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 reusable building blocks that can be plugged together as needed by the application to be built. The patterns try to identify essential properties of reusable software. In particular, we show that extensive standardization is not a necessary prerequisite of reusability 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 to design and develop software have been invented. Deciding which way to take in a given situation has become difficult not only for novices, but also 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 was successfully used to produce elegant software. However, no approaches does consistently produce *reusable software components*. Reusability still seems 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 reusable software, independently of any specific design method. Although most underlying ideas have been described by other authors before, often they were covered only briefly, or scattered within a huge body of other material. Presenting them as a compact pattern language has been of great help to me to better understand 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 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 gradually 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.

Design methods differ in how they facilitate abstraction and reuse to make complexity and change manageable in a given situation. The notion of *scale* to be explained next will give us a deeper understanding of these differences. The discussion of scale will be the basis for the fundamental definition of a *building block* which is the central unifying concept behind the patterns presented in this paper.

Scales and Building Blocks

The notion of scale in software development has received considerable attention recently. For example, scale considerations play a significant role in the CORBA standardization effort. In their „CORBA Design Patterns", [MOWMAL97] relate each pattern to the scale were it is most often applied. They argue that many serious software mistakes have been made because the relationship between the scale of the problem and the scale of the solution was poorly understood or applied. [COCKBURN98] defines scale as a means of hiding things that are not relevant at the current stage of development. It is also important to understand that scale is not merely a quantitative concept but that scale transitions involve changes in quality: Like a bunch of trees will become a forest at a coarser scale, a bunch of related classes may become a module or a component.

Scales are mainly characterized by (and named after) the nature of the abstractions that are typically implemented at each level. In this paper we will concentrate on the following scale levels:

Algorithms and Data Structures: This is the lowest software level. Concepts reside at this level, if they are meaningful in relative isolation. Consider a list: When the list is implemented so that it can hold any data type its operations can be described without referring to any collaborator. The same is true for a sorting algorithm: The concept of sorting makes sense in isolation, as long as we are able to describe the container holding the data and the comparison operation abstractly (cf., for example, the C `qsort()` function and the C++ `sort()` algorithm). These fine-grained, independent building blocks lay a very good foundation for reuse because they can be combined almost freely to meet the needs of many different applications. Of course, efficiency is a major concern at this level, because performance given away here can not be recovered at the higher levels. The recently invented *generic programming* method [MUSSTEP94] is especially suited to achieve flexibility and speed simultaneously.

Collaborations and Micro-Architectures: This level contains those abstractions that don't make sense in isolation – interactions and collaborations. For example, defining an *employee* is only useful in the context of an *employer*. Abstract base classes and interface inheritance allow for a certain degree of type anonymisation that helps to handle the interactions of huge numbers of objects of many different types in a uniform and maintainable way. Thus, this level is the realm of *object-oriented programming* and collaboration patterns defining the *micro-architecture* of an application, as is exemplified by the design patterns presented in [GHJV94].

Modules and Packages: Modules and packages encapsulate a larger body of related functionality such as a number of collaborating objects. Modules have been invented with the explicit goal of facilitating software reuse. The main point their creation is to set up *boundaries* between different parts of a system. Boundaries divide the system into clearly identifiable subsystems that can be handled and used separately. Boundaries confine what's going on in one part of a system so that local changes can not propagate and cause changes at the other end of the system. In this sense, modules have been the origin of the *interface* concept as a means to cross subsystem boundaries in a well-defined way.

Architecture: The architecture of an application determines how the modules are put together. I.e., it must control their interaction, provide the necessary adapters, pass information around and so on. Besides this technical responsibilities, the architecture has also an important non-technical purpose: It communicates the structure of the application to the developers and maintainers so that changes and extensions can be made without destroying the underlying abstractions. An architecture thus acts as a *design center* [MÄBISCH96] which supports system evolution by defining a stable core that preserves system integrity.

An example from image analysis, namely a system to reconstruct a 3D object from a set of images, shall illustrate the scales. At the lowest scale of such a system we have a bunch of independent data structures like images, features (such as points, edges, and surfaces) in both 2D and 3D, and cameras as well as a large number of generic algorithms for image filtering, feature extraction, feature matching between images, back projection from 2D into 3D and so on.

At the micro-architecture scale we group the generic functionality into more specific objects such as the picture (which relates an image, the corresponding camera, view point coordinates, a collection of features found in this image, and the algorithms needed to extract this information) and the model (containing the features describing the 3D shape we are interested in). In addition, we have to carefully design the collaboration between pictures and models. This is especially important as the solution of the reconstruction problem can only be obtained by processing many features from many pictures at once. We must also keep spatial and temporal relationships between the features, update the structure as analysis proceeds, and detect erroneous measurements that don't fit into the model.

At the modules' level we might define domain specific modules like a feature extraction package, a matching package, and a photogrammetry package, as well as standard modules such as a user interface library and data import/export facilities. Finally, at the architecture level we could use a Model-View-Controller architecture for the interactive parts of the system and a Pipes-and-Filters approach in conjunction with a high-level control language for the non-interactive parts of our system.

Some additional remarks about objects seem appropriate. Object-based techniques have the advantage of being applicable at many different scales. For example, objects can be used to implement algorithms (e.g. the STRATEGY pattern), to organize collaboration patterns, to build module interfaces (e.g. the FACADE pattern, see [GHJV94] for explanations), and to define an architecture (e.g. an object-oriented framework). Although such versatility may contribute to design simplification, it also tends to blur the scale boundaries. Blurring has gone so far that some designers questioned the separation of the „Algorithm and Data Structures" and „Collaboration and Micro-Architectures" levels. Experience in algorithm intensive domains like numerics and image analysis suggests, however, that this merge is not helpful. Instead we must strive to keep the scale boundaries against the blurring forces.

Of course, not all levels have the same significance in all applications. For example, algorithms are relatively simple in most business applications, at least in comparison with the object coordination problems. Similarly, architecture will receive much more attention in a large application than in a small one. In certain well-understood domains we can even build applications almost entirely at the module level by using tools like VisualBasic or Fourth Generation Languages. Here, reuse of solutions at the algorithm, object, and architecture levels leads to extremely fast development. The important point is, however, that it is absolutely necessary to identify the scale where each problem resides, and to choose a corresponding design method. We will come back to this issue in the pattern section.

If we step back a little from the list, we see a recurring pattern: levels that mainly implement functionality (algorithms and data structures, and modules) alternate with levels that mainly coordinate the application of this functionality (objects and architecture). This is interesting because it highlights the fact that implementing some functionality and coordinating its application are indeed two different tasks that should both receive equal attention. Neither of the two can exist without the other, but nevertheless they must be separated due to the different abstractions they require.

To build a reusable system we need to balance two opposing forces – flexibility and encapsulation. In the above discussion of scales we have seen that this is the explicit purpose of the module level. The key observation of reusability is, however, that these forces must be balanced at every scale level, although sometimes in a less strict sense. Therefore, it is a natural conclusion to generalize the notion of a module to any scale level. This is the purpose of the term *building block.*

A *building block* establishes the necessary boundaries that facilitate reuse. That is, it encapsulates some reusable functionality, but at the same time provides well defined mechanisms (the interface) to adapt this functionality to the requirements of a larger whole. When a building block delegates some tasks to another one, we will call this a client-server relationship, and the building blocks involved will be termed collaborators. Building blocks are the granule of the *Open-Closed Principle* [MEYER94]: they should be open for enhancements and adaptation, but closed against modification of the existing functionality by not allowing changes to the encapsulated source code.

The building block concept turns out to be a very powerful generalization that allows us to find the interesting parallels between different design approaches presented in the patterns below. Since design methods differ widely in what their building blocks are and how they cooperate, these commonalties are often hidden behind different names or different conventions, and only the unified building block concept brings them to light. Many methods that are often regarded as competitors 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 such as the ones we are presenting here. 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	Purpose
Separation of Concerns	building block definition	decompose problem into narrowly focused building blocks
Abstraction of Dependencies	building block definition	make building block independent by abstract description of its dependencies
Offered Interface	building block specification	specify building block's exported behavior
Required Interface	building block specification	specify building block's expectations with respect to its collaborators
Descriptive Interface	building block specification	provide machine-readable documentation
Flexible Integration	building block integration	connect building blocks so that independence is not lost
Interface Standardization	building block integration	optimize and further abstract interfaces as to exploit commonalty
Interface Translation	building block integration	define interfaces so that adaptation becomes easy
Self-configuration	building block usability	provide meta-information to set up collaborations automatically
Flexibility on Demand	building block usability	hide flexibility behind a thin shell

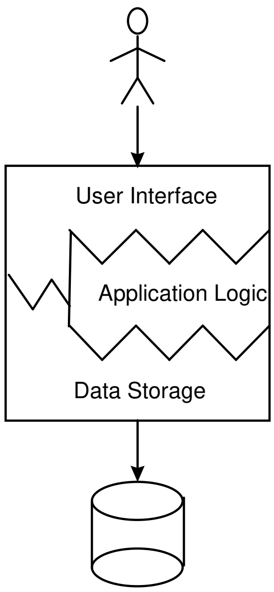
Building Block Definition

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.

SEPARATION OF CONCERNS

Building blocks that implement several tasks simultaneously are harder to reuse.

The evolution of multi-tier systems nicely exemplifies this at the architecture and modules' levels. 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.



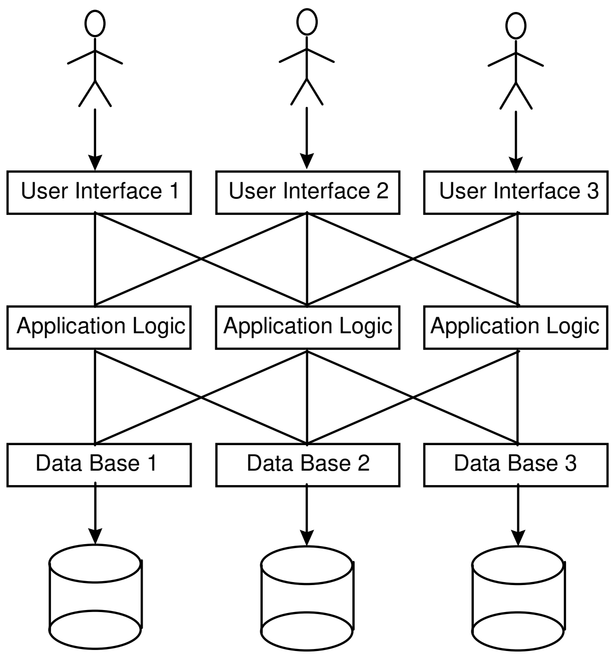


**Figure 1: Left: Monolithic design, insufficient separation of concerns,
Right: Three-tier architecture, better separation of concerns**

This example mentions some common instances of SEPARATION OF CONCERNS at the modules and packages level: the primary task of the system (the application logic) is concentrated in the middle tier, while the secondary tasks – storing and accessing persistent data in a database, building a user interface, and organizing communication between independent components – are delegated to separate building blocks. Other typical secondary tasks include: memory management, performance optimization and parallelization, concurrency control, error and failure handling, massage passing and event dispatch, printing and rendering.

While decomposition of primary and secondary tasks is equally important, secondary tasks deserve special attention in an attempt to improve reusability. First, they often account for more than half of the code of a software system. Thus, reusable building blocks for the secondary tasks would free considerable resources to deal with the primary tasks of a system. Second, no one can have expert knowledge in all these areas at once. So, if one cannot afford hiring an expert for everything, reusable components will be the only way to get high quality solutions for the secondary tasks. Finally, feasible solutions to secondary tasks are very much dependent on the environment. For example, an optimization working on one machine might not be possible on other machines. Thus, the ability to exchange secondary tasks is crucial for the reusability of our solution to the primary tasks.

SEPARATION OF CONCERNS has three important effects:

1. It makes building blocks focused. A building block containing just one well-defined functionality will be easier to learn, to maintain and to transfer into a new context.
2. It localizes functionality. If changes are required, only a single building block must be modified or exchanged. Localized behavior is also necessary to locally certify correctness.
3. It makes dependencies explicit. Implicit dependencies will lead to unexpected incompatibilities which are hard to repair. Only explicit dependencies can be abstracted (see the pattern ABSTRACTION OF DEPENDENCIES).

An interesting problem is the question exactly which granularity is optimal. Too fine granularity results in a huge number of building blocks that do almost nothing which unnecessarily complicates the design. Too coarse granularity will result in suboptimal reusability because certain aspects of a computation cannot be manipulated. Although I know of no complete answer to this question, experience suggests that the granularity typically found in real systems tends to be too coarse rather than too fine. Considering the scales of software design is also of some help. Once you have identified the scale on which a problem resides, granularity follows to some extend from the kinds of building blocks dominating at this scale. By and large, collaborating building blocks should reside at the same scale level, and building blocks crossing the scale boundaries deserve special attention.

Therefore:

Solve complicated problems by decomposing them into simpler ones. Make building blocks narrowly focused on one well-defined task and delegate other tasks to collaborating building blocks. Clearly identify the scale level of each problem to get an idea what the right granularity of decomposition should be.

ABSTRACTION OF DEPENDENCIES

Making dependencies explicit is necessary, but not sufficient to create reusable building blocks.

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;         // 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 thereby generalizing binary 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;         // 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. I.e., the case of adding a `complex<float>` to a `complex<double>` is not covered. Thus, 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[1]:

```
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); // promote arguments to
    PromoteTraits<T1, T2>::Promote tmp(b);    // result type
    result += tmp;    // add-assign promoted right argument to result
    return result;    // return result
}
```

---

[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:

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

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. In the second version, this dependency is formulated more abstractly: Now, the operator accepts any type which supports copying and add-assignment. But we still know, that the right and left argument are 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 defined for the two operand types.

This is what I call ABSTRACTION OF DEPENDENCIES: A building block becomes more reusable, if it formulates its dependencies more abstractly. Instead of saying specifically, which collaborators it needs, the building block generalizes itself by describing only abstract properties of the necessary collaborators (SEPARATION OF CONCERNS). In doing so, it becomes more and more invariant with respect to changing collaborators. I.e. by describing its dependencies abstractly, the building block becomes itself abstract.

There are two different ways to implement ABSTRACTION OF DEPENDENCIES in practice. The first is to make the implementation match the degree of abstraction of the intention. This was done in the above example: 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 reusability 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). In these cases, generative techniques such as Aspect-Oriented Programming [KICZALES+97] are successfully used: the abstract description of the collaborators is formulated as an algorithm to generate the actual building block given concrete instances of the collaborators. A proper treatment of these techniques is currently beyond the scope of this paper.

When one abstracts dependencies, it is extremely important to pay attention to the scales of software design: According to the building block's scale level, the right kind of dependency abstraction varies greatly. For example, consider access to the data of a digital image. To implement image processing algorithms at the lowest scale, we need the fastest possible access to every single pixel, i.e. we would call for an abstract iterator. At the level of micro-architectures, we will usually need an entire abstract image rather than single pixel values. At an even higher scale, where network communication might be involved, passing entire images (let alone single pixel values) is to expensive, so that our abstract description might request a compressed low resolution image first, and the complete data are only requested when really needed. Of course, scale has also strong influence on the technical means used to describe abstract dependencies. 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 accordingly.

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

OFFERED INTERFACE

Since the inner workings of a building block must be hidden, its behavior must be described by an abstract interface.

The traditional way of looking at interfaces corresponds mostly to what I call OFFERED INTERFACE. A building block's OFFERED INTERFACE describes the assumptions a client of this building block can make about its behavior. 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 SEPARATION OF CONCERNS: Every building block should contain just one well-defined piece of functionality. This directly translates into the interface being narrowly focused as well – 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 tend to advocate minimal offered interfaces. A minimal interface must at least ensure *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 piece of functionality. For a stack, four functions are sufficient in principle: createStack, isEmpty, push, and pop. We can transform every state into another by pushing and popping the 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 problems with this approach. First, minimal interfaces lead 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. So we might replace isEmpty with a function getSize (isEmpty then becomes getSize == 0). But whatever you choose as the minimal set, many operations will still be inefficient.

Second, there are compatibility problems: If an offered interface has to be changed, it is often preferable to implement the change as an extension, so that clients relying on the old variant do not break. Of course, allowing extensions destroys minimality.

Third, in most cases minimality is not even a well-defined concept because it may depend on very subtle variations in what clients want to do. For example, the algorithm to compare stacks outlined above temporarily modifies the stack. Thus, as long as this algorithm runs, normal stack operation must be suspended. Often this is not acceptable: Consider a client that periodically monitors whether certain elements are in the stack. This functionality should not interfere with normal stack operation. To allows the client to inspect the stack non-destructively, we might add an iterator to the stack's interface. This would also give rise to a more efficient comparison algorithm, but is the interface still minimal? These questions get more and 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. It is clear that 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 this power makes the language extremely versatile: Pixel graphics, line graphics and text are all handled easily regardless of paper size and printer resolution. Printers which offer a PostScript interface are thus supported by a wide variety of different applications and operating systems.

However, extensive offered interfaces create their own bag of problems. First, 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 the 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. Since the criteria to include or exclude something can not be based on inherent properties of the building block itself, an extreme amount of preplanning is required to determine what clients might want to do. Accordingly, only experienced designers are normally allowed to define offered interfaces, but even then errors can not be avoided.

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 are shielded from offered interfaces by their required interfaces, the interface segregation principle is automatically fulfilled. Clients cannot know which functionality a server actually provides so that additional server capabilities do not generate new dependencies. Thus, there is no longer a need to arbitrarily hide functionality a building block could easily provide. The extensive offered interface becomes feasible.

Therefore:

Describe 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

If only offered interfaces are provided, clients tend to depend on the interfaces of their servers. Thus, clients are not truly independent.

Consider an operating system that provides a service „Printing" to application programs. Usually, the operating systems doesn't itself implement this service (SEPARATION OF CONCERNS), 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 this pattern: the requirements which were derived in the curse of abstraction are exactly the required interface of the final implementation of the addition building block.

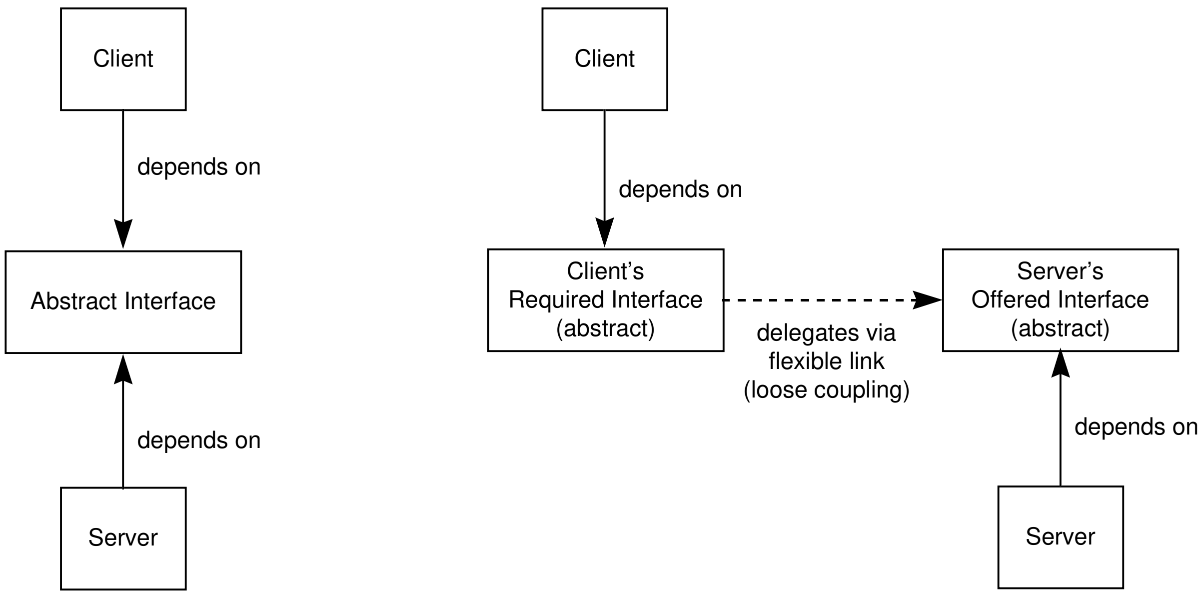


Figure 2: Left: client and server depend on the same interface and are thus not independent. Right: separation between client's required interface and server's offered interface results in truly independent building blocks

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. But besides this integration functionality, the original application has no idea which services a new plug-in will offer. In case of the WWW browser, it could, for instance, contain 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* if it collaborates with a *university*, while it 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 objects collaboration, and the core objects happen to take part in many such collaborations simultaneously. 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.

Separating required and offered interfaces is a fundamental principle of reusability:

1. It gives the idea of minimal interfaces a better meaning. Since we know exactly what the client wants to do with the server, the required interface can specify exactly this set of functionality. By careful design we can actually make the specified set minimal with respect to the known requirements. This solves the interface segregation problem: 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 – the required interface is a contract not to use any unspecified functionality. By accepting or rejecting the contract, the 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 was changed. 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:

Describe 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 reuse 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 or any other size, you have to search for all the hardwired 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. This means that a building block can work well without documentation, but it's much harder to maintain and reuse.

The independence of the descriptive interface has a very nice consequence: we can move this information into dedicated *meta building blocks*. Meta building blocks have a number of important advantages. First, they allow us to extend descriptive interfaces without modification of the building blocks described[2]. Consequently, when a client requests descriptive information about its collaborators, we can implement meta building blocks as necessary without touching the original building blocks.

Second, meta building blocks can refer to several building blocks simultaneously. This is very useful to express constraints and other relationships *between* building blocks. The `PromoteTraits` introduced earlier are an example for this. 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 should be optimized 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.

---

[2] This is not true for offered interfaces: Here, extensions always require corresponding extensions of the building blocks' implementations.

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 *offered interfaces must be mapped onto required interfaces*. This section describes several issues that need to be considered when doing so.

FLEXIBLE INTEGRATION

If the mapping of offered interfaces onto required interfaces gets hard-wired during building block integration, reusability is lost.

Several years ago I wrote a nice color reduction algorithm. Since the algorithm had some unique features, several colleagues approached me with the wish to reuse this algorithm in their projects. However, reuse (without source code modification) failed because it was not possible to extract the algorithm from its old environment: The algorithm depended on a specific image data structure. This image data structure was in turn derived from a base class needed for persistence, as is typical in object-oriented database systems. Compiling this base class finally required to link a huge database library into the application, even if we did not actually put any object into a database. Clearly, it was undesirable to transfer all these dependencies to the new environment, so source code modifications were needed to adapt the algorithm to its new environment.

The problem with this design is now obvious: Although it applied SEPARATION OF CONCERNS, the dependencies between the building blocks were hard-wired instead of being described abstractly and connected flexibly. In the previous patterns we have already seen how dependencies can be abstracted. Here we will discuss how separation can be maintained during building block integration as well. First of all: editing a building blocks' source code is not considered an instance of flexible integration – even if the source code is available, gaining the thorough understanding of the inner workings needed to avoid mistakes often takes longer than a reimplementation.

The right choice of an integration mechanism critically depends on the scale where the building blocks reside. Most importantly, we have to balance flexibility versus performance: The more run-time flexibility we need, the more performance overhead we must accept. Thus, on the lowest level we will prefer compile-time solutions, while at the architecture level interpreted languages and distributed environments are appropriate. The integration technique also influences, how required and offered interfaces are best written down. Let us briefly review design methods at different scales:

At the level of **algorithms and data structures**, programming (especially structured programming) has suffered for its lack of appropriate support for flexible integration. Procedure calls and data access are hard-wired into the program. Meanwhile, this problem has been solved by the introduction of *type parametrization* into important programming languages (e.g. „templates“ in C++, „generic packages“ in Ada, „genericity“ in Eiffel). Built on 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 a very simple example of this technique. The Standard Template Library (part of the Standard C++ Library, see [MUSSERSAINI95]) is currently the most advanced application of this programming style.

At the scale of **collaborations and micro-architecture**, inheritance in conjunction with abstract interface classes have been established as a preeminent approach. There is, however, a problem with inheritance: It is a static relationship, which can only be changed be editing the source code. Thus, inheritance alone does not fulfill our requirements for flexible integration – it must be combined with different forms of delegation. Many of the well known design patterns [GHJV94], e.g. COMPOSITE, MEDIATOR, and CHAIN OF RESPONSIBILITY, give detailed descriptions of how inheritance and delegation are effectively combined to achieve flexible integration. While delegation is directly supported by dynamically typed languages such as Smalltalk, in statically typed languages like C++ it must be implemented manually. [WEIHE98] gives a detailed discussion (with examples) how templates help in combining inheritance and delegation in C++.

At the **modules and packages** level, a relatively high degree of run-time flexibility is needed. Therefore, dynamic libraries in conjunction with abstract FACADES and FACTORIES have been successfully. Even more flexibility is provided by *middleware technologies* such as JavaBeans, DCOM and CORBA. Their dynamic invocation capabilities, even across process and machine boundaries, are exploited by several novel design techniques, including the promising *Component Based Development*[3] approach. In addition, modules are often configurable so that they can be adapted to the specific needs of the client. In extreme cases, a configurable interface can be specified as a programming language (like SQL for database access or PostScript for printing) .

At the **architecture** level, we can often reuse technical means from lower levels to implement loose coupling. For example, if the application requires data to flow through a flexible set of algorithms the pipes-and-filters architecture is successfully used, resembling a typical programming style at the level of algorithms and data structures. Object-oriented designs typically result in a framework architecture. Middleware technologies are the basis of client-server, multi-tier and broker architectures. These techniques are complemented by techniques specific for the architecture level such as event loops and high-level control languages (both in text-based – Python, the MatLab language etc. – and graphical variants – KHOROS, Fourth Generation Languages etc.). A more comprehensive treatment can be found in [BUSCH+96].

Sometimes, a very tight interaction of the different building blocks may be required which cannot easily be handled by the integration techniques listed above. [KICZALES+97] have introduced the term *aspect* to describe these entangled tasks. They propose a new programming technique, *Aspect Oriented Programming*, that uses specific *weaving* tools to combine the input from different building blocks into a consistent and efficient program. This approach introduces a completely new look on the loose coupling problem, and, if successful, could have a great impact on how we will design independent building blocks in the future.

Therefore:

When integrating reusable building blocks, don't hard-wire the mapping from required onto offered interfaces. Choose an appropriate flexibility mechanism according

[3] 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.

to the scale and the context of the problem which ensures flexible, exchangeable integration.

INTERFACE STANDARDIZATION

If all interfaces are defined completely independently of each other, the number of interfaces will explode. You will have a very hard time to find building blocks that are compatible with each other without extensive adaptation.

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 organizations 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 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: Since the two interface types are separate, 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 very complicated translations and algorithms 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. If someone wants a particular client-server pair to cooperate, he simply implements an appropriate connector object using the public offered interface of the server[4]. 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.

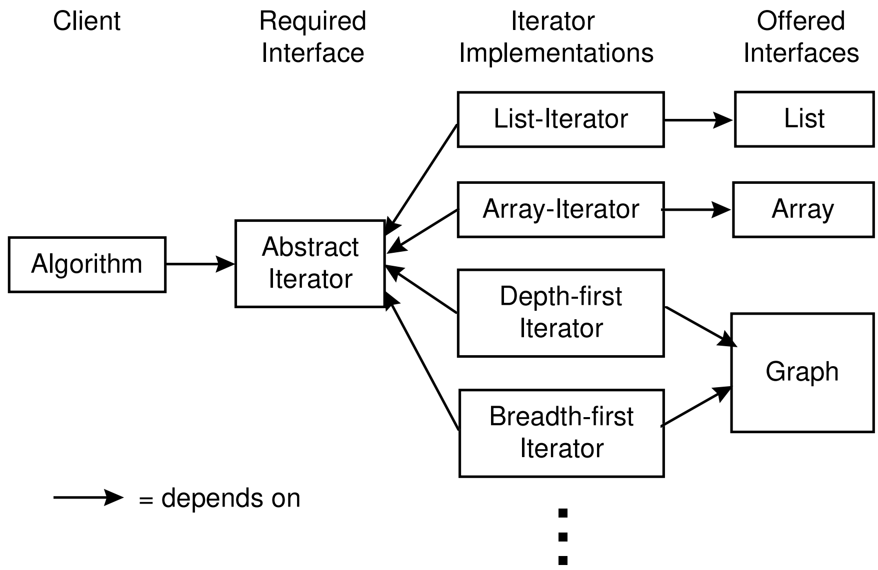


Figure 3: One algorithm works with many different data structures via iterators

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 reusability 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.

---

[4] If the translator programmer happens to be the server programmer, he can also make use of private server properties as an optimization. Then the new translator becomes an extension of the original offered interface.

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 thus serves to automate the process of setting up and controlling a network of collaborating building blocks. Meta-information allows 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 will create other building blocks. By using an abstract factory, a client is freed from the need to know the mechanisms of building block creation. According to the requirements of the current context, many different mechanisms can be chosen by simply exchanging or reconfiguring the factory (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 to construct 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` already discussed), 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. More general, formal approaches to this 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 by the third possibility where building blocks adjust their internals to the properties of their environment. For example, a numeric building block could adjust internal accuracy according to the clients 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 explicit client knowledge.

Therefore:

Standardize meta information such as descriptive interfaces and factories so that it can be used by building blocks to automatically adapt to their environment. Provide meta building blocks which are responsible for configuration, check, and adjustment of 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. Consequently, although it provides less functionality it is much harder to learn than the other systems.

In a known context, it is very useful to build the black box by wrapping a version of the building block configured for this context 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.

Hiding flexibility is also a side effect of the introduction of *atomic building blocks* at the scale boundaries. Atomic building blocks encapsulate the flexibility mechanisms of the lower scale level and map the low level functionality to the mechanisms of the new level. Consequently, implementing atomic building blocks as a thin wrapper is probably also a good idea.

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.

Although many of the underlying ideas are not new, some issues have, to the best of my knowledge, not been presented as definitely as in the present paper. In particular, this applies to the consequent treatment of the relationship between scale levels and design techniques, the emphasis on the separation of required and offered interfaces, and the observation that a proactive attitude towards interface adaptation will be needed. The latter is especially significant since it also shows that standardization is not a necessary prerequisite for reusability.

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 presented ideas. I'm also very grateful to Wolfgang Keller (the shepherd), Karsten Weihe, and Alistair Cockburn 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: „Beyind the Black-Box: Open Implementation“, IEEE Software, vol. 13, no. 1, January 1996

[KICZALES+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Loes, J.-M. Loingtier, J. Irwin: „*Aspect-Oriented Programming*“, Xerox PARC Technical Report, February 97, SPL97-008 P9710042

[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