Requested Interface

Ullrich Köthe
Fraunhofer Institute for Computer Graphics, Rostock, Germany
Joachim-Jungius-Str. 9, 18059 Rostock
Email: koethe@egd.igd.fhg.de

Abstract: This paper introduces the *Requested Interface* pattern which describes ways to implement truly independent software components that can be plugged together as needed in order to make reuse more attractive than reimplementation. It encourages components to delegate subtasks to collaborating servers so that they can be adapted to a new context by simply exchanging those subtask servers. The delegating objects must specify minimal and abstract requested interfaces that describe the subtasks independently of existing server interfaces. An adaptation layer mediates between the requested interface of a client and the offered interface of a server implementing the subtask.

Purpose

Make reuse of services more attractive than reimplementation by providing independent components that can be plugged together according to the needs of the application to be built.

Motivation

Although highly desirable, software reuse is still much less customary than it could be. Programmers often choose to reimplement a service because this is much easier than reusing any existing one. For a service to be reusable, two fundamental requirements must be met:

1. It must be easy to extract a reusable service from its old context. In particular, the service must not indirectly depend on a lot of unrelated code which we don't want to carry to the new environment.
2. It must be easy to adapt the service to the new context. In particular, it should not be necessary to touch the source code of the service („Open-Closed Principle", see [MAY94], [MART94]).

We will discuss these requirements using an algorithm that transforms an RGB image into a corresponding gray level image as a running example. In the old days of structured programming such an algorithm might have been implemented like this (abusing C++ for structured programming):

```
void rgbToGray(RGBImage * rgb, GrayImage * gray)
{
    for(int i=0; i < rgb->width*rgb->height; ++i )
    {
       // calculate weighted average of colors
       gray->data[i] = 0.30 * rgb->red[i] + 0.59 * rgb->green[i] +
                       0.11 * rgb->blue[i];
    }
}
```

where we assume that the images are structured data types defined like this:

```
struct RGBImage {
  int width, height;
  unsigned char *red, *green, *blue;
};
```

```
struct GrayImage {
  int width, height;
  unsigned char * data;
};
```

This implementation certainly does not fulfill the second requirement (one may argue about the first): It is in no way adaptable to a new context without modifying the source. Everything is fixed: the data types, the particular method to calculate gray values, the region of interest (the entire image) etc.

Better reusability is achieved by means of object-oriented programming. We use abstract interfaces and inheritance in order to allow servers to be rewritten without changing the clients. Algorithms are assigned to objects as member functions, so that the RGB to gray transformation will look like this:

```
class GrayImage {
 public:
   virtual unsigned char & pixel(int x, int y); // read/write pixel (x,y)
};

class RGBImage {
  public:
    // other member functions ...

    virtual void transformToGray(GrayImage * g)
    {
      for(int y=0; y<height_; ++y)
        for(int x=0; x<width_; ++x)
          // calculate weighted average of colors at pixel (x,y)
          g->pixel(x,y) = 0.30 * red_[x+width_*y] +
                          0.59 * green_[x+width_*y] +
                          0.11 * blue_[x+width_*y];
    }

  private:
    int width_, height_; // width and height of the image
    unsigned char * red_, * green_, * blue_; // the color bands
};
```

Since the algorithm is implemented as a virtual member function, it is inherited by any subclass of `RGBImage` and can be reused or reimplemented there. Access to the `GrayImage`'s pixels is encapsulated in the virtual function `pixel(x,y)` so that the algorithm runs on any subclass of `GrayImage` as well. Thus, we have some degree of freedom in adapting the algorithm without touching the original source code. In practice, however, this is still not sufficient:

- If our application must use image data types which are not subclasses of `RGBImage` and `GrayImage` (for example, the KHOROS VIFF format or one of the image classes of the Image Understanding Environment), we must convert data back and forth. Apart from code bloat and maintenance problems due to multiple image formats in one application, this takes time, and we must always ensure that the different copies remain consistent.
- The algorithm is only defined for `unsigned char` pixels. If we have other pixel types like `float` we cannot convert without loss of data.
- The basic loop structure is common for many image processing algorithms. We would like to abstract it so that we need not implement it again and again. Also, as written the algorithm always operates on the entire image. We would like to restrict its application to an arbitrary subimage.
- If a class has many member functions, adaptation by derivation often leads to an exponential explosion in the number of classes since any combination of function variants needs its own subclass.

These drawbacks alone would, however, not suffice to prevent us from reusing a worthwhile algorithm. A much more severe obstacle is encountered when we consider the first requirement - extracting the algorithm from its original context. Since we can reuse a class only as a whole, we have to carry all member functions into the new environment, even if we wanted to reuse just one. Moreover, these member functions depend on yet other classes so that we end up with a cascade of dependencies which must all be kept intact in the new environment.

Thus inheritance naturally leads to the *framework* approach to reuse: For a given application domain, a number of server interfaces and corresponding implementations are provided (often as a commercial system), and applications only extend the framework according to their needs. This works nicely for well understood functionality like a user interface, but it breaks down as soon as the framework does not suffice to accomplish the task at hand - it is very hard to just extract some nice

services from the old framework and switch to another one for the overall system (because everything tends to depend on each other), or to combine several such frameworks into one application (at best, this will duplicate much functionality, at worst it won't work, see for example [GARL96]).

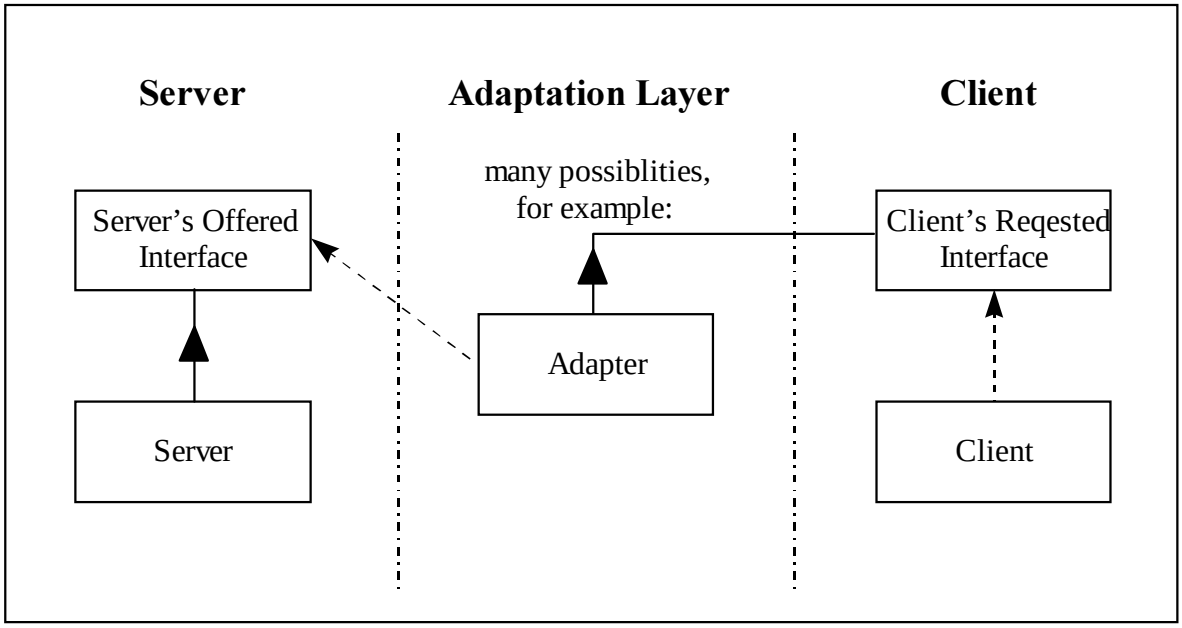
Consequently, to meet both requirements we need a set of small, independent components that can be freely plugged together according to the needs of the task at hand. Each component knows how to accomplish a certain, well defined task but needs the collaboration of others to do anything at all. To meet the first requirement these dependencies must not be static so that collaborators can easily be exchanged in order to adapt them to a new context. Such a *component based* approach has been envisioned for a long time, but it cannot be build with traditional object-oriented programming alone. The *Requested Interface* pattern is an attempt to summarize ways to make component based programming work.

Solution

The *Requested Interface* pattern is applied in three steps:

1. When implementing a service, identify subtasks that should be separated. Typical candidates include subtasks that might need to be changed independently of the rest of the computation (cf. the „Open-Closed Principle") or might be reusable by other services. Explicitly delegate these subtasks to collaborating subtask servers. The original service becomes a client of these subtask servers.
2. Look at the collaboration from the perspective of the delegating object (the client) and make its assumptions about the subtask servers explicit by specifying a „requested interface" for each collaboration. A requested interface is a formal specification of the operations the client wants a server to execute. This interface should be minimal and not directly depend upon any particular server's actual interface so that client and server become truly independent of each other.
3. To build an application, the programmer selects suitable components and adapts the offered interfaces of the servers to the requested interfaces of the clients via a separate adaptation layer. Since there are several possibilities to implement the adaptation layer, some of them already known as patterns, *Requested Interface* may be considered a meta-pattern, that generalizes existing design patterns.

Structure



Participants

- **Client:**
  - delegates subtasks to collaborating servers to accomplish its task
  - shall be adaptable to many applications
- **Client's Requested Interface:**
  - specifies the operations the client wants its server to execute
  - is minimal and abstract
  - does not directly depend on the interface of any concrete server
- **Servers:**
  - process some subtask for the client
  - should be exchangeable as easily as possible
- **Server's Offered Interface:**
  - abstract interface to the services of the server
  - may or may not directly conform to the requested interface of the client
- **Adaptation Layer:**
  - mediates between offered and requested interfaces
  - can be implemented by many different techniques
  - can be automated if offer and request happen to fit together or differ in deterministic ways

Consequences

Benefits

- **Selfish Clients:** The traditional style of designing collaborations between clients and servers is pretty much biased towards the server side. Servers actively specify their (possibly abstract) interfaces, and the client has to accept them. As compared to this dictatorial style of programming the *Requested Interface* approach is much more democratic: The client takes on an active, selfish role by specifying what it needs irrespective of what offers already exist. This is the key to make extraction of the client from any given context simple, as requirement 1 demands.

- **Exchangeable Servers:** The existence of the requested interface is a necessary prerequisite for different servers to be exchangeable: It constitutes the client side of the collaboration and states clearly what the result of the adaptation should be. If no requested interface is specified you must work around its lack by naming some old server interface as requested interface. However, this would be a bad solution since it causes the client to depend on the *entire* interface of the server so that the notion of a *minimal* requested interface is lost and a lot of unnecessary dependencies are created - contradictory to requirement 1. This makes adaptation much harder since the adapter must provide an implementation for every function of the old server so that you may end up investing lots of effort into functions that are never called.

- **Separation between Algorithms and Data Structures:** This is a special case of the previous statement and one of the most important applications of the *Requested Interface* pattern. Algorithms are clients of the data structures that serve them the necessary data to run. Usually, one and the same data structure can serve many algorithms, and an algorithm can be applied to many different, but similar data structures. For complex data structures, one single algorithm seldom uses all the data, but each one needs a specific view on them. The requested interface tells exactly and independently from any particular data representation what kind of view the algorithm needs.

- **Reusable Components:** The *Requested Interface* pattern is the key to *component based programming*: Servers and clients are truly separated and can be reused independently of each other. The programmer selects suitable components that might fit together, and just writes the

necessary adapters. It is not even necessary that each requested interface is delivered by exactly one server. Instead, several servers may cooperate, and only when viewed through the adapter do they appear as a monolithic object conforming to the requested interface. For this to work the specification of both requested and offered interfaces is absolutely necessary.

- **Mediating Adaptation Layer:** On first glance this might appear more like a drawback than a benefit. However, by looking closer we see that the adaptation layer is not an artifact caused by the proposed pattern but is actually a direct consequence of our desire to make components pluggable: In most cases we will not have control over the interfaces of both clients and servers. Therefore, adaptation is necessary to meet requirement 2, and an independent requested interface helps making it as easy as possible. Sometimes, offered and requested interfaces may be so different that the adaptation layer is difficult to build, but if the service to be reused is very complicated (like a typical image understanding algorithm) you will happily invest a fair amount of work into writing adapters.

- **Minimal Requests and Maximal Offers:** The *Requested Interface* pattern also allows to resolve a long-lasting controversy: Should interfaces be minimal, i.e. provide only the functionality absolutely necessary to do something useful, or should they provide several options to achieve the same results? In the light of the proposed pattern the answer is simple: Requested interfaces are more reusable (i.e. it is more likely to find a suitable server) if they are minimal. However, servers are easier to reuse if they offer a broader spectrum of possibilities to access their functionality since different clients need not agree on what the best minimal interface is (which, of course, is also task dependent). Therefore, we should restrict ourselves to the necessary when writing reusable clients, but should be splendid when writing server interfaces.

- **Extended Design by Contract:** In the light of Design by Contract [MEY94] the *Requested Interface* may be seen as a promise of the client not to use any services and functions except those specified. Such a promise is inherited by subclasses of the base client, i.e. subclasses that are to be used polymorphically through base class pointers can not rely on the server having any more functionality than requested by the base (unless you give up static type checking and rely on downcasting and run-time checks instead).

Drawbacks

- **More Complicated Design:** As we need explicit subtask servers and extra adaptation classes the design may be considerably more complicated. This is the price we pay for reusability. There are, however, techniques to reduce this price: Frequently, the requested interfaces of several clients are closely related. These interfaces may be classified into categories (cf. the iterator categories in the Standard Template Library [SL94]) so that clients can share them by simply referring to the appropriate category.

  Moreover, programming techniques and tools to automate adaptation are available or can be developed (see Implementation section) so that the adaptation layer will become invisible in many cases.

- **Potential Loss of Performance:** The additional levels of indirection may cause a considerable loss of performance. On the other hand, you save implementation effort due to reuse, which overall may outweigh the performance lost, at least during rapid prototyping. Also, there are implementation techniques like just-in-time compilation (Smalltalk) and inlining (C++) that eliminate or reduce the overhead so that it becomes acceptable for almost all applications.

Implementation

Explicit Subtask Servers

There is always a trade-off between what should be done in the server itself (as to justify its very existence) and how much should be delegated. The optimal solution is, of course, task dependent and may also evolve over time. In general, you should always consider to delegate subtasks that may need to be changed independently of the rest of the computation. Also, subtasks that might be reusable by many other clients are primary candidates for separation. A very detailed discussion on how to identify subtask servers can be found in [MART94].

Three tiered architectures, which are widely used in business applications, provide a nice example [FOW96]: The user interacts with the user services tier. All computations are delegated to the business services tier which may consist of any number of exchangeable servers containing the application logic. These servers in turn delegate data access to the data services tier which is formed by one or more (distributed) data bases. The tiers are connected by adaptation layers which may, for example, mediate between relational databases and object-oriented application programs.

Writing Requested Interfaces

There are several possibilities to write requested interfaces depending on the implementation environment: In mixed language and distributed environments you should use a dedicated interface description language such as CORBA´s IDL [OMG95]. In object-oriented languages the requested interface is often specified as an abstract base class from which the adapters can be derived (cf. *Generic Bridge* below). If you want to use *generic programming* the requested interface is best described in a table listing the functions the client wants to call (see „Example Resolved“ section). Here an abstract base class would be to restrictive, as the same functionality can often be provided in different ways (in C++: global operator vs. member function, automatic type conversion etc.), and certain types of compiler optimizations would be prevented by using abstract base classes only.

Implementing the Adaptation Layer

When implementing the adaptation layer we must keep in mind that we must not introduce static dependencies between clients and servers. I.e., in statically typed languages we can not solely rely on inheritance since it is a static relationship. We need to distinguish two cases: (1) the offered interface of the server is a superset of the requested interface, i.e. automatic adaptation is possible, and (2) the two interfaces don’t fit exactly, i.e. explicit adaptation is required.

(1) **The Interfaces Fit Together:** Of course, adaptation is easiest if you can control both clients and servers as to make their interfaces fit together. This may be achieved by a design style which I call *Micro Use Cases:* Start design by specifying requested interfaces, separately for each client. Then take related, partially overlapping, requested interfaces and make them as consistent as possible. Try also to generalize them into interface categories. Finally write servers that can efficiently deliver the requests without the need of explicit adaptation. If this is not possible iterate to improve the requests.

Now, if we are using a dynamically typed language like Smalltalk we are done - the run-time system automatically looks up the correct functions. In statically typed languages like C++ and Eiffel we may use genericity to let the compiler automatically generate forwarding functions, resulting in *Generic Programming* or *Generic Bridges.* In languages lacking genericity or in mixed language / distributed environments *Code Generators* are commonly used.

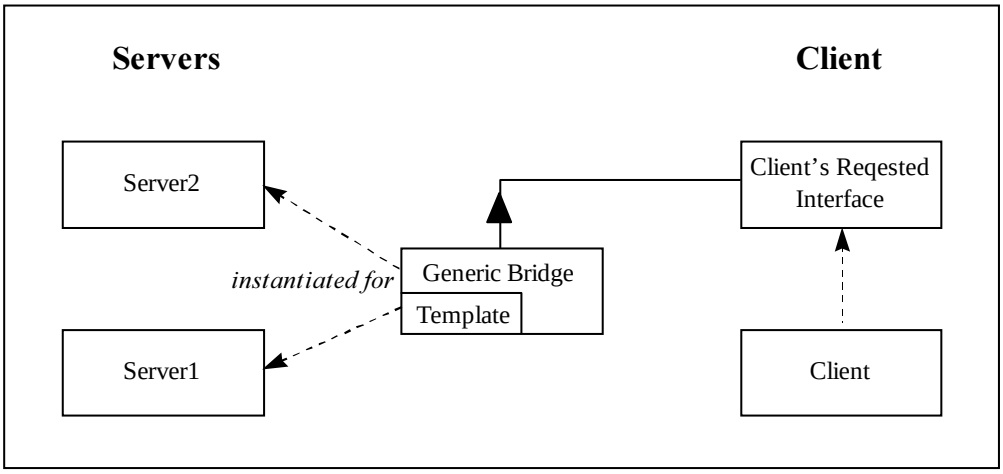
- **Generic Programming:** This is the approach of the Standard Template Library (STL) [SL94]. The requested interface is declared as a template parameter of the client:

```
/*   struct RequestedInterface
     {
          // list required functions
     };
*/

template <class RequestedInterface>
struct Client
{
     Client(RequestedInterface * itsServer);
     // etc.
};
```

Any object conforming to the requested interface can be used to instantiate the template. In this approach the requested interfaces need not exist as separate classes, they are pure specifications to which the servers must conform. Every time a new server is introduced, a new instance of the client template is generated. For large clients this may lead to code bloat and extensive compilation time. It is, however, the appropriate solution if speed of access is of utmost concern, e.g. if the requested interface is an iterator who's inlined functions are frequently called within a loop.

- **Generic Bridge:** This approach is a variation of the *Bridge* pattern from [GHJV94]. By using genericity, we modify the structure of the original pattern so that static coupling is avoided:



In the *Generic Bridge* pattern we implement the client in terms of an abstract requested interface and provide a template subclass of this abstract interface that automatically implements it by wrapping an appropriate server, for example:

```
struct RequestedInterface
{
  public:
     virtual int aFunction() = 0;
};


template <class Server>
struct GenericBridge: public RequestedInterface
{
     GenericBridge(Server * itsServer)
     : itsServer_(itsServer)
     {}
```

```
    virtual int aFunction() {
         return itsServer_->aFunction();
    }
  private:
    Server * itsServer_;
};

class Client  // not a template
{
  public:
    Client(RequestedInterface * itsServer);
};
```

In contrast to the original *Bridge* pattern the servers are independent of each other and need not inherit from a particular bridge interface class.

Sometimes it may be appropriate that a generic bridge inherits multiple, closely related requested interfaces. This will simplify the implementation process as there is only one bridge for several clients, but one should be very careful not to introduce new dependencies in doing so.

- **Code Generators:** If the generic solutions are not applicable (e.g. because your language doesn't support them or because client and server are written in different languages) you may use or write a code generator that automatically produces the adapter code from the requested interface. This approach is, for example, taken by CORBA [OMG95] where a code generator (misleadingly called „IDL compiler") translates the IDL specification into source code for the implementation language selected.

(2) **Requested and offered interfaces do not fit:** Here the solution depends largely on the type of misfit. Several patterns from [GHJV94] are among the implementation choices.

- The *Adapter* pattern is applied if one server semantically fulfills the requested interface but the generic methods above are not applicable (e.g., the function signatures differ). Often you can use a code generator which partly automates the adapter implementation, e.g. by producing an adapter skeleton where you need only fill in the missing function calls.

  Several variants of the *Proxy* pattern may also be used, e.g. if the server resides in a different process (*Remote Proxy*), if the server shall not be loaded completely (*Virtual Proxy*) or if the result of a complicated calculation will be requested multiple times (*Cache Proxy*).
- Often, cooperation of several servers is necessary to meet the client's needs. Then the *Mediator* pattern is a good implementation choice. If the appropriate server to respond depends on the request itself you may also use *Chain-of-Responsibility*.

  Another interesting option is offered by *Adaptive Programming* [LIEB96], although its original purpose is different, namely the enforcement of the *Law of Demeter* (see below). Adaptive programming is based on a graph representation of the relationships between cooperating servers. Given a requested interface and some information about the algorithm and the server(s) to respond, a code generator could automatically create a suitable adapter.
- The most difficult case is encountered if the adapter must modify the semantics of the server's interface. For example, in image analysis we must construct feature adjacency graphs (storing neighborhood relations between corners, edges and faces) out of the results of a segmentation algorithm. Since the results of the segmentation are typically not perfect, they may contain structures not allowed in a proper feature adjacency graph (say, edges not ending at a corner). An auxiliary data structure in the adaptation layer may temporarily store these structures until the inconsistencies are resolved.

  For the most part, the case of semantic adaptation is not yet formulated in pattern form. The only patterns that come close to this idea are the *Decorator* to add and the *Protection Proxy* to hide functionality. Further research is certainly necessary.

Example Resolved

By looking closer at the computations involved in the RGB to gray transformation we can identify the following subtasks:

- iterating over the pixels of two images
- reading the color channels of the RGB image, which may have arbitrary types
- writing the pixel values of the gray image, also of arbitrary type
- performing a certain computation with the pixel values

Flexibility would be maximized if we could separate these subtasks into exchangeable collaborators. Although it may look like to heavy a decomposition for such a simple algorithm, it very nicely exemplifies the ideas presented here. We use the *generic programming* approach as to maximize flexibility and performance. The core functionality is abstracted into a generic loop algorithm (where `iter1` and `iter2` mark the upper left corner of two images, `lowerright1` marks the lower right corner of the first image, and `transform` is a functor performing some computation on each pixel):

```
template <class ImageIterator1, class ImageIterator2, class Functor>
void foreachPixel(ImageIterator1 iter1, ImageIterator1 lowerright1,
                  ImageIterator2 iter2, Functor transform)
{
  int width = iter1.horizontalDistance(lowerright1);
  int height = iter1.verticalDistance(lowerright1);

    // iterate down the first column
  for(int y=0; y<height; ++y, iter1.incY(), iter2.incY())
  {
        // iterate across current row
      IterIterator1 row1(iter1); ImageIterator2 row2(iter2);
      for(int x=0; x<width; ++x, row1.incX(), row2.incX())
      {
          transform(row1, row2);  // do some computation
      }
  }
}
```

This function has the following requested interface (`i1` and `j1` are instances of `ImageIterator1`, `i2` of `ImageIterator2`, and `transform` of `Functor`):

Operation	Result	Semantics
`i1.horizontalDistance(j1)`	`int`	horizontal distance between two iterators (positive if `j1` is to the right of `i1`)
`i1.verticalDistance(j1)`	`int`	vertical distance between two iterators (positive if `j1` is below `i1`)
`i1.incY(), i2.incY()`	not used	advance in y direction (down)
`i1.incX(), i2.incX()`	not used	advance in x direction (to the right)
`ImageIterator1 k1(i1)` `ImageIterator1 k2(i2)`		copy constructor
`transform(i1, i2)`	not used	perform custom computation

Table 1: Requested interface of `foreachPixel()` function

Now we can apply the algorithm to any pair of images for which we can construct the necessary iterators, not just `RGBImage` and `GrayImage`. Also, the requested interface does not state where the iterators should point, so that we can restrict application of the algorithm to an arbitrary rectangular subregion by providing appropriate iterators. A functor doing RGB to gray conversion might look like this:

```
template <class RGBIterator, class GrayIterator>
struct RGBToGray
{
    void operator()(RGBIterator & rgb, GrayIterator & gray)
    {
       // calculate weighted average of colors
      *gray = 0.3 * rgb.red() + 0.59 * rgb.green() + 0.11 * rgb.blue();
    }
};
```

The functor also uses the iterators but with a different requested interface which must be merged with the request of the `foreachPixel()` function above (`rgb` is an instance of `RGBIterator`, `gray` of `GrayIterator`, and `d` is a `double`):

Operation	Result	Semantics

`rgb.red()`
`rgb.green()`
`rgb.blue()`
	convertible to `double`	read red, green, and blue components of the RGB pixel

`*gray = d`	not used	assign pixel value

Table 2: Requested interface of functor `RGBToGray`

An application that wants to use the transformation algorithm has to provide implementations of the iterators for the image formats it uses and could then call the function like this:

```
foreachPixel(getUpperLeft(rgbimage), getLowerRight(rgbimage),
             getUpperLeft(grayimage),
             RGBToGray<MyRGBIterator, MyGrayIterator>());
```

where the functions `getUpperLeft()` and `getLowerRight()` return iterators to the specified positions in the images. All the weak points of the old implementation have been resolved: By implementing iterators we can use the algorithm for arbitrary image formats, and by using a different functor we can change the algorithm implementation while reusing the loop implementation. Both of the fundamental requirements for reuse are met.

Related Patterns and Design Principles

- **Generic Programming** [MS94]: Generic Programming, and in particular the Standard Template Library, was one of the main inspirations for writing down this pattern (see the „Known Uses“ section below). The STL iterators are clearly applications of *Requested Interface,* in conjunction with the particular implementation technique as described above.
- **Patterns from** [GHJV94]: As has been shown in the previous section, several patterns from this book (most importantly *Bridge, Adapter,* and *Mediator)* may be used to implement the adaptation layer of the *Requested Interface* pattern.
- **Trader** (see for example [BR97]): The trader concept may be regarded as a consequence of the *Requested Interface* pattern. Given a requested interface, the *Trader* tries to find matching servers automatically by comparing the offered interfaces with the request. Different adaptation variants may be supported to improve chances for finding a suitable server. The Object Management Group (OMG) is currently working on a standardized trading service for the CORBA environment.
- **Roles** [REEN95]: Role based design is quite close to the *Requested Interface* pattern. The main difference lies in the fact that in the latter the clients play an active part by specifying the requested interfaces, while a *Role* is actively taken on by a server. I think that the change in viewpoint – from the server to the client – leads to an interesting new attitude to programming as a process of balancing demand (client's requested interfaces) and supply (server's offered interfaces) via negotiation and adaptation. This programming style encourages reuse by strongly reducing coupling between different parts of a software system towards really self-contained components that can be freely combined together.

- **Law of Demeter** [LIEB96]: This law makes a statement about which collaborators a client is allowed to use: namely objects that it contains or created itself, and objects that are explicitly passed as function arguments. It must *not* send messages to other objects, in particular not to those that may be queried from the 'allowed' objects because this requires unnecessary knowledge about collaborating servers. Instead, all requests should be sent to the 'allowed' objects, and these must delegate the request if necessary. Requested interfaces are a possibility to enforce this requirement as they completely hide which servers act behind the curtain.

Known Uses

- **Standard Template Library** [SL94]: To my knowledge, this is the best example for the application of the *Requested Interface* pattern to date. As Musser and Stepanov explain in section „Outline of the algorithm oriented approach" of their paper [MS94]:

  > „Start with the most efficient known algorithms and data structures, identify container access operations [...] on which the algorithms depend and abstract [...] those operations by determining the minimal behavior they must exhibit in order for the algorithm to perform a useful operation."

  According to this statement, STL iterators are designed from the point of view of algorithms and thus are, above all, *requested interfaces* of these algorithms. (Of course, the final design was the result of several iterations to balance what clients (algorithms) wanted and what servers (containers) could provide.) To reduce the number of requested interfaces, the iterators are classified into 5 categories: Input-, Output-, Forward-, Bidirectional-, and Random Access Iterators. All algorithms are generically defined in terms of one or more iterators from these categories, and the algorithm specifications explicitly state the required iterator types. Since the library was developed as a consistent whole, the container data structures 'happen' to provide exactly those iterators. This is, however, by no means necessary: By writing appropriate adapters, one can easily use STL algorithms for other data structures (e.g. persistent containers as provided by an object-oriented database system) that define different access methods than the STL.

- **CORBA** [OMG95]: This standard enables, among other things, *component based programming*. Its core component is an Interface Definition Languages (IDL) which is used to specify interoperable interfaces between clients and servers. Orbix' TIE approach for the implementation of CORBA conforming components [ORB96] is a nice example for the *Requested Interface* pattern: First, you specify the client's requested interface using IDL. Then you use an IDL compiler (a *Code Generator*) to implement the adapter layer which consists of a client side *Proxy* (stub) that sends requests over a network, a Basic Object Adapter (BOA) that receives the requests on the server side and interprets them, and a *Generic Bridge* (called TIE) which is called by the BOA and delegates the requests to the actual server for execution. The server is essentially independent of the CORBA system and may even be legacy code. Of course, for the delegation to work automatically it must conform to the requested interface (which is most easily achieved by letting the IDL compiler generate skeleton code for the server). However, this is not required, and you can always use a hand-coded bridge for the more difficult adaptation cases.

- **Object-Relational Mapping:** Sometimes an application that uses an object-oriented database for data storage must later be modified to use a relational database. Tools for object-relational mapping [KEL93] are designed to automatically generate the necessary adapters: By analyzing the definition of the objects to be stored (the *Requested Interface* of the application) they produce code to translate between the object-oriented and relational representations (in the application's language) and to store and retrieve the translated data (usually SQL).

- **VIGRA** [KÖTH97]: Our own framework for image processing, analysis, and visualization tries to translate the ideas of the STL into these fields. One of the main problems here is the existence of many different image formats and pixel data types. Under the traditional programming paradigms one keeps reimplementing algorithms whenever the data type changes. We use requested interfaces to implement algorithms that can run on any image format once we have written the necessary adapters (which is usually straightforward). We expect (and to some extend already witness) much greater reuse of parts of this framework outside the original context.

References

[BR97] D. Bäumer, D. Riehle: *„Product Trader“*, in: R. Martin, D. Riehle, F. Buschmann (eds.): „Pattern Languages of Program Design 3“, Addison-Wesley, 1997

[FOW96] M. Fowler: *„Reusable Object Models“*, Addison-Wesley, 1996

[GARL95] 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

[KEL93] A. Keller, R. Jensen, S. Agarwal: *„Persistence Software: Bridging Object-Oriented Programming and Relational Databases“,* ACM SIGMOD, 1993

[KÖTH97] U. Köthe: *„Reusable Algorithms in Image Processing“,* submitted

[LIEB96] K. Lieberherr: *„Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns“*, PWS Publishing Company, 1996

[MART94] R. Martin: *„Designing Object-Oriented C++ Applications using the Booch Method“*, Prentice Hall, 1994

[MEY94] B. Meyer: „*Object-Oriented Software Construction*“, Prentice Hall, 1994

[MS94] D.Musser, A. Stepanov: *„Algorithm Oriented Generic Libraries*“, in: Software - Practice and Experience, vol. 24, no. 7, pp. 623-642, 1994

[ORB96] *„Orbix 2 Programming Guide“*, IONA Technologies Inc. 1996

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

[REEN95] T. Reenskaug, P. Wold, O.A. Lehne: *„Working with Objects“*, Prentice Hall, 1995

[SM94] A. Stepanov, M. Lee: *„The Standard Template Library“*, Hewlett-Packard Laboratories Technical Report HPL-94-34, 1994