Servicing Components with Connector Systems

Joachim H. Fröhlich
Software Engineering Department
Johannes Kepler University of Linz
Altenbergerstr. 69, A-4040 Linz, Austria
+43 70 2468 9432
joachim.froehlich@acm.org

Manuel Schwarzinger
Racon Software GmbH Linz
Goethestr. 80, A-4021 Linz, Austria
+43 70 6929 1732
schwarzinger@racon-linz.at

Abstract

Interfaces bind components at dedicated points. Usually, despite their central role, interfaces are packed either with functionality-implementing components (call interfaces) or with functionality-using components (callback interfaces). Components that reference other components in order to implement or to use interfaces are directly coupled. This kind of coupling affects component implementations: integration of component services leads to implementations that are dependent on the component container or to a multiplication of implementation efforts.

We propose connectors as a mechanism to completely decouple components from each other and from their underlying component container. Connectors are special-purpose components that isolate component interfaces. Connectors optionally provide services to communicating components, e.g., checking bidirectional communication protocols (operation call sequences and data flows), exchanging components during run time, and parallelizing or synchronizing service requests in a non-intrusive manner. This frees components to focus on their core business. Connectors foster the standardization of interfaces, accelerate the development of components, improve the testability, portability and maintainability of component-based programs, and hence promote component markets. .NET provides an almost ideal implementation basis.

Keywords

interfaces, connectors, components, configuration, software architecture

1. INTRODUCTION

Mainstream component systems facilitate component-based programming but do not enforce it. This can partly be ascribed to the sensible wish for downward compatibility with object-oriented programming techniques and a white-box reuse style. This holds for .NET as well as for the Java.

In practice, object-oriented programs are usually organized in complex class graphs. More often than not, class libraries and frameworks expose many details at unwieldy, complex interfaces that are intended to cover various broad application scopes. This negatively impacts component architectures when classes are blurred with components, as in .NET. A component-based architecture calls for a different programming style that employs black-box reuse, interfaces (types) and contracts. Component services (such as controlling access rights, monitoring/profiling, object pooling, controlling concurrent access, and controlling transactions) are attached to components via a mix of marker classes (such as System.ContextBoundObject and System.EnterpriseServices.ServicedComponent) and attributes (such as ObjectPoolingAttribute and SynchronizationAttribute, both defined in namespace System.EnterpriseServices). Thus component services are applied intrusively and serviced components are directly coupled to the component container. Implementation of component services along a message sink chain with call interception, program reflection and container-dependent base classes in a robust and efficient way proves a major challenge [Löw05]. Although not directly referencing constructs of the component container, clients that reference serviced components (classes) become dependent not only on these components but also on the underlying component container.

It is fundamentally clear that components should be designed with high cohesion and low coupling. This leads to advantages well-known from proper class and method design. Functional diversity unfolded at component interfaces as lengthy or deeply structured public classes packed into large assemblies complicate the application and implementation of components. The resulting problems are best documented by complicated test procedures – most evidently for

.NET Technologies 2006

components wired into intrusive application servers. These components are loaded with operations that are foreign to their core business. To overcome these difficulties, lightweight component containers with minimal impact on applications have been emerging. Spring [Har05] serves as a prototypical example in the Java world; although Spring achieves decoupling through interfaces interposed between beans (components), interfaces are not treated as independent contracts.

Interfaces connect communicating components (or classes) and thus should be independent pivotal elements. In practice, however, interfaces are attached either to service-providing components or to service-requiring components. This asymmetry impairs specification, development and testing of independently installable components; this, in the long run, hampers the wide adoption of component technology. To overcome this obstacle, we propose an architectural style where every pair of interacting components is fully separated with independent, special-purpose components that isolate component interfaces and optionally implement nonfunctional component services.

The paper is organized as follows: Section 2 details the goals of the proposed architectural style. Section 3 presents basic concepts of the connector/component architecture style. Section 4 sketches the application of connectors. Section 5 presents basic connector variants on which extended variants in Section 6 build. Related work and consequences conclude the paper.

We back the presentation with code snippets in .NET/C# and semantically rich system diagrams documenting real implementations by abstracting away unnecessary coding details rather than describing the design of prospective systems. The whole work is based on experience gathered with experimental implementations and with several variants of a generic program for analyzing data streams [Edl05], [Frö05], [Frö06].

2. GOALS

We seek an architectural style that enables components to focus on their business without being distracted by intrusive component containers. Such a style must enable economically feasible structuring of general-purpose programs as well as domain- or application-specific programs. Thereby a program is either self-contained or embedded in a component container (application server). The architectural style must facilitate separate specification, implementation, testing, guarding, installation, substitution and monitoring of components and their interactions. Component services must be transparent as far as possible. The architectural style must enable independent component evolution in in-house and open-market situations. For practicability, existing container technologies, if needed at all, should be supplemented rather than be replaced. The mechanisms enabling this architectural style must be configurable and thereby provide only as much flexibility and cost only as much in resources as needed in various stages of a project, such as development, test, launch or production stages.

3. CONNECTOR BASICS

Interfaces rather than components carry software architectures. This contrasts with the usual view where software architectures focus on components and their interactions but tend to overlook the importance of component interfaces. We view software architectures as systems of component interfaces that service components. Like components, component interfaces are physical (i.e., binary) and identifiable concepts that we call *connectors*. Technically, a connector contains at least one interface in the sense of the programming language construct of the same name. All operations declared in interfaces of a connector form a functional closure; i.e., operations of connector interfaces use only parameters of basic data types, interfaces contained in the same connector or, in special cases, interfaces of neutral parts of .NET's framework class library, like System.Collection and System.Configuration. Logically, a connector specifies functional and nonfunctional properties of components using or implementing interfaces. Additionally, connectors may monitor, guard or change operation invocations and data transmissions across component boundaries as long as they conform to the contracted communication protocol without distracting adjacent components. Connectors do not execute any business- or application-specific functions.

Connectors define the points of variation at which components can be plugged in. At least two independent components communicate across the boundary that a connector establishes. We call them *functional components* (components for short where it is unambiguous) because they directly or indirectly implement functions that comprise the core business of a program. We speak of a *symmetric connector* when a functional component on the client side of connector uses the same interface(s) as the functional component on the provider side for communicating with each other. We speak of an *asymmetric connector* when a client component and a provider component use different interfaces and the connector maps interface concepts during communication. This article focuses on symmetric connectors.

Clutches serve as a metaphor for connectors. Clutches couple functional components, i.e.,

(driving) client components to (driven) provider components where these components might change their roles during communication. Thereby clutches transfer physical forces (data) in both directions, from clients to providers and vice versa. Real clutches optionally contain springs that dampen the transmission of exceptional forces. Connectors as defined above offer similar convenience. For example, they can log unspecified exceptions and map them onto exceptions specified in the connector because exceptions crossing component boundaries are part of the communication protocol. Another example is a connector that prohibits inadmissible input data or erroneous operation call sequences, e.g., faulty communication protocols.

Figure 1 illustrates a program that is minimal in terms of components and connectors.

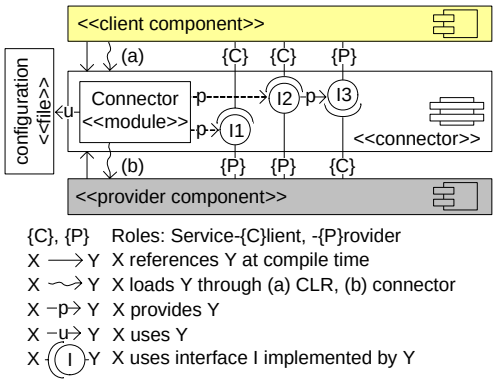

Figure 1. Connector and functional components

The connector in Figure 1 completely channels the communication between the sole service client and the sole service provider which includes the creation of service-providing objects. The connector module[1] processes data from the configuration file in order to relieve service clients as well as the connector itself from specifying concrete classes in the program code. The resulting constellation is characterized as follows:

- Components do not depend on each other.
- Components depend on connectors.
- Connectors do not depend on components.

The compilation procedure reflects this constellation:

```
csc /out:Connector.dll /t:library ...
csc /out:Provider.dll /t:library /r:Connector.dll ...
csc /out:Client.exe /t:exe /r:Connector.dll ...
```

Thus the architecture of a program can be modeled as a system of connectors that embed functional components (see Figure 2).

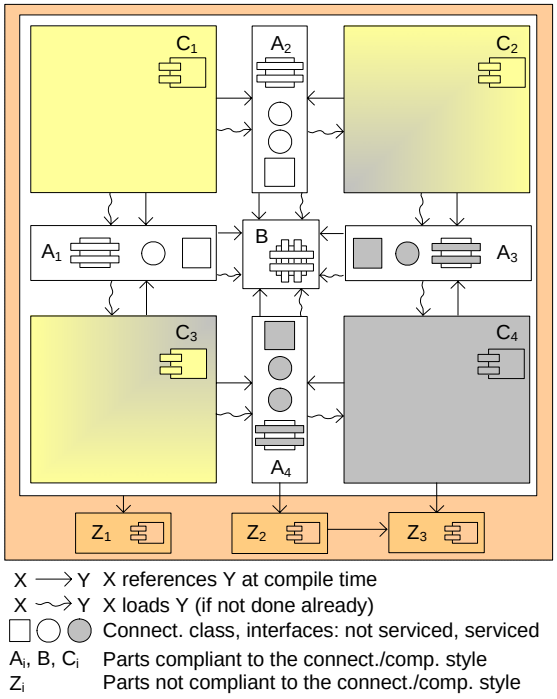

Figure 2. A connector / component architecture

Figure 2 depicts components that follow the architectural style (A_i, B, C_i) and those that do not (Z_i). Functional components (C_i) are connected to a central connector manager (B). The connector manager provides for a communication interface by which external clients can monitor and control connectors (A_i). In order to control a connector, interfaces must be wrapped in proxy objects that pre- or post-process operation calls crossing component borders as indicated in Figure 2 for connectors A_3 and A_4. We call connectors *heavy connectors* if they wrap interfaces in order to transparently hook component services like logging, profiling, security checks and protocol checks. We call connectors *light connectors* if they contain only interface declarations. The run-time overhead of light connectors is negligible. Light connectors can be exchanged for type (interface) compatible heavy connectors just by program reconfiguration before run time.

Another type of connectors not sketched so far are multiple-part connectors (see Figure 3).

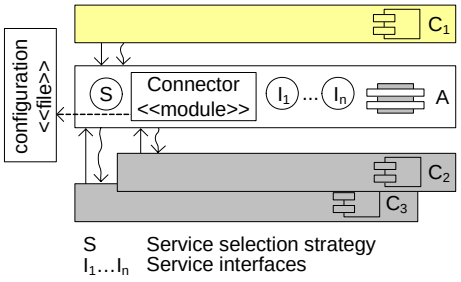

Figure 3. Multiple-part connector

[1] Only classes with (static) class members are modules.

Upon exceeding a certain breadth, the functional interface of a connector (I_1 … I_n in Figure 3) can be implemented via several components (C_2 and C_3 in Figure 3) instead of just one component. These components build a group. A component group is defined by a common connector and one or more partitioning attributes. Each component of a group must publish a value for each partitioning attribute. The combination of attribute values characterizes a component within a component group. Thus partitioning attributes are used to diversify components. Diversification narrows the application scope of a single functional component, which eases its implementation while raising the domain-specific service level. A *multiple-part connector* is a connector that can bind (load) more than one interface-implementing component and uses a strategy (S in Figure 3) to choose a component whose attribute values best fit the client requirements. The strategy is provided either by the connector as part of the contract or by a client component. In contrast, *single-part connectors* bind (load) at most one interface-implementing component. Table 1 provides examples of multiple-part connectors and partitioning attributes.

Mp connector	Attributes
String matchers	automaton, e.g., NFA, DFA
Report generators	file format, e.g., PDF, HTML
Memory systems	access time, durability
Numeric systems	accuracy, precision, run time

Table 1. Some attributes of multiple-part connectors

4. A CONNECTOR IN TEST USE

Before delving into various extensions of connectors, let us examine a typical application scenario as seen from a service using (client) side. You can find the complete C# code of an almost identical implementation elsewhere [Frö06]. Figure 4 sketches the architecture of the program.

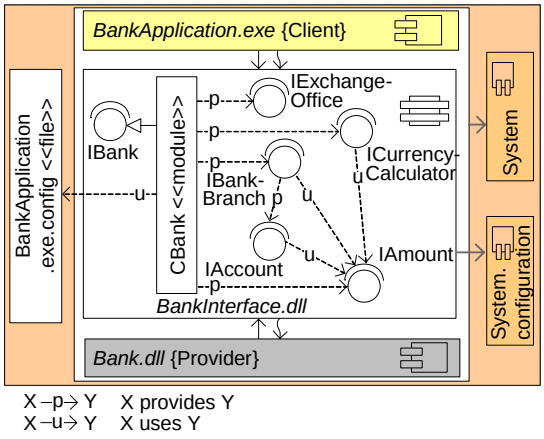

Figure 4. Architecture of a simplistic bankapp

The program implements a simplistic bank with several branches, accounts and customers. These concepts are directly reflected in the connector, whose interface operations build a functional closure, i.e., do only involve interfaces declared in this connector and basic data types:

```
namespace BankInterface { // Connector
  public interface IBank {
    void Provide(out IBankBranch branch);
    void Provide(out IAmount money, double val, string cy);
    ...
  }
  public interface IBankBranch {
    IAccount SetupAccount(IAmount initialValue);
    IAccount SetupAccount(); // initialValue= 0.00 EUR
    bool Transfer(IAmount money,
      IAccount source, IAccount target);
    ...
  }
  public interface IAccount {
    string Owner { get; set; }
    bool Deposit(IAmount money);
    ...
  }
  ...
}
```

The program applies a light, single-part connector (BankInterface.dll); i.e, the connector provides no services other then automatically loading one bank implementation (Bank.dll) at a time during first access by a bank client (BankApplication.exe). The concrete bank implementation is configured before run time, e.g., in the standard configuration file of a .NET application:

```
<configuration><appsettings>
  <add key="Provider" value="Bank.dll"/>
  ...
</appsettings></configuration>
```

An application scenario taken from the client illustrates the coding style, which resembles that prevailing for clients of COM components. Several amounts of money are transferred from different source accounts to a common target account:

```
namespace BankApplication { // Client
  // Set up bank branch, target account
  IBank bank= CBank.Get();
  IBankBranch branch; bank.Provide(out branch);
  IAccount target= branch.SetupAccount(); // 0.00 Euro
  IAmount amount1, amount2, ...;

  // Setup accounts
  bank.Provide(out amount1, 1000.00, "EUR");
  IAccount source1= branch.SetupAccount(amount1);
  bank.Provide(out amount2, 1500.00, "EUR");
  IAccount source2= branch.SetupAccount(amount2);
  ...
  // Transfer money
  bank.Provide(out amount1, 500.00, "EUR");
```

```
  bank.Provide(out amount2, 800.00, "EUR");
  ...
  branch.Transfer(amount1, source1, target);
  branch.Transfer(amount2, source2, target);
  ...
}
```

The service provider (Bank.dll) can be exchanged without changing the client's implementation. For instance, a test stub that is applied during development and component test of a bank client can be replaced with a production version for integration tests. Moreover, the light connector can be replaced with a type (interface) compatible heavy connector. For example, from a technical point of view the heavy connector checks whether the client component passes to the provider component objects that the same provider has created before. From a business point of view this check is necessary, e.g., when a bank branch charges an account. The account must be set up by the same bank branch or by one of the other branches of the bank. We assume this integrity check to be necessary for every bank; hence it is part of the bank contract.

5. BASIC CONNECTORS

5.1 The Lightest Connector

An application scenario as simple as the sketched bank program is typical of tests of functional components. Although light connectors are by no means restricted to test scenarios, they obviously demand easy connector implementations. This directly leads to the question of how to design the lightest connector (see Figure 5).

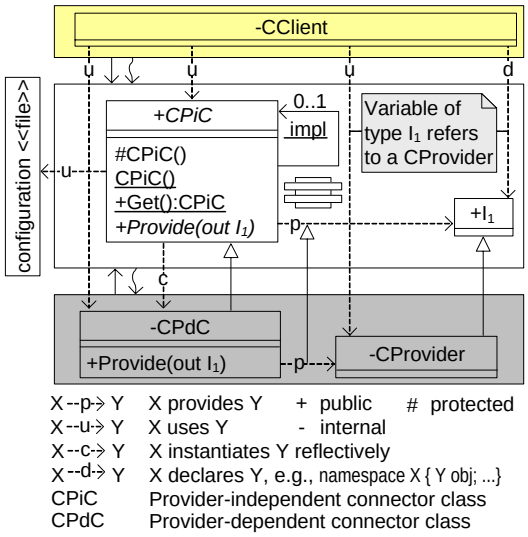

Figure 5. The lightest connector

Besides combining logically coherent interfaces into a separate component, every connector must implement just one nonfunctional task: the establishment of the first connection between a service-using and a service-providing component while preserving their independence as well as its own independence. For this purpose a connector contains what we call a *provider-independent connector class* (CPiC in Figure 5). On the one hand this class is a module with (static) class methods and variables for loading and anchoring a provider; on the other side it is a type declaring factory methods [Gam95] for letting providers decide which objects to deliver as roots of business process chains (sessions). Thus each provider must subclass exactly one *provider-dependent connector class* (CPdC in Figure 5) per supported connector.

The provider-independent connector class uses reflection techniques to create the sole object of this class (a singleton [Gam95]), the connector object. This object is created automatically in the background during the first access to a provider (triggered by, e.g., `bank= CBank.Get()` in the bank application and executed by the class constructor of the provider-independent connector class) immediately after the provider component specified in the configuration file is loaded. Once the connector has supplied the connector object, a client queries it for the first business object by means of a factory method (via, e.g., `bank.Provide(out IBankBranch)` in the bank application) declared in the provider-independent connector class and implemented in the provider-dependent connector subclass.

The implementation of the managing stuff of a light connector is delightfully cheap. It costs about 10 lines of code executed only once per provider component upon first access (compare with the CPiC `CBank` in CBank.cs, directory Bank.src\BankInterface [Frö06]). All other operation calls across a light connector, i.e. across interfaces in the sense of the programming language, do only cost as much as invocations of instance function members [Hej04]. Thus light connectors completely separate communicating functional components with no run-time overhead.

5.2 The Lightest Heavy Connector

Heavy connectors factor out nonfunctional services from functional components. For this purpose, heavy connectors wrap interfaces in proxy classes [Gam95]. They provide hooks for affixing component services like profiling and protocol checks to both call interfaces and callback interfaces. Connectors wrap both interface types with the same procedure but at different moments: call interfaces on the way out of an interface function and callback interfaces on the way into an interface function. Figure 6 sketches the structure of a heavy connector.

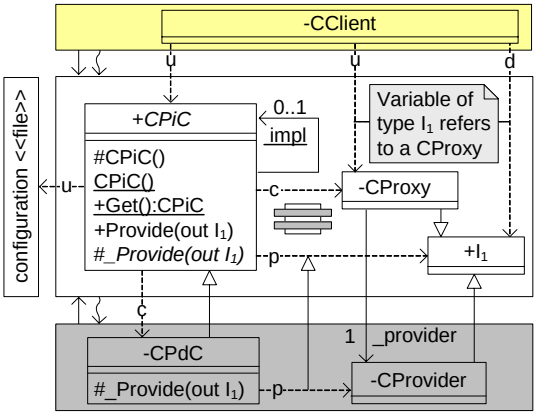

Figure 6. The lightest heavy connector

The decisive difference compared to a light connector is that all function calls of a client component first activate a wrapping function implemented in the heavy connector before they activate a function in a provider component. The prerequisite for wrapping all operation calls crossing a connector is template methods [Gam95] in the provider-independent connector class, as the following code excerpt demonstrates by wrapping the root object of a business process chain (implementing a call interface):

```
namespace Connector {
  public abstract class CPiC { // connector module
    public void Provide(out I1 p) { // the template method
      I1 provider; // the service provider
      this._Provide(out provider);
      p= new CProxy(provider); // wrap call interfaces on the
        // way out from a provider to a client
    }
    protected abstract void _Provide(out I1 provider);
      // the primitive operation of the template method
    ...
  }
  public interface I1 { ... }
  internal class CProxy : I1 {
    internal CProxy(I1 provider) { this._provider= provider; }
    ... // methods wrapping I1 functions
    private I1 _provider; // the wrapped service provider
  }
  ...
}
```

Syntactically, proxy objects and connected component services are completely hidden in the connector and therefore invisible to functional components.

5.3 The Lightest Multiple-Part Connector

Multiple-part connectors allow the differentiation and installation of several provider components that offer alternative or variant services. Moreover, heavy multiple-part connectors enable a different class of component services, like multiplexing (or parallelizing) of service request among several provider components and graceful failover from one service provider to another. Figure 7 sketches the structure of a light multiple-part connector.

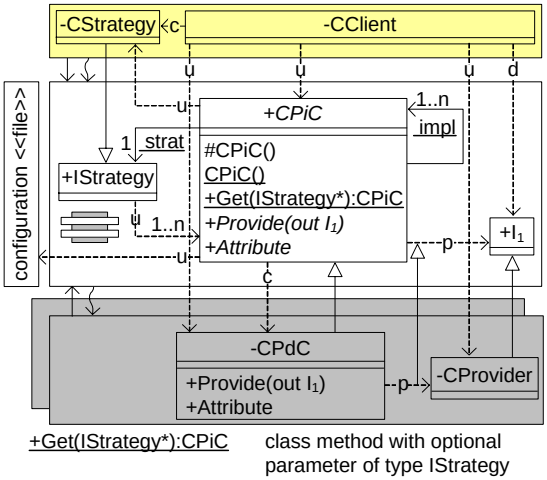

Figure 7. The lightest multiple-part connector

Provider components to hook into a multiple-part connector are specified in the configuration file with multiple-value entries such as

```
<configuration><appsettings>
  <add key="Provider" value="Bank1.dll;Bank2.dll"/>[2]
  ...
</appsettings></configuration>
```

All these provider components share one (structured) interface, i.e., one connector, and usually vary in their implementation with regard to at least one component attribute. A multiple-part connector offers clients the chance to dynamically select one of the configured providers. To make this work, the provider-independent connector class forces provider-dependent subclasses to return values that characterize their business with regard to a differentiating business attribute. The strategy pattern [Gam95] lends itself for a flexible implementation of the selection algorithm. In the context of the bank example, clients can now choose among several banks applying different interest and portfolio strategies.

6. EXTENDED CONNECTORS

Connectors can be extended at four sides (see Figure 8):

(a) Client side: several functional components use one connector.
(b) Provider side: several functional components provide alternative or supplementing services.

[2] Of course, the type-safe way for specifying an arbitrary number of provider components would be an xsd:element with a multiplicity range of minOccurs="1" maxOccurs="unbounded".

(c) Connector service side: several special-purpose components register component services for communicating functional components.
(d) Connector managing side: the behavior of a running program is monitored and controlled in terms of connectors and components.

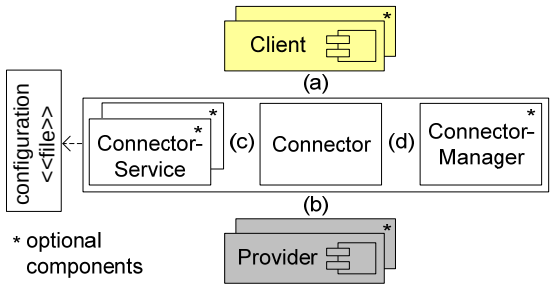

Figure 8. Connector extensions

6.1 Extending the Connector Service Side

Component services can be implemented in proxy classes directly in a heavy connector (see 6.1.1) or sourced out into separate classes in separate components (see 6.1.2).

6.1.1 Implementing Services Directly

Component services can be implemented with minimal effort directly in a connector. This implementation style well suits special component services like checks of highly specialized communication protocols while obviously compromising reusability of rather general applicable component services like logging[3]. To give an impression of a component service, we sketch a part of the life cycle management. The heavy connector checks objects that client components pass as operation parameters to a provider component for creation by the same provider.[4] Again, we demonstrate this for the sample bank application introduced in Section 4: A bank can only service its own bank accounts. We assume that this constraint is part of the contract holding for all banks. If this is true, then the connector is the place to implement the constraint. On violation of this constraint the connector throws a protocol exception:

```
namespace BankInterface { // Connector
 internal class CBBProxy  // Class Bank Branch Proxy
   : IBankBranch {
  public IAccount SetupAccount() { // public protocol
   IAccount provider= this._provider.SetupAccount();
   CAProxy accountProxy= new CAProxy(provider, this);
   this._issuedObjs.Add(accountProxy);
  }
  ... // more methods wrapping IBankBranch operations
  internal CBBProxy(IBankBranch bankBranch) {
   this._provider= bankBranch;
  }
  internal bool HasIssued(CAProxy proxy) {
   return this._ issuedObjs.Contains(proxy);
  }
  private IBankBranch _provider;
  private Utilities.ISet _issuedObjs= new Utilities.CSet();
 }
 internal class CAProxy // Class Account Proxy
   : IAccount {
  public void Withdraw(IAmount money) { // public protocol
   if (!this._creator.HasIssued(this))
     throw new CProtocolException("unknown account");
   this._provider.Withdraw(money);
  }
  ... // more methods wrapping IAccount operations
  internal CAProxy(IAccount provider, CBBProxy creator) {
   this._provider= provider;
   this._creator= creator;
  }
  private IAccount _provider;
  private CBBProxy _creator;
 }
 ...
}
```

6.1.2 Implementing Services Indirectly

Proxies that delegate requests for component services lead to service implementations that are extensible and reusable in the context of several connectors. Such proxies signal changes of relevant program states (method calls and returns across component boundaries) and delegate the provision of services to observers [Gam95] implemented in separate components. This raises the question of the sequence in which component service should be applied.

In general, component services can be applied in any sequence because they have no side effects. From a practical point of view, of course, it is useful to check, e.g., whether a client is allowed to use a provider before checking the communication protocol in case the two component services are implemented separately. Likewise, the communication protocol should be checked before a client is allowed to ask for exclusive usage of a provider. Thus ideally component services, their order of application and the associated connectors are specified in a program configuration file and set up with reflective programming techniques.

[3] During development and test phases of the generic data stream analyzer (mentioned in the Introduction) a heavy connector tests the communication protocol between the component providing the business logic (data stream pattern matcher) and various user interface components [Frö05]. The connector applies the state pattern [Gam05].

[4] This service is indeed rather generally applicable. It checks an integrity constraint for components that cast types of parameter objects to component-specific type implementations (classes).

6.2 Extending the Connector Managing Side

All connectors of a program may be connected at a central point which we call the connector manager. The connector manager is the place for querying and changing the state of a program in terms of components and connectors either from inside the program through API calls or from outside the program, e.g., through a web service. In particular the connector manager enables

- loading of functional components
- unloading of functional components
- switching component services on or off
- querying for current components and connectors
- querying for interaction states and histories
- coordinating several connectors

From a technical point of view, the most interesting feature of a connector manager is the coordination of connectors with regard to unloading stateful functional components. This requires life-cycle management of components.[5] A connector attached to a stateful functional component must check the communication protocols for each usage scenario (per business process chain) and indicate the functional component as being in a state allowing the component to be unloaded, as it is usually in initial states, end states, or error states (0-states for short). This requirement holds for all connectors directly attached to a component as well as for all dependent connectors.[6] Consider the program sketched in Figure 9.

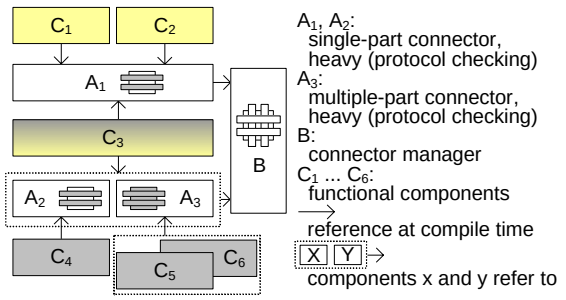

Figure 9. Managing connector systems

Provided that it is useful to unload C_3, all dependent connectors (A_1, A_2 and A_3) have to confirm dependent functional components to be in 0-states. These connectors contain at least 4 state machines that check the communication protocols between components

- C_1/C_2 and C_3,
- C_3 and C_4,
- C_3 and C_5, and
- C_3 and C_6.

Note that thereby we assume C_1 and C_2 to take part in a common usage scenario (session); i.e., they share one business process chain and therefore one protocol-checking state machine. Inversely, one component could take part in several usage scenarios of a connector so that, e.g., two or more state machines could be active in A_2 checking two or more applications of C_4 by C_3. Technically, unloading a component requires it to be installed in separate application domain (System.AppDomain) [Gun02], i.e., in a separate .NET process, which of course increases communication costs due to marshalling all calls between application domains.

Besides, the connector manager factors out code common to all connectors, such as that for loading and unloading functional components and standard component services such as logging operation call sequences.

7. RELATED WORK

This article focuses on physically separate connectors as a means to connect and at the same time to decouple components in the context of coherent programs or program parts.

Some of the presented concepts suggest concepts prevailing in the context of distributed programs. Here connectors are manifested as parts of the underlying infrastructure, e.g., in the form of networking protocols, pipes, SQL links between a database server and a database application program, event buses, and message brokers [Clem03], [Meh00], [Sha96]. Service–oriented architectures (SOA) provide the plumbing for the integration of components running on different technological foundations [Sko05]. Component interfaces are published, queried and translated into executable code for calling services across the Internet.

Connectors as separate compilation and deployment units of coherent programs are scarcely discussed elsewhere. In a coherent program, connectors usually occur at the abstraction level of a programming language as shared variables, buffers and procedure calls [Meh00], [Sha96]. This strongly contrasts with connectors at the architectural level of a program as discussed in this paper. At the architectural level a connector must not to be confused with a façade [Gam95] or a mediator [Gam95]. A façade provides a unified interface to a set of interfaces in a sub-

[5] Strictly speaking, only component instances can have state in a running program because components are just binary deployment units. As this should be clear from the context, we speak of stateful components.

[6] A functional component that implements (interfaces declared in) several connectors might indicate low binding or hint incomplete connector interfaces and so disobey the requirement for functional closure.

system. This is usually done when a class applies business logic to orchestrate (instances of) other classes. Thus a façade is coupled to the covered subsystem. A mediator coordinates interactions of a group of objects. Thus a mediator executes application-specific functions. A connector with its distinct orientation on improving nonfunctional system properties, such as reliability, adaptability, and testability, is independent of adjacent functional components and does not execute any application-specific functions.

However, the idea of including related interfaces in separate components is not new. Szyperski et al. [Szy02] emphasize the importance of viewing interfaces in isolation from any specific component that might implement or use such interfaces. Further-reaching concepts or implementation techniques are not discussed. In the context of .NET, Löwy [Löw05] suggests assemblies with interfaces to parallelize the development of adjacent components. Wienholt [Wie03] proposes a similar technique to shorten load time of assemblies and to save memory. He puts frequently and occasionally used types of an assembly into different netmodules[7] and separates them by netmodules that consist only of interfaces, which leads to multiple-module assemblies. This can also be achieved with the connector/component architectural style.

Interfaces play an important role in the realm of lightweight component containers; Spring [Har05] is a good example. Spring decouples components (beans) in the form of classes by externalizing the creation of instances of collaborating classes and injecting them at dedicated points of the class to be configured (dependency injection). Collaborating classes are expected to implement well-defined interfaces. Although the work on connectors presented in this article shares many of the goals of Spring, such as isolated component tests, externalization of component dependencies (in configuration files), and design in terms of the application domain (rather than in terms of the implementation domain or a mix of both), the solutions move in different directions. Spring abandons subclassing for Spring-conform components due to reflective programming techniques. In contrast, a functional component in the role of a service provider has to implement a provider-dependent subclass per connector, even though this subclass contains only domain-specific methods (in a special syntax). Spring does not support the transparent injection of non-functional services between communicating components. Spring has no notion of multiple-part components and provides no special means for coordinating semantic operations attached to related interfaces either in the form of protocol checking services or in the form of a connector manager for monitoring and controlling running programs.

[7] A netmodule is a *raw* module that must be associated with a full-fledged component (assembly) prior to deployment.

8. SUMMARY AND CONSEQUENCES

Connectors as discussed in this article are special purpose components that embody boundaries of functional components in the form of binary contracts. This allows functional components to focus on their core business. Moreover, functional components can be

- developed in several alternate or supplementary variants
- specified and tested separately
- relieved of intermingled nonfunctional services like logging, caching and checking communication protocols
- dynamically monitored and controlled if a connector manager supervises the connector system

Connectors may interpose nonfunctional services between functional components in a completely non-intrusive manner. This is achieved by means of a pattern language [Cun87] that combines several design patterns [Gam95], such as Factory Method, Template Method, Proxy, Strategy, State and Observer, and by encapsulating these patterns in special components (connectors). Classes of functional components shed any special base types (such as System.ContextBoundObject) or attributes (System.Attribute) for profiting from component services. Certainly these techniques can be used for implementing component services within connectors. Proxy classes in connectors expose suitable method call joint points to implement component services as aspects in the sense of AOP (aspect-oriented programming). Services that have well-defined effects on particular operations support the use of AOP [Mur01]. This is the case, e.g., for synchronization and accounting services but not for checks of complex, application-specific communication protocols. Due to the localization of services in connectors, functional components remain unchanged regardless of how services are intercepted, such as with context bound objects, code generation, modification of IL (intermediate language) code or .NET's profiling API.

If a program does not depend on a nonfunctional service, a heavy connector can simply be replaced with an interface-compatible light connector without changing the implementation of adjacent components. The implementation of the skeletal structure of a light connector is almost for free with regard to both development time and run-time efficiency while

still providing the fundamental advantages of connectors, i.e., separate specification, testing and development of functional components. The call of an operation across a light connector costs only as much as a call of an instance function member. The one-time loading of a component immediately before the first operation call does not impair performance in the long run. Even heavy connectors can boost the overall performance of a program. For instance, checks of communication protocols (pre- and post-conditions, invariants, operation call sequences) at clear-cut, contracted and rather stable component boundaries concentrate on essential and coherent system parts (components) while abstaining from checks of rather quickly changing implementation-specific (i.e. component-specific) objects scattered around the program.

Even demanding services like parallelizing service requests in a blocking or non-blocking manner among several service-providing components can be included in a heavy, multiple-part connector without distracting adjacent components. However, this holds only for unidirectional data flow where service clients just trigger service providers concurrently without needing any calculated value from them. Bi-directional data flow demands connectors that buffer data returned by providers and a special interface enabling clients to fetch this data for each provider. This exceeds the capabilities of symmetric connectors and moves towards asymmetric connectors that map deviating client and provider languages in terms of deviating interfaces.

In any case, separate connectors in different extension stages supply effective, non-intrusive mechanisms to solve challenges and issues in developing, testing and quality assurance of software components. Both isolated connectors and connector systems promote architecture-centric development of programs with variants. Connectors lend themselves for gluing common components and varying components with predictable capabilities even in order to build high-quality product families (product lines) [Wei99]. At the same time, connectors raise the productivity of component developers, testers and architects. Variants of a generic data stream analyzer [Frö05] and several experiments prove the practical feasibility of the connector/component architecture style. Coordinated life-cycle management (protocol checking) of several components is a key issue of further work.

9. REFERENCES

[Clem03] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford J.: Documenting Software Architectures – Views and Beyond. Addison-Wesley, 2003

[Cun87] Cunnigham, W.: Design Methodology for Object-Oriented Programming. OOPSLA'87, ACM SIGPLAN Notices 23 (5), 1987

[Edl05] Edlmayr, J., Fröhlich, J.H., Schwarzinger, M., and Stranzinger T.: Components for All Cases. (in German) SIGS Datacom OBJEKTspektrum 1/2005 (part 1), 2/2005 (part 2)

[Frö05] Fröhlich, J.H., and Schwarzinger, M.: Treating Interfaces as Components. In IVNET'05 ISBN 972-8688-31-8, 2005

[Frö06] Fröhlich, J.H., and Wolfinger, R.: .NET Profiling: Write Profilers with Ease Using High-Level Wrapper Classes. MSDN Magazine 21 (5), 2006

[Gam95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995

[Gun02] Gunnerson, E.: AppDomains and Dynamic Loading. http://msdn.microsoft.com/library/en-us/dncscol/html/csharp05162002.asp, 2002

[Har05] Harrop, R., and Machacek, J.: Pro Spring. Apress, 2005

[Hej04] Hejlsberg, A., Wiltamuth, S., Golde, P.: The C# Programming Language. Addison-Wesley, 2004

[Löw05] Löwy, J.: Programming .NET Components. O'Reilly, 2005

[Meh00] Mehta, M.R., Medvidovic, N., and Phadke S.: Towards a Taxonomy of Connectors. ICSE'00, conf.proc., Limerick Ireland, 2000

[Mur01] Murphy, G.C., Walker, R.J., Baniassad, E.L.A., Robillard, M.P., Lai, A., and Kersten, M.A.: Does Aspect-Oriented Programming Work? CACM 44 (10), 2001

[Sha96] Shaw, M., and Garlan, D.: Software Architecture-Perspectives on an Emerging Discipline. Prentice-Hall, 1996

[Sko05] Skonnard, A.: SOA: More Integration, Less Renovation. MSDN Magazine 20 (2), 2005

[Szy02] Szyperski, C., Gruntz, D., and Murer, S.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 2002

[Wei99] Weiss, D.M., and Lai, C.T.R.: Software Product Line Engineering. Addison-Wesley, 1999

[Wie03] Wienholt, N.: Maximizing .NET Performance. Apress, 2003