

An evolutionary Integration Approach using dynamic CORBA in a typical banking environment

Markus Pohlmann
GAD

Department SRV/ARC
Weseler Strasse 510, 48163 Münster, Germany
Markus.Pohlmann@GAD.De

Marc Schönefeld
GAD

Department SRV/ARC
Weseler Strasse 510, 48163 Münster, Germany
Marc.Schoenefeld@GAD.De

Abstract

The technical steps for migration of a large-scale banking application towards a new and open application architecture are discussed. Business needs and - derived from that - technical requirements demand a new flexibility in application design. Flexible solutions can be better built on solid ground, so the importance of a solid architectural foundation like CORBA is emphasized. The development of the CORBA/BCI-Bridge includes several middleware design and implementation recipes which may inspire other software architects in comparable projects.

1. Motivation: Migration of a large-scale banking Application

The transition phase banking software is currently undergoing can be best described by the following quote from the UBILAB [14] homepage.

“ Success in banking increasingly depends on the ability to master information technology (IT). The reason is simple: information is both the raw material and the product of a bank. But because IT changes at an ever faster rate, bringing it in line with banking’s long-term objectives becomes more difficult with each passing day.”

Today’s business challenges topics are Globalization and Resegmentation of markets, business deregulation, changing customer and product profiles as well as newly emerging information opportunities. From the strategic business goals like Time, Cost and Quality Leadership[11] (see figure 2) the tactical implications to the technological application environment can be derived. This means in other

words, that life-cycles of the different parts of banking software products have changed. Front-end components that use short-living technologies (WAP, UMTS, Browser and the “next best thing”) stay in service for about two years, the business processes on the middle-tier exist normally for 5 to 10 years but have to be adaptable to short term changes (e.g. new tax regulations). This helps the client bank to offer new front-office products with a reduced time to market latency. Because the back-end technology has the longest life-cycle of almost two decades, powerful investment intensive solutions with emphasis on scalability, quality and performance can be used.

1.1. The scenario

Reactions to Contextual forces like new legal regulations and changing business processes put the structure of monolithic software systems to the test. Monolithic software systems are often unable to adapt to changing business requirements, because the embedded workflows are tightly coupled to the atomic services and therefore do not offer adequate flexibility to depict emerging business processes.

1.2. Objective: Evolution to an Open Architecture

The objective is to make a seamless technical migration to an open architecture platform, that opens the opportunity to separate the workflow layer from the atomic services (in this context we call stateless functions *services*). Sneed[3] describes three (see figure 1) different approaches of software migration from past to the future, the first - and the most expensive - is to develop the whole system from scratch, the conversion approach replaces the existing programs by classes and functions by methods. For a migration

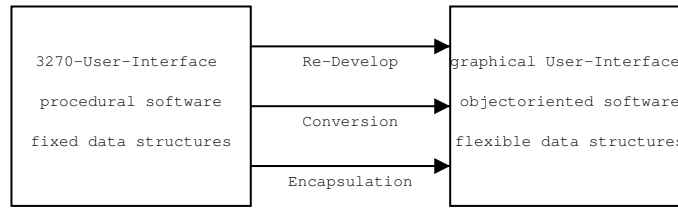


Figure 1. Three Ways of Migration

phase the whole system would have to be doubled, which would be very capital intensive. We chose the third alternative, which is based on a encapsulation strategy of the legacy system utilizing an object oriented wrapper, so the investments in the underlying system can be protected.

1.3. The Strategy

Without using such a encapsulation strategy, GAD is forced to change both its remote client application and its centralized application. There is a high risk in this unwieldy scenario on impacting customer service levels by thrusting a completely new environment upon front-office service personnel. CORBA technology offers an efficient means to upgrading its banking application to new distribution channels so it can use current information technology, entering the age of mobile and internet banking. Therefore GAD uses the “CORBA-way” as an important bridging technology to take advantage of the existing application investments[5] while providing a path to the future and filling the gap between requirements and functionality.

1.4. The Technology

The new CORBA-based solutions will allow GAD to maintain its existing backend application while replacing its legacy IBM 4700-based equipment and an OS/2-proprietary terminal emulation with a solution based on object technology. GAD chose CORBA-technology because of its interoperability benefits in large-scale implementations in favor of proprietary solutions. This anticipates the need to integrate new technologies, that will - due to the growing importance of CORBA on the market - offer IDL-specification and therefore interoperate seamlessly in a future-proof application architecture which aligns along a “Service-Bus” (see figure 4). New short-time living work flow processes can be assembled from re-used long-time living existing services. Therefore the time-to-market-process becomes shorter(see figure 2). The costs are lowered because re-use is possible, and using existing services instead of a newly implemented services can be used. The quality of the services is increased because they are used more often, and

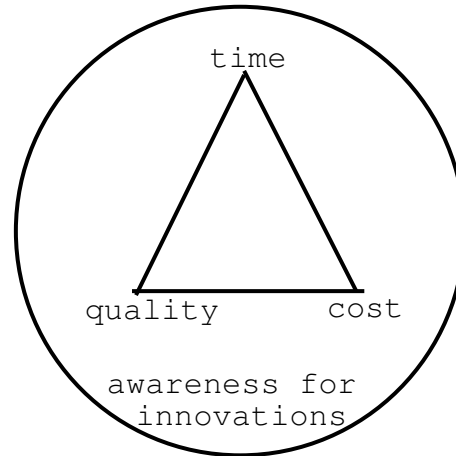


Figure 2. Time, Cost and Quality

therefore economies of scale and scope as well as learning curve-effects apply.

2. The BCI-Environment

GAD provides its client Volksbank institutes with the “BB3” banking application. The core BB3-architecture is depicted in Figure 3. For backend processing BB3 uses IMS databases and transaction monitoring functionality. The standard IMS-functionality provides only a simple stateless ‘request-reply’ model, which is not a sufficient pattern for banking software. This was the reason to extend this basic pattern with BCI (Banking Communication Interface).

2.1. Features of the BCI Runtime Environment

BCI provides session management framework, which takes care of routing the requests from clients to the responsible IMS system, handling user sessions, authentication, authorization and simple exception handling. The BCI-semantic framework introduced status information, input and output blocks to separate session data from business data. BCI also provides data-only-marshalling for different

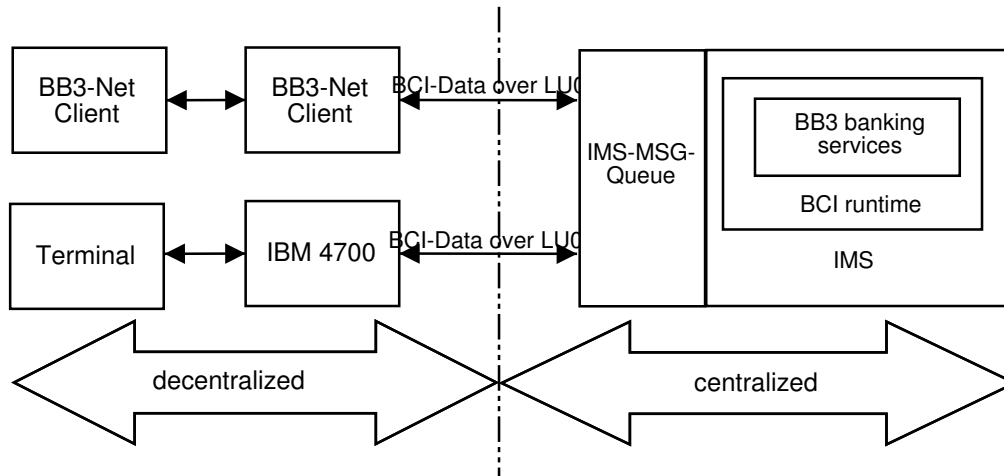


Figure 3. productive BB3-architecture

built-in data types. The further features of the BCI runtime environment are the following:

2.2. BCI-Data handling

Reducing bandwidth was a primary goal when data transmission facilities were an expensive resource. Therefore the BCI-backend transmits raw byte data to the client. Code page translation has to be explicitly handled by the client. There is no additional type and length information available as this meta data is deployed separately during the initial system installation process. As is obvious, this produces a lot of dependency problems when upgrading clients and backend system in sync. The meta information for each request-type is stored in a data dictionary as an development repository.

2.3. Network Protocol

The protocol between IBM mainframes and 4700 systems was the standard of the IBM-proprietary world, LU0, which does not fit well into the internet driven information age, where the standard protocol is TCP/IP.

3. GADs strategy: System Evolution

With the CORBA-BCI-Bridge GAD uses a “system evolution approach” [1, S.5]. CORBA was chosen as backbone middleware to provide a solid platform for building a bridge to the legacy system BB3. The function of the CORBA-BCI-Bridge is the role of an interface engine to the BCI-Based legacy. The “evolutionary approach” can be described by the following characteristics:

- Limit the range of used technologies to a few, but efficient solutions
 - Prefer semantic-rich middleware in favor of byte-level-middleware (for instance ITOC)
 - Build a integrated, high-available and scalable Servicebus-Infrastructure
- Integrated** , every type of software, including COTS products that export an IDL-interface can be service-enabled and integrated to the servicebus-architecture
- Available 365x24x7** , Proven Fail- and Takeover-Mechanisms can still be used
- Secure** , Can be integrated into an role-based-access-control Security-Management,
- Scalable** , Capacity and Performance can be met in the long term, there are no critical limitations

3.1. Technical Choice of migration strategy

The migration of the legacy system can be achieved on different levels of abstraction, the following list shows the alternative solutions.

- UI migration, screen scraping, does not fit to the requirements
- data migration, but all investment in semantic and process information will be lost
- functional migration, but data has also be migrated
- object migration

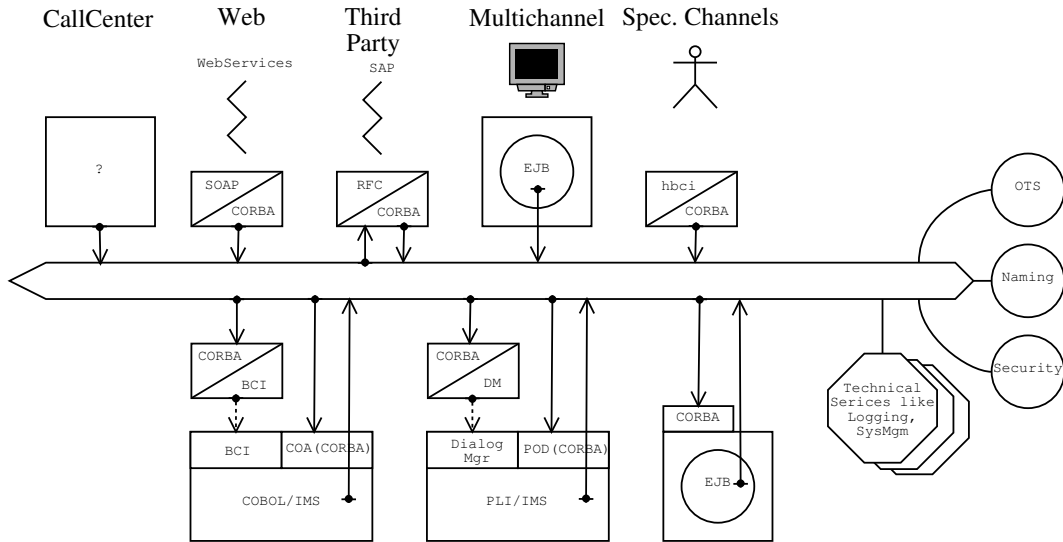


Figure 4. The GAD Service Bus Architecture

- component migration

We chose object migration as a foundation to take advantage of the benefits of object-orientation[3]. This approach can be extended to component migration once an open standard for component based development is established, which we expect with the Corba Component Model[15].

3.2. Modernize or Replace BCI

The migration alternatives were the Maintenance, Replacement or Modernization[8] of BCI. TheBCI-framework has proven itself over years as a scalable powerful runtime environment for BB3-application on the backend side. However as shown its communication model and deployment pattern does not keep pace with the requirements. The characteristics concerning the functionality of BCI can be summarized as follows:

- (-) The current framework does not fulfil all demands of flexibility, but
- (+) IMS database and BCI-protocol are a solid foundation for non-functional requirements, all banking business functions of BB3 are reachable via BCI so we do not start on a “green-meadow”
- (=) Additional functionality should extend the current system architecture in a harmonic manner, assuring backward compatibility for a smooth migration

Bisdal et al. [4] recommend a replacement strategy only for systems that are considered “undocumented, outdated, or not extensible”. Several risks come with a replacement

strategy . The replacement activities are very resource-intensive and may interfere with maintenance activities in the old system. Overall the ability to tune performance and stability have also to be trained from scratch, so the developers involved will have to achieve learning curve effects. However the robustness and functional level of the old system cannot be taken for granted for the initial stage of the new system. The modernization strategy was chosen because the existing running system can be replaced in incremental steps, the first one is described in the following paragraph.

4. The modernization strategy: Refactoring the BCI-framework

Refactoring is was by Opdyke in [13, S.71]. We extend his approach he applies to program code to the BCI-Framework. Therefore the benefits of BCI will be leveraged by extracting standard patterns and features to a dedicated software layer, which bridges the business logic from the legacy system to the “new” object-oriented world. Every client software has to use the bridge services and patterns, so re-use is enforced implicitly.

Looking on BB3 and BCI from a birds perspective, BB3 is a set

$$T = \{T_1, T_2, \dots, T_N\}$$

of IMS-Transactions.

Each Transaction T_n has a set

$$I_n = \{I_{n_1}, I_{n_2}, \dots, I_{n_M}\}$$

of valid inputmessages and a set

$$O_n = \{O_{n_1}, O_{n_2}, \dots, O_{n_K}\}$$

BCI artifact	CORBA/IDL artifact
byte tuple in data block	atomar or constructed CORBA/IDL type
data block	CORBA/IDL struct
transaction scedule	method call on an instance of a service interface

Table 1. Mapping between BCI and CORBA artifacts

of outputmessages.

Each input message contains a header, state-information and a set of data blocks. The output messages have the same structure. The header- and state blocks have a static design.

The data blocks are raw byte data. The structure and semantics are defined in the development-repository. The parts of the data block have the same order and byte-length as the description in the repository. The byte tuples map to different classical, COBOL-like “data types” like decimals, integers and strings. Some “data types” are COBOL picture clauses, e.g. date is encoded as a decimal with a fixed punctuation. Other types have a well defined domain. All data is encoded in EBCDIC.

Some approaches like the Java Connector Architecture [9] propose to use key-values lists or indexed string-arrays to access this data. As we know, that the number of “data types” is small and constant, we use a different approach. Every “data type” corresponds to a subclass of *Valuetype*. This classes implement a bidirectional mapping between a byte-tuple and a *CORBA::Any*, which typecode corresponds to an adequate atomar CORBA type. Besides these atomar types there are also subclasses representing constructed types like structures and unions. This class hierachy is the internal type-system of the CORBA-BCI-Bridge.

Based on this typesystem it is straightforward to map a BCI data block to a corresponding IDL structure or union and vise versa.

The mapping for the transaction call

$$O_{n_i} \leftarrow T_n(I_{n_j})$$

is also straight-forward. Each possible tuple (O_{n_i}, T_n, I_{n_j}) maps to a method signature on a service interface. This pattern is the most simple, RPC like communication pattern. Beside of this, there are two other fundamental communication patterns.

The first is a iterator like pattern. BCI has an option to send sequences of data. In the fist call it sends n of M tubles data. The next call gives the next j tubles. The calls can be repeated, until M is reached. This behavior is controlled by flags in the state information and can be mapped to an iterator. The call of the service gives an instance of an iterator object as the result. The iterator implements the discussed behavior.

The next fundamental pattern is the polymorph pattern. In some cases there is no one-to-one relationship between the input data block I_{n_i} and the outputblock O_{n_j} . There

is only a relation between one input block I_{n_i} and a set of possible output blocks

$$O_{n_i} = \{O_{n_{i_1}}, O_{n_{i_2}}, \dots, O_{n_{i_N}}\}.$$

For example assume the input block represents a customer key, then the output block may be a natural person or an juristical person. Or the input block is an account number and the output block is detailed account information, if there is only one account information, or a list of account information, if there is more than one account information. The list of account information is represented as with the known iterator pattern.

There are also some variations to this three fundamental pattern, like methods returning voids and methods using out parameters and in-out parameters as cookies.

Which methods together build one interface is a business design issue. The realtionship between interfaces and IMS-transactions is N to M.

4.1. Handling of Exceptions and Errors

Today BCI has a very simple concept to show the user, that something went wrong. The dialog client shows a message in the bottom row of the terminal. BCI has three different message classes: Errors, warnings and informational messages. Because all three classes use this software mechanism, the different classes are often “misused” in the legacy. Error messages are used as warnings etc. The transport mechanism for these messages is an additional data block (NR) in BCI. It is not a promising idea to use this data as the only source for an exception, because error messages may be also misused and this data block is additional, there are other results in different data blocks in the same BCI message. If the NR block would just be thrown as a CORBA exception, there is no way to get the other results to the client. This would ignore the transactional effects on business data that happened in the back-end.

Besides this there are three different ways in BCI to signal an error:

1. Setting a special “data processing not succeeded” flag in the header.

2. Using an IMS ABEND - a program termination - to show, that there is something wrong.
3. Sending an “unexpected” BCI output message.

If option one or three is used in conjunction with a NR block, the content of the message block is thrown in a standardized user exception. If option two is used or another IMS- or OTMA exception occurs, the content of this user exception is predefined.

There is also the possibility to map the occurrence of one BCI output message to a user defined exception per method. In this case the content of the datablock of this message is mapped to the content of the exception.

4.2. Implementation concepts

The CORBA-BCI-Bridge is a POA (Portable Object Adapter) [10] based server. As shown in figure 5, it has different components. This section will give a brief overview of how the different components interact in order to map a CORBA request to the appropriate BB3 transaction.

From the client-side perspective the CORBA-BCI-Bridge is a server object. This object is registered in a CosNaming server. This server object is constructed while starting the runtime. Its name, the nameserver to use etc. is part of the runtime configuration.

The server object is the primary entry point for a client to create a session. BCI is session-oriented, so we need user authentication and some information identifying the bank. In this sense the server object is the portal to a session manager. The session manager and its associated *PortableServer::ServantManager* is a threadsafe and manageable implementation of the evictor pattern [12]. The session can be deleted by an explicit logout, a timeout, cache overflow or from the systems management console.

The session object acts as a factory for service objects. If the client calls the *createService*-method, the session looks up the internal mapping dictionary to find the service. Each service is a *PortableServer::DynamicImplementation*. This means, each CORBA request to this object is routed to *invoke*-method. The implementation checks, whether the called method is registered, creates a CORBA-NameValueList and calls the method implementation to fill this list with the appropriate parameters. Then it calls the method implementation. The method implementation instantiates the needed BCI header and state blocks and creates a *TypeValue*-struct from the input parameters, using the mapping information from the internal mapping dictionary. As described before, the *TypeValue* object do the conversion from a *CORBA::Any* to the byte-tuples.

After this the implementation concatenates the BCI blocks and sends the data stream to the appropriate IMS transaction¹.

The way back is the same. The implementation parses the header and state information, the data block is assigned to the appropriate *TypeValue*-struct and this struct do the conversion from raw byte data to a struct of *CORBA::Any* using the typical features of the composit pattern [6].

The DSI invoke method takes this *CORBA::Any* and returns it to the request object. In the case of an exception the exception is returned is returned to the request object.

The other communication patterns mentioned are variations of this implementation.

4.3. Software-Development

The initial version of the CORBA-BCI-Bridge is written in C++ utilizing the open source MICO² orb. Later as thread safety became important, the “Bridge” was migrated to ORBIX from IONA. The source is based strictly on CORBA-specs, so porting to another ORB (TAO or back to a high-scalable MICO) will only imply minor recoding. The runtime configuration is stored in XML-files and is parsed during system start by the ServDicFeeder utilizing a DOM-Parser (Apache XerCes). Figure 6 shows the processes and resources of the software development workflow where the CORBA/BCI-Bridge is involved. Metadata which stems from the data dictionary is enriched with extra process data and converted to interfaces descriptions in IDL and XML. The IDL files are used on the client side to generate stub functions for the service, the XML are used to feed the Service-Dictionary of the CORBA-BCI-Bridge, which maps the incoming CORBA-calls to the appropriate IMS-Transactions. When building java applications, the stubs and the business logic are linked and distributed via jar-files. During run-time the client application never calls the BB3 directly instead it uses the CORBA-BCI-Bridge (directly or indirectly through an application server) via IIOP which then routes the request to BB3.

4.4 Programming the Client side

As with all CORBA applications the OID of the (remote) business object has to be found.

- Get NameServer standardized way is to use *CORBA::CosNaming*.

```
resolve(in Name name)
```

¹Today the BCI bridge uses the IONA IMS Adapter for this. From our point of view, it is the most efficient byte-stream adapter for IMS. But the bridge can use any other byte stream IMS adapter.

²www.mico.org

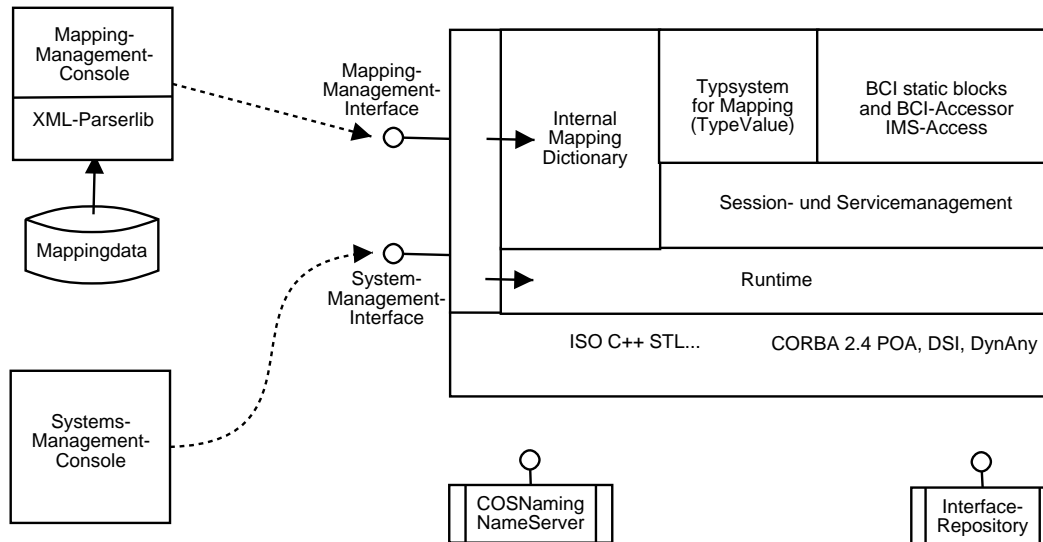


Figure 5. Implementation overview

- Get OID
- Get Object
- Narrow Object
- Get Bridge object
- Get Session object

```
// ncRef is a reference
// to a CORBA-Nameserver
NameComponent path[] =
    ncRef.to_name("BCIBridge/TheBridge");
BCIBridge.Server server =
    BCIBridge.ServerHelper.narrow(
        ncRef.resolve(path));
```

This reference is a constant value during the life cycle of the client, and is the logical connection to the remote bridge process.

4.5 Login and get a Session to establish a session with the bridge

The User creates an object of the *BCIBridge::Server::LoginInfo* and sets the member values. By calling method *Server::login()* a reference to a session object will be created for the user.

By using the session object the user can access the services which are available for his user type profile. The BCI-Bridge makes use of the role and access concept of BB3, therefore a front-office clerk will be able to access a limited number of services compared to the services that are

available to the CEO of the bank. The lifecycle of a service object is bound to the session it is called from. The call of *Session::logout()* explicitly removes the dependent service objects.

Several event driven frameworks might prevent the client keeping the reference to the remote session object, therefore the *myID* can be used to store it in a scalar variable. When the session must be used again, it can be retrieved by calling the method *Server::getSession(in SessionID id)*.

4.6 Retrieving the available Services

The Services can be retrieved by calling *Session::listServices()*. An implementation in plain java would like the following:

```
//----- set Login info -----
BCIBridge.ServerPackage.LoginInfo linfo =
    new BCIBridge.ServerPackage.LoginInfo();

linfo.institutsNr = 4985;

linfo.buchungsDatum =
    new GAD.Types.Datum();
linfo.buchungsDatum.tag = 25;
linfo.buchungsDatum.monat = 4;
linfo.buchungsDatum.jahr = 2001;

linfo.datum =
    new GAD.Types.Datum();
linfo.datum.tag = 24;
linfo.datum.monat = 4;
linfo.datum.jahr = 2001;
```

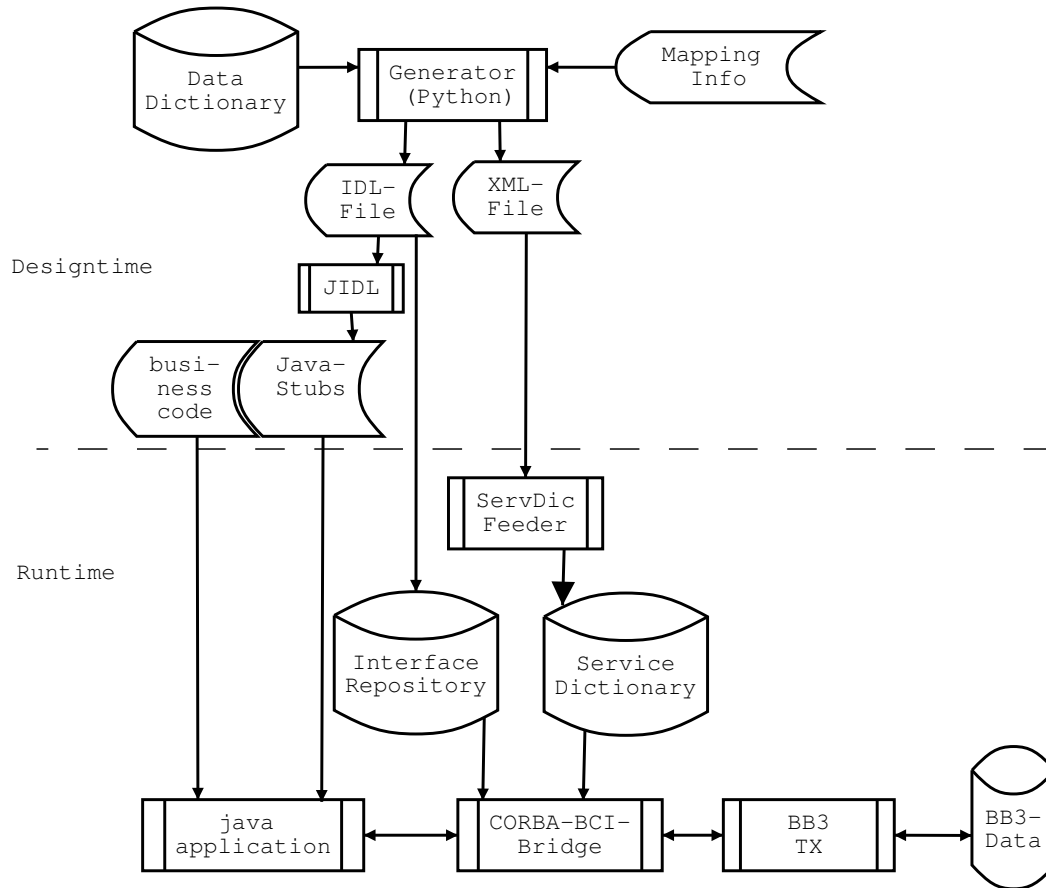


Figure 6. The Development und Runtime process

```

linfo.zeit =
    new GAD.Types.Zeit();
linfo.zeit.stunde = 10;
linfo.zeit.minute = 10;
linfo.zeit.sekunde = 10;
linfo.uidAnmeld = "YXXXXXX";
linfo.uidAutor = "ABCG4XX";

//- Login -
BCIBridge.Session session =
    server.login(linfo);

//- Print available Services -
java.lang.String[] servicelist =
    session.listServices();

for (int i=0; i < servicelist.length ; i++)
{
    System.out.println(servicelist[i]+"");
}

```

Calling the method *Session::getService(in ServiceType s)* creates a new instance of a service. A *BCIBridge::Session::ServiceNotFound-Exception* will be thrown if the service does not exist. If the service is not available to the user a *ServiceDisabled-Exception* is thrown.

The result of the method *getService* is an object which is an instance of *BB3::Service*. In Java this object reference has to be casted (with *narrow()*) to the appropriate type. Because of the harmonic way of embedding CORBA to the java language there are no further requirements than knowing the IDL, in order to call BCI-transactions from java applications.

5 Conclusions and Further Enhancements

The CORBA-BCI-Bridge is the first major step in an evolution towards an open application architecture. Based on this CORBA-compliant foundation, other functionality using open standards can be integrated. The following list depicts the possible further steps.

Enhance Parallelism by Distributed Callback Techniques

can be used to decouple back-end transaction from the front-end, blocking calls and lockups of the client can be avoided by utilizing modern client threading mechanisms and CORBA callback functions.

The role of the BCI-Bridge in next-generation-CORBA environment

In CORBA 2.x-based scenarios a server is understood as a monolithic program that operates by Request and Response, which is quite similar to the model of batch programming that was introduced with the first mainframes. With CORBA 3 there will be a new instrument in the architects toolbox, the lightweight, modular, re-usable components. By the use of these components programming can be enriched and re-use will be enforced through the assembly process utilizing existing components.

Refactoring of the backend transactions The number of business transactions that are only used by CORBA clients they can be redesigned in order to “slim” them, leaving just the core business functionality, establishing layer separation and enforce functional consolidation.

Integration in CCM-Framework The CORBA components specification is the basis for incorporating business components. In a CCM-based-scenario the legacy services established by the BCI-Bridge will show up as CCM-services.

Supporting MDA-Applications The fulfillment of business needs can be described without referring to a particular middleware technology. The choice of a middleware platform is often not based on the “best possible solution” criteria, it is often necessary, due to (cost, time, contract, skill) restrictions, to use a “best available solution”. As a long as replacement of such a suboptimal solution is anticipated this part of the system can be modernized later in harmonic way. To achieve this is the primary goal of the MDA [2], where platform independent models are used as a starting point to derive platform dependent models once the implementation middleware is chosen.

Separating Role-Based Access Control Nowadays BB3 handles authentication and functional access control in a proprietary way. When business processes are spawned across application frontiers, the correct delegation of user contexts and access rights becomes a critical requirement. Therefore a application-independent role-based access control [16] will be established and interoperate with the CORBA-BCI-Bridge, matching CORBA-principals authenticated by PKI-certificates to their BB3-access right.

References

- [1] Bergey, J.; Northrop, L.; Smith, D.: Enterprise Framework for the Disciplined Evolution of Legacy Systems
- [2] Model Driven Architecture: <http://www.omg.org/mda>, 2001.
- [3] Sneed, H. : Objectorientierte Softwaremigration, 1997.
- [4] Bisdal, Jesus; Lawless, Deirdre; Wu, Bing; Grimson, Jane; Wade, Vincent; Richardson, Ray; & O’Sullivan, D. “An Overview of Legacy Information System Migration” Proceedings of the 4th Asian-Pacific Software Engineering and International Computer Science Conference (APSEC 97, ICSC 97), 1997.
- [5] Danielczyk, P.; Pohlmann, M. ; Schönfeld, M.; Schulte, P.: CORBA/BCI-Bridge, Ein skalierbares Konzept zur serviceorientierten Wiederverwendung von Produktionssystemen in einer Multi-Channel-Architektur unter der Prämisse der Investitionssicherung (technical paper, GAD internal), 2001.
- [6] Gamma, E.; Helm, R.; Johnson, R., Vlissides, J.: Design Patterns , Elements of Reusable Object-Oriented Software, 1995.
- [7] Enterprise Application Integration with CORBA, Component and Web-Based Solutions, 2000.
- [8] Santiago Comella-Dorda, Kurt Wallnau, Robert C. Seacord, John Robert: A Survey of Legacy System Modernization Approaches , 1997
- [9] Java Connector Architecture, <http://java.sun.com/j2ee/connector/>, 2001.
- [10] The Common Object Request Broker Architecture, OMG, 2001. <http://www.corba.org>
- [11] Osterloh, M.; Frost, J.: Prozessmanagement als Kernkompetenz, 1999.
- [12] Henning, M.; Vinoski, S.: Advanced CORBA Programming with C++, 1999.
- [13] Opdyke, W.: Refactoring Object-Oriented Frameworks, 1993.

- [14] Ubilab homepage:
<http://www.ubilab.com>, 2001.
- [15] Homepage of the MicoCCM-Project
<http://www.fpx.de/MicoCCM>, 2001.
- [16] Beznosov, K.; Deng, Y.; Blakley, B.; Burt, C.,
Barkley, J.: A Resource Access Decision Service for CORBA-based Distributed Systems, 1999.