Abstract

This paper describes the online brokerage extension of GEOS (Global Entity Order System), a mainframe-based financial back-office system. The extension is implemented as a satellite system of GEOS utilising CORBA, JAVA, Servlet API and XML. It supports multiple front-ends using different user interfaces. The paper starts with a discussion of the general software requirements and gives an overview of the resulting software architecture. It then goes on to describe the architectural impact of requirements such as high performance and availability, middleware independence, non-intrusive security policies, design for testability and the reuse of the client-side code for different presentation layers like Swing, HTML or WAP.

Keywords: System Evolution, Online Brokerage, Distributed Systems, Internet, Intranet, CORBA, XML, WAP, JWAM, Servlet API, JSP, Struts Framework, Unit Testing, JUNIT, DROPTEST, Regression Testing
1. Introduction

GEOS is a mainframe-based financial software system developed by SDS (Software Daten Service) for straight-through real-time processing of securities and derivatives transactions, supporting front, middle and back office operations. GEOS consists of more than 6 million lines of C code and is accessed by a C++ front-end running on Win32 and OS/2. In 1999 it was decided to extend the functionality to facilitate online stock brokerage with multiple front-ends such as a Java Swing application, HTML browser or WAP-enabled device. Due to its size and complexity, enhancing the mainframe system beyond its original design scope is difficult and might decrease its reliability and availability. Consequently the new functionality was moved into a satellite system (SDS Internet Banking) and the mainframe system was considered as a black box. The challenge of this project was to fulfil the non-functional requirements typically associated with mission-critical financial software.

What are the non-functional requirements for such an online stock brokerage extension?

1) High Performance and Availability
The overall system must fulfil the same demands of performance and availability as the mainframe system. Therefore performance and stress testing has to be considered as high priority.

2) Middleware Independence
The online stock brokerage extension will be in use for many years. Consequently any particular middleware technology will be outdated or obsolete. Customers can have a strong opinion about which middleware to use or not to use and thus they demand a change of middleware.

3) Well-defined Mainframe Access
As the mainframe system is still under development, the access must be well-defined to minimise the impact of changes and to keep both systems maintainable.

4) Security Requirements
The original system (C++ front-end) has always been used within a trusted environment, i.e. operated by bank employees within the LAN. Opening such a system for the Internet requires strict security checks to detect intruders.

5) High Degree of Code Reuse
Supporting multiple front-end benefits from reuse of the client-side code since a lot of code is concerned with control flow and input parameter validation. Without proper reuse strategy this code has to be replicated for each front-end technology (e.g. Swing, AWT, HTML or WAP)

6) Design for Testability
The online stock brokerage extension must be designed with emphasis on testability since debugging a distributed system is time consuming. It has been noted that testing takes at least one half of the effort required to develop object-oriented, distributed systems. [Binder94a]

2. Software Architecture Overview

The bank customers and front office users can access GEOS by using multiple channels such as WAP, HTML Browser or stand-alone JAVA application. All the requests are processed by the GEOS Internet Server which uses the SDS Server Manager as a TCP/IP-LU62 Gateway. The classic C++ front-end is still used for the back office operation because it provides a much broader functionality than the other front-ends. All front-ends (C++, Swing, HTML, WAP) are using the same interfaces to access the mainframe.
The GEOS Internet Server is a CORBA-based distributed application. The Login Manager implements a Facade Pattern [GoF96] and delegates requests to the Interface Manager and Session Manager which are not used directly by any client application. The Interface Manager encapsulates the CORBA Naming Service and provides versioning of interfaces and static load balancing. The Session Manager is implemented as persistent process pair [GrayReuter93] and communicates with the customer specific Authentication Module. The Session Manager is responsible for managing all of the sessions and supports data exchange between servers on different computers. The JGEOS Server processes the client requests and is implemented by a CORBA Business Object. The security policy is enforced by a dynamically loaded Security Manager which communicates with the Session Manager.

How does the client interact with the server? The GUI framework generates requests which are delegated to a Datasource. The Datasource in turn delegates the requests to the Transport Layer, which shields the client from the underlying middleware. The requests are transmitted to the CORBA-based Business Server implemented by a CorbaBusinessObject, which maps the requests to various Data Access Objects. The Data Access Object is ultimately responsible for contacting the host-based GEOS and contains all implementation-specific information how to communicate with the mainframe. The CorbaBusinessObject and the SecurityManager are part of an application framework developed in-house. The results are propagated back through the various layers until they are displayed by the front-end.
3. Impact of the non-functional Requirements

3.1. High Performance and Availability

High performance and availability are essential for financial software systems and must be taken into account from the very beginning of the project. It starts with the application design and the definition of the IDL interfaces considering call latency, data marshalling rate, object model granularity, client-side caching, distributed garbage collection and fail-over scenarios. Depending on the ORB implementation and the environment you can expect 200 to 1,000 operation invocations/sec and a data marshalling rate between 200 Kb/sec and 800 Kb/sec [HeVi99]. It was decided to keep the object model extremely simple by implementing stateless servers and avoiding object creation on behalf of a client.

All CORBA remote method invocations are done by using the CORBA Multiproxy pattern [SGR99] which allows transparent fail-over to a replica server process. The system doesn’t contain any Single Point of Failure and all server processes apart from the CORBA Naming Service and the Session Manager are stateless. The CORBA Naming Service can be replicated and the Session Manager is implemented as persistent process pair [GrayReuter93] which guarantees high availability. Accessing the Server Manager is done by a similar TCP/IP-Multiproxy allowing to use multiple Server Managers transparently.

Load and stress testing are important for a distributed system to identify performance bottlenecks but putting a proper test environment into place requires a significant amount of planning. A bottom-up approach was used for load and stress testing starting with the Data Access Objects (DAOs), the CORBA-based application framework, the JGEOS Server and finally the Servlet Engine. This testing was done using JUNIT++ [Goe01] and Microsoft Web Application Stress Tool [Chang00]. The ability to combine load testing with simultaneous profiling of the Servlet Engine proved extremely helpful for optimising the performance. During the development we were able to increase the performance from 40 to 145 simultaneous users on our test server (PIII-800, Orion Servlet Engine, 256 Mbyte memory, JDK 1.2.2).

3.2. Middleware Independence

The clients use the Transport Layer which is determined and instantiated at run runtime to support various middleware technologies such as CORBA, SOAP or the internal SDS protocol. All of the Transport Layers have the same behaviour so that it is easy to switch between different implementations:

• The client is completely unaware of the underlying middleware. A client application could use the XML/HTTPS-based transport layer instead of CORBA to communicate with the server through a firewall.
3.3. Well-defined mainframe access

The classic C++ front-end communicates with the mainframe by using highly complex data structures (implementing a Composite Pattern [GoF95]). In the past the interfaces between C++ front-end and mainframe were considered private by the C++ programmers with the following consequences:

- The interfaces were not sufficiently documented
- The input parameter values were stored in complex data structures containing many more parameters than actually needed. One particularly complex data structure contains more than 400 parameters where actually 37 of them were necessary for this transaction.

It was decided not to modify the existing interfaces but to reverse-engineer them and to encapsulate the resulting knowledge in a Data Access Object (DAO). A Data Access Object is either responsible for performing a single transaction on the mainframe or using other DAOs to fulfill its task (Composite Pattern [GoF95]). To simplify the reverse reengineering task a SDS protocol analyser was developed which creates a DAO and a corresponding unit test by analysing the byte stream between C++ front-end and mainframe.

Technically a DOA is implemented as a Java Bean and all static information is encapsulated in the corresponding Java Bean Descriptor:

- Information how to invoke the CICS transaction or which DAOs are used internally.
- Description and versioning of the DAO
- Definition of all input parameters including name of the parameter, verbose description, data type and potential input domain restrictions
- All valid combinations of input parameters (similar to method overloading in C++)
- Data Type of the result

The usage of this descriptor class has a couple of advantages:

- It facilitates automatic input parameter validation through the application framework
- It allows building a repository where a programmer can browse through the available functionality or perform impact analysis of changing a CICS transaction or data structures.

The concept of DAOs, i.e. explicitly defining the preconditions of function call in software, proved to be very helpful during development because many coding errors are caught automatically.

3.4. Security Requirements

The architecture supports non-intrusive security policies by borrowing the idea of CORBA Interceptors [OMG99], where a security policy can be implemented by intercepting the ORB request processing at various stages. The CORBA Business Object generates the following events, which are processed by dynamically loaded Security Manager.

How can you enforce a non-intrusive security policy with such events?

- Ensure that every function of the IDL interfaces contains a session identifier
- Check if the corresponding session exists when the function is invoked
- Check if the caller is authorised to invoke this function
- Check if all relevant entities referenced in the input parameters actually belong to the caller
- Mark all relevant entities referenced in the result as belonging to the caller
Relevant entities could be globally unique identifiers (GUIDs), security account numbers, contract numbers, user id and son on. The SDS applications use a very rich type system defining more than 200 different data types. Using this algorithm, it is possible to keep track of all relevant entities of a user and detect attempts to break into the system.

### 3.5. High Degree of Code Reuse

The GUI framework is based on the JWAM [Gryczan99a-f], architecture and supports the reuse of client-side code by decoupling it from the user interface library. Using the JWAM architecture allows the development team to develop multiple front-end simultaneously (e.g. a Java Swing or a HTML).

The JWAM architecture is centred around the idea of tool layer, a presentation layer and an interaction layer. A tool implements the client-side business logic and communicate with the Datasources to access the server. A tool communicates with the presentation layer by using the interaction layer. Therefore the interaction layer insulates the tools from the actual user interface presentation.

The Java front-end uses Swing as user interface library. The HTML front-end utilises Java Server Pages (JSP), the Struts framework [Davis01] and Cascading Style Sheets (CSS). The Struts framework is part of the JAKARTA project (http://jakarta.apache.org) of the Apache Software Foundation (http://www.apache.org).

### 3.6. Design for Testability

Testing was considered as a high priority and therefore taken into account during design and implementation:

- A custom assertion package [Payne98] is used for implementing Design By Contract [Mey97]
- Usage of the Lightweight Coherence Pattern [Binder99]
- A logging package is used which allows dynamic loading of different logging facilities, e.g. use a centralised CORBA logging service without modifying the application code similar to [Modi01]
- All SDS exceptions contain the history of all previous exceptions even if this exception was thrown on a remote machine. This feature allows a developer to see all the stack traces of all involved server process spanning multiple computers.
- Unit testing and integration testing was extensively used during the project utilising JUNIT and the DROPTEST [Wild97]. JUNIT is a Java based unit test framework whereas DROPTEST is based on PERL.

The use of a unit and integration tests to detect errors in the components and their interfaces as soon as the code was compiled was a great advantage and contributed significantly to the lack of faults found in system testing where they are much more difficult to locate and remove. However, this requires that the tests are designed into the system architecture from the start which is a critical success factor for such projects. [Binder94b].

### 4. Conclusion

The SDS Internet Banking project team consists of seven software engineers who spent thirteen man years. The following tables contains basic source code metrics about the sub-projects of SDS Internet Banking:

<table>
<thead>
<tr>
<th>Subproject</th>
<th>Classes</th>
<th>Lines</th>
<th>Statements</th>
<th>%Branches</th>
<th>Calls</th>
<th>%Comments</th>
<th>Methods/Class</th>
<th>Avg. Stmt./Method</th>
<th>Biggest Method</th>
<th>Max. Depth</th>
<th>Avg. Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORE-BASE</td>
<td>123</td>
<td>16237</td>
<td>5032</td>
<td>13,00</td>
<td>2461</td>
<td>29,8</td>
<td>6,98</td>
<td>3,6</td>
<td>54</td>
<td>7</td>
<td>1,80</td>
</tr>
<tr>
<td>CORE-DOMAIN</td>
<td>511</td>
<td>54040</td>
<td>11186</td>
<td>4,60</td>
<td>3673</td>
<td>44,9</td>
<td>4,12</td>
<td>1,9</td>
<td>125</td>
<td>7</td>
<td>1,27</td>
</tr>
<tr>
<td>CORE-CORE</td>
<td>268</td>
<td>38048</td>
<td>13435</td>
<td>12,50</td>
<td>6868</td>
<td>22,6</td>
<td>7,45</td>
<td>4,0</td>
<td>104</td>
<td>8</td>
<td>1,85</td>
</tr>
</tbody>
</table>
The CORE project consists of three subprojects: CORE-BASE implements commonly used infrastructure, CORE-DOMAIN implements the 200+ data types and their formatting and CORE-CORE contains the SDS-specific infrastructure. The CORBA and JGEOS projects are split into a client and server part, whereas the JGEOS server project contains many data structure needed for communicating with the mainframe (therefore %Branches is so low).

The XML project contains all XML-related functionality, like data transformation and request parsing which is highly recursive. The SDS project contains the code needed for accessing the SDS customer management system. The UI project covers the JWAM architecture. PROXY is a SDS protocol analyzer which generates DAOs and corresponding unit tests.

Statements: in Java, computational statements are terminated with a semicolon character. Branches such as if, for and while are also counted as statements.

Branch Statements: Statements that cause a break in the sequential execution of statements are counted separately. These are the following: if, else, for, do, while, break, continue, switch, case and default. The exception block statements try, catch and finally are also counted as branch statements.

Method Call Statements: All method calls are counted, in statements as well as in logical expressions.

The main effort (nearly 60%) of the SDS Internet Banking project went into fulfilling of non-functional requirements such as

- high availability and performance
- middleware independence
- well-defined interfaces for accessing the mainframe
- stringent security requirements
- a high degree of code reuse due to the support of multiple front-ends
- design for testability.

The functional requirements are implemented by the JGEOS-CLIENT and JGEOS-SERVER package. These packages represents roughly 40% of the overall code size and implements 25 dialogs using 35 DAOs and 196 data structures.

The project is considered a success because

- the layered architecture and high reuse allows new team members to be very productive. The first step to incorporate new functionality is to generate the skeleton of a Data Access Object. The functionality of the DAO is incrementally refined and tested with JUNIT tests. On the average it takes two to five days to implement a moderately complex DAO and the corresponding unit tests. Accessing the server through a Datasource is very simple because the programmer is unaware of any underlying middleware. Creating the user interface using the JWAM architecture is a complex task but the user interface logic is implemented only once due to the code reuse. At the end of the development cycle we have multiple user interfaces and a regression test suite for the new functionality which will decrease the maintenance costs
- the focus on high availability and performance combined with early load an stress testing allowed to minimise the project risk. Many of our initial stress tests were successful in breaking the server system in the test environment.
- the non-intrusive security policy boosts the programmer’s productivity. The programmers can concentrate on the business logic while safely ignoring the security related aspects. The class implementing the security
policy is determined and loaded at runtime which allows to customise the security policy depending on the environment (e.g. Intranet versus Internet usage of the software)

- the system is largely independent from the middleware. It currently supports multiple ORB implementations and even the usage of a completely different middleware technology (e.g. MQSeries) is feasible.

All in all, it can be concluded that evolving a mission-critical mainframe system in the era of the Internet implies focusing on non-functional requirements. Taking the non-functional requirements seriously will create software with high performance, high availability and significant lower maintenance costs.
5. References


[Wild97] Fred Wild: „Design for Testability“, Dr. Dobb’s Journal 2/97