Architecting in the Face of Uncertainty: An Experience Report

Ian Gorton, Jereme Haack

Information Sciences and Engineering,
Pacific Northwest National Laboratory, Richland, WA 99352,USA
Ian.Gorton@pnl.gov

Abstract

Understanding an application's functional and non-functional requirements is normally seen as essential for developing a robust product suited to client needs. This paper describes our experiences in a project that, by necessity, commenced well before concrete client requirements could be known. After a first version of the application was successfully released, emerging requirements forced an evolution of the application architecture. The key reasons for this are explained, along with the architectural strategies and software engineering practices that were adopted. The resulting application architecture is highly flexible, modifiable and scalable, and therefore should provide a solid foundation for the duration of the application's lifetime.

1. Introduction

It is by no means uncommon for software projects to proceed without fully understanding their requirements. Often, this represents a lack of attention to the desired functionality of the system, and leads to applications being delivered that do not satisfy customer needs.

It is perhaps less common for a project to be required to deliver functionality when the requirements are not actually understood by the clients, beyond a very broad description and scope. Such a situation inevitably forces the engineering team to attempt to anticipate requirements from what little knowledge and direction is available. As software engineers rarely possess perfect foresight, this too can be a risky proposition.

It is precisely this latter situation that confronted a project team at the Pacific Northwest National Laboratory. A major, multi-year project was commenced whose requirements were only broadly understood by the project sponsors. Essentially the project, known as Glass Box (GB), had to capture detailed information on a user's workstation activities during information gathering and analysis tasks. This information had to be recorded over long time periods, and made available to a range of client research projects.

The client's research projects were all attempting to develop new approaches and technologies to support and enhance user activities in their information gathering and analysis. These approaches would subsequently be validated using the data captured by the GB application. However, at the time of the project commencement, the exact set of client research projects was unknown, as the contracts had not been awarded. This of course made detailed requirements gathering essentially impossible. It was though a necessity to commence at this stage, as the data from the GB had to be available for the client projects as soon as possible, so that their research could be grounded on the observed behavior of the users.

This paper briefly describes the initial solution developed by the GB team to provide core data capture and distribution. We then describe how the gradually emerging requirements from the client projects made it necessary to expand the solution architecture that had been adopted. The driving forces for this architecture evolution are explained, along with the design and engineering strategies adopted to create a flexible solution that should be able to endure for the lifetime of the GB project. The paper concludes with a summary of the approach taken and the major lessons learned.

2. Background – Glass Box v1.0

At the start of the GB project, the requirements were extremely loosely defined. Essentially, the first version of the software was to monitor and store potentially useful data about a user's activities on a Window's based workstation. This data would then be made available to the client projects to post-process and use as a test-bed and validation vehicle for new technologies. The users of the GB application should also have some tools to search through the data that has been collected, annotate it, and delete data that was inadvertently captured.

The development team initially began to prototype and gradually evolve a number of tools that would capture user activity, in terms of mouse and keyboard tracking, windowing activity, file access and Internet browsing. Numerous technical issues were encountered and solved. These typically were concerned with the details of the operation of the operating system, web browsers and the applications that the users worked with. In addition, the GB software was not straightforward to test, as it involved hooking in to operating system level events, which were not easy to generate in a controlled manner.

Despite these difficulties, the first version of the application was delivered on time. It was successfully deployed on the user's workstations, to a very positive

response. After the user's started working with the GB software, the collected data was distributed to the clients, who by this time had been selected and had commenced their research projects.

The deployed application architecture is depicted as a UML component diagram in Figure 1. The roles of the three components are explained below:

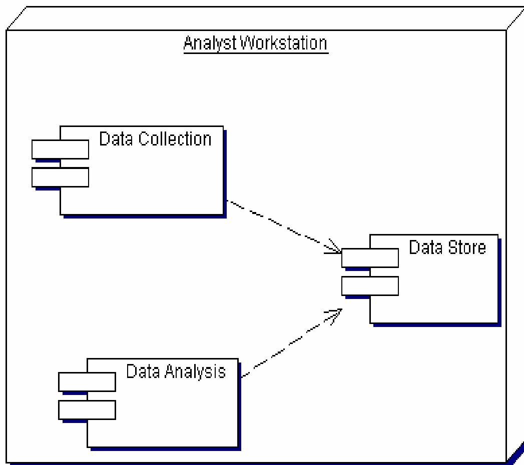

Figure 1 Logical Two-Tier GB Application Architecture

Data Collection: The collection components comprise a number of loosely coupled processes that track user activities and store them in the *Data Store*. The captured events relate to windowing system events, file saves, Internet accesses and copy/paste edits. Users can also insert data about their activities and notes about their line of thinking at any given time.

Data Store: This component comprises a commercial-off-the-shelf (COTS) relational database and a hierarchically structured file store. The relational database stores information on user activities in various tables, with timestamps added so that the order of events can be reconstructed. The database also contains meta-data on the files that the user accesses, along with pointers to actual files that are stored in the file system. The files that are captured are the Web pages and word processing documents that the user views and modifies during their work. The Web pages are especially important, as many pages have dynamic content, which makes post-analysis of the user's activities impossible, as the page content will have changed. Hence the GB software stores snapshots of the page content locally, so that the exact page content seen by the user can be replayed, even though the original URL will have changed.

Data Analysis: The user is supplied with a set of tools collectively known as the *Control Panel*. With these, they can specify when activity recording starts and stops, store free-hand notes that expand on their activities, and submit various *canned* queries to the *Data Store* to view the information that has been stored. These queries exploit a number of heuristics that have been designed to allow many of the raw events in the database to be aggregated and presented to the user at a more meaningful level (e.g. individual key stroke events are aggregated to show what a user typed in a particular application window).

When deployed, all these components resided on the user's workstation[1]. Each user consequently had their own copy of the *Data Store*. The *Data Store* was designed however to allow these individual copies to be merged in to a single master database. Scripts were written to perform nightly *Data Store* backup and periodic *Data Store* mergers. The merged *Data Store* was then distributed on a monthly basis on CD to the client research projects for local installation.

In order to facilitate access to the data, the client project teams were provided with a comprehensive set of documentation on the *Data Store*. This made it possible for them to write SQL queries to retrieve data that was relevant to their research. Several sample queries were provided in the documentation to demonstrate some 'best practice' techniques to create queries. These served their purpose, and soon after the first data was released, several client projects were working with the GB data in their research.

3. Non-functional Requirements for v2.0

Once the Glass Box v1.0 software was released, planning commenced immediately for version 2. Some specific functional requirements from client projects had emerged, and satisfying these was built in to the project plan. In addition, it had become apparent through discussions with clients that:

- More flexibility, in terms of platform and language support, and distribution was likely to be required in how the client projects accessed GB data.
- A framework would be required to allow the clients to 'plug' their research tools in to the GB environment, and obtain immediate feedback on user activities.

However, beyond these general requirements, few if any of the client projects were at a stage to be able to specify concrete needs.

[1] Although, through the use of JDBC, it is trivial to locate the database on a remote machine.

In response, the GB team held several brainstorming sessions. These attempted to anticipate the detailed implications of these broad requirements, and began the process of designing an application programming interface (API) for client projects to use in accessing GB data. The results of these sessions are summarized below:

Heterogeneous platform support: Several of the research teams were developing technologies on platforms other than MS Windows. The GB software was tightly coupled to Windows, as this was exclusively the user platform in the Glass Box. Also, the relational database used in the *Data Store* was supported only on the Windows platform. Hence, it was clear that GB v2.0 must adopt strategies to make it possible for software not executing on Windows to access GB data and plug in to the GB environment.

Instantaneous event notification: Many of research tools being developed by the clients aimed to provide timely feedback to the users on their work. A direct implication of this was that these tools needed access to the events recorded by the GB as they occurred. Hence, some mechanism was needed to distribute GB events, as they were being captured and recorded in the *Data Store*.

Remote data access: The initial GB installation and users was based on the USA east coast. The research teams were distributed across North America. Clearly then, it would be inconvenient (at the minimum) if all experiments had to be carried out by installing and testing software on the user's workstation. A requirement therefore was to make it possible for the research tools to access the GB data remotely. If this access was required instantaneously, it will obviously have higher latency than if the software was co-located with the GB. Also, the data access must safely navigate the security (firewalls, etc) provisions in place at both organizations.

Ease of data access: The GB database and related file store comprises a moderately complex software component. The relational database has 38 tables and 46 views, with some complex interrelationships. This made the SQL queries to retrieve data non-trivial to write and test. Also, as the functional requirements evolved with each release, changes to the database schema were inevitable. For these reasons, a mechanism to insulate the client's tools from database changes and to make it easier to retrieve data was deemed a requirement.

4. Architecture Design

With these non-functional requirements enumerated, the architectural design phase commenced. As a guiding principle, we were aiming to design as much flexibility as possible in to the resulting design. The intent was to create general mechanisms that could cater for more detailed behavior as the requirements for these emerged from the client projects.

The fundamental architectural change was to move to a strictly layered architecture [1]. This is depicted in Figure 2.

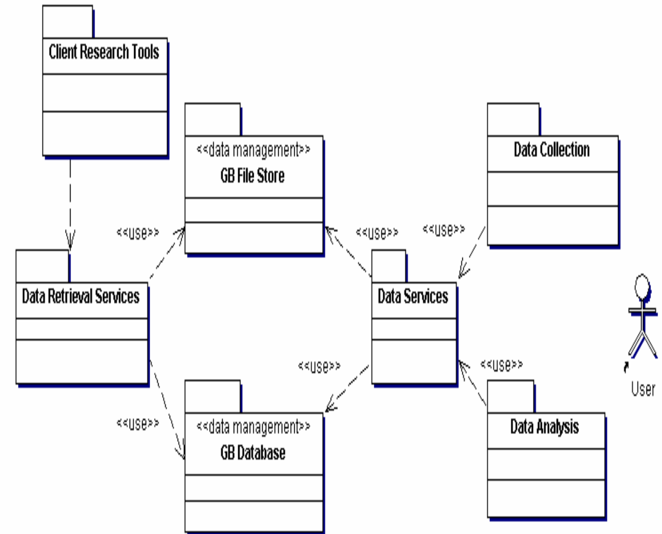

Figure 2 Layered Architecture

In the layered architecture, the *Data Store* remained unchanged, comprising the *GB File Store* and *GB Database*. However, another layer was introduced which isolated the *Data Collection* and *Data Analysis* components from the *Data Store*. The intermediate *Data Services* package offered an abstract set of services for the collection and analysis components to use. Consequently, these components required modification to replace their direct SQL and file store calls with calls to the abstract service layer. Once these changes were made, the collection and analysis components were considerably simplified.

Similarly, the *Data Retrieval Services* package offered a set of abstract services for the research tools to read from the *GB File Store* and *GB Database*. These services were designed around supporting a set of use cases that were elicited from two client projects, and by drawing on experience from other similar API designs, anticipating the way the research tools will want to navigate and retrieve information.

Briefly, the API was designed around sets of services to allow client tools to navigate GB data by particular tasks, users and sessions (activity time ranges explicitly recorded by the users of the GB software). For example, a request could require all the Internet home page URLs accessed by a specified user working on a specified task, and optionally specify which sessions were of interest. It could then iterate through the URLs in the result set,

optionally retrieving more detailed information, including the actual web page, for further analysis. Similar services were designed for all the major abstractions in the GB data, such as window and keyboard events, files accessed/modified and analyst annotations. In addition, a set of primitive services was designed, which essentially gave access to the individual abstractions by key values. Together, these services greatly simplified the code needed in the research tools, reducing complex SQL queries to single line service requests.

Introducing an intermediate service layer therefore effectively simplified the components responsible for collection and access of GB data. It also insulated them from changes in the database and file system organization, as they no longer had a direct dependency on these structures. Hence, the 'ease of data access' requirement was satisfied.

Crucially however, the service layer provided a single location that 'knew' about all the data storage requests that originated from the *Data Collection* components. This made it the ideal candidate for the dissemination of events to any research tools that wished to instantaneously receive GB events. Upon receiving a data storage request, it could check to see if any tools wish to know about the event, and if so, distribute it to them using some efficient mechanism.

Event dissemination has a number of possible solutions. The simplest to provide a mechanism for a research tool to request events of some kinds are sent to it as they occur. As the research tools will be running in a different address space, sockets or some more abstract mechanism (e.g. Java RMI) could be used as transport. This solution works well when the number of research tools is small and fixed, and the types of events requested are from a fixed set.

It seemed likely though that these conditions may not hold for the foreseeable lifetime of the GB project. We anticipated deployments where multiple client tools would be simultaneously requesting notification of multiple and varying event types. Although no single research team had requested such a requirement, it seemed sensible to incorporate a more flexible and scalable architecture than the simple alternative described above.

The solution adopted used a publish-subscribe architecture, as depicted in Figure 3. The research tools register their desire to receive event notifications by subscribing to a logical named topic supported by the *Event Notification Service* component. The *Event Notification Service* then informs the relevant services in the *Data Services* components that they should publish new events using the logical topic name. The *Event Notification Service* takes responsibility for delivering the events published on the topic to every subscriber registered. Note that no events are published when there are no subscribers. This is an optimization for publish-subscribe architectures, reducing message traffic. We call this a *publish-on-demand* design pattern.

Publish-subscribe provided a very loosely coupled architecture. Publishers have no knowledge of the number or location of subscribers. Topics can support multiple subscribers, facilitating one-to-many and many-to-many communications. Further, responsibility for connection management and message delivery is designated to the *Event Notification Service* component, simplifying the publishers considerably. This architecture therefore satisfied the 'instantaneous event notification' requirement.

With a layered, publish-subscribe architecture in place, supporting both co-located and remote research clients became relatively straightforward. The layers in the architecture could be deployed on the same machine, or distributed across multiple nodes in a multi-tiered deployment. One or more logical layers map to a single physical tier. Such a deployment required the use of suitable distributed protocols to communicate between the layers, so that they can be deployed flexibly to match the needs of various clients and deployment scenarios.

Consequently, in order to support the requirement for remote GB clients, standard Internet protocols can be used to create a multi-tier application. For example, a Web Services-based façade [2] could mirror the interfaces supported by the *Data Retrieval Services and Event Notification Service* components. Web Service requests are firewall friendly when running on HTTP. The façade localized all knowledge of connection management and remote client location, leaving its supporting components unchanged.

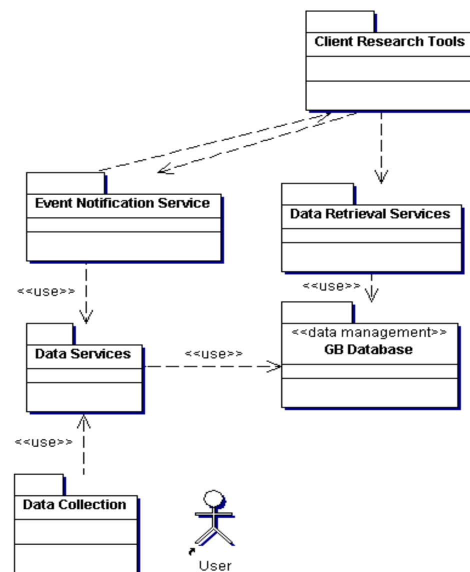

Figure 3 Event notification architecture

As a side effect, providing Web Service based interfaces to the GB also helped support heterogeneous clients. Web Services are programming language and platform neutral, with mappings supported for most popular technologies.

In addition, we chose to implement the core collection and event notification using the Java 2 Enterprise Edition (J2EE) platform. Java is platform neutral, satisfying the requirement for platform heterogeneity. In addition, J2EE has inherent support for distributed component-based systems, publish-subscribe event notification and database access [3]. Research clients may therefore choose to access the GB using a Java or Web Services application-programming interface (API). The choice will be governed by their anticipated execution location and the programming language used. This architecture is shown in Figure 4.

5. Architecture Analysis

In order to analyze the architecture, we explored a number of explicit modifiability scenarios [4]. This exercise led us to consider how the architecture and supporting technology could support anticipated changes to GB applications and deployment scenarios. The most important of these are described below:

Deploy GB software with a shared database: The database schema is designed to support multiple simultaneous users on the same database instance. As the database is accessed using JDBC, this modification reduces to a simple JDBC driver configuration change. This would facilitate a deployment with a shared database running on its own machine, with no application code changes.

Deploy GB software in a distributed fashion: Communications with J2EE components inherently uses protocols that support transparent distributed deployments. The J2EE *Data Services* components can consequently be hosted on a different machine to the GB client software simply by making changes to the J2EE directory service. No code changes are required in the GB client. Further, the distributed J2EE components can support multiple simultaneous users. This makes a full three tier [5] distributed deployment possible simply through configuration parameter changes. Such a deployment would then have the well-known scalability attributes of multi-tier applications [6]. In fact, in the current single user deployment, the three tiers are basically all just executing on the same machine, supporting a single user.

Modify GB database schema/file store: Changes to the database organization or file store structure will necessitate code changes in the *Data Services* and *Data Access Services* components. Structural changes that do not add new data attributes will be contained totally within these components. Modifications that add new data items for GB or research client use will require interface changes in *Data Services* and *Data Access Services* components. Interface versioning can be used to control how these interface changes effect client components.

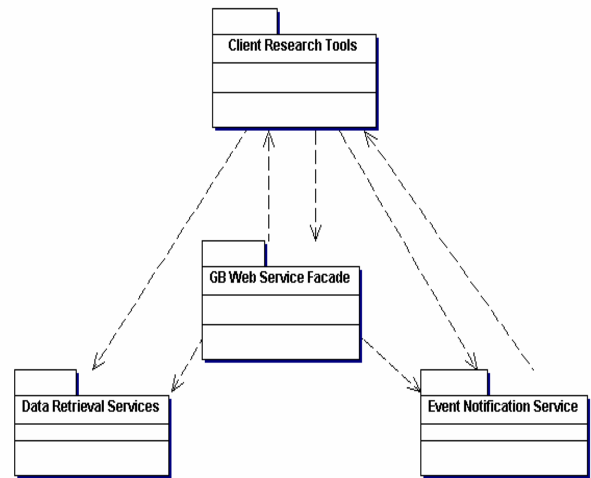

Figure 4 Supporting remote clients

Remote a co-located research tool: A co-located research tool can communicate with the GB using the native J2EE API. If this tool is then located on a remote machine/site, it will have to be modified to use the Web Services version of the GB API. Web Services development tools can automatically generate much, if not all of the required changes, making this relatively straightforward. Research clients may in fact choose to simply use the Web Services API. They may then execute remotely or be co-located with the GB software without code changes. The trade off is that the co-located version will almost certainly run more slowly using the Web Services API rather than the native J2EE API.

Change J2EE Vendor: The current version of the GB software utilized the JBoss open source application server. As J2EE defines standard interfaces and contracts with the J2EE container, changing the J2EE vendor will be transparent to the application code. However, it is likely that changes will be required to the deployment descriptors for the various J2EE components. This is because different vendors use custom deployment descriptor formats. Experience has shown the effort

required to make these format changes to be small, though still unwelcome!

This analysis demonstrated that the proposed architecture and technology realization in J2EE has the required modifiability and scalability.

6. Detailed Design

6.1 Design Requirements

With the architectural principles decided, we progressed to a more detailed design phase. The design needed to elaborate on issues such as API interaction styles, state management and persistent data access mechanisms.

In addition, we expanded on the non-functional requirements described above, to specify more detailed design needs. These were divided in to GB client programmer requirements, and the GB development team requirements.

It was decided that from the GB client programmer's perspective, the client API must be:

1. Easy and intuitive to learn, abstracting any implementation issues that were exposed in the server API (e.g. J2EE calls, state management)
2. Easy to comprehend and modify code using the API
3. Provide 'good' performance, ideally returning result sets in a small (1-5) number of seconds on a typical hardware deployment.

From the GB development team's perspective, the client and server APIs must:

1. Completely abstract the database and file store structure, insulating GB clients from the details of, and changes to, the GB *Data Store* structure
2. Support ease of modification with minimal impact on the existing GB client code
3. Support concurrent access from multiple threads or GB applications running in different processes and/or on different machines
4. Be easy to document and clearly convey usage to client API programmers
5. Provide scalable performance as client request loads increase, with minimal or ideally no changes to the server API implementation.
6. Significantly reduce or ideally remove the capability for 'badly behaved' GB clients to cause failures, consequently reducing support efforts
7. Not be unduly expensive to test

6.2 Concrete Design

Figure 5 presents a simple block diagram representing the architecture for a GB application in the initial release. GB clients are provided with a set of Java classes, the client API, which provides abstractions over GB data structures and access mechanisms.

The server API comprises a collection of stateless session Enterprise Java Beans (EJBs), hosted by a J2EE container, supporting service-based interfaces. Each bean's interface has a set of methods that facilitate access to the key GB data abstractions. These abstractions map to one or more tables and file structures in the underlying GB *Data Store*. Internally, the server API is structured as two layers. A façade layer encapsulates all J2EE-specific issues such as parameter marshalling and database connection pooling. The façade layer calls a service layer that implements the business logic of the request, with no J2EE dependencies. This layering makes it possible to, for example, relatively easily create a version of the server that does not use J2EE, relying perhaps on RMI-only, or building a library for inclusion in the client address space.

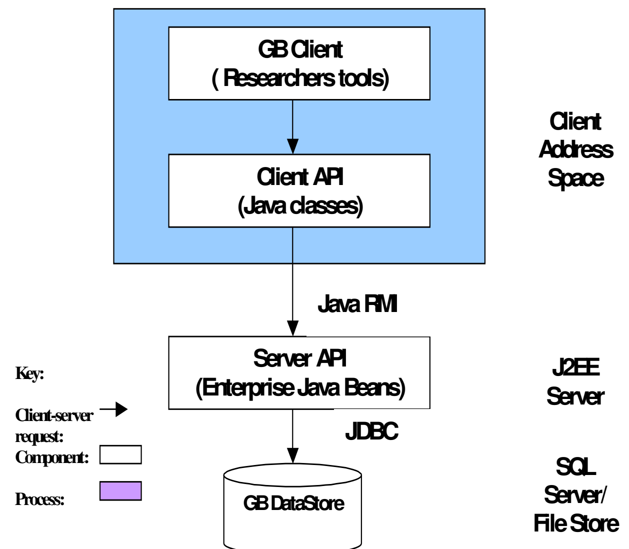

Figure 5 Detailed Architecture Block Diagram

6.3 GB Abstractions

A GB client API programmer should have no insight in to the structure of the underlying GB *Data Store*. This decouples the GB 'internals' from GB clients. To facilitate this, the API was designed around a set of key abstractions identified from the GB *Data Store*. Some examples are:

- Task
- Session
- WebEvent
- WindowEvent

- DataEvent (e.g. document edit)
- File

The attributes of these abstractions were grouped within a Java class. The mapping of the attributes to the underlying Data Store is performed within the server API layer. Hence structural changes in the persisted data do not affect GB client codes.

Within each abstraction, members were designated as mandatory and optional. Mandatory data members always have valid values when returned from a server API call. Optional members are not returned by default, but the API provides a mechanism (see below) for the API caller to designate that these member values are desired to be returned from an API call. This gives the GB client programmer the opportunity to ignore data items they do not intend to use, and optimize query performance.

6.4 Server API Design

The server API was based around a set of stateless services, forming a service-oriented architecture (SOA) [7]. SOAs are engineering best practice in terms of design for multi-tier distributed client-server applications. Modern component-based technologies inherently provide support for SOAs, making them an attractive option for the GB project.

The general form of the most prevalent server API call was as follows:

GBAbstraction[]	*returned value*
Get{GBAbstraction}By{searchcriteria}	*API name*
Search criteria values, MemberSelector, ResultSetSize	*Parameter list*

Where:
GBAbstraction is a Java class representing one of the key GB abstractions listed earlier. An API call will return a collection of objects that satisfy the query defined within the parameter list.
Search criteria values are valid key values for data items such as *Task* and *Session*, which have previously been retrieved through appropriate API calls.
MemberSelector is a class containing a set of Booleans, with one Boolean for each optional member of the *GBAbstraction* class. By default, the values are set to false.
ResultSetSize is an integer set by the caller to control the number of *GBAbstraction* objects returned from a single call.

The API name was designed to be explicit about its intent and function. The purpose of each API is conveyed clearly by the name alone. There is no need to understand the precise parameter settings and values unless the GB client programmer is specifically interested in using or understanding the API. Consequently, this made it easy for GB client programmers to learn and understand the API in existing code. Although the server API is not called directly by GB clients, the client API exposes these API names through a façade layer, and provided helper classes to make them simpler to use.

The *Search criteria values* provide the values to be used in the WHERE clause in the SQL query executed internally by the API. For example, the API *GetTasksByUser()* have a search criteria parameter requiring a valid key value identifying the user of interest.

The *MemberSelector* object provides a type-safe and extensible mechanism that allows the caller to control the member values that are returned from an API. The caller need only take action if a certain field should be included in the result set. If new members are added to the abstraction, no code changes are needed in existing clients. Member name changes or removal will be caught at compile-time, and there is no validation burden placed on the server API code.

The *ResultSetSize* provides a mechanism for the caller to control query execution time, and has a tuned, default mechanism for each GB abstraction class. Its value determines the number of results returned from each API call.

6.5 Client API

The client API abstracted the server API by hiding J2EE-specific implementation code and the issues related to retrieval and access of large result sets. To illustrate this, a simple example is shown below, with detailed parameters values omitted for brevity:

```
GBInit ( );
User a = GetUserByKey (  );
TaskIterator t = GetCurrentTasksByUser (a,
…)
while(( myTask = t.GetNext()) != NULL)

       // process task data
}
```

The call to *GBInit()* performs the necessary J2EE calls to connect the GB client to the J2EE server. Once the environment is initialized, the *GetUsertByKey()* API retrieves a specific object representing a user of the GB system. The user key is then used to retrieve a collection of task objects with *GetCurrentTasksByUser(),* which are subsequently used to initialize an iterator class. The GB client then uses the iterator's interface to process individual task objects.

The *GetCurrentTasksByUser()* API maps directly to an underlying server API call. However, in calling the

server, a *ResultSetSize* value is added to the API, and if at least sufficient numbers of results exist, a collection of objects of this size is returned. The iterator then encapsulates the behavior needed for the retrieval of results from the server API. If the GB client walks through the whole result set, and then requests another result object, the iterator automatically calls the server API again, using the last key value returned as the start of search criteria. This will return another fixed size result set, and the first element will be returned to the GB client. Hence the management of the result set sizes is transparent to the GB client code, and the implementation resembles a page fault handler in a virtual memory system.

6.6 Server API State Management

The server API was designed to hold no conversational state, and delegate all data store management to the GB database. The underlying implementation mechanism exploited common database features that only return a specified number of rows for a query. In SQL Server, the `TOP` feature is supported in its SQL dialect, and it returns the first N rows that satisfy the query. Once the specified number of rows is retrieved, the query terminates. By simply modifying the starting key value in a query (as described earlier), TOP made it possible to efficiently page through large database tables or views, returning each 'page' in constant time. This feature was carefully tested in a prototype to ensure adequate performance. The prototype clearly demonstrated the ability of SQL server to return, for example, successive 10K row result sets in constant time as a client retrieved all rows in a 700K row table.

Delegating result set state management to the database had important advantages. These are:

'Badly behaved' clients cannot be written. If the session beans that implement the API cached pages of the result set in memory, there must be a mechanism for the client to inform the server that it no longer requires further results. A badly behaved client would not do this, creating a quandary for the server bean over how long it cached the results. J2EE stateful beans can be automatically flushed after a certain period of receiving no requests, however setting this value is notoriously difficult. Too short a period causes the client to have to regularly handle failed requests and the query to be re-issued. Too long a period causes server resources to be, often wastefully, consumed. Regardless, exception handling code in the client and server must be written and tested. When the server delegates state management to the database, this problem does not exist.

Server memory management is trivial: If beans can cache data for 'long' periods, with multiple simultaneous clients, it is possible (in fact, likely) for server memory to become exhausted. Some memory management mechanism in the server could be built, but this is problematic in the J2EE component model. It would also be complex and error-prone, and in the worst case could still cause server JVM thrashing and eventual failure due to memory exhaustion. With a stateless server implementation, this issue again does not exist, as server memory is released immediately after each call completes.

Hence, delegated server state promotes simplicity and clarity of implementation, reduces test cases and downstream support costs. Simply, less can go wrong. And no performance compromises have to be made.

Maintaining no conversational state in the server also has advantages, which may accrue as the GB software becomes deployed on a larger scale. These are:

Enhanced reliability: With no conversational state, server beans can be replicated in two or more J2EE servers. If one should fail (process/machine/network), the J2EE technology automatically fails over requests to a replica bean, transparently to the client and server code. This is not possible with a stateful design.

Enhanced scalability: Replicating server beans makes it possible to distribute client requests across multiple J2EE servers. If they hold no conversational state, this is again transparent to the client and server code.

Importantly, these benefits are achieved without writing any code. The trade-off is the cost of maintaining the state in the client, and passing it to the server with each request. In the GB application, this is trivial.

7. Summary and Lesson Learned

Designing software systems when requirements are only loosely understood is a complex task. It is however a fact of life in many software engineering projects. Inevitably, new functional requirements will be continually discovered and implemented. Hence, from an architectural perspective, the key is to create a flexible infrastructure that can accommodate emerging functional, as well as non-functional requirements.

In this paper, we have described an approach to successfully designing such an architecture. The design approach and the general principles it embodies can be summarized as follows:

Define architectural requirements: Based on client discussions and team member experience and intuition, define the key attributes the architecture must have. These attributes are both application-focused, such as scalability

and resource usage, and engineering-focused, such as ease of modification, upgrade and testing.

Design the abstract architecture and implementation strategy: This defines the key abstract mechanisms that the architecture will use to satisfy its requirements. For any complex system, the abstract architecture will likely embody a number of well known architectural patterns (e.g. publish-subscribe, layered, client-server), which often need customizing to meet application needs (e.g. publish-on-demand). It is also important to simultaneously consider the implementation strategy to be used. It is again reality that implementation technologies influence architecture [8]. A quality solution must therefore meld the abstract architecture needs with the strengths of the implementation platform, and avoid its weaknesses. It is after all the implemented solution that is the final arbiter of success, not the paper-based architecture. The latter is just an, albeit important, means to an end.

'Test' the design: Define a set of architecture scenarios that can be used to see how the architecture is affected by change. Architecture scenarios relate to non-functional concerns such as modifiability, performance and reliability. By exploring how these scenarios are accommodated by the architecture and accompanying technology infrastructure, the solution can be refined and trade-offs explicitly stated and understood.

The project has also benefited greatly from using a number of software engineering practices.

Prototyping: Prototyping has been used extensively to validate design decisions. Prototypes have provided concrete evidence that a design is suitable. It also helps to give confidence to all team members that the approaches and technologies being proposed will work. Architecture affects everyone, and it is important to allay team member concerns and demonstrate that the changes will not cause the team as a whole to fail.

Exploit COTS components: Component-based technologies are feature-rich, reliable and high performance. Exploiting these makes the design, implementation and testing of an application easier. COTS technologies such as J2EE facilitate design and code reuse, and provide tested, reliable infrastructures. Simply, they reduce project effort and create higher quality and more adaptable applications.

Exploit the database features: Modern commercial databases are incredibly powerful systems. They have been developed, enhanced and tuned since the early 1980's, and are excellent persistent state managers. Learning and exploiting their capabilities means the project has less custom code to develop, test and maintain. This is nearly always a good thing.

References

1. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, Documenting Software Architectures: Views and Beyond, *Addison-Wesley, 2003*
2. F. Curbera, R. Khalaf, N. Mukhi, S. Tai, S. Weerawarana. *Web Services, The Next Step: Robust Service Composition. Communications of the ACM, 46 (10), October 2003*
3. I.Gorton, A.Liu, Software Component Quality Assessment in Practice: Successes and Practical Impediments, *in Proceedings 24th International Conference on Software Engineering, ICSE 2002, Orlando, USA May 2002, pages 555-559, ACM Press*
4. P. Clements, R. Kazman and M. Klein, Evaluating Software Architectures: Methods and Case Studies, *Addison-Wesley 2001*
5. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture - A System of Patterns. *Wiley and Sons, 1996*.
6. I.Gorton, A Liu, Performance Evaluation of Alternative Component Architectures for Enterprise JavaBean Applications, *in IEEE Internet Computing, vol.7, no. 3, pages 18-23, 2003*
7. Ali Arsanjani: Enterprise Components and Services. *CACM 45(10): 30-34 (2002)*
8. I.Gorton, A.Liu, P. Tran, The Devil is in the Detail, A Comparison of CORBA Object Transaction Services, *in 6th International Conference on Object-Oriented Information Systems, pages 211-221, 18-20 December 2000, London*