Proceedings of

"The Three-Tier Architecture Pattern Language Design Fest"

Introduction

Three-tier applications have gained increasing acceptance and popularity in the software industry. Three-tier applications usually consist of a thin client providing presentation logic, a middle-tier containing the business logic, and a back-end database. Three-tier applications provide several benefits over traditional client-server applications including:

- *Scalability*: A three-tier architecture allows distribution of application components across multiple servers thus making the system much more scalable.
- *Reliability*: A three-tier architecture makes it easier to increase reliability of a system by implementing multiple levels of redundancy.
- *Flexibility*: By separating the business logic of an application from its presentation logic, a three-tier architecture makes the application much more flexible to changes.
- *Reusability*: Separating the application into multiple layers makes it easier to implement re-usable components.

As a result of these benefits, three-tier architectures have been used in many large-scale distributed systems and enterprise applications including a large number of e-commerce solutions. Component technologies such as Enterprise Java Beans (EJB) and CORBA Component Model (CCM) support the middle-tier of three-tier architectures. They provide frameworks for component development and deployment. Similarly, many web services based on HTTP and XML make use of three-tier architectures.

While no two three-tier systems may be alike, they share similar requirements and consequently similar system designs. It was the goal of the workshop to capture these similarities in the form of design patterns and to further integrate them into a pattern language. The net outcome of the workshop was the beginning of a pattern language that describes the fundamental architecture of a three-tier system.

Discovering Patterns

Three-tier systems share many similar requirements. A typical three-tier system includes the following requirements:

- *User interface*: A web-based interface is commonly provided to access the rest of the system.
- *Persistent Storage*: An important requirement is to be able to persistently store the data, for example, in a database.
- *Adaptive Business Logic*: Systems need to be adaptive to changing business logic. They should provide hooks in the framework to add new features or plug in new services easily.
- *Data Consistency*: The state of the data in a system needs to be consistent across the lifetime of the system. This usually refines into some form of transaction and concurrency control mechanisms.

- *Fault Tolerance*: Systems need to be fault tolerant as well as highly available.
- *Lifecycle and Resource Management*: Since systems must typically be highly scalable, special care needs to be taken regarding the lifecycle of the components and general resource management.
- *Security* : Each layer of a three-tier system needs to support some form of security.

Design patterns can serve as a valuable tool to describe the system architecture that meets these requirements. Design patterns abstract away from any specific language, platform, or domain. The goal of the workshop was to discover the patterns common among three-tier architectures and interweave them into a pattern language. Several patterns and pattern languages have already been documented that address the requirements of three-tier systems. Here is a small subset of these patterns along with the requirements that they address.

Requirement	Pattern References
User Interfaces	Interaction Patterns in User Interfaces [WeTr00], Pattern Language for User Interface Design [CoLe96]
Persistent Storage	Mapping Objects to Tables: A Pattern Language [Kell97], Relational Database Access Layer [KeCo97]
Adaptive Business Logic	Component Pattern Language [Voel01], Component Configurator [POSA]
Data Consistency	Monitor Object [POSA2] , De-Centralized Locking [Schu01]
Fault Tolerance	Reliable Hybrid [Dani97] , Master-Slave [POSA], Object Group [Maff96]
Lifecycle and Resource Management	Leasing [JaKi00], Evictor [Jain01], Lazy Acquisition [Kirch01] , Eager Acquisition [KiJa01]
Security	Authenticator [BrFe99] , Architectural Patterns for Enabling Application Security [YoBa97] , Pattern Language for Cryptographic Software [BRD98]

The goal of the workshop was to expand on this list of patterns as well as integrate them into a pattern language that effectively describes the architecture of a three-tier system.

Results of the Design Fest

The Design Fest began with a discussion of layers. We first asked the question why we need layers. Here are some of the forces that drive the need for a layered architecture:

- To manage complexity
- Separate concerns
- Scalability issues, such as load balancing
- Ability to exchange user interfaces
- Support for multiple communication channel

We then asked the question how many layers are needed in a system. The answer for this question was simple: “It depends.” It depends on the domain problem one is trying to solve.

We agreed that when we usually talk about three-tier architectures, we refer to three layers – presentation layer, business layer and data layer. However, to separate concerns even more, it is more appropriate to group responsibilities into additional logical layers. We then spent a

considerable amount of time identifying a list of layers that are common among most *n*-tiered architectures. Along with each layer, we identified the responsibilities of the layer, the forces that come into play and finally any existing patterns that address the forces.

The following list shows the layers we agreed on, which are most common:

Layer	**Responsibility**	**Forces**	**Patterns**
Presentation	• Presentation of the user interface	• Avoid (hard) state • No business Logic	• MVC • User Interface patterns
View/Dialog	• Manage the display state • Support for multiple user interfaces (presentation layers) and associated communication channels • Keeping track of the conversational state		
Process	• Contains business rules • Lifecycle of business use cases • Is stateful, keeps application/session state • Connects and coordinates services		
Service/ Business Process	• Stateless • Atomic actions/services, also allows composite services • No user interaction • Covered by one transaction		• Service Abstraction Layer • Server-side Component Pattern Language • Business Components
Business Entity	• Represents persistent data in the application • Hides implementation details of the data layer • Might provide optimization of data access	• Fast access, relative to data layer	• Dynamic Object Model • Business Components
Data	• Represents persistent data	• Slow access	• Object-to-relational mapping

In addition, we identified some general patterns that applied to a majority of the layers.

- Resource Management Pattern Language
- Command [GHJV]
- Distribution and Communication patterns
- Security patterns

Discussion on Legacy Integration

We then had a discussion on integration with legacy systems. Legacy Systems include Enterprise-Information-Systems, Back-Ends, and Mainframes (with overlap of course). They can provide pure services, pure data access, or any combination of them. We agreed that depending on the major focus of the legacy system, it is best to integrate them into the service layer or the data layer.

Pattern Language Discussion

We concluded the Design Fest with a discussion on what kinds of patterns are important to put into the pattern language. We agreed upon:

- Patterns to separate the layers
- Patterns to find the right amount of layers
- Documentation for newbies
  - start with simple patterns
  - continue with more complex patterns

We then asked what kind of pattern language we could document.

- Finding the suitable architecture, finding forces to shape the system architecture
- Catalog existing patterns and pattern languages in the context of 3-tier architectures
- How should the business process be structured according to the domain problems?

List of Participants

Prashant Jain, Siemens AG, Germany
Michael Kircher, Siemens AG, India
Nicolai Josuttis, Solutions in Time, Germany
Oliver Vogel, Systor AG, Switzerland
Markus Voelter, Mathema AG, Germany
Thomas Neumann, Systor AG, Germany

References

[BRD98] A. Braga, C. Rubira, and R. Dahab, A Pattern Language for Cryptographic Software, Pattern Language of Programs Proceedings, 1998

[BrFe99] F.L. Brown and E.B. Fernandez, The Authenticator pattern, Pattern Language of Programs Proceedings, 1999

[CoLe96] T. Coram and J. Lee, A Pattern Language For User Interface Design, Pattern Language of Programs Proceedings, 1996

[Dani97] F. Daniels, The Reliable Hybrid Pattern: A Generalized Software Fault Tolerant Design Pattern, Pattern Language of Programs Proceedings, 1997

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

[JaKi00] P. Jain and M. Kircher, Leasing Pattern, Pattern Language of Programs conference, Allerton Park, Illinois, USA, August 2000

[Jain01] P. Jain, Evictor Pattern, To be submitted to Pattern Language of Programs conference, Allerton Park, Illinois, USA, September 2001

[KeCo97] W. Keller and J. Coldewey, Relational Database Access Layer, Pattern Language of Programs conference, Allerton Park, Illinois, USA, September 1997

[Kell97] W. Keller, Mapping Objects to Tables: A Pattern Language, Proceedings of the 1997 European Pattern Languages of Programming Conference, Irsee, Germany, 1997

[KiJa00] M. Kircher and P. Jain, Lookup Pattern, European Pattern Language of Programs conference, Kloster Irsee, Germany, July 2000

[KiJa01] M. Kircher and P. Jain, Eager Acquisition, To be submitted to Pattern Language of Programs conference, Allerton Park, Illinois, USA, September 2001

[KiJa01] M. Kircher and P. Jain, Ad-hoc Networking Pattern Language, Submitted to European Pattern Language of Programs conference, Kloster, Irsee, Germany, July 2001

[Kirch01] M. Kircher, Lazy Acquisition Pattern, Submitted to European Pattern Language of Programs conference, Kloster Irsee, Germany, July 2001

[Maff96] S. Maffeis, The Object Group Design Pattern, Proceedings of the Second Conference on Object-Oriented Technologies and Systems, Toronto, Canada, June 1996

[OOPSLA00] M. Kircher, P. Jain, Kirthika Parameswaran, The Jini Pattern Language , OOPSLA 2000, Minneapolis, October 2000, http://www.cs.wustl.edu/~mk1/AdHocNetworking

[POSA] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland and M. Stal, Pattern-Oriented Software Architecture--A System of Patterns, John Wiley and Sons, 1996

[POSA2] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, Pattern-Oriented Software Architecture--Patterns for Concurrent and Distributed Objects, John Wiley and Sons, 2000

[Schu01] D. Schuetz, De-Centralized Locking Pattern, Submitted to European Pattern Language of Programs conference, Kloster Irsee, Germany, July 2001

[Voel01] M. Voelter, The Component Pattern Language, Submitted to European Pattern Language of Programs conference, Kloster Irsee, Germany, July 2001

[WeTr00] M. van Welie, H. Trætteberg, Interaction Patterns in User interfaces, Pattern Language of Programs conference, August 2000, Allerton Park, Illinois

[YoBa97] J. Yoder and J. Barcalow, Architectural Patterns for Enabling Application Security, Pattern Language of Programs conference, 1997, Allerton Park, Illinois