Using a Blackboard Architecture in a Web Application

Christiane Metzner, Leonardo Cortez, and Doritza Chacín
Universidad Central de Venezuela, Caracas Venezuela

cmetzner@isys.ciens.ucv.ve, corvle@yahoo.com, doritzac@cantv.net

Abstract

In this work we discuss the development of a web application in the domain of movie chains using a Blackboard architecture, which is a well-established style for solving the problem of control, communication and collaboration in a system; it has traditionally been accepted as adequate for heuristic problem solving though not generally used for web applications. We present and discuss how the Blackboard architecture is used as the style for a graduate course project in Software Engineering. Issues about the implementation of the architecture are described and an assessment using some software metrics is presented.

Keywords: Software architecture, Blackboard architecture, web development, software metrics.

Introduction

Software architecture encompasses the set of significant decisions about the organization of a software system: selection of the structural elements and interfaces by which a system is composed, behavior as specified in collaborations among those elements, composition of structural and behavioral elements into larger subsystems, all comprise the architectural style that guides an organization (Booch, Rumbaugh & Jacobson, 1999). Therefore choosing the architecture should be a deliberate decision and not determined by the evolution of a software or chosen based on traditional ways of building software.

Frequently used and reused architectural solutions are defined as architectural patterns. Hohmann suggests that "By focusing on a specific class of problems, an architectural pattern helps you decide if that kind or style of architecture is right for your system." (Hohmann, 2003) Later, he establishes that the "pragmatic approach when creating a software architecture is to explore the various architectural patterns that have been documented and choose one that is reasonably thought to address your particular situation. From there, you must tailor the architecture to meet your needs, ultimately realizing this architecture in a working system."

The architecture generally used for web applications separates the parts of an application into so-called "tiers", or "layers" and then names the tiers, which is a straightforward process – these structures are called presentation, business and data tiers. The idea is to give each tier a main responsibility, and put the operations that work with each responsibility into their respective tier. A class should not contain code with responsibilities from more than one tier. The standard main responsibilities for defining a 3-tier architecture are:

- Presentational functionality: Presenting and collecting user data.
- Business functionality: Validating user input data and performing business processes.

- Data functionality: Storing and retrieving persistent application data.

A well-known example of a 3-tier architecture is J2EE (Sun Microsystems, 2004), which provides a unified platform for developing distributed, server-centric applications. A question that we asked ourselves was if this is the only architecture that should be considered in practice for development of web applications and why. The Blackboard architecture is a well established style for solving the problem of control, communication and collaboration in a system, and has traditionally been suggested as adequate for heuristic problem solving (Shaw & Garlan, 1996) but is not generally used in web applications. We present a solution using this architecture for a commercial web application developed as a course project in a graduate course in Software Engineering and some first results. Currently we are refactoring the solution in order to obtain more conclusive results about the viability of this architecture in web applications.

Background

The main goal of the course project was the development of the first release of a web component for an integrated movie administration system: ticket selling, standard accounting, and management processes in movie chains. In this context, a movie chain owns movie centers; every movie center defines the movie guide on a weekly basis and has one or more showrooms that present movies according to a schedule. The web component is aimed at web customers for an improved service offering show information, and on-line reservation and purchase of tickets. Scheduled shows can be looked up in a billboard page, and comprehensive information about the movies (including multimedia content) is available through a detailed view. Tickets can be purchased against a member account, which is set up and paid for when registering, though credit or debit cards are not emphasized as the system is targeted primarily at young people who generally do not own either. The web component comprises two areas:

- Public: shows the daily and weekly program. Anonymous customers can make reservations for a particular presentation. The administrative movie center staff reviews the reservations and notifies the result via email or sms.
- Registered members: can make reservations and buy tickets charging to their account. They can also view their data and statistics of their reservations as well as make some updates. The administrative movie center staff reviews the reservations and sales notifying the result via mail or sms.

On-line reservations are accepted until a fixed time before a show starts; this is a strategic decision that can be defined when configuring the application. When that time is reached, pending reservations are canceled and no new reservations are accepted. Policies related to reservation expiration time, percentage of available tickets for on-line reservations and negotiations with movie distributors are handled by the chain. Ticket pricing and account management are controlled by movie centers.

A logging subsystem has to record relevant events in the web application, including execution of automatic processes, cancellation of pending reservations, problems with access of registered members and failures in general.

The development process used was eXtreme Programming (Beck, 2000) and for clarity reasons, Figure 1 shows a use case diagram for the web component, though it was not an artifact generated during development (Chacín et al., 2004).

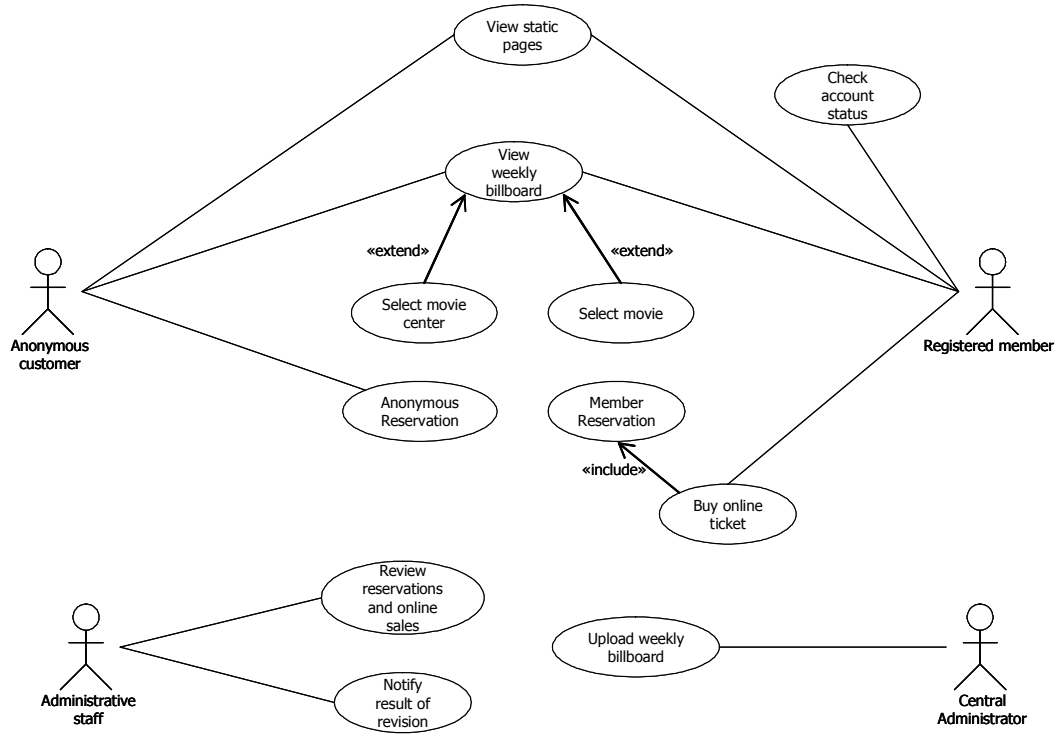

Figure 1. Use Cases for the web component.

Blackboard Architectures

The Blackboard, as an architectural pattern, has been traditionally used in the development of systems with artificial intelligence techniques. Blackboard architectures are particularly suited to solve nondeterministic problems, such as decision support, signal processing and speech recognition (Sadeh et al., 1998). The name of the pattern was chosen because it is reminiscent of the situation in which human experts sit in front of a real Blackboard and work together to solve a problem (Buschmann et al., 1996).

In Blackboard architectures, as shown in the class diagram in Figure 2, a central Blackboard (BB) data structure holds the entire state of a solution. A BB is often hierarchically structured and contains non-homogeneous content. Domain and world knowledge are represented in separate independent Knowledge Sources (KS), which hold computations that respond to changes in BB, and with direct access to it; KS interact through the

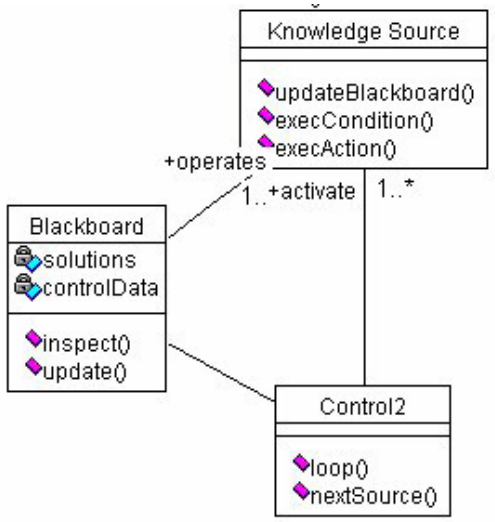

Figure 2. Structure of a blackboard architecture.

BB to yield solutions. A Control (C) monitors changes in BB and determines the next action to be executed, plans evaluations and activations according to a strategy that decides which KS is the one that will next change the BB.

Implementation

The project was developed for a J2EE platform using eXtreme programming (Beck, 2000). Issues about the development can be found in (Chacín et al. 2004) as part of a report on our academic experience in a Software Engineering graduate course. Two different groups of students had to develop the project, one group applied RUP (using a 3-tier architecture) and a second group applied eXtreme Programming (using the Blackboard architecture). Some process qualities were assessed and compared but measurement of the architectural and code properties was not considered at that time.

In this paper, we are focused instead on analyzing and presenting the solution with the Blackboard architecture, which was the architectural requirement of the second group of students. As in eXtreme programming the definition of the architecture is not explicitly considered and the students had no previous experience with this architectonic style; a spike solution was used for exploration. In general, and due to the nature of the development process, the architecture was fleshed-out iteratively and incrementally.

The initial approach used a mega class supporting queries or computations on the system's data. The business logic was embedded in jsp pages and mixed with presentation logic. This mega class was reduced to a minimum of services supporting database operations and the business logic was factored out to new classes, which later became KSs. The control C acted as the logical tier between presentation and business logic, and was responsible for managing the KS and linking jsp and business objects. In this way the Blackboard architecture also has layered KS as shown in Figure 3. The computations with direct access and which will respond to changes in the BB are defined around the business processes that are relevant to a customer.

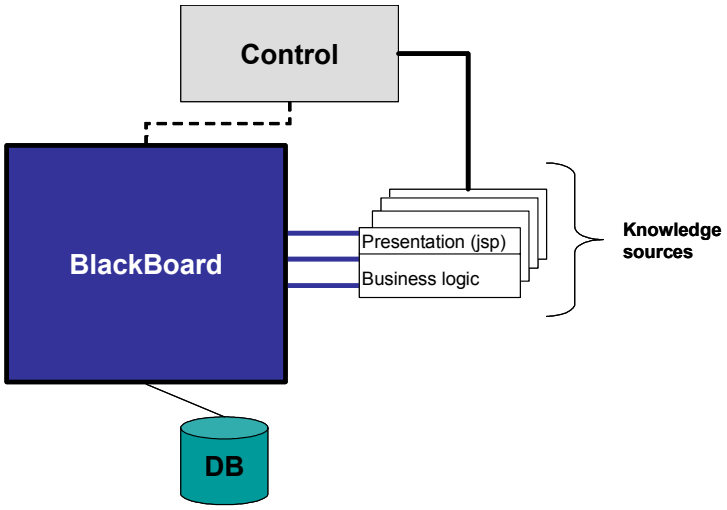

Figure 3. Components of the Blackboard architecture

The following tasks were identified as the most significant ones in the development of the first release, and they can be traced to the implementation steps suggested in (Buschmann et al. 1996):

1. **Determine user interface requirements.** This was quite straightforward, as the client (course advisor) was very specific about the interface.
2. **Define persistent data requirements.** User stories written by the client led to an initial data structure for the web application. Later on, the database was refined to support the functional and nonfunctional requirements.
3. **Design the Blackboard as a centralized supporting class with generic functionality.** As mentioned before, initially the Blackboard was a wrapper around the database. However, as

the architecture evolved, its responsibilities were modified and it became a component offering information sharing and fundamental support to the rest of the application.

4. **Translate recurrent and mechanical tasks into automated knowledge sources.** Usually, these tasks are executed when certain events are triggered or a special condition is met in the Blackboard. Actually, this is the behavior expected for control-activated knowledge sources. Thus, the described translation seemed to be the next logical step.
5. **Translate user requests handling into manually activated knowledge sources.** The business logic for dealing with a specific request is encapsulated in a unique knowledge source.
6. **Create a control class for orchestrating the whole web application.** Responsibilities of this class include the logical connection between the presentation and the business logic layers in the manually activated knowledge sources. The control class must also activate the automated knowledge sources if their condition is positively evaluated.
7. **Develop the required mechanisms for starting and stopping the web application.** A context listener class was created for the interaction with the web server.

Description of Components

The components of the web application that correspond to the architecture shown in Figure 3 are:

Knowledge sources

- *Iknowledgesource* defines an interface for all the KS. Classes implementing this interface have to register with the Control if they will be used by system processes.
- Movie Information (*InformacionpeliculasKS*): allows dynamic pages accessing movie information by generating and returning movie objects.
- Anonymous reservations (*ReservacionAnonimaKS*): provides ticket reservation services to anonymous customers.
- Member reservations (*ReservacionMiembrosKS*): provides movie ticket reservation and buying services to registered customers. Only registered members can buy tickets in advance of the shows.
- Show Movie Guide (*PresentarFuncionesKS*): offers information about movie centers, films and show times.
- Close reservations (*CierreReservacionesKS*): is a control-activated KS; closes pending reservations for on-going or past shows.
- Close functions (*CierreFuncionesKS*): is a control-activated KS; closes reservations for shows that are about to start.

Blackboard

The Blackboard holds all the data concerning the weekly and daily movie guide; it also provides special methods for supporting the updating process carried out by knowledge sources. It is a Singleton, therefore only one instance of the movie guide exists.

Control

The control extends the *Thread* java class and runs a separate thread monitoring constantly the Blackboard's state. It contains a list of registered knowledge sources for their automatic activation (if that is the case). In addition, it allows the communication among the jsp pages and the manual knowledge sources.

Additional components

- *ContextListener* implements the S*ervletContextListener* java interface and listens for events from the web application servlet. A *ContextListener* gets the reference of the BB creating and starting the Control thread; it also frees resources when the application context is destroyed.
- *LoggerSingleton* implements the Singleton pattern and is responsible for message logging.

Figure 4 shows the dependency graph between these main components generated using a freely available automated tool (Aqris Software, 2004a); the results clearly show the strong dependency of classes with the class *Blackboard* that may have a negative impact on the application's complexity. The behavior of BB may be degraded due to multiple simultaneous requests from clients; in other words, it might become a bottleneck.

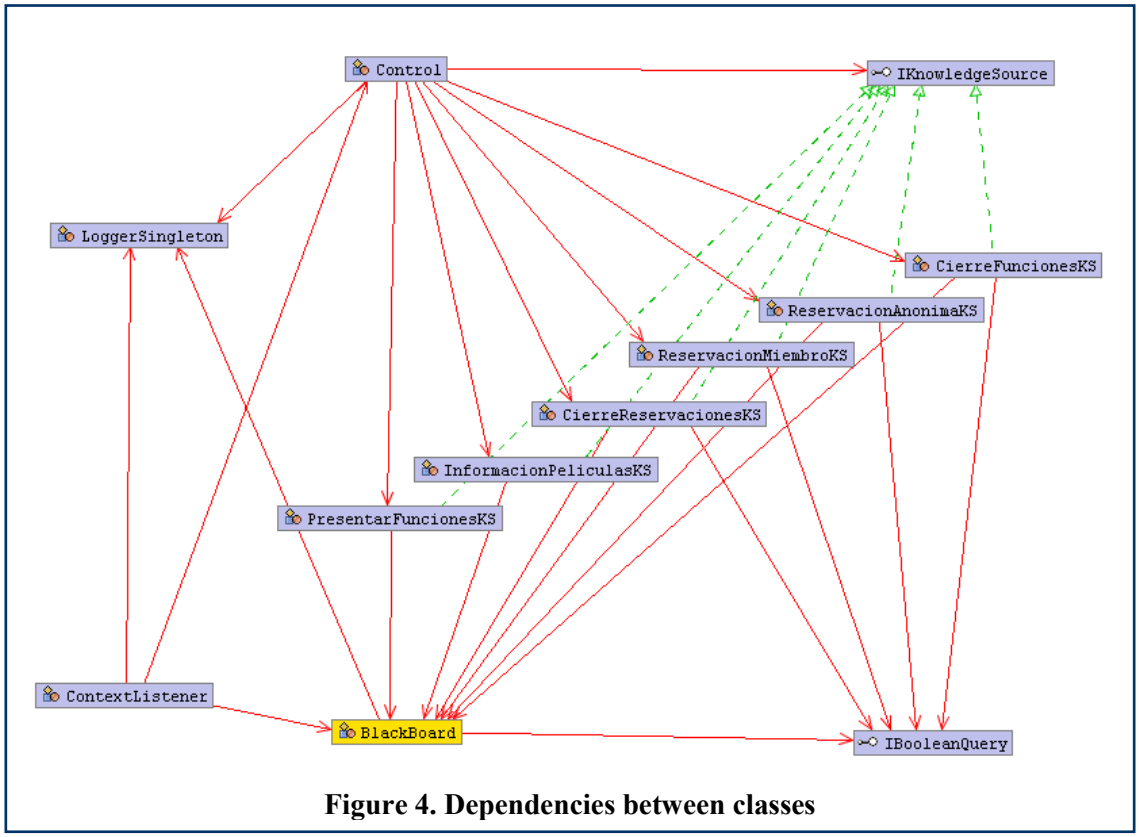

Figure 4. Dependencies between classes

Results & Evaluation

The eXtreme programming process generated a total number of 57 user stories; 19 user stories were implemented in five iterations of the first release of the project, each one taking between 2 to 3 weeks. The total effort was 1458 hours/person during 12 weeks and the implementation consisted of 34 java classes, 14 java server pages and a MySQL database with 14 tables.

Software testing was done during the whole development process, as suggested by eXtreme programming. At the end of the project, an acceptance test was made by the course advisor, with positive results. However, behavior of the web component was analyzed in a controlled testing environment, with only a few numbers of simultaneous connections. In the next release currently under development by a different group of students, concurrency issues will be exhaustively studied.

The Metrics

Measuring has been the basis for science and engineering since the Middle Ages. Galileo wrote over 500 years ago “Measure what is measurable and render measurable what is not yet so” (Kline, 1959). However, software development remains, for the most part, an unmeasured industry. Fundamental O-O metrics such as number of classes or methods per class are still not well understood by many developers and procurement people. Several authors (Bansiya, 1999; Fenton & Pfleeger, 1997; Henderson-Sellers, 1996) define O-O metrics to assess design artifact properties that contribute to the development of high quality products. However, "Quality” has different perspectives. For business, which usually does not evaluate design or code, quality means features at the boundary of a system. For Customer Service, quality means attributes that are appreciated by end users, such as usability or response time. For Finance, quality means costs, and for process-orientated business people it means "well-documented" or traceable. For developers, a high-quality design entails features that are harder to measure and that can be detected during maintenance, such as extensibility. However, from any viewpoint, quality must be understood and should be subject to manipulation and whoever the audience is, the questions and decisions that will be made based on the selected metrics have to be known. Quantification of the quality of a design and/or code is essential for developers and the information provided by product metrics can be used to reduce unnecessary complexity and irregularities and improve quality during the development process. There is as yet no single measure that describes quality, but there are several properties by which various features can be evaluated. Counting object-oriented properties is a way of assessing quality of designs and code; there are a good number of such properties at about the same level of granularity and the average is a good estimate of the mean. According to (Bansiya 1999) properties can be considered at system or at class level; at the system level, interactions between classes, inheritance hierarchies, size and the number of high-level abstractions impact the quality of a product; at the class level, structure, functionality and relations with other classes are considered. Data declaration and the number and parameter types will also influence the evolution of a product (Cattafi & Metzner, 2003).

The RefactorIT tool analyzes code and provides a number of industry standard object-oriented and static metrics used to measure code. The type of a metric discriminates between simple metrics (S) representing fundamental measures of a code such as physical lines of code, executable statements, functional points, blank lines, and the number of comment lines. Object-oriented metrics (OO) dealing with OO structure quality and providing information about the OO properties of the source code. Quality metrics (Q) concern the structure and connectivity of objects (Aqris Software, 2004b). The metrics are summarized below in Tables 1a, 1b and 1c; they are applied at the project, package, and method level. An in-depth discussion on software metrics can be found in Bansiya (1999), Henderson-Sellers (1996) and Fenton and Pfleeger (1997).

Table 1a: The Simple Metrics

Notation	Description	Preferred values	Object
CLOC	Comment Lines of Code	(*) [5,1000]	C
V (G)	McCabe's Cyclomatic Complexity	[1,10]	M
DC	Density of Comments	[0.2,0.4]	C
EXEC	Executable Statements	[0,20]	C
NCLOC	Non-Comment Lines of Code	(*) {60,80}%	C
NP	Number of Parameters	[0,4]	M
LOC	Total Lines of Code	[5,1000]	C
P = Package; C = Class; M = Method			
* Values apply for a class, method, field, package, and file			

Table 1b: The Object Oriented Metrics

Notation	Description	Preferred values	Object
A	Abstractness = *NOTa* / *NOTc*	[0.0,0.5]	P
Ca	Afferent Coupling	[0,500]	P
Ce	Efferent Coupling	[0,20]	P
DIT	Depth of Inheritance Tree	[0,5]	C
I	Instability	[0.0,0.3] or [0.7,1.0]	P
NOTa	Number of Abstract Types	[0,20]	P
NOC	Number of Children in Tree	[0,10]	C
NOTc	Number of Concrete Types	[0,80]	P
NOTe	Number of Exported Types	[3,50]	P
NOT	Number of Types	[0,80]	P
RFC	Response for Class	[0,50]	C
WMC	Weighted Methods per Class [V(G)]	[1,50]	C
P = Package; C = Class; M = Method			

Table 1c: The Quality metrics

Notation	Description	Preferred values	Object
CYC	Cyclic Dependencies	(**)	P
DIP	Dependency Inversion Principle	[0.3,1.0]	C
DCYC	Direct Cyclic Dependencies	(**)	P
D	Distance from the Main Sequence	[0.0,0.1], ideal value *D* = 0	P
EP	Encapsulation Principle	[0.0,0.6]	P
LSP	Limited Size Principle	(**) max. 10	P
P = Package; C = Class; M = Method			
** Values apply for a package			

The RefactorIT Java Refactoring Tool (Aqris Software, 2004a) calculated these metrics for our code and the results are summarized in Table 2 and those whose description is non self explanatory are discussed next.

DC provides the ratio of comment lines to all lines and is used as a quality indicator for how much of the code is commented. RefactorIT recommends a commenting of at least 20% of the code and a maximum of 40%.

EXEC counts the number of executable statements. A large EXEC implies that a code is harder to maintain and comprehend. It is recommended an EXEC range between [0,20].

The RFC is the "Number of Distinct Methods and Constructors invoked by a Class." This is the set of all methods and constructors that can be invoked as a result of a message sent to an object of the class; i.e. methods in the class, inheritance hierarchy, and methods located in other objects. RefactorIT recommends a default threshold of [0,50] for a class, but it is acceptable to have a RFC up to 100. If the RFC for a class is large, it can be difficult to test the behavior of the class

and debug problems since comprehending class behavior requires a deep understanding of the package.

WMC measures usability and reusability via cyclomatic complexities by summing V(G) of the declared methods and constructors of class. The lower limit for WMC in RefactorIT is a default 1 and the upper default limit is 50. A class with a low WMC usually points to greater polymorphism, while a class with a high WMC indicates that the class is complex and therefore harder to reuse/maintain. The larger the number of methods in a class, the greater the potential impact on children since children will inherit all the methods defined in a class, thus, limiting the possibility of reuse.

NOTe reports the number of classes and interfaces that are exported outside of a package. The default limits for NOTe in RefactorIT range between [3,50] and a large value of NOTe indicates extensive coupling. These values apply for a package.

DIP can be summarized as "High-level modules should not depend upon low-level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions." (Aqris Software, 2004a) Thus, DIP is an indication of maintenance quality: code can be easily modified since details are hidden with abstraction. Classes with low values for DIP have a high relation between abstraction and implementation making them harder to reuse. Values apply for a class where a DIP between [0.3,1.0] is preferred.

Ce (also known as Outgoing Dependencies or the Number of Types outside a Package that Types of the Package Depend on) indicates the number of classes and interfaces in other packages that classes and interfaces in the analyzed package depend upon. In short, this measure includes all the types referred to anywhere within the source of the measured package. Values apply for a package. Preferred values for Ce range between [0,20].

Instability (I) is a ratio statistic that measures the stability of a package's design. Stability is measured by calculating the effort to change a package without affecting other packages within the application. Preferred values for I range between [0.0, 0.3] or [0.7, 1.0]. 0.0 indicates a maximally stable package and 1.0 indicates a maximally unstable package. Designs of packages should intentionally be made as stable [0.0,0.3] or unstable [0.7,1.0] as possible.

D is often referred to as "The Perpendicular Distance of a Package from the Idealized Line A + I = 1," (Aqris Software, 2004a) this metric measures the square of the distance of the package from the main sequence A + I = 1, where the ideal value is D = 0. The primary rationale is that abstractness and stability of packages are closely connected. RefactorIT recommends a maximum distance of 0.1 from the main sequence before re-examination should occur.

The Encapsulation Principle can be paraphrased by "A substantial part of a package should not be used outside of the package." The EP metric is the ratio of objects that are used outside of the package, thus a low value indicates "good" encapsulation of data / algorithms within a package. Preferred values for EP range between [0.0,0.6].

Measurement and Results

In this section, we analyze those metrics shown in Table 2 that are out of range of the preferred values. They highlight properties of the design and code that should be closely examined and eventually refactored.

Table 2: Measurements for the web application

	Package	Class			Interface		Class						
Target / **Metric**	com.is.cinesweb	BlackBoard	Control	ContextListener	IBooleanQuery	IKnowledgeSource	CierreFuncionesKS	CierreReservacionesKS	InformacionPeliculasKS	PresentarFuncionesKS	ReservacionAnonimaKS	ReservacionMiembroKS	LoggerSingleton
LOC	3162	463	100	47	9	29	74	66	50	197	162	190	73
NCLOC	1602	307	55	32	1	3	46	34	25	94	106	126	41
CLOC	1311	137	39	14	8	24	24	28	20	87	46	52	25
DC	0.415	0.296	0.390	0.298	0.889	0.828	0.324	0.424	0.400	0.442	0.284	0.274	0.342
EXEC	242	83	19	15			2	2	2	5	4	4	12
WMC		39	11	2			10	6	7	30	27	31	9
RFC		38	26	16			13	11	3	11	22	27	13
DIT		1	2	1			1	1	1	1	1	1	1
NOC		0	0	0			0	0	0	0	0	0	0
Ca	11												
Ce	34												
I	0.756												
A	0.111												
D	0.133												
NOT	34	1	1	1	1	1	3	3	1	1	2	3	1
NOTa	3	0	0	0	1	1	0	0	0	0	0	0	0
NOTc	31	1	1	1	0	0	3	3	1	1	2	3	1
NOTe	27	1	1	1	1	1	1	1	1	1	1	1	1
DIP		0.125	0.125	0.0			0.4	0.4	0.333	0.111	0.3	0.25	
EP	0.704												

The density of comments (DC) in our code, which lies above the preferred values and is a consequence of documenting method's signatures in the interfaces, can be easily improved to meet those values.

The numbers of executable statements (EXEC), weighted methods per class (WMC) and responses for class (RFC) are high for the Blackboard class but WMC and RFC are in the range of the preferred values. The EXEC metric for the Blackboard class shows that it is a class hard to maintain and comprehend; its size can be reduced by eliminating some of its responsibilities.

Ce is above the preferred values indicating dependencies on external classes and interfaces. However, in our opinion, it is also an indication of reuse if the classes and interfaces depended on are stable enough.

Relationship between instability (I) and abstraction (A) suggests that the package tends to be unstable, but at the same time it has a high number of concrete classes. Therefore it is less sensitive to changes and D, the distance from the main sequence, is close to optimal. We consider that carrying out changes enhancing these metrics is not our first priority.

EP indicates that more than 70% of classes are used outside the main package, which can be interpreted as implementation details being revealed. The main reason for this is the relationship among the knowledge sources and their presentation layers (jsp pages). They should be decoupled in the next release, applying for example the *Composite View* pattern (Alur, Crupi & Malks, 2003).

The results for DIP show that classes *ReservacionMiembroKS, Control, ContextListener* and *BlackBoard* have values below the preferred range and should be refactored, decoupling their abstraction from the implementation.

Not expressed in these metrics is the high dependency between most classes and the *Blackboard*, which reduces the flexibility of the overall design and can be appreciated in Figure 4. Changes in the Blackboard class might affect the rest of the application. However, as the RefactorIT tool evaluates the required metrics (Ca, Ce) at package level, measurement of this dependency requires the Blackboard related classes to be relocated in a different package. Usually, the three types of components in the Blackboard architecture (C, BB, and KS) are implemented as classes. Therefore, there is an inherent dependency on this small number of classes, which is a consequence of the architecture's definition; there is not much choice regarding the elements that can be used in each one of the components. That was also our trail of thoughts motivating the definition of layers in each KS.

Conclusion and Future Work

Based on the assessment of the Blackboard architecture we can conclude that, although it was defined for solving heuristic and artificial intelligence related problems, its flexibility allows adaptation to web applications. It is an architecture that allows the progressive definition and implementation of the different KSs, which can also be reused. Modifying and/or adding new KS is straightforward. As a drawback, the centralization of the data in the Blackboard can negatively affect the performance and maintainability is definitely an issue. In order to reduce structural and dynamic complexity and risk elements, the Blackboard class has to be refactored reducing or even eliminating the outlined negative aspects. Finally, from a teaching point of view usage of the Blackboard architectural pattern for a web application was a success: creating an operational product using an architectural pattern in a semester period and motivating the development of a new release by a different group of students.

The currently on-going work focuses on:

1. Definition of a hierarchically structured Blackboard, one Blackboard per movie center. The general Blackboard for a movie chain is built when needed.
2. A Component Configurator (CC) has been introduced to dynamically select the components (Schmidt et al., 2000), one CC for the BB and one CC for the KSs. As a matter of fact, the CC is also a KS.
3. Composite View is used to decouple presentation code from business logic in the KSs.
4. The data and data access was removed from the BB's responsibilities and defined in a separate layer decoupling the way that data is stored in memory from the persistence logic. This is also how J2EE manages persistence.

Future enhancements of the web application also include incorporation of software agents offering recommendations to registered members through heuristic analysis of user profiles and movie popularity. In this context, Blackboard architecture is an adequate choice, since the software

agents can be easily included as new knowledge sources. In a 3-tier architecture making this kind of extension is not straightforward.

Acknowledgement

The authors wish to acknowledge Eric P. Elshtain for his invaluable and thorough editing and proof reading of this paper. We also thank the anonymous reviewers for their numerous helpful comments.

This work is supported by project No. PG-03134241-2003, Consejo de Desarrollo Científico y Humanístico, Universidad Central de Venezuela.

References

Alur, D., Crupi, J., & Malks, D. (2003). *Core J2EE patterns, Best practices and design strategies*. Upper Saddle River, NJ: Prentice Hall.

Aqris Software AS. (2004a). RefactorIT(tm) Version 2.0.7. http://www.refactorit.com/

Aqris Software AS. (2004b). Metrics: Summary table and references. Retrieved June 15, 2004 from http://www.refactorit.com/

Bansiya, J. (1999). Evaluating structural and functional stability. In M. E. Fayad. D. C. Schmidt, & R. E. Johnson (Eds.), *Building Application Frameworks.* Wiley Computer Publishing.

Beck, K. (2000). *Extreme programming explained – Embrace change*. Reading, MA: Addison Wesley Longman.

Booch G., Rumbaugh, J., & Jacobson, I. (1999). *UML user guide*. Addison-Wesley.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern – oriented software architecture: A system of patterns*. John Wiley & Sons.

Cattafi, C., & Metzner, C. (2004). Comparing designs for a chat system. *Proceedings of The 2003 International Business Information Management Conference*. Cairo, Egypt.

Chacín, D., Cortez, L., Metzner, C., Rivas, S., Romero, A., Esteves, Y., et al. (2004). Explorando extremos en un contexto académico: RUP y programación extrema. *Proceedings of 3rd International Business Information Management Conference*. Cozumel, Mexico.

Fenton, N., & Pfleeger, S. (1997). *Software metrics* (2nd ed.). Boston, MA: PWS Publishing Company.

Henderson-Sellers, B. (1996). *Object-oriented metrics: Measures of complexity*. Upper Saddle River, NJ: Prentice – Hall, Simon & Schuster.

Hohmann, L. (2003). *Beyond software architecture*. Addison-Wesley.

Kline, M. (1959) *Mathematics and the physical world.* New York: Thomas Y. Crowell Company.

Sadeh, N., Hildum, D., Laliberty, T., McA'Nulty, J., Kjenstad, D., & Tseng, A. (1998). A Blackboard architecture for integrating process planning and production scheduling. *Concurrent Engineering: Research and Applications, 6* (2).

Schmidt, D. Stal, M., Rohnert, H., & Buschmann, F. (2000). *Pattern-oriented software architecture 2 – patterns for concurrent and networked objects*. Wiley Press.

Shaw, M., & Garlan, D. (1996). *Software architecture perspective on an emerging discipline*. NJ: Prentice Hall.

Sun Microsystems. (2004). Java 2 enterprise edition (J2EE) homepage. Retrieved June 08, 2004 from http://java.sun.com/j2ee

Biographies

Christiane Metzner is a full-time professor at the Computer Science School, Universidad Central de Venezuela. Her professional experience spans over 25 years, with the last several years focused on software engineering, particularly design patterns, metrics and technologies for web development.

Leonardo Cortez earned his major in Systems Engineering in 2002, and is a graduate student in Computer Science at the Universidad Central de Venezuela. His interests include design patterns, emerging programming paradigms, simulation and numerical methods.

Doritza Chacín earned her Systems Engineering degree at Universidad Metropolitana, and holds a specialization in development of e-business applications. Her professional experience is focused on Java technologies and integration platforms. Currently, she is a graduate student in Computer Science at the Universidad Central de Venezuela.