A Comparative Study of Maintainability of Web Applications on J2EE, .NET and Ruby on Rails

Lok Fang Fang Stella, Stan Jarzabek and Bimlesh Wadhwa*
National University of Singapore
stellalok@gmail.com, stan@comp.nus.edu.sg, bimlesh@comp.nus.edu.sg

Abstract

With Web Services predicted to become distributed computing architecture in near future, maintainability of the web applications (WAs) will rank high on selection criteria while choosing a platform for development of a WA. The goal of this paper is to evaluate maintainability of small-scale WAs built on J2EE, .NET and Ruby On Rails (RoR). The maintainability criteria considered comprised of modifiability, testability, understandability and portability. We found that the RoR implementation fared better on modifiability, testability, and understandability, while J2EE implementation was the most portable. The results led us to comment on the maintainability of small WAs with respect to underlying architecture and development environments the three platforms provide. We believe that results are expected to vary for medium and large-size WAs. The work included here is part of an effort to build a decision framework for platform selection for WAs.

Keywords: *software maintainability, evolution, component platforms*

1. Introduction

Web Applications (WAs) are gaining unprecedented popularity due to their integration in business models. Leading platforms for WA developments are J2EE, and .NET. Recently, Ruby On Rails (RoR) has also received attention of industry. These platforms provide support for variety of development aspects such as managing presentation and persistence, thus allowing developers to concentrate on business logic of their applications.

The reported work aims at a decision framework to help software developers in platform evaluation and selection for WA development. Selection of a platform for WA projects is often driven by technical considerations based on ease of software design and high productivity requirements. For WAs with long shelf-life, maintainability can be one of the major concerns. Here, we consider maintainability as the extent to which it facilitates updating to satisfy new requirements or to correct deficiencies [1]. WAs' requirements change often, and the development is usually iterative, with each iteration also involving many changes. Therefore, maintainability of WAs is a crucial factor to consider.

While comparison of platforms for WA development can be found on the Web [2][3], we have not come across a study that compares development platforms from the point of view of maintainability. This paper attempts to fill that gap. We believe that work reported here will help system architects make more informed decisions regarding platform selection when maintainability is crucial.

We conducted a comparative study by conceptualizing and implementing a small-scale WA, called *Blogissimo*, on the three platforms J2EE, .NET and RoR. We examined general WA architecture requirements induced by platforms, and their impact on the architecture of *Blogissimo*. We also evaluated and compared the three implementations with respect to the four criteria of maintainability - modifiability, testability, understandability and portability [4]. Our evaluation reveals that the J2EE fares better in terms of portability, while RoR fares better in terms of modifiability, understandability and testability. We discuss the comparison of the three implementations for platform selection in WA development. The evaluation results lead us to comment on the impact of design patterns enforced by platform on the maintainability of WAs.

The paper is organized as follows: Section 2 discusses the maintenance characteristics used for our work. Section 3 introduces the *Blogissimo,* the WA that forms the basis of comparative evaluation. Section 4 provides a summary of architectural patterns and practices of each of the platforms chosen for implementing *Blogissimo.* Section 5 provides a comparative evaluation of the platforms based on criteria listed in Section 2. Section 6 discusses the results and conclusions.

2. Maintenance characteristics for WAs

Boehm et al's [1] software quality model classifies modifiability, testability and understandability as factors of maintainability. Frappier, Matwin and Mili [4] added portability as a factor of maintainability. Genero et al [5] have explored structural complexity and size metrics as predictors of maintainability. In our study, we use four maintainability criteria as described below.

Modifiability is the extent to which a software is able to incorporate changes [1]. In order to compare the modifiability of a WA on the three platforms, we conducted a change propagation analysis on each implementation for the same enhancement of requirements. The modifiability metrics - number of modified files, number of modified Lines of Code (LOC), and development time (man-hours) to incorporate the change – are used to compare the extent to which each platform facilitates modifiability.

Testability is the extent to which a software allows the establishment and evaluation of its acceptance criteria. We perform a qualitative evaluation by examining the platform features that help in the testing of the respective implementation.

Understandability is the extent to which the software is comprehensible to the maintainer. We compare it for the three implementations by comparing the code size (LOC), and examining the platform features that offer to make the WA modular.

Portability is the extent to which a software can be easily and effectively operated in a variety of computing environments [1]. Migrating software to a new platform is viewed as a type of adaptive maintenance [4]. We provide a qualitative comparison by examining the platform features that help in porting the respective implementation to a different database, persistence mechanisms, and operating system.

3. Blogissimo: A Blogging WA

We studied a number of public domain blogging applications to formulate requirements for a blogging WA, *Blogissimo*. It can be categorized as heavy on user-interface requirements as opposed to business logic or data requirements. *Blogissimo*, is implemented on each of the J2EE, .NET and RoR platforms. Each implementation of *Blogissimo* adopted the best architectural patterns and practices of the respective platform.

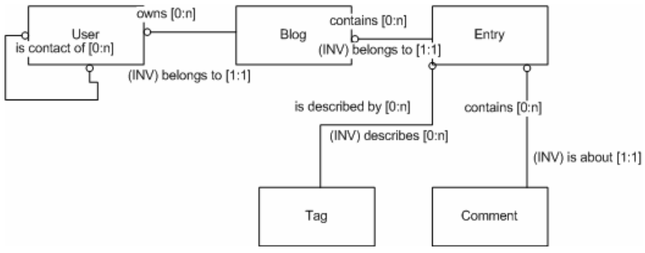


Figure 1. Conceptual classes in Blogissimo

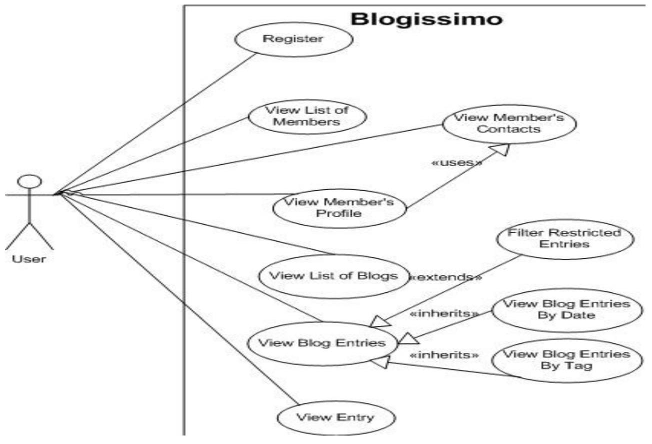


Figure 2. Use cases for User actor

Figure 1 shows the five conceptual entities in Blogissimo, and their interrelationships. Use cases shown in Figure 2 highlight the basic functionalities of the Blogissimo application.

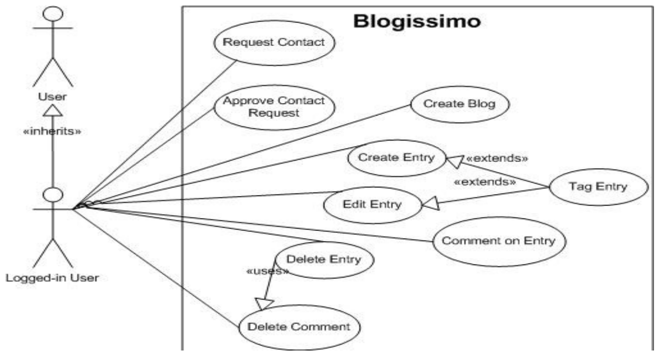


Figure 3. Use cases for Logged-in User actor

Figure 3 illustrates the additional functionalities for logged-in members. Logged-in members can request to add other members as contacts, approve contact requests, create blogs, create, edit and delete and tag entries, create comments, and delete comments.

4. Comparison of Blogissimo Architectures

In this section, we first summarize basic architectural assumptions and mechanisms of the three platforms. Next, we discuss how these affected *Blogissimo* architecture.

J2EE includes Web Services standards such as JAX-RPC ands JAXR. .NET offers language independence, through a common API, and a limited platform independence. Both J2EE and .NET are supported by IDEs. Relatively newer, RoR claims to provide ten times more productivity than J2EE [6]. Ruby is an interpreted, OO language, while Rails is an application development framework [7]. .NET WAs may be written in a mix of .NET languages supported by the Microsoft CLR. In contrast, JVM supports only Java, and RoR supports only Ruby. Ruby is more flexible and concise than Java and C#. For instance, Ruby allows the modification of class definitions at runtime. Furtheremore, J2EE and RoR WAs are portable across multiple platforms including Windows, MacOS and Linux. In contrast, .NET WAs are supported mainly on Windows. RoR's overall philosophy follows 'Don't repeat yourself' (DRY) and and 'Convention over configuration' principles. The former means that any piece of information is located in a single, unambiguous place in the code while the latter means that most of conventional elements are pre-configured by the framework.

We structured *Blogissimo* into web-UI, business and database layers. On J2EE platform, we chose Hibernate framework, Spring framework, and Spring MVC for the data access, business layer, and web layer repectively. On .NET platform, we used the ASP.NET, .NET Framework classes, and ADO.NET for the web layer, business layer, and data access layer respectively. RoR provides an integrated ActionPack-ActiveRecord mechanism for implementing all three layers. Action Pack contains ActionView and ActionController modules within it. It handles web layer, receives a request and returns view to the client, and also establishes connection to the Active Record which in turn establishes the connection between data objects and the database following the Object Relational Mapping framework. Table 1 below summarizes the design patterns used in *Blogissimo* architecture on different platforms.

Table 1. Design patterns used in *Blogissimo*

Application Layer	**Embedded Design Patterns**		
	J2EE	*.NET*	*RoR*
Web	Model-2, Front Controller	MVP, Page Controller	Model-2, Front Controller
Business	Inversion of Control (IoC)		Active Record
Data Access	Data Mapper	Data Access Layer	

4.1. Web Layer Comparison

The Web layer was split into presentation and processing logic layers in each of the implementations.

The web layer of *Blogissimo* J2EE was implemented with Spring MVC and JSPs. The Spring IoC container manages the controllers' references to business layer objects via dependency injection. For each use case, a controller returned at least one possible JSP-rendered view. The controller also updated and made the model accessible to the JSP. The JSP accessed the model via JSTL (JSP Standard Tag Library) and had minimal processing logic.

In *Blogissimo*.NET, the separation of presentation and processing logic layers was not as clean as in the J2EE and RoR implementations. For each use case, a Web Form was coupled with a code-behind that contained the code to handle events due to user interactions with the Web Form.

In the web layer of *Blogissimo* RoR processing logic was encapsulated in the ActionController subclasses. Each entity's controller contained a method for a use case that involved the entity. Presentation logic resided in .rhtml pages which accessed the model objects that were made accessible to the view by the respective controllers.

Both Spring in J2EE and Action Pack in RoR follow the MVC design pattern. The Front Controller pattern receives HTTP requests, and dispatches them to their respective handlers. Handlers process the request by making calls to the business layer, and selecting views for rendering. In Spring MVC, views are typically JSPs, while in RoR, Views are Embedded Ruby (.rhtml) pages.

ASP.NET adopts the Model-View-Presenter (MVP) pattern. Each dynamic web page consists of a visual component as well a logic component, located in separate files. The visual component - a "Web Form", which comprises static HTML and/or ASP.NET server controls, acts as a View. The logic component acts as a combination of the Model and Controller. It contains event handlers, invoked by user interactions with the Web Form, which interact with the business layer to update the model and select the next view. In contrast to MVC of J2EE and RoR, where Controller and View are cleanly separated, every dynamic web page in ASP.NET acts as a combination of a View, Model and Controller.

The three platforms are based on different programming models: Spring MVC and RoR's Action Pack are action-based web frameworks, that provide a thin layer of abstraction from the HTTP protocol. A developer or maintainer therefore needs to know the HTTP request/response protocol. In contrast, ASP.NET is a component-based web framework, that abstracts the HTTP request/response paradigm into page lifecycle events. Web pages, viewed as assemblies of components, respond to events triggered by the framework. Maintainers thus need to have understanding of code for event handlers, which is more intuitive , and should aid in overall understanding, than that of Spring MVC and RoR.

4.2. Business Layer Comparison

Business layer has been separated into domain and service sub-layers to keep use-case specific logic separated from reusable domain logic.

In *Blogissimo* J2EE, the domain sub-layer consisted of domain classes encapsulating entity behavior, while the service sub-layer consisted of the EntryService, BlogService and UserService classes. Every service sub-layer object method encapsulated a use case and delimited the scope of a transaction. The Spring framework was configured to manage transactions for service layer methods. *Blogissimo* .NET was structurally similar to *Blogissimo* J2EE. However, it required programmatic transaction coding. In *Blogissimo* RoR, the business layer consisted only of the domain model objects. ActionControllers in the web layer were responsible for delimiting transaction scopes, programmatically.

In a typical J2EE and .NET WA, the business and data access layers are distinctly separate. In J2EE, for example, domain model objects contain only behavioral logic and are not responsible for persistence. The interdependencies between application objects, and links between different layers are externalized and managed by Spring's IoC container. The Spring also provides an Aspect Oriented Programming (AOP) framework that allows crosscutting concerns to be modularized and provided as services to the objects managed by the IoC container [8].

In RoR, on the other hand, the business and data access layers overlap due to RoR's ActiveRecord mechanism. ActiveRecord provides object/relational mapping (ORM) based on the Active Record pattern [9], which describes model objects that encapsulate both a database row and domain logic for the data. Domain model classes extend the ActiveRecord::Base class, and inherit the capability to persist, find and delete themselves. Developers are insulated from the details of database operations as changes made to model classes are automatically propagated to the database. Thus RoR's domain model classes serve towards both, business and data access layers.

4.3. Data Access Layer Comparison

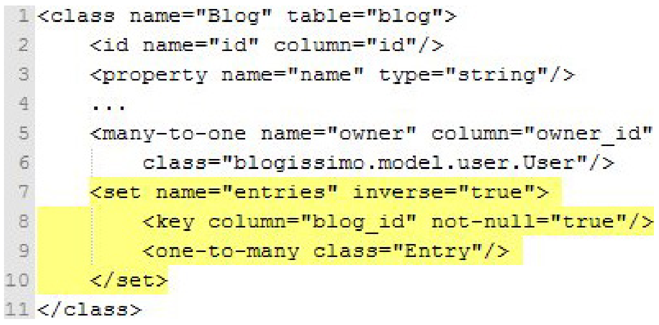
In *Blogissimo* J2EE, Hibernate provided a database-agnostic persistence mechanism for entities. XML mapping files were used to configure Hibernate to map each domain model entity to its corresponding database table. The mapping files also described entity relationships that exist in the relational model, in order for Hibernate to recreate in the domain model. In *Blogissimo* .NET, ADO.NET was used as a persistence mechanism as it provides data structures for working with database data. However, ADO.NET does not provide ORM. Thus, code must be written to map the object model to the database model. In *Blogissimo* RoR , the ActiveRecord ORM framework was used in the data access layer to build domain model objects that could persist themselves.

Unlike J2EE and .NET, RoR does not require writing attribute accessor methods because ActiveRecord dynamically generates them at runtime by examining the Blog database table definition [10]. ActiveRecord also recreates the relationships, between entities in the database model, in the object model.

In J2EE applications, Hibernate ORM framework helps developers work with entities as objects instead of database rows. The mapping of the object model to the database model is declared via configuration files. Figure 4 illustrates how the Blog domain model is mapped to the Blog database table.

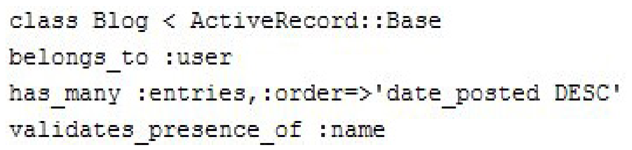
RoR's ActiveRecord is also an ORM framework. Thus, in Hibernate, the domain object model is kept separate from the database model, while ActiveRecord combines the two. Figure 5 shows how the 1-1 relationship between the Blog and User entities and the 1-N relationship between the Blog and Entry entities are recreated by specifying the "belongs_to" and "has_many" associations. Further, by following database table and entity class naming conventions, ActiveRecord dynamically maps object model attributes to database model attributes without configuration. This is achieved via Ruby's reflection and meta-programming features.

In .NET applications, the ADO.NET framework is typically used in the data access layer. ADO.NET is based on the Data Access Layer (DAL) pattern: the domain object model is kept separate from the database model by containing the mapping code in DAL objects. ADO.NET does not provide for ORM. Figure 6 illustrates mappings for the data layer.



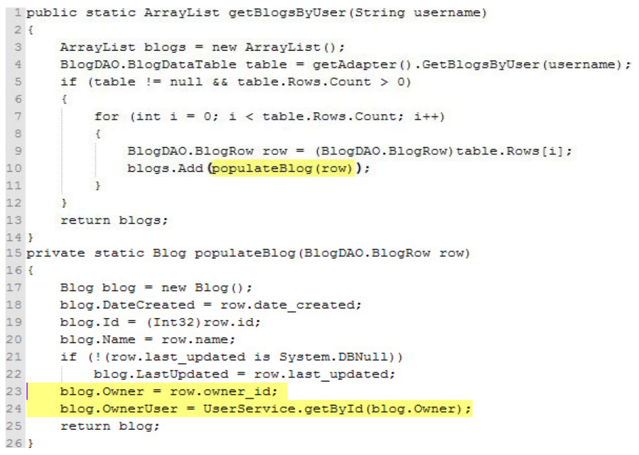
```
<class name="Blog" table="blog">
    <id name="id" column="id"/>
    <property name="name" type="string"/>
    ...
    <many-to-one name="owner" column="owner_id"
        class="blogissimo.model.user.User"/>
    <set name="entries" inverse="true">
        <key column="blog_id" not-null="true"/>
        <one-to-many class="Entry"/>
    </set>
</class>
```

Figure 4. Modeling the Blog entity and its relationships with Hibernate's ORM



```
class Blog < ActiveRecord::Base
belongs_to :user
has_many :entries,:order=>'date_posted DESC'
validates_presence_of :name
```

Figure 5. Modelling the Blog entity and its relationships with ActiveRecord



```
public static ArrayList getBlogsByUser(String username)
{
    ArrayList blogs = new ArrayList();
    BlogDAO.BlogDataTable table = getAdapter().GetBlogsByUser(username);
    if (table != null && table.Rows.Count > 0)
    {
        for (int i = 0; i < table.Rows.Count; i++)
        {
            BlogDAO.BlogRow row = (BlogDAO.BlogRow)table.Rows[i];
            blogs.Add(populateBlog(row));
        }
    }
    return blogs;
}
private static Blog populateBlog(BlogDAO.BlogRow row)
{
    Blog blog = new Blog();
    blog.DateCreated = row.date_created;
    blog.Id = (Int32)row.id;
    blog.Name = row.name;
    if (!(row.last_updated is System.DBNull))
        blog.LastUpdated = row.last_updated;
    blog.Owner = row.owner_id;
    blog.OwnerUser = UserService.getById(blog.Owner);
    return blog;
}
```

Figure 6. Mappings for data layer

5. Comparative evaluation of maintainability

In this section, we discuss the impact of platforms on maintainability of a WA. We consider factors such as WA architecture induced by platforms, component modularity, component coupling, interactions between components, and separation of concerns. We evaluate maintainability in terms of modifiability, testability, understandability, and portability.

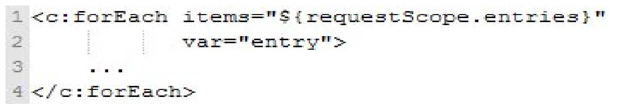
In order to compare the modifiability of software on three platforms, we implemented an enhancement by introducing a new entity Tag, to each of the three implementations. We then analyzed change propagation triggered by the enhancement. The Tag entity has a many-to-many relationship to the Entry entity in *Blogissimo* model Besides changes to data access and business layer, this new requirement also involved changes to the web layer, as the tags had to be displayed on the Entry-related pages. We collected the values of three maintainability metrics, namely the number of modified files, the number of modified LOC and the effort to implement change in man-hours.

Table 2. Change Propagation Analysis Results

App Layer	**No. of Modified Files**			**No. of Modified LOC**			**Effort** (man-hours)		
	JEE	*.NET*	*RoR*	*JEE*	*.NET*	*RoR*	*JEE*	*.NET*	*RoR*
Web	15	16	13	296	349	142	8	9	10
Business	3	3	3	114	217	34	3	5	1
Data Access	4	1		29	129		1.5	4	
Total	**22**	**20**	**16**	**439**	**695**	**176**	**12.5**	**18**	**11**

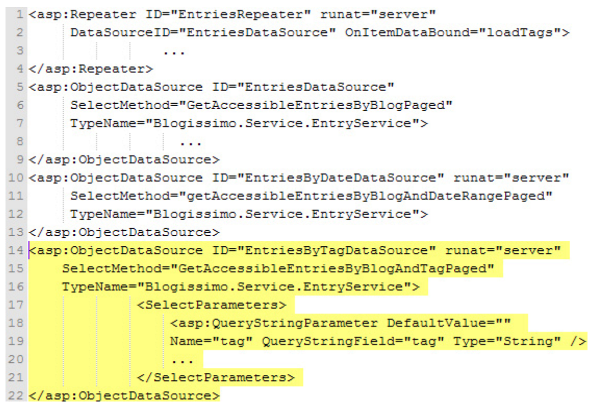
From Table 2 we observe that *Blogissimo* .NET required the most, while *Blogissimo* RoR required the least number of modifications to source code, as well as the least effort. The observation can be attributed mainly to two reasons. Firstly, enhancement for *Blogissimo* .NET required hand coding of the mapping from the Tag database entity to the Tag entity object, whereas J2EE's Hibernate and RoR's ActiveRecord automated much of this mapping. The second reason could be that .NET offers tighter coupling between concerns versus the cleaner separation of concerns in J2EE and RoR. For instance, in the .NET web layer, there was greater coupling of the View (Web Form) to the Controller (code-behind), whereas in J2EE and RoR, View and Controller were cleanly separated.

Figure 7 to Figure 9 show code excerpts for displaying a collection of entries on the View Entries page in each of the implementations. Figure 7 and Figure 9 show that the Views in J2EE and RoR are highly decoupled from the Controller logic. Therefore, neither of the Views in J2EE and RoR had to be changed in order to implement entry filtering by tag. In contrast, Figure 8 shows that in *Blogissimo* .NET, the View contains more than presentation logic as it specifies different Controller methods used to perform different types of entry filtering. Thus, to implement filtering by tag, the lines 14 to 22 (highlighted in Figure 8) had to be added to the Web Form. This demonstrates the tighter coupling between concerns in .NET, which in turn can be considered as a reason for poorer modifiability of *Blogissimo* .NET as compared to *Blogissimo* J2EE and *Blogissimo* RoR.



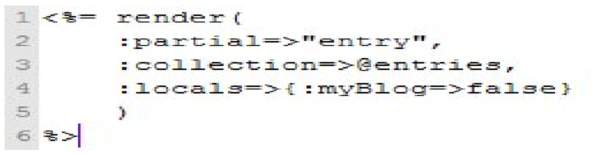
```
<c:forEach items="${requestScope.entries}"
           var="entry">
    ...
</c:forEach>
```

Figure 7. Excerpt from View Entries on J2EE



```
 1 <asp:Repeater ID="EntriesRepeater" runat="server"
 2      DataSourceID="EntriesDataSource" OnItemDataBound="loadTags">
 3                ...
 4 </asp:Repeater>
 5 <asp:ObjectDataSource ID="EntriesDataSource"
 6      SelectMethod="GetAccessibleEntriesByBlogPaged"
 7      TypeName="Blogissimo.Service.EntryService">
 8                  ...
 9 </asp:ObjectDataSource>
10 <asp:ObjectDataSource ID="EntriesByDateDataSource" runat="server"
11      SelectMethod="getAccessibleEntriesByBlogAndDateRangePaged"
12      TypeName="Blogissimo.Service.EntryService">
13 </asp:ObjectDataSource>
14 <asp:ObjectDataSource ID="EntriesByTagDataSource" runat="server"
15     SelectMethod="GetAccessibleEntriesByBlogAndTagPaged"
16     TypeName="Blogissimo.Service.EntryService">
17              <SelectParameters>
18                  <asp:QueryStringParameter DefaultValue=""
19                  Name="tag" QueryStringField="tag" Type="String" />
20                  ...
21              </SelectParameters>
22 </asp:ObjectDataSource>
```

Figure 8. Excerpt from View Entries on .NET



```
<%= render(
      :partial=>"entry",
      :collection=>@entries,
      :locals=>{:myBlog=>false}
      )
%>
```

Figure 9. Excerpt from View Entries on RoR

WAs are difficult to test because components expect to be interacting in the context of HTTP requests, responses and sessions making it hard to test an individual component with dependencies on the HTTP context.

Testing of *Blogissimo* .NET was difficult because an aspx page cannot be instantiated without access to HTTPContext, Request and Response objects that are provided by ASP.NET when run in the web server. *Blogissimo* J2EE was relatively more testable, as Spring provides a mock implementation for each key Servlet interface, such as HttpServletRequest. A mock object allows to write code that tests a component without running the WA in a web server. The RoR, besides providing mock objects, offers a *breakpoint* feature that can be used to interactively test a WA. Invoking the *breakpoint* method causes a WA to execute in an Interactive Ruby (*irb)* interpreter session. This allows variable values at that point to be inspected or changed, by issuing Ruby statements to be run in the *irb* [10]. Table 3. below summarizes these observations.

Table 3. Testability comparison results

Platforms / **Testability Factors**	**J2EE**	**.NET**	**RoR**
Ability to test web components w/o web server	Yes, with mock objects	No built-in support	Yes, with mock objects
Interactive debugging of running WA	-	-	Yes, with breakpoint and irb
Overall	**Medium**	**Low**	**High**

We evaluate understandability in terms of size [5] and modularity [1]. LOC, a metric of program size, influences understandability adversely because the larger the program, the more effort is needed to understand it. Modularity, degree by which a system is decomposed into loosely-coupled, cohesive components, is influenced by module cohesiveness, inter-module coupling and the extent to which a separation of concerns is observed.

Table 4. *Blogissimo* size comparison

Application Layer	**Lines of Code (LOC)**		
	J2EE	*.NET*	*RoR*
Web	4347	3048	1224
Business	1643	1469	138
Data Access	557	740	
Total	**6547**	**5257**	**1326**

Table 4 compares the lines of code for each of the three implementations of *Blogissimo*. *Blogissimo* RoR is smallest due to RoR's Rapid Application Development (RAD) features. On the other hand, J2EE *Blogissimo* J2EE implementation required 125% more LOC than .NET and 487% more than RoR. Thus, in terms of size, the J2EE implementation requires the most effort to understand, while RoR requires the least.

J2EE's Spring IoC container externalizes the references between components. This allows components to contain only business logic, thus promoting cohesiveness and decreasing coupling [8]. Spring's declarative Aspect-Oriented Programming capabilities facilitate modularizing of crosscutting concerns [8]. For instance, transaction management, a crosscutting concern, was externalized and shared among business layer components. Spring MVC provides the *HandlerInterceptor* class, which can intercept requests and modify or execute actions before or after designated Controllers handle it [8]. This lets code that handles common concerns be shared among web Controllers.

In .NET, ASP.NET's custom user controls are reusable page components that can be included in any page that requires its services. This allows common code to be shared among pages. However, the coupling of code-behinds and Web Forms makes it difficult to separate Controller and View logic.

RoR encourages modularity through its *helper*, *partial* and *filter* features. For example, *Partials* were used in *Blogissimo* RoR to share code for the rendering of forms and entries among different views. This increased the cohesiveness of pages that included the *partials*, as extraneous responsibilities were delegated to the partials.

Table 5 summarizes the modularity comparison results.

The combination of Spring and Hibernate made *Blogissimo* J2EE the most portable implementation, as it was database-agnostic, persistence mechanism-agnostic and supported on multiple operating system platforms. Hibernate facilitates portability of data access code, as usage of Hibernate API eliminates the need for database-specific SQL. The use of Spring, together with the Data Access Object (DAO) pattern, made *Blogissimo* J2EE not only database-agnostic, but also persistence mechanism-agnostic. In *Blogissimo* J2EE, the DAO pattern was implemented by creating a DAO interface for each entity. A set of DAO classes implementing the interfaces via Hibernate was created. Lastly, the Spring IoC container was used to declaratively inject references to the Hibernate DAO classes into the business layer, thus externalizing the business layer's dependencies on Hibernate DAO classes.

Blogissimo RoR is database-portable and operating system-portable, but not persistence mechanism-portable. This is because RoR is based on the Active Record pattern, which means domain model objects contain not only business logic but are also responsible for persistency.

Blogissimo .NET was the least portable, because it was neither database-portable or persistence mechanism-portable nor was it operating system-portable. This was because ADO.NET does not provide database abstraction. Furthermore, Microsoft's .NET implementation is supported only on the Windows platform.

6. Discussion of results

Our current study is based on comparison of implementations of one small-scale WA with requirements similar to the basic requirements of most WAs, mainly, CRUD operations on domain objects. Blogissimo was developed by a single developer, while most enterprise WAs involve large developer teams. Also, the specifications and requirements of the Blogissimo WA are relatively simple compared to large- scale, enterprise-level WAs.

Thus, the results of our comparison are most applicable to small-scale WAs with simple CRUD requirements. In such a context, our results suggest that RoR WAs are more maintainable than J2EE and .NET WAs due to the RoR platform's positive influence on modifiability, understandability and testability of WAs implemented on it.

Our study can be supplemented by extending the comparison to WAs with enterprise-level requirements, and larger-scale WAs, as we believe the different nature of such WAs may yield results different from those reported in this paper.

Table 5. Modularity comparison results

Platforms / **Modularity Factors**	**J2EE**	**.NET**	**RoR**
Cohesion	High	High	High
Coupling	Very low, due to dependency injection	High	Low
Support for modularization of cross-cutting concerns	Very high; cross-cutting concerns are easily modularized with Spring without knowledge of AOP	Lower; .NET has features like interception, but usage requires advanced programming knowledge. Unlike Spring, there is no declarative support.	High; Ruby's open class definitions allow cross-cutting services to be added. RoR makes it easy via *partials* and *filters*.
Overall	**High**	**Low**	**Medium**

Enterprise WAs have more complex requirements such as interacting with external systems, and accessing one or more shared databases. In contrast, Blogisssimo accesses a single database exclusively, which means the schema can be defined to suit the application, and does not interact with external systems. Thus, the availability of platform support for enterprise needs such as interfacing with external resources, would not feature in our study. We conjecture that the J2EE platform, with its large number of specifications for enterprise functionalities, which are implemented by many software vendors, would rate favourably in a comparison of maintainability of enterprise WAs.

We believe that the dynamic class capabilities of Ruby may lead to maintainability issues in large-scale WAs, whereby requirements changes propagate across many WA modules. For instance, the ability to reopen class definitions means that a class definition could be redefined and spread over multiple files, without a developer's knowledge. This can adversely affect a developer's ability to understand the workflow of a WA. Furthermore, Ruby does not require type declarations for variables, parameters and return arguments which serve as a form of concise code documentation. The lack of type declarations may have little impact on understandability in a small-scale WA, where the small code size makes programmers to have an intimate understanding of the entire system's code. However, understandability (and hence maintainability) issues may emerge in a large-scale WA with many modules, as a result of the greater effort needed for programmers to understand code, especially for code written by others. In summary, the dynamic nature of Ruby may impede program understandability and create control and coordination issues, as the scale of the WA development project grows.

A related work on measuring maintainability of WAs is a model for assessing the maintainability of a WA proposed in [11]. This model was adapted from Oman and Heigemaster's model for estimating maintainability of traditional software [12], and comprises metrics specific to WAs, such as web page data coupling. An approach to reverse engineering WAs, with the support of a software tool called WARE, to better comprehend and maintain WAs, has been presented in [13]. The field of WA platforms continues to evolve. For instance, TurboGears [14] is an up-and-coming Python-based WA platform popular for its productivity. Hence, future work should take into account new developments in the field of WA platforms, and extend the comparison to include new platforms that are popular enough to merit analysis of the maintainability of WAs built on them.

Acknowledgement : This work was supported by NUS Research Grant R-252-000-239-112.

References

[1] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," Second Int'l Conf. Software Eng., 1976, IEEE C.S. Press, pp. 592-605.

[2] http://www.ibm.com/developerworks/linux/library/wa-rubyonrails/

[3] http://www.theserverside.com/news/thread.tss?thread_id=35202

[4] M. Frappier, S. Matwin, and A. Mili, "Maintainability: Factors and Criteria", Software Metrics Study, Tech. Memo. 1, Canadian Space Agency, St-Hubert, Canada, 1994.

[5] M. Genero, M.Piattini, E. Manson and C. Cantone, "Building UML Class Diagram Maintainability Prediction Models Based on early Metrics", Proc. 9th International Software Metrics Symposium, Sep 3-5, 2003, pp. 263-275

[6] Curt Hibbs, "Rolling with Ruby on Rails" , The Open Source Web Platform onlamp.com, January 2005

[7] http://www.rubyonrails.com/

[8] R. Johnson, J. Hoeller, A. Arendson, T. Risberg, and C. Sampaleanu, Professional Java Development with the Spring Framework: Wrox, 2005.

[9] M. Fowler, Patterns of Enterprise Application Architecture: Addison-Wesley Professional, 2002.

[10] D. Thomas and D. H. Hansson, Agile web development with rails: Pragmatic Bookshelf, 2005.

[11] G.A. Di Lucca, A.R. Fasolino, P. Tramotana, and C. A. Visaggio, "Towards the definition of a maintainability model for web applications," Proc. of the 8th IEEE European Conference on Software Maintenance and Reengineering, CSMR 2004, IEEE C.S. Press, pp. 279 - 287.

[12] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability," Proc. of IEEE International Conference on Software Maintenance, ICSM 1992, IEEE C.S. Press, pp. 337-344.

[13] G.A. Di Lucca, A.R. Fasolino, F.Pace, P.Tramontana, and U. D. Carlini, "WARE:a tool for the reverse engineering of Web applications," Proc. of 6th European Conference on Software Maintenance and Reengineering, CSMR 2002, , pp. 241-250.

[14] http://turbogears.org