Implementing the Microsoft® .NET Pet Shop using Java

Featuring JPetStore: Open Source Edition

Version 1.2.0

"Java is not a product, it's an industry."
--James Gosling

Created: November 18, 2002

Prepared by Clinton Begin

Table of Contents

A note about this Revision

This is a very important update to the original JPetStore white paper. In the original comparison an error was made while counting the lines of code for JPetStore. After correcting the error it was found that JPetStore was in fact developed using fewer lines of code than the .Net Pet Shop. The error was made by accidentally counting all of the lines of JSP code in the JPetStore application. Later it was discovered that the CLOC utility used by Microsoft only counts server side blocks (<%%>) in JSP files. This made a dramatic difference in the line counts in favor of JPetStore and resulted in JPetStore using 330 fewer lines of code. All new line counts are now done using Microsoft's CLOC utility that is included with their .Net Pet Shop download available from their website at the following URL:

http://www.gotdotnet.com/team/compare/petshop-1.5.2.tar.gz

Although these are the rules created by Microsoft, the author of JPetStore disagrees with this methodology and believes that all lines of code in JSP files should be counted. The author also maintains that line counts are not alone indicative of developer productivity. For all respects and purposes the author would be happy to consider the .Net Pet Shop "equal" in terms of lines of code.

Abstract

The Java 2 Platform is more productive, cheaper and provides more platform choices than Microsoft's .Net. This white paper and the JPetStore application proves this while avoiding any marketing hype or corporate agendas.

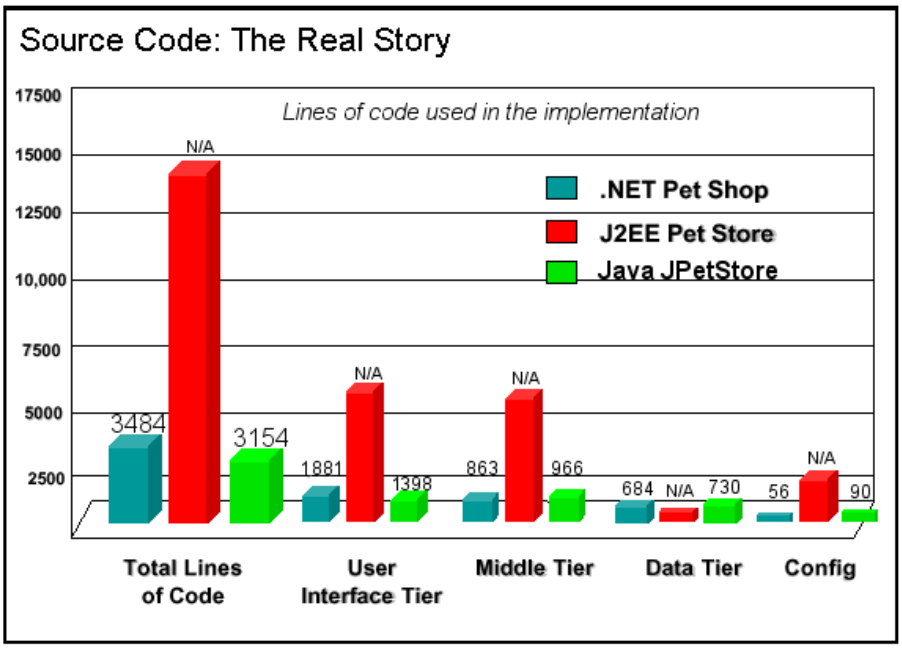
JPetStore is a completely new implementation of Pet Store demo application written in Java. While using Java to develop the Pet Store application, the following benefits were realized when comparing Java to .Net:

- Java Enabled Higher Developer Productivity: JPetStore was implemented in less time, with fewer resources and using fewer lines of code than the .Net Pet Shop. Java was found to be 4x more productive than .Net in the implementation of the Pet Store application.

- The Java Platform is Lower Cost (FREE): JPetStore can (and was) implemented using completely free tools and runtime platforms, including the development tool, frameworks, application server and the database management system.

- Java is Vendor Independent: At the time of this writing JPetStore runs on at least 4 operating systems, 6 application servers and 7 database management systems. This portability is achieved with no changes to the application source code. Such vendor independence enables greater choice of operating systems, more scalable hardware options and a much wider price range (starting at $0.00).

The source code for JPetStore is available at *http://www.ibatis.com*, the source code for the .Net Pet Shop is available at *http://www.gotdotnet.com/compare/*, and the source code for the Sun's J2EE Pet Store is available at *http://java.sun.com/j2ee/blueprints/index.html*.



Background

When Sun Microsystems created their original Java Pet Store, they clearly stated that it is a sample application to illustrate basic usage of J2EE technology and demonstrate current best practices in system design. The intent of the Sun Java Pet Store is to cover as much of the platform as possible, as clearly as possible, in a relatively small application. The intent of Sun's implementation is not to implement the Pet Store requirements using as few lines of code as possible, nor is it to be used as a performance benchmark. Although it demonstrates best practices, it tries to demonstrate all J2EE features and every best practice imaginable. So, it is bloated to say the least. One could simply say that Sun's J2EE Pet Store is "overarchitected". Shortly after Sun created the Java Pet Store, Oracle opened the door to benchmarking the Pet Store application in order to demonstrate the performance of the Oracle application server. Microsoft quickly followed suit and made what many consider to be an unfair comparison of the J2EE Pet Store to their .Net Pet Shop. The comparison is considered unfair because the designs of the applications are so dramatically different, that almost any comparison of the two would be technically flawed. Furthermore, the .Net Pet Shop was clearly written *for the benchmark*. This can be clearly seen by their use of known anti-patterns (worst practices) to achieve the highest possible performance. This makes the .Net Pet Shop a poor example of a real world application and therefore is irrelevant.

Introduction

This document describes a new Pet Store implementation called JPetStore. The new JPetStore application is based on open-source tools and frameworks that are freely available. JPetStore has been fully developed and deployed without spending a dime on tools (e.g. IDE) or runtime software (e.g. application server).

A Note on Terminology

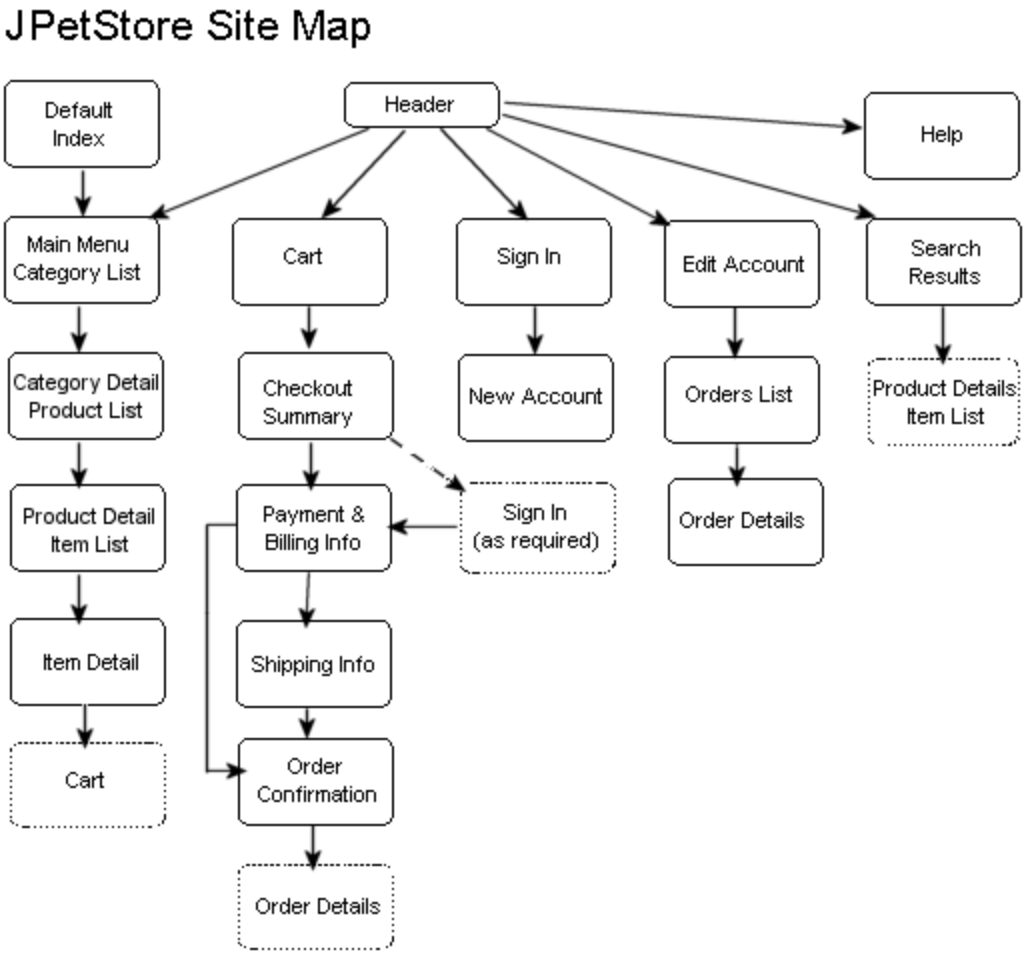
Now that there are at least three Pet Store like applications, it may be confusing as to which one is being referred to in a particular case. For that reason this document will use the following terms for the various Pet Store flavors:

- The term *Pet Store* will be used generically to refer to any and all Pet Store implementations and is usually used to refer to the requirements or the general functionality of the application.
- *JPetStore* will refer to the new application, based on open-source frameworks and tools, that this document was written to discuss.
- *Java Pet Store* or *J2EE Pet Store* will refer to Sun's original Pet Store implementation and/or Oracle's enhanced Pet Store application. The differences between the two were minimal and not relevant to this document.
- *.Net Pet Shop* or *Microsoft Pet Shop* refer to Microsoft's implementation of the Pet Store application.

What does Pet Store Demo Application do?

The Pet Store application was originally implemented by Sun and had many more features than the .Net Pet Shop. The Microsoft .Net Pet Shop included only a subset of the J2EE Pet Store application features that includes a basic online storefront and shopping cart.

The Pet Store application feature set will be familiar to most users of the Web. The customer can browse through a catalog of pets that vary by category, product type and individual traits. If the customer sees an item they like, they can add it to their shopping cart. When the customer is done shopping, they can checkout by submitting an order that includes payment, billing and shipping details. Before the customer can checkout, they must sign in or create a new account. The customer's account keeps track of their name, address and contact information. It also keeps track of profile preferences including favorite categories and user interface options (banners etc.). Below is a site map of the general flow of the application.



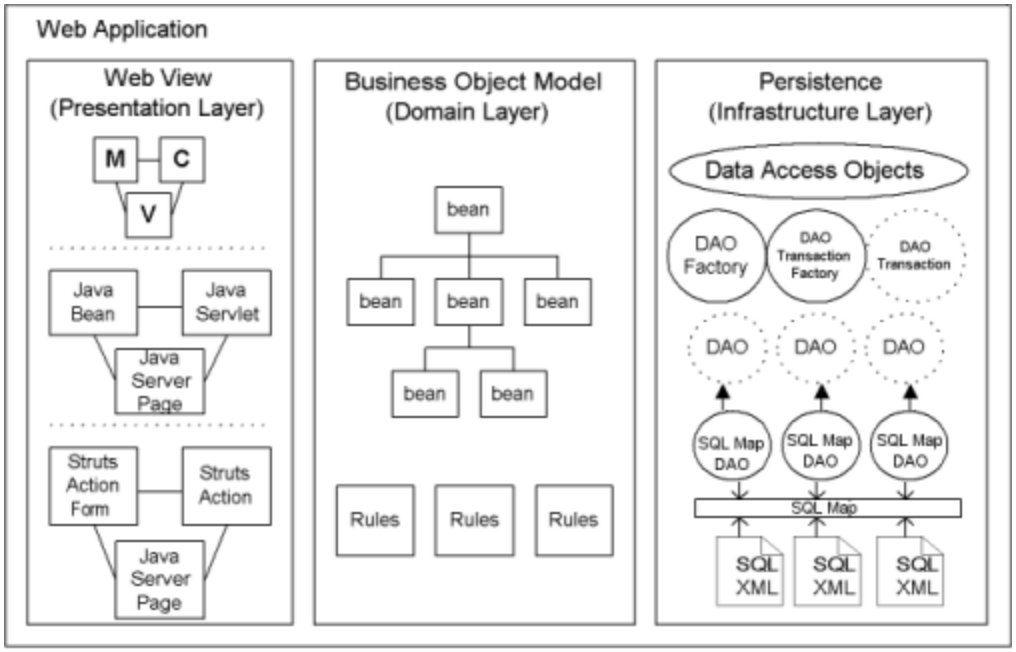
Enter JPetStore: Open Source Edition

JPetStore is a completely rewritten Pet Store application that is implemented using a design much like Microsoft's .Net Pet Shop application, but without many of it's shortcomings. JPetStore is completely vendor independent. It can run on various application servers and use a wide range of databases with no changes to the Java code.

The most noteworthy design differences between JPetStore and Microsoft's .Net Pet Shop are:

- JPetStore does not use Stored Procedures, nor does it embed SQL in the Java code.
- JPetStore does not store HTML in the database.
- JPetStore does not use generated code.
- JPetStore uses the Model View Controller presentation pattern.
- JPetStore was implemented using completely open-source freeware, including the development tool, runtime environment and database management system.

JPetStore uses the Model View Controller pattern to improve the maintainability of the presentation layer –the layer which is often the most likely to change. The JPetStore persistence layer uses SQL mapped to Java classes through XML files. The advantages to this are clear: the elimination of stored procedures means JPetStore is independent of the database vendor, and eliminating the SQL from the Java source code means improved maintainability in the data tier. The picture below shows a high level view of the architecture of the application.



Some Point-for-Point Design Comparisons

One of the goals of the JPetStore application was to implement the Pet Store application using a design similar to Microsoft's, therefore creating an application that can be more fairly compared to the .Net Pet Shop. This is a difficult task when the implementations are in such drastically different environments. This section identifies some of the similarities and differences in the design. For a discussion of Microsoft's design, please see their white paper at: *http://www.gotdotnet.com/compare*.

Database & Persistence Layer

The database design was largely unchanged. The major differences were in the elimination of stored procedures and some improvements to the table definitions. During implementation of JPetStore, the author was uncomfortable with having HTML code in the database. This is a very poor practice and complicates user interface design as well as reduces the value of the data in the database. In summary the differences in database design/implementation are:

- Eliminated all stored procedures.
- Added "image" column to the Product table to store image filename and normalized the original "DESCN" column data.
- Added "image" column to the Category to store image filename and normalized the original "DESCN" column data.

To replace the stored procedures without having to write complicated JDBC code, a simple object/relational mapping framework was employed. The framework, called iBATIS Database Layer, uses simple XML descriptor files (included in the line count) to describe the inputs and outputs of an SQL statement. It allows the programmer to simply pass a JavaBean into a MappedStatement as a parameter (input) and receive a JavaBean as a result (output).

The following example shows how JPetStore uses a clean, vendor independent API to create a new record in the Account and Profile tables. The method looks like this:

JPetStore Uses a Very Clean API for Object/Relational Mapping

```
public void insertAccount(DaoTransaction trans, Account account)
    throws DaoException {
  executeUpdate("insertAccount", trans, account);
  executeUpdate("insertProfile", trans, account);
}
```

The same functionality in the .Net application used a call to a proprietary stored procedure that only works on Microsoft's SQL Server. It's on the next page because it needed an entire page to itself!

The .Net Pet Shop uses a complex, unhelpful and vendor dependent persistence API

```
public string Add(string userid, string password, string email, string
firstName, string lastName, string address1, string address2, string
city, string state, string zip, string country, string phone, string
languagePref, string favoriteCategory, int mylistOption, int bannerOptions) {

  Database data = new Database();

  SqlParameter[] prams = {
    data.MakeInParam("@userid",      SqlDbType.VarChar, 80, userid),
    data.MakeInParam("@password",    SqlDbType.VarChar, 25, password),
    data.MakeInParam("@email",       SqlDbType.VarChar, 80, email),
    data.MakeInParam("@firstname",   SqlDbType.VarChar, 80, firstName),
    data.MakeInParam("@lastname",    SqlDbType.VarChar, 80, lastName),
    data.MakeInParam("@addr1",       SqlDbType.VarChar, 80, address1),
    data.MakeInParam("@addr2",       SqlDbType.VarChar, 80, address2),
    data.MakeInParam("@city",        SqlDbType.VarChar, 80, city),
    data.MakeInParam("@state",       SqlDbType.VarChar, 20, state),
    data.MakeInParam("@zip",         SqlDbType.VarChar, 20, zip),
    data.MakeInParam("@country",     SqlDbType.VarChar, 80, country),
    data.MakeInParam("@phone",       SqlDbType.VarChar, 80, phone),
    data.MakeInParam("@langpref",    SqlDbType.VarChar, 25, languagePref),
    data.MakeInParam("@favcategory", SqlDbType.VarChar, 25,
favoriteCategory),
    data.MakeInParam("@mylistopt",   SqlDbType.Int,      4, mylistOption),
    data.MakeInParam("@banneropt",   SqlDbType.Int,      4, bannerOptions)
  };

  try {
    DataReader
    int retval = data.RunProc("upAccountAdd", prams);

    if (retval == 0 )
        return userid;
    else
        return null;
    } catch (Exception ex) {
        Error.Log(ex.ToString());
        return null;
    }
}
```

The method above shows not only how the use of stored procedures actually complicated the C# code, but it also shows how poor the implementation of the .Net Pet Shop really was. The Add() method has sixteen (16!) parameters. Imagine what would happen in a more realistic enterprise application. Even if the method parameters were wrapped up in a single convenient parameter (e.g. CustomerDetails class), the number of individual calls to data.MakeInParam() would continue to increase as columns were added. Worse yet, the database column type and size are hard coded inside the application code! This means that to allow for greater field width in a particular column the application code needs to be recompiled (e.g. State needs to accommodate "Northwest Territories" during internationalization of .Net PetShop).

Presentation Layer: Model-View-Controller

JPetStore implements a very well established and widely accepted pattern for separating presentation code from business logic. This pattern is the Model View Controller pattern. To avoid writing yet another description of MVC, here is an excerpt from Sun's website that describes the pattern.

Source: http://java.sun.com/blueprints/patterns/j2ee_patterns/model_view_controller/index.html

By applying the Model-View-Controller (MVC) architecture to a J2EE application, you separate core data access functionality from the presentation and control logic that uses this functionality. Such separation allows multiple views to share the same enterprise data model, which makes supporting multiple clients easier to implement, test, and maintain.

The MVC architecture has its roots in Smalltalk, where it was originally applied to map the traditional input, processing, and output tasks to the graphical user interaction model. However, it is straightforward to map these concepts into the domain of multi-tier enterprise applications:

- The **model** represents enterprise data and the business rules that govern access to and updates of this data. Often the model serves as a software approximation to a real-world process, so simple real-world modeling techniques apply when defining the model.
- A **view** renders the contents of a model. It accesses enterprise data through the model and specifies how that data should be presented. It is the view's responsibility to maintain consistency in its presentation when the model changes. This can be achieved by using a *push* model, where the view registers itself with the model for change notifications, or a *pull* model, where the view is responsible for calling the model when it needs to retrieve the most current data.
- A **controller** translates interactions with the view into actions to be performed by the model. In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in a Web application, they appear as `GET` and `POST` HTTP requests. The actions performed by the model include activating business processes or changing the state of the model. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.

The MVC architecture has the following **benefits**:

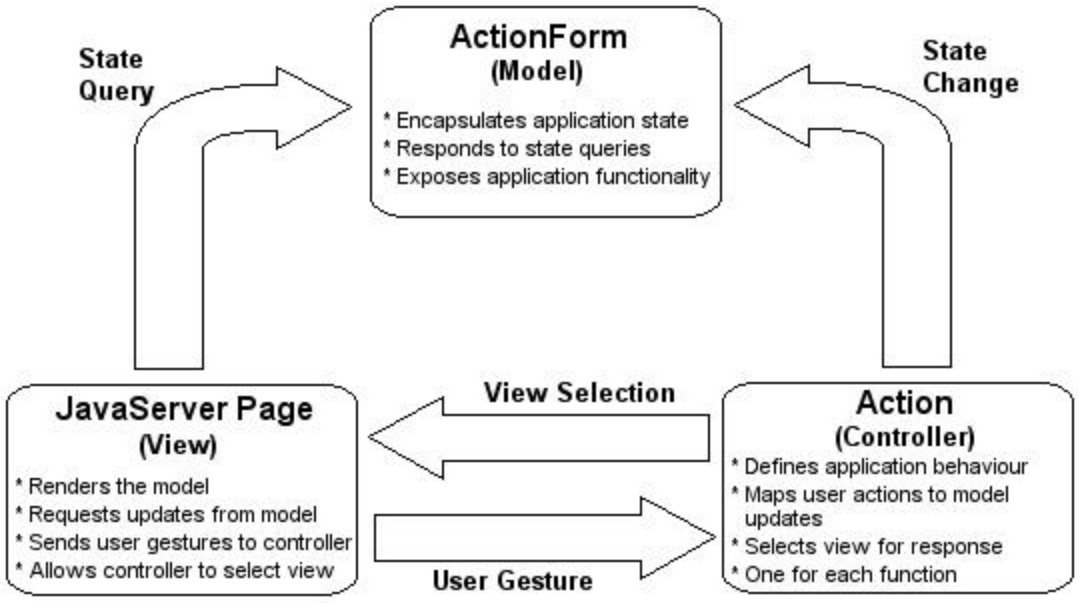
- **Multiple views using the same model.** The separation of model and view allows multiple views to use the same enterprise model. Consequently, an enterprise application's model components are easier to implement, test, and maintain, since all access to the model goes through these components.
- **Easier support for new types of clients.** To support a new type of client, you simply write a view and controller for it and wire them into the existing enterprise model.

An MVC Framework: Struts

The framework that was used by JPetStore to implement the Model View Controller pattern was the popular framework known as Struts (http://jakarta.apache.org). Struts, like everything used by JPetStore, is freely available and open-source. The Struts framework played a key role in the design and implementation of the JPetStore presentation layer. Combined with JSP, it helped maintain a consistent look and feel as well as good flow control throughout the application. It did this while helping reduce the overall code size for JPetStore and improving the overall design.

Struts has three main components: the ActionForm (model), the JavaServer Page (view) and the Action (controller). Struts uses an XML descriptor file to connect these three components together, which helps simplify future code maintenance on the presentation layer –a layer prone to change. The diagram below illustrates the three main components and how they interact with each other.

It is important to note that writing these components is very simple, lightweight and is almost exactly like writing a normal Servlet. The largest Struts-based source file (Action or ActionForm) in JPetStore is 121 lines of code.



Based on work from: http://java.sun.com/blueprints/patterns/j2ee_patterns/model_view_controller/index.html

Implementing JPetStore: Java Infrastructure

In Microsoft's white paper titled "Implementing Sun's Java Pet Store using Microsoft .Net", Microsoft claims that the code in the J2EE implementation was required to make up for a lack of infrastructure. The fact is, Sun simply chose to stick to the APIs that are core to J2EE. They didn't make use of any other frameworks or infrastructure. Sun never promised the universe, they just started the fire. The big bang was left to the rest of the industry.

Tools & Frameworks

As mentioned previously, JPetStore was implemented completely using open-source freeware. In the Java industry, there are numerous tools, frameworks and infrastructure components to satisfy nearly any requirement. Many of these are freely available with source code included. The JPetStore implementation used a number of such frameworks, including the very popular Struts MVC framework. The following table lists the tools and frameworks along with their associated cost (as unnecessary as it may be):

Category	Product	Cost
Integrated Development Environment	NetBeans IDE http://www.netbeans.org	$0.00
Model View Controller Framework	Jakarta Struts http://jakarta.apache.org	$0.00
Object/Relational Mapping	iBATIS DAO/SQL Map Framework http://www.ibatis.com	$0.00
Format Tag Library	Formatter TagLib, by Tak Yoshida http://www.ibatis.com	$0.00
Compiler & Software Development Kit	Sun Java 2 SDK http://java.sun.com	$0.00
		Total: $0.00

Runtime Environment

In addition to the tools and frameworks above, there are a number of free, open-source runtime environments for Java, including application servers, web servers, Servlet/JSP containers and relational database management systems. The following table lists the runtime environment that JPetStore was primarily intended for, although JPetStore can run on any J2EE compliant application server and with any JDBC compliant database.

Category	Product	Cost
Web Server & Servlet/JSP Container	Jakarta Tomcat http://jakarta.apache.org	$0.00
Relational Database Management System	PostgreSQL http://www.postgres.org	$0.00
Operating System	Red Hat Linux http://www.redhat.com	$0.00
		Total: $0.00

Developer Productivity

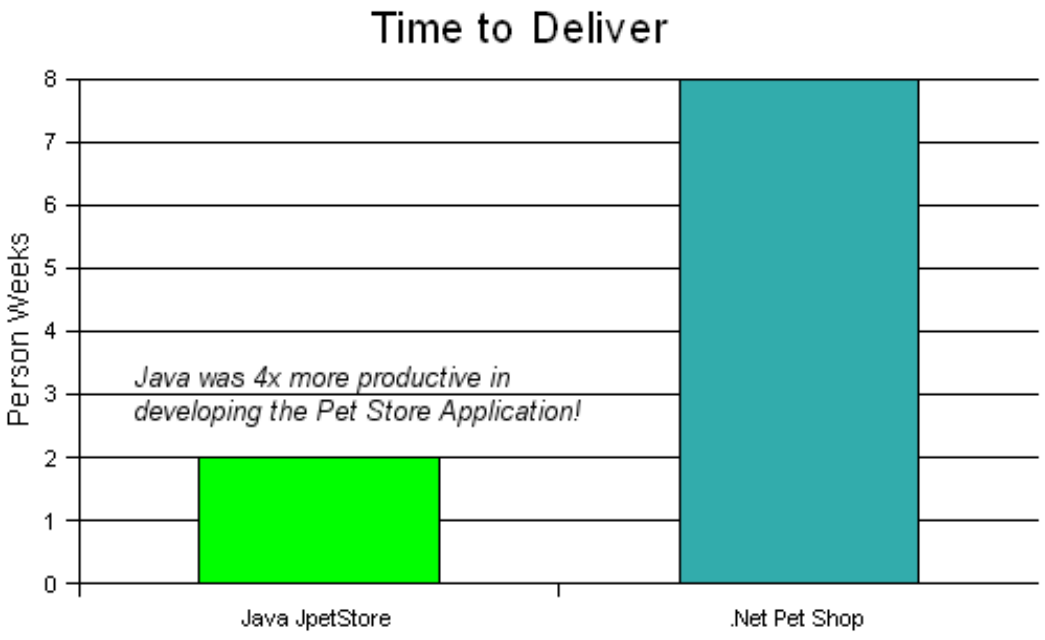
The .Net Pet Shop was developed, at Microsoft's request, by Vertigo Software. In an interview by Lamont Adams, published by Builder.com, Scott Stanfield, CEO of Vertigo Software made the statement: "We did the whole [.Net Pet Shop] with two developers in a total of five weeks, which includes about three to four days of load testing and benchmarking at the end." Hence a conservative estimate would be that the development time was about four weeks with two developers, or about eight person weeks.

In contrast, JPetStore was developed by a single developer in his spare time (i.e. not his full time job) over about two weeks. Assuming 4 hours per evening and 12 hours on weekend days a more-than-fair estimate would be about 11 person days or about 2 person weeks. Even if these estimates are doubled, that would still mean that Java was about twice (2x) as productive as the .Net development environment.

Are these productivity gains constant? No, of course not. There will be situations where any given technology will outperform another. The important point to consider here is that the claim made by Microsoft is absolutely false and Java is just as productive or possibly more productive than .Net and C#.

The skill level of the developer(s) has a lot to do with these types of comparisons. The benefit of Java is that it is a mature technology and has a good following of developers. The pool of talented Java developers available is much greater than that of C# and Java has a 7 year head start in developer mind-share. Therefore Java has greater network effect*. Although there is a great demand for Java skills, the number of talented Java developers is also great and the ratio of supply to demand is significantly in favor of Java.

*Metcalfe's Law: *the usefulness, or utility, of a network equals the square of the number of users*.

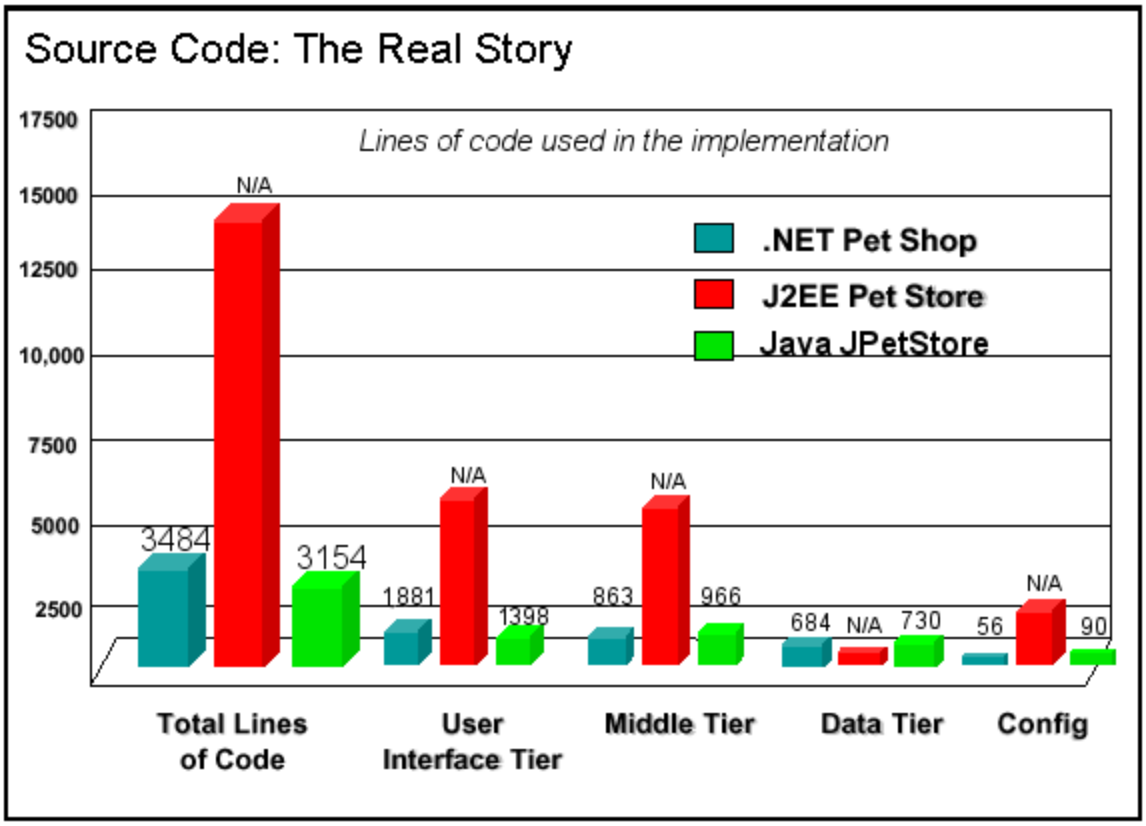


Lines of Code "Required" vs. "Used"

In the same white paper mentioned above, Microsoft claims that using Java to implement the Pet Store application "required" over 14,000 lines of code. This is simply not the case. No piece of software ever written has absolutely required any particular amount of code (although there may be theoretical minimums and maximums). Assuming somewhat similar languages (e.g. C# and Java), the lines of code required to implement a solution is generally a matter of design and is more the result of decisions made by the designers and developers than anything else. There are so many design and infrastructure choices in Java that the possibilities are quite endless.

JPetStore demonstrates a design that the author decided was appropriate for the requirements of the Pet Store application. It was implemented not to simply reduce the line count to below that of the .Net implementation, but rather to implement the Pet Store requirements using a smart design with no dependencies on any particular vendor. The author decided that Pet Store was best implemented in Java using a design that required just over 3100 lines of code –300 fewer than the .Net Pet Shop.

On the other hand, the .Net Pet Shop was developed using anti-patterns (worst practices) to achieve low line counts and false performance. In addition, this was done at the expense of being tied to Microsoft SQL Server through the excessive use of stored procedures. Furthermore, the implementers of the .Net Pet Shop have discarded the industry accepted, best-practice presentation pattern called Model-View-Controller and instead chose a design that is unproven and unique to .Net.



Is it possible to implement Pet Store in Java with fewer lines of code?

Absolutely. The functional requirements of the Pet Store surely could have easily been worked out in a couple thousand lines of Java code. If lines of code was the only measure of success in software development, then JPetStore would have probably been implemented to that effect.

How to Eliminate another 300 Lines of Code from JPetStore

It's important to note that simple coding conventions between languages can have a dramatic effect on the number of lines of code. For example, examine the following two implementations of the Product ID property below. The example clearly shows that there is a difference of about 2 lines of code *per property* that is simply due to different coding conventions between the two languages. By eliminating such differences, the author could have easily shaved off another 300 lines of code in JPetStore by following a convention similar to that of C# (140 lines from properties alone). He chose not to do so to keep consistent with Java coding conventions.

.Net Property; 5 lines	**JavaBeans Property; 7 lines**

`private string m_productid;`
 	
`public string productid {`
`  get { return m_productid; }`
`  set { m_productid = value; }`
`}`
	`private String productId;`
	
	`public String getProductId(){`
	`  return productId;`
	`}`
	
	`public void setProductId(String productId){`
	`  this.productId = productId.trim();`
	`}`

A Word on File Extensions

The extensions included in the line counts were not consistent with those that Microsoft counted in their comparison, as it would not have made sense. For example, to implement the JPetStore application the author did not have to write a single TLD (tag library descriptor) file. This would have resulted in a configuration line count of zero, which would have been inaccurate. Therefore for the JPetStore application "properties" files were counted as configuration.

What was Counted?

The following extensions were included in the line counts: java, xml, jsp, properties, sql. All source code was counted excluding blank lines and Java comments. The code was counted using Microsoft's own CLOC utility, the same utility that was used to count the lines of code for both the .Net Pet Shop and the Sun J2EE Pet Store.

Generated Code: The Ultimate Vendor Dependency

Some implementations of the Pet Store application use generated code. This includes Microsoft's .Net Pet Shop (e.g. ASP Forms) as well as a few Java/J2EE implementations. In both cases, it is generally a very bad idea to generate code beyond very simple code. For example, generating JavaBeans properties is not usually harmful, because JavaBeans properties can be hand-written very easily and with very little difference in implementation cost. Also, generating JavaBeans properties will not result in any particular vendor dependency.

On the other hand, generating complex application architecture code is very risky because the foundation of the application is at risk of becoming dependent on a particular vendor tool or product. Furthermore, it is often the case that even though the code is generated initially, there are very few (if any) code generators that are sophisticated enough such that the code will not have to eventually be manually modified to suit the application requirements. To avoid these and other potential risks, the following guidelines were used in implementing JPetStore:

- Never generate code that will result in a dependency on a particular vendor or product.
- Beware of generating code that would be significantly more costly to write by hand (it will likely have to maintain it by hand anyway –think Total Cost of Ownership)
- Keep generated code to a minimum (one J2EE Pet Store implementation generated over 22000 lines of Java code with only a few thousand hand written lines).
- Reserve generated code for the uppermost layers of the application (i.e. Presentation Layer).
- Never generate code that you would never write by hand (because it will eventually need to maintained by hand anyway –count on it!).
- Avoid generating Infrastructure Layer and Domain Layer code.

```
Example of Vendor/Tool Dependency: From .Net PetShop Product.aspx.cs
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
Moral: Don't generate code that results in a dependency on a tool!
```

Fewer Files is not Necessarily Better

In their comparison of Sun's J2EE Pet Store to the .Net Pet Shop, Microsoft showed how they could jam pack 3484 lines of code into 65 files. Once again, this was done at the cost of their design which was in total conflict with known best practices. This can clearly be seen in how the .Net Pet Shop domain layer is infected with persistence code that ties the application not only to the specific implementation approach (use of stored procedures), but also ties them to the vendor (Microsoft, ADO.Net). Worse yet, it creates bloated classes that are difficult to maintain. For example: in implementing the Customer/Account/Profile functionality of the Pet Store application, the .Net Pet Shop used over 300 lines of code and 3 classes that were jammed into a single source file. The JPetStore instead chose to separate the persistence layer from the domain layer and in doing so used less than 200 lines of code, split into 2 logically separated files. Most any developer would surely agree that the latter approach is simpler, more maintainable and a better design approach overall.

Conclusion

Sun's J2EE Pet Store application was never intended to be a benchmark application for measuring efficiency, productivity or performance. It was simply intended to demonstrate the functionality of the core J2EE APIs. Despite the intent, Oracle opened the doors for the "Pet Store performance benchmark". Microsoft then took it a step further with its .Net Pet Shop and unfairly and irresponsibly dragged Sun's Pet Store into a competition comparing of lines of code as well as performance.

JPetStore is an implementation of the Pet Store application that has successfully demonstrated that Java is more productive, cheaper and offers more choice than Microsoft's .Net. In addition it has shown how it can do this while achieving a superior design. Most importantly though, JPetStore has proven that Java truly "is an industry" in which anyone can implement a professional solution without paying a dime for development tools or runtime software.

Where are the Performance Benchmarks?

JPetStore has not been tested for performance because the conditions under which the .Net Pet Shop were tested could not be recreated. At this time, the testing software and testing platforms are not accessible to the author. Some performance benchmarks may be posted at http://ww.ibatis.com, however it is not guaranteed that these will be directly comparable to the .Net performance statistics gathered by Microsoft. If you would like to help performance test JPetStore, please contact clinton.begin@ibatis.com. A description of the test configuration and tools used by Microsoft .Net are available at http://www.gotdotnet.com/team/compare/Benchmark_ShortRepFinal.pdf.

Who is Clinton Begin and what is iBATIS

Clinton Begin is a software developer from Calgary Alberta (Canada). For nearly four years he has been employed as a Java Software Specialist in the Canadian oil and gas industry. iBATIS is Clinton's website (www.ibatis.com), nothing more. iBATIS is not a company, nor is it a product. Therefore, other than shameless self promotion, you will not be exposed to any sort of sales pitch or marketing campaign.

If you have any comments or suggestions regarding JPetStore or this document, please feel free to contact Clinton at: clinton.begin@ibatis.com