Component Oriented Software

Java Perspectives

By
Robin Sharp

www.softwarereality.com
www.javelinsoft.com

21 April 2002

Copyright © Robin Sharp 2002

This article may be freely distributed, under the following conditions:

1. The article is not modified in any way.
2. The article and this Acrobat file are left intact, i.e. nothing is left out (including this title page).
3. Distribution of the article is strictly for personal and not-for-profit use (e.g. forwarding it to a colleague). To discuss distribution for other purposes (e.g. magazine publication), please contact Robin Sharp at: robin.sharp@javelinsoft.com

Table of Contents:

1. Introduction

This paper introduces the concept of Component-Oriented software from a Java perspective. The purpose of this paper is to provide a high level description of the direction and momentum behind Java, its associated architectures, and to show where Java as a programming language could be heading.

Components are an over-used word in the Java language, and there is often confusion about where the classification begins and ends. This paper tries to put some flesh on the bones and explain how Java programming language features and architectures relate to Component-Oriented concepts.

In this discussion the paper makes a comparison between Java, C++ and .Net and takes a look at the evolution of the languages.

The paper concludes that languages develop by moving closer to the solutions, but evolve by moving closer to the problems. It further concludes that Java as a language has stagnated by focusing too much on moving closer to solutions and needs to make greater efforts in moving closer to the problems.

The paper is divided into the following sections:

1. Introduction
2. Language Evolution
3. Language Lifecycle
4. Object-Oriented Java
5. Component-Oriented Software
6. Component-Oriented Java
7. Aspect-Oriented Java
8. Conclusion & References

About the author:
Robin Sharp is the Managing Director of Javelin Software (http://www.javelinsoft.com). He can be reached at robin.sharp@javelinsoft.com

2. Language Evolution

The evolution of programming languages is best characterized as moving closer to the problem (business) domain. Moving closer to the problem domain means allowing a language to better express problem domains. This is contrasted to language development, which can be expressed as moving closer to the solution (technical) domain.

A classic example of moving closer to the problem domain is the move from functional programming to Object-Oriented programming. A classic example of moving closer to the technical domain is the development of networking API's from TCP to HTTPS.

Programming language evolution must be differentiated from technological evolution. Language and technical evolution have different frequencies and different manifestations. Technology's lifecycle is based on innovation and technical evolution and is generally manifested in declarative standards, for example TCP, SQL, HTML and XML. These standards, whilst disrupted by new languages, continue evolving independently.

As a programming language goes through its lifecycle, API's evolve to solve problems. When the language becomes inefficient, a new language emerges by extracting patterns from previous generation solutions.

Interesting fuel for language evolution does not come from the positive path of API successes, but the negative path of API inefficiencies. For example, the motivation for the uptake of C++ was that rigorously well written Object-Oriented software ended up passing a this pointer to method pointers on structures.

When watching language evolution it is important to separate the emergence of API features that represent language evolution from the emergence of APIs that represent technical evolution.

In terms of programming language evolution less is often more. Language evolution often takes the form of small extensions and small restrictions - assumptions are everything, and one-step backward means two-steps forward. Adding and removing a few small features often opens the door to new and efficient programming models. An example of an extension would be the movement from function pointers in C structures to methods as first class objects. An example of a restriction would be the removal of goto statements and then functions.

To date languages have evolved in 10-year cycles, and have taken off in economic upswings. The current state of the computing market sees Java and C# competing as two dominant programming models with enormous barriers to entry in terms of their libraries. Given these barriers to entry, Java and C# will probably remain competing languages over at least the next generation. The component oriented aspects of C# means it has leapfrogged over Java's libraries and VM.

It is important to understand where Java and C# are in their lifecycles. Java is on the cusp of its maturity phase and its inefficient stage, whilst C# is still early in it's adoption stage.

3. Language Lifecycle

In order to understand the direction of a programming language, it is important to understand its dynamics from both a technical and economic perspective. I have divided the lifecycle into the following stages:

1. Conception
2. Adoption
3. Acceptance
4. Maturation
5. Inefficiency
6. Deprecation
7. Decay

Conception
A language is conceived to meet a requirement that other languages do not meet. Countless languages have been written, but those C++ and Java have come from corporate research labs, rather than Universities (C++ from AT&T) and (Java from Sun), both have take off.

Adoption
Languages are adopted to make the programmer more efficient. This is driven by the programmers because they are bright and don't like mundane coding. Languages have API's back into the libraries of those they are replacing.

Acceptance
Once the market sees that tools are sufficiently bug free and early adopters have made profits there is a general acceptance of a language. Green field projects expose requirements and ideas for tools and libraries are conceived.

Maturation
Languages, libraries and tools mature. They are capable and deliver profitable solutions. There is an increased demand for functionality, rather than efficiency, in order to reap profits. Standards bodies attempt to control the spread of ideas.

Inefficiency
Development has become slow as libraries become more inefficient. The rush to provide functionality has created a market that is fragmented subtly fragmented by vendors implementing standards differently. There is a large increase in the number of case tools and code generators.

Deprecation
Developers start to understand that it is costly to develop using the language. Frustration begins to set in a designers take a deeper look at the issues causing the inefficiencies.

Decay
A young pretender appears and challenges the leader. Lean, mean and fast the new language has apparently come from a research department of a large technology company. Marketing departments lock horns and invariably the old order is over thrown.

4. Object-Oriented Java

Now that Java has passed the maturation point, it has fragmented into several architectures, such as J2SE (standard), J2EE (enterprise) and J2ME (mobile), each designed to cope with the different requirements imposed on it.

In addition to architecture fragmentation, within these architectures we are now seeing 2nd and even 3rd generation iterations. This has led to problems of API bloom, particularly in the area of User Interface and Persistence APIs.

Here is a list of just some of the various User Interface and Persistence APIs:

	User Interface	Persistence
Standard	• AWT • Swing	• JDBC/SQL-J • EJB • JDO • JavaSpaces • Long Term Persistence
Community	• HTML DOM • Java Faces • JSP Tag Library • MIDP • Java TV • Long Term Persistence	• Java Cache • Orthogonal Persistence • MIDP

Some circles have argued that J2EE represents a new language. In reality J2EE is the antithesis of Java in that it represents an attempt to build on the Java language. The early specification was oriented around remote entities. The current thinking is that the Sessions should be remote and that entities should be local, and the JDO camp is currently jostling for a place at the high table. J2EE has been stumbling from one solution to another, with no theoretical underpinning. Remarkably, they look no closer to understanding why they need a clear separation between data and architecture.

With all these various persistence APIs, we (at JavelinSoft) set about building a code generator to provide a generic persistence API to hide the ever-changing implementations. The Component-Oriented solution we achieved blended the Beans types with the EJB Session modules. With JGenerator we are able to generate persistence layers for In-memory, JDBC and EJB objects. Our Beans are Types and our Sessions are Modules.

One of the most telling moments when developing JGenerator was when we discovered that the ratio between our business descriptors and the (optimised) EJB code we had generated was a whopping 1:100 (and this code was minimal). We needed no further proof that Java had moved too close to the solution and too far from the problem.

Another area of class bloom is Java UI's. The Java Server Faces early goals made a reference to provide a universal UI interface. This has since been watered down to provide a server UI interface. Sun is not alone in maintaining this separation. The .NET Grid component has both a client and server implementation, both with different interfaces.

With all these UI API's we set about developing a generic UI API called Swinglets. The solution we achieved was almost identical to the Swing API. With Swinglets we are able to render a UI component model as HTML, WML, JFC or PDF.

5. Component-Oriented Software

The component-oriented thesis is best expressed as making the distinctions:

Modules vs. Types

and

Methods vs. Messages

Modules refer to the static architecture in a system, whilst Types refer to the dynamic data in a system. Modules are then supported in a framework (or matrix). The best analogy is between a product and the production line, where the framework is the factory.

When designing a system using an Object-Oriented language, the developer has to be disciplined not to merge the two aspects. This discipline is similar to that required when treating a functional language as Object-Oriented: such as C and C++.

You'd think the separation of architecture from data was easily achieved, but it's surprising how often you see the mistake made. Once you spot the mistake once, you'll see it repeated. For example the EJB Object has a remove method. The data has wrongly become part of the architecture.

The distinction in Component Oriented Software that's harder to make is the separation of Methods vs. Messages. Simply put, a method is an implementation whilst a message defines a set of interfaces. The thesis here is that modules conform to a message interface and can be implemented differently.

The idea that modules such as a UI or a persistence API can be generalized seems difficult to achieve, but Swing/Swinglets and JGenerator show that it is possible. The separation is more easily achieved by keeping a clear separation between data and architecture. The Model, View, Control design is a less abstract form of the Module, Type, Framework thesis.

6. Component-Oriented Java

So what is the best way for Javasoft to move, in order to provide a fully Component-Oriented language?

Component-Oriented class modifiers, if added to the language, let you declare a class or interface as a type, or module.

For example:

```
public type class Invoice
{
    public double amount;
}

public module class InvoiceSession
{
    public void process(Invoice invoice) throws ProcessException;
}
```

Declaring classes with these modifiers should give instructions to compilers to treat the classes meaningfully as data or architecture. Module and Type hooks could be included in the compiled code. The benefits of the Component-Oriented programming model would be immediate through the use of introspection.

Classes that were declared with the type modifier would allow accessor methods to be assumed as method declarations with the same access as the member declaration. The compiler could also make some compile time checks to enforce the bean standards, such as the existence of a default constructor.

Classes declared as types should be assumed to be Serializable. In addition type classes should be capable of mapping to XML binding and Persistence modules, and type classes should not reference module classes.

Classes that were declared with the module modifier would allow the methods to be assumed to be services. The compiler could also make some compile time checks to ensure that their methods only passed Serializable objects. This would mean they could be exposed either locally or remotely.

Classes declared as modules should be assumed to be Remotable. In addition module classes Remote Interfaces would need to be extended to seamlessly handle other protocols such as XML Schema or SOAP method bindings.

If these features were added to Java, aspect and feature oriented extensions would have something to get their teeth into. Property methods could be designed to have hooks for prefix and postfix processing.

7. Aspect-Oriented Java

Aspect-Orientated is often related to Component-Oriented programming. Aspect-Orientated software is the idea that code fragments need to be applied in a consistent manner across a large number of classes.

Some good examples of this are:

- Pre and post method calls
- Validation
- Logging
- Debugging
- Monitoring
- Firing Property Change Events

Languages like AspectJ allow Java developers to consistently modify their code by modularising crosscutting concerns. AspectJ applies declared aspects by modifying the bytecode.

Component-orientation and aspect-orientation are orthogonal. Aspect-oriented programming adds most value when there are standard components to be modified. For example, the ability to add pre and post actions and conditions to out of the box components is extremely useful functionality.

A useful addition to providing Component-Oriented features in the next phase of Java would be to add Aspect-Oriented features.

8. Conclusion

Java is entering into its inefficient phase. C# is in the middle of its acceptance phase.

When the .NET architect, Hejlsberg, says, "Yeah, and I just think that we're a generation ahead when it comes to the thinking in this space," he is probably right. In the scheme of things the .NET language extensions are particularly small. However .NET now offers Component-Oriented support in the form of properties and events as first class language constructs.

There is a lot of talk amongst senior figures about where Java is heading. JavaLobby, The Server Side and Java Developer Journal have all carried articles in recent months about the way ahead for Java. Looking at the readers' responses to these articles I'd say the vast majority of Java developers don't even appreciate what Hejlsberg is saying.

Many Java programmers can't see that C# has leapfrogged Java. Whilst many of its Component-Oriented aspects are supported by Java, they are not built in, and they involve significant programmer effort to get to work. Thankfully the next version of Java (1.5) is planned to have boxed types built in, which shows they are at least responding to .NET.

When languages evolve, they move closer to problems rather than the solutions. Javasoft should stop thinking about solutions and start thinking about our problems. The publication of module interfaces or displaying a type and its related dependents in a UI should be one line of code.

It's time for Javasoft to start getting aggressive in their vision of what Java is. It's easy to get self-congratulatory and lazy when you're in a dominant market position. I believe that Javasoft can deliver the next big thing but it's going to involve swallowing a lot of pride, and I'm not sure they can do that.

9. References

Brown W, et.al, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, 2001.

Javelin Software: JGenerator, November 2001 (http://www.javelinsoft.com/)

Javelin Software: Swinglets, May 1999 (http://www.swinglets.com/)

Jonathan Aldrich Craig Chambers David Notkin, Component-Oriented Programming in ArchJava
http://www.cs.washington.edu/homes/jonal/archjava/oopsla01-cop.pdf

Microsoft Corporation: The Component Object Model 0.9. July 1995.

P. H. Frohlich, Component-Oriented Languages: Messages vs. Methods, Modules vs. Types. (http://nil.ics.uci.edu/~phf/pub/honnef-2000.pdf)

Sun Microsystems : The Java Beans Specification 1.01. July 1997.
(http://java.sun.com/products/javabeans/)

Sun Microsystems: JavaTM Architecture for XML Binding (JAXB), May 2001.
(http://www.javasoft.com)

Sun Microsystems: JavaDataObjects, version 1.0, 2000.
(http://jcp.org/aboutJava/communityprocess/first/jsr012/index.html)

Sun Microsystems : JavaTM Architecture for XML Binding (JAXB), May 2001.
(http://www.javasoft.com)

W3C. Simple Object Access Protocol (SOAP) 1.1, May 2000
(http://www.w3.org/TR/SOAP).

W3C, XML Schema Part 2: Datatypes XML, May 2001
(http://www.w3.org/TR/xmlschema-2)

Xerox. AspectJ™: Aspect-Oriented Programming Using Java Technology.
http://aspectj.org.

JavaWorld: Aspect-Oriented Programming.
http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html
http://www.javaworld.com/javaworld/jw-03-2002/jw-0301-aspect2.html
http://www.javaworld.com/javaworld/jw-04-2002/jw-0412-aspect3.html