Why Java is Not Suitable for Object-Oriented Frameworks

Dragos A. Manolescu
Applied Reasoning Systems Corporation and the University of Kansas
10955 Lowell Ave. Suite 300
Overland Park, KS, 66210
+1 (913) 319 0900
dragos.Manolescu@acm.org

Adrian E. Kunzle
Skillgames
233 Broadway, 19th floor
New York, NY 10279
+1 (212) 471 3514
adrian.kunzle@skillgames.com

ABSTRACT

Many business applications involve Java and object-oriented frameworks. Several characteristics of Java conflict with some key features of frameworks. These conflicts force the creation of "work-arounds" by developers. We show several examples that illustrate the tensions that exist between Java and object-oriented frameworks, and discuss how we solved them.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *classes and objects, constraints, frameworks, inheritance.*

General Terms

Languages.

Keywords

Java, Object-Oriented Frameworks.

1. JAVA DOES NOT SUPPORT COVARIANT RETURN TYPES

The Java type system doesn't support covariant return types (i.e., subclasses can't change the return type of an inherited method to a subtype). The lack of covariant return types has a significant impact on how developers use Java. This ranges from simple messages like accessors, through idioms like polymorphic copy, through design patterns like Factory Method, Manager or Singleton [1]. In effect, Java's type system introduces roadblocks that developers must code around. Sometimes they may get around with a type cast. Other times, to avoid downcasting, they may have to widen the API with methods that simply narrow the return type (e.g., getMySessionContext vs. getSessionContext). This makes applying the white-box and black-box reuse techniques specific to object-oriented frameworks cumbersome [2]. It also breaks layering, making developers aware of the objects at different levels of abstraction.

2. JAVA'S STATIC TYPE CHECKING PROVIDES A FALSE SAFETY NET

Java is intended for building robust, reliable, and secure software. One of the mechanisms used to achieve these goals is static type checking. We have studied how well this works on a large-scale.[1] eBusiness project involving several object-oriented frameworks [3]. Java's static type checking catches mainly trivial mistakes that seasoned developers and unit tests would catch anyway, without guaranteeing the elimination of run time problems. Without covariant return types, many Java frameworks involve explicit casts from a framework generic type to an application-specific type (customization through class composition is typical in white-box frameworks [4]). The compiler doesn't check casts. At run time the Java virtual machine reports casting problems by throwing a ClassCastException and usually terminating the application. Catching these exceptions involves a significant amount of manual type checking. This cycle greatly reduces the value of simple compile time type checking.

3. REMOTE EXCEPTION = COUPLING

Well-crafted distributed systems exhibit low coupling between the subsystem and object design and the deployment packaging structure. Once developers start to learn how an application performs in a distributed environment, they fine-tune it through re-partitioning the functionality between the server-space and the client-space. This requires the ability to relocate components around the client-server boundary seamlessly. Java supports distributed programming natively through RMI. Java's RMI RemoteException subclasses Exception and requires explicit catch statements. This violation of the Liskov Substitution Principle[2] essentially introduces coupling between domain objects and their location. Refactoring components from server-space to client-space involves wrapping remote message calls in try-catch blocks, which translates into hard-coding the location within the code. In effect, it hinders developers' ability to experiment with "concretizing" object locations when fine-tuning distributed applications [5]. It also litters the code and makes maintenance harder.

[1] The project involves over 700 Java classes and almost 9,000 methods.

[2] The Liskov Substitution Principle (LSP) states that modules using references to base types must be able to use references to derived types without knowing the difference.

4. ADDITIONAL PROBLEMS

4.1 Visible source code

Java lets developers separate the source from the byte codes. Although the core JDK classes ship with source code, developers don't have to do so. Typically third party frameworks and libraries ship as class files without source. But not having framework source code hampers your project. You can't see how the framework works; you can't add hooks that the original developers never thought would be needed; you'll have a hard time finding bugs. In short, you have to hope that the framework developers thought of all the things that you might possibly want to do with it, or else you are groveling at their door. The increasing popularity of the Open Source model provides a clear sign that developers benefit from having access to the source code. Note that distributing source code doesn't hinder the business aspect; Smalltalkers have made money for over 20 years from applications shipped with source code.

4.2 Deprecated is a comment instead of a reserved word

As frameworks evolve, some of their components become obsolete. Java lets developers deprecate methods, classes, or interfaces. This allows for the phasing out of functionality in a controlled way, giving framework users adequate time to adapt their code to the new mechanisms. However, Java hides this important tag in a comment, thereby reducing its visibility. When looking at a class to work out how to use it, most developers look at the class and method definitions first. It would make more sense for "deprecated" to be a keyword and to live in the definition (e.g., `public static deprecated void MyClass()`) rather than in the comment. This would make compiler parsing easier, and would increase the visibility of this useful function.

5. SUMMARY

We have sketched the impedance mismatch between object-oriented frameworks and the Java programming language. More specifically, we have described some of the problems we encountered on a large Java project involving several frameworks. Software developers walking this path will benefit from our discussion. They will understand the tension between object-oriented frameworks and Java, and will learn how we dealt with it.

6. REFERENCES

[1] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns. Addison-Wesley, Reading, MA 1994.

[2] Johnson, R., and Foote, B. Designing Reusable Classes. Journal of Object-Oriented Programming, June/July 1988, Volume 1, Number 2, pages 22-35.

[3] Manolescu, D., and Kunzle, A. Several Patterns for eBusiness Applications. Proceedings of Pattern Languages of Programs 2001, Monticello, IL, USA. Available on the Web from `http://micro-workflow.com`.

[4] Roberts, D., and Johnson, R. Patterns for Evolving Frameworks. In Martin, R., Riehle, D., and Buschmann, F., editors. Pattern Languages of Program Design, Volume 3, Addison-Wesley, Reading, MA 1997.

[5] Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. A Note on Distributed Computing. Sun Microsystems Laboratories, Inc. Technical Report SMLI TR-94-29, November 1994.