Why Java is Not Suitable for Object-Oriented Frameworks

Dragos A. Manolescu
dragos.manolescu@acm.org

Adrian E. Kunzle
adrian@kunzle.com

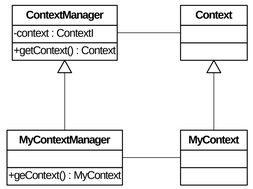
Background

We have worked on a large-scale (i.e., over 700 classes and almost 9,000 methods) eBusiness Java project. We've observed tensions between generic and domain-specific object-oriented frameworks, and the Java programming language. Here are some of the highlights of our experience with using and refactoring rich Java frameworks.

Java doesn't Support Covariant Return Types

Problem

Framework developers tailor white-box frameworks through class composition. Typically customization involves objects that receive messages and objects returned by messages. The former requires subclass polymorphism, and the latter requires covariant return types. Java subclasses can't change the return type of an inherited method to a subtype. You can customize framework objects through inheritance. However, messages sent to these objects will return generic framework objects instead of application objects.



Why does this matter?

The lack of covariant return types affects:

- Simple messages like *accessors*
- Idioms like *polymorphic copy*
- Design patterns like *Factory Method*, *Manager*, *Singleton*, etc.

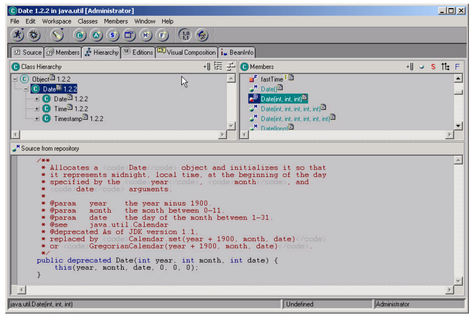
Question

How can you customize a Java framework through inheritance?

Deprecation through Comments is Easy to Miss

Problem

As frameworks evolve, new components replace old ones. To transition smoothly, framework developers should be able to phase out the old components in a controlled way, giving framework users adequate time to update application code. Java lets developers deprecate methods, classes, or interfaces. However, Java hides this important tag in a comment, thereby reducing its visibility. When browsing code to work out how to use it, most developers look at the class and method definitions first.



Why does this matter?

Developers have to read the tags embedded within comments as well as the source code. They could easily miss methods, classes, or interfaces marked as `deprecated`.

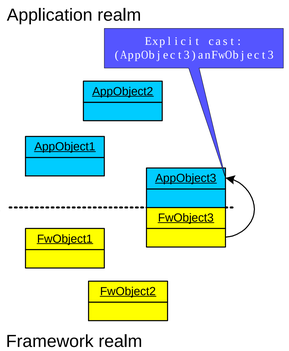
Question

How can you mark deprecated methods, classes, and interfaces in a way that is not easy to miss?

Java's Static Type Checking Provides a False Safety Net

Problem

Java is intended for building robust, reliable, and secure software. One of the mechanisms used to achieve these goals is static type checking. However, without covariant return types using Java frameworks involves explicit type casts from a framework generic type to an application-specific type. For example, even the containers from the Java class library require developers to cast down from Object.



Why does this matter?

Static type checking catches many of the trivial mistakes that seasoned developers and unit tests would discover anyway. However, it can't guarantee that a cast will succeed. Application developers may think that they have fixed all type errors once the type checker signals it. At run time the virtual machine will throw a `ClassCastException` for any explicit cast that fails. If uncaught, this run time type error will terminate the application.

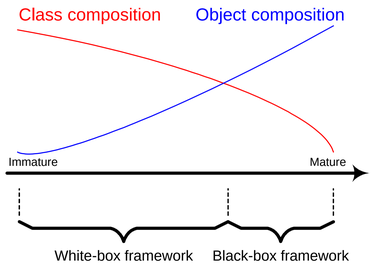
Question

How can you ensure that crossing from the framework realm to the application realm won't produce a `ClassCastException`?

Class Composition Requires Visible Source Code

Problem

Developers use and customize object-oriented frameworks through a combination of class and object composition. Typically frameworks start as white-box, with inheritance as the mechanism for using them. As they mature, they become black-box, and object composition replaces (most of the) class composition. Access to the source code is crucial, particularly at the beginning of frameworks evolution. 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 with no source.



Why does this matter?

Without source code:

- You can't have white-box frameworks; you'll have immature black-box frameworks
- You can't see how the framework really works
- You can't extend the framework beyond what its developers imagined
- You'll have a hard time fixing bugs

Question

How can you ensure that your users have access to the framework's code?

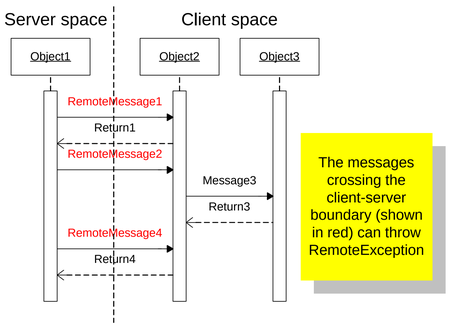
`RemoteException` = Coupling

Problem

Well-crafted distributed system exhibit low coupling between the subsystem and object design, and the deployment packaging structure. Developers fine-tune the system by repartitioning the functionality between the server-space and the client-space. Java provides native support for distributed programming through RMI. However, the explicit catch blocks required by `RemoteException` effectively hardcodes an object's location within the code.

Why does this matter?

The above problem introduces coupling between domain objects and their location. Refactoring components from server-space to client-space involves wrapping remote message sends in try-catch blocks. This hinders the developers' ability to experiment with concretizing object locations when fine-tuning distributed applications.



Question

How can you reduce the coupling between your objects and the deployment location when using RMI?

Static Type Checking Dependencies Prevent Independent Development

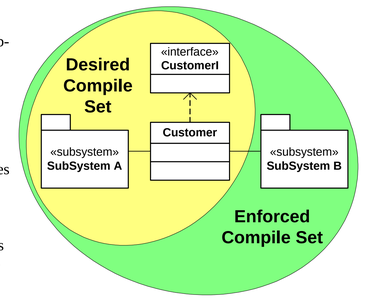
Problem

You have a system that is broadly split into two parts, A and B, who share one common object—let's call it `Customer`. `Customer` is a collection of discrete business objects, some from A and some from B. These objects' interfaces project as methods on `Customer`. The groups working on A and B want to work independently, and it is meaningful at a systems level to do so (i.e., A can perform a lot of things using `Customer` without B's code being there). However, if the group working on A wants to compile their code—including `Customer`—the compiler also needs access to most of B code.

Why does this matter?

This type of compile dependency causes the following problems:

- Sharing code can be troublesome for groups working in different physical locations
- Compile issues in one group can often leave the other group unable to test their work, even in the presence of good version control
- You pollute your system through building lots of interfaces to stop compile dependencies



Question

How can you prevent the static type checking dependencies from interfering with your breaking the system in independent parts?

About Us

Dragos A. Manolescu has earned a Ph.D. in Computer Science from the University of Illinois and now teaches at the University of Kansas. He is also consulting in object-oriented workflow in particular, and object technology, software architecture, frameworks and patterns in general. Visit Dragos on the Web at `http://micro-workflow.com/`.

Adrian E. Kunzle has a 9 year history of building and architecting large distributed object systems for both financial institutions and eBusiness companies. His language of choice is Smalltalk, but earns a living writing Java. He is constantly fascinated by the human side of systems development. Adrian is currently the CTO of a new B2B startup focusing on Internet conversion marketing.