Some Preliminary Rules of Engagement for Java

Joseph E. Hollingsworth
Computer Science Department
Indiana University Southeast
4201 Grant Line Road
New Albany, IN 47150-2158 USA

jholly@ius.edu
Phone: +1 812 941 2425
Fax: +1 812 941 2637
URL: http://homepages.ius.edu/JHOLLY

Bruce W. Weide
Department of Computer Science and Engineering
The Ohio State University
2015 Neil Avenue
Columbus, OH 43210-1277

weide.1@osu.edu
Phone: +1 614 292 1517
Fax: +1 614 292 2911
URL: http://www.cse.ohio-state.edu/~weide/

Abstract

We propose some "rules of engagement" for developing software in Java in order to achieve the following goals: simplified specification and reasoning when compared to alternatives such as JML, efficiency comparable to typical Java software, and the ability to make use of existing Java components (e.g., Swing). It has been said that the best place to start is at the beginning, and [Weide02, Harms91] tell us that the beginning has to be with the movement of data. If we do not get that aspect of software development right to begin with, then we are already fighting a loosing battle with respect to achieving the stated goals. Next comes the development of what we call "clean and safety net" foundational components which aid in building of larger scale applications [Hollingsworth00] that meet our goals. Finally, we need some additional rules that must be followed in order to achieve the stated goals.

Keywords

data movement, generics, move, transfer, program reasoning, clean and safety net components

Paper Category: technical paper
Emphasis: research

1. Introduction

Since the early 1990's the Reusable Software Research Group (RSRG) has advocated swapping [Harms91] as the primary means by which data should be moved within a sequential, imperative program. Only one research language (Resolve [Ogden94] created by RSRG) and no commercial languages (that we know of) support swapping as the built-in, primary method for moving data within a program. RSRG has proposed disciplines for Resolve/Ada [Hollingsworth92-2] and Resolve/C++ [Hollingsworth94] that permit building software in those languages using the swapping paradigm.

The contribution of this paper is that it proposes a data movement paradigm for Java, using existing built-in Java constructs, along with a preliminary set of "safety net" foundational components, and some preliminary "rules for engagement" for developing Java programs, all intended to reduce the cognitive load with respect to reasoning about Java programs and object references inherent in them, and still be efficient.

Section 4.1 discusses why retrofitting swapping into Java (as was done in Resolve/Ada, and Resolve/C++) cannot be done easily, and our choice as the next best alternative appears in Section 4.2. Section 4.3 mentions (without detail) some of the clean and safety net components, while Section 4.4 introduces some of the preliminary rules for engagement for using Java in a disciplined manner.

2. The Problem

How to design and implement software written in Java to improve ease of reasoning and specification, efficiency, and the ability to make use of existing Java components if so desired. Note: "ease" is almost always directly dependent on the difficulty associated with computing the solution to the problem, therefore "ease" is a relative term. That is, we are not claiming everything is going to be "easy".

3. The Position

By building on previous RSRG work, we have developed a prototype approach that addresses the above problem with a system of rules that even hard core Java hackers can embrace.

4. Justification

4.1 In Java, "Move" or "Transfer" Data, but Do Not "Swap" it

[Weide02] explores the various ways in which data can be moved within a program, i.e., to address the *data movement problem*. That paper also proposes some *data movement evaluation criteria* for evaluating these different methods for moving data. We are not going to rehash that work here, rather, we plan to use its results to pick the best data movement paradigm for Java programs. In this section, we first discuss why we are not picking the swapping data movement paradigm for Java, which by the data movement evaluation criteria turns out to be an excellent choice for many commercially available, sequential, imperative languages. Next we propose that data be "moved" or "transferred" in Java. This *move* approach is not new - it is described in [Weide02], and by the evaluation criteria it is rated the next best choice for moving data within a program.

4.2 In Java, We Are *Not* Going to Swap

Anyone who has been even peripherally acquainted with RSRG's research knows that swapping of data has been (and continues to be) at the core of the Resolve software discipline and approach. That we are *not* choosing swapping for Java must come as a great surprise. But do not be fooled. The heart of the problem within sequential, imperative programs is how best to move data. The answer is swapping in the Resolve research language and for many commercial languages (except Java). We are not married to swapping. We are dedicated to solving the data movement problem, and if that means choosing a method other than swapping, then so be it.

Because of the way in which Java is defined, swapping does not turn out to be the most effective method for moving data. The roots of this data movement problem in Java lie with the fact that all non-scalar type objects are implemented by using a reference variable to store the address to the object data (as was done in Modula), but no direct access is given to where this reference is stored. By direct access, we mean, a Java software developer cannot gain access to the reference variable in order to change the existing reference stored there so that the variable can reference a different object. If one *could* access where the references are stored in Java reference variables, then one should be easily able to implement the swapping paradigm.

A folk theorem of computing is that every problem can be solved by adding a level of indirection [Weide01]. So why not stick with swapping and implement all objects with a second level of indirection as is mentioned in [Sridhar02]? That is, when creating a new object class C, have the only data member inside C be a reference to the actual data that needs to be stored by object instances of C. Then to swap the data between two object instances of C, say A and B, one would just call an operation exported from C which would have access to A's and B's data members (both of which are references to the real data) and and just swap them. The original Java variable references for A and B would remain the same. (As a side note, this double level of indirection is often referred to as a *handle*, and is used by some operating systems to permit the operating system to do memory compaction on the fly while the program remains in an execution state.) Why not choose this approach? Every dereference of an object now requires two dereferences, and as was seen in [Hollingsworth00] actual commercial systems can be highly layered. Thus when a dereference is made at a high level this dereference might cause a cascading of dereferences down through the layers to the bottom layer. Performing two dereferences for each access of an object at each level of a highly layered system most certainly will cause a performance hit. We have not measured this effect because there is a viable alternative to swapping; we leave this for future work.

Ruling out two levels of indirection leaves us with trying to implement swapping in Java by using the built in language constructs, e.g., assignment, parameter passing, returning of values, etc.

- Call-by-reference - *If* Java supported call-by-reference, then every class C could export an operation which would take two objects from that class and do a straightforward three line swap using a locally declared temporary variable. The call to such an operation might look like: `x.swap(y);` After the call, x would reference y's data, and vice versa.
- Return multiple values - *If* Java supported the returning of more than one value, then one might construct an operation which would take two objects and return those two objects in "reverse" order. The call to such an operation might look like: `y, x = swap(x, y);`
- Preprocessor magic - *If* Java supported a preprocessor then one might be able to use *preprocessor magic* (as it is known in the C/C++ world) to implement swapping.

Our proposal is to implement data movement by using Java's assignment statement in combination with the recently introduced generic class and interface capability. We found in [Hollingsworth00] that dealing with a piece of data gives rise to one of two situations: 1) we either momentarily need to hand off an object to some other operation (which implements some algorithm that uses the data, and possibly changes it); or 2) we need to put it in some container for safe keeping for later use. For both situations, Java is well suited with its call-by-value for small and large objects. Why? Because at most, all that is required for passing a scalar is the (small) value itself, and all that is required for passing large objects is the (small) variable reference.

But what about ease of specification and reasoning? The problem is visible aliasing, i.e., aliased references to mutable objects, so that changes to an object through one reference are visible through another reference that is not mentioned in the statement that changes the object. We want to drive the number of references to each object to the bare minimum, i.e., one. Think of this as a program wide invariant, similar to a loop invariant, where there might be short times when we need to have multiple visible aliases, but after that need is satisfied, we go back to our invariant of just one reference (just like reestablishing the loop invariant). That is, if in a particular part of a Java program, we have a Java variable reference to an object (and its value), and we know (because of our program-wide invariant) that no other part of the program aliases this object, then we can reason with certainty and confidence about the before and after values of the object when working with it. We can do this without introducing the complications of modeling or reasoning about references [Weide01].

When handing off an object - In the situation where our sequential program needs to hand off the object to another operation, there will be two references in existence momentarily, one held by the caller and one held by the callee. For now, we are not going to allow the callee to make and save a copy of the reference, so that when the callee finishes, its reference automatically gets eliminated. Later we will discuss what has to be done if the callee needs to save a copy of the reference.

When saving an object for safe keeping - When inserting an object into a container object for safe keeping and for later use, the container gets and stores the reference, and simultaneously we eliminate the reference stored in the caller by replacing it with 'null'. Below (in Figure 1) is some sample code using a Queue for a container:

Client of Queue

```
import queue.*;

void Op1 ()
{
   SomeMutableOrImmutableObject x = new SomeMutableOrImmutableObject()
   IQueue<SomeMutableOrImmutableObject> q1 = new Queue1<SomeMutableOrImmutableObject>();

   x.modify(...);
   x = q1.enqueue(x); // returns null
   ...
}
```

IQueue.java

```
package queue;

public interface IQueue<E> extends Cloneable {
   public void clear();
   public IQueue<E> clone();
   public E dequeue();
   public E enqueue(E value);
   public boolean equals (Object other);
   public E peek();
   public int size();
   public String toString();
}
```

Figure 1.

All container components are implemented so that:

- when the reference goes into the container, 'null' gets returned and the calling operation assigns this result to the reference variable that was passed to the container's insert operation;
- when the reference comes out of the container, no reference to it is maintained within the container, and the calling operation assigns the returned reference to its own local reference variable

4.3 A Set of Clean, and Safety Net Components

This part of the effort involves encapsulating, into a family of well designed, understandable, and carefully implemented software components, support for clean components that encapsulate the most important uses of indirection with simple contract specifications; and safety-net components to handle all residual uses of indirection not otherwise covered. There is not enough room to describe them here. A demonstration will be provided at the workshop, however. Details basically follow [Hollingsworth92-1, Hollingsworth92-2].

4.4 Rules for Engagement

In the past it has been called a "discipline". It's time for a new name, and that name is "rules for engagement". Think of it as how we are going to engage the language (Java in this case) in a way that helps us achieve our goals. The incomplete list of rules that follow are not in any particular order of importance. One must remember that it is allowable to break any of these rules for engagement; however, one needs a very good reason to do so, and also needs to be prepared to pay the consequences.

- Use value-based modeling and specification.
- Make 'null' a member of the value space for all object models. That way when an object is assigned 'null' it has a legal value for its type.
- Use the "generic" construct so that the compiler can aid in helping identify type mismatches in client code at compile time.
- Use the "interface" construct to capture the abstract idea of a component, use one or more "class" constructs to implement the interface.
- Build all container components by layering on the clean and safety-net components mentioned in Section 4.3.
- Implement 'clone' so that it makes a true deep copy.
- Implement 'toString' so that it produces a string representing the abstract value of the object.
- Implement a 'clear' operation to reset the object's abstract state to be exactly as if the object had just been created.
- Implement a 'terminate' operation to be called when the object is no longer of any use. This is not the same as 'clear', because after 'clear' is called the object can continue to be used.
- At declaration time, assign a value to an object reference. One might use 'new' and call the object's constructor, or one might 'clone' a value, or one might set the reference to 'null'. There are many options, but one way or the other, the object reference must get assigned a value at declaration time.
- Move data around as is described in Section 4.2.
- At all times drive the number of references to bare minimum by moving/transferring references rather than copying them.
- If you make use of components that do not follow these rules of engagement, be aware that operations such as 'clone', 'clear', etc., will probably not work as advertised.

5. Related Work

We have made reference to other work throughout the paper.

6. Conclusion

As was mentioned in Section 3 (The Position), we have developed a prototype approach that simplifies reasoning and specification about Java programs without sacrificing efficiency or existing libraries. We can think of no better way in which to make it stronger (or discredit it) than to hold it up for scrutiny by the attendees of the workshop. We will come prepared to show the clean and safety-net components, containers layered upon these components, and even a small GUI application developed using these components.

Acknowledgments

This work is supported in part by Indiana University by providing sabbatical release time for this research.

References

[Harms91]

Harms, D.E., and Weide, B.W., "Copying and Swapping: Influences on the Design of Reusable Software Components", *IEEE Transactions on Software Engineering 17*, 5 (1991), 424-435.

[Hollingsworth92-1]
Hollingsworth, J.E., and Weide, B.W., "Engineering 'Unbounded' Reusable Ada Generics," *Proceedings 10th Annual National Conference on Ada Technology*, Arlington, VA, February 1992, 82-97. [PDF]

[Hollingsworth92-2]
Hollingsworth, J.E., *Software Component Design-for-Reuse: A Language Independent Discipline Applied to Ada*. Ph.D. thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, 1992. (If you obtain this document electronically, please read the file `README.txt` in the document's directory, then take all the files.) [PDF: 12 files]

[Hollingsworth94]
Hollingsworth, J.E., Sreerama, S., Weide, B.W., and Zhupanov, S., "RESOLVE Components in Ada and C++," *Software Engineering Notes 19*, 4 (October 1994), 53-63. [PDF]

[Hollingsworth00]
Hollingsworth, J.E., Blankenship, L., and Weide, B.W., "Experience Report: Using RESOLVE/C++ for Commercial Software", *Proceedings of the ACM SIGSOFT Eighth International Symposium on the Foundations of Software Engineering*, 2000, ACM Press, 11-19.

[Ogden94]
Ogden, W.F., Sitaraman, M., Weide, B.W., and Zweben, S.H., "The RESOLVE Framework and Discipline- A Research Synopsis," *Software Engineering Notes 19*, 4 (October 1994), 23-28. [PDF]

[Sridhar02]
Sridhar, N., Weide, B.W., and Bucci, P., "Service Facilities: Extending Abstract Factories to Decouple Advanced Dependencies", in C. Gacek, ed., Software Reuse: Methods, Techniques, and Tools (Proceedings Seventh International Conference on Software Reuse), Springer-Verlag LNCS 2319, 2002, pp. 309-326. [PDF]

[Weide01]
Weide, B.W., and Heym, W.D., "Specification and Verification with References", Proceedings OOPSLA Workshop on Specification and Verification of Component-Based Systems, October 2001, http://www.cs.iastate.edu/~leavens/SAVCBS/papers-2001.

[Weide02]
Weide, B.W., Pike, S.M, Heym, W.D. "Why Swapping?", *Proceedings of the RESOLVE Workshop 2002*, Columbus, OH, June 2002, Technical Report TR-02-11, Dept. of Computer Science, Virginia Tech, Blacksburg, VA, June 2002, 72-78.