SmallJava

Smalltalk to Java

Using Language Transformation
to Show Language Differences

v0.1 [mlf-970507]

Mark L. Fussell

Mark.Fussell@ChiMu.com
www.chimu.com

ChiMu Corporation
1220 N. Fair Oaks Ave, #1314
Sunnyvale, CA 94089
Phone: 408 734-9068
Email: info@chimu.com
www.chimu.com

Table of Contents

Copyright © 1997, Mark L. Fussell

Overview and Introduction

In this document I will compare and contrast Smalltalk and Java using a somewhat unusual approach: I will transform one language (Smalltalk) into the other (Java) by a series of small steps in an intermediary language (SmallJava). With each step I will discuss how the new properties of the intermediate language compare and contrast to the properties of the previous version of the language.

The reason to use this approach is it allows two languages' features to be compared without the baggage of all the other differences in the languages. Frequently language discussions are difficult because there are so many differences in syntax and unrelated semantics between the two languages that discussing a single point is impossible unless all parties are fluent in both languages. The approach of transforming a single language removes this particular hurdle to analysis. In addition, by using two publicly known languages as "end-points", the intermediate language will also be very familiar.

The reason I can use this approach is the incredible amount of similarity between Smalltalk and Java. This similarity is surprising because the syntax of the languages look very different and, more importantly, because Java is called "statically typed" and Smalltalk is called "untyped" or "dynamically typed". But the similarity is there at the core and by doing the language transformation we can have very concrete examples of what terms like "statically typed" and "dynamically type" mean in the context of a "single" language.

This document will require at least introductory background in both Java and Smalltalk. Basically the syntax and major concepts of both languages should be familiar to you. If you get lost during the postings I suggest you look at FAQ's and book related postings for "comp.lang.smalltalk", "comp.lang.java.programmer" and "comp.object" [using www.dejanews.com might be the easiest approach]. Although I don't ever consider a single book to be enough for any language, my current favorite two books may be:

Smalltalk-80: The Language and its Implementation.
Adele Goldberg and David Robson.
Addison-Wesley, Reading, MA, 1983.

The Java Language Specification.
James Gosling, Bill Joy, Guy Steele.
Addison-Wesley, Reading, MA, 1996.

But these are not exactly introductory books and will not be the best choices for all readers.

Copyright © 1997, Mark L. Fussell

SmallJava

SmallJava-97, The Language

This document is comparing Smalltalk and Java through an intermediary language, SmallJava. SmallJava will start as the common elements from both Smalltalk and Java. It will then be used to discuss how new properties brought into SmallJava from Smalltalk or Java impact the language. The first item on the agenda is to specify what the base version of SmallJava will look like in both semantics and syntax.

SmallJava Semantics

SmallJava semantics will be the intersection of the common semantic features of both Smalltalk and Java. This will enable SmallJava to easily grow with features in either direction and will allow developers familiar with either language to also be familiar with SmallJava. To accomplish this goal requires simplifying the properties of both languages until the remaining cores overlap sufficiently. Because of the similarities in the languages not very much has to be removed from either.

Smalltalk and Java Simplification

First I will simplify Smalltalk to get to its core. To do this SmallJava will:

(S1) Use class declaration instead of class construction
(S2) Drop all the Metaclass/Class capabilities
(S3) Ignore how blocks and control messages are different from control structures.
(S4) Only have a single root class "Object" that all classes must inherit from directly or indirectly.

This gives us a dialect of Smalltalk like “Little Smalltalk” [Budd 87]. We have dropped (or not mentioned) a lot of capabilities of professional Smalltalk dialects, but we still have a valid dialect of the Smalltalk language.

Next I will simplify Java to match up with the above simple Smalltalk. To do this SmallJava will:

(J1) Drop access control. All methods will be public and all instance variables will be protected which closely matches Smalltalk’s standard access controls.
(J2) Ignore the static side of Java classes.
(J3) Consider all primitive data types to inherit from Object. Viewed differently, you could say that all the Java control structures, operators, and other expressions work with the Wrapper versions of the primitive data types.
(J4) Make allowances and simplifications with certain packages and operators. For example the “Math” functions will be made available without needing the “Math” prefix.
(J5) Drop the use of static typing. All variables, parameters, and return types will be typed as “Object” and messages can be successfully sent to an object that has a method with the right signature. Since static typing is removed, interfaces are removed also and we simply have classes extending other classes.

The only dramatic change to Java is by J5. An interesting aspect of J5 is that if you take a working program it still works after applying J5 to it. This will be shown in the example below.
The change J3 is also sizable but I believe it is an “unquestionable” improvement so less interesting although this will also be discussed in later sections.

SmallJava Semantics Summary

Stated again, SmallJava’s semantics are the intersection Smalltalk and Java with each simplified as specified. A more formal description might be useful but is not in the scope of this document. It would also have to be many formal descriptions because SmallJava will be constantly changing as we add new

Copyright © 1997, Mark L. Fussell

properties to it. Beside the above informal description, examples of SmallJava will be provided as the document progresses. Next we have to specify SmallJava's syntax.

SmallJava Syntax

SmallJava uses primarily Java syntax. This is to allow easier discussion of adding in Java properties (e.g. static typing and interfaces) since I can use Java syntax for those properties. It is also true that many programmers are more familiar with 'C++' like syntax than with Smalltalk syntax. Adding in Smalltalk properties to SmallJava generally does not require extra syntax (blocks "[|]" being the notable exception) but instead either requires objects to become "smarter" (control messages for Booleans and Collections) or requires developing new types of objects (Class objects).

If you are interested in a syntax level comparison between Smalltalk and Java, you might want to look at: http://www.chimu.com/publications/SmalltalkJavaSyntax.html. Also some of my naming conventions make Java look a bit more like Smalltalk so the syntax difference will not be quite as noticeable.

The SmallJava Language

This resulting SmallJava is a language you can view as either a simple Java without static typing or as a simple Smalltalk with Java syntax. I will show each of these views in different ways. First I will show the similarity to Java by applying the "J" transformations to a Java program and turn it into SmallJava. Second I will translate the example's syntax between SmallJava and Smalltalk. This will result in a total of three versions of the same example code in each of Smalltalk, Java, and SmallJava.

From Java to SmallJava

My primary example will be a simple Point class. The following is the original, 100% pure Java.

```
public class Point {
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double x() {return x;}
    public double y() {return y;}

    public double r() {return Math.sqrt(Math.pow(x,2)+Math.pow(y,2));}
    public double theta() {return Math.atan2(y,x);}

    public Point vectorFrom(Point point) {
            return new Point(x - point.x(), y - point.y());
    }

    protected double x,y;
}
```

Now I will start applying the SmallJava changes. J2 doesn't apply because there are no static methods. After applying J1, J3, and J4 to the example we get:

```
class Point {
    Point(Double x, Double y) {
        this.x = x;
        this.y = y;
    }

    Double x() {return x;}
```


Copyright © 1997, Mark L. Fussell

```
    Double y() {return y;}

    Double r() {return sqrt(x^2+y^2);}
    Double theta() {return atan2(y,x);}

    Point vectorFrom(Point point) {
            return new Point(x - point.x(), y - point.y());
    }

    Double x,y;
}
```

The major dramatic effect of these changes is that all our numbers can now be used like all other objects (e.g. directly placed in Vectors and Hashtables). Except for these numbers and the simplified Math calls, the above is completely legal Java.

Next we apply J5 part 1: make all the variables typed to Object.

```
class Point {
    Point(Object x, Object y) {
        this.x = x;
        this.y = y;
    }

    Object x() {return x;}
    Object y() {return y;}

    Object r() {return sqrt(x^2+y^2);}
    Object theta() {return atan2(y,x);}

    Object vectorFrom(Object point) {
            return new Point(x - point.x(), y - point.y());
    }

    Object x,y;
}
```

Strangely enough, this is almost legal Java also. Basically it is missing a lot of casting to make it work. For example the method #vectorFrom needs to have casts like this to be valid Java:

```
Object vectorFrom(Object point) {
            return new Point(
                (Double) x - (Double) ((Point) point).x(),
                (Double) y - (Double) ((Point) point).y()
              );
}
```

Well, that is a whole lot of casting, so it is fortunate we are dropping it. We will allow a message to be sent to any Object and if it the actual object does not implement that method we will throw a "DoesNotUnderstandException" (this behavior will be discussed in depth in the next section).

Now we can finish the J5 transformation and drop all the superfluous "Object"s. This gives us:

```
class Point {
    Point(x, y) {
        this.x = x;
        this.y = y;
    }
```


Copyright © 1997, Mark L. Fussell

```
    x() {return x;}
    y() {return y;}

    r() {return sqrt(x^2+y^2);}
    theta() {return atan2(y,x);}

    vectorFrom(point) {
            return new Point(x - point.x(), y - point.y());
    }

    x,y;
}
```

The first SmallJava example

The above is the first example of SmallJava. From these transformations you can see that SmallJava is just like Java but with a lot fewer words. An important observation is, returning to an earlier comment, if the original Java Point class executed correctly then the above class would execute correctly too. The use of the static typing does not impact the correct behavior of the code at all [1]; it only helps detect incorrect specifications that will result in incorrect behavior (as does having multiple types of access control).

SmallJava to Smalltalk

Next, to show how similar SmallJava is to Smalltalk, the following is the SmallJava program converted to Smalltalk-like (see [Budd 87]) syntax (the bar "|" separates method):

```
class Point  | x y | [
    newX: newX y: newY
        x := newX.
        y := newY.
|
    x
        ^x
|
    y
        ^y
|
    r
        ^((x^2) + (y^2)) sqrt
|
    theta
        ^(y atan2: x)
|
    vectorFrom: point
        ^Point newX: x - point x
                y: y - point y
]
```

The point of defining the SmallJava language is so people familiar with either Smalltalk or Java will all have a single common language they also consider familiar. For Smalltalk developers the transition is mostly a change in syntax. For Java developers the transition is mostly relaxing over typing: the Point class will execute correctly whether the types are there or not.

Copyright © 1997, Mark L. Fussell

Summary: SmallJava as a Vehicle

SmallJava is a vehicle to compare and contrast Smalltalk and Java. SmallJava's existence is the first comparison: these languages are very similar which makes SmallJava possible. In the next sections I will start the contrasting by adding properties of Java into SmallJava. I will begin by getting "pessimistic".

---

[1] Static typing can also be used to choose among overloaded message names, but I would argue that these Type overloaded messages are a very bad thing. It should be obvious to the programmer reading or writing a message what will happen. Overloading excessively hinders this: you also have to read the Type declarations of all the message parameters and know all the possible overloaded methods to determine what the compiler determines (which better be the same thing). This is not an objection to multi-methods which are a very different thing. See the comments on method naming http://www.chimu.com/publications/javaStandards/ for a slightly longer discussion of this.

Becoming Pessimistic: Changing when an object's type is verified

SmallJava's most significant difference from Java is that a message can be sent to an object that is not known to understand that message. If the object has implemented a method matching the message signature then everything executes as expected. If the object has no matching method then SmallJava throws a "DoesNotUnderstandException".

This behavior I call "***optimistic***" messaging: you assume an object can understand a message and only handle the special cases when the object does not. This is as opposed to "***pessimistic***" messaging where you make sure an object understands a message before sending it to the object [1]. So let us compare these two approaches.

Optimistic and Pessimistic Messaging

Optimistic and pessimistic messaging have identical behavior if the message will be successfully understood. The main difference between the two approaches is when an unsuccessful message send is recognized. For optimistic messaging it will not be recognized until you send the message and for pessimistic it will be recognized at some time before sending the message. Using our Point#vectorFrom [2] example:

```
vectorFrom(point) {
        return new Point(x - point.x(), y - point.y());
}
```

If we want to make sure messages like #x and #y are understood by 'point' then for an optimistic approach we would have to do something like this:

```
vectorFrom(point) {
   try {
        return new Point(x - point.x(), y - point.y());
   } catch (DoesNotUnderstandException e) {
      //Do the right thing
   }
}
```

The approach is identical to what you would do for handling any message that could throw an Exception; the only difference is that the message is never actually "received" by the object [3].

So what would pessimistic messaging look like for our example? Assume we add to SmallJava a "message-check" with syntax "(#message)" that allows us to check whether an object understands a particular

Copyright © 1997, Mark L. Fussell

message. A message-check will do nothing if the object understands the message, but will throw a "DoesNotUnderstandException" if the object does not. Then we can change our example to:

```
vectorFrom(point) {
   try {
         return new Point(x - ((#x) point).x(), y - ((#y) point).y());
   } catch (DoesNotUnderstandException e) {
      //Do the right thing
   }
}
```

Now we have guaranteed that 'point' will successfully respond to #x before sending 'x()' and will successfully respond #y before sending 'y()'. We do not need the optimistic messaging anymore.

OK, that really didn't provide us with ANY benefit. We still throw the same exception on failure and we throw that exception "just a fraction of a second" before we would actually have sent the message. Why bother?

Advantages of pessimistic messaging

The advantage of pessimistic messaging is that we have more control of when the test is done. For example, we can make sure nothing happens in the method if we didn't actually get a 'point' that responds to #x and #y:

```
vectorFrom(point) {
   try {
      (#x,#y) point;
   } catch (DoesNotUnderstandException e) {
      //Do the right thing
   }
   return new Point(x - point.x(), y - point.y());
}
```

This is now different behavior than what our optimistic version of the method produced. Our optimistic version would have sent 'x()' to the point before checking whether the point responded to #y. This version verifies [4] that 'point' understands both #x and #y before sending any messages to it.

Forgetting and Remembering verification

SmallJava's message-check verification happens to the object in a variable at a given time. What happens when that variable changes or when the object moves to a new variable? We loose the verification and must verify again. The following is not completely pessimistic because of the second assignment to 'pointCopy':

```
vectorFrom(point) {
   try {
      temp = (#x,#y) point;
   } catch (DoesNotUnderstandException e) {
      //Do the right thing
   }
   newX = temp.x();
   temp = this;
   newX = newX-temp.x();

   return new Point(newX, y - point.y());
}
```

In the example above we do not know whether the second 'temp.x()' will be successful. We have to do another check after assigning to the variable a second time:

```
   try {
      temp = (#x,#y) this;
```


Copyright © 1997, Mark L. Fussell

```
    } catch (DoesNotUnderstandException e2) {
      //Do the right thing
    }
    newX = newX-temp.x();
```

Remembering verification

Well, that produces some very noisy methods. It would be better if we could have a little more memory of previous verifications. We can remember verification of an object by keeping track of the object during variable assignments:

```
vectorFrom(point) {
  try {
    temp1 = ((#x,#y) point);
  } catch (DoesNotUnderstandException e) {
    //Do the right thing
  }
  temp2 = temp1;
  newX = temp1.x();
  newY = temp2.y();

  return new Point(x - newX, y - newY);
}
```

Unfortunately this doesn't provide us with too much benefit in most programs.

Our other option is to insist that all assignments to a particular variable will always be message-checked before the assignment:

```
vectorFrom(point) {
  (#x,#y) temp1;
  (#x,#y) temp2;
  try {
    temp1 = (#x,#y) point;
  } catch (DoesNotUnderstandException e) {
    //Do the right thing
  }
  temp2 = temp1;
  newX = temp1.x();
  newY = temp2.y();

  return new Point(x - newX, y - newY);
}
```

All assignments to 'temp1' and 'temp2' must now check whether the value of the assignment passes the message-check. This has no impact to our assignment to 'temp1', but it does allow us to assign to 'temp2' without doing a further check. We have defined invariants for the variables [5] temp1 and temp2 that guarantee that the assignment from temp1 to temp2 will succeed. Since we can do this invariant check at compile-time we now have a simple "compile-time message-check" capability for SmallJava.

Pessimistic inside: Optimistic outside

While adding all this support for pessimistic checking, we unfortunately have been cheating a bit. We left out the implementation of "Do the right thing". What is the right thing to do? Well, it is possible that a method can handle different types of 'point's and by finding out which type of point it has it will behave differently. For example, we could use a default 'z' value if we are given a 2d point when we expect a 3d point. This is useful behavior but is not the most common behavior.

Copyright © 1997, Mark L. Fussell

The more common answer is that if a method doesn't get what it expects it doesn't know what to do. In this case we really can't catch the error at all: we have to let it go to the caller. That makes us an optimistic method from the caller's point of view. For example:

```
pointA.vectorFrom(pointB)
```

will throw a DoesNotUnderstandException if pointB is not able to respond to (#x,#y). So whether we are optimistic or pessimistic within our method, we are still optimistic as far as the caller is concerned.

Pessimistic in: Pessimistic out

How can we change this? We need more invariants within our #vectorFrom method so we don't have any message-checks inside it. Our only message-check is to 'point' itself, so if we can make the caller guarantee that 'point' passes our message-check then we will guarantee we can execute the method without further verification. Notationally this is simple enough:

```
vectorFrom((#x,#y) point) {
   return new Point(x - point.x(), y - point.y());
}
```

We are now forcing the caller to explicitly satisfy our requirement that 'point' understands (#x,#y) before they can even call our method instead of making the caller handle our DoesNotUnderstandException if 'point' does not understand (#x,#y). We just passed the verification requirement in a different and more-explicit manner than before [6].

Repassing the pessimistic requirement

SmallJava could previously handle the DoesNotUnderstandException through the usual exception handling mechanisms, so only the ultimate handler of the DoesNotUnderstandException (who ever that may be) would need to be involved [7]. Now we have to handle the pessimistic checking explicitly at all levels. If the caller to our #vectorFrom method looked like this under optimistic messaging:

```
class Line {
   Line(pointA, pointB) {
      this.pointA = pointA;
      this.pointB = pointB;
   }

   vector() {
      return vectorFrom(pointA,pointB);
   }

   pointA, pointB;
}
```

It would now have to look like this for pessimistic verification:

```
class Line {
   Line((#x,#y) pointA, (#x,#y) pointB) {
      this.pointA = pointA;
      this.pointB = pointB;
   }

   vector() {
      return vectorFrom(pointA,pointB);
   }

   (#x,#y) pointA, pointB;
}
```


Copyright © 1997, Mark L. Fussell

Where does the buck stop?

We keep passing the buck for the pessimistic message-check verification, but some SmallJava expression must be a "buck consumer" and "verification producer". We have already seen one of them: the explicit message-check when used as an expression instead of an invariant. All the pessimistic-invariants allowed us to do one thing: remember an earlier message-check. If we have a method:

```
(#x,#y) point;
try {
   point = (#x,#y) newPoint;
} catch (DoesNotUnderstandException e) {
   //Do the right thing
}
(new Line(point,point)).vector();
```

then we can take advantage of the single message-check of 'newPoint' to know all the other messages involved with creating a line and sending #vector to the line will succeed. We now have a pretty good memory caused by explicitly stating what we want to remember (require) about message-understanding throughout the flow of the program.

Having one message-check is better than many, but what if I want to get rid of that message check too? Aren't there any other "buck consumers"? There is one other case where the result of an expression is guaranteed to respond to certain messages, object construction. If we build a new Point we know what messages it responds to, the methods Point implements. The expression:

```
new Point(x,y);
```

is equivalent to:

```
(#x,#y,#r,#theta,#vectorFrom) new Point(x,y);
```

so we can finally get rid of our last message-check for our example program:

```
(#x,#y) point = new Point(x,y);
(new Line(point,point)).vector();
```

Now it is completely verifiable at compile time that all message sends will be successful (ignoring 'x' and 'y'). We have turned off the need for Optimistic messaging for SmallJava for this particular example and now have a completely pessimistic and compile-time verified program.

The Unmentionables

There were several aspects unmentioned in the above discussion of pessimistically verified message checking. What happens when we get a 'null'? What are the return values for the methods? How do we know that the method implementing a message (by name) is semantically equivalent to what we expect the method to do? These I will address in future sections on 'null' explicitly and on 'interface' instead of 'message' based typing. For the moment I will leave them unmentioned.

Where's the static typing?

We have shown how we can make SmallJava pessimistic, but how do we make it completely compile-time verified or "statically typed"? If all our pessimistic message-checks can be moved to the point of object construction then all of them can be verified at compile-time. This would be a statically message-checked program. Is this possible? Generally, no. At some point we will have to hope that a particular object understands more messages than we have been assured that it understands. For example, if we have a keyed collection object:

```
class KeyedCollection {
   atKey_put(key, value) {...}
   atKey(key) {…; return value;}
}
```


Copyright © 1997, Mark L. Fussell

Then what can we be sure about the object returned from #atKey? We can't be sure of anything. For example, in:

```
(#x,#y) inPoint = ...;
namesToPoints.atKey_put("test",inPoint);
outPoint = namesToPoints.atKey("test");
```

We can't be sure that 'outPoint' is able to respond to (#x,#y), so we will have to do a runtime message check to verify it. We can still be pessimistic by checking 'output' before sending a message to it, but we can not do it statically.

Sure we can! We can define a new class:

```
class KeyedCollectionOfPoints {
   atKey_put(key, (#x,#y) value) {...}
   (#x,#y) atKey(key) {...; return value;}
}
```

Well, that solves our problem but now we have added even more information (or "noise") to our program. We had to create a whole new class to support being able to statically verify that a "collection of points" is really a 'CollectionOfPoints'. Also note that a 'KeyedCollectionOfPoints' can not be just a "wrapper" of a 'KeyedCollection': We can not use a KeyedCollection to implement our KeyedCollectionOfPoints because we would stilly have to do a runtime check to convert the "atKey" to an "(#x,#y) atKey". We have to completely rewrite the KeyedCollectionOfPoints from scratch to have the new compile-time verifiable message-checks. So much for code reuse.

A solution to all this extra effort is to have parameterized classes that "effectively" (or actually) code-generate classes that are compile-time verifiable for a given set of message constraints. We can develop a "template" class:

```
class KeyedCollectionOf<valueMessages> {
   atKey_put(key, <valueMessages> value) {...}
   <valueMessages> atKey(key) {...; return value;}
}
```

And simply use it like so:

```
(#x,#y) inPoint = ...;
namesToPoints = new KeyedCollectionOf<(#x,#y)>();
namesToPoints.atKey_put("test",inPoint);
outPoint = namesToPoints.atKey("test");
```

So now we at least don't have to write a bunch of different classes for every variation we need, we can let the compiler do it for us. And then the compiler can statically verify the messages.

Some final remarks about static typing

One aspect to notice for the above classes is that they are not compatible. A KeyedCollectionOfPoints can not be used where you expect a KeyedCollection because the method #atKey_put(,(#x,#y)) is more restrictive than #atKey_put(,). A KeyedCollection can not be used where you expect a KeyedCollectionOfPoints because the method #atKey() is more lenient than #atKey()->(#x,#y). The classes are completely incompatible and effectively unrelated except for the design similarity.

The second remark is that Java doesn't support parameterized classes and interferes with developing your own specialized versions of classes because of weak interaction between types and polymorphism. For example, even if you have your own KeyedCollectionOfPoints, you can not develop a subclass of Enumeration that will return a (#x,#y) point for #nextElement. Java does not support covariant return types so even if you defined a 'PointEnumeration' it must either not inherit from 'Enumeration' or it has to return the same "type" as Enumeration returns, which knows nothing about (#x,#y) of point. This will be

Copyright © 1997, Mark L. Fussell

discussed again in a later section possibly titled “EiffelJava”, but for now we can say Java itself is incapable or poorly capable of making a program compile-time verifiable.

The final remark is that all this static typing ignores ‘null’ values, which would fail all the message-checks we have been applying and prevent the ability to statically type a program. To make that static typing work we will have to say what a ‘null’ means and how it interacts with the message-checking or type system. This will be discussed in the next posting.

Conclusion

Comparing Optimistic and Pessimistic messaging

What are the tradeoffs between optimistic and pessimistic messaging now that we have shown both for SmallJava?

Optimistic messaging requires much less noise to accomplish the same, if successful, result. It is also far easier to change: if we decide to send a new message to a ‘point’ we can just send the message and know (or hope) that the object will understand it. With pessimistic messaging we have to explicitly say what messages we require an object to understand. This caused us to put a lot more information into the program (much of it “obvious”) and means we have to update all this information if we decide to send a new message to an object. (I will discuss alternative ways to declaring what new messages can be understood in the section on interfaces.) Overall, optimistic messaging is much less painful and correct programs are still correct programs.

Pessimistic messaging allows us to move message-checks earlier in a program’s execution and to consolidate multiple message-checks into a single check. This allows us to identify and respond to failed message-checks long before a program needs to rely on those message checks. In many cases these message-checks can be moved all the way to the point of object construction, which allows them to be verified at compile-time. If this were possible then we could be surer of what our program does before execution verification. We have made a good step forward if our programs frequently have mistakes of expecting an object to respond to a message that it doesn’t understand.

Unfortunately complete compile-time checking is rarely possible because of weak language support, ‘null’s, or program behavior more complex than the capabilities of even a good language. Compile-time type checking also significantly reduces the reusability of classes: it requires generating new classes with all the proper “types” to be used in a particular context.

All the extra information for pessimistic messaging provided us with an additional form of documentation. Besides having the name of a method, the name of its parameters, and the context of the methods implementation (i.e. its class), pessimistic messaging allows us to express the expected methods on the parameters and the return value. Whether this is valuable documentation or not depends on the quality of expressiveness of the more core components: the method name, the parameter names and the context. If these are very descriptive and consistent through the whole application, then the pessimistic information may not be at all useful. The topic of documentation will be discussed in a later section.

Alternatives to explicit pessimistic messaging

There are other alternatives to explicitly declaring the pessimistic message-checks. We could have a program try to verify that the optimistic program will work correctly using either no extra information or much less information than the explicit pessimistic programs above. This would provide us with all the benefits of both optimistic and pessimistic messaging. It could also generate the additional documentation that a pessimistic program can provide. See [Brach+G 93][8] for a starting reference point to these types of languages. For this document I will ignore inference capabilities since they are not available in Java, Smalltalk, and other “mainstream” OO languages.

Copyright © 1997, Mark L. Fussell

Deciding on Optimistic vs. Pessimistic messaging

SmallJava_0 supports optimistic messaging. You can send a message to any object and if the object understands the message (it has implemented a matching method) it will respond. If not, the object would throw a "DoesNotUnderstandException" that the caller can catch and respond to. This is very clean and simple, relying on the same exception handling abilities in the rest of SmallJava. Optimistic messaging's problems are that a program can only be verified by running it and that there is less documentation of what is expected of a variable or parameter.

Should SmallJava_1 support pessimistic messaging: Should it support the ability to check whether an object understands a message before sending it to it? The answer would seem to be an emphatic yes. The only cost is the addition of the syntax "(#message,#message2,...)" that does a message check or that requires the user of a variable or parameter to do a message check. This is useful in several ways:

It allows a program to move the location of a message-check to the point where it can better handle a failure

It provides the possibility of compile-time verification.

It provides the possibility of extra documentation (that the program will actually use).

These all seem valuable enough to add them to SmallJava.

From Java and Smalltalk's point of view this is uncontroversial: this capability is in both languages. Java provides the "cast" operation and typed variables that we will discuss in future sections. Smalltalk provides the ability to ask an object whether it #respondsTo: a message. This returns a boolean instead of throwing an exception, but the meaning is the same. Smalltalk does not support an invariant on a variable [actually, some do or at least document the invariant], but that seems a useful capability within the spirit of commercial Smalltalk. So pessimistic message verification is available in both languages and should be available in SmallJava.

Requiring Pessimistic Checking

Now for the big question: Should SmallJava_1 REQUIRE only pessimistic message checking and abandon the optimistic message checking? So far, the answer would have to be no. In many cases the pessimistic checking is not providing us with any gain because there is no advantage to moving the message-check point, the checks are not compile-time verifiable, and the invariant provide poor extra documentation. But without the gain, there is no point in the pain: every time we want an explicit message-check we have to add a lot of extra words to our program and make sure all these message-checks are in agreement with each other.

In future sections we will be discussing other language features that may make pessimistic message checking more useful and less painful. We will also be discussing aspects that make pessimistic message checking less useful (e.g. for 'null's). After dealing with these features and aspects we can revisit the question of whether pessimistic checking is useful enough to be required. For now, SmallJava_1 supports both.

Summary

We defined and analyzed optimistic and pessimistic message checking and found that they are both useful enough to include in SmallJava_1 and that neither is so useful as to warrant excluding the other. Our change to SmallJava was the addition of a message-check with syntax (#message1, #message2) and of a message-requirement invariant which uses the same syntax.

---

[1] The terminology and behavior is similar to database transactions.

[2] I use "#foo" to label a message and "Class#foo" for a method in a particular class. The use of '#' is similar enough between both Smalltalk (where it indicates a Symbol) and Java (where for Javadoc it indicates a method) to be the best choice.

Copyright © 1997, Mark L. Fussell

[3] Or you could view it that Object defines all methods with a default behavior of: “throw new DoesNotUnderstandException();”
[4] The database transaction terminology would be a pessimistic “lock” on ‘point’
[5] We might now want to call them “invariables”
[6] We have changed the contract with the caller, see [Meyer 97]
[7] DoesNotUnderstandException is a subclass of RuntimeException, which explains why it does not have to be in a Throws clause.
[8] If anyone has a good collection of references for type inferencing, I will add them to this.

Typing: Better support for pessimism

SmallJava started with only optimistic messaging. You could send a message to any object and if the object did not have a corresponding method, a “DoesNotUnderstandException” would be thrown.

We added pessimistic messaging capabilities to SmallJava_1 but we did not require all messaging to be pessimistic. Pessimistic messaging allows us to move and combine message-checks to a point earlier in the execution of a program. This enables us to catch an error where we are more capable of handling it and to avoid some errors completely by compile time verification. Pessimistic messaging also provides extra documentation of how a variable is used and what a method does.

We did not make SmallJava_1 require all messaging to be pessimistic because it was not a big enough benefit in all cases to be worth the pain. The pain is explicitly defining all the messages we expect an object to understand in all contexts and then make sure all these message-checks agree with each other so the compiler is happy. This is not worth it in the cases where there is no advantage to moving the message-check, the check can not be compile-time verified, and the extra documentation is not meaningful.

Our decision might change if we had a better mechanism to manage our pessimistic message-checks. Either the mechanism has to be more convenient or it has to have added benefits that make it more useful. This chapter discusses making pessimistic behavior easier and more useful by using messageGroups, a preliminary form of “Type”. If the pessimism becomes easier we may make it the required behavior for SmallJava_2. In any case we will have something close enough to Java like static typing to be useful to compare.

Annoyances in pessimistic messaging

One of the biggest annoyances of pessimism so far is explicitly enumerating all the messages we expect of an object. Even our simple example program [1] has too much noise:

```
class PointClass {
   // ...
   vectorFrom((#x,#y) point) {
      return new PointClass (x - point.x(), y - point.y());
   }
}

class LineClass {
   LineClass ((#x,#y,#vectorFrom) pointA, (#x,#y) pointB) {
      this.pointA = pointA;
      this.pointB = pointB;
   }

   vector() {
      return pointA.vectorFrom(pointB);
   }

   (#x,#y,#vectorFrom) pointA;
   (#x,#y) pointB;
```


Copyright © 1997, Mark L. Fussell

```
    }
```

It already has twelve individual message references combined into five pessimistic invariants. What's worse is that changes in PointClass#vectorFrom could cause all the other invariants to have to change. Assuming a point could have a name, then changing #vectorFrom to:

```
    vectorFrom((#x,#y,#name) point) {
      return new PointClass(name()+point.name(), x - point.x(), y - point.y());
    }
```

will force all the other invariants to be updated to include #name also or the compile time verification will complain.

Combining messages into messageGroups

If we could define a larger unit of granularity that contains multiple messages, a messageGroup[2], we might have an easier time. Let's try the following simple syntax:

```
messageGroup #Xy = (#x, #y);
messageGroup #Point = (#Xy, #vectorFrom);
```

where mentioning a messageGroup is just like mentioning the messageGroup's messages explicitly. Message groups are compared by value: two message groups are equal if they have the same messages in them after expansion. Our previous pessimistic message checks and invariants become inline, unnamed message groups.

Using these messageGroups, our program becomes:

```
class PointClass {
  // ...
  vectorFrom((#Xy) point) {
    return new PointClass(x - point.x(), y - point.y());
  }
  // ...
}

class LineClass {
  LineClass ((#Point) pointA, (#Xy) pointB) {
    this.pointA = pointA;
    this.pointB = pointB;
  }

  vector() {
    return pointA.vectorFrom(pointB);
  }

  (#Point) pointA;
  (#Xy) pointB;
}
```

Which now has only two types of message checks in five invariants. This is a little simpler, but what about adding #name again? We can either mention it explicitly as before:

```
    vectorFrom((#Xy,#name) point) //...
```

which would cause all the callers to change their invariants or we can modify our #Xy message group to include #name:

```
    messageGroup #Xy = (#x, #y, #name);
```


Copyright © 1997, Mark L. Fussell

That is certainly simple. It also feels a bit disingenuous. We named the group #Xy for a reason: all we required from the object is that it support #x and #y. To sneak an extra, unrelated message into the group just because it is convenient does not seem appropriate and certainly would confuse the reader of our program.

If we don't modify #Xy we can still modify #Point:

```
messageGroup #Xy = (#x, #y);
messageGroup #Point = (#Xy, #vectorFrom, #name);
```

and

```
vectorFrom((#Xy,#name) point) //...
class Line {
   LineClass((#Point) pointA, (#Xy,#name) pointB) //...
```

Now we only have to change three invariants instead of five. But why could we modify #Point if we couldn't modify #Xy? Just because it had a different name? Well, yes.

Types

The messageGroup named #Point represents a "Type", a meaningful grouping of objects that provide the same interface. We can include the ability to respond to #name as part of the capabilities of a #Point. For the messageGroup named #Xy we were just providing a "shortcut" name for several messages without any fuller or independent logical meaning.

So why don't we just throw away #Xy and use just #Point? This would give us:

```
messageGroup #Point = (#x, #y , #vectorFrom);

class PointClass {
   // ...
   vectorFrom((#Point) point) {
      return new PointClass(x - point.x(), y - point.y());
   }
   // ...
}

class LineClass {
   LineClass ((#Point) pointA, (#Point) pointB) {
      this.pointA = pointA;
      this.pointB = pointB;
   }

   vector() {
      return pointA.vectorFrom(pointB);
   }

   (#Point) pointA;
   (#Point) pointB;
}
```

Now if we add #name as a capability because of:

```
   vectorFrom((#Point) point) {
      return new PointClass(point.name(),x - point.x(), y - point.y());
   }
```

we can just add it in one place

Copyright © 1997, Mark L. Fussell

```
messageGroup #Point = (#x, #y , #name, #vectorFrom);
```

and we don't have to worry about changing anything else. Life is easy.

Excessively Restrictive Typing

Unfortunately, what we have just done has a large negative impact. We just required that an object must support all of (#x,#y,#vectorFrom,#name) to be acceptable as a pointB in our LineClass. But that is not true. We only need the object to support (#x,#y,#name). Similarly we only need pointA to understand (#x,#y,#vectorFrom). We accidentally became excessively restrictive of acceptable objects through our simplification.

So what? We were planning on using only PointClasses anyway, and they support all of the #Point messages anyway.

Unfortunately that is a sign of being "near-sighted" in both time and development scope. As the system grows I could want to use an independently developed library of classes that includes a 'Point' which does not have a method #name. But I can not. Lines require that a Point understand #name even when it never sends a #name message to them.

Over time I could also want to reuse lines to handle some UI points which are floating point and also have names, but I could not because my #Line uses #Points and #Point says it only works with #Integers instead of objects that respond to just (#+, #-, #*).

There are words for this type of program: inflexible, limited, not reusable, and "bad".

If a language feature supports easily doing the wrong thing in the short term it is very hard to keep track of the larger picture and we have to very consciously strive for it. The new messageGroups are one of these features so we have to use them with care.

The principal problem preventing us from more exact and flexible typing is inertia. We do not want to change five different occurrences of verification checks to include #name and #vectorFrom. And then do the same thing again when some objects need to respond to #foo. And then change them back when we get rid of #foo. We are just lazy and want as little of the program to change as possible.

There are words for this type of program too: stable, maintainable, and "good".

Preventing Excessive Typing

So how do we get the "good" with as little of the "bad" as possible? The main way is to think harder about our object type models. Do we really want all #Points to respond to #name? Or do we have a special #NamedPoint? Is the protocol #Xy useful in general so we should still include it and try to use it when all we want are #x and #y? If we produce a good type model we will probably get the best maintainability and reusability possible[3].

Although our example is too small to really see much difference, we might choose the following as the best description of our domain:

```
messageGroup #Xy = (#x, #y);
messageGroup #Point = (#Xy , #vectorFrom(#Xy), ...other stuff...);
messageGroup #Named = (#name);
messageGroup #NamedPoint = (#Point , #Named);
messageGroup #Line = (#vector);

class PointClass {
   // ...
   vectorFrom((#Xy) point) {
      return new PointClass(x - point.x(), y - point.y());
```


Copyright © 1997, Mark L. Fussell

```
    }
    // ...
}

class LineClass {
    LineClass ((#Point) pointA, (#Xy) pointB) {
        this.pointA = pointA;
        this.pointB = pointB;
    }

    vector() {
        return pointA.vectorFrom(pointB);
    }

    (#Point) pointA;
    (#Xy) pointB;
}
```

If we later decide to use #names with our lines then we can either change the invariant on pointA (to either "(#Point,#Named)"or "(#NamedPoint)") or we can decide that we really have a new #NamedLine which would have a new NamedLineClass to implement it. We have reduced the excessive restrictions in exchange for what looks like more work (more Types) but should be more stable and less work over time.

Types and Classes

MessageGroups as simple types make the differences between classes and types very visible. A messageGroup is an alias for a set of messages. This documents a concept, the type, and allows pessimistic verification of that type. A class describes an implementation of messages through methods (behavior) and instance variables (state) so the program can actually build and execute objects. Types classify and verify. Classes implement.

Reviewing the variations of Typing

We now have three variations of typing, optimistic, message-level pessimistic, and type-based pessimistic. These are all different in terms of typing precision.

Optimistic messaging is exactly typed. The only messages ever required of an object are the ones actually sent to the object. It is even exact in terms of different program executions and flows. During one program execution a message might not be required, but in the next it would be.

The problem with optimistic messaging is that these types can only be verified by program execution or through a separate inference and pessimistic checking process.

Pessimistic messaging as described in SmallJava_1 is likely to be almost exactly typed. Since each message that should be verified must be explicitly listed it is unlikely that the messages will become excessive except through lack of maintenance. Another source of inexactness is that different program flows may have different requirements but the pessimistic checks will union all the program flows.

Pessimistic messaging was a useful addition to SmallJava_1 because it could help detect errors earlier in program execution or possibly at compile-time. It could not completely replace optimistic messaging because it was too much work in the cases where there was little or no gain.

Pessimistic typing is likely to be excessively restrictive. Pessimistic typing has messageGroups that can be used to collect messages into Types. This provides more power for using pessimistic verification: We can more easily express message requirements with one or two groups instead of many individual messages and we can reduce program maintenance by changing a single group instead of many verifications.

Copyright © 1997, Mark L. Fussell

Unfortunately these messageGroups lead to combining verifications that are not actually identical and restricting an object more than is needed. This is not inherently required for messageGroups, but is the trade-off for increased maintainability. It is up to the software developer to consciously pick the designs that are descriptive, maintainable, and do not cause excessive restriction.

SmallJava_2 and Pessimistic Typing

For SmallJava_2 we have to decide whether to use the pessimistic typing capabilities described above and ultimately whether to start mandating pessimistic behavior.

Should SmallJava_2 support messageGroups and pessimistic typing? Yes. MessageGroups make pessimistic behavior easier and more functional while costing very little. The main language cost is an addition in syntax for defining messageGroups. Other than that, a messageGroup works just like an alias for a collection of messages. It is a simple to understand addition and is consistent with the behavior of SmallJava_1. The other main cost is that it can encourage less flexible software, but it is only a mild encouragement and it is completely under the control of the developer.

Should we REQUIRE pessimistic behavior now that we have better language support for it? I will say "no" to defer until covering "null" values, but we are close to making that decision. For the moment I will compare optimistic and pessimistic behavior in SmallJava_2 and discuss choosing between them.

Comparing Optimism and Pessimism

We now have a much more useful and interesting pessimistic mechanism. We can now retouch bases with our original completely optimistic SmallJava PointClass and see how our pessimistic abilities change it.

The original SmallJava_0 code was:

```
class PointClass {
    PointClass (x, y) {
        this.x = x;
        this.y = y;
    }

    x() {return x;}
    //...

    vectorFrom(point) {
        return new PointClass (x - point.x(), y - point.y());
    }

    x,y;
}

class LineClass {
    LineClass(pointA, pointB) {
        this.pointA = pointA;
        this.pointB = pointB;
    }

    vector() {
        return pointA.vectorFrom(pointB);
    }

    pointA, pointB;
}
```


Copyright © 1997, Mark L. Fussell

A SmallJava_2 version with fully pessimistic typing would be:

```
messageGroup #Xy = (#x,#y);
messageGroup #Point = (#Xy,#r,#theta,#vectorFrom);
messageGroup #Number = (#^,#-,...);
messageGroup #Line = (#vector);

class (#Point) PointClass {
   PointClass((#Number) x,(#Number) y) {
      this.x = x;
      this.y = y;
   }

   (#Number) x() {return x;}
   //...

   (#Point) vectorFrom((#Xy) point) {
            return new PointClass (x - point.x(), y - point.y());
   }

   (#Number) x,y;
}

class (#Line) LineClass {
   LineClass((#Point) pointA, (#Point) pointB) {
      this.pointA = pointA;
      this.pointB = pointB;
   }

   (#Point) vector() {
      return pointA.vectorFrom(pointB);
   }

   (#Point) pointA;
   (#Point) pointB;
}
```

What are the differences between these two programs?

In terms of correct program execution, nothing. The optimistic and pessimistic programs both send the same messages and both get the same objects back as results.

In the case of incorrect calls to Point and Line, the pessimistic version will throw an exception earlier or quite possibly notify the developer at compile time. The optimistic version will throw an exception at run time.

The optimistic program is less “noisy” and is easier to change. We can simply add a new method to PointClass and use it in LineClass. But we can also accidentally send a new message without defining the method in PointClass. This will cause a runtime error if and when the program sends this message. It is best if this is sooner rather than later.

The pessimistic program is harder to change but will catch more errors at compile time. If we add a new method to PointClass we also need to add it to #Point before we can use it in LineClass. If we forget to add the message to #Point we will get a compile time error when we try to call the new message. If we forget to

Copyright © 1997, Mark L. Fussell

implement the method in PointClass we will get a compile time error saying PointClass does not implement all of the #Point type.

The pessimistic program has more description of its requirements and behavior. We can study the messageGroups to see what types are important and what messages these types support. This is completely separate from the actual Class implementations. Although an optimistic program could also provide this documentation there is no encouragement (to put compiler errors in the most positive light) to keep the documentation up to date.

The pessimistic program is less flexible. We can only use #Points in our LineClass. We can't use another type of point that only understand #x and #y because #Point requires #r and #theta too (even though LineClass may never need #r and #theta). You should also note the comment in the next section.

Comments on Reality

Unfortunately the type behavior in SmallJava_2 is more "ideal" and flexible than what most statically typed languages support, including Java. For example, instead of message based verification with types as message aggregates, most statically typed languages use named-type based verification. For these languages an object can only pass through a type-check if its class specifically "implements" the named type. You can not use a third party PointClass that supports all the #Point messages with LineClass unless it actually specifically mentions our #Point type. This is not too likely. This language "feature" severely punishes reusability for pessimistic typing. This I will address in a future chapter when we start bringing some of the idealizations of SmallJava down to the realities of Java.

Although this document is meant to describe the differences between Smalltalk and Java, I though it was unfair to discuss a language feature only in the context of a weak version. In many cases Java is simply missing features which can be discussed as add-ins. But in other cases Java has a poor implementation of a language feature that I would rather describe ideally and then explain how Java parts from that ideal. Pessimistic type checking is one of those features.

Environments makes a difference

Overall, the above programs are very similar, but coding them will probably feel very different. Besides language features this will be because of the development environments and compiling technology associated with each. Generally optimistic languages have much more interactive development environments, so programs feel "directly manipulated" and grown. Generally pessimistic language development environments have a specify, implement, and verify process.

Although language features have a significant impact on the development environment I will not consider that as part of SmallJava feature evaluation. This environment difference is becoming smaller[4] and comparing environments is beyond the scope of this document.

Choosing

Choosing between optimistic and pessimistic behavior in the SmallJava language will be hard because they each have tradeoffs. For most of the tradeoffs we would like a language that allows you to choose which approach to take at any point. If you want to document a class more fully, add and use types. If you want to be more pessimistic add some pessimistic checks and invariants. Generally as a subsystem matures it can and will become more pessimistic so people can understand and rely on its behavior.

The one tradeoff that is hard to swallow is the diminished flexibility and reusability of an excessively typed pessimistic program. It is a severe hindrance if a language or program prevents objects from being useable in appropriate circumstance solely because of pessimistic typing. We will have to spend more time recoding (and debugging and documenting) identical functionality to support different pessimistic variations of the identical optimistic program. This is by far worse in named-type verification languages (of which Java is a member).

But no choosing yet, we must first cover "nothing".

Copyright © 1997, Mark L. Fussell

---

[1] Note the change to suffixing with “Class”, this is to make discussions about types clearer later on. It also happens to be my naming convention for Java code. See [Fussell-1]

[2] I use the term messageGroup in this chapter because it is descriptive and it does not bring in currently unwanted connotations that other terms like interfaces or protocols due.

[3] Different project goals will adjust this too. Prototypes frequently care more about speed then either of these criteria. Larger projects and frameworks tend to care more for reusability than other types of projects would.

[4] For example, see the IBM VisualAge for Java beta (http://www.software.ibm.com/ad/vajava), which is a (currently immature) version of one of the best Smalltalk development environments.

Copyright © 1997, Mark L. Fussell

References

Brach+G 93 Gilad Bracha and David Griswold. "Stongtalk: Typechecking Smalltalk in a Production Environment" in *OOPSLA 1993 Conference Proceedings*. Addison-Wesley, Reading, MA, 1993.

Budd 87 Timothy Budd. *A Little Smalltalk*. Addison-Wesley, Reading MA, 1987.

Fussell-1 Mark L. Fussell. "Java Development Standards" http://www.chimu.com/publications/javaStandards/

Fussell-2 Mark L. Fussell. "Java-Smalltalk Syntax comparison" http://www.chimu.com/publications/javaSmalltalkSyntax.html/

Goldberg+R 83 Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

Gosling+JS 96 James Gosling, Bill Joy, Guy Steele. *The Java™ Language Specification*. Addison-Wesley, Reading, MA, 1996.

Meyer 97 Bertrand Meyer. *Object Oriented Software Construction, 2nd Edition*. Prentice-Hall, Englewood Cliffs, NJ, 1997.

Copyright © 1997, Mark L. Fussell