

ArchJava: An Evaluation

Andrei Alexandrescu
Konrad Lorincz

Abstract

Analyzing a tool intended to make large projects more manageable and increase programmer productivity is a daunting task. Not only applying transformations to a large body of code and analyzing its evolution over time is a large, time-consuming effort, but also even in the presence of the appropriate logistics, measuring and quantifying the results is hard.

In an attempt to focus our area of research, we decided to analyze the ArchJava language from two perspectives.

One perspective focuses on a comparative analysis of the features offered by ArchJava versus those features emulated in plain Java. Our goal is to obtain some data about ArchJava's benefits as measured in source code reduction, maintainability, decoupling, error prevention, and error correction.

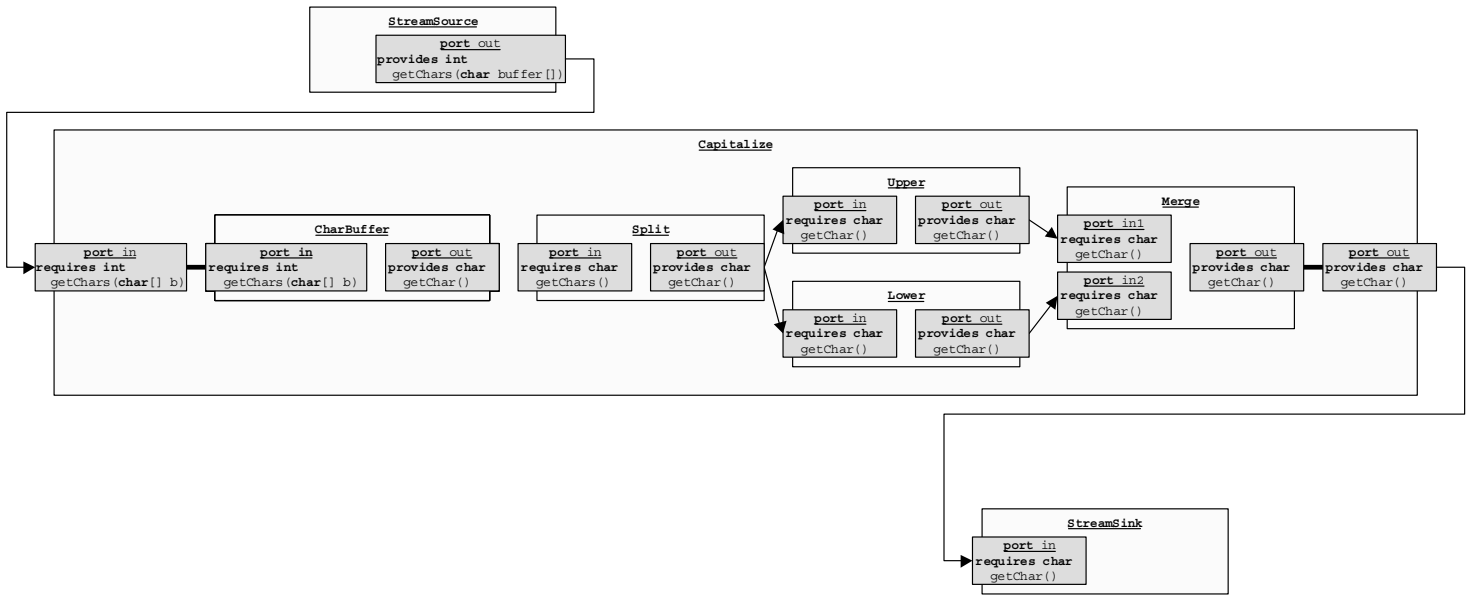
The other direction is performance. For each of ArchJava's features, we compared them in speed with counterparts implemented in plain Java.

The two perspectives are not orthogonal and are salient only when analyzed in simultaneity. On one hand, a feature that reduces coupling might be often useful even though it exacts a cost in performance. On the other hand, costly abstractions are a dime a dozen. A good tool strikes the right balance in between the two, and we will try to assess whether this is the case for ArchJava.

Introduction

ArchJava ensures a stronger connection between architecture and implementation than plain Java and than the architectural languages that predate ArchJava. We set out to measure to what extent an ArchJava-based project is better than an architecture hand-implemented in straight Java, architecture providing a comparable amount of decoupling and type safety.

Time would not allow us to develop two equivalent projects – one using ArchJava and one using Java only – and compare them. Thus, we decided to simply reverse engineer one of the – supposedly elegant – examples provided by the authors themselves, re-implement equivalent designs in Java, and compare the results.



The chosen example is Capitalize, a simple console-oriented program that reproduces its input with alternating lowercase and uppercase characters.

After rewriting Capitalize in plain Java, we measured two key aspects.

One is evaluating the extent to which the ArchJava project has had added value in terms of:

- Smaller code;
- Improved understandability/maintainability;
- Most important, less burden on programmer’s discipline when it comes about preserving the architecture’s communication integrity.

The other direction is efficiency. We measured the throughput of the two programs when routed to the null device, in a quest for finding out whether, if any, additional overhead ArchJava incurs.

Emulating ArchJava Constructs with Java Constructs

In order to analyze the benefits offered by an ArchJava-powered project, we need to compare them against a reference project implemented in plain Java, that offers a comparable architecture.

The main architectural artifact provided by ArchJava is the component type, which offers fine-grained access restriction to its methods via the port keyword.

A solution we considered was to implement a library that offers abstractions such as “Component”, “Port”, “Connection”, and such, in a classic object-oriented manner. This way, we would build a library that offers features allegedly similar to those offered by ArchJava.

Below is an excerpt of a possible Java source-level library implementing the Port and Component abstractions.

```
class Port
{
    private String name;
    private Port peerPort;
    private Vector broadcastPorts;

    public void connect(Port peerPort);
    public void addBroadcastPort(Port broadcastPort);

    public void receiveMsg(String msg); // provides
    public void sendMsg(String msg);    // requires
    public void broadcastMsg(String msg); // broadcast
}

class Component
{
    private Object objClass;
    private Vector ports;

    public Component(Object objParent);

    public Port getPort(String name);

    ...
}
```

However, we abandoned this path because it would introduce an architectural mismatch, forcing us to compare different program structures. The architecture of a system using ArchJava would be fundamentally different from the architecture of a system built using such a library. Essentially, ArchJava provides artifacts that are beyond what can be abstracted away in an object-oriented library. In particular, the level of type safety, decoupling, and integrity offered by ArchJava cannot be successfully matched by a Java library. To connect components in a way that's reminiscent of ArchJava's, a Java source-level component framework must rely on conventions, casts and introspection.

We decided that a reasonable comparison could be made between a given system created with ArchJava, and a similar system created from the ground up in straight Java, sporting the same coupling and visibility characteristics as the ArchJava project.

That is:

- The communication between the Java-implemented components must be made exclusively through interfaces that expose the same functionality as their ArchJava counterparts;
- The Java components must be hidden to the best extent possible from each other, so that they communicate exclusively through their dedicated interfaces;
- The level of type checking (safety) stays the same, that is, the Java code must use no additional casts or introspection to facilitate communication between components.

Following this course of action, we noticed that each port engenders *three interfaces*: one for each of port's *requires*, *provides*, and *broadcast* clauses. These three interfaces ensure typesafe communication between peer components.

The component defining a port implements that port's *provides* interface, and *requires* peer components to implement the *requires* and *broadcast* interfaces.

For example, consider the component *Upper* in *Capitalize*:

```
public component class Upper {
  port in {
    requires char getChar() throws IOException;
  }

  port out {
    provides char getChar() throws IOException {
      ...
    }
  }
}
```

This component defines two interfaces, an outgoing interface `in`, and an incoming interface `out`. `Upper` implements the incoming interface and connects with other components that provide the outgoing interface.

Below is the equivalent Java code that supports a similar level of decoupling:

```
interface Upper_port_in_requires {
  char getChar(Object sender) throws IOException;
}

interface Upper_port_out_provides {
  char getChar(Object sender) throws IOException;
}

class Upper implements Upper_port_out_provides {
  void connect(Upper_port_in_requires peer) {
    ... store peer in a private data member ...
  }
  char getChar(Object sender) throws IOException {
    ...
  }
}
```

This decomposition does allow preservation of communication integrity for the class `Upper` because the programmer can connect `Upper` to other components by using only the interfaces `Upper_port_in_requires` and `Upper_port_out_provides`. Thus, other components remain oblivious to the `Upper` class itself.

More steps are necessary to realistically enforce communication integrity in the Java project, even before implementing plumbing functions such as `connect`:

- The code above must be broken in several Java packages such that the `Upper` class itself is not visible to other classes and is accessed solely through its `Upper_port_out` interface.
- Worse, each interface definition must reside in its own file, which in the general case leads to $3N$ (`provides`, `requires`, and `broadcasts` clauses) files for each port that the component defines.
- Programmer discipline is still needed to make sure that each peer component does not unwittingly use the implementation class (the component) directly. Components are supposed to be used only from within their parent.

So far, the `Upper` class and its artifacts imitate ArchJava's `Upper` component only from a type safety viewpoint. To ensure the semantics are equivalent, in the general case, more code needs to be written:

- For each occurrence of the `connect` keyword in ArchJava, the component must store the connection peer in a private data member.
- For broadcast interfaces, there can be multiple receivers for one message; therefore, the initiator of the broadcast must store its receivers in a collection (such as an array).
- Whenever the component needs to broadcast a call, it must iterate through the collection and invoke that method.
- Also, when a port uses both `requires` and `provides` clauses, the interface interplay is slightly more intricate than illustrated above: when calling a method via the `provides` interface, a caller must identify itself as the sender. This is necessary for emulating the `sender` keyword in ArchJava, which identifies a calling port to its callee. In the example above, no port uses both `requires` and `provides` clauses simultaneously; therefore, the type identifying the sender is simply `Object`. We note that this interface interaction establishes a circular dependency between two interfaces, which increases cohesion (desirable) but also coupling (undesirable) so it is generally advised against. However, ArchJava does increase cohesion while masking the coupling to a certain extent (a less than perfect masking, as discussed in the next section).

Dependency Analysis of ArchJava Constructs

The previous section has shown that an ArchJava program can be expressed in Java with significant source code overhead, and in addition requires a certain amount of programmer discipline.

In addition, the Java source code must be split across several files, which practically might cause programmer confusion.

However, having a fine-grained interface distributed over multiple files is a blessing as much as is a curse. Conversely, we noticed an important drawback of ArchJava that might become a showstopper for realistically sized projects.

Consider an ArchJava component `widget` that defines several ports, each using some or all of the `requires`, `provides`, and `broadcasts` clauses. Several other components, mostly unrelated to each other, connect to the ports defined by `widget`.

Unfortunately, unless the ArchJava compiler largely improves its file generation algorithm, this setting is bound to be a dependency nightmare. Every time any port of `widget` is changed, *all* other components connecting to any of `widget`'s ports must be recompiled because they use and/or implement interfaces defined inside the `widget` component. For example, if you change, say, an output port of `widget`, the components connected to its input ports are affected as well.

We noticed that the current version of ArchJava simply neglects this issue altogether, practically regenerating everything whenever any `.archj` file is touched.

A better idea would be for the ArchJava compiler to do all the work on the side (compile to temporary files), and then compare the newly generated `.java` files with the `.java` files from the previous compilation. The `.java` files are to be replaced with the newly generated temporaries only if they are not identical. Otherwise, their timestamps shouldn't be affected and thus the `javac` compiler won't have to recompile them.

Still, this solution doesn't scale with respect to the ArchJava compiler itself, which does have to touch and generate a plethora of files no matter how small the change in a component is.

By comparison, the multiple-files approach in our equivalent Java project naturally allows minimal recompilation whenever an interface is changed.

While it was obviously not a problem with a toy project like Capitalize, the scaling problem might become overkill for many real-world projects, where compilation from scratch is measured in hours. For such projects, good incremental compilation is instrumental to rapid development, and currently ArchJava does not support it.

Benefit Analysis of ArchJava

This section contains some number comparisons in terms of lines of code, as well as our subjective experience in reading, understanding, and writing ArchJava code.

Overall, our experience with ArchJava is that it considerably increases expressiveness in implementing architectures of a certain nature – specifically, cascaded interconnected components. The communication between various components is clearer and safer. ArchJava enforces excellent communication integrity, thus facilitating the understanding of complex interconnected components.

Also, ArchJava provides some automatic one-to-many dispatch abilities in two ways. First, a port P can be connected to multiple ports simultaneously. The ports connected to P can then initiate communication with P. This connection is efficient because there's no central authority brokering the communication.

Second, a port's broadcasts clause allows defining methods that initiate communication with an unbound number of peer ports, feature that amounts to a simple and effective implementation of the Observer design pattern.

Compared to a pure Java implementation of an architecture that has the same level of communication integrity, the ArchJava implementation is more compact in both number of files and sheer source code size, results summarized in the table below:

	Files	Classes	Interfaces	Lines	Words	Bytes	Bytecode
ArchJava	9	8	0	195	465	3437	19262
Straight Java	18	8	9	190	565	5134	11090
Improve-ment	200%	0%		-2.5%	17%	33%	-75%

However, ArchJava does incur dependency problems, as discussed in the previous section.

Our general impression about ArchJava was that its component architecture with its artifacts (ports, port interfaces, connections and such) is an excellent feature with a large range of applicability. In particular, pipe/filter architectures can be expressed in a very terse and expressive manner.

Reactive systems such as Model-View-Controller can greatly benefit of the broadcasts facility and of the fine-grained communication control.

Event-based can also benefit of ArchJava's powerful connection mechanisms such as `glue` and `connect` pattern.

ArchJava addresses and simplifies complex communication patterns and therefore is not as useful for architectures that emphasize complex state transformations spanning relatively few objects, such as state machines.

Speed Analysis

We measured the speed of the ArchJava-generated code versus the speed of our equivalent Java program. Before stating the conclusions of that comparison, please allow us to make a couple of prefatory observations on measuring speed and comparisons in general.

When comparing two different programs, it is important that apples are compared with apples. Otherwise, the results of the measurements are biased and hide important differences on other dimensions than sheer performance.

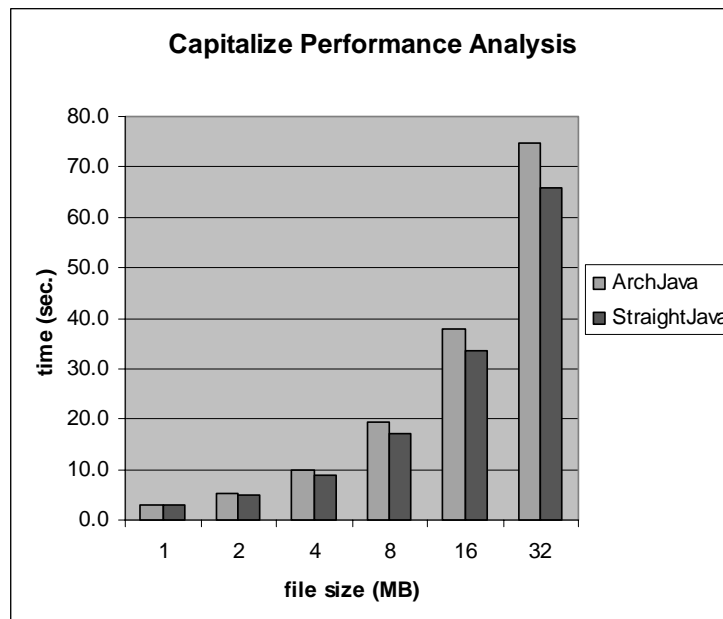
For example, we should not compare the performance of an ArchJava project with the performance of a functionally equivalent program, especially tuned for speed while completely throwing out the window any maintainability or modularity.

That's why we carefully engineered our pure Java implementation to exhibit not only the same functional behavior, but also the same amount of flexibility as the ArchJava program.

This being said, we were glad to notice that ArchJava does not exact a significant price in speed. ArchJava's overhead lies generally around 10%. We measured the throughput of the ArchJava and the Java program reading the same disk files and outputting to the `null` device. The overhead ratio increased slightly with input size. You can see the results of the measurements summarized in the table below.

The measurements were made with the UNIX tool `time`, which generates three outputs: *real*, *user* and *sys*. We included the results of all three values, although we concentrated on *real*, which according to the `time` man page is "the elapsed real time between invocation and termination." The percent performance difference and the graph all compare the elapsed real time.

File name	Size (kB)	ArchJava			Straight Java			Performance Diff.
		real (sec)	user (sec)	sys (sec)	real (sec)	user (sec)	sys (sec)	% difference
file_1MB	1,023	3.091	2.850	0.210	2.818	2.510	0.260	
		3.099	2.820	0.140	2.952	2.580	0.250	
		3.065	2.880	0.160	2.890	2.580	0.220	
		<i>Average</i>	<i>3.085</i>	<i>2.850</i>	<i>0.170</i>	<i>2.887</i>	<i>2.557</i>	<i>0.243</i>
file_2MB	2,046	5.508	5.390	0.160	5.019	4.600	0.240	
		5.335	5.150	0.080	5.124	4.630	0.280	
		5.365	5.270	0.160				
		<i>Average</i>	<i>5.403</i>	<i>5.270</i>	<i>0.133</i>	<i>5.072</i>	<i>4.615</i>	<i>0.260</i>
file_4MB	4,092	9.934	9.680	0.230	9.040	8.690	0.300	
		10.114	9.840	0.250	8.833	8.680	0.220	
		10.056	9.710	0.210	8.835	8.560	0.170	
		<i>Average</i>	<i>10.035</i>	<i>9.743</i>	<i>0.230</i>	<i>8.903</i>	<i>8.643</i>	<i>0.230</i>
file_8MB	8,185	19.492	19.160	0.300	17.266	16.610	0.340	
		19.299	18.780	0.430	16.920	16.710	0.200	
		19.365	19.030	0.240	16.976	16.720	0.180	
		<i>Average</i>	<i>19.385</i>	<i>18.990</i>	<i>0.323</i>	<i>17.054</i>	<i>16.680</i>	<i>0.240</i>
file_16MB	16,370	37.881	36.990	0.400	34.388	32.980	0.960	
		37.895	37.400	0.520	33.334	32.910	0.440	
		37.827	37.550	0.450	33.328	32.800	0.450	
		<i>Average</i>	<i>37.868</i>	<i>37.313</i>	<i>0.457</i>	<i>33.683</i>	<i>32.897</i>	<i>0.617</i>
file_32MB	32,741	74.421	73.770	0.550	65.669	64.670	0.580	
		75.074	74.500	0.560	65.621	64.990	0.370	
		74.569	73.880	0.570	65.999	65.160	0.700	
		<i>Average</i>	<i>74.688</i>	<i>74.050</i>	<i>0.560</i>	<i>65.763</i>	<i>64.940</i>	<i>0.550</i>



Conclusions

Overall, we have had a positive impression of ArchJava. We did have problems with installation, problems that pointed to bugs in the implementation, bugs promptly corrected by the developer.

The language improvements made by ArchJava seemed coherent and nicely integrated with the rest of the Java language. The only wrinkle, the `ComponentCastException` exception (which is thrown when an `Object` is cast to a component in an illegal context) is a necessary evil, allowing components to integrate within Java's type system.

The compiler is, in our opinion, not ready for handling large projects with complex dependencies. ArchJava's terseness and ability to cram multiple mini-interfaces and their implementations in a single source file can be abused and sometimes might create dependency hogs. This is a fundamental tension and what we believe to be the major conceptual drawback of ArchJava; only intricate file analysis and incremental compilation can address this problem satisfactorily.

On the bright side, dependencies remain under architect's control. We should not forget that a Java program is also an ArchJava program. The architect of the system can intelligently combine ArchJava components with straight interface-based design to effectively manage the architecture of a system.

Bibliography

[1] Aldrich, J, et al. *ArchJava: Connecting Software Architecture to Implementation*. To appear in ICSE 2002.

<http://www.cs.washington.edu/homes/jonal/archjava/icse02.pdf>

[2] Aldrich, J, et al. *Architectural Reasoning in ArchJava*. To appear in ECOOP 2002. <http://www.cs.washington.edu/homes/jonal/archjava/archjava-proofs.pdf>

[3] * * *. *ArchJava Language Reference Manual*.
<http://www.cs.washington.edu/homes/jonal/archjava/archjava-language.pdf>