

Dec15-12: The Final Report

@PaniniJ: Capsule Oriented Programming from Within Java

Trey Erenberger, David Johnston, and Dalton Mills

Advised by Dr. Hridesh Rajan

Table of Contents

[Section 1: Project Overview](#)

[Section 2: Revised Project Design](#)

[org.paninij.lang](#)

[org.paninij.runtime](#)

[org.paninij.proc](#)

[Overview of Annotation Processing Pipeline](#)

[Overview of Pipeline Within Annotation Processor](#)

[Sub-Component: Checks](#)

[Sub-Component: Models](#)

[Sub-Component: Capsule Artifact Generator](#)

[Sub-Component: Message Artifact Generator](#)

[Section 3: Implementation Process Details](#)

[Section 4: Testing Process and Testing Results](#)

[Appendix I: Operation Manual](#)

[Operation Manual Overview:](#)

[1 - Setup the project to use JRE 1.7 or greater](#)

[2 - Download the at-paninij.jar](#)

[3 - Enable annotation processing](#)

[4 - Add at-paninij annotation processor](#)

[5 - Add the at-paninij as a referenced library](#)

[Appendix II: Alternative Designs](#)

[Alternative: Eclipse Plugin](#)

[Alternative: panc Interoperability](#)

[Appendix III: Other Considerations](#)

[Package Structure](#)

[Naming Conventions](#)

[Coding Guidelines and Conventions](#)

[Lessons Learned](#)

[Appendix IV: FAQ](#)

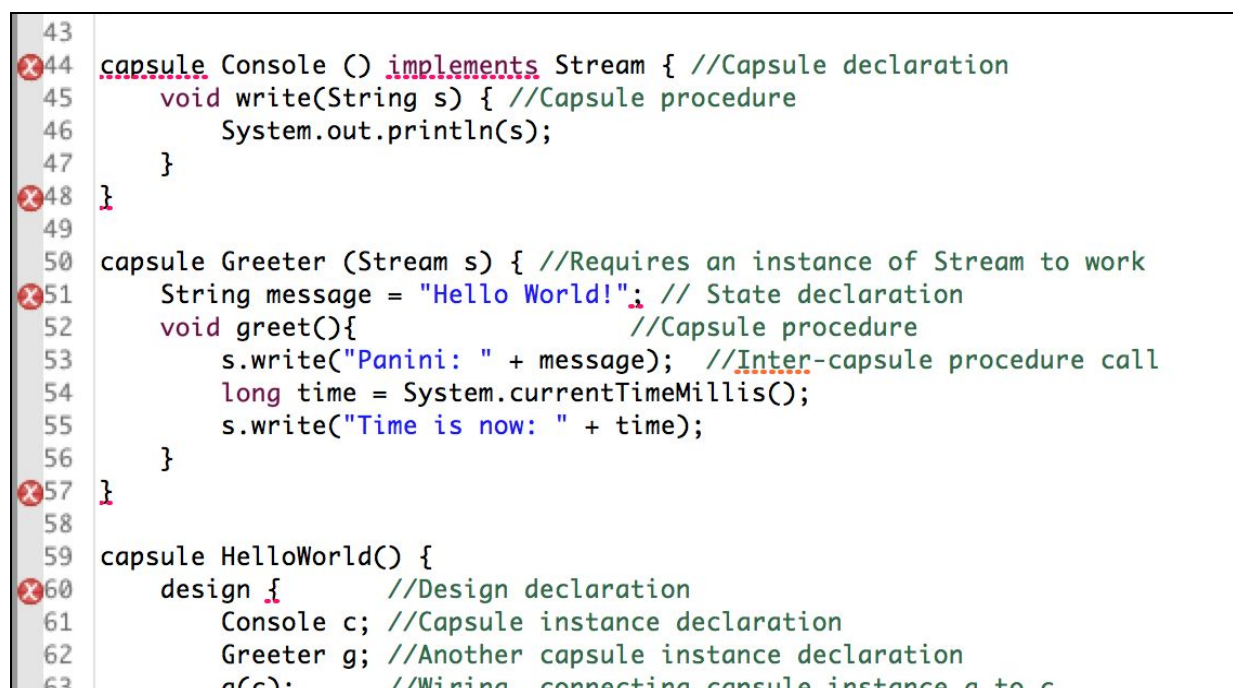
[Appendix V: Glossary](#)

Section 1: Project Overview

The @PaniniJ project is intended to provide tooling to enable the Panini programming model from within the Java ecosystem. We were initially given the source to the existing implementation of Panini in Java, PaniniJ, with the goal of developing Eclipse tools for the PaniniJ language.

PaniniJ was a fork of the entire standard Java compiler that had been modified by members of the Software Design Laboratory to allow new keywords and syntax specific to capsule programming. The PaniniJ language is similar to Java in many ways, however, important differences between Java and PaniniJ make many tools designed to be used with Java unusable with PaniniJ code.

We began our project by investigating possible ways to develop these Eclipse tools for PaniniJ. However, in discussion with our client, we soon realized that this route would not likely lead to a maintainable product, and we therefore chose to explore other designs.

A screenshot of the Eclipse IDE showing a Java file with PaniniJ code. The code is displayed with line numbers on the left. Lines 44, 48, 51, 57, and 60 are marked with red 'X' icons, indicating syntax errors. The code includes capsule declarations, procedures, and a design block. Comments explain the purpose of each line, such as '//Capsule declaration', '//Capsule procedure', and '//Design declaration'. The code uses keywords like 'capsule', 'design', 'implements', 'void', 'String', 'long', and 'System'.

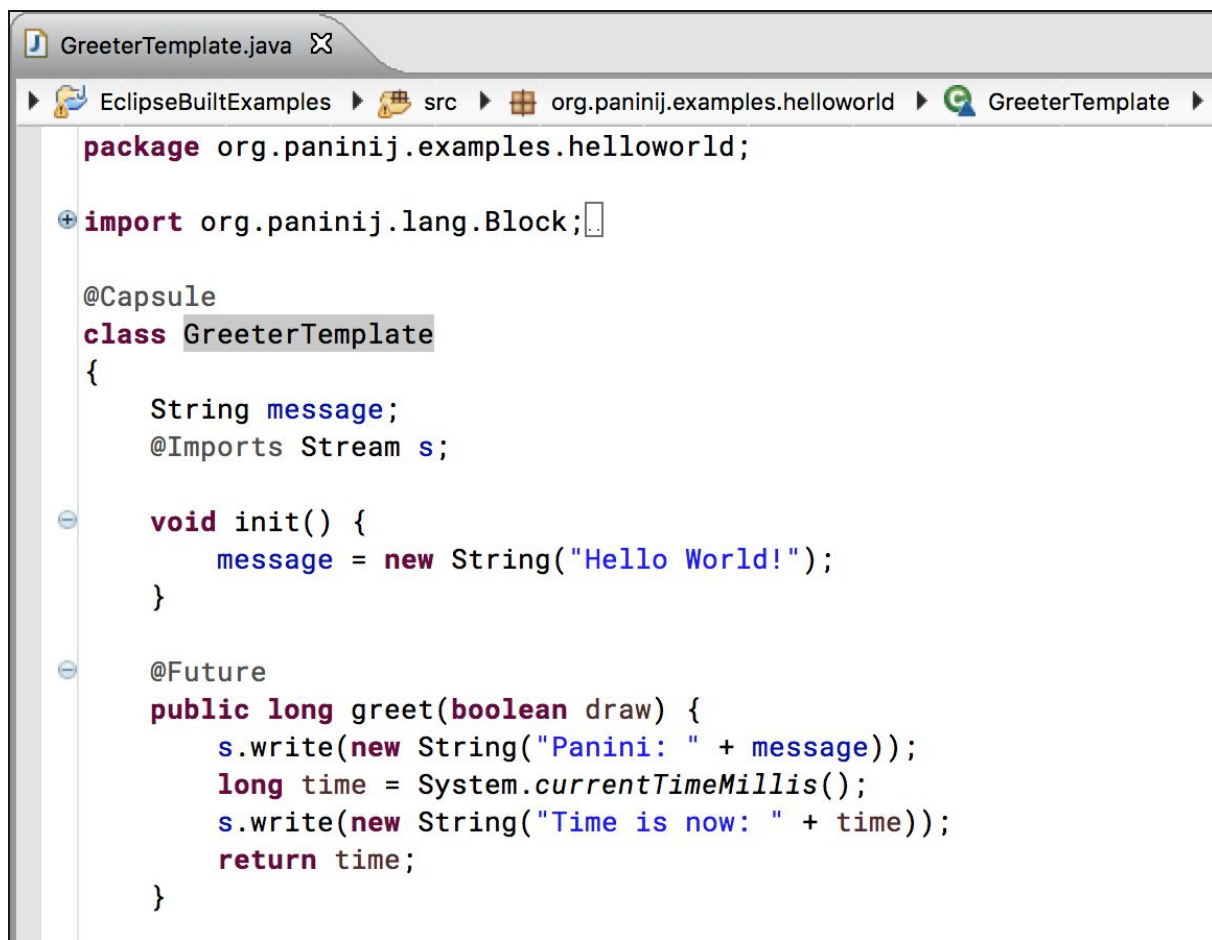
```
43
44 capsule Console () implements Stream { //Capsule declaration
45     void write(String s) { //Capsule procedure
46         System.out.println(s);
47     }
48 }
49
50 capsule Greeter (Stream s) { //Requires an instance of Stream to work
51     String message = "Hello World!"; // State declaration
52     void greet(){ //Capsule procedure
53         s.write("Panini: " + message); //Inter-capsule procedure call
54         long time = System.currentTimeMillis();
55         s.write("Time is now: " + time);
56     }
57 }
58
59 capsule HelloWorld() {
60     design { //Design declaration
61         Console c; //Capsule instance declaration
62         Greeter g; //Another capsule instance declaration
63         g(c); //Wiring connecting capsule instance g to c
```

Figure 1: The PaniniJ compiler worked, but other tools didn't know the PaniniJ language. This is an image of valid PaniniJ code displayed from within the Eclipse IDE.

During this exploration we considered numerous strategies to accomplish our overall goal to “provide tooling for PaniniJ” and we ultimately discovered that PaniniJ itself needed to be adjusted. Our client approached us with the idea of using annotation processing in order to write a Java compiler plugin. This would allow us to hook into the Java compiler and generate

code that would support the Panini model. Essentially, PaniniJ was already generating additional Java source code, but it did so using a fork of the entire Oracle Java compiler. Clearly, this would be hard to maintain, since should be updated with ongoing changes to the Java compiler.

We moved forward with developing an annotation processor, and we reimplemented the functionality of the existing PaniniJ compiler. We adapted the syntax of the PaniniJ language by converting the PaniniJ keywords into annotations. These annotations allowed for the code written by the user to be 100% valid Java. This method can be used with any standards-compliant Java compiler. This should allow our project to be much easier to maintain as future versions of Java are released.



```
package org.paninij.examples.helloworld;

import org.paninij.lang.Block;

@Capsule
class GreeterTemplate
{
    String message;
    @Imports Stream s;

    void init() {
        message = new String("Hello World!");
    }

    @Future
    public long greet(boolean draw) {
        s.write(new String("Panini: " + message));
        long time = System.currentTimeMillis();
        s.write(new String("Time is now: " + time));
        return time;
    }
}
```

Figure 2: PaniniJ features were ported to annotations which allowed Java IDE's to handle and perform Java syntax and type checking.

Our version, @PaniniJ, should now be completely compatible with any standards-compliant Java tooling (i.e. compilers, build tools, and IDEs). We have tested our compiler plugin with Eclipse, Netbeans, Oracle's javac, and Maven. We are also able to provide real-time error checking to make sure that the user's code follows the somewhat exacting requirements of

the Panini model. These error messages are integrated right along-side plain-old Java language syntax errors, so the way in which or system gives feedback to the user should be completely natural for Java developers. We were able to accomplish this in an IDE/tool agnostic way that ensures that @PaniniJ has can be used with a huge range of standard Java development tools, while keeping @PaniniJ easily maintainable by a small team.

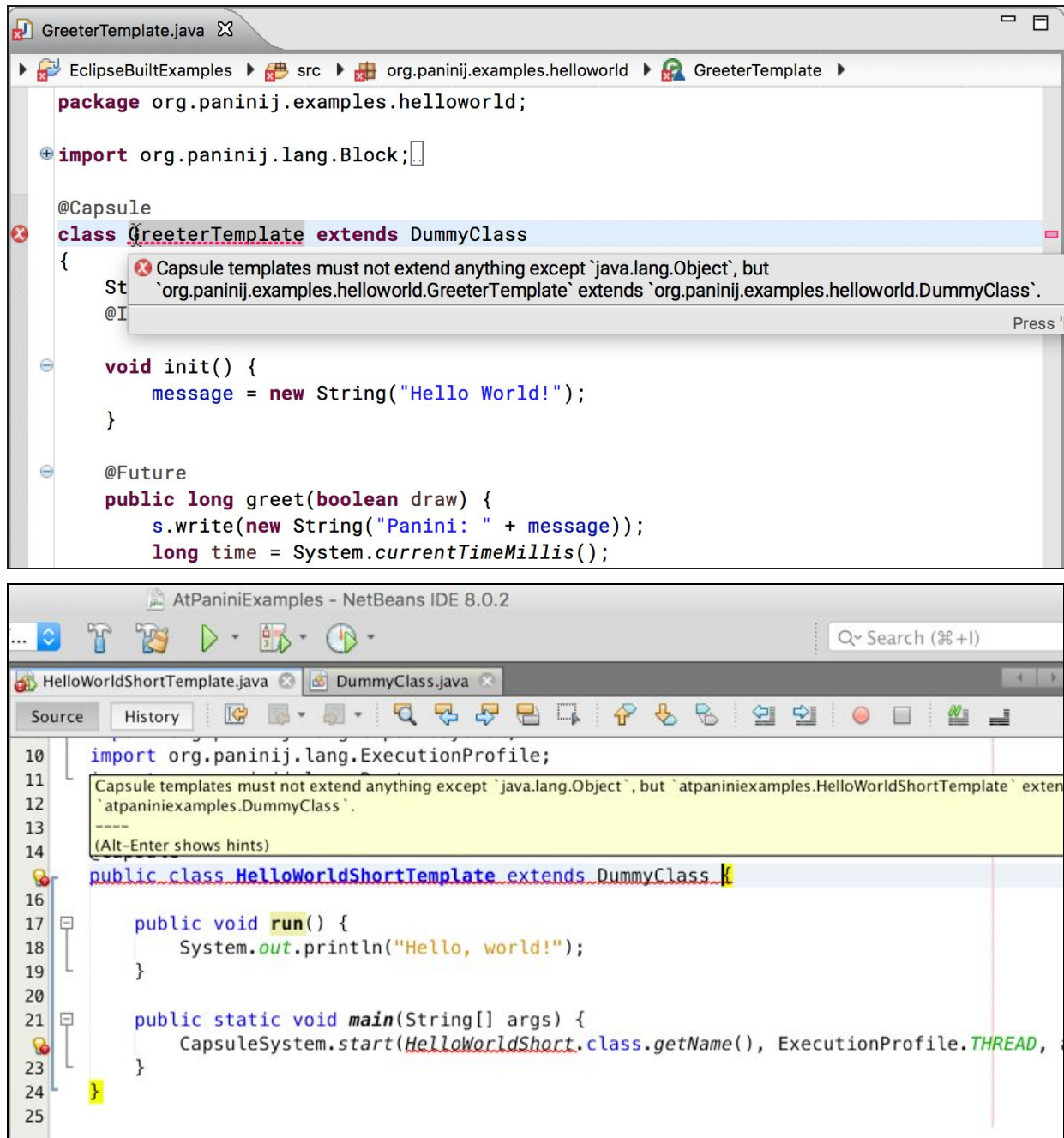


Figure 3: Using standard Java compiler methods, @PaniniJ is able to produce detailed custom errors and display them just like Java errors. This image is of @PaniniJ code in Eclipse and Netbeans that violates a tenet of the Panini model.

Section 2: Revised Project Design

The core of @PaniniJ is organized into three packages, each of which contain its own set of responsibilities:

1. `lang`: contains the annotations which `proc` consumes.
2. `runtime`: components which are necessary at runtime (e.g. thread pool managers).
3. `proc`: the annotation processor itself.

The first two, which are required for both compiling and running a capsule system, are included in the `at-paninij-runtime` JAR file. All three packages are included in the `at-paninij-proc` JAR file.

`org.paninij.lang`

This is where user-facing @PaniniJ class definitions are kept. These classes fall into two categories. The first category are the annotations which the user applies to their code (e.g. `@Capsule` and `@Signature`).

The second category arises from a technical challenge and requires a little more explanation. Many common Java classes in `java.lang` are marked `final` (e.g. `String`, `Integer`, etc.). Our system (like PaniniJ before it) cannot make a duck which mocks a class marked `final`. Therefore, procedures which return one of these types will need to return the actual type, not a transparent future. Consequently, we cannot provide the user with explicit concurrency when they use these these common Java classes cannot user with implicit concurrency.

As a workaround for this problem, we have placed reimplementations of these common types within `org.paninij.lang` types. Our package mirrors `java.lang`, but these reimplementations are not marked `final` and can thus be mocked as ducks and have implicit concurrency.

`org.paninij.runtime`

The `org.paninij.runtime` package is another major modules in our system is segmented into. This package includes the capsule interface, abstract capsule profile classes, and the interfaces and abstract classes for Duck Futures. The `Capsule` interface included in this package contains the methods that all threading profiles implement. These methods, which have analogs in the `Thread` class, include: `start`, `shutdown`, `push`, `join`, and `exit`. This interface is implemented by the abstract classes that are made for each threading profile: `Monitor`, `Serial`, `Task`, and `Thread`. These abstract classes contain the implementation details that all capsules of that profile share.

As mentioned, this package also includes the interfaces and abstract classes for Duck Futures that are utilized by the capsules to consume Ducks. There exists two types of procedure calls in the PaniniJ system: procedures with a return value and procedures with no return value. The procedures that have no return value are the baseline for our Duck Futures. Ducks based off these procedures implement the `Panini$Message` interface which is used to tie a Duck and the procedure it is based on together. The abstract class `SimpleMessage` is included to be used for the `shutdown` and `exit` calls on capsules. The capsule's run method relies on the `Panini$Message` interface to resolve the Duck Futures in its queue.

The second type of procedures, those that return values, utilize the remaining classes in the runtime package: `Future` and `ResolvableFuture`.

`org.paninij.proc`

Overview of Annotation Processing Pipeline

Java provides the ability write custom annotations and custom annotation processor. Java developers are already familiar with annotations such as `@Override` and `@Suppress`.

In order to approximate the expressiveness of the PaniniJ language, replaced PaniniJ-specific keywords with a create a handful of new Java annotations. For example, the `capsule` keyword in PaniniJ became the `@Capsule` annotation in equivalent `@PaniniJ` programs.

User-defined classes annotated with `@Capsule` are called *capsule templates*. The name follows from the fact that they serve as a template in the automatic generation of additional artifacts. If our compiler plugin is present on the classpath when when a Java compiler is tasked to compile one of these capsule template classes, the Java compiler will trigger our custom annotation processor, `org.paninij.proc.PaniniProcessor`.

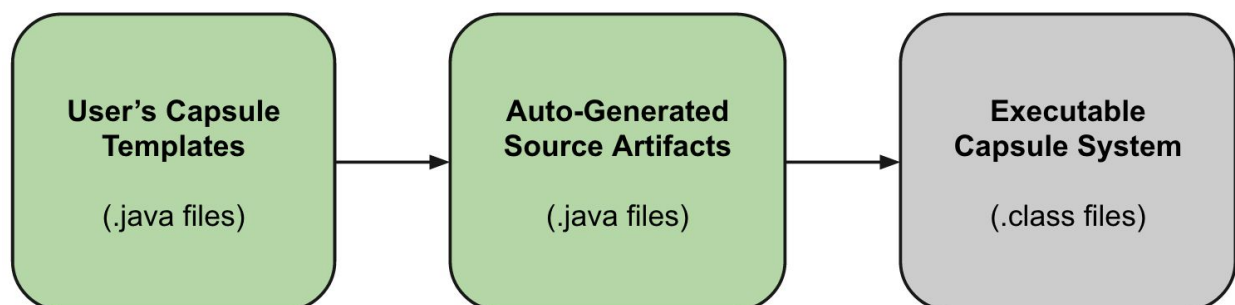


Figure 4: A high-level view of the process that converts user template code into capsule systems.

While executing, our annotation processor uses the annotation processing API to inspect the interfaces of the classes which the user has supplied. Note that this API is limited to only inspecting input classes, not modifying them. Therefore, like many other annotation processors, `proc` does its work by creating new source artifacts. These new artifacts essentially wrap the user-defined code with concurrent code according to the Panini programming model. (The technical details of the generated files are quite similar to files which would have been created by `panc`, however there are some important technical differences which arose from limitations of the compiler plugin API and restrictions induced by the Java type system.)

These generated artifacts essentially wrap the user's code to form an executable system of concurrent capsules. This lets us hide complicated and error-prone interactions between capsules hidden from the user.

Our processor emits these newly generated Java files, and they are passed along with the rest of the user's source code to be compiled. The user's classes and the `@PaniniJ`-generated classes are used together to form an executable capsule system.

Overview of Pipeline Within Annotation Processor

As discussed above, the job of the annotation processor is to take the user's capsule templates as input and to generate new concurrent wrapping classes as output. These outputs are generated in adherence to the Panini programming model and thus eliminate the possibility of certain concurrency bugs by construction.

Within this process of inspecting inputs and generating outputs are a number of stages. The following image shows the processing pipeline within our annotation processor.

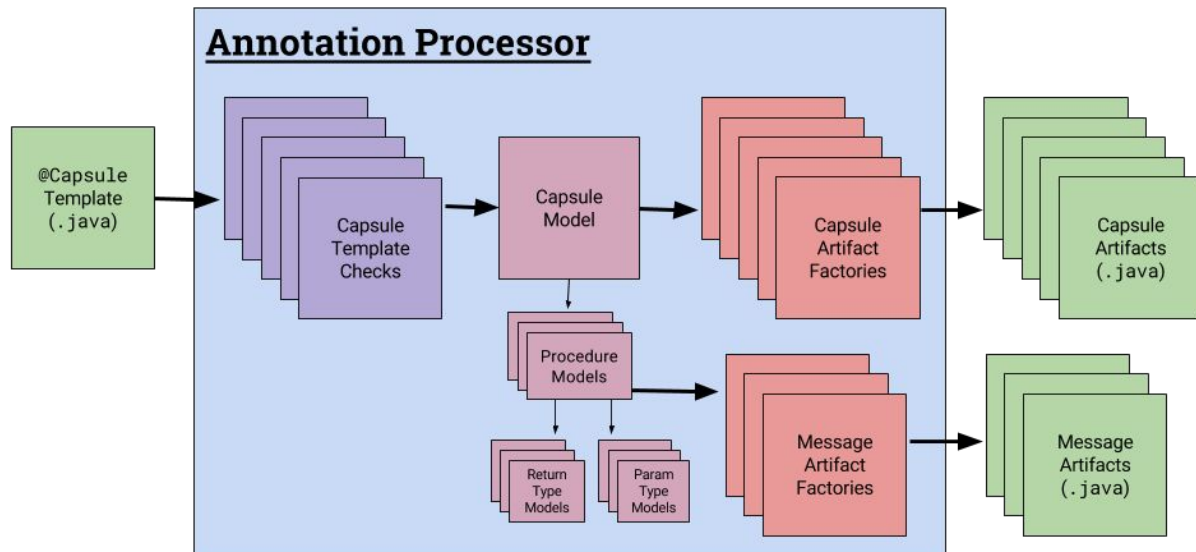


Figure 5: @PaniniJ processor pipeline within the processor itself. The user's source code is taken as input and various Panini-specific artifacts are generated as output.

The annotation processor system (a.k.a. @PaniniJ) drives all compile-time behavior. It is responsible for delegating to any necessary input validation components (e.g. CapsuleChecker) and any necessary artifact generation components (e.g. MakeDuck). It is the master control which delegates to other components.

The following subsections describe each of the components which are a part of the annotation processor system.

Sub-Component: Checks

@PaniniJ will generate errors when a user violates a some property of the Panini programming model or the @PaniniJ syntax. In all, we implemented about 45 distinct checks.

These errors are reported just as Java programmers expect: in IDE's like Eclipse and Netbeans, these errors are reported via red squiggly lines and context boxes at the point of failure; when running the compiler on the command line, error messages are printed along with line numbers.

Sub-Component: Models

Since the Panini programming model has several differences from the Java programming model, we decided it would be the most flexible to translate components of the Java model (classes, methods, interfaces, etc) into components of the Panini model (capsules, procedures, signatures, etc). With this abstract-syntax-tree like structure of Panini components, we are able to put the artifact generators in terms of the Panini programming model. Additionally, this structure could be created programmatically.

Sub-Component: Capsule Artifact Generator

User classes which are annotated with `@Capsule` are called Capsule Templates. Capsule Templates define a capsule to be generated. The Capsule Artifact Generator SubComponent creates Capsules based on the user-created template. The Capsule Artifact Generator will create four different runtime profiles (Thread, Monitor, Serial, and Task) for each capsule. These generated artifacts are named accordingly:

- `CapsuleName$Thread.java`
- `CapsuleName$Monitor.java`
- `CapsuleName$Serial.java`
- `CapsuleName$Task.java`

The Capsule Artifact Generator also tells the Duck Future Artifact Generator SubComponent which ducks will need to be generated.

Sub-Component: Message Artifact Generator

`@PaniniJ` uses several distinct methods to allow capsules to invoke procedures on each other. It is necessary to have these different methods because the primary method, duck futures, are unable to support procedures due to properties of their return type. We also use explicit futures via the `Future` interface in `java.util.concurrent` package of the standard Java API. Finally we allow a procedure to have blocking behavior where the calling capsule halts execution until the result is returned. A user specifies the behavior of a procedure by using the provided annotations: `@Duck`, `@Future`, and `@Block`.

Each of these methods extends the abstract class `MessageFactory` to produce the version of the message artifact requested by the user's template.

The `DuckMessageFactory` handles the production of the implicit duck futures. These duck futures are an essential part of making synchronous method calls act as asynchronous procedures invocations by acting as invisible futures which encapsulate the results of the procedure while still allowing the capsule continue executing.

The core functionality of duck futures remain the same in this system as it is in `panc`. `@PaniniJ`'s duck system differs in an attempt to reduce the number of duck classes generated by allowing "duck shapes" to be shared by different capsules. In the `panc` implementation, ducks were generated by capsule name and procedure return type. Our implementation instead creates ducks based on the return type and parameter types.

Additionally, any object (i.e. non-primitive) arguments (e.g. `String` or `BufferedReader`) are cast to an `Object` when they are stored in a duck future. This abstraction again reduces the number of ducks which need to be generated. When the duck is consumed by a capsule's

run() method, the abstracted parameters are cast back to their original types and passed into the correct method of the stored instance of the template class.

An example of the ducks generated by both systems follows:

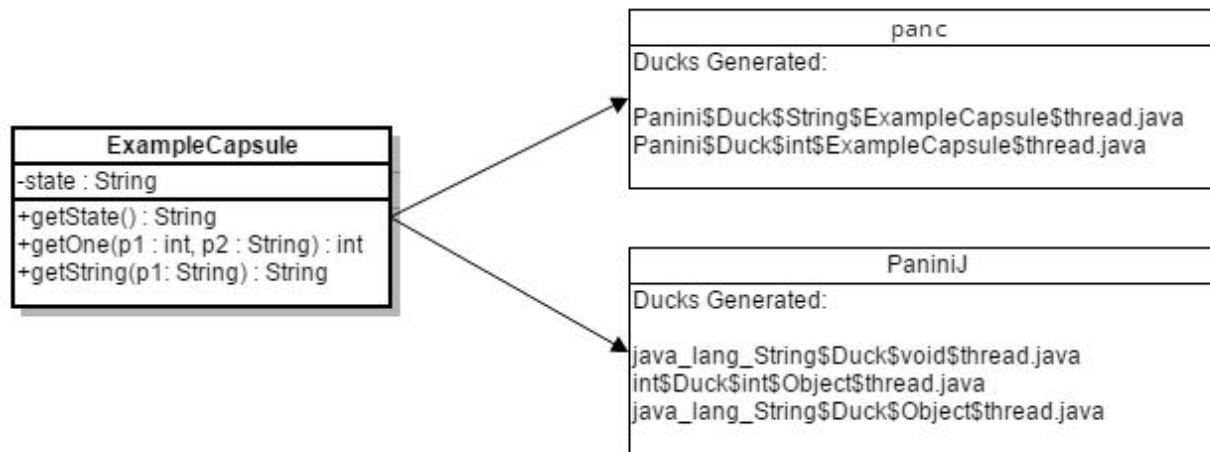


Figure 6: Duck Artifacts: Duck Generation using PaniniJ's `panc` vs. `@PaniniJ`.

In addition to ducks, we also allow for explicit futures to be used within a capsule system with the `@Future` annotation. Explicit futures are the standard method of calling java methods asynchronously in traditional concurrent java programming. Explicit futures do not have the restrictions that duck futures have with return types and yield better performance than blocking behavior. The downside to this method is that the user must handle the unpacking of the future instead of the panini system doing it for them.

Finally, `@PaniniJ` allows developers to specify blocking behavior for procedures with the `@Block` annotation. This behavior has no restrictions on return type, but has a performance downside as the calling capsule must wait for the result before it can continue executing. A potential application of the blocking behavior is when a user wants to intentionally synchronize the order of procedure calls.

The Java source files generated by these different capsule and message factories produce artifacts which are then combined and ran through the standard java compiler. The resulting class files represent the concurrency safe capsule system that the user described in their annotated templates.

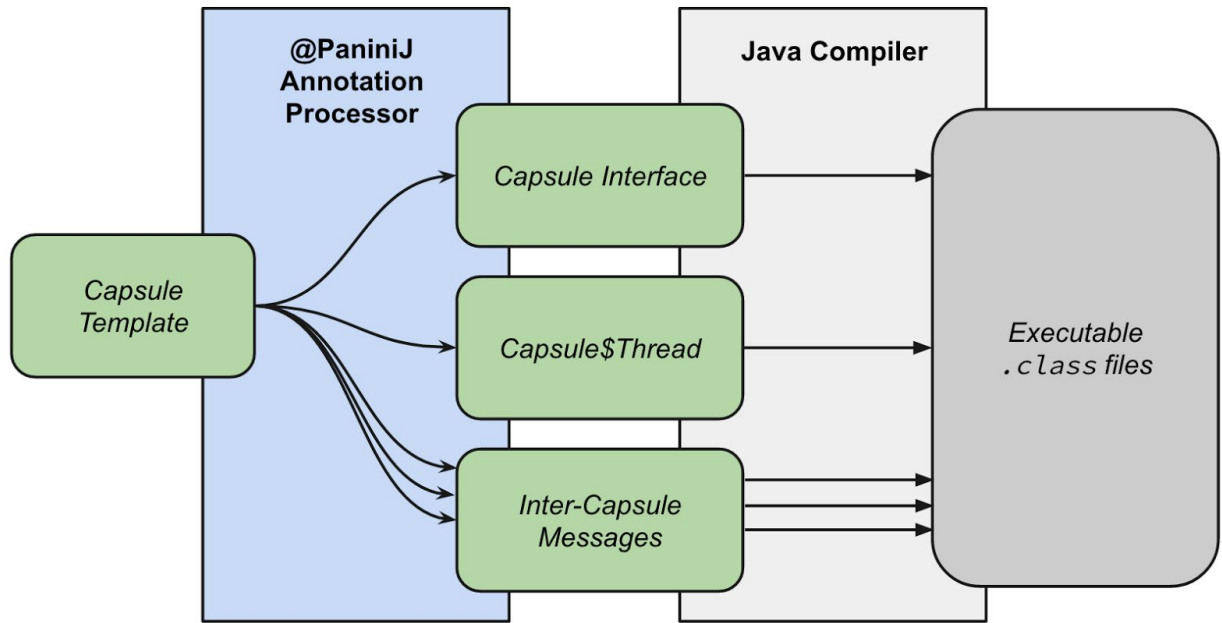


Figure 7: Example flow between user written code, generated code, and an executable capsule system.

Section 3: Implementation Process Details

We used the Rapid Application Development method.¹ This involved creating many prototypes that tackle small problems instead of doing a lot of up-front planning. The lack of up-front planning was suitable for this project since it is a small portion of a larger ongoing research project. We expected that the requirements of the project would change frequently and without warning.

Additionally, the technologies we will be used were completely new to us (e.g. annotation processing and potentially, pluggable type checkers); as such, too much up-front planning might provide too optimistic a view of the strength and appropriateness of these tools within the design. With rapid development of prototypes we became familiar with the capabilities of the new technologies without committing to a single plan and then gauged their appropriateness as we went.

Throughout the development of this project several design refactorings occurred in order to bring the smaller features and prototypes together in a way which best met the project's goals, constraints, and requirements, as well as the needs of the project's stakeholders.

Since features and prototypes developed with the Rapid Application Development method are somewhat independent, it was necessary that multiple people viewed code before it became a part of the current design. To accomplish this, we are used pair programming as often as possible and also used tools such as git² and github³ to manage pull requests and perform code reviews.

¹ http://en.wikipedia.org/wiki/Software_development_process#Rapid_application_development

² <http://git-scm.com/>

³ <https://github.com/>

Section 4: Testing Process and Testing Results

Developing tests for our annotation processor was very important, since our project will be continued on by the ISU Laboratory for Software Design after we are finished.

While developing the artifact generation code, we were able to perform a good deal of simple testing by simply writing input @PaniniJ programs and running our generated outputs through the Java compiler. We could then run these capsule artifacts to see whether they behaved as expected. However, automatic execution of these test capsule systems has not yet been added to the project.

This strategy works for capsule systems which follow all of the @PaniniJ syntax rules, but we also needed to check that all of our capsule checks worked. For this we needed to check that malformed inputs reported appropriate errors.

To test each of these checks, we used JUnit tests to start the annotation processor with some malformed input. The JUnit tests only pass when a panini check fails.

Finishing the 0.1.0 release of the @PaniniJ processor was our final goal for development on this project. It was imperative that the static checks of user code functioned properly as it is a primary tool for new developers to learn a new system. It was also requisite that our code generation produced valid, concurrently safe Java source code. We made a point of writing a unit test for each of the static checks that would clearly show whether they are working as anticipated which has allowed us to validate that that portion of the user experience is functioning.

We have also been maintaining a suite of example @PaniniJ projects that are run through our processor and then through the Java compiler. The pass through the Java compiler ensures that our generated code passes static type checking and will alert us to any problems with malformed generated source code. As of the release of 0.1.0, both prongs of our testing method were fully functioning and producing positive results.

Appendix I: Operation Manual

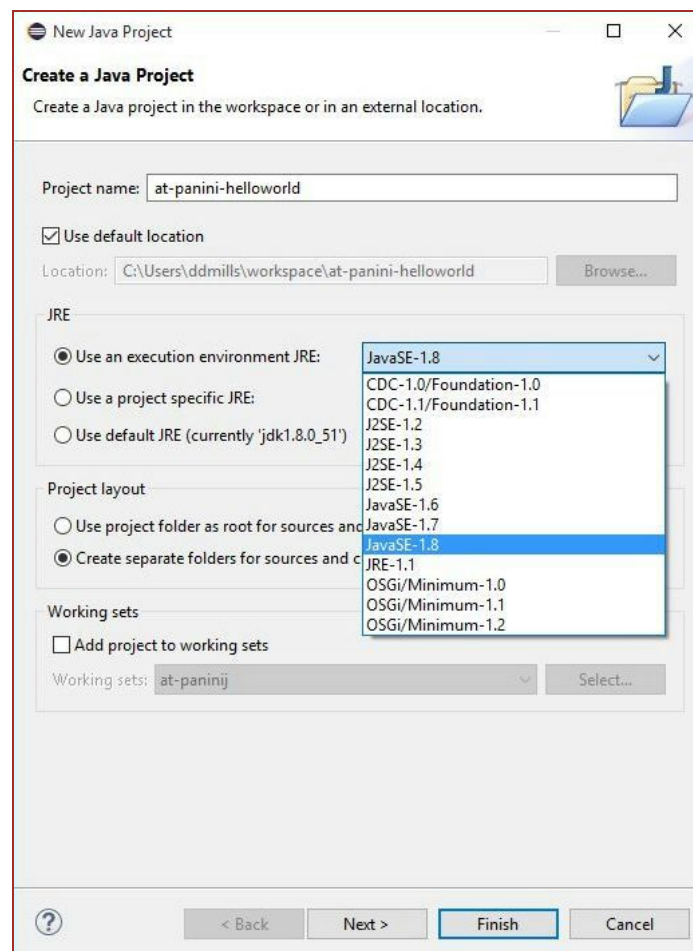
Operation Manual Overview:

These are the overall steps to setting up an @PaniniJ project in Eclipse:

1. Create an Eclipse Project
2. Download the @PaniniJ jar
3. Enable annotation processing
4. Add at-paninij annotation processor to your build
5. Add the at-paninij jar as a referenced library

1 - Setup the project to use JRE 1.7 or greater

When you create a new project, be sure to choose Java Runtime Environment 1.7 or greater, this is necessary for the annotation processing to work correctly.

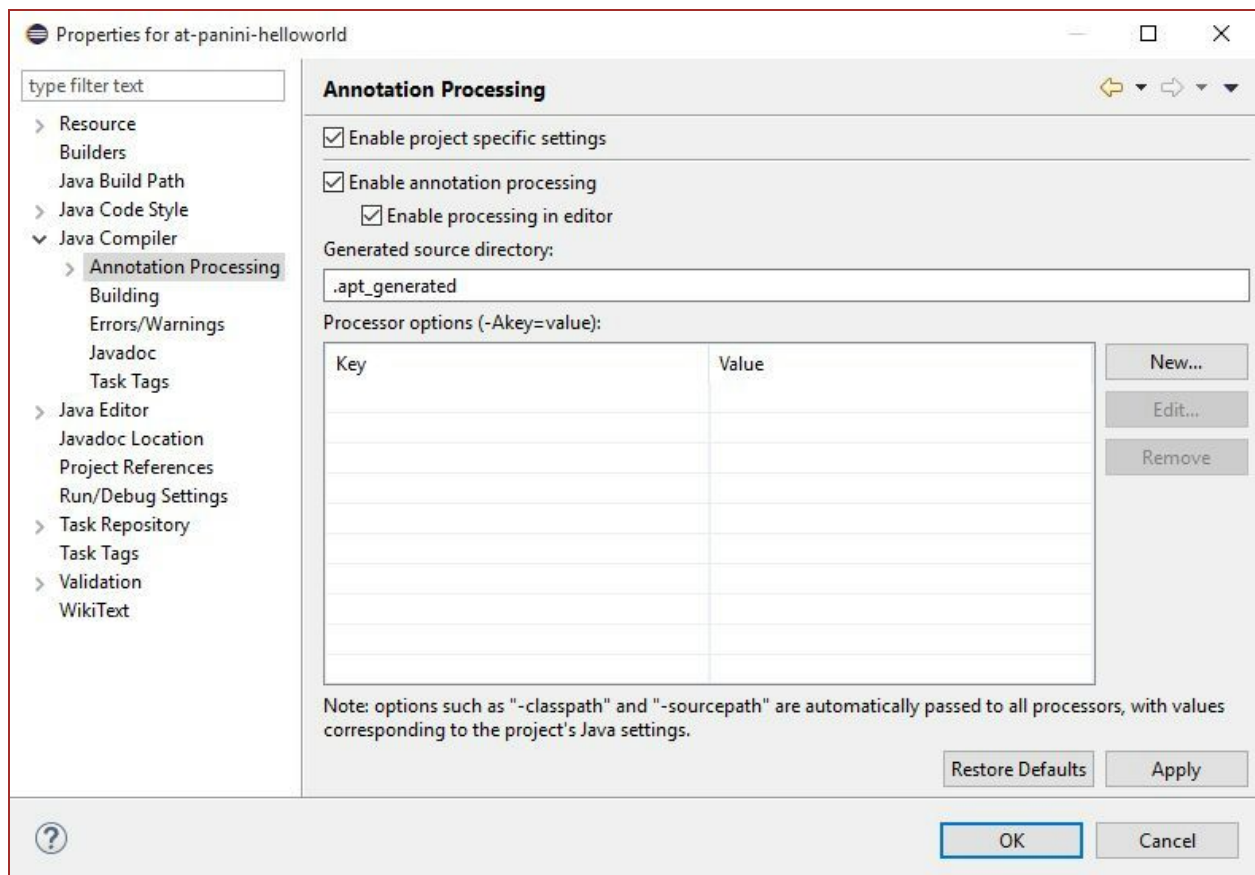


2 - Download the at-panini.jar

Download the latest processor (at-panini-jar-v0.1.0.jar) from the github releases page: <https://github.com/hrides/panini/releases>.

3 - Enable annotation processing

Enable annotation processing by right clicking on your project in the project explorer and choosing "properties." Browse to Java Compiler > Annotation Processing and check the "Enable project specific settings" checkbox and "Enable annotation processing".

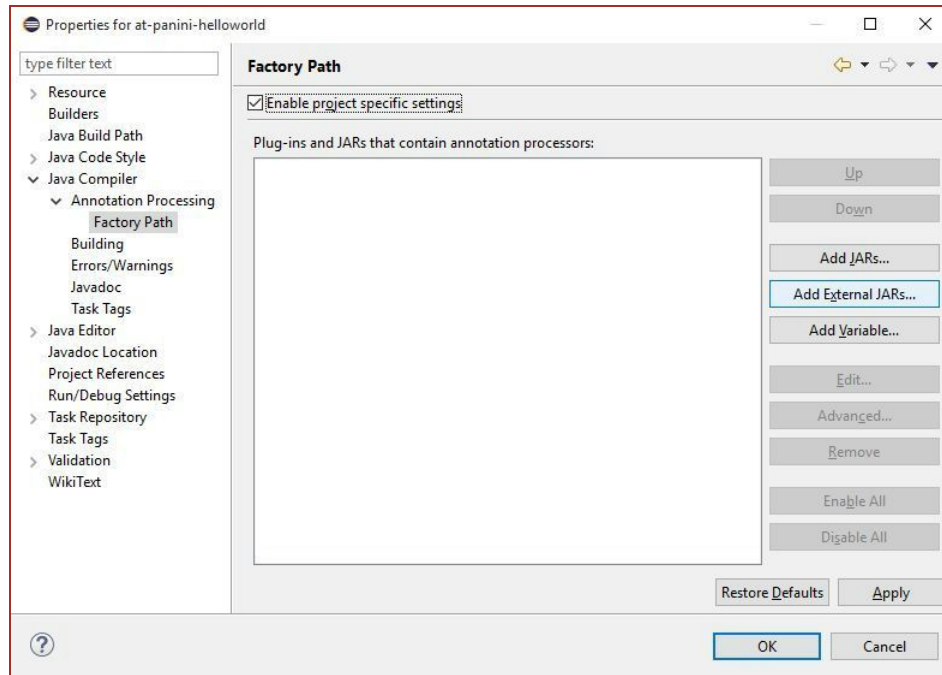


Once you hit Apply, Eclipse will inform you that a rebuild on the project is required. You can click yes to rebuild the project now.

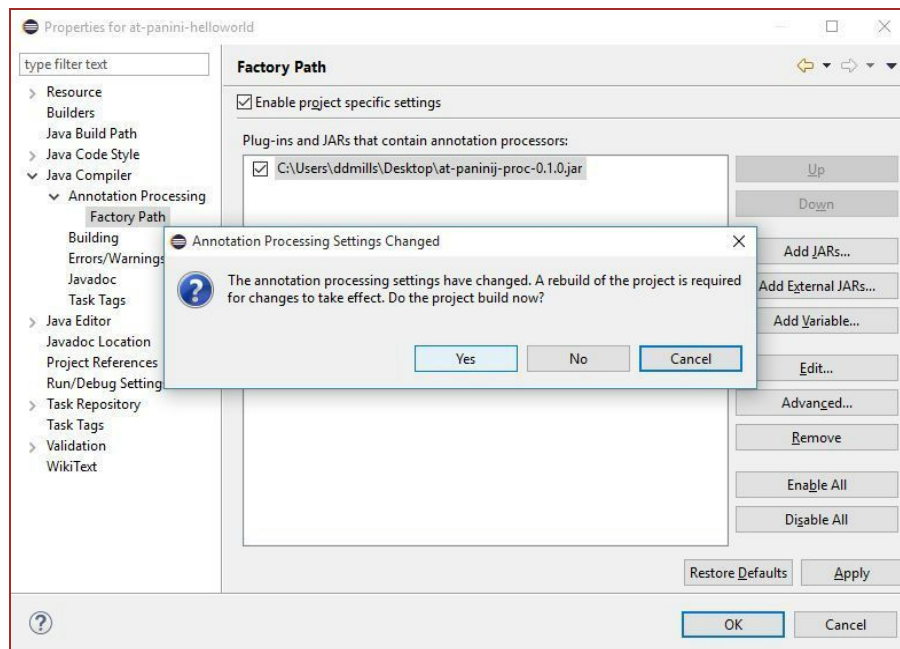
Note that the "Generated Source Directory" is where the sources that the annotation processor automatically generates will be stored. You can remove the "." from ".apt_generated" and it will become visible in Eclipse.

4 - Add at-paninij annotation processor

Navigate to the Factory Path section of the project properties. It is beneath the Annotation Processing option. Check the Enable project specific settings checkbox, and click the “Add External JARs...” button.

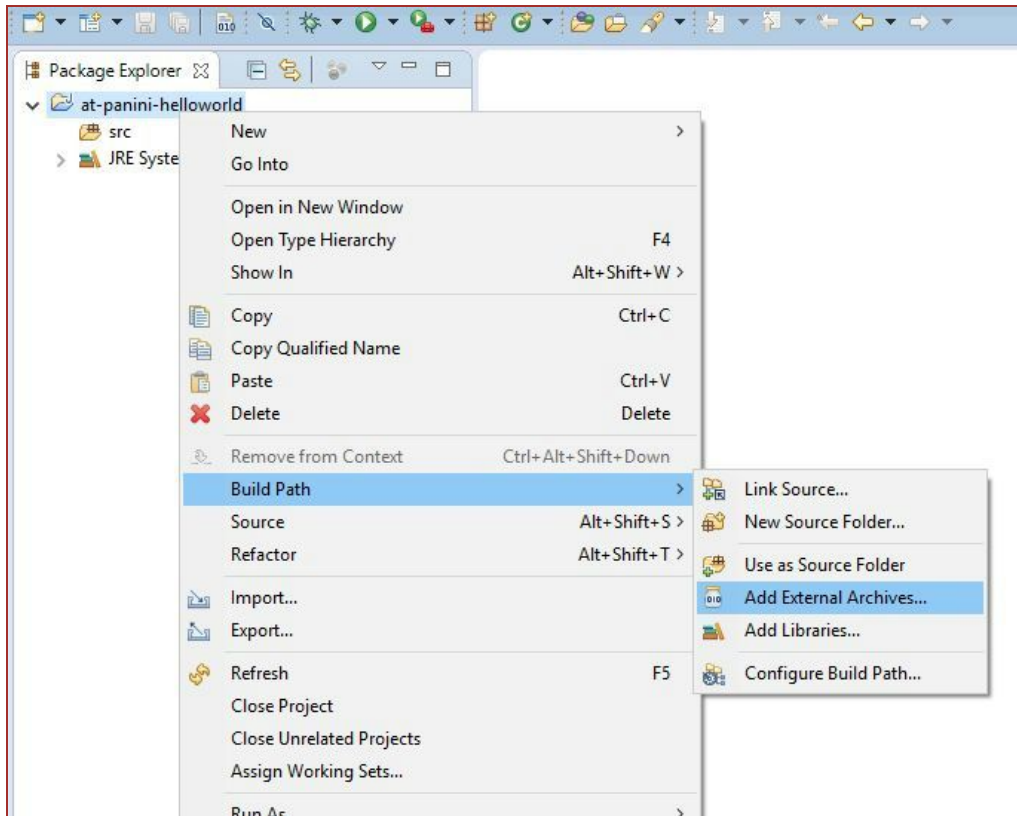


Browse to where you have downloaded the JAR file from step 2. Hit Apply and confirm the project rebuild.

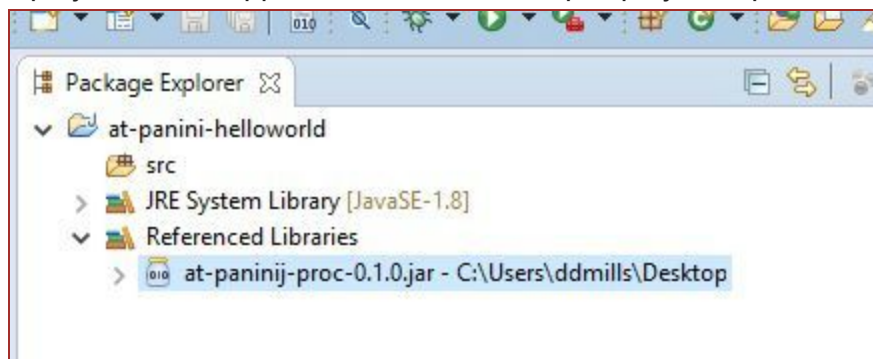


5 - Add the at-paniniJ as a referenced library

The @PaniniJ JAR file includes code necessary for the annotation processing and runtime. To include the JAR file as a referenced library, right click the project in the project explorer, and choose "Add External Archives...".



Browse to where you downloaded the JAR file from step 2, and include it in your project. Once it is included, the project should appear like this in the Eclipse project explorer:



Appendix II: Alternative Designs

Alternative: Eclipse Plugin

As mentioned in our introduction, the original client specification for this project was to develop an Eclipse plugin for the PaniniJ language. This plugin would enable syntax checking, code completion, and other standard IDE features. We explored this route and researched implementations using the XText framework which is used to define language grammar for Java-like languages and provide the bridge to Eclipse's Java features.

We met with another member of the client's lab who had been working on a similar project for the Boa language which gave us practical insight on the nature of this version of the project. It was not long before we realized that this plugin would need to be constantly maintained when the version of Eclipse, XText, or Java was updated. These maintenance challenges influenced our client and our team and ultimately led to this version of the project being altered.

Alternative: panc Interoptibility

We also had a version of the project that had strict interoability for the existing PaniniJ compiler. This required our outputs to match up exactly with those of the PaniniJ system. We ended up abandoning this requirement as we could not reproduce certain behaviors of the PaniniJ compiler within the framework of our annotation processor.

One example of a behavior we could not replicate is the mangling of user's source code. We did not have a view of the user's code that would give us line by line parsing but instead were limited to a view based on elements that would be directly and immutable linked to classes, methods, parameters, and state variables. In short, we could not alter the contents of a user's code in arbitrary ways like the PaniniJ system could.

This ended up being positive as the restriction enabled the use of standard Java debugging tools which provided functionality that was better than our initial specification. It also marked the time when we accepted that @PaniniJ would be a full replacement of the PaniniJ compiler.

Appendix III: Other Considerations

Package Structure

The naming conventions for the project are adopted from the system architecture; there is a direct mapping between package names and the names of systems, components, and subcomponents. With the exception of auto-generated classes, all @PaniniJ code is a subpackage of the `org.paninij` package.

Naming Conventions

Many of the class names in the project are delimited by a dollar sign (\$). These describe classes that are auto-generated or do the generating of said classes. Auto-generated classes need this in order to prevent collisions with the user's code (since the auto-generated classes are kept in the same package as the user's code). Additionally, many of the variables and method names on the generated classes start with `panini$`, this is again to prevent collisions with code written by the user.

Coding Guidelines and Conventions

The @PaniniJ codebase uses a slight modification of the standard Java code conventions. Any modifications, such as placement of return carriages before entering the body of a method, have been retained as artifacts of the original PaniniJ code conventions.

Lessons Learned

As a group, we were extremely satisfied with our final product. We do not think that the level of polish and detail that we have accomplished would be possible if it weren't for the consistency with which we met as a group (also with the client). We knew what each person was responsible for, held each other accountable and also did a lot of pair programming. To summarize, we believe that communication and consistency was key to accomplishing our goals.

Appendix IV: FAQ

Why didn't we just make better tooling for PaniniJ? Why didn't we just make an Eclipse Plugin?

The stated goal of the project is to make tools which make Panini capsule systems more accessible to programmers. This could have been achieved by making better tooling for PaniniJ.

However, we also believe that it may be worthwhile developing an annotation processor solution for a number of reasons. In particular, an annotation processor is likely much easier to develop and maintaining than the existing implementation of `panc`, a fork of the entire `Sun javac` compiler.

Furthermore, though `panc`, is an extension of the standard Java compiler, PaniniJ code is not easily integrated into existing Java projects. Our project, may make Capsule-Oriented Programming more usable in Java project than the existing PaniniJ tools can provide.

Note that there may be certain features that PaniniJ/`panc` provides, which our solution cannot provide, for example, certain code analyses and safety checks. However, these features are currently outside the scope of this project.

Why did we make the capsule declarations native Java classes?

This decision allows the user to use many existing Java tools when developing a Panini capsule system. Additionally, Java programmers can start making capsule systems without learning a new programming language, PaniniJ.

Why perform Java source generation?

The boilerplate code for making capsule-like entities is tedious and error prone, despite being highly a relatively regular translation process. We want to remove the boilerplate by providing a standard model which can be verified and tested. By using Java source generation, we can generate a layer of code that includes the boilerplate based on source code provided by the user.

Why use an Annotation Processor for source artifact generation?

Using an Annotation Processor gives us a detailed look at the user's source code through the Java standard library, `javax.api.model`. This library provides the tools to analyze Java source code which pairs with the Annotation Processor's ability to hook into specific sections of the source code. Together they provide a system of source analysis that does not require us to write a Java interpreter which would be of lesser quality compared to the Java standard API libraries.

Appendix V: Glossary

<i>Artifact, also Source Artifact, Generated Artifact</i>	A Java source code artifact created by @PaniniJ. Important kinds of artifacts generated include capsule classes and duck classes.
<i>Artifact Generation</i>	The process by which @PaniniJ processes a set of user-defined template classes and automatically generates/creates derived artifacts.
<i>Capsule</i>	An actor-like software construct defined by the Panini model which <ul style="list-style-type: none">● uniquely owns its state variables,● provides a set of procedures which can be invoked, and● has an execution profile by which computations of invoked procedures are performed.
<i>Capsule, Local</i>	A capsule declared within the definition of another capsule. Note that each design argument of some capsule C is not considered a local capsule of C (though they may be a local capsule of some other capsule). Aside from the root capsule, all capsules within a system are local in exactly one other capsule (its parent capsule).
<i>Capsule, Imported</i>	A capsule declared within the definition of another capsule which is imported (a.k.a. wired-in) from the capsule's parent.
<i>Capsule, Passive</i>	A capsule without a user-defined <code>run()</code> declaration.
<i>Capsule, Active</i>	A capsule with a user-defined <code>run()</code> declaration.
<i>Capsule, Root</i>	A capsule which has no dependencies and can serve as the entry point for a capsule system. Usually designated with the @Root annotation.
<i>Declaration, design()</i>	Where the user defines the set of design arguments and specifies what are to be wired to it's child capsules.
<i>Declaration, init()</i>	Where the user defines initialization code for a capsule's state variables.
<i>Declaration, run()</i>	Where the user defines custom run behavior for a capsule. If a capsule has a run declaration, it is called an active capsule. Otherwise, it is called a passive capsule.

<i>Declaration, Signature</i>	Analogous to a Java interface, except it applies to capsules.
<i>Capsule Imports</i>	The set of capsules and objects which must be passed to a capsule via its <code>imports()</code> method in order for that to be instantiated properly.
<i>Execution Profile</i>	The mechanism or policy by which a capsule's procedure invocations are processed. For example, in the case of the thread execution profile, procedure invocations are submitted to a queue and processed one-by-one by that capsule's own dedicated thread.
<i>Future</i>	A thread-safe object/class which represents a result of a task. We say that a future is resolved when the task is complete and the result is ready to be used. If a thread tries to use this result before it has been resolved, then the thread will block until it is resolved.
<i>Duck Future (a.k.a. Transparent Future)</i>	An object/class which is a mockup of one of the user's objects/classes but also acts as a future, resolvable by the panini runtime. This is the key to enabling implicit concurrency.
<i>Method</i>	A regular Java method. (This is distinct from the Panini concept of a procedure.)
<i>Method Call</i>	A regular call to a Java method. (This is distinct from the Panini concept of procedure invocation.)
<i>Panini</i>	The abstract programming model which defines the semantics of a system of interacting capsules.
<i>PaniniJ</i>	A research language similar to Java which adds support for the capsule-oriented programming as defined in the <i>Panini</i> programming model. See: http://www.paninij.org
<i>@PaniniJ</i>	The system described in this design document.
<i>Procedure</i>	A panini analog of a method. A procedure is the user-defined code on a capsule's interface which can be invoked (i.e. called), potentially by other capsules or other threads. Arguments can be passed and an object can be returned. Importantly, the returned object can be a duck future.
<i>Procedure Invocation</i>	A panini analog of a method call. (See <i>Procedure</i> .)
<i>Shape</i>	A description of a method's return and argument types. This is essentially the information in a method signature aside from its names. By extension, we also say that procedures have shape.

<i>Signature</i>	A Panini analog of a Java interface. Each signature specifies a set of procedures. In order for a capsule to implement a signature, it must have a definition matching the shape and name of each procedure in that signature.
<i>State Variable, also state</i>	A Panini analog of an instance variable on a Java object. A state variable is a variable attached to a capsule instance. They can only be accessed and modified by the init() declaration and procedures of the capsule which owns them.
<i>System Topology</i>	The structure of a network of capsules.
<i>Template Class</i>	A Java class annotated with either @Capsule or @Signature which specifies the elements of a capsule or signature, respectively. For example, some elements which a capsule template class is used to define are the procedure definitions, the define() declaration, and child capsule declarations. It is from processing a set of template classes that @PaniniJ generates a set of source artifacts.
<i>Wiring</i>	The process of initializing a system of capsules with references to one another according to the user-defined system topology.