@PaniniJ

An annotation-based realization of Panini capsule-oriented programming.

Dr. Hridesh Rajan
Project Advisor & Client

Dalton Mills
Webmaster

David Johnston
Team Lead

Trey Erenberger
Key Concept Holder

Introduction

Problem: The PaniniJ language does not work with existing Java tools.

Need: The PaniniJ language needs better tooling for better developer experience.

Solution: Re-implement PaniniJ as an annotation processor to make the language more familiar to users and more compatible with existing Java tools.

@PaniniJ is an implementation of the Panini programming model that enables the use of standard Java tools for capsule-oriented programming. Panini is a programming model (i.e. a set of rules for program behavior) in which certain classes of concurrency errors (endemic to other programming models) are not possible. This programming model has been the subject of years of research and development here at Iowa State.

Our project is a replacement for the existing implementation of the PaniniJ language (the existing implementation of Panini). PaniniJ is essentially a fork of the Sun/Oracle Java compiler which includes some additional Panini-specific keywords, syntax, and semantics.

We implemented an annotation processor, a standard method of hooking into a standards-compliant Java compiler, to generate concurrent code based on user-written templates. What were keywords in PaniniJ are now Java annotations in @PaniniJ.

This decision allows the user to write code for the Panini model while still allowing him/her to use standard Java tools.

Unlike its predecessor, @PaniniJ is not a fork of the Java compiler, so maintainers do not need with changes to the Java compiler and language. Additionally, with @PaniniJ, the user is able to choose his/her own Java compiler implementation, rather than being forced to use the PaniniJ compiler for all of their project's code.

Design Approach

Java annotation processing is a linear and relatively restrictive interface for compiler plugins. Our annotation processor takes in Java classes annotated with @Capsule and produce additional artifacts. These generated artifacts essentially wrap the user's code to form an executable system of concurrent capsules. This lets us hide complicated (thread-safe) interactions between capsules hidden from the user.

Before we generate these artifacts, we also perform some basic checks on and processing of the user's input. This process of taking in Capsule Templates and generating wrappers and messages occurs every time the compiler is called. In Eclipse, this is done every time the user saves updates to a file.

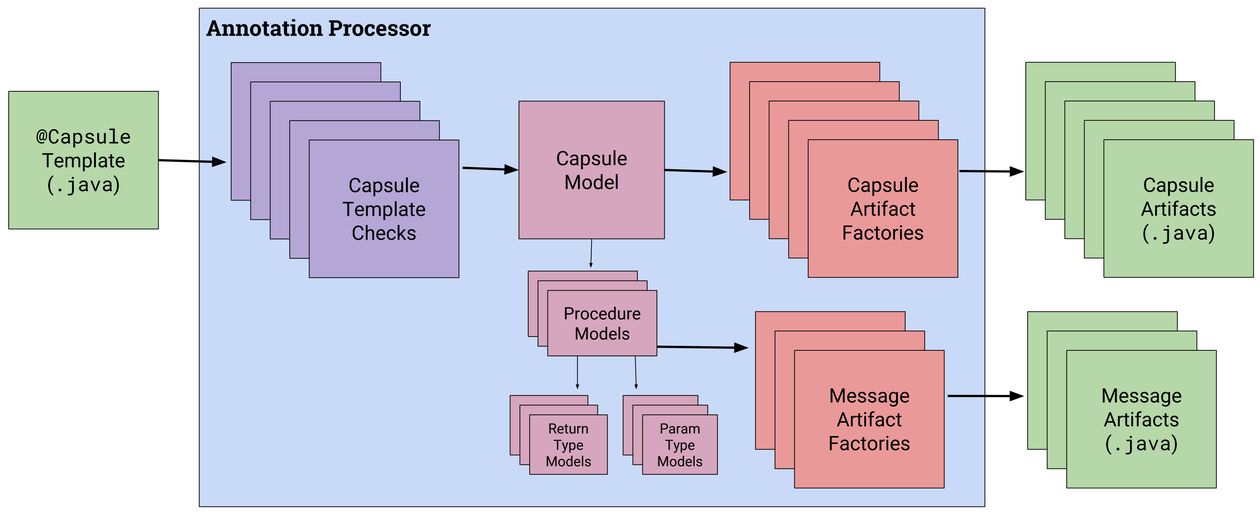

Figure 1: Processing pipeline within the @PaniniJ annotation processor. Within the processor, there are three primary phases: (1) user inputs are checked for Panini-specific errors; (2) a model is created; (3) concurrent Java source code is generated based on the model.

Design Requirements

Our project provides a maintainable way to allow capsule-oriented programs to be developed using standard Java tools while still giving the user specialized feedback to help them adhere to the Panini model.

- Provide a way to create a working concurrent capsule system without having to manually write any synchronization code.
- Generate artifacts for four basic execution strategies.
- Provide analysis of correctness (based on Panini model).
- Solution must work with standard Java toolchains.
- Capsule syntax must be simple and declarative.
- Provide meaningful feedback to user when they're doing something against the Panini model (see Figure 2).

Technical Details

@PaniniJ was developed using Eclipse, Maven, and standard Java libraries. We purposely avoided adding 3rd party code dependencies as maintainability was one of our primary concerns.

We mainly utilized the standard Java package `javax.lang.model` which includes the sub-packages element and type within our annotation processor.

We split the project into three main core modules: the processor, the runtime, and tests for the processor. This distinction allows for a smaller runtime jar to be easily coupled with deployed projects.

We also had modules for spin-off projects such as benchmarks, analyses, and examples of the @PaniniJ jar being used. These modules are not needed to run or use the @PaniniJ product itself.

Intended Users & Uses

@PaniniJ is for programmers who want to have implicitly concurrent code. The capsule-oriented model is especially useful for people who care about modularity and thread safety.

@PaniniJ was designed from the beginning to be approachable by Java programmers. Because it is packaged as an annotation processor, @PaniniJ is easy to add to any Java project, either by adding a jar file and enabling the processor, or by including it as a Maven dependency.

@PaniniJ will also 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.

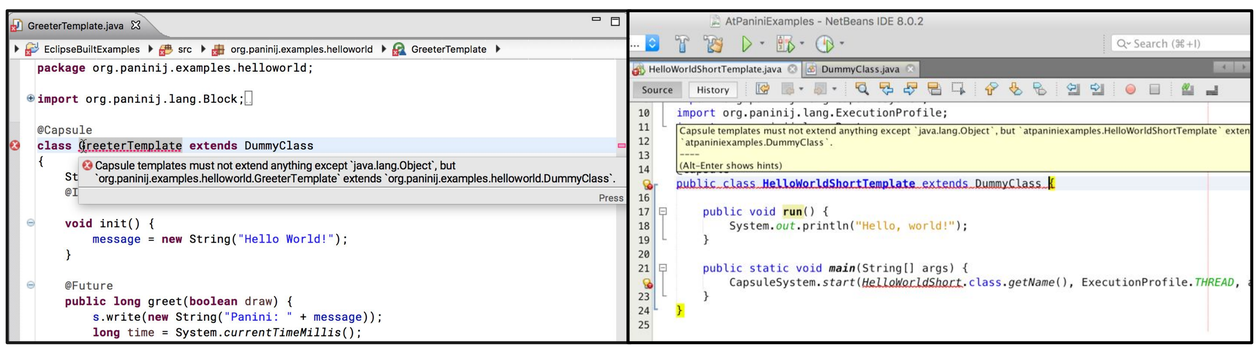

Figure 2: User error reporting in both Eclipse and Netbeans. When using these IDEs, Panini-specific syntax errors are reported just as Java errors are reported, with red squiggly lines and context boxes near the error's source.

Testing

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 testing 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 capule 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.