Software Design Specification

Team Dec 15-12: @PaniniJ
April 25th, 2015

Advisor	Dr. Rajan	
Client	Dr. Rajan	
Team Members	Dalton Mills	*Webmaster*
	David Johnston	*Team Lead*
	Trey Erenberger	*Key Concept Holder*

Table of Contents

1. Introduction to the SDS

The purpose of this Software Design Specification (*SDS*) is to provide a top-down view of the design and implementation of @PaniniJ. To supplement this approach, a glossary of important terms, acronyms, and abbreviations is included, as well as a list of relevant background resources with direct links to each where available.

The *SDS* touches on the ways in which @PaniniJ differs from and supplements PaniniJ, and how @PaniniJ fits into the larger Panini Project.[1][2] Described within is the overall system architecture of @PaniniJ, followed by the subsystem architecture and relationships between components, and finally as necessary a detailed design of each system and subsystem.

The section on System Architecture provides a high-level overview of the division of responsibilities and functionality of @PaniniJ into subsystems and components. In certain cases a more detailed description of an individual subsystem or component will be included in the subsequent section on Detailed System Design.

The intended audience of this document includes developers of @PaniniJ and the PaniniJ language, including but not limited to the Panini Project research group headed by Dr. Hridesh Rajan and researchers at other institutions.[3]

There are no current @PaniniJ version numbers as the majority of work thus has been researching design direction creating prototypes in order to understand PaniniJ and what is required to accomplish the goals of the project.

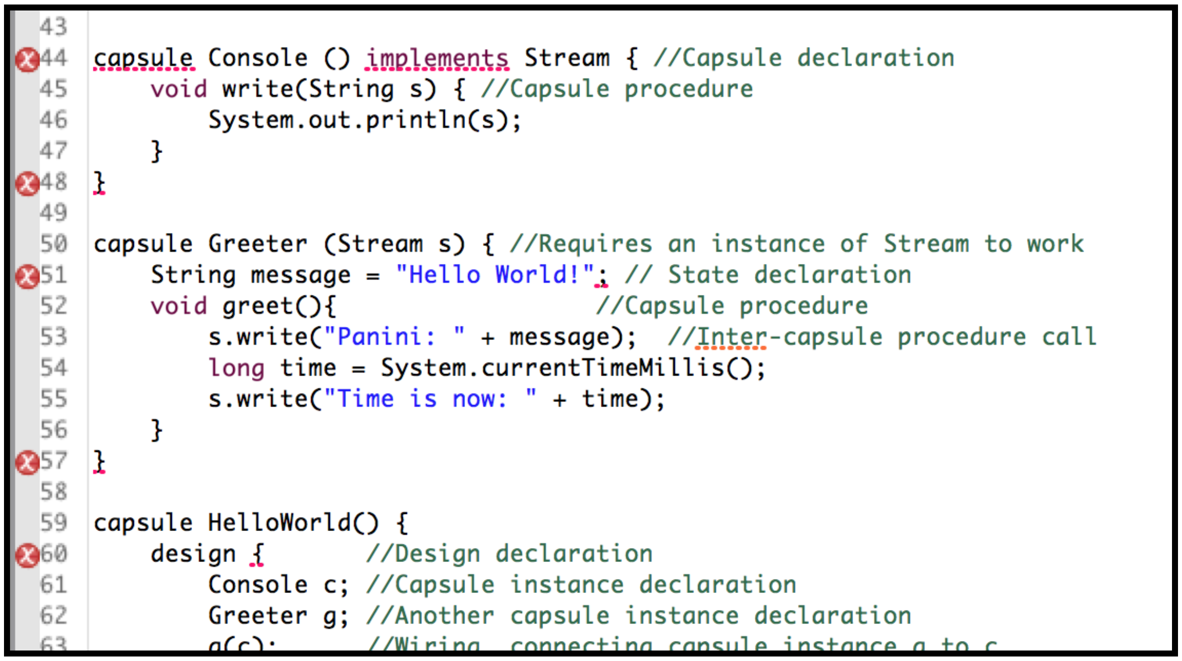
[1] The Panini programming language.
[2] More information on PaniniJ and The Panini Projet can be found at http://www.paninij.org/ (March, 2015).
[3] Dr. Rajan's professional homepage can be found at http://www.cs.iastate.edu/~hridesh/.

2. Design Considerations

2.1. Prior Work

@PaniniJ is a framework for generating capsules systems which follow the Panini programming model. There already exists a compiler for (among other things) generating Panini capsule system from the PaniniJ language. 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.

For example, consider the screenshot below. It shows an excerpt of a valid PaniniJ program viewed from within the Eclipse IDE. Though this is valid PaniniJ code, the IDE shows a number of very unhelpful errors.



```

capsule Console () implements Stream { //Capsule declaration
    void write(String s) { //Capsule procedure
        System.out.println(s);
    }
}

capsule Greeter (Stream s) { //Requires an instance of Stream to work
    String message = "Hello World!"; // State declaration
    void greet(){                    //Capsule procedure
        s.write("Panini: " + message);  //Inter-capsule procedure call
        long time = System.currentTimeMillis();
        s.write("Time is now: " + time);
    }
}

capsule HelloWorld() {
    design {        //Design declaration
        Console c; //Capsule instance declaration
        Greeter g; //Another capsule instance declaration
        g(c);       //Wiring, connecting capsule instance g to c
```

Image 2.1: Excerpt of a valid PaniniJ program viewed from within Eclipse IDE.

Any existing Java tool (such as Eclipse) cannot correctly interpret and handle almost any PaniniJ program. Importantly, Java compilers fail to correctly interpret *capsules* and thus fail to generate code for them.

Currently, there is very little tooling used in the development of PaniniJ programs. Generally, a PaniniJ programmer needs to use a plain text editor and manually invoke the command-line `panc` program, the custom PaniniJ compiler.[4] This is a far less

[4] Additionally, there is no practical way to use a debugger to analyze a PaniniJ program.

usable development environment than most modern programmers have come to expect.

2.2. Primary Design Goal

@PaniniJ is an alternative, re-engineered system for generating Panini capsule systems. Unlike the existing PaniniJ/`panc` method, @PaniniJ is designed to be used with most standard Java development environments and toolchains (e.g. Ant, Maven, Eclipse and `javac`).

There are two main reasons why this is possible. First, in our @PaniniJ solution, the inputs which describe a Panini capsule system are standard Java. Second, @PaniniJ is built on powerful standard Java tools and APIs, in particular, the Java annotation processing mechanism and the `javax.lang.model` API.

2.3. Capsule Declaration Syntax and Semantics

The @PaniniJ capsule declaration syntax is the syntax with which an @PaniniJ capsule programmer specifies a capsule template and thus the behavior and properties of an @PaniniJ capsule. We have some freedom to design a syntax and define associated semantics. @PaniniJ will use the capsule template checker component to verify that a capsule template's syntax is valid.

2.4. Assumptions and Constraints

Build Environment
We assume that the end-user environment is a standard Java build environment. Our annotation processor service should be portable across all such build environment. However, we are initially testing our system in a Maven/javac build environment and also Eclipse.

Java Version
We are assuming Java version 1.8 or greater is being used.

Distribution
@PaniniJ should be as plug-and-play as possible. The only steps required to run our program is

Performance requirements
The resulting code must have comparable speed, performance, etc., as the original PaniniJ panc compiler version.

Possible and/or probable changes in functionality.
This is a research language so it is anticipated that it will be continually changing. However, we do not anticipate any changes occurring while we are creating this for our senior design project.

Annotation Processor Limitations
The annotation processor can only generate new code, it cannot edit or modify code that the user writes.

2.5. Non-Functional Requirements

NFR 1: The user shall not need to directly manipulate modify or even look at any generated artifacts. The user need only write the template classes in order to specify a capsule or signature.
Motivation: We to provide a programming model which allows the developer to be code at a higher level of abstraction than the boilerplate generated code (i.e. we support capsule-oriented programming). Furthermore, we do not want to burden the user with the need to understand any of the generated artifacts in order to.

NFR 2: Limit name collisions and report any name collisions that do occur.
Motivation: We don't want the user to be unable to use certain words that are used in the implementations of generated artifacts.

NFR 3: The amount of code required to make a Panini capsule system with @PaniniJ should be comparable to the amount of code required to make a similar system using PaniniJ.
Motivation: Our goal is to minimize boilerplate for the user. We want to keep the amount of code that the user must write to a minimum to accomplish our stated goal.

NFR 4: The capsule declaration syntax should be straightforward, but have the flexibility to allow the user to solve their problems.
Motivation: We don't want to force the user to memorize complex syntax in order to utilize our system. We can save the developer from constantly referencing documentation by making the @PaniniJ syntax clear and declarative.

NFR 5: Procedure invocation performance should be comparable with that in `panc`.
Motivation: If performance is not comparable to `panc` then the annotation processor is much less effective. It attacks the ultimate purpose of making the application faster via multi-threading.

2.6. Development Methods

We are using the Rapid Application Development method.[5] This involves creating many prototypes that tackle small problems instead of doing a lot of up-front planning. The lack of up-front planning is suitable for this project since it is a small portion of a larger ongoing research project. We expect that the requirements of the project will change frequently and without warning.

Additionally, the technologies we will be using are 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 can become familiar with the capabilities of the new technologies without committing to a single plan and then gauge their appropriateness as we go.

Throughout the development of this project several design refactorings must occur in order to bring the smaller features and prototypes together in a way which best meets 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 is necessary that multiple people view code before it becomes a part of the current design. To accomplish this, we are using pair programming as often as possible and also using tools such as git[6] and github[7] to manage pull requests and code reviews.

[5] http://en.wikipedia.org/wiki/Software_development_process#Rapid_application_development
[6] http://git-scm.com/
[7] https://github.com/

3. Architectural Strategies

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.

Why did use Java for capsule generation?
We chose to implement the capsule generation in Java because of the team's familiarity with writing Java code. Java also has standard API's that allow us to cleanly work with source code. Java also has a robust annotation system that we can use to analyze source code.

User interface paradigm:
Our goal with @PaniniJ is to allow developers to use common Java tools to develop Panini programs. We want to get out of the way of the developer so that they can use a familiar environment and begin working with Panini quickly.

From a code perspective, we want @PaniniJ to have a clear, declarative syntax that feels very similar to classic java. We accomplish this by making the hooks into the @PaniniJ system each have a singular clear purpose.

Concurrency Model:
Our annotation processor is called from the standard javac process, which may be of a different concurrency model. Our code is synchronous and will execute in that manner when called from javac.

4. System Architecture

@PaniniJ can be described in three components which contain their own discrete responsibilities.

1. Annotation Processor
2. Runtime
3. Lang

In the codebase, these components have been divided into their own java packages and bundled under the at-panini-j JAR file.

4.1. System: Annotation Processor

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.

Furthermore, the annotation processor provides an interface to certain resources provided by the standard annotation processing API to be used by the components to which it delegates. For example, a `Filer` object is encapsulated by `@PaniniJ` ultimately used by `MakeDuck` for creating new duck artifacts.

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

4.1.1. SubComponent: 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.

4.1.2. SubComponent: Duck Future Artifact Generator

The primary responsibility of this module is to generate the Duck Future Java classes. These are an essential part of making synchronous methods calls act as asynchronous procedures invocations. They serve two roles in the system. Firstly, they act as messages added to be added to a capsule's queue. Secondly, they act as invisible futures which are returned to the user to encapsulate the results of a procedure call.

The core functionality of duck futures remain the same in this system as it is in `panc`. This 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:

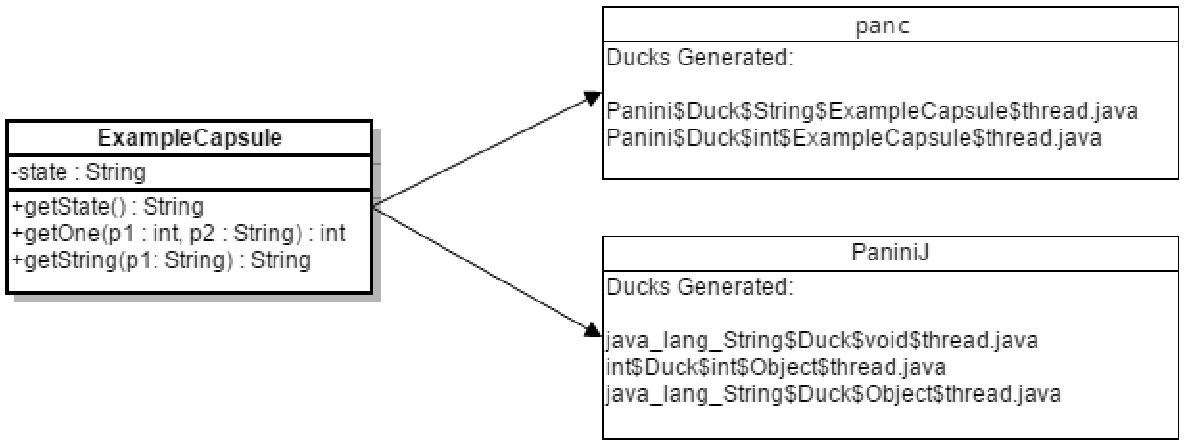

ExampleCapsule
-state : String
+getState() : String
+getOne(p1 : int, p2 : String) : int
+getString(p1: String) : String
panc
Ducks Generated:
Panini$Duck$String$ExampleCapsule$thread.java
Panini$Duck$int$ExampleCapsule$thread.java
PaniniJ
Ducks Generated:
java_lang_String$Duck$void$thread.java
int$Duck$int$Object$thread.java
java_lang_String$Duck$Object$thread.java

Figure 4.2 Duck Generation using PaniniJ's `panc` vs. `@PaniniJ`

Our Duck Future implementation also differs in the way that the passed parameters are stored. In `panc`, each time a procedure is matched to a Duck class instance fields are added to match the types of the procedure's parameters. The current `panc` implementation has an odd quirk when a Duck is created where a parameter is assigned to all instance fields that it can could possibly match. In a large system this can cause ducks to store much more information than is needed. Our method is tied tightly to the shape of the procedure (the composition of its parameter types) and avoid the aforementioned quirk by having clearly defined storage for the parameters.

4.2. System: Runtime

One of the major modules in our system is segmented into the `Runtime` package. 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`.

4.3. System: Lang

Many common Java classes in `java.lang` are marked final (e.g. String, Integer, etc.). Our system (as it is currently designed) and panc 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. Therefore, this is a case in which common classes (e.g. String) do not provide the user with implicit concurrency.

As a workaround for this problem, we have provided a package org.paninij.lang which mirrors java.lang which includes reimplementations. These reimplementations are not marked final and can thus be mocked as ducks and have implicit concurrency.

5. Policies and Tactics

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 ($). These describe classes that are auto-generated or do the generating of said classes. Auto-generated classes need this in order to prevent collisions with the user's code (since the auto-generated classes are kept in the same package as the users code). Additionally, many of the variables and method names on the generated classes start with `panini$`, 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.

6. Glossary

Artifact, also Source Artifact, Generated Artifact	A Java source code artifact created by @PaniniJ. Key examples include capsule classes and duck classes.
Artifact Generation	The process by which @PaniniJ processes a set of user-defined template classes and automatically generates/creates derived artifacts.
Capsule	An actor-like software construct defined in Panini which ● 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.
Capsule, Child	A capsule declared within the definition of another capsule. Note that each design argument of some capsule C is not counted as a child capsule of C (though they may well be child capsules of some other capsule).
Capsule, Leaf	A capsule having no children. A leaf capsule may be either passive or active.
Capsule, Passive	A capsule having no user-defined `run()` declaration.
Capsule, Active	A capsule having a user-defined `run()` declaration.
Capsule, Root	A capsule which is active and has no
Declaration, Capsule	
Declaration, `design()`	Where the user defines the set of design arguments and specifies what capsules are to be wired to it's child capsules.
Declaration, `init()`	Where the user defines initialization code for a capsule's state variables.
Declaration, Procedure	
Declaration, `run()`	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.

Declaration, Signature	
Capsule Requirements	The set of capsules S which must be passed to a capsule C in order for C to be well-defined.
Execution Profile	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.
Future	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.
Duck Future	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.
Method	A regular Java method. (This is distinct from the Panini concept of a procedure.)
Method Call	A regular call to a Java method. (This is distinct from the Panini concept of procedure invocation.)
Oracle	When testing whether some computation has computed some result correctly, an oracle can be queried for the result which that computation should have computed.
Panini	The abstract programming model which defines the semantics of a system of interacting capsules. **TODO: Add Reference**
PaniniJ	A research language similar to Java which adds support for the capsule-oriented programming as defined in the *Panini* programming model. **TODO: Add Reference**
@PaniniJ	The system described in this design document.
Procedure	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.
Procedure Invocation	A panini analog of a method call. (See *Procedure*.)

Shape	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.
Signature	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.
State Variable, also state	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.
System Topology	A network of capsules.
Template Class	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.
Wiring	The process of initializing a system of capsules with references to one another according to the user-defined system topology.