Software Design Specification

Team Dec 15-12: PaniniPress

Advisor	Dr. Rajan	
Client	Dr. Rajan	
Team Members	Dalton Mills	*Webmaster*
	David Johnston	*Team Lead*
	Kristin Clemens	*Communication 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 PaniniPress. 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 PaniniPress differs from and supplements PaniniJ, and how PaniniPress fits into the larger Panini Project.[1][2] Described within is the overall system architecture of PaniniPress, 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 PaniniPress 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.

TODO: Provide references for any other pertinent documents:

- **PaniniJ technical document for the language.**
- **The project plan document created earlier in the semester.**
- **Any tutorials for the original paniniJ language.**

The intended audience of this document includes developers of PaniniPress 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 PaniniPress 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.

TODO: We would like to include or what is, essentially, an abstract for this document in the styling of a typical research paper, since this document is primarily directed at researchers.

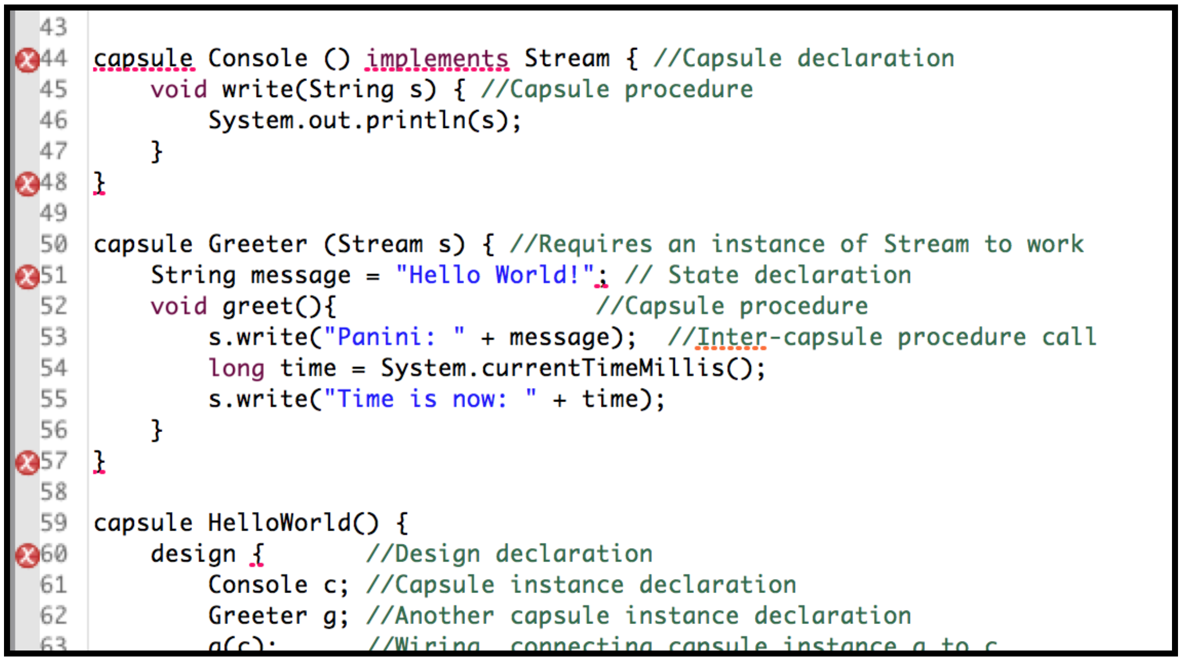
[1] The Panini programming language.
[2] More information on PaniniJ and The Panini Projet can be found at http://www.paninij.org/ (March, 2015).
[3] Dr. Rajan's professional homepage can be found at http://www.cs.iastate.edu/~hridesh/.

2. Design Considerations

2.1. Prior Work

PaniniPress 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
```

Image 2.1: Excerpt of 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.

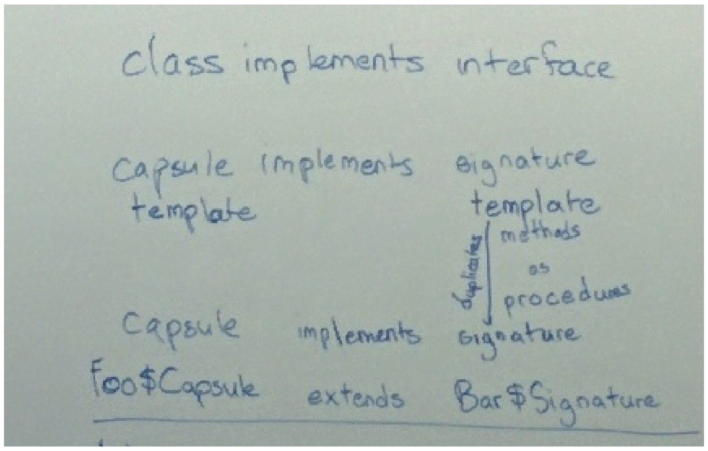
2.2. Primary Design Goal

PaniniPress is an alternative, re-engineered system for generating Panini capsule systems. Unlike the existing PaniniJ/`panc` method, PaniniPress 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 PaniniPress solution, the inputs which describe a Panini capsule system are standard Java. Second, PaniniPress is built on powerful standard Java tools and APIs, in particular, the Java annotation processing mechanism and the `javax.lang.model` API.

2.3. Introduction To Capsule Declaration Syntax

TODO: Give a simple example of a PaniniPress capsule declaration. Explain it and identify the following elements: procedure declarations, `init()` declaration, `run()` declaration, `design()` declaration. These explanations can be with respect to prior understanding of the Panini capsule model.



2.4. General Constraints

TODO: Explain purpose of this section to audience.

Describe any global limitations or constraints that have a significant impact on the design of the system's software (and describe the associated impact). Such constraints may be imposed by any of the following (the list is not exhaustive):

- Hardware or software environment
- End-user environment
- Availability or volatility of resources
- Standards compliance
- Interoperability requirements.
    - It is not necessary that the end result of this project be backwards compatible with the original PaniniJ which uses the panc compiler.
- Interface/protocol requirements.
    - Relevant? What is meant by an interface or protocol requirement? Something to look up on Wikipedia probably.
- ~~Data repository and distribution requirements~~
- Security requirements (or other such regulations)
    - Probably none. Probably ask Dr Rajan about this soon. Might already be included in existing PaniniJ documentation.
- Memory and other capacity limitations
    - Unknown.
- Performance requirements The resulting code must have comparable speed, performance, etc., as the original PaniniJ panc compiler version.
- ~~Network communications~~
- Verification and validation requirements (testing) Possibly not in the scope of this project. However, it would be nice to have a test harness in the future. We have been talking about the idea of possibly generating test code based on pre-existing test code that the developer him or herself has written. If we manage to pull this off, it would be pretty awesome. However we are unsure as to whether or not there will be enough time. Something to ask Dr Jean. We don't want to promise any features this early that we can't necessarily provide.
- Other means of addressing quality goals
- ~~Other requirements described in the requirements specification~~

2.5. Implementation Constraints

TODO: Explain the purpose of this section to the audience.

This specifies constraints on the way that the problem must be solved. Describe the mandated technology or solution. Also explain the reason for using the technology. The constraints are treated as a type of requirement. Also, see possible sub-headings below. We can use any of these, or none, or anything in between.

Implementation Environment of the Current System
Java 1.8 and custom Panini compiler (`panc`).

Implementation Environment of the Anticipated System
Java 1.8.

Partner or Collaborative Applications
Eclipse.

Off-the-Shelf Software
Refer to relevant section in technical document for Panini probably.

Schedule Constraints
See project plan.

2.4. Assumptions and Dependencies
TODO: Explain the purpose of this section to the audience.

Describe any assumptions or dependencies regarding the software and its use. These may concern such issues as:

- Related software or hardware Java SDK. PaniniJ language. No hardware. Eclipse. Version of Java?
- Operating systems Windows, Linux, and any other language that can run Eclipse realistically.
- End-user characteristics What does this part mean? What exactly is an end-user characteristic? Does it actually applied to this document and our project?
- Possible and/or probable changes in functionality. Functionality in what way? 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. Do we need to elaborate upon any future changes? Something to ask Dr Rajan.

2.1.1. Facts

2.1.2. Assumptions
- How panini works will not change while we are working on this software project.
- We will not have to worry about multiple versions of Java. Only Java 1.8.

2.1.3. Dependencies

- Possibly none?
- Perhaps just the Java SDK. Oh and of course Eclipse.

2.7. Supplementary Design Goals and Guidelines

Describe any goals, guidelines, principles, or priorities which dominate or embody the design of the system's software. Such goals might be:

- The KISS principle ("Keep it simple stupid!")
- Emphasis on speed versus memory use
- Working, looking, or "feeling" like an existing product

For each such goal or guideline, unless it is implicitly obvious, describe the **reason for its desirability**. Feel free to state and describe each goal in its own subsubsection if you wish.

TODO: Explain the purpose of this section to the reader.

NOTE: May be easier to think of this section as containing the Non-Functional Requirements, so to speak.

NFR 1: User 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 PaniniPress should be comparable to the amount of code required to make a similar system using PaniniJ.

Motivation: TODO

NFR 4: The capsule declaration syntax should be easy, though the behavior should be somewhat sophisticated. This is probably too broad.

Motivation: TODO

NFR 5: Procedure invocation performance should be comparable with that in `panc`.

Motivation: TODO

2.8. 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 thus expect that the requirements of the project can 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

TODO

- Use of a particular type of product (programming language, database, library, etc. ...)
- Reuse of existing software components to implement various parts/features of the system
- Future plans for extending or enhancing the software
- User interface paradigms (or system input and output models)
- Hardware and/or software interface paradigms
- Error detection and recovery
- Memory management policies
- External databases and/or data storage management and persistence
- Distributed data or control over a network
- Generalized approaches to control
- Concurrency and synchronization
- Communication mechanisms
- Management of other resources

TODO: Explain the purpose of this section to the reader.

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 make a system that does this automatically. ***TODO***

Why did use an Annotation Processor for source artifact generation? The Java source code inspection API, `javax.lang.model`, is exactly what we need. ***TODO***

Why did use Java for capsule generation? Because we are familiar with Java. Because the very useful annotation processing APIs are written for Java. ***TODO***

Why Java 1.8? ***TODO***

4. System Architecture

NOTE: Brief explanation moved to Introduction. Will add a more verbose description of this section here.

TODO: Explain the purpose of this section to the reader.
TODO: Explain meaning of a component service and a component secret.

1. Major responsibilities the software must undertake
2. Various roles the system (or portions of it) must play
3. How the system is broken down into components and subsystems
    a. Roles & responsibilities assigned to each
4. Relationship between higher-level components
    a. Describe how they interact to achieve required results
5. Rationale for choosing this system decomposition
    a. Other proposed decompositions
        i. Reasons they were rejected
6. Throughout the above, any design patterns that may apply.
7. Diagrams, models, flowcharts

Diagrams, models, flowcharts, documented scenarios/use-cases of system behavior and/or structure, either included here or in Detailed System Design section if overly complex.

Diagrams that describe a particular component or subsystem should be included within the particular subsection that describes that component or subsystem.

Subsections
1. Use for components (subsystems) requiring a more detailed analysis
2. Further divided? How?
3. Use above template for system architecture, applied to subsystem architecture.

4.1. System: Annotation Processor

The annotation processor system (a.k.a. `PaniniPress`) 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 `PaniniPress` 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.

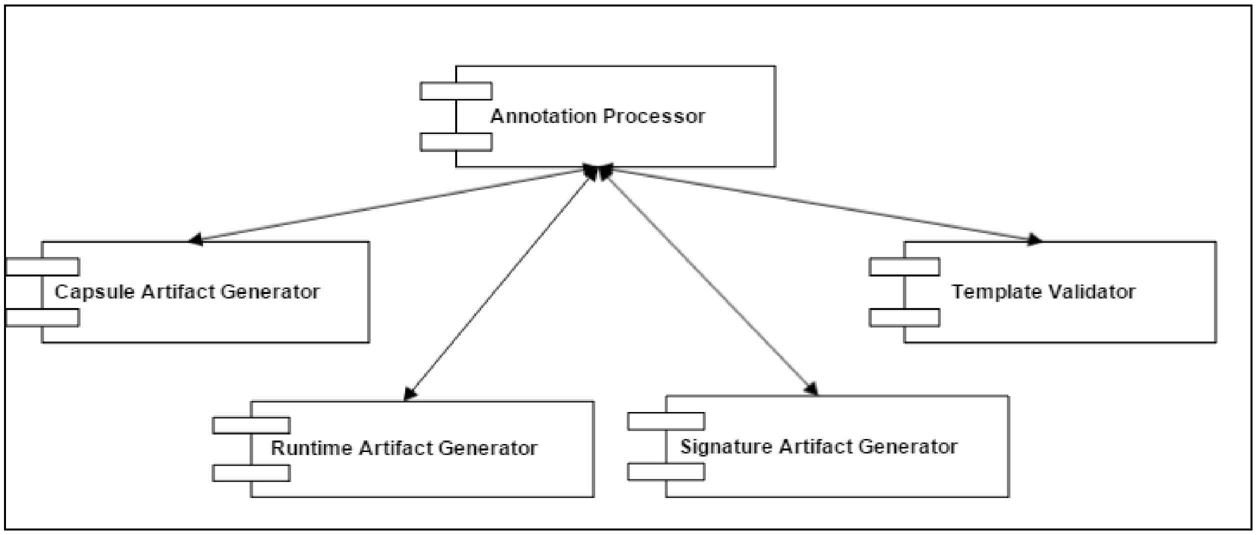

Annotation Processor
Capsule Artifact Generator
Template Validator
Runtime Artifact Generator
Signature Artifact Generator

Figure 4.1 System Diagram: Annotation Processor

4.1.1. Component: Capsule Artifact Generator

4.1.1.1. Subcomponent: Capsule$Thread Generator

4.1.1.2. Subcomponent: Capsule$Interface Generator

4.1.2. Component: Signature Artifact Generator

4.1.3. Component: Template Validator

TODO

4.1.4. Component: 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:

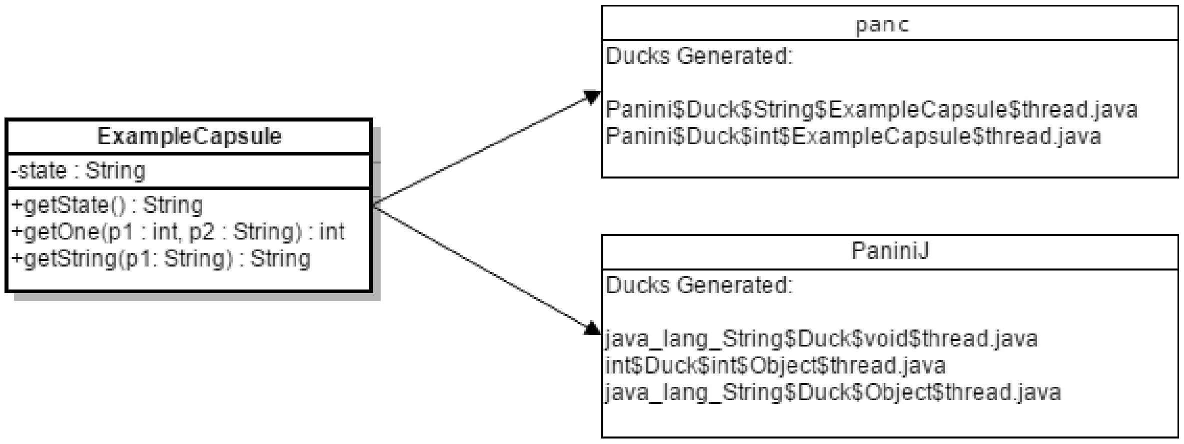

ExampleCapsule
-state : String
+getState() : String
+getOne(p1 : int, p2 : String) : int
+getString(p1: String) : String
panc
Ducks Generated:
Panini$Duck$String$ExampleCapsule$thread.java
Panini$Duck$int$ExampleCapsule$thread.java
PaniniJ
Ducks Generated:
java_lang_String$Duck$void$thread.java
int$Duck$int$Object$thread.java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

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

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.

4.1.5. Component: Runtime Artifact Generator

TODO

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`.

TODO: Changes in inheritance from `panc` (Future<T>, ResolvableFuture<T>, Message, SimpleMessage etc.)

4.3. System: Lang

TODO: Describe reimplementations of String, Integer, etc. Describe why this is necessary. Say that this comes directly from pre-existing work in panc.

5. Policies and Tactics

TODO: Explain the purpose of this section to the audience.

As stated in the PaniniPress Project Plan,

- The protocol of one or more subsystems, modules, or subroutines:
    - Protocol here is just the public interface / what governs interactions between things, right?
- **T**he choice of a particular algorithm or programming idiom (or design pattern) to implement portions of the system's functionality:
    - Are we using any specific design patterns? I think we are but just haven't used the names.
- Plans for ensuring requirements traceability
- Plans for testing the software:
    - Mention Oracle + Generated Test Artifacts here? Or elsewhere? (Some discussion of these may belong here, though we should also have the test generator as another component or set of components.)
- Plans for maintaining the software
    - Passed off to research group after completion of the project.
- Interfaces for end-users, software, hardware, and communications
    - Refer to implementation in Eclipse; works for other IDEs as well, ideally, using typical tools
- Hierarchical organization of the source code into its physical components (files and directories).
- How to build and/or generate the system's deliverables (how to compile, link, load, etc. ...)

NOTE (from template): Make sure that when describing a design decision that you also discuss any other significant alternatives that were considered, and your reasons for rejecting them (as well as your reasons for accepting the alternative you finally chose). For this reason, it may frequently be convenient to use one of the more popular "pattern formats" to describe a given tactic.

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 PaniniPress 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 PaniniPress 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. Detailed System Design

Given material in section four, for each component and their artifacts generated: each deserving of further elaboration will be elaborated upon as necessary. Describe each component that was listed in the System Architecture, as necessary.

TODO: Explain the purpose of this section to the audience.

For components described in the System Architecture section that require an even more detailed discussion; also, lower-level components and subcomponents as necessary. Each subsection refers to or contains a detailed description of a system software component.

For each:

1. Classification (the type of component)
2. Definition (purpose and semantic meaning of component)
3. Primary Responsibilities/Behavior
    a. Roles played; what the component accomplishes
    b. Services provided to its clients
    c. Refer to specific requirements as necessary
4. Constraints, Assumptions, Limitations relevant to the component, e.g.
    a. timing
    b. storage
    c. state
    d. rules for interacting with the component, e.g.
        i. preconditions
        ii. postconditions
        iii. invariants
        iv. constraints on input/output, local/global values
        v. data formats and access
        vi. synchronization
        vii. exceptions
5. Composition (use and meaning of subcomponents)
6. Uses/Interactions (description of component's collaborations with other components), e.g.
    a. other components used by this component
    b. other components that use this component
    c. any side-effects of the above
    d. methods of interaction for the above, and the interactions themselves
    e. subclasses, superclasses, metaclasses, etc.
7. Resources (any and all that are managed, affected, or needed by this component), e.g.

    a. anything external to the design (memory, processors, etc.)
    b. race conditions
    c. deadlock situations
    d. possible resolutions to the above

8. Processing (precisely how the component performs the duties necessary to fulfill its responsibilities), e.g.
    a. algorithms used
    b. changes of state
    c. time or space complexity
    d. concurrency
    e. creation, initialization, cleanup
    f. exception handling
9. Interface/Exports (set of services provided by the component) and definition/declaration of each element, plus comments, annotations, etc.
    a. services: resources, data, types, constants, subroutines, exceptions
    b. subcomponent attributes: classification, definition, responsibilities, etc., as above for the actual component

NOTE: Most of this should be in the source code, properly documented/commented. No need to reproduce information that is otherwise easily obtained by examining source and reading javadoc. This section should largely consist of references to or excerpts of annotated diagrams and source code.

7. Glossary

Artifact, also Source Artifact, Generated Artifact	A Java source code artifact created by PaniniPress. Key examples include capsule classes and duck classes.
Artifact Generation	The process by which PaniniPress 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	
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	
Design Arguments	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**
PaniniPress	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 PaniniPress 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.

8. Open Issues

1. What does comparable performance actually mean?
2. Limited understanding of how to tie in error checking and error revealing to the user / developer in Eclipse.
3. How do we ensure that the user code is actually being run properly by PaniniPress?

8. References

http://www.bradapp.com/docs/sdd.html

TODO: Complete this section!