Summer 8-5-2014

Capsule-oriented Programming in the Panini Language

Hridesh Rajan
Iowa State University, hridesh@iastate.edu

Steven M. Kautz
Iowa State University, smkautz@iastate.edu

Eric Lin
Iowa State University, eylin@iastate.edu

Sean L. Mooney
Iowa State University, smooney@iastate.edu

Yuheng Long
Iowa State University, csgzlong@iastate.edu

See next page for additional authors

Capsule-oriented Programming in the Panini Language

Hridesh Rajan, Steven M. Kautz, Eric Lin, Sean L. Mooney, Yuheng Long, and Ganesha Upadhyaya

TR #14-08
Initial Submission: August 5, 2014.

Keywords: Capsule-oriented programming, safe concurrency, implicitly-concurrent languages, pervasive interference problem, oblivious interference problem

Citation: Hridesh Rajan, Steven M. Kautz, Eric Lin, Sean L. Mooney, Yuheng Long, and Ganesha Upadhyaya, "Capsule-oriented Programming in the Panini Language", Technical Report #14-08, Dept. of Computer Science, Iowa State University, Aug 2014.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Capsule-oriented Programming in the Panini Language

Hridesh Rajan

With contributions from: Steven M. Kautz, Eric Lin, Sean L. Mooney, Yuheng Long, and Ganesha Upadhyaya

Revised: August 5, 2014 for Panini 0.9.3

Contents

Listings

Part I

Introduction

Chapter 1

Overview

An ounce of prevention is worth a pound of cure. – De Legibus (c. 1240) by De Bracton (d.1268)

Its a pity that engineering correct and efficient concurrent software remains a daunting task in this second decade of the 21st century. Programming language support for engineering concurrency-safe programs by construction in modern mainstream languages is inadequate at best. Even if we are to ignore the fact that deploying external tool support for verifying concurrency safety properties requires an additional step in the toolchain, such tools have had limited applicability due to the unbridled concurrency power of modern mainstream languages.

Prevailing dynamics has been that the research on and development of programming languages and that of tool support are in non-cooperative roles, which ultimately hurts productivity of software engineers.

The Panini programming language is an attempt to design concerted support for concurrency in both programming models and compile-time checking. Specifically, the project has four goals:

1. Simplifying concurrency support, while interesting patterns of concurrent program design are of significant intellectual interest to select few, we believe that simple constructs suffice for most needs,

2. ease the process of visualizing the interactions between the components, in order to make design decisions about concurrency and synchronization,

3. ease the process of producing safer concurrent programs by construction by providing automated, in-built support for error checking, and

4. ease the process of reasoning and understanding concurrent programs by significantly decreasing the number of program locations that necessitate reasoning about concurrent tasks.

To satisfy these goals, the Panini language proposes a programming model called *capsule-oriented programming*, where programmers describe a system in terms of its modular structure and write sequential code to implement the operations of those modules using a new abstraction that we call *capsule*. Capsule-oriented programs look like familiar sequential programs but they are implicitly concurrent. This programming guide describes Panini, the motivation behind the language in more detail, its main constructs and features, common programming patterns in the programming language, and finally installing and running Panini programs.

If you are completely new to the Panini programming language, you may want to read our guide on how to get started in chapter 3 or look at one of our examples in part III.

Chapter 2

Motivation

The Panini programming language is designed to enable implicit concurrency as a direct result of modularization of a system into capsules and to maintain modular reasoning in the presence of concurrency.

2.1 Concurrency: A Pressing Need

There is no escape: all programmers will soon be forced to consider concurrency decisions in software design. Most modern software systems tend to be distributed, event-driven, and asynchronous, often requiring components to maintain multiple threads for message and event handling. There is also increasing pressure on developers to introduce concurrency into applications in order to take advantage of multicore processors to improve performance.

Yet concurrent programming stubbornly remains difficult and error-prone. First, a programmer must partition the overall system workload into tasks. Second, tasks must be associated with threads of execution in a manner that improves utilization while minimizing overhead; note that this set of decisions is highly dependent on characteristics of the platform, such as the number of available cores. Finally, the programmer must manage the dependence, interaction, and potential interleaving between tasks to maintain the intended semantics and avoid concurrency hazards, often by using low-level primitives for synchronization. To address these issues, the invention and refinement of better abstractions is needed: that can hide the details of concurrency from the programmer and allow them to focus on the program logic.

The significance of better abstractions for concurrency is not lost on the research community. However, we believe that a major gap remains. There is an

impedance mismatch between sequential and implicitly concurrent code written using existing abstractions that is hard for a sequentially trained programmer to overcome. These programmers typically rely upon the sequence of operations to reason about their programs.

2.2 Running Example

To illustrate the challenges of concurrent program design, consider a simplified version of the classic arcade game *Asteroids*. The game consists of five components: a ship (`Ship`), game logic (`Logic`), user input (`Input`), controller (`Asteroids`) and a UI. The user input listens to the keyboard, parses the input, and directs the ship to make corresponding moves. The logic component maintains the board configuration, generates new asteroids, computes whether any previously generated asteroids have been destroyed by the ship, and computes whether the ship has collided with an asteroid. The ship moves and fires rockets as directed. The controller mediates the model and the view.

2.3 Difficult Concurrency-related Design Decisions

The modular structure of the system is clear from the description above, and it is not difficult to define five Java classes with appropriate methods corresponding to this design. However, the system will not yet work. The programmer is faced with a number of nontrivial decisions: Which of these components needs to be associated with an execution thread of its own? Which operations must be executed asynchronously? Where is synchronization going to be needed? A human expert might reach the following conclusions, shown in listing 2.1.

- A thread is needed to read the user input (line 49)
- The UI, as usual, has its own event-handling thread. The calls on the UI need to pass their data to the event handling thread via the UI event queue (lines 25 and 27)
- The game logic needs to run in a separate thread; otherwise, calls to update game state will "steal" the controller thread and cause the game to become jittery

- The `Ship` class does not need a dedicated thread, however, its methods need to be synchronized, since its data is accessed by both the user input thread and the controller thread

Listing 2.1: Program for a simplified arcade game Asteroids.

```
1  class Ship {
2    private short state = 0;
3    synchronized void die() { state = 2; }
4    synchronized void fire () { state = 1; }
5    synchronized boolean alive() { return state != 2; }
6    synchronized boolean isFiring() {
7      if (state == 1) { state = 0; return true; }
8      return false;
9    }
10   private int x = 5;
11   synchronized int getPos() { return x; }
12   synchronized void moveLeft() { if (x>0) x--; }
13   synchronized void moveRight() { if (x<10) x++; }
14 }
15 public class Asteroids extends Thread {
16   Ship s; UI ui; Logic l;
17   public void run() {
18     int points = 0;
19     while(s.alive()) {
20       int shipPos = s.getPos();
21       boolean isFiring = s.isFiring();
22       int result = l.step(shipPos, isFiring);
23       if (result>0) points += result;
24       else if (result<0) s.die();
25       ui.repaint(shipPos, isFiring, points);
26     }
27     ui.endgame();
28   }
29   public static void main(String[] args) {
30     Ship s = new Ship();
31     Input i = new Input(s);
32     Logic l = new Logic();
33     UI ui = new UI(l);
34     Asteroids c = new Asteroids(s, ui, l);
35     c.start();
36     i.start();
37     try {
38       i.join();
39       c.join();
40     } catch (InterruptedException e) {
41       e.printStackTrace();
42     }
```

```
43      }
44      public Asteroids(Ship s, UI ui; Logic l) {
45        this.s = s; this.ui = ui; this.l = l;
46      }
47    }
48    class UI { /* provides repaint, endgame */ }
49    class Input extends Thread { /* reads player input */ }
50    class Logic { /* provides step */ }
```

None of the conclusions above, in itself, is difficult to implement in Java. Rather, in practice it is the process of visualizing the interactions between the components, in order to reach those conclusions, that is extremely challenging for programmers [2, 1].

Chapter 3

Getting Started

3.1 Panini's Goals

A central goal of capsule-oriented programming and the Panini language is to help programmers deal with the challenges of concurrent program design.

The value proposition of the programming paradigm and the programming language is to enable greater program modularity and in doing so automatically enable greater program concurrency. In fact, Panini does not use explicit concurrency features. Instead, the programmer modularizes a program using capsules, which implicitly specify boundaries outside of which concurrency can occur. The Panini runtime will automatically enable concurrency in between the boundaries of capsules when safe to do so.

3.2 Hello World!

A Panini program is a collection of zero or more capsules. For example, a simple "hello world" program in Panini can be written as follows:

Listing 3.1: Hello World in Panini

```
1 capsule HelloWorld {
2   void run(){
3     System.out.println("Panini: Hello World!");
4     long time = System.currentTimeMillis();
5     System.out.println("Time is now: " + time);
6   }
7 }
```

This program declares a capsule called `HelloWorld`. The declaration of this capsule starts on line 1 and ends on line 7. The capsule `HelloWorld` contains only one procedure, `run` on line 2, which prints a message "Panini: Hello World!" on line 3 and prints current time on line 5.

This is a complete Panini program that can be compiled and executed.

When this program is executed, since it has only one capsule, and that capsule has a procedure named `run`, code inside that procedure is executed.

3.3 Compiling and running Hello World!

To compile and run this program, you will need the Panini compiler `panc` and the Panini executable `panini`. Both these applications are available from the Panini web-page `http://paninij.org` for download. For more information about installing and running the compiler see chapter 13.

Once you have downloaded and installed the Panini distribution, open your favorite text editor and save the HelloWorld program in listing 3.1 in a new file `HelloWorld.java`.

To compile this program simply run:

```
$ panc HelloWorld.java
```

You can then run this panini program with:

```
$ panini HelloWorld
  Panini: Hello World!
  Time is now: 1375940448626
```

The printed time is the difference, measured in milliseconds, between the time at which this command was issued and midnight, January 1, 1970 UTC.

3.4 Decomposing a Program into Capsules

A capsule-oriented program can have more than one capsules. To illustrate, let us decompose our HelloWorld program from previous section into two parts. Throughout this book we will use David Parnas's *information hiding principle* as our guide for program design. In essence, this principle says that one should decompose a program into parts in a manner such that each part is designed to "know about" and "hide" certain key decisions about how that program is

implemented. This is done so that, if necessary, those decisions can be changed later by us and others.[1]

We can decompose our HelloWorld program into three parts: a `Greeter` capsule that knows about the method of proper greeting, e.g. "Hello" in English, "Namaste" in Hindi, a `Console` capsule that knows about the medium that will be used to convey the greeting, e.g. standard output, a file, and a `HelloWorld` capsule that puts these parts together.

Listing 3.2: Hello World Decomposed!

```
1  capsule Console {
2    void write(String s) {
3      System.out.println(s);
4    }
5  }

7  capsule Greeter ( Console c ) {
8    void greet() {
9      c.write("Panini: Hello World, Decomposed!");
10     long time = System.currentTimeMillis();
11     c.write("Time is now: " + time);
12   }
13 }

15 capsule HelloWorld {
16  design {
17    Console c;
18    Greeter g;
19    g(c);
20  }
21  void run() {
22    g.greet();
23  }
24 }
```

This new version declares a capsule `Console` on line 1. This capsule declares a single procedure `write` on line 2 that writes its argument `s` on the standard output.

The `Greeter` capsule definition now contains a specification of what other capsules it requires. On line 7, it says that a `Greeter` requires a `Console` to work properly.

Once a capsule definition declares such requirements, procedures of the required capsules can be called. For example, the `write` procedure of the `Console` capsule is called on lines 9 and 11 in the `Greeter` capsule.

[1] **Further Reading:** David Parnas, "On the Criteria to Be Used in Decomposing Systems Into Modules", Communications of the ACM, December 1972.

As you might notice the `Console` and the the `Greeter` capsules do not have a `run` procedure, but the `HelloWorld` does. The run procedure is optional and signals that the capsule can start computation without external stimuli (i.e. if said capsule is instantiated in a program, then the run procedure of that capsule will be executed as soon as program initialization finishes).

On lines 16-20, this program has a *design declaration*, a new feature in Panini. The role of a design declaration is to define parts of a Panini program (typically capsules) and how these parts are connected. The design declaration on lines 16-20 simply states that this capsule `HelloWorld` will, as its internal parts, contain one *instance* of capsule `Console` (line 17), one *instance* of capsule `Greeter` (line 18). On line 19, the design declaration states that the `Greeter` instance `g` is connected to the `Console` instance `c`.

3.5 Implicit Concurrency in Capsule-oriented Programs

As mentioned previously, Panini does not use explicit concurrency features. Instead, the programmer modularizes a program using capsules, which implicitly specify boundaries outside of which concurrency can occur. The Panini runtime will automatically enable concurrency in between the boundaries of capsules when safe to do so.

When a procedure is called on an external capsule, e.g. call to `write` procedure on line 9, and if it is safe to do so, the call immediately returns allowing the caller capsule and the callee capsule to work independently. Here, the `Greeter` capsule can immediately continue to obtain the current system time, while the `Console` capsule prints first line of the greeting.

This is the main benefit of capsule-oriented programming and the Panini language. Implicit concurrency is achieved without having to introduce explicit concurrency features like threads, task pools, etc. This simplifies programming tasks. Not having to worry about concurrency is the main promise of capsule-oriented programming.

Now that you've written your first Panini program it is time to familiarize yourself with more complex features of the language in the next chapters.

Chapter 4

Capsule-oriented Design

Simplicity is the ultimate sophistication. – Leonardo da Vinci

4.1 Asteroids in Panini

We will illustrate capsule-oriented program design and Panini's new features using a simplified version of the classic arcade game `Asteroids`. In this game objective of the player is to score as many points as possible by destroying asteroids. In the simplified game, the player controls a ship that can move left or right. The ship can also fire.

For our problem, the subproblems are modeling the ship, game logic, user input, controller, and UI. The user input component would listen to the keyboard, parse the input, and direct the ship to make corresponding moves. The logic component would maintain the board configuration, generate new asteroids, compute whether any previously generated asteroids have been destroyed by the ship, and compute whether the ship has collided with an asteroid. The ship would move and fire rockets as directed by player. Finally, the controller would mediate the model and the view.

The modular structure of the system is clear from the description above, and it is not difficult to define five modules with appropriate methods corresponding to this design. However, the system will not yet work. The programmer is faced with a number of nontrivial decisions related to concurrency: Which of these components needs to be associated with an execution thread of its own? Which operations must be executed asynchronously? Where is synchronization going to be needed? A human expert might reach the following conclusions.

1. An independent thread of control is needed to read the user input
2. The UI, as usual, has its own event-handling thread of control. The calls on the UI need to pass their data to the event handling thread via the UI event queue
3. The game logic needs to run in a separate thread of control; otherwise, calls to update game state will "steal" the controller thread of control and cause the game to become jittery
4. The Ship class does not need a dedicated thread of control, however, its methods need to be synchronized, since its data is accessed by both the user input thread and the controller thread

None of the conclusions above, in itself, is difficult to implement in existing programming languages. Rather, in practice it is the process of visualizing the interactions between the components, in order to reach those conclusions, that is extremely challenging for programmers.

Capsule-oriented programming paradigm and the Panini language removes most of this burden so that the programmer can focus on their application's logic.

4.2 Architecture and Design

In capsule-oriented programming better design leads to better implicit concurrency, so it is valuable to start off with the architecture and design.

The Panini programmer specifies a system as a collection of `capsules`, `signatures`, and ordinary object-oriented classes. A `capsule` is an abstraction for decomposing software into its parts, and a `design` is a mechanism for composing these parts together. So the first order of business is to come up with this `capsule-oriented design`. This involves creating capsules, assigning subtasks to these capsules, designing how these capsules could be connected to each other, and finally integrating them to form a complete program. Each of these steps is fairly straightforward and is done entirely in the program.

1. *Create capsules and assign responsibilities to capsules.* In assigning responsibility, follow the information-hiding principle. There are four design decisions that are likely to change: ship's representation, game logic, UI design, and how we get input from the player. Therefore, we must hide

these responsibility behind interfaces of separate capsules. Finally, capsule Asteroids would encapsulate the design decisions related to controller logic.

Listing 4.1: Design skeleton for Asteroids

```
1 capsule Ship() { }
2 capsule Game() { }
3 capsule UI() { }
4 capsule Input() { }
5 capsule Asteroids () { }
```

This decision, to implement these components using the capsule abstraction, automatically handles four concurrency concerns mentioned above. Each capsule behaves as if it has an independent thread of control, which resolves issues 1-3 above. Only a single thread of control can ever access a capsule's data, which resolves the fourth issue.

This illustrates the value proposition of the programming paradigm and the programming language: to enable greater program modularity and in doing so automatically enable greater program concurrency. Observe that Panini does not use explicit concurrency features. Instead, the programmer modularizes a program using capsules, which implicitly specify boundaries outside of which concurrency can occur. The Panini runtime will automatically enable concurrency in between the boundaries of capsules when safe to do so.

Programmers familiar with the notion of actors may notice some similarities. However, capsules differs from extant work on actors in three significant ways that we believe is helpful for programmers. First, all inter-capsule communication is logically synchronous. Second, capsules, by default, provide confinement without requiring additional annotations from programmers. Third, topology of a capsule-oriented program is fixed at compile-time, which aids in static analysis of properties such as sequential consistency and confinement and with some implementation algorithms, e.g. garbage collection. Here, by topology we mean capsule instances and their interconnections.

Capsule-oriented programming eliminates two classes of concurrency errors by construction: sequential inconsistency and race conditions due to shared data.

2. *Design interconnection between capsules.* We do not yet know the interconnection between these five capsules, but it seems to be the case that Input ought to direct the ship to take actions, and UI might need information from Game to present it. Finally, controller Ateroids would need to talk to all of the capsules to be able to control their actions. We can use this knowledge to refine our architecture and design.

Listing 4.2: Interconnection between Capsules in Design for Asteroids

```
1  capsule Ship() { }
2  capsule Game() { }
3  capsule UI(Game g) { }
4  capsule Input(Ship s) { }
5  capsule Asteroids () { }
```

The third line says that the UI capsule could be connected with an external Game capsule. Alternatively, we could read the third line as: "the UI capsule requires a Game capsule." The fourth line says that the Input capsule could be connected with an external Ship capsule, and the first, the second, and the fifth lines says that the Ship, the Game, and the Asteroids capsules may not be connected to any other external capsules.

3. *Integrate capsules to form a complete program.* We now know that this program would require one Ship, one UI, one Game, and an Input. These will be co-ordinated by a controller. At this time, we can choose between two alternative designs: have the controller capsule Astroid contain other capsules, or create a new Capsule for that responsibility. We choose to assign this responsibility to the Astroid example, but the other choice is certainly feasible.

The listing below shows this design of the Asteroids program.

Listing 4.3: Complete Capsule-oriented Design for Asteroids

```
1  capsule Ship() { }
2  capsule Game() { }
3  capsule UI(Game g) { }
4  capsule Input(Ship s) { }
5  capsule Asteroids () {
6   design {
7    Ship s; UI ui; Game g; Input i;
8    ui(g); i(s);
9   }
10 }
```

Lines six to nine are the new parts of the design for this system. They define the internal design for the Asteroids capsule. This declarative design says that this capsule would have four internal components, one of each kind defined previously. In other word, one `capsule instance` s of kind Ship, another instance ui of kind UI, an instance g of kind Game, and another instance i of kind Input.

Unlike object instances, capsules instances do not need to be created using a `new` expression. It is sufficient to just declare them like line seven above.

Line eight defines interconnections between capsule instances. Line eight says that the capsule instance ui would be connected with the capsule instance g and capsule instance i would be connected with the capsule instance s.

4. *Check the design.* A nice property of Panini is that once you have written the high-level design above, you can check it using the Panini compiler to find out whether you got the capsule definitions and their interconnections right. Copy and paste the code above in a file Asteroids.java and compile it using the Panini compiler.

   To compile this program simply run:

   ```
   $ panc Asteroids.java
   ```

   This program is not yet executable, but with a small change we can make it executable by adding a `run` procedure in the `Asteroids` capsule.

Listing 4.4: Executable Capsule-oriented Design for Asteroids

```
1  capsule Ship() { }
2  capsule Game() { }
3  capsule UI(Game g) { }
4  capsule Input(Ship s) { }
5  capsule Asteroids () {
6   design {
7    Ship s; UI ui; Game g; Input i;
8    ui(g); i(s);
9   }
10  void run() {
11   System.out.println("TODO: fill in implementation");
12  }
13 }
```

To compile this program simply run:

```
$ panc Asteroids.java
```

You can then run this panini program with:

```
$ panini Asteroids
  TODO: fill in implementation
```

4.3 Implementation

1. *Capsule Ship.* We can now start specifying behavior of each of these capsules. The behavior of capsule ship is fairly straightforward, it should provide facilities to move left and right, to fire, to kill itself, and to check its position and state.

   The behavior of the capsule Ship requires keeping track of its position and its condition. In Panini, a capsule can declare states to keep track of such pieces of information. A state declaration is syntactically the same as a field declaration in object-oriented languages, however, it differs semantically in two ways: first, a state is private to a capsule (there are no public, protected, or static modifiers.), second, all the memory locations that can be reached via this state are uniquely owned by the containing capsule instance. Other capsule instances may not directly access it.

   Listing 4.5: Capsule Ship and its States

```
1 capsule Ship {
2   short state = 0;
3   int x = 5;
4 }
```

   The listing above shows two states on lines 2 and 3. You could also write state initializers to give them initial values, or you could write a capsule initializer as shown in the listing below.

   Listing 4.6: Capsule Ship with an Initializer

```
1 capsule Ship {
2   short state;
3   int x;
4   => {
5     state = 0;
6     x = 5;
7   }
8 }
```

   To allow other capsules to change its state, a capsule can provide capsule procedures, procedures for short. A capsule procedure is syntactically similar to methods in object-oriented languages, however, they are different

semantically in two ways: first, a capsule procedures is by default public (although private helper procedures can be declared using the private keyword), and second a procedure invocation is guaranteed to be logically synchronous. In some cases, Panini may be able to run procedures in parallel to improve concurrency in Panini programs. Several example procedures of the capsule Ship are shown below.

Listing 4.7: Complete Implementation of the Capsule Ship

```
1  capsule Ship {
2    short state = 0;
3    void die() { state = 2; }
4    void fire () { state = 1; }
5    boolean alive() { return state != 2; }
6    boolean isFiring() {
7      if (state == 1) { state = 0; return true; }
8      return false;
9    }

11   int x = 5;
12   int getPos() { return x; }
13   void moveLeft() { if (x>0) x--; }
14   void moveRight() { if (x<10) x++; }
15 }
```

Concurrency concerns in Ship's Design. Recall from our previous discussion that a ship's data is accessed by both the user input component and the controller component. Therefore, in an object-oriented design a human expert may conclude that all of its procedures need to be synchronized. A capsule's semantics gives this behavior by default: it ensures that the ship's data is accessed only by a single thread of control, ever. Thus, this concurrency concern is automatically addressed.

2. *Capsule Asteroids.* The behavior of the capsule Asteroids is specially interesting. This capsule declares an autonomous capsule procedure run on line six. Capsule Asteroids is a closed capsule

   A capsule is considered closed, if it does not require access to external capsule instances. In our example, Asteroids is a closed capsule, whereas Input is not. A closed capsule is a complete Panini program, and if it defines autonomous behavior, it can be executed.

Listing 4.8: Capsule Asteroids

```
1  capsule Asteroids {
2    design {
3      Ship s; UI ui; Game g; Input i;
```

```
4       ui(l); i(s);
5     }
6     void run() {
7       int points = 0;
8       while(s.alive()) {
9         int shipPos = s.getPos();
10        boolean isFiring = s.isFiring();
11        int result = g.step(shipPos, isFiring);
12        if(result>0) points += result;
13        else if (result<0) s.die();
14        ui.repaint(shipPos, isFiring, points);
15        yield(1000);
16      }
17      ui.endGame();
18    }
19  }
```

The execution of this program begins by allocating memory for all capsule instances, and connecting them together as specified in the design declaration on lines 2-5. Recall that capsule parameters define the other capsule instances required for a capsule to function. A capsule listed in another capsule's parameter list can be sent messages from that capsule. Design declarations allow a programmer to define the connections between individual capsule instances. These connections are established before execution of any capsule instance begins.

Next, any capsule with a run procedure begins executing independently as soon as the initialization and interconnection of all capsules is complete and may generate calls to the procedures of other capsules. For example, referring to the code above, capsule Asteroids will run code on 6-18. Capsules without a run procedure, such as Ship, perform computation only when their procedures are invoked. For example, on lines 8,9,10, and 13 procedures of the capsule Ship are invoked on the capsule instance s.

3. *Capsule Input.* A simple implementation of the capsule Input is shown below.

Listing 4.9: Capsule Input in Asteroids

```
1   capsule Input (Ship ship) {
2     void run(){
3       try {
4         while(ship.alive())
5           switch(System.in.read()) {
6             case 106: ship.moveLeft(); break; //Key j
7             case 108: ship.moveRight(); break;//Key l
8             case 105: ship.fire(); break; //Key k
```

```
9           }
10        } catch (IOException ioe) {}
11    }
12  }
```

This implementation continually checks for user input and directs the ship to move left, right or fire based on the key pressed.

Concurrency concerns in Input's Design. Note that the semantics of a capsule, i.e. each capsule instance runs as if it has a logically, independent thread of control, naturally satisfies the requirements of the Input capsule.

4. *Capsules Game and UI.* These capsules implement the game logic and a user interface that shows position of the ship, and the asteroids. A full implementation is available in the Panini distribution.

4.4 Analysis of Benefits

This example illustrates some of the key advantages of the capsule-oriented approach for programmers. These are:

- They don't need to create explicit threads or specify whether a given capsule needs its own thread of execution.
- They don't need to recognize or reason about potential data races due to shared data.
- They work within a familiar method-call style interface with a reasonable expectation of sequential consistency.
- All synchronization-related details are abstracted away and are fully transparent to them.

4.5 Compiling and running Asteroids!

If you haven't installed the Panini compiler yet then please see chapter 13 on installing and running the compiler.

You should have received a copy of the full `Asteroids` program as part of the Panini distribution. This program is located in the directory `examples` within the Panini distribution. Copy and save it, say, to the file `Asteroids.java`

in your local directory. If you have put the Panini compiler and the Panini executable in your path, you can compile the program by simply running:

To compile this program simply run:

```
$ panc Asteroids.java
```

You can then run this panini program with:

```
$ panini Asteroids
```

Where Asteroids is the name of the closed capsule that contains other capsule instances.

Now that you've done your first capsule-oriented design it is time to systematically familiarize yourself with different features of the Panini programming language.

Part II

Panini Language and its Features

To recall from previous chapters, capsule-oriented programming and the Panini language is designed to help programmers deal with the challenges of concurrent program design. The value proposition of the programming paradigm and the programming language is twofold:

1. to enable greater program modularity and in doing so automatically enable greater program concurrency, and

2. improve reasoning about programs in the presence of concurrency.

In fact, Panini does not use explicit concurrency features. Instead, the programmer modularizes a program using capsules, which implicitly specify boundaries outside of which concurrency can occur. The Panini runtime will automatically enable concurrency in between the boundaries of capsules when safe to do so. We will now discuss each part of the programming language in more detail.

Panini introduces three main features to extend the Java language. A *capsule* declaration, in short capsule, that is designed as a mechanism for decomposing a program into its parts, a *signature* declaration that serves as an interface for capsules. A capsule declaration may optionally contain a *design* declaration that is a mechanism for composing instances of capsules to form a subsystem or even an entire program. These three features and their parts are described below as follows:

- Capsule Declaration in chapter 5

- Design Declarations in chapter 6

- Signatures in chapter 7

Chapter 5

Capsule Declarations

Simplify, simplify. –Henry D. Thoreau

The Panini programmer specifies a design as a collection of *capsules* and ordinary object-oriented classes.

A *capsule* is an abstraction for decomposing a software into its parts. A capsule is like an *information-hiding module* in that it defines a set of public operations, hides the implementation details, and could serve as a work assignment for a developer or a team of developers. Beyond these standard responsibilities, a capsule also serves as a memory region, for some set of standard object instances and behaves as an independent logical process.

The notion of a capsule in the Panini programming language is designed to enable implicit concurrency at the interface of capsules as a direct result of the modularization of a design into capsules, and to maintain modular reasoning in the presence of implicit concurrency. Here, by modular reasoning we mean our ability to understand a software one module at a time by looking at the code for that module and only the interfaces of other modules referenced by name.

Inter-capsule calls look like ordinary method calls to the programmer. The object-oriented features are standard, but there are no explicit threads or synchronization locks in Panini.

A program in Panini can have zero or more signature declarations, zero or more class declarations, and zero or more capsule declarations. Panini classes are very similar to classes in other object oriented languages but with the restriction that they cannot contain any explicit concurrency constructs.

5.1 Syntax of Capsule Declaration

Capsules are designed to have the feel of ordinary classes, the intent being to capitalize on programmers' familiarity with object-oriented design and thus minimize the learning curve. Each capsule guarantees that its state is accessed by only one thread, thereby maintaining thread safety with respect to state mutations (by confinement in this case). Implicitly concurrent procedure calls on capsules are structured to have the appearance of ordinary method calls.

Capsules define a set of public operations as well as a memory region. They can have parameters, state declarations, inner class declarations, initializer, and procedure definitions.

The syntax of a capsule declaration is as follows.

Listing 5.1: Syntax of Capsule Declarations

```
capsule CapsuleName [ (Param p1, ..., Param pn) ] [ implements SignatureName+ ] {
  [  Initializer  ]
  [DesignDeclaration]
  StateDeclaration*
  ClassDeclaration*
  procedure*
}

Here, [s]  means s is optional,  s* means zero or more of s, and s+ means one or more of s.
```

A capsule declaration consists of the keyword 'capsule', the name of the capsule, zero or more formal parameters representing dependencies on other capsules, and zero or more signatures representing services that the capsule provides, followed by the capsule's implementation. Each procedure declaration in every signature implemented by the capsule must match with exactly one capsule procedure. Panini does not have capsule inheritance but does have class inheritance. The primary mechanism for reuse of capsules is composition.

Let us start with an empty capsule declaration shown in listing 5.2. This declaration just gives the capsule a name `C`.

Listing 5.2: An empty capsule declaration

```
1  capsule C {
2  }
```

Capsules can require access to instances of other capsules. For example, in listing 5.3 is a new version of the capsule `C` that requires access to an instance of another capsule `D`. Notice that when `C` did not have any requirements, e.g. in listing 5.2 we can omit '(' and ')' in capsule declaration.

Listing 5.3: An example capsule ‘C’ that requires a capsule instance of kind ‘D’

```
1  capsule D () {
2      ...
3  }
4  capsule C (D d) {
5      ..
6  }
```

Capsules can also require access to primitive, reference, or array values. For example, in listing 5.4 is another version of capsule `C` that requires access to an instance of another capsule `D`, a boolean value, a string, and an array of strings.

Listing 5.4: An example capsule ‘C’ that also requires other values

```
1  capsule D () {
2      ...
3  }
4  capsule C (D d, boolean b, String s, String [] args) {
5      ..
6  }
```

Capsules can implement signatures. An example is shown in listing 12.1. In this listing an empty signature `S` is implemented by the capsule `D`.

Listing 5.5: Signatures and capsules

```
1  signature S {
2  }
3  capsule D () implements S {
4      ...
5  }
6  capsule C (S d, boolean b, String s, String [] args) {
7      ...
8  }
```

Capsules can require access to another capsule instance that implement certain signatures. For example, in listing 12.1 capsule `C` now requires access to an instance of a capsule that implements signature `S`. Since the capsule `D` in this listing declares to implement this signature an instance of `D` could be provided to `C`, but it is also possible to provide instances of other capsules that also implement the signature `S`. More on signatures in chapter 7.

Closed capsules

A capsule is considered closed, if it does not require access to external capsule instances. In listing 12.1, `D` is a closed capsule, whereas `C` is not a closed capsule because it requires access to another capsule instance of type `S`. A closed capsule

is a complete Panini program. If a closed capsule provides autonomous behavior it can be executed.

5.2 Capsule States

A capsule can declare *states* to keep track of its internal information. A state declaration has a type, a name, and optionally an initialization expression. Legal types for a state declaration are primitive types and reference types.

For example, in listing 5.6, the capsule `D` declares a state of reference type `String` and the capsule `C` declares a state `i` of primitive type `int` and another state `privList` of reference type `ArrayList<Integer>`.

Listing 5.6: State declarations in capsules

```
1  import java.util.ArrayList;
2  signature S {
3  }
4  capsule D () implements S {
5      String name = "D";
6      ...
7  }
8  capsule C (S d, boolean b, String s, String[] args) {
9      int i;
10     ArrayList<Integer> privList = new ArrayList<Integer>();
11     ...
12 }
```

A state looks similar to a field in traditional class declarations, but there are two major differences.

- *State is always private.* All state declarations are private to a capsule, therefore, no visibility modifiers are necessary.
- *States are truly encapsulated.* A capsule instance controls all accesses to the object graph reachable from its states. More precisely, a capsule instance acts as a dominator for the object graph reachable from its states. More on that in chapter 8.

The listing 5.6 also illustrates using the type `ArrayList` available in the Java development kit (JDK). A type declared elsewhere can be imported using the import statements that has the same syntax as Java.

A capsule name is not a legal type for a state declaration.

5.3 Capsule Initializer

To setup the internal data structures of the capsule before it can interact with external entities, capsules can [optionally] declare a single initializer.

A capsule initializer runs before any other external procedure calls on a capsule instance.

As a concrete example, let us add a simple initializer to the capsule `C` that will initialize the values of `i` and `privList`.

Listing 5.7: Initializer in capsules

```
1  import java.util.ArrayList;
2  signature S {
3  }
4  capsule D () implements S {
5      String name = "D";
6      ...
7  }
8  capsule C (S d, boolean b, String s, String[] args) {
9      int i;
10     ArrayList<Integer> privList;

12     => {    // Start of an initializer
13       i = 42;
14       privList = new ArrayList<Integer>();
15     }
16     ...
17 }
```

A capsule initializer runs without interruption until completion. A desirable initializer should terminate, although there are no restrictions in Panini on initializer's structure to ensure termination.

A long running capsule initializer can render a capsule unresponsive.

5.4 Capsule Procedures

After the capsule initializer has completed setting up the data structures of the capsule instance, capsule instance can receive external procedure calls.

A capsule procedure has a return type, a name, zero or more arguments, and a body. For example, in listing 5.8 on lines 7-10 there is a procedure `append` that takes a single argument `s` of type `String`, adds `s` to `name` and returns `s`.

Listing 5.8: Capsule procedures

```
1  import java.util.ArrayList;
2  signature S {
3    String append(String s);
4  }
5  capsule D() implements S {
6    String name = "D";
7    String append(String s) {
8      name += s;
9      return s;
10   }
11 }
12 capsule C(S d, boolean b, String s, String[] args) {
13   int i;
14   ArrayList<Integer> privList;
15
16   => {   // Start of an initializer
17     i = 42;
18     privList = new ArrayList<Integer>();
19   }
20
21   private void add(Integer n) {
22     privList.add(n);
23   }
24
25   ...
26 }
```

A capsule procedure looks similar to a method in traditional class declarations, but there are three major differences.

- *Procedures run logically synchronously.* Unlike methods that run synchronously, i.e. when a method is called caller is blocked until caller finishes and returns, capsule procedures run *logically* synchronously, i.e. when a procedure is called caller is blocked only if it needs the result right away, otherwise caller and callee can compute concurrently.
- *Procedures have exclusive access to states.* A capsule procedure has exclusive access to the states of the capsule while it is running. Therefore, a programmer need not use locks or other synchronization mechanisms to gain exclusive access to states.
- *Procedures are by default public.* All procedure declarations are by default public for a capsule, therefore, no visibility modifiers are necessary to make them public.

Helper procedures can be declared by qualifying them with a modifier private. For example, in listing 5.8 on lines 21-23 there is a helper procedure `add`.

All capsule procedures, except helper procedures, constitute the interface of the capsule.

5.5 Autonomous Capsule Behavior

Capsule procedures like `append` and `add` are passive in that they cause the capsule to compute in response to a procedure call. A capsule can also declare autonomous behavior that does not require external call to run. There is one designated optional capsule procedure `run` representing that the capsule can start computation without external stimuli.

Listing 5.9: Autonomous capsule behavior

```
1  import java.util.ArrayList;
2  signature S {
3  }
4  capsule D () implements S {
5      String name = "D";
6      String append(String s) {
7        name += s;
8        return s;
9      }
10 }
11 capsule C (S d, boolean b, String s, String[] args) {
12     int i;
13     ArrayList<Integer> privList;

15      => {    // Start of an initializer
16          i = 42;
17          privList = new ArrayList<Integer>();
18      }

20     private void add(Integer n) {
21        privList.add(n);
22     }

24     void run() {
25      for(Integer i = 0; i < 25; i++) {
26        add(i);
27        d.append("_" + i);
28      }
29     }
30 }
```

For example, in listing 5.9 on lines 24-29 there is an autonomous procedure `run` that runs as soon as memory allocation and initialization for capsules is complete.

A capsule can either be autonomous by declaring a `run` procedure or passive and declare other procedures, but not both.

5.6 Capsule Procedure Calls

The semantics of capsule procedure calls is new and worth emphasis. In listing 5.9 the body of the autonomous procedure `run` shows two examples of capsule procedure calls on lines 26 and 27.

Calling an internal helper procedure. On line 26 is an example of calling helper procedure `add`, which runs to completion while the caller is blocked. This is to ensure that only a single capsule procedure has exclusive access to the states of the capsule.

Calling an external capsule procedure. On line 27 is an example of calling external capsule procedure `append`. This call immediately returns, while the callee continues to finish running the `append` procedure. This is because the caller does not make use of the return value of the `append` procedure.

Listing 5.10: Inter-capsule procedure call

```
1   void run() {
2    for(Integer i = 0; i < 25; i++) {
3      add(i);
4      String result = d.append("_" + i);
5    }
6   }
```

In listing 5.9 on line 4 is another example of calling external capsule procedure `append`. This call waits until the caller has finished running the `append` procedure. This is because here the caller needs the return value of the `append` procedure. So we must wait until that value is available.

A distinct advantage of capsule-oriented programming is that the programmer does not need to explicitly program these semantic variations.

5.7 Shutdown and Exit Procedures

Each capsule implicitly supports two built-in procedures: `shutdown` and `exit`.

When a capsule is requested to `shutdown`, and if it does not have any pending work, it terminates.

When a capsule is requested to `shutdown`, and if it has pending work, it puts the shutdown request at the end of its work queue and continues to serve other requests.

When a capsule is requested to `exit`, it terminates as soon as all of the pending work (as of the exit request) is done.

5.8 Example: Bank Account

To review features of a capsule declaration, we would now use them to build a simplified banking application. As a concrete example of capsules consider this simple model of a bank account in listing 5.11.

Listing 5.11: A Bank Account Capsule

```
1  capsule BankAccount() {
2      // this declaration represents the state of the capsule
3    double balance = 0.0;
4
5    /* Capsule procedures are defined in the same way object
6       methods are, and as far as the programmer is concerned
7       they behave largely like a normal class method */
8    void deposit(double money) {
9      balance += money;
10   }
11
12   void withdraw(double money) {
13     if (balance − money &lt; 0) {
14       throw new InvalidTransactionException();
15     }
16     balance −= money;
17   }
18 }
```

5.9 Implicit concurrency

Now consider an example where you have two clients who have a joint bank account. At some point, both of them might try to withdraw money at the same time. In traditional programming languages, if no explicit synchronization is used, this would be a data-race.

Here comes Panini's greatest strength, the programmer does not have to worry about any of that! He/she can write the out the logical design of the design and the concurrency related issues are dealt with behind the scenes.

Below is an example of such a design where the potential for error is eliminated by the language itself:

Listing 5.12: Multiple Clients of the Bank Account Capsule

```
1  capsule Bank{
2   design {
3    BankClient client1;
4    BankClient client2;
5    BankAccount jointAccount;

7    /* since client1 and client2 are both instantiations
8       of a capsule with a run procedure they will be
9       executed concurrently. And they will safely access
10      the bank account with no need to modify the original
11      implmentation of either the BankClient or BankAccount
12      capsules. */
13   client1(jointAccount);
14   client2(jointAccount);
15  }
16 }
```

Chapter 6

Design Declarations

A capsule declaration may optionally contain a design declaration. A design declaration is a declarative static specification of the topology of capsule instances that would be internal to that capsule. The topology of these capsule instances is fixed for a capsule declaration and does not change dynamically, which facilitates more precise static analysis of capsule interactions. Arrays of capsule instances of fixed length can also be declared.

For our banking example a design declaration could look like:

Listing 6.1: Design Declaration in Bank Capsule

```
1  capsule Bank {
2   design {
3    /* these two lines specify the capsule instances
4       that will be participating in the design */
5    BankAccount account;
6    BankClient client;
7
8    /* this line describes how the capsule instances
9       are connected with each other */
10   client (account);
11  }
12 }
```

This design declaration spans lines 2-11. On line 5 and 6 this design declaration specifies parts (or internal components) of the capsule Bank and on line 10 it says how these internal components are connected to each other.

6.1 Capsule Instance Declarations

You have already seen capsule instance declaration in the previous design example, where we declared a `BankAccount` instance and a `BankClient` capsule instance on lines 5 and 6 respectively. We do not need to explicitly allocate memory for a capsule instance with a `new`-like operator as this is handled implicitely.

6.2 Wiring Capsule Instances

A capsule instance can neither be returned nor passed as arguments to class methods (capsule instances are not first-class values); informally, capsule instances can only be `wired` or connected to other capsules.

Let us look at a BankClient capsule that wants to make use of a BankAccount:

```
1 /* the BankClient depends on the signature BankAccountSig */
2 capsule BankClient (BankAccountSig account) {
3   void run()  {
4     account.withdraw(10);
5   }
6 }
```

Two checks are performed on the design declaration to determine if all capsule instances are being properly wired.

First check disallows "null" values to be wired as capsule instances. For example, in the following capsule-oriented program the wiring declaration on line 6 is illegal.

```
1   Capsule C (C c, int  i ) {}
2   capsule Test {
3    design {
4        C c;
5     C c1;
6     c(null, 0);     // is  illegal
7     c(c1, 0);    // legal
8    }
9   }
```

The second check ensures that any capsule with arguments are properly wired. For example, in the following capsule-oriented program the wiring declarations are incomplete.

```
1   Capsule C{}
2   Capsule D(int i) {}
3   capsule Test {
4    design {
```

```
5       C c;
6       D d;
7     }
8   }
```

So the compiler will report an error `error: Capsule instance d may not be correctly initialized.` since the capsule instance d on line 6 expects an initial integer value and it has not been provided.

Arrays of Capsule Instances

In effort to allow for even further code reuse, capsule arrays may be instantiated. The syntax is the same as Java's.

```
1  capsule Bank{
2   design {
3    BankAccount accounts[5];
4    BankClient client [5];

6    for (int i = 0; i &lt; accounts.length; i++) {
7      client [ i ]( accounts[i ]) ;
8    }
9   }
10 }
```

6.3 Topology operators in design declarations

To make writing complex design declarations easier and to decrease tedious, repetitive wiring declarations in large designs, Panini provides some topology operators. These are: `wireall`, `ring`, `assoc`, and `star`.

These operators simplify wiring some of the common type of connections between capsules.

The wireall operator connects each element in a capsule array to the same set of arguments. For example, if cs is an array of capsule instances, of length n writing

```
1   wireall(cs, arg1, arg2, ...) ;
```

is the same as writing the following n wiring declarations.

```
1   cs[0](arg1, arg2, ...) ;
2   cs[1](arg1, arg2, ...) ;
3    ...
4   cs[n−1](arg1, arg2, ...) ;
```

The ring operator connects each element 'N' in a capsule array to element 'N+1'. It also connects the last element in the capsule array to the first element in the array. For example, if cs is an array of capsule instances, of length n writing

```
1  ring(cs);
```

is the same as writing the following n wiring declarations.

```
1  cs[0](cs[1]);
2  cs[1](cs[2]);
3   ...
4  cs[n-1](cs[0]);
```

The assoc (short for associate) operator connects elements of two capsule arrays from a starting index i, for a l items. For example, if cs and ds are arrays of capsule instances of length >=4 writing

```
1  assoc(cs, 3, ds, 2, args);
```

is the same as writing the following two wiring declarations.

```
1  cs[3](ds[3], args);
2  cs[4](ds[4], args);
```

The star operator connects a single capsule instance to all elements of a capsule array. For example, if c is a capsule instance and cs is an array of capsule instances following wires c to all elements of ds.

```
1  star(c, cs, args);
```

Chapter 7

Signature Declarations

A signature is to a capsule as an interface is to a class. A signature is the equivalent of an interface in object-oriented programming. It contains one or more abstract procedure signatures. Each procedure signature has a return type, a name, and zero or more formal parameters. This syntax is similar to interfaces in Java, except that for simplicity we do not allow signature inheritance. Capsules may implement multiple interfaces.

Let us extract a signature from the BankAccount capsule from above:

Listing 7.1: A signature for bank accounts

```
1  signature BankAccountSig{
2    void withdraw(double money);
3    void deposit(double money);
4  }
```

To have the BankAccount make use of this signature we write:

Listing 7.2: Implementing a signature

```
1  /* since both methods of declared in the signature where
2     already present in the original source code this is
3     the only line that needs modification. */
4  capsule BankAccount implements BankAccountSig{
5    double balance;
6    => {
7     balance = 100.0;
8    }
9
10   void deposit(double money) {
11     balance += money;
12   }
13
14   void withdraw(double money) {
15     if (balance - money &lt; 0) {
```

```
16          throw new InvalidTransactionException();
17        }
18        balance -= money;
19      }
20  }
```

Chapter 8

Capsule State Confinement

As mentioned previously all capsule states are private to a capsule. This notion of privacy is a bit stricter compared to object-oriented languages to promote stronger encapsulation. A capsule instance confines access to its state.

Stronger encapsulation aids with safe concurrency in capsule-oriented programs.

8.1 Confinement between Instances of a Capsule

For example, consider the listing below.

Listing 8.1: Confinement violation between capsule instances

```
1  import java.util.ArrayList;
2  capsule C(C other) {
3    ArrayList<Integer> privList = new ArrayList<Integer>();
4    void test() {
5      other.privList.add(42);
6    }
7  }

9  capsule TConfineInstance {
10   design {
11     C c1; C c2;
12     c1(c2); c2(c1);
13   }
14   void run() {
15     c1.test();
16   }
17 }
```

When compiled the Panini compiler will produce a compile-time error:

```
$ panc TConfineInstance.java
  TConfineInstance.java:8: error: States of capsules
cannot be accessed directly.
    other.privList.add(42);
    ^
    1 error
```

This is because in the capsule C, internal encapsulated state of the other capsule is directly accessed.

8.2 Confinement Violation in Procedure Call

Listing 8.2: Confinement violation in capsule procedure call

```
1  class TestC {
2    TestC next;
3    void setNext(TestC next) { this.next = next; }
4  }
6  capsule C {
7    void test(TestC tc) {   }
8  }
10 capsule M (C c) {
11   TestC tc = new TestC();
12   void mtest() {
13     tc.setNext(tc);
14     c.test(tc);
15   }
16 }
18 capsule ConfineTest {
19   design {
20     C c; M m;
21     m(c);
22   }
24   void run() {
25     m.mtest();
26   }
27 }
```

8.3 Confinement Violation in Return Statements

Listing 8.3: Confinement violation in return statements

```
1  class TestC {
2    TestC next;
3    void setNext(TestC next) { this.next = next; }
4  }

6  capsule M () {
7    TestC tc = new TestC();

9    TestC mtest2() {
10     return tc;
11   }
12 }

14 capsule ConfineTest {
15   design {
16     M m;
17     m();
18   }

20   void run() {
21     m.mtest2();
22   }
23 }
```

8.4 Resolving Confinement Violation

We suggest three strategies for resolving a confinement warning to the Panini programmer: (1) create a clone of the object, (2) if access to the entire object is not needed pass a portion of the object, and (3) if the object is large and shared, create a shared capsule whose sole purpose is to encapsulate that large object. In the rest of this section, we illustrate these strategies.

Resolving confinement violation using cloning

Listing 8.4: Resolving confinement violation with clones

```
1  A complete example that compiles and runs here.
```

Resolving confinement violation by passing parts

Listing 8.5: Resolving confinement violation by passing parts

```
1  A complete example that compiles and runs here.
```

Resolving confinement violation by creating shared capsules

Listing 8.6: Resolving confinement violation by creating shared capsules

```
1 A complete example that compiles and runs here.
```

Part III

Implicit Parallelism

In coarse-grained concurrent applications, such as the simplified Asteroids game illustrated in Chapter section 6, the main motivation is not necessarily to achieve parallel execution but rather to correctly and safely model components that are "logically autonomous". These kinds of asynchronous, event-driven systems are the obvious candidates for capsule-oriented design. However, the capsule abstraction also adapts easily to other styles of parallel programming, while retaining Panini's advantages of abstracting away the concurrency control mechanisms and ensuring data confinement.

Chapter 9

Master-Worker Pattern

9.1 Computing the constant Pi

To illustrate Panini's new features in practice, consider the classic problem of computing the constant pi using a Monte Carlo approximation. The idea behind the method is that we use the ratio, R, between the area of an enclosing square and the area on an enclosed circle, R = pi/4. We then proceed to randomly generating points within the above mentioned area of the square and count how many of them land in the enclosed circle. The ratio of points that land strictly withing the circle to the total number of points is a good approximation of the ratio R. We then multiply the result by 4 to get an estimate on the value of pi.

9.2 Architecture and design

In capsule-oriented programming better design leads to better implicit concurrency, i.e. better designed programs often run faster, so it is valuable to start off with the architecture and design.

1. Divide the problems into subproblems. In our case, the subproblems are:
    a) randomly generate a point and test if it's in the boundary of the circle
    b) agreggate the results
2. The Panini programmer specifies a system as a collection of capsules and ordinary object-oriented classes. A capsule is an abstraction for decomposing a software into its parts and a design block is a mechanism for

composing these parts together. So the first order of business is to come up with this capsule-oriented design. This involves creating capsules and assigning subtasks to these capsules.

3. Create capsules and assign responsibilities to capsules. In assigning responsibility follow the information-hiding principle. We should have a capsule that randomly generates points and tests whether or not they are within the circle. A master capsule that gathers the results from all generative capsules.

   This suggests capsules: Pi, Worker. For convenience we will be creating a wrapper class Number that implicitely handles conversions from integers to doubles and back.

Listing 9.1: Declaration of our capsules

```
1 capsule Pi(String args[])  {  }
2 capsule Worker() { }
```

As you can see above, capsule Pi will be the one that receives command line parameters.

4. Integrate capsules to form a design block. We now know that this program would require one Pi capsule, and multiple workers. At this time, we can make a decision about how many workers we want in this program. In this particular case we settle on a fixed number of Worker capsules, 10.

   Every capsule can have a design block, it effectively marks the capsule as a high level component that is composed out of other capsules. In our case the best choice would be to give the Pi capsule such a block. From the description of the problem we can see that the Pi capsule needs to know about the Worker capsules, but not the other way arround.

   Let us look at the public interfaces of each capsule and the design block:

Listing 9.2: Public interfaces of the capsules

```
1 capsule Worker (double batchSize) {
2     // Computes the number of points, from a total of batchSize,
3     // that fall within the area of the circle
4   Number compute(double batchSize) { ... }
5 }

7 capsule Pi (String[] args) {
8   design {
9         Worker workers[10];
```

```
10      }
11    void run(){  ...  }
12  }
```

This declarative design block(lines 10-12) states that the program should have a set of 10 Worker capsules.

9.3 Implementation

Capsule Worker. The behavior of capsule Worker is fairly straightforward: generate a given number of points and count whether or not they fall within the circle.

To allow other capsules to change its state, a capsule can provide capsule procedures, procedures for short. A capsule procedure is syntactically similar to methods in object-oriented languages, however, they are different semantically in two ways: first, a capsule procedures is by default public (although private helper procedures can be declared using the private keyword), and second a procedure invocation is guaranteed to be logically synchronous. In some cases, Panini may be able to run procedures in parallel to improve parallelism in Panini programs. In this particular case the only state of our capsule is the random number generator.

Listing 9.3: Public interfaces of the capsules

```
1  capsule Worker () {
2    Random prng = new Random ();

4    Number compute(double num) {
5      Number _circleCount = new Number(0);
6      for  (double j = 0;  j  < num; j++) {
7        double x = prng.nextDouble();
8        double y = prng.nextDouble();
9        if  ((x * x + y * y) < 1)  _circleCount.incr () ;
10     }
11     return _circleCount;
12   }
13 }
```

The implementation of the compute procedure should be easily understood by any Java programmer, it has the same syntax. As for the semantics, a call to a non-void external capsule procedure immediately returns a "future" value, while the procedure that is called runs concurrently. That value behaves exactly like normal values, so you won't need to modify your programs to make adjustments

for it. When you need the actual value, and if the called procedure has completed running execution proceeds as usual, otherwise execution is blocked until the called procedure completes running.

Capsule Pi. Line 5 declares a procedure run, every capsule can optionally declare such a method and it is implicitely invoked at the start of the program.

Listing 9.4: Public interfaces of the capsules

```
1  capsule Worker () {
2  capsule Pi (String[] args) {
3    design {
4      Worker workers[10];
5    }
6    void run(){
7      if (args.length <= 0) {
8        System.out.println("Usage: panini Pi [sample size], try several hundred thousand
               samples.");
9        return;
10     }

12     double totalSamples = Integer.parseInt(args[0]);
13     double startTime = System.currentTimeMillis();
14     Number[] results = foreach(Worker w: workers)
15           w.compute(totalSamples/workers.length);

17     double total = 0;
18     for (int i=0; i < workers.length; i++)
19       total += results[i].value();

21     double pi = 4.0 * total / totalSamples;
22     System.out.println("Pi : " + pi);
23     double endTime = System.currentTimeMillis();
24     System.out.println("Time to compute Pi using " + totalSamples +
25           " samples was:" + (endTime - startTime) + "ms.");
26   }
27 }
```

9.4 Implicit concurrency

The implicit concurrency in this example is on line 12 in the capsule Pi, where calling an external capsule procedure immediately returns, the foreach is simply a sugar for calling the procedure on a capsule and storing the result in a cell of an array, one capsule at a time.

On lines 15-16, each indexing of the results array might wind up blocking due to the fact that the result has not been computed up until that point.

When it is safe to exploit these sources of implicit concurrency, Panini compiler will automatically introduce parallelism to speedup this program without intervention from the programmer.

Chapter 10

Leader-Follower Pattern

10.1 Creating Server/Client applications in Panini

Servers are naturally concurrent applications, they have to be able to respond to requests from clients at any point in time and can make no assumptions about when these requests arrive. Panini's features allow the programmer to write a server application as if he/she were writting a sequential program and get the concurrency needed to make the application viable in a real world setting via implicit concurrency. To illustrate we will be coding a simple EchoServer that simply repeats everything the clients say.

10.2 Architecture and design

1. Divide the problems into subproblems. In our case, the subproblems are:

    a) accept incoming connections.

    b) handle these connections accordingly.

    c) for illustration purposes write a client program to help test the server.

2. The Panini programmer specifies a system as a collection of capsules, signatures and ordinary object-oriented classes. A capsule is an abstraction for decomposing a software into its parts and a design block is a mechanism for composing these parts together. So the first order of business is to come up with this capsule-oriented design. This involves creating capsules and assigning subtasks to these capsules.

3. Make key design decisions. In our case, we want our server to be able to respons to multiple quasi-simultaneous incoming requests with ease.

4. Create capsules and assign responsibilities to capsules. We will start by defining the capsule EchoServer.

Listing 10.1: Capsule EchoServer

```
1 capsule EchoServer {...}
```

Now that we have a simple name to refer to the server we will define the capsule ConnectionHandler which needs to communicate with the server.

Listing 10.2: Capsule ConnectionHandler

```
1 capsule ConnectionHandler(EchoServer server) {...}
```

As a separate program we define the client. As you can see bellow the EchoClient does not need to know about the capsule EchoServer, it will communicate with it via a custom network protocol that will be evident later.

Listing 10.3: The client

```
1 capsule EchoClient() {...}
```

5. integrate capsules to form a design block. Since we want our server to handle multiple connections at the same time it makes sense to have multiple such handlers.

Listing 10.4: Design block

```
1 capsule EchoServer() {
2   design {
3     ConnectionHandler connHandlers[10];
4     wireall(connHandlers, this);
5   }
6   // ...
7 }
```

Every capsule can have a design block, it effectively marks the capsule as a high level component that is composed out of other capsules. In our case, the best choice would be to give the Pipeline capsule such a block.

This declarative design block (lines 3-5) declares a set of 10 ConnectionHandler capsules. On line 4 we link all of these capsules to the current EchoServer capsule.

Since the EchoClient program is composed only out of that one capsule it does not require a design block.

10.3 Implementation

Capsule EchoServer. The only thing that the server does is listen on port 8080 and accept a connection (line 19), when requested.

Listing 10.5: Entire implementation of EchoServer

```
1  capsule EchoServer() {
2    design {
3      ConnectionHandler connHandlers[10];
4      wireall(connHandlers, this);
5    }
6    ServerSocket ss;
7    => {
8      try {
9        ss = new ServerSocket(8080);
10     } catch (IOException e){ e.printStackTrace(System.err); }
11   }
12   Socket getConnection() {
13     Socket s = null;
14     try {
15       s = ss.accept();
16     } catch (IOException e){ e.printStackTrace(System.err); }
17     return s;
18   }
19 }
```

To allow other capsules to change its state, a capsule can provide capsule procedures, procedures for short. A capsule procedure is syntactically similar to methods in object-oriented languages, however, they are different semantically in two ways: first, a capsule procedures is by default public (although private helper procedures can be declared using the private keyword), and second a procedure invocation is guaranteed to be logically synchronous. In some cases, Panini may be able to run procedures in parallel to improve parallelism in Panini programs. Calls to a non-void external capsule procedure immediately returns a "future" value, while the procedure that is called runs concurrently. That value behaves exactly like normal values, so you won't need to modify your programs to make adjustments for it. When you need the actual value, and if the called procedure has completed running execution proceeds as usual, otherwise execution is blocked until the called procedure completes running.

The => denotes a capsule initialization block. Every capsule may declare it. The semantics of the language ensure that it will be run before the capsule responds to any messages (procedure calls).

ConnectionHandler When a ConnectionHandler calls the getConnection procedure, the call returns immediately with a future representing the Socket object, and a task corresponding to the procedure body is queued for execution in the Host. When a worker attempts to use the socket in its handleConnection helper, it blocks until the Server provides the actual socket. A server that can handle variable size connections can also be implemented similarly by introducing a mediator capsule between Server and Worker.

Listing 10.6: ConnectionHandler implementation

```
1  capsule ConnectionHandler(EchoServer server) {
2    void run() {
3      while (true) {
4        Socket s = server.getConnection();
5        handleConnection(s);
6      }
7    }
8    void handleConnection(Socket s) {
9      try {
10       PrintWriter out = new PrintWriter(s.getOutputStream(), true);
11       BufferedReader in = new BufferedReader(
12                   new InputStreamReader(s.getInputStream()));
13       String clientInput;
14       while ((clientInput = in.readLine()) != null) {
15         System.out.println("client says: " + clientInput);
16         out.println(clientInput);
17       }
18     } catch (IOException e) { e.printStackTrace(System.err); }
19   }
20 }
```

Any capsule with a run procedure begins executing independently as soon as the initialization and interconnection of all capsules is complete and may generate calls to the procedures of other capsules. For example, capsule Pipeline will run the code on lines 3-8. Capsules without a run procedure, such as EchoServer, perform computation only when their procedures are invoked.

The implementation of the handleConnection procedures should be easily understood by any Java programmer familiar with network communication. For any given connection, the handler will simply read all input and print it to the standard output.

Capsule EchoClient. A client will simply open a connection to a running server through port 8080, send a "Hello Server!" and "Goodbye Server!" message, both times printing the server's response.

Listing 10.7: Entire implementation of EchoServer

```
1  import java.net.*;
2  import java.io.*;

4  capsule EchoClient() {

6    Socket echoSocket = null;
7    PrintWriter out = null;
8    BufferedReader in = null;
9    BufferedReader stdIn = null;

11   void run() {
12     try {
13       open();
14       out.println("Hello Server!");
15       System.out.println("Server replied: " + in.readLine());
16       out.println("" + System.currentTimeMillis() + ".");
17       System.out.println("Server replied: " + in.readLine());
18       out.println("Good bye.");
19       System.out.println("Server replied: " + in.readLine());
20       close();
21     } catch (IOException e) {
22       e.printStackTrace(System.err);
23     }
24   }
25   private void open() {
26     try {
27       echoSocket = new Socket("localhost", 8080);
28       out = new PrintWriter(echoSocket.getOutputStream(), true);
29       in = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));
30       stdIn = new BufferedReader(new InputStreamReader(System.in));
31     } catch (UnknownHostException e) {
32       e.printStackTrace(System.err);
33     } catch (IOException e) {
34       e.printStackTrace(System.err);
35     }
36   }
37 capsule EchoClient() {...}
38   private void close() {
39     try {
40       out.close();
41       in.close();
42       stdIn.close();
43       echoSocket.close();
44     } catch (IOException e) {
45       e.printStackTrace(System.err);
46     }
47   }
48 }
```

As you can see, capsules can declare helper procedures: open (line 25) and close (line 38). These are accessible only by the owner capsule.

10.4 Implicit concurrency

This code is very similar to how one would write a sequential program to model the same scenario, so the structure of this Panini program would be familiar to a sequential programmer. This code is also free of any concurrency-related concerns, such as setup and teardown threads, synchronization between the ConnectionHandlers to access the EchoServer's ServerSocket.

An example of implicit concurrency is the run procedure of the ConnectionHandler capsule. All ten capsules will ask the EchoServer capsule for a connection and once such a connection is obtained any other handler can go acquire another one without having to wait on any previous handlers to finish communicating with the clients.

When it is safe to exploit these sources of implicit concurrency, Panini compiler will automatically introduce parallelism to speedup this program without intervention from the programmer.

Chapter 11

Pipeline Pattern

11.1 Overview of pipeline parallelism implementation

Pipeline parallelism is usually employed in situations where computation is done in a loop, but instead of running each iteration in parallel, the computation is divided into discrete stages and the work associated to each of these stages is done in parallel.

In languages with an explicit model of concurrency the programmer would wind up having to map each stage to one or more threads, all this while ensuring that the threads operate on different stages from different operations. On top of all this, the programmer might also want to organize the program in such a way that the stages of the pipeline are easily interchangeable (when it makes sense), replaceable and ultimately have the program be maintainable.

Panini's features allow the programmer to focus on the latter issues that deal with design and maintainability while the compiler will implicitely provide concurrency where possible. To illustrate, consider the problem of maintaining the running average, total, minimum and maximum price of a stock in a day.

11.2 Architecture and design

In capsule-oriented programming better design leads to better implicit concurrency, i.e. better designed programs often run faster, so it is valuable to start off with the architecture and design. As an overview, the Panini programmer spec-

ifies a design block as a collection of capsules, signatures and ordinary object-oriented classes. A capsule is an abstraction for decomposing a software into its parts and a design block is a mechanism for composing these parts together. So the first order of business is to come up with this capsule-oriented design. This involves creating capsules and assigning subtasks to these capsules. To start off:

1. Divide the problems into subproblems. In our case:
   a) computing average, sum, min, max
   b) generate multiple values and feed them through the pipeline
2. Make key design decisions. In our case, we want to be able to easily create a program that can compute either of the above values in any order. To that end, Panini allows us to declare a signature which allows us to define a common interface for capsules.
3. Create signatures, capsules and assign responsibilities to capsules. We will start by defining a signature Stage. It declares two procedure that will have to be implemented by any capsules that wants to implement it, the semantics are similar to that of Java interfaces.

Listing 11.1: Signature of any of all our pipeline stages

```
1 signature Stage {
2   // handles pipeline stage input
3   void consume(long n);
4   // reports the current state of the pipeline stage
5   void report() ;
6 }
```

Now that we have a signature we can create the capsules that represent the pipeline stages. Each of the stages that are interchangeable expect a Stage parameter so that they can be composed freely:

Listing 11.2: Definition of concrete capsules

```
1 capsule Average(Stage next) implements Stage {...}
2 capsule Sum(Stage next) implements Stage {...}
3 capsule Min(Stage next) implements Stage {...}
4 capsule Max(Stage next) implements Stage {...}
5 //we create an additional stage that is used to seal off the pipeline
6 capsule Sink() implements Stage {...}
```

And the only capsule left to define is the one that feeds numbers into the pipeline:

Listing 11.3: Pipeline capsule

```
1 capsule Pipeline() {...}
```

4. Integrate capsules to form a design block. We know that we need one Pipeline capsule and at least one sink, all other capsules could be composed as often and in any way we would want to. For no reason other than simply demonstrating everything we will use one capsule of each.

Listing 11.4: Definition of design block

```
1 capsule Pipeline() {
2   design {
3     Average avg; Sum sum; Min min; Max max; Sink sink;
4     avg(sum); sum(min); min(max); max(snk); sink(num);
5   }
6   void run()  {...}
7 }
```

Every capsule can have a design block, it effectively marks the capsule as a high level component that is composed out of other capsules. In our case, the best choice would be to give the Pipeline capsule such a block. This declarative design block (lines 2-5) declares one of each Stage capsule types (line 3). On line 4 we link each pipeline stage in the order: Average » Sum » Min » Max » Sink.

11.3 Implementation

Capsules implementing Stage The behaviour of these capsules is fairly straightforward. Every time the consume is called they accumulate state and then call the consume procedure on the next capsule (line 4) in the pipeline. They behave in a similar manner for the report procedure as well.

Listing 11.5: Implementations of the pipeline stages

```
1  capsule Sum (Stage next) implements Stage {
2      long sum = 0;
3      void consume(long n) {
4          next.consume(n);
5          sum += n;
6      }
7
8      void report(){
9          next.report() ;
10         System.out.println("Sum of numbers was " + sum + ".");
11     }
```

```
12  }

14  capsule Min (Stage next) implements Stage {
15      long min = Long.MAX_VALUE;
16      void consume(long n) {
17          next.consume(n);
18          if (n < min) min = n;
19      }

21      void report(){
22          next.report() ;
23          System.out.println("Min of numbers was " + min + ".");
24      }
25  }

27  capsule Max (Stage next) implements Stage {
28      long max = 0;
29      void consume(long n) {
30          next.consume(n);
31          if  ( n > max) max = n;
32      }

34      void report()  {
35          next.report() ;
36          System.out.println("Max of numbers was " + max + ".");
37      }
38  }

40  capsule Sink(long num) implements Stage {
41      long count = 0;
42      void consume(long n) {
43          count++;
44      }

46      void report()  {
47          if  (count != num)
48              throw new RuntimeException("count should be: " + num + "; but was: " + count);
49          System.out.println("Successful " + count + " runs!!");
50      }
51  }
```

The implementation of the compute procedures should be easily understood by any Java programmer, it has the same syntax. As for the semantics, a call to a non-void external capsule procedure immediately returns a "future" value, while the procedure that is called runs concurrently. That value behaves exactly like normal values, so you won't need to modify your programs to make adjustments for it. When you need the actual value, and if the called procedure has completed running execution proceeds as usual, otherwise execution is blocked until the called procedure completes running.

Capsule Pipeline Line 10 declares a procedure run, every capsule can optionally declare such a method and it is implicitely invoked at the start of the program.

Listing 11.6: Implementation of Pipeline

```
1  capsule Pipeline () {
2      int num = 500;
4      design {
5          Average avg; Sum sum; Min min; Max max; Sink snk;
6          avg(sum); sum(min); min(max); max(snk); snk(num);
7      }
9      Random prng = new Random ();
10     void run() {
11         for (int j = 0; j < num; j++) {
12             long n = prng.nextInt(1024);
13             avg.consume(n);
14         }
15         avg.report() ;
16     }
17 }
```

The execution of this program begins by allocating memory for all capsule instances, and connecting them together as specified in the design declaration (lines 4-7). Recall that capsule parameters define the other capsule instances required for a capsule to function. A capsule listed in another capsule's parameter list or in its design block can be sent messages from that capsule. Design declarations allow a programmer to define the connections between individual capsule instances. These connections are established before execution of any capsule instance begins.

Next, any capsule with a run procedure begins executing independently as soon as the initialization and interconnection of all capsules is complete and may generate calls to the procedures of other capsules. For example, capsule Pipeline will run the code on lines 10-16. Capsules without a run procedure, such as Max, perform computation only when their procedures are invoked.

11.4 Implicit concurrency

This code is very similar to how one would write a sequential program to model the same scenario, so the structure of this Panini program would be familiar to a sequential programmer. This code is also free of any concurrency-related concerns, such as setup and teardown threads for running each stage in the pipeline

concurrently and synchronization between adjacent stages to hand off the input to the next stage, which is typical of a pipeline pattern. This code would, however, have all of the benefits of the explicitly concurrent implementation. Therefore, we believe that a sequential programmer would have a greater chance of success when writting such a program in Panini.

The implicit concurrency in this example is on line 13 in the capsule Pipeline, where calling an external capsule procedure immediately returns. Additionally, every call to a consume procedure on any Stage type capsules, at any point throughout the pipeline are subject to implicit concurrency.

When it is safe to exploit these sources of implicit concurrency, Panini compiler will automatically introduce parallelism to speedup this program without intervention from the programmer.

Chapter 12

Distributor Pattern

The Fibonacci example below computes the Fibonacci numbers via the collaboration of a set of worker and distributor capsules. The worker capsules, of type `FibWorker`, iteratively compute the Fibonacci numbers, whereas the distributor capsule, of type `Distributor`, distributes the computation work among the worker capsules.

Listing 12.1: Fibonacci Example

```
1  signature Worker {
2    Number execute(int num);
3  }
4
5  capsule FibWorker (Worker w) implements Worker {
6    Number execute(int n) {
7      if (n < 2) return new Number(n);
8      if (n < 13) return new Number(helper(n));
9      return new Sum (w.execute(n−1), w.execute(n−2));
10   }
11   private int helper(int n) {
12           int prev1=0, prev2=1;
13           for(int i=0; i<n; i++) {
14                int savePrev1 = prev1;
15                prev1 = prev2;
16                prev2 = savePrev1 + prev2;
17           }
18           return prev1;
19   }
20 }
21
22 capsule Distributor (Worker[] workers) implements Worker {
23   int current = 0;
24   Number execute(int num) {
25     Number result = workers[current].execute(num);
26     current++;
27     if (current == workers.length) current = 0;
```

```
28      return result;
29    }
30  }

32  capsule Fibonacci (String[] args) {
33    design {
34      FibWorker workers[10];
35       Distributor d;
36      d(workers);
37      wireall(workers, d);
38    }
39    void run(){
40      try {
41        System.out.println(d.execute(Integer.parseInt(args[0])).v());
42      } catch (java.lang.ArrayIndexOutOfBoundsException e) {
43        System.out.println("Usage: panini Fibonacci <Number>");
44      }
45    }
46  }

48  class Number {
49    int number;
50    Number(int number){ this.number = number; }
51    int v(){ return number;}
52    public String toString() { return "" + number; }
53  }

55  class Sum extends Number {
56    Number left; Number right;
57    Sum(Number left, Number right){ super(0); this.left = left; this.right = right; }
58    Overrideint v() return left.v() + right.v(); @@
```

The system capsule of `Fibonacci`, lines 32–46, declares a system design with 10 `Fibworker` capsule instances, line 34, and 1 Distributor capsule instance, line 35. Each worker capsule is connected to the distributor capsule, line 37, and vice versa, line 36. The `Fibonacci` capsule also runs the program to compute the Fibonacci number for the of the input value of `args[0]`, line 41, by invoking the `execute` method of the distributor capsule.

The interface for worker and distributor capsules is specified by the signature `Worker`, lines 1–3, containing a method `execute`, implemented by both capsule implementations. The `FibWorker` capsule implements the signature on lines 1–3. In the worker capsule, (1) the Fibonacci number for numbers less than 2 is equal to the number itself, line 7; (2) for Fibonacci numbers less than 13, the capsule uses a local `helper` method, line 8, that iteratively computes the Fibonacci number; and (3) finally for number greater than 13, the computation of the Fibonacci number is sent to worker capsule `w`, line 9. According the the system design, especially line 37, the worker capsule `w` is the distributor, which

in turn send the computation of the Fibonacci numbers to some other worker capsules. The `Distributer` capsule, on lines 22–30, implements the signature `Worker`. The distributor invokes `execute` on the first worker capsules it has available to compute the Fibonacci number `num`. If the worker number is less than 13, the worker capsule computes the Fibonacci number and returns it. Otherwise, the worker capsule invokes back the execute method of the distributor with messages to compute the Fibonacci for numbers `num-1` and `num-2`. The distributor in turn, invokes other worker capsules to do these computations.

The classes `Number`, lines 48–53 and `Sum`, lines 55–58, are wrapper classes that encapsulate numerical values and their additions.

Part IV

Appendix

Chapter 13

Installing and Running the Panini Compiler

13.1 Downloading the Compiler

The Panini compiler can be obtained in the following forms:

- Pre-built binaries: From the Panini download page `http://paninij.org/download`
- Source code: Available via Sourceforge at the URL `http://sf.net/projects/paninij`

For starting out, we would recommend the pre-built binaries that account for all dependencies. The following instructions assume the use of the pre-built package.

13.2 Structure of Panini Distribution

The archive file contains the following:

- bin: Executable versions of the Panini compiler for both *nix and WIN environments.
- lib: Jars files for panini.
- examples: Some small examples of new Panini features.

- license: License agreements under with the Panini Compiler and associated tools are available.
- README: A textual version of this page.

13.3 Requirements for Running Panini Compiler

- Java Runtime Environment(JRE) 1.6 or greater.
- Apache Ant is also recommended. It available from the URL `http://ant.apache.org/`

13.4 Running the Panini Compiler

The Panini compiler can be used in two ways:

- like javac from the command-line
- as a replacement builder for javac in the javac ant task.

The following sections describe usage.

13.5 Running the Panini Compiler from the command-line

The bin folder in this distribution contains scripts that can be used to run the Panini compiler and to start compiled Panini programs.

- bin/panc and bin/panini : the command-line interfaces for *nix systems.
- bin/panc.bat and bin/panini.bat : the command-line interface for WIN systems.

The `panc` and `panc.bat` scripts have an interface like `javac`, the standard Java compiler. The `panini` and `panini.bat` have an interface like `java`, the standard Java virtual machine.

13.6 Compiling Examples from Command-Line

Examples of some small Panini programs are located in the examples folder. To compile an example, navigate to its directory and use Panini compiler just like javac. For example, to manually compile the Dining Philosophers example, navigate to the examples directory and type

```
../bin/panc Philosophers.java
```

Running compiled Panini programs requires the panini runtime be on the JVM classpath. The required runtime classes are in `$PANC_HOME/lib/panini_rt.jar`. The `panini` script, packaged in the bin directory, automatically starts a JVM with the correct library on the classpath.

Use `$PANC_HOME/bin/panini AppEntry` to run these programs.

13.7 Running the Panini Compiler from Within Ant

The panini compiler can be used in an ant build script by replacing the jar file used by the `javac` task. See the OpenJDK notes in the Ant manual at `http://ant.apache.org/manual/Tasks/javac.html` for more information.

```
<pre class="brush: xml;}
<property name="panc.dir"
        value="path/to/panini/ install }</property>
<property name="panc.jar"
        value="${panc.dir}/ lib / dist / lib /panc.jar}</property>

<!-- Create a version of javac, which uses the panini javac. jar . -->
<presetdef name="panc}
    <javac fork="yes}
        <compilerarg value="-J-Xbootclasspath/p:${panc.jar}}</compilerarg>
    </javac>
</presetdef>

<target name="build" >
    <panc srcdir="src"
        destdir="bin}
    </panc>
</target>
```

13.8 Translating Panini into Java

The panini ant task can also translate Panini programs into pure Java programs. After translation, a Panini program can be compiled with any standard Java compiler. To translate a Panini program set the `dotranslate` attribute to yes. For example:

```
1 <panc dotranslate="yes" srcdir="${source}" destdir="${output}"
2   include="*.java}</panc>
```

will translate all the Panini source files in the `${source}` to Java files in the `${output}` directory. The translation attribute does not support a nested package structure for the source files. If the translation option is used, all the source files must be in the default package. When compiling with javac, panini_rt.jar must be on the classpath. The build files for the included examples use the translation/javac strategy to produce runnable byte code.

13.9 Acknowledgments and Licensing

The Panini compiler builds on the OpenJDK java compiler available at the URL `http://openjdk.java.net/`.

Chapter 14

Profiling Panini Programs

14.1 Using Panini Profiler

You could also profile a Panini program to compute runtime and memory consumption of various parts of that program. This can be done from command-line using the 'panp' (for Panini profiler) command available in the Panini distribution.

An example usage of 'panp' is shown below. The following assumes that you have downloaded and installed the Panini compiler distribution.

```
1  [examples] $ ../bin/panc HelloWorld.java
2  [examples] $ ../bin/panp HelloWorld
3  [examples] $ ../bin/panc HelloWorld.java
4  [examples] $ ../bin/panp HelloWorld
5   profiler : on
6  remote: off
7  port: 15599
8  thread-depth: compact
9  thread.compact.threshold.ms: 1
10 max-method-count: compact
11 method.compact.threshold.ms: 1
12  file :  profile . txt
13 track . object . alloc : on
14 output:  text
15 debug: off
16  profiler -class: com.mentorgen.tools.profile.runtime.Profile
17 output-method-signatures: no
18 clock-resolution: ms
19 output-summary-only: no
20 exclude:null
21 Accept ClassLoader: sun.misc.Launcher$AppClassLoader
22 ClassLoaderFilter.1:
23 com.mentorgen.tools.profile.instrument. clfilter .StandardClassLoaderFilter
24 Java Interactive  Profiler :  starting
```

```
25 ----------
26 Hello World!
27 Controller -- shuttingdown
28 [examples] $ cat profile.txt
```

The file profile.txt contains the runtime profile generated by the profiler.

14.2 Configuring Panini Profiler

You could change the settings of Panini profiler by modifying the file 'PANC_HOME/lib/profile.properties'. Here, 'PANC_HOME' is the directory in which your Panini distribution is installed.

Chapter 15

FAQs

15.1 FAQs about the Language

- What is capsule-oriented programming?
- Who may benefit from capsule-oriented programming?
- Who may NOT benefit from capsule-oriented programming?
- What do I need to know about concurrency in order to learn the Panini language?
- Will I be able to use my existing object-oriented code?
- How are capsules different from Erlang actors?
- How are capsules different from Scala actors?
- How are capsules different from actors in general?
- Why are capsules different from actors?
- How are capsules different from standard object-oriented classes?
- Why did you introduce new syntax for capsules and system?

What is Capsule-oriented Programming?

Capsule-oriented programming is a programming paradigm that aims to ease development of concurrent software systems by allowing abstraction from concurrency-related concerns.

Capsule-oriented programming entails breaking down program logic into distinct parts called capsule declarations and composing these parts to form the complete program using system declaration.

Who may benefit from capsule-oriented programming?
Capsule-oriented programming is a suitable approach for programmers who don't want to be distracted by concurrency concerns so that they can focus on their software's logic. If you want your programming language to take care of your concurrency concerns just like Java and C# handles your memory management concerns then capsules-oriented programming is for you.

Who may NOT benefit from capsule-oriented programming?
Capsule-oriented programming may not be a suitable approach for a software project if it is preferred in that project to manage every aspect of concurrency manually. This is similar to why Java and C# are not suitable, if a software project requires manual memory management and explicit pointer arithmetic.

What do I need to know about concurrency in order to learn the Panini language?
Nothing.

Will I be able to use my existing object-oriented code?
Mostly. All existing classes that do not have a main method would work as is. It is also advisable to avoid using explicit concurrency features, e.g. threads, locks, synchronized, etc since they may interfere with Panini's internal mechanisms.

How are capsules different from Erlang actors?
Capsules are different from Erlang actors in that capsules ensure messages arrive in sequentially consistent order, as well as providing mutable state within

the actor.

How are capsules different from Scala actors?
Capsules are different from Scala actors because of the enforced confinement of a capsule's state.

How are capsules different from actors in general?
Here is a precise comparison,

	_ = Actors	_ = Capsules
_ can send messages to other _	☑	☑
_ can create a finite number of new _	☑	☒
_ can send _ as messages	☑	☒
_ can run code in response to messages	☑	☑
_ have local heap and stacks	☑	☑
_ can send asynchronous messages	☑	☑
_ can send synchronous messages	☒	☑
messages are buffered	☑	☑
message delivery is guaranteed	☑	☑
message ordering is guaranteed	☒	☑
message fairness is guaranteed	☒	☒

Why are capsules different from actors?
The decision to make different choices in capsule's design was based on two factors.

- To decrease the impedance mismatch between mainstream languages like Java, C#, etc and the capsule-oriented programming model. In the resulting design, inter-capsule procedure calls look like ordinary method calls in mainstream programming languages.
- We also simplified the language model a bit to make it feasible to build efficient, precise, and more automated analysis and compilation strategies.

How are capsules different from standard object-oriented classes?
A capsule is like a class in that it also defines a set of public operations, hides the implementation details, and could serve as a work assignment for a developer or a team of developers. Beyond these standard responsibilities, a capsule also serves as a memory region for some set of standard object instances and behaves as an independent logical process.

Why did you introduce new syntax for capsules and system?
At first glance a capsule declaration may look similar to a class declaration, thus naturally raising the question as to why a new syntactic category is essential, and why class declarations may not be enhanced with the additional capabilities that capsules provide, namely, confinement (as in Erlang) and an activity thread (as in previous work on concurrent OO languages). There are three main reasons for this design decision in Panini.

- First, we believe based on previous experiences that objects may be too fine-grained to think of each one as a potentially independent activity.
- Second, we wanted to specify a system as a set of related capsules with a fixed topology, in order to make it feasible to perform static analysis of the system graphs; this implies that capsules should not be first-class values.
- Third, there is a large body of OO code that is written without any regard to confinement. Changing the semantics of classes would have made reusing this vast set of libraries difficult, if not impossible. In the current design of Panini, since syntactic categories are different, sequential OO code can be reused within the boundary of a capsule without needing any modification.

15.2 FAQs about the Compiler

- How do I report a bug?
- Why do my files have the extension .java if I am writing Panini code?
- Can I compile all Java code with the Panini compiler?

- I compiled the HelloWorld Panini program and now I have all these .class files, can some of them be removed?

I believe I have just run into a compiler bug, how do I report it?
All bugs in the Panini compiler can be reported by sending an e-mail to panini@ iastate.edu. We would very much appreciate a self-contained source code file that reproduces the bug along with your e-mail, but if you don't have it you could also describe what you were trying to do and send us buggy output of the Panini compiler.

Why do my files have the extension .java if I am writing Panini code?
The Panini compiler was built on top of the Java OpenJDK compiler and, currently, we re-use most of the infrastructure provided by it. Panini files will have their own extension in future releases of the compiler.

Can I compile all Java code with the Panini compiler?
All code that does not have use any explicit concurrency features (e.g. synchronized keyword, threads, etc.) can be compiled using the Panini compiler. Also, the Java main method no longer has any special meaning in the Panini language.

I compiled the HelloWorld Panini program and now I have all these .class files, can some of them be removed?
Assuming you want your program to run, no. Currently, Panini programs first compile to Java programs and then to bytecode, the extra files contain everything needed to express Panini's abstractions in Java.

Bibliography

[1] ACM/IEEE-CS Joint Task Force. Computer science curricula 2013 (CS2013). Technical report, ACM/IEEE, 2012.

[2] D. Meder, V. Pankratius, and W. F. Tichy. Parallelism in curricula an international survey. Technical report, University of Karlsruhe, 2008.