Getting Started with Avian Computing

Exploring Parallel Programming with ConcX 2.3

By Nelson Chamberlain
December 20, 2017
Table of Contents

Part 1: Introductions and Demonstrations................................................................. 6
Chapter 1: Introduction to Avian Computing............................................................... 7
  A Look Ahead ........................................................................................................... 8
  A First Example of Parallel Processing .................................................................... 9
  Discussion ............................................................................................................... 9
  Procedure .............................................................................................................11
  Wrap Up ................................................................................................................24
Chapter 2. Installing Concurrency Explorer (ConcX).................................................... 26
  Tested and Untested Configurations ...................................................................... 26
  Software Dependencies ......................................................................................... 26
  Downloading ConcX ..............................................................................................27
    ConcX.jar Zip File Instructions .........................................................................27
    Download Code Snapshot of Source Files .........................................................27
    Checking Out ConcX Source Files ....................................................................28
  Setup New Java Project with Existing Sources ......................................................30
  Adding Jar Files and Libraries to the Project .........................................................33
  Set the Main Class ...............................................................................................33
Chapter 3. Avian Computing Concepts and ConcX......................................................36
  What Is Avian Computing? ....................................................................................36
  Why Avian Computing? .......................................................................................36
  Avian Computing Is a Simplifying Abstraction ....................................................36
  Is Avian Computing Really Necessary? .................................................................37
  Brains Work in Parallel, Logic Is Single-Threaded ...............................................37
  Programming Languages Are Processed in the Frontal Lobe .........................38
  What Kind of Parallel Developer Are You? ..........................................................39
  In Search of a Better Paradigm ............................................................................39
  Mapping Computer Operations into Bird Behaviors ............................................40
  The Avian Lifecycle in ConcX .............................................................................41
  Advantages of Avian Computing using ConcX ....................................................42
    ConcX Handles Parallelizing the App so You Don’t Have to ............................42
    ConcX Provides Asynchronous Message Passing and Decoupling of Objects ....42
    ConcX Provides Visual Clues to Internal Status of Each Bird When Flying ....42
Chapter 4. ConcX Controls and Indicators ............................................. 44

The Left Panel – Bird Management and Configuration Features ......................... 44
  The Bird Info Tab ........................................................................ 45
  The Vitality Tab .......................................................................... 50
  The Code Info Tab ...................................................................... 51
  The XMV Tab ............................................................................. 51
  The History Tab ......................................................................... 52

The Right Panel – Bird Activity, Amount of Food, and TupleTree Viewing ............. 53
  The Activity Tab ........................................................................ 53
  The Food Supply Tab ................................................................... 55
  The TupleTree Tab ...................................................................... 56

Bottom Row Buttons ........................................................................ 57

ConcX Menu Choices ....................................................................... 60
  The File Menu ........................................................................... 60
  The Config Menu ....................................................................... 61

Chapter 5. AddxBirds Revisited ............................................................ 65

A Brief Explanation of AddxBirds .......................................................... 65

Avian Foundations of AddxBirds ............................................................ 67

Variations of AddxBirds ..................................................................... 68
  Variation #1: Adjusting Nap Length Adjusts How Frequently a Bird Eats .......... 68
  Variation #2: Birds Die When They Run Out of Stamina .............................. 69
  Variation #3: One Fast Bird ................................................................ 70
  Variation #4: Adding Birds to Manage Fast Birds ...................................... 71
  Variation #5: Duplicating Birds to Manage Fast Birds ............................... 73
  Discussion of Variation #5 ................................................................ 74

AddxBird Code .................................................................................. 75

More Variations with add5.bird Flock ..................................................... 76
  Variation #6: Loading More Birds ................................................... 76
  Variation #7: Loading Even More Birds ............................................. 78
  Variation #8: Loading the Same Birds Multiple Times ............................ 78
Part 1: Introductions and Demonstrations

This part of the Getting Started guide contains a dozen chapter or so that introduce and demonstrate the basics of Avian Computing as implemented by ConcX.

Turn to Part 2 of this guide to learn more about how ConcX version 2.3 was implemented and learn about creating your own birds and food pods.
Chapter 1: Introduction to Avian Computing

The Avian Computing project is the first project intentionally designed to improve how we think about parallel programs. We have plenty of tools that optimize our code to run faster and even automatically insert parallel sections. Unfortunately, we don’t have any tools that optimize the really slow part of the system, i.e., the human part.

That’s right; I’m referring to us, you and me, the humans who are responsible for visualizing the complex, dynamic, asynchronous interactions of the multitude of components that make up parallel programs. This lack of thinking tools has led to parallel programs gaining a reputation for being slow to develop and difficult to debug.

Avian Computing invites us to think differently about parallel programming, to visualize a parallel program as a flock of birds instead of lines of code or interacting layers or contracts or structures. It is easy to visualize a flock of birds, either looking for food in trees or flying together in flight. When you do, you automatically create a mental model of parallel operation. Avian birds all work together but also act individually, each of them dynamically adjusting to a wealth of environmental conditions and doing it all, ahem, on the fly.

The Concurrency Explorer (ConcX) extends the Avian Computing perspective by modeling each thread as a bird. This modeling is accomplished by mapping thread actions into common bird behaviors, such as eating and sleeping, so that we can think more naturally about parallel programs. Instead of starting a new thread, a bird hatches and starts flying. Instead of thread terminating, the birds just die. ConcX birds eat (find work to do), digest (process the data), and nap (thread.sleep) as if they were real birds.

In ConcX, we design simple birds that do just one part of the overall task but do it in parallel with all the other birds “in their flock” that are also doing just one part of the overall task. Together, these simple birds do all the work required in our parallel programs without getting mired in the complications and complexities of traditional parallel coding efforts.

To feed the birds, we need to divide the work into bite-sized chunks that can handled by each individual bird. Subdividing the work into chunks forces us to examine the program’s
inherent data and to consider the relationships between the various tasks to perform in parallel. Subdividing the work helps to reveal opportunities for parallel work, highlight data dependencies in the program, and point out where the natural task boundaries are located.

The net result of this subdividing and exploring is that we are thinking more effectively about how to perform our task in parallel because we are thinking about what we want each thread to do instead of how to do it. This is ultimate goal of Avian Computing; to think more easily and naturally about parallel programming.

**A Look Ahead**

This guide is divided into two parts: Part 1 introduces the basics of ConcX and demonstrates its usage in a variety of parallel scenarios; Part 2 describes the major code modules, describes their (inter)relationships and how to develop new parallel applications using ConcX.

The following chapters are included in Part 1:

- Chapter 1 (this chapter) introduces the basics of Avian Computing using ConcX and the AddxBird scenario. It gives you a quick taste of what to expect in ConcX and Avian Computing and is intended as fail-early instruction. It demonstrates enough of the capabilities of ConcX so you can tell right away if this isn't the right tool for you so you haven’t wasted a lot of your time on the wrong tool.
- Chapter 2 describes how to install the source code for ConcX and run it from NetBeans.
- Chapter 3 briefly describes the key concepts of Avian Computing and how they are implemented in ConcX.
- Chapter 4 describes in detail the various controls, fields, and tabs available in ConcX and how they are used to adjust the behavior of the birds.
- Chapter 5 repeats and expands the AddxBirds scenario presented in this introductory chapter, explaining what is happening in that scenario in more detail. It also loads other AddxBird examples and explores how they work. It also assumes that you have installed NetBeans (if required) and downloaded the ConcX source code as described in Chapter 2.
- Chapters 6 and the rest of the chapters in Part 1 present a wide variety of parallel program scenarios, such as the classic Dining Philosophers problem, estimating Pi in parallel, various simulations, working with files and with databases and more. Again, these chapters all assume that you have NetBeans installed and you are running the ConcX source code.

Part 2 examines the main program modules and studies the code they contain and their relationships to each other. Armed with this information, the general procedure of developing a new application is described.
There is no requirement that the chapters in Part 1 be read in sequence. However, it is recommended that this chapter (Chapter 1) be read all the way through just to properly set your expectations. This first chapter should be enough to determine if ConcX and Avian Computing are sufficiently interesting or valuable that you want to learn more.

Chapters 5 – 15 were written to be stand-alone chapters and don’t depend on the information presented in earlier chapters. However, be aware that the earlier chapters will have more how-to information, such as more screenshots about how to set up a scenario. The later chapters assume that you’ve become acquainted with ConcX and therefore don’t provide screenshots for every selection or step by step instructions for simpler operations, such as loading a bird file.

**A First Example of Parallel Processing**

**Discussion**

Let's look at the AddBirds scenario (one of my favorites). It is the parallel equivalent of the classic “hello world” program in c. In this scenario, a group of birds work in parallel to perform a sequence of math operations on a series of food pods (values). In this scenario, the following actions are all performed in parallel:

- The Add1Bird finds and eats the right kind of food pod, adds the numerical value of 1 to the value contained in the food pod and then stores it as a different kind of food pod. It then takes a nap and then repeats this process until stopped by old age or the stop button.

- The Add2Bird finds and eats the right kind of food pod (the kind that Add1Bird stored) and adds 2 to the value contained in the food pod and then stores it as a different kind of food pod. It then takes a nap and then repeats this process until stopped by old age or the stop button.

- The Add3Bird finds and eats the right kind of food pod (the kind that Add2Bird stored) and adds 3 to the value contained in the food pod and then stores it as a different kind of food pod. It then takes a nap and then repeats this process until stopped by old age or the stop button.

- The Add4Bird finds and eats the right kind of food pod (the kind that Add3Bird stored) and adds 4 to the value contained in the food pod and then stores it as a different kind of food pod. It then takes a nap and then repeats this process until stopped by old age or the stop button.

- The Add5Bird finds and eats the right kind of food pod (the kind that Add4Bird stored) and adds 5 to the value contained in the food pod and then stores it as a different kind of food pod. It then takes a nap and then repeats this process until stopped by old age or the stop button.

A food pod that has been processed by all five birds will have a value of 15 (1+2+3+4+5). The details of how all of these birds work together in parallel is discussed in more detail in
Chapter 3 (Avian Computing Concepts) but the important bit to get from this description is that each bird is running asynchronously and doing its own thing, eating whenever it can without checking or coordinating with any of the other birds in its flock. See the following illustration. Each bird is effectively isolated from all the other birds, flying without any central control thread, processing FoodPods asynchronously without any coordination. All a bird knows is whether or not the food that they want to eat is available in the TupleTree.

While this scenario may seem trivial and contrived, it actually provides a simple template of how more complicated processes can be decomposed into sequences of simple atomic operations that can be performed in parallel. For example, we could replace Add1Bird with FastFourierTransformBird and Add2Bird with InvertAndFlipBird, etc, and develop an extremely sophisticated sequence of math calculations that are all performed in parallel that operates on a stream of data.

And ConcX is not just for math calculations. This scenario could just as easily represent processing the customers in a billing cycle, where each bird performs only one part of the total process, such as one bird looking up a customer’s billing address while another bird is calculating a different customer’s usage minutes while another bird is calculating the tax for monthly services, etc., all being done in parallel.
Procedure

After downloading and unzipping or gunzipping the concx.zip file in any desirable folder, it will contain concx.jar plus three folders: lib, data, and flocks. See Chapter 2 for details.

At a command window or terminal window in the folder where the concx file was unzipped, type `java -jar concx.jar Nimbus`

After some info is displayed to the window, ConcX will be displayed as follows:

![ConcX Interface](image)

This is the main screen of ConcX. The birds are added and defined on the left and the results of the birds flying are displayed on the right.

**NOTE**

If you are running Windows, you may see a bunch of messages similar to “Qt: Untested Windows version 10.0 detected!” and log4clus error messages.

It is safe to ignore these messages. These messages don’t appear in Linux.

**NOTE**

The Nimbus parameter following “concx.jar” sets the Look & Feel of ConcX. If you omit the Nimbus parameter, the System Look & Feel will be used. You can experiment with these different appearances as desired by supplying different Look & Feel names such as “Metal” or “Motif” as the parameter.
For this document, the Nimbus Look & Feel will be used for all screenshots.

Do the following steps to run your first parallel Avian application:

1. In the top left corner, click on File → Open Flock File. The following is displayed:

   ![Open Flock File Dialog](image1)

   **NOTE**

   If the folder that contains the addbirds folder isn’t displayed, refer to [The Set Flock Files Location](chapter4) section of Chapter 4.

2. Navigate to the addbirds folder.
3. Select the add5.bird file and then click on Open.

   ![Open Flock File Dialog](image2)
4. When the birds have been loaded, five new tabs will have been added to the left side of the screen. These five new birds will also have been added to the list of birds on the Activity tab on the rights side of the screen, as shown below.

5. The Add5Bird (tab #5) will be selected. This bird eats a ChartreusePod any time it can find one, gets the running total from that pod, and then adds 5 to the value. It puts the new running total into a CoralPod and then stores the CoralPod back into the TupleTree (see Chapter 3 for info on tuplespaces and the TupleTree).

6. Click on the #4 tab and display the Add4Bird. It does the same thing as the Add5Bird except it eats BrownPods, adds 4 to the running total, and saves the new running total in ChartreusePods that will be eaten by the Add5Bird.

7. Same for tabs #1 thru #3, Add1Bird, Add2Bird, and Add3Bird. They add 1, 2, and 3 respectively to running total, resulting in a final total value of 15 (1 + 2 + 3 + 4 + 5) being saved in the food pods that the Add5Bird stores, the CoralPods. Click on those tabs if you want to see the color of the pods that the individual birds eat and store.

8. Click on Start All button at the bottom left of the screen.

All five birds will start flying. You’ll know that they’re flying because their individual Activity ProgressBars will increase each time each individual eats.
After 30 seconds the AddxBirds will all stop flying because they will each have reached the end of their individual lives of 30,000 milliseconds (each separately configurable).

The screen will look similar to the following:

1. Click on the TupleTree tab on the right side. The TupleTree contains all the food that is being processed by the birds. The screen will appear similar to the following:
2. The bottom line shows how many food pods have been stored in the TupleTree.

3. Drag the thumb (or knob or bar or elevator, etc) up to the top of the TupleTree History text box and you will see something similar to the following screenshot: (the left side of ConcX has been cropped off to better fit the dimensions of the page).

The first line of the History text box also shows the total number of food pods in the tree. In this example, we have 380 food pods, but the actual quantity isn’t important in this demo. Depending on the scenario being run, the number of birds, and various other configuration settings, different quantities will remain in the TupleTree after a run.
Scroll around in the History text box. Note that almost all of the pods are CoralPods and every CoralPod has a value of 15. Towards the bottom will be a few other color pods, each with the appropriate values (for example, all BlackPods will have a value of 1; all BluePods will have a value of 3, etc).

Also important to note is that every CoralPod has been eaten by all five of the AddxBirds in the desired order. Even though in this scenario we don't care about the order (because we’re just adding), this ordering would be critical if we were doing non-commutative operations.

*Note that there is absolutely zero control code that determines that the correct pod is processed in the correct sequence.* The desired ordering was achieved completely by configuring the food pods that each bird ate and stored and not in the code. The birds are all loosely coupled which makes it extremely simple to rearrange the order of operations without having to make any code changes. We’ll do just that in a moment.

Also note the times on the left side of the pod histories displayed on screen, formatted as hh:mm:ss:mmm (where mmm are milliseconds). We can see the exact millisecond of each operation that was performed on each pod as well as the bird that performed it. This allows us to look up each pod event in the history of the bird that performed the event and see if it recorded any errors or problems. More on this in a little bit.

Click on the Food Supply tab. It will look similar to the cropped image on the right.

The Food Supply tab shows in real time what foods are being eaten (or not eaten) while the program runs. When a bird eats, it eats one specific type of food and the ProgressBar for that type of food is *shortened* by 1 unit. When a bird stores food, it stores a specific type of food and the ProgressBar for that type of food is *increased* by 1 unit. The food ProgressBars will frequently appear to be flickering as food is being stored and just as quickly eaten.

The big winners after this first run are MagicPods and CoralPods (more about MagicPods later). There aren’t many in-between pods (Black, Blue, Brown and Chartreuse) but this is expected.
In this scenario, the Add1Bird is busy storing BlackPods (increasing the BlackPod ProgressBar) while the Add2Bird is just as busy eating BlackPods (decreasing the BlackPod ProgressBar) with a net growth on the BlackPod ProgressBar of approximately zero. The same goes for the Add2Bird: just as quickly as it is storing BluePods, the Add3Bird eats them, again with a net growth of about zero. And so on for all five birds until the Add5Bird which stores CoralPods that no bird eats, resulting in a continuously growing CoralPod ProgressBar.

So why are there a few in-between pods that didn't get eaten? This is due to the built-in randomness that adjusts how long each bird naps, which in turn affects how frequently they check for food pods. The birds all run independently from each other; depending on each bird's individual “luck” there will typically be a few leftover pods.

Let's clean them up, just to demonstrate how to run one bird at a time.

1. Click on tab #2, the Add2Bird, the one that eats BlackPods.
2. Click the Start Me button on tab #2.
3. The BlackPod ProgressBar will shrink to zero and the BluePod will grow. After 5 seconds of not eating (configurable), the Add2Bird will stop. Or you can click the Stop Me button after the BlackPod ProgressBar reaches zero.
4. Click on tab #3 and click the Start Me button. When the BluePod ProgressBar reaches zero, click the Stop Me button.
5. Repeat on tabs #4 and #5.

After completing the above steps, the Food Supply tab will look similar to the following screenshot, with only MagicPods and CoralPods showing anything on their ProgressBars. Note also that both ProgressBars show approximately the same amounts, just less than 400.

Now click on the TupleTree tab again. Note that there are still 380 pods in the tree, matching the value of just less than 400 shown on the Food Supply tab. The reason why the total number of pods is unchanged is that Birds 2 – 5 didn’t add any pods to the TupleTree; instead they ate all food pods for which they were configured and processed them until they were all CoralPods with a value of 15.
Now let’s rearrange the order that the birds perform their operations.

1. Click on the #1 tab. It will appear similar to the screenshot on the right.

2. Click in the Food Stored Food Type field, which highlights BlackPod.

3. Type “ch” and press tab.

4. ConcX tries to match “ch” to the name of a food and finds multiple foods that match “ch” so it displays the Duplicate food selector dialog box (see figure below).

5. Click on the dropdown arrow to view other matches. A couple of Cherry Cola foods will be displayed.

6. We want Chartreuse so click on it and then click on the OK button. ChartreusePod appears in the Food Type field, properly capitalized.

7. Click on Tab #2 and click in the Stored Food Type field.

8. Replace BluePod with “c” and press Tab. The Duplicate Store Food selector appears with a list of foods that begin with “C”.

9. Click the Cancel button. The Food Type changes to “Not Found”.

10. Replace “Not Found” with “cor” and press Tab. Since only one food pod begins with “cor”, CoralPod will automatically be selected.

11. Select Tab #3. Leave its Food Stored Food Type to BrownPod.

12. Select Tab #4 and change its Food Stored Food Type to BlackPod.
13. Select Tab #5 and change its Food Stored Food Type to BluePod.

14. In the bottom right corner of ConcX, click on the Clear All Activities button.

15. In the top center of ConcX, click on the Activities tab.

16. In the bottom left corner of ConcX, click on the Start All button. All of the Activity progress bars will start growing.

17. While the birds are flying, click on the Food Supply tab. Note how all the progress bars are flickering (growing and shrinking) as foods are stored and eaten.

When all the birds stop flying, click on the TupleTree tab. Scroll to the top of the History text box. It will look similar to the following screenshot:
Notice how every CoralPod equals 15, just like before. But this time the order that the operations were performed in is completely different than before. Every CoralPod has the identical (and new) sequence of events recorded. Don’t like that order? Change the order again in 2 minutes. It’s just that easy. Want to add another operation? If the bird exists, just click the Add New Bird button, select that type of bird, assign it the desired foods and run them all again.

Imagine trying to completely reorder your typical program’s sequence of operations when you’re coding in the conventional way, with modules and Objects and traditional lines of code. Now imagine you discover that you need to calculate the square root of negative infinity before calculating the inverse of perpetual motion. I guarantee it will take longer to revise the order of operations using normal code than it takes in ConcX.

Let's kick it up a notch.

1. Click the Clear All Activities button in bottom right corner.
2. Put all the stored foods back to the way they were before. Tab #1: stores BlackPod, Tab#2: stores BluePod, Tab#3: stores BrownPod, Tab #4: stores ChartreusePod and Tab #5: stores CoralPod.
3. Click on File->Open and navigate to the addbirds folder and open the add5.bird file, the same one that was opened before.
4. On the left hand side, change the Names on Tabs #6 - #10 by replacing the final “a” in their Names to a “B”. For example, “a5a” becomes “a5B”.
5. Click on the Start All button.
6. As before, the Activity tab shows the ProgressBars for all of the birds increasing. Note that all 10 birds are flying.
7. Click on the Food Supply tab. The MagicPod and CoralPod ProgressBars will be increasing rapidly while the BlackPod, BluePod, BrownPod, and ChartreusePods will all be flickering as they grow and shrink in rapid succession.
8. When all the birds stop (or after you click on the Stop All button), click on the TupleTree tab.
9. Scroll up to the top and it will look similar to the above screenshot. It
will also look similar to the previous results except that twice as many pods are in the TupleTree, which makes sense since there are twice as many birds flying as before.

A couple of items to note: first, even though there are twice as many birds working as before, every CoralPod is still equal to 15 because each CoralPod has been processed by only one Add1Bird, one Add2Bird, one Add3Bird, one Add4Bird and one Add5Bird.

The next item to notice is which birds actually processed each pod. In the first pod listed above, it was processed by bird “a1B” and then bird “a2B” and then bird “a3B” (all part of the second group of AddxBirds loaded) but then it was processed by “a4a” (from the original group of AddxBirds loaded) then by “a5B”.

The next pod however was processed by a different mixture of AddxBirds, mostly from the first group loaded. Not only was an arbitrary number of additional birds added to the scenario, there doesn’t seem to be any favoritism shown to either the earlier loaded birds or the last birds loaded.

Scroll down thru the CoralPods in the TupleTree and they will show a semi-random pattern of processing but always processing each pod exactly 5 times and always adding in exactly the same order but not by the same sequence of individual birds. This demonstrates that the birds are loosely coupled and eating any available food pod of the correct type and not showing any special favoritism or preference.

Now let’s look at how to use the timestamp on the left side of the TupleTree history screen.
The screenshot above shows that the first listed food pod was eaten at 14:50:11:383 by bird a5B, an add5Bird displayed on tab #10. By clicking on the History tab of tab #10, we can track its startup events that were followed 3 failed attempts to eat, followed by when it successfully ate a ChartreusePod at 14:50:11:383. If there were any unusual circumstances related to this eating of the food pod, it would probably be listed in the bird’s history, allowing you to research unexpected or inconsistent results.

Note that this time and event information is always collected for food pods and for each individual bird. The histories are always collected to avoid situations where an anomaly happens when logging is turned off and when logging is turned back on, the anomaly can’t be recreated.

And if more history information about an individual bird is desired, the History tab for each bird can have its Level of Info adjusted so more information can be recorded using familiar sounding logging levels (Severe, Warning, Info) or reduced information using (Fine, Finer, and Finest). Config is the default history level.

The final advantage of Avian Computing in ConcX is the reduced amount of code that needs to be written to implement parallel programs. Here’s the complete code for the Add1Bird:

```java
public class Add1Bird extends BasicBird {
    /**
     * Overrides afterDigestion in BasicBird with the instructions on how to add
     * 1 to the value of the food that it ate. This bird expects the value being
     * summed to be kept in the contentsString instead of the contents Object.
     * If digestFood1 is non-null, it tries to add 1 to that one's
     * contentsString. If digestFood2 is non-null, it tries to add 1 to that
     * one's contentsString.
     */
    @Override
    protected void afterDigestion(int eatResults) {
        int contentsValue;
        if(eatResults == PUT_BACK_1 || eatResults == PUT_BACK_FILE_1) {
            return;  //don't update the value
        }
        //uncomment the following block if you wish to automatically generate errors for testing
        /* if(ThreadLocalRandom.current().nextInt(1, 11) == 3) {
            mPod1.setEdible(false);
            //mPod2.setEdible(false); //uncomment this line if ALWAYS be eating food2
            System.out.println("afterDigestion: Random value = 3");
        } */
        if (!mPod1.isEmpty()) {
            try {
                final String localContents = mPod1.getDesc();
                contentsValue = Integer.parseInt(localContents);
            } catch (NumberFormatException nfe) {
                mPod1.addToPodHistory("Non-numeric value in contents");
            }
```
A couple of blanks were removed but the functionality for each of the five AddxBirds required only 45 lines. If you skip commented out lines, it’s only 31 lines. If you break coding style standards and move stand-alone curly braces onto lines with code, the code is only 24 lines long.

Understanding the code details isn’t relevant at this point. The important detail to get from this code is that to create a new bird doesn’t mean copying large blocks of boilerplate code or digging into the guts and modifying some existing code. Instead, by extending BasicBird, all of the behaviors described earlier are automatically available.

New birds extending BasicBird have access to a large number of empty methods that can be overridden and modify the behavior of the bird without modifying the code of BasicBird. Need to perform a special one-time action before starting the bird? There’s a method that can be overridden. Need to perform some special preprocessing of the retrieved info before it is digested? There’s a method for that. Need to replace the food retrieval process completely? There’s a method for that and a host of other methods. Ideally, just override the necessary methods and your new bird is complete.

Notice what’s missing? There is a complete lack of parallel programming logic normally needed to run Objects in parallel. Instead of forcing developers to consider parallel code requirements at the same time they’re trying to write the application code, all of the parallel code is pushed into the TupleTree and BasicBirds, borrowing heavily from the concepts pioneered by Linda and JavaSpaces. It is so much easier to think about what you need to do in parallel if you don’t also have to think about how you need to do it in parallel. Refer to Chapter 3 for more details on how the parallel code is managed in ConcX.
**Wrap Up**

This first chapter introduced Avian Computing and demonstrated how to run parallel programs using ConcX. The goal of this chapter was to properly set the reader's expectations for ConcX with a minimum of intellectual investment by the reader.

Another goal of this chapter was to demonstrate how programs can be developed using a flock of birds as a mental model for the desired parallel program. The mental model of a flock of birds looking for food in a small tree is easy to imagine because it is a familiar scene in nature. This familiarity allows us to zero-in on the important details, such as what food to find and what is done after the food is found. In the demonstrated AddxBirds scenario, a simple addition was performed, but it could just as easily have been much more sophisticated math operations.

The mental model of a flock of birds also lets us gloss over some critical details so we can concentrate on the big picture. In this chapter’s example there was never any discussion of synchronizing or semaphores or preventing deadlocks or any of the other parallel programming baggage. Those are implementation details that shouldn’t control how we think about a solution. Once we have a clear picture of what we want to do, we can implement those parallel-enabling details more easily.

Other important details that the birds model allowed us to gloss over was starting separate threads instead of running on Swing’s event-dispatch thread or terminating threads, etc. Conceptualizing the individual threads as birds allowed us to use natural terms like birth and death to clearly understand what is happening without having to know any of the gory details of how it is done.

Although ConcX is written in Java, ConcX is intended to be language agnostic. ConcX is a thinking tool, one that leverages nature-based abstractions to help us to think about parallel solutions, helping us to think about how the components (birds) will interact and their dependencies. Once we know how the birds (threads) should behave, we can implement our final solution in any language that can support multi-threading parallel programs.

ConcX provides an environment where it is cheap and quick to experiment with how to modularize our code. Among its goals is learning which actions should be grouped together as an atomic group and which ones should be broken into smaller atomic groups and to find out as easily and efficiently as possible.

Execution speed is not the goal of ConcX but instead thinking speed. We probably have all gotten deep into the development of a program only to discover that it couldn’t be done the way we were planning to do it, requiring a major rewrite. There is no substitute for gaining the profound understanding the application’s requirements that is gained by actually getting a version running.

If ConcX doesn’t sound right for you, don’t waste any more time with it. And be glad that you didn’t have to spend a long time figuring out that ConcX doesn’t meet your needs.
If you’re not sure about ConcX, skip ahead to Chapter 5, AddxBirds Revisited, and read through it. Not only does Chapter 5 provide multiple variations of the original add5.bird scenario, it demonstrates how easy it is to access ConcX’s dynamic performance behaviors can be used to “right-size” the number of threads in your application.

However, if ConcX already sounds like it could be of use, install ConcX (Chapter 2) and then mess around with the examples in Chapters 5 - 15. Chapter 5 is recommended to be reviewed next, but after that, look for chapters that interest you and do those next.

Load the flocks and run the examples and then change some of the configuration settings (see Chapter 4) and re-run the examples. Go ahead and mess around with stuff and then refer back to Chapter 3 when you want to peek behind the curtains more. Go ahead – explore and have some fun. I won’t tell.
Chapter 2. Installing Concurrency Explorer (ConcX)

ConcX can be downloaded in two formats: as a jar file and as source code. Download the jar file when you wish to run ConcX with its existing birds and foods. Download the source code when you wish to view the code, modify existing birds and foods, create new birds and foods, or wish to use the debugger to step thru the code.

This chapter covers both ConcX jar installation and ConcX source code installation. This chapter tries to avoid duplicating the instructions provided for installing other software packages. Always refer to the provider's installation instructions for the correct procedures to follow and setup other software packages.

Tested and Untested Configurations

ConcX has been developed and tested using NetBeans running on Windows 7 & 10 and Linux (Ubuntu 15). The majority of development was done on a Windows computer, so some Windows-centricity will probably be noticed. NetBeans was selected when this project started because it offered the best free Swing GUI Builder.

The ConcX source code can be built and executed from the Eclipse environment but it configures the build directory differently which throws off some of the settings so it can’t find the birds or the foods. However, I’m cautiously optimistic that the problems are not insurmountable.

ConcX on Mac has not been tested at this point. I have no excuses for not testing it other than a chronic shortage of time.

And then maybe someday I’ll get with the times and upgrade to Java FX. Maybe.

Software Dependencies

ConcX is built using Java 8 jdk, although it should be fairly compatible with Java 7 because I don't remember using any Java 8-specific language features. If possible, and you are not prohibited for configuration and/or compatibility and/or corporate reasons, I recommend that you run the latest version of Java 8, either the JRE (to run the jar version) or the jdk (to modify, compile, and run from IDE), both available for free from Oracle. If necessary, download and install Java following the instructions provided on Oracle's website.

As discussed above, ConcX is built using NetBeans, which is also available for free download from Oracle. If you need both Java and NetBeans, look to see if you can find a version that contains both the latest Java jdk and the latest NetBeans for your operating system and hardware – it installs both Java and NetBeans at the same time, simplifying the install process. Follow Oracle's instructions for installation.

The Guava jar from Google is also required because it offers the ArrayListMultimap object used by the TupleTree. Guava (2.5MB) offers a huge number of capabilities that are unused by ConcX, so a lighter replacement object may be sought.
**Downloading ConcX**

ConcX can be downloaded from [www.aviancomputing.net](http://www.aviancomputing.net). Look for the “Download ConcX” link on the website. This will download a zip file that includes the concx.jar file, all necessary lib jars, and the flocks and data folders with sample files. This site also contains the javadocs for ConcX as well as the latest version of this guide.

Download latest stable jar file of ConcX and support libraries, download it from:

1. Aviancomputing.net
2. [https://sourceforge.net/projects/aviancomputing/files/](https://sourceforge.net/projects/aviancomputing/files/)

Download latest source code from:

- [https://sourceforge.net/p/aviancomputing/code/HEAD/tree/](https://sourceforge.net/p/aviancomputing/code/HEAD/tree/)
  - Choose “Download Snapshot” if you want the all of the latest source code in one zip file
  - Use the provided svn command to use Subversion to download specific files, branches, specific revisions or all of the source code

**Concx.jar Zip File Instructions**

1. Follow one of the links above to either the Avian Computing website or the sourceforge website and click on the link to download the ConcX.zip file.
2. Select an empty folder or create a new folder that will be used for ConcX.
3. Unzip the zip file into your folder.

When it is finished, the folder should have the concx.jar and three folders: com, data, and flocks. The com folder contains all the .class files. The data folder has a few sample data files that can be read into ConcX. The flocks folder holds the sample .bird files that you can open to load preconfigured collections of birds that can be flown.

In Windows, you can:

- Double-click on the concx.jar file OR
- Open a cmd window in that folder and then type this command
  ```java
  java -jar concx.jar Nimbus
  ```

In Linux:

- Open a terminal window into the folder and type the same command
  ```java
  java -jar concx.jar Nimbus
  ```

**Download Code Snapshot of Source Files**

1. Point your browser to [https://sourceforge.net/projects/aviancomputing/](https://sourceforge.net/projects/aviancomputing/)
2. On the menu bar, click on the Code option.
3. Select the Download Snapshot option. The latest version of ConcX will be zipped together and then will download them to your computer.

Once the files have downloaded:

1. Select an empty folder or create a new folder.
2. Unzip the source files into the new folder.
3. Skip the next section and continue with the following section titled: Setup New Java Project with Existing Sources if you are using NetBeans. If you are using Eclipse or a different editor to edit and compile Java, it should be pretty similar. Let me know how it works out.

Checking Out ConcX Source Files

If you are familiar with Subversion version control software and have it installed, you can navigate to the location where you want to put the checked out files and then issue this command:

```
svn checkout svn://svn.code.sf.net/p/aviancomputing/code/trunk avian-code
```

Note that this command will put all of the code into a new folder named “avian-code” under the current folder. If you want the code to go into a different folder, replace “avian-code” with the folder name (and path) of your choice.

If you are using a GUI front-end for Subversion such as Tortoise, navigate to the location where you want the code to go and then follow the tool’s procedures to check out the source code.

In the example on the right, the URL of the repository is already entered, but if you want to browse to a different location, such as one of the branches, you can click the 3-dots button and navigate to the desired folder. When you click OK button, the source code will be copied from the repository to the checkout directory.

**IMPORTANT!** Checking out the source code will not grant you permission to check in any of your changes. Please contact the author if you have code changes, bug fixes, or improvements that you want to contribute to the Avian Computing project.
If you have Subversion turned on in NetBeans, you can also use NetBeans to check out a copy of the source code.

1. In NetBeans, select the Team menu
2. Select the Subversion and then the Checkout option.
3. The popup screen on the right will be displayed.
4. Type in the Repository URL – this will be the same URL used in the previous checkout examples, except without the .../trunk folder. This final folder will be set on the next screen.

Because anonymous access is allowed, the User and Password fields should be blank. Click the Next button and a new screen appears.

Click on the Browse button to select the desired Repository Folder. Choose trunk if you want the latest. Choose one of the branches if you want to checkout code from one of the branches.

Click the Skip “trunk” checkbox so the source code is copied directly into the local destination folder. If you don’t click it, you will have a folder named “trunk” created in the local folder with the source code under the trunk folder.

Click the Local Folder Browse button to select the desired folder. When everything has been
set appropriately, click the Finish button. The ConcX source code will be copied to the local folder you specified.

NetBeans offers to create a new project using the new source code but I've never had much luck with this option. Instead, I click No and then when all copying is finished, I follow the procedures in the following section.

**Setup New Java Project with Existing Sources**

Now that you have a full copy of the ConcX source code on your local system, it is time to create a new NetBeans project to access that source code as follows:

1. Click on the File menu and select New Project. The following screen is displayed.
2. Click on the Java Project with Existing Sources option and then click on the Next button.
3. When the next screen is displayed, update the Project Name if desired. You can leave it with the generated name or change it as shown below.

   **NOTE**

   Make sure you don't put the new project inside the same folder as the source code or NetBeans will complain.

4. Click the Next button.
5. When next screen pops up, click on the Add Folder... button for the Source Package Folders and navigate to the folder with the source code you unzipped or checked out.

6. Click the Next button.
7. When the next screen appears, it should list a bunch of files in the Included Files field. If not, make sure you selected the correct source code folder.

8. Click the Finish button.

NetBeans will grind away for a little while and then display the new project. When you expand the Source Packages folder, it will look similar to the following screenshot, complete with the red error flags. The error flags are shown because we need to add a couple of jar files to the project.
Adding Jar Files and Libraries to the Project

The first jar file to add is the Guava.jar file from Google. This file contains the ArrayListMultimap object that is required by the TupleTree.

To add Guava.jar, do the following:

1. Right-click on the Libraries folder to display the context menu.
2. Click on the Add JAR/Folder option.
3. Navigate to the location of the Guava jar file which was distributed with ConcX.
4. The red flag on com.avian.tuple will go away when the Guava.jar file has been correctly added.
5. Two JUnit libraries need to be specified to remove the red flags from the test folders.
6. Right-click on the Libraries folder to display the context menu.
7. Click on the Add Library . . . option.
8. The Add Library pop up is displayed.
9. Select the Hamcrest 1.3 library and then click the Add Library button.
10. Right-click on the libraries folder and select the Add Library. . . option from the context menu.
11. Select the JUnit library option and then click the Add Library button.
12. If all has gone correctly, all of the red flags should be gone.

Set the Main Class

Set the location of the Main Class as follows:

13. Select the Run menu.
14. Select the Project Configuration menu option.
15. Select the Customize option.

The following screen will display.
1. Click on the Browse . . . button to the right of the Main Class field.
2. Click on the `com.avian.gui.Aviary2` option (second screenshot following).
3. Click the Select Main Class button.
4. **Optional:** Enter the desired Look and Feel in the Arguments field. For example, enter “Nimbus” or “Metal” or “Motif” to use a different Look and Feel instead of the default “System” Look and Feel. All screenshots were made using Nimbus Look and Feel.
5. Click the OK button.
Press the F6 key or click the Run menu and then click the Run Project option.

If you see the above screen, your ConcX is running. If it fails and gives you an error message, go back thru the instructions and double-check them.

If that doesn’t resolve your issue, make sure that you are running Java 8 as follows:

6. In the NetBeans left-hand panel, select the Project tab (if required).
7. Right click on the project folder name
8. Select the Properties option at the bottom of the pop up menu.
9. On the Project Properties dialog, in the left-hand panel, select the Libraries option.
10. Make sure the Java Platform for ConcX is set to JDK 1.8.

If the above doesn’t resolve the issue, please contact support@aviancomputing.net
Chapter 3. Avian Computing Concepts and ConcX

This chapter introduces the fundamental concepts of Avian Computing and describes how these concepts are implemented in ConcX. It is intended to be a brief primer on Avian Computing, why Avian Computing got started and its intended goals without diving into the subject too deeply. For those who wish to know more about Avian Computing, please go to https://www.aviancomputing.net/.

What Is Avian Computing?

Avian Computing is a tool to help us think about how multiple threads (or processes or CPU's) can work together to accomplish our programming goal. Avian Computing replaces the code-based model of computer programs with a model based on a “Flock of Birds”. Avian Computing is like training wheels for our brains. Avian Computing is like the bumpers in the bowling alley gutters; it tries to keep our minds focused on how to express a programming solution in parallel instead of falling into the gutter of trying to make the program compile and thread-safe.

Why Avian Computing?

Avian Computing began with a moment's insight: Maybe the reason we don't have more parallel programs is because the methodology that we use to generate parallel programs is obsolete. It certainly isn't because we don't have enough smart people to create parallel programs and it isn't because we don't have parallel software/languages (Java had threads and multi-tasking capabilities since it was introduced in 1995).

Maybe it's just too hard to create parallel programs the old-fashioned way, the code-based way. Maybe if we start over and consider the strengths and weaknesses of humans we can find a more effective way of developing parallel programs.

Avian Computing Is a Simplifying Abstraction

Computing is full of abstractions; 1's and 0's are an abstract way of expressing the voltage states inside transistors, ASCII is an abstract way of expressing numbers and letters in a computer, assembly languages are abstract expressions of sequences of code and data (machine language) that are processed by the CPU, high-level programming languages are further abstractions for the machine language processed by the CPU.

Avian Computing is yet another abstraction but one that makes no pretense of directly representing computer operations. Instead, Avian Computing abstracts away the linear thinking necessary for coding and replaces it with images of parallel processes, with visualizations of simultaneously operating objects.

In advanced mathematics, there are times where a particular equation or proof is too difficult to solve as currently expressed. In such situations, the mathematicians will look for simplifying expressions; look for ways to replace an unsolvable portion with expressions.
that they know how to solve. Many times the replacing expression is only approximately right but it’s close enough and that them to solve the bigger problem.

Avian Computing is intended to fit in as that kind of simplifying expression, as the piece that can be more easily solved. Instead of struggling with sequential lines of code that has consistently proved to be difficult to develop, Avian Computing provides a simpler and easier model in which to express a parallel solution.

**Is Avian Computing Really Necessary?**

Nope. Not at all. If you're happy with making slow, unpredictable progress and lots of hours pursuing elusive bugs, then there's nothing wrong with using the old-fashioned methods. If you enjoy difficult logic puzzles, then continue to try to solve them with the tools you grew up with.

Avian Computing isn't some cutesy veneer randomly selected to drape over last century's techniques. It is a model that was specifically selected to address the deficiencies in human brains that make parallel programming difficult. Avian Computing is a model that tries to side-step the limitations of programming in computer languages/code.

The next two sections describe the structure, operation and shortcomings of the human brain.

**Brains Work in Parallel, Logic Is Single-Threaded**

The brain is a marvel of multi-processing, with various portions of the brain dedicated to different functions. For example, our vision system is extremely sophisticated and powerful, with the visible light detected in our eyes converted into nerve impulses that are transmitted to the Occipital Lobe at the back of the brain (in pink in the image below) where it is converted into the shapes and object that we recognize. At the same time the Cerebellum is automatically managing our balance and posture while our Brain Stem is maintaining our breathing, body temperature, blood pressure and heart rate.

These brain systems and many more all work in seamless and nearly invisible harmony for most of us (most of the time). The one exception to this invisible multi-tasking is the Frontal Lobe, the portion of the brain associated with problem solving and creative thinking. This is the part of the brain that we associate with ourselves, our personality, our command center, the part of the brain that gives us intention, logic, and free-will.

Unfortunately, the Frontal Lobe is effectively single-threaded. We like to think we can truly multi-task but we can only time-slice; we can only focus our brain's attention on one subject at a time before jumping onto
the next subject. Imagine being able to read a book at full speed at the same time we were talking on the phone and playing a game of chess and giving each our full attention. That would be true multi-tasking. But we can't.

Not convinced? Take a look at the image on the right. Do you see a candlestick or two people face to face? Can you see both at once? Most of us can only see the candlestick OR the faces but never both at the same time. For a more complete discussion of the single-threadedness of human thinking, refer to www.aviancomputing.net.

**Programming Languages Are Processed in the Frontal Lobe**

The reason that the single-threadedness of our Frontal Lobe is important to computer programming is that computer programming, as currently practiced, is purely a Frontal Lobe activity. The Frontal Lobe interprets the meaning of the code words that we are reading and translates them into a mental model of what we believe will be the sequence of operations that the computer will perform. Understanding what a computer program will do is strictly a logic exercise, an exercise that is performed in the Frontal Lobe.

Single-threadedness wouldn't be an issue if computer programs never branched, if they never tested for any conditions and responded accordingly. Such computer code wouldn't be very useful but it sure would be easy to understand.

Instead, computer code is full of conditional statements and branches. And every time we read a conditional, our single-threaded Frontal Lobe has to try to completely think thru and understand every step of each branch, remembering all the details that have been covered in branch so the details in the other branch are not incorrectly processed. Our brains can't think down both branches simultaneously but instead has to jump back and forth between the different branches, just like when we are time-slicing and trying to multiple things at the same time.

If you think this issue is non-sense or inconsequential, consider that one of the measures of code complexity and its probability of containing errors is the number of conditional statements and branches in a code module. And the importance of branches is not because the computer has trouble with the branches, it's because humans have trouble with program branches because of our single-threaded Frontal Lobes.

Now consider the article published by George A. Miller in 1956 titled, “The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information.” In this article, Miller argued that the average human can hold 7 +/- 2 objects in working memory. Other researchers since then have studied the issue and come up with similar results, some specifying that we can hold fewer chunks but each chunk can hold multiple pieces within a category, etc. But the general consensus is that our short term memory or working memory is pretty limited.

And if each conditional has two branches, it only takes 3 conditionals to nominally exceed
the magic number 7 and 4 conditionals produces 16 different branches that have to be processed by our single-threaded Frontal Lobe, significantly exceeding the number that our short term memories can process.

*The wonder of computer programming is not the amazing and sophisticated programs that have been created but that we are ever able to produce code that runs.*

And then when you add the additional unpredictable asynchronous interactions of the components of true parallel programs, it's no wonder that developing parallel programs are usually approached with a sense of dread and uncertainty.

**What Kind of Parallel Developer Are You?**

Most developers approach parallel programming in one of two ways:

1. They develop the basic program single-threaded and then, if it doesn't run fast enough, try to bolt on some of that multi-threaded stuff later.
2. They work thru a lot of simple parallel program examples from a lot of different sources and then try to adapt the example code that most resembled the parallel program they think they will need.

Personally, I've failed at both ways of developing parallel programs, all the while thinking that it was harder than it had to be, always too aware that too many unseen dangers awaited me. It was like playing chess against a Grand Master who keeps saying helpful things like, “That move will put you in checkmate in 7 moves.”

**In Search of a Better Paradigm**

Philosophers, psychologists, and politicians all insist that how we frame an issue determines the solutions of which we can conceive. Our personal frame of reference determines how we interpret what we see and that interpretation determines what we believe can be done or how it can be done.

Imagine we are trying to figure out how to get to the freeway. We look out the window and it looks like we're going to have to drive down a steep hillside, cross a river, and then climb back out of a ravine to get to the freeway. But if we look out a different window, we can see that there is a road right there that leads to the freeway.

What we consider as the correct solution from one window may be a totally different from the correct solution viewed from a different window. But if we always look out the same window, the window of lines-of-code and pure abstract logic, we will always end up proposing the same solutions and always encounter the same difficulties. It’s time to try a different window.

So the search began for a frame that naturally embodied parallelism, something that would allow me to simply think about performing operations in parallel. It occurred to me that the simplest examples of parallel operations were natural phenomena such as flocks of birds,
schools of fish, ant hills, and so on. I like birds so the Avian Computing project began. The “frame” of a flock of birds provides a familiar perspective to think about parallel operations. It is easy to mentally “zoom in” on the behavior of one bird and just as easy to “zoom out” and consider the behavior of the complete flock. Thinking about a flock of birds automatically frames our thinking as parallel operations while remaining easy to visualize and familiar conceptually – nothing new to learn.

**Mapping Computer Operations into Bird Behaviors**

In Avian Computing, the flock of birds is used as a metaphor for parallel program. This metaphor is emphasized and enhanced by mapping regular computer operations into the regular events in a bird's life. For example, a bird hatching is a thread starting, a bird napping is a thread sleeping, and a bird dying is a thread stopping.

While this mapping threatens to injure all of us with a near-fatal case of excessive cuteness, the mapping makes it possible to talk and think about parallel programs easily. But Avian Computing wouldn't have been anything more than a cute concept without Linda, a coordination language for concurrent program originally developed by David Gelernter and aided by Nick Carriero and others in the 1980's.

Linda uses three operations,” rd”, “in”, and “out”, to work on items in a global associative memory called a tuplespace. The in operation removes a tuple from the tuplespace, rd reads the tuple but doesn't remove it, and out writes the tuple into tuplespace. Linda formed much of the underlying structure for JavaSpaces, Jini and IBM's Tspaces.

Avian Computing borrows many concepts from Linda but doesn't try to implement all of its functions, such as associative matching, expiration time set and support for transactions. Instead, Avian Computing implements the functionality needed to support the bird behaviors of eating and storing food in the TupleTree, a stripped down version of tuplespace.
The Avian Lifecycle in ConcX

With the above concepts and tools, the lifecycle of every bird can now be defined:

- Hatch
- Look for Food
- Digest the food
- Store any resulting food
- Take a nap
- Reproduce when sufficiently well-fed
- Die when too old or starved

The Avian Lifecycle is shown in the drawing on the right.

Every bird in Avian Computing follows the same lifecycle. All of the functionality in their lifecycle is already coded in the BasicBird and AbstractBasicBird. The TupleTree object performs all of the Linda operations, allowing the birds to eat.

Some of you are of course curious about birds storing their food. Crows and ravens are well known to cache any extra food that they have, often going to great lengths to ensure that their fellow ravens don't know where they hid their food. Blue Jays, chickadees, nuthatches and more will also cache food to retrieve at a later time. Besides, storing the food for others to eat is conceptually much more pleasant than the other ways that digested food can leave the bird, especially if other birds are going to eat it.

The birds can be configured to find and store any kind of existing food pods. This means that developers don’t have to do any coding for the “Look for Food” and “Store Food” stages of the lifecycle unless there are unusual circumstances. So the digestion phase is where the developers will do most of their coding. During the digestion stage, any seeds (objects) contained in the food pod are removed and processed as desired by the developer.

For example, a curve-fitting food pod might contain multiple seeds with each seed containing a single point object \((x, y)\) coordinates. The developer just needs to write the code to draw the curves that fit the points contained in the seeds. Or each seed in the food pod could hold a term in a complicated math calculation.

The BasicBird is ignorant of the contents of any food pods that it eats. It is up to the developers to assign meaning to the contents of the food pod during digestion.
Advantages of Avian Computing using ConcX

By extending BasicBird, developers can leverage existing code to inherit a parallel processing environment. All the synchronizing code has already been written and implemented so developers don't have to. Developers can concentrate on developing the single-threaded code with which they are most familiar.

ConcX Handles Parallelizing the App so You Don't Have to

When a food pod is removed from the TupleTree, the bird that ate it doesn't share it with any other bird, just like in real life. The bird that ate it can do whatever it wants with that particular food pod. No other bird can affect that food pod because they don't know that it exists until the bird is finished with it and puts the pod back in the TupleTree.

ConcX Provides Asynchronous Message Passing and Decoupling of Objects

The food pods are normally eaten as one type and stored back in the TupleTree as a different type of food pod. This convention makes it simple to differentiate between data that needs to be processed and data that has been processed. And by designating different birds to eat different kinds of food pods, an asynchronous message passing protocol is implemented, one that is robust and highly decoupled.

ConcX Provides Visual Clues to Internal Status of Each Bird When Flying

Every BasicBird provides visual clues to developer to indicate its health and progress. It provides these visual clues in the form of ProgressBars that display the activity of each BasicBird and the amount of each type of food stored in the TupleTree. Developers can tell at a glance if one bird is eating too much or too little and if the foods are being consumed and stored as expected. BasicBird also implements all the code necessary to update the ProgressBars on the ConcX GUI screen.

ConcX Automatically Logs the Events at Configurable Levels

BasicBird also contains all of the code needed for logging the events of each bird. Each bird can be configured to have its own level of logging details based on java.util.Logging levels. When BasicBird is extended and new code is written, developers add as many logging messages at the appropriate logging levels as required to appropriately document the events that their new bird encountered. For example, if you want to verify that a specific method is being called, just add code like the following:

```
addToBirdHistory("beforeDigestion called", Level.FINER);
```

And every time the beforeDigestion method is called, it will be logged in any bird’s personal History that is set to the Finer Level of Detail.

ConcX Can Dynamically Adjust the Number of Birds (Threads)

BasicBird also has the code to test for starvation and old age and will have a bird die that doesn't get enough to eat (configurable) or lives too long (configurable). BasicBird also
contains the code for reproduction, with each bird duplicating itself when it has been successful enough (configurable) at finding food. These configurable controls allow ConcX to dynamically adjust the number of birds to match the current situation.

**ConcX Birds (Threads) Are Loosely Coupled**

All of the built in functionality reduces the amount of coding required, it is perhaps more important that birds in ConcX are very loosely coupled. Birds always find and store food pods and know nothing about the contents of the food pods until they digest them. If a step in the overall process needs to be added (or deleted), the changes can be made in seconds just by changing the foods eaten or stored. If the order of the steps in the process needs to be changed, the order can be rearranged just by changing the foods they eat or store.

**The Von Neumann Bottleneck and ConcX**

John von Neumann in 1945 defined the most common computer architecture in use today. He also identified a limitation on throughput caused by the computer architecture, the so-called Von Neumann Bottleneck.

One interesting aspect of Avian Computing and ConcX is that they side-step around the Von Neumann bottleneck. Because there is no master thread or scheduler thread in ConcX, it bypasses the bottleneck (at least to some extent). ConcX could run on a 100 core CPU or a 1000 core CPU without getting bottlenecked. The bottleneck is still there, but it is hidden inside each bird. Anytime you give a bird too much to do, it chokes on the excess and can’t go any faster. If you can split the excess with two or twenty more birds, the bottleneck goes away.
Chapter 4. ConcX Controls and Indicators

This is the basic ConcX screen. It has two primary panels. The left panel is used to manage and configure the birds you are using. The right panel is used to show the birds’ results, such as how active a bird has been or the amount of food eaten.

The Left Panel – Bird Management and Configuration Features

The left panel always has at least one bird tab available. As more birds are added, new bird tabs are added and numbered sequentially. Each bird tab contains five sub-tabs that categorize the controls, information, and settings for that bird.

The Bird Info tab is used to select the type of bird and the types of food that it eats and stores.

The Vitality tab is used to manage the life events of each bird, such as how long it lives and how long it survives without food.

The Code Info tab provides some high-level code details about the currently selected type of bird, such as its methods, constructors, and fields. It also shows the same info about the
bird’s immediate superclass.

The XMV tab provides 10 rows for Externally Managed Variables (XMV) that can be programmatically passed into each bird when it first starts.

The History tab contains the detailed history of the events that bird’s life, including the times when each of the events occurred. It also contains a Level of Info dropdown where you can select the amount of detail that will be recorded.

The Bird Info Tab

The Bird Info tab begins with only two fields enabled, the (Bird) Name field and the (Bird) Type field. The Name is optional but a valid Bird Type must be entered or the Food Eaten and Food Stored sections will not be enabled. A valid Eaten Food Type and a valid Stored Food Type must be selected to enable the Start Me button at the bottom of this tab. Until these three fields have valid entries, the bird cannot be started.

Select a Bird Type

To select a Bird Type, click in the Type field and enter the first part of the desired bird type, such as “add” for Add3Bird or “cons” for Consumer. Don’t worry about capitalization. When you press the Tab key, ConcX will validate what you typed against all the existing Bird Types. The field is updated as follows:

- If it doesn’t find any matches, the field is updated to read “Not Found”. Try typing the bird type again.
- If it finds one and only one type that begins with what you typed, the field will be updated with the properly capitalized Bird Type.
- If it finds more than one type that begins with what you typed, a dialog box will be displayed with a list of matching bird types to select from. Click on the dropdown arrow to view the complete list.
NOTES

- A bird with Type = “None” or “Not Found” cannot be run. If you want to keep a bird but don’t want it to execute in the next run(s), you can change the Bird Type to an invalid type and then it won’t execute.

Runtime Modes
ConcX can be run in two modes: Jar Mode and IDE Mode. If you downloaded the zip package that contains the ConcX jar file and start ConcX using the command line, it will run in Jar Mode. If you downloaded the source code access it through NetBeans, ConcX will run in IDE Mode.

The runtime mode is only important because it affects the usage of the Extrnl buttons and ellipses ( . . . ) buttons. These buttons use the JFileChooser object which only works on file systems and not on the contents of jar files. Since the ConcX jar file has every pre-defined Bird Type compiled into the jar file, the buttons cannot be used to find types within the jar file (only external Bird Types). This is why entering the first few letters of Bird Types and Food Types is emphasized.

Extrnl Button
Because of the differences of the two different modes, the file chooser button to the right of the (Bird) Type field produces different results. Its uses are described by mode below.

Jar Mode
Pressing the Extrnl button displays a file chooser that allows you to navigate thru the file system (not inside the jar). This feature allows custom Bird Types to be compiled outside of NetBeans and then use the Extrnl button to access their .class files while in Jar Mode. Refer to The Config Menu section later in this chapter for details on setting up external Birds.

IDE Mode
In IDE Mode, the Type field validator only searches the file system, which makes the Extrnl button somewhat redundant. However, if you prefer to visually browse thru Bird Types, you can configure the Extrnl button to search the same locations as the Type validator. Refer to The Config Menu section later in this chapter for details on setting up this button.

Enter a Bird Name (Recommended)
Entering a name for the bird isn’t required but it is recommended as an easy way to identify which bird performed which events that were recorded in a pod history. For example, you might have five of the same type of bird and giving each of them a unique name helps you know exactly which one of the five performed an event.

It is also helpful to name the birds in some kind
of meaningful way, such as an alphabetical or numeric sequence. For example, the first bird might be named Alice or Alpha or a1, the second bird Bob or Betty, etc. If multiple instances of the same kind of bird will be run, it can be useful to add a number to the end of the names. For example: Bob-1, Bob-2, Bob-3, etc. The name isn’t validated or evaluated so the choice is yours – have fun with it.

**Select First Food Eaten (Required)**
The first (upper) Food Eaten Food Type field (and all the other Food Type fields) works similarly to the (Bird) Type field: enter the first part of the desired food (case insensitive) and press Tab and ConcX will validate what you entered. The Food Type field will be updated as follows:

- “Not Found” is displayed if nothing matched what you entered
- The properly capitalized full name of the food will be displayed
- A pop up dialog box is displayed with a list of matching names to select from. Click on the desired food and then click OK, or click on the cancel button to return to the field which will display “Not Found”.

Every bird must eat some kind of food which is why the first (upper) food is a required field. However, this creates a “chicken or the egg” situation at the beginning: the right kind of food must be in the tree or a bird can’t eat but a bird can’t store the right type of food until it has eaten.

This stalemate is solved several ways. In many scenarios the first food eaten is a “MagicPod”. MagicPods prevent this situation because BasicBirds will automatically create food pods if they are set to eat MagicPods. The MagicPods will be digested as programmed and then store them as the appropriate type in the TupleTree. In scenarios which are more like simulations it is more common to have birds create that first food in their doFirstTimeProcesses method.

**Eat First Food from a File**
Birds can be configured to eat from a file by clicking on the File Name radio button. This enables the File Name field and disables the Food Type field. You can then type in the full path and file name or you can use the ellipses button to the right of the File Name field to
browse the file system to select the desired input file. See Chapter 12 for details.

There are several advantages to using input files. The input files can be used to ensure that the full range of values is tested, such as positive and negative values, zero values, empty Strings, extremely long Strings, etc. Input files can also be used to verify that the correct results are received when a set of standard input values are processed.

**Accessing Second Food Eaten Field (Optional)**

Some situations require that a bird eat two foods. Depending on your requirements for the second food, click on either the AND radio button or the OR radio button to enable the second (lower) food fields. The second food can then be chosen, either as a Food Type or a File Name.

The AND/OR radio buttons work as follows:

- Click the AND option if you want the bird to be required to **always** get both foods before it can eat. If the bird can only get the first food and can’t get the second food, then it can’t eat and has to give up the first food. The bird always tries to get the first food before it tries to get the second food, in that order.

- Click the OR option if you want the bird to eat if it gets either one of the two foods. The bird will always try to get the first food first. If it does get the first food, it won’t try to get the second food. The bird only tries to get the second food if it was unsuccessful getting the first food.

The second Food Type field works just like the first Food Type field; enter the first letters of the food type and ConcX will validate and respond appropriately. If you click the File Name radio button, you can type in the full path and file name or use the file chooser button to the right of the field. The file name and path are not validated so be careful if you type it.

**Select First Food Stored (Required)**

The first (upper) Food Stored Food Type field works just like the other Food Type fields: enter the first part of the desired food (case insensitive) and press Tab and ConcX will validate what you entered. The Food Type field will be updated with one of the following:

- Not Found if nothing matched what you entered
- The properly capitalized full name of the food
- A pop up dialog box with a list of matching names to select from. Click on the desired food and then click OK, or click on the cancel button to return to the field.

**Store First Food to a File**

Click the File Name radio button if you want the food pod written to a file instead of being put into the TupleTree. Enter the full path and file name of the file that should receive the
contents of the food pod. You will need to override one of the food storage methods if you want more sophisticated formatting than simple toString formatting.

**Access Second Food Stored**
Some situations require that a bird store two foods. Depending on your requirements for the second food, click on either the AND radio button or the OR radio button to enable the second (lower) food fields. The second food can then be chosen, either as a Food Type or a File Name.

The AND/OR radio buttons work as follows:

- Click the AND option if you want the bird to be required to always store two foods.
  - If the bird eats just one food, after the food pod has been digested, a copy of the pod will be created and then stored twice, once as the first stored food and once as the second stored food.
  - If the bird eats two foods, it will always store the first food eaten as the first food stored and the second food eaten as the second food stored.

- Click the OR option if you want the bird to store its food as either of the two foods.
  - If the bird eats just one food, then the food will be stored as either one of the two stored foods.
  - If the bird must eat two foods (AND) then the default behavior is to store the first food eaten as either the first stored food or as the second stored food. The second food eaten will then be stored as the other type of food.
  - If the bird will eat either of two foods (OR) then one food pod will be written, either as the first stored food or the second stored food.

**Food Storing Considerations**
One very useful food storage configuration is to have a bird eat one food but store two foods, one to the TupleTree and one to a file. The food stored in the TupleTree continues processing normally but the food stored to a file holds the interim results. This allows developers to review their results at each stage of the process instead of waiting for the final results and then trying to figure out how things went wrong.

The Food Stored section also allows you to select NullPods, which are food pods that never get stored. This is a good choice when working with databases and you don’t want to actually retain the food pod once it has been uploaded into the database.
The Vitality Tab

The Vitality tab contains indicators that display information about the bird as well as five fields that let you control the bird’s life cycle settings.

**The Life Length field** specifies how long the bird will live (in milliseconds) if it gets enough food to survive. Negative values mean that the bird will not stop unless you click the Stop Me/Stop All button or it doesn’t get enough to eat. The default value is 30 seconds.

**The Stamina field** specifies how long the bird can survive without eating. Negative values mean that the bird will never die from hunger. The default value is 5 seconds.

Making both these first two fields negative will ensure that the bird will stop only when you click the Stop Me/Stop All button. If you are debugging your code, it can be useful to set both of these fields to negative values so the bird doesn’t die while you are single-stepping thru your code.

**The Patience field** specifies how long the bird will try to eat a second food. This field has no effect if you specify only one food to be eaten. When you specify two foods must be eaten, the bird will eat the first food and then try to eat the second food. It will hold the first food until it gets the second food or it exceeds the length of time specified in this field (runs out of patience). It prevents any other bird from getting its first food while it tries to get its second food. Negative values in this field mean that the bird will never stop trying to get its second food, which will almost certainly lead to deadlock situations if the second food is shared with any other bird. Use negative values with caution. The default value is 5 seconds.

**The Nap Length field** determines the maximum length of time (in milliseconds) the bird will nap. The nap method selects a random duration up to the length of time specified in this field. Negative values are not accepted in this field because it would cause the bird to sleep endlessly and never wake up. The minimum length of time is 5 milliseconds which is hardcoded in AbstractBasicBird; override the nap method in your custom bird if you want a different minimum length or if you want a fixed length nap. The default value is 75 ms.

**The Reproduce field** specifies how many times in a row the bird must eat to qualify as “well-fed and ready to reproduce”. Small values (such as 30 – 50) mean that the bird is more likely to reproduce and create a duplicate bird. Each duplicate gets a unique name based on the original bird’s name. Larger values (300-500) mean that the bird is much less likely to reproduce. Negative values mean that it will never reproduce. The default is -1.
The Code Info Tab

The Code Info tab lists the methods, constructors, and fields of the current bird or the current bird’s superclass. Click the Show button to list its information. Click the Show Super button to list the information about its superclass.

The XMV Tab

The XMV tab provides you with 10 user-defined values (Externally Managed Variables) that you can use with a bird. You must provide the code required to get the values from the XMV fields and to use the XMV values in your bird. It is optional to enter values into these fields unless your code specifies that entries must be provided.

There are 10 rows on the XMV tab. The left column of fields contains the names (or descriptions or hints or labels) for the values in the right-hand column of fields. The contents of the left column of fields are for display only; they are not used as part of the bird’s program. They are there strictly to help you remember what the values in the right column of fields represent.

The contents of the right column of fields are the
actual values that will be passed into your bird.

The XMV values in the right column will always come into your program as Strings. Which means that any numerical values listed in the code must be intentionally cast or otherwise converted into Java numerical values. It is recommended that you catch the Number Format Exception to prevent your bird from trying to process any non-numeric values you may have entered into the field by accident.

The contents of both columns are retained when you save your birds as a file. This ensures that when you reload a bird file, the descriptions or explanations of the values in the right-hand columns will always be available.

The History Tab

The History tab shows the events in the bird’s lifetime and the times that the events happened.

This tab also contains the Level of Info dropdown that lets you select the amount of event detail that you want to have included in the history. There are 7 levels of information. These levels match the seven logging levels defined by java.util.logging, which are:

- Severe (highest) – indicates a serious failure.
- Warning – indicates a potential problem.
- Info – provides informational messages.
- Config (default) – includes configuration messages as well as event.
- Fine – provides tracing information.
- Finer – provides fairly detailed tracing messages.
- Finest (lowest) – provides most detailed tracing messages.

You cannot turn off history. It is always there, guaranteeing that it will be available when you need it. If you only turn it on when you think you need it, it means that you probably just missed an important event/condition that may not happen again for a long time.

Note that History is on a bird by bird basis so there is never any contention writing to a single log file.

The Clear button deletes the contents of the current History. This isn’t really required since every time the bird starts, its history is cleared.
The **Save button** allows you to save a bird’s history to a file. Enter the path and file name for the history that you want to save and then click on the Save button.

**The Right Panel – Bird Activity, Amount of Food, and TupleTree Viewing**

The right side panel contains information about the birds once they are flying or when they are done flying. This means that the right side panel mostly contains indicators and very few controls.

**The Activity Tab**

The Activity tab displays 10 identifier fields and 10 activity ProgressBars on each of the Current Activity tabs. The identifier fields display the user assigned name of the bird followed by a colon (":") followed by the bird’s Type. To the right of the identifier field is the activity ProgressBar assigned to that bird. Every new bird that is added receives its own activity ProgressBar. See first screenshot following.

When the number of birds exceeds 10, a new Current Activity tab is added that will hold 10 more birds. New tabs will be added for as long new birds are added, regardless of if they are manually added by pressing the Add New Bird button, or if they are loaded from a bird file, or if they are automatically added when a bird reproduces (second screenshot).

The activity ProgressBar is increased each time a bird successfully eats. If a bird doesn’t successfully eat, then its ProgressBar isn’t increased.

Activity ProgressBars always grow larger. When a ProgressBar has almost reached its maximum amount (99% or so), its maximum amount is automatically doubled. After doubling its maximum, the ProgressBar is displayed as only half as long as before. After doubling its maximum, the ProgressBar will grow more slowly than before because each unit of growth is worth twice as much as before. The activity ProgressBars always begin with a value of 100.
When a bird reproduces, it creates a duplicate of itself but gives it a new name derived from the name of the bird which reproduced. For example, the bird named a1 will have its first offspring named a1-1, followed by a1-2, a1-3, etc. If its first offspring reproduces, they will be named a1-1-1, a1-1-2, a1-1-3, etc. Each offspring will have its own ProgressBar assigned to it; if you stop that offspring, the original bird will continue flying until it starves or reaches the end of its natural life time.

At the top of the screen is the Errors ProgressBar. When your bird stores an Err pod in the TupleTree, it will also automatically update the Errors ProgressBar. This bar will increase every time an Err pod is stored. This ProgressBar behaves just like the activity ProgressBars; any time the number of errors is almost as great as the maximum value to the right of the Errors ProgressBar, the maximum value is doubled and its ProgressBar is updated with the new value, causing the completed progress to be cut in half. The initial maximum value always begins with 100.

**Important:** not finding a food pod is NOT considered an error. Errors are user defined in code and can be written to handle situations such as finding alpha characters when numeric digits were expected or finding seeds in a food pod when there should have been none. The erroneous pod is stored exactly as detected and stored in the TupleTree as an ErrorPod.

At the bottom of the screen are two clear buttons: Clear Selected Bird and Clear All Birds (screenshot at right).

The Clear Selected Bird resets the activity ProgressBar for the currently selected bird on the left side panel. You cannot select a bird from the right side panel. When the activity ProgressBar is cleared, the percentage complete is set to 0 and the maximum value for the ProgressBar is set to 100.

The Clear All Birds button resets all of the activity ProgressBars to their initial settings; every percent complete is set to 0 and the maximum value for every ProgressBar is set to 100. This clears all birds, including those whose ProgressBars are displayed on other tabs.
The Food Supply Tab

The Food Supply tab is similar in appearance and behavior to the Activity tab; it has 10 identity fields and 10 ProgressBars and when more than 10 food pods are used, a new tab is added to hold the additional food pods.

The Food Supply ProgressBars work differently than the Activity ProgressBars. The Food Supply ProgressBars show the net amount of food available. Every time a bird stores a food pod in the TupleTree, the ProgressBar for that food pod is increased by 1 unit. Every time a bird eats a food pod, the ProgressBar for that food pod is decreased by 1 unit.

The Food Supply tab shows the overall amount of food pods available and work independently from the Activity ProgressBars. It is common for the Activity ProgressBars to show a great deal of activity but the Food Supply ProgressBars to show very few food pods available because the food pods have been eaten as rapidly as they have been stored, leaving their ProgressBars “empty”.

The number of food pods shown is generally not important; what is important is whether or not the number is approximately what was expected. For example, if you expected all of the food pods to be consumed but one type of food pod isn’t being consumed, that highlights a bird which isn’t performing as expected.

The Food Supply tab also contains an Errors ProgressBar and two clear buttons. The Errors ProgressBar is a duplication of the Activities tab Errors ProgressBar; they should always show the same number of errors.

The clear buttons work just like the clear buttons on the Activity tab, except they clear the information about food pods instead of activity. Clicking the Clear This Bird’s Foods will reset the information about the foods for the currently selected bird on the left side panel. Note that as many as 4 food ProgressBars may be cleared, depending on how many foods the currently selected bird eats (two max) and how many foods the currently selected bird stores (two max). Clearing the foods for one bird may affect other birds that also eat or store these foods.

The Clear All Foods will reset the information about all of the foods currently listed on the Food Supply tab(s). The food ProgressBars displayed on other sub-tabs of the Food Supply tab will also be displayed.
Note that clearing the food pod ProgressBars does not affect the associated food pods that are stored in the TupleTree. Clearing the food pods on the Food Supply tab only resets the ProgressBars to their initial values.

**The TupleTree Tab**

The TupleTree tab shows a summary of the food pods contained in the TupleTree (see next screenshot). The History field contains a text listing which is refreshed when you switch to the TupleTree tab or when you click the Show Tree button.

The History list is not a dynamic listing. If you are viewing the TupleTree tab when you restart the birds flying, the list will not be updated while you are viewing the tab unless you click the Show Tree button.

At the top and bottom of the listing is the total number of food pods stored in the TupleTree. It is repeated for your convenience only, just in case you have a lot of food pods to scroll thru in the TupleTree.

A summary of each food pod is listed. Each summary begins with the name of the food type for the food pod.

Each type of food pod has its own unique (user assigned) key value that is used to store and retrieve that type of food pod. The unique key value allows you to re-use the names for the food pods. For example, the name CyanPod shown on the right could potentially come from the colorfoods folder or the printerInk folder or the dye folder; as long as their “treeID” values are unique, ConcX will keep them straight and never give you the wrong CyanPod.

After the Key value is the Contents Desc, which is a user assigned description of what is contained in that food pod. The Contents Desc is just a String field into which you can put anything that will go into a String. In the AddxBirds example, the Contents Desc holds the String equivalent of the numeric running total. In other scenarios, the Contents Desc could hold a brief summary of the contents or even an object’s toString value.

Following the Contents Desc is the number of seeds that are currently held in that food pod.

Following the Seeds in Pod entry is the History of that food pod. Every food pod has its own personal history (log) of the events that happened to it along with the times that these events happened. Food events are not recorded based on the Level of Info set for a bird; instead the events which the developer believes are important enough to record are always...
written to each pod’s history.

At the bottom of the TupleTree tab are three buttons and a field.

**The Show Tree button** tells the TupleTree tab to discard the current History listing (list of TupleTree contents) and generate a new listing. Sometimes the History list isn’t updated when it should have been; click the Show Tree button before panicking and assuming that something horrible has gone wrong.

**The Save button** will write the listing to the path and filename entered in the field to its right. If there is nothing entered in the field, clicking the Save button will be ignored. There is no path validation or JFileChooser dialog box; you have to enter the correct path and filename (no duplicates) or an error message will be displayed on the NetBeans Output window.

The Save button also provides the opportunity to get more information about the contents of the food pods in the TupleTree. If the seeds in the food pods in the TupleTree contain a toString method, this will be detected by the TupleTree and this method will be called while saving the contents to a file, providing detailed information about the seeds contained the food pods. Even without this seed method, saving to a file will list each seed contained in each pod as well as each seed’s description and info about each seed’s kernel object and info using its toString method.

**The Clear Tree button** will delete any food pods currently in the TupleTree and then refresh the History field to show that there are zero objects in the TupleTree.

**Bottom Row Buttons**

Below Left (Birds) Panel and the Right (Activities) Panel is a row of buttons that affect all birds and/or all activities. See the cropped screenshot below. Each of these buttons is described below.

![Bottom Row Buttons](image)

**The Start All button** is used to launch all birds that have met the minimum requirements for flying: a valid Bird Type has been entered, a valid eaten Food Type has been entered for the first (upper) food, and a valid stored Food Type has been entered for the first (upper) food.

When the Start All button is pressed, the birds on each of the bird tabs are examined in order, starting with the lowest numbered (leftmost) tab, and finishing with the highest numbered
rightmost) tab. If the bird on the examined tab meets the minimum requirements for flight, it is launched as a new thread. Otherwise, the bird is not launched. In either case, the examination for that tab is finished and examination begins on the next higher tab.

This standard behavior has two consequences.

- An easy way to temporarily NOT fly a bird is to change its bird type to the easiest non-name and pressing tab. Most of its settings are retained for later use.

- Birds always launch in the same order. A non-determinist will protest this guaranteed sequence, but this standard behavior is considered an issue to be aware of but not considered to materially affect the outcomes of most scenarios because of the significant amount of randomness incorporated into the unmodified lifecycle of BasicBirds.

For example, the standard Nap of a BasicBird and its descendants is 75ms, which means that after every digestion cycle (not just the ones where it ate) the bird will nap for a random length of time between 5ms and 75ms. And if the DigestionDelay method isn’t changed, the bird will use the configured Nap length to for a random length of time between 5ms and 75ms pause during every digestion cycle. Which makes the combined standard thread.sleep time on each lifecycle loop to be anywhere between 10ms and 150ms. If that range is too small, the maximum nap length can be set to as many seconds as you see fit.

The above random nap lengths makes the impact of a predetermined starting sequence negligible for most scenarios. However, this standard starting behavior should always be kept in mind when designing new scenarios.

Pressing the Start All button will NOT attempt to launch a bird that is already flying – it will only write a message to System.out that the bird(s) is current alive and can’t be Started and allows the currently flying birds to continue flying. This means that the Start All button can be used to “keep alive” birds in a flight that stopped before the others in the flock.

The Start All button is always enabled and does not reflect the current status of the birds the way that the Start Me/Stop Me buttons do.

**The Stop All button** is used to stop all flying. It begins stopping birds at the lowest numbered (leftmost) tab and continues stopping each bird in order to the highest numbered (rightmost) tab. It is not an error to press the Stop All button when no bird is flying and no message is printed.

The Stop All button is always enabled and does not reflect the current status of the birds the way that the Start Me/Stop Me buttons do.

**The Remove All button** is use to delete all Bird tabs as well as to delete all Activity tabs and all Food Supply tabs. It also deletes the contents of the TupleTree history text area.

The Remove All button makes it easy to dump the currently loaded scenario so a new
scenario can be loaded. This is especially useful in cases when so many changes have been made to a flock of birds that it easier to reload the flock than it is to undo the changes.

The Remove All button is always enabled. If you press the Remove All button while birds are flying, it will first try to stop all birds and then delete all the bird/activity/food supply tabs. Because of the way that the system works, the TupleTree history text area may have a few residual items left in it if you click the Remove All while the birds are flying. Click the Clear Tree button to remove the residual items left in the TupleTree, if required.

**The Add New Bird button** is used to add a blank new bird available to be configured. Refer to the section on the Left Panel configuration for more details on the available bird configuration options.

The Add New Bird button is always enabled. If you click on the Add New Bird button while other birds are flying, it will work just fine. And if you successfully configure that new bird while the others are still flying, the new one will join the flock without a problem.

**The Clear Errors button** is used to zero out the Errors progress bar. It preserves any Errors detected by leaving all error pods in the TupleTree. It is always enabled.

**The Clear All Activities button** is used to zero all progress displayed on the Activity tab, to zero all progress displayed on the Food Supply tab, and to remove food pods from the TupleTree. All birds listed on the Activity tab(s) remain in place; only their progress information is cleared. All food pods listed on the Food Supply tab(s) remain in place; only their progress information is cleared.

The Clear All Activities button also deletes all food pods placed in the TupleTree. That means that this button should not be pressed until you are completely done with the pods that were placed in the TupleTree. The food pods cannot be un-deleted or restored after this button has been pressed.

The Clear All Activities button is normally used to clear the data collected during a flight in preparation for another flight using the same flock.

The Clear All Activities button is always enabled and can be used while the flight is in progress.
**ConcX Menu Choices**

In the top left corner of ConcX are two menus: the File menu and the Config menu. Click on either of these menus to display its menu items.

**The File Menu**

The File menu contains three options:

1. Open Flock File (to load a flock of birds into ConcX).
2. Save Current Birds (to create or update a Flock File to load later)
3. Exit.

**The Open Flock File option** launches a file chooser that allows you to navigate to any location in your file system and select the Flock File (saved collection of preconfigured birds) that you wish to open. Note that you will need the appropriate permissions to open the selected file which is more of an issue on Linux systems.

If you open a file, ConcX will expect to find an XML file that defines all the birds that you wish to have added to ConcX. If you open a file that doesn’t contain birds or XML, it will fail in unpredictable ways. The Open Flock File option will not overwrite any existing birds.

The default folder in the file chooser is the flocks folder. The flocks folder is normally in the same folder that holds the concx.jar file but it is not required. Use one of the Config menu options to select the desired folder that will hold your bird files. See the Config Menu section for details on setting the configuration.

**The Save option** launches a file chooser that allows you to navigate to any location in your file system to save the currently loaded birds into a Flock File. Note that the Save option doesn’t automatically save to the location where you opened the flock file. Note that you will need the appropriate permissions to save to the selected location.

Saving will write every active bird currently in ConcX to an XML file. Any bird that has a Type of “None” or “Not Found” will be skipped and not be recorded to the XML (bird) file. So if you want to save just some of the currently loaded birds, you will need to change the Type of the birds that you don’t want to save to a type of bird not in ConcX. For example, I usually add “xx” to the end of any bird that I don’t want saved and press Tab – the validation on the Type field tries to find that bogus name, fails, and then changes the Type field to “Not Found”. The other fields for the bird are not affected, so I can save just the desired birds. After saving them, I can go back and re-select the desired Bird Type(s) and the bird(s) will be all setup, just like before the save.
The Config Menu

ConcX creates a configuration file (avian.cfg) the first time you run ConcX. Any changes made using the Config Menu causes the config file to be rewritten. The new settings are usually used immediately, but a Re-Read the Config file option is available just in case.

The config file initially contains a variety of settings based on the current directory and the location of concx.jar. For example, if you unzipped the distribution file into /home/test/ , changed to it, and then issued the standard startup command (java –jar concx.jar Nimbus) then ConcX would start with /home/test/ as its current directory and would look for the flocks, data, and lib folders there.

If you execute ConcX from a different folder, the avian.cfg file will point to that folder and write the flocks, data, External Birds and External Foods as pointing to that folder. It is your responsibility to update the appropriate config settings before trying to use them.

You might want to use these other file locations if you are working in a group situation and you want to share common flocks for consistent testing results. Or perhaps you want to run the birds that someone else developed, you can set the External Birds location to a shared file location.

The Config menu contains seven options that will help you set up ConcX:

1. Set Flock Files Location
2. Set Data Files Location
3. Set Bird Files Location
4. Set Food Files Location
5. Set External Bird Location
6. Set External Food Location
7. Re-read Config and Use

The Set Flock Files Location option allows you to identify which folder should be opened when you click on the Open Flock Files option. The Flock files are always part of your computer’s file system (and not part of the jar) so they can be set to any location you desire. For example, to a shared file location so multiple users (testers) can use the same flocks.

The default location for the flocks folder is in the same folder where ConcX was unzipped. For example, if you unzipped the ConcX.zip file into the C:\test\testConcx\ folder, that folder will contain concx.jar as well as the flocks, data, and lib folders. Use this option is you wish to use flocks from a different location.

This setting does not affect the validation routines of Bird Types.

The Set Data Files Location option allows you to identify which folder should be opened
when you click on the button to the right of one of the File Name fields. This folder is used when you want to read (eat) food from a file instead from the TupleTree or when you want to put (store) food into a file instead of the TupleTree.

The default location for the data folder is in the same folder where ConcX was unzipped. For example, if you unzipped the ConcX.zip file into the C:\test\testConcx\ folder, that folder will contain concx.jar as well as the flocks, data, and lib folders. Use this option is you wish to use data from a different location.

This setting does not affect the validation routines of the Food Types fields.

**The Set Bird Files Location** option allows you to identify which folder should be searched when validating the Bird Types. This location is normally automatically set for you and shouldn’t require changing. The configuration routines automatically determine if you are running Linux or Windows and if you are running a jar file or from an IDE. If you are running a jar file, this will be set to com/avian/birds inside the jar file. Changing it to point to a file location will cause the validation routine to fail. Use the Set External Bird Location setting if you want to access .classes outside of your jar file.

This setting *does* affect the Bird Types validation routines and should be modified with care. However, if ConcX does crash because of this setting, simply delete the avian.cfg file and start over.

**The Set Food Files Location** option allows you to identify which folder should be searched when validating the Food Types. This location is normally automatically set for you and shouldn’t require changing. The configuration routines automatically determine if you are running Linux or Windows and if you are running a jar file or from an IDE. If you are running a jar file, this will be set to com/avian/foods inside the jar file. Changing it to point to a file location will cause the validation routine to fail. Use the Set External Food Location setting if you want to access .classes outside of your jar file.

This setting *does* affect the Food Types validation routines and should be modified with care. However, if ConcX does crash because of this setting, simply delete the avian.cfg file and start over.

**The Set External Bird Location** option allows you to identify which folder the file chooser should open when you click the button to the right of the (Bird) Type field. The original intent of this field was to allow you to access files outside the jar file. For example, you have the standard concx.jar file installed and you want to use a new bird that you created and compiled with Eclipse or IntelliJ. Instead of generating a new jar file, the External Bird Location can be set to your .class file location.

This setting does not affect the Bird Type validation routines because they are by-passed.
When you click on the button to the right of the (Bird) Type field, this config setting will be used as the default folder for the file chooser. Any bird selected from this external file location will be added to the (Bird) Type field without validation. So if you select BalanceCheckbook.class, you deserve whatever disastrous results that occur.

When running ConcX from the IDE, the button to the right of the (Bird) Type field isn’t really needed. In IDE mode, you’re probably going to be creating or modifying bird files that the validation routines are going to be searching anyway. Which opens up a new use for this button and setting; by setting the External Bird location to your current project’s build location and then drill down to the com\avian\birds folder, you can repurpose this button to be a visual bird selector. Instead of typing the first part of a valid bird type, you can click on this button to pop up the file chooser so you can navigate to the desired folder and select the desired bird. This is handy if you forgot the Bird Type.

**The Set External Food Location** option allows you to identify which folder the file chooser should open when you click the button to the right of the Food Type fields. The original intent of this field was to allow you to access files outside the jar file. For example, you have the standard concx.jar file installed and you want to use a new food pod that you created and compiled with Eclipse or IntelliJ. Instead of generating a new jar file, the External Food Location can be set to your .class file location.

This setting does not affect the Food Type validation routines because they are by-passed. When you click on the button to the right of the Food Type field, this config setting will be used as the default folder for the file chooser. Any food pod selected from this external file location will be added to the Food Type field without validation. So if you select BalanceCheckbook.class, you deserve whatever disastrous results that occur.

When running ConcX from the IDE, the buttons to the right of the Food Type fields aren’t really needed. In IDE mode, you’re probably going to be creating or modifying food pods that the validation routines are going to be searching anyway. Which opens up a new use for this button and setting; by setting the External Food location to your current project’s build location and then drill down to the com\avian\foods folder, you can repurpose this button to be a visual food type selector. Instead of typing the first part of a valid food type, you can click on this button to pop up the file chooser so you can navigate to the desired folder and select the desired food pod. This is handy if you forgot the Food Type.

**The Re-read Config and Use** option reads the avian.cfg file so they will be used during the current run of ConcX. This option isn’t normally needed since the AvianConfigFile object is updated any time you change one of the above options and it is then used to write the config options to the avian.cfg file. So the AvianConfigFile object and avian.cfg should always have the same settings.

However, if you are testing both the IDE and jar modes from the same folder, the wrong
settings for the avian.cfg file can get loaded by mistake (for example, the IDE settings get loaded when you’re running in jar mode). In this case, just delete avian.cfg and then select this option. This will create the appropriately formatted config file and then load these default (but mode appropriate) settings into the AvianConfigFile object.
Chapter 5. AddxBirds Revisited

A Brief Explanation of AddxBirds

Chapter 1 introduced the AddxBird scenario that demonstrated how a group of birds can cooperatively process multiple items in parallel. Instead of coding the end to end process in a single thread and then trying to figure out how to make it parallel, the reverse approach is taken. The entire process is broken down into atomic steps and then performs these steps in parallel. This approach allows us to discover and exploit opportunities for parallelism that are not otherwise obvious.

Once the add5.bird file has been loaded and the Start All button clicked, all five birds start trying to eat their designated foods. The first bird to successfully eat is the Add1Bird. It eats a MagicPod and sets the running total to a value of 1 and then stores the food pod as a BlackPod in the TupleTree. After a nap, the Add1Bird tries to eat again, repeating the standard bird lifecycle.

Birds that eat MagicPods always find something to eat – that’s why they’re magic. The MagicPod is necessary in this scenario because every bird needs something to eat and there is nothing to eat in the TupleTree when Start All is first pressed. If one of the birds doesn’t get food pods into the TupleTree, the birds will all starve. Instead of using MagicPods, a data file could have been created that contains every value between 0 and 1000 or every even value, etc. But MagicPods are the easiest way to get started.

Once the Add1Bird has stored a BlackPod in the TupleTree, the BlackPod is available to be eaten by any bird that eats BlackPods. If no birds eat BlackPods, the pod will just sit in the tree uneaten. If two or more birds eat BlackPods, the first one who finds it is the one who eats it. There is no fairness in Avian Computing but there is luck. More on luck later.
In the AddxBird scenario, the Add2Bird eats the BlackPod and then digests it by adding 2 to the running total. It then stores the running total as a BluePod in the TupleTree for the other birds to eat. After a brief nap, Add2Bird repeats the life cycle and looks for BlackPods again.

This process repeats with the Add3Bird, Add4Bird, and the Add5Bird, using other color food pods until the Add5Bird has added 5 to the running total and stored it in the TupleTree as a CoralPod. And since no bird is assigned to eat the CoralPod, the CoralPods just accumulate in the TupleTree.

Every bird in this scenario is simultaneously repeating its lifecycle until it either starves to death, until it dies of old age, or until you click on one of the Stop buttons. While all the birds are flying, if you click on the stop button of the Add3Bird, that thread stops running but all of the other birds keep flying, repeating their lifecycle. Nothing crashes. It is normal for a bird to look for food and not find any food. Clicking the Add3Bird Start Me button restarts that bird and it begins its life cycle again.

If you select the Code Info tab of any of the AddxBirds, they all implement only one method that overrides the afterDigestion method in BasicBird. There’s no need to write code that locks the food objects, no synchronizing of Objects. Finding food pods, storing food pods, it’s all just handled for you so you can focus on how to break your application into the appropriate pieces. The code
for one of the AddxBirds was shown in Chapter 1. It shows that there is no locking, no synchronizing, not mutexes or critical sections or semaphores. It’s all handled for you.

**Avian Foundations of AddxBirds**

The TupleTree in ConcX is a simple implementation of a tuplespace, first proposed by David Gelernter back in the 1980’s and worked on with Nicholas Carriero as Linda, described as “coordination language for concurrent programming” or as a “logically global associative memory”. The idea of a tuplespace was seen as sufficiently powerful that it became the basis for JavaSpaces in the 1990's and IBM's Tspaces.

The TupleTree is a very simple tuplespace and doesn't do any of the more sophisticated features of a tuplespace. Instead it stores food pods by name and passes out food pods requested by name. Using ConcX, any bird can be configured to eat any food pod as well as configured to store any food pod. This simplicity reduces the coupling between birds, making it easy to change or rearrange the sequence that birds work on the food pods.

The beauty of the TupleTree is that it frees developers from having to manage parallelism from the code that they are writing because the TupleTree does all the synchronizing needed any time it adds or removes food pods. Gelernter’s tuplespace provides a certain amount of “indirection” that makes message passing between objects more robust. The tuplespace is like a post office and various threads can come and check for mail in their individual PO boxes. The structure of the tuplespace makes it easy for threads to pass information to one another without interfering with each other.

ConcX builds on this the strength of tuplespaces. Any time a bird receives an instance of a food object from the TupleTree, it has the only copy of that instance and doesn’t have to share with any other birds. Other birds can’t modify or corrupt that instance of a food object because they don’t know it exists and couldn’t access a food object held by a different bird because there are no methods for sharing between birds.

In ConcX, the TupleTree contains a private ArrayListMultimap so no other Object can make changes to it. The TupleTree synchronizes (locks) the ArrayListMultimap any time one of its public methods attempts to modify its ArrayListMultimap. Birds are free to request food pods from the tree or try to store food in the tree and remain blissfully oblivious to any locking or synchronizing or any other parallel processing idioms.

When a bird looks for a food pod in the TupleTree, such as a CoralPod or a BlackPod, it receives any one of the requested food pods, with no order or sequencing intended. The bird would just get one food pod of the desired type. It cannot request a specific instance of a type of food pod. This behavior was specifically incorporated to allow programs to gracefully scale on systems that contain hundreds or thousands of cores. ConcX uses anonymous food pods so any bird assigned to eat that type of food pod can process it appropriately. Avian Computing’s simplified method of controlling threads is why it is so easy to dynamically adjust computing resources to meet the current workload.

Note that if a food pod is requested by a bird and there isn't one in the TupleTree, it isn't a
problem; the bird simply tries again later, just like a real bird would do. This try-again-later methodology not only resembles the real world behaviors, it makes it easier to work with ConcX because it reduces the time-dependencies. As long as the system is running, a food pod can be picked up milliseconds after being placed in the TupleTree or hours later. This time insensitivity makes Avian Computing even more loosely coupled and simplifies how information gets passed between birds.

The work performed by the AddxBirds doesn’t really result in a useful product. Instead what the AddxBirds scenario demonstrates is a five-step framework that has been streamlined into its simplest possible components, where each component can be performed asynchronously at its own rate without interfering with other components.

If in the first step, Add1Bird was replaced by GetCustomerBird, and in the second step, Add2Bird was replaced by GetBillingAddressBird, the third step replaced by GetCustomerChargesBird, the fourth step replaced by a CalculateAllChargesBird, and the last step replaced by PrintInvoicesBird, this simplified framework suddenly looks a lot like a parallel program that does monthly customer invoicing. All five steps run concurrently and asynchronously. Need it to run faster? Add more of the same five types of birds. Need it to run even faster? Add multiple computers that run large numbers of these same five types of birds.

What if you actually wanted to save the results? Don’t change any of the existing five birds – just add a sixth bird, SaveChargesBird that eats the food the fourth bird stored and then store a new food that the fifth bird will eat. How about that? We just made a major revision to our billing system and it only took 2 minutes to incorporate the changes (once the bird was created).

**Variations of AddxBirds**

Close and then re-start ConcX (that’s the easiest way to remove unwanted tabs). Load the add5.bird file again by clicking the File menu and then selecting Open Flock File option. Then navigate to the addbirds folder and select the add5.bird file.

**Identify Baseline:** Re-run the AddxBirds scenario again to identify the baseline quantity of food pods (probably between 300 and 400 food pods).

1. Click the Start All button in the bottom left corner.
2. When the birds all stop, click on the TupleTree tab and note how many food pods are contained in the TupleTree. This is your computer’s baseline quantity.
3. Click the Clear All Activities button.

**Variation #1: Adjusting Nap Length Adjusts How Frequently a Bird Eats**

1. Click on the #1 tab to select Add1Bird a1a.
2. Click on the Vitality tab. The screen will look like the following screenshot.

3. Change the Nap Length value from 75ms to 750ms.

4. Click the Start All button.

The birds will all start flying, but now they are all “gated” to the Add1Bird. On the Activity tab, there will be a pause while the Add1Bird naps and then when it wakes up and eats, its ProgressBar advances, followed by the ProgressBars of the other birds in rapid succession as they eat. The other birds are actively looking for their own types of food in the TupleTree, but until the Add1Bird wakes up and eats and stores a BlackPod in the TupleTree, the other birds can do nothing but look for food.

This is even more apparent on the Food Supply tab. Click on it and watch as each time the MagicPod ProgressBar grows, there is a brief flicker on the BlackPod, then BluePod, then BrownPod, then ChartreusePod ProgressBars as a food pod is stored and then eaten by the next bird. And then finally the CoralPod grows 1 unit.

When all the birds have stopped, click on the TupleTree tab.

The TupleTree size will likely be about 40 food pods, almost all of them CoralPods. There are so many CoralPods because the other birds are so much faster than the Add1Bird that any food pods put into the TupleTree is immediately eaten and then stored by the other birds.

**Variation #2: Birds Die When They Run Out of Stamina**

5. On the #1 tab, change the Add1Bird's Nap Length to 7500ms.

6. Click the Clear All Activity button.

7. Click Start All button. The Add1Bird will eat once, followed by all the other birds, but when it eats a second time, it normally will be the only one to eat.

8. When all of the birds have stopped, click on the TupleTree tab. It will show only one CoralPod and then a handful of BlackPods.

Every bird has a Stamina setting, which is the length of time that it can survive without eating, just like with birds in nature. The default setting is 5000ms (5 seconds). If a bird can’t find food to eat within that time frame, it will die of starvation. In the addxbirds scenario, if any bird dies (for example Add2Bird) than any birds depending on that bird’s
food (Add3Bird, etc) will also die. In this scenario, it’s a cascade effect, as Add4Bird and Add5Bird also die of starvation. It doesn’t have to be this way – it’s just the way this scenario is setup.

Starvation is one of the ways that Avian Computing and ConcX dynamically adjust the usage of resources to match the current demand. Birds that aren’t being fed regularly enough are unneeded and their thread is stopped.

What if you want your bird to never die from starvation? It’s easy – just set the Stamina value to less than zero. The Stamina label says “–1” prevents starvation, but it can actually be any negative value. I normally just put a minus sign in front of whatever value is in the Stamina field and remove the minus sign if the bird should be mortal once again.

For those sharp-eyed folks who noticed that pods aren’t added to the TupleTree at exactly 7.5 second intervals, there are two reasons for this. First, the Nap Length is a maximum nap length and the actual length of each nap event is a random value between 5ms and the value entered in this field. Because it is a random value, there is the potential for two or three naps in a row to be short, allowing the bird to try to eat more frequently than other birds. But it is just as likely a few actual naps in a row will be long, resulting in reduced feeding.

A second reason for the unpredictability is the standard DigestionDelay method built into every BasicBird. This method also produces a random value between 5ms and the Nap Length value. Its purpose is to simulate some computations being performed during the digestion phase. If you don't want this delay, simply override the DigestionDelay method in the code for the bird and make it an empty method, thereby zeroing out the standard DigestionDelay. Or make it extra long. It’s your choice, depending on what you want to represent.

**Variation #3: One Fast Bird**

1. On Tab #1, set the Nap Length of Add1Bird to 25ms.
2. Click Clear All Activities.
3. Click Start All. The Add1Bird Activity tab ProgressBar grows a lot faster than the others (screenshot at the right).

The Add1Bird line looks shorter but by checking the auto-adjusting Max values
on the right side of the ProgressBars, it appears that the Add1Bird (1600) ate 3 to 4 times more than the other birds (400), which makes sense since the Add1Bird naps for only 1/3 as long as the other birds (25ms v 75ms). The actual numbers for the ProgressBars don't matter, but their relative sizes give a good indication of the amount of processing done by the individual birds.

Now look at the Food Supply tab (shown at the right). Again we can see the Add1Bird has been eating MagicPods (1600) much more frequently that the other birds are eating and storing BlackPods (800) quicker than the Add2Bird can process them, so a bunch of BlackPods have accumulated.

The AddxBirds scenario, as currently configured, represents a fairly normal situation, where one thread processes much more quickly than other threads. In this situation, the common way to try fix this speed mismatch is to try to optimize the slower thread(s) to better match the speed of the faster thread.

However, in Avian Computing, the preferred solution is to just add more of the slower bird(s). It is preferred because every code change can introduce mistakes into the code. It is safer and easier to just assign more birds (that you know are executing correctly) to the task rather than risking broken code.

**Variation #4: Adding Birds to Manage Fast Birds**

To meet the now plentiful output of the Add1Bird, add a new bird as follows:

1. Click Add New Bird at the center bottom of the screen.
2. On the new tab that pops up, in the (Bird) Type field, key in “ad” (the first two letters of Add2Bird) and then press the Tab key.
3. A list of birds that match what you entered is displayed. Click on the drop down arrow to show all of them.
4. Close the list and then click the Cancel button. Not Found is display in the field.
5. Replace “Not Found” with “add2” and press the Tab key.

6. The Type field now displays Add2Bird. Only one bird type begins with Add2 so the field automatically was updated with this bird and the dialog box wasn’t displayed.

7. Click in the Name field and replace “None” with “a2B”.

8. In the Food Eaten section, click on the Food Type field.

9. Replace “BasicPod” with “bl” and press the Tab key.

10. Select the BlackPod and click on the OK button.

11. In the Food Stored section, click in the Food Type field.

12. Replace “BasicPod” with “blu” and press the Tab key. The Stored Food Type field is updated with “BluePod”.

13. Click the Clear All Activities button in the bottom right corner.

14. Click the Start All button in the bottom left corner.

The Activity ProgressBars will all be growing somewhat more evenly, but if you click on the Food Supply tab (shown at the right) it is obvious that even with two Add2Birds working together they can’t keep up with the one superfast Add1Bird. BlackPods are still accumulating, but not as fast as before. And the Add3Bird can’t keep up with two Add2Birds, causing even more BluePods to accumulate. Adding a second Add2Bird just pushed the logjam downstream. We still have a mismatch in speeds, where two Add2Birds still can’t keep up with the Add1Bird.

Repeat steps 1 – 9 above and add another Add2Bird (named “a2C” that eats BlackPods and stores BluePods. Now when you Clear All Activities and click Start All, the Add2Birds do a better job of keeping up with the Add1Bird but, unfortunately, now the Add3Bird is totally overwhelmed.

So we could add two more Add3Birds so they can keep up, but that means that we’ll have to add two more Add4Birds and then two more Add5Birds. Not only does that seem like a lot of dull work, it also makes a lot of assumptions about the capabilities and timings of the AddxBirds (for example, Add1Bird’s naps got even shorter). Instead, let’s use the dynamic adjustment capabilities of ConcX and have the birds automatically reproduce when they are well fed.
Variation #5: Duplicating Birds to Manage Fast Birds

1. Click on #3 tab to select the Add3Bird.
2. Click on the Vitality tab.
3. Change the value in the Reproduce field from -1 to 150. The -1 sets the bird to never reproduce. The 150 sets the bird to reproduce when it has eaten 150 times in a row.
4. Repeat steps 1 – 3 for tabs #4 and #5.
5. Click Clear All Activities button.
6. Click Start All button.

After the birds have been flying for a while some extra food pods are accumulating because they are being created faster than they can be consumed. These surplus food pods ensure that the birds that eat them will always find food to eat. When these birds have eaten 150 times in a row, they qualified for reproducing and started hatching birds that were duplicates of themselves.

In the screenshot following, the last two birds listed were automatically generated when the Add3Bird a3a and Add4Bird a4a successfully met their reproduction criteria. The -1 at the end of their names was automatically added by ConcX and signify that this was the first duplicate bird to be hatched for Add3Bird a3a and Add4Bird a4a.

If the a3a-1 and a4a-1 birds reproduce, their first offspring will end in -1-1 (for example, a3a-1-1). If the original Add3Bird and Add4Bird reproduce again, their next offspring will have their names end with -2 and if those birds reproduce, their offspring’s names will end in -2-1, -2-2, -2-3, etc. This naming convention makes it possible to track which birds reproduced and how many times each one reproduced.
As the birds reproduce, the ConcX screen is updated accordingly. In this example, 16 new birds were added, enabling the scroll arrows on the tab row so you can see any of the birds (bird tabs).

As the birds reproduced, they quickly totaled more than the 10 birds that could fit on the first Activity tab. A new Activity tab was added to display 10 additional birds. And when there were more than 20 total birds, another tab was added to display these birds. ConcX will continue adding tabs past 100 birds but a lot more than 100 birds gets unwieldy in the current version of ConcX.

**Discussion of Variation #5**

This behavior of automatic sizing to match environment is more than just a convenience. It also fits the natural model that is experienced in the real world. When a new species of bird moves into a region and finds abundant food to eat, it reproduces often and its population quickly grows until it consumes so much food that the food starts to become scarce. When it is hard to find food, the reproduction rate goes down and may stop expanding altogether. Avian birds work the same way.

In other words, by emulating birds’ lives, we have simplified our programming lives. We no longer have to look for places where it makes sense to add additional threads or increase parallelization thru vectorizing the program because ConcX can automatically show us which birds are slow because those are the birds that are reproducing the most. And birds that die of starvation need to have the birds that provide them with food speeded up. No guessing. Set up the scenario and let it show you if there are performance issues and where.

It seems like this dynamic adjustment to demands would easily provide extremely useful information during the prototyping phase of development. Different birds could be configured to represent the performance of different parts of the system being developed. For example, a database record could be expected to arrive every 10ms so a bird could be configured to nap 10ms before putting a food pod into the TupleTree. The bird that eats that food pod could be estimated to perform an XYZ operation on a data record in 30ms so it would be configured appropriately. Any subsequent birds can also be configured to match their expected performance.

When the birds are all started, they will show you not only where performance bottlenecks
are located, but also show you where performance optimization could yield the biggest improvements to the system. By adjusting the nap time of a bird, it will emulate the performance of an optimized function and the overall system performance analyzed to determine if the improvements increased system performance.

**AddxBird Code**

Before we go any further, let’s take a quick look at the code that runs the AddxBirds. For now, let’s just say that AddxBirds extend BasicBird.java. If you want to know more, Chapter 3 covers the conceptual background for Avian Computing as well as how ConcX implements Avian Computing and Part 2 will cover more of the details of how the code modules work together.

Here is all the code for Add1Bird.java, just 53 lines including comments and blank lines. It’s just 30 lines if comments and blank lines are removed.

```java
package com.avian.birds.addbirds;

import com.avian.birds.basebirds.BasicBird;

/**
 * Eats a food object from the TupleTree, gets desc(ription), adds 1 to it, and
 * then stores it back in the TupleTree. It can only add 1 to the desc. Whatever food
 * it eats (BlackPod, GreenPod) it will always try to add 1 to whatever value it finds
 * in its desc.
 *
 * @author nchamberlain
 */

public class Add1Bird extends BasicBird {

    /**
     * Overrides afterDigestion in BasicBird with the instructions on how to add 1
     * to the value of the food that it ate. This bird expects the value being summed
     * to be kept in the contentsString instead of the contents Object. If digestFood1
     * is non-null, it tries to add 1 to that one's contentsString. If digestFood2 is
     * non-null, it tries to add 1 to that one's contentsString.
     */

    @Override
    protected void afterDigestion() {
        int contentsValue;

        if (!pod1.isEmpty()) {
            try {
                final String localContents = pod1.getDesc();
                contentsValue = Integer.parseInt(localContents);
            } catch (NumberFormatException nfe) {
                pod1.addToPodHistory("Non-numeric value in contents");
                contentsValue = 0;
            }

            contentsValue += 1;
            pod1.setDesc(Integer.toString(contentsValue));
        }

        if (!pod2.isEmpty()) {
            try {
                final String localContents = pod2.getDesc();
                contentsValue = Integer.parseInt(localContents);
            } catch (NumberFormatException nfe) {
                pod2.addToPodHistory("Non-numeric value in contents");
                contentsValue = 0;
            }

            contentsValue += 1;
            pod2.setDesc(Integer.toString(contentsValue));
        }
    }
}
```
contentsValue = Integer.parseInt(localContents);
} catch (NumberFormatException nfe) {
    pod2.addToPodHistory("Non-numeric value in contents");
    contentsValue = 0;
}

contentsValue += 1;
pod2.setDesc(Integer.toString(contentsValue));
3. Click on the Start All button. All 10 birds will start flying.

4. When the birds stop, select the Food Supply tab. The results will be just like before, with all but the first and last foods having just a few food pods. The last food will be shown on the second tab.

5. Select the TupleTree tab. The results will be similar to the following screenshot.

The IndigoPod is the last food type and all of them will have a value of 30 in their Description. Every IndigoPod will have the exact same sequence of birds eating that food pod, even though each one of the birds ate their food pod without coordinating or checking with any of the other birds.

And if you compare the event times recorded for the food pods, it can be seen that the birds never access a food pod at the same time. If you save this TupleTree to a file as described in Chapter 4 and then open the file with a program that will allow you to sort, you can view all of the times the individual events occurred and see that the events were spread across multiple birds, all with different event times. The image below shows the saved file opened with Excel and all of the lines loaded into column A.
Variation #7: Loading Even More Birds

1. Click on the Remove All button. This will delete all the existing birds.
2. Select File -> Open Flock File and navigate to the addbirds folder and open the add20.bird file.
3. Click on the Start All button. All 20 birds will start flying.
4. Before the birds stop, select the Food Supply tab and watch how all the food ProgressBars are flickering, except the first (MagicFood) and the last (SalmonPod) on the third tab. It’s all working just like before.
5. When the birds stop, select the TupleTree tab. It will have results similar to this screenshot.

Only one SalmonPod is shown because so many birds (20) digested it. If you review the other Salomon pods in the TupleTree, they will all have the value of 60 and they will all have been eaten by all 20 AddxBirds in the flock in the exact same sequence: first the axa birds, then the axb birds, followed by the axc and then axd birds.

Note that the total number of food pods in the TupleTree is pretty close to the results on previous runs with unmodified nap lengths: more than 350 and less than 400. This indicates that even though we have 20 birds flying concurrently, the system is still fairly lightly loaded. Older systems and single-core systems may have different results (YMMV).

Variation #8: Loading the Same Birds Multiple Times

1. Select File -> Open Flock File and navigate to the addbirds folder and open the add20.bird file.
2. Edit the names of birds 21 – 40 so the last digit E – H. I like to make these capital letters so the stand out as being in the second flock of birds loaded.
3. Click the Clear All Activities button in the bottom right corner.
4. Click on the Start All button. All 40 birds will start flying.
5. When the birds stop, select the TupleTree tab. The results will be similar to the screenshot at the right. Note that there are now approximately twice as many food pods in the TupleTree as before and the every SalmonPod has a value of exactly 60. And also that a mixture of the first flock and second flock digested each SalmonPod.

6. Open add20.bird file again.

7. Edit the names of birds 41 – 60 so their name ends with –I thru –L. Adding a hyphenated suffix to the bird’s name makes them stand out from the previously loaded flocks.

8. Click the Clear All Activities button.

9. Click the Start All button and all of the birds start flying.

10. When they stop flying, the results on the TupleTree tab will look similar to the screenshot at the right.

Note that once again every SalmonPod has a value of 60 and that every food pod has been eaten by the exact same sequence of bird types differing only in which flock they were loaded with.

In the SalmonPod on the right, the MagicFood was digested by the Add1Bird from the third flock, but the Add2Bird that ate the BlackPod was from the first flock. The next two pods were digested by third flock birds, but the Add5Bird is from the second flock.

This interchangeability is what gives Avian Computing and ConcX its scalability: multiple instances of a specific bird type can be added without special considerations or requirements. Add 10 or 100 of the same type of birds and the only thing that matters is food availability.
Adding more add20.bird flocks is left as an exercise for the reader to continue according to their interest and patience at editing bird names.

All of my results are run on a 5+ year old computer with an early Intel i-7 quad core CPU with HyperThreading. On my system, it has to have about 100 AddxBirds loaded before it starts to show noticeable CPU load.

The screenshot on the right shows the CPU load on my system. All four cores and all four hyper-threaded cores are experiencing pretty much the same CPU load: they all start to ramp up slowly and then quickly build to the right. I suspect that the big ramp up is when the CPU is starting to fall behind and starting to thrash because it is handling so many in-process threads. And then all the cores level off, I suspect because the CPU speed is bumped to its maximum frequency at that moment. And then all of loads suddenly drop to zero, probably because that is 30 seconds into the run and the birds’ lifetimes have all expire. Interested readers can change all of the birds lifetimes to 60,000 (or 1 minute) to see if this changes how the CPU load is experienced by the system.

**Variation #9: Limiting MagicPods**

1. Tabs #21 and #41 should both be Add1Birds that eat MagicPods. If your numbering is different, find the appropriate tabs. On these tabs, add “xx” to the end of the Bird Type and press the Tab key. They will change to Not Found and the other fields will grey out (disabled).

2. Click the Clear All Activities button.

3. Click the Start All button. The birds will start flying normally except for birds #21 and #41.

4. Click on the Food Supply tab. All of the foods will be flickering but almost all foods will show zero or just one food pod.

5. When they stop flying, note that only a few food types will have any pods at all.

6. Click the TupleTree tab. Almost every food pod in the tree will be a SalmonPod.

Because only one bird (Add1Bird named a1a on tab #1) was putting BlackPods into the system and three birds were eating BlackPods, any BlackPods put into the tree were eaten.
almost instantaneously. The same was true for all subsequent food pods; too many consumers and not enough producers for more than one or two food pods to remain in the tree except for the final product, the SalmonPods. There was no planning or controlling or coordinating the efforts of any of the birds, and yet they all managed to share the food that was there well enough that they all survived.

**Variation #10: Circular Eating**

1. On Tab #21, replace Not Found with “ad” and then press the Tab key.
2. On the pop up dialog, highlight the Add1Bird and click on OK button or press Enter. The previously entered food types will become enabled again.
3. On Tab #41, change the Not Found to “add1” and the Add1Bird will automatically be filled in for the Type and the food types will become enabled again.
4. Change the Food Type from MagicPod to SalmonPod.
5. Click on Clear All Activities button.
6. Click on the Start All button and the birds will all start flying and everything will look normal.
7. When they have stopped flying, click on the TupleTree tab. The total number of food pods in the tree should be fairly normal for two birds eating MagicPods, but notice that some of the values for SalmonPods are not 60 but are 120 or even 180.

In the screenshot at the right the SalmonPod has a value of 180, which means that it went thru all 20 birds three times. Any SalmonPod that has a value greater than 60 will have the lines that are boxed on the right that shows that bird #41, a1a-I ate the SalmonPod and saved it as a BlackPod where various Add2Birds could eat it and start the whole process all over again.

The purpose of this exercise was to demonstrate that the birds don’t care if they are processing the same food pods multiple times; food is food. However, if you find that you are getting odd results, be sure to review the events for the deviant food pods to see if more birds are eating the food pod than should be eating it.
**Variations Using MathBird**

One of the shortcomings of the AddxBirds is that they can only do addition and they can only add set amounts; an Add1Bird can only add 1 and a whole different bird is needed if you want to add 2 (or 17). The MathBird provides the functionality to do any one of the basic math operations (addition, subtraction, multiplication and division) using any numeric value.

For example, the math5.bird will perform this formula in parallel:

\[
\frac{(3+2) \times 5 - 8}{4}
\]

It’s not a meaningful equation. It exists just to illustrate the basics of MathBirds.

MathBirds also illustrates a useful feature of ConcX: Externally Managed Variables or XMV. XMV allows values to be passed into the system without using command line arguments being passed into the program. While passing arguments into the system are functional, they are not “self-evident”. There is no way that you can look at a program and tell that it requires runtime parameters that need to be passed. I think we’ve all had interactions similar to the following:

```
C:\mystuff> doIt.exe
C:\mystuff> Please supply name
C:\mystuff> doIt.exe Nelson
C:\mystuff> Please supply date
C:\mystuff> doIt.exe Nelson May 15, 2016
C:\mystuff> Please supply date
C:\mystuff> doIt.exe Nelson 5/15/2016
C:\mystuff> Please supply date
C:\mystuff> doIt.exe Nelson 5-10-2016
C:\mystuff>Nelson - today’s date is October 5, 2016
```

You’ve probably noticed the XMV tab available for each bird. This tab provides 10 fields in which you can enter the XMV values that you want to pass into your program. Each of the 10 XMV Value fields also has a matching Name field so you can document what the value represents. If you use XMV Values, it is strongly recommended that each value has a matching entry in its XMV Name field. The XMV Names and Values are both saved in the flock file so you will have a long term record of what values you were passing into your program.
MathBirds can then be configured to perform different math operations using different values. In this screenshot, the m5 MathBird divides whatever amount contained in the food pod that it eats by 4.0. Each of the birds in the math5.bird file is exactly the same kind of bird, a MathBird. And yet each of them performs a different operation using different values simply by configuring them.

Imagine trying to configure 5 birds using 10 different command line arguments. Imagine having to type all 10 arguments in correctly every time you run your program. XMV variables can make your life much simpler.

Let’s load and run the math5.bird file.

1. If required, close ConcX to clear out any loaded birds and then restart ConcX.
2. Click File menu and then select Open and navigate to the addbirds folder.
3. Open the math5.bird file. Be sure to click on the XMV tab for each of the birds and look at the values and operations associated with each of the birds.
4. Click the Start All button. All the birds will start flying.
5. When you get bored, click the Stop All button.

When the math5.bird file loaded, it should have looked pretty familiar, with tabs #1 - #5 containing 5 MathBirds, named m1 – m5. The first MathBird eats MagicPods just to get things started, gives it an initial value of 3 and then stores it in the TupleTree as a BlackPod. The other MathBirds then eat their food pods in order asynchronously and in parallel and resulting in a bunch of RedPods being placed in the TupleTree, each with a value of 4.25

Wrap Up

The AddxBirds scenario(s) demonstrated several parallel programs of deceptive simplicity. The scenario(s) demonstrated that it is fairly simple to develop and modify parallel applications by putting together sequences of birds. The output (food pod) of one bird was the input (food pod) for a different bird with all the birds working asynchronously and concurrently. This easy to imagine method of sequencing the operations makes it possible to put together “programs” that work in parallel without conditional statements or timing/testing or any kind of master scheduler. Just start them flying and they figure it out on their own.

Because ConcX is considered a development tool, it features a suite of “trackers” that always run when you start one or more birds. These trackers provide real-time feedback (ProgressBars) on the birds as they are flying as well as logging all the events that occurred as they were flying for examination when the birds stop flying.
While the AddxBirds scenarios are quite simple, they are really abstractions or skeletons of more complicated real-world applications. Two potential applications that could be implemented using advanced versions of AddxBirds are ETL (Extract, Translate, Load) applications and Customer Billing applications.

Most importantly, this chapter introduced many of the built-in features available in ConcX, such as configurable performance settings (nap times, reproduction counts, stamina, etc.) and XMV variables. It also showed how short the code can be to produce fully-functional birds.
Chapter 6. The Producer-Consumer Scenario

The Producer-Consumer Problem
If you look it up on Wikipedia, the Producer-Consumer problem will be described as follows:

The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

ConcX (and the Avian Computing perspective) provides a simple solution to this problem without customization. Simply add two BasicBirds, configure them, and then run them. The “producer” bird eats MagicPods and puts them into the buffer (TupleTree) at an independent, self-determined rate. The “consumer” bird removes one food pod at a time from the buffer (TupleTree), also at an independent self-determined rate (although this isn’t explicitly stated).

One reason that this problem even exists as a “problem” in concurrency is that the buffer is assumed to be a passive component in the system. The assumption is that the buffer doesn’t participate in the management of itself; only the Producer and Consumer have the smarts needed to manage the buffer. In Avian Computing, the TupleTree performs the role of the buffer but instead of being passive, it is an active participant in the exchange of the piece of data produced by one bird and consumed by the other bird.

By creating a “smart buffer” we reduce the Producer-Consumer problem to the status of triviality.

Producer-Consumer Setup
Although we could just load the ProducerConsumer1Pair.bird file, it is helpful to set it up all ourselves, especially since we are only configuring two birds. We are going to use BasicBirds that are just slightly modified for this initial solution to the Producer-Consumer problem. A more sophisticated solution will be presented later in this chapter, and a more realistic solution will be presented in the following chapter.

For this scenario, start ConcX from NetBeans and then:

1. On Tab #1 (or first available bird tab), click in the (Bird) Type field and key in “prod” and the press the Tab key.
2. The Bird Type Selector dialog box is displayed.
3. Click on the ...prodcon\Producer option and then click on the OK button.

4. Click on the Name field and enter the name “prod1” (without the quotes).

5. Click in the Food Eaten Food Type field and key in “magic” and press the Tab key. MagicPod is displayed in the Food Type field.

6. Click in the Food Stored Food Type field and key in “prod” and press the Tab key. Select the ProductPod option from the dropdown list and click the OK button.

7. When you’re done it should look like the screenshot on the right.

8. Click on the Add New Bird button in the bottom center of the screen.

9. Tab #2 will be added, a blank bird ready to be configured.

10. Click in the (Bird) Type field and key in “cons” and press the Tab key.

11. Select the prodcon\Consumer option and then click the OK button.

12. Click on the Name field and enter the name “cons1” (without the quotes).

13. Click in the Food Eaten Food Type field and key in “prod” and press the Tab key.

14. Select the prodcon\ProductPod option and then click the OK button.

15. Click in the Food Stored Food Type field and key in “prod” and press the Tab key.

16. Select the ProductProcessedPod option and then click the OK button.

17. The second bird will look like the screenshot at the right.
Two birds have now been set up to run the Producer-Consumer problem. The first bird, prod1, is the Producer, so it eats MagicPods (that don’t have to previously exist) and stores them as ProductPods in the TupleTree. After storing the ProductPod, the Producer takes a nap and then repeats eating and storing MagicPods and ProductPods, respectively.

The Consumer bird on tab #2 eats the ProductPod and then stores ProductProcessedPods back in the TupleTree. The Producer-Consumer problem definition doesn’t state what is supposed to happen to the “product” once it has been consumed; the actual piece of data is of no interest to the problem, so it would be safe to assume that it is discarded. But for this initial solution, let’s put it back in the TupleTree as a ProductProcessedPod so we can readily identify which items have been processed, how many items have been processed, and which ones are still waiting processing. This new name also prevents from the Consumer from accidentally consuming the same ProductPod twice, which could happen if we put ProductPods back into the TupleTree.

1. Click that Start All button. The ProgressBars on the Activity tab will start growing at approximately even rates.

2. Click the Food Supply tab – the ProgressBars for MagicPod and ProductProcessedPod will both be growing but the ProductPod will be flickering under the stress of the Producer constantly putting new ProductPods in the TupleTree while the Consumer is constantly removing the ProductPods.

3. Click the Stop All button (or wait for 30 seconds for the birds to stop on their own). Click on the TupleTree tab and it should look similar to this screenshot. There is one ProductPod and a bunch of ProductProcessedPods. The actual number of each type of pod will vary depending on how long you let them run.

The Desc of the ProductPod mentions only that it was “Produced by prod1” but the Contents Desc of the ProductProcessedPods all mention both prod1 and cons1, demonstrating that the products were both processed by both birds. Adding the names of the birds that digested the food pods to the Desc of the pods is the only difference between BasicBirds and the Product birds and Consumer birds.

ConcX has fulfilled the basic requirements of the Producer-Consumer problem almost without
changes. The Producer can put items (ProductPods) into the buffer (TupleTree) without conflicting with the Consumer and the Consumer can remove items from the buffer without conflicting with the Producer. No buffer over-runs, no crashes when the Consumer attempts get data from an empty buffer. The smart buffer (TupleTree) handles it all for us.

Fussy readers will have noticed that the original problem statement says that a “fixed-size buffer” is used to pass data. So for this example, the fixed-size of the buffer is assumed to be the maximum practical size of the TupleTree or many thousands of food pods. And since the Producer and Consumer are producing and consuming at approximately the same rate, the number of items to be consumer (ProductPods) in the buffer remains small and under control.

If we follow the assumption that the ProductPod can be discarded, that is simple enough to do. Instead of having the Consumer store ProductProcessedPods, the Consumer can be set to store NullPods. When a BasicBird (or any of its subclasses) is told to store a NullPod, it simply discards whatever food pod it digested. It stores nothing back in the tree. This is useful in birds that write data to databases and have no further need for the food pod.

To see how this scenario changes using a NullPod:

1. Click on the Consumer tab.
2. Click in the Food Stored Food Type field and replace ProductProcessedPod with “nu” and press the Tab key.
3. The Food Stored Food Type field is updated with NullPod.
4. Click Clear All Activities button.
5. Click the Start All button.

The Activity ProgressBars should grow normally. However, selecting the Food Supply tab shows that the ProgressBar for the ProductProcessedPod is now zero and the ProductPod ProgressBar either zero or very short.

The NullPod will show the same quantity as the MagicPods, but if you click on the TupleTree tab there will be only a few ProductPods in it. The NullPods were in fact discarded but their processing was still recorded on the ProgressBar. All that is left is a few ProductPods that had not yet been processed by the Consumer.
Variation #1: Slow Producer

Let’s see what happens when the Producer puts pieces of data (ProductPods) into the buffer (TupleTree) more slowly than the Consumer eats them.

1. Click on the Producer tab.
2. Click on the Vitality tab.
3. Change the Nap Length field to a much larger number, such as 750ms or 2275ms or about 2.25 seconds. The Producer is now set to taking much longer to generate a piece of data (ProductPod) than it takes for the Consumer to eat it.
4. IF required, click on the Consumer tab and set the Food Stored Food Type to be ProductProcessedPod.
5. Click the Clear All Activities button.
6. Click the Start All button.

It works pretty much the same as it did earlier, but the Activity ProgressBars advance more slowly than before. Sometimes you may even detect a relatively long pause before the Producer's progress bar advances, followed almost immediately by the Consumer's progress bar advancing. This jumpiness happens because the Producer took a long nap before generating the next ProductPod, and all the Consumer could do is keep checking the TupleTree until it could finally eat (and update its ProgressBar).

Click on the Clear All Activities button.

Note the asymmetrical dependencies in this configuration. If you select the Producer tab and then click on the Start Me button, the Producer will produce ProductPods without any problems but very slowly.

Click on the Clear All Activities button.

However, if you select the Consumer tab and then click the Start Me button, the Consumer refuses to run because it depends on the Producer to put ProductPods into the TupleTree. Without ProductPods to eat, the Consumer eventually starves and terminates early.

Variation #2: Multiple Slow Producers

One of the values of ConcX is that we can cheaply experiment with the application’s configuration. In this variation, add three more Producer birds per steps 1 – 6 above, but name them
prod2 – prod4. Set the new Producers to the same nap length that Producer 1 was set for (in my example, 2275ms). The Activity ProgressBars should look similar to this screenshot.

Click Clear All Activities button and then the Start All button. Now four Producers are generating pieces of data that the Consumer is eating but the Consumer is still waiting much of the time for a ProductPod.

While it’s possible to just keep adding slow Producer birds until they can produce as much as the Consumer can eat, it is simpler in this variation to use their nap lengths to figure out the needed values. The Consumer is set to nap for 75ms, so with four Producers, they can each be set to 300ms (4x75ms = 300ms) and their combined performance should be able to keep up with the Consumer.

If you make those changes, clear the Activities and Start All birds, you will get Activity similar to the upper screenshot, with the Consumer bird eating about four times more than the Producers. And if we look at the lower screenshot, the number of ProductProcessedPods is almost as large as the number of MagicPods, meaning that the Producers are just slightly outproducing the Consumer. This is confirmed by the number of ProductPods waiting to be eaten by the Consumer (bottom screenshot).

This is actually a useful observation. If we are designing a multi-threaded system that we want to run with the minimum amount of restrictions or bottlenecks, we need to know that all threads are running at full speed and not just waiting for some other task to complete. By modeling our design in ConcX, we can assign nap times to each of the birds based on real-world constraints, such as sensor readings per
second or disk reads per second, etc. And then by running the app in ConcX, the areas where performance mismatches will not only reveal themselves but also indicate if optimization is the best choice or if simply providing additional threads can meet the performance requirements.

**Variation #3: Multiple Pairs of Producer-Consumers**

1. Close ConcX and then start it again.
2. Select Open Flock File on the File menu and then navigate to the prodcon folder. Open the ProdCon2Pair.bird file.

Four birds will have been added, two Producers and two Consumers. They all eat the same foods as before and store the same foods as before.

3. Now click Start All button and all the birds will start flying like before. On the Food Supply tab, it will be just like before with the only difference that MagicPod and ProductProcessedPod ProgressBars will grow faster than before.

When the run is over, the TupleTree will be much like before but the ProductProcessedPods will have been processed by all combinations of Prod1/2 and Cons1/2.

4. Now select Open Flock File on the File menu and then navigate to the prodcon folder. Open the ProdCon3Pair.bird file.

Six birds will have been added, three Producers and three Consumers. They all eat the same foods as before and store the same foods as before.

5. Click the Clear All Activities button and then click the Start All button. All the birds will start flying like before. On the Food Supply tab, it will be just like before with the only difference that MagicPod and ProductProcessedPod ProgressBars will grow faster than before.

When the run is over, the TupleTree will be much like before but the ProductProcessedPods will have been processed by all combinations of Prod1/2 & PROD1/2/3 and Cons1/2 & CONS1/2/3.

**Saving Bird Configurations**

Before going any further, click on the File menu and then select the Save Current Birds option. Navigate to the prodcon folder and then give your current configuration a name such as ProdCon5Pair.bird. I use “.bird” as the file name extension but this is just something I like to use. The file will be saved as an XML file, so you could give it a .xml file extension or any file extension that you like.

Here’s an excerpt from the file just saved:
<?xml version="1.0" encoding="utf-8" ?>
<birds>
  <bird>
    <birdname>prod1</birdname>
    <birdtype>com\avian\birds\prodcon\Producer.class</birdtype>
    <eatsfood1>com\avian\foods\basefoods\MagicPod.class</eatsfood1>
    <eatsfood2>Not Found</eatsfood2>
    <eatsandor>ONE</eatsandor>
    <storesfood1>com\avian\foods\prodcon\ProductPod.class</storesfood1>
    <storesfood2>Not Found</storesfood2>
    <storesandor>ONE</storesandor>
    <infolevel>Config</infolevel>
    <lifetime>30000</lifetime>
    <stamina>5000</stamina>
    <patience>5000</patience>
    <naplength>75</naplength>
    <reproduce>-1</reproduce>
  </bird>
  . . . . .
</birds>

All of the fields that have values were saved. Empty fields, such as the XMV fields in this example, are not saved.

**Variation #4: Tightly Linked Producer-Consumer Scenario**

The original scenario let the Producer generate ProductPods at its natural rate (based on its Nap Length setting) regardless of the status of the Consumer and how many ProductPods it was eating. And the Consumer was free to look for ProductPods at its natural rate (based on its Nap Length) regardless of whether or not the Producer was producing. The birds were unaware of each other and ran independently of each other, as long as there was food.

How would the application change if the Producer-Consumer problem was restated so the Producer could only generate a piece of data when the Consumer was ready to consume it? In this scenario the Producer and Consumer are tightly linked, needing to communicate their status to the other bird so they never produce or consume at the wrong time. For example, a Just-In-Time manufacturing system, where parts (seats, radios, door panels, etc.) arrive on the production line at the exact moment that they are needed instead of the traditional procedure of having the parts delivered early and stockpiled in a warehouse.

A common solution might include a shared variable that operates as a flag of some sort indicating which thread was allowed to produce or consume. A significant amount of time has been spent developing optimal schemes for ensuring that the flag is managed effectively
and is never accessed with a stale state.

Fortunately, the tightly linked scenario is easily managed in ConcX because the TupleTree again functions as an intelligent participant in the process. The TupleTree can be told to store any food pod as a Singleton. This effectively makes that food pod function as a shared flag that controls the operation of Producer and Consumer.

The TupleTree will allow zero or one Singleton food pod in the tree at any given moment (more than one Singleton food pod is an error). If the Singleton food pod is in the tree and a bird requests/receives it, then the bird has permission to perform its operation (produce or consume). If there is not a Singleton food pod in the tree, the bird that requested it cannot receive it so the bird does not have permission to perform its operation.

In this scenario, the Producer bird stores a ProductPod in the TupleTree as a Singleton. The Consumer gets the one and only instance of the ProductPod, processes the pod, and then stores a ProductMakeMorePod in the tree as a Singleton. The Producer gets the one and only instance of the ProductMakeMorePod and delivers a new ProductPod to store in the TupleTree. By using separate food pods, it prevents a bird from interpreting its own food pod (flag) as permission for it to operate again.

Singleton food pods use “lock” pods help manage the process. Without locks, it is impossible to tell the difference between the situations where a food pod has never been stored in the TupleTree and where a type of food pod has been stored in the TupleTree but someone is using it. In the first case, it is OK to store the absent food pod but in the second case the absent food pod should not be stored because there would then be two food pods of that type in the system.

Let’s try out this scenario.

1. Click on Remove All button.
2. On the File menu, select the Open Flock File option and then navigate to the prodcon menu.
3. Open the TightProdcon1Pair.bird file.
4. Click the Start All button. The Activity tab ProgressBars should both advance at a leisurely pace (they both take 275ms naps instead of the 75ms of earlier examples).
5. After 15 seconds, they will both stop flying.
6. Click on the TupleTree tab. It will
look similar to the screenshot and list the two locks needed to allow Singletons and only one other food pod, the flag food pod that is being passed back and forth between the Producer and Consumer.

Note that the lines beginning with “Last eaten by…” are part of the food pod’s Desc, with one line for each time the food was eaten. The times and events are still there in the file, they are just further down, in the History section.

The New Value (number at the end of each line) is a random number that is generated by each bird each time they digest their food. The random number is stored as a seed in the food pod. Note how the New Value for the Producer becomes the Old Value for the Consumer, and vice versa. Each time the Producer or Consumer processes the food pod, they add to their own info the Desc of the food pod.

If you wish to investigate or validate the random numbers this further:

1. Click on Tab #2 (ConsumerTight).
2. Click on the History tab.
3. Set the Level of Info to Fine.
4. Click on Tab #1 (ProducerTight).
5. Click on its History tab.
6. Set the Level of Info to Fine.
7. Click on Clear All Activities button.
8. Click on Start All button.

When it has finished running, click on the TupleTree tab and compare the numbers in the Pod History to the numbers recorded in the Producer and Consumer History files. The Producer will have generated every-other-number and the Consumer will have generated the other every-other-number.

To demonstrate that the Producer and Consumer really are tightly linked:

1. Click Clear All Activity button and then
2. Click the Start All button and then
3. After a second or two, click the Stop Me button on either bird. Their ProgressBars will both stop advancing.
4. After a few seconds, click the Start Me button and both ProgressBars will once again start moving.

5. Select the other bird tab and click its Stop Me button. Again, the ProgressBars for both birds will stop.

6. After a few seconds, click Start Me again and they will both start flying again.

If you check the History tab of the bird you did NOT stop, you will find a long string of Napping events. While the other bird was stopped, this one is dutifully checking for food until either you restart the other bird or until this one runs out of stamina and dies from not eating.

**Producer-Consumer Wrap Up**

The Producer-Consumer problem can be easily solved using ConcX because the Producer-Consumer model fits very closely to the Avian Computing model. The Producer-Consumer problem can be solved in ConcX using BasicBirds without modifications – just configure them to eat corresponding food pods and they will happily do so all day long.

The TupleTree is the star in this scenario. The TupleTree is a smart and active participant in this scenario. Instead of flags and semaphores and all that other junk that developers usually have to worry about to share a flag, the TupleTree manages all of those details and lets birds be birds and just look for food.

The Producer-Consumer specific birds and foods developed for this problem were developed just to more fully demonstrate that they are fulfilling the goals of the problem statement.

It took more specialized code to extend and enhance the Producer-Consumer scenario to manage a buffer with a single item and to use that item to control the flow of food pods through the intelligent buffer (TupleTree). In this example, two birds were “tightly linked” by storing their food pods in the TupleTree as Singletons. The food pods also passed data (the random integer) in two directions to prove that they really were interacting.

A more realistic (and complex) Producer-Consumer scenario is studied next.
Chapter 7. A More Realistic Producer - Consumer

A more realistic Producer-Consumer scenario might be a soft drink bottling plant. It might have several different vats where they mix different soft drinks, such as orange soda, grape soda, and a cola drink. Each type of drink has different mixing durations and bottling rates, so they arrive at the boxing station at irregular intervals.

For this example, we assume that each bottle or can of soda arrives at the boxing station as soon as the container has been filled and closed (capped or sealed). Each bottle or can is numbered with a unique ID number and needs to be put into case of similar types of soda. Each type of soda has a different quantity of containers in the case to accommodate different size bottles or cans (8oz, 12oz, 16oz, etc). When a case of soda is full, it is moved to a storage location in the warehouse, managed by an ERP system or the warehouse manager.

Another example that would fit this scenario is a pharmaceutical manufacturing facility where every vial or bottle would have its own ID that would be tracked from the time the vial or bottle was filled until it was delivered to a customer. Every vial or bottle would be associated with a batch and if there was ever a problem with a specific batch, every vial or bottle could be tracked to the case that it was packed in and who purchased and distributed the contents of each case. Setting up such a system based on the Soda Producer-Soda Consumer system presented here is left as an exercise for the reader.

For the scenario in this Getting Started guide, a Soda Producer bird and a Soda Consumer bird were created. The Soda Producer extends the ProducerTight explored in the last chapter; the only thing that it does differently is that it assigns an ID to the individual bottles or cans of soda. Since this is a realistic example (and not real world), the IDs are random integers with no checking to verify that they are unique or sequential or whatever. Random integers is good enough for this example.

The Soda Consumer extends the regular ConsumerTight from the last chapter and only does a few things differently. Since the Soda Consumer always puts a bottled soft drink into the appropriate case of soda, it needs to be sure that the appropriate type of “case of soda” food pod exists before it starts working. To do this, it overrides the doFirstTimeProcesses method to add this food pod (case being loaded with soda) to the TupleTree the first time after the bird has been started.

The other difference is that after the bird has digested its food (soda), the afterDigestion method is overridden to put the soda bottles into the case of soda. When the case is full, a new food object, a FullCaseOfSoda, is made from the case of soda object and put into the TupleTree.

Although it is not explicitly done in this simulation, a FullCase bird could be created that would continuously check for full cases in the TupleTree, and when it found one, it would move the full case of soda to the ERP system for moving to the warehouse.

Load and run this scenario as follows:
1. If necessary, click on Remove All button to get rid of any pre-existing birds.

2. From the File menu, click on the Open Flock File option and then navigate to the prodcon folder. Open the Soda1flavor.bird file.

3. After loading, it should look similar to the following screenshot.

![Screenshot of Avian Computing interface]

The Producer and Consumer birds are set up just like they were set up in the last chapter with the exception that the SodaConsumer eats two foods and stores two foods. Second foods eaten or stored can be added manually to scenarios when required by clicking on the AND/OR radio buttons, which enables the second Food Type fields. Selecting the second food is just like selecting the first food; replace any text in the field with the start of the second food and press the Tab key. If more than food pod matches your entry, select the desired food pod, CaseOrangeSodaPod in this case.

The logic for the SodaConsumer works like this: it looks for the first food (OrangeSodaPod) and when it finds one, it tries to get the second food, CaseOrangeSodaPod. When it has both, it moves the OrangeSodaPod (1 container of orange soda) into the Orange Soda case. It then tells OrangeProd1 that it is ready to receive another one by putting the OrangeSodaOkPod into the TupleTree and then puts the second food (CaseOrangeSodaPod) into the TupleTree.
NOTE

The SodaConsumer eats and saves the same food pod, a CaseOrangeSodaPod, which is somewhat unusual. Most of the time, a food pod is processed just once by a bird and the transformed version of the food pod is stored as a different type of food pod. In this scenario, however, it takes it out of the tree when it needs it and then puts it back when it's done.

Why not just hold the CaseOrangeSodaPod until it is full and then store the full case? The first reason for always putting the case back in the TupleTree is because holding the CaseOrangeSodaPod introduces a lot of complexity into the SodaConsumer.

- **First complication:** once the SodaConsumer has eaten an OrangeSodaPod, the SodaConsumer must figure out if it already holding a CaseOrangeSodaPod or if it needs to get one from the TupleTree. If it already is holding a CaseOrangeSodaPod, custom code will have to be written to by-pass looking for the CaseOrangeSodaPod. It is much simpler to let the SodaConsumer do what it does naturally which is to eat food and then to store the resulting food. No custom code to maintain.

- **Second complication:** in situations where you have more than one SodaConsumer and each will hold a CaseOrangeSodaPod until the case is full, what if the second SodaConsumer is ready to put a bottle in the case and the CaseOrangeSodaPod is being held by the other SodaConsumer? Should the second SodaConsumer start a new case or should it wait for the original CaseOrangeSodaPod to become available? If it creates a new case, what if it doesn’t get enough bottles to fill it, what becomes of the partial case? If it waits for the case to become available, how long should it wait? If their wait time is exceeded, should they put back their OrangeSodaPod or should they start a new case? If they start a new case, which of the SodaConsumers should create it. And then what if the previous case does become available? Will the system that’s supposed to work on one case at a time suddenly have more than one case to fill? It is much simpler to let the SodaConsumer do what it does naturally which is to eat food and then to store the resulting food.

The second reason is that Avian Computing and ConcX expect to get food pods from the TupleTree, digest (modify) them, and then put them back in the TupleTree. This simple sequence matches real life bird behavior - when was the last time you saw a bird carrying around a box of stuff that it was going to need later? This also makes life easier for developers since they don’t have to write the code that insures that objects are correctly and fairly shared.

The scenario in this chapter also makes extensive use of BasicSeeds that are held in the food pods. The first seed in every case pod (CaseCherryColaPod, CaseGrapeSoda, etc) defines the maximum number of items that go into that case. When a unit of soda is added to a case, what really happens is a new seed is added to the case pod. By comparing the quantity of seeds in a case pod to the value in its first seed, it knows when the case is full. Full cases are then stored in the TupleTree and the case pod is emptied and readied to be refilled.
New seeds are added to pods similar to the following code snippet.

```java
this.addSeed(new BasicSeed("Max Qty in Case", "Orange Soda 12 fl oz can", 16));
```

“this” is the case Food pod which extends BasicPod which has the addSeed() method that adds a BasicSeed to the end of its internal array of seeds. In the above code snippet, it also creates a new BasicSeed to hold the case info. “Max Qty in Case” is inserted into the seed’s “desc” field, “Orange Soda 12 fl oz can” is inserted into the seed’s “type” field, and “16” is inserted into the seed’s “kernel” field. The seed’s kernel is actually an Object field so it can hold any kind of object. In this line of code, the 16 numeric primitive is autoboxed into an Integer Object with a value of 16. By changing the kernel value in the first seed in a case pod, the number of seeds (soda containers) that will be held in that type of case pods will be changed correspondingly.

When you click the Start All button, the activity progress bars for the SodaProducer and the SodaConsumer will advance evenly. If you click on the Food Supply tab while running, you'll see that the amount of food remains pretty small. The small amount of food pods is a good indicator that all of the generated units of soda are being consumed.

After the birds stop (or after clicking the Stop All button) the TupleTree tab will look similar to this screenshot. The FullCaseOfSoda in this screenshot has 17 seeds in its pod. The first seed defines the maximum number of sodas in the case, followed by this maximum number of seeds (units of soda) defined in the first seed. Following the number of seeds in the pod are history events for that pod that show exactly when each unit of soda was added to the case and which bird added the soda.

What about the ID numbers of all of the bottles or cans in the case? How do we see those? The short answer is that the pod details are not shown in the TupleTree to keep its display manageable. However, the pod details are available when you save the TupleTree contents to a file. Do the save as follows:

1. In the field at the center bottom of this tab is the Save filename field.
2. Type in an appropriate file path and name, such as “C:\asp\caseOfSoda.txt”
3. Click the Save button (pressing Enter key doesn’t save).
Use the editor of your choice to open the file that you just saved. Its contents will look similar to the following:

```
Key = FullCaseOfSoda
  Contents Pod: Full Case of Orange Soda
    Contents Object = class con.avian.foods.BaseFoods.BasicSeed
      ... BasicSeed(desc=Max Qty in Case, type=Orange Soda 12 fl oz can, kernel=16)
    Contents Object = class con.avian.foods.BaseFoods.BasicSeed
      ... BasicSeed(desc=1 unit of soda, type=Orange Soda 12 fl oz can, kernel=390274)
    Contents Object = class con.avian.foods.BaseFoods.BasicSeed
      ... BasicSeed(desc=1 unit of soda, type=Orange Soda 12 fl oz can, kernel=242276)
    Contents Object = class con.avian.foods.BaseFoods.BasicSeed
      ... BasicSeed(desc=1 unit of soda, type=Orange Soda 12 fl oz can, kernel=129222)
    Contents Object = class con.avian.foods.BaseFoods.BasicSeed
      ... BasicSeed(desc=1 unit of soda, type=Orange Soda 12 fl oz can, kernel=92644)
    Contents Object = class con.avian.foods.BaseFoods.BasicSeed
      ... BasicSeed(desc=1 unit of soda, type=Orange Soda 12 fl oz can, kernel=371361)
    Contents Object = class con.avian.foods.BaseFoods.BasicSeed
      ... BasicSeed(desc=1 unit of soda, type=Orange Soda 12 fl oz can, kernel=263827)
    Contents Object = class con.avian.foods.BaseFoods.BasicSeed
      ... BasicSeed(desc=1 unit of soda, type=Orange Soda 12 fl oz can, kernel=532093)
    Contents Object = class con.avian.foods.BaseFoods.BasicSeed
      ... BasicSeed(desc=1 unit of soda, type=Orange Soda 12 fl oz can, kernel=935167)
    Contents Object = class con.avian.foods.BaseFoods.BasicSeed
      ... BasicSeed(desc=1 unit of soda, type=Orange Soda 12 fl oz can, kernel=440267)
    Contents Object = class con.avian.foods.BaseFoods.BasicSeed
      ... BasicSeed(desc=1 unit of soda, type=Orange Soda 12 fl oz can, kernel=203102)
    Contents Object = class con.avian.foods.BaseFoods.BasicSeed
      ... BasicSeed(desc=1 unit of soda, type=Orange Soda 12 fl oz can, kernel=535426)
    Contents Object = class con.avian.foods.BaseFoods.BasicSeed
      ... BasicSeed(desc=1 unit of soda, type=Orange Soda 12 fl oz can, kernel=520058)
    Contents Object = class con.avian.foods.BaseFoods.BasicSeed
      ... BasicSeed(desc=1 unit of soda, type=Orange Soda 12 fl oz can, kernel=955058)
    Contents Object = class con.avian.foods.BaseFoods.BasicSeed
      ... BasicSeed(desc=1 unit of soda, type=Orange Soda 12 fl oz can, kernel=963778)
    Contents Object = class con.avian.foods.BaseFoods.BasicSeed
      ... BasicSeed(desc=1 unit of soda, type=Orange Soda 12 fl oz can, kernel=129454)
  History -
```

The saved file shows the details of the full case that were omitted on the TupleTree tab. Here we can clearly see the type of objects in the pod (BasicSeeds) and the contents of each seed (description, type of container, and the random number assigned to the kernel that is used as a unique ID for each unit of soda).

These extra details are available when you save the TupleTree IF you have implemented the method “toDetailString” in your food pods. In this scenario, the CaseOfSodaPod implements the toDetailString method and then all of the other cases (CaseCherryColaPod, etc.) inherit this method. In this example, the toDetailString method shows us details about every seed (unit of soda) in the full pod. These details are omitted from the TupleTree display just so you don’t get buried in details on screen.
Imagine how difficult the TupleTree would be to use if the Object in the seed’s kernel was a sophisticated Object instead of an Integer, it could easily take multiple lines of data to list all the fields in the Object. For example, if the kernel Object was a CustomerAddress object, the Name, Street Address, Street Address2, City, State, zip code fields would probably all need to be listed making an unwieldy on-screen listing. And what if you had hundreds of CustomerAddress objects listed? However, once the TupleTree has been saved to a file, you can open it in a more sophisticated editing environment where you may have access to more advanced search tools and more powerful analytical tools.

**Variation #1: Controlling the Birds**

While the birds are flying, if you click the Stop Me button for the SodaConsumer, the Soda Producer will stop and wait until it either receives the OrangeSodaOkPod or until it runs out of Stamina. If you start the SodaConsumer and then stop the SodaProducer, the SodaConsumer will stop and wait until the OrangeSodaPod is available or it runs out of Stamina. Starting and stopping the birds separately shows that they still have the tightly linked behavior. You can also select the Vitality tabs and change how long they will live (negative for never die) and how long they can go without eating (Stamina – negative for never die). Adjusting these values makes it easier to start and stop the birds and keep the scenario running.

**Variation #2: Adding Another Soda Consumer**

Click the Add New Bird button at the bottom center of the screen. Set up this new bird just like the SodaConsumer but with a different name. It’s also instructive to change the SodaConsumer Nap Lengths to 175ms or 275ms to increase their variability and improve their chances of both catching bottles and putting them in the case. Rerun the scenario and it will work pretty much like before but reviewing the pod history on the TupleTree tab will show that both of the two SodaConsumers are eating the OrangeSodaPods and storing them in the CaseOrangeSodaPod.

**Variation #3: Processing Multiple Simultaneous Soda Types**

You can add other SodaProducers and SodaConsumers for other flavors of soda, or you can load them from files. The file Soda3Flavor.bird has all of the birds and food pods necessary to run three different flavors of soda concurrently. This scenario processes Grape soda, Orange soda, and Cherry cola and each type of soda has a different number of units required to fill a case (Grape = 8, Cherry cola = 12, and Orange = 16).

If you decide to add them manually yourself, be sure to set up the Producers for the other flavors the same way as for OrangeProd1 and to set up the Consumers for the other flavors the same as for OrangeCons1 but having them eat different flavors (Cherry Cola, Grape Soda, etc.).

If you prefer, close ConcX and then restart it and then load Soda3flavor.bird described earlier. When all six birds are loaded, click the Start All button and all of the birds will start
flying normally with their activity ProgressBars growing approximately equally. The ProgressBars for all the sodas will begin flickering as they are eaten and stored in quick succession.

When the run is over, click on the TupleTree tab (click the Show Tree button if it isn’t automatically updated). The number of full cases will vary by type of soda based on the number of units in each case. Reviewing the full cases, you’ll notice that every full case of Grape Soda will have the same number of units as all the other Grape Soda cases and so on for the other drink flavors. You’ll also note that only one bird type digested its drink flavor: regardless of which type of soda, only the desired types of birds processed it.

**Final Variations**

Click the Clear All Activities button and then click the Start All button. Once the birds are all flying, randomly stop and start various birds in the scenario by clicking the Stop Me and Start Me buttons. When their lives have expired, check the TupleTree and all of the Full Cases will have been filled only by the correct type of birds. The birds in these tightly linked scenarios are nicely isolated from the other birds.

If you have the source package installed in NetBeans, you can also add new types of soda that can be produced; for example, Apple Cola, Berry Soda, Danube Cola, Elephant Soda, and so on. Each type of soft drink requires a pod, an Ok pod, as well as a case pod (they all have to be pods). For example: ElephantSodaPod, ElephantSodaOkPod and CaseElephantSodaPod.

**More Realistic Producer - Consumer Wrap Up**

The SodaProducer-SodaConsumer scenario presented in this chapter demonstrated a more complicated and arguably more realistic scenario. It showed how individual pairs of Producers and Consumers can be configured to work together to achieve a specific outcome (full cases of soda). It also showed that multiple pairs of Producers and Consumers can concurrently share the same resource (TupleTree) without blocking or interfering with each other.

Best of all, we didn't have to spend a lot of time worrying about synchronizing or creating/managing critical sections or mutexes or any of the other complications of parallel programming. Instead, we concentrated on what we wanted each individual bird to do and the data chunks (food pods) that they would operate on.

By leveraging the ConcX framework, we were able to build the full complement of a dozen food pods and Producer & Consumer birds in a few hundred lines of code. And when we finished, we had a dynamic model that represented the relationships between the actors and objects needed to build a real system.
Chapter 8. The Dining Philosophers Problem

Description
The Dining Philosophers problem was originally proposed in 1965 by Edsger Dijkstra as a student exam exercise and given its present description by Tony Hoare soon thereafter. Wikipedia describes it like this:

“Five silent philosophers sit at a table around a bowl of spaghetti. A fork is placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when he has both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it's not being used by another philosopher. After he finishes eating, he needs to put down both forks so they become available to others. A philosopher can grab the fork on his right or the one on his left as they become available, but can't start eating before getting both of them.”

The problem was designed to investigate deadlocks and resource starvation and how to prevent them. The straight-forward solution is to instruct each philosopher to behave as follows:

- think until the left fork is available; when it is, pick it up;
- think until the right fork is available; when it is, pick it up;
- when both forks are held, eat for a fixed amount of time;
- then, put the right fork down;
- then, put the left fork down;
- repeat from the beginning.

This solution is said to fail because it allows the system to reach a deadlock state where each philosopher is holding a left fork and waiting for the right fork to become available.

A number of solutions have been proposed over the years. A brief summary of some of the more common proposals follow.

- Overbearing waiter: a “waiter” presides over the dinner and supervises the philosophers and determines who can eat and when
- Spying on neighbors: the philosophers spy on each other and determine when they can pick up the forks without getting deadlocked
• Hierarchical dining: the philosophers have a pre-defined dining hierarchy that sets preferences or priority to some philosophers over others. Think of it as tie-breaker rules.

**Feeding the Dining Philosophers in ConcX**

Let’s take a look at the simplest solution ConcX can offer. Navigate to the phil folder and open the file dining5Basic.bird file. When the birds have all loaded, a screen similar to the following will be displayed. The first bird loaded is Jeeves, the waiter, who puts out the forks and then quits. The philosophers are five BasicBirds (unmodified) who are configured to pick up (eat) two forks (food pods) and to put down (store) two forks (food pods). The only code written for this Dining Philosophers problem is for the waiter to set the table and then to override the digest and store methods with empty methods. Everything else in the Dining Philosophers problem is solved by ConcX out of the box. Just configure them to pick the correct forks and click the Start All button.

The code for the Waiter is simple. It just creates ten new fork objects and adds them to the TupleTree, performing the code below for Fork1Pod thru Fork10Pod.

```java
Fork1Pod f1 = new Fork1Pod();
f1.setEmpty(false);
mEatBeak1.storeItem(f1, this);
//myTupleTree.put(f1.getTreeID(), f1);
```
The flock that was loaded for this scenario only uses Fork1 thru Fork5, but the Waiter puts out all 10 forks anyway so special versions of the Waiter aren’t required for dining5Basic and dining10Basic. The extra forks are ignored in the dining5Basic scenario.

1. Once the dining5Basic.bird file has loaded, click the #1 tab to view the Waiter.
2. Click the Start Me button for the Waiter bird. The Waiter adds the forks as quickly as it can and then stops.
3. Click on #2 tab and then click the Start Me button for Aristotle. His activity progress bar will grow rapidly because he isn't competing for the forks.
4. Click on #3 tab and then click the Start Me button for Bacon (Francis). His activity progress bar will start growing and Aristotle's progress bar will grow less rapidly because he now has to share a fork with Bacon.
5. Start birds 4thru 6 and their progress bars will start advancing more or less evenly.

As the birds' reach their individual Life Length, they will stop eating and the birds around them will eat more frequently because they no longer have to compete for one of their forks. Eventually they all stop.

6. Click on the TupleTree tab. It will look similar to the following screenshot.

In the pod history for Fork5Pod shown, Fork5Pod is eaten initially only by Descartes but after about 3 seconds, he starts to share with Epicurus. Epicurus doesn’t start eating immediately because it took me a few seconds to switch from the tab for Descartes to the tab for Epicurus and click his Start Me button.

Note how long their sharing goes on and how frequently one or the other of these two philosophers will eat twice before the other eats. This is the non-deterministic random timing that is built into ConcX. Embrace the randomness and the chaos; birds in the real world don’t fly in rank and file fixed formations and neither should Avian birds in ConcX.

The pod histories all go on so long that it’s hard to understand what’s going on.

1. Click on the Clear All button and then click on the Start All button.
2. After a few seconds, click on the Stop All button
The following screenshot gives a better feel for what’s going on.

There are ten forks (food pods) in the TupleTree put there by the Waiter before he quit work. The five philosophers all tried to grab Fork1 - Fork5 (completely unaware of Forks6 – Fork10 because of semantic blindness, no doubt) as soon as they started, even if the Waiter hadn’t finished putting them out. But if they didn’t get a fork, they just waited and tried again a little while later. Sometimes they shared evenly, sometimes they shared unevenly; it just depended on their individual “luck” (random nap lengths) at getting forks.

If you click on one of the BasicBirds (tabs #2 - #6) and then click on their History tab, there will be a line repeated throughout the history, “patientEatFromTree, ate on try: x”. When a BasicBird (and any descendent birds) eats two foods with an AND relationship, the BasicBird will perform a “patientEatFromTree”, meaning that after successfully obtaining its first food (e.g., Fork1Pod), it will hold that food pod while it tries to get the second food. If it doesn’t find it within the length of time specified in that bird’s Patience field, the bird will give up and release any food pod it holds.

In other words, the bird must get both food 1 AND food 2 to successfully eat, just like the philosophers must get both forks to eat. But if the philosophers held their first food pod indefinitely while trying to get the second food pod a deadlock would eventually occur. The deadlock would happen when each philosopher got their first fork but couldn’t get their second fork because it was held by another philosopher as their first fork.

So to prevent deadlocks, BasicBirds have a patience setting, measured in milliseconds. It also has a number of attempts setting that is defined programmatically as 10. See Part 2, Patient Eating, for more info). If the bird runs out of patience (either time or attempts) before it finds the second fork, it will put back the first fork and its history will record that it put back Food1. If the bird finds the second food before it runs out of patience, its history will contain a statement such as “patientEatFromTree, ate on try: x” where “x” is the number of tries it made.

BasicBirds are hardwired to look for second foods about 10 times per second. The default Patience setting for BasicBirds is 500ms, so it will look for the second food about 5 times before giving up. Increasing birds’ patience improves the chances of a bird getting the
second fork but also increases the chances that the birds will slow each other down while they try to get their second fork.

If you had specified the bird’s foods had an OR relationship, the bird will first try to get its food 1 and IF that food isn’t available, it will try to get its food 2. It always follows that order and if it always gets food 1, it will never try to get food 2. The OR relationship is provided for situations where the bird has a preference in foods, perhaps situations where the exact food is preferred (food 1) but a more generic food that could be processed after more effort would be eaten if food 1 wasn’t available, similar to someone who wants to eat mashed potatoes but if those are gone, will eat the broccoli. Other ways of looking for food are left as exercises to be implemented by the readers, such as alternating (eating food 1, digesting and storing, and then eating food 2, digesting and storing, repeat) and random (randomly deciding to eat either food 1 or food 2, and then digesting and storing what it found), and so on.

It is interesting to change all of the BasicBirds (philosophers) to eat their Food1 fork OR eat their Food2 fork. Click the Clear All Activities button and then Start All. The birds all fly more smoothly and consistently. Click Stop All and look at the TupleTree tab.

If you guessed that each one of the Forks would be eaten by just one bird philosopher, then you’re right. Because of the way that the birds/philosophers were configured, they never have to share their Food1 fork, which means that they always get their first choice so they never have to look for their second fork and therefore never use that second fork.

Again, clicking on one of the BasicBird tabs (#2 - #6) and then clicking on their History tab will confirm the above. It will show that each bird is eating their first food every time (...eating foodType,philos\Fork5Pod,Epicurus) and never eating their second food (...Eating,philos\Fork1Pod,Nothing to eat,Epicurus).

**Summation of Simple ConcX Solution**

The preceding example shows that this simple ConcX solution works remarkably well for the simple reason that the Dining Philosophers problem happens to fit the basic behavior of birds in Avian Computing exceptionally well. Philosophers in the problem definition get a first fork and if they get that fork then they look for a second fork, and if they get the second fork, they eat and then put both forks down. It is as if the Dining Philosophers problem was written to define birds’ behavior in Avian Computing.

However, ConcX implements a variety of features that automatically prevent resource-sharing problems. Deadlocks are prevented because default ConcX birds never hold any resource without letting go of it; they try to get a food pod from the TupleTree and if it isn’t available, they don’t block and wait for it to become available. Instead, they just take a nap and try again later. And if they are trying for two resources (forks in this case) and they get one, they only hold it until they get the second resource or until they run out of patience. Either way they will always release any resources they were holding.
ConcX uses nap randomness to automatically prevent “livelock” conditions that could potentially happen if all of the philosophers arrive in the dining room at exactly the same moment, grab their left fork at exactly the same moment, hold it for exactly the same length of time while trying to get the second fork (held by the others), and then put it down at the same time and then waiting the same length of time before trying again.

In ConcX, however, this livelock situation effectively can’t happen because every bird naps for a random length of time every time they look for food, even if they don’t find food. This built-in randomization prevents the birds from syncing up and blocking each other, reducing the chance of a livelock to effectively zero. Reviewing the pod histories and the bird histories documents just how asynchronously the birds perform. Sometimes they nap a short time and sometimes they nap close to the maximum length.

**Variations**

A first variation would be to change the Life Length of the philosophers from 30,000ms to 300,000ms (5 minutes). This longer lifetime shows that over time the ProgressBars tend to even out, that in general the philosophers all get to eat about the same amount, that the “luckiness” of some birds is just an initial condition that tends to disappear with longer run times (assuming nap times are all equal).

A second variation would be to experiment with the Nap Length setting. For example, reduce the Nap Length for Aristotle and Camus to 20ms, making them greedier than the other philosophers. The result will be similar to this screenshot. Not only are Aristotle’s and Camus’ progress bars longer, but they have higher Max values (400 for Aristotle and 200 for Camus), showing that they are eating much more frequently.

What happens if more Philosophers get greedy and you reduce all of their Nap Lengths? What happens if one has a long nap length and all the others are very short? Will that one starve?

What happens if you “optimize” the problem and reduce all of their Nap Lengths to 10ms? Instead of higher through-put, the increased competition for forks means that there is a lot more time spent grabbing for forks instead of eating.
By increasing in small increments the Nap Lengths for all the Philosophers, we can search for an actual optimum through-put. In my testing, the Nap Lengths have to increase to 50ms before they start sharing the forks, but even then they are frequently unable to get both forks and run out of patience a lot.

Interestingly, on my system, around 150ms the Philosophers are eating frequently and rarely run out of patience. There are many times when they can't get the first fork so they have nothing to eat, but most of the time when they can get the first fork, they can also the right fork and then eat. And looking at the food histories, it is obvious that they are sharing much better and they are eating more.

Isn't it odd that by making all of the birds equally less “greedy” that they can all eat more often and as a group do better? Less surprising is if a minority of the birds are greedy eaters and the others are slower eaters, the greedy ones can eat significantly more than the slower eaters without interfering too much with the amount or frequency of eating for the slower ones. Are there lessons here in how to run a society, that there is an optimum level of greediness? Or that there is a threshold in the number of greedy individuals that a society can tolerate before they start detracting from the overall performance of the society? See the next chapter for a further investigation of these possibilities and more.

Another variation is to add more Philosophers. Just keep adding BasicBirds and assign them the appropriate forks. Fork6 thru Fork10 are automatically put out by the Waiter. Just be sure to reassign the right fork for Epicurus from Fork1 (Aristotle's left fork) to Fork6, and so on. I find that it helps to draw a circle with the philosophers numbered (and the forks they use) around the circle. You can also just load dining10Basic.bird from the phil folder.

Another variation is to have teams of philosophers competing for the forks. For example, Aristotle and Bacon could both try to get the same forks and block out the other users of those forks. Or three of them could team up and elect one member to be the “designated eater” who gets full access to both forks (for example, Fork2Pod and Fork3Pod) while its teammates compete with the other Philosophers for Fork1Pod and Fork4Pod, reducing the others ability to eat.

**Advanced Variations or Living without the Waiter**

It has always seemed like cheating to use the Waiter to put out the forks. Besides, everyone knows that philosophers always carry their own silverware – one of the occupational hazards of being a philosopher is that they never know when they will find their next meal. Carrying their own silverware is useful for those lean times when their philosophical preferences fall out of favor.

So let's create custom Philosopher birds that set the table for themselves. And let's also change their digestion behavior so it is not dependent upon their nap times. After all, the Wikipedia description said that they eat for a fixed period of time.

It turns out that adding those two abilities is relatively simple. Adding silverware to the table (TupleTree) can be done by overriding the doFirstTimeProcesses method, while
setting the eating time to a fixed length of time can be accomplished by overriding the digestionDelay method so they digest (eat) for a fixed length of time instead of a random time based on Nap Length.

The doFirstTimeProcesses method for Philosopher birds now puts out the forks it will use. For example, Anscombe tries to puts out Fork1Pod and Fork2Pod, while Burke tries to put out Fork2Pod and Fork3Pod. Note that they both try to put out Fork2Pod, but only one Fork2Pod is allowed in the TupleTree. The trick is that Philosophers try to put their forks into the TupleTree as a Singleton, meaning that only one instance of each fork (ForkxPod) is allowed in the TupleTree. The forks are interchangeable; it doesn’t matter who actually puts them into the tree as long as there is only one of each kind of fork in the TupleTree so the Philosophers are forced to share.

Overriding the digestionDelay method allows the Philosopher birds to eat for a fixed length of time. This can be done by simply inserting a Thread.sleep(100) in the new digestionDelay method, but this solution is excessively rigid. Every Philosopher bird will now eat for exactly the same length of time and cannot be changed without modifying the code. A better solution is to set the digestionDelay to default to a standard length of time, and then use one of the XMV values (Externally Managed Variables) to override the default digestionDelay (eating time).

Load the dining5Phil.bird file; five Philosopher birds will be loaded. Click the Start All button and they will run slowly and jerkily because they are all using the default digestionDelay of 1000ms. If you check the XMV tab, the first field, DigestionDelay, is blank, meaning that the default value remains set to 1000ms. The Philosophers all run more smoothly when their digestionDelay is set to shorter values.

If you check the TupleTree after their run, you'll see that an image similar to the accompanying screenshot. The forks will show that they were shared fairly well. On the right, both Emerson and Diogenes eat approximately the same number of times, sometimes a few times in a row but always sharing with the other Philosopher.

Sometimes as they run, one or more Philosophers may run out of Stamina and stop running altogether. When a Philosopher runs out of Stamina and stops, it was because that Philosopher had random nap times that correlated unfortunately with the random nap times of its neighbors.
If their random nap lengths always had them awakening when their neighboring Philosophers were eating, they were “unlucky” and couldn’t eat. When they ran out of stamina they ended up face down on the tablecloth.

Note that the number of items in the TupleTree is 10, twice the number of forks. That is because when the forks are added, a “lock” is also added that prevents a later Philosopher from mistaking a fork that is in use for a missing fork.

Checking other ForkxPods in the tree however, may show a very different story. In the following screenshot, the Fork5Pod is initially shared by both Diogenes and Emmerson but after a few seconds, Diogenes’ luck runs out on him and he is unable to get control of this fork and so he never eats again (in this run). He’ll be luckier next time.

If you check the bird history of one of the Philosophers that ran out of stamina, you can watch its Stamina shrink the longer it goes without eating. In its history will be lines similar to:

```
Stamina = 3052
......
Stamina = 2955
......
Stamina = 2503
```

In this scenario, all of the birds were saved with their Level of Info set to “Fine” which provided this additional info about their stamina and other settings, such as its age, its napping duration, and so on.

For this scenario we could adjust the Stamina setting of all the Philosophers until they are sufficiently resilient that they can outlast the length of time that their neighboring Philosophers eat. Or their Stamina could be set to a negative value and then they’d never run out of Stamina.

A different and arguably better solution is to adjust the DigestionDelay value on the XMV tab for each bird. Click on the XMV tab of one of the Philosophers and the left field on the first line will contain the text “DigestionDelay”. You can change this text of this label to “Eat Time” or “Time with 2 Forks” or whatever will help you remember what this line represents. Then fill in an appropriate number of milliseconds in the field to the right. In this example, 300 (ms) was entered, but any positive value can be entered here. The value entered here will override the default DigestionDelay.
By adjusting the XMV DigestionDelay value on all of the Philosophers, you can affect the overall system performance. By reducing the XMV DigestionDelay on all of the Philosophers, the system will run more smoothly and evenly. By increasing the XMV DigestionDelay on all the Philosophers will run more erratically with more of the Philosophers being starved out. The XMV DigestionDelay provides a convenient way to experiment with system performance without coding. It also allows you to explore how various settings of one Philosopher affect the performance of others.

If you find a set of parameters that you like, be sure to save it. If you save it to the original filename, it will overwrite the original values without checking to see if that is really what you want. If you’d like to preserve the original file settings, just give it a different name or save it to a different folder.

**Summation of Advanced Variations**

This last section looked at how the BasicBirds could be extended to provide advanced features, such as each (Philosopher) bird loading the resources (forks) that it required before it started running. The downside of this feature is that it requires the forks be stored as Singletons, requiring you to override the strFoodsWithBeaks method.

The Philosopher bird also provides the ability to fine-tune the performance of the system using an XMV value to override the default settings. The XMV values allow you to make adjustments to settings or to provide additional values to your birds that are saved along with your birds.

If you’d more about how the Philosopher bird is coded, you can either open the code in NetBeans or you can turn to Part 2 that (eventually) contains a discussion of the code for the Philosopher bird.
Chapter 9. Estimating Greed Scenario

Discussion

The Estimating Greed scenario flows directly from the experiments with the Dining Philosophers scenario. In that scenario, the Philosophers all competed for forks with their neighbors; always with the expectation that eventually every Philosopher would be fed. You might have experimented with setting up teams and alliances, but it was all optional.

The Estimating Greed scenario is a more deliberate investigation of how an entity’s willingness to put itself first affects themselves, how it affects other entities, and how it affects the overall system. Using ConcX, it is possible to precisely adjust specific elements of “sharing” and “greed” and then measure the resulting outcomes. This scenario is of course an abstraction and in no way pretends to represent the real world; however this scenario can provide insights into the consequences of selfish behavior. ConcX provides a tool where you can methodically adjust “selfishness” settings for one or more entities and then recording the results of each entity and on the society of entities in this scenario.

Implementation

This scenario investigates sharing of resources, just like the Dining Philosophers scenario in the previous chapter but it extends that scenario in several ways.

- It changes the scenario from the whimsical situation of philosophers all dining together to a situation where 10 farmers with adjacent lands all have to share the water supply to irrigate their individual crops.
- Each farmer has to get control of the two faucets shared with the neighboring farmers to get the water to flow. Note how Cate in the illustration shares Faucet 3 with her neighbor Bill and shares Faucet 4 with her neighbor Dave.
- Every farmer records every attempt that they make to get control of the faucets. The successes and failures are both tracked.
- Additionally, every time a farmer turns on the water, the amount of water delivered to the
farmer is recorded. The amount of water delivered per unit of time is assumed to always be the same constant rate for all farmers.

- It is assumed that every farmer will consume as much water as they can get. Rumor has it that they all have additional farm lands that will receive any excess water that they receive.
- Two settings control the “greediness” of each individual farmer: how often they try to turn on their faucets and how long they keep the water running.
- At the conclusion of each run (configured to be 30 seconds for all farmers but you can change this), each farmer reports their water usage and the number of times they tried to get water and the actual number of times they successfully got water.
- The individual farmers’ amounts are summed and grand total amounts are reported as well as the largest amount received by one farmer and the least amount of water received by one farmer.
- To prevent unusually lucky or unlucky farmers from inappropriately affecting the reported results, multiple baseline runs can be made with all farmers set up identically and their results kept.
- Subsequent sets of runs were made with one or more farmers configured to have increasing levels of greediness. The results of those runs can then be compared to the baseline results.

**Results Summary**

While this scenario is not a realistic representation of the real world, it does provide a controlled environment that can be methodically adjusted to represent many different “social” conditions. ConcX provides a laboratory in which to experiment and measure results, allowing observations about the results to be made. For example, after running many different setups, the following observations have been made:

- A farmer who is a little bit greedy only has a minor negative effect on the immediate neighbors. Surprisingly, in this controlled environment, the amount of water used in the whole system goes up slightly, indicating that the greedy farmer received more water than the unfortunate neighbors lost.
- A significantly greedy farmer has major negative impacts on the immediate neighbors, measurably decreasing their ability to get water. This in turn has a ripple effect that benefits the neighbors of the neighbors; the “secondary neighbors” are also able get water more often because of the reduced ability to get water by the immediate neighbors of the greedy farmer. And again, the total amount of water used by all of the farmers goes up, even though a few are suffering.

One may speculate that the “beneficial” ripple effect could be at the root at US society’s ambivalent attitude towards self-interested behavior that improves one
individual at the cost of other(s). The attitude is sort of like, “Too bad about those guys who lost their jobs/businesses/lives. But I’m doing a little better, even though I didn’t do anything differently, so it’s not all bad.”

- The greatest amount of water is received by farmers when the greedy ones are arranged so they don’t share faucets. In this configuration, non-greedy farmers are surrounded by and effectively wiped out by their greedy neighbors, giving the greedy farmers unimpeded access to as much water as they can use.

- Significantly less water is received by farmers if the greedy farmers are configured to be competing with each other. In this configuration, the greediness of one farmer is effectively negated by the other greedy farmers.

- If all the farmers are configured to be very greedy it creates a situation where the smallest amount of water is delivered to the system. The farmers are all working so hard to get water that they actively prevent the other farmers’ from getting water.

This result indicates that there may well be limits to competitiveness and aggressive self-interested behaviors. As a group of competitors improves because of increased competition, the amount of advantage gained by each increment of improvement effort decreases. Increased levels of competition, instead of continually increasing the total amount of water being used, it actually reduced it.

- Based on the observations noted above, the much heralded “competitive advantage” of one company or country is only useful if most of the other competitors do not adopt the competitive advantages. If one company increases efficiency by streamlining, it is only a competitive advantage if the other competitors do not also streamline. Implementing a competitive advantage may ultimately backfire on a company because it may cause the other competitors to implement an improved form of competitive advantages that outperforms their original advantage.

**Setup and Procedure**

The easiest way to setup the Measuring Greed scenario is to click the File menu and select the Open Flock File option. Navigate to the sharing folder and open Farmer10.bird.

Eleven birds will get loaded: ten Farmers and one Summarizer. The farmers try to get water and the Summarizer eats (summarizes) the individual farmer results.

Click the Start All button and the Farmer birds will start flying normally. After 30 seconds, the Farmer birds will stop (end of life time) and the Summarizer bird will start eating ResultsPods, summing the results of individual Farmers and storing the grand totals in the SummaryPod.

**Viewing and Interpreting Results**

Scrolling thru the TupleTree screen, the Summary pod will look something like this:
The SummaryPod has the total amounts listed, as promised, but it’s short on details. If you want more details about how those amounts were calculated, you need to look at seeds contained in the one and only seed in the SummaryPod. The best way to do that is to enter a file name in the field at the bottom of the TupleTree tab and then click the Save button.

Open that file with a text editor and scroll down until you find the SummaryPod. It will look similar to the following (without the word wrap included below):

```
Key = SummaryPod
   .. Contents Desc: SummaryDetails{Total Failed=687, Total Succeeded=305, Total DelivAmt=91500, Total DiedEarly=0, Max DelivAmt=10800, Min DelivAmt=7500}
   .. Seeds in Pod = 1
   .. History =
   12:50:29:724,SummaryPod,Added during Summation phase
```
The two highlighted sections are formatted so they can become part of a CSV file that can be imported into a spreadsheet program. Once the CSV data is in a spreadsheet, it can be copy-pasted into a master spreadsheet where the data can be used to generate charts.

For example, here is the beginning of one of the CSV files. The first two lines in the file are the first two highlighted lines above. On subsequent runs, only the second line highlighted above is added to the beginning of the CSV file.

```
Total Failed, Total Succeeded, Total DelivAmt, Total DiedEarly, Max DelivAmt, Min DelivAmt
720,313,93900,0,11100,8400
721,306,91800,0,11100,6900
652,305,91500,0,10500,7800
```

The second group of highlighted values contains the individual values for each farmer in one individual run. These values, including the header line, are copied directly into the CSV file.

It is important to run each configuration at least three times because the random feature built into ConcX may produce uncharacteristic results in any single run. By running each configuration at least three times you have pretty good confidence that the results that you received actually are typical for the greed settings that were used.

This CSV file was opened by a spreadsheet program which put the values in the CSV file into neat columns in the new spreadsheet. Once in the new spreadsheet, their rows and columns could be pasted into a master spreadsheet, as shown below. I like to use a separate tab for each set of runs for each configuration.

Once the values have been copy-pasted into the master spreadsheet, you can generate charts from the results. I find it more convenient to use a master spreadsheet to hold all of the results instead of using multiple spreadsheets, but the choice is yours; use what works best for you.
The screenshot above shows the baseline results where all of the farmers are configured the same. Note how consistent the results are; there are individual differences but these are all caused by “luck” from built in randomness. A farmer that does well in one run may do poorly on the next run.

In this next screenshot, Edna has been configured to be greedy by trying to turn on the faucets more frequently than her neighbors Dave and Fred.
In this configuration, we can see that Edna is receiving significantly more water than any other farmer and has effectively prevented her neighbors (Dave and Fred) from getting any water. But the Total Amt Delivered chart shows that more water is being delivered overall than when every farmer got the same chance to get water.

If the greedy farmers are set to not interfere with each other (think oligopoly) the greedy farmers do exceptionally well. In this scenario, the non-greedy farmers are quickly eliminated from the competition for water so they die and no longer put up even token competition. From an external perspective, this is a thriving society. Water usage is way up which should translate into more food being grown. Everything is great, as long as your name isn’t Bill, Dave, Fred, Hank, or Jack. They did nothing wrong other than picking the wrong neighbors (competitors).

**Inserting Greed into the Simulation**

The first step to estimating greed is to select a farmer to be the greedy farmer. It can be any one of the farmers, which is one of the strengths of this simulation in ConcX: there is not a pre-ordained choice who behaves differently. The same results are achieved by any of the farmers (within the limits of nap randomness) because of their configuration and NOT because of the programmer’s preferences.

The farmer selected to be the greedy one needs to have either their nap time shortened or their watering duration increased. To shorten their nap time, click on their Vitality tab and reduce the Nap Length value (5ms or less causes an exception because it violates the minimum setting). To increase their watering duration, click on the XMV tab and change...
the numeric value on the line labeled Digestion Delay. Digestion Delay is the method that is called during the digestion portion of the bird’s lifecycle; the default behavior inserts a delay based on a random number of milliseconds up to a maximum of the Nap Length value. The value on the XMV tab is a fixed length delay – the farmer will always digest (use the water) for this length of time. If you clear out the value on this line, the default duration of 1 second (1000 milliseconds) will be used.

After changing and adjusting farmer values as desired, feel free to explore other possibilities using the Farmer20.bird and Farmer40.bird files. These files increase the number of farmers competing for water without increasing the number of faucets – there are still only 10 faucets. Be patient while these files load – it takes a while to get all those birds set up on their own tabs. This delay seems to be an issue with Windows because this delay loading birds is not seen when running Linux.

The results of these additional configurations are similar to the results found using the Farmer10.bird file. The main difference is how the greed of one farmer affects so many other farmers. The span or scope of one farmer’s greed is amplified in the Farmer20 and Farmer40 configurations.

**Summary**

One of the interesting aspects of this scenario is that it allows us to see both the winners and the losers in a “society” and to quantify how the actions of some can affect the lives of others. In a society that celebrates winners, the losers are easily forgotten and their businesses/lives swept up and disposed of in the dustbin of history.

Identifying the losses that accompany successes is an important part of the calculus of sustainability. If we can’t identify the tradeoffs and estimate the losses that accompany a success, we can’t accurately predict how a course of action will affect our society. And if we can’t look beyond the impassioned rhetoric and truthfully estimate the effects of changes then our mutual chances of surviving in the future are limited.

This scenario is an attempt to study one of the foundational concepts of Capitalism: that individuals acting in their own self-interest will produce the best overall outcome for society at large. From Adam Smith onwards, economists have taken this concept as a pillar of their faith but never really empirically proved that the best overall outcome for the most people was achieved. The abundant presence of lots of shiny new cars owned by lots of people is taken to be sufficient evidence that Capitalism works and therefore Capitalism should never be limited.

While it may be argued that an abundance of material goods does not automatically lead to individual happiness or improved lives, this argument tends to focus on moral, religious, or philosophical reasons. Using a tool such a ConcX to study the potential impacts of economic changes could provide us with improved awareness of the possible future impacts of those changes. This improved awareness of the potential impacts can hopefully lead us to making better decisions for all members of society.
One certainly can’t fault economists for this lack of study because the only realistic way to study the effects of selfishness would be to isolate individual cities and then to force the residents of each city to adopt various levels of selfishness. Not an easy study to perform in free societies.

It is also worth noting that, in the past, 90% of society consisted of peasants trying to coax enough food to grow from the land owned by others. Economics was founded on the perception that a few pennies difference in the price of a loaf of bread determined if large portions of society went hungry that night. Those 18th Century perceptions no longer accurately describe the lives of billions of people but we continue to base our economic foundations on those outdated realities. New tools are needed for new realities.

This scenario is not an attempt to invalidate Capitalism or attempt to prove that an alternative economic system is superior. Instead, this scenario is an attempt to quantify the effects of self-serving behaviors and their impacts on others in society so that we can make informed decisions on the trade-offs between personal profit and the well-being of their society.

It is hoped that this scenario can in some way be useful in initiating or sustaining sincere discussions about the role of an individual in society, about the rights of the individual versus the rights of society, and the appropriateness of rules and limitations being forced on individuals by society. It is hoped that by shining a bright light on the effects of competition that we can have honest and beneficial discussions of how much economic freedom should an individual have.
Chapter 10. Estimating Pi in Parallel

The value of pi can be calculated using the following infinite series:

\[ 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots = \frac{\pi}{4} \]

This method of calculating pi was first discovered by Madhava of Sangamagrama in the 14th century. It was then rediscovered in the 17th century by Leibniz and by James Gregory who published two works that included it in 1668. This series of calculations is known variously as the Leibniz series or the Gregory-Leibniz series or the Madhava-Leibniz series and probably also the Madhava-Gregory-Leibniz series.

This method of calculating pi lends itself to computerization because it can be put into a loop and run for as many terms as desired. This is fortunate because, as Wikipedia states, the results converge slowly. Very slowly. In fact, it takes about 5 million terms to get the first 6 digits correct and more than 5 billion terms for it to get to the first 10 digits of pi correct. And as you’d expect, the more correct pi digits that you want, the longer it takes to calculate them.

To get started, open the pi-4.bird file. This will add a Distributor bird, a Reducer bird, and 4 Calculator birds.

When the app first starts, the Distributor bird breaks the overall range (the number of terms to process) into subranges (bite-size pieces) that are stored in the TupleTree as RangePods and then terminates. The default overall range is 10,000 terms and the default number of subranges is 100. These values are hard-coded in PiSeedPod but you can change their values by entering new values into the first two XMV Value fields of the DistributorBird. This means that you don’t have to modify/save/re-compile PiSeedPod every time you want to experiment with a different number of terms or subranges. It’s kind of fun. Enter a new maximum number of terms in the top field, for example, 5000000 (5 million) and then press the Start Me button. Click on the TupleTree tab and all 100 RangePods will be listed, each with a range of 50,000. If you experiment with these values, don’t use any punctuation in the numbers, such as commas, or the default value will be used.

Each Calculator bird eats any RangePods it can find. It gets the minimum and maximum values of their subrange and then loops thru that range of numbers and calculating its subtotal using the formula above. The subtotal result is inserted into a ResultPod and stored in the TupleTree.

The Reducer bird eats a ResultPod and the one and only ReducePod. The Reducer bird takes the value out of the ResultPod and adds it to the running total held in the ReducePod and then to the ReducePod back into the tree.

Note that the pi-x.bird files contain the number of Calculator birds indicated in the file name; the pi-2.bird file has two Calculator birds and the pi-16.bird file contains 16 Calculator birds. The smallest pi-x.bird file has just one Calculator bird while the largest pi-x.bird file has 50 Calculator birds.
As usual, click the Start All button. The birds will all start showing activity with Calculator birds active for about 1 second. The Reducer bird will run longer (about 4 seconds) because it can’t eat results as fast as they are being calculated.

Click on the TupleTree tab and there should be just one item; the ReducePod item. Its Contents Desc field should contain 3.141692643590543213460768320877938 or similar depending on how many terms you used. Its history should be 100 lines that list when and how much was added to the Reduce food item. Note that it may list some of the values using scientific notation instead of a bunch of leading zeros.

If you select one of the Calculator birds and then select its History tab, you can view when it ate a RangePod and when it stored the ResultPod. If you change the Level of Info to Fine and rerun the app, it will also include when it started and finished calculating the sum total value for that subrange.

The code that performs the calculations is quite short.

```java
addToBirdHistory("Start Computing " + startNumber + " thru " + endNumber, Level.FINE);
for(long i = startNumber; i <= endNumber; i++) {
    //BigDecimal denom = new BigDecimal(i * 2.0 + 1.0, new MathContext(900));
    BigDecimal denom = new BigDecimal(i * 2.0 + 1.0, MathContext.DECIMAL128);
    if (i % 2 == 1) {
        //denom = denom.negate( new MathContext(900));
        denom = denom.negate(MathContext.DECIMAL128);
    }
    //BigDecimal one = new BigDecimal(1, new MathContext(900));
    BigDecimal one = new BigDecimal(1,MathContext.DECIMAL128);
    //mySum = mySum.add(one.divide(denom,new MathContext(900)),new MathContext(900));
    mySum = mySum.add(one.divide(denom,MathContext.DECIMAL128),MathContext.DECIMAL128);
}
addToBirdHistory("End Computing " + startNumber + " thru " + endNumber);
```

Note that all the calculations are performed using BigDecimals because doubles lack sufficient number of digits for these calculations. The commented out lines above show how to increase the precision (number of digits to the right of the decimal point) for the calculations from the default of DECIMAL128 to 900 digits. Other MathContext values (ranging from DECIMAL32 to over 1000 digits) can be selected but it turns out that in this scenario increasing the precision does NOT noticeably improve the accuracy of the results, although it does significantly increase the length of time it takes to perform the calculations.

Here are the results for 5 million terms in DECIMAL128 and 900 digits followed by the actual value of pi (from table of pi values):

3.14159245389873238432643393679541 (DECIMAL128 )
3.14159245389873238432643393679499... (900 digits)
3.141592653589793238462643383279502... (actual value of pi)

The bold + underlined digits above show where there are differences. The first difference from the actual value of pi is in the 7th digit but then the next 5 values are correct, followed by 2 more off digits, followed by 5 correct digits, followed by 1 wrong digit, etc. One of the odd things about this method of calculating pi is that it provides answers that are mostly
correct, percentage wise at least. In the example above, it has correct digits about 70% of the time, with only 7 wrong digits out of 33. And the difference between DECIMAL128 and 900 digits doesn't show up until 31 digits after the decimal point.

Clearly, increasing precision won't improve the accuracy of the results, at least not out to the first 30 places or so. And since the calculations for 900 digits of precision run 100 times slower, increasing precision won't help you more quickly get the true value of pi.

Increasing the number of terms does improve the accuracy. Going out to 5 billion terms (0-5,000,000,000) increases the number of correct digits to 9 (3.141592653) and this is when the value of using multiple Calculator birds becomes apparent. Using 2 Calculator birds, it takes 26 minutes to process 5 billion terms. Using 4 Calculator birds, it takes 15 minutes. Using 8 birds drops the time to 12 minutes, while using 16 birds reduces the time to 11:30 (no real improvement over using 8 birds). Processing 5 billion terms is also when the number of cores available becomes critical – my 4-core (8 hyperthreads) CPU simply can’t go any faster. But if I had one of the fancy new 10-core (20 hyperthreads) CPU that Intel just announced, we could probably show solid improvements all the way to 20+ threads.

Interestingly, if we stick with the default number of terms (10,000), we can show improvements using 32 birds and even 50 birds. In this configuration, 1 bird took about 4 seconds, 2 birds took 2 seconds, 4 birds about 1 second, 8 birds took about 0.5 second, 16 birds about 0.2 second, 32 birds about 0.07 second and 50 birds about 0.04 second. This almost straight-line improvement seems to be because the duration of each thread was relatively short, allowing the threads to efficiently process their assigned tasks. In the previous configuration (5 billion terms), the duration of each thread was so great that adding more threads didn’t help.

In general, though, note how useful it is to be able to easily add more birds to a task. No attempts have been made to optimize the efficiency of the Calculator birds but by just adding more birds, we were able to reduce the runtime by more than 50%. In more powerful systems (16 cores or more), the time could have been reduced even more, all the way down the theoretical minimum, Amdahl's Law, that is, the length of time that it would take one single core (thread) to compute its one and only range.

**Variations**

The first variation that you’ll probably want to make is to change the number of terms that will be processed. You can change the value in the PiSeedPod code but it’s easier to change the settings on the XMV tab for the Distributor bird.

Load one of the Pi-x.bird files, select the Distributor tab and then select the XMV tab. The first line (xmv1) will be labeled Max Nbr of Terms and the second line...
will be labeled Nbr of Ranges. If the value fields are blank, then the default settings are used (10,000 terms and 100 ranges).

If you happened to load the Pi-8.bird file, you’ll see that it’s been pre-configured to process 100,000 terms in 1,000 ranges as shown in the screenshot. You can change these values if you want, just be sure to save the file if you want to keep these new settings.

Note the format for the number of terms: you can change the zero to the left of the hyphen to get a different starting value, but the final value that it calculates won’t be anywhere close to the value of pi. Turns out that the very first term that is processed adds the greatest amount to the grand total (it adds “0.7828982….” while the next value added is “0.0012499….” and all subsequent values that are added are progressively smaller).

This is why the starting value to the left of the hyphen is optional; if you skip the first number and the hyphen, the Distributor bird will assume that you want to start at 0. Starting at any other value than zero produces a nonsensical (but not irrational) result.

You can change the Max Nbr of Terms to any value you want (or are willing to wait for). If you go past 50 million, you'll probably need to increase the Life Length of the Calculator birds or they’ll die before they finish processing all the RangePods.

You’ll definitely want to experiment with the number of ranges. Increasing the number of ranges makes each subrange smaller, making it quicker to calculate its subtotal but increasing the amount of processing done by the Reducer bird and the amount of time spent shuffling items in and out of the TupleTree. Reducing the number of ranges allows the Java HotSpot compiler to optimize the calculations performed on each subrange to produce some amazingly fast calculations – up to a point. If the subranges get too large, the ability to produce superfast results seems to be overwhelmed by the large size of the subrange. Plus, the ProgressBars are updated less frequently so it looks like the program has frozen.

Another option to experiment with is the Nap Length of the Calculator birds. If the number of ranges requires the Calculator birds to eat multiple ranges, then the built in delay (Nap Length) can affect the overall performance of the system. However, if you are running more threads than the number of cores in your machine, reducing the Nap Lengths may cause the Calculator birds to interrupt each other more often, reducing performance. Just remember that the minimum time you can enter is 6ms; if you enter a time shorter than that, an IllegalArgumentException will be thrown and everything will stop. If you really want less than 6ms, you’ll need to find every place where ThreadLocalRandom is called and change the minimum time there.

Another variation that is possible in the NetBeans installation is to comment out the System.out.println statements. Println statements are rumored to be slow; their impact on the overall through-put can be determined by running the same number of terms in the same number of subranges and then comparing the length of time between the first result is processed and the last result is processed.

Another coding variation would be to move the calculation of the final value of pi from the afterDigestion method to the finalCleanup method that overrides this method in
As currently coded, the running total is kept as a BigDecimal which is multiplied by 4 to get the present value of pi. This value is then converted into a String that is held in the Contents Desc of the ReducePod. This multiplication and then String conversion is performed once for each interim result or a default 10,000 number of times and possibly millions of times. Moving this calculation so it is only performed once could yield large improvements, however, it would mean that you couldn’t click Stop All button as see what the currently calculated pi value was because it would only be calculated after all the threads have stopped.

And of course, if you want to mess around with the precision, in the NetBeans installation, go into the Calculator bird and the Reducer bird and uncomment out the lines that set the precision to 900 digits and comment out the lines that refer to DECIMAL128. You can change the 900 to any integer value less than 1024 or so, which gives a wide range of performance opportunities. Be sure to record the value of pi and duration at each difference precision so you’ll have a solid baseline for comparison.

Eventually, I will get around to installing a video card with a large number of Cuda graphic processing units to see if we can move these calculations into 100 or so GPU cores without significant re-coding. Because GPU cores are numerical calculating heavyweights, it is hoped that the large number of cores would yield improvements that are 100 or 1000 times faster.

**Summary**

Estimating pi in parallel is probably not the most efficient method of calculating the digits of pi. It seems that the number of terms that must be processed increases by one order of magnitude (10,000 - 100,000 - 1,000,000, etc) for each digit of precision (correct digit to the right of the decimal point) that you want to calculate for pi, making this method impractical for determining the first 1000 digits of pi.

However, this method does demonstrate how parallel processing can be used to perform extremely large calculations and why graphical processing units are so successful in accelerating calculations that can be performed in parallel.
Chapter 11. The BarberShop Scenario

Discussion
The BarberShop scenario is a simulation of a real world barber shop. In the real world, customers go into a barbershop to get haircuts. If there is no one waiting, the customer gets their haircut right away. If a customer is currently getting a haircut, every customer who walks in adds their name to the waiting list. When the barber finishes cutting the current customer, the customer pays the store manager and leaves. The barber calls the name of the next customer waiting (the one whose name is at the top of the list because they have been waiting the longest) and gives the customer the haircut that they desire. If a customer has to wait too long (defined by the individual customer), the customer leaves the barber shop.

The BarberShop scenario uses three bird types: the Barber, the Customer, and the ShopManager. Customers “come into the shop” and put their names onto the waiting list (barber backlog) and indicate the kind of haircut they want. One or more Barbers look at the waiting list and find the Customer who has been waiting the longest and give that Customer their requested haircut. Customers who have been waiting too long give up and leave the shop. The ShopManager keeps track of all the transactions and produces multiple reports.

When the Start All button is pressed, all of the birds start flying. As each Customer starts up (runs their individual doFirstTimeProcesses methods) they each create a pod with their name and the kind of haircut they want, as well as adding their name to the waiting list. The Customer then begins waiting for their name to be called.

Waiting Customers check their status regularly to see when their haircut begins. If it takes too long for their name to be called, the Customer “crosses their name off” and then leave the barber shop. The patience of each Customer can be individually configured.

Each time a Barber finishes a haircut, he marks the Customer as completed and then finds the next Customer by finding which one has been waiting the longest. That Customer’s name is then called (searched for in the TupleTree); if the Customer is found, the Barber begins giving them a haircut. If the Customer isn’t found after four tries, the Barber crosses that Customer off the list and then finds the next Customer.

As the ShopManager starts up (runs doFirstTimeProcesses method), it creates a DailyTransactionsPod that will accumulate all of the results as the simulation runs. Only one DailyTransactionsPod is created regardless of how many ShopManagers are in the shop that day; making the DailyTransactionsPod a singleton prevents the results from being spread across multiple pods which would make the reports harder to understand (how many total customers gave up on that day, etc).

Setup
The BarberShop scenario requires at least one Barber, one Customer, and one ShopManager to run. Normally there are more Customers than Barbers or ShopManagers, such as 5-10 Customers for each Barber and 3-5 Barbers for each ShopManager.
Two pre-built flocks are provided: barbershop5cust.bird and barbershop20cust.bird. These flocks differ in the number of Customers (5 and 20 customers) but both have just one Barber and one ShopManager. Additional Customers, Barbers, and ShopManagers can be added to either flock and the scenario re-run to see how the results of the scenario are changed.

The above screenshot shows ConcX after barbershop5cust.bird was loaded. Note that the ShopManager eats DailyTransactionsPod and stores DailyTransactionsPod. Eating and storing the same food accomplishes two things:

- the ShopManager tries to capture the latest shop transactions every time it gets the DailyTransactionsPod so the data in the pod are always current
- multiple ShopManagers can run without interfering or coordinating with each other. If one ShopManager held the day’s transactions in memory, it would be very difficult to run a second ShopManager.

All ShopManagers would have to either pass information between themselves and keep track of which ShopManager held the most up-to-date results, or they would have to establish custom locking rules to safely control their access shared memory locations.

By having the ShopManager eat and store the same food, a simple foolproof locking mechanism is implemented that allows painless sharing of info between multiple threads.
Customers are also setup to eat and store CustInfoFindByNamePods. As their names indicate, their TreeID is actually their name, such as “Alvin” and “Dave”. Using their Bird Name prevents us from having to create a FoodPod for each unique name but creates a new problem – unwanted duplicate keys if two Customers have the same name. Duplicate keys are prevented by adding a unique suffix to each customer’s name (Bird Name).

Customers specify the haircut they want by specifying the duration for their haircut on their XMV screen. If no duration is specified, a default value of 1500ms will be used. The following screenshot shows that Edgar (Bird tab #6) has requested a 2500ms haircut which will cost him $25.00. No attempt has been made to accommodate the myriad of all possible haircuts by name because this level of “authenticity” is beyond the scope of this scenario but is left as an exercise for interested programmers.

Both Barbers and Customers access CustInfoFindByNamePods and pass status information in these pods. These pods notify Customers when their haircuts have started and finished and notify Barbers that the Customer gave up waiting.

Barbers officially read BarberBacklogPods and officially store NullPods. The Barbers grab all of the BarberBacklogPods all at once and sort them to identify the Customer who has waited longest. The other BarberBacklogPods are put back into the TupleTree. The Name of the selected (longest waiting) Customer is then used to retrieve the appropriate CustInfoFindByNamePod. Retrieving and storing bunches of BarberBacklogPods are prevented from clogging up the system bandwidth by using the getMultiItem and
storeMultiItem methods of their beaks which pass the BarberBacklogPods as ArrayLists. These methods also depend on synchronized methods inside the TupleTree itself to move individual BarberBacklogPods into ArrayLists and to move the BarberBacklogPods out of ArrayLists and back into individual BarberBacklogPods. Putting these methods inside the TupleTree and synchronizing them prevents other Barbers from getting an incomplete waiting list because some of the BarberBacklogPods are in process.

Because the Barbers have already stored all the BarberBacklogPods, it has nothing to store during the storeFood method. NullPods are “stored” which skips the storage process.

**Results**

Press the Start All button to start the BarberShop scenario. It will run pretty much like any other scenario. See the next image.

One aspect of the BarberShop scenario to be aware of is that ConcX always starts birds in ascending tab number order, which in this example means that Alvin will always start before Bert, who will always start before Chase, etc. Although execution will always begin in this pre-determined sequence, it isn’t expected to materially affect the results of the run. The Customers that launch later will have to wait longer and will be more likely to give up. In this scenario the pre-determined start sequence makes the results more predictable, which is useful when trying different modifications to the scenario. If one really wanted a less
predictable start up sequence, the Customer’s doFirstTimeProcesses could be modified to include a short random-duration nap which would randomly change the order that the Customers added their names to the waiting list. That is left as an exercise for the reader.

The two screenshots above show the appearance of ConcX after the BarberShop scenario is completed. The first screenshot shows the appearance of the Activity progress bars after the run. The lower screenshot shows the typical final results in the TupleTree with it containing just two pods. The LOCK pod is used to ensure that only one instance of the DailyTransactionsPod exists while the second pod is the DailyTransactionsPod itself. That pod contains the “general ledger report” generated by the ShopManager(s) in its descr(ription) field. That report contains four untitled sub-reports:

- Customers with Completed Haircuts
- Customers who Gave Up Waiting (and when)
- Barber(s) and the Haircuts They Gave
- Barber(s) who Called Customer’s that Did Not Respond (they gave up)

More information about the DailyTransactionsPod can be viewed by entering a filename in the field at the bottom of the TupleTree screen and clicking the Save button. Saving the file will show details similar to the following screenshot.

If you are interested in learning more about the BarberShop program details, go to Part 2, Code Details, and find the BarberShop Programming Details section.
This shows how information is stored in seeds in the DailyTransactionsPod. The first four seeds [0-3] in the array contain the formatted results used in the report with all following seeds [4-13] containing the raw information about each transaction in the order that the transactions were completed.

Close inspection of the report shows that Customer Dave564 gave up at 05:784 but the Barber didn’t call the Customer’s name until 07:104, more than a full second later. This information could be used to decide how to modify the scenario in future runs.

**Variations**

There are three different ways a real business could respond to the results of the 5-Customer scenario that showed a customer walked out:

- Do nothing. Losing one impatient customer is cheaper than having a second barber sit around and do nothing sometimes.
- Find a way to make the customers willing to wait longer, such as playing music or a TV in the waiting room
- Add a second barber

The last option is demonstrated in the screenshot below.

The second Barber, York, is setup the same way as the first Barber, Zeb. They both eat BarberBacklogPods and store NullPods (nothing is stored in their storeFood methods). Now when the BarberShop scenario is run, all 5 of the customers received their desired haircuts and none of them gave up waiting. Here’s a cropped image of the results.
Adding a second Barber isn’t the only choice. It is consistently the Customer Dave who gives up. By switching to Dave’s Vitality tab, Dave’s Patience can be increased. In the screenshot below, Dave’s Patience has been increased from 5000 ms to 8000 ms.

To really test this solution, it is also necessary to turn off York, the second Barber. Select York’s tab and in his (Bird) Type field type a few random letters and press tab. York’s tab will be changed to “Not Found” and can’t be run.

Now when the BarberShop scenario is re-run, all five Customers receive their haircuts. The following cropped image shows the results of this second modified run.

This cropped image shows that Dave is now listed as having received a complete haircut.
and that the only Barber was Zeb. The Customer Gave Up table and the Barber Called at table are both empty as well.

For a further variation, load the barbershop20cust.bird flock file without clicking Remove All. This will load another 20 Customers, a second barber, and a second ShopManager. Change the name of Bird #9, the second Barber, to a new name such as Yak or Yorba or Fred. Now when the BarberShop scenario is run, it will behave very much like it did with 5 customers and produce a result similar to the following cropped screenshot.

![Screenshot](image.png)

Note that this run has many duplicate bird names but because of the Customers always adding a short UUID, these duplicate names are made unique so they can be used as keys. If you wish to experiment with the names and UUID, you can turn off the UUID in the Customer.doFirstTimeProcesses method.

But with all these Customers, two Barbers are not enough. So go back to York, Bird #8, the Barber we added manually earlier, and change his type back into a Barber. With any luck, the foods that York eats will still be set to BarberBacklogPod and the foods that York stores will still be set to NullPod.

Now when the BarberShop scenario is run again, it will produce results similar to the following screenshot, where all of the Customers received a haircut.

If one reduces the haircut durations on the XMV tabs, it is easier to give more Customers haircuts with a single Barber. With two or three Barbers, it is easier to accommodate Customers who want long haircuts; the Barbers who didn’t get assigned the long haircut just do more of the shorter duration haircuts.
Summary

The BarberShop scenario is more of a simulation than it is an attempt to compute a specific solution in parallel. It blends some advanced techniques, such as retrieving and storing arrays of BasicPods, with sharing information using singleton pods and extracting info from history files and formatting it to form reports.

While the BarberShop scenario is somewhat contrived, it could just as easily be a sophisticated control system that has multiple birds that eat values from different sources, such as temperature or pressure or RPM, etc, and saves/accumulates their values, evaluates the values, and potentially even responds to the received values to raise or lower temperature, etc.
Chapter 12. File Reading and File Writing

Discussion

As discussed earlier, every Food Eaten and every Food Stored has a File Name option that can be selected by using the appropriate radio button. This option allows you to read in values from a file or to write pod desc values out to a file.

Perhaps the best reason for reading from a file is for testing purposes. For example, if you are comparing two versions of some code and you need to be able to verify that both versions produce the same results. By reading specific values from an input file, you are guaranteed that you are processing exactly the same values so the results should be exactly the same.

Another possibility is to modify the readPodFromFile method so that it reads from a stream, allowing real-time processing of data from some source. This would be useful for situations where you want to process wheel-speed data or voltage levels. I haven’t needed these features so far, so this is left as an exercise for the reader.

Two modes of reading a file are available:

- read a line at a time
- read a single digit at a time

By changing the Boolean parameter to the readLineFromFile method to false, the method will read a single digit from the designated file instead of a single line. Currently, reading a single digit from the file can only be enabled programmatically because I haven’t been able to come up with a use case that was general enough to be useful. In either case, however, the BirdInputFile object will keep track of the location of where the last data was read from in the file, simplifying the coding required to read a file.

Writing to a file is done in text mode, writing one line at a time. Again, more sophisticated outputs are certainly possible but I haven’t needed this capability so these more sophisticated outputs are left as exercises for the reader.

Setup

The first file reading and writing example begins with our old friends, the AddxBirds.

1. Open the Add5.bird file from the addbirds folder.
2. Select tab #1, the Add1Bird. This bird comes configured to eat MagicPods but we will change it to read from a file.
3. Click on the File Name radio button, just below the MagicPod Food Type field.
4. Click on the now-enabled Browse button to the right of the File Name.
5. Open the addBirdInputFile.txt file. This file contains 20 four-digit numbers, each number on its own line.
6. Click the Start All button. The Activity ProgressBars will all start moving and then after a few seconds, they will all stop. Basically, the Add1Bird reached the end of its input file so it stopped putting BlackPods into the TupleTree, effectively starving all the subsequent birds.

7. Click on the TupleTree tab. There will be 20 CoralPods, each with a value that is 15 greater than the values in the input file (1111 became 1126, 2222 became 2237, etc). If you click the Clear All Activities button and rerun this setup, you should get exactly the same results, with the exception that the order of the pods in the TupleTree may differ from run to run.

The order of food pods in the TupleTree is NOT guaranteed because it is contrary to the basics of the TupleTree; requesting a type of food pod will get you any instance of that type of food pod and if you are expecting or depending on a specific ordering or sequence of food pods, then you are serial thinking and not parallel thinking.

Let’s kick it up a notch.

1. Click on the Food Eaten AND radio button in the middle area; the second Food Type will be enabled.
2. Click on the second File Name radio button.
3. Click on the Browse button to the right of the File Name field.
4. Open the addBirdInputFile2.txt file. This file has 25 two-digit numbers.

5. In the Food Stored area, click on the AND radio button.

6. In the Food Type field, type in “bla” and press Tab key. The Food Type field will be updated with BlackPod.

7. Click the Clear All Activities button.

8. Click the Start All button. The birds will all start flying and then stop after a few seconds when the Add1Bird reaches the end of both of its input files.

9. Click on the TupleTree tab. It will contain 40 CoralPods, 20 four-digit numbers from addBirdInputFile.txt and 20 numbers from addBirdInputFile2.txt, mostly 2-digit numbers but also including the number 00000001. Because Food1 & Food2 are eaten with the AND relationship, when the bird reaches either End of File, it stops finding foods to eat. The resulting CoralPod values in the TupleTree will all be 15 greater than the values contained in the input files.

10. Click on Tab #1, Add1Bird. Swap the Food1 and Food2 files, so Food1 reads from addBirdInputFile2.txt and Food2 reads from addBirdInputFile.txt.

11. Click on Clear All Activities and then Start All. When the birds stop flying, the TupleTree will again have 40 CoralPods.

12. Click on Tab #1, Add1Bird. Click on the OR radio buttons between Food1 and Food2.

13. Click on the None radio button (between Stored Food1 and Stored Food2).

14. Click on Clear All Activities and then Start All. Note that the Food1 file is read completely and then the Food2 file is read. When the birds stop flying, the TupleTree will have 45 CoralPods – 20 from addBirdInputFile.txt and 25 from addBirdInputFile2.txt

**NOTES**

When eating two foods in an OR relationship, the bird attempts to get Food1 first and only tries to get Food2 if it doesn’t find a Food1. When the bird is reading from a file, it always finds a food until it reaches the end of file (EOF). This explains the way the files are read – it always finds a Food1 from file1 so it never looks for Food2 until file1 reaches EOF at which time the bird never finds a Food1 so only reads from the Food2 file.

If you had skipped Step 13, there would be 90 CoralPods in the TupleTree at the end of the run instead of 45 because each single line of a file would be saved twice. When the Stored Food AND radio button is selected and only one food pod is eaten, it is saved to both Food1 AND Food2. Only one food pod is eaten when either the OR radio button or the None radio button is selected. If the Food Eaten AND radio button is selected, two food pods must be eaten: Eaten Food1 → Stored Food1 and Eaten Food2 → Stored Food2.
Variations

The first set of variations is experiment with using different Stored Food Types.

1. Click on the Food Eaten AND radio button.
2. Click on the Food1 ellipses button and select addBirdInputFile.txt.
3. Click on the Food2 ellipses button and select addBirdInputFile2.txt.
4. Click on the Food Stored AND radio button.
5. Change the second Food Stored type to BluePod
6. Click the Clear All Activities button.
7. Click on Start All button.

After the birds stop, verify that the TupleTree holds 40 CoralPods. The final values for the CoralPods will not all be 15 larger than the values read from the input files; some will be 15 larger and some will be 13 larger. This is because the values read from the Food2 file (addBirdInputFile2.txt) will be saved to BluePods. These pods will by-pass Add2Bird so will never get 2 added to their values. These CoralPods will also have shorter histories because the Add2Bird never ate them.

Let’s take this a little further.

1. Change the second Food Stored type to BrownPod.
2. Click the Clear All Activities button.
3. Click the Start All button.

After the birds stop, verify that the TupleTree holds 40 CoralPods. Verify that the values for the second food CoralPods are only 10 greater than the values in the addBirdInputFile2.txt file and that their pod histories are shorter because they don’t include events for Add2Birds or Add3Birds.

Now let’s use a totally different pod.

1. Change the first Food Stored to PinkPod.
2. Click the Clear All Activities button.
3. Click the Start All button.

When the birds stop, click on the TupleTree tab; it will hold 40 pods but this time it will be 20 CoralPods and 20 PinkPods. The values of the CoralPods will all be 15 higher than the values in the input file. However, the values of the PinkPods will all be only 1 greater than their values in the input file because none of the other birds processed PinkPods.
So far we have only read from input files; let’s write to output files.

1. If required, click on Tab #1 to select the Add1Bird.
2. In the first Food Stored, click on the File Name radio button.
3. Click on the ellipses button to the right of the File Name field.
4. Navigate to the desired location in your file system and then type in the desired file name, such as out1.txt.
5. In the second Food Stored, click on the File Name radio button.
6. Click on the ellipses button to the right of the File Name field.
7. Navigate to the desired location in your file system and then type in the desired file name, such as output2.txt.
8. Click Clear All Activities button.
9. Click Start All button. On the Activities tab, only the Add1Bird shows any activity. The other birds are looking in the TupleTree but the Add1Bird didn’t store anything there so the other birds starve.
10. Click on the Food Supply tab and only the input files and the output files show any activity.
11. Click on the TupleTree tab. It will show that it contains zero food pods (all the results were sent to the output files).
12. Navigate to the location selected for the output files and open one of the output files. It will have contents similar to the next screenshot, where all the values will be 1 greater than the values in the input file.
13. Navigate to the location of the second output file and it will have similar contents to this screenshot, except that the values will all be greater than the values in the second input file.

Note that the output files put one value on each line, similar to the input file that were read. If a different output format is required, you can modify the behavior of the BirdOutputFile as required. Once again, this is left as an exercise for the reader.

**Summary of File Reading and File Writing**

Reading from files and writing to files in ConcX was made as simple as possible so you could spend your time thinking about the important things, like how to make the desired events happen in parallel.

Reading from an input file provides two advantages over eating MagicPods:

1. The exact same sequence of input values can be read every time so the validity of changes can be easily demonstrated
2. Values that caused failures or produced undesired results can be saved in the input file so future modifications can be tested against these values to ensure that the future modifications didn’t reintroduce the undesired results.

If you have specific file formats that you need to support, then you’ll need to dig in and learn the code details for BirdInputFiles and BirdOutputFiles. If you do, you’ll notice that all the file operations are done using standard Java input/output. This means that you’ll need to make sure that you don’t have two different birds reading or writing to a file concurrently. Java nio (new Input Output) holds promise for more advanced file reading and writing, but those promises won’t be investigated for a while. This is all open source, so if you have a need for java.nio features or capabilities, feel free to dive in and upload your improvements.
Chapter 13. Working with Databases

Discussion

Working with databases in ConcX turns out to be simpler than one would expect. Each bird knows how to connect to one DBMS and perform one SQL operation, simplifying the programming of the bird. This sounds limiting but it’s really not. Database birds are endpoints, shuttling info from and back into the chosen DBMS. Data retrieved by database bird(s) moves into the TupleTree where a variety of other birds can operate on the data until it is ready to be moved back into the DBMS by the appropriate database bird(s).

The details of the database connection are held in the XMV fields and used to establish a JDBC connection in the doFirstTimeProcesses method. The actual SQL statements are executed in either the preferredFeedingMethod or preferredStoringMethod, depending on which way the data is moving. When the bird stops, it closes its database connection in the finalCleanup method.

In the included examples, a simple ETL process is demonstrated. Data is extracted from a MySQL database and put into the TupleTree, then eaten in parallel by multiple database birds that transform the data and then load them into an Apache Derby database.

Setup

If necessary, close ConcX and then restart it. Open the database1.bird file from the db folder. When the bird is loaded, click on its XMV tab. It will look similar to the following screenshot:

![XMV Fields Screenshot]

The settings in the XMV fields above show that we’re connecting to a MySQL database hosted on my computer (localhost) but it could be hosted on any other computer that could
be reached from your network.

Note that the user’s password is stored in plaintext. If this was a production system, this wouldn’t be acceptable but it’s acceptable for development efforts. For better security, either a real human would be required to manually provide the password or an automated method of providing the authorization info would be required.

As required, modify the database configuration settings to match what is needed to work with your database. This example uses the sample database “world” that is provided with MySQL v. 5.7. If your SQL command syntax doesn’t match MySQL’s that is shown, modify the SQL statement as required.

If required, start your database(s).

Click the Start All button and the DatabaseBird flies for barely a second and then stops. Click on the TupleTree tab and there should be 3 BrownPods listed for Oregon (assuming you are using the MySQL sample database world). The SQL query statement returned a result set containing 3 items and each one of these items was copied into its own seed that was stored in its own BrownPod.

If you save the contents of the TupleTree, you can see the details of the BrownPods, each with the details of one city in Oregon, similar to the screenshot below.

If you click Clear All Activities and then modify the SQL statement so the district (state) is “Texas” or “California” or any of the other states. Now when you click Start Me button, all of the cities in the state you selected will be put into the TupleTree. If you make the district = ‘California’, there will be 68 BrownPods in the TupleTree.

If desired, you can change the statement to:

Select * from city where countryCode = ‘USA’

When you run this statement, there will be 274 or so cities in the TupleTree representing the largest cities in the USA.
Now, either exit and restart ConcX or change the Type of the bird to something unrecognized in ConcX such as InDatabaseBirdxq and the press Tab. Its Type will change to be “Not Found” so it will be ignored when you click the Start All button.

Open the database2.bird from the db folder. Click on the XMV tab of the OutDatabaseBird and it will look similar to the following screenshot.

![Picture of XMV tab](image)

This shows that it is configured to access an Apache Derby DBMS. Because all of the database operations on happening on my computer (//localhost) I didn’t bother to set up a user name and password – NEVER DO THIS ON A PRODUCTION SYSTEM.

I set up a database named avian in Derby and created a table named “DerbyCities” with the following layout:

```sql
ij> describe derbycities;
COLUMN_NAME | TYPE_NAME | DECimals | NUMimals | COLUMN_DEF | CHAR_OCTets | IS_Null
-------------|-----------|-----------|-----------|-------------|-------------|---------
CITYID       | INTEGER   | 0         | 10        | NULL        | NULL        | YES     
COUNTRYCODE  | VARCHAR   | NULL      | NULL      | 3           | 6           | YES     
STATE        | VARCHAR   | NULL      | NULL      | 20          | 40          | YES     
CITY         | VARCHAR   | NULL      | NULL      | 24          | 48          | YES     
POPULATION   | INTEGER   | 0         | 10        | 10          | NULL        | YES     
```

I used the following statement to create the derbycities table:

```sql
ij> create table derbycities (cityid int, countrycode varchar(3), state varchar(20), city varchar(24), population int);
```

You can use any desired Java-accessible DBMS and create a database & table similar to the above. Update the database info (URL, db name, etc.) as required to match your DBMS.
Derby has the option to run it as an embedded DBMS or as a network server. In the examples in this chapter, Derby was run as a network server using the startNetworkServer.bat file, provided with the Derby installation. Hopefully running in this mode most closely resembles what would be encountered in a network and/or corporate environment.

If necessary, click the Clear All Activities button and then click Start All button. The OutDatabaseBird will run for a second and then stop. If you click on the Food Supply tab, its ProgressBars will be blank. If you click on the TupleTree tab, it will show that it is empty. The data that was extracted from the MySQL database was inserted into the Derby database. If you query your (Derby) database, you should see results similar to the following:

```
\$ Command Prompt - ij

1j> select * from derbycities;

<table>
<thead>
<tr>
<th>CITYID</th>
<th>USA</th>
<th>STATE</th>
<th>CITY</th>
<th>POPULATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>39953</td>
<td>usa</td>
<td>OREGON</td>
<td>portland</td>
<td>529121</td>
</tr>
<tr>
<td>33834</td>
<td>usa</td>
<td>OREGON</td>
<td>eugene</td>
<td>137893</td>
</tr>
<tr>
<td>35546</td>
<td>usa</td>
<td>OREGON</td>
<td>salem</td>
<td>138924</td>
</tr>
</tbody>
</table>

3 rows selected
1j>
```

Note that the USA is now all lowercase, Oregon is all uppercase and the city is all lowercase. These were all transformed from their original (upper or lower) cases by the OutDatabaseBird on the fly before inserting them into the (Derby) database. Also note that it created random CITYID values before inserting them into the db – a real DBMS would have created id’s that were guaranteed to be unique.

If you have been successful to this point, great! It means that you have correctly set up your system to concurrently access two different DBMS. If you didn’t get any records in the second database, look at the bottom of the NetBeans screen for error messages and read the history of each bird for error messages to help resolve any problems.

**ETL Procedure**

1. Shutdown and restart ConcX (or make the Types Not Found).
2. Open the database3.bird file from the db folder.
3. Click on the XMV tab on the #2 tab (InDatabaseBird). It will show that the district (state) has been changed from Oregon to California.
4. If necessary, click the Clear All Activities button and the click the Start All button.
5. The birds will fly for a second or two. Click on the Food Supply and TupleTree tabs. They will both show as empty.
6. Switch to the second dbms and review the contents of the derbycities table. It will
contain 68 city records for the state of California (plus any left over from previous runs).

7. Delete the records in derbycities with a statement similar to:
   ```sql
   ij> delete from derbycities where cityid > 0;
   ```

8. Change the district name to Texas or Ohio and repeat steps 4 thru 8.

9. Change the SQL Statement for the InDatabaseBird to be similar to:
   ```sql
   select * from city where countrycode = 'USA'
   ```

10. Repeat steps 4 thru 7. The second database should now hold 274 or so records with the names of the largest cities in the USA.

11. Repeat steps 1 thru 9 but open database5.bird. It should run the same way as before but should finish sooner because more OutDatabaseBirds are consuming BrownPods and inserting them into the second database.

12. Repeat steps 1 thru 9 but open database9.bird. It should run the same way as before but should finish sooner because even more OutDatabaseBirds are consuming BrownPods and inserting them into the second database.

13. Change the SQL Statement for the InDatabaseBird to be similar to:
    ```sql
    select * from city
    ```
    This will select all 4300+ records in the MySQL db.

14. Repeat steps 4 thru 7. It should take longer to finish now because of the increased number of records it is processing. Most importantly, it should generate a bunch of SQL Exceptions because 20 or 30 cities have names that are longer than can be held in the City varchar(24) field.

At the top of the right side of the screen, the Errors ProgressBar has been updated to show that errors were detected as shown below. Click on the TupleTree tab.

The TupleTree now contains one Err (error) record for every SQL exception.
Save the TupleTree to a file. It will contain all of the cities whose full names are too long to fit into the CITY field of the derbyCities table.

As a database designer, you now have the statistics you need to make some hard decisions:

- Should the CITY field be made wide enough to hold every city name? Nice for the <1% of the cities with names longer than 24 chars but at a cost of additional empty (wasted) space for 99% of the records.
- Should these city names be manually entered into the second db? A good solution if this is a one time data transfer, but a poor solution if this data transfer happens frequently.
• Should the OutDatabaseBird be modified to check the length of every city name and truncate them to an acceptable length? This could be a good choice if the number of records being checked isn’t significant or else the extra operations could slow down the whole process. And so on.

Fortunately, we don’t have to make this decision. Instead, it is important to note the existence and use of the Err pods. In this last scenario, the Err pods were generated and stored in the TupleTree with just 2 lines of code. Err pods are highly visible on the right side of ConcX screen, making it easy to detect problems in large batch runs.

**Summary**

This scenario demonstrated that ConcX can provide a relatively simple and flexible interface to interact with one or more DBMS concurrently. While no attempt has been made to present a real-world system, it doesn’t take too much to imagine extending the database birds presented here to issue SQL statements that access multiple tables in a database (or data in a legacy non-relational DBMS) to extract some desired information, have other birds perform more sophisticated transforms, and then have the data uploaded into a DBMS that can slice and dice the data and provide the required Business Intelligence Reports.

Consider, for example, one of the legacy hierarchical databases with a “header” record followed by one or more repeating groups of detail records. Each group of records could be stored to the TupleTree by one bird, another bird could eat those groups of records and store one or more food pods in the TupleTree, where other birds would format them into proper INSERT statements based on the DBMS that they will go into, and finally other database birds would do the actual uploading of data.

One nice aspect of processing databases in parallel is that each chunk of data is downloaded or uploaded by a separate bird, so if one bird has to perform extensive processing (for example, looking up a city’s GeoCode using the Google Maps API) or encounters network conflicts or gets hung up, none of the other birds are affected — they can all keep processing their own data.

This scenario also demonstrated the Err pod usage, with front-page visibility, familiar TupleTree placement, and the capability thru coding to capture whatever level of details needed. Developers are frequently tempted to code solutions to problems or error conditions that might never actually happen. The Err pods allow developers to code for the most common cases and put any data that don’t match these cases into Err pods for later analysis. By running against realistic data, the developers can see how many times certain error conditions actually occur and make their coding decisions accordingly.
Part 2: Code Details

This part of the guide contains information about how ConcX is structured and how it runs parallel apps developed for ConcX. This part begins with a brief introduction to the main components of ConcX and how they work together. Following this introduction will be a discussion of how to create your own birds and food pods.

**Basic Components**

The above diagram shows the basic components of ConcX. The boxes in the upper half are GUI components while the boxes in the lower half represent the birds that fly when you push the Start Me/Start All button.
NOTE:

All birds provided in ConcX are descendants of BasicBird

The BasicBird implements most of the functionality used in ConcX. BasicBirds can fly without any additional code or modifications. Creating custom birds that extend BasicBird allows you to inherit all of the functionality of BasicBird without the any programming. If these behaviors are unsuitable then one is free to implement AdvancedBirds or MarvinBirds as required to meet one’s needs.
NOTE:
All pods provided in ConcX are descendants of BasicPod

Food pods are containers that hold data to be shared through the TupleTree. The Desc(ription) field can be used to hold simple values like strings or numeric amounts. Anything more sophisticated than simple types should be put into one or more seeds and added to the appropriate pod.

The pods hold arrays of BasicSeeds; pods have the usual set of functions to manage the array of seeds. Each seed has one kernel of type Object so it can hold any type of object.
BirdFile (interface)
isFull()
getType()
getFullName()

BirdFileImpl extends File
isFull()
getType()
getFullName()
getBaseName()
getBaseName(boolean)
getPathToBird()
getPathToBird(boolean)
getPathAndBaseName()
getShortPath()
getShortPathAndBaseName
getPathAndFullName()
getDotPathToBird()
getDotPathAndBaseName()
getDotPathAndFullName()
constructors, getters and setters

BirdInfo extends BirdFileImpl
getType()
constructors

PodInfo extends BirdFileImpl
getType()
constructors

BirdInputFile extends BirdFileImpl
implements BirdIOFile
isInputFile()
isReady()
open()
close()
readLine()
constructors, getters and setters

BirdInputFile extends BirdFileImpl
implements BirdIOFile
isInputFile()
isReady()
open()
close()
readLine()
constructors, getters and setters

BirdIOFile (interface)
isInputFile()
isReady()
BirdFiles are helper objects that BasicBirds rely upon to simplify accessing and managing bird types, food pods, and input/output files. BirdFileImpl objects extend Java’s File object to gain its variety of file system neutral methods. BirdFileImpl then adds numerous ConcX-specific methods to provide the functionality needed.

**BeakFactory**
getBeak()

Returns a Beak that is either a BeakForTree or a BeakForFile based on the parameters provided.

---

**Beak (interface)**

- getItem() (1, 2, & 3 param versions)
- getSingleton()
- patientGetItem()
- getMultiItem()
- getTargetPod()
- storeItem()
- storeMultiItem()
- storeSingleton()
- hasNext()
- getBeakType()
- getPodType()
- getPodTypeID()
- open()
- close()
- setPodInfo()

**BeakForTree**

getBeak() (1, 2, & 3 param versions)

**BeakForFile**

getBeak() (1, 2, & 3 param versions)

---

**BeakForTree implements Beak**

- getItem() (1, 2, & 3 param versions)
- getSingleton()
- patientGetItem()
- getMultiItem()
- getTargetPod()
- storeItem()
- storeMultiItem()
- storeSingleton()
- hasNext()
- getBeakType()
- getPodType()
- getPodTypeID()
- open()
- close()
- setPodInfo()

**BeakForFile implements Beak**

- getItem() (1, 2, & 3 param versions)
- getSingleton()
- patientGetItem()
- getMultiItem()
- getTargetPod()
- storeItem()
- storeMultiItem()
- storeSingleton()
- hasNext()
- getBeakType()
- getPodType()
- getPodTypeID()
- open()
- close()
- setPodInfo()
A beak provides a unified method of accessing data, regardless of whether it is in the TupleTree or in a file. Other kinds of Beaks can be created to meet future needs, such as a database beak or a stream beak. It is anticipated that creating new BeakFor types will allow new data sources to be added with a minimum of modifications to BasicBird code or disruption of functionality.

**Programming in ConcX**

**Starting New Scenarios**

New scenarios are created by creating new packages in the ConcX package hierarchy for any new bird types and for any new food pods. For example, if you were creating a parallel billing system, you would add a package such as com.avian.birds.billing and then add any new birds to that package. And you would add a package such as com.avian.foods.billing and then add any new food pods to that package.

**Creating New Birds**

New birds normally extend BasicBird and then override any necessary BasicBird methods. Common methods to override are:

- **doFirstTimeProcesses** – this method is probably the most frequently overridden method. It is commonly used to read XMV values or to put necessary food pods into the TupleTree.

- **eatFood1/2Beak** – these methods are typically overridden to allow access to non-standard ways of eating, such as getSingleton or getMultiItem. It is also overridden if you wish to change the messages that are written to the bird history or pod history.

- **setLastAteTime** – this method is a massive switch statement that compares the results of a bird’s eating to how it was configured to eat and decides if the bird did in fact successfully eat and if so, updates the appropriate activity and food supply progress bars. If you wish to change (or remove) the bird history or pod history messages, it could be done here. It is best to always copy this method from BasicBird instead of from a different bird so that you always start with the full and complete method.

- **digestSeeds** – this method is typically where the unique work of the new bird would be done, such as performing calculations, formatting Strings or transforming values. If you know that your food pods will not have seeds, you can override the default method with an empty method.

- **afterDigestion** – this method can be used to perform final checks or summarize all seeds, etc.

- **digestionDelay** – this method was created to allow scenarios to simulate the length of time the digestion processing will require. Once the digesting processes have been coded, the digestionDelay can be changed to do nothing except return.
• strFoodsWithBeaks – this method is typically overridden to access non-standard ways of storing food, such as storeSingleton or storeMultiItem. This method is large and comprehensive so if you are going to override it, it is best to copy the method from BasicBird and then reduce or remove cases that don’t apply. It also is responsible for converting inedible pods into error pods so if you don’t include this functionality in your new bird then the dynamic on screen reporting of errors will be lost. It is also responsible for updating the food progress bars. It is also overridden if you wish to change the messages that are written to the bird history or pod history.

It is totally acceptable to create a new bird type for your new scenario and do nothing more than extend BasicBird and not override any of its method. By creating a new bird type you have made it clear that this bird belongs to a specific scenario and what role it plays in your scenario. For example, a bird type of Butcher or DJ or Trimmer would give some indication of how you expect it to behave within your scenario.

Creating New Food Pods

New foods can be easily created by copying an existing similar food pod and changing its constructor to assign new treeID, a description, and a new podType.

The treeID must be a unique String in ConcX so the correct food pods are found. The convention in ConcX is to assign a 2 or 3 character prefix based on their food pod type (folder). For example, the colorfoods all begin with “col” and pi foods all begin with “pi”. The character prefix is followed by a numeric identifier, such as “01” or “005”. This convention makes it relatively simple to add new treeIDs because only the food pods in the appropriate package need to be checked for duplicates. However, any method of assigning treeID Strings can be adopted, as long as it always provides each food pod type a unique id.

The treeID is the key used by the TupleTree to retrieve matching food pods. It is assumed that shorter keys provide better performance than longer keys such as “TotallyBerserkerMadSkillsAtAccessingTheLandOfPlentyPod” but that assumption has never been tested.

The description and the podType are Strings that aid us humans in remembering what food pod type is in the TupleTree. Without the description and podType, we would have trouble remembering that “col02” represented a BluePod or that “bar03” represented a BarberCompletedCustomerHaircutPod.

Specifics - Patient Eating

ConcX is set up by default to use PatientEating when it must eat BOTH Food1 and Food2 (the AND radio button is selected). Every BasicBird (and descendants) always tries to get Food1 before trying to get Food2. If every bird could always get Food2 immediately, this wouldn’t be a problem. However, the question is what to do if a bird can only get its Food1?

• Should it give up and put back its Food1 pod?
• Should it hold onto its Food1 pod until it finally gets its Food2?
The first option would cause a lot of unnecessary churning with birds getting and putting back unpaired food pods. And the second option would lead to deadlock over shared food pods; eventually every bird would be holding their Food1 and wouldn’t release it until they got their Food2 that a different bird was holding as their Food1.

ConcX addresses this situation by retrieving their Food2 food pods using the three-parameter version of beak.getItem. This version of getItem gets a copy of Food1 as one of its parameters. It uses this copy of Food1 to decide how to eat its Food2. The following is the actual code that is executed.

```java
public BasicPod getItem(String podId, BasicPod pod1, BasicBird bBird) {
    if (bBird.isEatsAND() && pod1.isEmpty()) {
        return new BasicPod(); // empty pod
    } else if (bBird.isEatsONE()) {
        return new BasicPod(); // empty pod
    } else if (bBird.isEatsOR() && !pod1.isEmpty()) {
        return new BasicPod(); // empty pod
    } else if (bBird.isEatsAND()) { // pod1 ! empty so must have a second pod
        // will try multiple times to get it
        // 10 = number of tries, 100L = how long to pause between attempts
        return patientGetItem(podId, 10, 100L, bBird);
    } else {
        return getItem(podId, bBird); // bBird.isEatsOR AND pod1 is empty so
        // do regular retrieve
    }
}
```

In most situations this code returns an empty food pod, signifying that nothing was found. For example, the first if statement returns an empty pod if the bird’s food eaten radio button is set to AND (both Food1 & Food2 must be eaten) and Food1 was not found. The two situations where the Bird tries to get a Food2 are:

- Either Food1 OR Food2 can be eaten AND Food1 was not found
- Food1 was successfully eaten AND the bird was set to eat both Food1 & Food2

The second possibility results in trying to eat Food2 patiently, which means that it will try a configurable number of times to get Food2. In the above code, the number or tries is set to 10. If it tries and fails 10 times in patientGetItem to get Food2, it will put back the Food1 that it was holding.

The 100L is the number of milliseconds that the bird will sleep between each attempt. If the bird tries 10 times with 100ms pause between each attempt, the total time that the bird would hold its Food1 while it attempts to get its Food2 is 1 second. All the code for the attempts to get Food2 and for the pauses is contained in the patientGetItem.

Now 1 full second is a long time in computer-land so one of the configuration settings for each bird’s Patience is how long it should try to retrieve its Food2. The default value is 500ms, which means it will try approximately 5 times (5 * 100ms/try) before giving up. Selecting a value less than 500 will produce fewer tries before giving up. Selecting more than 1000 ms will result in no more than 10 attempts(10 * 100ms) because 10 is hard coded into the call to patientGetItem, as is the time to pause between attempts.
If changing the Patience setting is insufficient, the hard coded values can be changed, preferable by overriding the three parameter version of beak.getItem. Just remember that the bird is holding Food1 while it trying to get Food2 so if you make the bird’s patience too long, that it will force any other birds to wait longer to access the bird’s Food1.

**Specifics - Barbershop Programming Details**

The Barbershop has three kinds of birds:

- **Barber** – finds longest waiting Customer and gives that person a haircut
- **Customers** – put their names on the waiting list. Sit down and wait for their turn or give up and leave if they can’t wait any longer.
- **ShopManager** – tracks completed haircuts and customers who gave up and dynamically generates the Daily Transactions report.

**Barber Details**

The Barber reads BarberBacklogPods, and writes either BarberClientNotFoundPods or BarberCompletedPod. Before giving a haircut, the Barber finds the customer who has been waiting the longest on the waiting list (BacklogPods). The selected Customer then gets a haircut according to their preferences.

The Barber does almost all of the interesting things in this scenario. For example, when looking for the oldest customer, the Barber gets all of the BarberBacklogPods in a single call to the TupleTree.readAndRemoveSome method. Because it gets all of the backlog records in a single synchronized method, it guarantees that multiple Barbers can access the backlog without more than one Barber trying to cut the hair of the same customer. The first Barber who successfully retrieves the backlog holds the entire backlog and can safely find and remove the longest waiting customer without fear that another Barber could find the same customer.

A small possibility exists of a Customer “taking cuts” if a Customer adds their name to the waiting list and a second Barber then takes that name from the waiting list before the first Barber puts the other waiting Customers back in the TupleTree. In the real world, this kind of thing happens now and then and is normally handled by hurt feelings and angry customers, so it is considered a non-consequential risk. However, if the proper sequence is absolutely required, it is recommended a new method be created that sorts the found records in the desired order, such as readAndRemoveOldest or readAndRemoveLargest, etc.

This TupleTree method does what its name indicates; it finds and removes all matching key-value pairs in the TupleTree up to a maximum number specified in a parameter. The matching key-value pairs are moved into an ArrayList and then removed from the TupleTree. Since this is a synchronized method, there is no danger of key-value pairs being added or deleted while the matching key-value pairs are being moved to the ArrayList and then removed from the TupleTree. The ArrayList is added as a single object to the kernel of a BasicSeed which is then added to a single BasicPod. This single BasicPod is then returned...
to the calling method.

Note that the max number to retrieve parameter is normally set to a value much larger than would reasonably be expected to exist in the TupleTree because there is no performance penalty for entering 100 or 1000 – the TupleTree automatically creates a new ArrayList with a size that matches the actual number of key-value pairs that were found in the TupleTree. This ensures that the ArrayLists are always created as the correct size and that there will never be any time-consuming resizing of the ArrayList.

The next interesting thing the Barber does happens during digestSeeds2; the Barber extracts the ArrayList of matching items from the kernel of seed[0] of the BasicPod it received and then sorts the ArrayList using a custom comparator. Since the items are sorted in ascending order based on the time that Customers put their names on the waiting list, which makes the longest waiting Customer the first item in the list. This first item is removed and the remaining items in the ArrayList are put back into the TupleTree, again using a single call to the TupleTree.putSome method. If one were to create a custom retrieve some TupleTree method, using a sort() and custom Comparator similar to those in the Barber would probably make sense.

The next interesting thing the Barber does also happens during digestSeeds2. The Barber gets the Customer’s name from the selected (retained) item and then gets the Customer from the TupleTree using the Customer’s name as the key. This is a departure from standard ConcX behavior, where each FoodPod TYPE is assigned a treeID. For example, BluePods have a TreeID of “col02” and GreenPods have a TreeID of “col09” and retrieving a BluePod is normally done by searching the TupleTree for keys matching “col02”.

In the BarberShop scenario, however, it is impractical (impossible) to provide a FoodPod for every possible name (imagine having thousands of first name pods, such as AbePod, AllenPod, AlanPod, AlPod, AlicePod, AliciaPod, and so on). Instead, the Barber retrieves the appropriate Customer record by using the Customer’s name (retrieved from the BarberBacklogPod) for the actual TreeID. And blank and duplicate names are handled correctly using UUID’s. See Customer Details for more info.

While other methods of assigning names or TreeIDs could be developed, the method used in the BarberShop scenario provides a general purpose algorithm that could be applied in other scenarios, such as invoicing customers, tracking phone call usage, etc., where creating a unique FoodPod for every possible ID (existing or future) is impractical.

Once the correct Customer record has been retrieved, the Barber gives a haircut by performing the giveHaircut method, which basically just delays (Thread.sleep) for the length of time required to give the Customer the specified haircut. See Customer Details for more info.

If the Customer record cannot be retrieved, that means that Customer has given up waiting for their haircut. On the chance that the Customer’s record is just temporarily unavailable, the Barber tries three times to retrieve, the same as a real barber would call a customer’s name several times before getting the next name from the list. If after several tries the
Customer wasn’t located, the Barber creates a BarberClientNotFoundPod with the details about the customer name that was called and when they were called.

**Customer Details**

When a Customer enters the BarberShop (performs its doFirstTimeProcesses method, it creates a CustInfoFindByNamePod and a BarberBacklogPod. Each custInfo pod has a unique name (TreeID) assigned to it as follows:

- If the Customer’s name is blank or “none”, a 7-digit UUID is created and used.
- Otherwise, a 3-digit UUID suffix is added to the end of the Customer’s name

The scheme described above ensures that every Customer has a unique name and will be called in the correct order. In a real barber shop, if two or more Dave’s are waiting, the barber calls out “Dave” and multiple customers will respond and then they will try to straighten out who was waiting longest. By having each Customer assign themselves their own unique name (and then remember it for later use), the problem of duplicate names (or multiple blank names) is solved.

When each Customer put their name on the waiting list, they also indicated what kind of haircut they wanted by specifying how long (in milliseconds) the haircut will take. The underlying assumption is that the longer the haircut takes, the “fancier” the haircut will be. The duration of the haircut also determines how expensive it will be; longer haircuts cost more money.

The kind of haircut (duration) also is an effective way of demonstrating that each thread is running independently. To show this capability, change one of the Customers to have a very long haircut (7000 instead of the typical 1500). When this scenario is run in this configuration, most of the Customers following the long haircut give up. If a second Barber is added to the scenario, however, one of the Barbers will be tied up with the slow Customer while the other Barber gives the bulk of the other haircuts.

While the Customer is waiting, it repeatedly checks its status. Each Customer checks its status by retrieving its own CustInfoFindByNamePod using its own name and checks its desc(ription) field.

- If the value is “waiting for haircut” and the Customer’s waiting time is exceeded, the Customer gives up and leaves otherwise the Customer continues waiting.
- If the Barber has called the Customer’s name, the Barber will have changed the desc to “haircut in process”. When the Customer retrieves its record, it changes its status in both its bird history and in its CustInfoFindByNamePod. It also adds 300ms to its own personal lifetime – this ensures that if the Customer was called just before it was set to expire, it will live long enough to complete its haircut (everyone wants to look good for their funeral).
- If the Barber has completed the Customer’s haircut, the Barber will have changed its desc to “completed haircut”. When the Customer retrieves its record, it changes its
status in both its bird history and transforms its CustInfoFindByNamePod into a CustomerCompletedPod and updates its pod history. The CustomerCompletedPod is then eaten by the ShopManager (see ShopManager Details for more info). The Customer then leaves the barber shop.

**ShopManager Details**

The ShopManager is responsible for managing the haircut results and collecting (tallying) the payments. There are four possible haircut results (results pods):

- Barber Completed Haircut (BarberCompletedPod) [Seed 2]
- Barber Did Not Find Customer (BarberClientNotFoundPod) [Seed 3]
- Customer Haircut Completed (CustomerCompletedPod) [Seed 0]
- Customer Gave Up (CustomerGaveUpPod) [Seed 1]

The ShopManager continuously searches the TTree for these four types of pods, eats (removes) them whenever found and records their transactions.

After starting, the doFirstTimeProcesses method creates an instance of DailyTransactionsPod that only the ShopManager uses. The DailyTransactionsPod is created as a Singleton using the beak.StoreSingleton which in turn uses the TupleTree.putCheckedSingleton method to ensure that one and only one instance of DailyTransactionsPod is created. (If each ShopManager is allowed to create its own DailyTransactionsPod, there will be multiple DailyTransactionPods updated by random ShopManagers which spreads the results over all the DailyTransactionsPods in the TTree. By using the StoreSingleton method to create only one instance of DailyTransactionsPod, every ShopManager will use this singleton (DailyTransactionsPod) and consolidating all of the results.) This is similar to having all store managers using the same General Ledger to record all transactions.

After the ShopManager successfully gets the DailyTransactionsPod, it uses its afterEating method to gather any results. It first searches for and removes one BarberCompletedPod, and then finds and removes one BarberClientNotFoundPod, then one CustomerCompletedPod, and finally one CustomerGaveUpPod. Each type of pod is empty if its corresponding pod wasn’t found. If more than one pod of any type is found, it doesn’t matter; only one is returned. Any remaining pods of any result types just remain in the TTree until the next time the ShopManager searches for results to put into the DailyTransactionPod.

The ShopManager overrides the setLastAteTime2 method for purely esthetic reasons; the standard method writes a message to the DailyTransactionsPod every time it finds that pod which results in the DailyTransactionPod holding hundreds of unnecessary unimportant lines. Comment out this method if you wish to view all of the pod messages.

ShopManager also overrides the strFoodsWithBeaks method for the same reason; to eliminate hundreds of unnecessary unimportant lines in the DailyTransactionsPod.
Comment out the strFoodsWithBeaks method if you wish to view all of the pod messages.

The ShopManager overrides the digestFood method and uses it to transfer information from any non-empty results pods into the DailyTransactionsPod. The DailyTransactionsPod maintains four seeds, one for each result, in its array of seeds as follows:

- Seed[0]: Customer Completed Haircuts Results
- Seed[1]: Customer Gave Up Results
- Seed[2]: Barber Completed Haircuts Results
- Seed[3]: Barber Did Not Find Customer Results

Each seed has a kernel which holds objects. For the DailyTransactionsPod, each seed’s kernel is an array of Strings with each String in the array representing one line of results. Seed[0] and Seed[1] correspond to Seed[2] and Seed[3], respectively. There should always be corresponding Completed Haircut information in Seed[0] and Seed[2]. There should also be corresponding info in Seed[1] and Seed[3]; some useful information can be determined by comparing these two seeds. The info in Seed[1] records which Customers gave up, if any, and the times when they gave up. The info in Seed[3] lists which Customers were called by the Barber(s) for a haircut and the time at which they called them. Every Customer name in Seed[1] should be listed in Seed[3] but their times will be different because the Barber won’t have noticed that a Customer left until the Barber calls the Customer’s name (or more accurately, when the Barber can’t find a CustInfoFindByNamePod for the name the Barber pulled from the waiting list (BarberBacklogPod).

The ShopManager maintains four BasicPods that temporarily hold the results pods that it retrieved from the TupleTree. The ShopManager is responsible for extracting the info that it needs from these temporary results pods, formatting the info, and adding that info to the appropriate String arrays held in Seed[0] thru Seed[3].