Concurrent Programming in Pharo

Stéphane Ducasse and Guillermo Polito

December 20, 2019

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

Part I

TaskIt

>Anything that can go wrong, will go wrong. – Murphy's Law

Expressing and managing concurrent computations is indeed a concern of importance to develop applications that scale. A web application may want to use different processes for each of its incoming requests. Or maybe it wants to use a "thread pool" in some cases. In other case, our desktop application may want to send computations to a worker to not block the UI thread.

Processes in Pharo are implemented as green threads scheduled by the virtual machine, without depending on the machinery of the underlying operating system. This has several consequences on the usage of concurrency we can do:

- Processes are cheap to create and to schedule. We can create as many as them as we want, and performance will only degrade if the code executed in those processes do so, what is to be expected.
- Processes provide concurrent execution but no real parallelism. Inside Pharo, it does not matter the amount of processes we use. They will be always executed in a single operating system thread, in a single operating system process.

Also, besides how expensive it is to create a process, to know how we could organize the processes in our application, we need to know how to synchronize such processes. For example, maybe we need to execute two processes concurrently and we want a third one to wait the completion of the first two before starting. Or maybe we need to maximize the parallelism of our application while enforcing the concurrent access to some piece of state. And all these issues require avoiding the creation of deadlocks.

TaskIt is a library that ease Process usage in Pharo. It provides abstractions to execute and synchronize concurrent tasks, and several pre-built mechanisms that are useful for many application developers. This chapter explores starts by familiarizing the reader with TaskIt's abstractions, guided by examples and code snippets. At the end, we discuss TaskIt extension points and possible customizations.

CHAPTER 1

Introduction

1.1 Downloading

Current stable version of taskit can be downloaded using metacello as follows:

```
Metacello new
  baseline: 'TaskIt';
  repository: 'github://sbragagnolo/taskit';
  load.
```

If you want a specific release such as v0.1, you can load the associated tag as follows

```
Metacello new
  baseline: 'TaskIt';
  repository: 'github://sbragagnolo/taskit:v0.2';
  load.
```

Otherwise, if you want the latest development version, take a look at the development branchs and load the latest:

```
Metacello new
  baseline: 'TaskIt';
  repository: 'github://sbragagnolo/taskit:dev-0.3';
  load.
```

Adding it as a Metacello dependency

Add the following in your metacello configuration or baseline specifying the desired version:

```
spec
    baseline: 'TaskIt'
    with: [ spec repository: 'github://sbragagnolo/taskit-2:v0.2' ]
```

For developers

To develop TaskIt on github we use [iceberg](https://github.com/npasserini/iceberg). Just load iceberg and enter github's url to clone. Remember to switch to the desired development branch or create one on your own.

1.2 Changeslog

v0.2

Major Features

- Task Runners

- NewProcessTaskRunner - LocalProcessTaskRunner - Worker - WorkerPool

- Futures with callbacks
- Future combinators
- Future synchronous access
- Services

[Entire changes log](changeslog.md)

CHAPTER 2

Asynchronous Tasks

TaskIt's main abstraction are, as the name indicates it, tasks. A task is a unit of execution. By splitting the execution of a program in several tasks, TaskIt can run those tasks concurrently, synchronize their access to data, or order even help in ordering and synchronizing their execution.

2.1 First Example

Launching a task is as easy as sending the message 'schedule' to a block closure, as it is used in the following first code example:

```
[ 1 + 1 ] schedule.
```

>The selector name 'schedule' is chosen in purpose instead of others such as run, launch or execute. TaskIt promises you that a task will be "eventually" executed, but this is not necessarilly right away. In other words, a task is "scheduled" to be executed at some point in time in the future.

This first example is however useful to clarify the first two concepts but it remains too simple. We are schedulling a task that does nothing useful, and we cannot even observe it's result ("yet"). Let's explore some other code snippets that may help us understand what's going on.

The following code snippet will schedule a task that prints to the 'Transcript'. Just evaluating the expression below will make evident that the task is actually executed. However, a so simple task runs so fast that it's difficult to tell if it's actually running concurretly to our main process or not.

```
[ 'Happened' logCr ] schedule.
```

The real acid test is to schedule a long-running task. The following example schedules a task that waits for a second before writing to the transcript.

While normal synchronous code would block the main thread, you'll notice that this one does not.

```
[ 1 second wait.
'Waited' logCr ] schedule.
```

2.2 Schedule vs fork

You may be asking yourself what's the difference between the 'schedule' and 'fork'. From the examples above they seem to do the same but they do not. In a nutshell, to understand why 'schedule' means something different than 'fork', picture that using TaskIt two tasks may execute inside a same process, or in a pool of processes, while 'fork' creates a new process every time.

You will find a longer answer in the section below explaining "runners". In TaskIt, tasks are not directly scheduled in Pharo's global 'ProcessScheduler' object as usual 'Process' objects are. Instead, a task is scheduled in a task runner. It is the responsibility of the task runner to execute the task.

2.3 All valuables can be Tasks

We have been using so far block closures as tasks. Block closures are a handy way to create a task since they implictly capture the context: they have access to 'self' and other objects in the scope. However, blocks are not always the wisest choice for tasks. Indeed, when a block closure is created, it references the current 'context' with all the objects in it and its "sender contexts", being a potential source of memory leaks.

The good news is that TaskIt tasks can be represented by almost any object. A task, in TaskIt's domain are **valuable objects** i.e., objects that will do some computation when they receive the 'value' message. Actually, the message 'schedule' is just a syntax sugar for:

```
(TKTTask valuable: [ 1 logCr ]) schedule.
```

We can then create tasks using message sends or weak message sends:

```
TKTTask valuable: (WeakMessageSend receiver: Object new selector:
    #yourself).
TKTTask valuable: (MessageSend receiver: 1 selector: #+ arguments: {
    7 }).
```

Or even create our own task object:

```
Object subclass: #MyTask
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'MyPackage'.
```

```
MyTask >> value
    ^ 100 factorial
```

and use it as follows:

```
TKTTask valuable: MyTask new.
```

CHAPTER 3

Retrieving a Task's Result with Futures

In TaskIt we differentiate two different kind of tasks: some tasks are just "scheduled" for execution, they produce some side-effect and no result, some other tasks will produce (generally) a side-effect free value. When the result of a task is important for us, TaskIt provides us with a "future" object. A "future" is no other thing than an object that represents the future value of the task's execution. We can schedule a task with a future by using the 'future' message on a block closure, as follows.

```
aFuture := [ 2 + 2 ] future.
```

One way to see futures is as placeholders. When the task is finished, it deploys its result into the corresponding future. A future then provides access to its value, but since we cannot know "when" this value will be available, we cannot access it right away. Instead, futures provide an asynchronous way to access it's value by using "callbacks". A callback is an object that will be executed when the task execution is finished.

>In general terms, we do not want to **force** a future to retrieve his value in a synchronous way. >By doing so, we would be going back to the synchronous world, blocking a process' execution, and not exploiting concurrency. >Later sections will discuss about synchronous (blocking) retrieval of a future's value.

A future can provide two kind of results: either the task execution was a success or a failure. A success happens when the task completes in a normal way, while a failure happens when an uncatched exception is risen in the task. Because of these distinction, futures allow the subscription of two different callbacks using the methods 'onSuccessDo:' and 'onFailureDo:'.

In the example below, we create a future and subscribe to it a success callback. As soon as the task finishes, the value gets deployed in the future and the callback is called with it.

```
aFuture := [ 2 + 2 ] future.
aFuture onSuccessDo: [ :result | result logCr ].
```

We can also subscribe callbacks that handle a task's failure using the 'onFailureDo:' message. If an exception occurs and the task cannot finish its execution as expected, the corresponding exception will be passed as argument to the failure callback, as in the following example.

```
aFuture := [ Error signal ] future.
aFuture onFailureDo: [ :error | error sender method selector logCr ].
```

Futures accept more than one callback. When its associated task is finished, all its callbacks will be "scheduled" for execution. In other words, the only guarantee that callbacks give us is that they will be all eventually executed. However, the future itself cannot guarantee neither **when** will the callbacks be executed, nor **in which order**. The following example shows how we can subscribe several success callbacks for the same future.

```
future := [ 2 + 2 ] future.
future onSuccessDo: [ :v | FileStream stdout nextPutAll: v asString;
    cr ].
future onSuccessDo: [ :v | 'Finished' logCr ].
future onSuccessDo: [ :v | [ v factorial logCr ] schedule ].
future onFailureDo: [ :error | error logCr ].
```

Callbacks work wether the task is still running or already finished. If the task is running, callbacks are registered and wait for the completion of the task. If the task is already finished, the callback will be immediately scheduled with the already deployed value. See below a code examples that illustrates this: we first create a future and subscribes a callback before it is finished, then we wait for its completion and subscribe a second callback afterwards. Both callbacks are scheduled for execution.

```
future := [ 1 second wait. 2 + 2 ] future.
future onSuccessDo: [ :v | v logCr ].

2 seconds wait.
future onSuccessDo: [ :v | v logCr ].
```

CHAPTER 4

Task Runners: Controlling How Tasks are executed

So far we created and executed tasks without caring too much on the form they were executed. Indeed, we knew that they were run concurrently because they were non-blocking. We also said already that the difference between a ‘schedule‘ message and a ‘fork‘ message is that scheduled messages are run by a **task runner**.

A task runner is an object in charge of executing tasks ”eventually”. Indeed, the main API of a task runner is the ‘schedule:‘ message that allows us to tell the task runner to schedule a task.

```
aRunner schedule: [ 1 + 1 ]
```

A nice extension built on top of schedule is the ‘future:‘ message that allows us to schedule a task but obtain a future of its eventual execution.

```
future := aRunner future: [ 1 + 1 ]
```

Indeed, the messages ‘schedule‘ and ‘future‘ we have learnt before are only syntax-sugar extensions that call these respective ones on a default task runner. This section discusses several useful task runners already provided by TaskIt.

4.1 New Process Task Runner

A new process task runner, instance of ‘TKTNewProcessTaskRunner‘, is a task runner that runs each task in a new separate Pharo process.

```
aRunner := TKTNewProcessTaskRunner new.
aRunner schedule: [ 1 second wait. 'test' logCr ].
```

Moreover, since new processes are created to manage each task, scheduling two different tasks will be executed concurrently. For example, in the code snippet below, we schedule twice a task that printing the identity hash of the current process.

```
aRunner := TKTNewProcessTaskRunner new.
task := [ 10 timesRepeat: [ 10 milliSeconds wait.
        ('Hello from: ', Processor activeProcess identityHash
    asString) logCr ] ].
aRunner schedule: task.
aRunner schedule: task.
```

The generated output will look something like this:

]]] 'Hello from: 887632640' 'Hello from: 949846528' 'Hello from: 887632640' 'Hello from: 949846528' 'Hello from: 949846528' 'Hello from: 887632640' 'Hello from: 949846528' 'Hello from: 887632640' 'Hello from: 949846528' 'Hello from: 887632640' 'Hello from: 949846528' 'Hello from: 887632640' 'Hello from: 949846528' 'Hello from: 887632640' 'Hello from: 949846528' 'Hello from: 887632640' 'Hello from: 949846528' 'Hello from: 887632640']]]

First, you'll see that a different processes is being used to execute each task. Also, their execution is concurrent, as we can see the messages interleaved.

4.2 Local Process Task Runner

The local process runner, instance of 'TKTLocalProcessTaskRunner', is a task runner that executes a task in the caller process. In other words, this task runner does not run concurrently. Executing the following piece of code:

```
aRunner := TKTLocalProcessTaskRunner new.
future := aRunner schedule: [ 1 second wait ].
```

is equivalent to the following piece of code:

```
[ 1 second wait ] value.
```

or even:

```
1 second wait.
```

While this runner may seem a bit naive, it may also come in handy to control and debug task executions. Besides, the power of task runners is that they offer a polymorphic API to execute tasks.

4.3 The Worker Runner

The worker runner, instance of 'TKTWorker', is a task runner that uses a single process to execute tasks from a queue. The worker's single process removes one-by-one the tasks from the queue and executes them sequenceally. Then, schedulling a task into a worker means to add the task inside the queue.

A worker manages the life-cycle of its process and provides the messages 'start' and 'stop' to control when the worker thread will begin and end.

```
worker := TKTWorker new.
worker start.
worker schedule: [ 1 + 5 ].
worker stop.
```

By using workers, we can control the amount of alive processes and how tasks are distributed amongst them. For example, in the following example three tasks are executed sequenceally in a single separate process while still allowing us to use an asynchronous style of programming.

```
worker := TKTWorker new start.
future1 := worker future: [ 2 + 2 ].
future2 := worker future: [ 3 + 3 ].
future3 := worker future: [ 1 + 1 ].
```

Workers can be combined into "worker pools".

4.4 The Worker pool

A TaskIt worker pool is pool of worker runners, equivalent to a ThreadPool from other programming languages. Its main purpose is to provide several worker runners and decouple us from the management of threads/processes. A worker pool is a runner in the sense we use the 'schedule:' message to schedule tasks in it.

In TaskIt we count with two kind of worker pools:

- TKTWorkerPool - TKTCommonQueueWorkerPool

TKTWorkerPool

Internally, all runners inside a TKTWorkerPool pool have a task queue. This pool counts with a worker that is in charge of scheduling tasks into one of the available workers, taking in account the work load of each worper.

Different applications may have different concurrency needs, thus, TaskIt worker pools do not provide a default amount of workers. Before using a pool, we need to specify the maximum number of workers in the pool using

the 'poolMaxSize:' message. A worker pool will create new workers on demand.

```
pool := TKTWorkerPool new.
pool poolMaxSize: 5.
```

TaskIt worker pools use internally an extra worker to synchronize the access to its task queue. Because of this, a worker pool has to be manually started using the 'start' message before scheduled messages start to be executed.

```
pool := TKTWorkerPool new.
pool poolMaxSize: 5.
pool start.
pool schedule: [ 1 logCr ].
```

Once we are done with the worker pool, we can stop it by sending it the 'stop' message.

```
pool stop.
```

TKTCommonQueueWorkerPool

Internally, all runners inside a TKTCommonQueueWorkerPool pool share a common queue. This pool counts with a watchdog that is in charge of ensuring that all the workers are alive, and in charge of reducing the amount of workers when the load of work goes down.

Different applications may have different concurrency needs, thus, TaskIt worker pools do not provide a default amount of workers. Before using a pool, we need to specify the maximum number of workers in the pool using the 'poolMaxSize:' message. A worker pool will create new workers on demand.

```
pool := TKTCommonQueueWorkerPool new.
pool poolMaxSize: 5.
```

TaskIt worker pools use internally an extra worker to synchronize the access to its task queue. Because of this, a worker pool has to be manually started using the 'start' message before scheduled messages start to be executed.

```
pool := TKTCommonQueueWorkerPool new.
pool poolMaxSize: 5.
pool start.
pool schedule: [ 1 logCr ].
```

Once we are done with the worker pool, we can stop it by sending it the 'stop' message.

```
pool stop.
```

4.5 Managing Runner Exceptions

As we stated before, in TaskIt the result of a task can be interesting for us or not. In case we do not need a task's result, we will schedule it usign the 'schedule' or 'schedule:' messages. This is a kind of fire-and-forget way of executing tasks. On the other hand, if the result of a task execution interests us we can get a future on it using the 'future' and 'future:' messages. These two ways to execute tasks require different ways to handle exceptions during task execution.

First, when an exception occurs during a task execution that has an associated future, the exception is forwarded to the future. In the future we can subscribe a failure callback using the 'onFailureDo:' message to manage the exception accordingly.

However, on a fire-and-forget kind of scheduling, the execution and results of a task is not anymore under our control. If an exception happens in this case, it is the responsibility of the task runner to catch the exception and manage it gracefully. For this, each task runners is configured with an exception handler in charge of it. TaskIt exception handler classes are subclasses of the abstract 'TKTExceptionHandler' that defines a 'handleException:' method. Subclasses need to override the 'handleException:' method to define their own way to manage exceptions.

TaskIt provides by default a 'TKTDebuggerExceptionHandler', accessible from the configuration 'TKTConfiguration errorHandler' that will open a debugger on the raised exception. The 'handleException:' method is defined as follows:

```
handleException: anError
  anError debug
```

Changing a runner's exception handler can be done by sending it the 'exceptionHandler:' message, as follows:

```
aRunner exceptionHandler: TKTDebuggerExceptionHandler new.
```

4.6 Task Timeout

In TaskIt tasks can be optionally schedulled with a timeout. A task's timeout limits the execution of a task to a window of time. If the task tries to run longer than the specified time, the task is cancelled automatically. This behaviour is desirable because a long running tasks may be a hint towards a problem, or it can just affect the responsiveness of our application.

A task's timeout can be provided while scheduling a task in a runner, using the 'schedule:timeout:' message, asFollows:

```
aRunner schedule: [1 second wait] timeout: 50 milliSeconds.
```

If the task surpasses the timeout, the scheduled task will be cancelled with an exception.

A task's timeout must not be confused with a future's synchronous access timeout (explained below). The task timeout governs the task execution, while a future's timeout governs only the access to the future value. If a future times out while accessing its value, the task will continue its execution normally.

4.7 Where do tasks and callbacks run by default?

As we stated before, the messages #schedule and #future will schedule a task implicitly in a "default" task runner. To be more precise, it is not a default task runner but the **current task runner** that is used. In other words, task scheduling is context sensitive: if a task A is beign executed by a task runner R, new tasks scheduled by A are implicitly scheduled R. The only exception to this is when there is no such task runner, i.e., when the task is scheduled from, for example, a workspace. In that case a default task runner is chosen for scheduling.

> Note: In the current version of taskit (0.2) the default task runner is the global worker pool that can be explicitly accessed evaluating the following expression 'TKTConfiguration runner'.

Something similar happens with callbacks. Before we said that callbacks are eventually and concurrently executed. This happens because callbacks are scheduled as normal tasks after a task's execution. This scheduling follows the rules from above: callbacks will be scheduled in the task runner where it's task was executed.

CHAPTER 5

Advanced Futures

5.1 Future combinators

Futures are a nice asynchronous way to obtain the results of our eventually executed tasks. However, as we do not know when tasks will finish, processing that result will be another asynchronous task that needs to start as soon as the first one finishes. To simplify the task of future management, TaskIt futures come along with some combinators.

- **The 'collect:' combinator**

The 'collect:' combinator does, as its name says, the same than the collection's API: it transforms a result using a transformation.

```
future := [ 2 + 3 ] future.
(future collect: [ :number | number factorial ])
    onSuccessDo: [ :result | result logCr ].
```

The 'collect:' combinator returns a new future whose value will be the result of transforming the first future's value.

- **The 'select:' combinator**

The 'select:' combinator does, as its name says, the same than the collection's API: it filters a result satisfying a condition.

```
future := [ 2 + 3 ] future.
(future select: [ :number | number even ])
    onSuccessDo: [ :result | result logCr ];
    onFailureDo: [ :error | error logCr ].
```

The 'select:' combinator returns a new future whose result is the result of the first future if it satisfies the condition. Otherwise, its value will be a 'NotFound' exception.

- **The 'flatCollect:'combinator**

The 'flatCollect:' combinator is similar to the 'collect:' combinator, as it transforms the result of the first future using the given transformation block. However, 'flatCollect:' excepts as the result of its transformation block a future.

```
future := [ 2 + 3 ] future.
(future flatCollect: [ :number | [ number factorial ] future ])
    onSuccessDo: [ :result | result logCr ].
```

The 'flatCollect:' combinator returns a new future whose value will be the result the value of the future yielded by the transformation.

- **The 'zip:'combinator**

The 'zip:' combinator combines two futures into a single future that returns an array with both results.

```
future1 := [ 2 + 3 ] future.
future2 := [ 18 factorial ] future.
(future1 zip: future2)
    onSuccessDo: [ :result | result logCr ].
```

'zip:' works only on success: the resulting future will be a failure if any of the futures is also a failure.

- **The 'on:do:'combinator**

The 'on:do:' allows us to transform a future that fails with an exception into a future with a result.

```
future := [ Error signal ] future
    on: Error do: [ :error | 5 ].
future onSuccessDo: [ :result | result logCr ].
```

- **The 'fallbackTo:' combinator**

The 'fallbackTo:' combinator combines two futures in a way such that if the first future fails, it is the second one that will be taken into account.

```
failFuture := [ Error signal ] future.
successFuture := [ 1 + 1 ] future.
(failFuture fallbackTo: successFuture)
    onSuccessDo: [ :result | result logCr ].
```

In other words, 'fallbackTo:' produces a new future whose value is the first's future value if success, or it is the second future's value otherwise.

- **The 'firstCompleteOf:' combinator**

The 'firstCompleteOf:' combinator combines two futures resulting in a new future whose value is the value of the future that finishes first, wether it is a success or a failure.

```
failFuture := [ 1 second wait. Error signal ] future.
successFuture := [ 1 second wait. 1 + 1 ] future.
(failFuture firstCompleteOf: successFuture)
    onSuccessDo: [ :result | result logCr ];
    onFailureDo: [ :error | error logCr ].
```

In other words, 'fallbackTo:' produces a new future whose value is the first's future value if success, or it is the second future's value otherwise.

- **The 'andThen:' combinator**

The 'andThen:' combinator allows to chain several futures to a single future's value. All futures chained using the 'andThen:' combinator are guaranteed to be executed sequenceally (in contrast to normal callbacks), and all of them will receive as value the value of the first future (instead of the of of it's preceeding future).

```
([ 1 + 1 ] future
    andThen: [ :result | result logCr ])
    andThen: [ :result | FileStream stdout nextPutAll: result ].
```

This combinator is meant to enforce the order of execution of several actions, and this it is mostly for side-effect purposes where we want to guarantee such order.

5.2 Synchronous Access

Sometimes, although we do not recommend it, you will need or want to access the value of a task in a synchronous manner: that is, to wait for it. We do not recommend waiting for a task because of several reasons: - sometimes you do not know how much a task will last and therefore the waiting can kill's your application's responsiveness - also, it will block your current process until the waiting is finished - you come back to the synchronous world, killing completely the purpose of using TaskIt :)

However, since experienced users may still need this feature, TaskIt futures provide three different messages to access synchronously its result: 'isFinished', 'waitForCompletion:' and 'synchronizeTimeout:'.

'isFinished' is a testing method that we can use to test if the corresponding future is still finished or not. The following piece of code shows how we could implement an active wait on a future:

```
future := [1 second wait] future.
[future isFinished] whileFalse: [50 milliseconds wait].
```

An alternative version for this code that does not require an active wait is the message 'waitForCompletion:'. 'waitForCompletion:' expects a duration as argument that he will use as timeout. This method will block the execution until the task finishes or the timeout expires, whatever comes first.

If the task did not finish by the timeout, a ‘TKTTimeoutException‘ will be raised.

```
future := [1 second wait] future.
future waitForTimeout: 2 seconds.

future := [1 second wait] future.
[future waitForTimeout: 50 milliSeconds] on: TKTTimeoutException do:
    [ :error | error logCr ].
```

Finally, to retrieve the future’s result, futures understand the ‘synchronizeTimeout:‘ message, that receives a duration as argument as its timeout. If a successful value is available by the timeout, then the result is returned. If the task finished by the timeout with a failure, an ‘UnhandledError‘ exception is raised wrapping the original exception. Otherwise, if the task is not finished by the timeout a ‘TKTTimeoutException‘ is raised.

```
future := [1 second wait. 42] future.
(future synchronizeTimeout: 2 seconds) logCr.

future := [ self error ] future.
[ future synchronizeTimeout: 2 seconds ] on: Error do: [ :error |
    error logCr ].

future := [ 5 seconds wait ] future.
[ future synchronizeTimeout: 1 seconds ] on: TKTTimeoutException do:
    [ :error | error logCr ].
```

CHAPTER 6

Services

TaskIt furnishes a package implementing services. A service is a process that executes a task over and over again. You can think about a web server, or a database server that needs to be up and running and listening to new connections all the time.

Each TaskIt service may define a ‘setUp‘, a ‘tearDown‘ and a ‘stepService‘. ‘setUp‘ is run when a service is being started, ‘shutDown‘ is run when the service is being shutted down, and ‘stepService‘ is the main service action that will be executed repeateadly.

Creating a new service is as easy as creating a subclass of ‘TKTService‘. For example, let’s create a service that watches the existence of a file. If the file does not exists it will log it to the transcript. It will also log when the service starts and stops to the transcript.

```
TKTService subclass: #TKTFileWatcher
  instanceVariableNames: 'file'
  classVariableNames: ''
  package: 'TaskItServices-Tests'
```

Hooking on the service’s ‘setUp‘ and ‘tearDown‘ is as easy as overriding such methods:

```
TKTFileWatcher >> setUp
  super setUp.
  Transcript show: 'File watcher started'.

TKTFileWatcher >> tearDown
  super tearDown.
  Transcript show: 'File watcher finished'.
```

Finally, setting the watcher action is as easy as overriding the ‘stepService‘ message.

```
TKTFileWatcher >> stepService
  1 second wait.
  file asFileReference exists
    ifFalse: [ Transcript show: 'file does not exist!' ]
```

Making the service work requires yet an additional method: the service name. Each service should provide a unique name through the ‘name‘ method. TaskIt verifies that service names are unique and prevents the starting of two services with the same name.

```
TKTFileWatcher >> name
  ^ 'Watcher file: ', file asString
```

Once your service is finished, starting it is as easy as sending it the ‘start‘ message.

```
watcher := TKTFileWatcher new.
watcher file: 'temp.txt'.
watcher start.
```

Requesting the stop of a service is done by sending it the ‘stop‘ message. Know that sending the ‘stop‘ message will not stop the service right away. It will actually request it to stop, which will schedule the tear down of the service and kill its process after that.

```
watcher stop.
```

Stopping the process in an unsafe way is also supported by sending it the ‘kill‘ message. Killing a service will stop it right away, interrupting whatever task it was executing.

```
watcher kill.
```

6.1 Creating Services with Blocks

Additionally, TaskIt provides an alternative means to create services through blocks (or valuables actually) using ‘TKTParameterizableService‘. An alternative implementation of the file watcher could be done as follows.

```
service := TKTParameterizableService new.
service name: 'Generic watcher service'.
service onSetUpDo: [ Transcript show: 'File watcher started' ].
service onTearDownDo: [ Transcript show: 'File watcher finished' ].
service step: [
  'temp.txt' asFileReference exists
    ifFalse: [ Transcript show: 'file does not exist!' ] ].

service start.
```

CHAPTER 7

Process dashboard

TaskIt provides as well a far more interesting process dashboard, based on announcements.

For accessing this dashboard, go to World menu > TaskIt > Process dashboard, as showed in the following image.

7.1 TaskIt tab

The first one showing the processes launched by TaskIt looks like

The showed table has six fields.

- # ordinal number. Just for easing the reading.
- Name: The name of the task. If none name was given it generates a name based on the related objects.
- Sending: The selector of the method that executes the task. If the task is based on a block, it will be #value.
- To: The receiver of the message that executes the task.
- With: The arguments of the message send that executes the task
- State: [Running|NotRunning].

Some of those fields have attached some contextual menu.

Do right-click on top of the name of the process for interacting with the process

The options given are

- Inspect the process: It opens an inspector showing the related TaskIt process.
- Suspend|Resume the process: It pause|resume the selected process.
- Cancel the process: It cancel the process execution.

Do right-click on top of the message selector for interacting with selector|method

The options given are

- Method. This option browses the method executed by the task.
- Implementors. This option browses all the implementors of this selector.

Finally, do right-click on top of the receiver for interacting with it

The option given is

- Inspect receiver. What does exactly that. Inspects the receiver of the message.

7.2 System tab

Finally, to allow the user to use just one interface. There is a second tab that shows the processes that were not spawnend by TaskIt.

7.3 Based on announcements

The TaskIt browser is based on announcements. This fact allows the interface to be dynamic, having allways fresh information, without needing a pulling process, as in the native process browser.

[image-menu]: https://github.com/sbragagnolo/taskit/raw/dev-0.2/images/AccessMenu.png ”MenuTaskit” [image-main]: https://github.com/sbragagnolo/taskit/raw/dev-0.2/images/FirstScreen.png ”Main tab” [image-process]: https://github.com/sbragagnolo/tas 0.2/images/ProcessMenu.png ”Process menu” [image-receiver]: https://github.com/sbragagn 0.2/images/ReceiverInspector.png ”Receiver menu” [image-selector]: https://github.com/sbr 0.2/images/SelectorInspection.png ”Selector (method) menu” [image-system]: https://github.com/sbragagnolo/taskit/raw/dev-0.2/images/SystemScreen.png ”System process tab”

CHAPTER 8

Debugger

TaskIt comes with a debugger extension for Pharo that can be installed by loading the 'debug' group of the baseline (the debugger is not loaded by any other group):

```
Metacello new
  baseline: 'TaskIt';
  repository: 'github://sbragagnolo/taskit';
  load: 'debug'.
```

After installation the TaskIt debugger extension will automatically be available to processes that are associated with a task or future. You can manually enable or disable the debugger extension by evaluating 'TKTDebugger enable.' or 'TKTDebugger disable.'.

The TaskIt debugger shows an augmented stack, in which the process that represents the task or future is at the top and the process that created the task or future is at the bottom (recursively for tasks and futures created from other tasks and futures). The following visualisation shows one future process (top) with frames '1' and '2' and the corresponding creator process (frames '3' and '4'):

```
-------------------
|     frame 1     |
-------------------
|     frame 2     |
-------------------
-------------------
|     frame 3     |
-------------------
|     frame 4     |
-------------------
```

The implementation and conception of this debugger extension can be found in Max Leske's Master's thesis entitled ["Improving live debugging of concurrent threads"](http://scg.unibe.ch/scgbib?query=Lesk16a&display=abstract).

CHAPTER 9

Configuration

TaskIt comes with a configuration based on the DynamicVariables (process local variables).

```
profiles
  ^ {(#profile -> #development).
  (#development
    -> [ {(#initialize
        -> [ TKTDebugger enable.
          self class runner start ]).
      (#runner -> TKTCommonQueueWorkerPool createDefault).
      (#poolWorkerProcess -> TKTDebuggWorkerProcess).
      (#process -> TKTRawProcess).
      (#errorHandler -> TKTDebuggerExceptionHandler).
      (#processProvider -> TKTTaskItProcessProvider new).
      (#serviceManager -> TKTServiceManager new)} asDictionary ]).
  (#production
    -> [ {(#initialize
        -> [ TKTDebugger disable.
          self class runner start ]).
      (#runner -> TKTCommonQueueWorkerPool createDefault).
      (#poolWorkerProcess -> TKTWorkerProcess).
      (#process -> TKTRawProcess).
      (#errorHandler -> TKTExceptionHandler).
      (#processProvider -> TKTPharoProcessProvider new).
      (#serviceManager -> TKTServiceManager new)} asDictionary ]).
  (#test
    -> [ {(#initialize
        -> [ TKTDebugger disable.
          self class runner start ]).
      (#runner -> TKTCommonQueueWorkerPool createDefault).
      (#poolWorkerProcess -> TKTWorkerProcess).
```

```
      (#process -> TKTRawProcess).
      (#errorHandler -> TKTExceptionHandler).
      (#processProvider -> TKTTaskItProcessProvider new).
      (#serviceManager -> TKTServiceManager new)} asDictionary ])}
    asDictionary.
```

This configuration object contains many profiles. The profile loaded by default is the specified at the first dictionary key '#profile'

For changing the profile, you just need to send the message profile with the required profile as parameter:

'TKTConfiguration profile: #production.'

Since this is a process variable, we can do also some magic for executing specfiic code with specific configurations as:

'TKTConfiguration profile: #test during: ["run tests"].'

Or also do smaller changes as:

'TKTConfiguration errorHandler: #MyHandler during: ["do something"] .'

In this last case the the block will be executed with the actual profile, but changing the errorHandler by default.

CHAPTER 10

Future versions

- ActIt: an actor/active object implementation on top of taskit
- Service manager
- Future inspection. (Or processing network combination)
- Autoorganized worker pool
- Inter-innerprocess debugging