Beacon: a slim announcement-based logging engine for Pharo

Logging is a pervasive analysis tool, but most logging systems focus on text. There is a historical reason for this, but we can certainly do better in this century. Particularly in Pharo, where everything is an object, it should follow that logs should be made of objects and that scrolling endlessly through large text files should make room for more sensible engineering approaches.

So, how would an object-oriented logging system look like?

The SystemLogger proposes a Pharo solution that focuses on objects. The central concept is the Log object which represents a single logging event. The event can be specialized via subclassing with various types of events. Similar to Toothpick, it features a central object that collects log objects, and has several concrete loggers that consume the log objects through various bindings such as the standard output or a database. The size is also rather tiny, the core containing some 535 lines of code.

However, there are still some lesser objects involved in the engine. Like in other logging frameworks, you can specify both levels of severity and tags that can be used for filtering. However, just like in other frameworks, these levels are numbers (even if encapsulated), and the tags are just strings.

At the same time, Sven described his thoughts about how a simple logging system could look like in Pharo. He talks about using event mechanisms that already

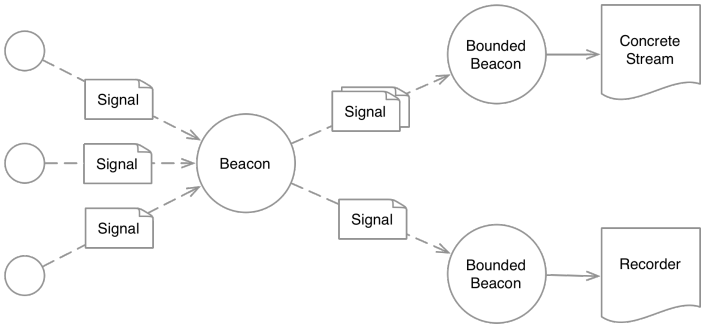
exists in Pharo such as: Notifications and Announcements. Indeed, this makes plenty of sense. Of these two, I like Announcements more because they are meant to announce events from objects to objects.

So, I set myself to implement an engine based solely on Announcements. The goal was to get to pragmatic solution that should be usable in practice, yet be elegantly slim at the same time. After several iterations and a couple of feedback sessions with Sven and Andrei, the Beacon project was born:

```
Gofer new
     smalltalkhubUser: 'girba' project: 'Beacon';
     configuration;
     loadDevelopment
```

I know. Yet another logging framework! Just bare with me for a while. At the very least, it's an exercise. And just to entice you: it has 200 lines of code. Let's go.

The basic structure is similar to the SystemLogger. Objects log `BeaconSignals` that are captured by a receiving announcer. To make things easy, there exits a global announcer in the `Beacon` class. The beacon announcer does not do anything by itself. To actually produce an effect either on some sort of a stream like the Transcript, or on some sort of a recording support, it needs a dedicated `BoundedBeacon` .



Take a look at some simple examples. In the first one, we record signal objects that are logged in between the starting and stopping of the recorder:

```
RecordingBeacon start.
CurrentStackSignal log.
RecordingBeacon stop.
```

spike (20)

story (31)

strategic (1)

tooling (37)

visualization (1)

In the the second one, we simply output to the transcript the textual representation of the signals logged within the scope of a

```
TranscriptBeacon runDuring: [
     StringSignal log: 'This is a message'. ]
```

Basic signals

Beacon works with Announcements. However, for logging purposes, we typically need a timestamp. To this end, `BeaconSignal` offers a handy Announcement that stores the current timestamp during the initialization.

The code also ships with three concrete signals for convenience:

- `StringSignal` is used for simple string messages. This is used mostly for example purposes, and it is actually not meant to be used much in practice. Why not? Because it's a string.
- `WrapperSignal` is useful to log an arbitrary object without necessarily creating a class for it. Again, it is not meant to be used extensively, but can prove useful for quick prototypes.
- `CurrentStackSignal` is a utility useful for capturing the current stack as objects (how cool is that?!).

Other than that, each application has to define its own, much like how it is supposed to define Exceptions and Announcements.

O yes. Another thing to remember is that each `Signal` is triggered using a `log` instance side message. That is it.

Basic bindings

To achieve anything useful, we need a `BoundedBeacon` to bind the general `Beacon` to a concrete medium, such as a stream. The core engine comes with a couple of concrete bounded beacons.

As seen in the previous examples, we have seen a

`RecordingBeacon` that records the signals in an ordered collection, and a `TranscriptBeacon` that outputs the textual representation to the Transcript. Similarly, we can also output a textual representation to any custom string stream:

```
String streamContents: [ :stream |
     (CustomStringStreamBeacon with: stream)
          runDuring: [
               StringSignal log: 'This is a messag
e' ] ]
```

An example of a custom binding to Fuel

A funny way of binding can be found in the `FuelBeacon` package from the same repository:

```
Gofer new
     smalltalkhubUser: 'girba' project: 'Beacon';
     package: 'FuelBeacon';
     load
```

As signals are objects, and as Fuel can store any such objects, it follows that we can simply combine the two and store a fuelized version of each signal.

```
FuelBeacon new runDuring: [
     StringSignal log: 'This is a message'.
     StringSignal log: 'This is another message'.
     StackSignal log ].
```

Each signal gets stored as a fuel file in a dedicated fuelbeacon folder. This is not particularly efficient and it should be used with caution because of two reasons:

1. the creation of each file is not particularly fast, and
2. serializing an object with Fuel will take with it all referenced objects, so each file size depends on the shape of your Signal. Thus, this can influence both the size and the speed of the logging.

Nevertheless, it does provide an example of how to create a custom `BoundedBeacon`.

No fancy text formatting

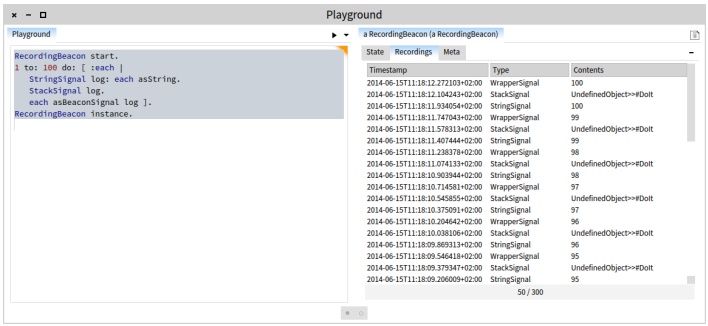
One common logging concept that did not make it in the engine is that of a formatter. This is typically an essential piece in text focused logging engines.

Perhaps there are cases in which it can still be useful for having special formatting on top of Beacon. If needed, most common of the formatting use cases can be achieved through concrete bounded beacons. For example, the `StringStreamBeacon` uses polymorphism to delegate the printing to the `Signal` hierarchy. This is enough.

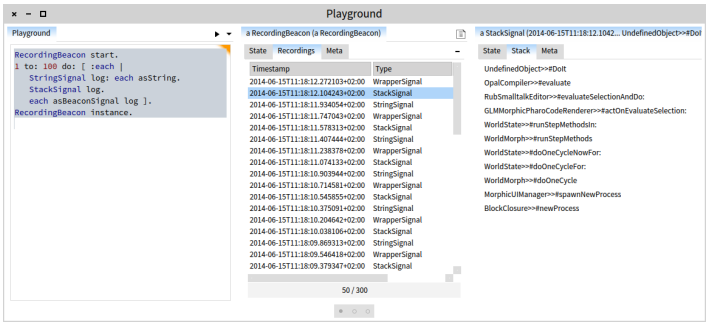
Given that serialization should focus on objects, formatting text is neither essential nor general enough to warrant more prominent modeling. Once we have objects, we can approach presenting them through smarter tools like it can be seen below.

Inspecting beacon objects in the GTInspector

The Beacon comes with a dedicated set of extensions for GTInspector. In the picture below, you can see an inspector on an instance of `RecordingBeacon`. The list updates every time a new signal is recorded. By default, the latest one is shown on top, but the table is sortable. As you can see, in this example, we simulated 100k + 1 signals.



The `CurrentStackSignal` also comes with a dedicated presentation that shows the stack.



Filtering signal objects

Logging can quickly spawn a ton of data. Filtering is essential in this situation.

But, Beacon has no tags and no levels. Why not? Because these filtering methods pose a rather arbitrary constraint as they require us to know in advance at what level we want a certain logging event to be. They might work well in some situations, but in others they are more of a hindrance as often when using logging for debugging purposes, we mainly want to see only the events we nominally want and not others.

A approach for this problem is to take advantage of objects. Indeed, the Announcement framework allow us to register our interest for a set of announcement types of our choice. For example, the snippet below:

```
RecordingBeacon new
     runFor: StringSignal, StackSignal
     during: [
          StringSignal log: 'This should be record
ed.'.
          StackSignal log.
          DummySignal new log ]
```

will only log the first two signals, but not the dummy one. For each application, we can define various combinations of arbitrary signals that should be logged together without having an extra need of levels and tags.

Of course, applications can still create their own specialized `BoundedBeacons` and introduce another level of filtering if it makes sense within that application. However, please note that the default filtering

mechanism offers developers an incentive to design fine grained logging classes that can be easily picked up at a later time. That can only be for the better.

Filtering signal data

Another filtering issue is related to the data associated with a signal object. Let's take a concrete example. The goal of `StackSignal` is to store the entire current stack. This in turn can lead to including an extensive object graph that can produce large files when stored using Fuel.

To limit the size, we use two mechanisms. First, the `StackSignal` does not store the actual context objects. Instead it only stores the references (i.e., Ring definitions) to the methods being present in the stack. Indeed, this is not quite a stack, but it is a reasonable approximation for the point of view of logging. A benefit of doing so, is that the logged objects become constant and thus they are better suited for longer term inspection (see also the thoughts of Sven).

Second, the FuelBeacon itself ensures that only a limited amount of stack items are being serialized. This decision is essentially dependent both on the `FuelBeacon` serialization strategy and on the `StackSignal`. To handle it, we need a double dispatch. For example, in our case, the `FuelBeacon` delegates to the `Signal` hierarchy the job of actual serialization via extensions:

```
Signal>>serializeInDirectory: aDirectoryReference
     ...
     FLSerializer
          serialize: self objectToSerialize
          toFileNamed: (aDirectoryReference / oneL
inerAsFileName , 'fuel') fullName
```

And the `StackSignal` can specify the acceptable copy to be serialized:

```
StackSignal>>objectToSerialize
     | trimmedCopy |
     trimmedCopy := self copy.
```

```
	trimmedCopy
		instVarNamed: #stack
		put: (self stack first: 100).
	^ trimmedCopy
```

This is but an example of a pattern, and the solution described above can be used in other cases as well.

Beyond global bindings

Up to now, we have mostly looked at global signals that are captured by a global `Beacon` and passed to other global `BoundedBeacon` objects. These globals make it cheap to add a new log entry and have it captured in a central place. Yet, the engine does not limit the usage to these globals. The last example from above shows that we can equally well collect the results through an instance of a `BoundedBeacon` (e.g., `RecordingBeacon new`).

This becomes interesting when we want to debug selectively only a dedicated scenario without having to dig through the entire log that might contain many irrelevant details. For example, while working with the Rubric text morph, we noticed that when the syntax highlighting is activated, it gets slow. At the same time, the same syntax highlighting engine does not cause a problem in the commonly used PluggableTextMorph. Thus, we wanted to understand what execution paths lead to the `SHTextStyler>>style:`. To this end, we added:

```
SHTextStyler>>style:
	StackSignal log.
	...
```

And, after selecting a Rubric morph, we simulated the troubled scenario like this:

```
RecordingBeacon new runDuring:
	[ ActiveHand simulateKeyStrokes: 'aa.' ]
```

Inspecting the resulting RecordingBeacon revealed that indeed, the style method is being used at every character change, while in the case of the

PluggableTextMorph, it is triggered much more seldom. For this type of analysis it is critical to be able to collect only the signals we care about and only for the duration we care about.

Logging any announcements from any announcer

Not only can we use multiple instances for bindings, but we do not necessarily need the `Beacon` either. By using Announcements as the transmission engine, we can also log the transmissions of announcements from any announcer. To achieve this, we simply have to point a bounded beacon to the announcer. For example:

```
announcer := Announcer new.
b := RecordingBeacon for: announcer.
b
     runFor: Announcement
     during: [ announcer announce: Announcement ne
w ].
```

This feature is particularly interesting given that the Announcements framework is becoming the primary means of communication between decoupled objects in Pharo. As announcements tend to be hard to debug, logging the activity of an announcer inexpensively can save a lot of trouble.

Closing remarks

I was surprised by how little code I needed to write the engine. At first sight, the implementation can appear too simple. Indeed, it is based on a small amount of concepts. However, I believe that the examples above show that by combining these concepts, we can handle a wide range of classic requirements we might have for a logging system.

There is certainly room for improvements. One direction of development is to create more bindings to support various backends. Another direction is to support filtering based on the instance of a signal, and not only based on its type. Yet another one, is to work on how to

effectively trim the stored data.

Some things in Beacon are happening in a radically different manner than what is considered the norm, but I think this is what makes it interesting. Filtering is simpler and more flexible without imposing restrictions deep in the framework. Textual formatters do not exist at all. Add to that the announcement debugging option and we get a different bread of an engine that is more versatile at less cost.

Posted by Tudor Girba at 15 June 2014, 2:48 pm with tags assessment, moose, pharo, analysis, tooling link

Home
Guide
Stories
Courses
Services
Resources
About
Blog

You are here:
Home /
Blog /
Beacon: a slim announcement-based logging engine for Pharo /

Contact | Follow @humaneA on twitter | Subscribe to RSS

This site is powered by Pier | Login