Chapter 1

Fuel

Fuel is an open-source general-purpose object serialization framework.

1.1 General Information

Goals

Concrete

We don't aspire to have a dialect-interchange format. This enables us to serialize special objects like contexts, block closures, exceptions, compiled methods and classes. Although there are ports to other dialects, Fuel development is Pharo-centric.

Flexible

Depending on the context, there could be multiple ways of serializing the same object. For example, a class can be considered either a global or a regular object. In the former case, it will be encoded just its name; in the latter case, the class will be encoded in detail, with its method dictionary, etc.

Fast

We worry about to have the best performance. We developed a complete benchmark suite to help analyse the performance with diverse sample sets, as well as compare against other serializers. Our pickling algorithm allows outstanding materialization performance, as well as very good serialization performance too.

Object-Oriented Design

From the beginning it was a constraint to have a good object-oriented design and to do not need any special support from the VM. In addition, Fuel has a complete test suite, with a high coverage. We also worry about writing comments on classes and methods.

Features

- Is a fast, concrete, general-purpose and flexible binary serializer.
- Object-Oriented design.
- No special VM-support needed.
- Modular (clear division in packages).
- Can serialize/materialize not only plain objects but also classes, traits, methods, closures, contexts, packages, etc.
- Support for global references.
- Very customizable: ignore certain instance variables, substitute objects by others, pre and post serialization and materialization actions, etc.
- Supports class rename and class reshape.
- Good test coverage (almost 600 unit tests).
- Large suite of benchmarks.

Development

Some links

- Issue tracker
- Source repository
- Continuous integration

Collaborators

- Martin Dias - tinchodias (at) gmail (dot) com (Developer)
- Mariano Martinez Peck - marianopeck (at) gmail (dot) com (Developer)

- Max Leske (Developer)
- Pavel Krivanek (Developer)
- Tristan Bourgois (Past Developer)
- Stephane Ducasse (Promotor and financer)

1.2 Installation

Fuel 1.9 is already loaded in Pharo 2.0. The **default packages** work out of the box in Pharo 1.1.1, 1.1.2, 1.2, 1.3, 1.4, 2.0 and Squeak 4.1, 4.2, 4.3, 4.4.

Default

For most users should be enough to install just the default packages:

```
Gofer it
   squeaksource: 'MetacelloRepository';
   package: 'ConfigurationOfFuel';
   load.
((Smalltalk at: #ConfigurationOfFuel) project version: '1.9')
   load.
```

Basic demo

Open the Transcript and evaluate code below in a Workspace.

```
| serializedArray materializedArray |
serializedArray :=
   Array
      with: (Set with: 42)
      with: Transcript
      with: [:aString | Transcript show: aString; cr ].

"Store (serialize)"
FLSerializer serialize: serializedArray toFileNamed: 'demo.fl'.

"Load (materialize)"
materializedArray := FLMaterializer materializeFromFileNamed: 'demo.fl'.

Transcript
   show: 'The sets are equal: ';
   show: serializedArray first = materializedArray first;
   cr;
```

```
    show: 'But not the same: ';
    show: serializedArray first ∼∼ materializedArray first;
    cr;
    show: 'The global value Transcript is the same: ';
    show: serializedArray second == materializedArray second;
    cr.
materializedArray third
    value: 'The materialized block closure can be properly evaluated.'.
```

1.3 Getting Started

Basic examples

We give some class-side messages to facilitate the more common uses of serialization and materialization. The next example writes and reads from file:

```
FLSerializer serialize: 'stringToSerialize' toFileNamed: 'demo.fuel'.
materializedString := FLMaterializer materializeFromFileNamed: 'demo.fuel'.
```

We also provide messages for storing into a ByteArray. This can be fine for users of a NoSQL database:

```
anArray := FLSerializer serializeToByteArray: 'stringToSerialize'.
materializedString := FLMaterializer materializeFromByteArray: anArray.
```

FileStream

In this example we work with files. Note that we set the file in binary mode:

```
FileStream forceNewFileNamed: 'demo.fuel' do: [:aStream |
    FLSerializer newDefault
        serialize: 'stringToSerialize'
        on: aStream binary].

FileStream oldFileNamed: 'demo.fuel' do: [:aStream |
    materializedString := (FLMaterializer newDefault
        materializeFrom: aStream binary) root].
```

Note also that we are no longer using the class-side messages of previous examples. Now, for both FLSerializer and FLMaterializer, we first create instances with #newDefault to then perform the desired operations. As we will see in next example, creating the instances brings more possibilities.

Compressing

Of course, you could use stream compressors provided by the system. However, we have detected some errors serializing WideStrings. An example of use:

```
FileStream forceNewFileNamed: 'number.fuel.zip' do: [:aFileStream |
   |gzip|
   aFileStream binary.
   gzip := GZipWriteStream on: aFileStream.
   FLSerializer newDefault serialize: 123 on: gzip.
   gzip close. ].
```

```
FileStream oldFileNamed: 'number.fuel.zip' do: [:aFileStream |
   |gzip|
   aFileStream binary.
   gzip := GZipReadStream on: aFileStream.
   materialization := FLMaterializer newDefault
      materializeFrom: gzip.
   gzip close. ].
```

Showing a progress bar

Sometimes it is nice to see progress updates on screen. Use #showProgress in this cases.

```
FileStream forceNewFileNamed: 'numbers.fuel' do: [:aStream |
   FLSerializer newDefault
      showProgress;
      serialize: (1 to: 200000) asArray
      on: aStream binary ].

FileStream oldFileNamed: 'numbers.fuel' do: [:aStream |
   FLMaterializer newDefault
      showProgress;
      materializeFrom: aStream binary ].
```

Package FuelProgressUpdate must be installed. You can use:

```
(ConfigurationOfFuel project version: '1.9')
   load: 'FuelProgressUpdate'.
```

1.4 Managing Globals

Let us assume a CompiledMethod is referenced from the graph to serialize. Sometimes we may be interested in storing just the selector and name of

the class, because we know it will be present when materializing the graph. However, sometimes we want to really store the method with full detail.

This means that given an object graph, there is not an unique way of serializing it. Fuel offers dynamic and static mechanisms to customize this.

Default globals

By default, Fuel considers following objects as globals, i.e. will store just its name:

- nil, true, false, and Smalltalk globals.
- Any Class, Trait, Metaclass or ClassTrait.
- Any CompiledMethod (except when either it #isInstalled not or #isDoIt, for example, the code is evaluated from Workspace).
- Some well-known global variables: Smalltalk SourceFiles Transcript Undeclared Display TextConstants ActiveWorld ActiveHand ActiveEvent Sensor Processor ImageImports SystemOrganization World.

Custom globals are duplicated

In this following code snippet we show that by default the global value is not serialized as a global, and so it is duplicated on materialization.

```
"Define a global variable named #SomeGlobal."
SomeGlobal := Set new.

"Serialize and materialize the value of #SomeGlobal."
FLSerializer
  serialize: SomeGlobal
  toFileNamed: 'g.fuel'.

"The materialized object *is not* the same as the global instance."
[ (FLMaterializer materializeFromFileNamed: 'g.fuel') ~~ SomeGlobal ] assert.
```

But...

How to avoid duplication

Instead, in the code below #considerGlobal: is used to specify that it should be stored as global.

```
| aSerializer |

"Define a global variable named #SomeGlobal."
SomeGlobal := Set new.

aSerializer := FLSerializer newDefault.

"Tell the serializer to consider #SomeGlobal as global."
aSerializer analyzer considerGlobal: #SomeGlobal.

aSerializer
  serialize: SomeGlobal
  toFileNamed: 'g.fuel'.

"In this case, the materialized object *is* the same as the global instance."
[ (FLMaterializer materializeFromFileNamed: 'g.fuel') == SomeGlobal ] assert.
```

This feature is tested in tests-globals protocol of FLBasicSerializationTest as well in FLGlobalEnvironmentTest.

Changing the environment

It is possible to specify where the global will be looked-up during materialization. The method #globalEnvironment: exists for that purpose, as the following example shows.

```
| aSerializer aMaterializer anEnvironment |

"Define a global variable named #SomeGlobal."
SomeGlobal := Set new.

"Tell the serializer to consider #SomeGlobal as global."
aSerializer := FLSerializer newDefault.
aSerializer analyzer considerGlobal: #SomeGlobal.
aSerializer
  serialize: SomeGlobal
  toFileNamed: 'g.fuel'.

"Override value for #SomeGlobal."
anEnvironment := Dictionary newFrom: Smalltalk globals.
anEnvironment at: #SomeGlobal put: {42}.

"In this case, the materialized object *is the same* as the global instance."
FileStream oldFileNamed: 'g.fuel' do: [ :aStream |
  aStream binary.
  aMaterializer := FLMaterializer newDefault.

  "Set the environment"
```

```
aMaterializer globalEnvironment: anEnvironment.

[ (aMaterializer materializeFrom: aStream) root = {42} ] assert ]
```

This feature is tested in the class FLGlobalEnvironmentTest. The global environment can be setted also for serialization (not only materialization), but we don't include an example for that case.

1.5 Customizing the Graph

Ignoring Instance Variables

It can happen that instance variables should never be serialized. A practical way to do this is overriding the hook method #fuelIgnoredInstanceVariableNames.

Let's say we have the class User and we do not want to serialize the instance variables 'acumulatedLogins' and 'applications'. So we implement:

```
User class >> fuelIgnoredInstanceVariableNames
  ^#('acumulatedLogins' 'applications')
```

When materialized, such instance variables will be nil. If you want to re-initialize and set values to those instance variables, you can use #fuelAfterMaterialization for that.

Be aware that in case of renaming those instance variables, you should rename that method as well. Notice also that the method #fuelIgnoredInstanceVariableNames is implemented at class side. This means that **all** instances of such class will ignore the defined instances variables.

We test this feature in FLIgnoredVariablesTest.

In StOMP serializer this same hook is called #stompTransientInstVarNames and in SIXX it is #sixxIgnorableInstVarNames.

Post-Materialization Action

The method #fuelAfterMaterialization let us execute something once an object has been materialized. For example, let's say we would like to set back the instance variable 'acumulatedLogins' during materialization. Hence, we can implement:

```
User >> fuelAfterMaterialization
 acumulatedLogins := 0.
```

Substitution on Serialization

Sometimes you may want to serialize something different than the original object, without altering them.

Dynamic way

You can establish a pluggable substitution to a particular serialization.

Let's illustrate with an example, where your graph includes a Stream and you want to serialize nil instead.

```
objectToSerialize := Array with: 'hello' with: '' writeStream.

FileStream forceNewFileNamed: 'demo.fuel' do: [ :aStream |
  aSerializer := FLSerializer newDefault.
  aSerializer analyzer
    when: [ :o | o isStream ]
    substituteBy: [ :o | nil ].
  aSerializer
    serialize: objectToSerialize
    on: aStream binary ].
```

So, when loading you will get #('hello' nil), without any instance of a stream.

You can find this code in FLUserGuidesTest>>testPluggableSubstitution.

Static way

You have to override #fuelAccept: in the class of the object to be substituted. Fuel visits each object in the graph by sending this message, to determine how to trace and serialize it. Note that this will affect every serialization, in contrast with the 'dynamic way' we explained above; but it could be much faster.

As an example, imagine we want to replace an object directly with nil. In other words, we want to make a whole object transient, say CachedResult. For that, we should implement:

```
CachedResult >> fuelAccept: aGeneralMapper
  ^ aGeneralMapper visitSubstitution: self by: nil
```

As another example, we have a Proxy class and when serializing we want to serialize its target instead of the proxy. So we implement:

```
Proxy >> fuelAccept: aGeneralMapper
  ^ aGeneralMapper visitSubstitution: self by: target
```

Notice that #fuelAccept: is the same as the previous example. The last example is when an object needs to change the value of its instance variables. Say we have again the class User and we want to nil the instance variable 'history' when its size is greater than 100.

```
User >> fuelAccept: aGeneralMapper
  ^self history size > 100
    ifTrue: [
      aGeneralMapper
        visitSubstitution: self
        by: (self copy history: Array new) ].
    ifFalse: [ super fuelAccept: aGeneralMapper ]
```

Note we are substituting the original user by another instance of User, which Fuel will visit with the same #fuelAccept: method. We could easily fall in an infinite sequence of substitutions if we don't take care. To avoid this problem, it is useful #visitSubstitution:by:onRecursionDo:, where you define an alternative mapping for the case of mapping an object which is already a substitute of another one:

```
User >> fuelAccept: aGeneralMapper
  aGeneralMapper
    visitSubstitution: self
    by: (self copy history: #())
    onRecursionDo: [ super fuelAccept: aGeneralMapper ]
```

In the case, the substitute user (i.e. the one with the empty history) is will be visited via its super implementation.

You can see tests for this functionality at FLHookedSubstitutionTest.

Substitution on Materialization

Global Sends

Suppose we have a special instance of User that represents the admin user, and it is an unique instance in the image. In case the admin user is referenced in our graph, we want to treat that object as a global. We can do that in this way:

```
User >> fuelAccept: aGeneralMapper
  ^self == User admin
    ifTrue: [
      aGeneralMapper
        visitGlobalSend: self
        name: #User
        selector: #admin ]
    ifFalse: [ super fuelAccept: aGeneralMapper ]
```

So what will happen is that during serialization, the admin user won't be completly serialized (with all its intance variables) but instead its global name and selector are stored. Then, at materialization time, Fuel will send #admin to the class User, and use what that answers as the admin user of the materialized graph.

We test this feature in FLGlobalSendSerializationTest.

Hooking instance creation

Fuel provides two hook methods to customise how instances are created: #fuelNew and #fuelNew:.

For (regular) fixed objects, the method #fuelNew is defined in Behavior as:

```
fuelNew
  ^ self basicNew
```

But we can override it to our needs, for example:

```
fuelNew
  ^ self uniqueInstance
```

This similarly applies to variable objects through the method #fuelNew:, which by default answers #basicNew:.

We test this feature in FLSingletonTest.

Not Serializable Objects

You may want to be sure that some objects are not serialized. For this case we provide #visitNotSerializable:, which in next example forbids serialization of any instance of MyNotSerializableObject.

```
MyNotSerializableObject >> fuelAccept: aGeneralMapper
  aGeneralMapper visitNotSerializable: self
```

We test this feature in FLBasicSerializationTest>>testNotSerializableObject.

1.6 Errors

We provide a hierarchy of errors which allow to clearly identify the problem if something went wrong:

- FLError
  - FLSerializationError

    - FLNotSerializable
    - FLObjectNotFound
    - FLObsolete
  - FLMaterializationError
    - FLBadSignature
    - FLBadVersion
    - FLClassNotFound
    - FLGlobalNotFound
    - FLMethodChanged
    - FLMethodNotFound

As most classes of Fuel, they have class comments that give an idea their meanings:

FLError

I represent an error produced during Fuel operation.

FLSerializationError

I represent an error happened during serialization.

FLNotSerializable

I represent an error which may happen while tracing in the graph an object that is forbidden of being serialized.

FLObjectNotFound

I represent an error which may happen during serialization, when trying to encode on the stream a reference to an object that should be encoded before, but it is not. This usually happens when the graph changes during serialization. Another possible cause is a bug in the analysis step of serialization.

FLObsolete

I am an error produced during serialization, signaled when trying to serialize an obsolete class as global. It is a prevention, because such class is likely to be absent during materialization.

FLMaterializationError

I represent an error happened during materialization.

FLBadSignature

I represent an error produced during materialization when the serialized signature doesn't match the materializer's signature (accessible via FLMaterializer>>signature). A signature is a byte prefix that should prefix a well-serialized stream.

FLBadVersion

I represent an error produced during materialization when the serialized version doesn't match the materializer's version (accessible via FLMaterializer>>version). A version is encoded in 16 bits and is encoded heading the serialized stream, after the signature.

FLClassNotFound

I represent an error produced during materialization when a serialized class or trait name doesn't exist.

FLGlobalNotFound

I represent an error produced during materialization when a serialized global name doesn't exist (at Smalltalk globals).

FLMethodChanged

I represent an error produced during materialization when is detected a change in the bytecodes of a method serialized as global. This error was born when testing the materialization of a BlockClosure defined in a method that changed. The test produced a VM crash.

FLMethodNotFound

I represent an error produced during materialization when a serialized method in a class or trait name doesn't exist (at Smalltalk globals).

1.7 Object Migration

Often, we need to load objects whose class has changed since it was saved. In this document how to load them in the different cases. Figure 1.1 is useful to explain some of them. Imagine we serialized an instance of Point and we need to materialize it when Point class has changed.

Let's start with the easier cases. If a variable was **inserted**, its value will be nil. If **removed**, it is also obvious: the serialized value will be ignored. In

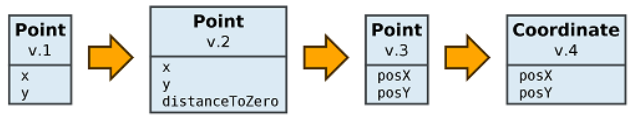

Figure 1.1: Example of changes in a class.

the case the variables are the same the the **order changed**, Fuel also tolerates it automatically.

A more interesting case is when a variable was **renamed**, where the user can map old names to new ones. In our example:

```
FLMaterializer newDefault
  migrateClassNamed: #Point
  variables: {'x' -> 'posX'. 'y' -> 'posY'}.
```

Not surprisingly, if nothing specified the change will be understood by Fuel as two independent operations, an insertion and a removal.

The last change in the figure is a **class rename**. This should be specified this way:

```
FLMaterializer newDefault
  migrateClassNamed: #Point
  toClass: Coordinate.
```

It is also available #migrateClassNamed:toClass:variables: to combine both **class and variable rename**.

Although not illustrated in the figure, a class could also change its **layout**. For example, Point could change from being **fixed** to **variable**. This should be also automatically tolerated by Fuel. Unfortunately, the inverse (variable to fixed) is not supported so far.

You can find tests related to this guide in FLMigrationTest.

Additionally, the method globalEnvironment:, showed in 1.4, might be useful for migrations: you can prepare an ad-hoc environment dictionary with the same keys that were used during serialization, but with the new classes as values.

1.8 Fuel Format Migration

Until now, each Fuel version has its own stream format. Furthermore, each version is **not** compatible with the others. This means that when upgrading Fuel version, we will need to convert our serialized streams.

We include below an example of migration. Let's say we have some files serialized with Fuel 1.7 in a Pharo 1.4 image and we want to migrate them to Fuel 1.9.

```
| oldVersion newVersion fileNames objectsByFileName
  materializerClass serializerClass |
oldVersion := '1.7'.
newVersion := '1.9'.
fileNames := #('a.fuel' 'b.fuel' 'c.fuel' 'd.fuel' 'e.fuel').
objectsByFileName := Dictionary new.


(ConfigurationOfFuel project version: oldVersion) load.
materializerClass := Smalltalk at: #FLMaterializer.

fileNames do: [ :fileName |
  objectsByFileName
    at: fileName
    put: (materializerClass materializeFromFileNamed: fileName) ].


(ConfigurationOfFuel project version: newVersion) load.
serializerClass := Smalltalk at: #FLSerializer.

objectsByFileName keysAndValuesDo: [ :fileName :objects |
  serializerClass
    serialize: objects
    toFileNamed: 'migrated-', fileName
  ].
```

Note: Note 1: We assume in this example that the number of objects to migrate can be materialized all together at the same time. This can be false. In such case, we could fix the script to split the list of files and do it in parts.

Note: Note 2: It is necessary to fetch the classes in the System Dictionary after the desired Fuel version has been loaded.

Note: Note 3: This script should be evaluated in the original image. For example, we don't guarantee that Fuel 1.7 loads in Pharo 2.0, but we know that Fuel 1.9 loads in Pharo 1.4.

1.9 Debugging

There are a couple of packages that help us debugging Fuel. To understand the output of the tools in this guide, you should know some basics of how Fuel internally works.

Serialization

The most important thing to remark is that serialization is split in two main steps: analysis and encoding.

Analysis

It consists in a graph iteration, mapping each traversed object to its correspondent grouping, called **cluster**.

Encoding

After analysis, we linearly write on the stream, in these steps:

1. header
2. for each cluster, instances part
3. for each cluster, references part
4. trailer

Materialization

It consists on progressively recreating the graph.

Decoding

This is done by linearly reading from the stream. So, steps are obviously analogous to the ones above:

1. header
2. for each cluster, instances part
3. for each cluster, references part
4. trailer

Debug Tools

Ensure you have them with:

```
(ConfigurationOfFuel project version: '1.8.1')
   load: #(FuelDebug FuelPreview).
```

Next, a transcript of some useful class comments.

FLGraphViewBuilder

I add draw capabilities to analysis in FuelDebug package.

Right-click a node for inspect it. Some examples:

```
(FLAnalyzer newDefault
   setDebug;
   analysisFor: #((1) (2) (3) (4)))
   open.
```

```
(FLAnalyzer newDefault
   setDebug;
   analysisFor: #((1) (2) (3) (4)))
   openPathsTo: 3.
```

```
(FLAnalyzer newDefault
   setDebug;
   analysisFor: #((1) (2) (3) (4)))
   openPathsToEvery: [:o | o isNumber and: [o > 2] ].
```

Figure 1.2 shows how they look like.

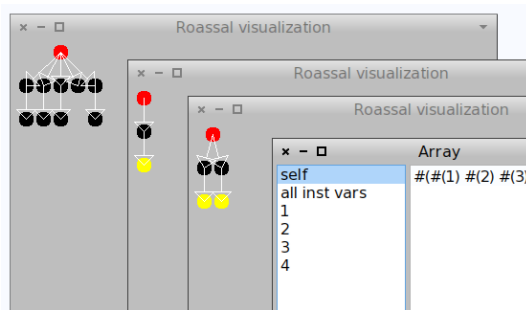

Figure 1.2: Visual preview of graph that would be serialized.

FLDebugSerialization

I am a serialization which facilitates debugging, by logging the stream position before and after main steps of FLSerialization, including cluster information. Obviously, you should be familiar with such class and the algorithm to understand the output log.

To use, send #setDebug to your serializer and run as usually. For example:

```
FileStream forceNewFileNamed: 'debug.fuel' do: [:aFile |
  FLSerializer newDefault
    setDebug;
    serialize: "hello" on: aFile binary ]
```

Then, inspect the output log:

```
FLDebugSerialization last log
```

FLDebugMaterialization

I am a materialization which facilitates debugging, by logging the stream position before and after main steps of FLMaterialization, including cluster information. Obviously, you should be familiar with such class and the algorithm to understand the output log.

To use, send #setDebug to your serializer and run as usually. For example:

```
FileStream oldFileNamed: 'debug.fuel' do: [:aFile |
  FLMaterializer newDefault
    setDebug;
    materializeFrom: aFile binary ]
```

Then, inspect the output log:

```
FLDebugMaterialization last log
```

1.10 Built-in Header Support

Since the graph of objects serialized in a file can be large, and it can be useful to query some small extra info, Fuel supports the possibility to easily add such information in a header. The following examples show this set of features:

```
| serializer |
serializer := FLSerializer newDefault.

serializer header
  at: #timestamp
```

```
    putAdditionalObject: TimeStamp now.

serializer header
    addPreMaterializationAction: [
        Transcript show: 'Before serializing'; cr ].

serializer header
    addPostMaterializationAction: [ :materialization |
        Transcript
            show: 'Serialized at ';
            show: (materialization additionalObjectAt: #timestamp);
            cr;
            show: 'Materialized at ';
            show: TimeStamp now;
            cr ].

serializer
    serialize: 'a big amount of data'
    toFileNamed: 'demo.fl'
```

Then, you can just materialize the header info:

```
| aHeader |
aHeader := FLMaterializer materializeHeaderFromFileNamed: 'demo.fl'.
aHeader additionalObjectAt: #timestamp.
```

Printing it, the result is:

```
'28 March 2013 12:44:54 pm'
```

If we normally materialize the whole file with:

```
FLMaterializer materializeFromFileNamed: 'demo.fl'
```

Then, the print of the results is:

```
'a big amount of data'
```

And this is shown in Transcript:

```
Before serializing
Serialized at 28 March 2013 12:50:50 pm
Materialized at 28 March 2013 1:01:21 pm
```

For additional examples, you can see tests in FLHeaderSerializationTest.