Settings refactoring

rydnr

April 5, 2021

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

Contents

Illustrations

Part I

PharoEDA

CHAPTER 1

Introduction

PharoEDA is a Pharo framework to simplify the development of event-driven applications, using Event Sourcing, following DDD principles.

At runtime, PharoEDA provides primary adapters for consuming messages and publishing events. Its objective is to take complexity out of the application as much as possible.

It encourages TDD, and also provides a test framework.

1.1 Limitations

Currently, only one PharoEDA application can be running in a Pharo image at the same time.

An `EDAApplication` has configurable properties, that can be inspected and modified using `SettingBrowser`. The `SettingBrowser` customization is done in `EDASettings` class, using the `<systemsettings>` pragma, which requires the method to be on the class side. Currently, the methods in charge of creating the customization tree expects each setting is stored in specific classes, available via accessors, in class-side attributes. The main benefit of this approach is to make `EDASettings` unaware of the existence of the `EDAApplication` class. However, this only applies on the surface, because both classes know which classes contain the actual setting values.

Additionally, the configuration settings are read from a json file, which gets identified by an environment variable. But that file is only used for the initial values, and its contents are not synchronized with the values of the application after they are modified using `SettingBrowser`. Ideally, each setting could be managed differently: either from an environment variable, from a json file, from a global value, and so on.

1.2 Motivation

This refactoring attempts, on one hand, to avoid using classes as setting containers, and sharing them between `EDASettings` and `EDAApplication`. On the other hand, to be able to manage more than one `EDAApplication` instance. Every instance should have its own settings, and its own subtree in `SettingBrowser`. Finally, to abstract the holder of each setting so the actual value can be obtained from different sources.

1.3 Objective

This refactoring is divided in different parts. We'll focus first on customizing settings available through `EDAApplication` instances. Then, we'll use Ports and Adapters to decouple `EDAApplication` from the actual settings. We'll go through all settings one by one, replacing the current solution with a new adapter. This will enable us to implement new adapters for specific settings. For example, to retrieve credentials from external vaults. At the end, we'll be able to have different `EDAApplication` instances running, each with its own configuration settings, independent of each other.

1.4 Load the project

To load the project,

```
Metacello new repository: 'github://osoco/pharo-eda:settings';
    baseline: #PharoEDA; load
```

CHAPTER 2

Application registry

We need a way to know which `EDAApplication` instances exist, and a way to interact with them. A simple registry is enough for now.

Let's start by creating the class with a **registry** attribute we'll use to annotate the references to the existing applications.

```
Object subclass: #EDAApplications
  uses: EDATLogging
  instanceVariableNames: 'registry'
  classVariableNames: ''
  package: 'EDA-Application'
```

The **registry** should be accessed via dedicated methods, under the *accessing* protocol.

```
EDAApplications >> registry: aDictionary
  registry := aDictionary
```

```
EDAApplications >> registry
  | result |
  result := registry.
  result ifNil: [
    result := Dictionary new.
    self registry: result
  ].
  ^ result
```

We'll throw an `EDAApplicationAlreadyRegistered` exception in case the application is already registered.

```
Error subclass: #EDAApplicationAlreadyRegistered
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'EDA-Application'
```

Now we can implement a method to register applications. We'll identify each application by its name.

```
EDAApplications >> register: anEDAApplication under:
    anApplicationName
  self registry
    at: anApplicationName
    ifPresent: [ :app | EDAApplicationAlreadyRegistered signal: app
    printString ]
    ifAbsentPut: [ anEDAApplication ]
```

```
EDAApplications >> register: anEDAApplication
  self
    register: anEDAApplication
    under: anEDAApplication applicationName
```

Now each application needs a name, but it could be optional at instance level. We'd like to omit the instance name if only one instance will ever be created. In other words, an instance can have its own name, but can rely upon the class'.

We need to add the new attribute to `EDAApplication`:

```
Object subclass: #EDAApplication
  uses: EDATLogging
  instanceVariableNames: 'applicationName eventStore eventAnnouncer
    commandConsumer commandDispatcher eventProducer
    commandListenerClientDebug isConfigured isStarted
    projectorsByTenant projectionSynchronizationEnabled'
  classVariableNames: ''
  package: 'EDA-Application'
```

The following accessors deal with that:

```
EDAApplication >> applicationName
  ^ applicationName ifNil: [ self class applicationName ]
```

```
EDAApplication >> applicationName: aString
  applicationName := applicationName
```

We'll delegate on the class itself to provide a default name.

```
EDAApplication class >> applicationName
  ^ 'default'
```

Now that every application instance has a name, and it's used to reference the application in the registry, we should provide a way to inspect the applications already registered:

```
EDAApplications >> edaApplications
  ^ self registry values
```

Recall that `EDAApplication` was previously meant to be used as a class, not to create instances. It might be useful to guide developers on how to create new instances. If they use **new** and forget to provide a name, they'll end up with unnamed applications, and won't be able to register more than one in the registry. Therefore, we'll miss precisely the point we're trying to make.

This factory method (class-side) ensures the instances are created with a name. Notice we don't use **self new**. We're dealing with that soon enough.

```
EDAApplication class >> withName: anApplicationName
  | result |
  result := super new.\n
  result applicationName: anApplicationName.
  ^ result
```

Analogously, we can create "unnamed" instances as well.

```
EDAApplication class >> unnamed
  ^ self withName: self applicationName
```

However, instances can be created without a name (being "unnamed"). If we want to ensure instances are created via the factory methods above, we can override the **new** method.

```
EDAApplication class >> new
  MessageNotUnderstood
    signal:
      'Use ' , self class printString , ' >> withName: or ' , self
    class printString , ' >> unnamed instead'
```

We'd need to convert `EDAApplications` into a singleton, to ensure there's only one registry of running applications. To do so, we add a **uniqueInstance** attribute on the class itself.

```
EDAApplications class
  uses: EDATLogging classTrait
  instanceVariableNames: 'uniqueInstance'
```

Again, we use a lazy approach in the getter:

```
EDAApplications class >> uniqueInstance
  | result |
  result := uniqueInstance.
  result
    ifNil: [
      result := self new.
      self uniqueInstance: result
    ].
  ^ result
```

For the sake of completeness, we define the trivial setter.

```
EDAApplications class >> uniqueInstance: anInstance
  | result |
  uniqueInstance := anInstance
```

2.1 Registration of EDAApplications

At the expense of introducing a cyclic coupling between `EDAApplication` and `EDAApplications` classes, it's convenient to make sure all `EDAApplication` instances get registered automatically.

```
EDAApplication class >> withName: anApplicationName
  | result |
  (EDAApplications uniqueInstance edaApplications
    select: [ :app | app applicationName = anApplicationName ])
    ifEmpty: [
      result := super new.
      result applicationName: anApplicationName.
      EDAApplications uniqueInstance register: result ]
    ifNotEmpty: [ :c | result := c first ].
  ^ result
```

Finally, we can create applications and register them.

```
EDAApplications uniqueInstance register: (EDAApplication withName:
    'test-1')
```

And inspect the registered applications as well.

```
EDAApplications uniqueInstance edaApplications
```

CHAPTER 3

Different subtrees for each application

In `EDASettings` we need to build subtrees dynamically, based on the already registered applications. We can create a new entry using the **group:** method of the `SettingTreeBuilder` that is passed to the `<systemsettings>`-tagged method: `aBuilder group: #subtreeSymbol`

We need to ensure `#subtreeSymbol` is unique. Otherwise the subtree will be created under an existing entry.

Let's remove the current `<systemsettings>` pragma of `EDASettings class>>edaSettingsOn:` and rename it to `EDASettings class>>edaSettingsOn: aBuilder for: anEDAApplication under: aSymbol`, and let's write a new `EDASettings class>>edaSettingsOn:` from scratch.

Since we haven't created (or registered) any application yet, we can choose either to skip creating anything, or display a message.

```
EDASettings class >> edaSettingsOn: aBuilder
  <systemsettings>
  | root rootSymbol |
  rootSymbol := #edaApps.
  root := aBuilder group: rootSymbol.
  EDAApplications uniqueInstance edaApplications
    ifEmpty: [
      root
        label: 'No PharoEDA applications' translated;
        description: 'No PharoEDA applications registered' ]
    ifNotEmpty: [ :apps |
      root
        label: 'PharoEDA application(s)' translated , ': '
```

```
                , apps size printString translated;
        description: 'Registered PharoEDA applications';
        noOrdering.
      apps do: [ :app |
        self edaSettingsOn: aBuilder for: app under: rootSymbol
      ]
    ]
```

However, the old **edaSettings:** method used a fixed symbol to place the application settings in the tree. Let's fix that. We'll start with the `environmentSettingsOn:under:on:`. The rest of the methods will be fixed later.

```
EDASettings class >> edaSettingsOn: aBuilder for: anEDAApplication
    under: aSymbol
  | parent parentGroup |
  parent := anEDAApplication applicationName.
  parentGroup := aBuilder group: parent.
  parentGroup
    label: parent translated;
    parent: aSymbol;
    description: parent;
    noOrdering.
  self
    environmentSettingsOn: aBuilder
    under: parent
    on: anEDAApplication
```

As you can see, we've renamed the previous method from **environmentSettingsOn: aBuilder under: aSymbol** to **environmentSettingsOn: aBuilder: under: aSymbol on: anEDAApplication**. To display and manage each application's settings, we need the application instance. Previously, we used class-scoped attributes of predefined classes as settings. Now we're going to access them through the application instance itself. We need to create the `config/` folder and two files (**development.json** and **integration-tests.json**) with an initial empty json file (`{}`) in them.

Additionally, we'll comment some settings code in `EDATestSettings` class as well. We'll deal with it later.

```
EDATestSettings class >> edaTestSettingsOn: aBuilder
  <systemsettings>
  (aBuilder group: #edatests)
    label: 'EDA Tests' translated;
    description: 'EDA tests';
    noOrdering;
    parent: #eda
```

We might just add the settings as instance attributes of the `EDAApplication` itself. Such solution would not be flexible enough.

CHAPTER 4

First Ports and Adapters

To accomodate different ways to access and customize setting values, we can abstract them as a Port. `EDAApplication` instances will include a reference to a port, regardless of the actual implementation used.

The required ports are: environment, command listener, event publisher, event store, projections, and logging.

4.1 Minimal behavior shared among all Settings Ports

Even though in Smalltalk there's no such thing as an interface (a definition of the contract exposed by all implementations), we'd like to ensure all adapters share certain behavior. For that, we'll use a trait.

```
Trait named: #EDATSettingsPort
  uses: {}
  package: 'EDA-Settings-Ports'
```

The most basic behavior all adapters should support is telling us a brief description of themselves.

```
EDATSettingsPort classTrait >> description
  self subclassResponsibility
```

Part II

Refactorings

CHAPTER 5

Environment setting

This setting is used to identify the environment. Currently, it shows the value of the **PHARO_ENV** environment variable. The first adapter would be one that supports this feature.

If we inspect the current logic used in `EDASettings` to inject the environment setting into de `SettingBrowser` tree, we see the following:

```
EDASettings class >> environmentSettingsOn: aBuilder under: aParent
    on: app
  (aBuilder pickOne: #currentEnvironment)
    parent: aParent;
    target: EDADUEnvironment;
    label: 'Environment' translated;
    description: 'Environment';
    order: 1;
    domainValues:
      (EDADUEnvironment environments
        collect: [ :level | level translated -> level greaseString ])
```

The nature of this setting is to choose one of the available environments. It's mainly informative, but it might impose certain restrictions or influence other settings as well.

5.1 Environment-variable adapter

As its value is currently provided by the **PHARO_ENV** environment variable, it cannot be changed. We'll change the **pickOne:** above with a simple label, as we did before, when there're no registered applications. We can create a new `EDA-Settings-Adapters-EnvVars` package for this kind of adapters.

```
Object subclass: #EDAEnvironmentSettingsEnvVarAdapter
  uses: EDATSettingsPort
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'EDA-Settings-Adapters-EnvVars'
```

Now we need to distinguish between the case in which the value is a single string, and the previous assumption that it'd always be a list of items. A simple way to accomplish it is to ask the port itself. Remember that `EDASettings` relies on the `EDAApplication` instance to provide the settings, and the latter only deals with ports, not adapters. It makes sense to create a new custom `EDATSettingsPort` specifically for the environment-related adapters.

```
Trait named: #EDATEnvironmentSettingsPort
  uses: EDATSettingsPort
  package: 'EDA-Settings-Ports'
```

Of course, we'll use that trait in our current adapter instead of the generic one we were using so far.

```
Object subclass: #EDAEnvironmentSettingsEnvVarAdapter
  uses: EDATEnvironmentSettingsPort
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'EDA-Settings-Adapters-EnvVars'
```

This variation can now be accomodated in the new `EDATEnvironmentSettingsPort`.

```
EDATEnvironmentSettingsPort >> supportsMultiple
  self subclassResponsibility
```

Now, `EDASettings` can include both variations:

```
EDASettings class >> environmentSettingsOn: aBuilder under: aParent
    on: app
  | entry port |
  port := app environmentSettingsPort.
  (port supportsMultiple)
    ifTrue: [
      entry := aBuilder pickOne: #currentEnvironment.
      entry
        domainValues:
          (EDADUEnvironment environments
            collect: [ :level | level translated -> level
    greaseString ]) ]
    ifFalse: [
      entry := aBuilder setting: #currentEnvironment ].
    entry
      parent: aParent;
      target: port;
```

```
      label: 'Environment' translated;
      description: 'The current environment' translated;
      order: 1
```

`EDAApplication` has to provide an adapter for its **environmentSettingsPort** somehow. We'll deal with a mechanism to choose from a list of available adapters soon. For now, let's return the one we've just implemented: `EDAEnvironmentSettingsEnvVarAdapter`.

```
Object subclass: #EDAApplication
  uses: EDATLogging
  instanceVariableNames: 'applicationName commandConsumer
    commandDispatcher commandListenerClientDebug
    environmentSettingsPort eventAnnouncer  eventProducer eventStore
    isConfigured isStarted projectorsByTenant
    projectionSynchronizationEnabled'
  classVariableNames: ''
  package: 'EDA-Application'.
```

```
EDAApplication >> environmentSettingsPort
  | result |
  result := environmentSettingsPort.
  result ifNil: [
    result := EDAEnvironmentSettingsEnvVarAdapter new.
    self environmentSettingsPort: result
  ].
  ^ result
```

```
EDAApplication >> environmentSettingsPort: anAdapter
  anAdapter = environmentSettingsPort
    ifTrue: [ ^ self ].
  environmentSettingsPort := anAdapter
```

When we launch a `SettingBrowser` we get an `SubclassResponsibility` error. Let's implement it in our adapter. But does it support a list of values? Actually, no. We cannot change the value of the environment value.

```
EDAEnvironmentSettingsEnvVarAdapter >> supportsMultiple
  ^ false
```

Running again `SettingBrowser` will try to call **currentEnvironment** on our adapter, but we haven't implemented it yet. Let's do it now.

```
EDAEnvironmentSettingsEnvVarAdapter >> currentEnvironment
  ^ OSPlatform current environment at: 'PHARO_ENV' ifAbsent:
    '(PHARO_ENV not set)'
```

`SettingBrowser` should render the tree now, but we're not done yet with this adapter. `EDASettings` assumes the setting can be customized. That is, the adapter supports setting a new value. How can we make it read-only? First of all, `EDASettings` mustn't make any assumption. It's responsibility of the port to provide that information.

So if the port's **isReadOnly** returns **true**, then `EDASettings` will use a `LabelMorph` to display the value provided by the port itself.

We're implementing the ports as traits. It makes sense to indicate all adapters need to override this method.

```
EDATEnvironmentSettingsPort >> isReadOnly
  self subclassResponsibility
```

Our adapter clearly needs to return **true**:

```
EDAEnvironmentSettingsEnvVarAdapter >> isReadOnly
  ^ true
```

All adapters will share the same entry in `SettingsBrowser`. We'll define that label in `EDATEnvironmentSettingsPort`'s **description**:

```
EDATEnvironmentSettingsPort classTrait >> description
  ^ 'Environment'
```

Based on the port and adapter's metadata, `EDASettings` will choose how to properly display the entry it the tree.

```
EDASettings class >> environmentSettingsOn: aBuilder under: aParent
    on: app
  | entry port |
  port := app environmentSettingsPort.
  port supportsMultiple
    ifTrue: [
      entry := aBuilder pickOne: #currentEnvironment.
      entry
        domainValues:
          (EDADUEnvironment environments
            collect: [ :level | level translated -> level
    greaseString ]) ]
    ifFalse: [
      port isReadOnly
        ifTrue: [
          entry := aBuilder group: #currentEnvironment.
          entry dialog: [ LabelMorph newLabel: port
    currentEnvironment ] ]
        ifFalse: [ entry := aBuilder setting: #currentEnvironment ]
    ].
    entry
      parent: aParent;
      target: port;
      label: port class description greaseString;
      description: port class description greaseString;
      order: 1
```

Refactoring

Even though our adapter is working, it's not reusable. Every other setting whose value comes from an environment variable would need to share most of its behavior. Before continuing with the next adapter, let's refactor this one a bit.

First, let's extract the logic to retrieve values from environment variables into its own trait.

```
Trait named: #EDATEnvVarAdapter
  uses: {}
  package: 'EDA-Settings-Adapters-EnvVars'
```

```
EDATEnvVarAdapter >> getValueFor: envVarName orElse: defaultValue
  ^ OSPlatform current environment
    at: envVarName
    ifAbsent: defaultValue
```

```
EDATEnvVarAdapter >> supportsMultiple
  ^ false
```

```
EDATEnvVarAdapter >> isReadOnly
  ^ true
```

Our adapter now uses this new trait instead of `EDATSettingsPort`:

```
Object subclass: #EDAEnvironmentSettingsEnvVarAdapter
  uses: EDATEnvironmentSettingsPort + EDATEnvVarAdapter
  instanceVariableNames: ''
  lassVariableNames: ''
  package: 'EDA-Settings-Adapters-EnvVars'
```

Now **currentEnvironment** implementation can delegate the logic to the new trait:

```
EDAEnvironmentSettingsEnvVarAdapter >> currentEnvironment
  ^ self
      getValueFor: 'PHARO_ENV'
      orElse: '(PHARO_ENV not set)'
```

Additionally, `EDAEnvironmentSettingsEnvVarAdapter` uses a hard-coded environment variable. Let's fix it as well.

```
EDAEnvironmentSettingsEnvVarAdapter >> environmentVariableName
  ^ 'EDA_ENV'
```

We can refactor the **currentEnvironment** implementation:

```
EDAEnvironmentSettingsEnvVarAdapter >> currentEnvironment
  ^ self
      getValueFor: self environmentVariableName
      orElse: '(' , self environmentVariableName , ' not set)',
```

5.2 Predefined list adapter

This adapter allows the user to choose among a predefined list of possible values for the environment setting. From the point of view of `EDASettings`, if the port supports multiple values, it needs to provide them in advance.

For the new adapter, let's create it under a new package `EDA-Settings-Adapters-InMemory`.

```
Object subclass: #EDAPredefinedEnvironmentSettingsInMemoryAdapter
  uses: EDATEnvironmentSettingsPort + EDAPrintOnHelper +
    EDATCollectionHelper
  instanceVariableNames: 'currentEnvironment'
  classVariableNames: ''
  package: 'EDA-Settings-Adapters-InMemory'
```

```
EDAPredefinedEnvironmentSettingsInMemoryAdapter >> currentEnvironment
  ^ currentEnvironment ifNil: [ self availableEnvironments first ]
```

It needs to store the selected value in memory.

```
EDAPredefinedEnvironmentSettingsInMemoryAdapter >>
    currentEnvironment: aByteString
  currentEnvironment := aByteString
```

The list of predefined values is provided by the **availableEnvironments** method.

```
EDAPredefinedEnvironmentSettingsInMemoryAdapter >>
    availableEnvironments
  ^ OrderedCollection with: 'dev' with: 'test' with: 'PRO'
```

We need to tell that the new adapter supports multiple values, so `EDASettings` can use **pickOne:** to render the setting.

```
EDAPredefinedEnvironmentSettingsInMemoryAdapter >> supportsMultiple
  ^ true
```

`EDASettings` can now display the options regardless of the adapter used.

```
EDASettings class >> environmentSettingsOn: aBuilder under: aParent
    on: app
  | entry port |
  port := app environmentSettingsPort.
  port supportsMultiple
    ifTrue: [
      entry := aBuilder pickOne: #currentEnvironment.
      entry
        domainValues: (
          port availableEnvironments
            collect: [ :level | level translated -> level
    greaseString ]) ]
    ifFalse: [
```

```
      port isReadOnly
        ifTrue: [
          entry := aBuilder group: #currentEnvironment.
          entry dialog: [ LabelMorph newLabel: port
    currentEnvironment ] ]
        ifFalse: [ entry := aBuilder setting: #currentEnvironment ]
    ].
  entry
    parent: aParent;
    target: port;
    label: port class description greaseString;
    description: port class description greaseString;
    order: 1
```

Also, even though now it's not strictly necessary, this adapter supports read-write values, in case anyone needs to know it.

```
EDAPredefinedEnvironmentSettingsInMemoryAdapter >> isReadOnly
  ^ false
```

Now, to test it we can just change `EDAApplication` to use it instead of the current one. We'll introduce a mechanism to inject adapters into ports dynamically soon.

```
EDAApplication >> environmentSettingsPort
  | result |
  result := environmentSettingsPort.
  result
    ifNil: [
      result := EDAPredefinedEnvironmentSettingsInMemoryAdapter new.
      self environmentSettingsPort: result ].
  ^ result
```

The only missing piece is to provide a description. It'd be great if We'll use some methods from existing traits in PharoEDA.

```
Object subclass: #EDAPredefinedEnvironmentSettingsInMemoryAdapter
  uses: EDATEnvironmentSettingsPort + EDAPrintOnHelper +
    EDATInMemoryAdapter + EDATCollectionHelper
  instanceVariableNames: 'currentEnvironment'
  classVariableNames: ''
  package: 'EDA-Settings-Adapters-InMemory'
```

```
EDAPredefinedEnvironmentSettingsInMemoryAdapter class >> description
  ^ 'Predefined list of environments'
```

Refactoring

As we did with the environment-variable adapter, let's introduce a new intermediary trait, common to all "in-memory" adapters, to indicate **isReadOnly** is by default **false**.

```
Trait named: #EDATInMemoryAdapter
  uses: EDATSettingsPort
  package: 'EDA-Settings-Adapters-InMemory'
```

```
Object subclass: #EDAPredefinedEnvironmentSettingsInMemoryAdapter
  uses: EDATInMemoryAdapter
  instanceVariableNames: 'currentEnvironment'
  classVariableNames: ''
  package: 'EDA-Settings-Adapters-InMemory'
```

5.3 In-memory adapter

Another useful choice is to let the user decide the name of the environment, and store it in memory.

```
Object subclass: #EDAEnvironmentSettingsInMemoryAdapter
  uses: EDATEnvironmentSettingsPort + EDAPrintOnHelper +
    EDATInMemoryAdapter
  instanceVariableNames: 'currentEnvironment'
  classVariableNames: ''
  package: 'EDA-Settings-Adapters-InMemory'
```

We just need to provide the accessors for the attribute, and we're done.

```
EDAEnvironmentSettingsInMemoryAdapter >> currentEnvironment
  ^ currentEnvironment ifNil: [ '' ]
```

```
EDAEnvironmentSettingsInMemoryAdapter >> currentEnvironment: anObject
  currentEnvironment := anObject
```