

Reddit.st in 10 elegant classes

(A more recent and up to date version of this nice tutorial is available at: <https://medium.com/@svenvc/reddit-st-in-10-cool-pharo-classes-1b5327ca0740>)

CRUD refers to the Create, read, update and delete operations in persistent storage. The term CRUD applications often refers to simple applications managing simple data and that have to be created fast. Often CRUD development is done with languages such as PHP. In this Chapter we want to show that Smalltalk is a really good alternative and that in addition, your CRUD applications can evolve (which is often the problems of the use of traditional languages). We will show you that it is perfectly possible to write nice web applications in Pharo and really fast and Reddit.st adds persistency in a relational database, unit tests as well as web application components to the mix.

We will show how to implement a small but non-trivial web application in Smalltalk using Seaside, Glorp and PostgreSQL. Reddit, is web application where users can post interesting links that get voted up or down. The idea is that the 'best' links end up with the most points automatically. Many other websites exist in the area of social bookmarking, like Delicious, Digg and Hacker News.

The 10 main sections of this chapter follow the development of the 10 classes making up the application. The focus of the Smalltalk version is not so much on the small size or the high developer productivity, but more on the fact that we can cover so much ground using such powerful frameworks, as well as the natural development flow from model over tests and persistence to web GUI.

The material shown in this Chapter was originally written by Sven Van Caekenberghe and we thanks him for his permission to use it to create this book chapter. We assume that you understand what web applications are and how

Seaside basically works. If not, you should read the Seaside Chapter for an introduction. We also will assume that you have a basic understanding of relational databases and/or SQL.

1.1 First: a Model

The central object of our application will be `RedditLink`, an interesting URL with a title, a created timestamp and a number of points. It has the following properties: `id url title created points`.

These are naturally instance variables of our class. Create a new class inheriting from `Object` subclass by editing the class template.

```
Object subclass: #RedditLink
  instanceVariableNames: 'id url title created points'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Reddit'
```

Next, use the class refactoring tool to automatically generate accessors (getters and setters) for all our instance variables. With these implemented we can write our `initialize` and `printOn:` methods.

```
RedditLink>>initialize
  self initialize.
  self points: 0.
  self created: TimeStamp now

RedditLink>>printOn:
  super printOn: stream.
  stream nextPut: $(.
  self url printOn: stream.
  stream nextPut: $,.
  self title printOn: stream.
  stream nextPut: $)
```

We also add a method named `posted` that will return the Duration of time the link now exists. We will need that when rendering links later on.

```
RedditLink>>posted
  ^ TimeStamp now - self created
```

Apart from creating and displaying `RedditLinks`, users should be able to vote them up and down. Therefore, we add two action methods, `voteUp` and `voteDown`.

```
RedditLink>>voteUp
  self points: self points + 1

RedditLink>>voteDown
  self points > 0 ifTrue: [ self points: self points - 1 ]
```

We also introduce the class method `withUrl:title:` to create new instances as follows

```
RedditLink class>>withUrl: url title: title
  ^ self new url: url; title: title; yourself
```

The core of the `RedditLink` object is now finished. Everything is ready to make instances and use them.

1.2 RedditLinkTests

Units tests are very important, not so much in small examples like this one, but especially in larger applications. Having a good set of unit tests with descent coverage helps protect the code during changes. At the same time, unit tests function as working documentation. Instead of writing scratch test code in some workspace, you can just as well write a unit test. We created the class `RedditLinkTests` as a subclass of `TestCase` and add 3 test methods.

After creating a `RedditLink` object, and/or manipulating it, these methods assert that certain conditions hold. To improve code sharing (we'll need it again in Section 5) we add a method called `assertContractUsing:` to `RedditLink` that checks the basic contract of the receiver, using the method `assert:` on an arbitrary object. For completeness, we also implement a general `isValid` testing method.

```
RedditLink>>assertContractUsing: object
  object perform: #assert: with: (self url isNil or: [ self url
    isKindOf: String ]).
  object perform: #assert: with: (self title isNil or: [ self title
    isKindOf: String ]).
  object perform: #assert: with: (self created isKindOf: TimeStamp).
  object perform: #assert: with: (self points isKindOf: Integer).
  object perform: #assert: with: (self posted asSeconds >= 0).
  object perform: #assert: with: (self printString isKindOf: String)

RedditLink>>isValid
  self assertContractUsing: self
```

Now we can take advantage of these methods in our tests.

```
RedditLinkTests>>testInitialState
  | link |
  link := RedditLink new.
  link assertContractUsing: self.
  self assert: link points isZero

RedditLinkTests>>testCreate
  | link url title |
  url := 'http://www.seaside.st'.
```

```

title := 'Seaside'.
link := RedditLink withUrl: url title: title.
link assertContractUsing: self.
self assert: link points isZero.
self assert: link url = url.
self assert: link title = title

```

```

RedditLinkTests>>testVoting
| link |
link := RedditLink new.
link assertContractUsing: self.
self assert: link points isZero. link voteUp.
self assert: link points = 1.
link voteDown. self assert:
link points isZero.
link voteDown.
self assert: link points isZero

```

1.3 Describing Database Data

To make our application less trivial, we are going to make our collection of links persistent in a relation database. For this we are going to use Glorp, an object-relational mapping tool. Glorp will take care of all the SQL! To do its magic, Glorp needs a `DescriptorSystem` that tells it 3 things: the class models involved, the tables involved and the way the two map (which it calls a descriptor). In this simple case we are just mapping one object into one table, so the descriptor system might look a bit verbose. Just remember that Glorp can do much, much more advanced things. Furthermore, there exist extensions to Glorp that implement `ActiveRecord` style automatic descriptors.

Loading Glorp.

Here is how you should load Glorp

Defining Schema

First let us define a new class description named `RedditSchemaDescriptor` subclass of `DescriptorSystem` for Glorp.

```

DescriptorSystem subclass: #RedditSchemaDescriptor
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'Reddit-DB'

```

Using this class we can tell Glorp about our class model, `RedditLink`, and which instance variables are to be persistent attributes. Instead of some XML

description, a much more powerful Smalltalk object model is being built. Conventions in methods names are being used to glue things together.

```
RedditSchemaDescriptor>>constructAllClasses
  ^ super constructAllClasses add: RedditLink; yourself

RedditSchemaDescriptor>>classModelForRedditLink: aClassModel
  #(id url title created points) do: [ :each | aClassModel
    newAttributeNamed: each ]
```

The second step is to describe which tables are involved. So here we list all the fields (columns) of our table REDDIT_LINKS. Glorp shields us from the differences between different SQL dialects. Note how id is designated to be a primary key. The serial type will result in an SQL sequence being used.

what is that a serial type

```
RedditSchemaDescriptor>>allTableNames
  ^ #( 'REDDIT_LINKS' )

RedditSchemaDescriptor>>tableForREDDIT_LINKS: aTable
  (aTable createFieldNamed: 'id' type: platform serial)
  bePrimarykey.
  aTable createFieldNamed: 'url' type: (platform varchar: 64).
  aTable createFieldNamed: 'title' type: (platform varchar: 64).
  aTable createFieldNamed: 'created' type: platform timestamp.
  aTable createFieldNamed: 'points' type: platform integer
```

The third and final step is the actual descriptor describing the mapping between the class model `RedditLink` and the table `REDDIT_LINKS`. In this simple case, direct mappings are used between attributes and fields. As we will see in the next sections, Glorp is now ready to do its work.

```
RedditSchemaDescriptor>>descriptorForRedditLink: aDescriptor

  | table |
  table := self tableNamed: 'REDDIT_LINKS'.
  aDescriptor table: table. (aDescriptor newMapping:
  DirectMapping) from: #id to: (table fieldNamed: 'id').
  (aDescriptor newMapping: DirectMapping) from: #url to: (table
  fieldNamed: 'url').
  (aDescriptor newMapping: DirectMapping) from: #title to: (table
  fieldNamed: 'title').
  (aDescriptor newMapping: DirectMapping) from: #created to:
  (table fieldNamed: 'created').
  (aDescriptor newMapping: DirectMapping) from: #points to:
  (table fieldNamed: 'points')
```

1.4 Connecting to the Database

Let's assume you installed and configured PostgreSQL on some machine and that you created some database there. ^{how} We now have to specify how Glorp has to connect to PostgreSQL. We do this creating a new class named `RedditDatabaseResource`

```
Object subclass: #RedditDatabaseResource
  instanceVariableNames: ''
  classVariableNames: 'DefaultLogin'
  poolDictionaries: ''
  category: 'Reddit-DB'
```

We will add some class methods (as well as a class variable called `DefaultLogin`).

```
RedditDatabaseResource class>>login
  DefaultLogin ifNil: [ DefaultLogin := self createLogin ].
  ^ DefaultLogin

RedditDatabaseResource class>>login: aLogin
  "see #createLogin for an example of how to create a Login
  object"
  DefaultLogin := aLogin

RedditDatabaseResource class>>createLogin
  ^ Login new
    database: PostgreSQLPlatform new;
    username: 'svc';
    password: 'secret';
    connectionString: 'localhost:5432_playground';
    yourself
```

The username and password speak for themselves, the `connectString` contains the hostname, port number and database name (after an underscore). Glorp accesses a database through sessions. A session is most easily started from a descriptor system given a login as argument, that's what we do in the session helper class method.

```
RedditDatabaseResource>>session
  ^ RedisSchema sessionForLogin: self login

RedditDatabaseResource>>createTables
  "self createTables"
  "This has to be done only once, be sure to set #login"
  |session|
  session := self session.
  session accessor login; logging: true.
  session inTransactionDo: [ session createTables ].
  session accessor logout
```

Glorp can even help us to create our REDDIT_LINKS table, that is what the createTables class method does. So we truly don't have to use any SQL! The flow should be familiar: get a session, login, do some work in a transaction and logout. By setting logging to true, the generated SQL statements will be printed on the Transcript (comment this out for production use).

1.5 Time to test: RedditDatabaseTest

With our RedditSchema descriptor system and our RedditLinksDatabaseResource we are now ready to test the persistency of our model. We create another class named RedditDatabaseTest which inherits from TestCase. These tests need an instance variable called session to hold the Glorp session, as well as setUp and tearDown methods.

```
Object subclass: #RedditDatabaseTest
  instanceVariableNames: 'session'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Reddit-DB'

RedditDatabaseTest>>setUp
    session := RedditDatabaseResource session.
    session accessor logging: true; login

RedditDatabaseTest>>tearDown
    session accessor logout
```

Our first test reads all RedditLinks from the database, making sure they are valid and of the expected type. Querying doesn't have to be done in a unit of work or transaction.

```
RedditDatabaseTest>>testQuery
|links|
links := session readManyOf: RedditLink.
links do: [ :each |
    each assertContractUsing: self.
    self assert: (each isKindOf: RedditLink) ]
```

The second test creates a new RedditLink and then registers it with the session inside a unit of work. This will effectively save the object in the database. The id of the RedditLink will have a value afterwards. Next we reset the session and query the RedditLink with the known id. After making sure that what we put in got out of the database we delete the object.

```
RedditDatabaseTest>>testUpdate
| link url title id |
url := 'http://www.seaside.st'.
title := 'Seaside Unit Test'.
link := RedditLink withUrl: url title: title.
session inUnitOfWorkDo: [ session register: link ].
```

```

    id := link id.
    session inUnitOfWorkDo: [ session register: link. link voteUp ].
    session reset.

```

The third test checks if updating an existing persistent object works as expected. Note that the actual modification, the `voteUp`, has to be done inside a unit of work to a registered object for it to be picked up by Glorp.

```

RedditDatabaseTest>>testCreate
| link url title id |
url := 'http://www.seaside.st'.
title := 'Seaside Unit Test'.
link := RedditLink withUrl: url title: title.
session inUnitOfWorkDo: [ session register: link ].
id := link id.
self assert: id notNil.
session reset.
link := session readOneOf: RedditLink where: [ :each | each id
= id ].
link assertContractUsing: self.
self assert: link url = url.
self assert: link title = title.
session delete: link

```

1.6 RedditSession

We are almost ready to start writing the GUI of our actual web application. Seaside web applications often have a session object that keeps the application's state during the user's interaction with it. We need to extend that session with a database session. We define a new class `RedditSession` as a subclass of `WASession`. In addition, it has an instance variable called `glorpSession` to hold a Glorp session to the database.

```

WASession subclass: #RedditSession
    instanceVariableNames: 'glorpSession'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Reddit-DB'

```

Note how we are using lazy initialization in the `glorpSession` accessor. In `newGlorpSession` we're making use of our `RedditDatabaseResource`.

```

RedditSession>>glorpSession
    glorpSession ifNil: [ glorpSession := self newGlorpSession ].
    glorpSession accessor isLoggedIn ifFalse: [ glorpSession accessor
        login ].
    ^ glorpSession

RedditSession>>newGlorpSession
    | session |

```

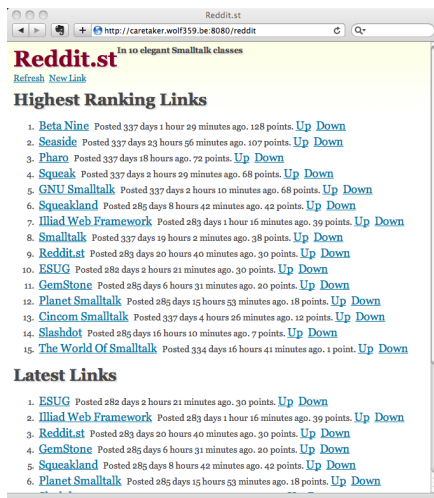



Figure 1.1 Reddit

```

    session := RedditDatabaseResource session. "session accessor
logging: true."
    ^ session

```

The `unregistered` is a hook called by `Seaside` whenever a session expires, we use it clean up our `Glorp` session by doing a log out.

```

RedditSession>>unregistered
    super unregistered.
    self teardownGlorpSession

RedditSession>>teardownGlorpSession
    self glorpSession logout

```

1.7 Web Part: the Component WAREddit

We can finally start with our web app itself. Figure 1 shows the main page of the `Reddit.st` app. There are four sections in this page: a header or title section, some action links, a list of some of the highest or top ranking links and a list if some of the latest or most recent links.

We create a `WAComponent` subclass called `WAREddit`. This will become our central or root web app component. We start by writing the rendering methods.

```

WAREddit>>renderContentOn: html
    html heading: 'Reddit.st'.
    html heading level: 3; with: 'In 10 elegant Smalltalk classes'.
    self renderActionsOn: html.

```

```

:      self renderHighestRankingLinksOn: html.
      self renderLatestLinksOn: html

[WAReddit>>renderActionsOn: html
  html paragraph: [ html anchor callback: [ ]; with: 'Refresh'.
    html anchor callback: [ self inform: 'Not yet implemented' ];
    with: 'New Link' ]

[WAReddit>>renderHighestRankingLinksOn: html
  html heading level: 2; with: 'Highest Ranking Links'.
  html orderedList: [
    self highestRankingLinks do: [ :each | self renderLink:
      each on: html ] ]

[WAReddit>>renderLatestLinksOn: html
  html heading level: 2; with: 'Latest Links'.
  html orderedList: [
    self latestLinks do: [ :each | self renderLink: each on:
      html ] ]

[WAReddit>>renderLink: link on: html
  html listItem: [ html anchor url: link url; title: link url;
    with: link title.
    html text: ' Posted ', (self class durationString: link posted),
      ' ago. '.
    html text: link points asString, ' points. '.
    html anchor
      callback: [ self voteUp: link ];
      title: 'Vote this link up';
      with: 'Up'.
    html space.
    html anchor
      callback: [ self voteDown: link ];
      title: 'Vote this link down'; with: 'Down' ]

```

Starting with the main `renderContentOn:` method, the rendering of each section is delegated to its own method. Note how `renderLink:on:` is used 2 times. For now, we're not yet implementing the 'New Link' action. Our rendering methods depend on 5 extra methods: `highestRankingLinks`, `latestLinks`, the class method `durationString:` and the actions methods `voteUp:` and `voteDown:`. Only the first three are needed to render the page itself.

```

[WAReddit>>highestRankingLinks
  | query |
  query := (Query readManyOf: RedditLink)
    orderBy: [ :each | each points descending ]; limit:
    20; yourself.
  ^ self session glorpSession execute: query

[WAReddit>>latestLinks
  | query |

```

```

query := (Query readManyOf: RedditLink)
    orderBy: [ :each | each created descending]; limit:
    20; yourself.
^ self session glorpSession execute: query

WAReddit>>durationString: duration
^ String streamContents: [ :stream |
    | needSpace printer |
    needSpace := false.
    printer := [ :value :word |
        value isZero ifFalse: [
            needSpace ifTrue: [ stream space ].
            stream nextPutAll: (value pluralize:
word).
            needSpace := true ] ].
    printer value: duration days value: 'day'.
    printer value: duration hours value: 'hour'.
    printer value: duration minutes value: 'minute' ]

```

In `highestRankingLinks` and `latestLinks`, we explicitly build up and execute Query objects with some more advanced options. For duration string conversion we use the powerful `pluralize` method. With these in place we can already render the page. Since we did not yet add any CSS styling, the result will look rather dull.

```

WAReddit>>voteDown:link
    self session glorpSession
        inUnitOfWorkDo: [ :session | session register: link.
            link voteDown]

WAReddit>>voteUp: link
    self session glorpSession
        inUnitOfWorkDo: [ :session | session register: link. link
            voteUp ]

```

Voting links up or down is trivial, like with our database test, we only have to make sure to do the object modifications inside a Glorp unit of work and the actual SQL update will be done automatically.

1.8 RedditFileLibrary

To style our web app, we'll be reusing the CSS file from `Reddit.lisp`. This CSS code references one small GIF for its background gradient. We need to make sure our application makes use of the CSS file and that we serve the actual files. Seaside can serve these files in a couple of ways, we'll be using the `FileLibrary` approach. This is a class where each resource served is implemented as a method.

We define a new class, subclass of `WAFileLibrary`, named `RedditFileLibrary` will thus have 2 methods: `mainCss` and `bgGif`, returning a string and

bytes respectively. These are long methods which are not our main focus so we do not list them here. Based on some naming conventions, Seaside will figure out what mime types to use.

```
RedditFileLibrary>>updateRoot: anHtmlRoot
    super updateRoot: anHtmlRoot.
    anHtmlRoot title: 'Reddit.st'.
    anHtmlRoot stylesheet url: (RedditFileLibrary urlOf: #mainCss)
```

By implementing the `updateRoot: hook` method on `WAReddit`, we can set our page title and CSS. Note again how everything happens in Smalltalk. To install a Seaside application, a class side `initialize` method is typically used.

```
RedditFileLibrary class>>initialize
    (WAAdmin register: self asApplicationAt: 'reddit')
        preferenceAt: #sessionClass put: RedditSession;
    addLibrary: RedditFileLibrary
```

We register our application under the handler `'reddit'` so its URL will become something like `http://localhost:8080/reddit`. Then we tell it to use our custom session class and finally add our file library. We now have a nicely styled, working web app.

1.9 WARedditLinkEditor

One of Seaside's main advantages over other web application frameworks is its support for components. Especially for large and complex projects this makes a huge difference. We'll be introducing a new component to allow the user to enter the necessary information when adding a new link. Consider the difference between figure 1 and figure 2: when the user clicks the `'New Link'` anchor, we'll add an editor just below (while hiding the `'New Link'` anchor). The editor will have its own `'Save'` and `'Cancel'` buttons. Both of these will dismiss the editor, saving or cancelling the new link.

`redditNewLink`

How is Seaside's component model powerful? As we will see next, the component is written without any knowledge of where it will be used. Its validation logic is independent. It is used just by embedding it and by wiring it to its user in a simple way. The subcomponent functions independently from its embedding parent while each keeps its own state: whether the component is visible or not, you can keep on voting links up or down, and doing so will not alter the contents of the component.

To prove our point, we'll be using yet another component inside our link editor: a simple CAPTCHA component. This will be implemented in the final section, but used here as a black box.

The first step is to make the necessary additions and modifications to WAReddit to accommodate the link editor component. We add an instance variable called `linkEditor` with its accessors.

```
WAReddit>>renderContentOn: html
    html heading: 'Reddit.st'.
    html heading level: 3; with: 'In 10 elegant Smalltalk classes'.
    self renderActionsOn: html.
    self linkEditor notNil
        ifTrue: [ html render: self linkEditor ].
    self renderHighestRankingLinksOn: html.
    self renderLatestLinksOn: html

WAReddit>>renderActionsOn: html
    html paragraph: [
        html anchor callback: [ ]; with: 'Refresh'.
        self linkEditor isNil ifTrue: [
            html anchor callback: [ self showNewLinkEditor ]; with: 'New
Link' ] ]

WAReddit>>showNewLinkEditor
    self linkEditor: WARedditLinkEditor new.
    self linkEditor onAnswer: [ :answer |
        answer ifTrue: [
            self session glorpSession inUnitOfWorkDo: [ :session |
                session register: self linkEditor createLink ] ].
        self linkEditor: nil ]

WAReddit>>children
    ^ self linkEditor notNil
        ifTrue: [ Array with: self linkEditor ]
        ifFalse: [ super children ]
```

There are 2 possible states: either we have a link editor subcomponent or not. So the main `renderContentOn:` method conditionally asks the link editor to render itself. Likewise, in `renderActionsOn:` the 'New Link' anchor is only rendered when there is no link editor yet.

In the `showNewLinkEditor` action method we instantiate our subcomponent and hook it up. We could have reused just one instance, creating a new one is easier and clearer. The wiring is done by supplying a block to `onAnswer:`. A component can answer a value, in our case true or false for save or cancel respectively. So when the link editor answers true, we save a new link object and hide the editor.

In Seaside, the `children` method is a hook method that has to be implemented to list all subcomponents. Again this happens conditionally.

We can now implement the component itself: `WARedditLinkEditor` is a subclass of `WComponent` with 3 instance variables and their accessors: `url`, `title` and `captcha`.

```

[WARRedditLinkEditor>>renderContentOn: html
  html form: [
    html paragraph: 'Please enter a URL and title for the link that
      you want to add:'.
    html textInput size: 48; title: 'The URL of the new link'; on:
      #url of: self.
    html textInput size: 48; title: 'The title of the new link'; on:
      #title of: self.
    html render: self captcha.
    html submitButton on: #cancel of: self.
    html submitButton on: #save of: self ]

[WARRedditLinkEditor>>initialize
  super initialize.
  self url: 'http://';
  title: 'title';
  captcha: WARRedditCaptcha new

[WARRedditLinkEditor>>children
  ^ Array with: self captcha

[WARRedditLinkEditor>>updateRoot: anHtmlRoot
  super updateRoot: anHtmlRoot.
  anHtmlRoot title: 'Reddit.st - Submit a new link'.
  anHtmlRoot stylesheet url: (RedditFileLibrary urlOf: #mainCss)

[WARRedditLinkEditor>>cancel
  self answer: false

[WARRedditLinkEditor>>save
  self isUrlMissing ifTrue: [ ^ self inform: 'Please enter an URL' ].
  self isTitleMissing ifTrue: [ ^ self inform: 'Please enter a
    title' ].
  self captcha isSolved ifFalse: [ ^ self inform: 'Please answer the
    correct sum using digits' ].
  self isValid ifFalse: [ ^ self inform: 'The URL you entered did
    not resolve' ].
  self answer: true

[WARRedditLinkEditor>>isTitleMissing
  ^ self title isNil or: [ self title isEmpty or: [ self title =
    'title' ] ]

[WARRedditLinkEditor>>isUrlMissing
  ^ self url isNil or: [ self url isEmpty or: [ self url = 'http://'
    ] ]

[WARRedditLinkEditor>>isValid
  ^ (WebClient httpGet: self url) isSuccess

[WARRedditLinkEditor>>createLink
  ^ RedditLink withUrl: self url title: self title

```

Most of the code should be familiar by now. New is how cancel and save use `answer:` to return to whoever called upon this component. Before save returns successfully, a number of validation tests are done. When one of these tests fails, a message is shown and the operation is aborted. The `isUrlValid` method actually tries to resolve the URL. Finally, `createLink` instantiates a new `RedditLink` instance based on the valid fields entered by the user. Note how the CAPTCHA is used as a true component.

1.10 WARedditCaptcha: A last Web Component

The last and simplest web component is a CAPTCHA that presents a simple addition in words. This component does not need answer logic. The class `WARedditCaptcha` is again a subclass of `WComponent` with the following instance variables and accessors: `x`, `y`, and `sum`.

```
WARedditCaptcha>>renderContentOn: html
  self x: 10 atRandom.
  self y: 10 atRandom.
  html paragraph: 'CAPTCHA: How much is ',
    self x asWords, ' plus ', self y asWords, ' ?'.
  html textInput
    title: 'This functions as a CAPTCHA, type the answer using
      digits';
    on: #sum of: self

WARedditCaptcha>>initialize
  super initialize.
  self x: 0; y: 0; sum: 0

WARedditCaptcha>>isSolved
  ^ self sum asInteger = (self x + self y)
```

Each time the CAPTCHA is rendered, `x` and `y` get a new random value between 1 and 10. Next, the addition is presented in words. The `isSolved` method checks if the user answered correctly.

1.11 Conclusion

The source code discussed in this chapter is available the `SqueakSource` project called `ADayAtTheBeach`. Look for the package called `Reddit`.

Measuring the quality of the design is often a challenge. We believe that having classes with clear responsibilities. This is what we applied when designing such little application. `Reddit.st` consists of 10 classes for a total of 8 class methods and 75 instance methods. More than half are just one (1) line long, the rest averages just a few lines.