Script:Basic software engineering for Haskell

Headline

Basic software engineering practices for Haskell

Summary

Basic software engineering principles are instantiated for Haskell. That is, Haskell programs aremodularized (based on a module system), organized (in terms of scoping), documented (based on idiomatic comments), packaged (in terms of dependencies), and tested (specifically unit-tested). To this end, Haskell's where clauses and itsmodule system as well as the Haskell technologies Technology:Haddock, Technology:Cabal, Technology:HackageDB, and Technology:HUnit are leveraged.

Concepts

- Software engineering
- Modularization
- Module
- Local scope
- Build tool
- Package
- Haskell package
- Package management system
- Testing
- Unit testing
- Testing framework
- Documentation
- Documentation generation
- Documentation generator

Languages

- Language:Haskell

Technologies

- Technology:Cabal
- Technology:HackageDB
- Technology:Haddock
- Technology:HUnit

Features

- Feature:Total
- Feature:Median
- Feature:History

Contributions

- Contribution:haskellStarter
- Contribution:haskellEngineer
- Contribution:haskellBarchart

Metadata

- Course:Lambdas in Koblenz
- https://github.com/tibbe/haskell-style-guide/blob/master/haskell-style.md
- Script:First steps in Haskell

Build tool

Headline

A tool for build automation

Metadata

- http://en.wikipedia.org/wiki/Build_automation
- Software technology
- Vocabulary:Software engineering
- Vocabulary:Programming

Documentation

Headline

Documentation accompanying programs or software systems

Illustration

See Contribution:haskellEngineer for Language:Haskell style documentation based on Technology:Haddock.

Citation

(http://en.wikipedia.org/wiki/Software_documentation, 2 May 2013)

Software documentation or source code documentation is written text that accompanies computer software. It either explains how it operates or how to use it, and may mean different things to people in different roles.

Metadata

- http://en.wikipedia.org/wiki/Software_documentation
- Vocabulary:Software engineering
- Vocabulary:Programming
- Concept

Documentation generation

Headline

Application of a documentation generator

Metadata

- Documentation generator
- Vocabulary:Software engineering
- Vocabulary:Programming
- Concept

Documentation generator

Headline

A programming tool generating documentation

Illustration

See Contribution:haskellEngineer for Language:Haskell style of documentation based on Technology:Haddock.

Citation

(http://en.wikipedia.org/wiki/Software_documentation, 2 May 2013)

A documentation generator is a programming tool that generates software documentation intended for programmers (API documentation) or end users (End-user Guide), or both, from a set of specially commented source code files, and in some cases, binary files.

Metadata

- https://en.wikipedia.org/wiki/Documentation_generator
- Software technology
- Vocabulary:Software engineering
- Vocabulary:Programming

Haskell package

Headline

A distribution unit for Haskell

Metadata

- Vocabulary:Haskell
- Technology:Cabal
- Technology:HackageDB
- Package

Modularization

Headline

The process towards modular software

Illustration

Consider, for example, Contribution:haskellStarter, which is a non-modular (Language:Haskell-based) implementation of the 101system. Also, consider Contribution:haskellEngineer, which is a modular (Language:Haskell-based) implementation. In fact, the latter was obtained from the former by modularization. That is, both implementations implement the same features with the same code, except that the former implementation collects all code in one monolithic module, whereas the latter separates concerns by dedicating modules to the different implemented features.

Metadata

- Modularity
- Modular programming
- Separation of concerns
- Vocabulary:Programming
- Vocabulary:Software engineering
- Concept

Module

Headline

A unit of composition and separation of concerns

Illustration

Consider the following "naive" (inefficiently recursive) definition of the Fibonacci numbers in Language:Haskell:

```
-- Naive definition of Fibonacci numbers
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib x = fib (x-2) + fib (x-1)
```

For what it matters, we may place the function in a module as follows:

```
module Fibonacci.Inefficient where

-- The Fibonacci numbers
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib x = fib (x-2) + fib (x-1)
```

Suppose you want to compute the Fibonacci numbers more efficiently. To this end, you need an auxiliary function which keeps track of the two previous Fibonacci numbers so that binary (exponential) recursion can be avoided:

```
-- The Fibonacci numbers
fib :: Int -> Int
fib x = fib2 x 0 1

-- Helper function for efficient Fibonacci numbers
fib2 :: Int -> Int -> Int -> Int
fib2 0 y _ = y
fib2 1 _ y = y
fib2 x y1 y2 = fib2 (x-1) y2 (y1+y2)
```

These two functions are reasonably embedded into a module as follows:

```
module Fibonacci.Efficient (
  fib
) where

-- The Fibonacci numbers
fib :: Int -> Int
fib x = fib2 x 0 1

-- Helper function for efficient Fibonacci numbers
fib2 :: Int -> Int -> Int -> Int
fib2 0 y _ = y
fib2 1 _ y = y
fib2 x y1 y2 = fib2 (x-1) y2 (y1+y2)
```

Notably, the primary function for the Fibonacci numbers is exported while the helper function isn't. In this manner, we express that one function is of general interest whereas the other one is an implementational artifact.

Metadata

- Vocabulary:Programming
- Vocabulary:Software architecture
- Abstraction mechanism
- Modularization
- Modular_programming
- http://en.wikipedia.org/wiki/Modular_programming
- http://www.haskell.org/tutorial/modules.html

Package

Headline

A unit of distribution of software

Illustration

See, for example, the package database ofTechnology:HackageDB for Language:Haskell.

Metadata

- Abstraction mechanism
- http://en.wikipedia.org/wiki/Package_(package_management_system)
- Vocabulary:Software engineering

Package management system

Headline

A system for managing packages

Metadata

- http://en.wikipedia.org/wiki/Package_management_system
- Vocabulary:Software engineering
- Software technology

Contribution:haskellStarter

Headline

Basic functional programming in Language:Haskell.

Characteristics

The contribution demonstrates basic style of functional programming in Language:Haskell. Only very basic language constructs are exercised. Companies are represented via tuples over primitive data types. (No algebraic data types are used; type synonyms suffice.) Only flat companies are modeled, i.e., nested departments are not modeled.Pure, recursive functions implement operations for totaling and cutting salaries by pattern matching. The types for companies readily implement read and show functions for closed serialization.

Illustration

101companies contribution haskellStarter

The data model relies on tuples for data composition:

```
-- | Companies as pairs of company name and employee list
type Company = (Name, [Employee])

-- | An employee consists of name, address, and salary
type Employee = (Name, Address, Salary)
```

Basic types for strings and floats are leveraged for names, addresses, and salaries.

```
-- | Addresses as strings
type Address = String

-- | Total all salaries in a company
total :: Company -> Float
total = sum . salaries

-- | Salaries as floats
type Salary = Float
```

A sample company looks like this:

```
-- | A sample company useful for basic tests
sampleCompany :: Company
sampleCompany =
  ( "Acme Corporation",
    [
      ("Craig", "Redmond", 123456),
      ("Erik", "Utrecht", 12345),
      ("Ralf", "Koblenz", 1234),
      ("Ray", "Redmond", 234567),
      ("Klaus", "Boston", 23456),
      ("Karl", "Riga", 2345),
      ("Joe", "Wifi City", 2344)
    ]
```

```
 )
```

Features for functional requirements are implemented by families of functions on the company types. For instance, Feature:Total is implemented as follows:

```
-- | Total all salaries in a company
total :: Company -> Float
total = sum . salaries

-- | Extract all salaries in a company
salaries :: Company -> [Salary]
salaries (n, es) = getSalaries es

-- | Extract all salaries of lists of employees
getSalaries :: [Employee] -> [Salary]
getSalaries [] = []
getSalaries (e:es) = getSalary e : getSalaries es

-- | Extract the salary from an employee
getSalary :: Employee -> Salary
getSalary (_, _, s) = s
```

We may test these functions with the followingfunction application:

```
total sampleCompany
```

The function application evaluates to the following total:

```
399747.0
```

All the remaining functions are implemented in the same module:

```
-- | Companies as pairs of company name and employee list
type Company = (Name, [Employee])

-- | An employee consists of name, address, and salary
type Employee = (Name, Address, Salary)

-- | Names as strings
type Name = String

-- | Addresses as strings
type Address = String

-- | Salaries as floats
type Salary = Float

-- | A sample company useful for basic tests
sampleCompany :: Company
sampleCompany =
  ( "Acme Corporation",
    [
      ("Craig", "Redmond", 123456),
      ("Erik", "Utrecht", 12345),
      ("Ralf", "Koblenz", 1234),
      ("Ray", "Redmond", 234567),
      ("Klaus", "Boston", 23456),
      ("Karl", "Riga", 2345),
      ("Joe", "Wifi City", 2344)
    ]
  )

-- | Total all salaries in a company
total :: Company -> Float
total = sum . salaries

-- | Extract all salaries in a company
salaries :: Company -> [Salary]
salaries (n, es) = getSalaries es

-- | Extract all salaries of lists of employees
getSalaries :: [Employee] -> [Salary]
getSalaries [] = []
getSalaries (e:es) = getSalary e : getSalaries es

-- | Extract the salary from an employee
getSalary :: Employee -> Salary
```

```
getSalary (_, _, s) = s

-- | Cut all salaries in a company
cut :: Company -> Company
cut (n, es) = (n, cutEmployees es)

-- | Cut salaries for lists of employees
cutEmployees :: [Employee] -> [Employee]
cutEmployees [] = []
cutEmployees (e:es) = cutEmployee e : cutEmployees es

-- | Cut the salary of an employee in half
cutEmployee :: Employee -> Employee
cutEmployee (n, a, s) = (n, a, s/2)

-- | Illustrative function applications
main = do
  print (total sampleCompany)
  print (total (cut sampleCompany))
```

Relationships

In the interest of maintaining a very simple simple beginner's example, the present contribution is the only contribution which does not commit to modularization, packaging, unit testing. See Contribution:haskellEngineer for a modularized and packaged variation with also unit tests added.

Architecture

The contribution only consists of a single module "Main.hs" which includes all the code as shown above.

Usage

See a designated README.

Metadata

- Language:Haskell
- Language:Haskell 98
- Technology:GHCi
- Feature:Flat company
- Feature:Closed serialization
- Feature:Total
- Feature:Cut
- Contributor:rlaemmel
- Theme:Starter
- Theme:Haskell introduction
- Theme:Haskell data

Scoping

Headline

The placement of abstractions in appropriate scopes

Illustration

See the notion of local scope for an illustration of a specific scoping option.

Metadata

- http://en.wikipedia.org/wiki/Scope_(computer_science)
- Concept

Software engineering

Headline

The application of engineering to software

Citation

(http://en.wikipedia.org/wiki/Software_engineering, 2 May 2013)

Software engineering (SE) is the application of a systematic, disciplined, quantifiable approach to the design, development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software.

Metadata

- http://en.wikipedia.org/wiki/Software_engineering
- Vocabulary:Software engineering
- Concept

Testing framework

Headline

A framework for authoring and running automated tests

Metadata

- Software technology
- http://en.wikipedia.org/wiki/Test_automation_framework
- http://c2.com/cgi/wiki?TestingFramework

Testing

Headline

Testing software systems or programs

Illustration

See unit testing, for example.

Metadata

- Vocabulary:Software engineering
- Vocabulary:Programming
- http://en.wikipedia.org/wiki/Software_testing
- Concept

Contribution:haskellBarchart

Headline

Analysis of historical company data with Language:Haskell

Characteristics

Historical data is simply represented as list of year-value pairs so that company data is snapshotted for a number of years and any analysis of historical data can simply map over the versions. A simple chart package for Haskell is leveraged to visualize the development of salary total and median over the years. In this manner, the contribution demonstrates how to declare external dependences via Technology:Cabal. Further, the contribution also demonstrates modularization and code organization. In particular, where clauses for local scope and export/import clauses for modularization are used carefully.

Illustration

We would like to generate barcharts as follows:

These barcharts are generated by the following functionality. Given a filename, a title (such as "Total" or "Median") and a year-to-data mapping, generate a PNG file with the barchart for the distribution of the data.

```
-- | Generate .png file for development of median
chart :: String -> String -> [(Int,Float)] -> IO ()
chart filename title values = do
  let fileoptions = FileOptions (640,480) SVG empty
  renderableToFile fileoptions (toRenderable layout) filename
  return ()
    where
      layout
        = def
        & layout_title .~ "Development of salaries over the years"
        & layout_plots .~ [plotBars bars]
      bars
        = def
        & plot_bars_titles  .~ [title]
        & plot_bars_spacing .~ BarsFixGap 42 101
        & plot_bars_style   .~ BarsStacked
        & plot_bars_values  .~ values'
      values'
        = map (\(y,f) -> (y, [float2Double f])) values
```

Metadata

- Feature:Flat company
- Feature:Total
- Feature:Median
- Feature:History
- Contributor:rlaemmel
- http://hackage.haskell.org/package/Chart-0.16
- Contribution:haskellEngineer

Contribution:haskellEngineer

Headline

Basic software engineering for Haskell

Characteristics

The contribution demonstrates basic means of modularization (using Haskell's nativemodule system), code organization (using where clauses for local scope), packaging (using Technology:Cabal), documentation (using Technology:Haddock), and unit testing (using Technology:HUnit). Other than that, only basic language constructs are exercised and a very limited feature set of the 101system is implemented. The contribution is indeed more of a showcase for a pattern for modularization, code organization, packaging, documentation, and unit testing.

Illustration

Modular organization

The contribution consists of the following modules as listed in name: haskellEngineer version: 0.1.0.0 synopsis: Basic software engineering for Haskell homepage: http://101companies.org/wiki/Contribution:haskellEngineer build-type: Simple cabal-version: >=1.9.2 library exposed-modules: Main Company.Data Company.Sample Company.Total Company.Cut build-depends: base >=4.4 && < 5.0, HUnit hs-source-dirs: src test-suite basic-tests main-is: Main.hs build-depends: base, HUnit hs-source-dirs: src type: exitcode-stdio-1.0 :

```
Main
Company.Data
Company.Sample
Company.Total
Company.Cut
```

The modules implement features as follows:

- {-| A data model for the 101companies System -}

module Company.Data where -- | Companies as pairs of company name and employee listtype Company = (Name, [Employee]) -- | An employee consists of name, address, and salarytype Employee = (Name, Address, Salary) -- | Names as strings type Name = String -- | Addresses as stringstype Address = String -- | Salaries as floatstype Salary = Float : Feature:Flat company.

- Company/Sample.hs: A sample company.
- Company/Total.hs: Feature:Total.
- Company/Cut.hs: Feature:Cut.
- Main.hs: Unit tests for demonstration.

For instance, the implementation of Feature:Total takes this form:

```
{-| The operation of totaling all salaries of all employees in a company -}

module Company.Total where

import Company.Data

-- | Total all salaries in a company
total :: Company -> Float
total = sum . salaries
  where

    -- Extract all salaries in a company
    salaries :: Company -> [Salary]
    salaries (_, es) = getSalaries es
      where

        -- Extract all salaries of lists of employees
        getSalaries :: [Employee] -> [Salary]
        getSalaries [] = []
        getSalaries (e:es) = getSalary e : getSalaries es
          where

            -- Extract the salary from an employee
```

```
getSalary :: Employee -> Salary
getSalary (_, _, s) = s
```

Please note how "where clauses" are used to organize the declarations in such a way that it is expressed what function is a helper function to what other function. The declaration of such local scope also implies that the helper functions do not feed into the interface of the module.

Haddock comments

Technology:Haddock comments are used to enable documentation generation. Consider again the module shown above. Haddock comments are used for the functions *total* and *salaries* but not for the remaining functions, as they are not exported and thus, they do not need to be covered by the generated documentation.

External dependencies

The contribution has the following dependencies as listed in name: haskellEngineer version: 0.1.0.0 synopsis: Basic software engineering for Haskell homepage: http://101companies.org/wiki/Contribution:haskellEngineer build-type: Simple cabal-version: >=1.9.2 library exposed-modules: Main Company.Data Company.Sample Company.Total Company.Cut build-depends: base >=4.4 && < 5.0, HUnit hs-source-dirs: src test-suite basic-tests main-is: Main.hs build-depends: base, HUnit hs-source-dirs: src type: exitcode-stdio-1.0 :

build-depends: base >=4.4 && < 5.0, HUnit

These packages serve the following purposes:

- base: This is the Haskell base package; a range of versions is permitted.
- HUnit: This is the package for Technology:HUnit; its version is not explicitly constrained.

HUnit testcases

The contribution is tested by the following test cases:

```
-- | The list of tests
tests =
  TestList [
    TestLabel "total" totalTest,
    TestLabel "cut" cutTest,
    TestLabel "serialization" serializationTest
  ]
```

For instance, the test case for serialization looks as follows:

```
-- | Test for round-tripping of de-/serialization of sample company
serializationTest = sampleCompany ~=? read (show sampleCompany)
```

Relationships

- The present contribution is an "engineered" variation on Contribution:haskellStarter. That is, modularization, packaging, documentation, and unit testing was applied.
- Several other contributions derive from the present contribution more or less directly by demonstrating additional language or technology capabilities or implementing additional features of the 101system.

Architecture

Modules to feature mapping:

- Company.Data: Feature:Flat company
- Company.Sample: A sample company
- Company.Total: Feature:Total
- Company.Cut: Feature:Cut
- Main: Unit tests for demonstration

Usage

See https://github.com/101companies/101haskell/blob/master/README.md

Metadata

- Language:Haskell
- Language:Haskell 98
- Technology:GHC
- Technology:Cabal
- Technology:HUnit
- Technology:Haddock
- Feature:Flat company
- Feature:Closed serialization
- Feature:Total
- Feature:Cut
- Contributor:rlaemmel
- Theme:Haskell introduction
- Contribution:HaskellStarter

Feature:History

Headline

Maintain and analyze historical company data

Description

Company data is to be maintained for a number of years so that some analysis are applicable. Specifically, management is interested in the development of salaries in terms of the total and the median of salaries over the years. To this end, the data may be visualized via a bar chart.

Motivation

The feature triggers the need to deal with historical data, which, in a practical setting would need to be obtained by regular snapshotting or the use of a temporal database. Simple implementations of the feature may simply assume sufficient historical data via test data. More interesting implementations may also model (simulate) the process of obtaining historical data.

Illustration

See Contribution:haskellBarchart for an illustration.

Relationships

The feature leverages Feature:Total and Feature:Median.

Metadata

- http://en.wikipedia.org/wiki/Temporal_database
- Functional requirement
- Data requirement
- Optional feature
- Feature:Total
- Feature:Median

Feature:Median

Headline

Compute the median of the salaries of all employees

Description

Management would like to know the median of all salaries in a company. This value may be used for decision making during performance interviews, e.g., in the sense that any employee who has shown exceptional performance gets a raise, if the individual salary is well below the current median. Further, the median may also be communicated to employees so that they can understand their individual salary on the salary scale of the company. In practice, medians of more refined groups of employees would be considered, e.g., employees with a certain job role, seniority level, or gender.

Motivation

This feature triggers a very basic statistical computation, i.e., the computation of the median of a list of sorted values. Of course, the median is typically available as a primitive or from a library, but when coded explicitly, it is an exercise in list processing. This feature may also call for reuse such that code is shared with the implementation of Feature:Total because both features operate on the list of all salaries.

Illustration

The following code stems from Contribution:haskellStarter:

```
-- Median of all salaries in a company
median :: Company -> Salary
median = medianSorted . sort . salaries
```

First, the salaries are to be extracted from the company. Second, the extracted salaries are to be sorted, where a library function *sort* is used here. Third, the sorted list of salaries is to be processed to find the median.

```
-- Extract all salaries in a company
salaries :: Company -> [Salary]
salaries (n, es) = getSalaries es

-- Extract all salaries of lists of employees
getSalaries :: [Employee] -> [Salary]
getSalaries [] = []
getSalaries (e:es) = getSalary e : getSalaries es

-- Extract the salary from an employee
getSalary :: Employee -> Salary
getSalary (_, _, s) = s

-- Median of a sorted list
medianSorted [] = error "Cannot compute median on empty list."
medianSorted [x] = x
medianSorted [x,y] = (x+y)/2
medianSorted l = medianSorted (init (tail l))
```

Relationships

- See Feature:Total for another query scenario which also processes the salaries of all employees in a company.

Metadata

- Functional requirement
- Optional feature
- Query

Feature:Total

Headline

Sum up the salaries of all employees

Description

The salaries of a company's employees are to be summed up. Let's assume that the management of the company is interested in the salary total as a simple indicator for the amount of money paid to the employees, be it for a press release or otherwise. Clearly, any real company faces other expenses per employee, which are not totaled in this manner.

Motivation

The feature may be implemented as a query, potentially making use of a suitable query language. Conceptually, the feature corresponds to a relatively simple and regular kind of query, i.e., an iterator-based query, which iterates over a company' employees and aggregates the salaries of the individual employees along the way. It shall be interesting to see how different software languages, technologies, and implementations deal with the conceptual simplicity of the problem at hand.

Illustration

Totaling salaries in SQL

Consider the following Language:SQL query which can be applied to an instance of a straightforward relational schema for companies. We assume that all employees belong to a single company; The snippet originates from Contribution:mySqlMany.

```
SELECT SUM(salary) FROM employee;
```

Totaling salaries in Haskell

Consider the following Language:Haskell functions which are applied to a simple representation of companies.

```
-- Total all salaries in a company
total :: Company -> Float
total = sum . salaries

-- Extract all salaries in a company
salaries :: Company -> [Salary]
salaries (n, es) = salariesEs es

-- Extract all salaries of lists of employees
salariesEs :: [Employee] -> [Salary]
salariesEs [] = []
salariesEs (e:es) = getSalary e : salariesEs es

-- Extract the salary from an employee
getSalary :: Employee -> Salary
getSalary ( ,  , s) = s
```

Relationships

- See Feature:Cut for a transformation scenario instead of a query scenario.
- See Feature:Depth for a more advanced query scenario.
- The present feature should be applicable to any data model of companies, specificallyFeature:Flat company and Feature:Hierarchical_company.

Guidelines

- The *name* of an operation for summing up salaries thereof should involve the term "total". This guideline is met by the above illustration, if we assume that the shown SQL statement is stored in a SQL script with name "Total.sql". By contrast, if OO programming was used for implementation, then the names of the corresponding methods should involve the term "total".
- A suitable *demonstration* of the feature's implementation should total the salaries of a sample company. This guideline

is met by the above illustration, if we assume that the shown SQL statement is executed on a database which readily contains company data. All such database preparation and query execution should preferably be scripted. Likewise, if OO programming was used, then the demonstration could be delivered in the form of unit tests.

Metadata

- Optional feature
- Functional requirement
- Aggregation

Technology:Cabal

Headline

A build automation tool for Language:Haskell

Illustration

Consider the Hello world program for Haskell:

```
main = putStrLn "Hello, world!"
```

Now let's do packaging and build automation for this program. To this end, we set up the following Cabal file:

```
-- Initial helloWorld.cabal generated by cabal init.  For further
-- documentation, see http://haskell.org/cabal/users-guide/

name:                helloWorld
version:             0.1.0.0
synopsis:             Demonstration of Cabal
description:          Just filled in to make "cabal check" go quiet.
homepage:               http://101companies.org/wiki/Technology:Cabal
license:             MIT
license-file:        LICENSE
author:              Joe Hackathon
maintainer:            101companies@gmail.com
-- copyright:
category:            Testing
build-type:          Simple
cabal-version:       >=1.8

executable helloWorld
  main-is:             Main.hs
  other-modules:
  build-depends:       base ==4.5.*
  hs-source-dirs:      src
```

As one can see at the top, the initial file was actually generated with "cabal init" such that some parameters are filled in interactively, but a few subsequent modifications were applied to the file manually.

With this Cabal file in place, the program can be built and ran at the command line as follows:

```
cabal configure
cabal build
dist/build/helloWorld/helloWorld
```

The configure step checks the Cabal file and resolves external dependencies, if necessary. The build step compiles all involved modules. Thus, an executable can be invoked in the last step.

Metadata

- Build tool
- Haskell technology
- http://www.haskell.org/cabal/
- Technology:HackageDB

Technology:HUnit

Headline

A testing framework for unit testing for Language:Haskell

Illustration

Consider the following test suite that tests some properties of logical negation forLanguage:Haskell with the help of Technology:HUnit:

```
import Test.HUnit

-- The tests
tests =
    TestList [
      TestLabel "notNotTrue" (doubleNegation True),
      TestLabel "notNotFalse" (doubleNegation False)
    ]
  where
   doubleNegation x = x ~=? not (not x)

-- Run all tests
main = runTestTT tests
```

Thus, there are two test cases, one for double negation of*True* and another one for *False* as the operand. The helper function *doubleNegation* illustrates the structure of test cases. That is, an expected value is compared with the actual value as described by an expression or computation. The specific operator "~=?" represents equality but it makes up for monitored test-case execution.

Metadata

- http://hunit.sourceforge.net/
- http://www.haskell.org/haskellwiki/HUnit_1.0_User's_Guide
- http://hackage.haskell.org/package/HUnit
- Testing framework
- Unit testing
- Namespace:Technology

Technology:HackageDB

Headline

A collection of releases of Language:Haskell packages

Illustration

Have a look at the website of HackageDB:

http://hackage.haskell.org/packages/hackage.html

Specifically, have a look at the packages available at HackageDB:

http://hackage.haskell.org/packages/archive/pkg-list.html

For instance, here is a pointer to a specific package forSYB style of generic programming:

http://hackage.haskell.org/package/syb

Metadata

- http://hackage.haskell.org/
- Source code repository
- Technology:Cabal
- Namespace:Technology

Technology:Haddock

Headline

A documentation generator for Language:Haskell

Illustration

Haddock relies on module headers and simple comment conventions to generate documentation from Haskell source code. Consider, for example, the following module:

```
{- | This comment is placed before the module header and thus
     is seen as the general description of the module. Since
     the general description may be a bit longer, it is quite
     common to see a multi-line comment in this position. -}

module Main (
 foo
) where

-- | The "|" character in the comment expresses that this
-- comment should contribute to the generated documentation.
-- Haddock does indeed search for such comments.
-- We note that 'foo' is indeed exported and thus it
-- deserves documentation. We could also use a multi-line
-- comment of course.
foo :: () -> ()
foo = id

-- This function is not exported.
-- Thus, no Haddock comment is needed.
-- That is, the function will not appear in generated documentation.
bar :: () -> Bool
bar = const True
```

It uses Haddock comment conventions for the module description and the exported function*foo*. Haddock supports much more conventions and features; see the documentation. Haddock also nicely integrates with Technology:Cabal such that one can simply invoke "cabal haddock" to generate documentation for a given Haskell package.

Metadata

- Documentation generator
- http://www.haskell.org/haddock
- Namespace:Technology

Unit testing

Headline

A method of software testing

Illustration

See Contribution:haskellEngineer for Language:Haskell style of unit testing based on Technology:HUnit.

Metadata

- http://en.wikipedia.org/wiki/Unit_testing
- Vocabulary:Software engineering
- Vocabulary:Programming
- Testing

Language:Haskell

Headline

The functional programming language Haskell

Details

There are plenty of Haskell-based contributions to the 101project. This is evident from corresponding back-links. More selective sets of Haskell-based contributions are organized in themes: Theme:Haskell data, Theme:Haskell potpourri, and Theme:Haskell genericity.

Metadata

- http://www.haskell.org/
- http://en.wikipedia.org/wiki/Haskell_(programming_language)
- Functional programming language
