Task Oriented Programming
with
itasks
dynamic workflow system

-

A Domain Specific Language embedded in
Clean

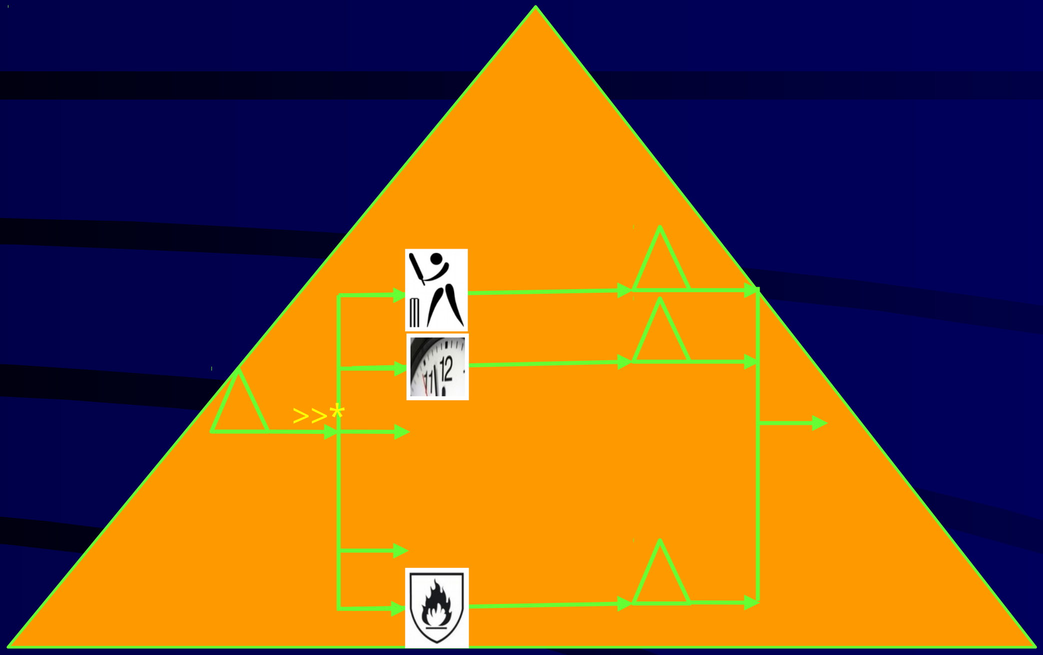
Rinus Plasmeijer – Bas Lijnse
Peter Achten – Pieter Koopman - Steffen Michels
Jurriën Stutterheim - Markus Klinik - Tim Steenvoorden - Mart Lubbers – Arjan Oortgiesen
Jan Martin Jansen (NLDA) – John van Groningen - Laszlo Domoszlai (ELTE)
Radboud University Nijmegen

Tasks, Tasks Combinators, Data Exchange

- Tasks
  - Basic Tasks
    - Non-interactive
      - return, throw, …
    - Interactive editors
      - interact , and derived combinators like enterInformation , showInformation , …
  - Combinators
    - Sequential
      - step, and derived combinators like >>* , >>= , >>|,…
    - Parallel
      - parallel, and derived combinators like -&&-, -||- , …
- Data exchange between tasks
  - Locally Observable Task Values: :: Task a
  - Globally Observable Shared Data Sources : :: RWShared r w

*Sequential Combinator : Step >>**

- (Task a) >>* [TaskCont a b] → Task b



- *Observe* Task a, continue with one of the Task b's:

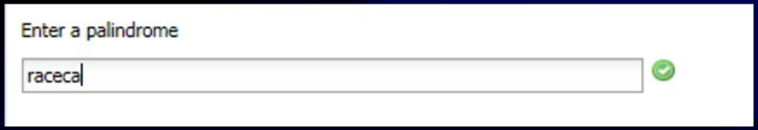
  - if a certain action is performed by the end-user (normal priority)

  - if the value of the observed task is satisfying a certain predicate (high priority)

  - or the observed task has raised an exception to be handled here (highest priority)

*Sequential Combinator: >>**

```
palindrome :: Task String
palindrome =   enterInformation "Enter a palindrome" []
```

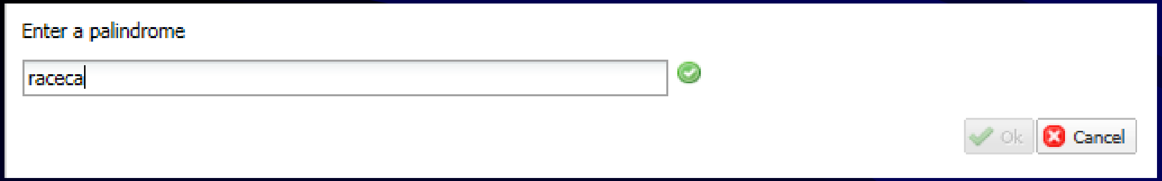


Enter a palindrome

raceca

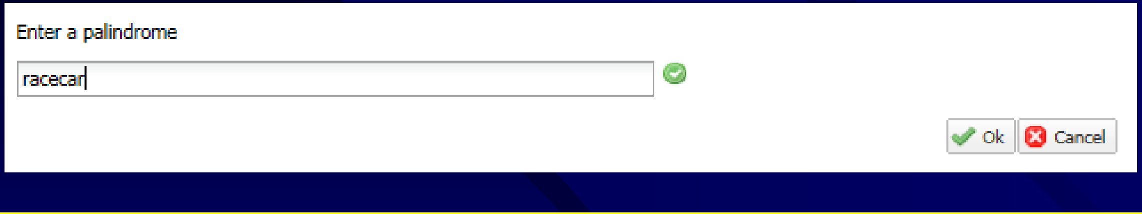
*Sequential Combinator: >>**

```
palindrome :: Task (Maybe String)
palindrome =   enterInformation "Enter a palindrome" []
            >>* [ OnAction  ActionOk      (ifValue isPalindrome  (\v → return (Just v)))
                ,  OnAction  ActionCancel (always                  (return Nothing))
                ]
```



Enter a palindrome

raceca



Enter a palindrome

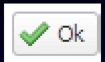
racecar

Sequential Step Combinator

- Combinator for *Sequential* Composition

```
(>>*) infixl 1 :: (Task a)  [TaskCont a b]  → Task b                | iTask a & iTask b

:: TaskCont a b
   =         OnAction Action     ((TaskValue a) → Maybe (Task b))
   |         OnValue         ((TaskValue a) → Maybe (Task b))
   |  E.e:  OnException  (e → Task b)                         & iTask e

:: Action                = Action String
```

```
ActionOk     :== Action "Ok"
```

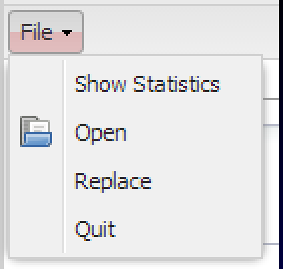


Sequential Step Combinator

- Combinator for *Sequential* Composition

```
(>>*) infixl 1 :: (Task a)  [TaskCont a b]  → Task b                 | iTask a & iTask b

:: TaskCont a b
   =         OnAction Action     ((TaskValue a) → Maybe (Task b))
   |         OnValue        ((TaskValue a) → Maybe (Task b))
   |  E.e:   OnException  (e → Task b)                       & iTask e

:: Action             = Action String
```

```
ActionOpen:== Action "/File/Open"
```



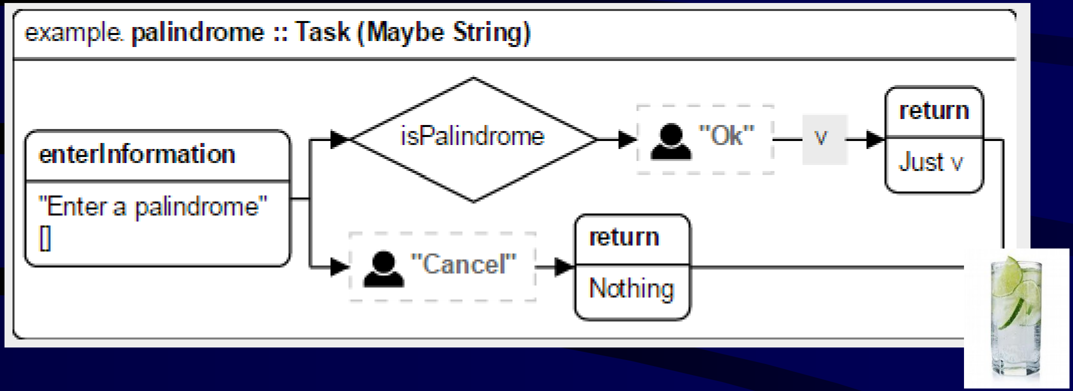
Sequential Step Combinator

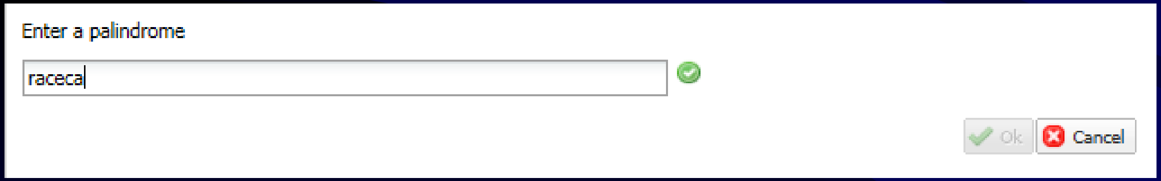
Combinator for *Sequential* Composition

```
(>>*) infixl 1 :: (Task a)  [TaskCont a b] → Task b              | iTask a & iTask b

:: TaskCont a b
 =          OnAction Action    ((TaskValue a) → Maybe (Task b))
   |        OnValue          ((TaskValue a) → Maybe (Task b))
   |  E.e:  OnException  (e → Task b)                              & iTask e
always :: (Task b)          (TaskValue a) → Maybe (Task b)
always taskb _              = Just taskb

ifValue :: (a → Bool) (a → Task b)  (TaskValue a) → Maybe (Task b)
ifValue pred ataskb (Value a _)       = if (pred a) (Just (ataskb a)) Nothing
ifValue _ _ _          = Nothing

hasValue  :: (a → Task b)        (TaskValue a) → Maybe (Task b)
hasValue  ataskb (Value a _)  = Just (ataskb a)
hasValue _ _                = Nothing

ifStable :: (a → Task b)   (TaskValue a) → Maybe (Task b)
ifStable ataskb (Value a True)         = Just (ataskb a)
ifStable _ _           = Nothing
```

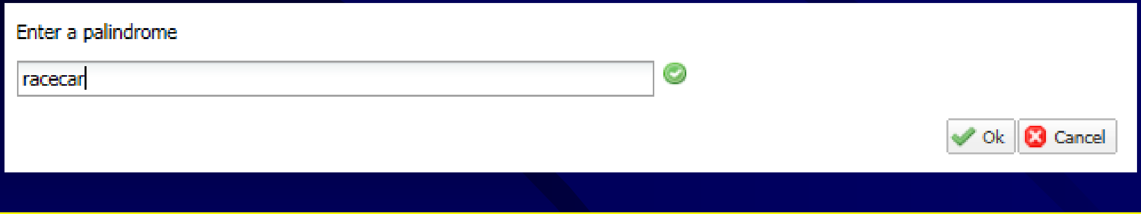
*Sequential Combinator: >>**

```
palindrome :: Task (Maybe String)
palindrome =   enterInformation "Enter a palindrome" []
               >>* [ OnAction  ActionOk      (ifValue isPalindrome  (\v → return (Just v)))
                   , OnAction  ActionCancel (always                  (return Nothing))
                   ]
```




Enter a palindrome

raceca


Enter a palindrome

racecar

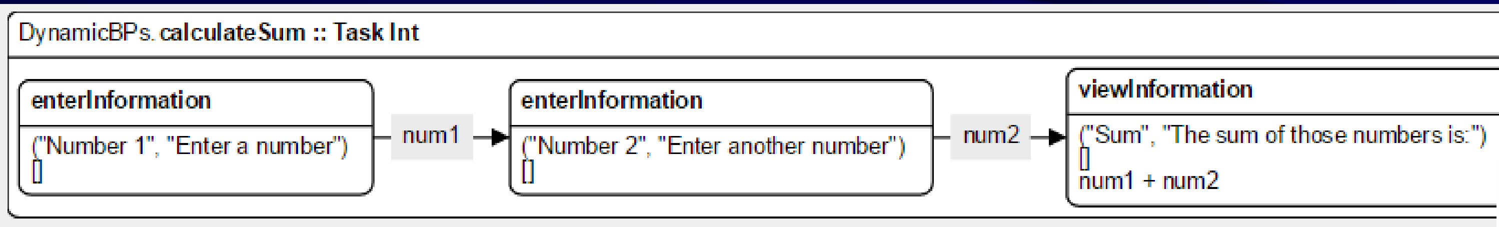
Derived Sequential Combinators: Monadic-style

Monadic style:

```
	(>>=)  infix  1       :: (Task a) (a → Task b) → Task b              | iTask a & iTask b
	return          :: a                        → Task a              | iTask a

	(>>|)  infix  1       :: (Task a) (Task b)              → Task b              | iTask a & iTask b


(>>=)  infix  1       :: (Task a) (a → Task b) → Task b              | iTask a & iTask b
(>>=) taska ataskb
	=               taska
	        >>*   [OnAction  ActionContinue      (hasValue  ataskb)
	              , OnValue                       (ifStable   ataskb)
	              ]
```

Simple Sum

```
calculateSum :: Task Int
calculateSum
  =                     enterInformation ("Number 1","Enter a number") []
         >>= \num1 →       enterInformation ("Number 2","Enter another number") []
         >>= \num2 →       viewInformation   ("Sum","The sum of those numbers is:") [] (num1 + num2)
```



DynamicBPs.calculateSum :: Task Int

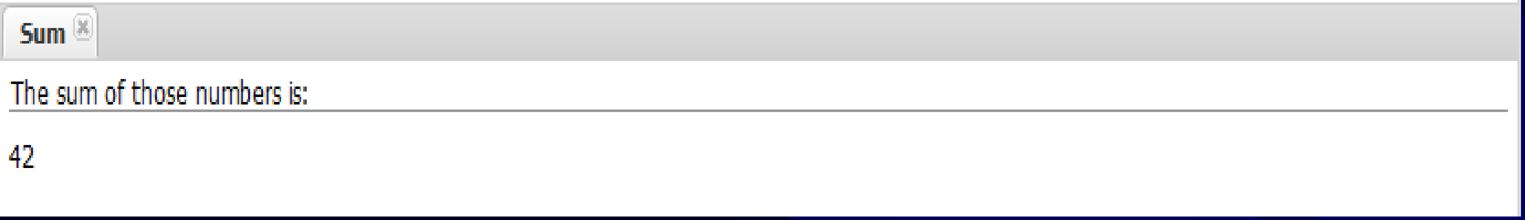
enterInformation
("Number 1", "Enter a number")
[]

num1

enterInformation
("Number 2", "Enter another number")
[]

num2

viewInformation
("Sum", "The sum of those numbers is:")
[]
num1 + num2



The sum of those numbers is:

42

Derived Combinators of the Parallel Combinator

Any thinkable parallel way of working can be expressed with *one-and-the-same Parallel Core – Combinator* !
Here are some handy derived instantiations:

and : return values of all (embedded) parallel tasks:

```
(-&&-) infixr 4     :: (Task a) (Task b)     → Task (a, b)        | iTask a & iTask b
allTasks            :: [Task a]              → Task [a]           | iTask a
```

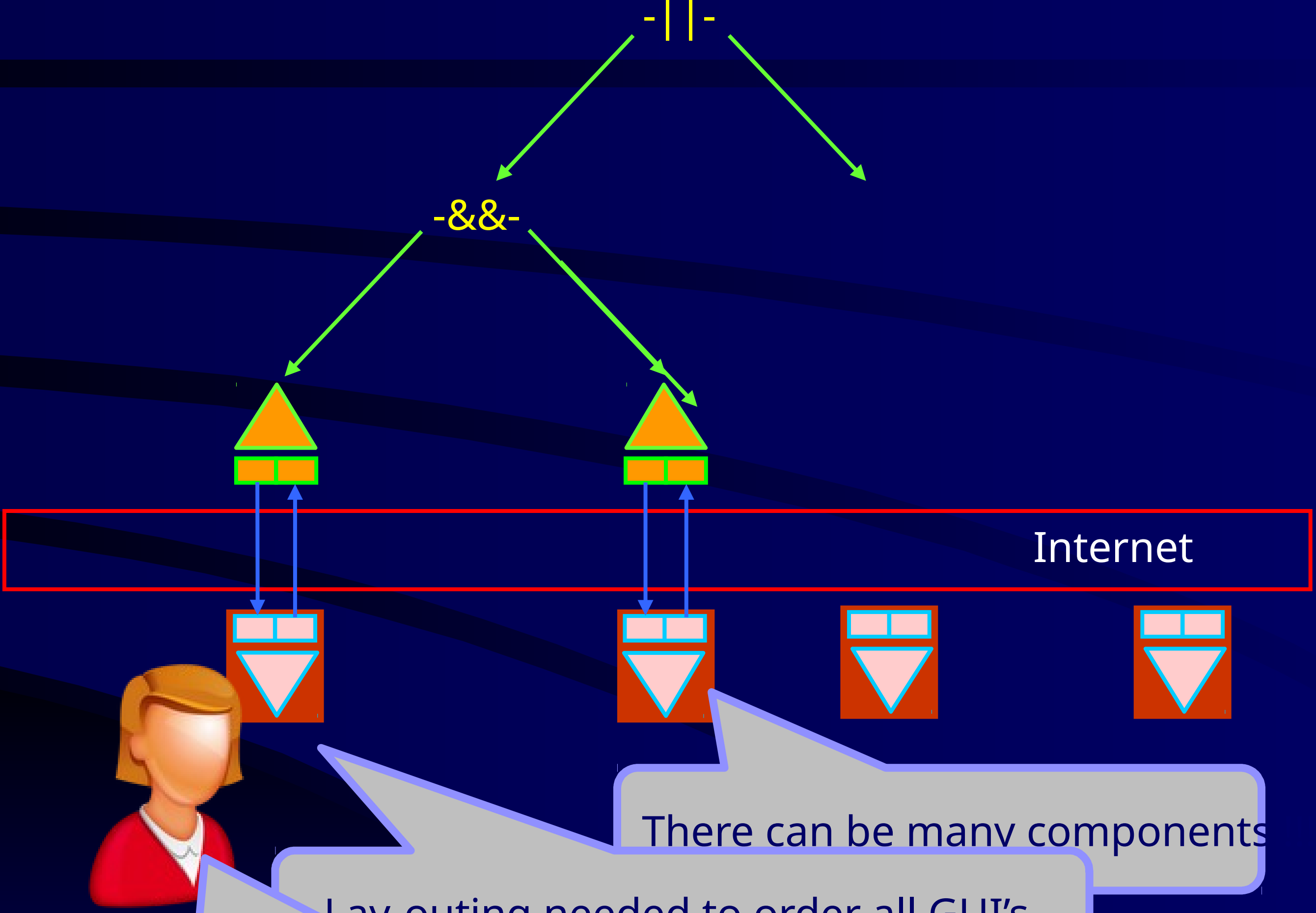
or: return result of (embedded) parallel tasks yielding a value as first:

```
(-||-) infixr 3     :: (Task a) (Task a)     → Task a             | iTask a
eitherTask          :: (Task a) (Task b)     → Task (Either a b)  | iTask a & iTask b
anyTask             :: [Task a]              → Task a             | iTask a
```

one-of: start two tasks, but we are only interested in the result of one of them, use the other to inform:

```
(-||)  infixl 3     :: (Task a) (Task b)     → Task a             | iTask a & iTask b
(||-)  infixr 3     :: (Task a) (Task b)     → Task b             | iTask a & iTask b
```

One User may have many parallel tasks to work on..



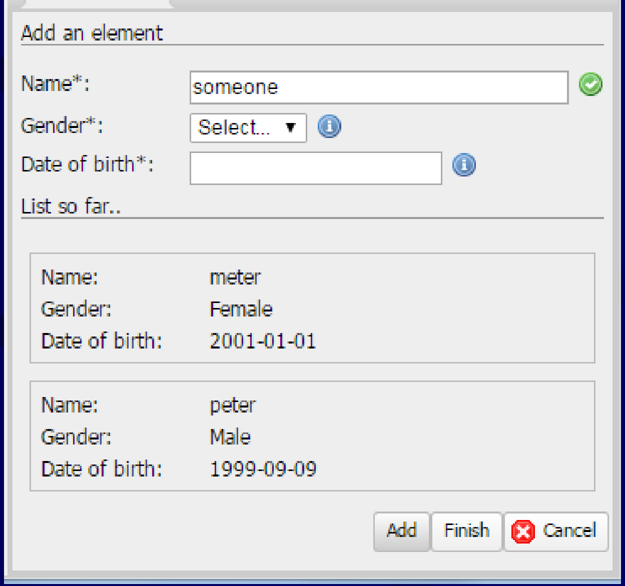
There can be many components

Lay-outing needed to order all GUI’s

Default lay-out algorithme + lay-out directives

Recursive Tasks

```
add1by1 :: [a] → Task [a]       | iTask a
add1by1 list_so_far
      =         enterInformation "Add an element" []
                  -||
                  viewInformation "List so far.." [] list_so_far
              >>*  [ OnAction  (Action "Add")     (hasValue  (\elem   → add1by1 [elem : list_so_far])
                 ,  OnAction  (Action "Finish")   (always (return list_so_far))
                 ,  OnAction  ActionCancel          (always (return []))
               ]

person1by1 :: Task [Person]
person1by1 = add1by1 []
```



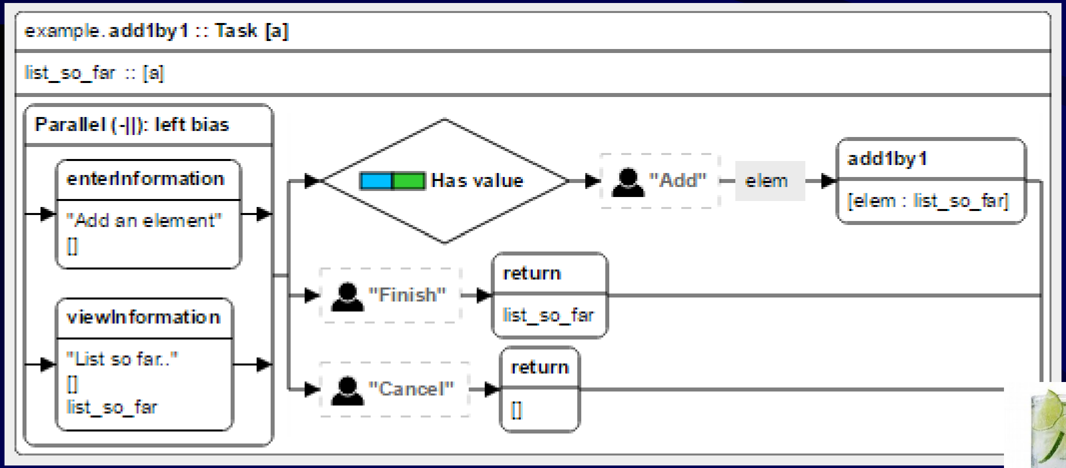
Add an element

Name*:	someone
Gender*:	Select... ▾
Date of birth*:	

List so far..

Name:	meter
Gender:	Female
Date of birth:	2001-01-01

Name:	peter
Gender:	Male
Date of birth:	1999-09-09



example.add1by1 :: Task [a]

list_so_far :: [a]

Derived Combinators of the Parallel Combinator

Any thinkable parallel way of working can be expressed with *one-and-the-same Parallel Core – Combinator* !
Here are some handy derived instantiations:

and : return values of all (embedded) parallel tasks:

```
(-&&-) infixr 4   :: (Task a) (Task b)   → Task (a, b)         | iTask a & iTask b
allTasks          :: [Task a]        → Task [a]        | iTask a
```

or: return result of (embedded) parallel tasks yielding a value as first:

```
(-||-) infixr 3   :: (Task a) (Task a)   → Task a              | iTask a
eitherTask        :: (Task a) (Task b)   → Task (Either a b)       | iTask a & iTask b
anyTask           :: [Task a]        → Task a              | iTask a
```

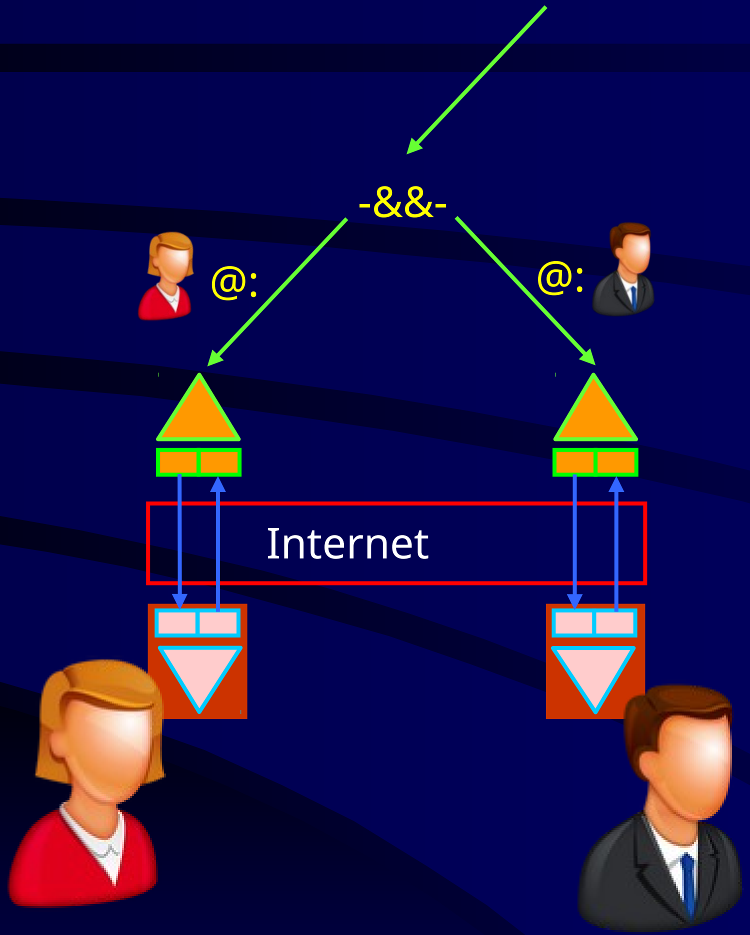
one-of: start two tasks, but we are only interested in the result of one of them, use the other to inform:

```
(-||)  infixl 3   :: (Task a) (Task b)   → Task a              | iTask a & iTask b
(||-)  infixr 3   :: (Task a) (Task b)   → Task b              | iTask a & iTask b
```

assign a task to a specific user:

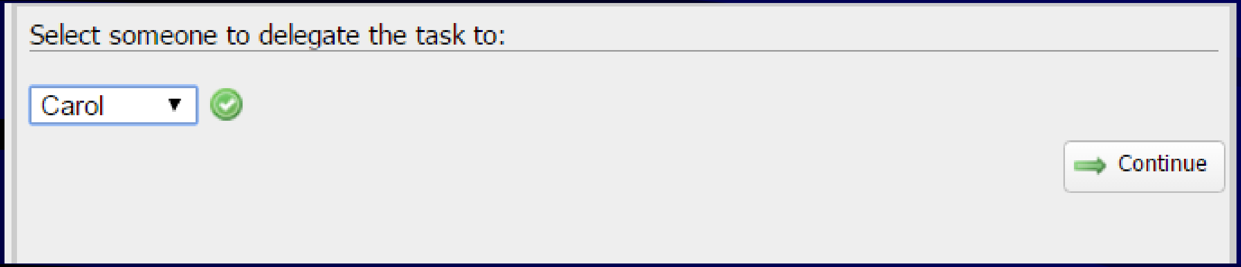
```
(@:) infix 3  :: User (Task a)   → Task a              | iTask a
```

Assigning Tasks to Users



Multi-users

```
delegate :: (Task a) → Task a  | iTask a
delegate task
        =               enterChoiceWithShared "Select someone to delegate the task to:" [] users
        >>= \user →         user @: (task >>= return)
        >>= \result → >   viewInformation "The result is:" [] result
```



Select someone to delegate the task to:

Carol ▼ ✓

Shared Data Sources

There are many different types of data storages, sources, sinks, one can use to exchange information:

- Shared Memory Files Cloud Time Sensors , ….

SDS: *one* abstraction layer for *any* type of shared data: easy to use for the progammer

```
:: RWShared  r w
```

- Reading and Writing can be of *different* type
- It includes a publish-subscribe system:
  * task *looking* at a share are automatically notified when the share has changed
- Fine-tuning :
  * which kind of change should trigger a notification ?
  * how to react on a race-condition ?
- SDS's can be composed from others using special Share Combinators

```
:: Shared a           :== RWShared a  a

:: ReadOnlyShared a   :== RWShared a Void
:: WriteOnlyShared a  :== RWShared Void a
```

Shared Data Sources

Creating an SDS:

```
withShared       :: a  ((Shared a) → Task b)      → Task b     | iTask b    // Shared memory

sharedStore      :: String  a                  → Shared a        | iTask a    // Special File
externalFile     :: FilePath                   → Shared String          // Ordinary File
sqlShare         :: SQLDatabase String  …       → ReadWriteShared r w        // SQL Database
```

Reading an SDS:

```
get :: (RWShared r w)                          → Task r       | iTask r     // read once

currentTime      :: ReadOnlyShared Time
currentDate      :: ReadOnlyShared Date
currentDateTime  :: ReadOnlyShared DateTime
currentUser      :: ReadOnlyShared User
users            :: ReadOnlyShared [User]
```

Updating an SDS:

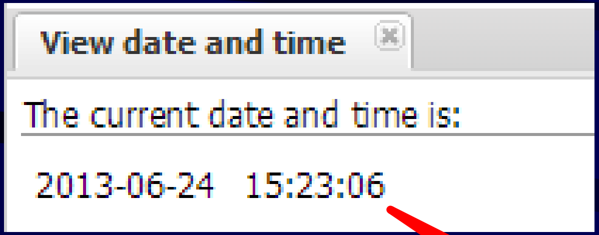
```
set              :: w (RWShared r w)             → Task w     | iTask w    // write once

update        :: (r → w) (RWShared  r w)      → Task w      | iTask r & iTask w
```

Interactive Editors on SDS’s

```
viewSharedInformation         :: p [ViewOption r] (RWShared r w) → Task r
                                    | toPrompt p & iTask r

updateSharedInformation       :: p [UpdateOption r w] (RWShared r w) → Task w
                                    | toPrompt p & iTask r & iTask w

enterSharedChoice             :: p [ChoiceOption a] (RWShared [a] w) → Task a
                                    | toPrompt p & iTask a & iTask w
updateSharedChoice            :: p [ChoiceOption a] (RWShared [a] w) a → Task a
                                    | toPrompt p & iTask a & iTask w

enterSharedMultipleChoice     :: p [MultiChoiceOption a] (RWShared [a] w) → Task [a]
                                    | toPrompt p & iTask a & iTask w
updateSharedMultipleChoice        :: p [MultiChoiceOption a] (RWShared [a] w) [a] → Task [a]
                              | toPrompt p &  iTask a & iTask w
```

Editors on SDS's

```
viewCurDateTime :: Task DateTime
viewCurDateTime
        = viewSharedInformation "The current date and time is:" [] currentDateTime
```



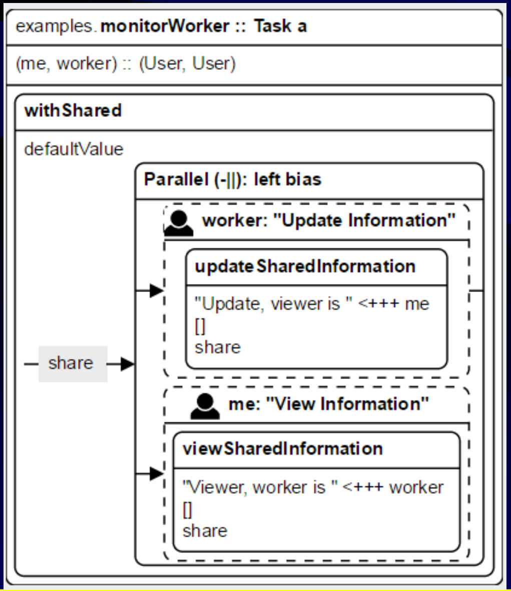
The current date and time is:

2013-06-24 15:23:06

Ticking !!

Editors on SDS's

```
monitorWorker :: ((User, String), (User, String)) → Task a  | iTask a
monitorWorker ((me,my_prompt), (you,your_prompt))
= withShared defaultValue
  (\share → (you @:  updateSharedInformation  (your_prompt, "viewer is " <+++ me) [] share)
                      -||
                      (me @: viewSharedInformation    (my_prompt, "worker is " <+++ worker)  [] share)
  )
```



examples. **monitorWorker :: Task a**

(me, worker) :: (User, User)

withShared

defaultValue

share

Parallel (-||): left bias

worker: "Update Information"

updateSharedInformation

"Update, viewer is " <+++ me
[]
share

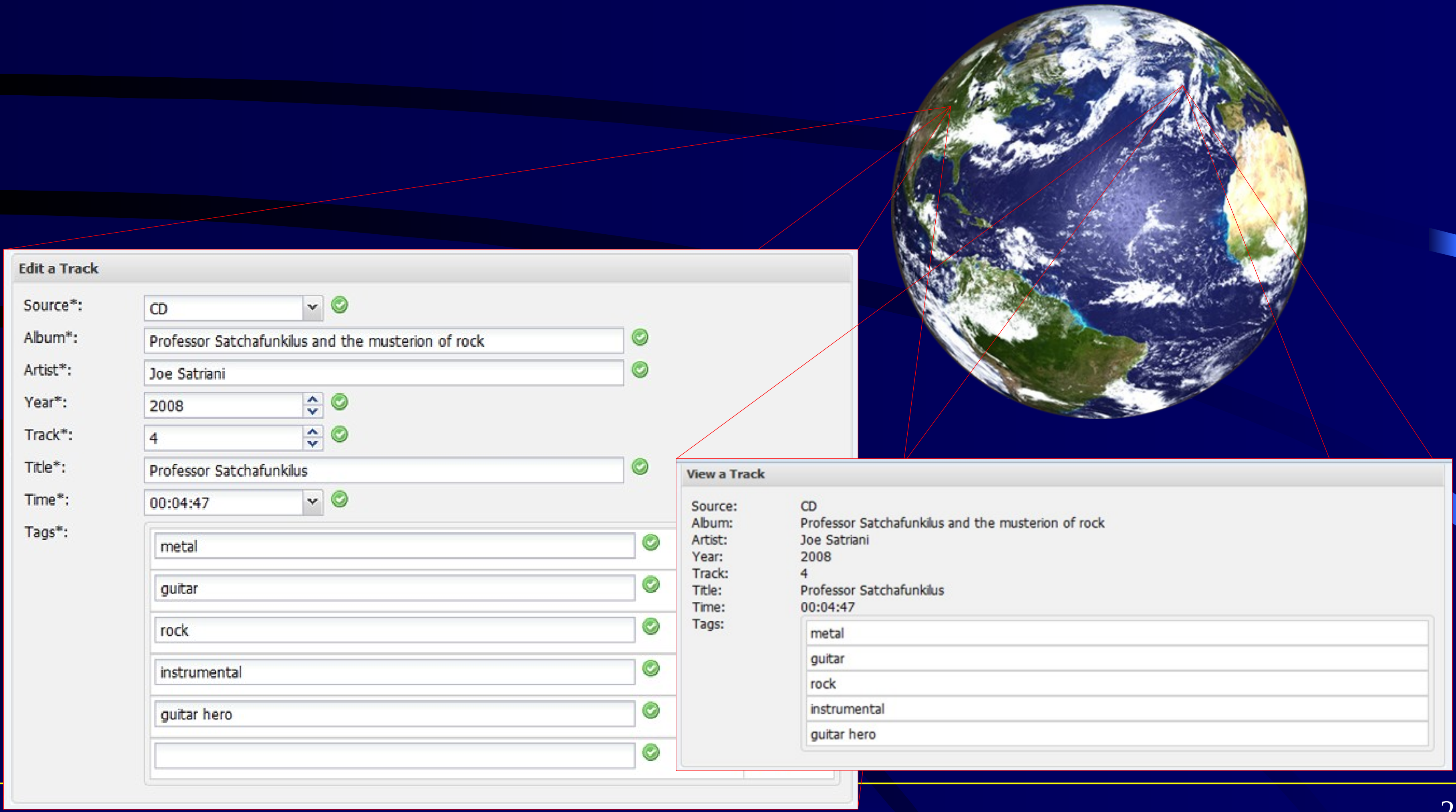
me: "View Information"

viewSharedInformation

"Viewer, worker is " <+++ worker
[]
share

Editors on SDS's

```
addTrack :: Task Track
addTrack = monitorWorker (("peter",“View a Track"),(“rinus",“Edit a Track"))
```



Edit a Track

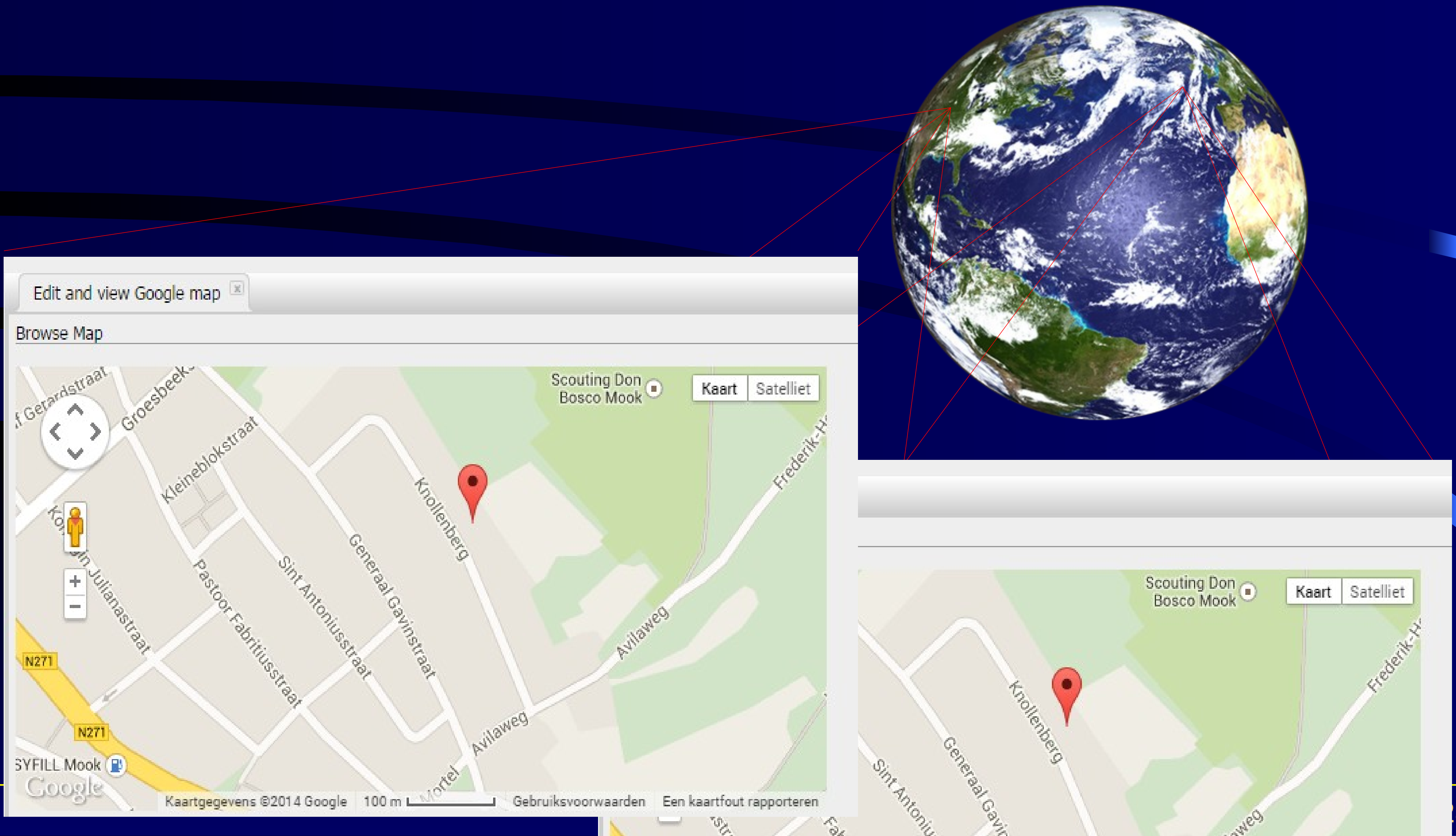
Source*:	CD ✓
Album*:	Professor Satchafunkilus and the musterion of rock ✓
Artist*:	Joe Satriani ✓
Year*:	2008 ✓
Track*:	4 ✓
Title*:	Professor Satchafunkilus ✓
Time*:	00:04:47 ✓
Tags*:	metal ✓ guitar ✓ rock ✓ instrumental ✓ guitar hero ✓ ✓

View a Track

Source:	CD
Album:	Professor Satchafunkilus and the musterion of rock
Artist:	Joe Satriani
Year:	2008
Track:	4
Title:	Professor Satchafunkilus
Time:	00:04:47
Tags:	metal guitar rock instrumental guitar hero

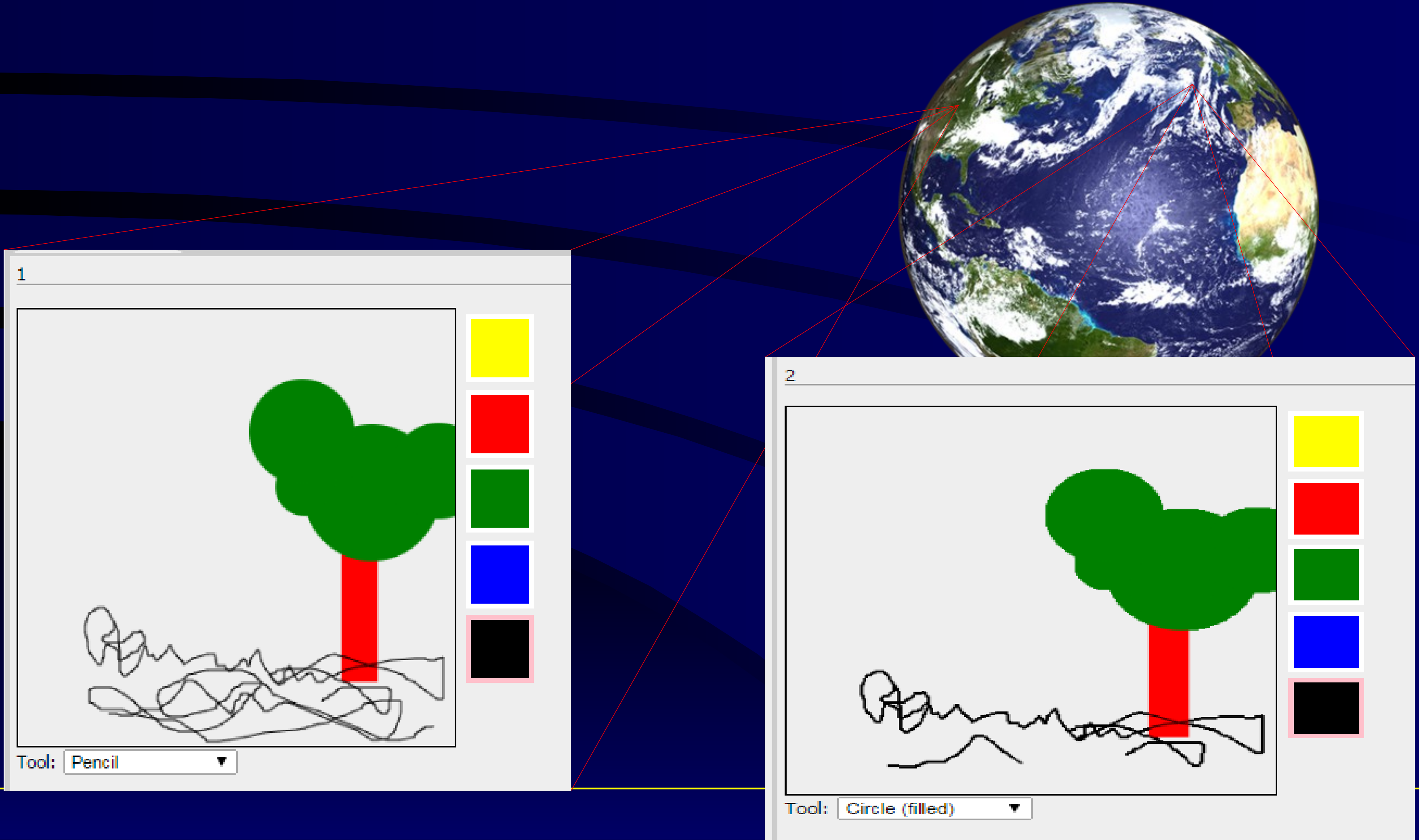
Editors on SDS's

```
changeMap :: Task GoogleMap
changeMap = monitorWorker (("peter", "View Map"),("rinus", "Browse Map"))
```



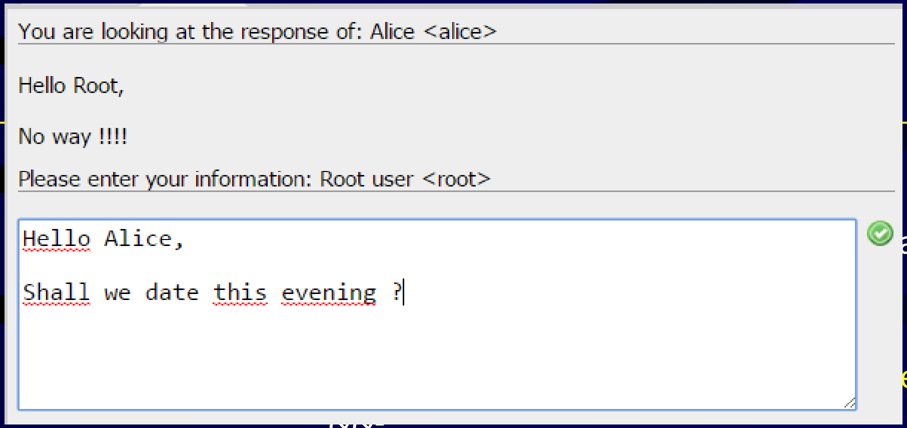
Editors on SDS's

```
changeMap :: Task Drawing
changeMap = monitorWorker ((“bert", "View Drawing"),(“ernie", “Make Drawing"))
```



Tool: Pencil

Tool: Circle (filled)



You are looking at the response of: Alice <alice>

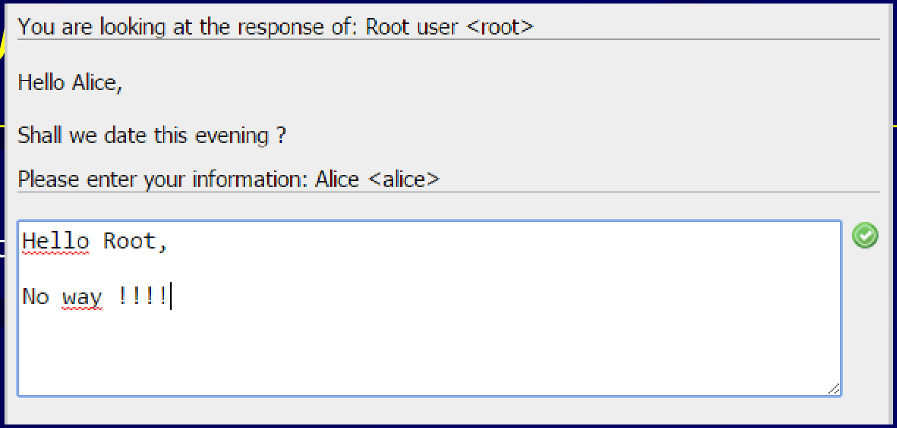
Hello Root,

No way !!!!

Please enter your information: Root user <root>

```
Hello Alice,

Shall we date this evening ?
```

U



You are looking at the response of: Root user <root>

Hello Alice,

Shall we date this evening ?

Please enter your information: Alice <alice>

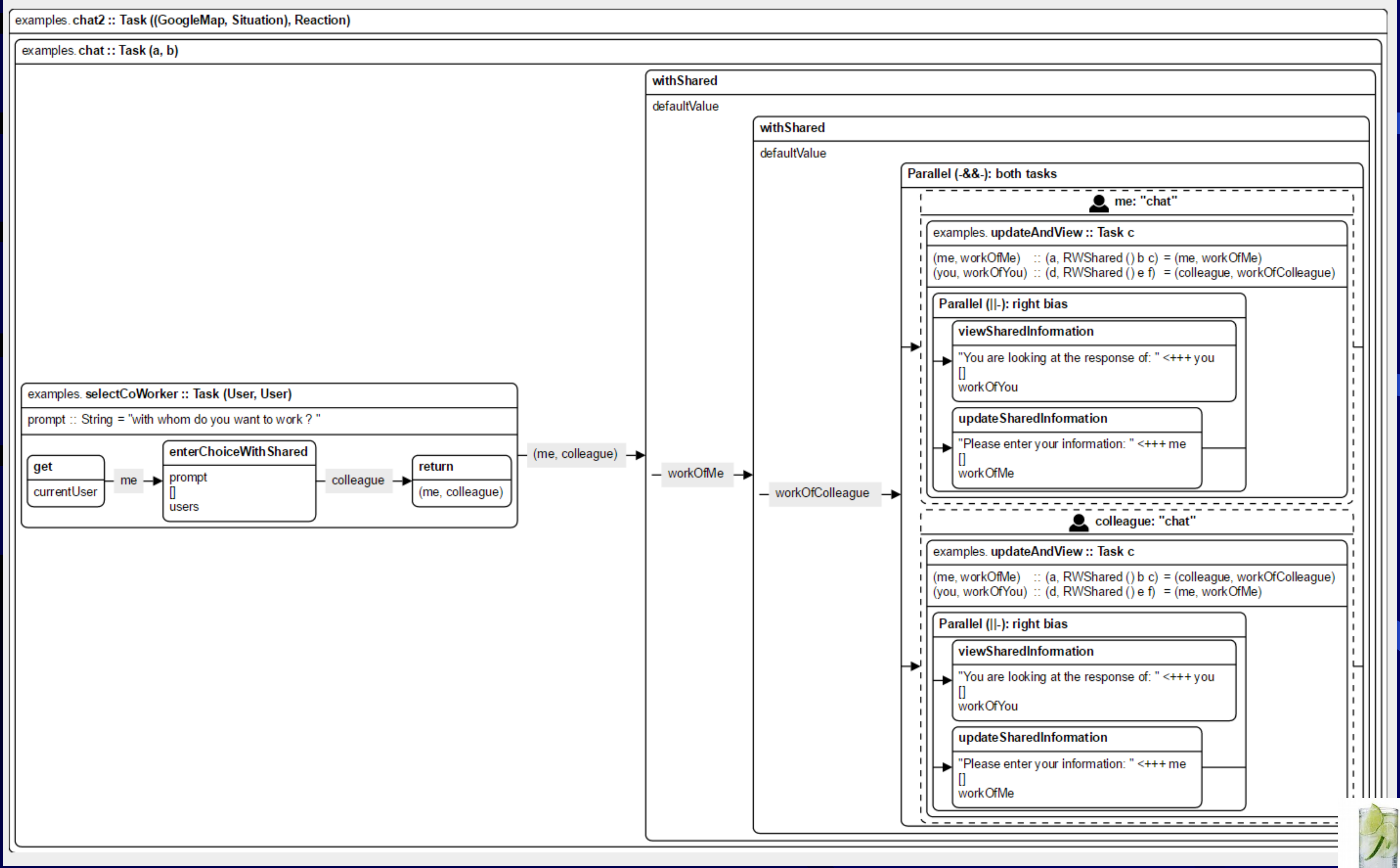
```
Hello Root,

No way !!!!
```

ant

ew

```
          &&-
          ((colleague,"chat") @: updateAndView (colleague,workOfColleague) (me,workOfMe))
          ))

selectCoWorker :: String → Task (User, User)
selectCoWorker  prompt
=                         get currentUser
>>= \me ->                enterChoiceWithShared prompt [] users
>>= \colleague ->         return (me,colleague)

updateAndView :: (User, Shared a) (User, Shared b) → Task a          | iTask a & iTask b
updateAndView (me, workOfMe) (you, workOfYou)
=       updateSharedInformation  ("Please enter your information: " <+++ me) [] workOfMe
        -||
        viewSharedInformation    ("You are looking at the response of: " <+++ you) [] workOfYou


chat1 :: Task (Note, Note)
chat1 = chat
```

Editors on SDS's



examples. **chat2 :: Task ((GoogleMap, Situation), Reaction)**

examples. **chat :: Task (a, b)**

examples. **selectCoWorker :: Task (User, User)**

prompt :: String = "with whom do you want to work ? "

get
currentUser

me

enterChoiceWithShared
prompt
[]
users

colleague

return
(me, colleague)

(me, colleague)

withShared

defaultValue

workOfMe

withShared

defaultValue

workOfColleague

Parallel (-&&-): both tasks

me: "chat"

examples. **updateAndView :: Task c**

(me, workOfMe) :: (a, RWShared () b c) = (me, workOfMe)
(you, workOfYou) :: (d, RWShared () e f) = (colleague, workOfColleague)

Parallel (||-): right bias

viewSharedInformation
"You are looking at the response of: " <+++ you
[]
workOfYou

updateSharedInformation
"Please enter your information: " <+++ me
[]
workOfMe

colleague: "chat"

examples. **updateAndView :: Task c**

(me, workOfMe) :: (a, RWShared () b c) = (colleague, workOfColleague)
(you, workOfYou) :: (d, RWShared () e f) = (me, workOfMe)

Parallel (||-): right bias

viewSharedInformation
"You are looking at the response of: " <+++ you
[]
workOfYou

updateSharedInformation
"Please enter your information: " <+++ me
[]
workOfMe

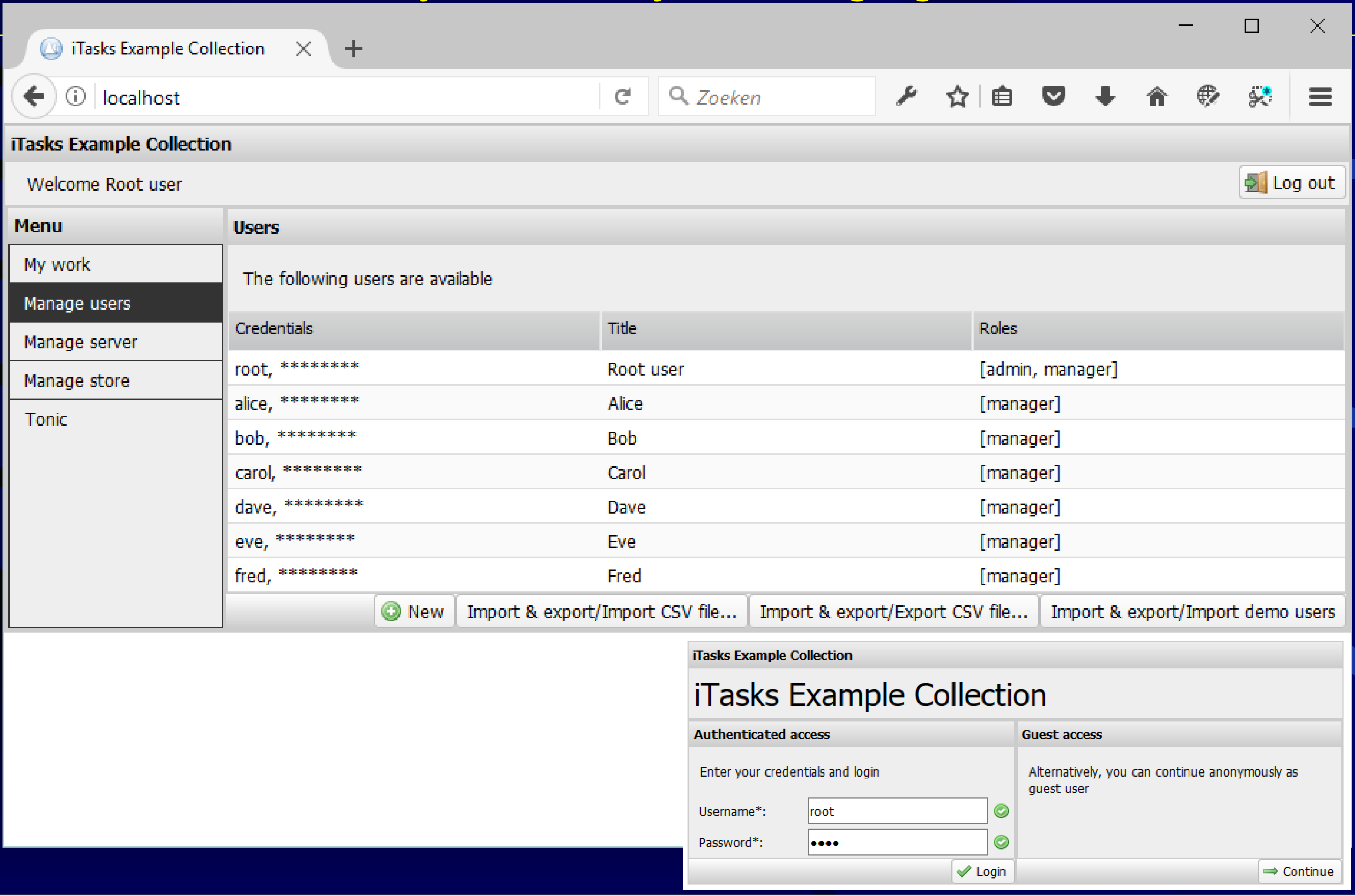
Predefined Tasks for managing tasks

```
module example

import iTasks

Start :: *World → *World
Start world = startEngine palindrome world









palindrome :: Task (Maybe String)
palindrome = …

person1by1 :: Task [Person]
person1by1 = …
```

Predefined Tasks for managing tasks

```
module examples

import iTasks

Start :: *World → *World
Start world  = doTasks myTasks world

myTasks
 =      installWorkflows myWorkFlows
 >>|  loginAndManageWork "welcome to my examples"


myWorkFlows :: [Workflow]
myWorkFlows
 =  [ workflow "palindrome"                "accepts palindrome string "       palindrome
    , workflow "create list of persons“    "one by one"                person1by1
    , ...
    , workflow "Manage users"              "Manage system users..."    manageUsers
    ]


palindrome :: Task (Maybe String)
palindrome = ...

person1by1 :: Task [Person]
person1by1 = ...
```

Predefined Tasks for managing tasks


iTasks Example Collection
localhost
Zoeken
Log out
Menu
My work
Manage users
Manage server
Manage store
Tonic
New
Import & export/Import CSV file...
Import & export/Export CSV file...
Import & export/Import demo users
Login
Continue

iTasks Example Collection

Welcome Root user

Users

The following users are available

Credentials	Title	Roles
root, ********	Root user	[admin, manager]
alice, ********	Alice	[manager]
bob, ********	Bob	[manager]
carol, ********	Carol	[manager]
dave, ********	Dave	[manager]
eve, ********	Eve	[manager]
fred, ********	Fred	[manager]

iTasks Example Collection

iTasks Example Collection

Authenticated access

Enter your credentials and login

Username*: root

Password*: ••••

Guest access

Alternatively, you can continue anonymously as guest user