TOP

Rinus Plasmeijer – Bas Lijnse - Peter Achten
Pieter Koopman - Steffen Michels - Jurrien Slutterheim (TNO-RU)
Jan Martin Jansen (NLDA) - Laszlo Domoszlai (ELTE)

Radboud University Nijmegen

Task Oriented Programming

Rinus Plasmeijer – Bas Lijnse - Peter Achten
Pieter Koopman - Steffen Michels - Jurrien Slutterheim (TNO-RU)
Jan Martin Jansen (NLDA) - Laszlo Domoszlai (ELTE)

Radboud University Nijmegen

From Workflow Specifications in FP to TOP

How can we define nicely

- workflows management systems in a pure FP ?
- multi-user web based GUI applications ?
- merge sever-side and client-side evaluation ?
- the management of tasks, as a task ?
- soft real-time complex collaborations, e.g. support for crisis management ?

Why Task Oriented Programming ?

- *Tasks are a common notion in daily life / in any organization*
  - People increasingly work together distributively on the internet
  - Focus on complex collaborations, arbitrary ways of working
    - (sub) tasks and their interdependencies are dynamically determined
  - Any kind of task (involving computers) should be expressible
  - Huge Application Area
    - CC2, Crisis management, (e) Health Care, Insurance Market, Systems for Economical Market, (e) Government, Legal Systems, ERP, Social Media
- *Tasks are useful building blocks when developing software*
  - function call, procedure call, method invocation, calling a web-service, a query
  - web form handling, email handling
  - process, thread, "app"
- *Tasks are suited to communicate ideas between Domain Expert – TOP Programmer*

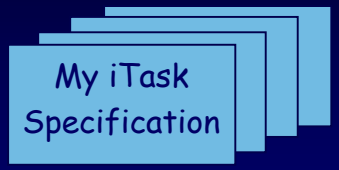
Task Oriented Programming

- New style of (functional) programming
- Tasks as basic building blocks
- Reactive system
- Declarative
  - High level of abstraction
  - No worry about technical realization !
- Yields application coordinating the work of collaborating people & systems
- But, it can also be used
  - for Rapid Prototyping
  - to formalize how work should be organized
  - to investigate different ways of doing work using simulation with agents
  - for training: mix of real people and agents
  - to check properties by testing, analysis or by formal proof (semantics formally defined)
  - to communicate desired ways of working between domain experts and programmers

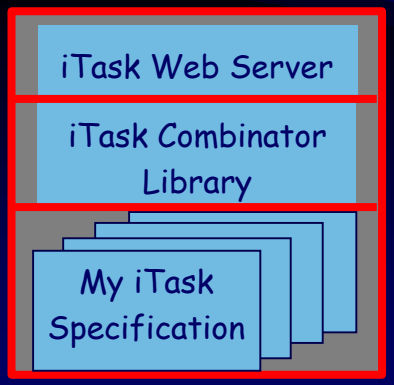
- implementation of TOP

- Domain Specific Programming Language, embedded in Clean
- "just" another Combinator Library
- Abstracts as much technical stuff as possible (thanks to generic functions):
  - graphical user interfaces & handling of user-interaction
  - persistent storage of information
  - (client-server) communication
  - evaluation on client
  - informing tasks about the progress in tasks others work on
  - informing tasks when shared information is changed
- Yields Web-Service coordinating the tasks to be done…
  - Tasks can run on server, on client , or on both
  - Clean is standard compiled twice:
    1. to native code (Windows, Mac-OS, Linux)
    2. to SAPL, and just-in-time on demand by client to javascript

iTasks Architecture



iTasks Architecture



iTasks Architecture

Computer

iTask Client

Laptop

iTask Client

Phone

iTask Client

Tablet

iTask Client

Sensor

iTask Client

iTask User | iTask User | iTask User | iTask User | iTask User

WEB

Web Service | Web Service | Web Service | RPC | Cloud / Database

iTask Web Server

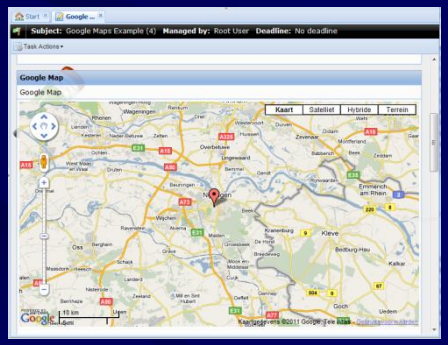
iTask Combinator Library

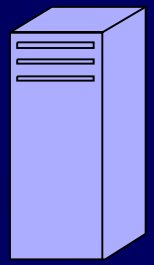
My iTask Specification

iTask Web Server

iTask Combinator Library

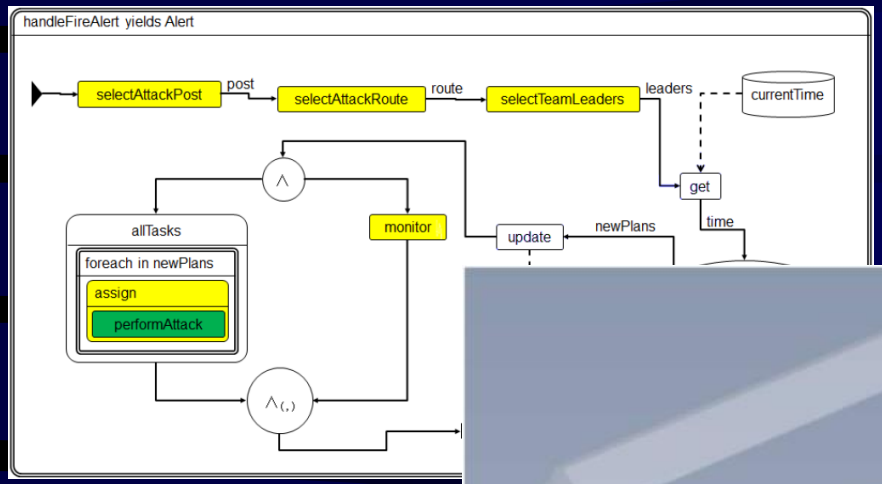
iTask Specification

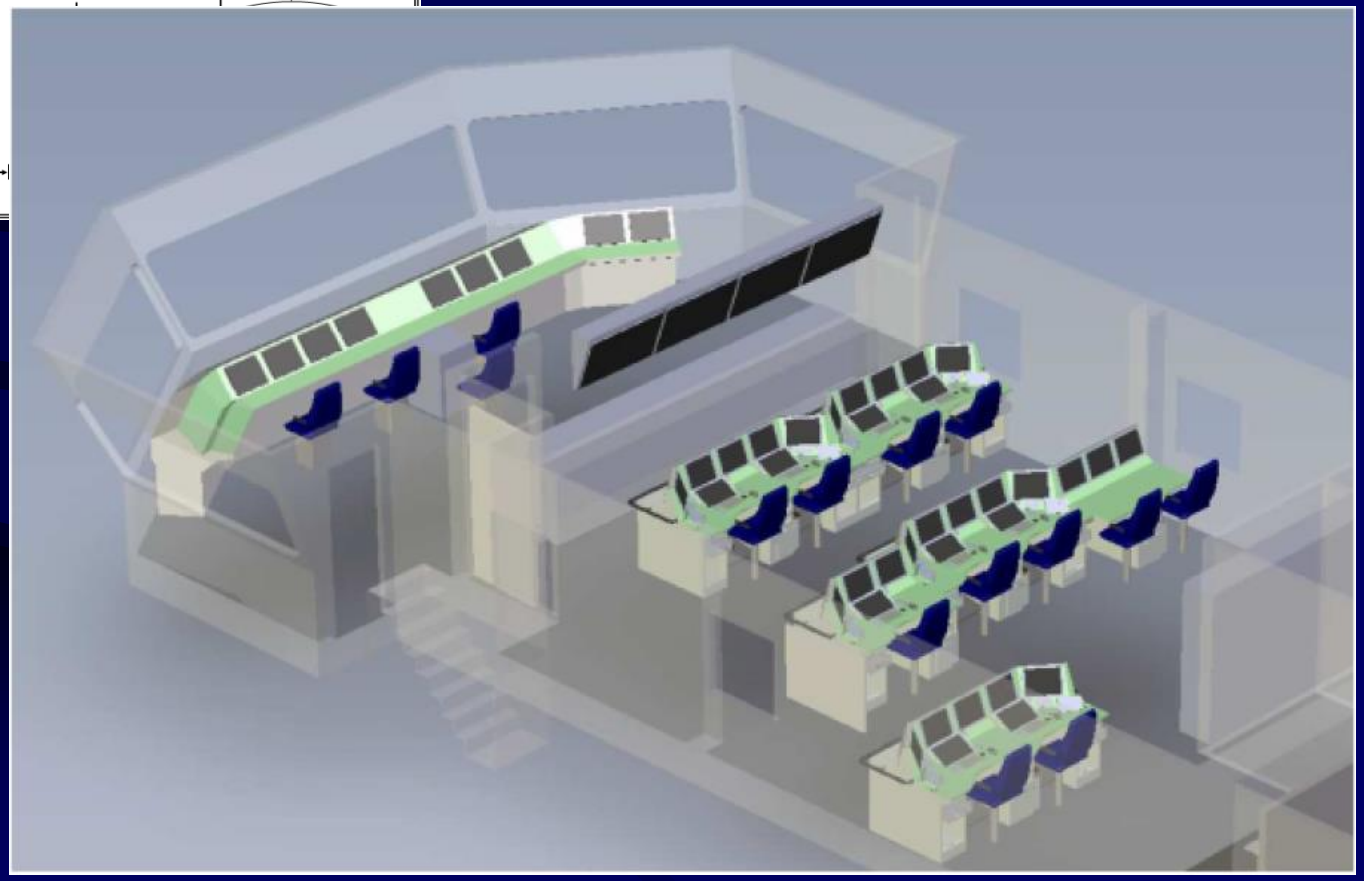




Case study -> Prototype: Coast Guard Search And Rescue

Prototype : Vessel Crew Optimization – TNO





iTask Core

- Tasks: typed, a task value may change over time
- Basic tasks:
  Interactive Tasks : editors
  Simple Tasks : return, …
  Foreign stuff : web-service, OS-call, sensors, …
- Sequential and Parallel Combinators for combining tasks
  Defines control flow and data flow between tasks
- Shared Information: *one* concept for sharing any kind of information

+ growing iTask Library to support frequently occurring work patterns

+ Clean
  pure, higher order, polymorphic, overloaded, generic functions
  hybrid typing: strongly statically typed + dynamic typing

Task Values

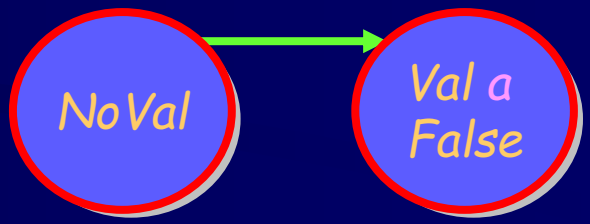
- `:: Task a` *typed* unit of work which should deliver a task result of type a
- While the task is going on, its value may change over time

Task Values

- `:: Task a` *typed* unit of work which should deliver a task result of type a
- While the task is going on, its value may change over time

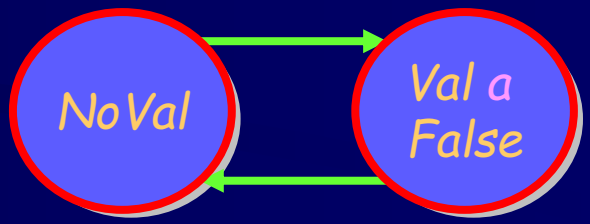
Task Values

- `:: Task a` *typed* unit of work which should deliver a task result of type a
- While the task is going on, its value may change over time



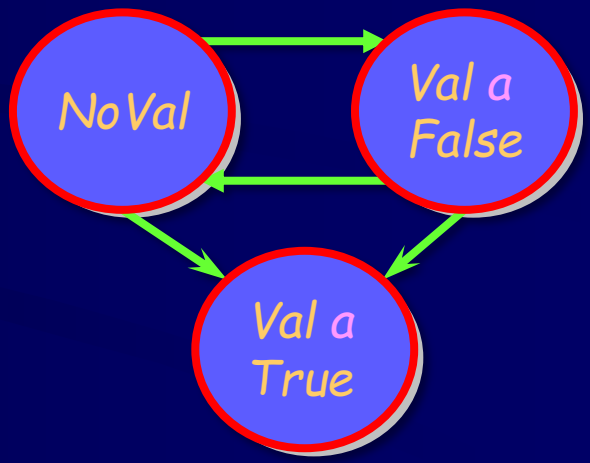
Task Values

- `:: Task a` *typed* unit of work which should deliver a task result of type a
- While the task is going on, its value may change over time



Task Values

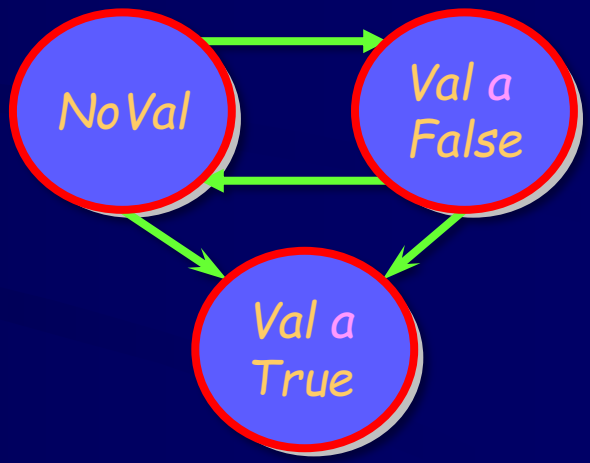
- `:: Task a` *typed* unit of work which should deliver a task result of type a
- While the task is going on, its value may change over time



Task Values

- :: Task a *typed* unit of work which should deliver a task result of type a
- While the task is going on, its value may change over time

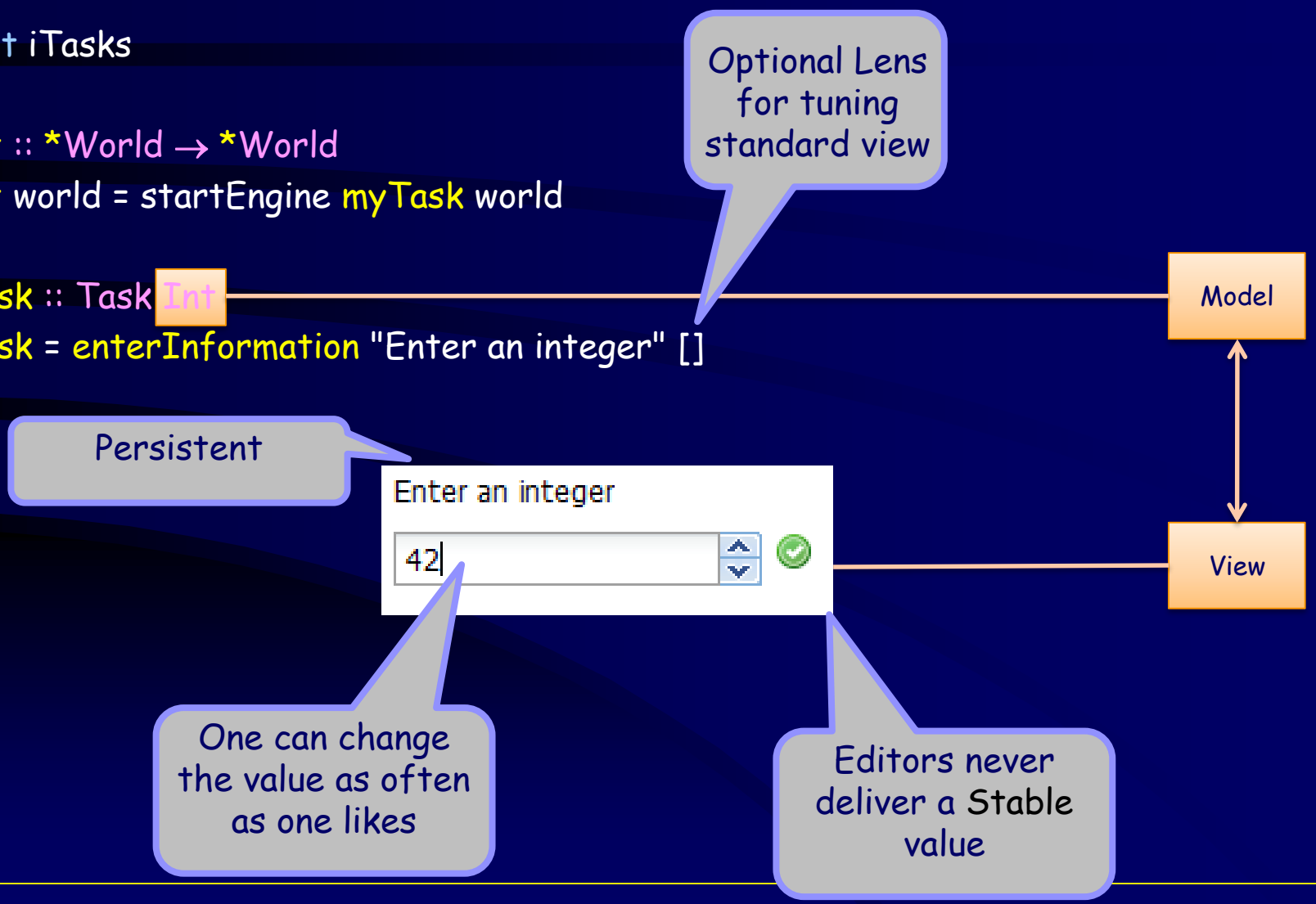
```
:: TaskResult a     =        ValRes TimeStamp (Value a)
                    | ∃ e:   ExcRes e              & iTask e
:: Value a          =         NoVal
                    |         Val a Stability
:: Stability        :==       Bool
```



- Task values
  - can be observed by other tasks
  - may influence the work of others
  - can be of *any* type: user defined, higher order (e.g a task or a function)
  - must satisfy the iTask context restriction
- A Task may raise an exception
- A Task *never* finishes (although the work may be done)
  - but its value may not be needed anymore by the environment…

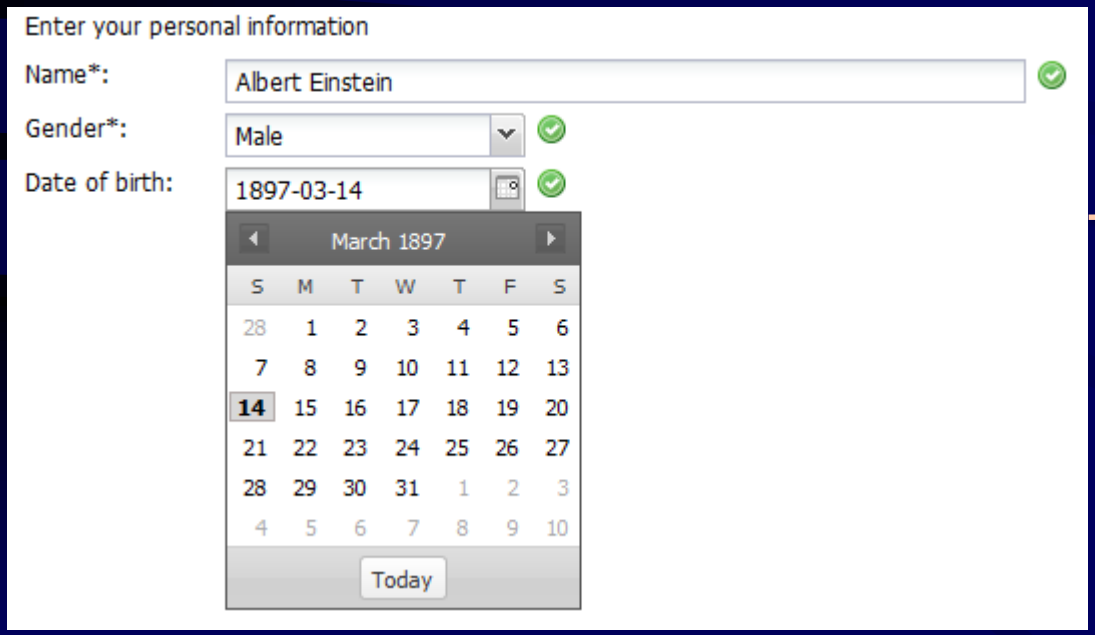
Editors

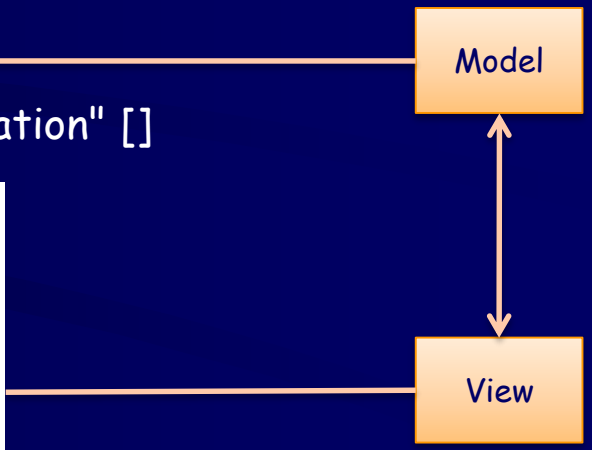
```
module example

import iTasks

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

myTask :: Task Int
myTask = enterInformation "Enter an integer" []
```



Editors - 2

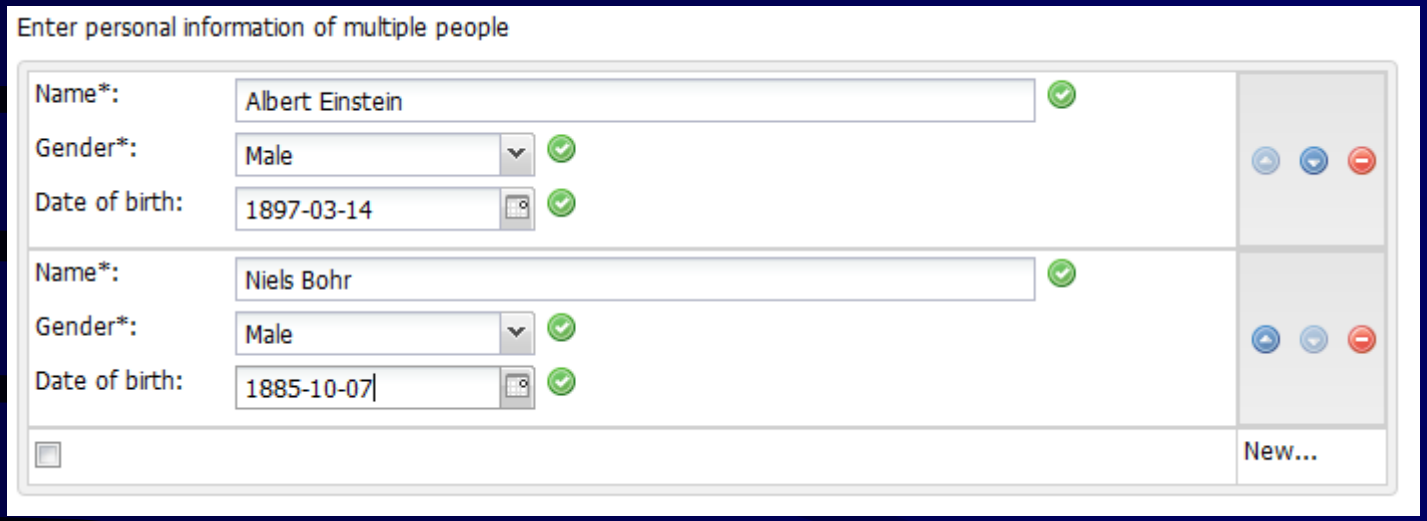
```
:: Person  =        { name :: String, gender :: Gender, dateOfBirth  :: Maybe Date}
:: Gender =         Male | Female

derive class iTask Person, Gender

myTask :: Task Person
myTask = enterInformation "Enter your personal information" []
```





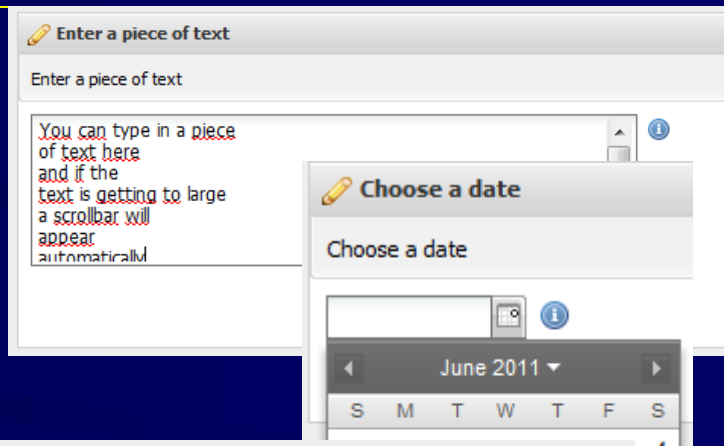
Editors - 3

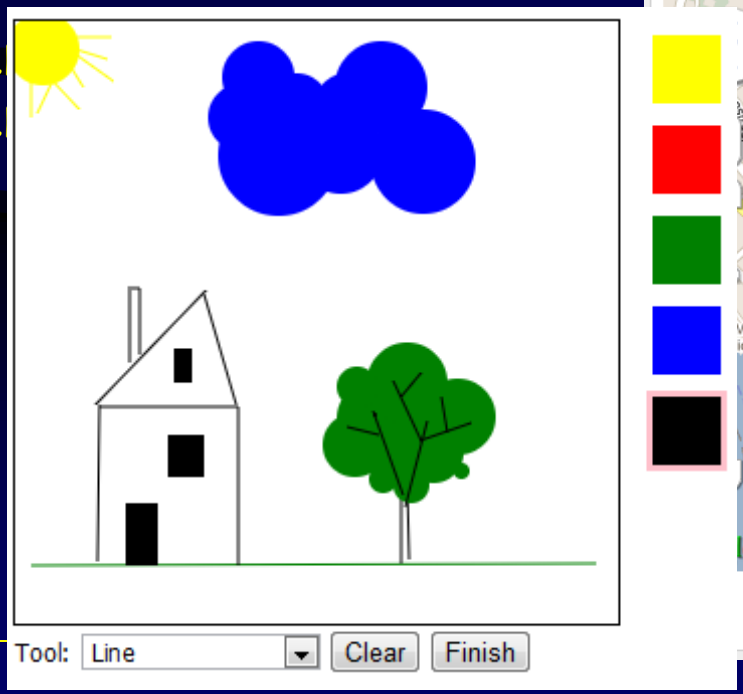
```
myTask :: Task [Person]
myTask = enterInformation "Please personal information of multiple people" []
```

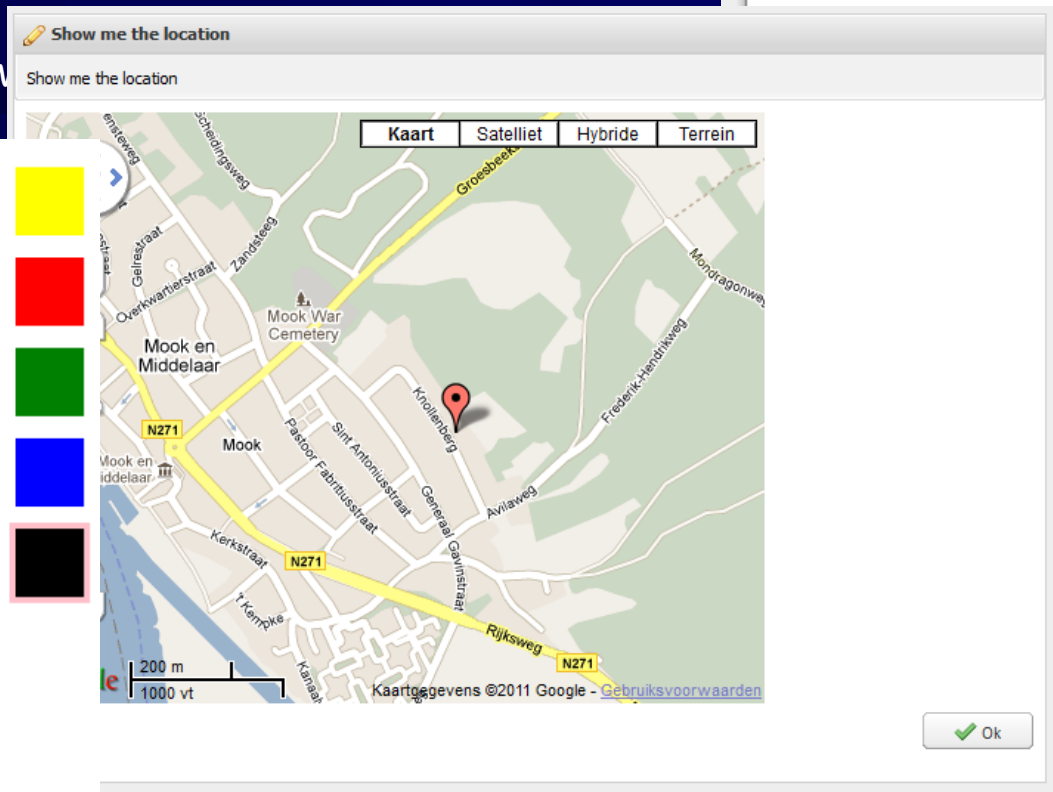


Editors - 4

```
simpleEditor :: Task Note
simpleEditor = enterInformation "Enter a piece of text" []

chooseDate :: Task Date
chooseDate = enterInformation "Choose a date" []

pointOnMap :: Task GoogleMap
pointOnMap = enterInformation "Show

simple
simple
```







Editors - 5

```
pizzaWith :: Task [String]
pizzaWith = enterMultipleChoice "What do you like on your pizza ?" []
                                ["Cheese","Tomato","Ansjofish","Salami"]
```

Variant of Interactive Editors

Basic tasks: Interactive editor for filling in forms of a certain type:

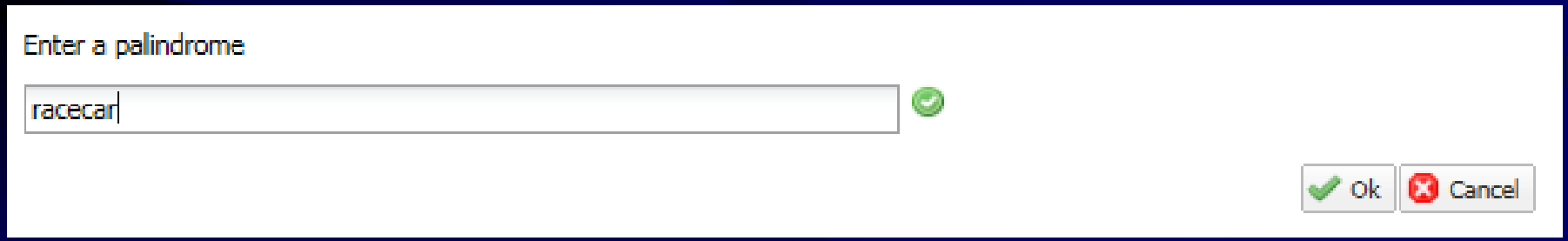
```
viewInformation     :: d [ViewOption a] a          → Task a          | descr d & iTask a
enterInformation    :: d [En
updateInformation   :: d [Up
enterChoice         :: d [Ch
updateChoice        :: d [Ch
enterMultipleChoice :: d [M
updateMultipleChoice :: d [
```

```
generic gVisualizeEditor a   | gVisualizeText a, gHeaders a, gGridRows a
                             :: (Maybe a) *VSt                → (VisualizationResult *VSt)
generic gVisualizeText a     :: StaticVisualizationMode a     → [String]
generic gUpdate a            :: (UpdateMode a) *USt           → (a, *USt)
generic gHeaders a           :: a → [String]
generic gGridRows a          | gVisualizeText a
                             :: a [String]                    → Maybe [String]
generic gVerify a            :: (Maybe a) *VerSt              → *VerSt
generic JSONEncode t         :: t                             → [JSONNode]
generic JSONDecode t         :: [JSONNode]                    → (Maybe t, [JSONNode])
generic gEq a                :: a a                           → Bool
```

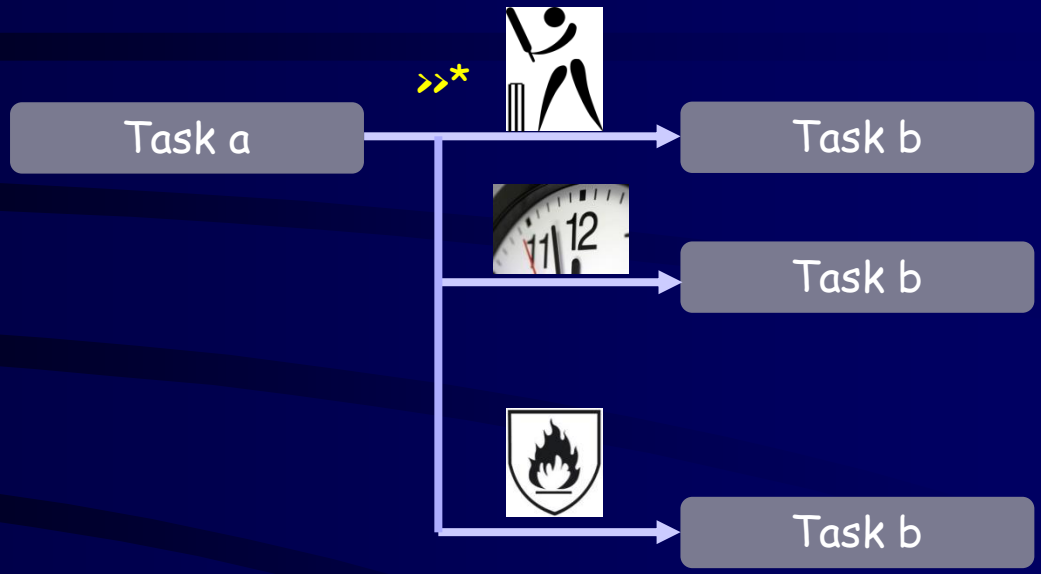
- All instances of one Core editor
- Options: definable view: between task value type a and arbitrary view type v
- descr d: can vary from a simple string to html code
- iTask a : bunch of type driven generic functions for doing the real work

Sequential Combinator: >>*

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



Sequential Combinator: >>*



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

- if a certain action is performed by the end-user

- if the value of the observed task is satisfying a certain predicate

- or the observed task has raised an exception to be handled here

Core - Sequential Combinator

- Combinator for *Sequential* Composition

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

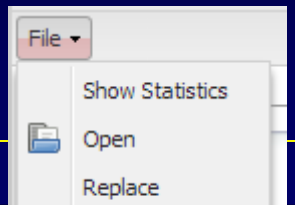
```
:: TaskStep a b
   =       OnAction Action     ((Value a) → Maybe (Task b))
   |       OnValue             ((Value a) → Maybe (Task b))
   | E.e:  OnException         (e → Task b)                          & iTask e

:: Action                  = Action String [ActionOption]

:: ActionOption            = ActionKey     Hotkey
                           | ActionWeight Int
                           | ActionIcon    String
                           | ActionTrigger DoubleClick

:: Hotkey                  = { key :: Key, ctrl :: Bool, alt :: Bool, shift :: Bool }
```

```
ActionOk      :== Action "Ok"   [ActionIcon "ok", ActionKey (unmodified KEY_ENTER)]
```

Core – Sequential Combinator

- Combinator for *Sequential* Composition

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

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

:: Action                = Action String [ActionOption]

:: ActionOption          = ActionKey     Hotkey
                         | ActionWeight Int
                         | ActionIcon    String
                         | ActionTrigger DoubleClick

:: Hotkey                = { key :: Key, ctrl :: Bool, alt :: Bool, shift :: Bool }

ActionOpen    :== Action "/File/Open" [ActionIcon "open", ActionKey (ctrl KEY_O)]
```



Core - Shared Data Sources

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

- Shared Memory  , Files  , Database  , Time  , Sensors , ...

- Reading and Writing can be of different type

- SDS's can be composed from others

- Tasks depending on an SDS are automatically informed when it is being changed

```
:: RWShared  r w
:: Shared a      :== RWShared a  a
```

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

Variants of Interactive Editors

viewInformation

enterInformation
updateInformation

enterChoice
updateChoice

enterMultipleChoice
updateMultipleChoice

Variants of Interactive Editors

viewInformation

enterInformation
updateInformation

enterChoice
updateChoice

enterMultipleChoice
updateMultipleChoice

viewSharedInformation

updateSharedInformation

enterSharedChoice
updateSharedChoice

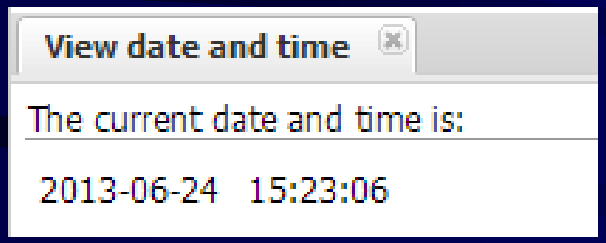
enterSharedMultipleChoice
updateSharedMultipleChoice

- All instances of one Core editor:

```
interact :: d (ReadOnlyShared r) (r → (l,v)) (l → r → v → (l,v)) → Task l
                                                  | descr d & iTask l & iTask r & iTask v
```

Editors on SDS's

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

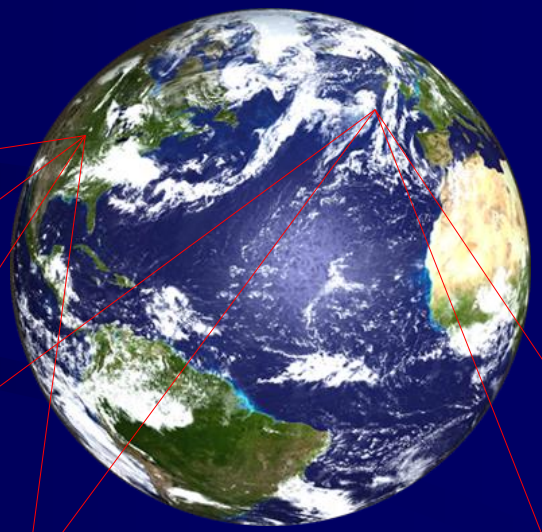


Editors on SDS's

```
twoTasks :: a → Task a      | iTask a
twoTasks v  = withShared v  doTasks

doTasks :: (Shared a) → Task a      | iTask a
doTasks sv = user1  @: updateSharedInformation  sv
           -||
           user2 @: viewSharedInformation sv
```

Assign task to someone

do both tasks in parallel, return value first



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

Handy predefined functions based on parallel

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

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

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

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

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

assign a task to a specific user:

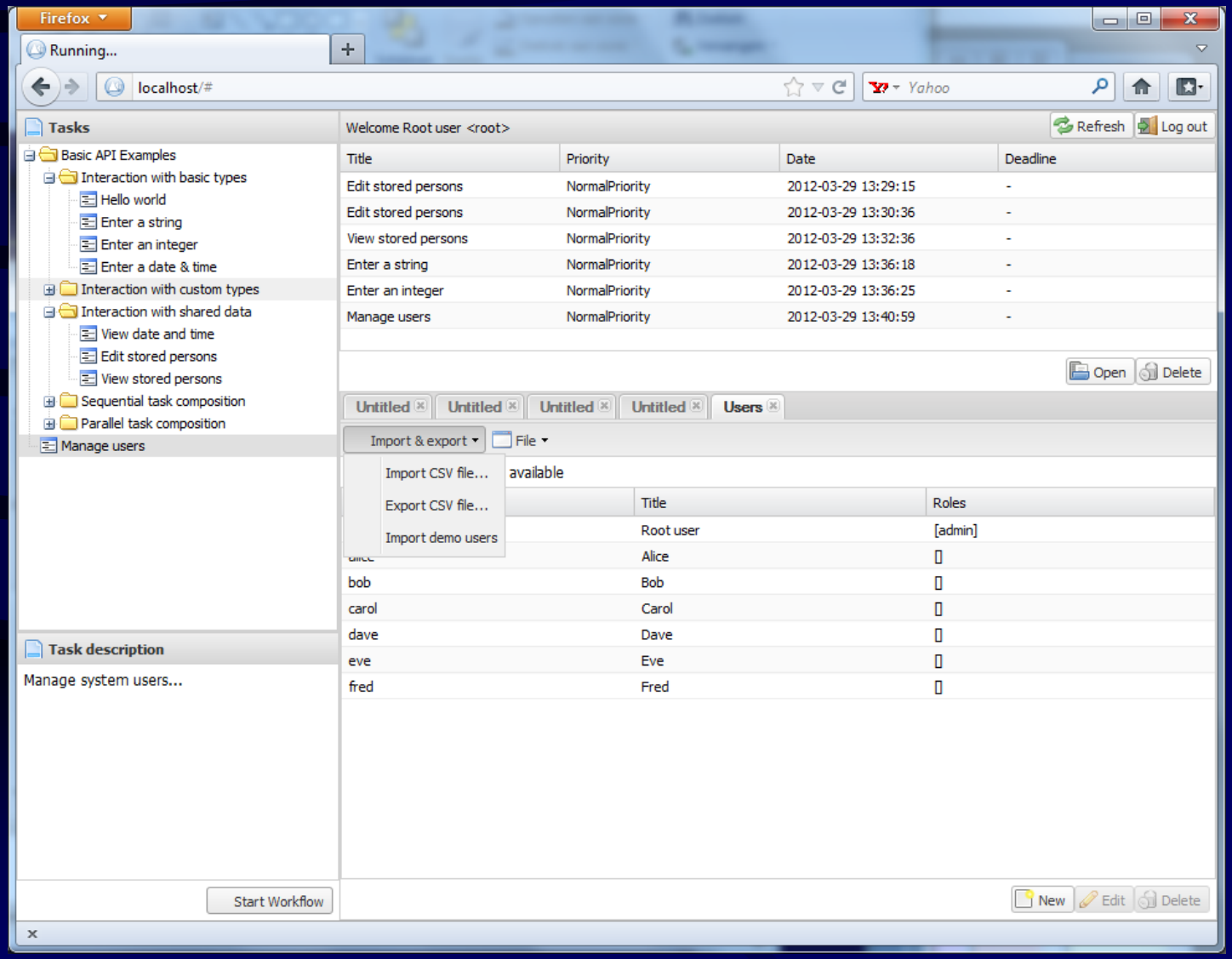
```
(@:) infix 3      :: User (Task a)     → Task a            | iTask a
```

- All instances of one Core parallel task combinator:

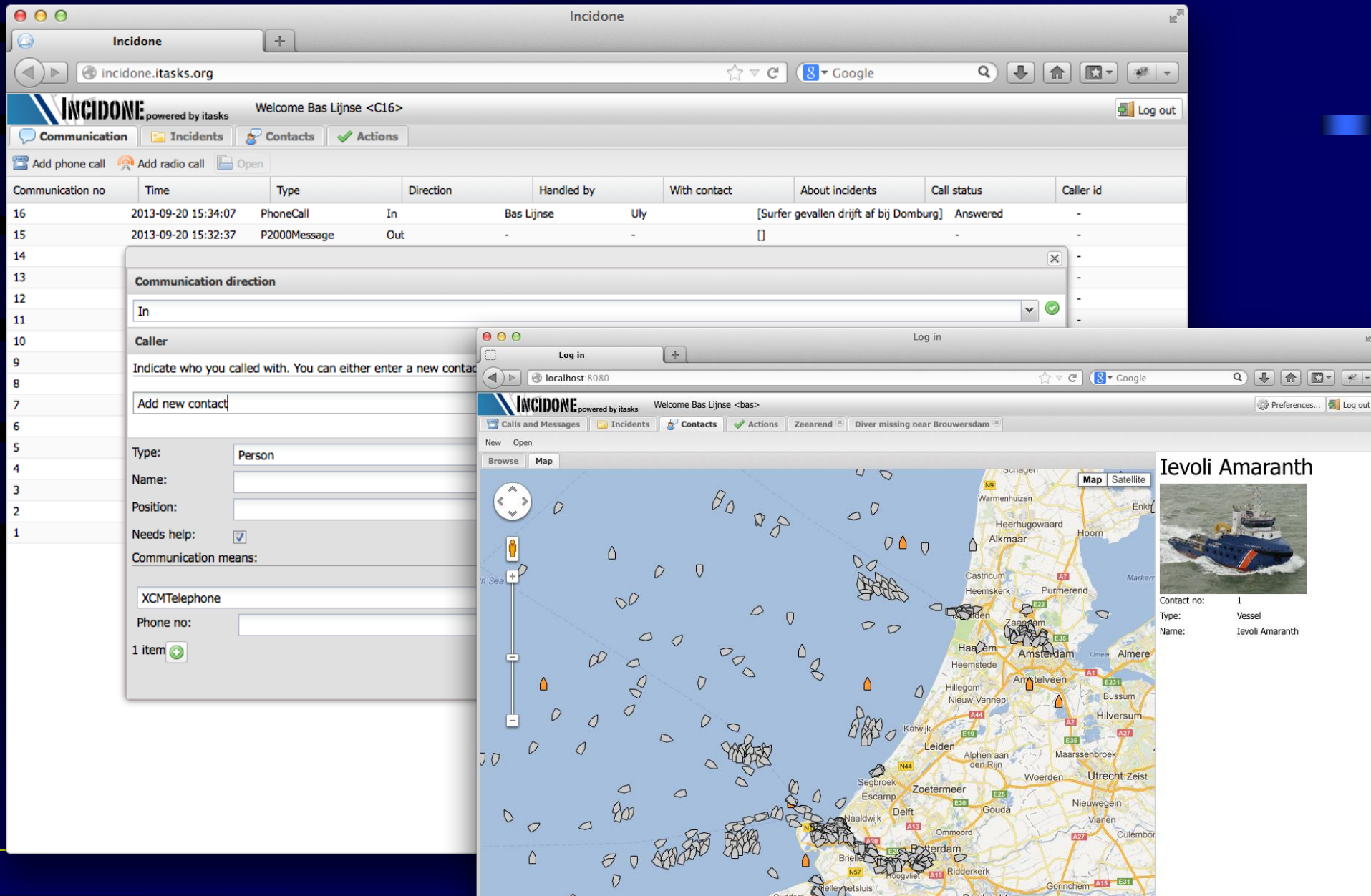
```
parallel :: d [(ParallelTaskType, (ReadOnlyShared (TaskList a)) → Task a)]
                                       → Task [(TaskTime, TaskValue a)]  | descr d & iTask a
```

itasks
dynamic workflow system

Standard iTask Client



Incidone - Coast Guard Search and Rescue Support



Semantics - What is a Task ?

```
:: Task a      :==     Event → *State → *((Reduct a, [(TaskNo, Response)]), *State)

:: Reduct a    =       Reduct (TaskResult a) (Task a)
```

Current Value

Remaining Task To do

```
rewrite :: (Task a) → *State → *(Maybe a, *State)                    | iTask a
rewrite task st
# (ev, world)                                  = getNextEvent     st.world
# (t, world)                                   = getCurrentTime   world
# ((Reduct result ntask, responses), st)       = task ev {st & timeStamp = t, world = world}
= case result of
    ValRes _ (Val a Stable) → (Just a, st)
    ExcRes _                → (Nothing, st)
    _                       → rewrite ntask  {st & world = informClients responses st.world}
```

Conclusions

- *Task Oriented Programming*
  - New style of programming for implementing multi-user web applications
  - Focusing on tasks, not on the underlying technology
  - All source code in one language

- *Core*
  - reactive tasks working on local and shared data
  - shared data sources abstracting from any type of shared data
  - editor: can handle all interactions
  - sequential and parallel combinators

- *Operational Semantics*
  - defined in Clean
  - readable, concise, type-checked, executable
  - blueprint for implementations

Future Work

- *Real* real-world applications
  - Coast Guard
  - TNO Vessel Crew

- Applicability
  - efficiency, scalability, security, version management, collaboration existing systems..

  - Parallel & distributed servers

  - *Simulation*
    - *What is the best way to do the work ?*
    - *Can we do the work with less resources ?*

  - *How to communicate task specifications with Domain Experts, End-Users ?*
    - Graphical Representations of iTasks, …

- Semantics
  - Reasoning ? Proving ? Testing ?

Questions ?

Papers on iTasks

First paper on iTasks:

- iTasks: Executable Specifications of Interactive Work Flow Systems for the Web (ICFP 2007)

Extensions:

- iTasks for a change - Type-safe run-time change in dynamically evolving workflows (PEPM 2011)
- GiN: a graphical language and tool for defining iTask workflows (TFP 2011)
- iTask as a new paradigm for building GUI applications (IFL 2010)
- Getting a grip on tasks that coordinate tasks (LDTA 2011)

Semantics:

- An Executable and Testable Semantics for iTasks (IFL 2008)

⇒ Task Oriented Programming in a Pure Functional Language (PPDP 2012)

Client site evaluation of tasks:

- Transparant Ajax and Client-Site Evaluation of iTasks (IFL 2007)
- iEditors: Extending iTask with Interactive Plug-ins (IFL 2008)

Applicability:

- A Conference Management System based on the iData Toolkit (IFL 2007)
- Web Based Dynamic Workflow Systems for C2 of Military Operations (ICCRTS 2010)
- Managing COPD exacerberations with telemedicine (AIME 2010)
- Towards Dynamic Workflows for Crisis Management (ISCRAM 2010)
- Capturing the Netherlands Coast Guard's SAR Workflow with iTasks (ISCRAM 2011)
- A Task-Oriented Incident Coordination Tool (ISCRAM 2012)

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
```