Ajax and Client-Side Evaluation of i-Tasks Workflow Specifications

Rinus Plasmeijer – Jan Martin Jansen - Peter Achten – Pieter Koopman

University of Nijmegen - Dutch Defense Academy

clean.cs.ru.nl http://www.cs.ru.nl/~rinus/iTaskIntro.html

- Recap on Workflow Systems & iTasks (ICFP 2007)
- Implementation of i-Tasks
  - Basic implementation: Task Tree Reconstruction
  - Optimized: Task Tree Rewriting
  - Local Task Rewriting using "Ajax" technology
  - Client Side Local Task Rewriting using the SAPL interpreter
- Conclusion & Future Research

1. What is a Workflow System?

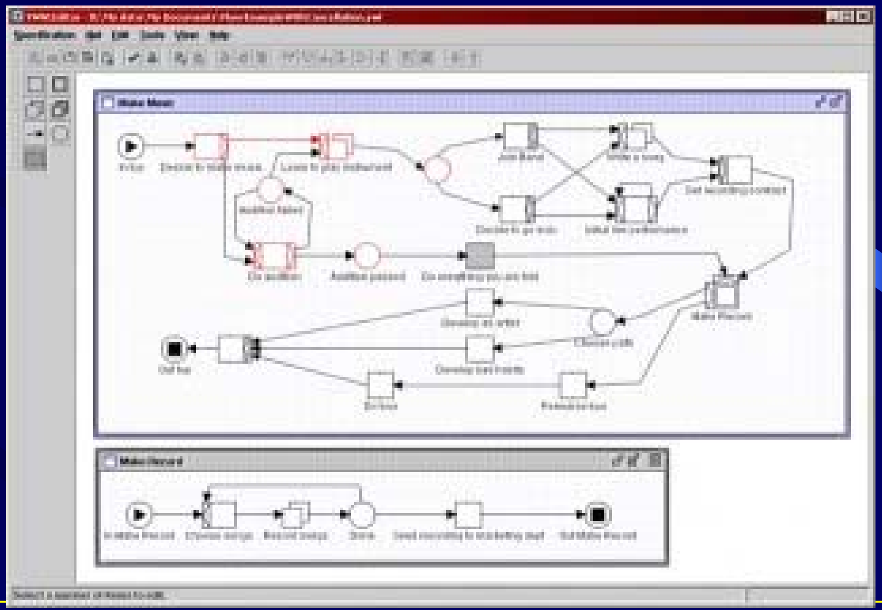
- A Workflow describes the operational aspects of work to be done
  - What are the tasks which have to be performed to achieve a certain goal ?
  - How do these tasks depend on each other?
    - In which order should the work be done ?
  - Who should perform these tasks ?

- A Workflow System is a computer application which coordinates the work, given
  - the workflow description
  - the actual work to be done
  - the actual resources available

2. How do existing Work Flow Systems look like?

- Common characteristics of Commercial Workflow Systems
  - Semantics based on (simple) Petri Nets
  - Workflows are commonly graphically defined: flow graphs
    - Workflow specification abstracts from concrete work and resources
    - Databases are used to store the actual work and progress made
  - > 25 "Workflow Patterns" identified (Van der Aalst et al.)

    sequencing, repetition, exclusive choice,
    multiple choice,
    parallel or, parallel or, ...



  - Descriptions are un-typed
  - Descriptions are static

3. *i -Tasks* Approach

Initiative from industry: *why not apply techniques known from Functional Languages?*

- Dutch Applied Science (STW) project: “Demand Driven Workflows”
- *i*-Tasks is our first "simple" try out

3. *i -Tasks* Approach

Initiative from industry: *why not apply techniques known from Functional Languages?*

- Dutch Applied Science (STW) project: “Demand Driven Workflows”
- *i*-Tasks is our first "simple" try out

We offer *all* "standard" Workflow Patterns as combinator functions

- Sequencing of tasks, repetition, exclusive choice, multiple choice, …

3. *i -Tasks Approach*

Initiative from industry: *why not apply techniques known from Functional Languages?*

- Dutch Applied Science (STW) project: “Demand Driven Workflows”
- *i*-Tasks is our first "simple" try out

We offer *all* "standard" Workflow Patterns as combinator functions

- Sequencing of tasks, repetition, exclusive choice, multiple choice, …

Typical features known from functional languages like Haskell and Clean

- Strongly typed, dynamically constructed, compositional, re-usable

3. *i -Tasks* Approach

Initiative from industry: *why not apply techniques known from Functional Languages?*

- Dutch Applied Science (STW) project: "Demand Driven Workflows"
- *i*-Tasks is our first "simple" try out

We offer *all* "standard" Workflow Patterns as combinator functions

- Sequencing of tasks, repetition, exclusive choice, multiple choice, …

Typical features known from functional languages like Haskell and Clean

- Strongly typed, dynamically constructed, compositional, re-usable

New useful workflow patterns

- Higher order tasks, Processes, Exception Handling, …

3. *i -Tasks Approach*

Initiative from industry: *why not apply techniques known from Functional Languages?*

- Dutch Applied Science (STW) project: "Demand Driven Workflows"
- *i*-Tasks is our first "simple" try out

We offer *all* "standard" Workflow Patterns as combinator functions

- Sequencing of tasks, repetition, exclusive choice, multiple choice, …

Typical features known from functional languages like Haskell and Clean

- Strongly typed, dynamically constructed, compositional, re-usable

New useful workflow patterns

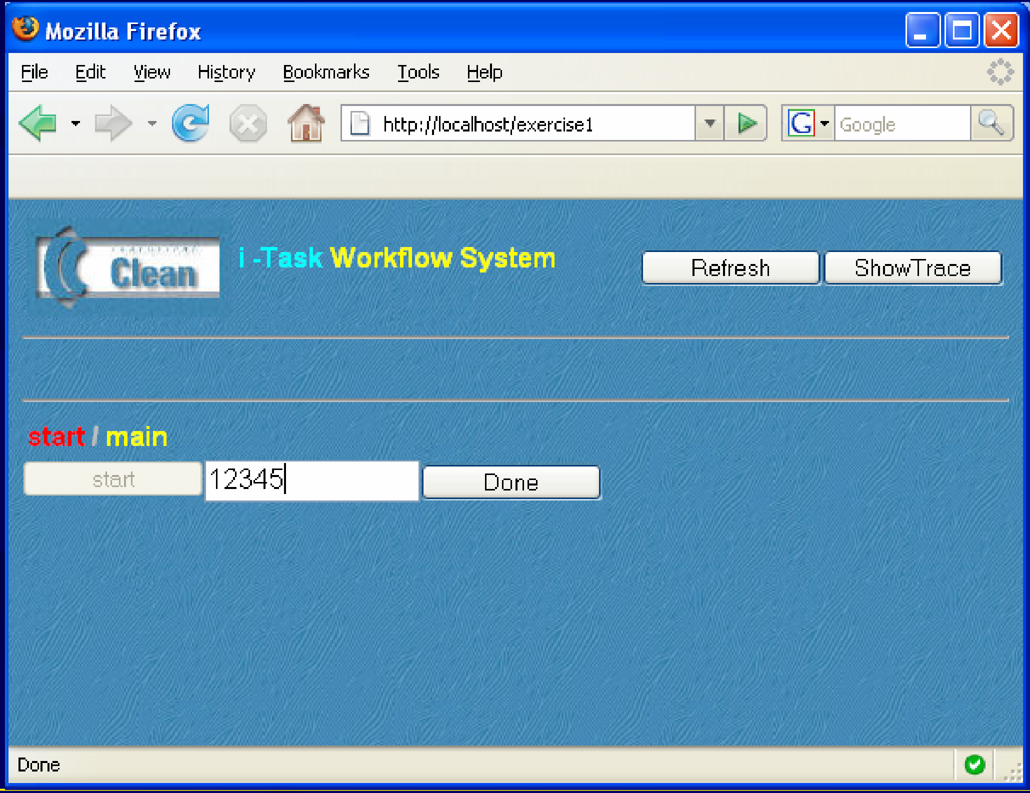
- Higher order tasks, Processes, Exception Handling, …

Executable workflow specification using standard web browsers

- All low level I/O handled automatically using *generic* programming techniques
  - Storage and retrieval of information, web I/O handling
- Declarative style of programming
  - Complexity of underlying architecture hidden
- One single application running distributed on server *and* clients

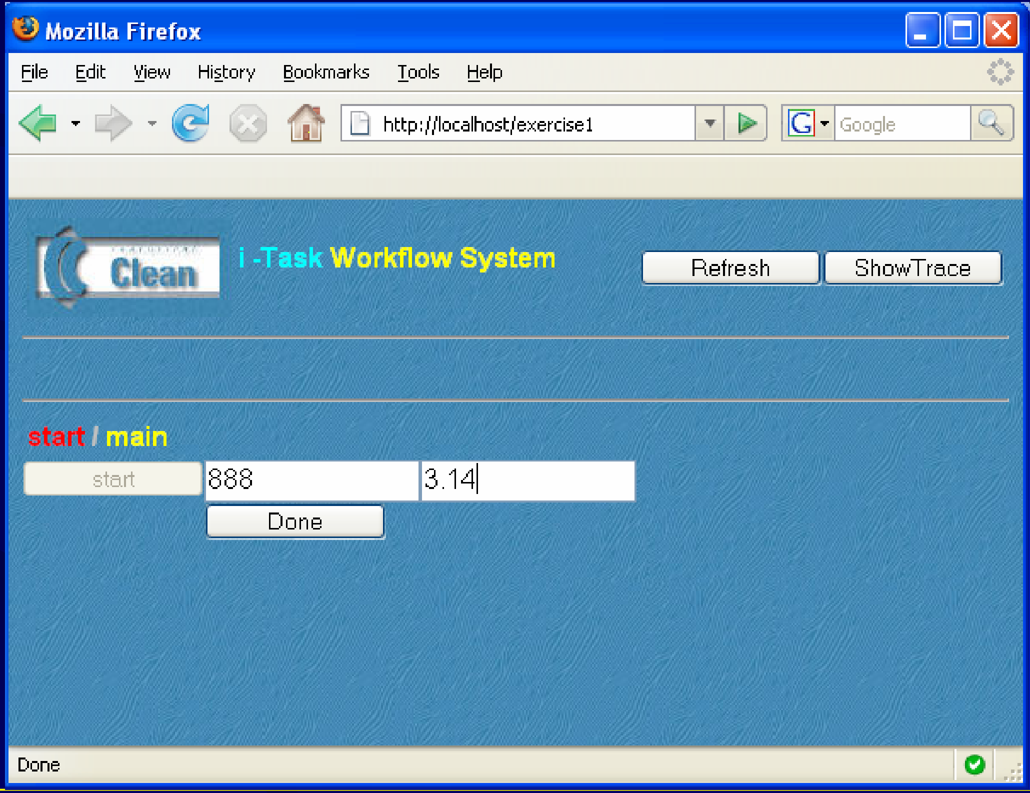
A very small *complete* example I

```
module exercise1

import StdEnv, StdiTasks

Start world = singleUserTask [ ] simple world



simple :: Task Int
simple = editTask "Done" createDefault
```



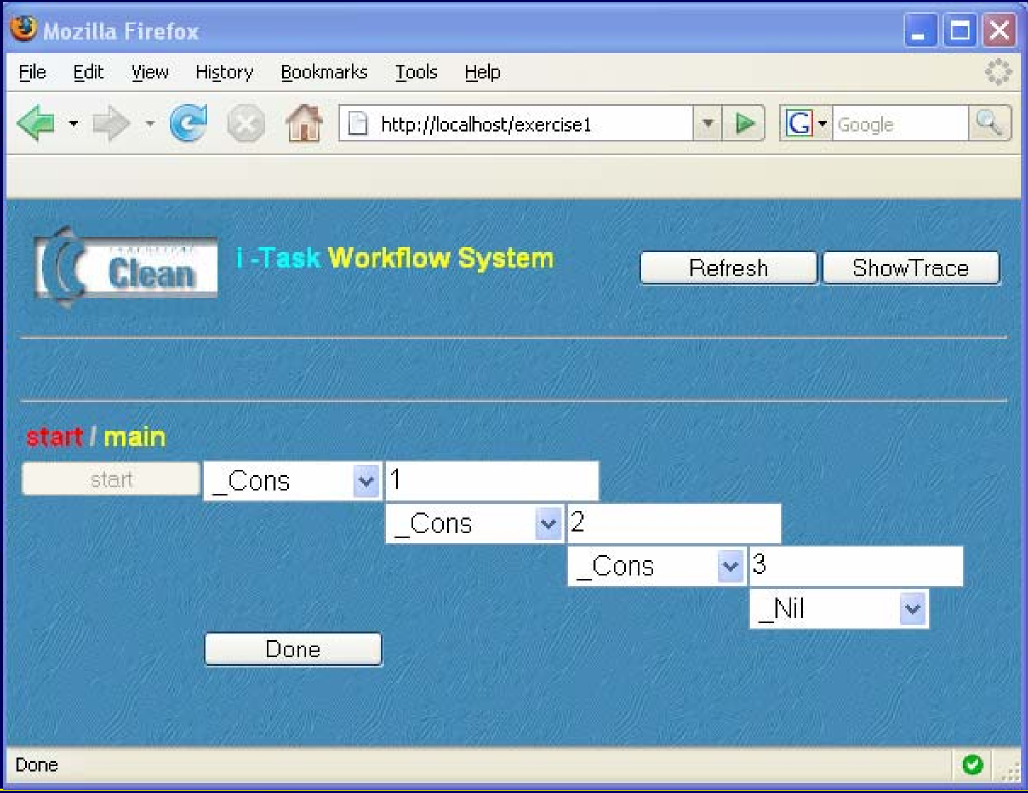
A very small *complete* example II

```
module exercise1

import StdEnv, StdiTasks

Start world = singleUserTask [ ] simple world

simple :: Task (Int, Real)
simple = editTask "Done" createDefault
```



A very small *complete* example III

```
simple :: Task [Int]
simple = editTask "Done" createDefault
```

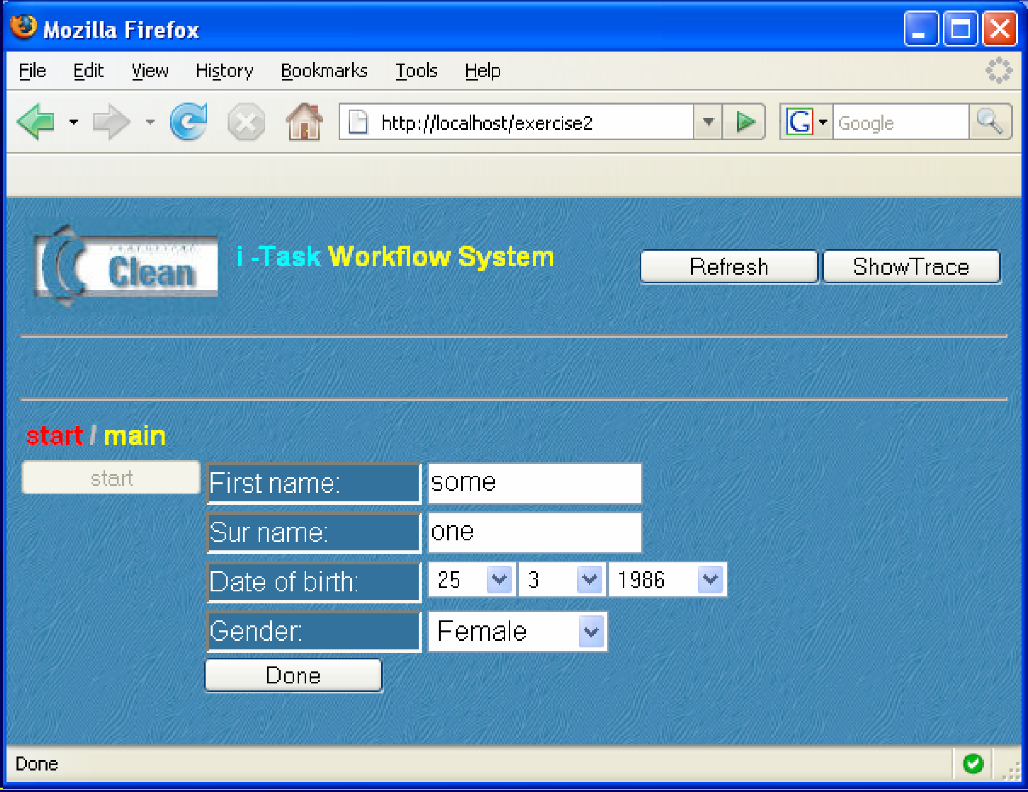


A very small *complete* example IV

```
:: Person =          { firstName      :: String
                     , surName        :: String
                     , dateOfBirth    :: HtmlDate
                     , gender         :: Gender
                     }
:: Gender =          Male
          |          Female

simple :: Task Person
simple = editTask "Done" createDefault
```

A very small *complete* example IV

```
:: Person =     { firstName     :: String
                , surName       :: String
                , dateOfBirth   :: HtmlDate
                , gender        :: Gender
                }
:: Gender =     Male
           |    Female

simple :: Task Person
simple = editTask "Done" createDefault
```



editTask

```
editTask        :: String a  → Task a                  | iData a
editTaskPred    :: a (a → (Bool, HtmlCode)) → Task a   | iData a

:: Task a  :== *TSt → *(a, *TSt)        // a Task is state transition function
:: TSt                                  // an abstract type
```

A task consist of an amount of work to be performed by the user involving ≥ 0 interactions

It is either not active, active, or finished.

editTask

```
editTask        :: String a  → Task a                    | iData a
editTaskPred    :: a (a → (Bool, HtmlCode)) → Task a     | iData a

:: Task a  :== *TSt → *(a, *TSt)          // a Task is state transition function
:: TSt                                    // an abstract type
```

A task consist of an amount of work to be performed by the user involving ≥ 0 interactions
It is either not active, active, or finished.

iData a is a context restriction for type a

In Haskell one would write:

```
editTask :: iData a => String → a  → Task a
```

- In Clean it is used not only to demand instances of overloaded functions for type a
- But it can also be used to demand instances of generic functions…

generic functions used by i-Task system

```
class iData a              | gForm {|*|} , iCreateAndPrint, iParse, iSpecialStore a
class iCreateAndPrint a    | iCreate, iPrint a
class iCreate a            | gUpd  {|*|} a
class iPrint a             | gPrint  {|*|} a
class iParse a             | gParse  {|*|} a
class iSpecialStore a      | gerda  {|*|},  read  {|*|},  write  {|*|}, TC a
```

It requires the instantiation of several generic functions for type "a" e.g.

gForm	gUpd	html form creation / form handling

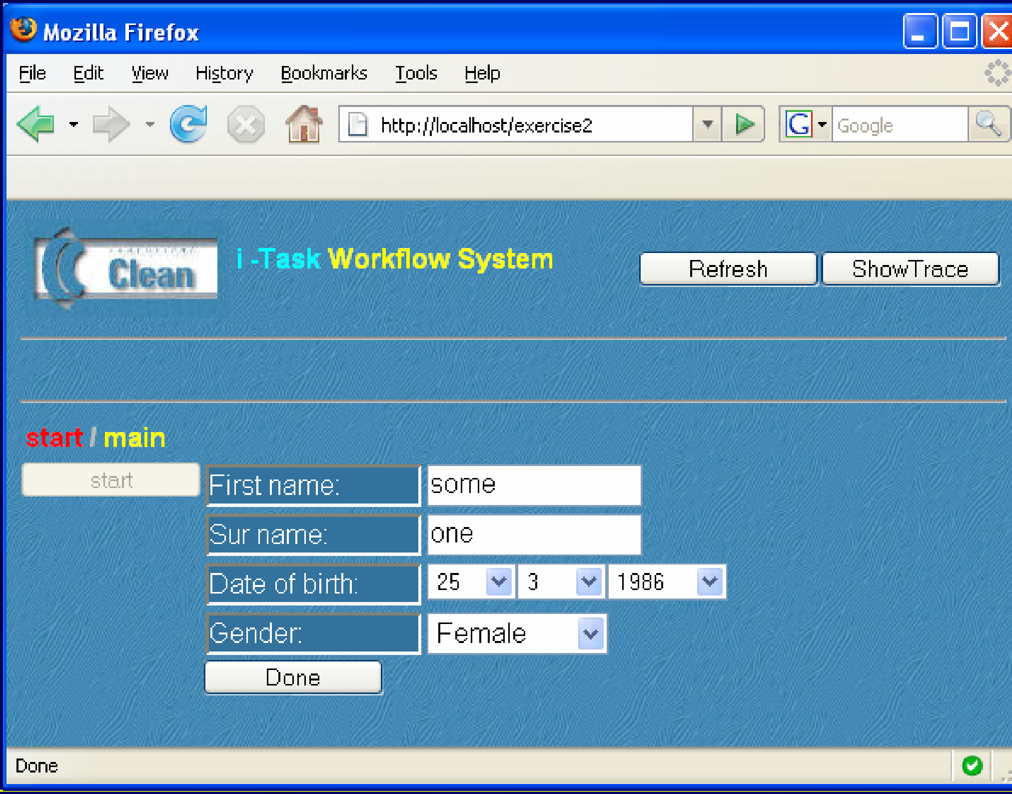
Serialization / De-Serialization for storage

gParse	gPrint	parsing / printing (in TxtFile, Page, Session)
gerda		storage and retrieval (in Database),
read	write	efficient binary reading / writing (in DataFile)
TC		conversion to and from Dynamics option used to store functions

all generic functions can, on request, automatically be derived by the compiler

A very small *complete* example IV

```
:: Person =      { firstName      :: String
                 , surName        :: String
                 , dateOfBirth    :: HtmlDate
                 , gender         :: Gender
                 }
:: Gender =      Male
          |      Female

simple :: Task Person
simple = editTask "Done" createDefault

derive gForm   Person, Gender
derive gUpd    Person, Gender
derive gParse  Person, Gender
derive gPrint  Person, Gender
derive gerda   Person, Gender
derive read    Person, Gender
derive write   Person, Gender
```



Options

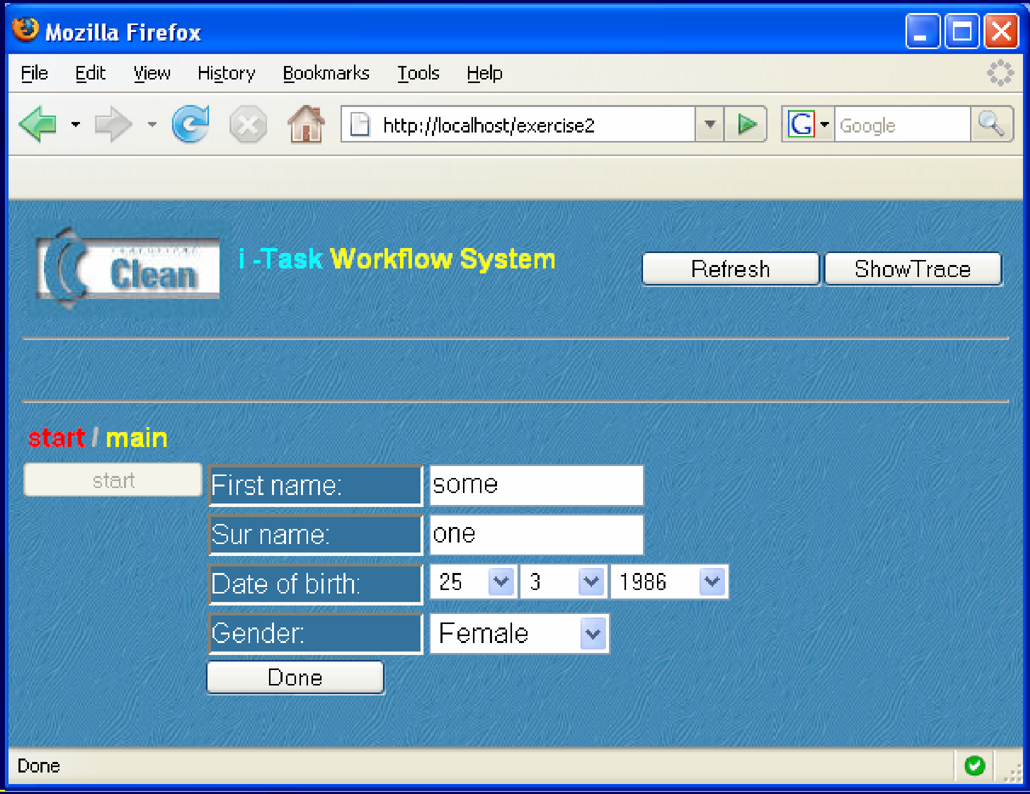
A task or any combination of tasks, can have several options:

```
class (<<@) infixl 3 b :: (Task a) b → Task a

instance <<@            Lifespan                          // default: Session
                 ,      StorageFormat                     // default: PlainString
                 ,      Mode                              // default: Edit
                 ,      GarbageCollect                    // default: Collect

:: Lifespan       = TxtFile | DataFile | Database         // persistent state stored on Server
                  | Session | Page                        // temp state stored in browser
                  | Temp                                  // temp state in application
:: StorageFormat  = StaticDynamic                         // to store functions
                  | PlainString                           // to store data
:: Mode           = Edit | Submit                         // editable
                  | Display                               // non-editable
                  | NoForm                                // not visible, used to store data
:: GarbageCollect = Collect | NoCollect                   // off: used for debugging & logging
```

A very small *complete* example IV

simple :: Task Person
simple = editTask "Done" createDefault

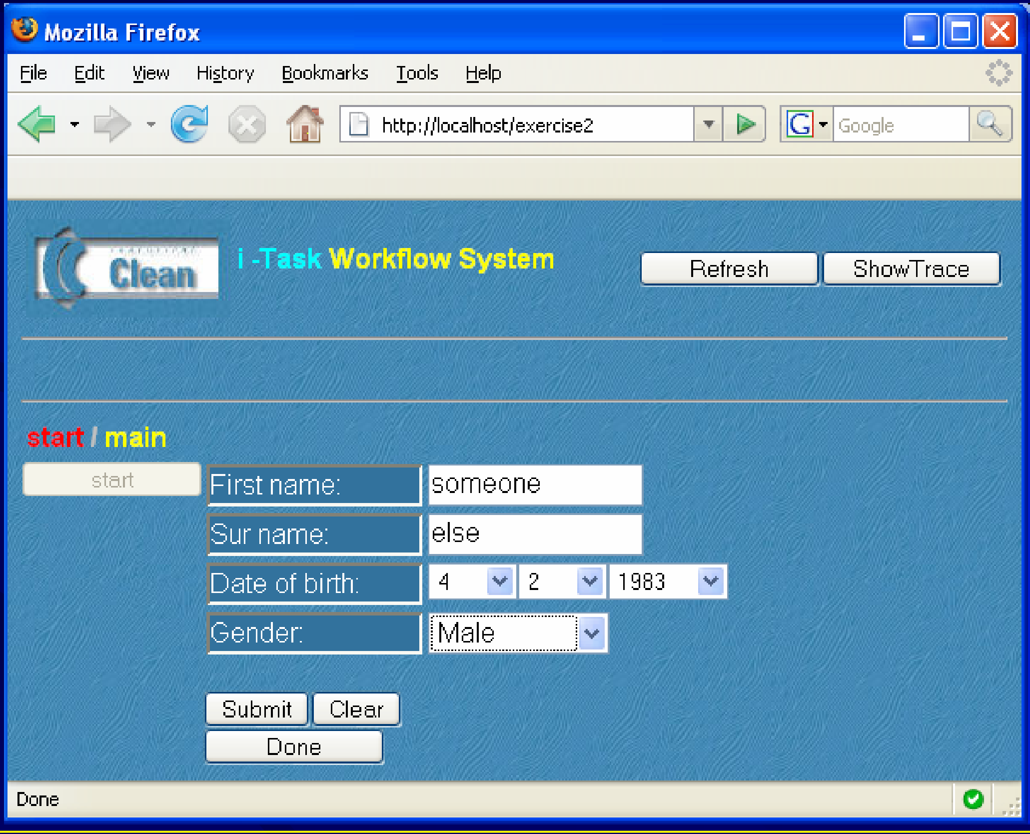
By default *any* change made in a form is transmitted to the clean application
Pressing "Done" means: task is finished



A very small *complete* example IV Submit

simple :: Task Person
simple = editTask "Done" createDefault <<@ Submit

Common behaviour: form is submitted when Submit is pressed, yet task not finished
Pressing "Done" means: task is finished

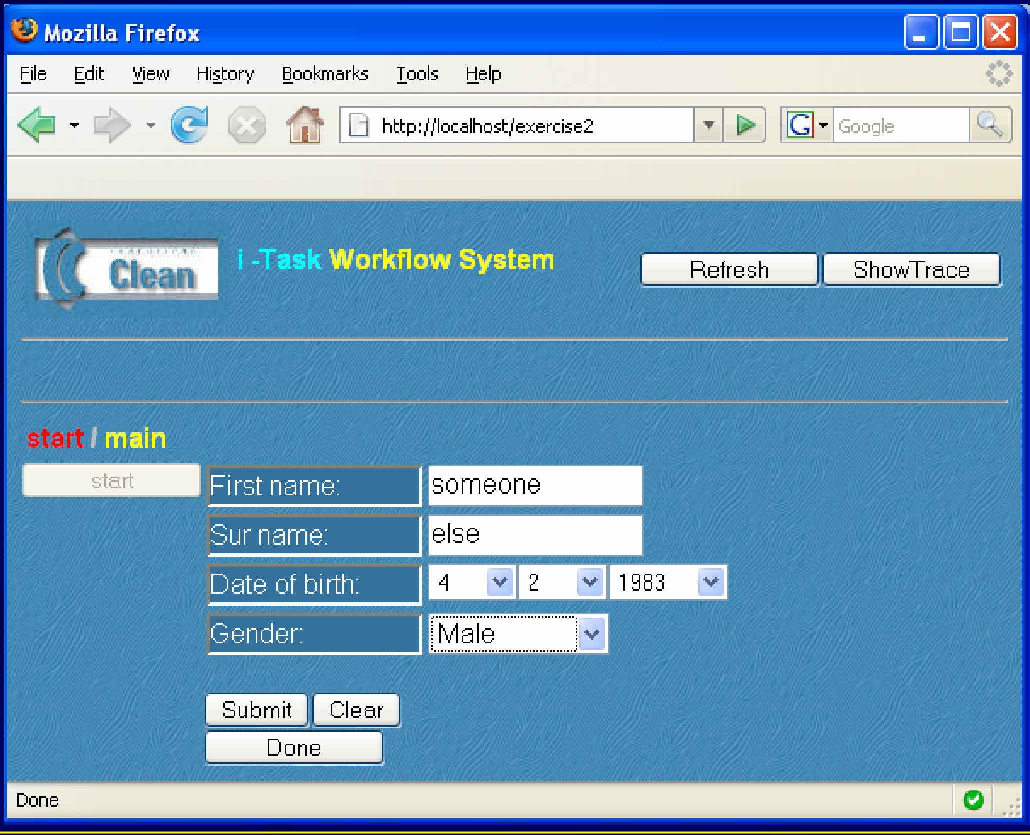


A very small *complete* example IV, Submit, TxtFile

```
simple :: Task Person
simple = editTask "Done" createDefault <<@ Submit <<@ TxtFile
```

Task(s) becomes persistent: status of the (partially evaluated) task is remembered

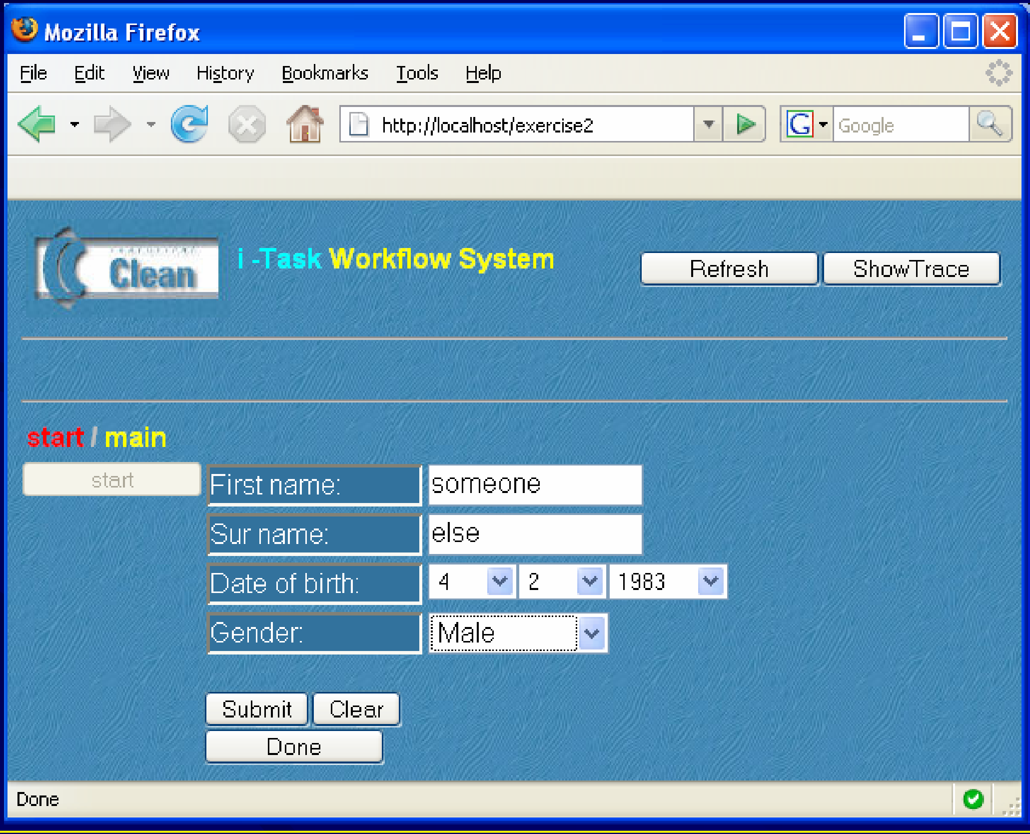
Important for multi-user applications.



A very small *complete* example IV, Submit, Database

simple :: Task Person
simple = editTask "Done" createDefault <<@ Submit <<@ Database

Task(s) becomes persistent, now stored in relational database
Important for multi-user applications.
Options switched by toggling flags



Some predefined combinators…

Sequencing of tasks: monads

```
(=>>)  infix 1 :: (Task a) (a → Task b)   → Task b    | iData b
return_V       :: a                       → Task a    | iData a
```

Assign a task to a user, every user has a unique id (UserId :== Int)

```
(@::)  infix 3 :: UserId (Task a)         → Task a    | iData a
```

Select 1 task to do out of n:

```
chooseTask     :: [(String,Task a)]       → Task a    | iData a
```

Or Task: do both tasks concurrently in any order, finish as soon as *one* of them completes

```
(-||-) infixr 3 :: (Task a) (Task a)      → Task a    | iData a
```

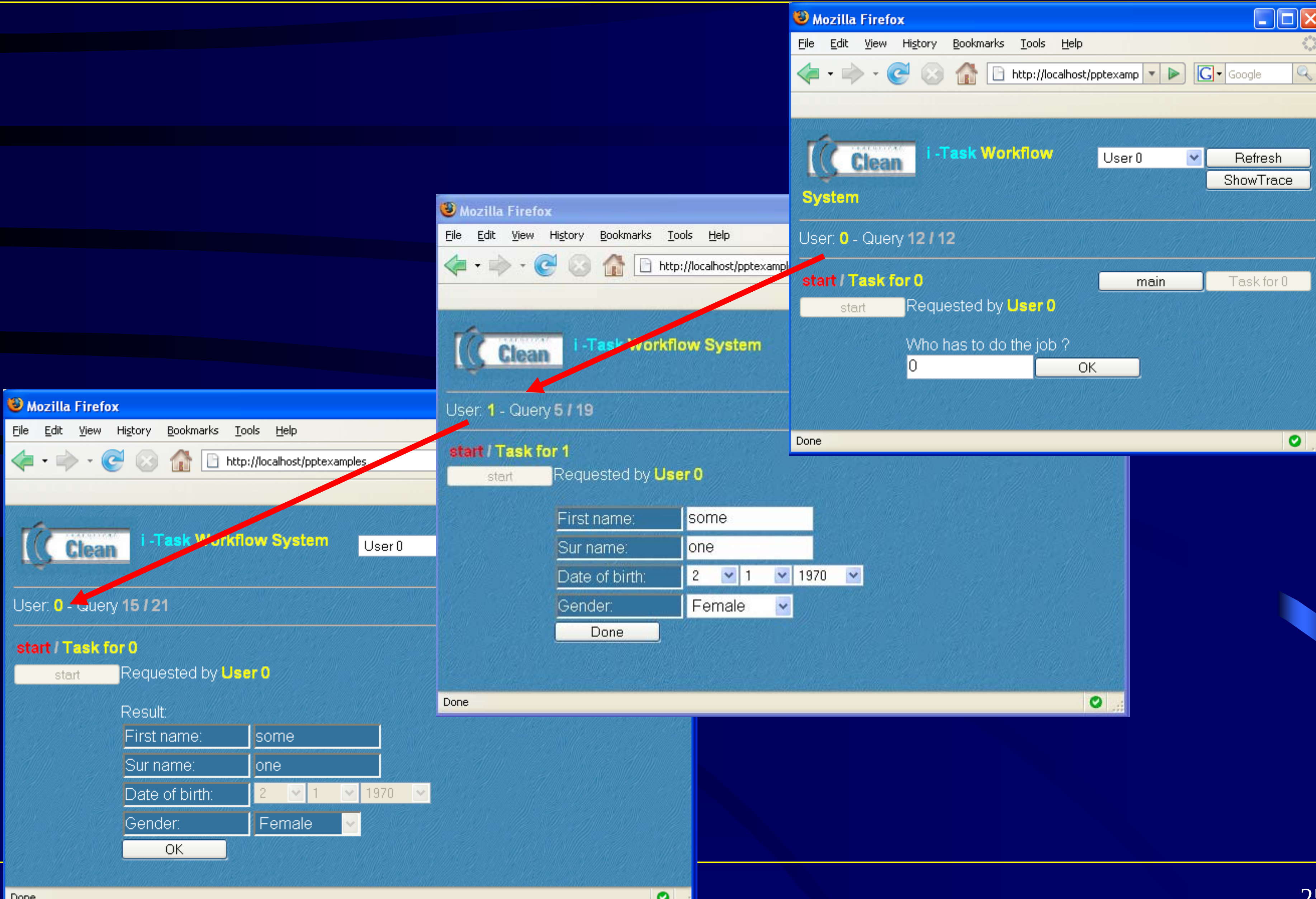
Repeat forever:

```
foreverTask    :: (Task a)                → Task a    | iData a
```

Prompting operator: displays Html text as long as a task is activated:

```
(?>>) infix 5  :: HtmlCode (Task a)       → Task a    | iData a
```

Assigning Tasks to Users



Assigning Tasks to Users

The actual assignment of tasks to users can be calculated dynamically:

```
delegate :: UserId (Task a) → Task a  | iData a
delegate boss task
=                                     boss       @:: [Txt "Who has to do the job ?"]
                                                     ?>> editTask "OK" createDefault
```

Assigning Tasks to Users

The actual assignment of tasks to users can be calculated dynamically:

```
delegate :: UserId (Task a) → Task a   | iData a
delegate boss task
=                                  boss      @:: [Txt "Who has to do the job ?"]
                                                 ?>> editTask "OK" createDefault
       =>> \employee →             employee @:: task
```

Assigning Tasks to Users

The actual assignment of tasks to users can be calculated dynamically:

```
delegate :: UserId (Task a) → Task a  | iData a
delegate boss task
=                               boss     @:: [Txt "Who has to do the job ?"]
                                             ?>> editTask "OK" createDefault
      =>> \employee →           employee @:: task
      =>> \result   →           boss     @:: [Txt "Result:", toHtml result]
                                             ?>> editTask "OK" Void
```

Assigning Tasks to Users

The actual assignment of tasks to users can be calculated dynamically:

```
delegate :: UserId (Task a) → Task a  | iData a
delegate boss task
=                              boss      @:: [Txt "Who has to do the job ?"]
                                             ?>> editTask "OK" createDefault
      =>> \employee →          employee @:: task
      =>> \result   →          boss      @:: [Txt "Result:", toHtml result]
                                             ?>> editTask "OK" Void
      =>>  \_       →                        return_V result
```

```
Start world = multiUserTask [ ] (delegate 0 some_nice_task) world
```

Different ways to start a workflow application…

```
definition module iTasksHandler

singleUserTask    :: [StartUpOptions] (Task a) *World → *World          | iData a

multiUserTask     :: [StartUpOptions] (Task a) *World → *World          | iData a

workFlowTask      :: [StartUpOptions] (LoginTask a) (TaskForUser a b)
                                              *World → *World          | iData b

:: LoginTask a        :== Task ((Bool, UserId), a)
:: TaskForUser a b :== UserId a → LabeledTask b
```

Semantics I - Types

```
:: ITask        = { val    :: Val
                  , ident  :: ID
                  , done   :: Done
                  }
:: Done         = Yes | No
:: Val          = Int  Int
                | Tuple (Val, Val)
:: ID           :== Int
:: Event        :== ITask
:: TasksToDo    :== [ ITask ]

:: ITaskComb    = Editor ITask                              // editor, input device
                | Sequence ITaskComb (Val -> ITaskComb)     // sequence, monadic bind
                | Return  Val                               // normal form, monadic return
                | Or   ITaskComb ITaskComb                  // or combinator
                | And ITaskComb ITaskComb                   // and combinator
```

Semantics II – Reduction Rules

Normal Form:

```
inNF :: ITaskComb → Bool
inNF (Return val) = True
inNF _            = False
```

One Step Reduction + Determining Active Editors for the next Reduction Step

```
Reduce :: ITaskComb (Maybe Event) TasksToDo → (ITaskComb, TasksToDo)

Reduce (Editor itask) Nothing todo = (Editor itask, [itask : todo])
Reduce (Editor itask) (Just event) todo
| event.ident == itask.ident
      | isFinished event.done        = (Return event.val, todo)
      | otherwise                    = (Editor event, [event : todo])
| otherwise = (Editor itask, [itask : todo])
where
      isFinished :: Done → Bool
      isFinished Yes = True
      isFinished No  = False
```

Basic Implementation Scheme: Task Tree Reconstruction

- Flow is specified in ***one*** Clean application serving ***all*** users
- An i-Task specification reads like a book
  - because it gives the ***illusion*** that it step-by-step interacts with the user like standard IO for a desktop application
  - In *reality* it starts from scratch every time information is committed, and dies
  - It reconstructs the Task Tree, starting from the root
    - finds previous evaluation point
  - It deals with Multiple Users
    - Sequential handling of requests: users are served one-by-one
  - It determines the resulting html code for *all* users
    - but it shows only the html code intended for a *specific* user
  - It stores state information in the html page, databases, files for the next request
    - Depending on the task options chosen

Optimization I: Global Task Rewriting

- *Can this be efficient?*
  - Over time, more and more tasks are created
  - the reconstruction of the Task Tree will take more and more time as well

- Speed-up re-construction of the Task Tree: *Global Task Rewriting*

  - Tasks are rewritten in (persistent) storages just like functions
    - The result of a task is remembered, not how a task accomplished

  - Tail recursion / repetition is translated to a Loop
    - Task Tree will not grow infinitely

  - Garbage collection of stored iTasks which are not needed anymore

- *The efficiency is not bad at all, but for large systems we can do better*

Optimization II: Local Task Rewriting – Basic idea

- Local Task Rewriting
  - Avoid complete Task Tree reconstruction all the way from the root
  - Only locally rewrite the different tasks (sub tree) a user is working on
  - Use "Ajax" technology and *only* update on web page what has to change

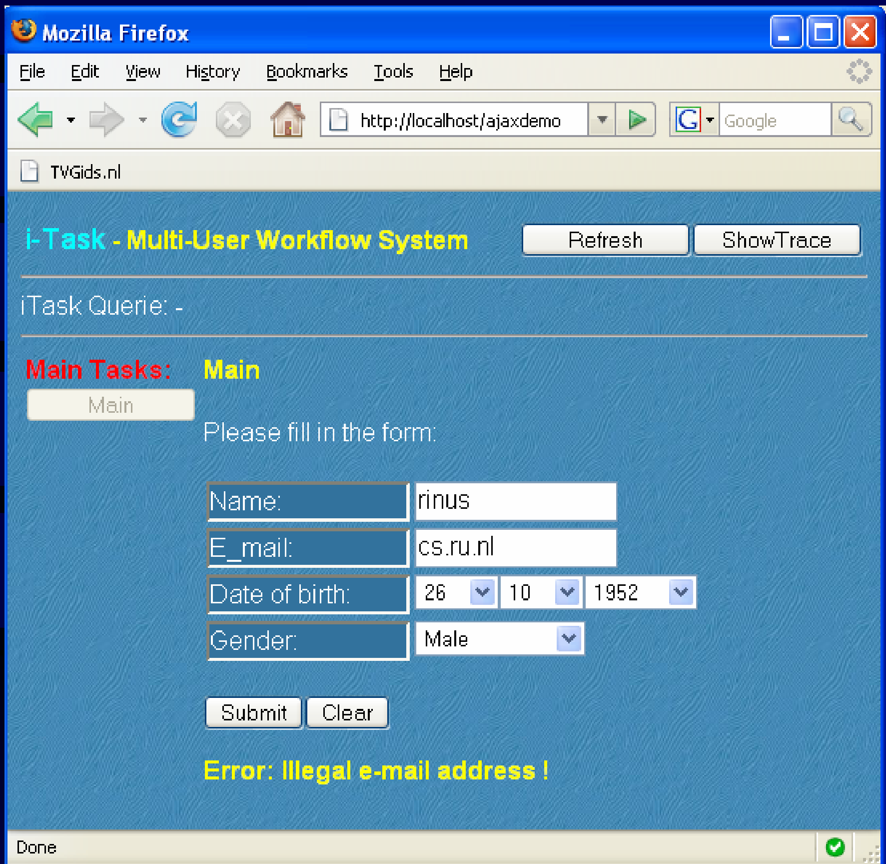
- Transparent: (almost) no changes in the original workflow specification
  - Each tasks assigned to a user with the @:: combinator is rewritten "locally"
  - Fine grain control: any i-Task can assigned to be rewritten "locally"
    - UseAjax @>> any_task_expression

Optimization II: Local Task Rewriting - Implementation

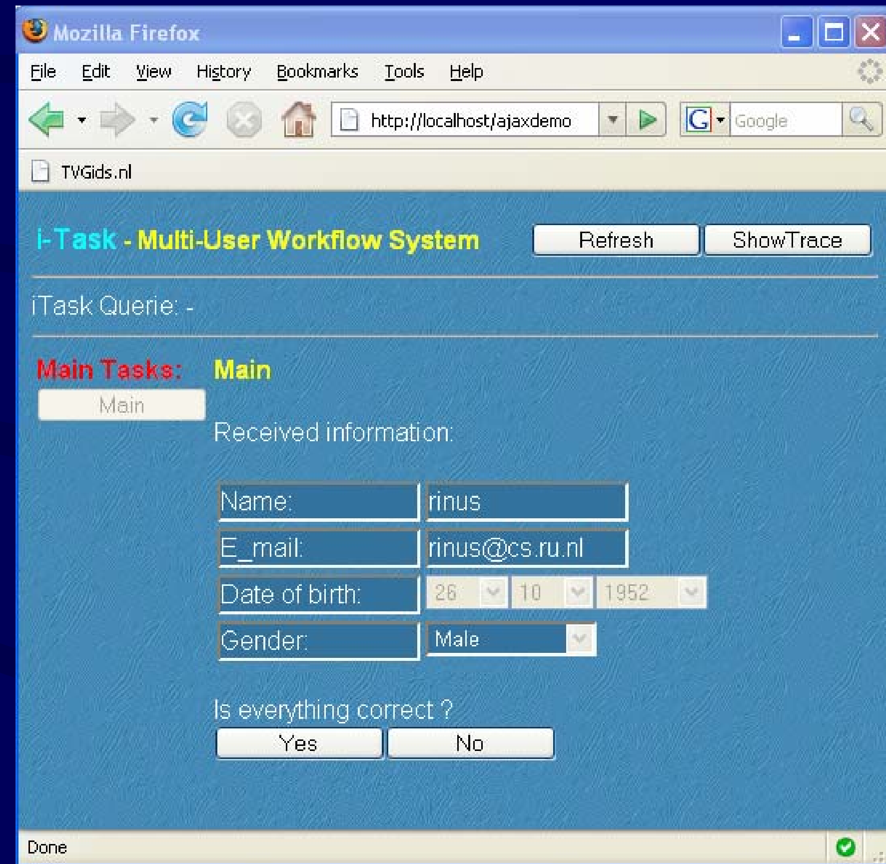
- Property: *any* Sub-Tree in the Task Tree can be reconstructed from scratch
- Thread Storage: to store closures: an iTask combinator call + its arguments
  - stored closure serves as kind of call-back function or thread which can handle *all* events of *all* subtasks in the subtree
- Global Effects Storage for every user
  - locally one cannot detect *global* effects
  - administrate which tasks are deleted, the fact that new tasks are assigned
- Rewrite-o-matic: from *Local* Task Rewriting stepwise to *Global* Task Rewriting
  - Threads can be nested, and can partly overlap
    - when a thread is finished locally rewrite parent thread, and so on…
  - Switch back to top level Global Task Rewriting
    - when parent thread belongs to another user
    - when there are global effects administrated affecting the user

Example: Check and Double-Check

Check 1: by predicate



Check 2: by application user



One can imagine that this is all done on the Client side

Check and Double-Check i-Task Specification

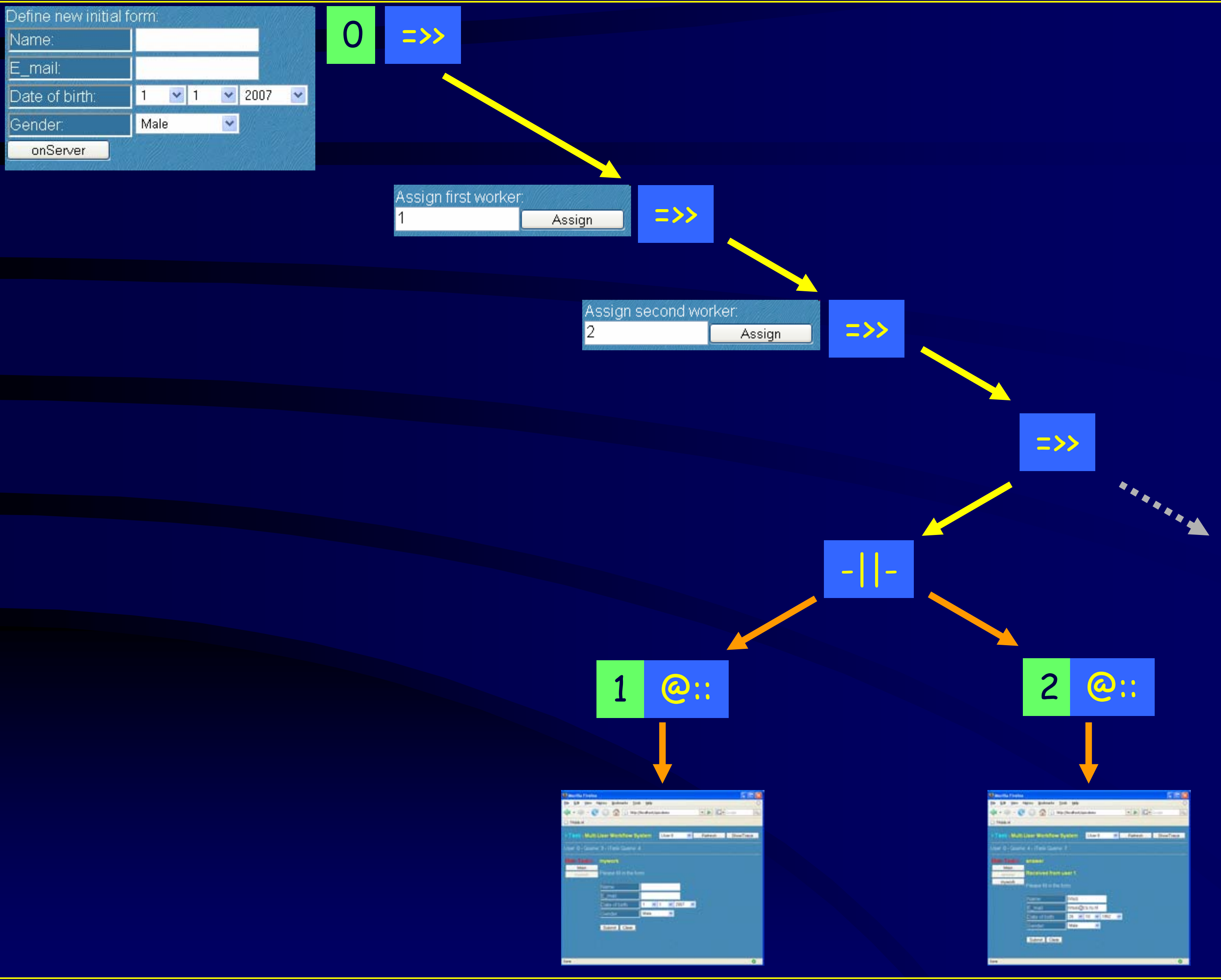
General Recipe to check and double-check the correctness of any value of any type…

```
doubleCheckForm :: a (a → (Bool, [BodyTag])) → Task a    | iData a
doubleCheckForm a preda
=                    [Txt "Please fill in the form:"]
                     ?>> editTaskPred a preda

      =>> \na →      [Txt "Received information:", toHtml na, Txt "Is everything correct ?"]
                     ?>> chooseTask       [ ("Yes", return_V na)
                                          , ("No",  doubleCheckForm na preda)
                                          ]

doubleCheckPerson :: Person → Task Person
doubleCheckPerson = doubleCheckForm  createDefault checkPerson
where checkPerson person = …

example = doubleCheckPerson createDefault
```

Delegate: assigning tasks to users

```
example :: Task Person
example = foreverTask delegate

delegate
=                       [Txt "Define new initial form:"]
                        ?>> editTask "onServer" createDefault

      =>> \fi →         [Txt "Assign first worker:"]
                        ?>> editTask "Assign" 1

      =>> \w1 →         [Txt "Assign second worker:"]
                        ?>> editTask "Assign" 2

      =>> \w2 →         fillform w1 fi -||- fillform w2 fi

      =>> \fr →         [Txt "resulting form received from fastest worker:", toHtml fr]
                        ?>> editTask "OK" Void
where
      fillform w f = w @:: doubleCheckPerson f
```

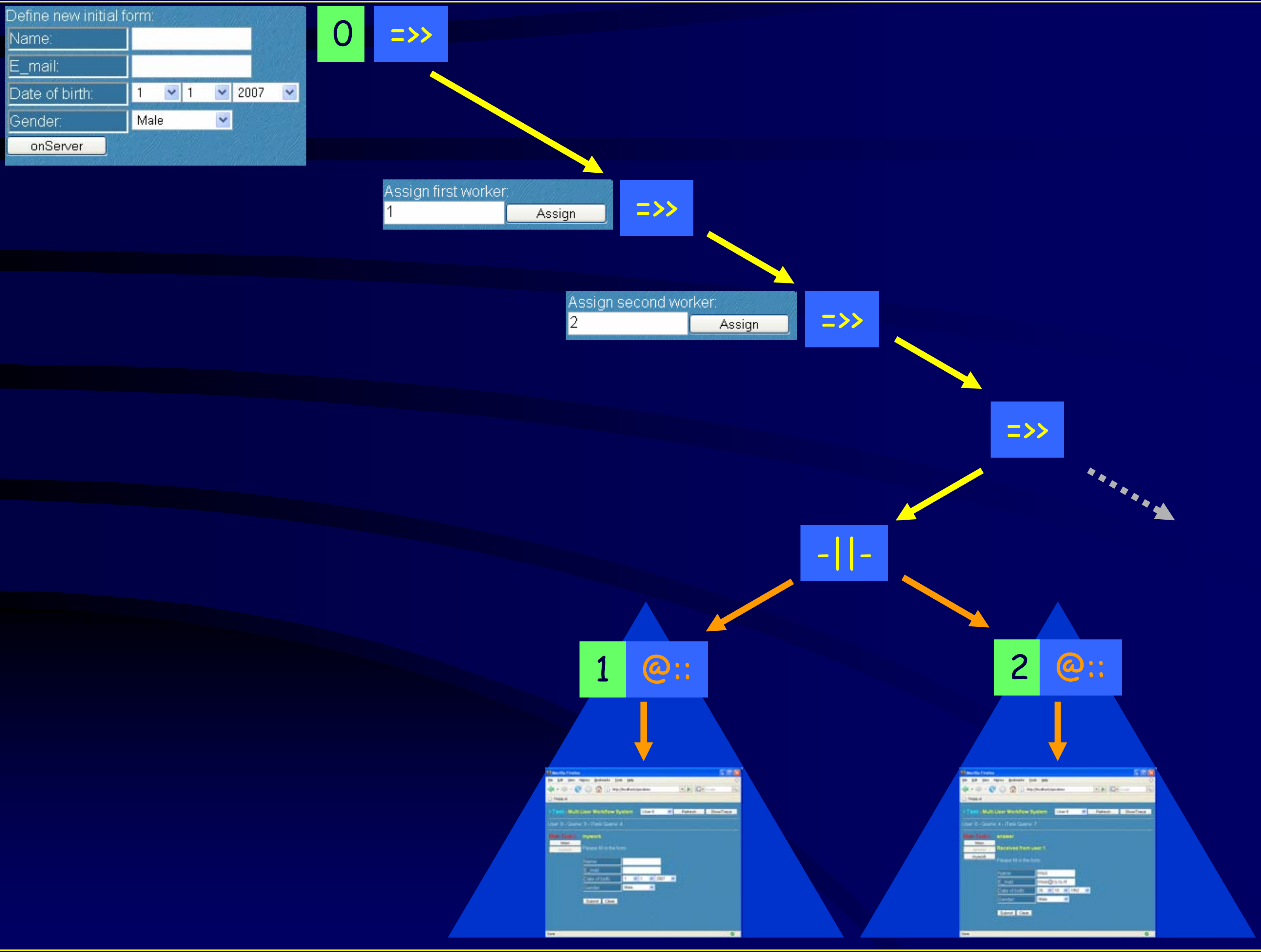
Delegate – Task Tree Snapshot



Delegate using Ajax

```
example :: Task Person
example = foreverTask delegate

delegate
=                  [Txt "Define new initial form:"]
                   ?>> editTask "onServer" createDefault

     =>> \fi →     [Txt "Assign first worker:"]
                   ?>> editTask "Assign" 1

     =>> \w1 →     [Txt "Assign second worker:"]
                   ?>> editTask "Assign" 2

     =>> \w2 →     fillform w1 fi -||- fillform w2 fi

     =>> \fr →     [Txt "resulting form received from fastest worker:", toHtml fr]
                   ?>> editTask "OK" Void
where
     fillform w f = w @:: doubleCheckPerson f
```

Delegate Ajax – Task Tree Snapshot



Optimization III: Client Side Local Task Rewriting

- Even better to avoid web traffic overhead: *Client Side* Local Task Rewriting

- Transparent: (almost) no changes in the original workflow specification
  - In the workflow specification, *any* i-Task can be turned into a *Client Thread*
    - OnClient @>> any_task_expression

Delegate using Sapl & Ajax

```
example :: Task Person
example = foreverTask delegate

delegate
=                       [Txt "Define new initial form:"]
                        ?>> editTask "onServer" createDefault

       =>> \fi →        [Txt "Assign first worker:"]
                        ?>> editTask "Assign" 1

       =>> \w1 →        [Txt "Assign second worker:"]
                        ?>> editTask "Assign" 2

       =>> \w2 →        fillform w1 fi -||- fillform w2 fi

       =>> \fr →        [Txt "resulting form received from fastest worker:", toHtml fr]
                        ?>> editTask "OK" Void
where
       fillform w f = w @:: OnClient @>> doubleCheckPerson f
```

Optimization III: Client Side Local Task Rewriting

- *The whole i-Task machinery has to run in the browser as well*
  - We use Jan-Martin Jansen's SAPL interpreter: fastest, small, in C & Java (TFP '06)
  - The whole Clean iTask application is compiled to SAPL code
    - "simple" iTask: > 7000 functions, functions can be large (> 20.000 chars)
  - The SAPL interpreter + SAPL iTask code is loaded as Java Applet in the web page
  - 2 *almost* identical iTask images: Clean .exe on server, SAPL code on Client
  - A Clean function call can be translated to an equivalent SAPL function call
  - When a Client thread is created (SAPL), a Server thread is made as well (Clean)
    - We can *choose* where to evaluate: Client or Server
    - If it cannot be done on the Client, we can do it on the Server

Optimization III: Client Side Local Task Rewriting

- When an event occurs, we know it's prime destination: Client or Server
  - The Client basically performs the same actions as the Server but it cannot deal with
    - global effects
    - persistent storage handling (access to files, databases)
    - parent threads from other users
    - threads to be evaluated on server
    - new threads created for other users
  - Rewrite-o-matic
    - in case of panic the Client evaluation stops
    - switch back to *Server Side* Local Task Rewriting

Conclusions

Advantages over Commercial Systems

- Executable specification, but not yet as declarative as envisioned

- Workflows are dynamically constructed
  - Flow can depend on the actual contents
- Workflows are statically typed, input type checked as well
- Highly reusable code: polymorphic, overloaded, generic
- Fully compositional
- Higher order: resulting work can be a workflow -> shift work to someone else

- It generates a multi-user web enabled workflow system
- Runs on client or server, as demanded
-
- One application => easier to reason
  - Technical very interesting architecture, general applicable
  - Distributed Database, operating system, not only for web applications

- Intuitive for functional programmers
  - but probably not for other programmers ???

Lots of work to do…

- More Real Life Examples needed:
  - Car Damage Subrogation System (IFL 2007, Erik Zuurbier)
  - Conference Management System (AFP 2008 Summerschool)
  - Planned:
    - Logistic Control System (Dutch Navy)
    - Crisis Management System (Navy, Ministry of National Affairs)

- Improve Practical Application
  - Robustness ? Performance ? Scaling ? Security ? Software evolution ?
  - Embedding with existing databases, workflow systems, main stream web tools
  - Improve implementation:
    - Controlling parallel applications
    - Distributed Servers
  - Exploit flexibility and total overview:
    - Improve feedback and control given to the manager: adjust a running system
  - Powerful editors on Client: full text editors, drawing of pictures, etc.

- Theoretical foundation
    - Semantics ? Soundness ?

- Can we define a declarative system on top of it ?