Keeping the flow going

Data-flow oriented workflow systems

Annette Bieniusa

joint work with Adriaan Middelkoop(vwd) and Alexander Mattes

„How do you want to invest your money?“

Financial Advisory Systems

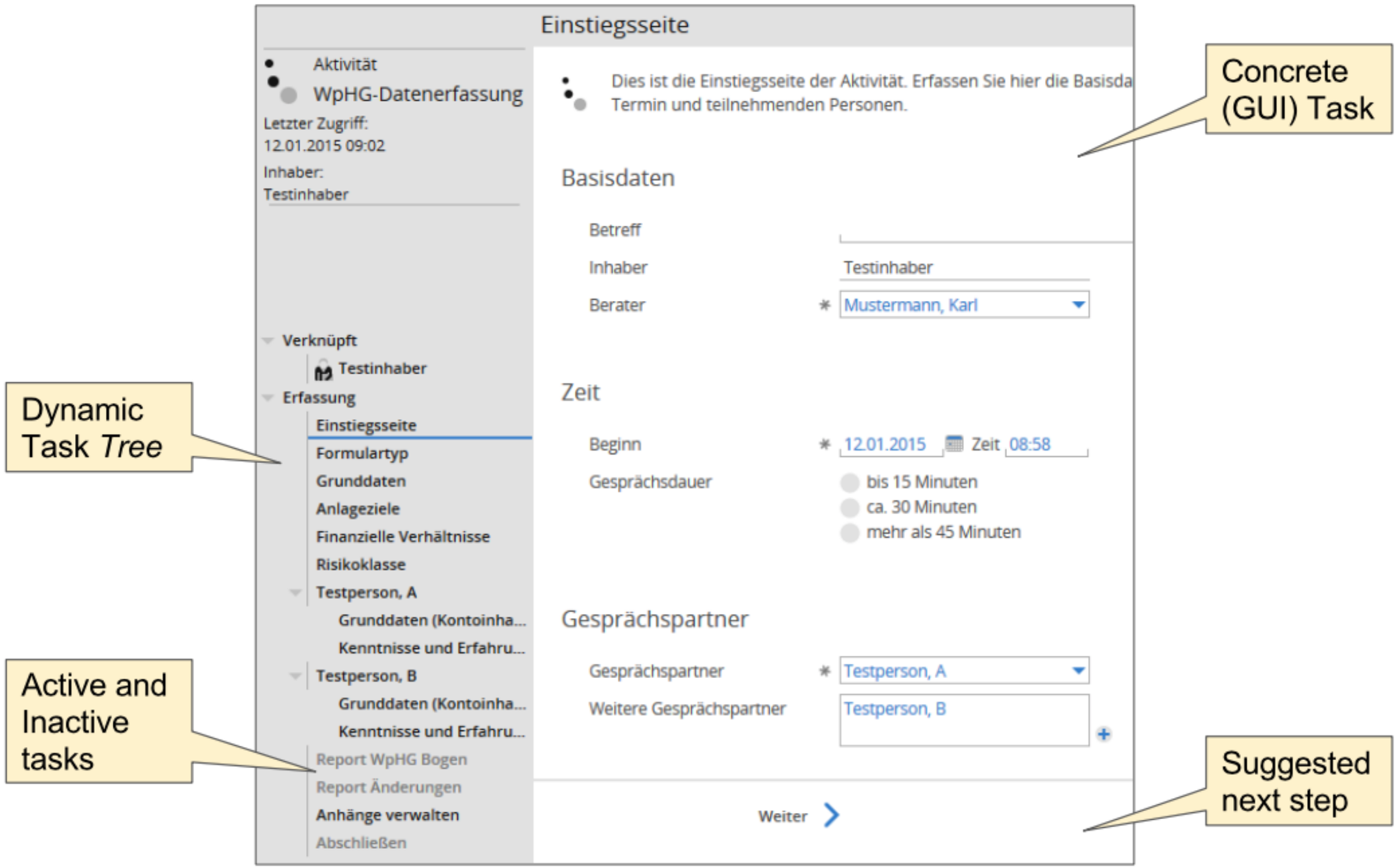
- Financial Advisory
  - Manage estate of clients
  - Giving clients individual advice

Workflows

Dynamic collection of interdependent tasks

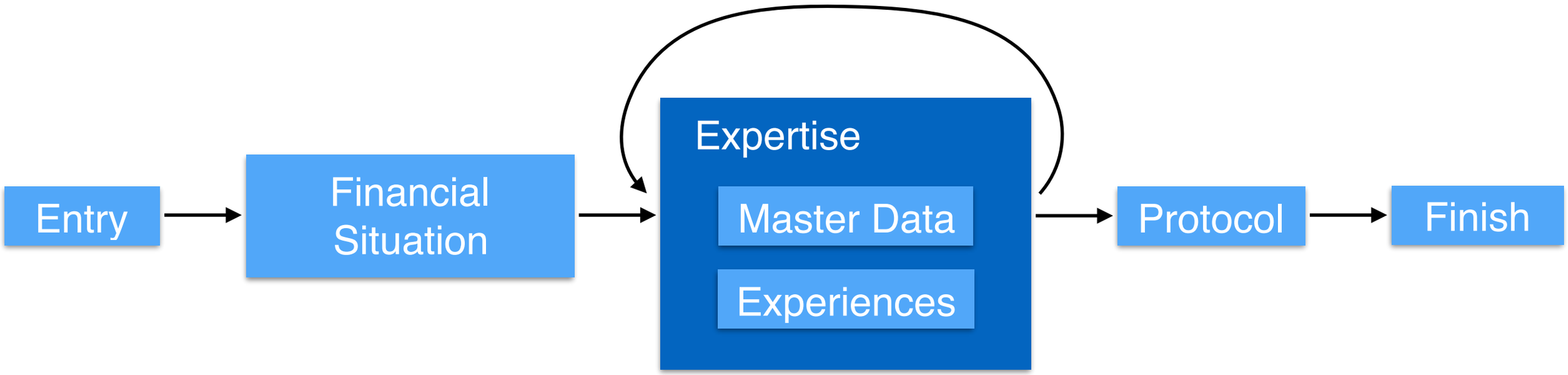
- Infrastructure required
  - Enforce regulation and support supervision
  - Automate parts of the processes

User Interface



The control-driven approach

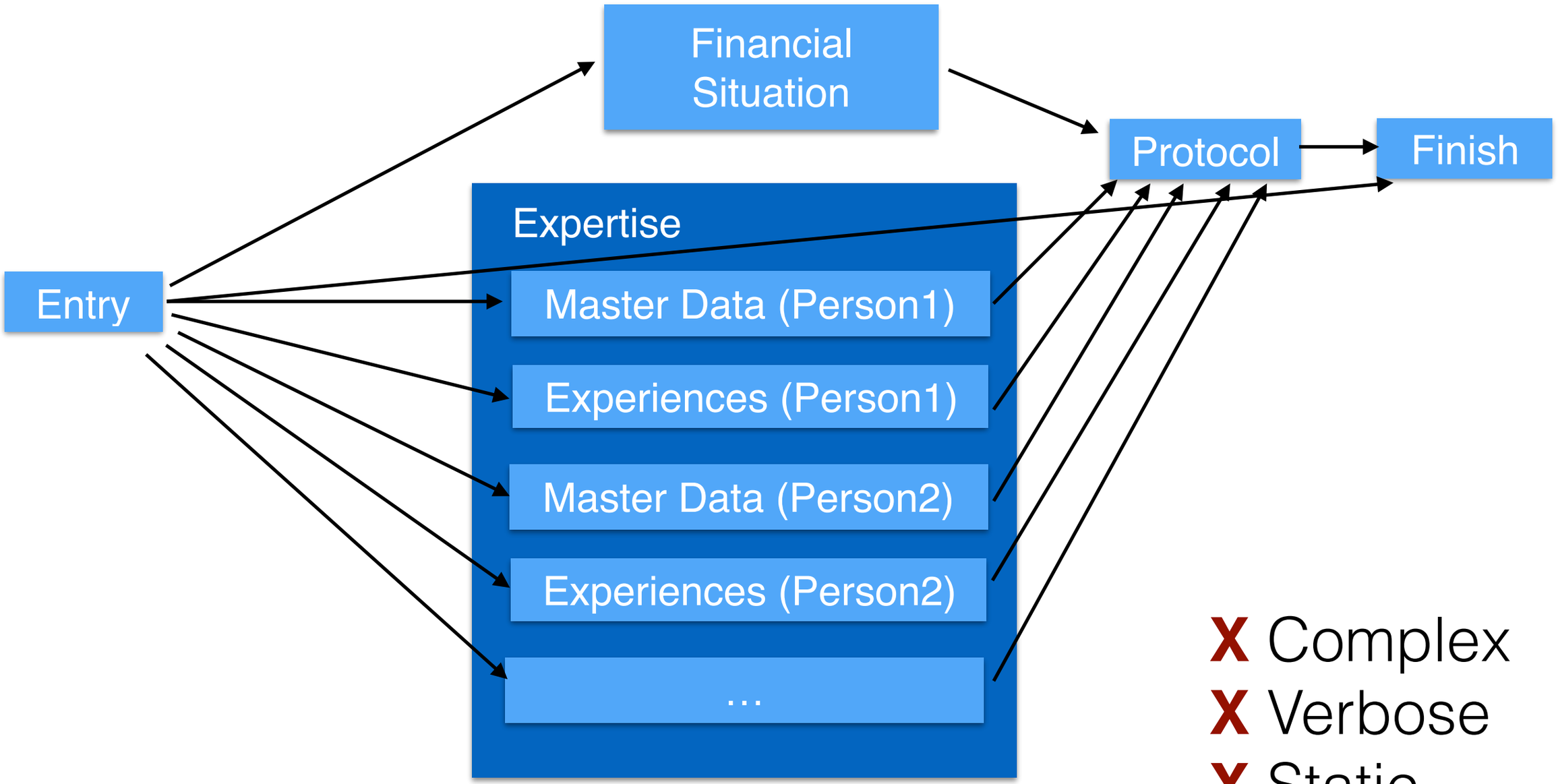
Workflow Example: Acquisition of client profile



X Fixed task order

X Explicit management of shared state

The data-driven approach



X Complex
X Verbose
X Static

Use dynamic, higher-order reactive programming to define the (dynamic) dataflow graph

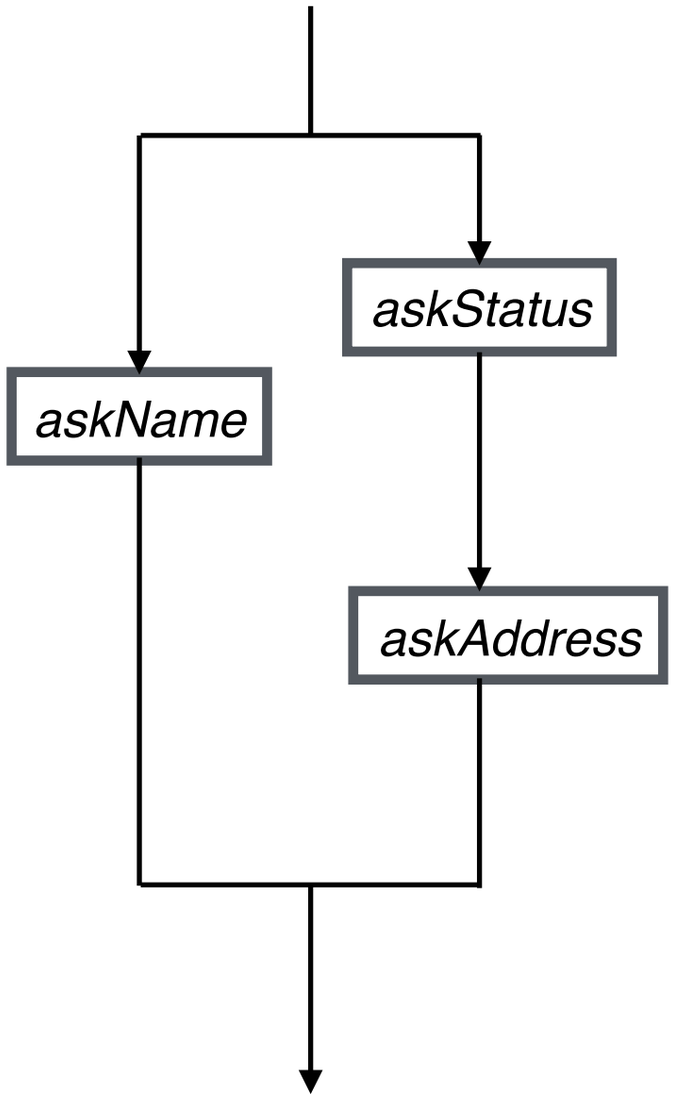
Defining the Tasks

askName :: Task () String
askName = …

askStatus :: Task () Status — *private person or company*
askStatus = …

askAdress = Task Status String
askAdress = …

workflow :: Task () (String, String)
workflow = … — *combination of Tasks*



Types of Tasks

- Need to program the (dynamic) dataflow graph for the workflow

- Idea: Dynamic, higher-order reactive programming

```
data Task a b where
    Pure     :: (a -> b)      -> Task a b
    Impure   :: (a -> IO b) -> Task a b
    Serial   :: Task a x -> Task x b -> Task a b
    Parallel :: Task u x -> Task v y -> Task (u,v) (x,y)
```

here: Task

Tasks as Arrows

- An arrow a b c represents „a computation with input type b delivering something of type c“ [Hughes 00]

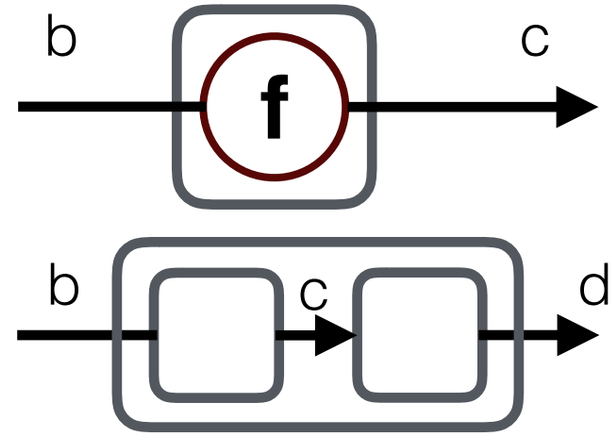
- Generalizes the monad concept by introducing dependence on input, e.g.

  - If you are a finance novice, only safe financial products may be proposed

  - Different type of address depending on the status of the advised person

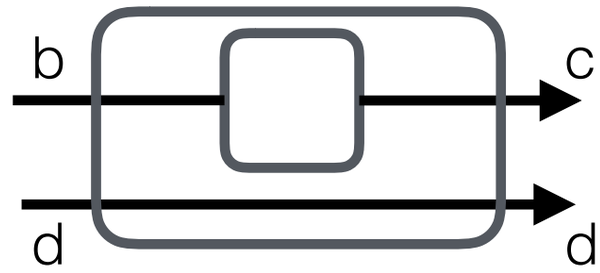
How to construct an Arrow

- Build an arrow out of a function:
  arr :: (Arrow a) => (b -> c) -> a b c

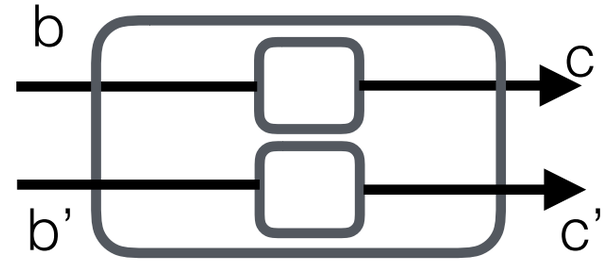


- Arrow composition (sequentiell)
  (>>>) :: (Arrow a) => a b c -> a c d -> a b d

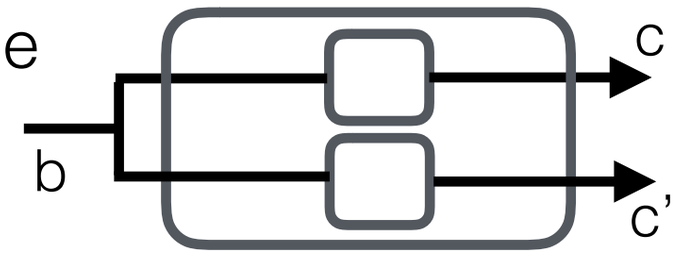
- Take an arrow and transform first
  first :: (Arrow a) => a b c -> a (b, d) (c, d)



- Run one arrow on the first item of the pair and one arrow on the second item of the pair
  (***) :: Arrow a => a b c -> a b' c' -> a (b, b') (c, c')



- Combine two arrows by running both on the same value
  (&&&) :: Arrow a => a b c -> a b c' -> a b (c, c')

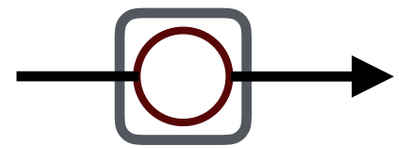


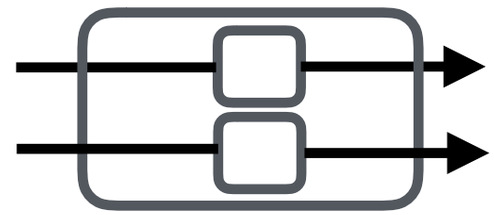
Tasks as Arrows

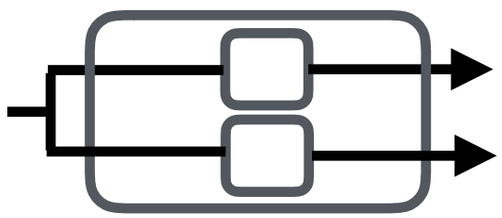
```
data Task a b where
    Pure     :: (a -> b)      -> Task a b
    Impure   :: (a -> IO b) -> Task a b
    Serial   :: Task a x -> Task x b -> Task a b
    Parallel :: Task u x -> Task v y -> Task (u,v) (x,y)
```

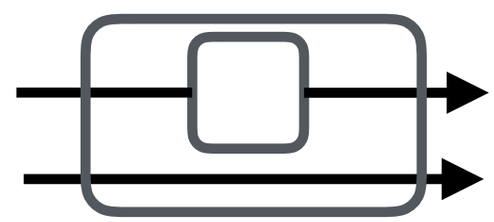
```
instance Cat.Category Task where
  id = Pure id
  t1 . t2 = Serial t2 t1

instance Arrow Task where
  arr = Pure
  t1 *** t2  = Parallel t1 t2
  t1 &&& t2 = Pure (\a -> (a,a)) >>> t1 *** t2
  first t = t *** Cat.id
  second t = Cat.id *** t
```







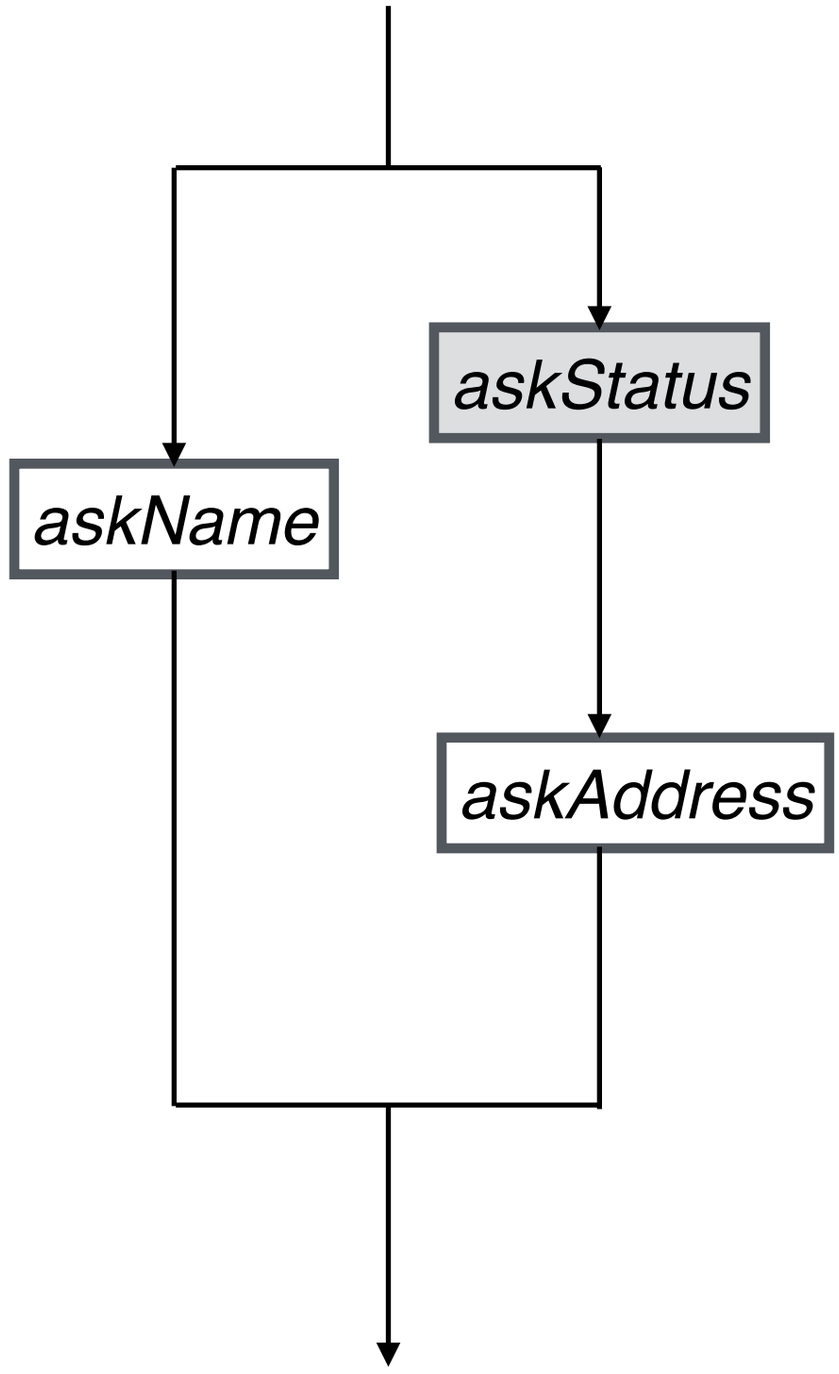


Keep the data flowing

- Dynamic workflow execution
  runTask :: Task a b -> a -> IO b

- Finds the tasks can be processed next

- Allows re-execution of tasks



ArrowTasks

- Haskell library for task-based workflows
- Integrated in Web-Framework (based on HappStack)
- Work in progress:
  - Collaborative workflows
  - Tasks as microservices with workflow API that wraps around a conventional microservice