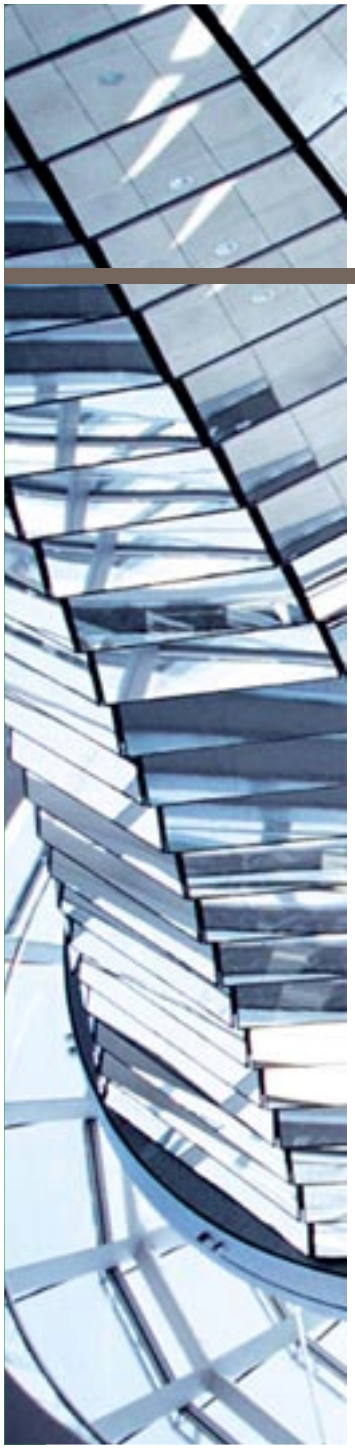
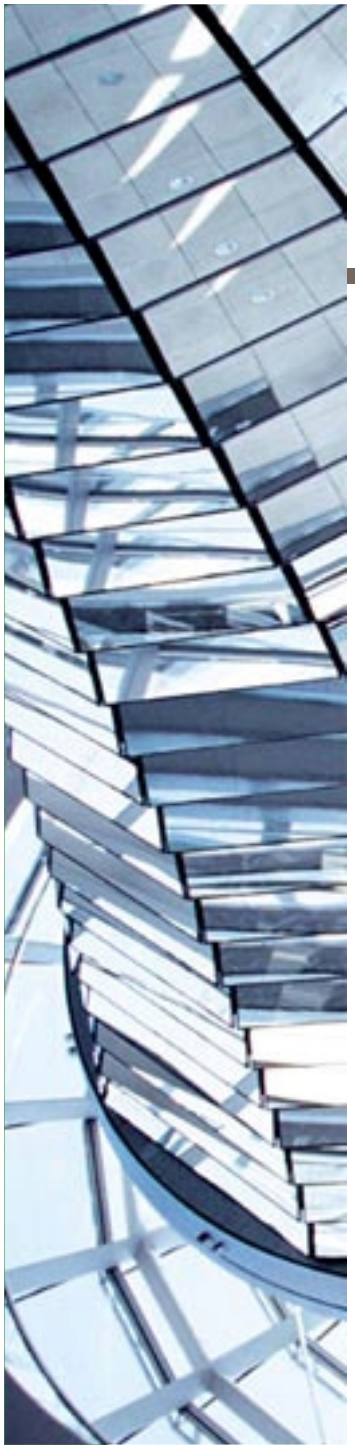


Language-Oriented Programming
am Beispiel Lisp

Arbeitskreis Objekttechnologie Norddeutschland
HAW Hamburg, 6.7.2009

Prof. Dr. Bernhard Humm
Hochschule Darmstadt, FB Informatik
und Capgemini sd&m Research

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES



Agenda

- Introduction & Example
- A few words on Lisp
- DSLs for business information systems
- The big picture
- Agile development
- Conclusion

My Background

- Diplom-Informatiker, Universität Kaiserslautern:
  - Lisp as first programming language
  - Focus on artificial intelligence (AI): thesis on machine learning
- Ph.D. Computer Science, University of Wollongong, Australia:
  - Thesis on transactions in distributed object-oriented systems
- 11 years with sd&m:
  - Development of large-scale business information systems
  - Developer, chief architect, project manager, department manager, head of sd&m Research
- Hochschule Darmstadt – University of Applied Sciences
  - Professor for software engineering and project management
  - Focus on software architecture

Language-Oriented Programming

Instead of simply writing your application in the base language, you build on top of the base language a language for writing programs like yours, then write your program in it.

Paul Graham, Hackers & Painters, 2004

- DSL = Domain-Specific Language

{ZUR DISKUSSION GESTELLT / IDEALE PROGRAMMIERWERKZEUGE

Auf dem Weg zu idealen Programmierwerkzeugen – Bestandsaufnahme und Ausblick

Johannes Brauer · Christoph Crasemann · Hartmut Krasemann

Einleitung

Geprägt durch jahrzehntelange Erfahrungen in der Praxis der Entwicklung und dem Betrieb großer Softwaresysteme diskutieren die Autoren, inwieweit die nach wie vor mannigfaltigen Probleme der Produkt- und Prozessqualität bei Software ihre Ursache in den Programmierwerkzeugen haben. Die Programmiersprache als das grundlegendste Werkzeug steht dabei zwar im Vordergrund, aber für den effektiven Einsatz einer Sprache in der Praxis sind Entwicklungsumgebung und Laufzeitsystem ebenso erforderlich.

Wenn im Zusammenhang mit Problemen oder Unzulänglichkeiten von Programmiersprachen häufiger Java zitiert wird, geschieht dies stellvertretend für alle industriell relevanten Sprachen, wie z.B. C#, C++ oder auch COBOL.

Die Autoren glauben nicht, dass die in der Praxis so „populäre" Programmiersprache Java einen Endpunkt der Entwicklung dieses Teilgebiets der Informatik markiert und nur noch an der Weiterentwicklung dieser Sprache gearbeitet wird. Im Gegenteil befinden sich die gängigen Sprachen u.E. eher auf toten Ästen des Programmiersprachenstammbaums, die möglicherweise längst abgebrochen wären, wenn sie nicht durch immer neue „Stützungsmaßnahmen" daran gehindert würden. So könnte z.B. der stete Strom neuer Rahmenwerke und Werkzeuge, die offenbar erforderlich sind, um mit Java produktiv arbeiten zu können, als Indiz für die Unreife dieser Sprache angesehen werden. Diese Entwicklung führt zur Steigerung der Komplexität [1] von Java-basierten Systemen und damit auch zu einer zunehmenden Steilheit der Lernkurve, was die Praxistauglichkeit eher abnehmen lässt. Auch wenn sich z.B. „Eclipse" heute symbiotisch anbietet, zahlreiche Werkzeuge für die Java-Entwicklung zu integrieren, so ist doch zu beobachten, dass nur noch ganz wenige Entwickler mit dieser Komplexität souverän und zielführend umgehen können.

In der „Programmierung im Großen" fehlt nach wie vor, Anwendungen durch „Zusammenstecken" wiederverwendbarer Komponenten entwickeln zu können. Das liegt u.a. daran, dass es keine normierten Schnittstellenspezifikationen gibt, die diesen Namen verdienen. Meist wird darunter nur die Festlegung der Aufrufsyntax von Diensten verstanden, während deren Semantik nicht präzise definiert werden kann. Dies wird auch an den Standards für Web-Services – wie z.B. WSDL – deutlich, die lediglich die Aufrufsyntax und die Konnektivität betreffen, aber nichts über die Semantik der Dienste auszudrücken erlauben (vgl. auch [12]).

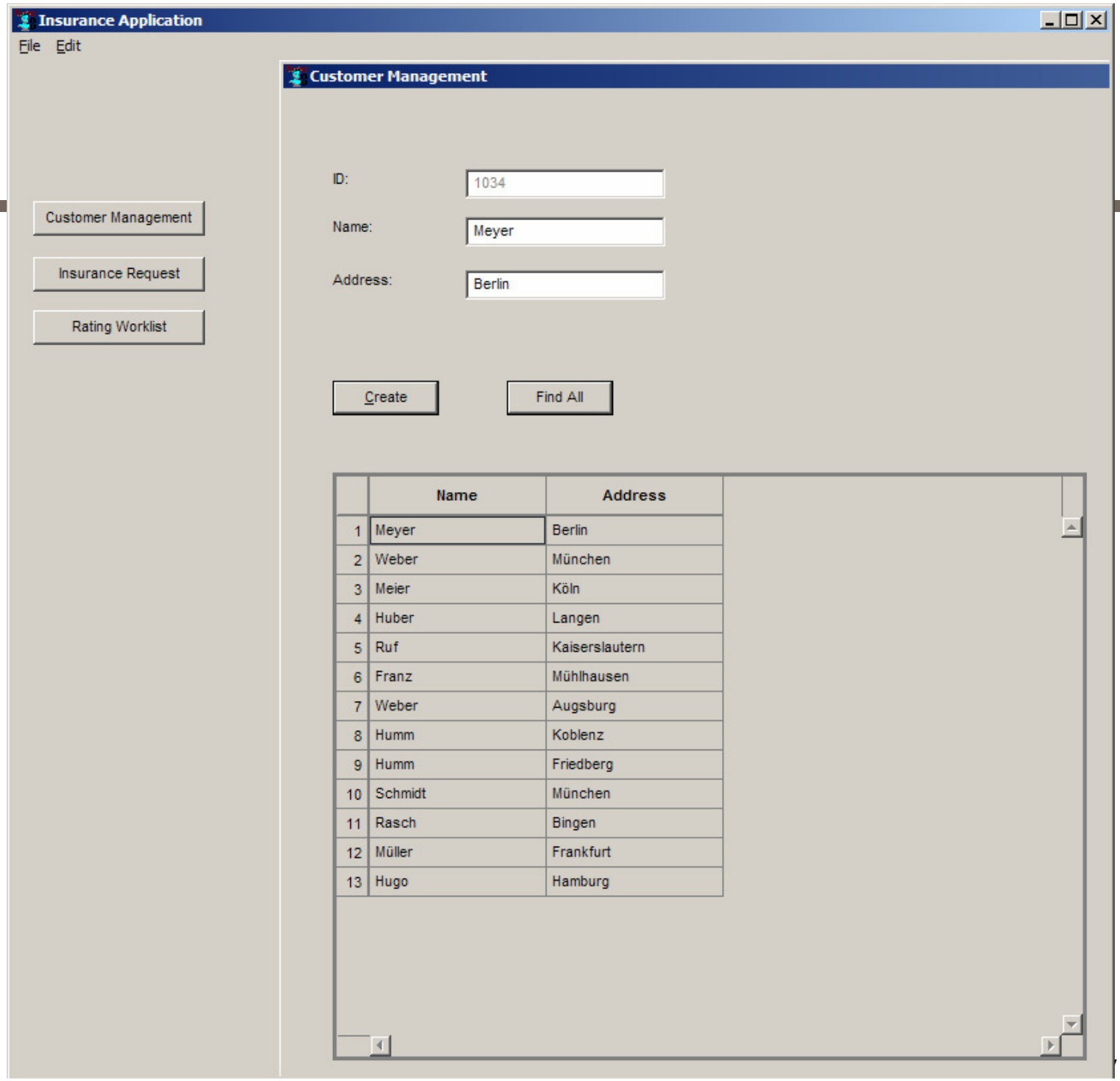
In der „Programmierung im Kleinen" behindern unsere Programmiersprachen oft den Entwickler, z.B. dadurch dass sie ihn zwingen,

DOI 10.1007/s00287-007-0211-3

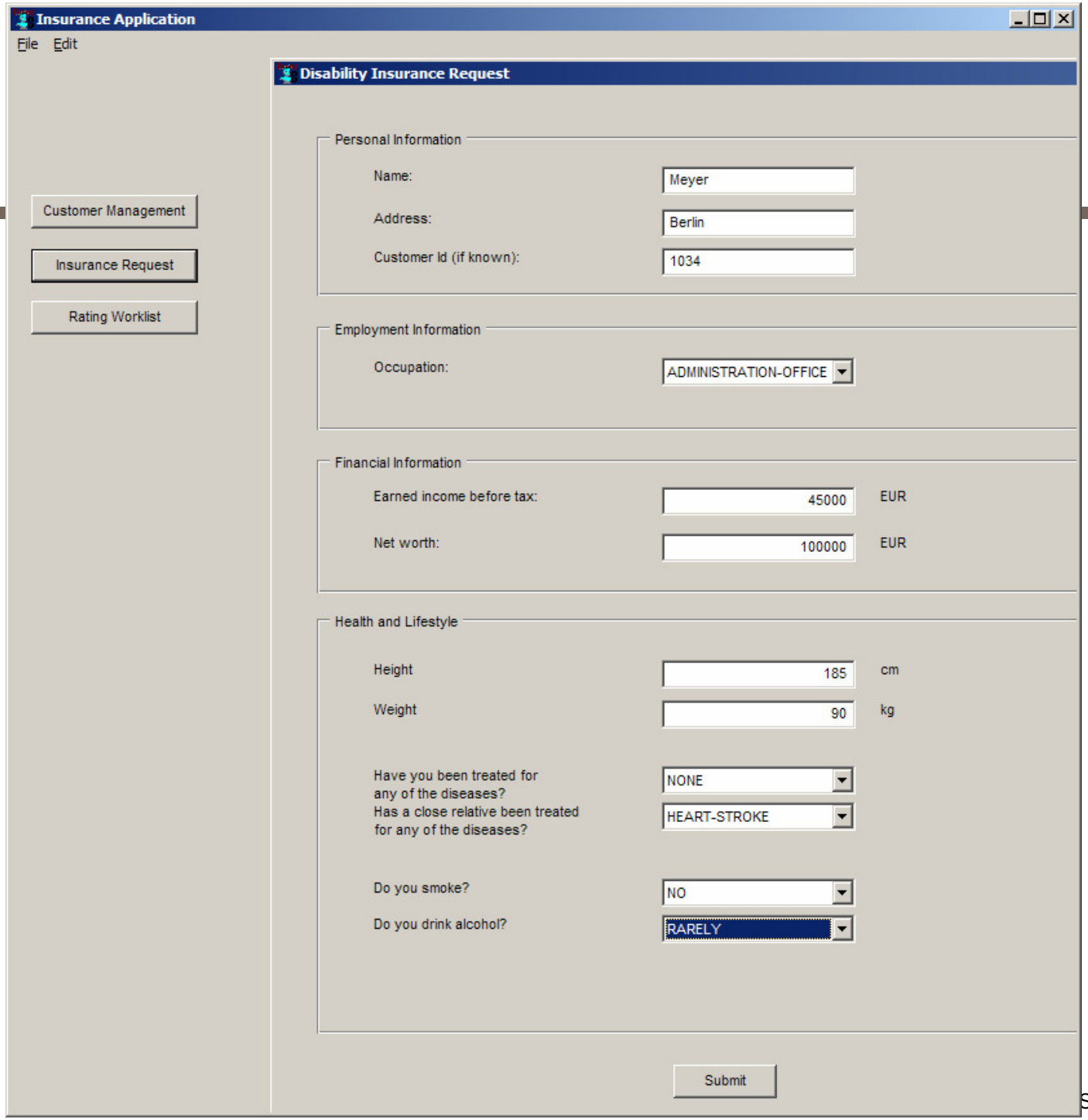
Prof. Dr. Johannes Brauer
FB Informatik, NORDAKADEMIE Hochschule der Wirtschaft,
Köllner Chaussee 11, 25337 Elmshorn
E-Mail: brauer@nordakademie.de

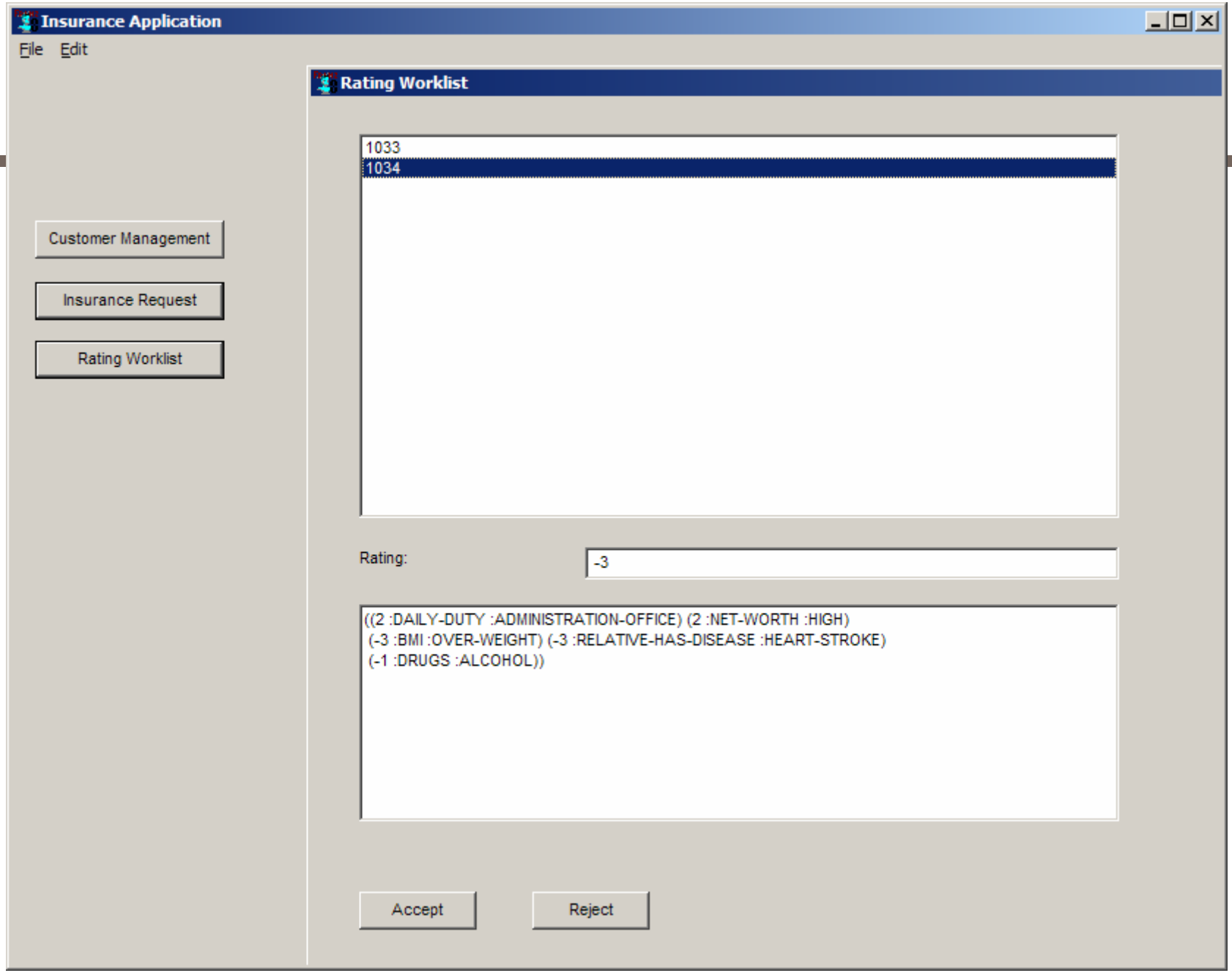
Dr. Christoph Crasemann
IC&C GmbH,
Papenhöhe 14, 25335 Elmshorn
E-Mail: christoph.crasemann@ic-gmbh.de

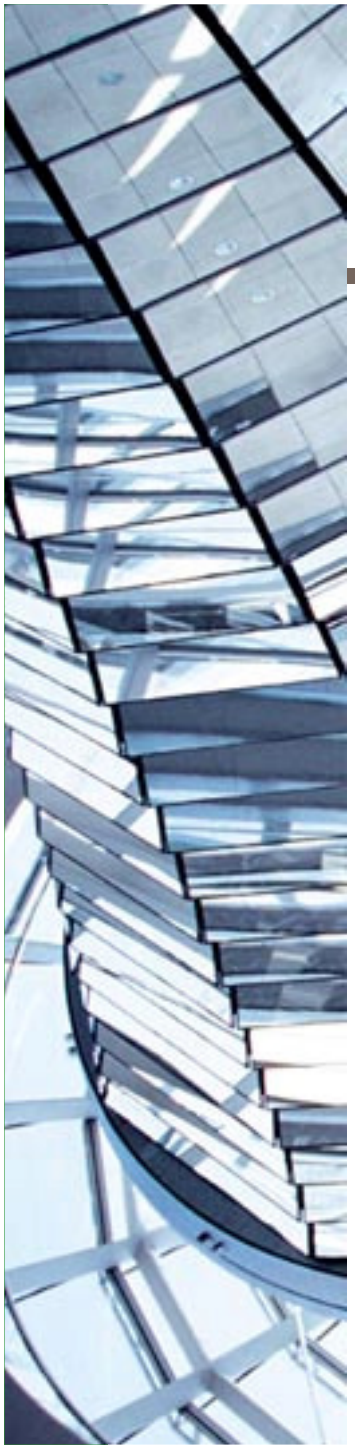
Dr. Hartmut Krasemann
IT-Architekt,
Königsberger Str. 41c, 22869 Schenefeld
E-Mail: krasemann@acm.org



IDE:
Allegro Common Lisp
Express Edition
Franz Inc.


Insurance Application
File Edit
Customer Management
Insurance Request
Rating Worklist
Disability Insurance Request
Personal Information
Name:
Meyer
Address:
Berlin
Customer Id (if known):
1034
Employment Information
Occupation:
ADMINISTRATION-OFFICE
Financial Information
Earned income before tax:
45000
EUR
Net worth:
100000
EUR
Health and Lifestyle
Height
185
cm
Weight
90
kg
Have you been treated for any of the diseases?
NONE
Has a close relative been treated for any of the diseases?
HEART-STROKE
Do you smoke?
NO
Do you drink alcohol?
RARELY
Submit


Insurance Application
File
Edit
Rating Worklist
1033
1034
Customer Management
Insurance Request
Rating Worklist
Rating:
-3
((2 :DAILY-DUTY :ADMINISTRATION-OFFICE) (2 :NET-WORTH :HIGH)
(-3 :BMI :OVER-WEIGHT) (-3 :RELATIVE-HAS-DISEASE :HEART-STROKE)
(-1 :DRUGS :ALCOHOL))
Accept
Reject



Agenda

- Introduction & Example
- A few words on Lisp
- DSLs for business information systems
- The big picture
- Agile development
- Conclusion

Lisp = List Processing

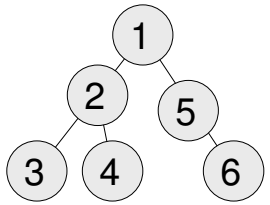
- List: the basic Lisp data structure
  - (1 2 3)
  - (“This is a list” “ of strings”)
  - ((:key1 . value1) (:key2 . value2))
  - (:key1 value1 :key2 value2)
  - (((3) 2 (4)) 1 (() 5 (6)))

Association list

Property list

Binary Tree

:key1 → value1
:key2 → value2



- Lisp Programs are data:
  - (factorial 5)
  - (+ 2 3 4)
  - (if condition (then-operation) (else-operation))
  - (defun identity (x) x)
  - (defclass Customer (name address))

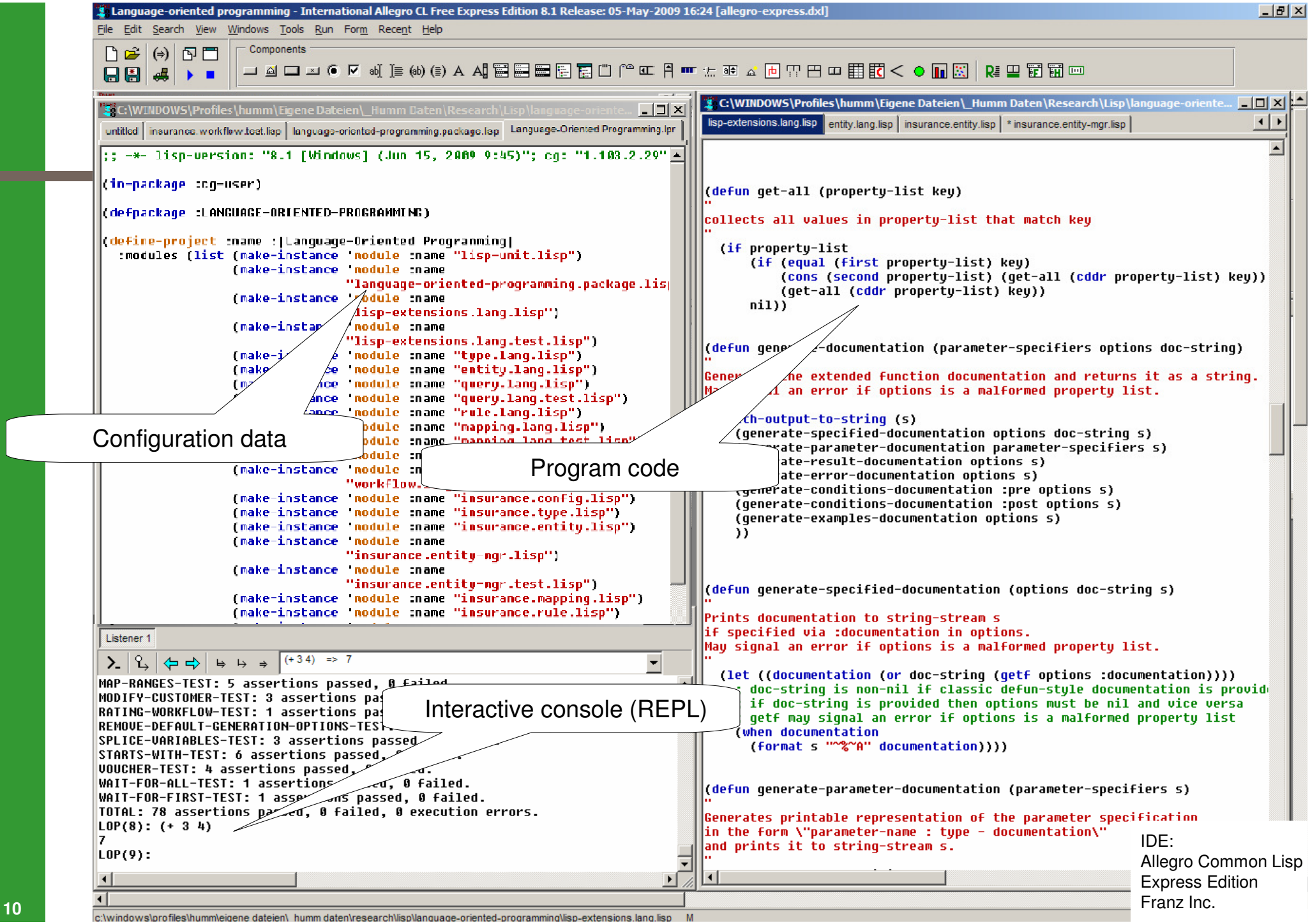
Function call: prefix notation

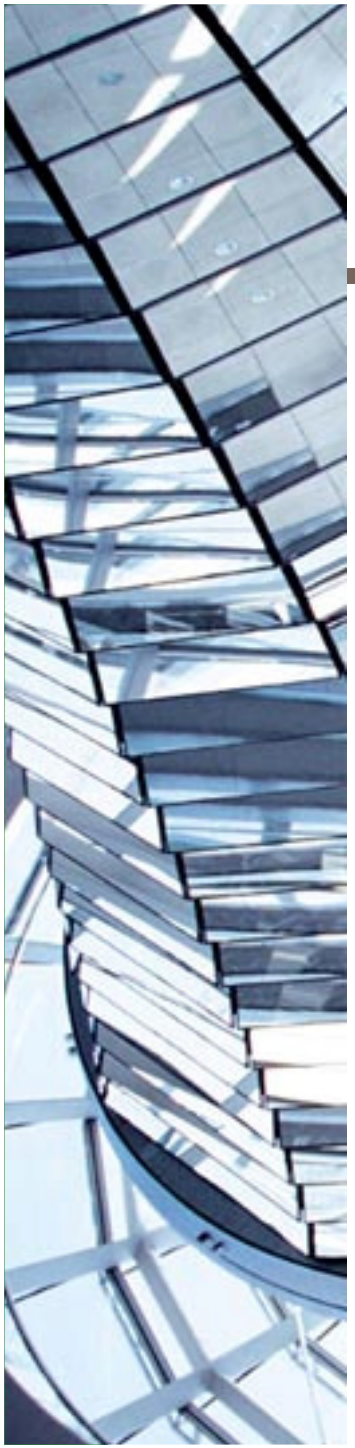
Control structures

Function definition

Class definition

- Lisp Programs are represented by their syntax trees
  → convenient transformation of Lisp programs via macro processor

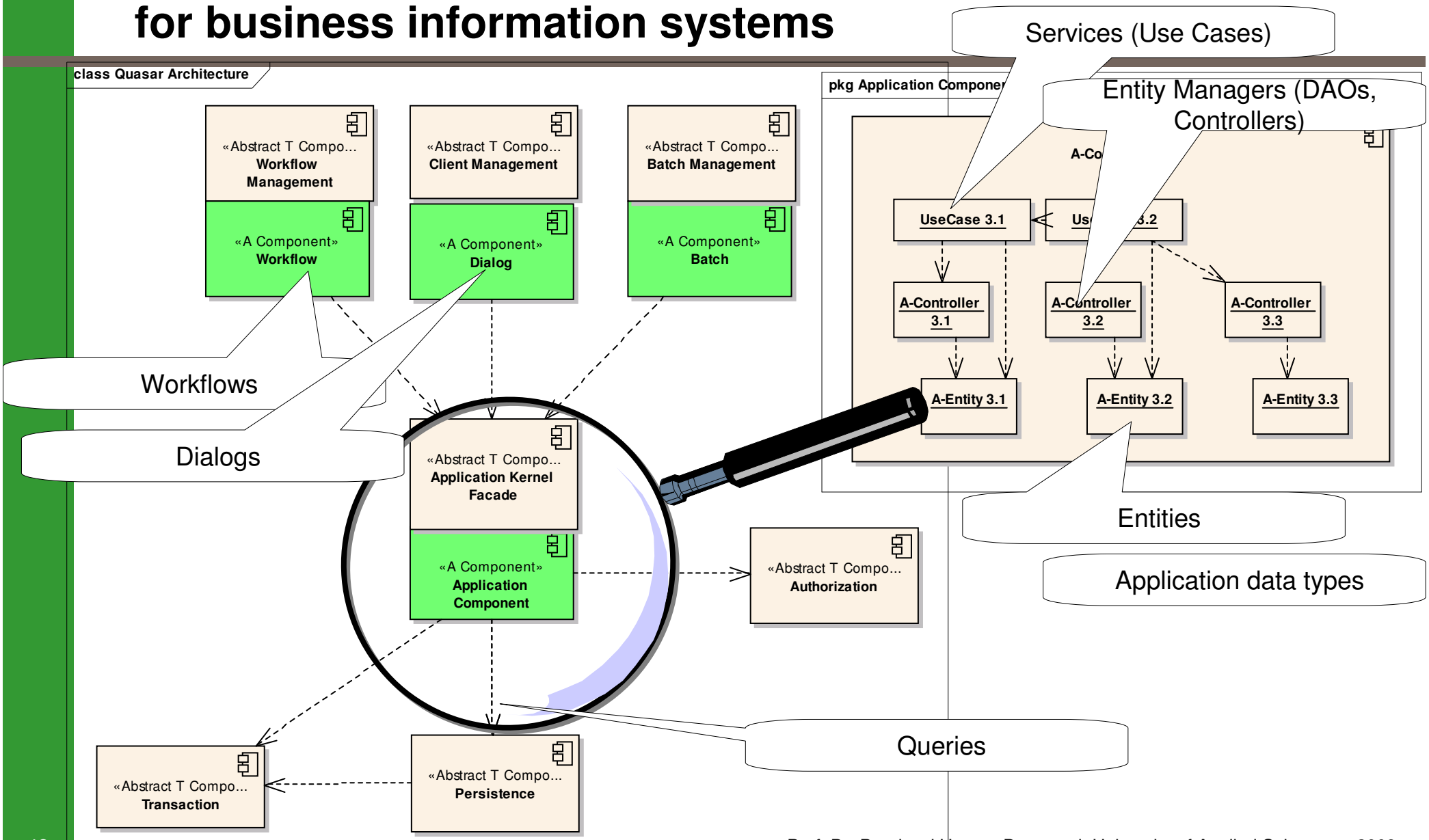

Language-oriented programming - International Allegro CL Free Express Edition 8.1 Release: 05-May-2009 16:24 [allegro-express.dxl]
File Edit Search View Windows Tools Run Form Recent Help
Components
C:\WINDOWS\Profiles\humm\Eigene Dateien_Humm Daten\Research\Lisp\language-oriente...
untitled
insurance.workflow.test.lisp
language-oriented-programming.package.lisp
Language-Oriented Programming.lpr
;; -*- lisp-version: "8.1 [Windows] (Jun 15, 2009 9:45)"; cg: "1.103.2.29"
(in-package :cg-user)
(defpackage :LANGUAGE-ORIENTED-PROGRAMMING)
(define-project :name :|Language-Oriented Programming|
:modules (list (make-instance 'module :name "lisp-unit.lisp")
(make-instance 'module :name
"language-oriented-programming.package.lis
(make-instance 'module :name
"lisp-extensions.lang.lisp")
(make-instance 'module :name
"lisp-extensions.lang.test.lisp")
(make-instance 'module :name "type.lang.lisp")
(make-instance 'module :name "entity.lang.lisp")
(make-instance 'module :name "query.lang.lisp")
(make-instance 'module :name "query.lang.test.lisp")
(make-instance 'module :name "rule.lang.lisp")
(make-instance 'module :name "mapping.lang.lisp")
(make-instance 'module :name
"workflow.
(make-instance 'module :name "insurance.config.lisp")
(make-instance 'module :name "insurance.type.lisp")
(make-instance 'module :name "insurance.entity.lisp")
(make-instance 'module :name
"insurance.entity-mgr.lisp")
(make-instance 'module :name
"insurance.entity-mgr.test.lisp")
(make-instance 'module :name "insurance.mapping.lisp")
(make-instance 'module :name "insurance.rule.lisp")
Listener 1
(+ 3 4) => 7
MAP-RANGES-TEST: 5 assertions passed, 0 failed.
MODIFY-CUSTOMER-TEST: 3 assertions pa
RATING-WORKFLOW-TEST: 1 assertions pa
REMOVE-DEFAULT-GENERATION-OPTIONS-TES
SPLICE-VARIABLES-TEST: 3 assertions passed
STARTS-WITH-TEST: 6 assertions passed,
VOUCHER-TEST: 4 assertions passed,
WAIT-FOR-ALL-TEST: 1 assertions
WAIT-FOR-FIRST-TEST: 1 asse
TOTAL: 78 assertions pa
LOP(8): (+ 3 4)
7
LOP(9):
C:\WINDOWS\Profiles\humm\Eigene Dateien_Humm Daten\Research\Lisp\language-oriente...
lisp-extensions.lang.lisp
entity.lang.lisp
insurance.entity.lisp
* insurance.entity-mgr.lisp
(defun get-all (property-list key)
"
collects all values in property-list that match key
"
(if property-list
(if (equal (first property-list) key)
(cons (second property-list) (get-all (cddr property-list) key))
(get-all (cddr property-list) key))
nil))
(defun gene -documentation (parameter-specifiers options doc-string)
"
Gener e extended function documentation and returns it as a string.
M l an error if options is a malformed property list.
ch-output-to-string (s)
(generate-specified-documentation options doc-string s)
ate-parameter-documentation parameter-specifiers s)
ate-result-documentation options s)
ate-error-documentation options s)
(generate-conditions-documentation :pre options s)
(generate-conditions-documentation :post options s)
(generate-examples-documentation options s)
))
(defun generate-specified-documentation (options doc-string s)
"
Prints documentation to string-stream s
if specified via :documentation in options.
May signal an error if options is a malformed property list.
"
(let ((documentation (or doc-string (getf options :documentation))))
doc-string is non-nil if classic defun-style documentation is provid
if doc-string is provided then options must be nil and vice versa
getf may signal an error if options is a malformed property list
(when documentation
(format s "~%~A" documentation))))
(defun generate-parameter-documentation (parameter-specifiers s)
"
Generates printable representation of the parameter specification
in the form \"parameter-name : type - documentation\"
and prints it to string-stream s.
"
c:\windows\profiles\humm\eigene dateien_humm daten\research\lisp\language-oriented-programming\lisp-extensions.lang.lisp M
Configuration data
Program code
Interactive console (REPL)
IDE:
Allegro Common Lisp
Express Edition
Franz Inc.



Agenda

- Introduction & Example
- A few words on Lisp
- DSLs for business information systems
  - A query language
  - A rules language
  - A workflow language
- The big picture
- Agile development
- Conclusion

Excursus: The Quasar reference architecture for business information systems



Example:
A simple DSL for application data types

```
(define-data-type ISBN string ISBN-p)


(define-enum-type policy-status
    (:initial :sent :active :passive :binding-period-passed :rejected :closed)
  "Possible status of insurance policies"
  )
```

```
(defmacro define-data-type (name base-type predicate &optional documentation)
  "
Defines an application datatype based on base-type with predicate as restriction

Example:
(define-data-type positive-number integer positive-p)
"

  `(deftype ,name ()
     ,documentation
     '(and ,base-type (satisfies ,predicate))
     )
  )



(defmacro define-enum-type (name values &optional documentation)
  `(deftype ,name ()
     ,documentation
     '(member ,@values)
     )
  )
```

```
LOP(20): (TYPEP "978-3-89864-506-5" 'ISBN)
T
LOP(21): (TYPEP :sent 'policy-status)
T
LOP(22): (TYPEP :send 'policy-status)
NIL
```

Example:
A simple DSL for entity types

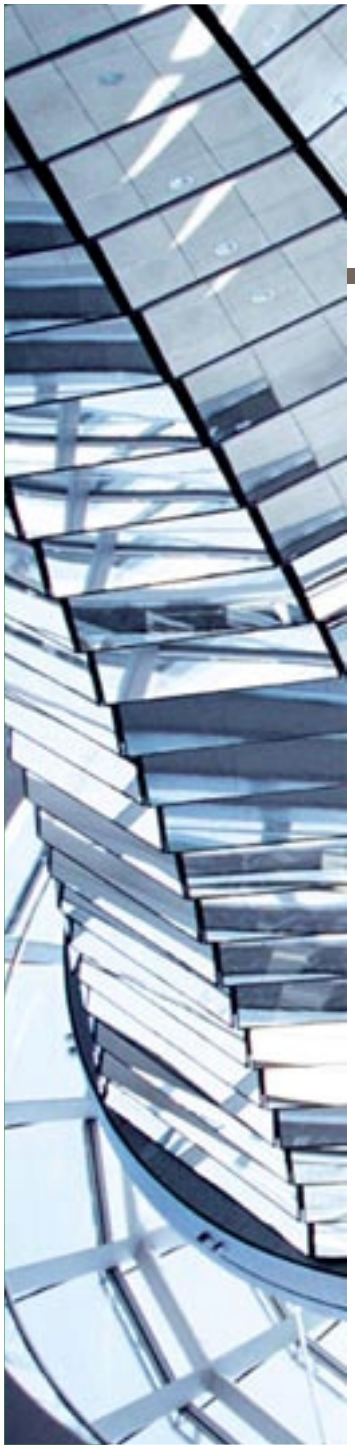
```
(define-entity Customer ()
  ((name
    :type string
    )
   (address
    :type string
    :documentation "Street, number, ZIP code, city and country"
    )
   )
  (:documentation "Entity representing a registered customer or potential customer")
  )
```

```
(defmacro define-entity (name  superclasses slots &rest options)
  "
Defines a persistent class with initargs, getters and setters for all slots
Also defines a default print-object method showing oid and all slot values
"
  `(progn
     (define-class ;; generate class definition
         ,name
         ,(cons 'Entity superclasses)
       ,slots
       ,(append
         '(:metaclass persistent-class :default-initargs t :default-getters t :default-setters t :all-indexed t)
         (first options))
       )
     (defprinter ,name ,@(slotnames slots)) ;; generate default print-object method
     (find-class (quote ,name))  ; return the class - not the print-object method - for better readability
     )
  )
```

```
NIL
LOP(23): (make-instance 'Customer :name "Huber" :address "Hamburg")
#<CUSTOMER [1035]* "Huber" "Hamburg">
```

Criteria for a good DSL

- A DSL consists of:
  - Syntax – needs an editor to program in
  - Semantics – needs a compiler and a runtime environment to execute
- Good Syntax is:
  - Concise
  - Easy and intuitively to understand
- Semantics should be:
  - Expressive / powerful
  - Efficient implementation
- Example DSLs:
  - Query language
  - Rules language
  - Workflow language



Agenda

- Introduction & Example
- A few words on Lisp
- DSLs for business information systems
  - A query language
  - A rules language
  - A workflow language
- The big picture
- Agile development
- Conclusion

A general object database query language

Local variable

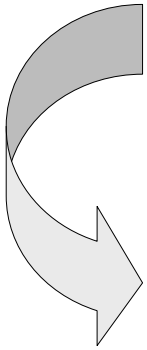
Persistent class

Auxiliary variable – not needed in result set

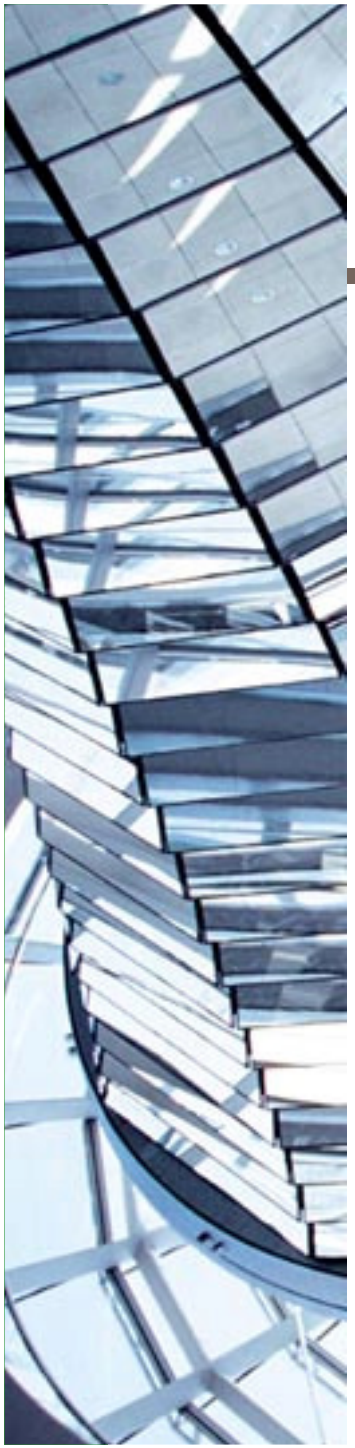
```
(select-entities ((cust 'Customer) (pol 'Policy :aux) (prod 'Product))
                 (equal :initial (get-status pol))
                 (equal :life-insurance (get-prod-type prod))
                 (equal cust (get-customer pol))
                 )
```

Conjunctive clauses

Any form of Lisp expression allowed. Here: object navigation



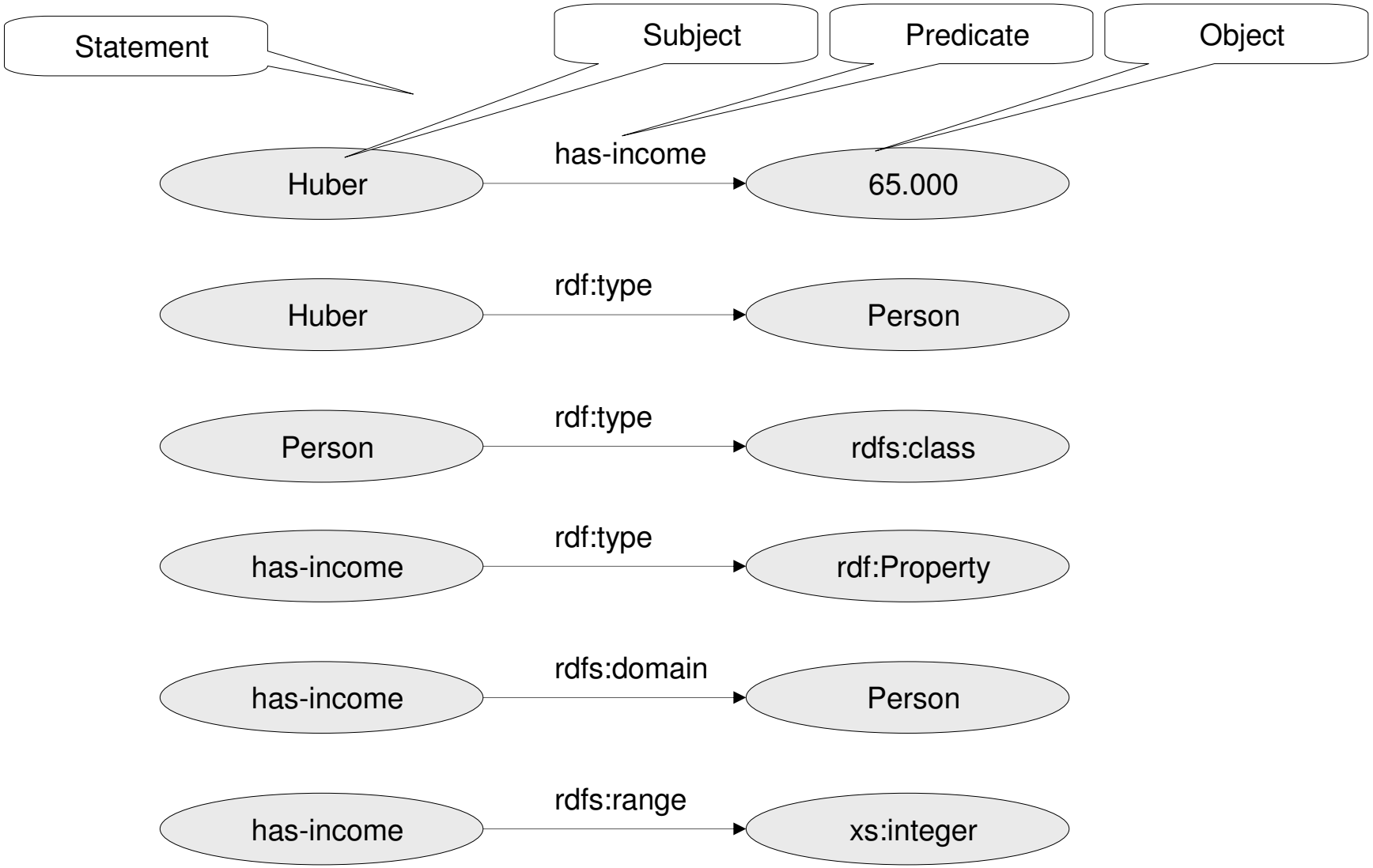
```
LOP(30): *
((#<CUSTOMER [1027]* "Ruf" "Kaiserslautern"> #<PRODUCT [1028]* :LIFE-INSURANCE>)
 (#<CUSTOMER [1027]* "Ruf" "Kaiserslautern"> #<PRODUCT [1017]* :LIFE-INSURANCE>)
 (#<CUSTOMER [1023]* "Franz" "Mühlhausen"> #<PRODUCT [1028]* :LIFE-INSURANCE>)
 (#<CUSTOMER [1023]* "Franz" "Mühlhausen"> #<PRODUCT [1017]* :LIFE-INSURANCE>)
 (#<CUSTOMER [1022]* "Weber" "Augsburg"> #<PRODUCT [1028]* :LIFE-INSURANCE>)
 (#<CUSTOMER [1022]* "Weber" "Augsburg"> #<PRODUCT [1017]* :LIFE-INSURANCE>)
 (#<CUSTOMER [1015]* "Rasch" "Bingen"> #<PRODUCT [1028]* :LIFE-INSURANCE>)
 (#<CUSTOMER [1015]* "Rasch" "Bingen"> #<PRODUCT [1017]* :LIFE-INSURANCE>))
LOP(31):
```



Agenda

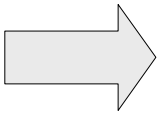
- Introduction & Example
- A few words on Lisp
- DSLs for business information systems
  - A query language
  - A rules language
  - A workflow language
- The big picture
- Agile development
- Conclusion

Excursus on semantic web technology: RDF(S) – Resource Description Framework (Schema)



XML syntax for RDF(S): no appropriate langauge for representating knowledge

```
<rdf:Description rdf:about="CIT2112">
 <uni:isTaughtBy>
  <rdf:List>
   <rdf:first>
    <rdf:Description rdf:about="949111"/>
   </rdf:first>
   <rdf:rest>
    <rdf:List>
     <rdf:first>
      <rdf:Description rdf:about="949352"/>
     </rdf:first>
     <rdf:rest>
      <rdf:List>
       <rdf:first>
        <rdf:Description rdf:about="949318"/>
       </rdf:first>
       <rdf:rest>
        <rdf:Description rdf:about="&rdf;nil"/>
       </rdf:rest>
      </rdf:List>
     </rdf:rest>
    </rdf:List>
   </rdf:rest>
  </rdf:List>
 </uni:isTaughtBy>
</rdf:Description>
```

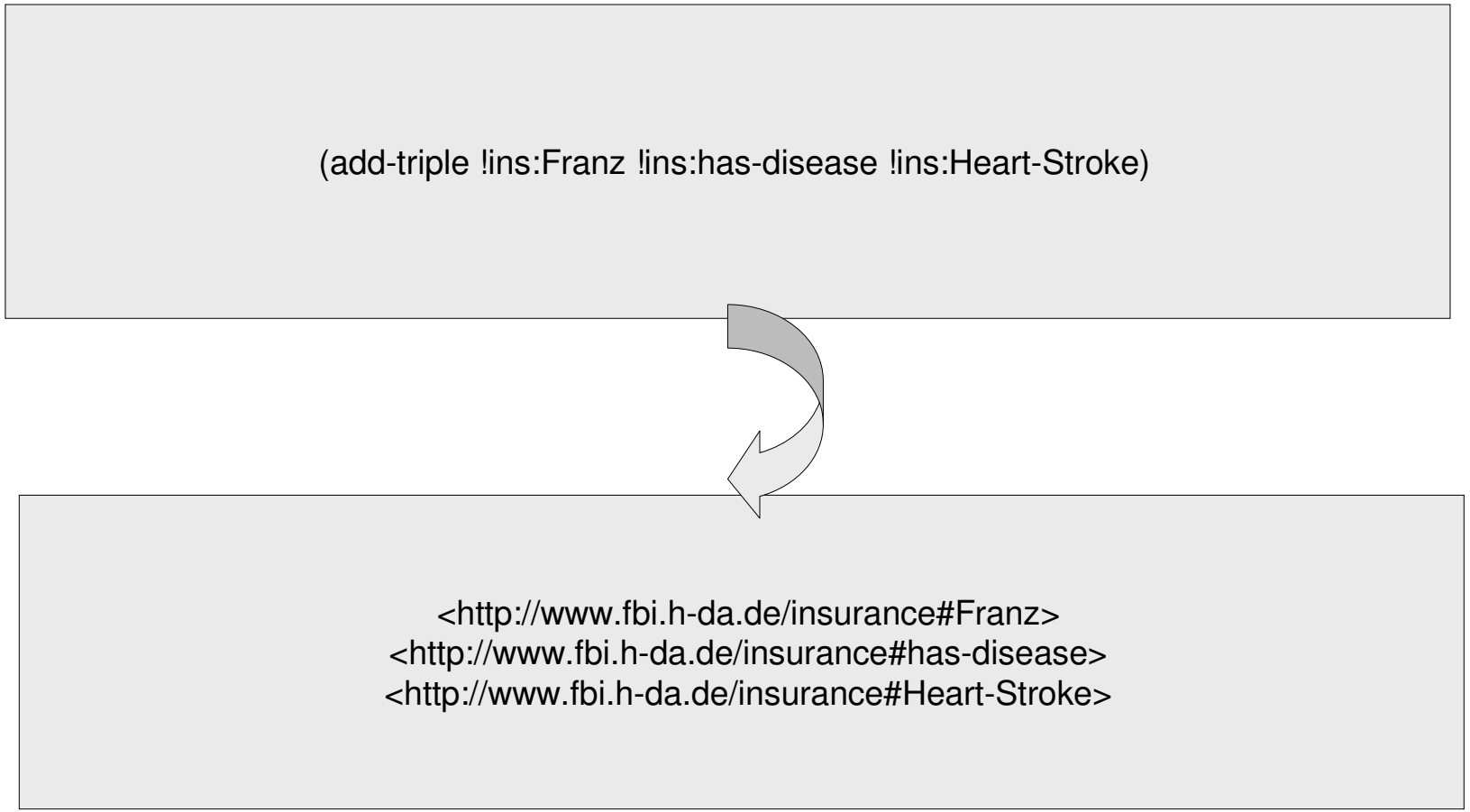


Meaning:

Course CIT2112 is taught by teachers 949111, 9493252, and 949318

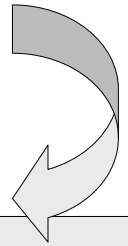
Source: Antoniou, van Harmelen: „A Semantic Web Primer“

Simplify URI syntax



Provide means for declaring classes and instances

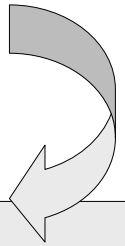
```
(add-class !ins:Person :comment "Natural Person" :see-also !ins:Profession)

(add-instance !ins:Huber !ins:Person)
```



```
(ADD-TRIPLE !ins:Person !rdf:type !rdfs:class)
(ADD-TRIPLE !ins:Person !rdfs:comment (LITERAL "Natural Person")) NIL
(ADD-TRIPLE !ins:Person !rdfs:seeAlso !ins:Profession))

(ADD-TRIPLE !ins:Huber !rdf:type !ins:Person)
```

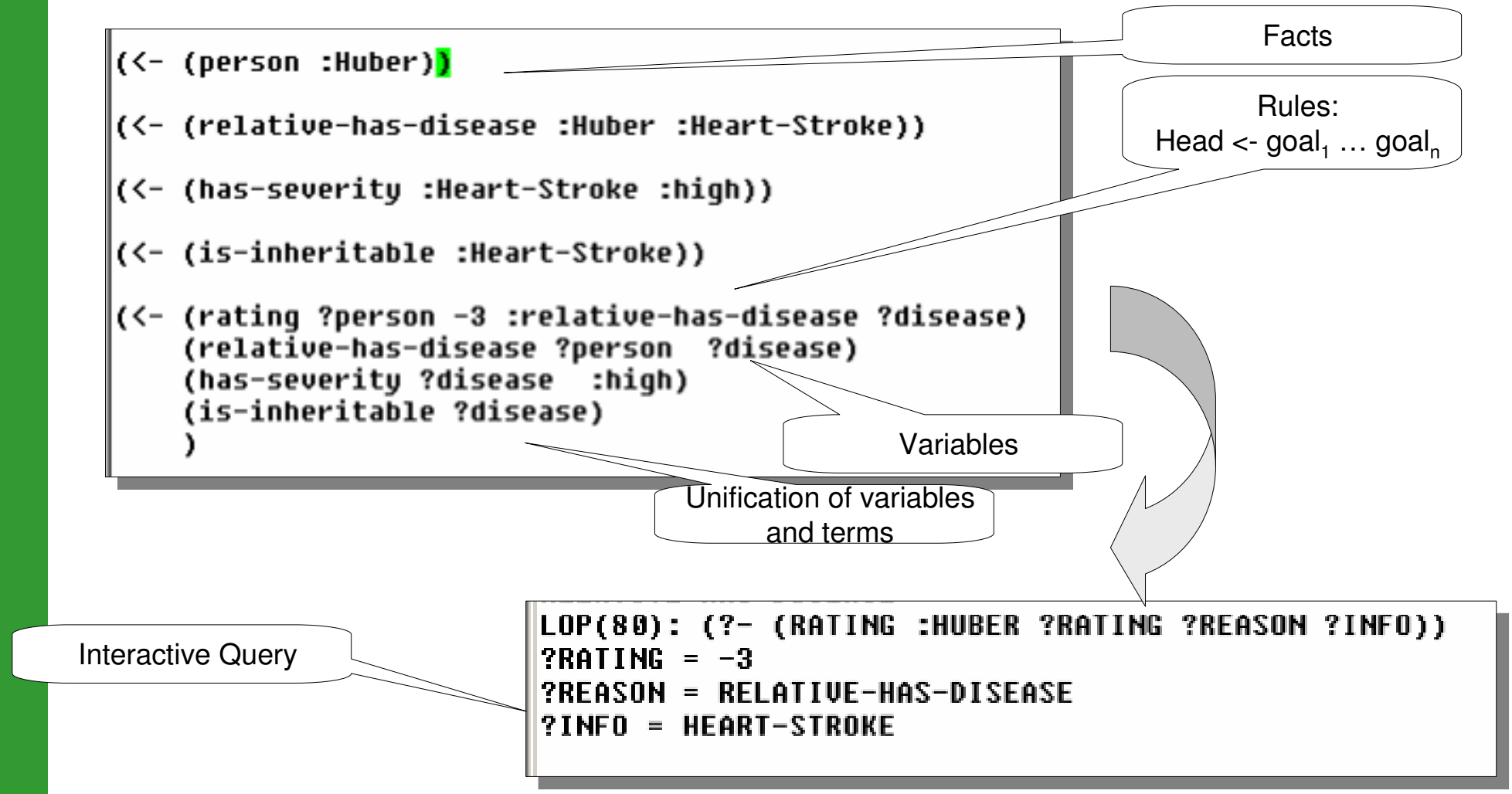
Provide means for declaring properties

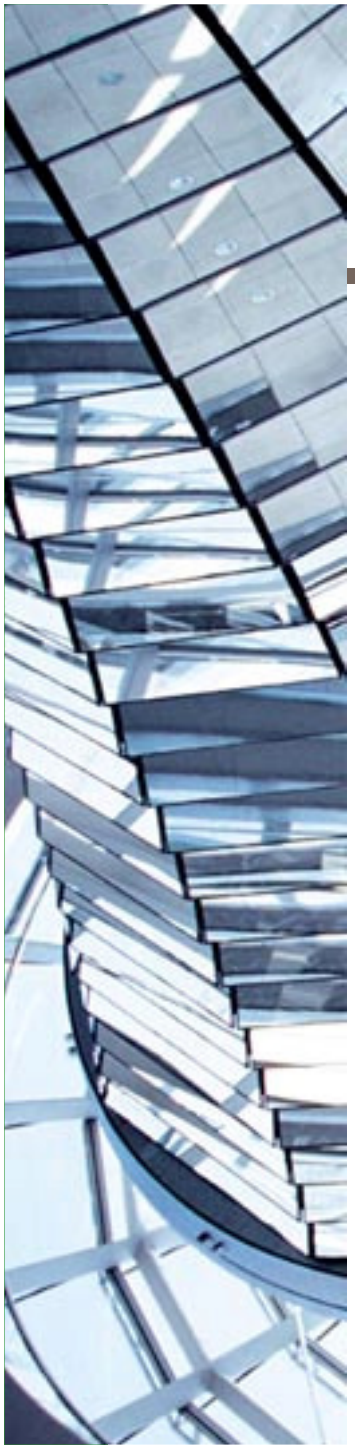
```
(add-property !ins:Person !ins:has-desease !ins:Disease)

(add-triple !ins:Franz !ins:has-disease !ins:Heart-Stroke)
```



```
(ADD-TRIPLE !ins:has-desease !rdf:type !rdf:Property)
 (ADD-TRIPLE !ins:has-desease !rdfs:domain !ins:Person)
 (ADD-TRIPLE !ins:has-desease !rdfs:range !ins:Disease)

(add-triple !ins:Franz !ins:has-disease !ins:Heart-Stroke)
```

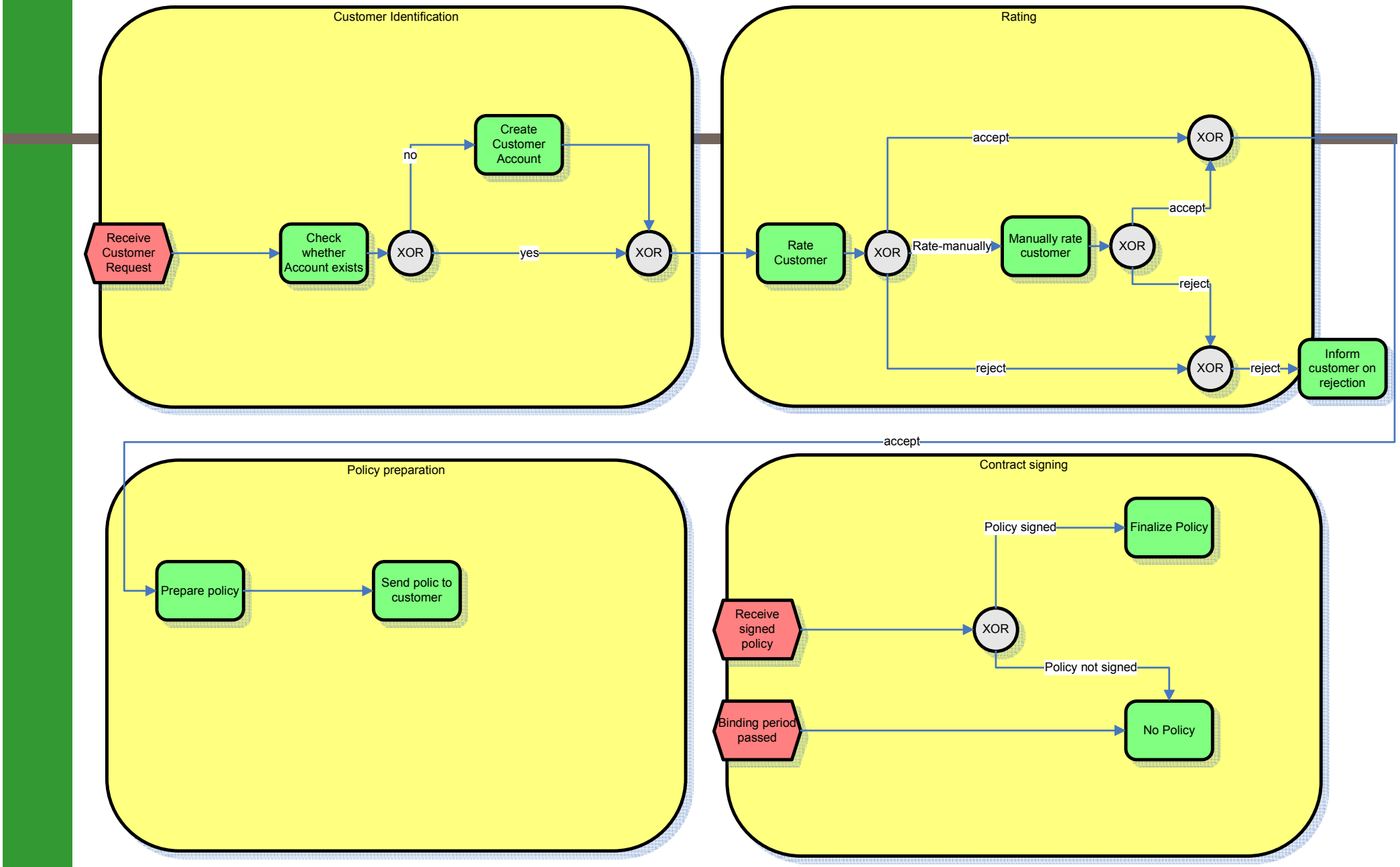
A rules language oriented on Prolog





Agenda

- Introduction & Example
- A few words on Lisp
- DSLs for business information systems
  - A query language
  - A rules language
  - A workflow language
- The big picture
- Agile development
- Conclusion


Customer Identification
Receive Customer Request
Check whether Account exists
XOR
no
Create Customer Account
yes
XOR
Rating
Rate Customer
XOR
accept
XOR
Rate-manually
Manually rate customer
XOR
accept
reject
XOR
reject
Inform customer on rejection
accept
Policy preparation
Prepare policy
Send polic to customer
Contract signing
Policy signed
Finalize Policy
Receive signed policy
XOR
Policy not signed
Binding period passed
No Policy

```
(define-workflow Rating-Workflow (customer-id request-form)
  ()
  (invoke-synch #'assert-insurance-facts customer-id request-form)
  (multiple-value-bind (rating num-rating info) (rate-insurance-request customer-id request-form)
    (if (equal :manual-check rating)
        (let ((activity (make-instance 'Manual-Insurance-Rating-Activity
                          :customer-id customer-id
                          :request-form request-form
                          :rating rating
                          :num-rating num-rating
                          :rating-info info
                          )
                        )
              )
          (setf rating (get-value (invoke-activity activity)))
          )
      )
    (cond
     ((equal :accept rating)
      (invoke-workflow 'Policy-Preparation-Workflow customer-id request-form)
      )
     ((equal :reject rating)
      (invoke-asynch #'inform-customer-on-reject customer-id)
      )
     )
    rating
    )
  )
```

Named workflow

Synchronous invocation of service

Manual activities can be accessed via worklist dialogs

Voucher mechanism: waits until value has been set

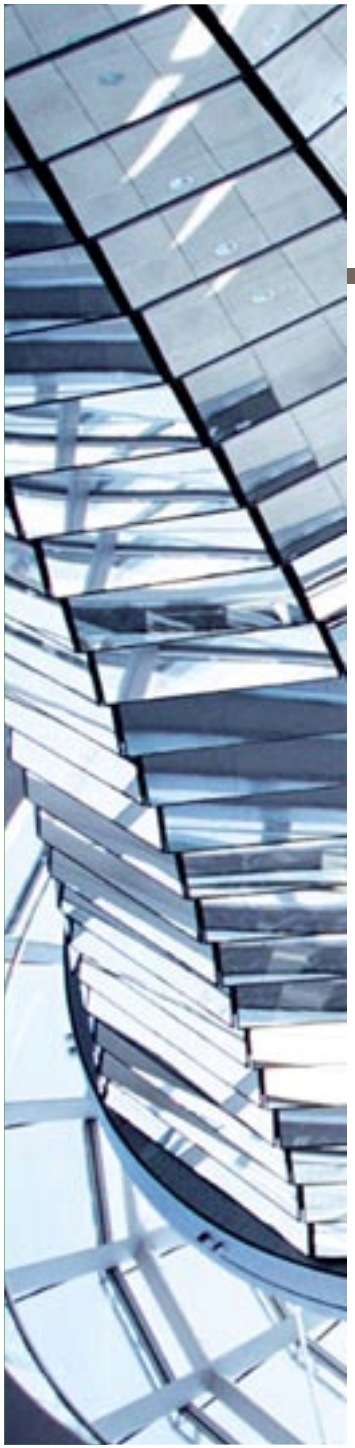
Worflows are services that can be invoked

Asynchronous invocation of service

```
(define-workflow Contract-Signing-Workflow (customer-id policy)
  ()
  (let* ((event-voucher (event-voucher
                         'Signed-Policy-Received-Event
                         (lambda (event) (equal customer-id (get-customer-id event)))
                         )
                        )
         (timer-voucher (timer-voucher :days 60))  ; binding period
         (result (wait-for-first event-voucher timer-voucher))
         )
    (if (equal :timer-finished result)
        (set-status :binding-period-passed policy)
      (set-status :active policy)
      )
    policy
    )
  )
```

Waiting for external events

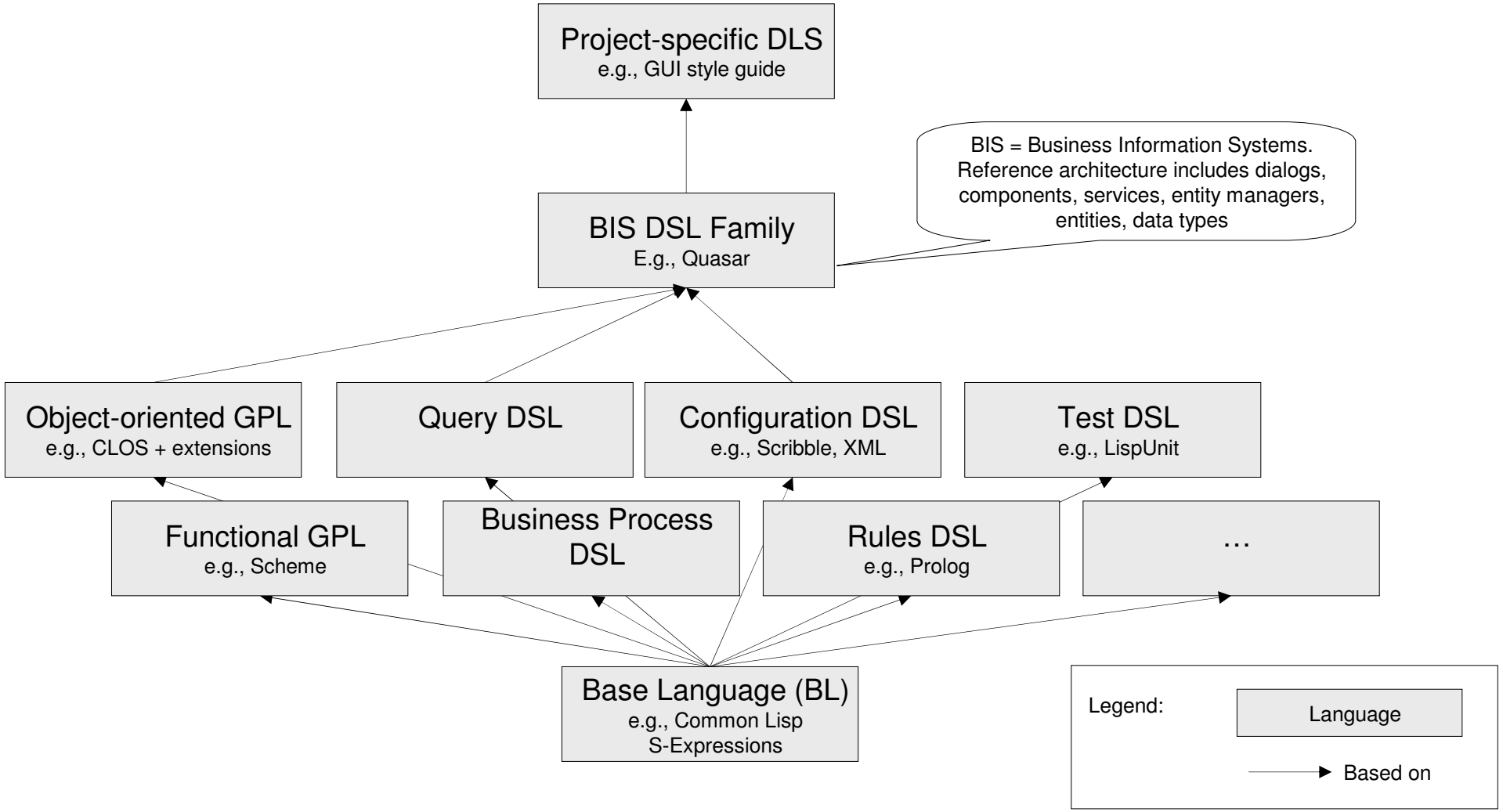
Waiting for timing conditions



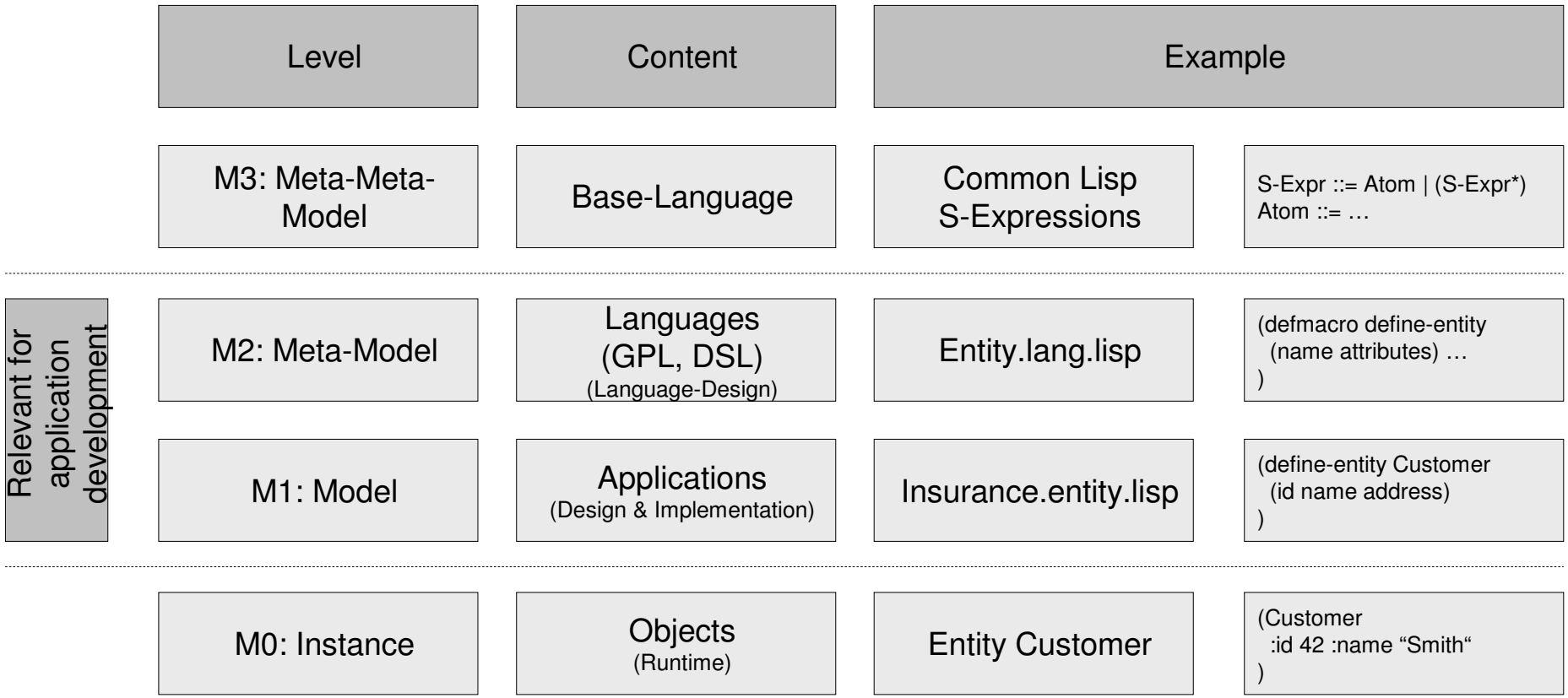
Agenda

- Introduction & Example
- A few words on Lisp
- DSLs for business information systems
- The big picture
- Agile development
- Conclusion

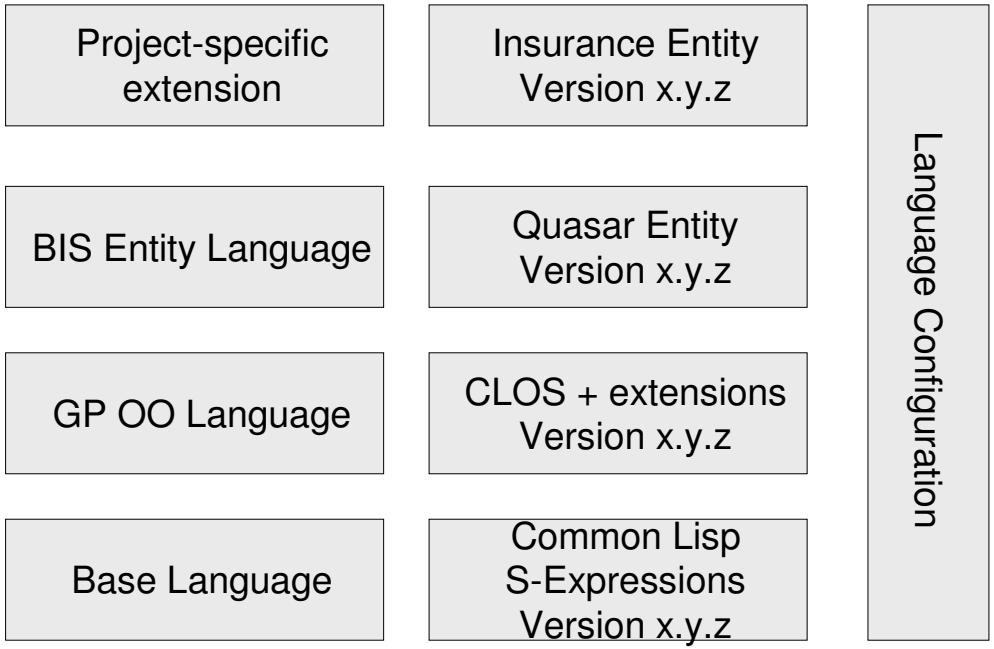
Domain-specific languages (DSL) or different general-purpose languages (GPL) may be derived from a base language (BL)

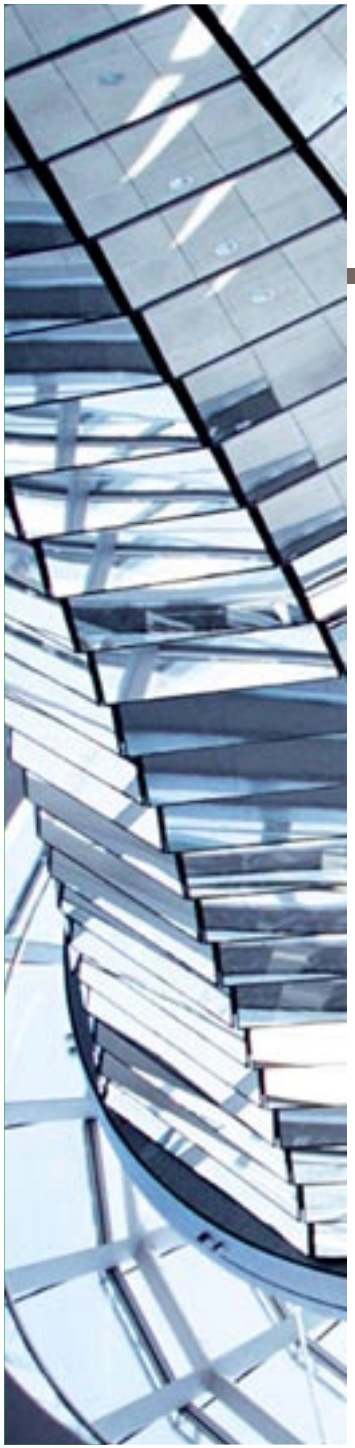


Language design and application design are relevant for application development



The language stack describes the configuration of language versions (GPL, DSL) based on each other

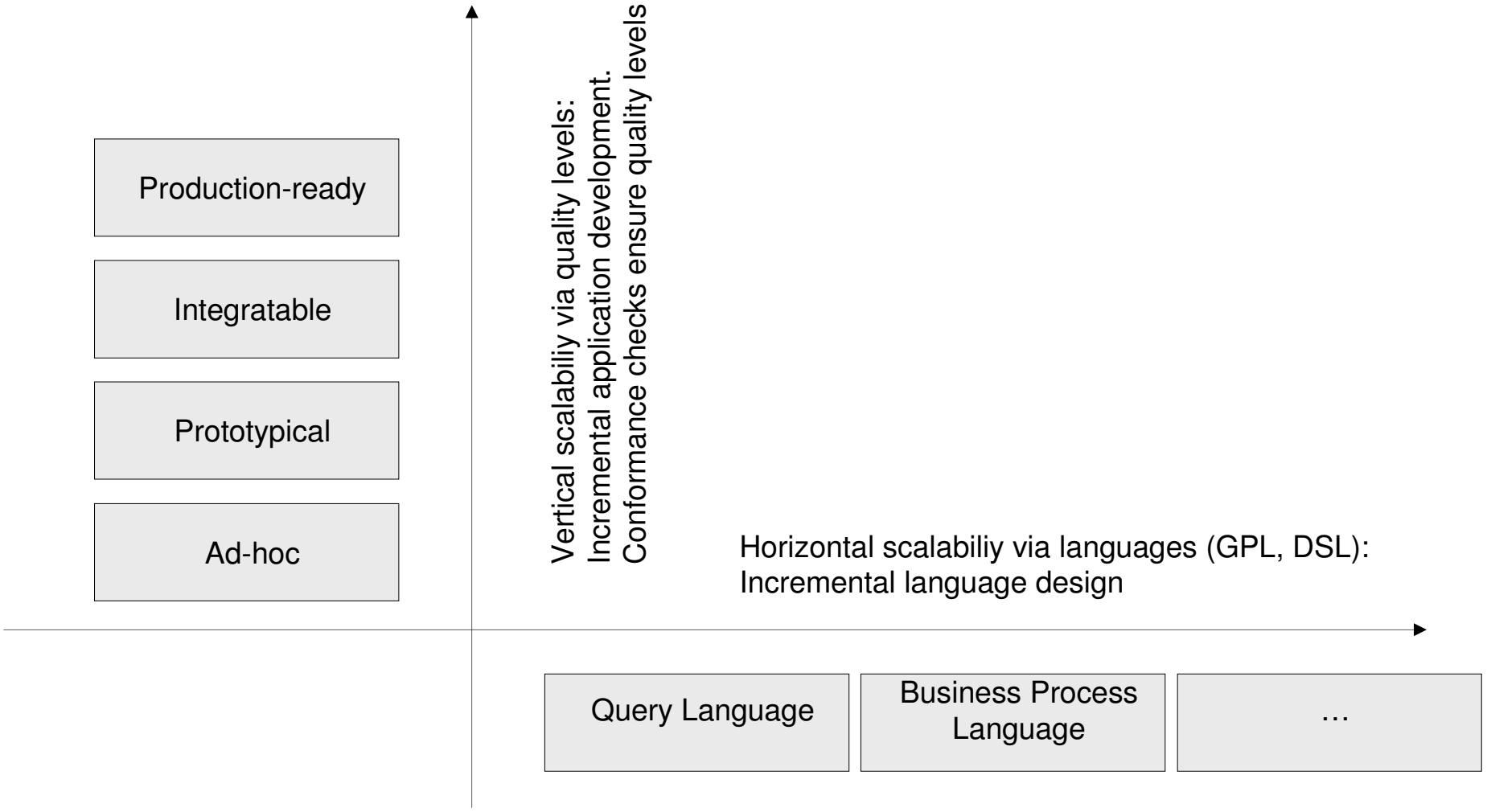




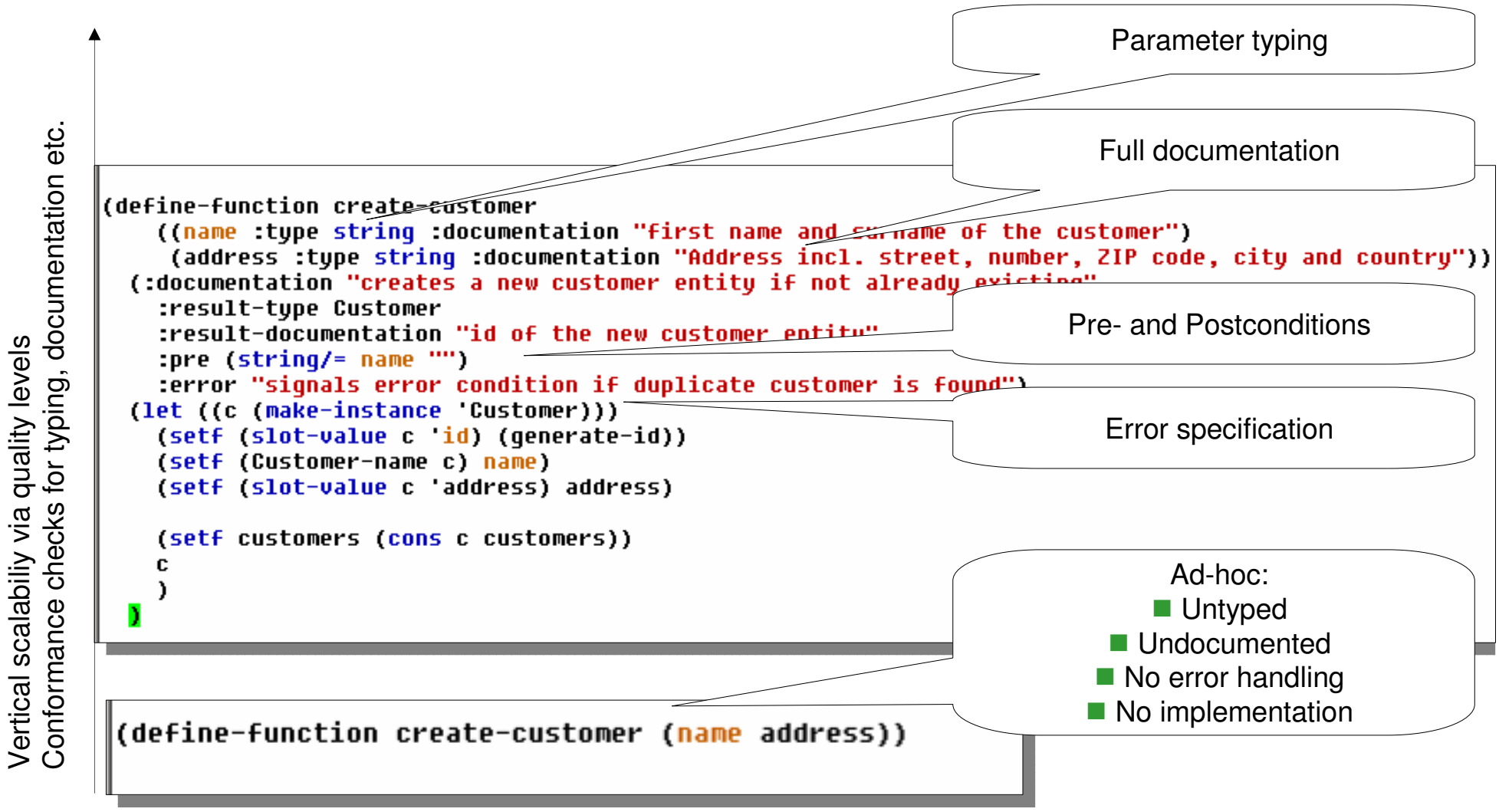
Agenda

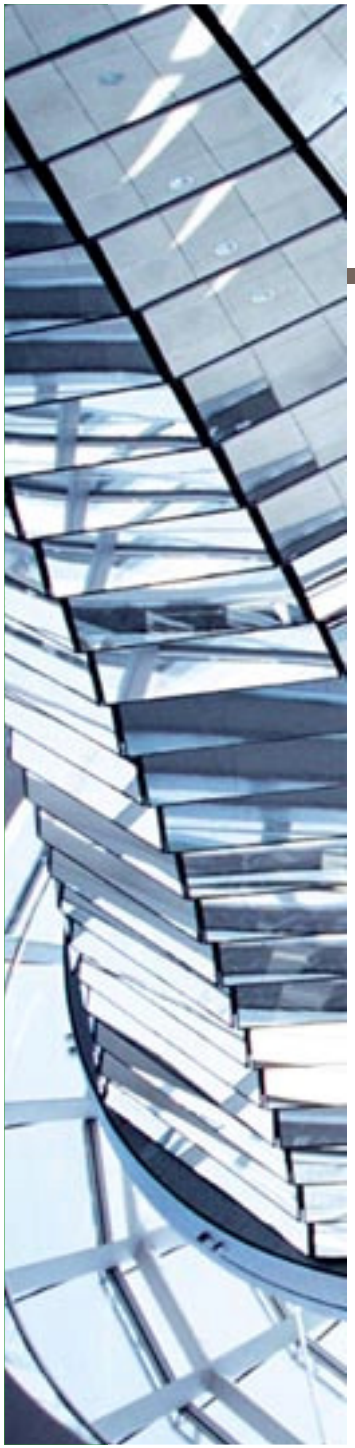
- Introduction & Example
- A few words on Lisp
- DSLs for business information systems
- The big picture
- Agile development
- Conclusion

Incremental language design allows scaling the language to the problem domain.
Incremental application development allows stepwise increase of quality



Example for vertical scalability: Successively adding typing, documentation, error handling, pre- / post-conditions etc.





Agenda

- Introduction & Example
- A few words on Lisp
- DSLs for business information systems
- The big picture
- Agile development
- Conclusion

Conclusion and discussion

- Language oriented programming (LOP) allows specifiying a solution to a problem on the appropriate level of abstraction
- Lisp is well suited for LOP
- Programming of a high level of abstraction is useful

BUT
- Defining languages is hard
- Language versioning is a problem

BUT
- Lisp is not widely adopted in industry

BUT
- Every complex abstration is leaky (Spolsky)

What is your opinion?