



Dylan (Dynamic Language)

A multi-paradigm language

Oliver Juwig

Aachen, 12. Februar 2003

Agenda

- **History of Dylan**
- Concepts of the language
- Multidimensional polymorphism - a silver bullet
- Functional programming with Dylan
- Dylan in a *real* project

Dylan has been inspired by functional programming languages

- The language was inspired by Scheme and CommonLisp
 - Dylan is a superset of Scheme
 - Extensions:
 - CommonLisp Object System
 - CommonLisp Condition System
 - A production-rule based macro processor
- But: Language syntax is more like Pascal or Modula
- Dylan is a best of breed language; it combines object-oriented, functional and algorithmic programming paradigms

Dylan has been developed by several partners

- Three major partners has been involved in the development of the language
 - Carnegie-Mellon University (Project Gwydion)
 - Digital Corporation
 - Apple Computers, Inc.
 - Harlequin
- Dylan is a general purpose language, but it was targeted for small devices in the beginning (similar to Java)
 - Apple Newton
- The first language draft appeared 1993

Current resources for Dylan

The image displays the Functional Developer IDE interface, which is used for developing Dylan programs. It consists of several windows:

- Functional Developer (Main Window):** Shows the menu bar (File, Options, Tools, Window, Help) and a toolbar with icons for file operations and a search bar containing the text "test".
- Project ueb7 - Functional Developer (Project Browser):** Displays a tree view of the project structure. The "Sources" tab is active, showing a hierarchy:
 - library.dylan
 - library ueb7
 - module.dylan
 - module ueb7
 - ueb7.dylan** (selected)
 - method times (<number>, <number>)
 - method fibo (<integer>)
 - method fibo1 (<integer>)
 - method faculty (<integer>)
 - method gcd1 (<integer>, <integer>)
 - method rev (<sequence>)
 - method main ()
 - begin
 The status bar at the bottom indicates "3 sources".
- ueb7.dylan - Editing ueb7 - Functional Developer (Code Editor):** Shows the source code for the selected file. The code defines a module named "ueb7" with a synopsis "uebung 7", author "Johannes Siedersleben", and copyright information. It contains two methods:


```
Module:      ueb7
Synopsis:   uebung 7
Author:     Johannes Siedersleben
Copyright:  (C) 2001, sd&M Research.  All rights reserved.
Version:    $version$

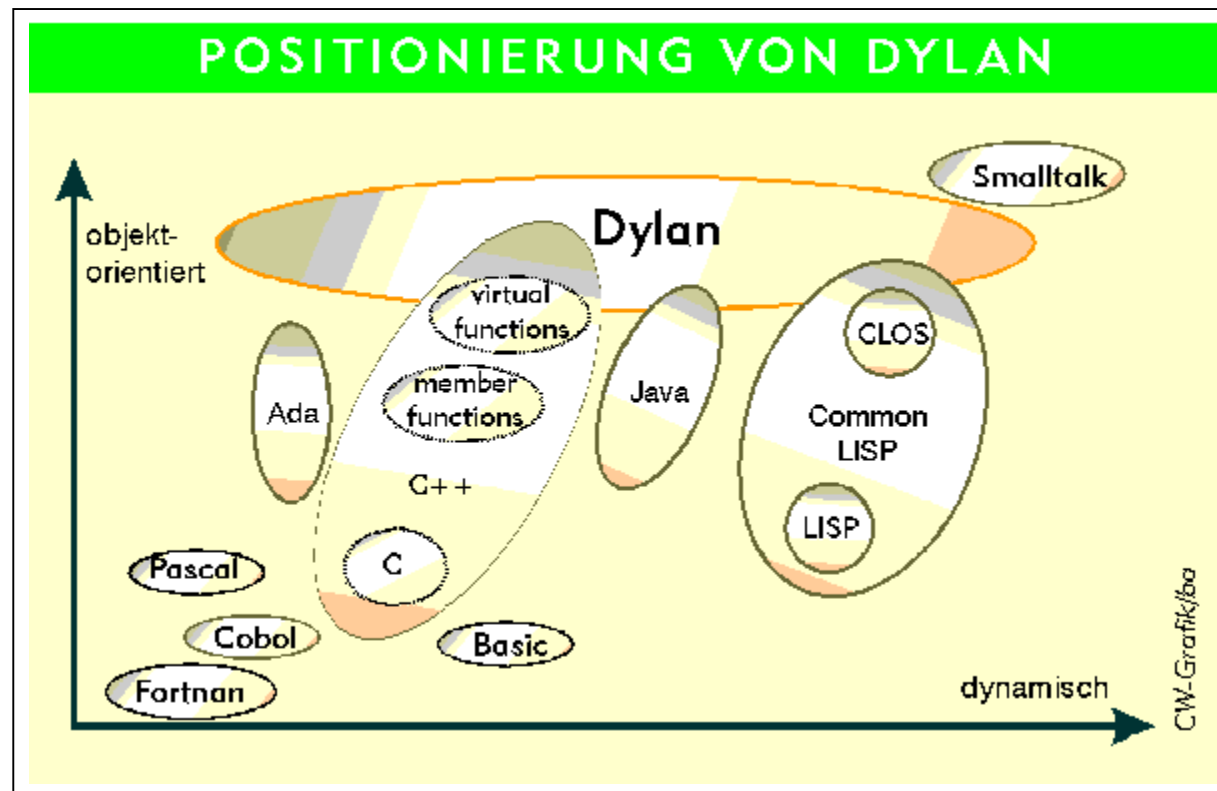
- define method times(x :: <number>, y :: <number>)
-           => (product :: <number>)
-   x * y;
- end method times;

- define method fibo(n :: <integer>) => (f :: <integer>)
-   if (n == 0)
-     0
-   elseif (n == 1)
-     1
-   else
-     let(a, b) = values(0, 1);
-     for (i from 0 below n - 1)
-       let(u, v) = values(b, a + b);
-       a := u;
-       b := v;
-     end for;
-     b;
-   end if;
- end method fibo;
```

Agenda

- History of Dylan
- **Concepts of the language**
- Multidimensional polymorphism - a silver bullet
- Functional programming with Dylan
- Dylan in a *real* project

Dylan combines extrem object-oriented concepts with static and dynamic programming style



Every little thing in Dylan is an object

- All objects are first class, even:
 - Numbers and characters
 - Classes
 - Functions
- Every object can be used as a function argument
- All objects are typed and type-safe
- Variables can be strongly typed
- All objects devive from class `<object>`

A strong subset of the CommonLisp Object System is used in Dylan

- Strong support of multiple inheritance
 - Slots are functions (so called slot methods, they can be specialized)
 - Classes define no methods in addition to the slot methods
 - Scope of names is not defined by classes
- Dylan has an explicit name space concept based upon libraries and

```
define open abstract class <presentation> (<object>)
  keyword cached:, type: <boolean>, init-value: #f;

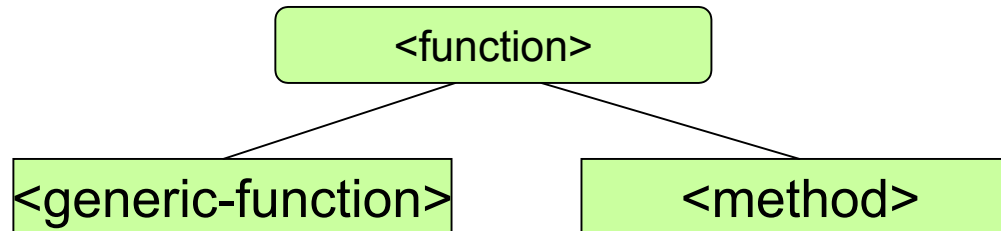
  sealed slot name :: <normalized-descriptor>, init-keyword: name;;
  sealed slot controller-class :: limited(<class>, subclass-of: <presentation-controller>);

  sealed slot state :: <object>, init-value: #f;

  sealed slot available-controllers :: <vector>, init-value: #[];
  sealed slot active-controllers :: <vector>, init-value: #[];

  sealed slot lru-count :: <integer>, init-value: 0;
  sealed slot cached? :: <boolean>, setter: #f, init-value: #f, init-keyword: cached;;
end class <presentation>;
```

The core concept of Dylan is the <function>



- A <method> is a callable unit of code identified by a fixed parameter signature

- <generic-function>


```

define open generic (&sequence1 :: <sequence>,
                    &sequence2 :: <sequence>, #key) => result :: <sequence>;

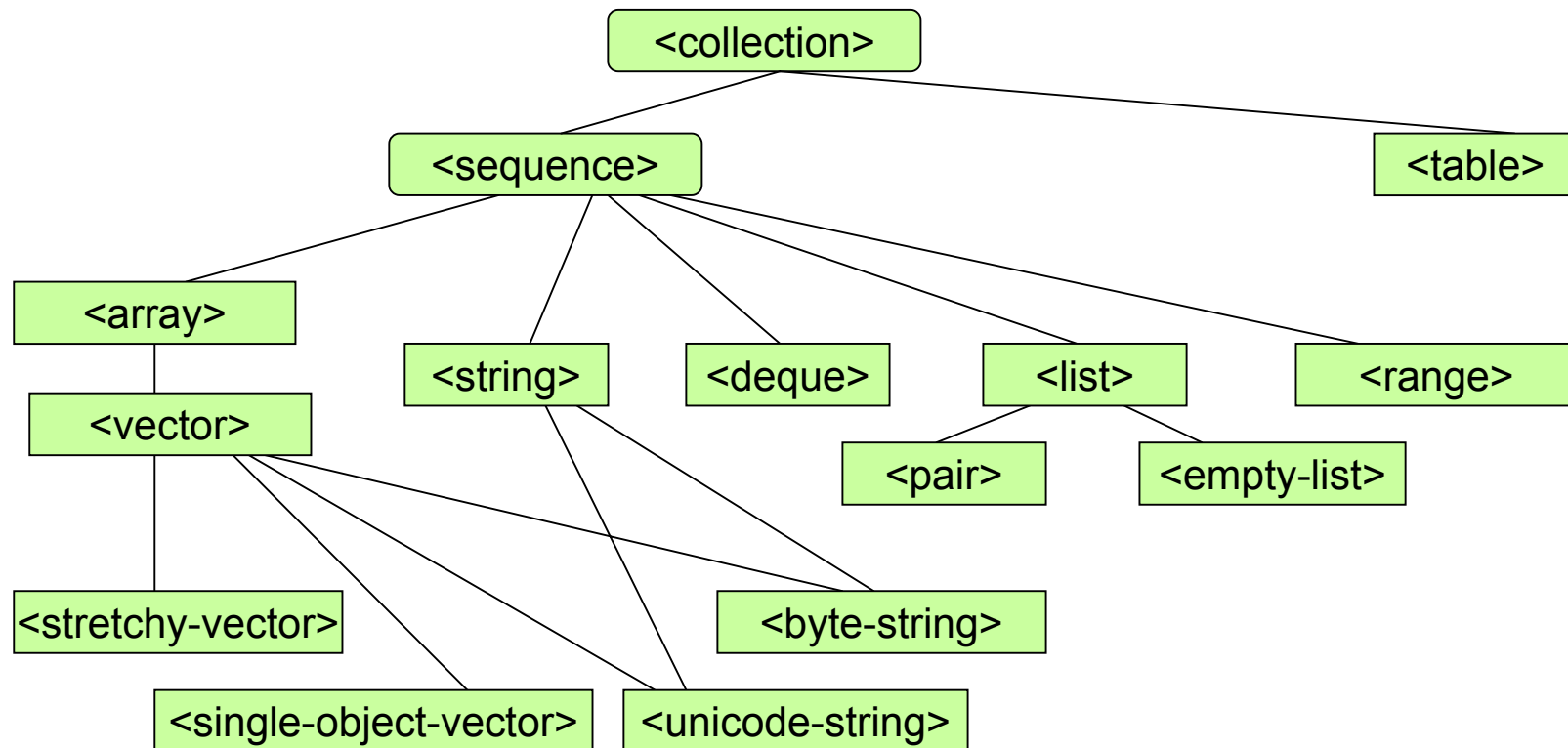
```
- Function


```

define method intersection (&sequence1 :: <lazy-sequence>, &sequence2 :: <sequence>,
                           #key key: &key :: <function> = identity,
                           test: &test :: <function> = \==)
  => result :: <lazy-sequence>;
  choose(method (&item)
          member?(&item, &sequence2, key: &key, test: &test)
        end method,
        &sequence1);
end method intersection;

```

A complete collection framework including functional iteration and mapping is supplied



- Supplied iteration and mapping functions:
 - do, map, map-as, map-into, member?, any?, every?, choose, ...

The condition (exception) handling of Dylan surpasses everything known in the C-world

- Both exceptions and handlers are objects
- Handlers can fix the problem and return to the signalling code block or they can return a new object
- Exceptions are objects
- Exceptions can be caught by handlers
- Several handlers can be attached to a block
- Simple exception handling
- Nested exception handling

```

local method run-block-lock ()
  let &run :: <integer> = 0;

  while (#t)
    block (return)
      let handler <kill> = method (&condition :: <kill>,
                                &next :: <function>)
        if (&run > 1000)
          return();
        end if;

        &next();
      end method;

      ... // Test code comes here...

      &run := &run + 1;
    end block;
  end while;
end method;

```

Agenda

- History of Dylan
- Concepts of the language
- **Multidimensional polymorphism - a silver bullet**
- Functional programming with Dylan
- Dylan in a *real* project

Dylan combines multiple inheritance with the so called multimethod dispatch

- Classes define no direct methods beside the slot methods
- Dylan uses generic functions for the active part of a program
- Each generic function can be compromised of methods that adhere to the parameter contract of this function
- Parameters of methods are strongly typed and define the type of arguments the method can be applied to
- The number of the so called applicable methods are computed for every concrete argument situation prior to the function invocation
- The applicable methods are sorted according to their specificity using a class precedence list (short: CPL) algorithm
- The most specific method is invoked
- A method can invoke the next most specific method by calling `next-method()` ;

Even operators are generic functions in Dylan and can be specialized

```
define generic method \(x :: <object>,
                        y :: <object>) => z :: <object>;

define method \(r :: <ratio>, i :: <integer>) => (s :: <rat>)
  r + make(<ratio>, numerator: i)
end \(+;

define method \(s :: <string>, t :: <string>) => (r :: <string>)
  concatenate(s, t);
end method \(+;
```

After this definition a program can write:

```
let h :: <string> = "Hello";
let w :: <string> = "World";

let hw :: <string> = h + " " + w;
```

Multimethods allows much more cleaner code than using traditional object-oriented techniques

```

class Shape {
  bool intersect(Shape s) {
    /* generic case - slow */
  }
}
class Rect {
  bool intersect(Shape s) {
    if ( s instanceof Rect ) {
      /* simple and fast */
    }
    else {
      super.intersect(s);
    }
  }
}
class Circle {
  bool intersect(Shape s) {
    if ( s instanceof Circle ) {
      /* simple and fast */
    }
    else {
      super.intersect(s);
    }
  }
}

```

```

define generic intersect (s1 :: <shape>,
                          s2 :: <shape>)
  => <boolean>;

define method intersect (s1 :: <shape>,
                          s2 :: <shape>)
  => <boolean>;
  /* generic case - slow */
end method;

define method intersect (s1 :: <rect>,
                          s2 :: <rect>)
  => <boolean>;
  /* simple and fast */
end method;

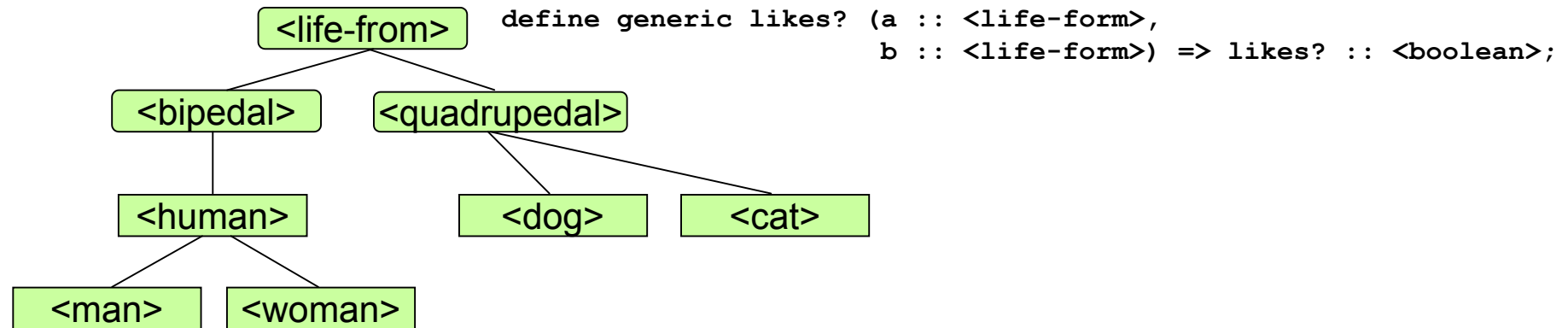
define method intersect (s1 :: <circle>,
                          s2 :: <circle>)
  => <boolean>;
  /* simple and fast */
end method;

```


Methods can be added to generic functions at any time

- When a generic function is not sealed, methods can be added by everyone at any point in the program
- Methods can be added syntactically according to the name scope or by using a call to `add-method(...)`;
- Every method must comply with the parameter signature of the generic function
- Methods can belong to several generic functions (esoteric feature)
- Methods can be even removed from a generic function by calling `remove-method(...)`;
- A complete set of introspection function are available

A complex sample



```

define method likes? (a :: <life-form>, b :: <life-form>) => likes? :: <boolean>
  #f;
end method likes?;

```

```

define method likes? (cat :: <cat>, whoCares :: <life-form>) => likes? :: <boolean>
  gives-food?(whoCares);
end method likes?;

```

```

define method likes? (dog :: <dog>, human :: <human>) => likes? :: <boolean>
  pet?(human, dog);
end method likes?;

```

```

define method likes? (man :: <man>, woman :: <woman>) => likes? :: <boolean>
  looks-good?(woman);
end method likes?;

```

```

define method likes? (woman :: <woman>, man :: <man>) => likes? :: <boolean>
  intelligent?(man) & rich?(man);
end method likes?;

```

Agenda

- History of Dylan
- Concepts of the language
- Multidimensional polymorphism - a silver bullet
- **Functional programming with Dylan**
- Dylan in a *real* project

Dylan provides a complete set of functional programming constructs

■ Functional com

— Closures

```
define method addX (n :: <number>, x :: <number>)
  n + x;
end;
```

■ Func

— apply

```
define method add2 (x :: <integer>, y :: <integer>)
  # do(method (a, b) print(a + b); end,
  er      #(100, 200, 300, 400), #(1, 2, 3));
```

■ Iteration

— do, etc

```
101
202
303
=> #f
# map(\+, #(1, 2, 3), #(4, 5, 6));
# => #(5, 7, 9)
# => # reduce(max, 10, #(2, 4, 6, 11));
=> 10 # => 11
# score => # reduce1(\+, #(1, 2, 3, 4, 5));
=> 11 # => 15
# score => # choose(even?, #(3, 1, 4, 5, 6, 2));
=> 130 => #(4, 6, 2)
# score # choose-by(even?, range(from: 1),
=> 470      #("a", "b", "c", "d", "e", "f", "g", "h"));
=> #("b", "d", "f", "h")
```

Sample from a real application

```

define function build-from-part (&environment :: <sequence>,
                                &expression :: <select-query-expression>,
                                #rest &args) => (from-part :: <string>,
                                                  hints :: false-or(<string>));

let &variables :: <sequence> = remove-duplicates!(as(<vector>, first(&environment).variables));
let &classes :: <sequence> = choose(rcurry(instance?, <transformed-class>), &variables);
let &structs :: <sequence> = choose(rcurry(instance?, <transformed-struct>), &variables);
let &sequences :: <sequence> = choose(rcurry(instance?, <transformed-sequence>), &variables);
let &first? :: <boolean> = #t;
let &from-part :: <string> = $empty-string;
let &hints = #f;

local method make-table-from-string (&alias :: <string>, &table :: <string>)
    ...
end method;

unless (empty?(&classes))
    &classes := choose(complement(compose(empty?, specificity)), &classes);

    unless (empty?(&classes))
        let &class :: <transformed-class> =
            first(sort!(&classes, key: compose(curry(reduce, \+, 0), specificity)));
        let &specificity :: <integer> = reduce(max, 0, &class.specificity);

        ...
    end unless;
end unless;

```

Agenda

- History of Dylan
- Concepts of the language
- Multidimensional polymorphism - a silver bullet
- Functional programming with Dylan
- **Dylan in a *real* project**

Dylan can be used successfully in very large commercial information systems

- Project FIM for GEMA
 - Central application of the company
 - About 50 person-years development effort
 - About 4 million lines of code written completely in Dylan
 - 1.5 million lines of code for technical framework
 - 2.5 million lines of code for the application
 - About 400 dialogs and 45 batches
 - About 800 persistent classes (tables) in the underlying database
 - Some tables have about 30 million entries
- The project has been started before Java matured and has gone into production before the first J2EE application server was commercially available

Some screenshots of FIM (1/4)

27458 - Dostal, Frank

Maske Objekt Navigieren Geschäftsvorfälle Liste

Name: **Dostal, Frank** Adresse: **Hochallee 43
D-20149 Hamburg** Aktuelle Vorgänge

Berec-Nr.: **27458** Kto.-Nummer: **FIM-1-0-27458-1**

Kontobewegung | Auszüge / Salden | Vollmacht | Mitglied

– Berufszugehörigkeit

Beginn	Ende	Beendet?	BG	Status	Bezeichnung	Koopt.-Knz.	Koopt.-Art
01.01.1967	31.12.1970	N	T	M	Angeschlossenes Mitglied		
01.01.1971	31.12.1976	N	T	A	Außerordentliches Mitglied		
01.01.1977		N	T	O	Ordentliches Mitglied		

Navigator (Server: fim)

Navigator Liste

Format: Standard

- Kontoinhaber
 - Geschäftspartner
 - Buchen
 - Konten
 - Zahlungen an FRW
 - Zahlungseingang von FRW
 - Abrechnungen (BUCHU)
 - MG
 - Steuer
 - UST 100,00 7,00 %
 - Adressen
 - Vermerke
 - 23 Zahlungsauftrag
 - 80 Kredit
 - 28 Verlagskonto beachten
 - 92 DTV-Beitrag
 - 43 Vorauszahler
 - 98 Siehe freie Texte
 - Berufsgruppe
 - Ordentliches Mitglied**
 - Verband
 - DTV
 - GEMA
 - Spartensummen
 - Dokumente
 - Unterkontokorrent
 - Auftragskreditkonto

Kreditorenkonto

Some screenshots of FIM (2/4)

B 44440050-Kalinski, Maria

Bericht: Standard

Name:
Zahlungsart: Überweisung
Gültigkeitsbeginn: 05.11.2002
Gültigkeitsende:

Zahlungsverwendungen:
Darlehensauszahlung
Tantiemenauszahlung

Bankverbindung:
Kontonummer BLZ/SWIFT Institut
45645474 10010111 BFG BANK
Abweichende Bankverbindung: ..

Auftragspositionen:

Lfd. Nr.	Ausf.-Tag	Wert	Verwendungszweck	ME	FREQ	FREQ-BEZ.
0		2		REST	Alle FIM-Auszahlungen	0

Adressbezug:
Adressbezug: Simmernerstr 134 tralala 56075 Koblenz DEUTSCHLAND

Zahlungsvereinbarung-Konto (Neu)

Kreditorenkonto Bestätigen Ablehnen Abbrechen

Some screenshots of FIM (3/4)

The screenshot shows a window titled "Dokumentation" with a search bar containing "Adressinhaber". Below the search bar, there are checkboxes for "Name", "Kurzname", "Synonyme", and "Beschreibung". The search results are displayed in a table with columns "Typ", "Name", and "Beschreibung".

Typ	Name	Beschreibung
▲	Adressinhaber	Ein Adressinhaber ist eine Person eine eigenständige Organisat
▲	Adressinhaber-Rolle	Die Adressinhaber-Rolle spezifiziert zum einen den Adressinhab
▲	Art eines weiteren Codes einer Adressinhaberrolle	Die Art eines weiteren Codes einer Adressinhaberrolle ist eine S
▲	Kategorie einer Adressinhaberrolle	Eine Adressinhaberrollen-Kategorie ist eine Gruppierung von Ad
▲	Kommunikationsverbindung eines Adressinhabers	Eine Kommunikationsverbindung eines Adressinhabers ist eine I

Below the table, there is a tree view showing the hierarchy of the "Adressinhaber" entity. The tree view shows "Geschäftspartner" as the parent, with "Adressinhaber" as a child. Under "Adressinhaber", there are several sub-entities: "Adressbezug", "Adresse", "Adresse-ISO", and "Adressinhaber" (highlighted). The "Adressinhaber" sub-entity has several attributes: "Anlagedatum", "Bearbeitet?", "Löschdatum", "Name", "Schlüssel", "Vorname", "Adressbezüge", and "Adressinhaber".

The right pane shows the details for the selected "Adressinhaber" entity. It includes the following information:

- Bezeichnung:** Adressinhaber
- Beschreibung:** Ein Adressinhaber ist eine Person eine eigenständige Organisationseinheit oder eine Örtlichkeit die über eine eigene Adresse verfügt wobei diese der GEMA nicht bekannt sein muß. -> Anmerkungen Dazu gehören zum einen die Geschäftspartner der GEMA wie z.B. Kunden oder Lieferanten und zum anderen sog. Standorte (Örtlichkeiten) wie z.B. die Olympiahalle München oder das Kongreßzentrum Berlin auch wenn diese der GEMA nicht bekannt sind.
- Unterklassen:** [Personengruppe](#), [Juristische Person](#), [Natürliche Person](#), [Standort](#)
- Abstrakt:** Ja
- Persistent:** Ja

Some screenshots of FIM (4/4)

The screenshot displays the EDW (Projekt: System1-Development) application window. The interface is divided into several panels:

- Struktur-Browser:** Shows a tree view of sub-classes (Unterklassen) including:
 - <function-tree>
 - <import-object-tree>
 - <inspector-tree>
 - <navigator-profile-tree>
 - <navigator-tree>
 - <object-class-tree>
 - <object-tree>
 - object-tree (vs)
- Objekt-Browser:** Shows a tree view of the development repository (X-DEVELOPMENT:Repository) with sub-entries like DYLAN, X-COMMUNICATION, X-DATABASE, X-DATABASE-TOOLS, X-DEPLOYMENT-TOOL, and X-DEVELOPMENT. The 'Repository' folder is expanded, showing sub-entries like Internal, Archive, and Private Constants & Variables.
- Objekt-Auswahl:** A search panel with a search field containing '<object-tree>'. It includes search options for 'Bezeichner' (checked), 'Source', and 'Dokumentation'. The search results table shows:

Typ	Bezeichnung
▲	EDW-Framework:XDL-FRAMEWORK-COMPONENTS:Object-T
- Objekt-Info:** A panel displaying metadata for the selected object:
 - COPYRIGHT: Qcentic GmbH, Germany
 - AUTHOR: Oliver Juwig, Juergen Krey
 - LIBRARY: X-DEVELOPMENT
 - MODULE: Repository
 - REVISION: 2.0.OP
 - DESCRIPTION: Alle Objekte, die mit der Entwicklungsumgebun modifiziert werden, werden in einem Reposito: Repository teilt diese Objekte in Gruppen vo: Workspaces auf, die einem Entwickler zugeord: kleinste konfigurierbare Einheit in einem Wo: Bibliothek und alle enthaltenen Objekte.
 - HISTORY:
- Protokoll:** A log panel showing a list of events:

Art	Uhrzeit	Nachricht
☑	10:45:52	Werkzeug Protokoll geöffnet
☑	10:45:53	Werkzeug Fehler/Warnungen geöffnet
☑	10:45:54	Werkzeug Prozess-Browser geöffnet
☑	10:45:55	Werkzeug Task-Manager geöffnet
☑	10:45:56	Werkzeug Listener geöffnet
☑	10:46:01	Werkzeug Haltepunkte geöffnet
☑	10:46:25	Werkzeug Struktur-Browser geöffnet

Discussion

Contact:

Oliver Juwig

sd&m Research

software design & management

Thomas-Dehler-Str. 27, 81737 München, Germany

Tel +49 89 63812-653, Fax -911

Tel +49 2241 9737-413

Mobile +49 171 3105384

<mailto:oliver.juwig@sdm.de>

<http://www.sdm-research.de>