Model-Driven Development with Executable UML and SOLoist

Prof. Dr. Dragan Milićev
University of Belgrade, School of Electrical Engineering, Dept. of Computing
dmilicev@etf.rs, www.rcub.bg.ac.rs/~dmilicev

Presentation for UMLChina, September 3rd, 2015

OUTLINE

- Introduction (5')
- UML Schools: Models as Sketches or Blueprints vs. Executable Models (10')
- About SOLoist (5')
- Modeling Structure in SOLoist (15')
- Querying in SOLoist: OQL, Query Builder (10')
- Database Features in SOLoist: Optimizations, Performance, Scalability (5')
- Modeling Behavior in SOLoist: Operations/Methods, Commands, State Machines (15')
- Web GUI Development in SOLoist: Concepts, Principles, Library (20')
- Conclusions (5')
- Q&A (30')

INTRODUCTION

…of the speaker…

- Full professor at University of Belgrade, School of Electrical Engineering, Department of Computing (www.etf.rs)
- Founder and CEO of SOL Software (www.sol.rs)
- Fields of expertise: software engineering, model-based engineering, model-driven development, UML, software architecture and design, business process modeling, information systems, and real-time systems
- Member of PCs of several premier international conferences on model-based engineering: MODELS, ECMFA, and MODELSWARD
- Member of the Editorial Board of Springer's Software and System Modeling journal (SoSyM)

INTRODUCTION

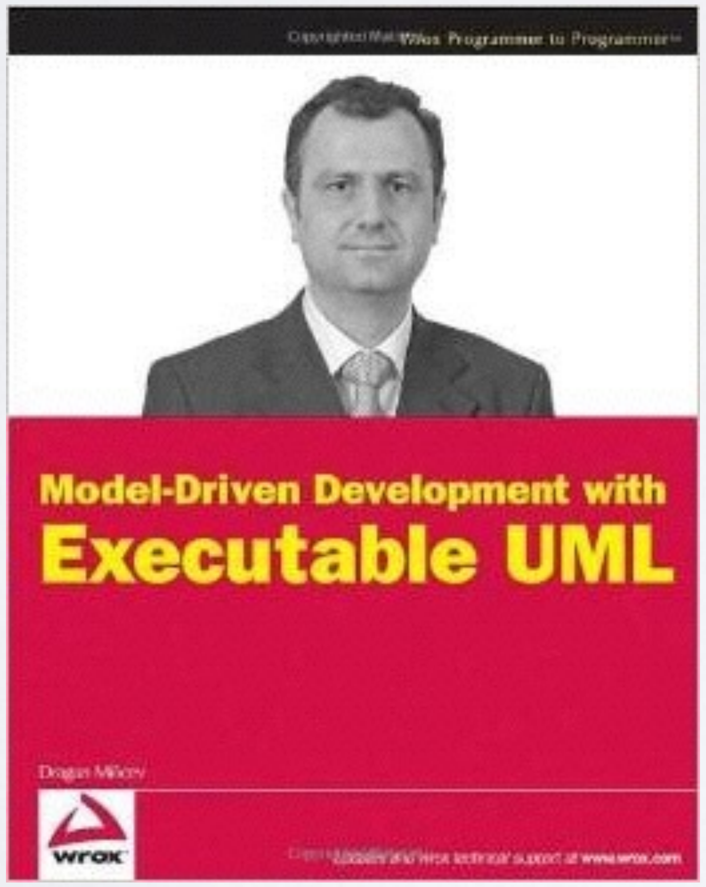
About my research and publications:

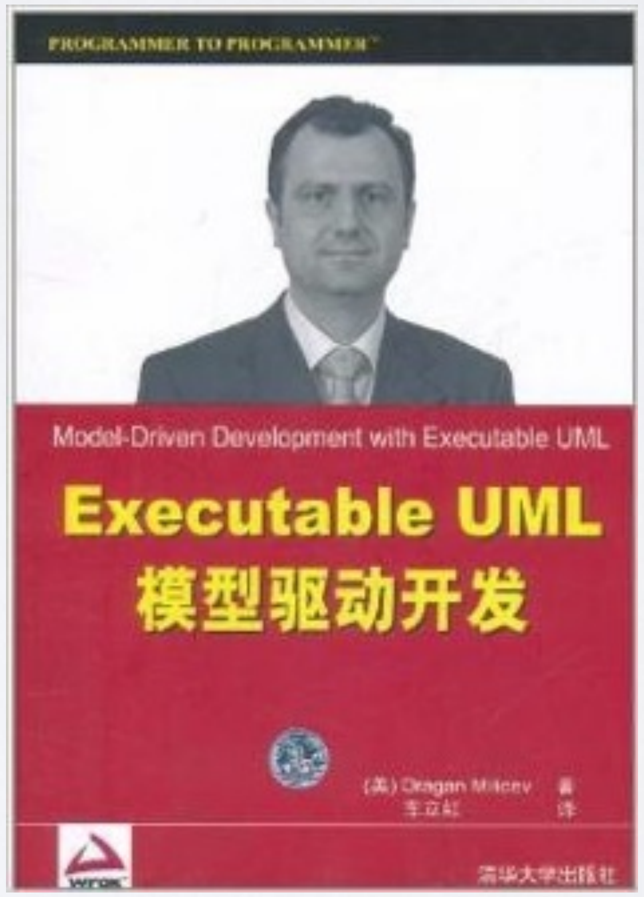
- Three books (bestsellers) in Serbian on OOP, C++, and UML
- Papers in journals and conferences, some contributing to modeling and UML. A short selection:
    - Milićev, D., "Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments," *IEEE Transactions on Software Engineering*, Vol. 28, No. 4, April 2002
    - Milićev, D., "On the Semantics of Associations and Association Ends in UML," *IEEE Transactions on Software Engineering*, Vol. 33, No. 4, April 2007
    - Milićev, D., "Towards Understanding of Classes versus Data Types in conceptual Modeling and UML," *Computer Science and Information Systems*, Vol. 9, No. 2, June 2012
    - Milićev, D., Mijailović, Ž., "Capsule-Based User Interface Modeling for Large-Scale Applications," *IEEE Transactions on Software Engineering*, Vol. 39, No. 9, pp. 1190-1207, September 2013
    - Milovanović, V., Milićev, D., "An Interactive Tool for UML Class Model Evolution in Database Applications," *Software and Systems Modeling*, September 2013
    - etc. (full list available at www.rcub.bg.ac.rs/~dmilicev)

INTRODUCTION

About my research and publications:

- The Wiley/Wrox book on MDD with Executable UML (2009)
- Chinese translation published by Tsinghua University Press (2011)





INTRODUCTION

About my professional activities:

- 30 years of industrial experience in building complex commercial software systems
- Served as chief software architect, project manager, consultant, or developer in over 30 large industrial projects:
  - With customers and partners in: USA, Germany, France, Italy, Norway, Serbia, …
  - In different domains: e-government, telecom, health and social care, customer relationship, human resources, document management, engineering, …
  - Of different size: from SME to national-level (e.g. National Cadastre of Serbia, National e-ID Document Issuance of Macedonia, National Civil Register of Iraq, etc.)

CHAPTER 1
UML SCHOOLS

Models as Sketches or Blueprints vs. Executable Models

UML SCHOOLS

Two schools of using and applying UML:

- “**Informal UML** School” (Martin Fowler et al.):

  UML used to:

  - sketch,
  - make blueprints,
  - document

  architecture, design, etc. of a software system.

- “**Formal UML** School” (Bran Selic, Steve Mellor, et al.):

  UML used as an executable (“programming”) language

THE INFORMAL UML SCHOOL

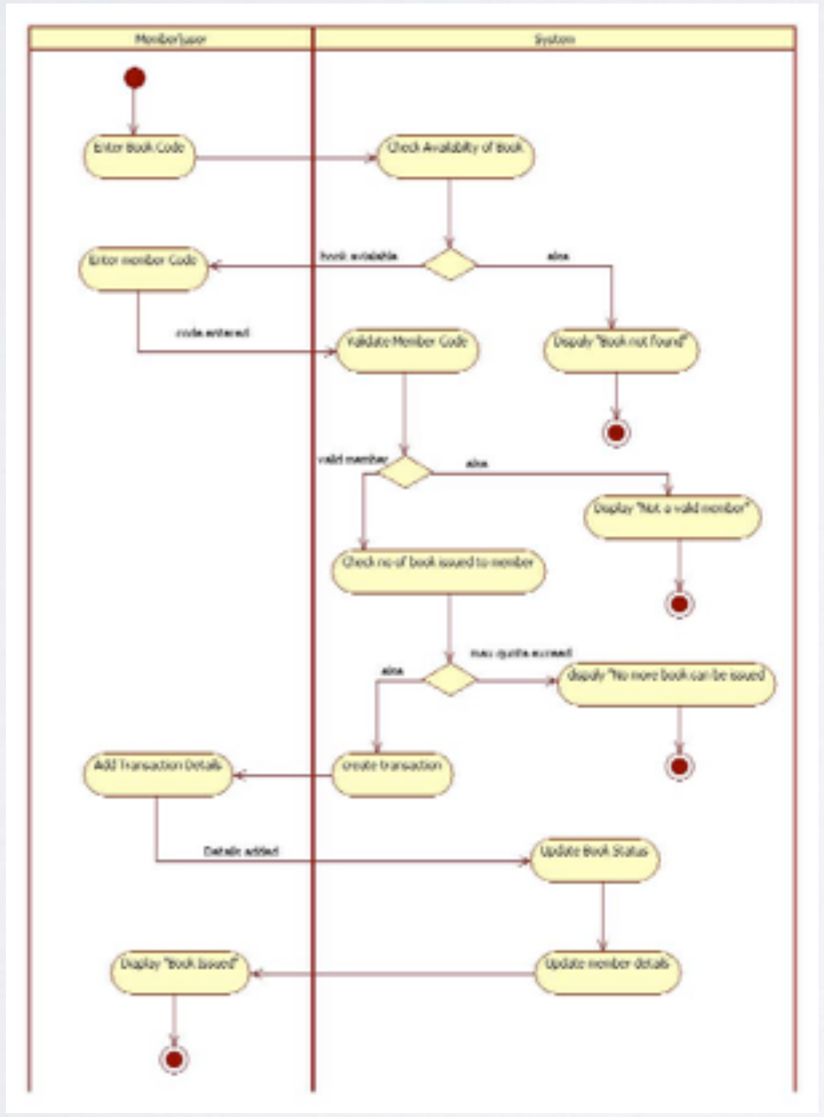
Characteristics:

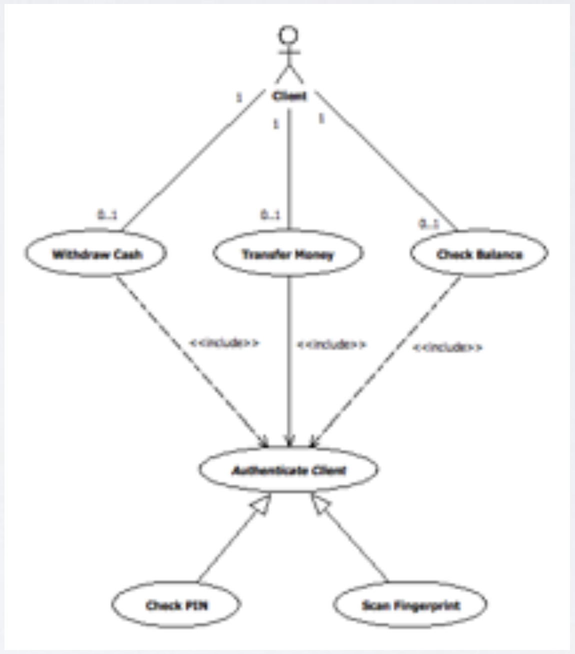
- UML used to *sketch*, make *blueprints*, or *document* architecture, design, pieces of structure or behavior, key concepts or mechanisms, business processes/ workflows, etc.
- Used in early phases of requirements engineering, conceptualization or design (sketches), and in late phases of design and documentation (blueprints)
- Convenient as a vehicle to convey an idea or a message to others
- Does not require deep knowledge of UML advanced concepts and their semantics
- Very lightweight and flexible

THE INFORMAL UML SCHOOL

Characteristics:

- UML models used in early stages, for requirements engineering, conceptualization, or architecture and design sketching are usually very abstract, but completely vague with semantics - can be interpreted in many different ways, depending on the context, assumptions, viewpoint, implementation strategy, etc.





THE INFORMAL UML SCHOOL

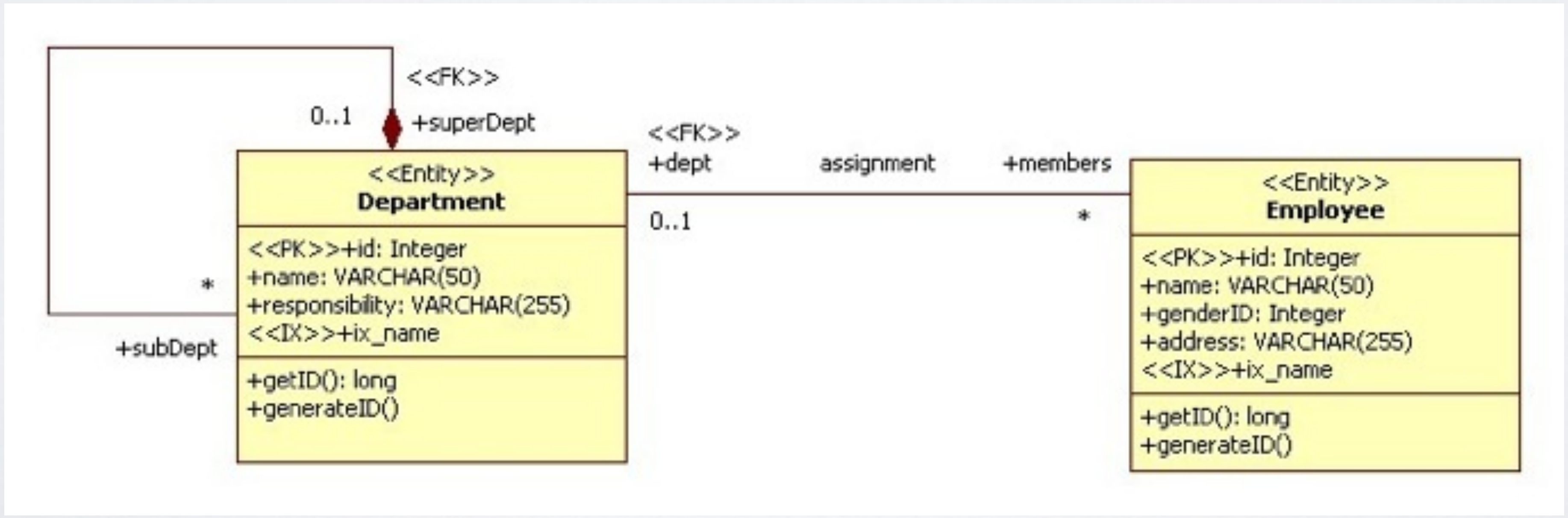
Consequence: ***rush-to-code syndrome!***

“A pervasive unease during the early development phases, a prevailing attitude among the developers that requirements definition and design models are ‘just documentation,’ and a conviction that the ‘real work’ has not begun until code is being written.” [Selic et al., *Real-Time Object-Oriented Modeling*, Wiley, 1994]

THE INFORMAL UML SCHOOL

Characteristics:

- UML models used as blueprints of implementation artifacts usually do not carry additional (higher-level abstraction) semantics, but simply describe the artifacts implemented in other "implementation-level" technologies, e.g., OOPL code, database schema, implementation framework, etc.
  - Lower-level abstraction == reduced expressiveness => models are over-specified: unnecessary additional work => reduced productivity!
  - Models are simple "drawings of code/schema/…". May help in understanding the artifacts, but do not bring additional value to the implementation process - significant additional work => reduced productivity!



THE INFORMAL UML SCHOOL

Characteristics:

- UML models used for documentation only suffer from *phase discontinuities*:
  - discrepancies between the real (executable) artifacts and the documentation - documentation is never fully up-to-date
  - updating documentation is tedious => significant unnecessary additional work => reduced productivity

Overall consequences:

  - switching to UML modeling (from just simply traditional programming/coding) in an ***inappropriate*** way adds burden to developers and reduces development productivity - moves us one step backwards!
  - many people have been disappointed by using UML for that reason!

THE FORMAL UML SCHOOL

Solution:

- Use UML in a controlled, appropriate way - with *formal, executable semantics*
- Formal, executable semantics is a key to usage of a software design language:
    - unambiguity of interpretation
    - understanding of a concept or rule is much easier by watching its runtime effects (*runtime semantics*) - programmers usually learn a new language by experimenting with simple examples that illustrate the runtime semantics

      (otherwise, one has to map the semantics of a new concept to something (s)he is familiar with, in a different semantic domain; in this context of UML, this is usually an OOPL, relational database or similar - turns modeling into "drawing code")
    - UML models become *authoritative, executable artifacts* - the *real software*!
    - No more rush-to-code syndrome
    - No discrepancy between software and documentation - models are accurate documentation

THE FORMAL UML SCHOOL

- A modeling language is *executable* if the concepts of a modeling language have formal semantics that enable models to be transformed completely automatically into forms that can be executed or interpreted in a way that results in running applications.

- However, even traditional programming languages fit in this definition. What is the difference?

    - *Abstraction level*: modeling languages, either general-purpose (such as UML) or especially domain-specific ones (DSL), are typically more abstract, expressive and closer to the problem domain. (Note: even classical programming languages have been increasing their level of abstraction by time, so this is a vague and relative criterion!)

    - *Notation*: modeling languages often use a combined visual (diagrammatic) and textual notation, while classical programming languages use textual notations only. (This is not not a strict criterion, either as many modeling languages have also purely textual notation.)

    - *Strictness*: programs in classical programming languages must be fully complete and filled in with details to compile - compilers are very rigid and rigorous. Models do not have to be fully detailed, and can still remain executable (as long as they are well-formed).

ABOUT UML AND PROFILES

- UML is (mostly but not fully deliberately) designed to be vague in many points, with the lack of formal semantics and with many semantic variation points, in order to be flexible, adaptable, and usable in many ways
- fUML (Foundational Subset for Executable UML Models): an initiative and standard of OMG for tightening up the semantics of some parts of UML
- UML *profiles*: adaptations of the core language that allow you to:
  - select a subset of the language of interest for a particular domain, kind of applications, etc.
  - extend the core language with new concepts, or by interpreting the semantics of the existing concepts
  - define the semantics of the used concepts in a formal and executable way.
- OOIS UML - a profile for UML defined for object-oriented information systems, i.e. applications with:
  - complex conceptual model (vocabulary of a domain)
  - massive persistent instantiation of key abstractions (i.e., need for a database)
  - complex business logic (behavior)
  - rich user interface

CHAPTER II
ABOUT SOLoist

A Java-Based Framework for Model-Driven Development of Information Systems with Executable UML

ABOUT SOLoist

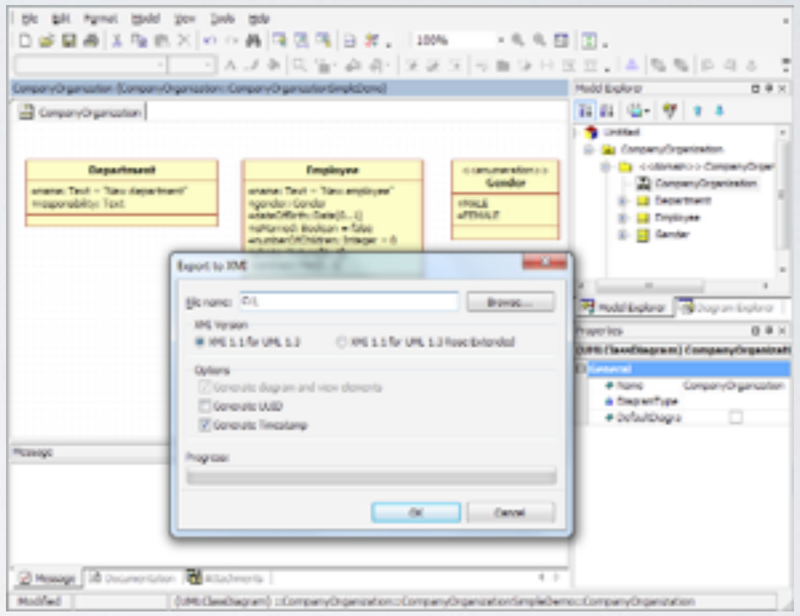
- SOLoist is a Java-based framework for model-driven development of web-oriented information systems with Executable UML (OOIS UML profile)
- SOLoist is *not*:
  - a UML modeling tool; instead, it uses a third-party UML tool
  - an object/oriented database (although it does provide the functionality of an OO database with UML semantics); instead, it stores the data in a standard, third-party relational database management system (RDBMS)
  - a programming language; instead, it uses Java as the implementation language of details.
- Current generation of SOLoist - G4. History:
  - G1, desktop,1999-2001: Rational Rose, Visual Basic
  - G2, desktop, 2002-2003: Rational Rose, C++, Microsoft Foundation Classes (MFC)
  - G3, desktop, 2003-2008: Rational Rose, C++, Qt
  - G4, Web, 2009-2015: StarUML, Java, Google Web Toolkit (GWT)
- G5 under development!

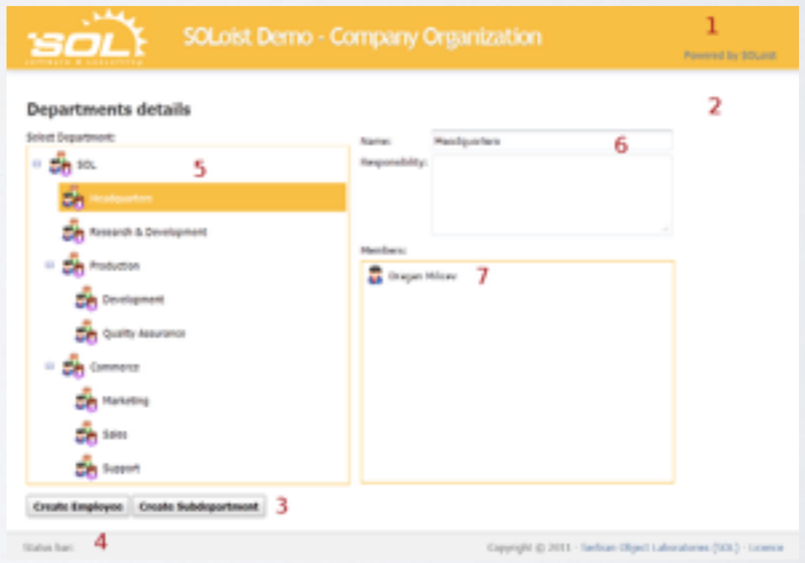
SOLoist DATASHEET

- SOLoist consists of:
  - plug-ins for StarUML for Java code generation for class models and state machine models
  - runtime environment (kind of a UML virtual machine) that provides UML runtime semantics over an API (“UML system calls”), object persistence, and UML reflection
  - basic model library
  - GUI library for rapid development with UML-semantically coupled widgets
- Development:
  - UML modeling tool: StarUML™ (an open-source, free UML modeling tool)
  - Target language: Java
  - IDE: Eclipse
- Execution:
  - Application server: Tomcat or any other (e.g. Wildfly)
  - DBMS: MySQL, Oracle, MS SQL Server, Sybase, adaptable to any other SQL-compliant RDBMS
- GUI framework: Google Web Toolkit (GWT)

DEVELOPMENT PROCEDURE

- Basic development procedure:
  - Develop a model in StarUML
  - Generate XMI for the SOLoist Runtime
  - Generate Java code from model
  - Write GUI code if necessary
  - Integrate and compile the code in Eclipse
  - Execute on a Web server
- Follow the Tutorial at www.soloist4uml.com

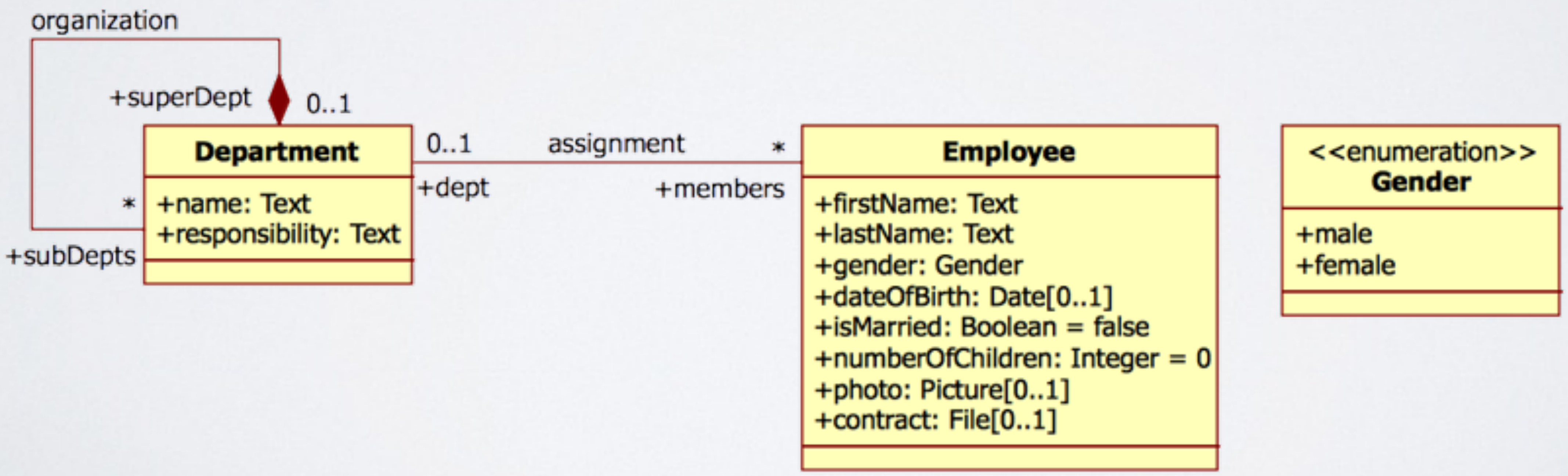




CHAPTER III
MODELING STRUCTURE IN SOLOIST

MODELING STRUCTURE IN SOLOIST

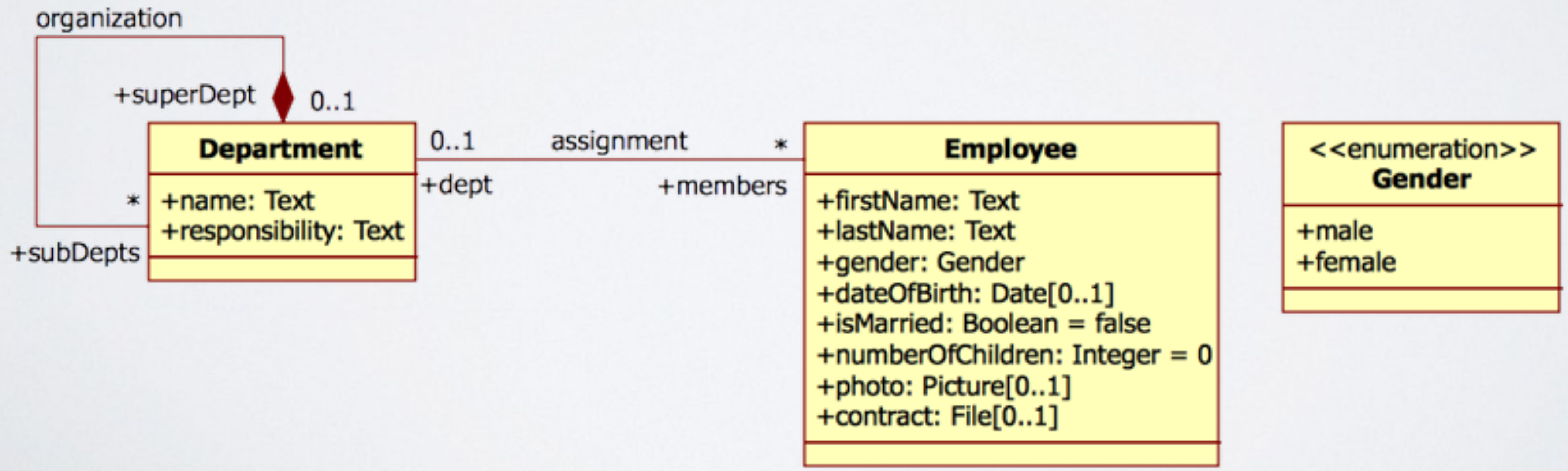
- Design a structural (class) model in UML and generate the app:
  - generate Java code
  - generate database schema using a SOLoist utility



MODELING STRUCTURE IN SOLOIST

You get immediately, without any additional actions in model or code:

- all classes are persistent by default (unless you specify them differently)
- no boilerplate manual coding and annotations (everything is generated)
- direct and transparent object persistence
- no need for object ID management, all this is managed by the framework



MODELING STRUCTURE IN SOLOIST

Compare it with using another persistence framework, based on an OOPL (e.g. Java) and its semantics. Example:

"Every entity must have a primary key (simple, composite or auto-generated):"

```
@TableGenerator(name="employeeGen", table="EJB_ORDER_SEQUENCE_GENERATOR",
    pkColumnName="GEN_KEY", valueColumnName="GEN_VALUE",
    pkColumnValue="EMPLOYEE_ID", allocationSize=10)

@Id

@GeneratedValue(strategy=GenerationType.TABLE, generator="employeeGen")

public Long getEmployeeID() { return employeeID; }
```

- Is this all really necessary?
- No. It is *accidental complexity* caused by a heavy influence of the underlying technology (relational database)

MODELING STRUCTURE IN SOLOIST

Complexity:

- *Essential complexity*: an inevitable component of complexity that is inherent to the very problem domain
- *Accidental complexity* arises purely from mismatches in the particular choice of tools and methods applied in the solution

We cannot avoid essential complexity, but we can (and should!) reduce accidental complexity

MODELING STRUCTURE IN SOLoist

Deal with an object as usual, and think of an object as a separate identity, residing in an abstract *object space*, from its creation to its destruction:

- Plain and natural coding
- No confusion wrt. object in memory vs. record in database
- No burden about loading the object and updating its database copy

```
Employee emp = new Employee();
…
emp.dept.set(dept);
…
emp.destroy();
```

`emp` is already persistent.
No need for "persist" or "save".

`dept.members` immediately includes `emp`

`emp` is removed from the object space (database)

MODELING STRUCTURE IN SOLOIST

Compare it with using another persistence framework, based on an OOPL (e.g. Java) and its semantics. Example:

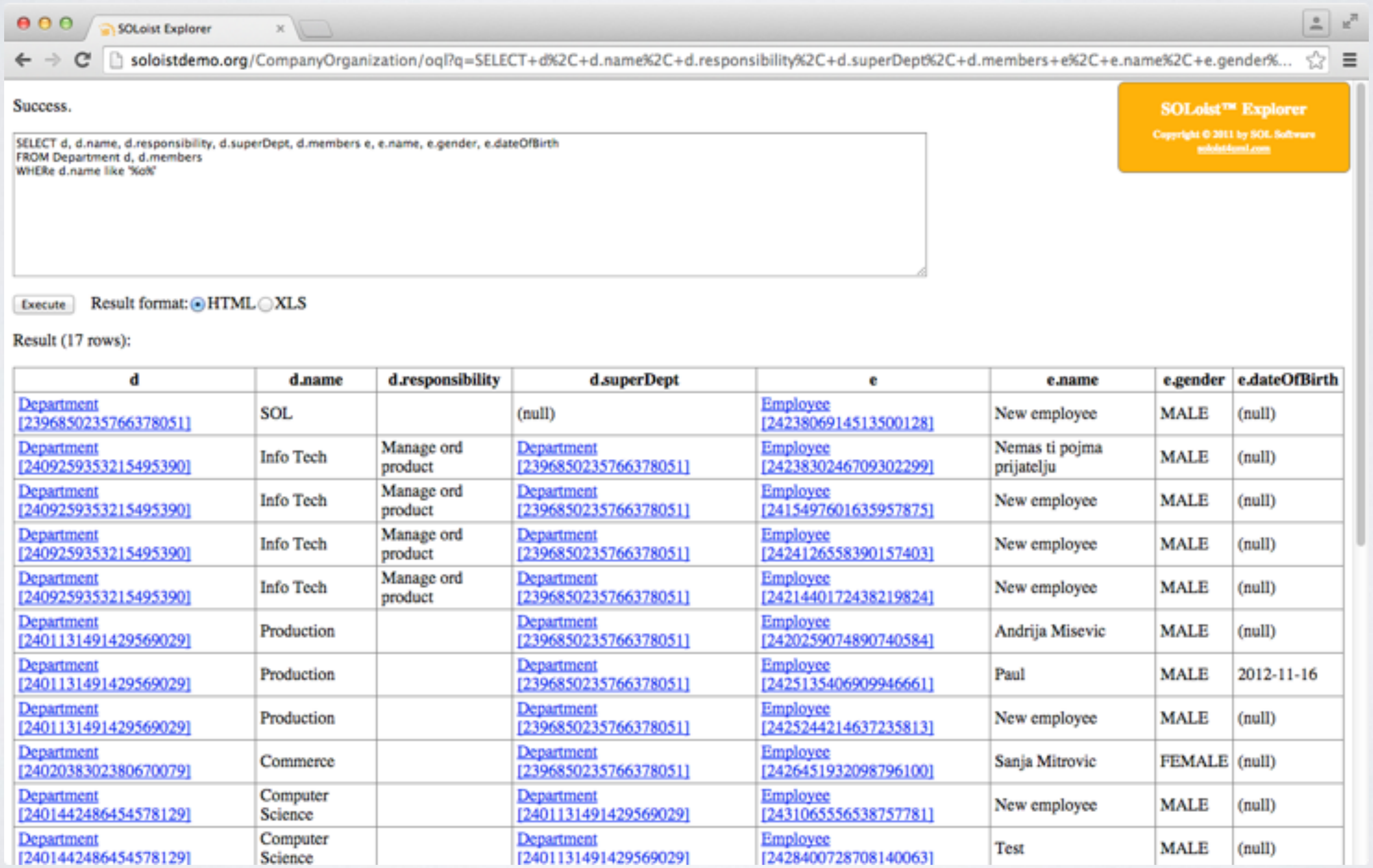
```
@PersistenceContextEntityManager em;
...
public Employee createEmployee(Department d){
  Employee empl = new Employee();
  d.getMembers().add(empl);
  em.persist(empl);
  return empl;
}
```

The developer must be aware of the fact that the Java object and its database representation are separate items that have to be kept in sync.

…Yet another instance of accidental complexity caused by the influence of the underlying implementation

MODELING STRUCTURE IN SOLoist

You also get - generic browsing of object space (SOLoist Explorer):


SOLoist Explorer
soloistdemo.org/CompanyOrganization/oql?q=SELECT+d%2C+d.name%2C+d.responsibility%2C+d.superDept%2C+d.members+e%2C+e.name%2C+e.gender%...
SOLoist™ Explorer
Copyright © 2011 by SOL Software
soloist4uml.com

Success.

```
SELECT d, d.name, d.responsibility, d.superDept, d.members e, e.name, e.gender, e.dateOfBirth
FROM Department d, d.members
WHERe d.name like '%o%'
```

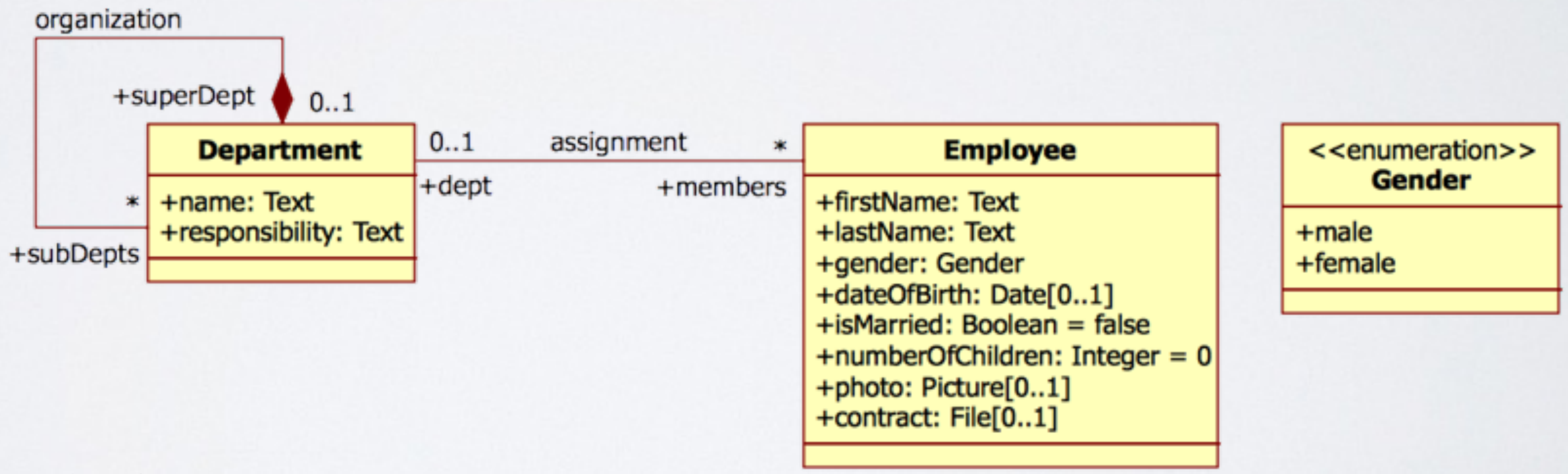
Execute Result format: [✓ HTML] XLS

Result (17 rows):

d	d.name	d.responsibility	d.superDept	e	e.name	e.gender	e.dateOfBirth
Department [2396850235766378051]	SOL		(null)	Employee [2423806914513500128]	New employee	MALE	(null)
Department [2409259353215495390]	Info Tech	Manage ord product	Department [2396850235766378051]	Employee [2423830246709302299]	Nemas ti pojma prijatelju	MALE	(null)
Department [2409259353215495390]	Info Tech	Manage ord product	Department [2396850235766378051]	Employee [2415497601635957875]	New employee	MALE	(null)
Department [2409259353215495390]	Info Tech	Manage ord product	Department [2396850235766378051]	Employee [2424126558390157403]	New employee	MALE	(null)
Department [2409259353215495390]	Info Tech	Manage ord product	Department [2396850235766378051]	Employee [2421440172438219824]	New employee	MALE	(null)
Department [2401131491429569029]	Production		Department [2396850235766378051]	Employee [2420259074890740584]	Andrija Misevic	MALE	(null)
Department [2401131491429569029]	Production		Department [2396850235766378051]	Employee [2425135406909946661]	Paul	MALE	2012-11-16
Department [2401131491429569029]	Production		Department [2396850235766378051]	Employee [2425244214637235813]	New employee	MALE	(null)
Department [2402038302380670079]	Commerce		Department [2396850235766378051]	Employee [2426451932098796100]	Sanja Mitrovic	FEMALE	(null)
Department [2401442486454578129]	Computer Science		Department [2401131491429569029]	Employee [2431065556538757781]	New employee	MALE	(null)
Department [2401442486454578129]	Computer Science		Department [2401131491429569029]	Employee [2428400728708140063]	Test	MALE	(null)

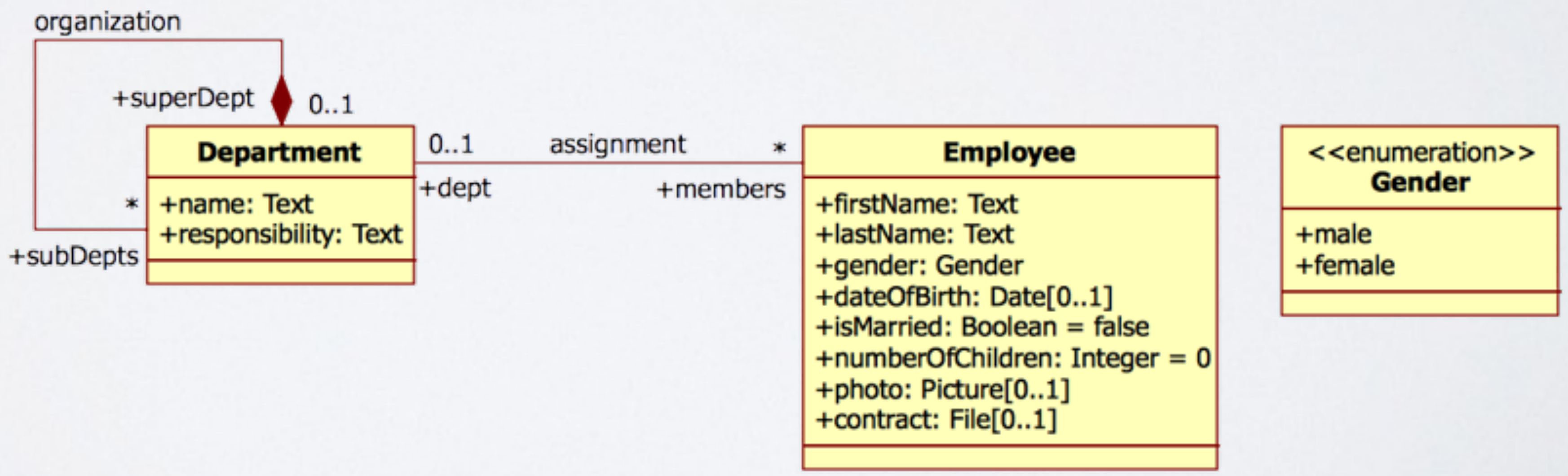
MODELING STRUCTURE IN SOLOIST

- Supported concepts:
  - Classes
  - Attributes
  - Associations (binary associations only)
  - Generalization/specialization (single inheritance only)
- Attribute types:
  - Boolean
  - Integer
  - Real
  - Currency
  - Text
  - Enumeration (user-defined)
  - File (binary)
  - Picture



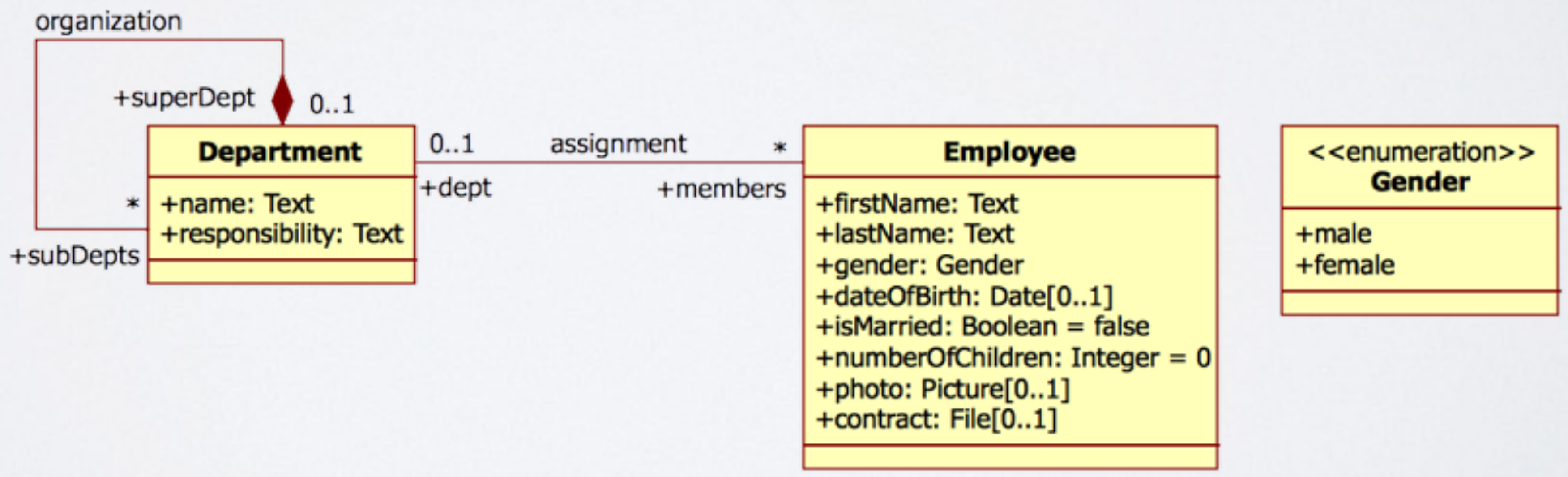
MODELING STRUCTURE IN SOLOIST

- Supported concepts for attributes:
  - name and type
  - visibility
  - initial value
  - multiplicity (single- or multi-valued): affects only the database schema and management, does not impose constraints!
  - *unique* constraint



MODELING STRUCTURE IN SOLOIST

- Supported concepts for associations and association ends:
  - associations only between classes
  - name (of association or end) is optional
  - aggregation (no semantic impact) and composition (with the meaning of propagated deletion)
  - multiplicity (single- or multi-valued): affects only the database schema and management, does not impose constraints!
  - visibility
  - ordering
  - navigability



MODELING STRUCTURE IN SOLOIST

For objects space manipulation, the SOLoist UML API ensures full UML semantics of *actions*:

```
Employee empl = new Employee();
dept.members.add(empl);
// or:
empl.dept.set(empl); // bot not both!
String emplName = empl.name.val().toString();
empl.name.set(newName);
dept.destroy();
```

Create object action (incl. constructor)

Create link (write slot) action
`dept.members` immediately includes **emp**

Read slot action.

Destroy object action. Implicitly destroys a
links of the object, and sub departments in
this case (due to the composition end).

MODELING STRUCTURE IN SOLOIST

Compare this with using another persistence framework, based on an OOPL (e.g. Java) and its semantics:

```
class Department {
   public List<Employee> members = new ArrayList<Employee>;
   …
}

class Employee {
   public Department dept;
   …
}

// And both of these are needed:
dept.members.set(empl);
empl.dept.set(empl);
```

Problems with the expressiveness of the OOP level:

- It is the responsibility of the developer to keep the fields in sync – prone to error
- More primitive semantics: unidirectional references instead of (bidirectional) links (relationships)

MODELING STRUCTURE IN SOLOIST

- Problem with using the "traditional" approach =
  UML class model + actions in OOPL (or, even worse, in SQL):

  semantic discontinuity: the lack of formal coupling between the elements of different kind (e.g., structure and behavior)

- Semantic discontinuity is another cause of the rush-to-code syndrome

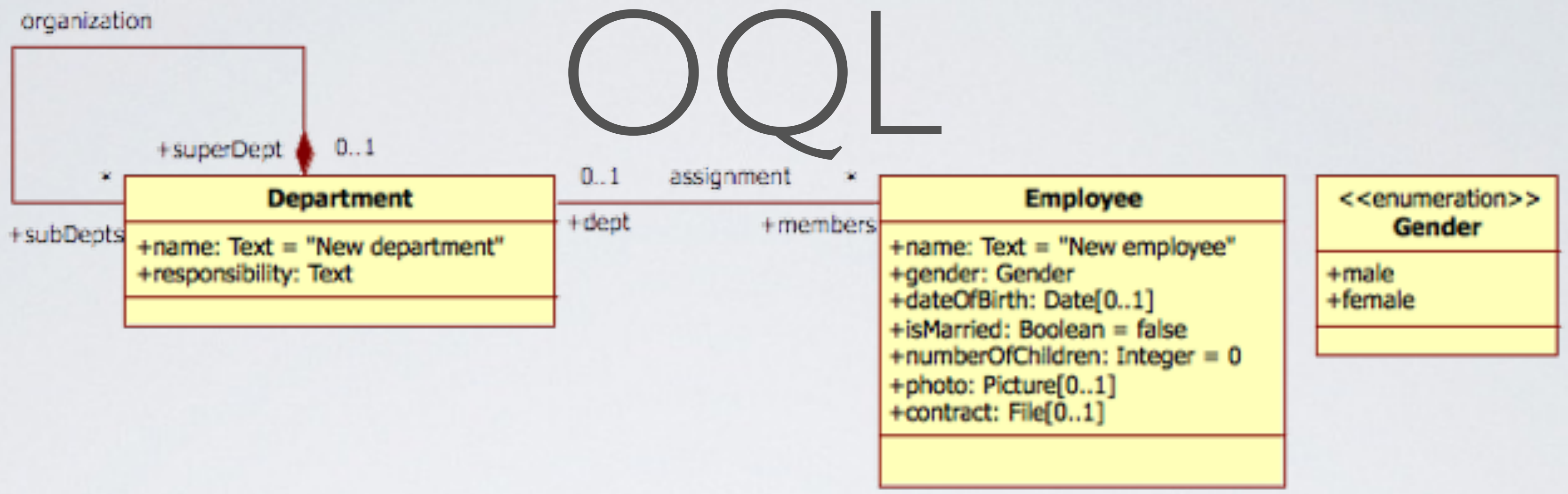
- Another bad example of semantic mismatch causing accidental complexity (=additional unnecessary work): DAO layer — *Anemic Object* anti-pattern

- **Conclusion**: Structural UML modeling in OOIS UML with SOLoist eliminates accidental complexity (=> improved productivity) and does not suffer from semantic discontinuity (= full semantic coupling)

CHAPTER IV
QUERYING IN SOLOIST

OQL and Query Builder

OQL



- Object Query Language (OQL) is an object-oriented descendent of the Structured Query Language (SQL)
- In OQL, you deal with objects of classes and with links of associations, instead of tables, records, and fields
- Example: get the names and date of births of all employees of the department 'R&D'

```
SELECT     e.name, e.dateOfBirth
FROM       Department d, d.members e        // or: FROM Employee e, e.dept d
WHERE      d.name = 'R&D'
```

- The equivalent SQL query must join the tables:

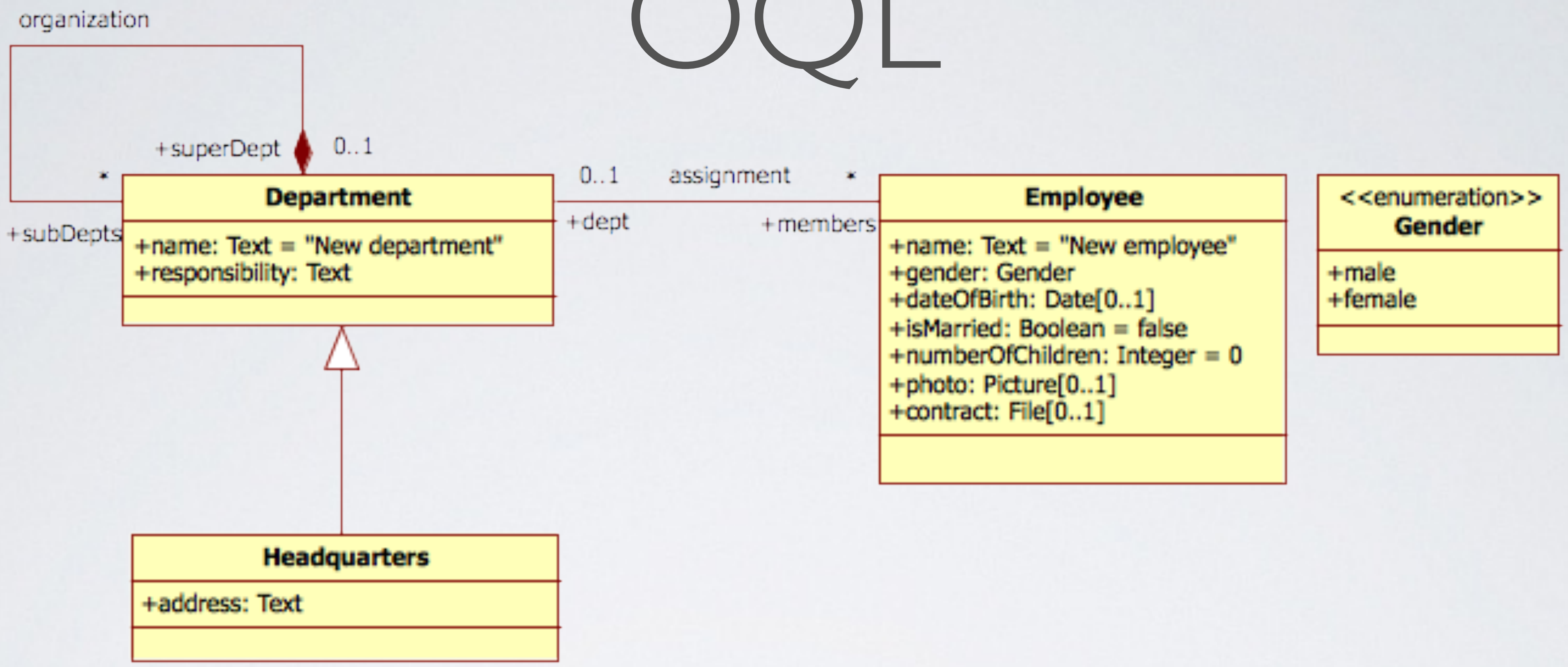
```
SELECT     e.name, e.dateOfBirth
FROM       Department d, Employee e         // or: FROM Department d INNER JOIN Employee e …
WHERE      e.dept = d.ID AND d.name = 'R&D' //    … ON e.dept = d.ID WHERE d.name = 'R&D'
```

- In case of a many-to-many association, the equivalent SQL query must join three tables (think about maintenance!):
    - OQL: `SELECT e.name, e.dateOfBirth FROM Department d, d.members e WHERE d.name = 'R&D'`

      Exactly the same
    - SQL: `SELECT e.name,e.dateOfBirth FROM Department d, Assignment a, Employee e WHERE d.ID=a.dept AND a.members=e.ID AND d.name = 'R&D'`

      Significantly different, especially in case of several joins

OQL



- Inheritance and specialization (down-casting) example: select Employees of Headquarters

```
SELECT     e.lastName, e.dateOfBirth, h.name, h.address
FROM       Employee e, e.dept:Headquarters h
WHERE      …
```

Inheritance

Specialization (down-casting)

OQL

- Queries can return:
  - slots, holding a value or a collection of values:

```
SELECT e.name, e.dateOfBirth, e.dept FROM…
```

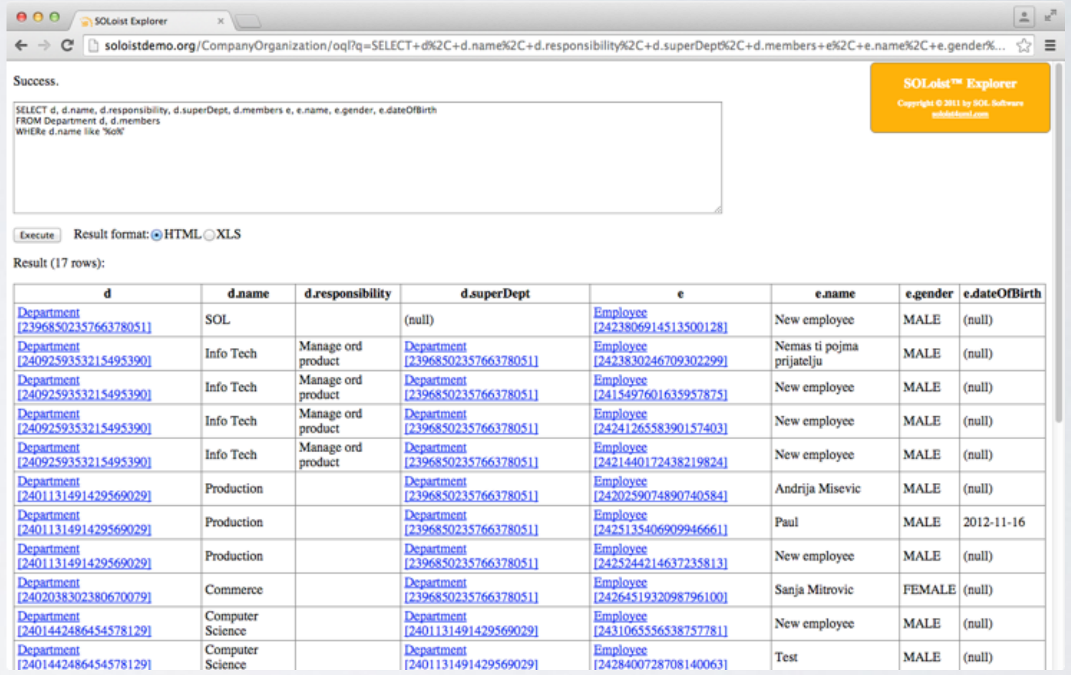
  - objects

```
SELECT e, e.name, e.dateOfBirth, d FROM…
```

- Queries can have parameters:

```
SELECT d, d.name, …
FROM Department d, d.members
WHERE d.name like #name#
```

- Queries can be executed:
  - interactively, with SOLoist Explorer
  - programmatically, via the API


SOLoist Explorer
soloistdemo.org/CompanyOrganization/oql?q=SELECT+d%2C+d.name%2C+d.responsibility%2C+d.superDept%2C+d.members+e%2C+e.name%2C+e.gender%...
SOLoist™ Explorer
Execute

Success.

```
SELECT d, d.name, d.responsibility, d.superDept, d.members e, e.name, e.gender, e.dateOfBirth
FROM Department d, d.members
WHERe d.name like '%o%'
```

Result format: [✓ HTML] XLS

Result (17 rows):

d	d.name	d.responsibility	d.superDept	e	e.name	e.gender	e.dateOfBirth
Department [2396850235766378051]	SOL		(null)	Employee [2423806914513500128]	New employee	MALE	(null)
Department [2409259353215495390]	Info Tech	Manage ord product	Department [2396850235766378051]	Employee [2423830246709302299]	Nemas ti pojma prijatelju	MALE	(null)
Department [2409259353215495390]	Info Tech	Manage ord product	Department [2396850235766378051]	Employee [2415497601635957875]	New employee	MALE	(null)
Department [2409259353215495390]	Info Tech	Manage ord product	Department [2396850235766378051]	Employee [2424126558390157403]	New employee	MALE	(null)
Department [2409259353215495390]	Info Tech	Manage ord product	Department [2396850235766378051]	Employee [2421440172438219824]	New employee	MALE	(null)
Department [2401131491429569029]	Production		Department [2396850235766378051]	Employee [2420259074890740584]	Andrija Misevic	MALE	(null)
Department [2401131491429569029]	Production		Department [2396850235766378051]	Employee [2425135406909946661]	Paul	MALE	2012-11-16
Department [2401131491429569029]	Production		Department [2396850235766378051]	Employee [2425244214637235813]	New employee	MALE	(null)
Department [2402038302380670079]	Commerce		Department [2396850235766378051]	Employee [2426451932098796100]	Sanja Mitrovic	FEMALE	(null)
Department [2401442486454578129]	Computer Science		Department [2401131491429569029]	Employee [2431065556538757781]	New employee	MALE	(null)
Department [2401442486454578129]	Computer Science		Department [2401131491429569029]	Employee [2428400728708140063]	Test	MALE	(null)

```
String oql =  "SELECT d, d.name, e, e.name, e.dateOfBirth " +
              "FROM Department d, d.members e " +
              "WHERE d.name like #name#";

BasicParameterValueStore param =
              new BasicParameterValueStore();
param.setParameterValue("name", Text.fromString(nameValue));

List<ITuple> results = Queries.executeOQL(oql,param).asList();
```

QUERYING API

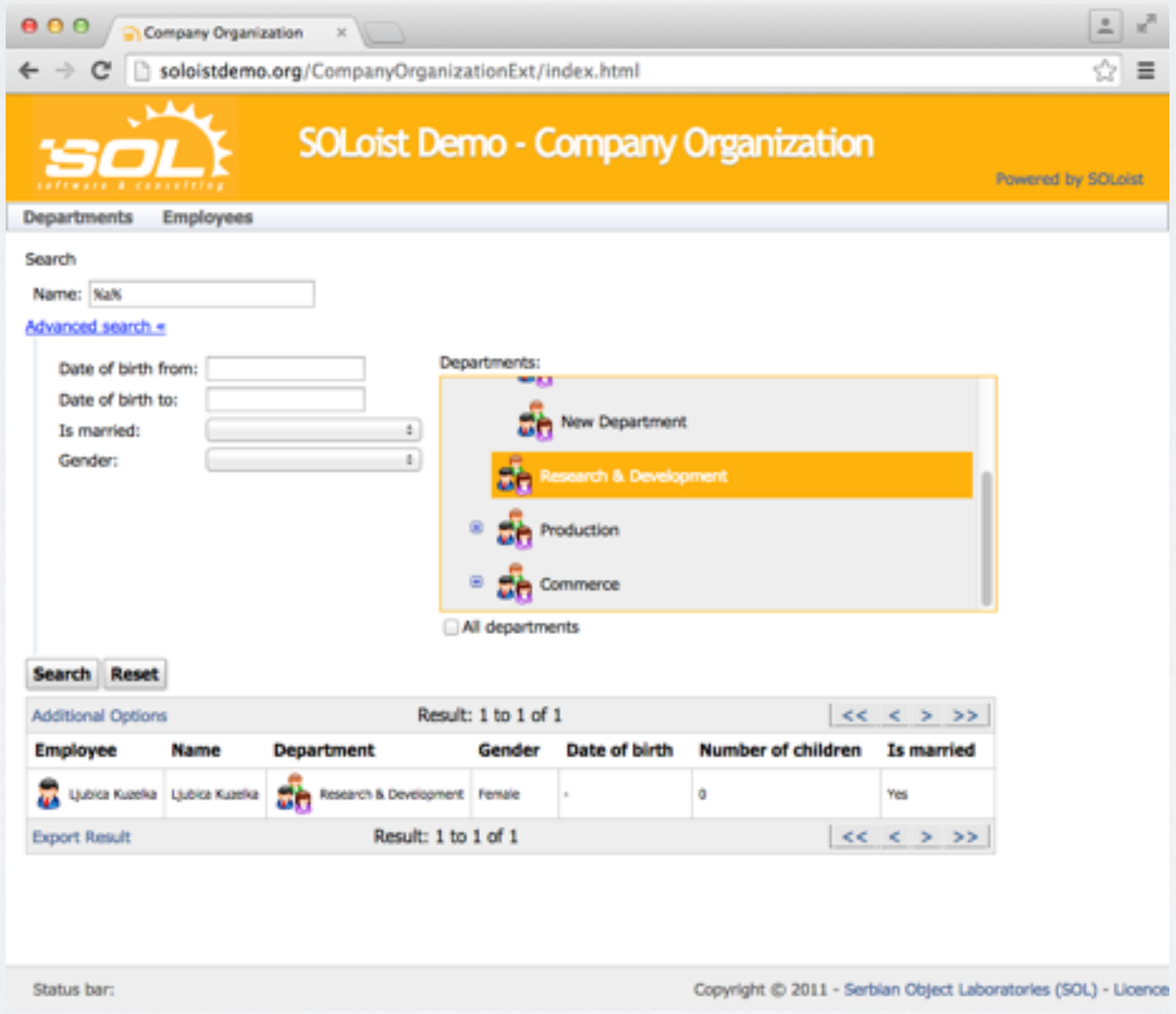
- For fetching of individual objects of a single class programmatically, there is a simple API:

```
List<Department> depts =
    return QueryUtils.getAllInstances(Department.CLASSIFIER);

List<Employee> employees =
    QueryUtils.findInstancesBy(Employee.CLASSIFIER,
    Employee.PROPERTIES.isMarried, Boolean.TRUE,
    fetchOffset, fetchSize);
    // to fetch all: fetchOffset=0, fetchSize=0

Employee employee =
    QueryUtils.findSingleInstanceBy(Employee.CLASSIFIER,
    Employee.PROPERTIES.name, Text.fromString("John Doe"), bMustExist);
```

- For more complex queries, use QueryBuilder

QUERY BUILDER

- QueryBuilder is an API for building queries programmatically, using their internal representation
- Can support everything as OQL, but has a bigger expressive power
- Used mostly in the implementation of interactive searches



Company Organization

soloistdemo.org/CompanyOrganizationExt/index.html

SOLoist Demo - Company Organization

Powered by SOLoist

Departments Employees

Search

Name: %a%

Advanced search «

Date of birth from:

Date of birth to:

Is married:

Gender:

Departments:

New Department

Research & Development

Production

Commerce

All departments

Search Reset

Additional Options Result: 1 to 1 of 1 << < > >>

Employee	Name	Department	Gender	Date of birth	Number of children	Is married
Ljubica Kuzelka	Ljubica Kuzelka	Research & Development	Female	-	0	Yes

Export Result Result: 1 to 1 of 1 << < > >>

Status bar:

Copyright © 2011 - Serbian Object Laboratories (SOL) - Licence

QUERY BUILDER

```
package companyorganization;

import java.util.Arrays;

import rs.sol.soloist.server.guiconfiguration.components.GUISearchResultComponent.Parameter;
import rs.sol.soloist.server.guiconfiguration.components.GUISearchResultComponent.Result;
import rs.sol.soloist.server.uml.queries.AssocEndTerm;
import rs.sol.soloist.server.uml.queries.AttributeTerm;
import rs.sol.soloist.server.uml.queries.ObjectTerm;
import rs.sol.soloist.server.uml.queries.SimpleQueryDefinition;
import rs.sol.soloist.server.uml.queries.builder.QueryBuilder;

public class EmployeesQueryBuilder extends QueryBuilder {

     private static final SimpleQueryDefinition prototypeQuery;

     @Parameter@Result("${Name}")
     public static final String NAME = "NAME";

     @Parameter@Result("${Department}")
     public static final String DEPARTMENT_NAME = "DEPARTMENT_NAME";

     @Parameter@Result("${Department}")
     public static final String DEPARTMENT = "DEPARTMENT";
     @Result("${}") @Parameter
     public static final String DEPARTMENT_ALL = “DEPARTMENT_ALL";
     …
```

QUERY BUILDER

```
…
@Parameter@Result("${Gender}")
public static final String GENDER = "GENDER";

@Result("${Date of birth}")
public static final String DATE_OF_BIRTH = "DATE_OF_BIRTH";

@Parameter
public static final String DATE_OF_BIRTH_FROM = "DATE_OF_BIRTH_FROM";

@Parameter
public static final String DATE_OF_BIRTH_TO = "DATE_OF_BIRTH_TO";

@Parameter@Result("${Number of children}")
public static final String NUMBER_OF_CHILDREN = "NUMBER_OF_CHILDREN";

@Parameter@Result("${Is married}")
public static final String IS_MARRIED = "IS_MARRIED";

@Result("${Employee}")
public static final String EMPLOYEE = "EMPLOYEE";

…
```

QUERY BUILDER

```
…
    protected EmployeesQueryBuilder() {
        super(prototypeQuery);
        contributions.addAll(Arrays.asList(
            new ContributePrefixMatch(NAME),
            new ContributePrefixMatch(DEPARTMENT_NAME),
            new ContributeEqual(IS_MARRIED),
            new ContributeGreaterOrEqual(DATE_OF_BIRTH_FROM, DATE_OF_BIRTH),
            new ContributeLessOrEqual(DATE_OF_BIRTH_TO, DATE_OF_BIRTH),
            new ContributeEqual(NUMBER_OF_CHILDREN),
            new ContributeEqual(GENDER),
            new ContributeIn(DEPARTMENT)
        ));

        includeInResultOnce(
            EMPLOYEE,
            NAME,
            DEPARTMENT,
            GENDER,
            DATE_OF_BIRTH,
            NUMBER_OF_CHILDREN,
            IS_MARRIED
        );
    }
…
```

QUERY BUILDER

```
    …
    @Override
    public SimpleQueryDefinition buildCountQuery() {
          require(EMPLOYEE);
          return super.buildCountQuery();
    }

    @Override
    public SimpleQueryDefinition buildQuery() {
          require(EMPLOYEE);
          return super.buildQuery();
    }

    static {
          ObjectTerm employee = new ObjectTerm(Employee.FQ_TYPE_NAME).as(EMPLOYEE);
          new AttributeTerm(employee, Employee.PROPERTIES.name).as(NAME);
          new AttributeTerm(employee, Employee.PROPERTIES.gender).as(GENDER);
          new AttributeTerm(employee, Employee.PROPERTIES.dateOfBirth).as(DATE_OF_BIRTH);
          new AttributeTerm(employee, Employee.PROPERTIES.numberOfChildren).as(NUMBER_OF_CHILDREN);
          new AttributeTerm(employee, Employee.PROPERTIES.isMarried).as(IS_MARRIED);

          AssocEndTerm dept = new AssocEndTerm(employee, Employee.PROPERTIES.dept).as(DEPARTMENT);
          new AttributeTerm(dept, Department.PROPERTIES.name).as(DEPARTMENT_NAME);
          prototypeQuery = new SimpleQueryDefinition(employee)
                .from(employee, dept)
                .nullable(dept);
    }
}
```

CHAPTER V
DATABASE FEATURES IN SOLOIST

Optimizations, Performance, Scalability

DATABASE FEATURES

- Database schema automatic update: SOLoist checks the version of the schema in DB and does the necessary upgrade through all intermediate versions
- Model evolution problem in production: how to keep the production data in DB when updating the schema on model change
- SOLoist utility for model evolution: does the diff-ing of two UML (class) model versions (with the user's confirmations or interventions) and generates an upgrade (`ALTER TABLE`) script
- See details in:

  Milovanović, V., Milićev, D., "An Interactive Tool for UML Class Model Evolution in Database Applications," *Software and Systems Modeling*, September 2013

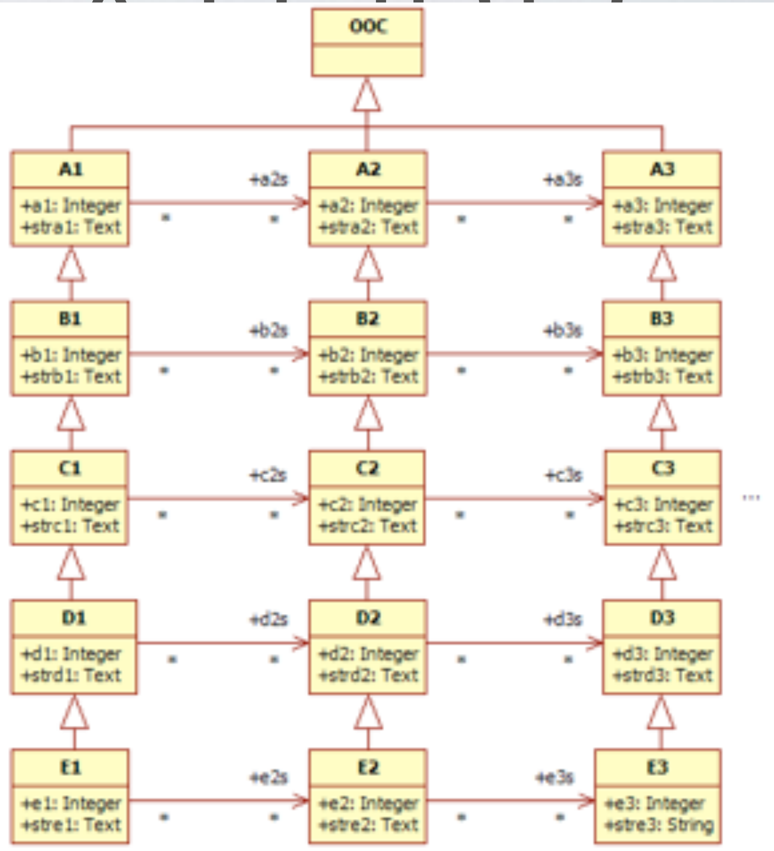
DATABASE FEATURES

- SOLoist is constructed and prepared for high volume and scalability, in terms of
  - data volume (hundreds of millions of objects)
  - throughput (hundreds of concurrent users)
- There are many clever optimizations in queries posed to the database, to significantly reduce query execution time for huge volumes of data, such as:
  - redundant copy of an inherited attribute value in each table of a derived class to accelerate read queries in case of rare writes (configure attribute's tagged value `AttrMappingStrategy`)
  - avoiding unnecessary joins of tables

DATABASE

Example:

SOLoist vs. Hibernate performing very complex queries spanning multiple many-to-many associations and deep hierarchies:



Hibernate HQL

```
select count(*)
from E1 x1 join x1.e2s x2 join x2.e3s x3
where x1.e1 > 0 and x3.e3 > 0 and x2.e2 > 0
```

SQL

```
select count(*) as col_0_0_
from OOC e1x0_
left outer join A1 e1x0_1_ on e1x0_.id=e1x0_1_.id
left outer join B1 e1x0_2_ on e1x0_.id=e1x0_2_.id
left outer join C1 e1x0_3_ on e1x0_.id=e1x0_3_.id
left outer join D1 e1x0_4_ on e1x0_.id=e1x0_4_.id
left outer join E1 e1x0_5_ on e1x0_.id=e1x0_5_.id
inner join E1#e2s_E2 e2s1_ on e1x0_.id=e2s1_.e1_ID
inner join OOC e2x2_ on e2s1_.e2s_id=e2x2_.id
left outer join A2 e2x2_1_ on e2x2_.id=e2x2_1_.id
left outer join B2 e2x2_2_ on e2x2_.id=e2x2_2_.id
left outer join C2 e2x2_3_ on e2x2_.id=e2x2_3_.id
left outer join D2 e2x2_4_ on e2x2_.id=e2x2_4_.id
left outer join E2 e2x2_5_ on e2x2_.id=e2x2_5_.id
inner join E2#e3s_E3 e3s3_ on e2x2_.id=e3s3_.e2_ID
inner join OOC e3x4_ on e3s3_.e3s_id=e3x4_.id
left outer join A3 e3x4_1_ on e3x4_.id=e3x4_1_.id
left outer join B3 e3x4_2_ on e3x4_.id=e3x4_2_.id
left outer join C3 e3x4_3_ on e3x4_.id=e3x4_3_.id
left outer join D3 e3x4_4_ on e3x4_.id=e3x4_4_.id
left outer join E3 e3x4_5_ on e3x4_.id=e3x4_5_.id
where  e1x0_.DTYPE='E1'
       and e1x0_5_.e1>0
       and e3x4_5_.e3>0
       and e2x2_5_.e2>0
```

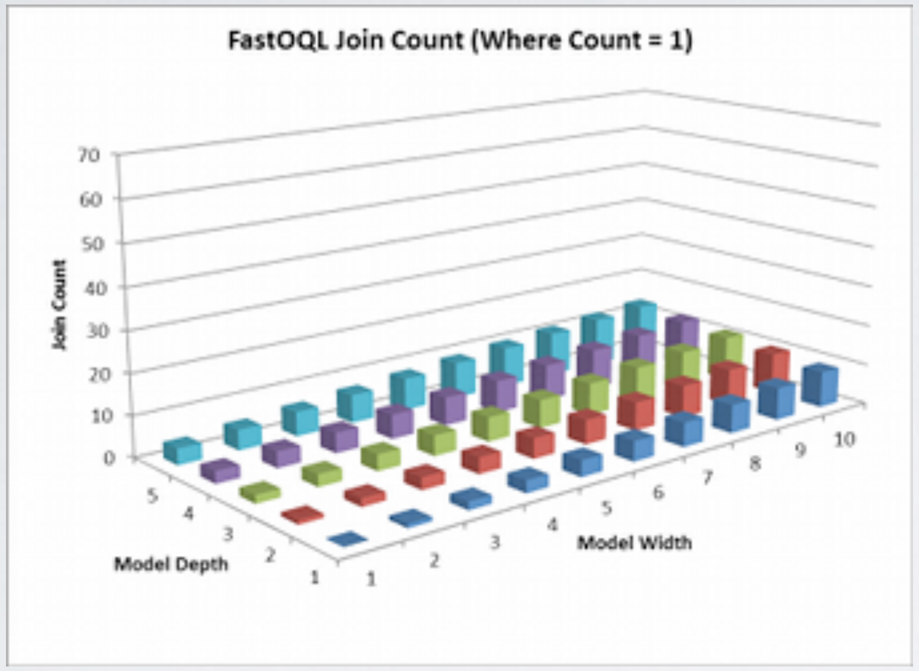
FastOQL

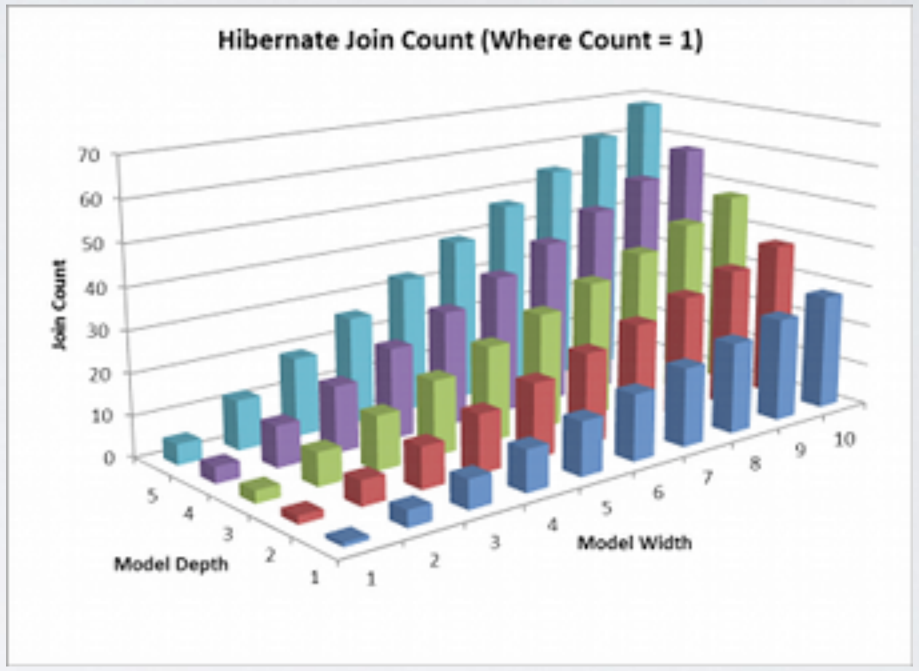
```
select count(*)
from E1 x1, x1.e2s x2, x2.e3s x3
where x1.e1 > 0 and x3.e3 > 0 and x2.e2 > 0
```

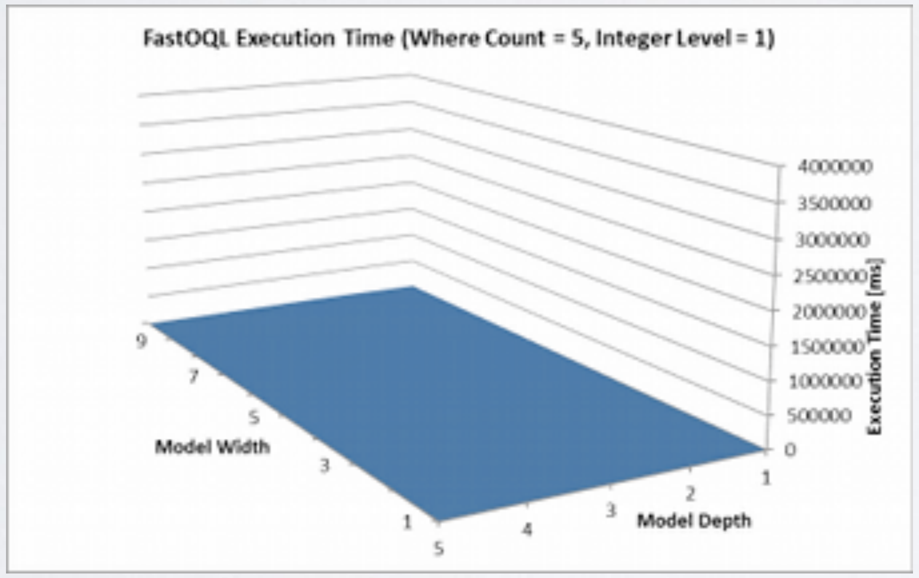
SQL

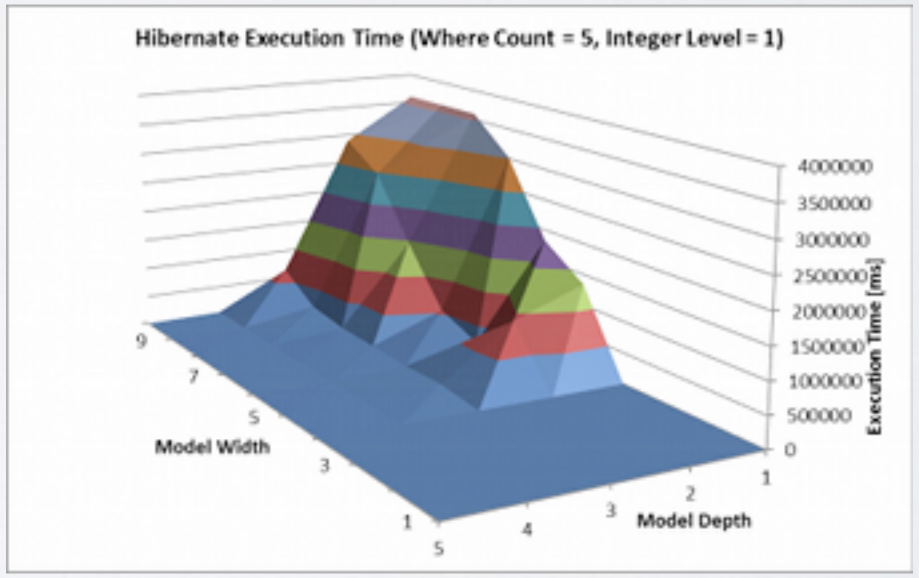
```
select count(*)
from E1 AS x1
join E1#e2s_E2 as x2 on x2.e1_id = x1.id
join E2#e3s_E3 as x3 on x3.e2_id = x2.e2s_id
join E3 as _attrs1_x3 on _attrs1_x3.id=x3.e3s_id
join E2 as _attrs2_x2 on _attrs2_x2.id=x2.e2s_id
where  x1.e1>0
       and _attrs1_x3.e3>0
       and _attrs2_x2.e2>0
```

DATABASE FEATURES

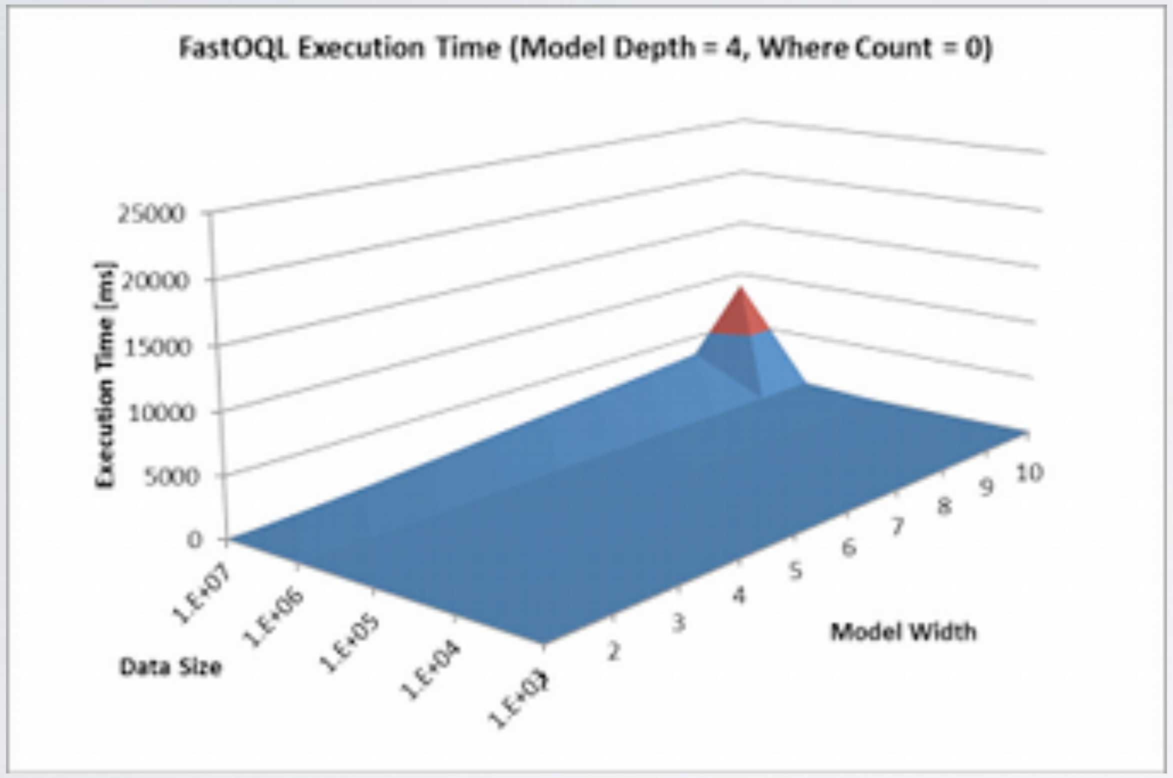


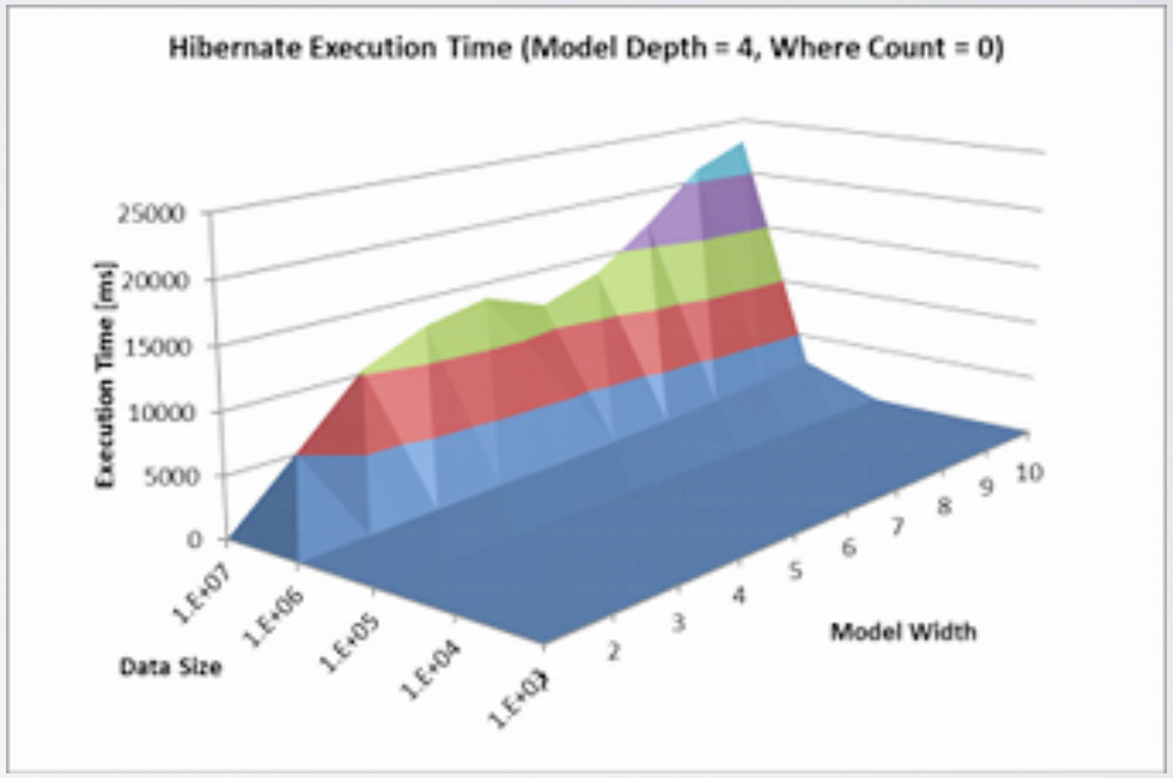






DATABASE FEATURES





CHAPTER VI MODELING BEHAVIOR IN SOLOIST

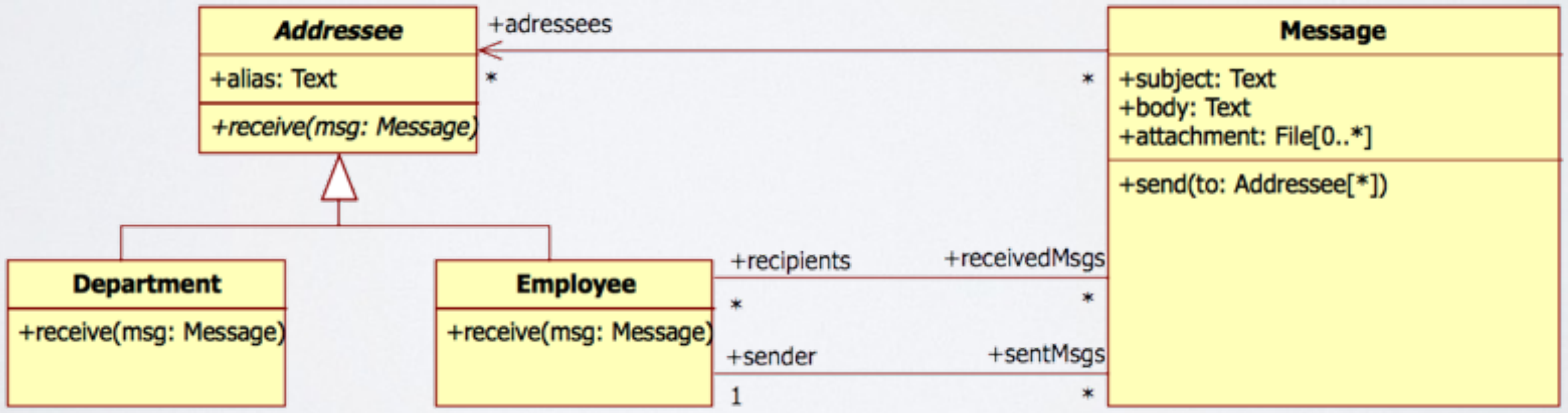
Operations/Methods, Commands, State Machines

OPERATIONS AND METHODS

- *Operation* is a feature of a class representing a specification of a service that can be requested from any instance of that class in order to activate an associated behavior
- An operation specifies that a service may be requested from any (direct or indirect) instance of that class
- At runtime, an operation of an object can be invoked. The actual arguments are then supplied. The invocation of the operation is manifested by the behavior specified as the implementation of the operation
- An implementation of an operation is called a *method*. A method is the specification of behavior that is activated when the operation of an object is invoked
- If a class does not attach a method to an operation of that class, the operation is called *abstract*

OPERATIONS AND METHODS

- In SOLoist, you model operations as usual in UML
- Supported features of operations:
    - arguments: can be of any type (a reference to a class, to a SOLoist data type, or any Java type)
    - return value: any type
    - abstract operations



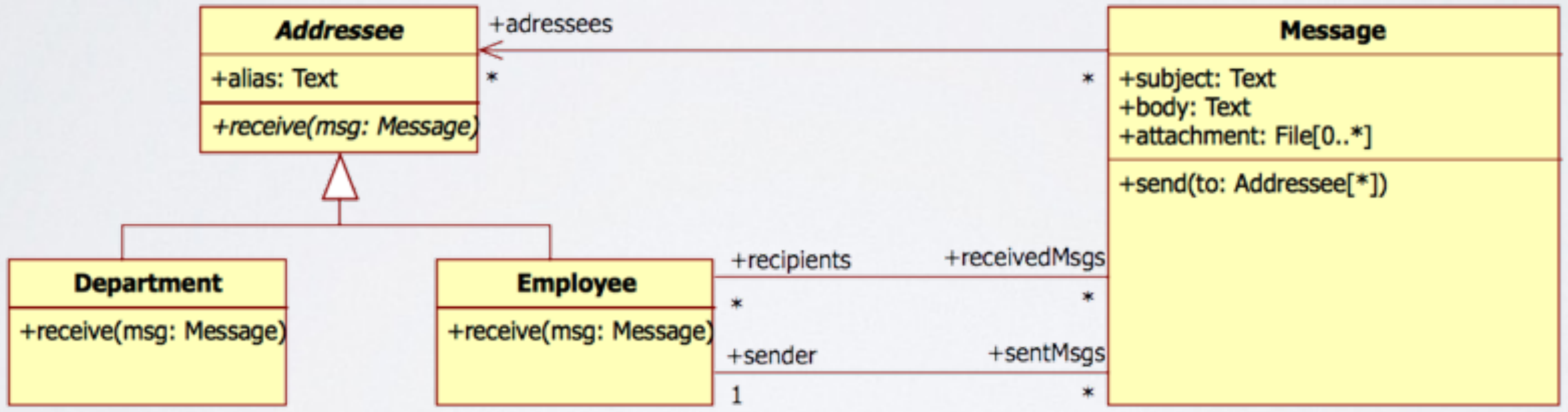
OPERATIONS AND METHODS

- In SOLoist, methods are written in "preserved sections" of Java methods generated for the class
- The contents of the "preserved sections" in the generated code are preserved when the code is regenerated — all other code will be overwritten when regenerated (do NOT change it manually)
- This ensures a strict development policy and maintenance - every artifact has its sole and unique "source form" (model and code) and is to be changed there.
- No reverse engineering, no round-trip engineering!

```
public int countEmployees()
// -------------<SOL id="06c35243-832e-4213-9195-9f7757a55651:___throw__" />
// -------------<LOS id="06c35243-832e-4213-9195-9f7757a55651:___throw__" />
{
    // ---------<SOL id="06c35243-832e-4213-9195-9f7757a55651:___body___" />
    int count = 0;
    for (Department d: this.subDept.read()) {
         count += d.countEmployees();
    }
    return count;
    // ---------<LOS id="06c35243-832e-4213-9195-9f7757a55651:___body___" />
}
```

OPERATIONS AND METHODS

- A method is written in Java (as the *host/detail/action* language), and can contain:
  - control of flow (if-then-else, loops, blocks)
  - UML actions, in terms of SOLoist API calls
  - operation calls (with the semantics of Java - compliant to UML; polymorphism assumed by default)
  - all other Java native code



<u>Method of operation in UML: Message::send(to:Addressee[*]):</u>

```
    for (Addressee addr : to) {
        this.addressees.add(to);
        to.receive(this);
    }
```

UML action “add a value to a slot” (creates a link)

UML action “call operation” (usual invocation)

OPERATIONS AND METHODS

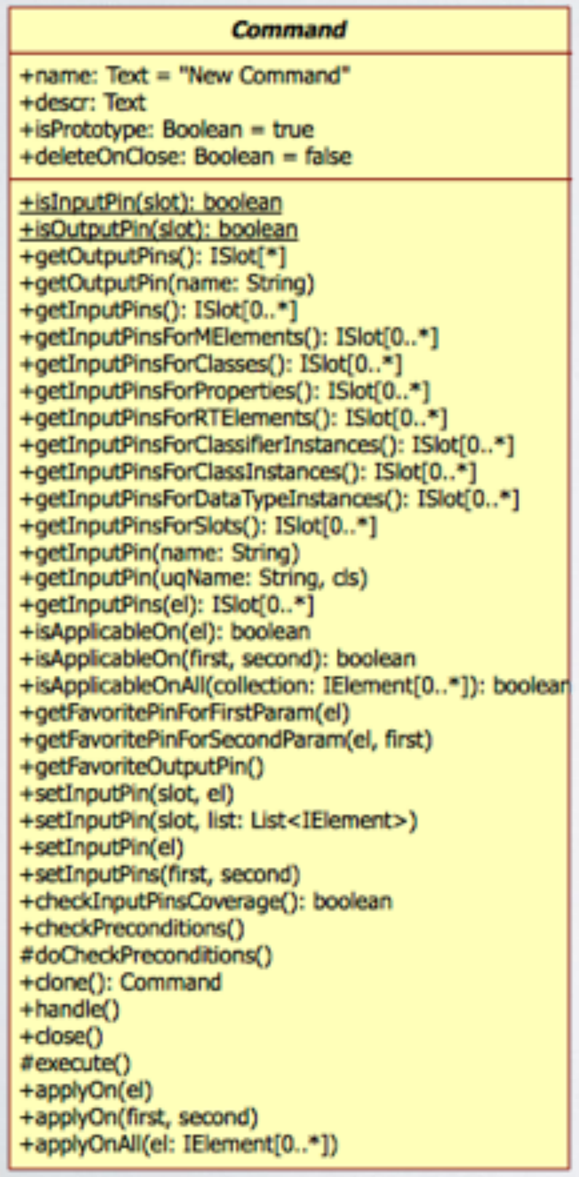
- Concurrency control:
    - optimistic concurrency control: exceptions raised on concurrency conflicts
    - the subject of control (detection of concurrency conflicts) are *objects*, not database records (another semantic discontinuity existing in other ORM frameworks that leave the concurrency control to the DBMS)
- Isolation and fault tolerance:
    - transaction mechanism implemented by SOLoist
    - the database transaction is issued when the SOLoist transaction is committed
    - actions issued to the database are optimized and compacted: only the net effects are sent to the RDBMS => smaller transactions => reduced RDBMS workload => better throughput
    - transactions can be nested: RDBMS transaction is issued on the topmost transaction's commit
    - UML actions are implicit transactions
    - requests from GUI are also performed transactionally

COMMANDS

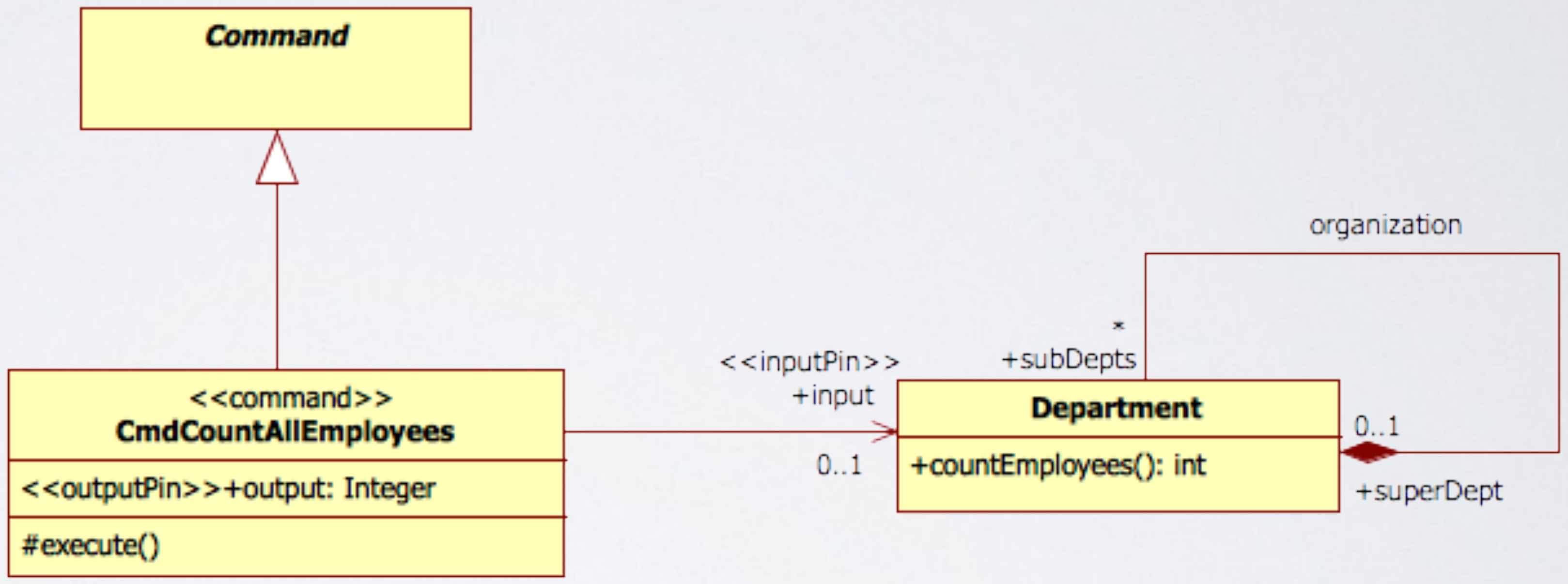
- *Command* is a (usual SOLoist UML) class whose objects represent requests for a service from the system, issued interactively from the GUI or another external system (*Command* design pattern)
- When a command is issued from the GUI, an instance of the command is created and its operation `execute` is ultimately invoked. The behavior of the command is specified within the method of this operation.
- A command can have its *input pins*, which are parameters whose values are provided by the GUI dynamically, at the time of command's activation.
- A command can have its *output pins*, which are results provided by the command's execution. These values can be used in the GUI dynamically, once the execution has completed.
- The elements used in the GUI to help the user understand the command are specified in its name and description attributes.
- A command can be applied on an object or a collection of objects.

COMMANDS

- Command is modeled as a class, stereotyped with `<<command>>` and derived from the built-in class Command (directly or indirectly).
- Input and output pins are modeled as properties (attributes or association ends) of the command, stereotyped with `<<inputPin>>` or `<<outputPin>>`



Command
+name: Text = "New Command"
+descr: Text
+isPrototype: Boolean = true
+deleteOnClose: Boolean = false
+isInputPin(slot): boolean
+isOutputPin(slot): boolean
+getOutputPins(): ISlot[*]
+getOutputPin(name: String)
+getInputPins(): ISlot[0..*]
+getInputPinsForMElements(): ISlot[0..*]
+getInputPinsForClasses(): ISlot[0..*]
+getInputPinsForProperties(): ISlot[0..*]
+getInputPinsForRTElements(): ISlot[0..*]
+getInputPinsForClassifierInstances(): ISlot[0..*]
+getInputPinsForClassInstances(): ISlot[0..*]
+getInputPinsForDataTypeInstances(): ISlot[0..*]
+getInputPinsForSlots(): ISlot[0..*]
+getInputPin(name: String)
+getInputPin(uqName: String, cls)
+getInputPins(el): ISlot[0..*]
+isApplicableOn(el): boolean
+isApplicableOn(first, second): boolean
+isApplicableOnAll(collection: IElement[0..*]): boolean
+getFavoritePinForFirstParam(el)
+getFavoritePinForSecondParam(el, first)
+getFavoriteOutputPin()
+setInputPin(slot, el)
+setInputPin(slot, list: List<IElement>)
+setInputPin(el)
+setInputPins(first, second)
+checkInputPinsCoverage(): boolean
+checkPreconditions()
#doCheckPreconditions()
+clone(): Command
+handle()
+close()
#execute()
+applyOn(el)
+applyOn(first, second)
+applyOnAll(el: IElement[0..*])



<u>Method of CmdCountAllEmployees::execute():</u>

```
protected void execute() {
    Department d = this.input.val();
    if (d != null) {
        int count = d.countEmployees();
        this.output.set(Integer.valueOf(count));
    }
}
```

COMMANDS

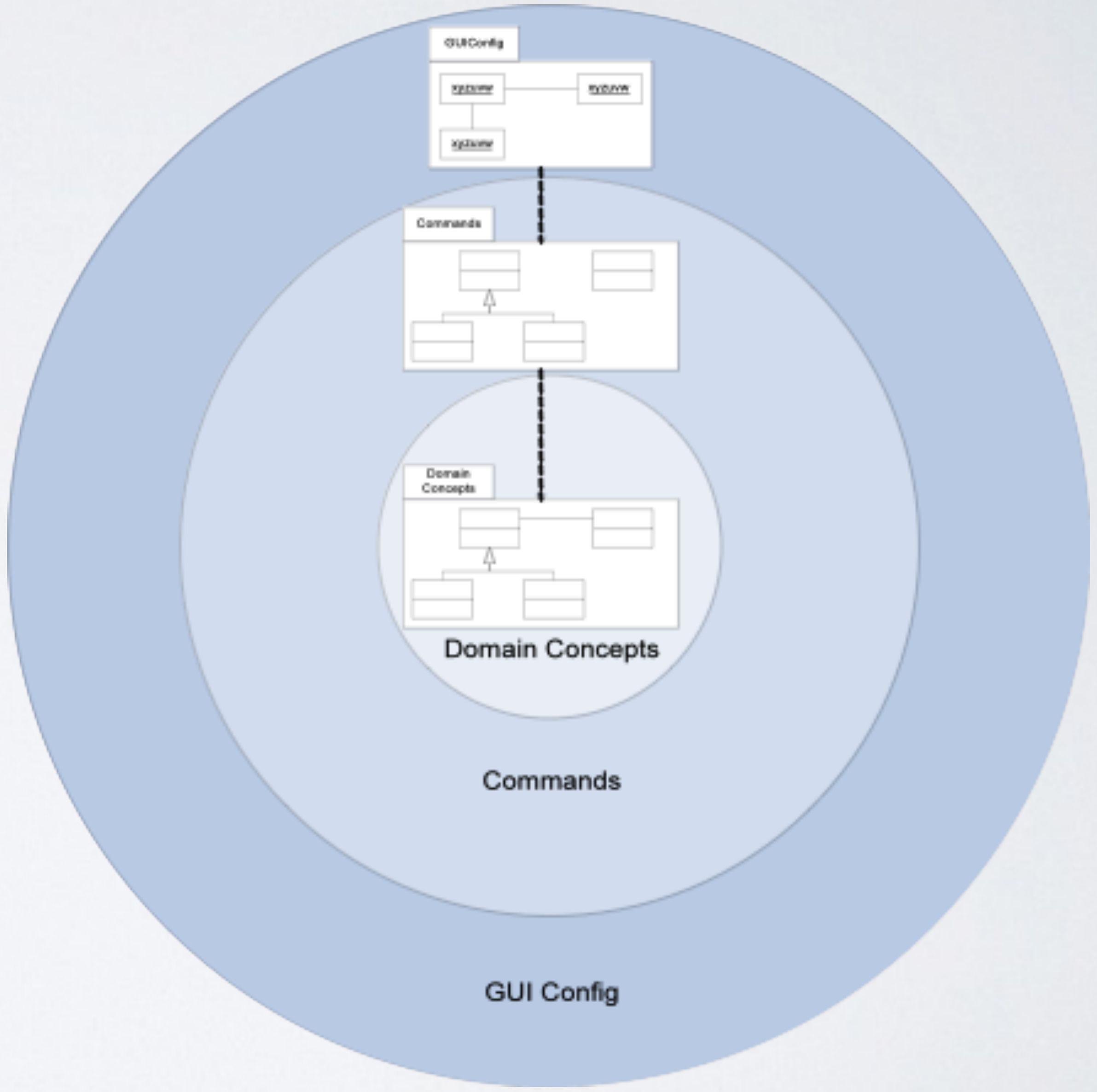
- *Generic* or *built-in* commands are commands that embody certain UML actions upon the object space. They are implicitly defined in the system and accessible from the application. They are part of the SOLoist UML model library.
- *Domain-specific* commands represent entry points to the implementation of the specific functionality of the system, not directly supported by the execution environment and generic commands. They are introduced into the model by the modeler.
- Generic commands in SOLoist:

```
CmdCreateObject
CmdCreateObjectOfClass
CmdDestroyObject
CmdCloneObject
CmdCopySlots
CmdReorderValuesInSlot
CmdMoveValueInSlot
CmdCopySlot
CmdClearSlot*
CmdAddValueToSlot*
CmdRemoveValueFromSlot*
CmdSetSlot*
CmdReadSlot*
CmdCreateObjectAndLinkToOne
CmdCreateObjectAndLinkToTwo
CmdCreateObjectAndLinkToObject
```

* Currently not available, or available in separate forms for attributes and association ends, but will be revised in future versions.

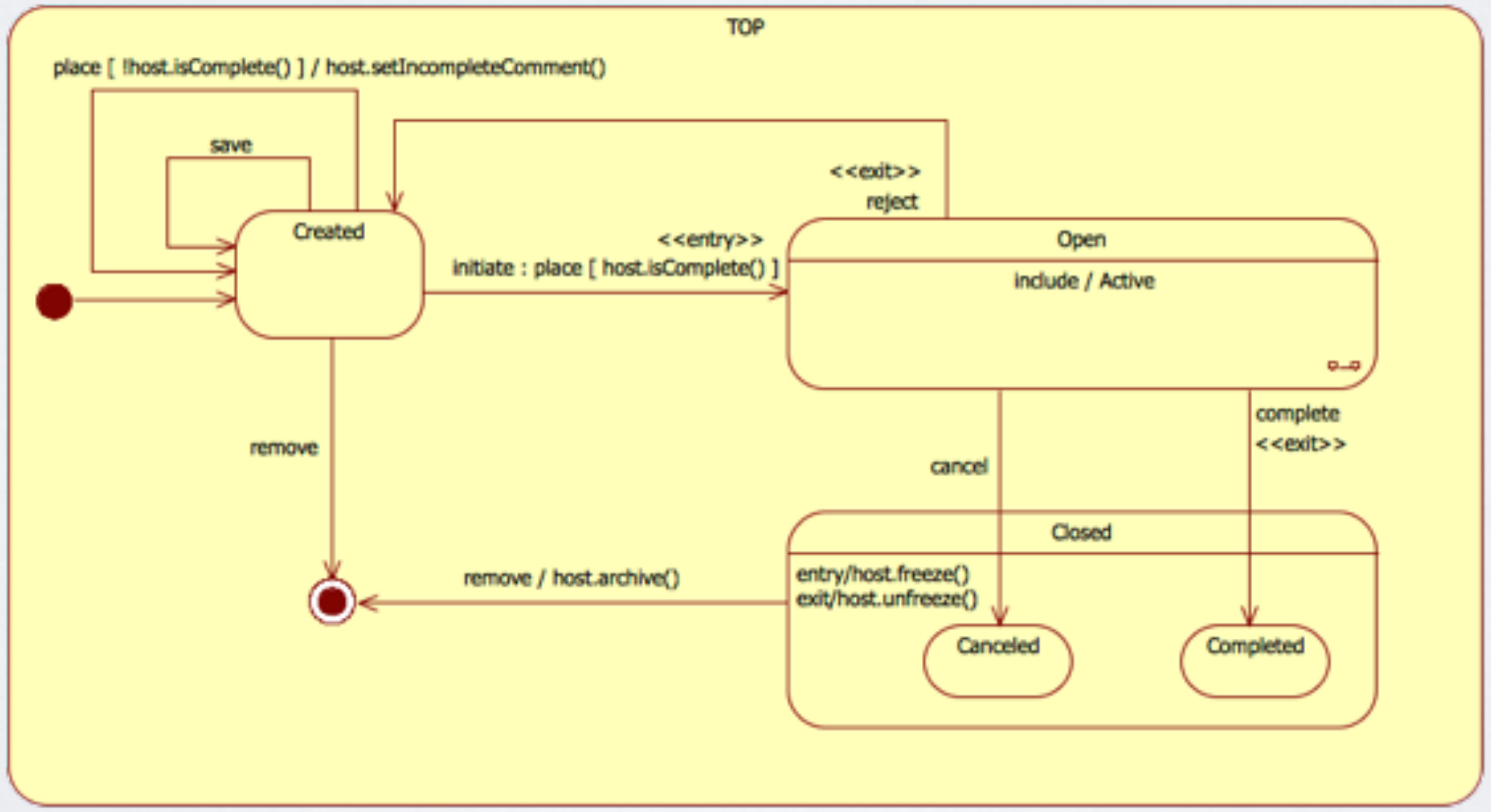
COMMANDS

- Advantages of using commands (Command DP):
  - encapsulation of business logic
  - clear architecture
  - generic handling of user's requests
  - dynamic configuration
  - logging
  - authorization (access rights)
  - encapsulation of common maintenance:
  - transaction mechanism
  - locking
  - parameters processing
  - exception handling



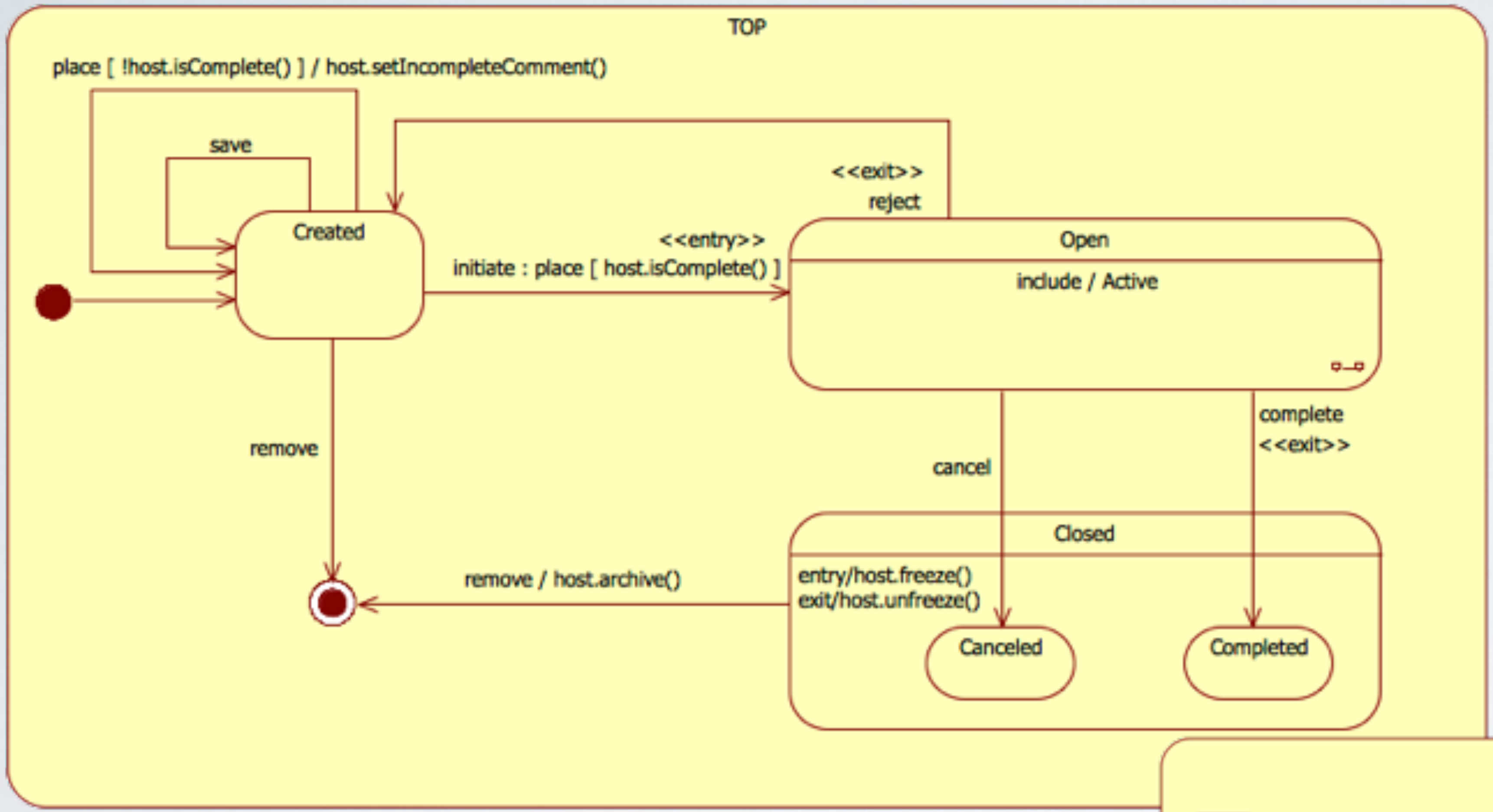
STATE MACHINES

- *State machines* are often very useful for modeling event-driven behavior of entities whose reaction on an incoming stimulus depends not only on the kind of that stimulus, but also on the history of the previously received stimuli, i.e., on the current *state* of the receiver.
- In essence, they model the lifetime of the entity in terms of *states* and *transitions*.
- SOLoist supports hierarchical UML state machines, with nested states and submachines.

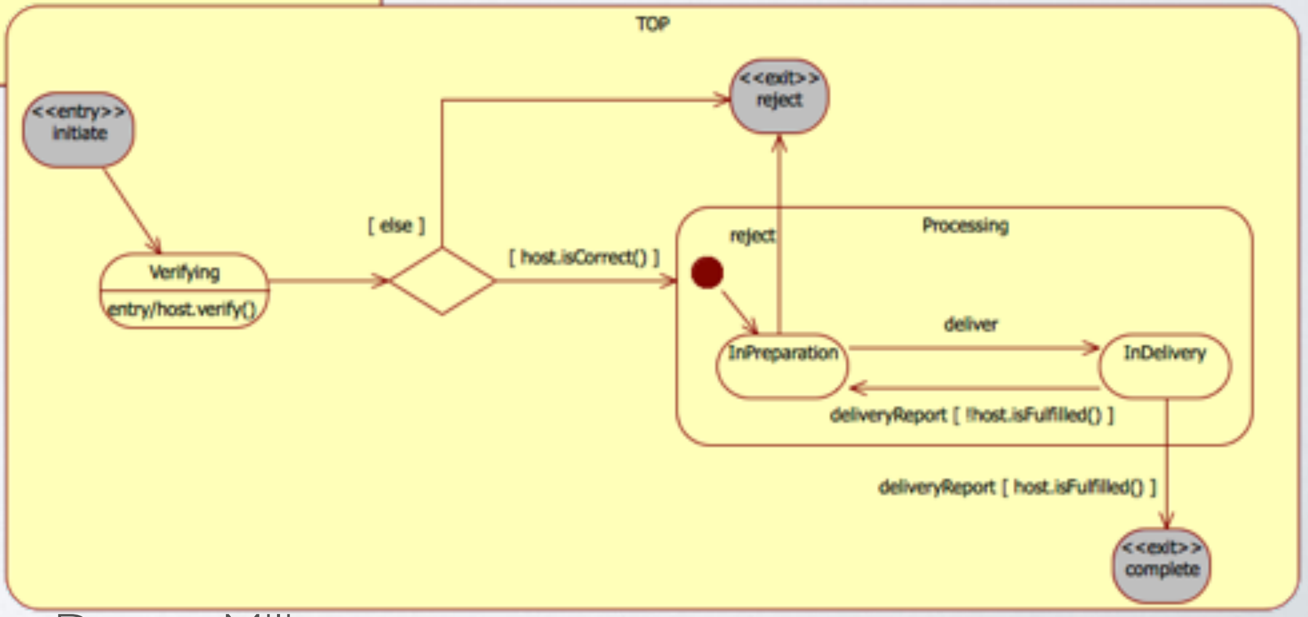


STATE MACHINES

Main state machine diagram for order processing (lifecycle of objects of class `Order`):



Included sub-machine diagram (`Active`):

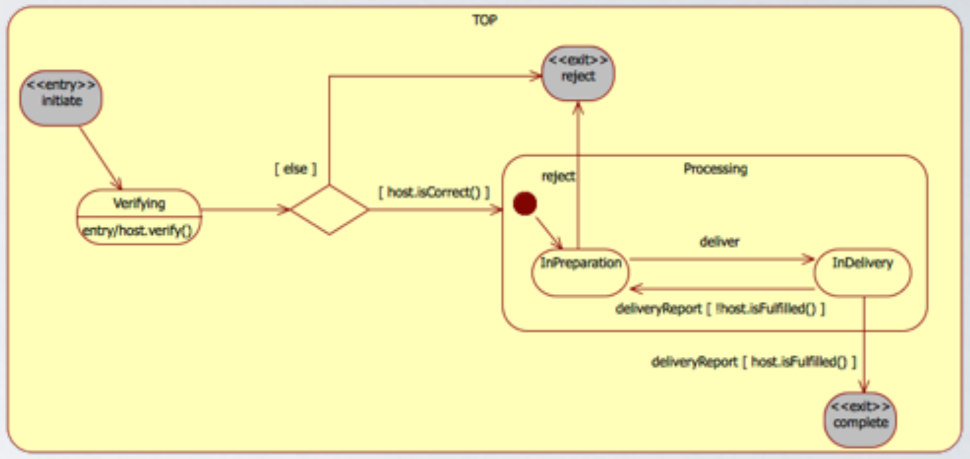


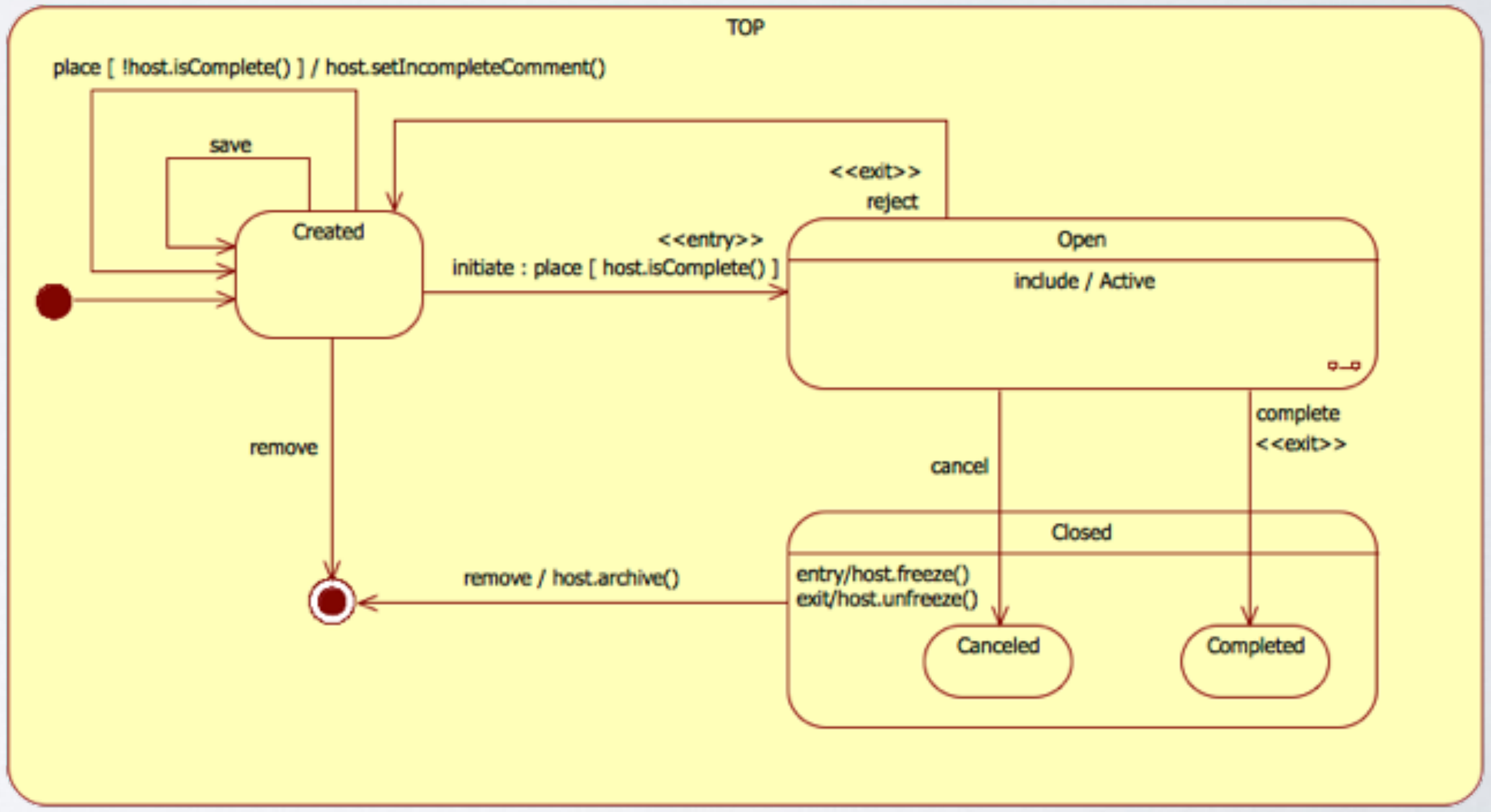
STATE MACHINES

Supported concepts:

- Expressions used in guards and actions: `host` refers to the host object
- Transitions, with triggers, guards, and actions: `trigger[guard]/action`
- Triggerless transitions: enabled as soon as the source state completes
- Primitive states
- Composite states
- Initial pseudo states
- Choice pseudo states (optional `else` branch)
- Final states
- Submachines
- Entry and exit points
- Exit from state on timeout

Not supported: history, concurrency (fork/join/regions)





STATE MACHINES

- An SM is normally modeled as a behavioral feature attached to a class called the *host class*.
- The SM then describes the lifecycle of objects of that host class.
- At runtime, each object of the host class, called the *host* object, can change its current state according to the definition of the SM.
- If an SM is not associated with a class in the model, and submachines are typically stand-alone modeling elements, the host class of the SM or submachine is specified through the `HostClass` tagged value available in the SOLoist profile.
- The behavior specified with an SM is ensured by the Java code generated for the SM (the "SM class").
- The SM class is a stateless pure Java class, while the state has to be embodied in the host class (object) (*Strategy* design patter with a stateless Strategy, the SM class playing the role of the Strategy).
- In order to implement the necessary behavior, methods of the SM class call back methods of the host class to:
    - read the current state of the host object,
    - execute guard conditions and actions on transitions,
    - change the current state of the host object,
    - perform other auxiliary processing, such as optional locking, logging, setting the timeout, etc.
- To trigger the state machine and process it, call the operation of the SM class:

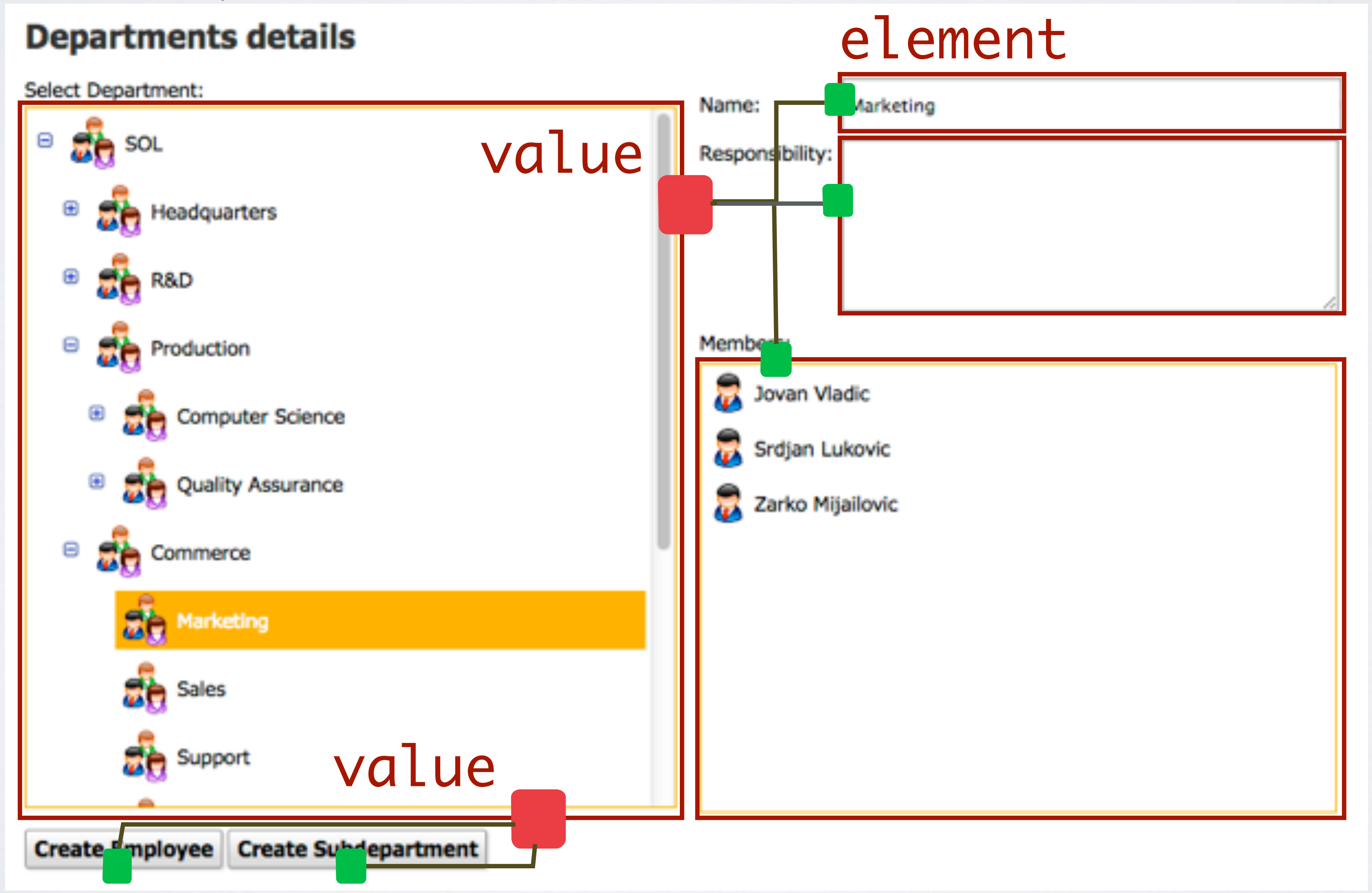
```
boolean process (HostClass hostObject, String triggerName);
```

CHAPTER VII
WEB GUI DEVELOPMENT IN SOLoist

Concepts, Principles, Library

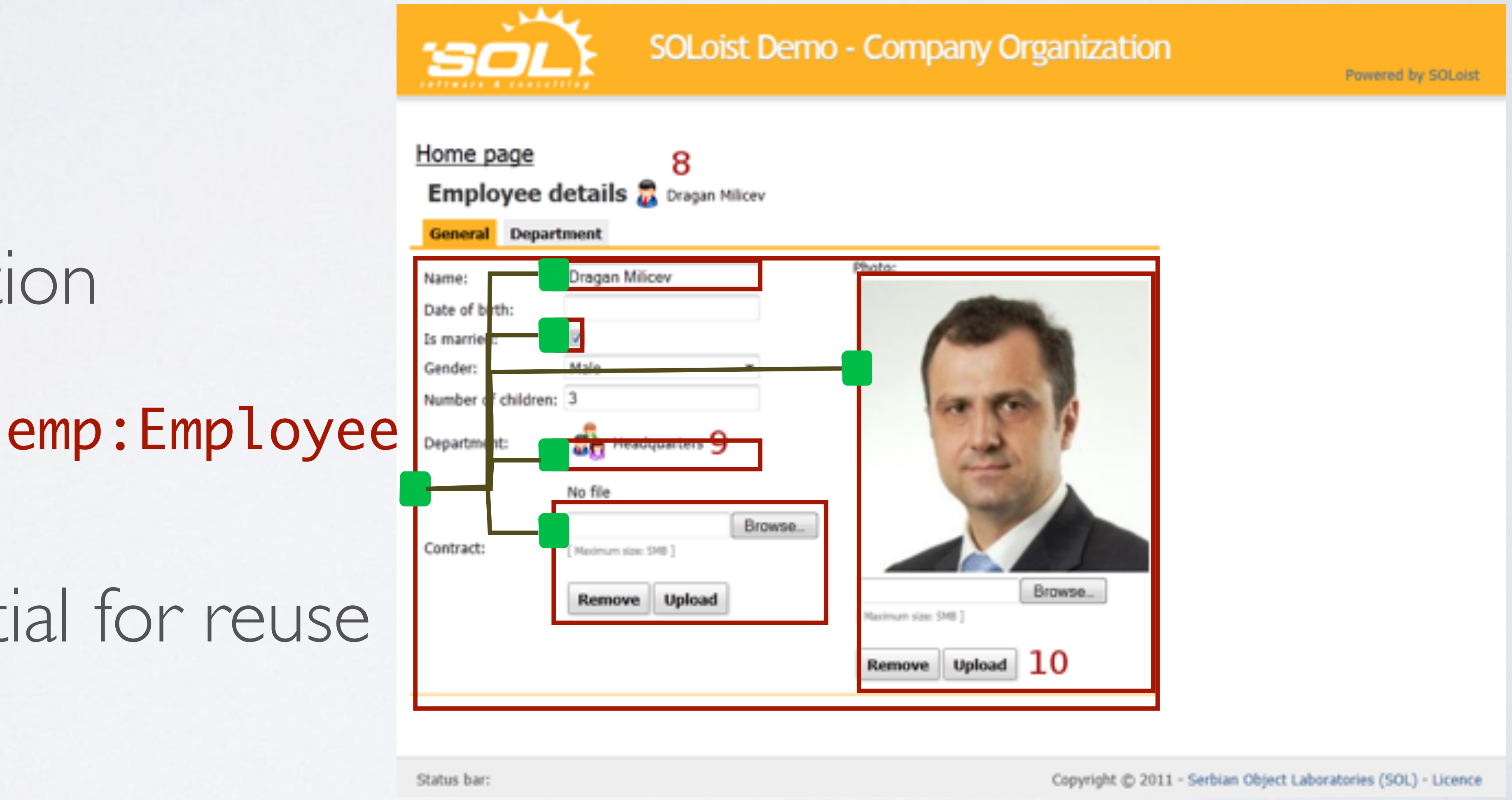
CONCEPTS

- GUI components: ready-to use building blocks from the SOLoist library, configurable and fully coupled with the UML object space
- *Input* and *output pins*: "ports" of components for receiving and sending messages
- *Wires*: connectors between ports
- Data flow



CONCEPTS

- *Capsule*: a profiled structured class that models a simple UI component or a coherent UI fragment of logically and functionally coupled components or other fragments with a clear interface.
- Capsule ensures:
  - proper abstraction
  - encapsulation
  - inherent potential for reuse



Employee details Dragan Milicev

Name: Dragan Milicev
Date of birth:
Is married: ✓
Gender: Male
Number of children: 3
Department: Headquarters
Contract: No file [Maximum size: 5MB]
Photo: [Maximum size: 5MB]

Copyright © 2011 - Serbian Object Laboratories (SOL) - Licence

CONCEPTS

- In the current version of SOLoist, a capsule is defined as a pure Java class in code tha extends the built-in class `Capsule`:

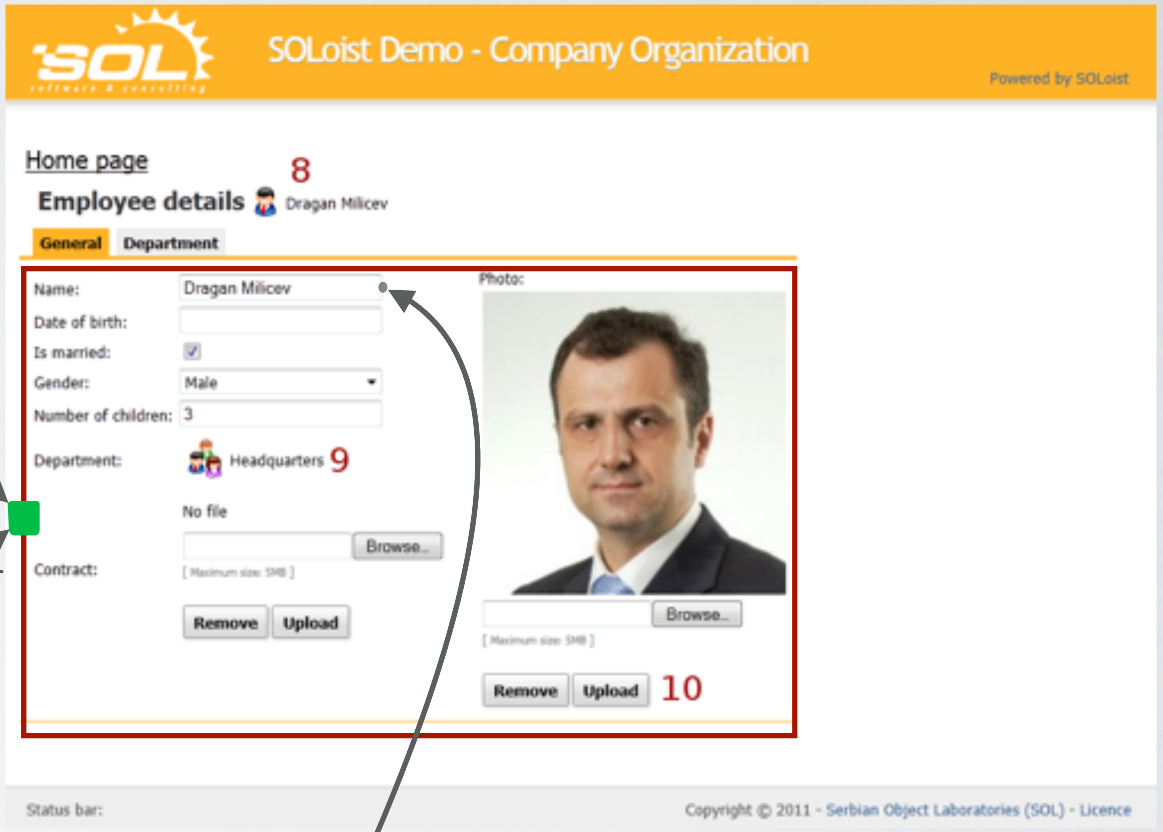
```
public abstract class Capsule {

     protected GUIContainerComponent root;

     protected Capsule(GUIContainerComponent parent) {
          this(parent, false);
     }

     protected Capsule(GUIContainerComponent parent, boolean deferred) {
          root = createRoot();
          if (parent != null)
               parent.add(root);

          if (!deferred)
               build();
     }

     protected Capsule(Capsule parent) {
          this(parent, false);
     }
     …
```

CONCEPTS

```
    …
    protected Capsule(Capsule parent, boolean deferred) {
          root = createRoot();
          if (parent != null)
                parent.root.add(root);

          if (!deferred)
                build();
    }

    protected GUIContainerComponent createRoot(){
          return GUIPanelComponent.createFlow(null);
    }

    public abstract void build();
}
```

CONCEPTS

```
public class EmployeeDetailsGeneral extends Capsule {

     private GUIRelayComponent employeeRelay;

     public ISlot ipEmployee(){
          return employeeRelay.ipRelay();
     }

     public EmployeeDetailsGeneral(GUIContainerComponent parent) {
          super(parent);
     }

     @Override
     public void build() {
          employeeRelay = GUIRelayComponent.create(root);

          GUIPanelComponent wrapPanelGeneral = GUIPanelComponent.createHorizontal(root);

          GUIPanelComponent panelGeneral = GUIPanelComponent.createTable(wrapPanelGeneral);
          panelGeneral.setStyle("tablePanels");

          GUILabelComponent.create(panelGeneral, "Name: ", 0, 0);
          GUIEdit employeeNameSlotComponent = GUIEdit.createField(panelGeneral, Employee.PROPERTIES.name, 0, 1);
     …
```


SOLoist Demo - Company Organization

Powered by SOLoist

Home page 8

Employee details Dragan Milicev

General | Department

Name:	Dragan Milicev
Date of birth:	
Is married:	✓
Gender:	Male
Number of children:	3
Department:	Headquarters 9
	No file
Contract:	Browse... [Maximum size: 5MB]
	Remove Upload

Photo:

Browse... [Maximum size: 5MB]

Remove Upload 10

Status bar:

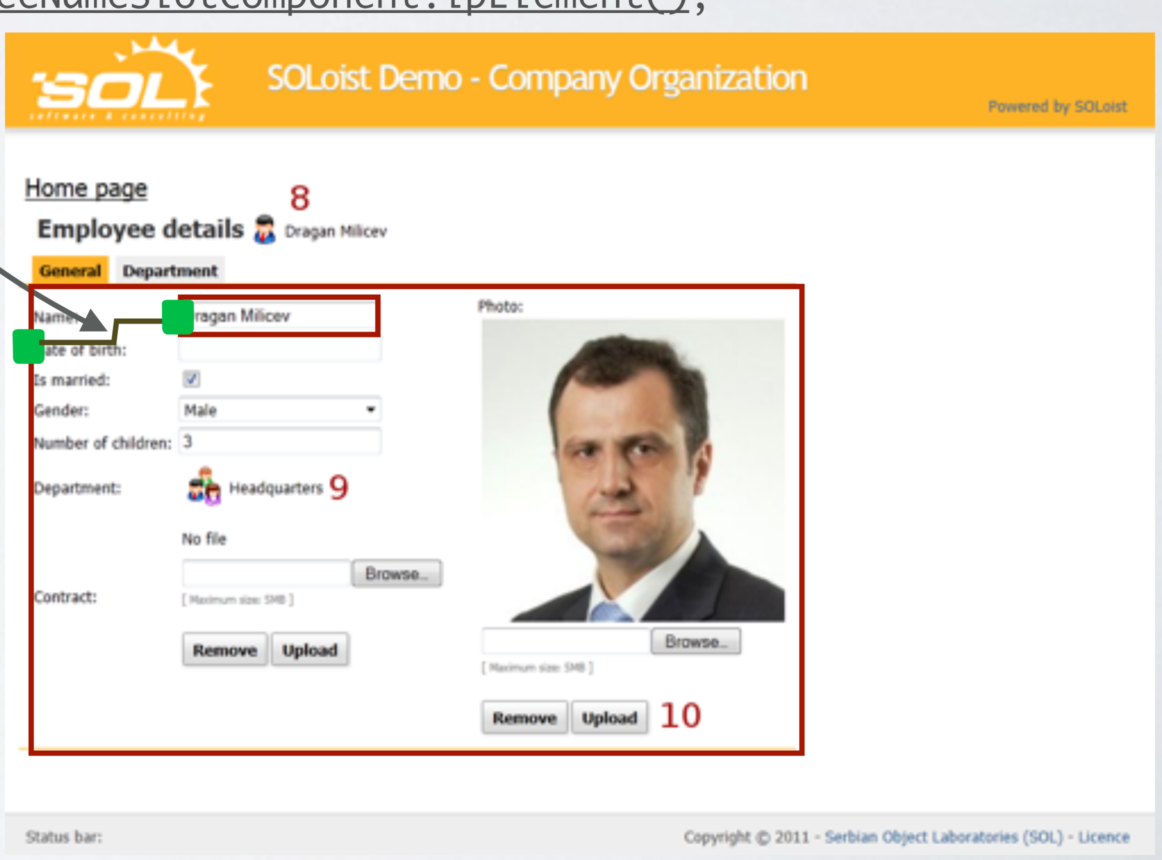
Copyright © 2011 - Serbian Object Laboratories (SOL) - Licence

CONCEPTS

```
…
     GUILabelComponent.create(panelGeneral, "Date of birth: ", 1, 0);
     GUIEdit dateSlotComponent = GUIEdit.createField(panelGeneral, Employee.PROPERTIES.dateOfBirth, 1, 1);
     dateSlotComponent.setSize("167px", null);

     GUILabelComponent.create(panelGeneral, "Is married: ", 2, 0);
     GUIEdit isMarriedSlotComponent = GUIEdit.createField(panelGeneral, Employee.PROPERTIES.isMarried, 2, 1);

     GUILabelComponent.create(panelGeneral, "Gender: ", 3, 0);
     GUIEdit genderSlotComponent = GUIEdit.createField(panelGeneral, Employee.PROPERTIES.gender, 3, 1);

     GUILabelComponent.create(panelGeneral, "Number of children: ", 4, 0);
     GUIEdit numOfChSlotComponent = GUIEdit.createField(panelGeneral, Employee.PROPERTIES.numberOfChildren, 4,
1);

     GUILabelComponent.create(panelGeneral, "Department: ", 5, 0);
     GUIInput departmentGeneralList = GUIInput.createList(panelGeneral, Employee.PROPERTIES.dept);
     departmentGeneralList.setLayoutData(TableLayoutData.create(5, 1));

     GUILabelComponent.create(panelGeneral, "Contract: ", 6, 0);
     GUIPanelComponent panelFile = GUIEdit.createFile(panelGeneral, Employee.PROPERTIES.contract, true, true);
     panelFile.setLayoutData(TableLayoutData.create(6, 1));
…
```

CONCEPTS

```
…
     panelFile.setLayoutData(TableLayoutData.create(6, 1));

     GUIPanelComponent wrapPicturePanel = GUIPanelComponent.createVertical(wrapPanelGeneral);
     wrapPicturePanel.setStyle("photo");
     GUILabelComponent.create(wrapPicturePanel, "Photo: ", TableLayoutData.create(0, 2));
     GUIPanelComponent panelPicture = GUIEdit.createFile(wrapPicturePanel, Employee.PROPERTIES.photo, true, 
true);

     // Wires (“bindings”):
     GUIComponentBinding.create(employeeRelay.opRelay(), employeeNameSlotComponent.ipElement(),
               dateSlotComponent.ipElement(),
               isMarriedSlotComponent.ipElement(),
               genderSlotComponent.ipElement(),
               numOfChSlotComponent.ipElement(),
               panelFile.ipElement(),
               departmentGeneralList.ipSlotValueElement(),
               panelPicture.ipElement());
     }
```


SOLoist Demo - Company Organization
Powered by SOLoist

Home page 8

Employee details Dragan Milicev

General | Department

Name:	ragan Milicev
ate of birth:	
Is married:	✓
Gender:	Male
Number of children:	3
Department:	Headquarters 9
Contract:	No file [Maximum size: 5MB]

Photo:
[Maximum size: 5MB]
10

Status bar:
Copyright © 2011 - Serbian Object Laboratories (SOL) - Licence

CONCEPTS

- *GUI item configuration setting* is a set of presentational and behavioral parameters defined for one particular kind of elements in the system:
    - icons (small, large, for drag-and-drop, etc.)
    - texts that are displayed as the name, type, description, label, and tip of the element
    - how sub-nodes in a tree view are obtained from the element
    - the behavior on double mouse click on the element
- Elements can be objects of classes, but also elements of the UML model (e.g. classes, properties, etc.), accessible through UML reflection
- At runtime, GUI item settings are objects of an UML built-in class (from SOLoist model library)
- A GUI item configuration setting can be a sub-setting of another setting, meaning that it inherits all the parameters from the latter, but it can also redefine (i.e., override) any of the parameters, specifying different appearance or behavior.

CONCEPTS

- *GUI context* is a collection of GUI item configuration settings that define the presentational parameters for one part of the system's GUI.
- A GUI context can be a sub-context of another context, meaning that it inherits all the configuration settings from the latter, but it can also redefine (i.e., override) any of the settings.
- At runtime, GUI contexts are objects of an UML built-in class (from SOLoist UML model library)

CONCEPTS

```
// Object setting for department
GUIObjectSetting departmentOS = GUIObjectSetting.create(application.getContext(), Department.CLASSIFIER);
GUIPictureFeature.createSmallIcon(departmentOS, "res/img/CompanyOrganizationIcons/3.01_Department1.png");
GUITextFeature.createName(departmentOS, "name");
GUINavigatorFeature.createSubnodes(departmentOS, "subDepts");

// Object setting for employee
GUIObjectSetting employeeOS = GUIObjectSetting.create(application.getContext(), Employee.CLASSIFIER);
GUIPictureFeature.createSmallIcon(employeeOS, "res/img/CompanyOrganizationIcons/3.03_MaleEmployee2.png");
GUITextFeature.createName(employeeOS, "name");

// Tooltip for department
GUITextFeature tooltipDepr = new GUITextFeature();
tooltipDepr.setFixed(true);
tooltipDepr.setTextValue("Double click to see details");
departmentOS.addFeature(tooltipDepr);

// Tooltip for employee
GUITextFeature tooltipEmpl = new GUITextFeature();
tooltipEmpl.setFixed(true);
tooltipEmpl.setTextValue("Double click to see details");
employeeOS.addFeature(tooltipEmpl);

…
```

CONCEPTS

```
// Bindings feature for Department (double click)
GUIBindingsFeature bfDepartment = new GUIBindingsFeature();
GUIComponentBinding.create(bfDepartment.opDoubleClick(), departmentPanel.ipDepartment());
GUIComponentBinding.create(bfDepartment.opDoubleClick(), departmentPanel.ipShow());
departmentOS.addFeature(bfDepartment);

// Bindings feature for Employee (double click)
GUIBindingsFeature bfEmployee = new GUIBindingsFeature();
GUIComponentBinding.create(bfEmployee.opDoubleClick(), employeePanelTab.ipEmployee());
GUIComponentBinding.create(bfEmployee.opDoubleClick(), employeePanelTab.ipShow());
employeeOS.addFeature(bfEmployee);
```

- This ensures:
  - easy, but flexible customization
  - consistent appearance and look-and-feel throughout the application
  - less errors

PRINCIPLES

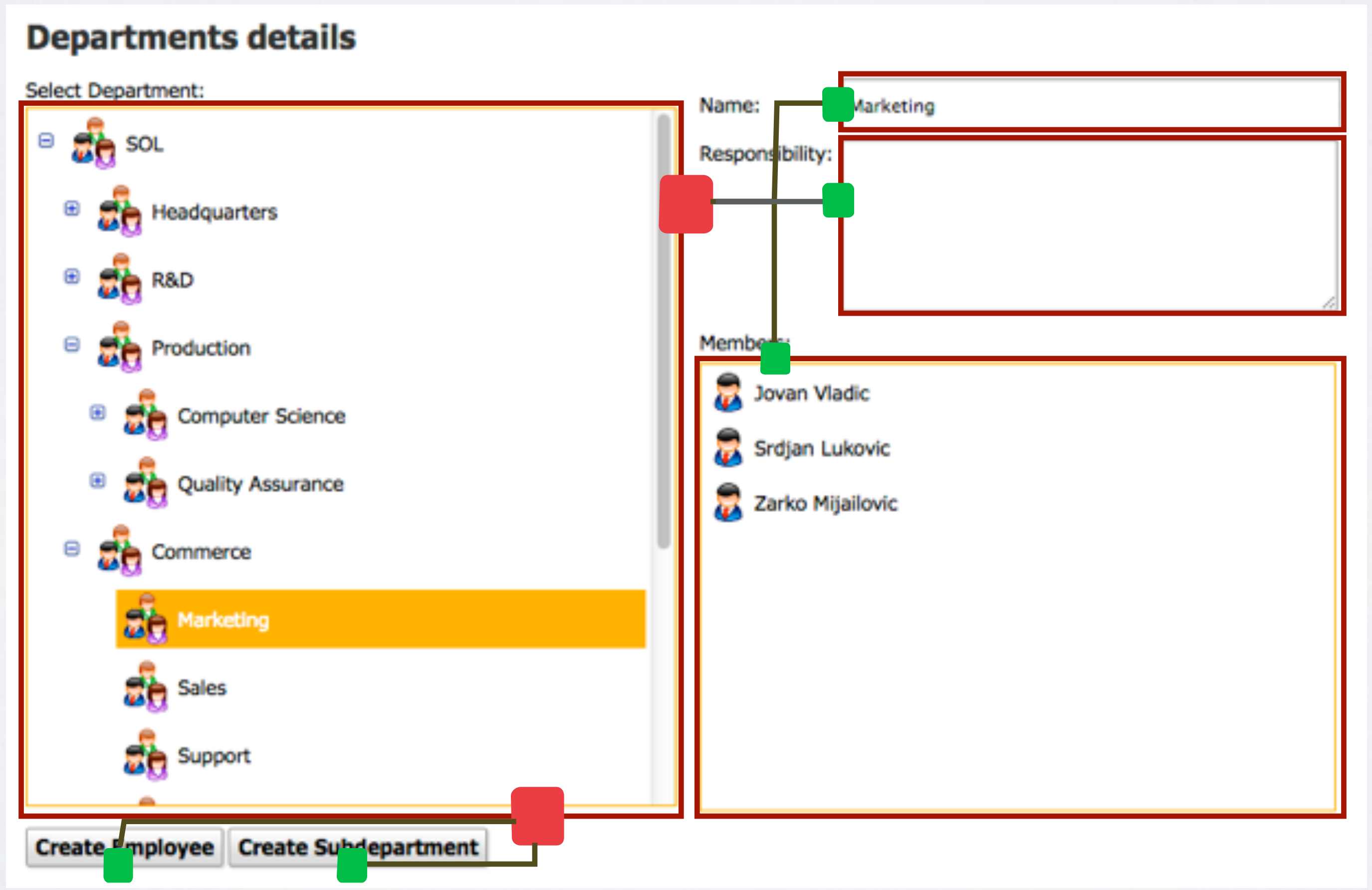
Proper and full semantic coupling of GUI and object space (no semantic discontinuities):

1. data binding of components:

`GUIEdit.createField(panelDeptDetails, `<u>`Department.PROPERTIES.name`</u>`)`

…

`GUIInput.createList(detailsPanel, `<u>`Department.PROPERTIES.members`</u>`)`



Departments details

Select Department:

- SOL
  - Headquarters
  - R&D
  - Production
    - Computer Science
    - Quality Assurance
  - Commerce
    - Marketing
    - Sales
    - Support

Name: Marketing

Responsibility:

Members:

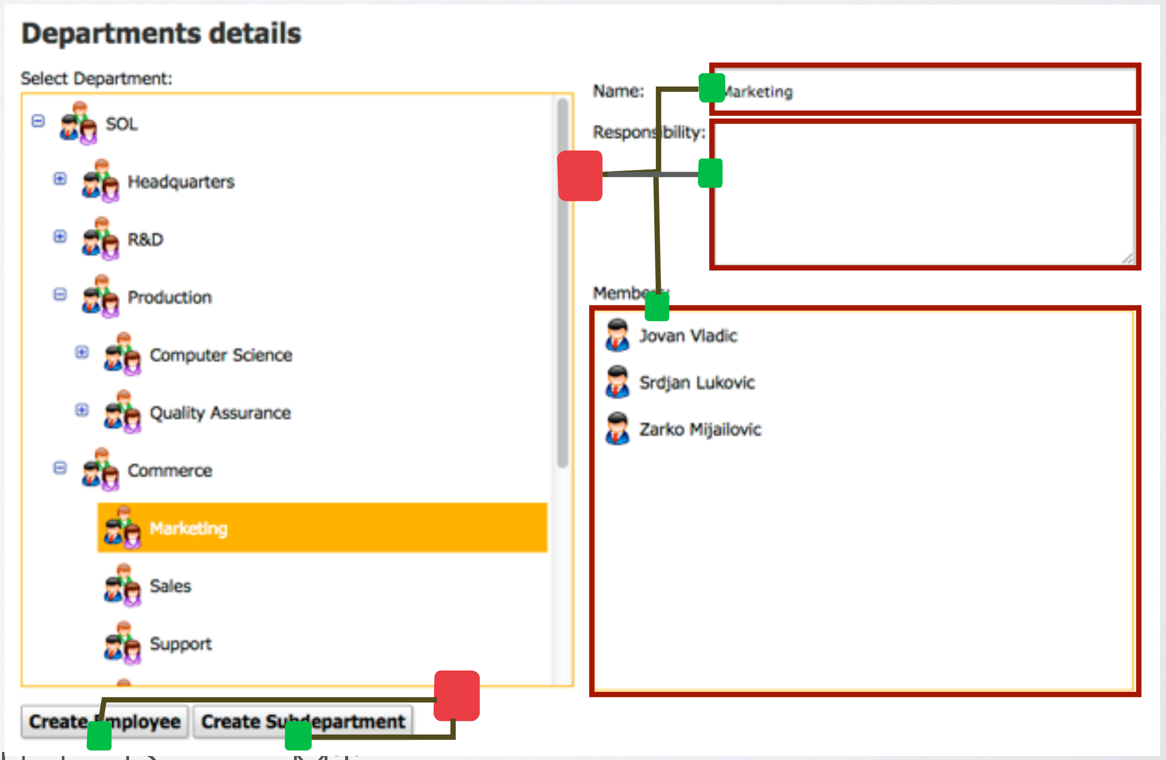
- Jovan Vladic
- Srdjan Lukovic
- Zarko Mijailovic

PRINCIPLES

Proper and full semantic coupling of GUI and object space (no semantic discontinuities):

2. coupling with commands:

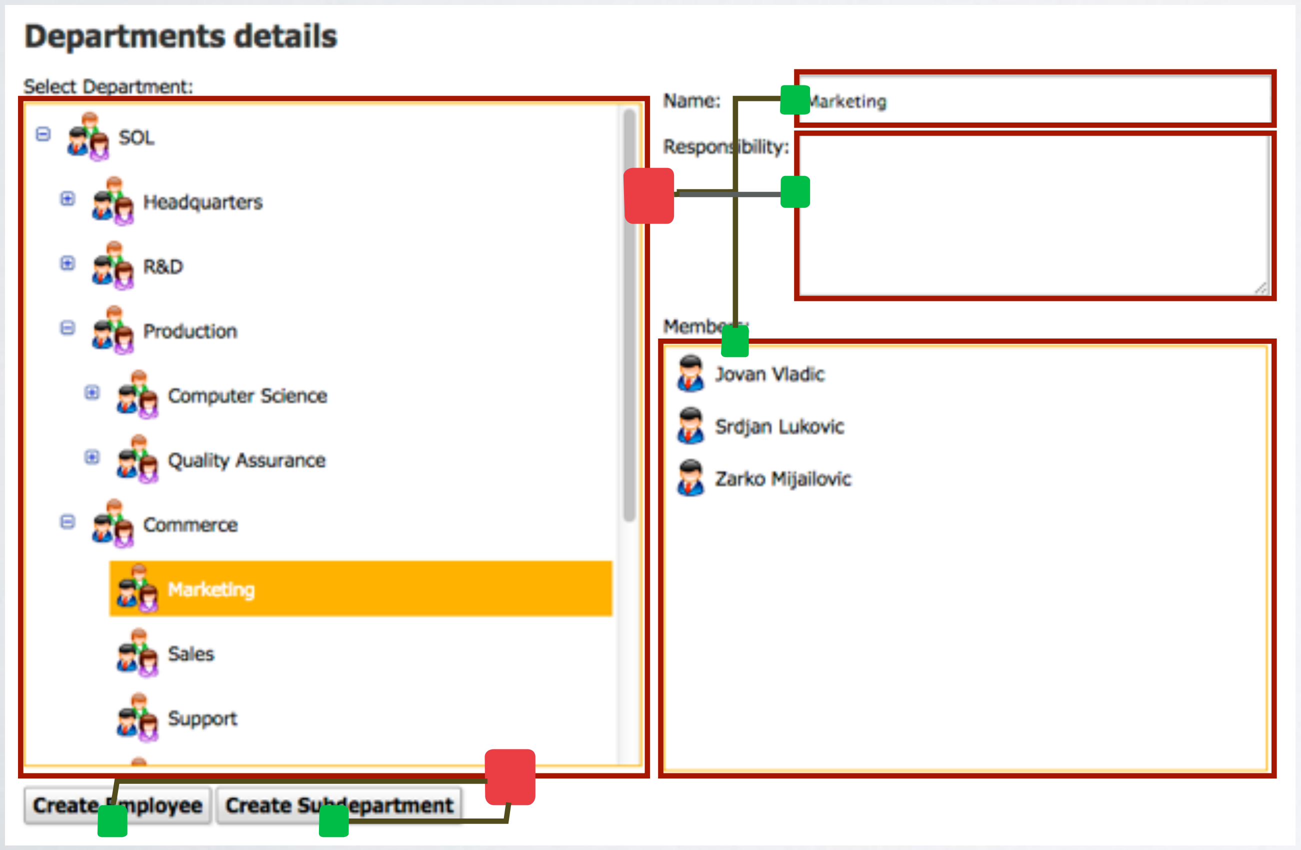
```
GUIButtonComponent btnCreateEmpl = GUIButtonComponent.create(departmentButtonPanel, "Create Employee");
          btnCreateEmpl.setStyle("button");

CmdCreateObjectAndLinkToObject createEmplCommand = new CmdCreateObjectAndLinkToObject();
     createEmplCommand.setClassName(Employee.CLASSIFIER);
     createEmplCommand.setAssocEndName(Department.PROPERTIES.members);
createEmplCommand.setType(Department.CLASSIFIER);
     btnCreateEmpl.setCommand(createEmplCommand);

GUIComponentBinding.create(departmentTreeView.opValue(), btnCreateEmpl,
     CmdCreateObjectAndLinkToObject.PROPERTIES.target);
```

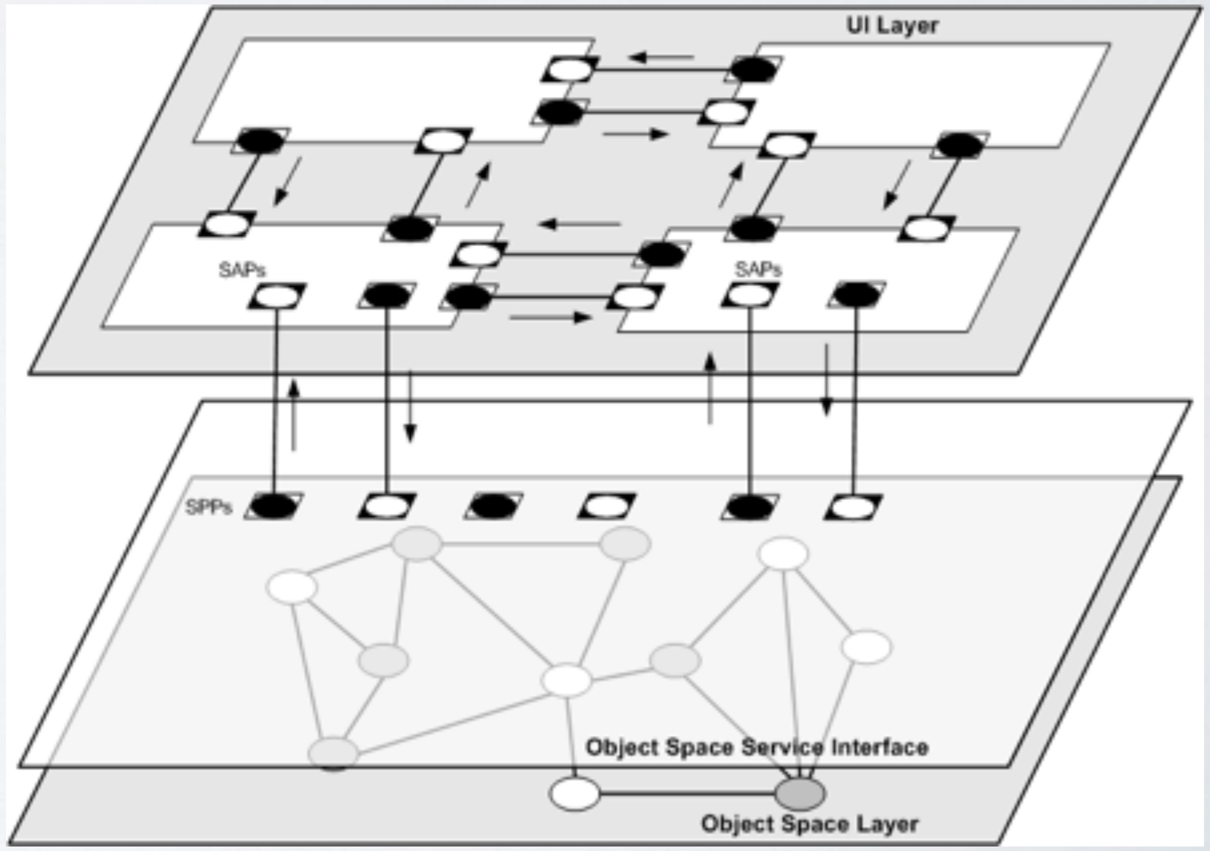


PRINCIPLES

Behavior:

- When such a component receives a new value (a reference to the host object) on its input pin, it issues an AJAX request to the server and fetches the value of the configured slot to display.
- If the component is an edit component, it issues a Write Slot request to the server when input value is changed.
- *Input* components do not change the object space, but provide the user's input or selection to its output pin.
- A button receives command parameters on its input pins. When pressed, it issues a request to execute the attached command on the server, passing the parameters to the corresponding input pins of the command.





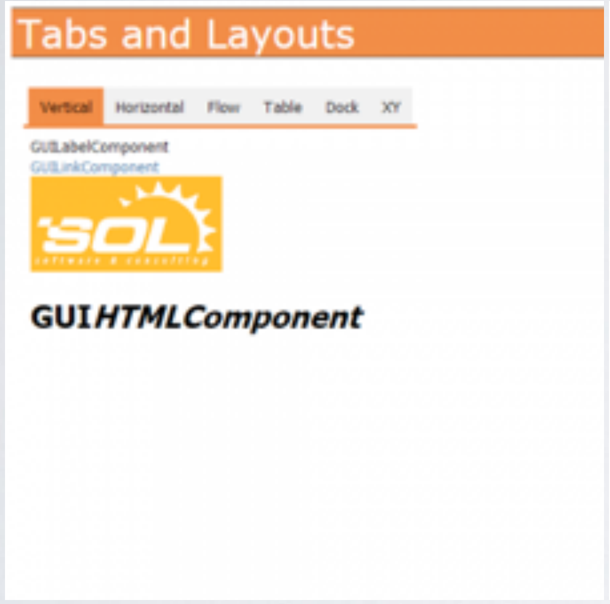
PRINCIPLES

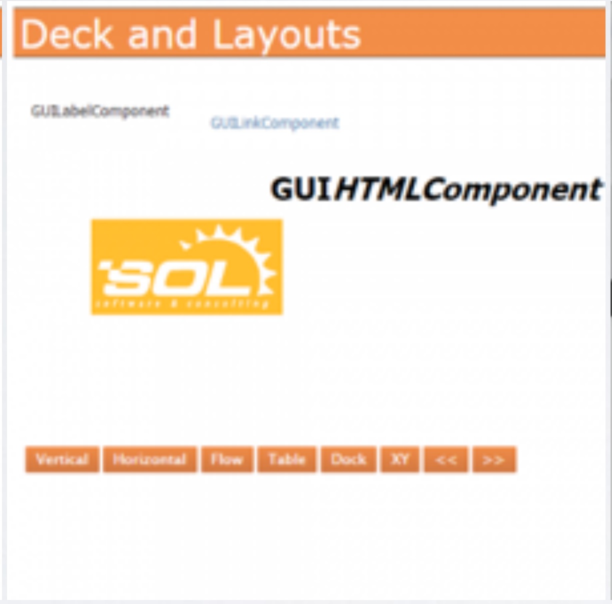
Consequences and features:

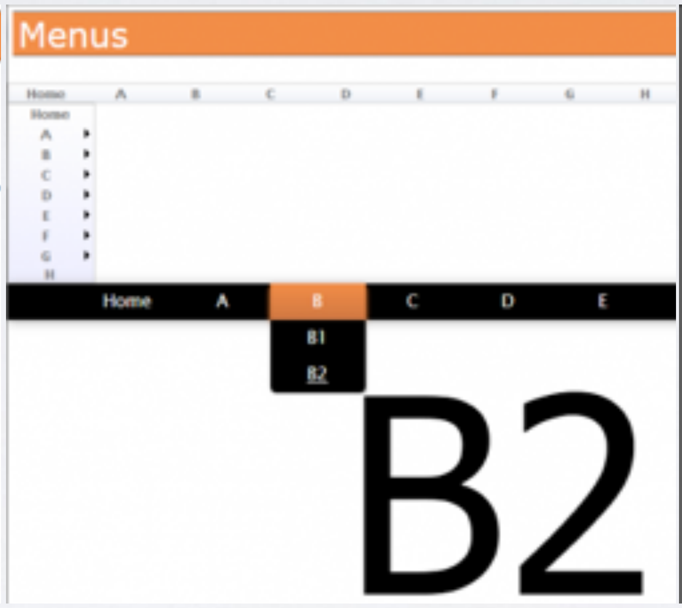
- *Notification mechanism*: for each change in the object space, SOLoist runtime notifies all GUI widgets in the same user's session (in which the transaction has been performed) — the widgets are updated automatically, no need for any programming!
- No semantic discontinuities
- "Single-page" application paradigm
- The entire application loads in a couple of seconds
- Then, the UI is very fast and responsive (immediate response for entire static contents), as in desktop applications
- Significantly reduced network traffic and load for the server, and more workload for the client (dedicated to GUI and one user session anyhow)
- Deferred (lazy) loading of pieces of application is possible (load on demand)
- Dynamic contents is also supported: fetch the contents of a capsule from the server, on each request

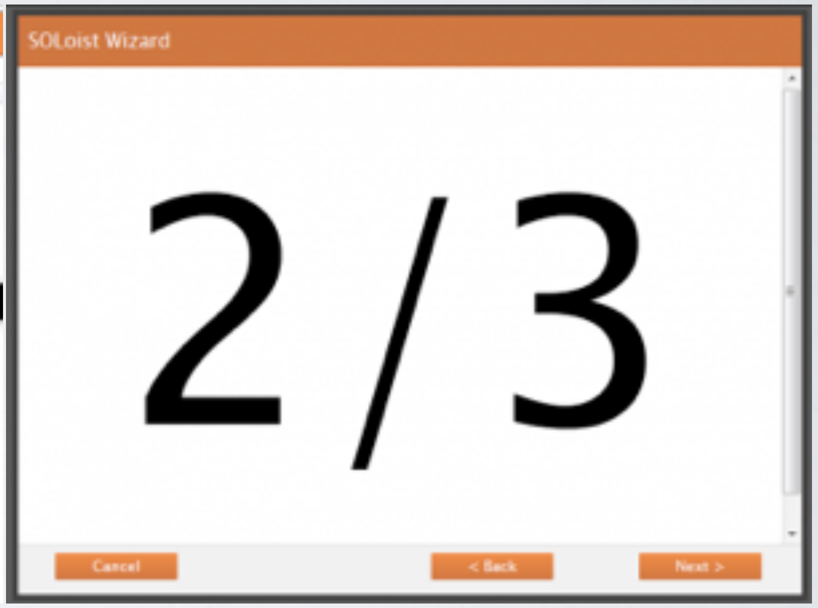
LIBRARY

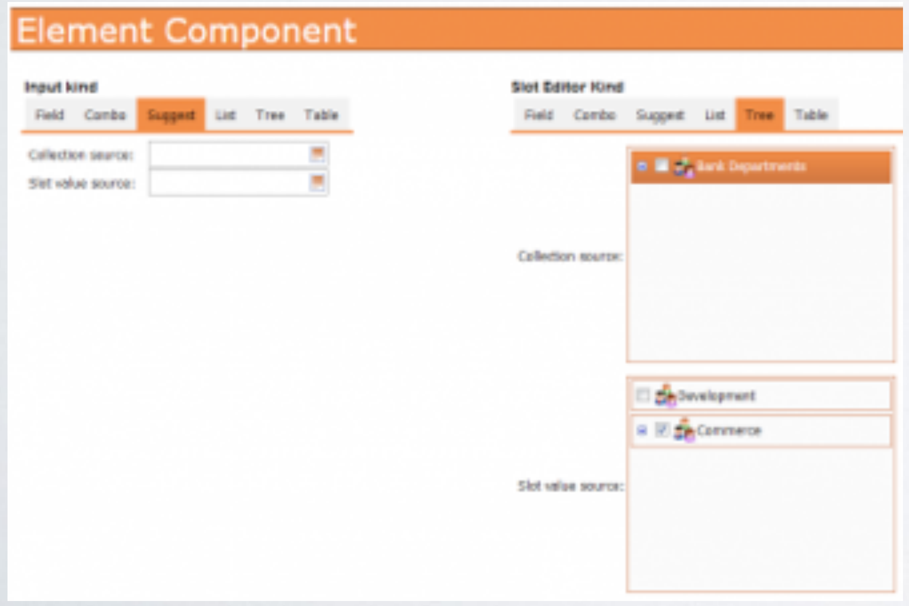
- Rich library of built-in components
- Custom-built components are also possible
- Sample applications:

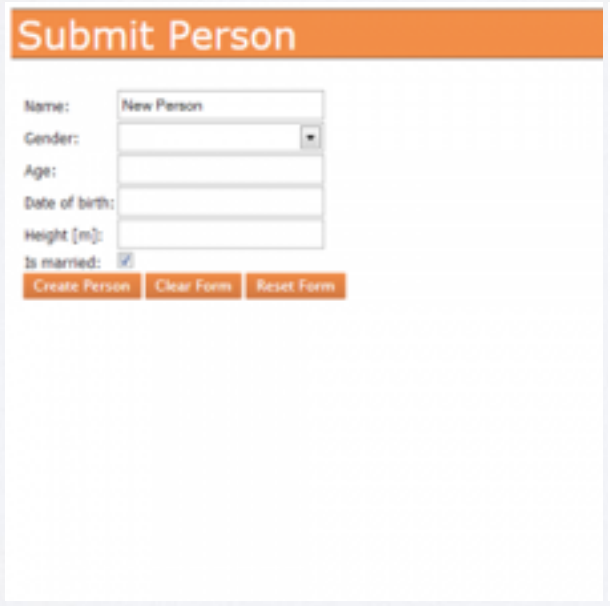




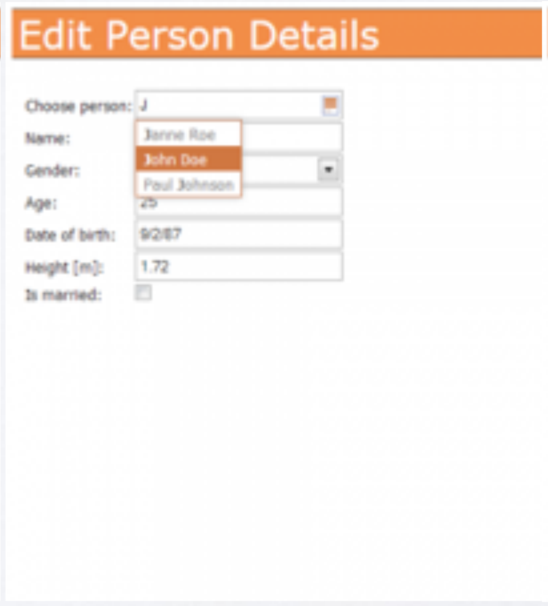




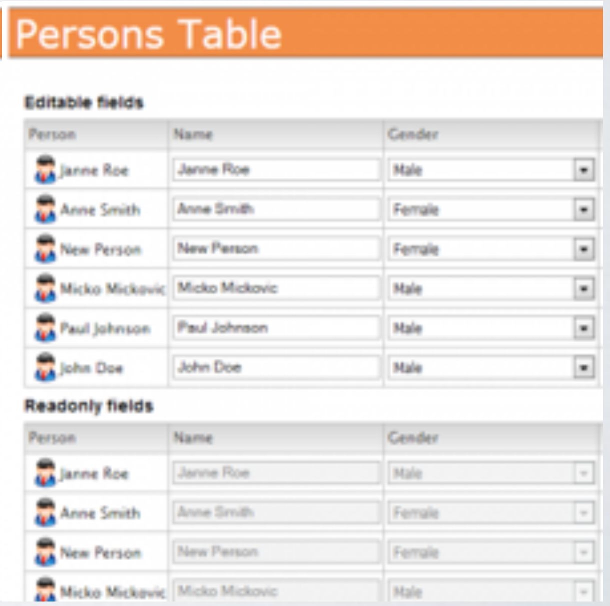




Name:	New Person
Gender:	
Age:	
Date of birth:	
Height [m]:	
Is married:	✓



Choose person:	J
Name:	
Gender:	
Age:	25
Date of birth:	9/2/87
Height [m]:	1.72
Is married:	



Editable fields

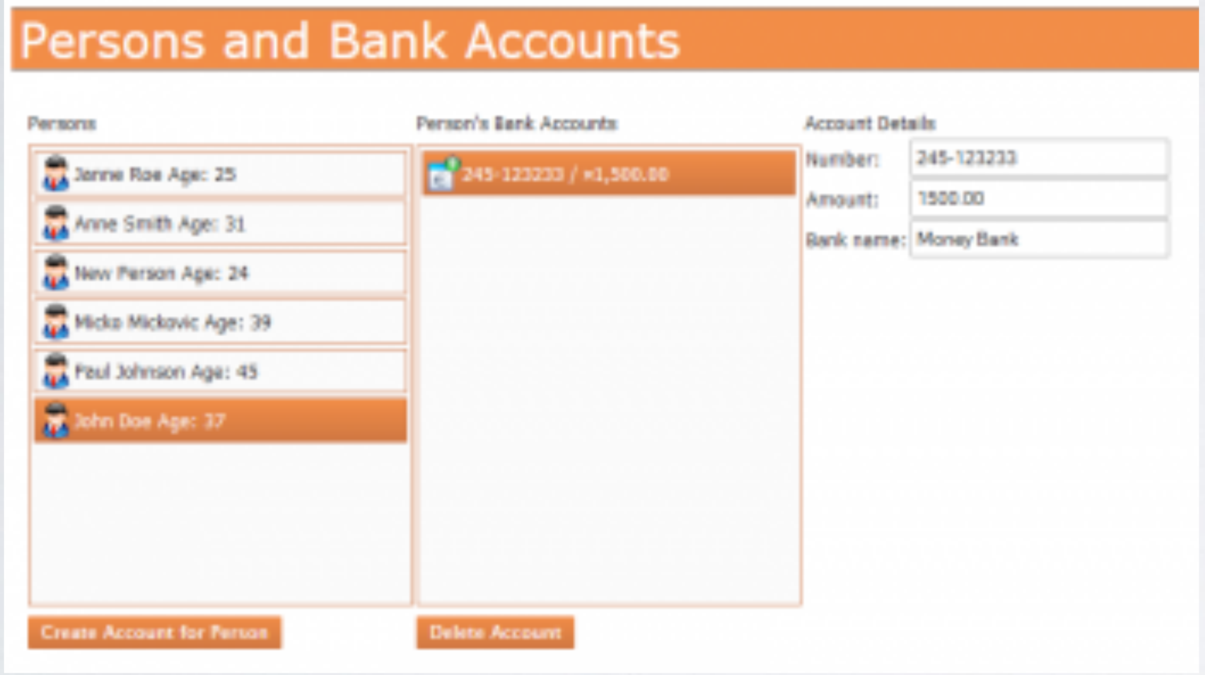
Person	Name	Gender
Janne Roe	Janne Roe	Male
Anne Smith	Anne Smith	Female
New Person	New Person	Female
Micko Mickovic	Micko Mickovic	Male
Paul Johnson	Paul Johnson	Male
John Doe	John Doe	Male

Readonly fields

Person	Name	Gender
Janne Roe	Janne Roe	Male
Anne Smith	Anne Smith	Female
New Person	New Person	Female
Micko Mickovic	Micko Mickovic	Male

LIBRARY

- Sample applications:



Persons and Bank Accounts

Persons

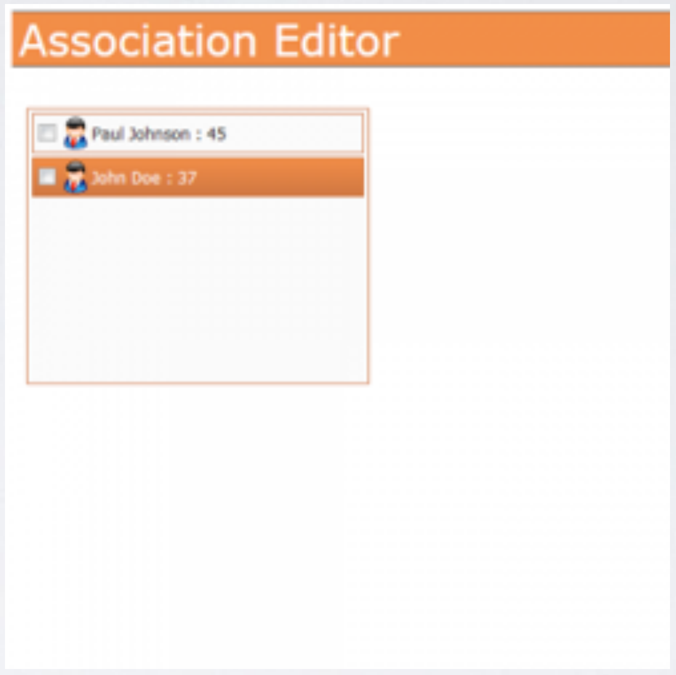
- Janne Roe Age: 25
- Anne Smith Age: 31
- New Person Age: 24
- Micko Mickovic Age: 39
- Paul Johnson Age: 45
- John Doe Age: 37

Person's Bank Accounts

- 245-123233 / ¤1,500.00

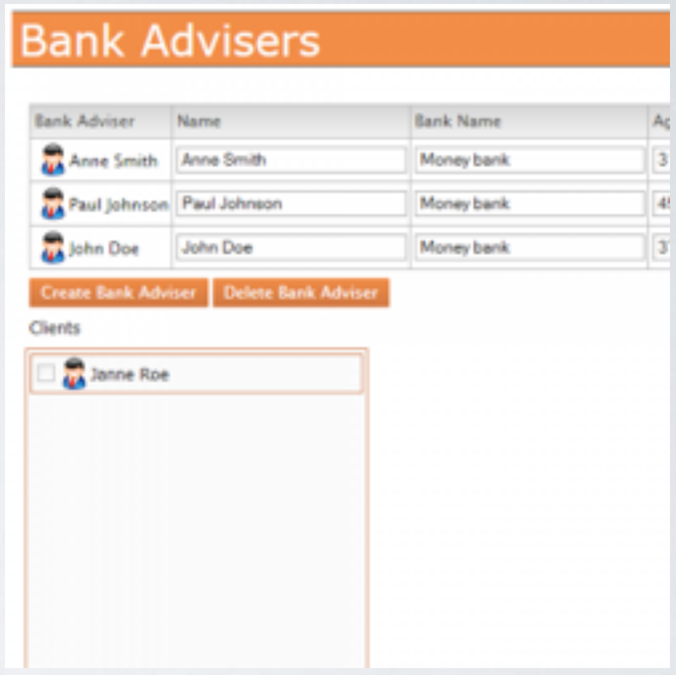
Account Details

Number:	245-123233
Amount:	1500.00
Bank name:	Money Bank



Association Editor

- ☐ Paul Johnson : 45
- ☐ John Doe : 37

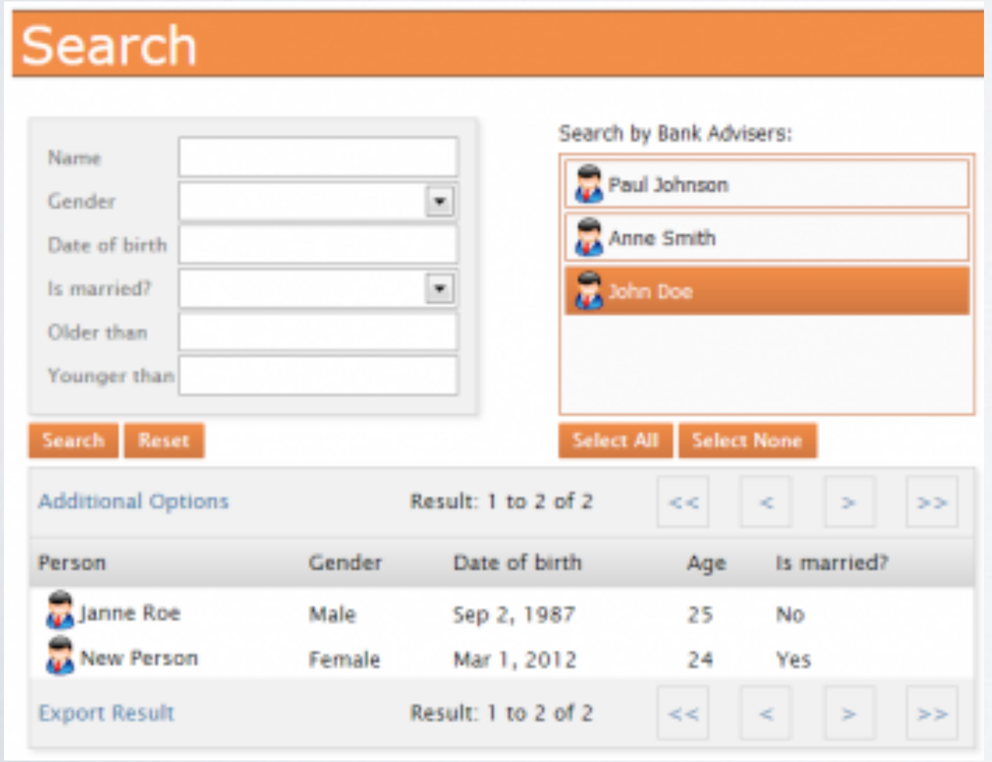


Bank Advisers

Bank Adviser	Name	Bank Name	Ag
Anne Smith	Anne Smith	Money bank	3
Paul Johnson	Paul Johnson	Money bank	4
John Doe	John Doe	Money bank	3

Clients

- ☐ Janne Roe



Search

Name	
Gender	
Date of birth	
Is married?	
Older than	
Younger than	

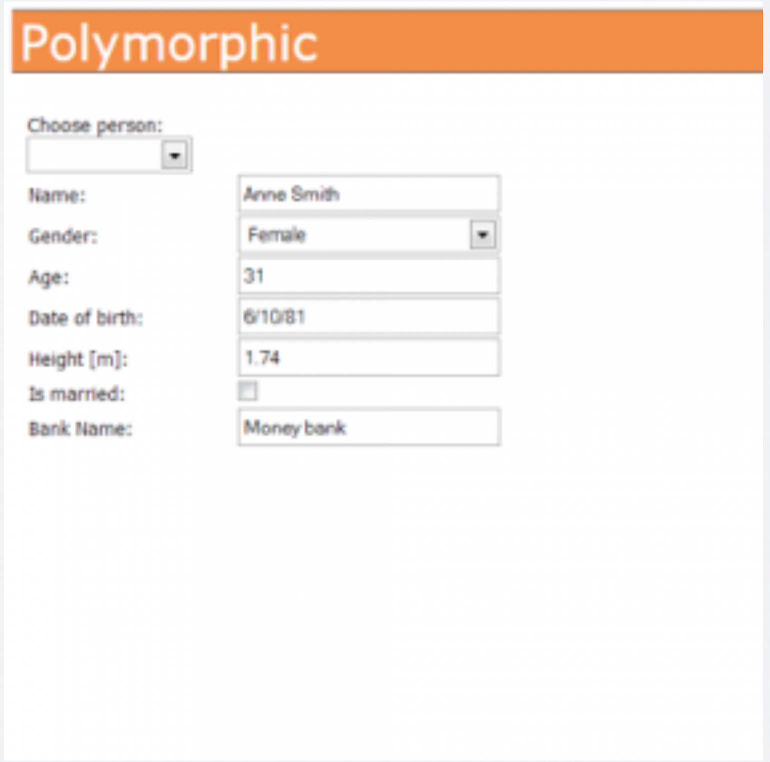
Search by Bank Advisers:

- Paul Johnson
- Anne Smith
- John Doe

Result: 1 to 2 of 2

Person	Gender	Date of birth	Age	Is married?
Janne Roe	Male	Sep 2, 1987	25	No
New Person	Female	Mar 1, 2012	24	Yes

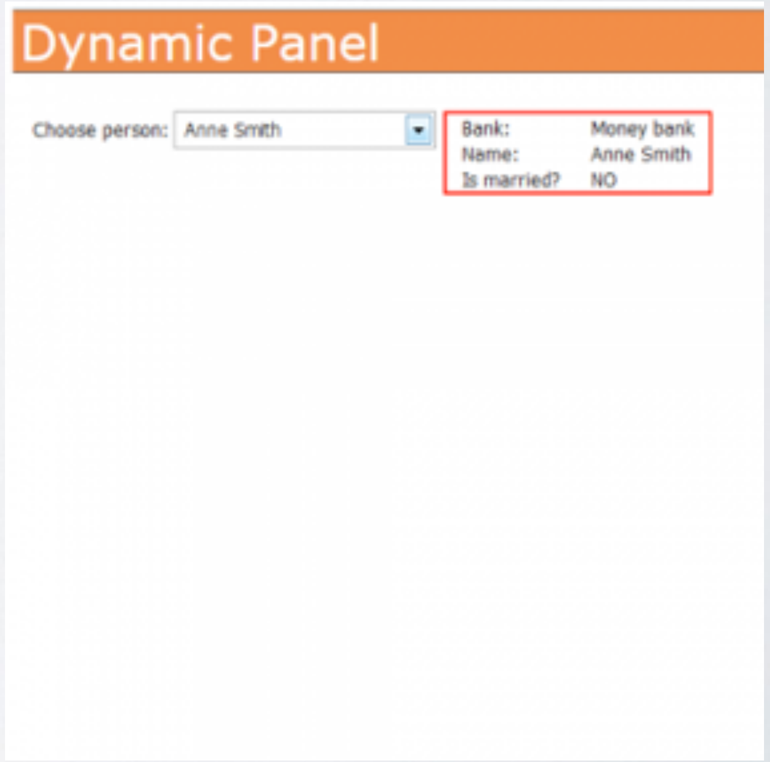
Result: 1 to 2 of 2



Polymorphic

Choose person:

Name:	Anne Smith
Gender:	Female
Age:	31
Date of birth:	6/10/81
Height [m]:	1.74
Is married:	☐
Bank Name:	Money bank



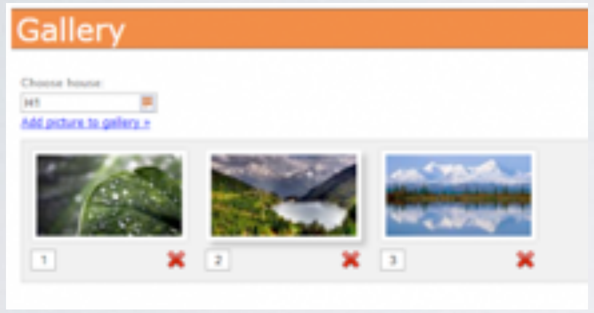
Dynamic Panel

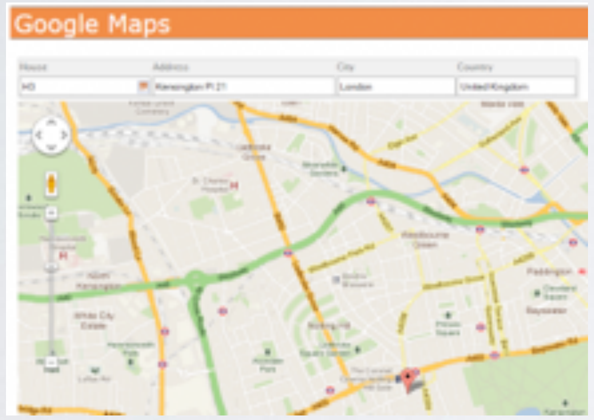
Choose person: Anne Smith

Bank:	Money bank
Name:	Anne Smith
Is married?	NO

LIBRARY

- Sample applications:





CHAPTER VIII
CONCLUSIONS

CONCLUSIONS

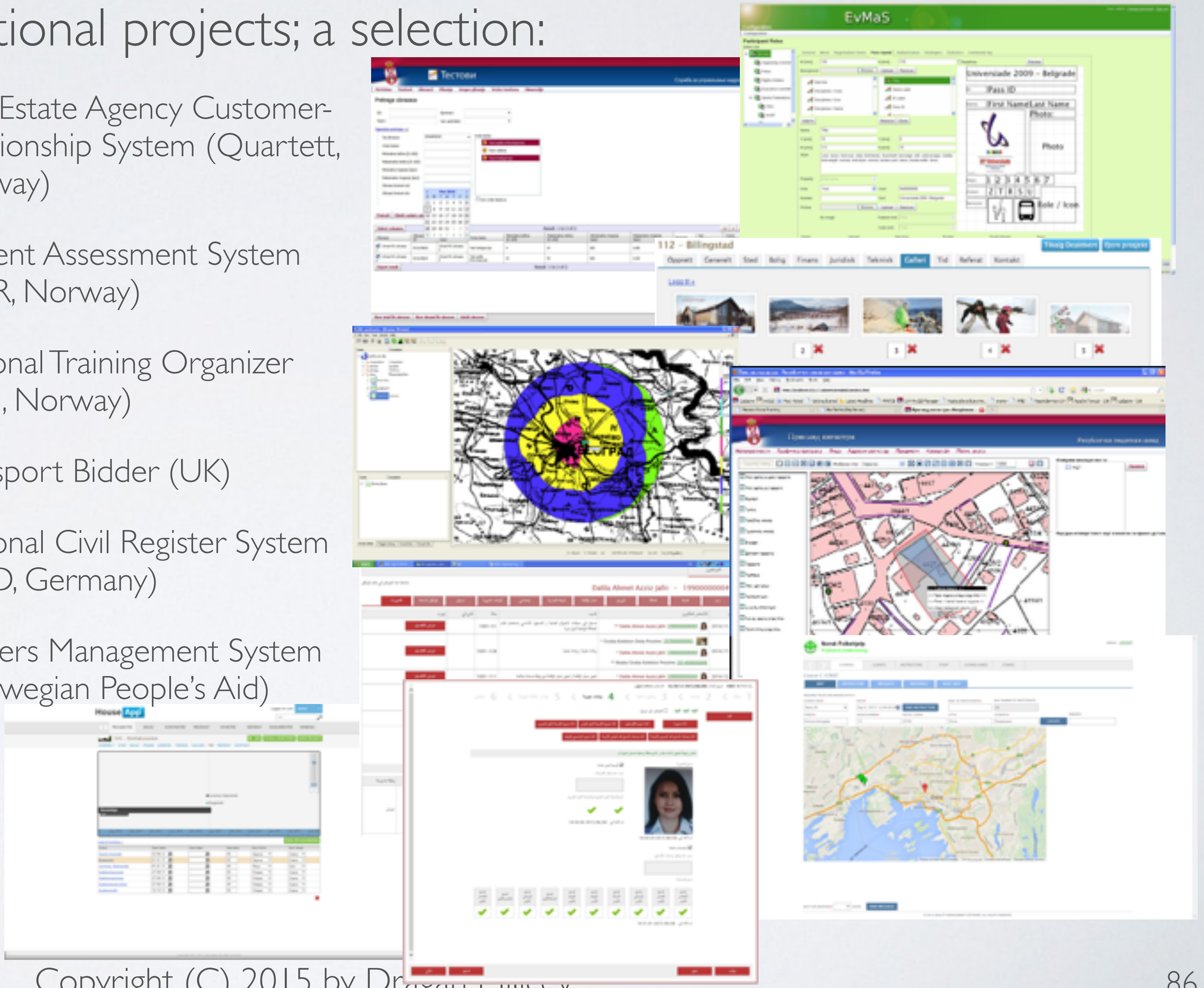
Model-driven development with executable UML and SOLoist:

- ensures proper usage of UML with full power
- eliminates discontinuities due to:
    - proper and formal coupling of different kinds of details (semantic discontinuities),
    - proper and formal coupling of elements of different level of details (scope discontinuities),
    - use of the same linguistic environment in all phases (phase discontinuities)
- raises the level of abstraction in software development, both in conceptual modeling, implementation of business logic, and UI development
- significantly reduces accidental complexity existing in mainstream technologies
- improves development productivity
- reduces mistakes and makes maintenance easier

SOLoist REFERENCES

SOLoist is a mature, stable, robust, and scalable product that has been applied in tens of large-scale international projects; a selection:

- Workshop Management System (RailConsult, Germany)
- Engineering Drawings Management System (RCData, Germany)
- The Analysis of VHF/UHF, Radar, and Radio-Relay Services (Serbian Air Traffic Control Agency)
- Event Management System (Indas, Serbia)
- Human Resources System (Government of Serbia)
- National Real-Estate Cadastre (Government of Serbia)
- First-Aid Assistant (QMSoft, Norway)
- Real Estate Agency Customer-Relationship System (Quartett, Norway)
- Student Assessment System (HCR, Norway)
- Personal Training Organizer (IMG, Norway)
- Transport Bidder (UK)
- National Civil Register System (G&D, Germany)
- Trainers Management System (Norwegian People's Aid)



www.soloist4uml.com

Q&A

Thank you for your attention!