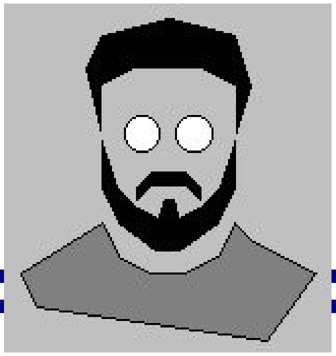
Model-Driven Development

Model-Driven Methods in Software Engineering

Alar Raabe

Content

Alar Raabe



- 30 years in IT
  - held various roles from programmer to a software architect
- Last 15 years in insurance domain
  - developed model-driven technology for insurance applications product-line
    - models
    - method/process
    - tools and platform framework
- Interests
  - software engineering (tools and technologies)
  - software architectures
  - model-driven software development
  - industry reference models (e.g. IBM IAA, IFW)
  - domain specific languages

Content

- Introduction
        - Common Language – some Definitions
        - The Problem
        - Beginning (Excursion into the History)
- Models in Software Development
    - Direct Modeling
        - Convergent Engineering
        - Domain-Driven Design
    - Models as Primary Artifacts
        - Generative Programming
        - Domain Specific Languages
        - Model-Driven Development Methods
- Practical Aspects
        - Model Management
        - Best Practices
        - Examples
- Conclusions
- References

Common Language – some Definitions 1

- Abstraction
  - a view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information
  - the process of formulating a view

- **Model**
  - a representation of a real world process, device, or concept
  - a representation of something that suppresses certain aspects of the modeled subject
  - a semantically closed abstraction of a system, or a complete description of a system, from a particular perspective
  - **structured information NOT A PICTURE!**

- **Metamodel**
  - a logical information model that specifies the modeling elements used within another (or the same) modeling notation
  - specification of the concepts, relationships and rules that are used to define a methodology
  - **a model of models**

Common Language – Some Definitions $_2$

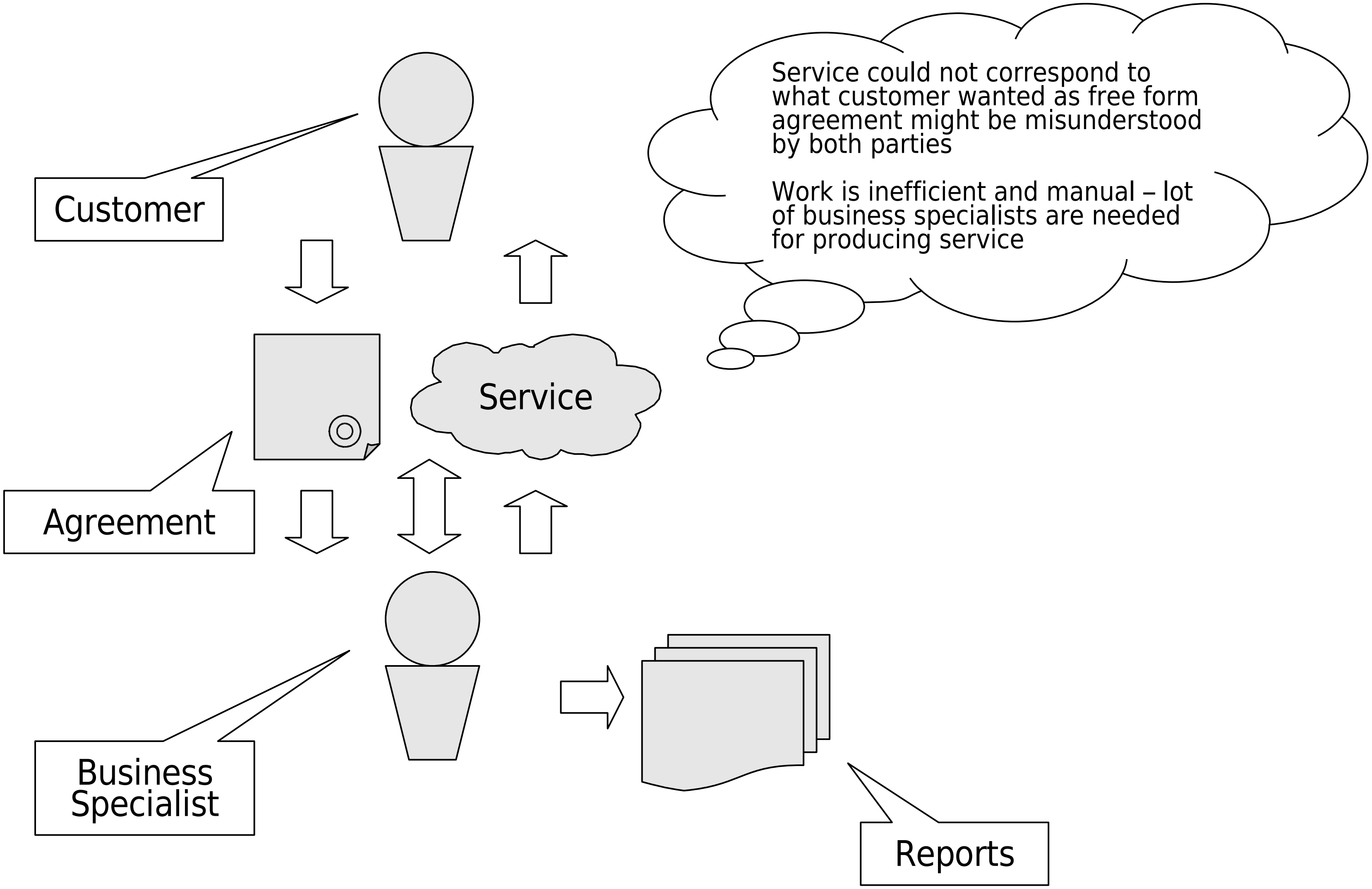
- Model **Transformations**
  - changing the form of the model while preserving semantics and some desirable properties (like correctness)

- Model **Refinements**
  - changing (enlarging) the content of the model – adding details

- **Domain**
  - a problem space
  - a distinct scope, within which common characteristics are exhibited, common rules observed, and over which a distribution transparency is preserved
  - an area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area (UML)

- Domain Specific Language (DSL)
  - language dedicated to a specific problem domain, problem representation technique, and/or problem solution technique

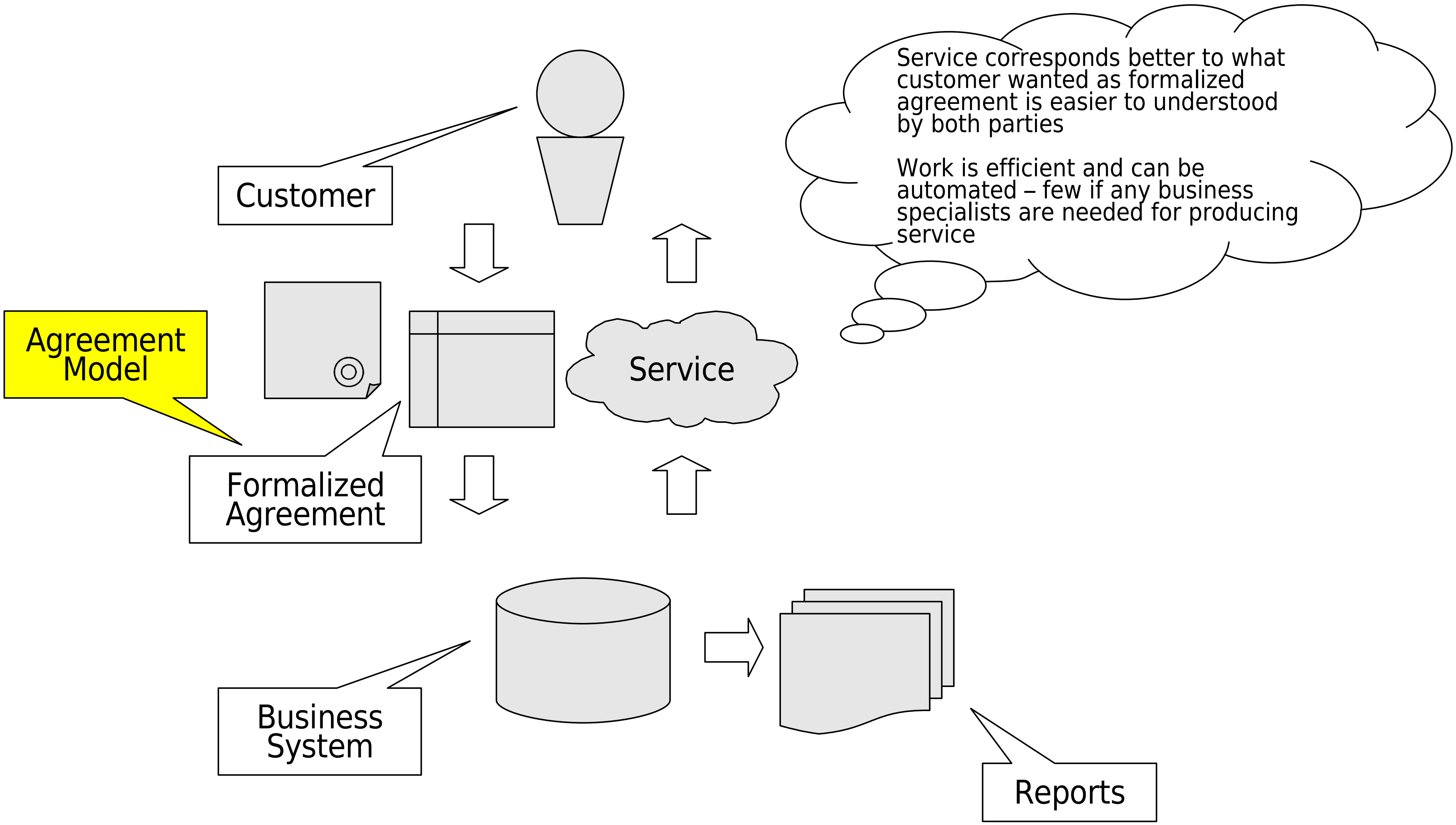
How we did Business Yesterday



Service could not correspond to what customer wanted as free form agreement might be misunderstood by both parties

Work is inefficient and manual – lot of business specialists are needed for producing service

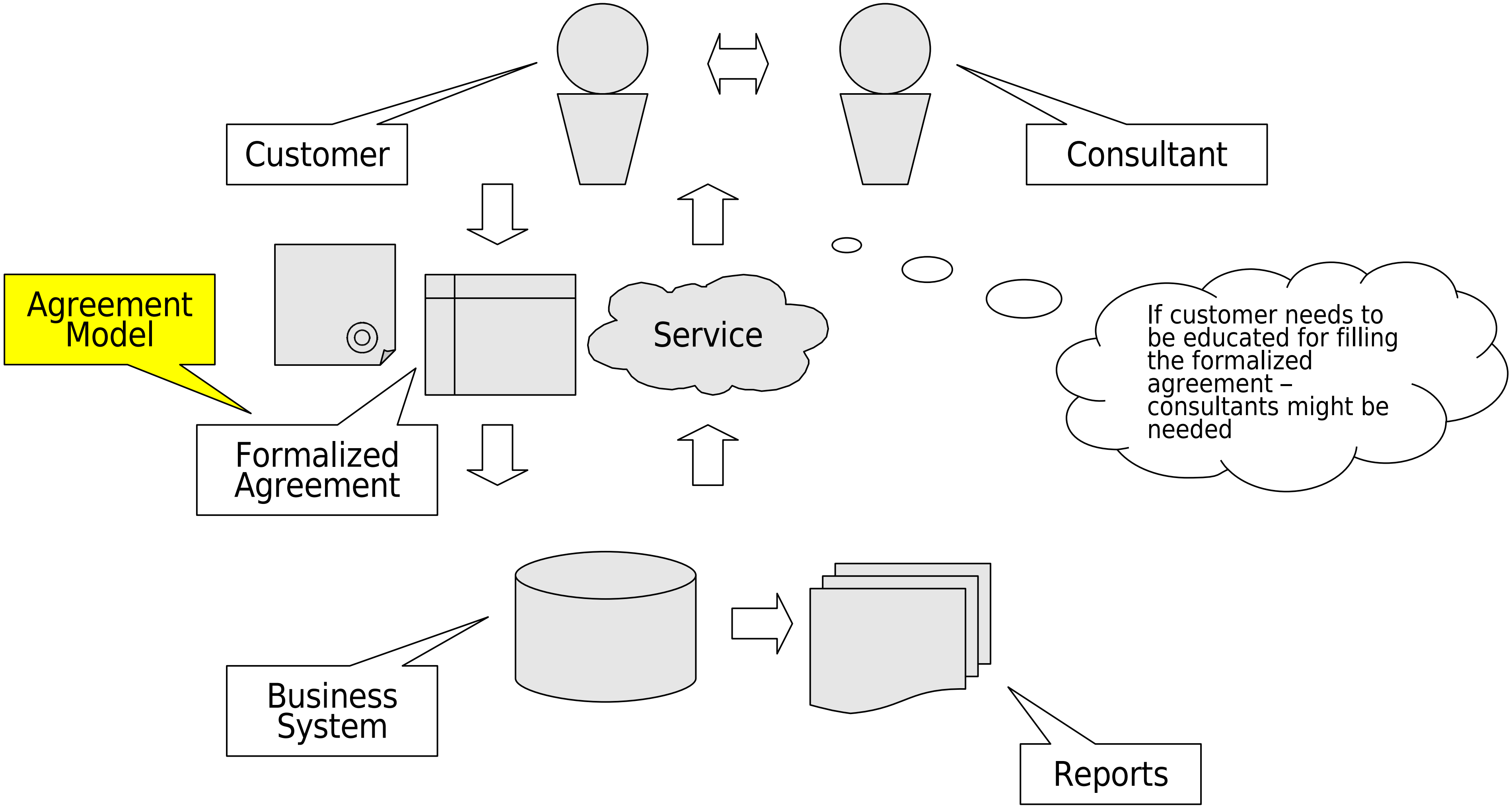
How we do Business Today



Service corresponds better to what customer wanted as formalized agreement is easier to understood by both parties

Work is efficient and can be automated – few if any business specialists are needed for producing service

How we do Business Tomorrow



Customer

Consultant

Agreement Model

Formalized Agreement

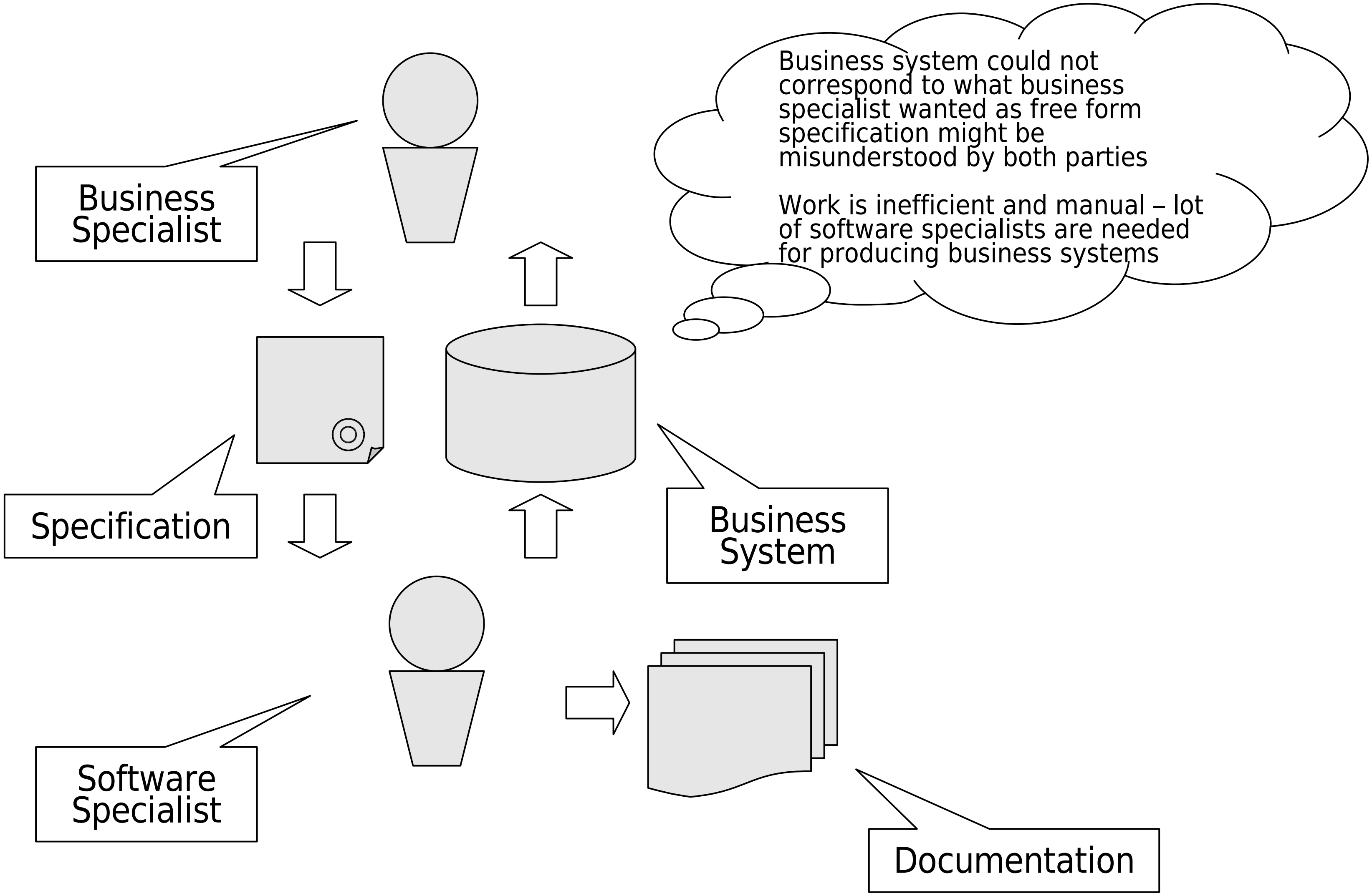
Service

If customer needs to be educated for filling the formalized agreement – consultants might be needed

Business System

Reports

How we Develop Software Today



Business Specialist

Specification

Business System

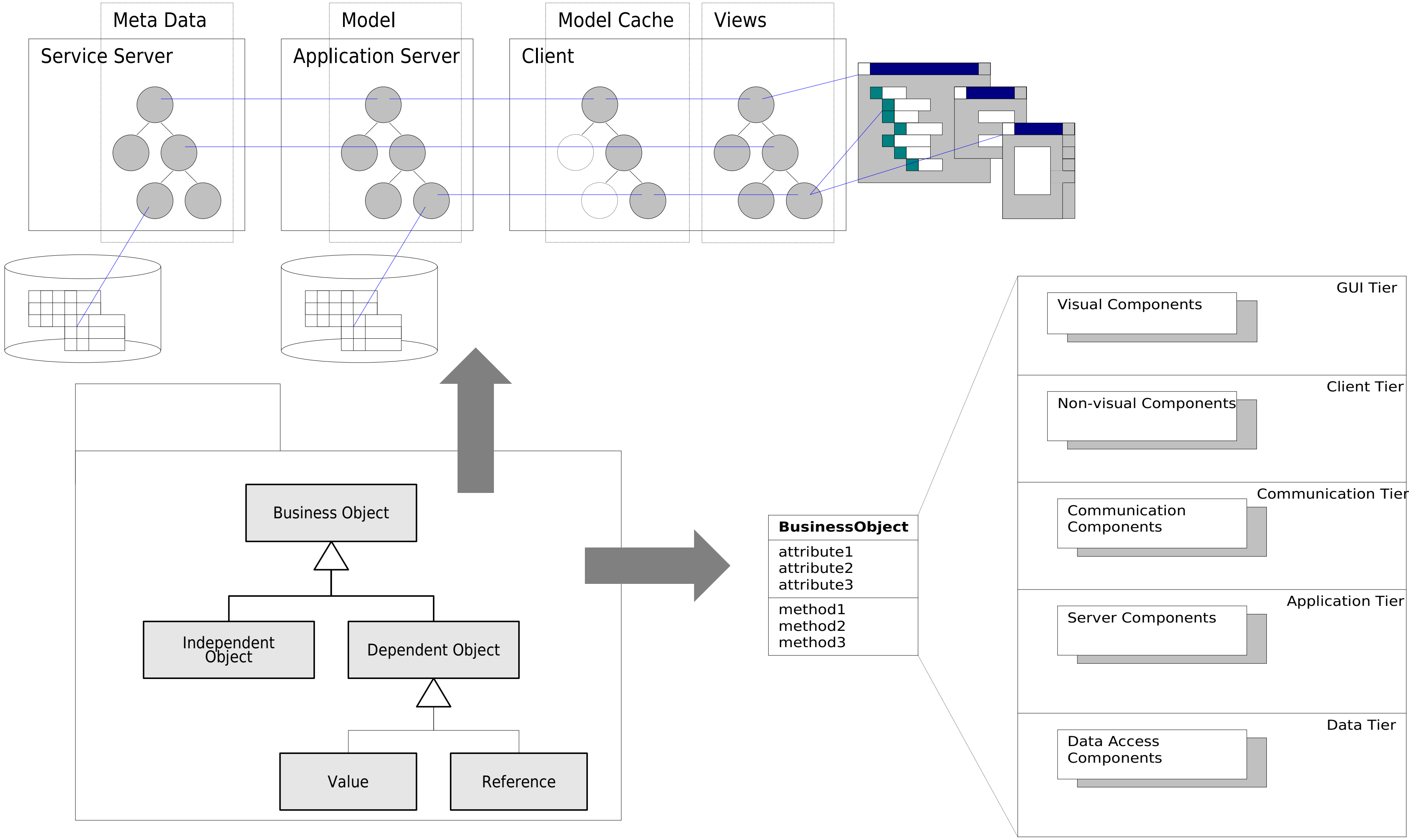
Software Specialist

Documentation

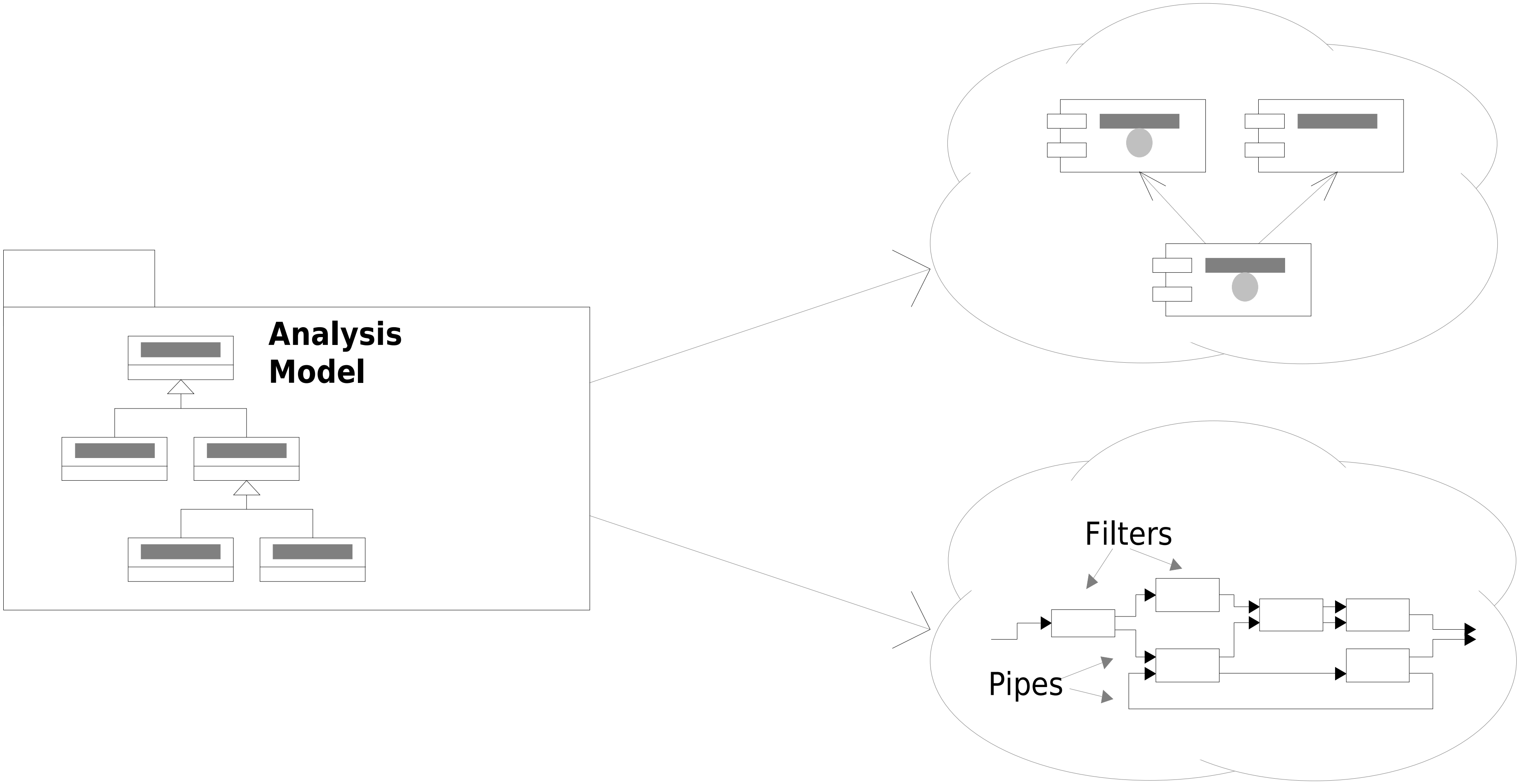
Business system could not correspond to what business specialist wanted as free form specification might be misunderstood by both parties

Work is inefficient and manual – lot of software specialists are needed for producing business systems

Consistency of Implementation



Mapping to Different Implementations



Possible Architecture Styles

Analysis Model

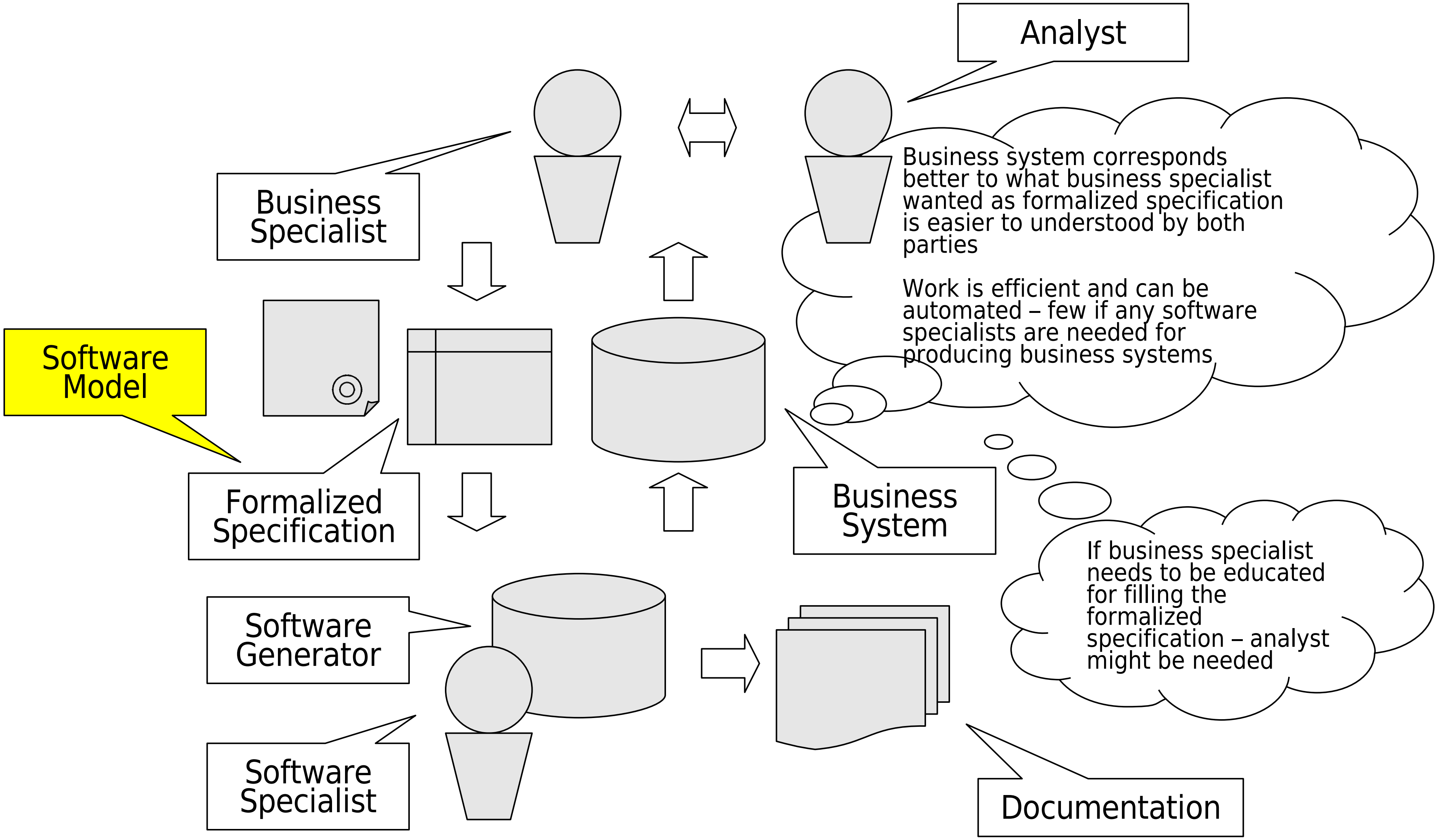
Filters

Pipes

Problem

- Requirements for today's business information systems
  - fast time-to-market – rapid delivery of initial results
  - flexibility – effortless and cheap change during the life-cycle
  - independence of business know-how from technology know-how
  - minimal (acquisition and ownership) cost
  - independence of technological platform
- Problem ⟶ Manual work
  - communication errors (systematic defects)
  - construction errors (random defects)
  - insufficient scalability of development process (sourcing)
  - difficult transfer of knowledge (continuity)
  - low reuse of both analysis and construction results (high cost)
  - long development time (low productivity)
  - insufficient flexibilitty of systems (high cost of changes)
- Solution ⟶ **Automation**

How we should Develop Software



Business system corresponds better to what business specialist wanted as formalized specification is easier to understood by both parties

Work is efficient and can be automated – few if any software specialists are needed for producing business systems

If business specialist needs to be educated for filling the formalized specification – analyst might be needed

Beginning (Excursion into the History)

"What has been will be again,
what has been done will be done again;
there is nothing new under the sun."
-- Ecclesiastes 1:9

- Programming Languages – to automate coding
  - FORTRAN (1954)
  - Lisp (1956)
  - Algol

- Problem-Oriented Languages/Systems – to automate programming
  - ICES (MIT 1961)
    - COGO, STRUDL, BRIDGE, ...
  - PRIZ

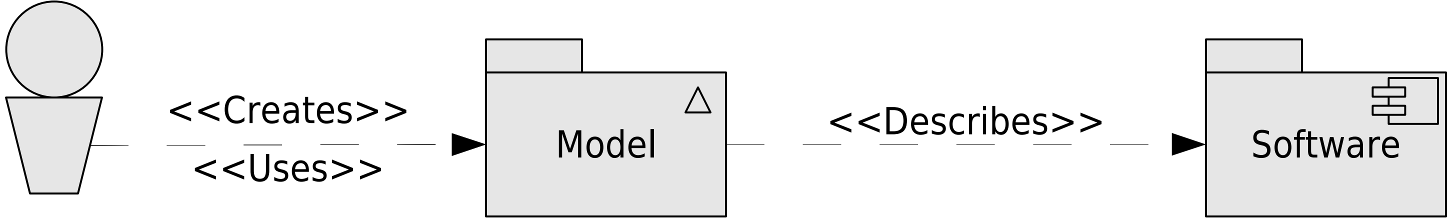
- Compiler Generators – generation of solution from model of problem
  - Yacc/Lex (1979)

- Application Generators
  - MetaTool & ... (Bell Labs 1988)
  - GENOA

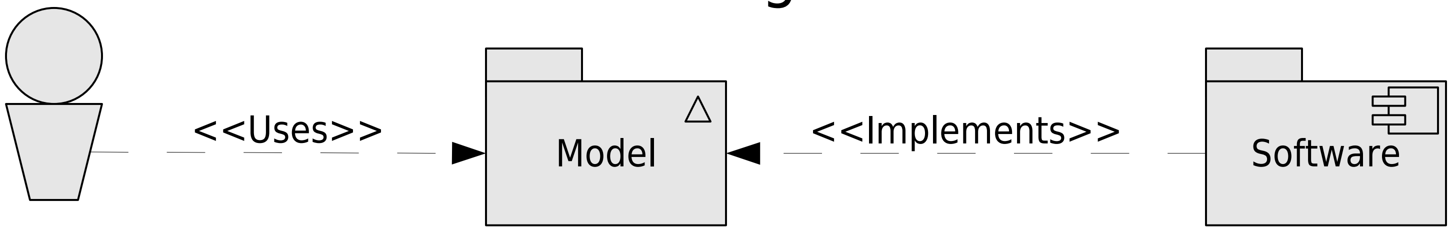
Content

Using Models in Software Development

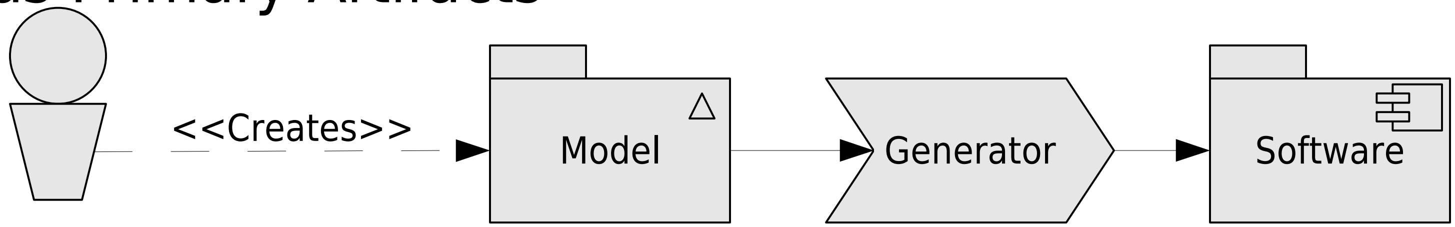
- Models as Descriptions and Illustrations



- Software as Model – Direct Modeling



- Models as Primary Artifacts



Direct Modeling

- History
  - Structured Programming / Structured Design (Jackson 1975)

“program structure should correspond to the structure of the problem”

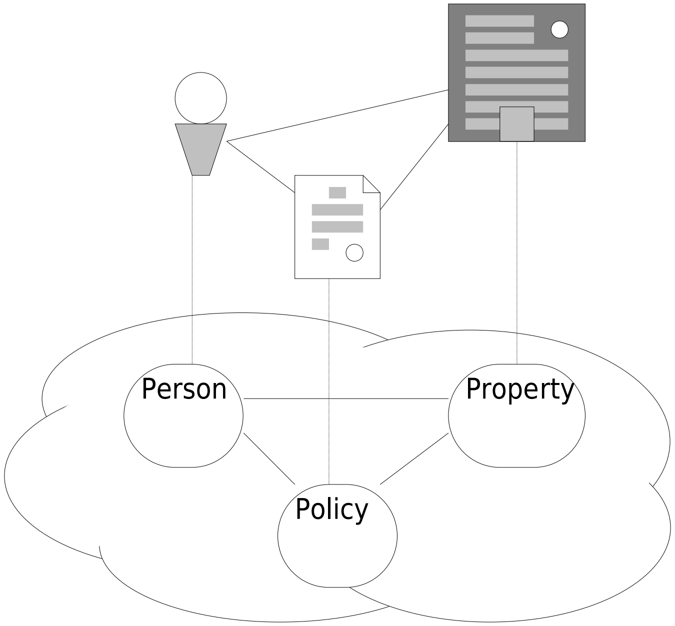
- Convergent Engineering
  - structure of business and business software should converge
  - flexibility and multiple usages of same software
- Domain-Driven Design
- Examples
  - Modeling Programs – programs that directly model something
    - Recursive Descent Parser
  - Generative Programs – programs, which are models and generate other programs

Convergent Engineering

- Convergent engineering – construct business software as a model of business (organization and processes) [Taylor]
  - business and the supporting software can be designed together
  - changes in business are easier – greater flexibility of software
  - same software can be used to:

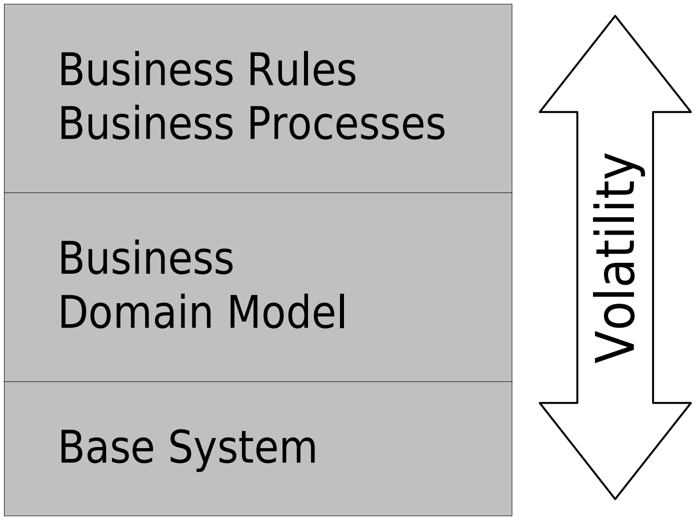
    1) run the day-to-day business, and

    2) plan (do “what-if” analysis)



Business Software Layers

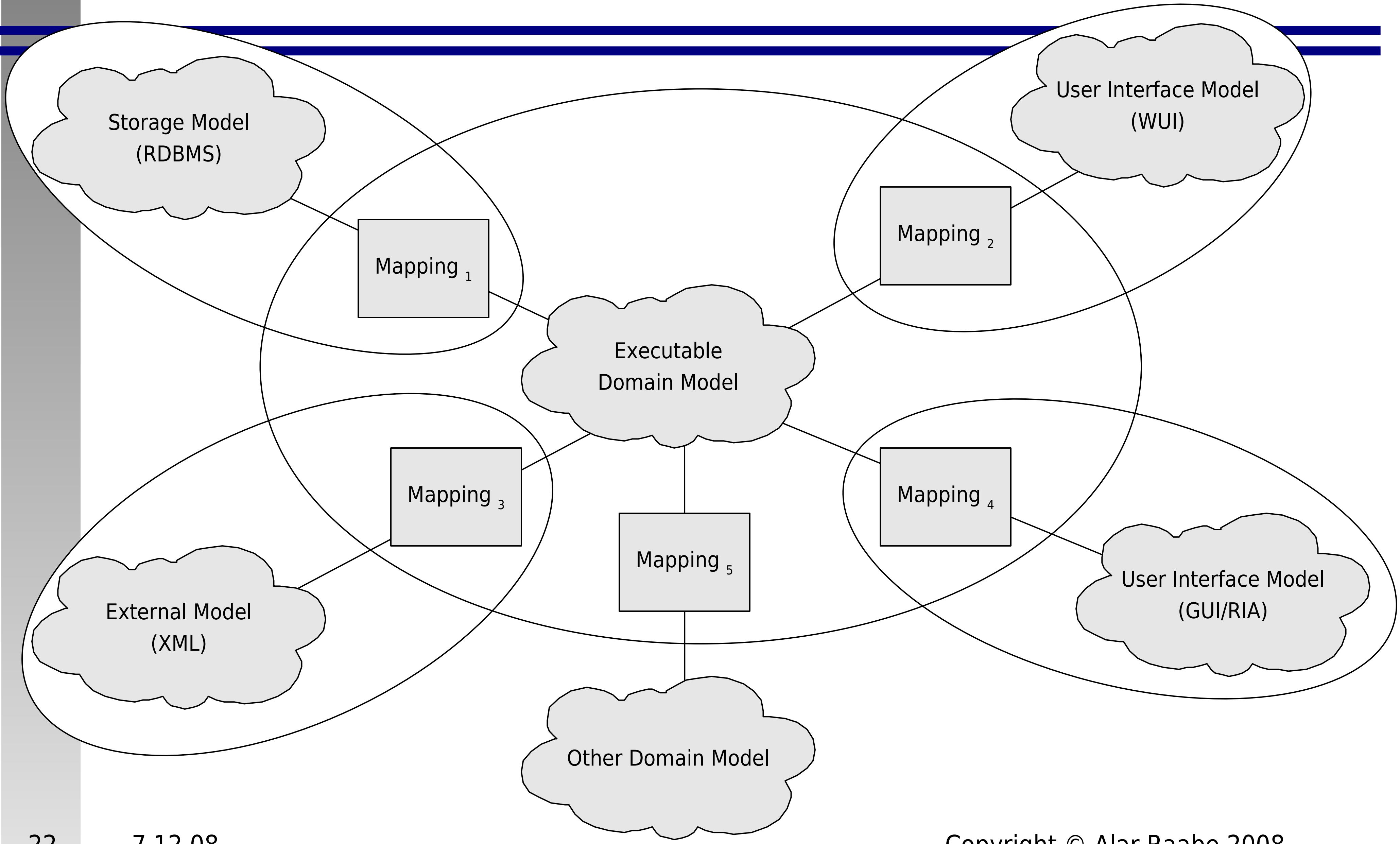
- Business rules and business processes
  - – Most volatile part of the software
  - – Depends on business context (e.g. product, task, user role, ...)
- Business logic (functionality) and business domain model
  - – More stable than business rules and business processes
- Base system(s)
  - – Base functionality
    - rules, data inheritance, events & notifications, relationships, queries & navigation, transactions, communications, persistence
  - – Interface to the external systems
  - – Interface to the supporting technology



Domain-Driven Design

- Domain-Driven Design – a way of thinking and a set of priorities, for accelerating software projects, which deal with complicated domains [Evans]
  - the primary focus should be on the domain and domain logic
  - complex domain designs should be based on a model
- Some techniques an practices of Domain-Driven Design
  - declarative design
  - intention revealing interfaces (fluent interfaces)
  - side-effect-free functions
  - assertions (explicit constraints)
  - conceptual contours (modules)
  - standalone classes (low coupling)
  - closure of operations (for value objects)
  - bounded context (explicit context)
  - context map (connecting models)
  - shared kernel (common subset of models)
  - anticorruption layer (interface between models)

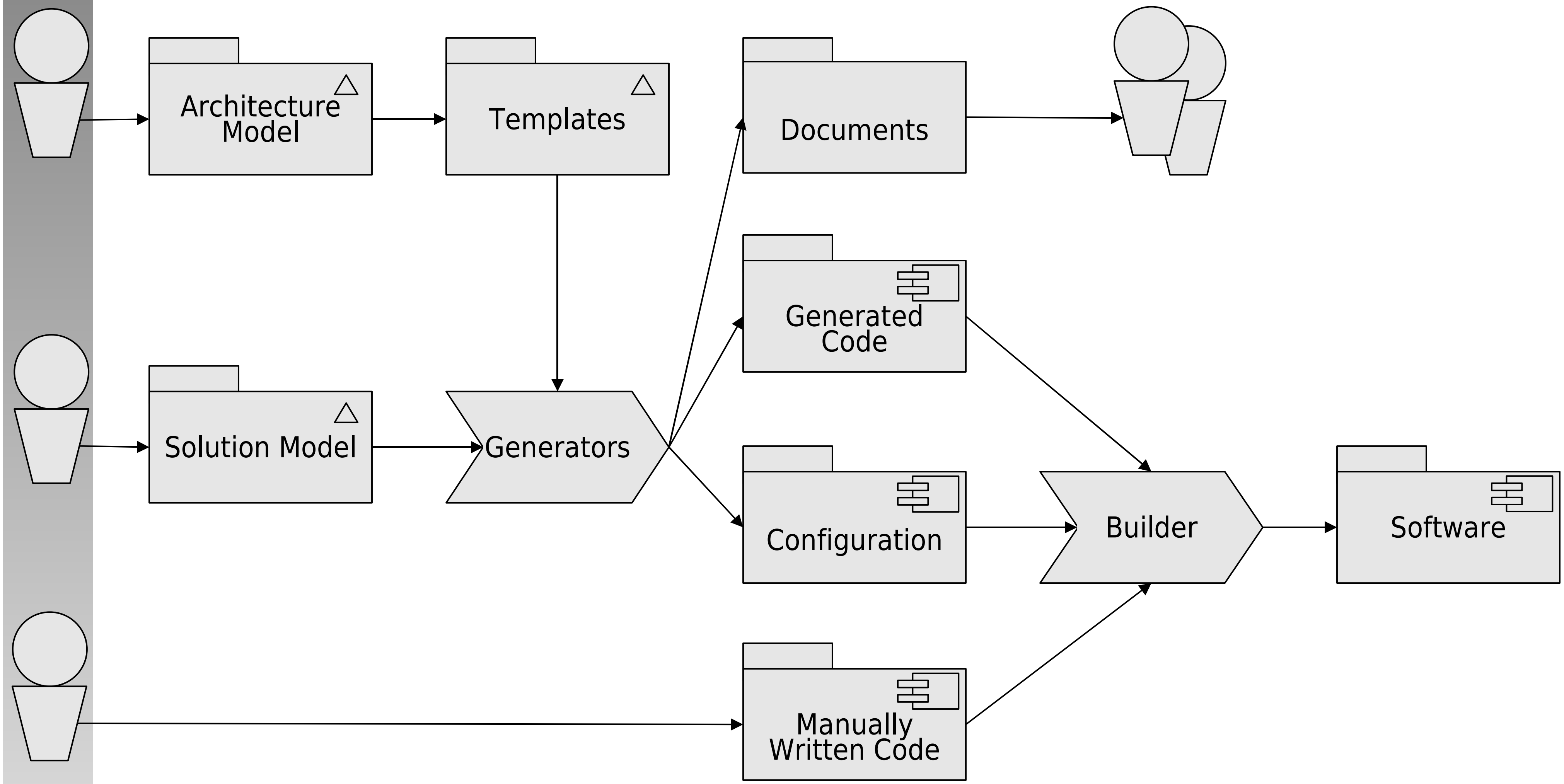
Relationships of Domain-Models



Content

- Introduction
        - Common Language – some Definitions
        - The Problem
        - Beginning (Excursion into the History)
- Models in Software Development
    - Direct Modeling
        - Convergent Engineering
        - Domain-Driven Design
    - Models as Primary Artifacts
        - Generative Programming
        - Domain Specific Languages
        - Model-Driven Development Methods
- Practical Aspects
        - Model Management
        - Best Practices
        - Examples
- Conclusions
- References

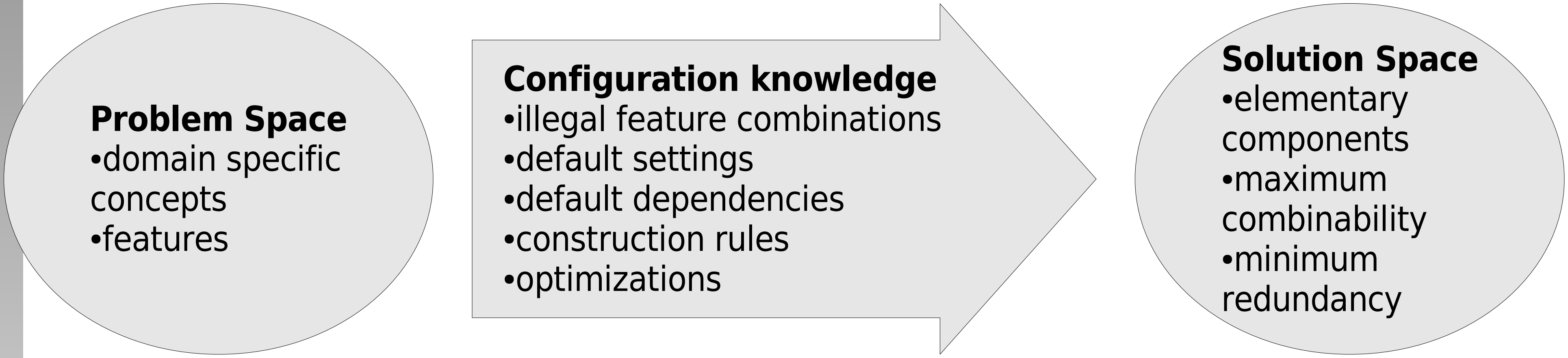
Models as Primary Artifacts



Model-Based Software Development

- Real-time and embedded systems
  - Model-Integrated Computing (MIC) and model-based software synthesis – (Vanderbilt Univ. (ISIS), 1993; Abbott et al., 1994)
  - Model-based development – (Mellor, 1995)
- Generative programming
  - GenVoca – (Batory, 1992)
  - Family-Oriented Abstraction, Specification, and Translation (FAST) – (Weiss, 1996; AT&T, Lucent, 1999)
- Software system families (a.k.a. product-lines)
  - Model-Based Software Engineering (MBSE) – (SEI, 1993)
- Integration and interoperability
  - Model-Driven Architecture (MDA) – (OMG, 2001)

Generative Programming



Problem Space
- domain specific concepts
- features

Configuration knowledge
- illegal feature combinations
- default settings
- default dependencies
- construction rules
- optimizations

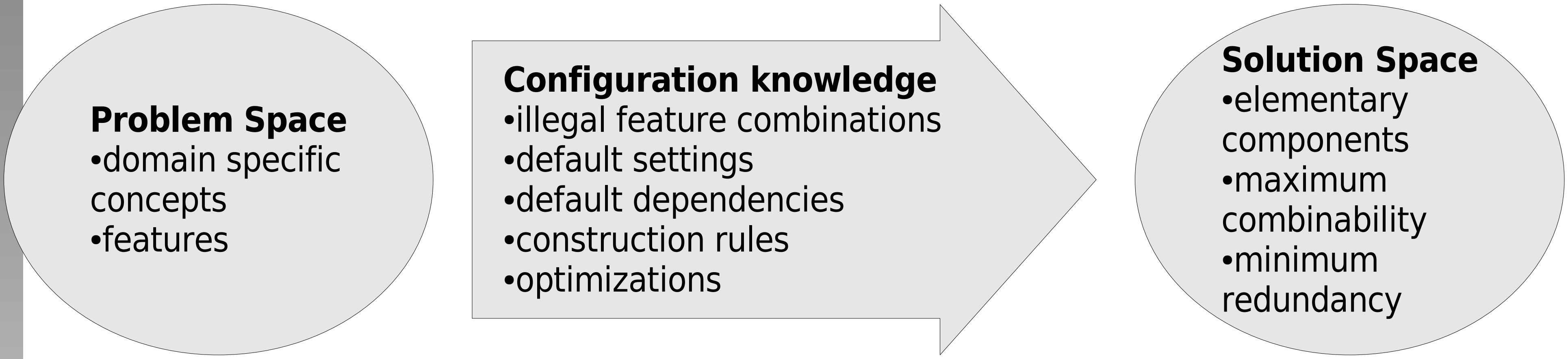
Solution Space
- elementary components
- maximum combinability
- minimum redundancy

Domain Specific Language (DSL)

Generator Reflection

Components + System Family Architecture

Generative Programming Technologies



Problem Space
- domain specific concepts
- features

Configuration knowledge
- illegal feature combinations
- default settings
- default dependencies
- construction rules
- optimizations

Solution Space
- elementary components
- maximum combinability
- minimum redundancy

DSL Technologies
- programming language
- extensible languages
- textual languages
- graphical languages
- interactive wizards
- any mixture of above

Generator Technologies
- simple traversal
- templates and frames
- transformation systems
- languages with meta-programming support
- extensible programming systems

Component Technologies
- generic components
- component models
- AOP approaches

Generator Technologies

- Model traversal
- Templates and frames
  - text with meta-instructions (referencing model)
    - retrieval of information from domain/problem model
    - conditional configuration of output
  - JSP, XSL, Velocity
- Transformation systems
  - operate on abstract syntax trees
    - rewrite rules
    - transformation procedures
  - DMS, XT, QVT
- Languages with meta-programming support
  - template meta-programming in C++

Domain Specific Languages

- Domain-Specific Languages (DSLs) – customized languages that provide a high-level of abstraction for specifying a problem concept in a particular domain

- Defining DSL
  - concrete syntax
    - specific representation of a DSL in a human-usable form
      - style: declarative | imperative
      - representation: textual, graphical, table, form(wizard), ...
  - abstract syntax
    - elements + relationships of a domain without representation consideration
  - semantics
    - the meaning of the phrases and sentences that the domain expert may express
      - static semantics: typing rules, truth value
      - dynamic semantics: evaluation rules, change in context
      - defined: formally | informally (interpreters, generators, transformers, ...)

DSL Technologies

Don't be too Clever!

- Internal DSLs
  - Built-in features of programming languages
    - C++ templates
    - Lisp Macros
  - Extendible languages
    - XML, Seed7
    - Ruby, Groovy, JavaScript, ...
  - Well-Designed APIs
- External DSLs
  - Textual languages
  - Graphical languages
    - UML, MetaCASE
  - Interactive wizards

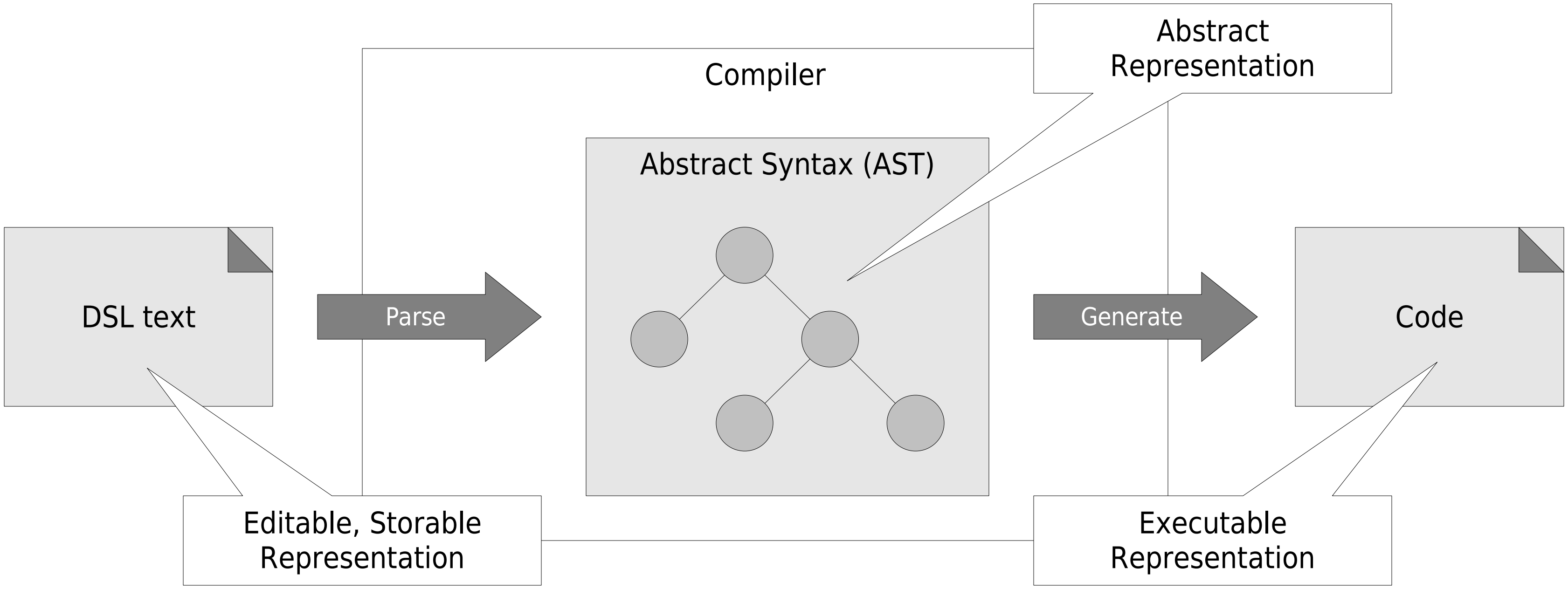
DSL Example $_1$

- Ojay (JavaScript internal DSL)

```
...
// Define some validation rules

    form('signup')
        .requires('username')   .toHaveLength({minimum: 6})
        .requires('email')      .toMatch(EMAIL_FORMAT, 'must be a valid email address')
        .expects('email_conf')  .toConfirm('email')
        .expects('title')       .toBeOneOf(['Mr', 'Mrs', 'Miss'])
        .requires('dob', 'Birth date').toMatch(/^\d{4}\D*\d{2}\D*\d{2}$/)
        .requires('tickets')    .toHaveValue({maximum: 12})
        .requires('phone')
        .requires('accept', 'Terms and conditions').toBeChecked('must be accepted');
...
```

DSL Implementation $_1$

- Compiler-Based



DSL Example $_2$

- Simple External DSL (yacc)

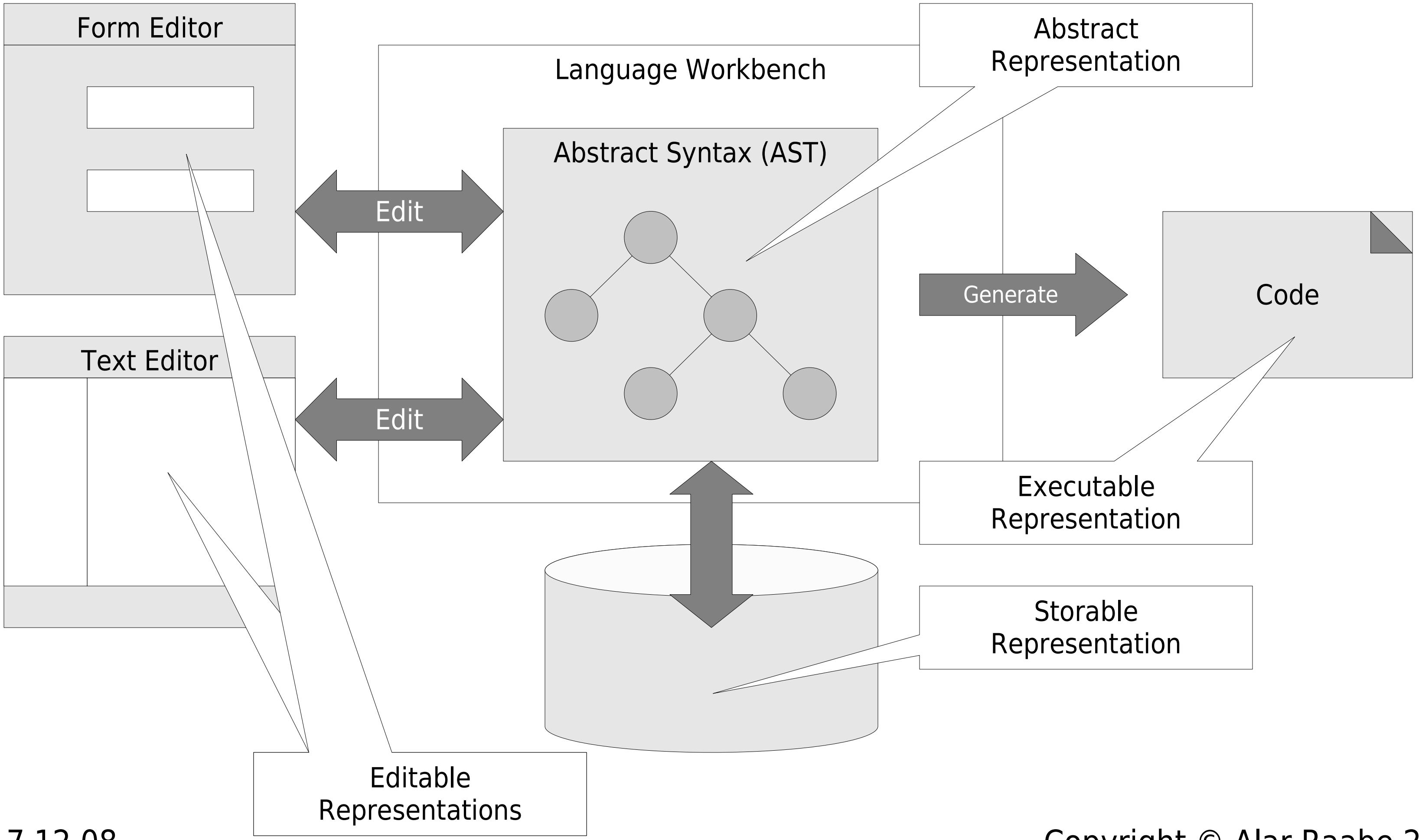
```
...
list:   /*empty */ | list stat '\n' | list error '\n' { yyerrok; } ;
stat:   expr { printf("%d\n",$1); } | LETTER '=' expr { regs[$1] = $3; } ;
expr:   '(' expr ')' { $$ = $2; } |
        expr '*' expr { $$ = $1 * $3; } | expr '/' expr { $$ = $1 / $3; } |
        expr '%' expr { $$ = $1 % $3; } |
        expr '+' expr { $$ = $1 + $3; } | expr '-' expr { $$ = $1 - $3; } |
        expr '&' expr { $$ = $1 & $3; } | expr '|' expr { $$ = $1 | $3; } |
        '-' expr %prec UMINUS { $$ = -$2; } |
        LETTER { $$ = regs[$1]; } | number ;
number: DIGIT { $$ = $1; base = ($1==0) ? 8 : 10; } |
        number DIGIT { $$ = base * $1 + $2; } ;
...
```

- Example

```
...
a = 10
b = 5
a + 4 * (b - 3)
...
```

DSL Implementation $_2$

- Language Workbench



DSL Example $_3$

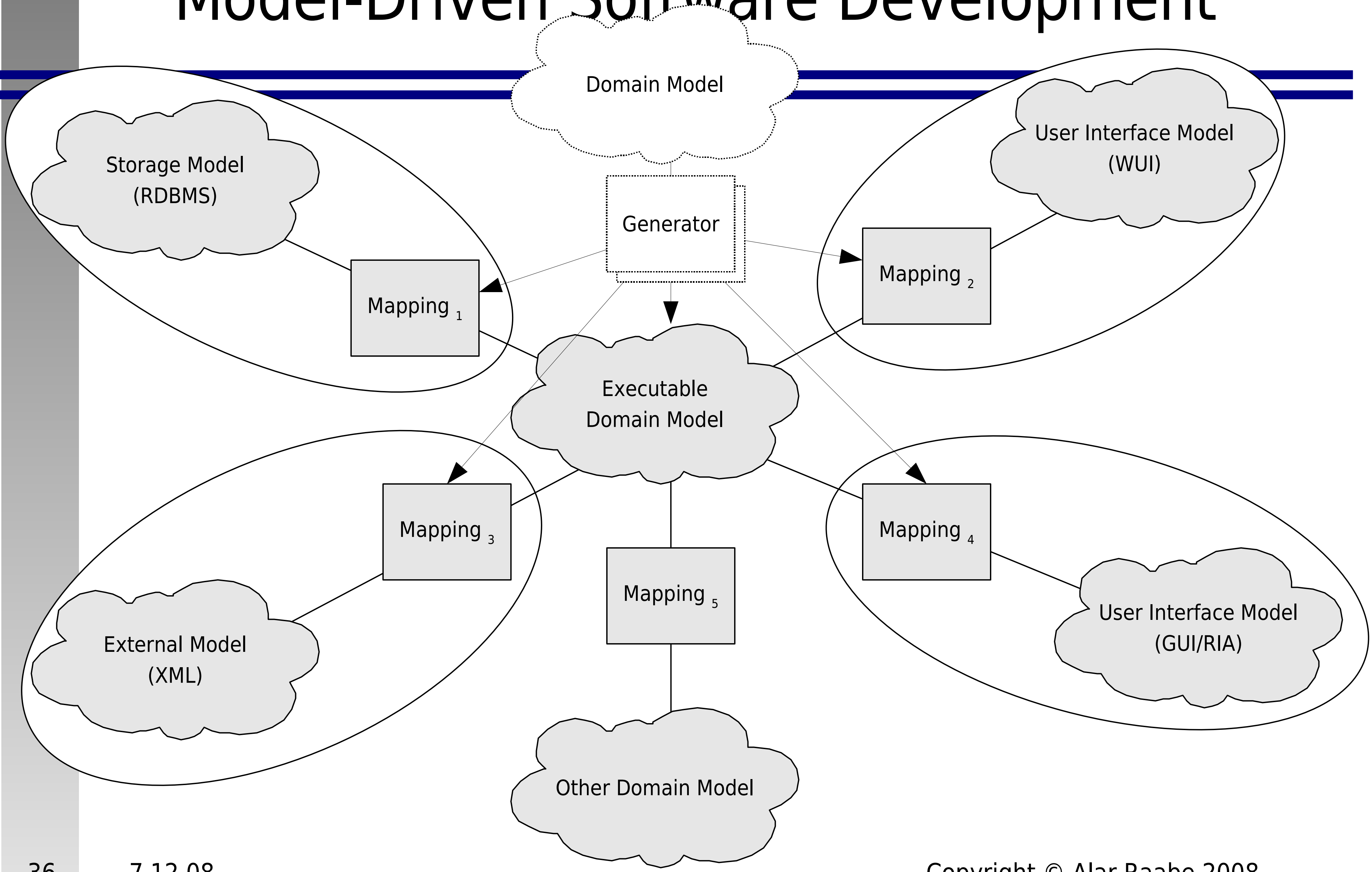
- xText (oAW)

```
Entity :
  "entity" name=ID ("extends" superType=[Entity])?
  "{"
     (features+=Feature)*
  "}";
Feature :
  Attribute | Reference;
Attribute :
  type=ID name=ID ";";
Reference :
  "ref" (containment?"+")? type=ID name=ID ("<->" oppositeName=ID)? ";";
```

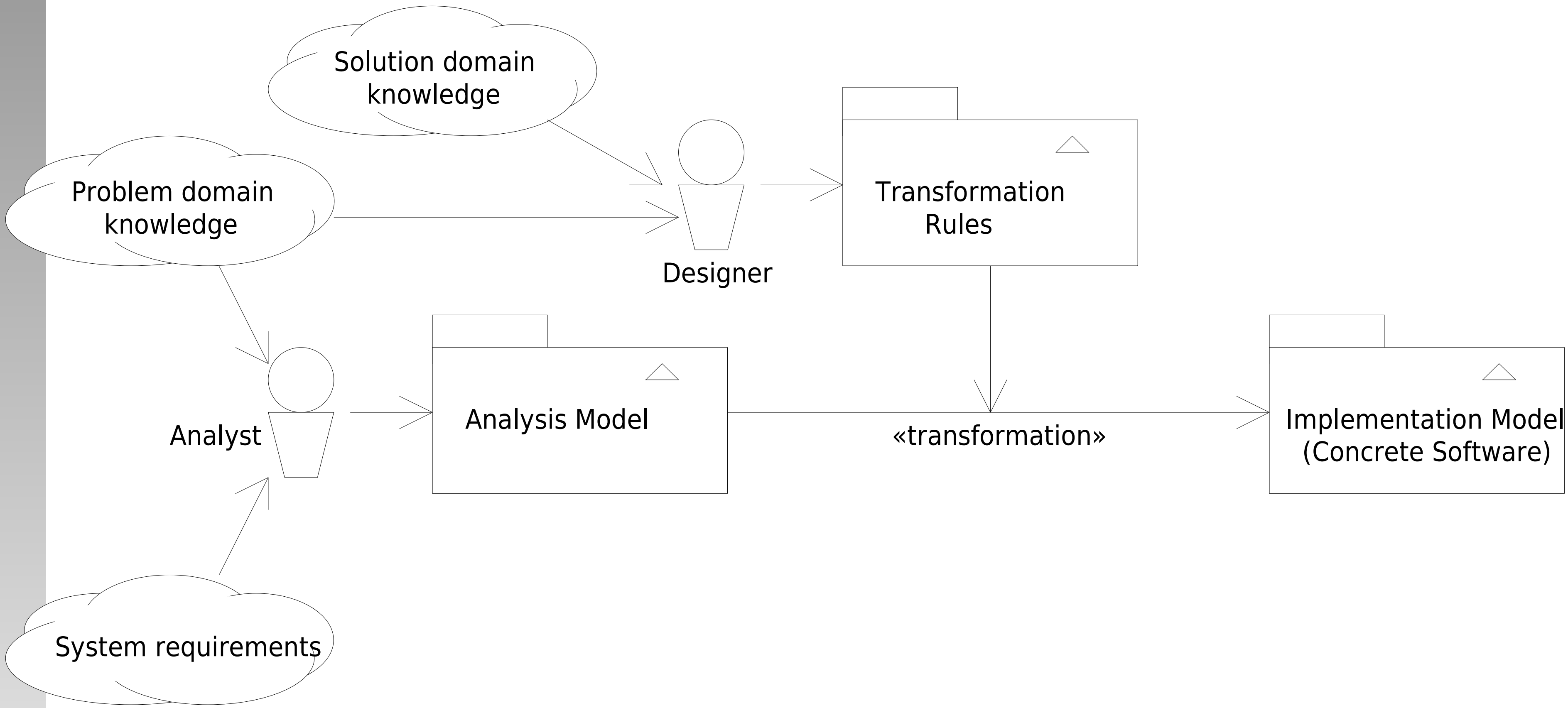
- Example

```
entity Customer {
  String  name;
  String  street;
  Integer age;
  Boolean isPremiumCustomer;
}
```

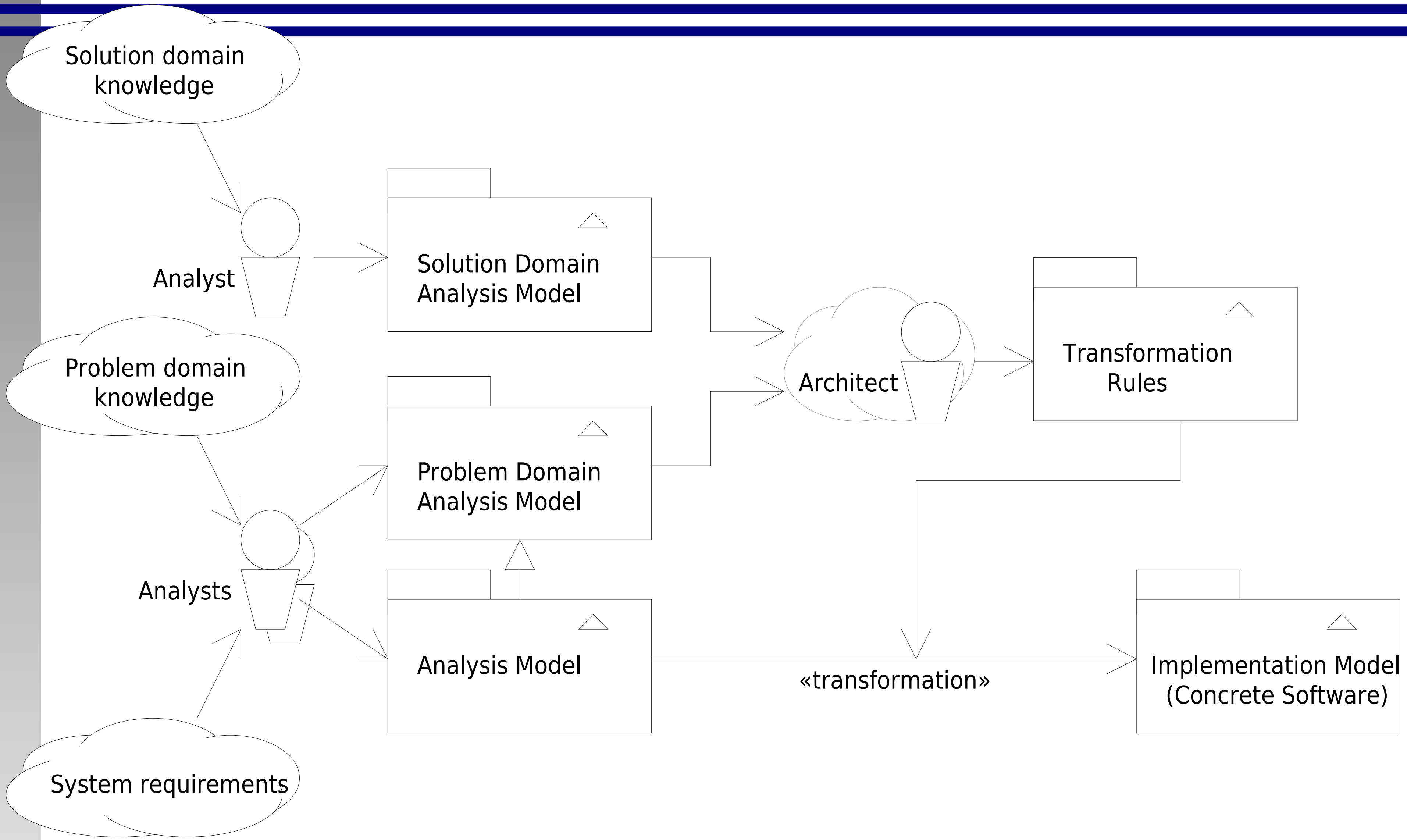
Model-Driven Software Development



Traditional MDSD Approach



Extended MDSD Approach



Solution domain knowledge

Analyst

Solution Domain Analysis Model

Problem domain knowledge

Problem Domain Analysis Model

Analysts

Analysis Model

System requirements

Architect

Transformation Rules

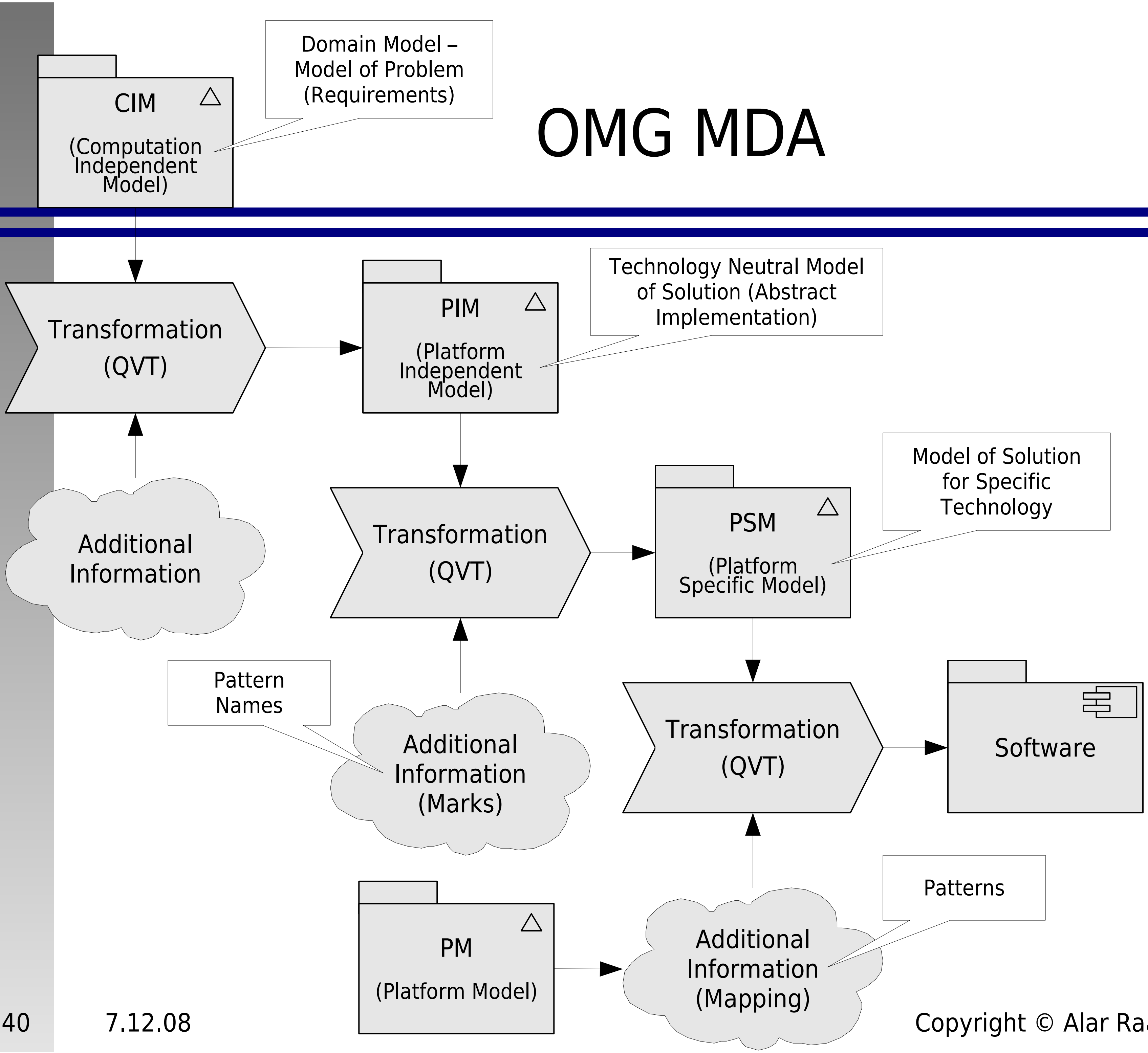
«transformation»

Implementation Model (Concrete Software)

Four Components of MDSD

- Models
  - Analysis and design meta-models
  - Reference models
- Architecture
  - Architecture style
  - Reference architecture
- Process
  - Generation rules
  - Process of application of generation rules
- Tools
  - Model manipulation tools
  - Generators

OMG MDA



CIM (Computation Independent Model)

Domain Model – Model of Problem (Requirements)

Transformation (QVT)

Additional Information

PIM (Platform Independent Model)

Technology Neutral Model of Solution (Abstract Implementation)

Transformation (QVT)

Pattern Names

Additional Information (Marks)

PSM (Platform Specific Model)

Model of Solution for Specific Technology

Transformation (QVT)

Software

PM (Platform Model)

Additional Information (Mapping)

Patterns

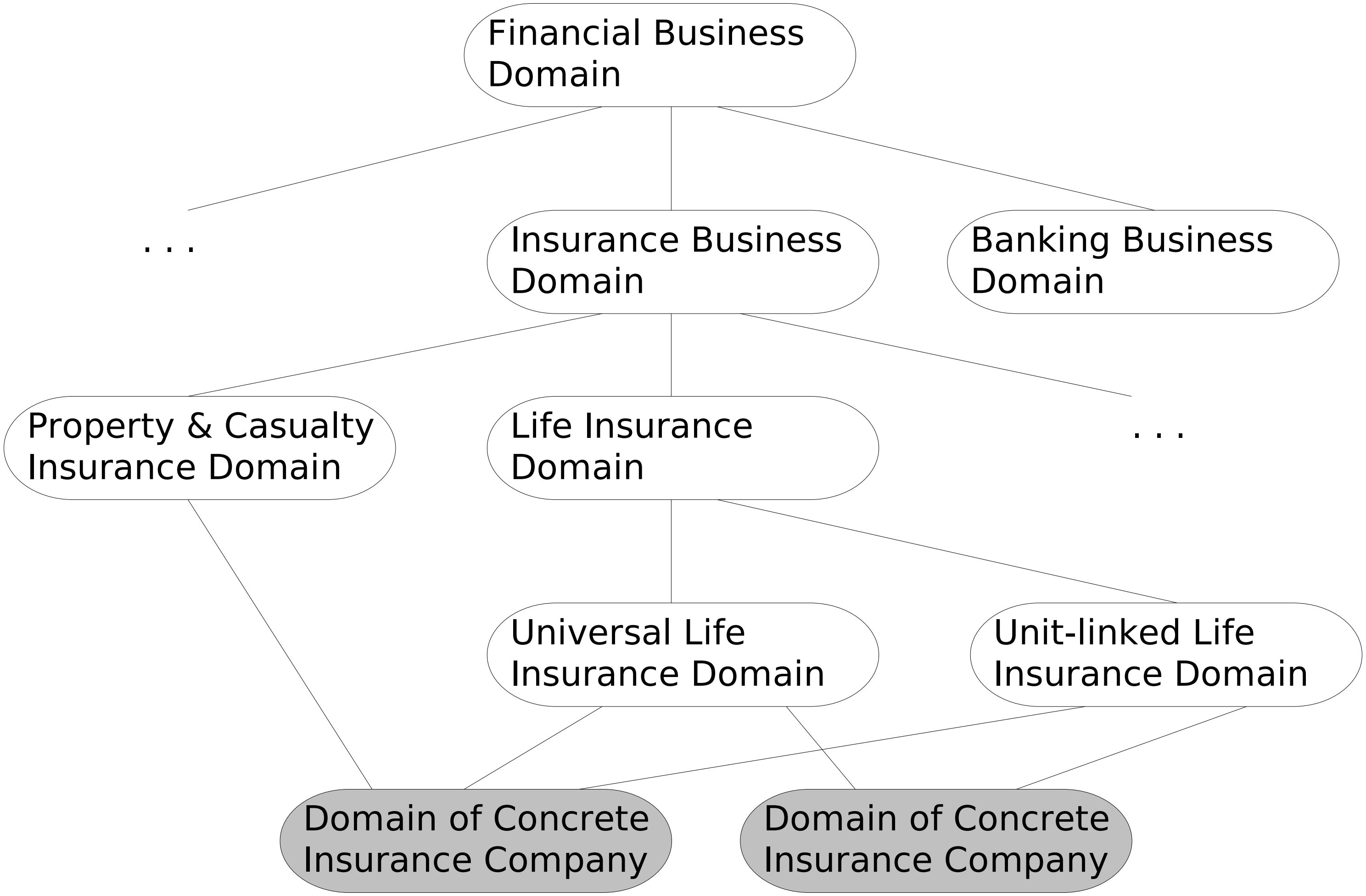
Content

- Introduction
    - Common Language – some Definitions
    - The Problem
    - Beginning (Excursion into the History)
- Models in Software Development
    - Direct Modeling
        - Convergent Engineering
        - Domain-Driven Design
    - Models as Primary Artifacts
        - Generative Programming
        - Domain Specific Languages
        - Model-Driven Development Methods
- Practical Aspects
    - Model Management
    - Best Practices
    - Examples
- Conclusions
- References

Model Management

- Relationships between Models
  - correspondence mappings between models
  - references to external models
  - “inheritance” – extension of models
- Operations on Models
  - editing models
    - graphical model editors
    - form-based model editors
    - text-based model editors
  - storing models
    - repository
    - source code control
    - embedding into code

Need for Combination of Models



Financial Business Domain

. . .

Insurance Business Domain

Banking Business Domain

Property & Casualty Insurance Domain

Life Insurance Domain

. . .

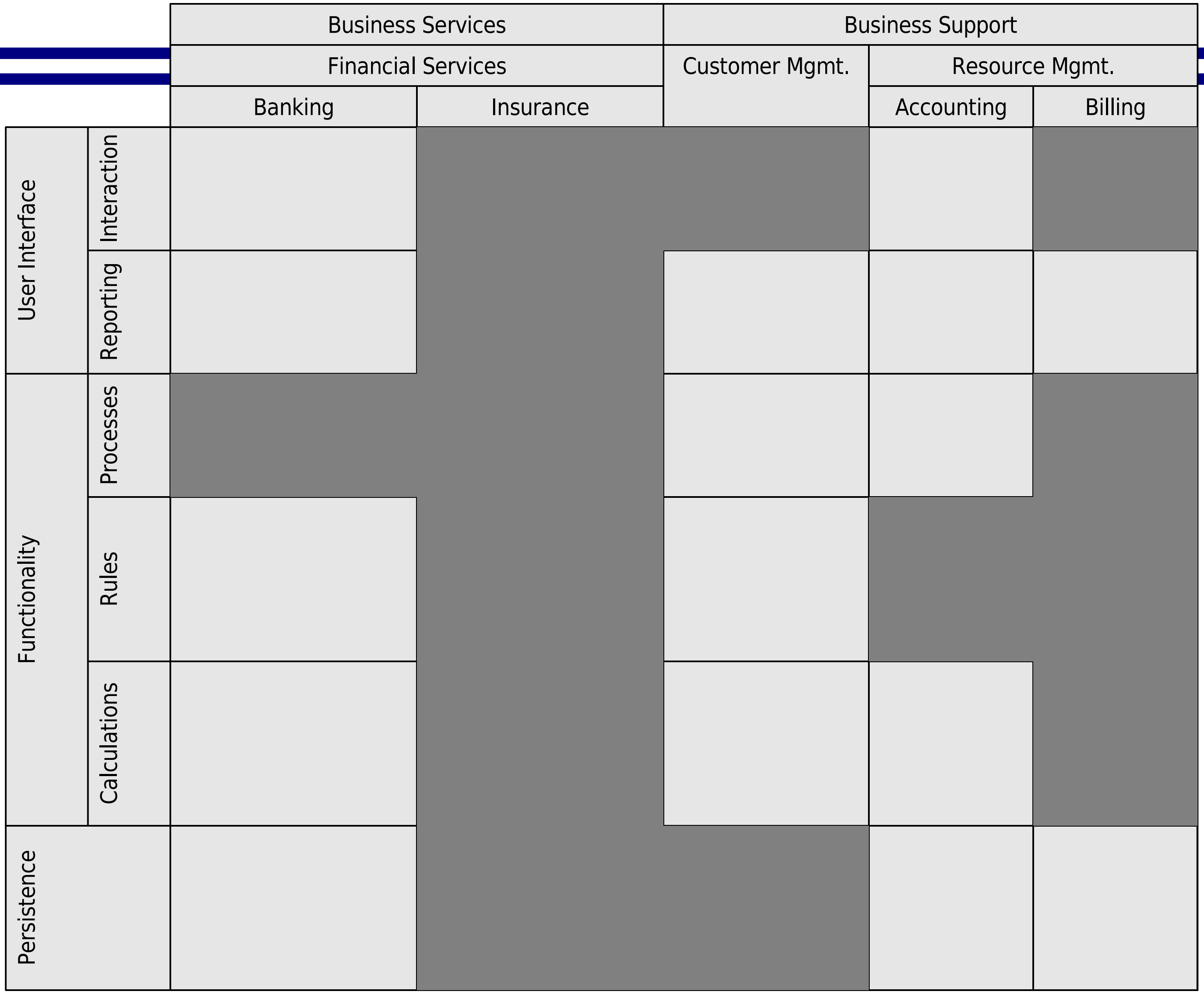
Universal Life Insurance Domain

Unit-linked Life Insurance Domain

Domain of Concrete Insurance Company

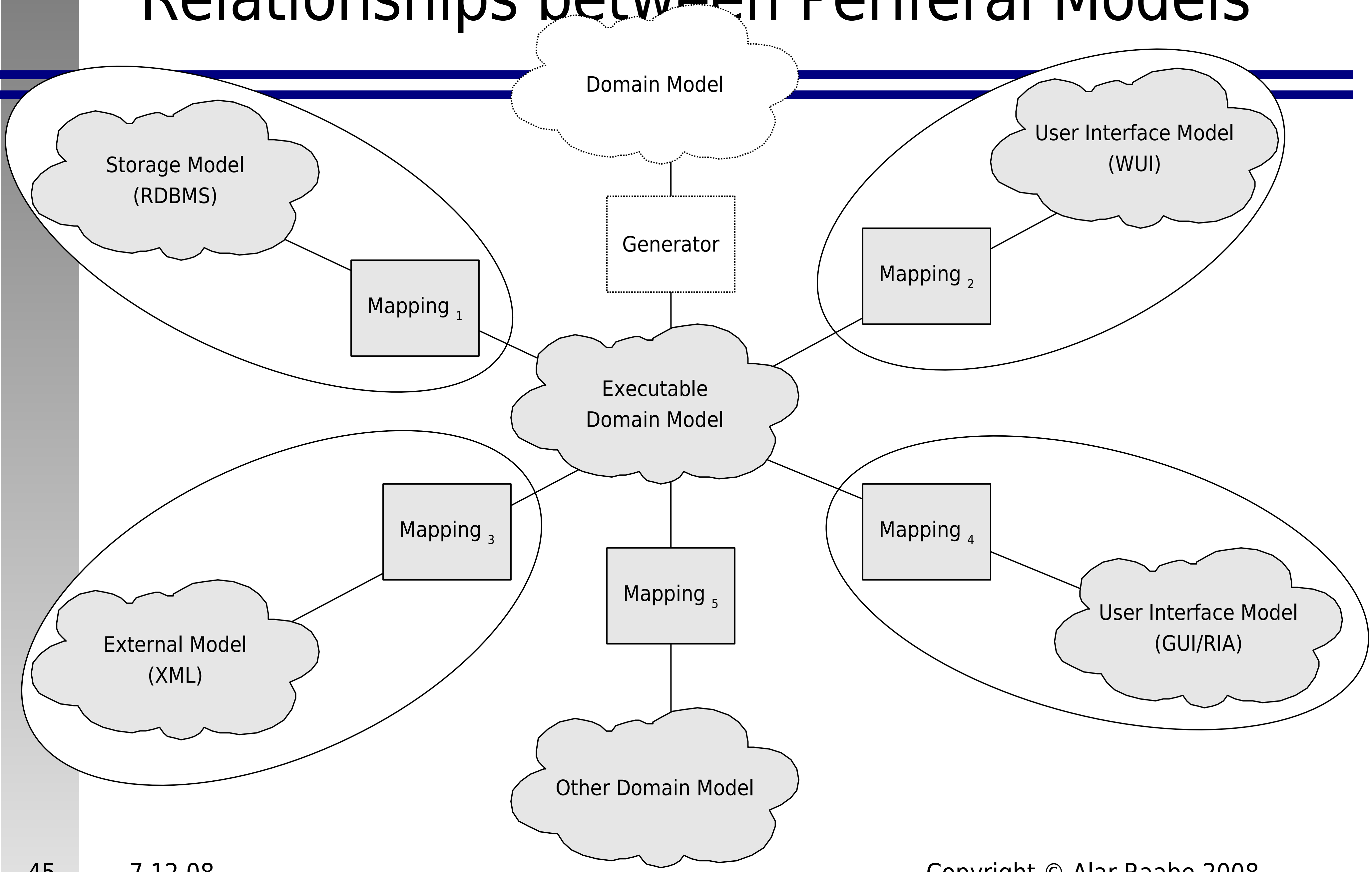
Domain of Concrete Insurance Company

Combine Domains for Specific System



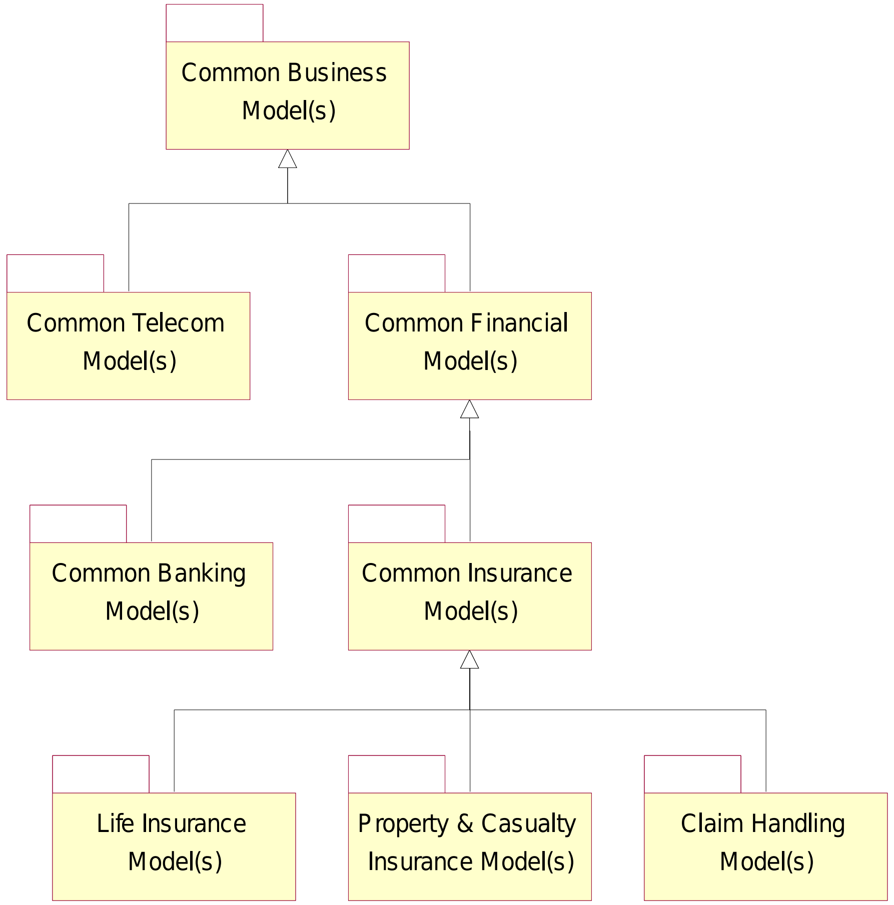
		Business Services		Business Support		
		Financial Services		Customer Mgmt.	Resource Mgmt.	
		Banking	Insurance		Accounting	Billing
User Interface	Interaction		▨	▨		▨
	Reporting		▨			
Functionality	Processes	▨	▨			▨
	Rules		▨		▨	▨
	Calculations		▨			▨
Persistence			▨	▨		

Relationships between Periferal Models



Relationships between Domain Models

- Domain reference model
  - developed during the domain analysis
  - represents formalized knowledge about domain
- Domain models form a (inheritance) hierarchy
  - where models of more specific domains inherit from the models of more generic domains
- Extendability of domain model
  - must be planned during the development of domain model



Best Practices

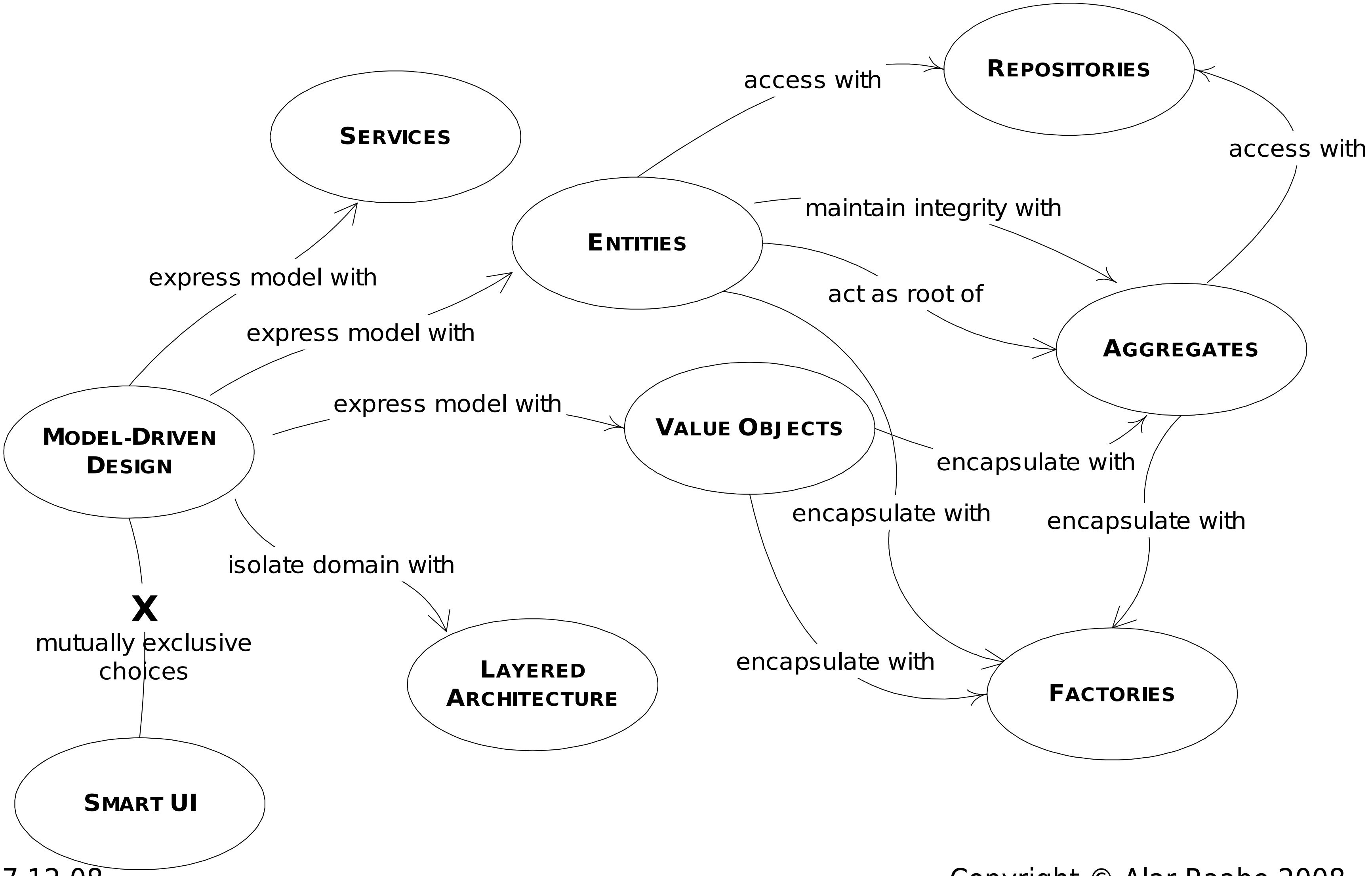
- Domain-Driven Design
  - Domain-Driven Development Best Practices
  - Domain-Driven Design Patterns

- Model-Driven Software Development
  - Model-Driven Software Development Approach
  - Model-Driven Software Development Best Practices

Domain-Driven Design Best Practices 1

- Use the Domain Model as **Ubiquitous Language**
- Design Part of the System to Reflect Domain Model – Avoid Divide between Analysis and Design
  - Domain Model is Constrained to Support Efficient Implementation
- Express Domain Model in Code – Hands-On Modeling
- Building Blocks

Express Domain Model in Software



Repositories

access with

Services

access with

Entities

maintain integrity with

express model with

act as root of

express model with

Aggregates

Model-Driven Design

express model with

Value Objects

encapsulate with

encapsulate with

encapsulate with

isolate domain with

X

mutually exclusive choices

Layered Architecture

encapsulate with

Factories

Smart UI

Domain-Driven Design Best Practices 2

- Express Model with Services, Entities and Value Objects
- Isolate Domain with Layered Architecture
  - Presentation Layer
  - Application Layer
  - Domain Layer
  - Infrastructure Layer
- Maintain Integrity with Aggregates
- Entities act as roots of Aggregates
- Access Entities and Aggregates with Repositories
- Encapsulate Value Objects with Aggregates
- Encapsulate Entities, Value Objects and Aggregates with Factories

MDSD Best Practices 1

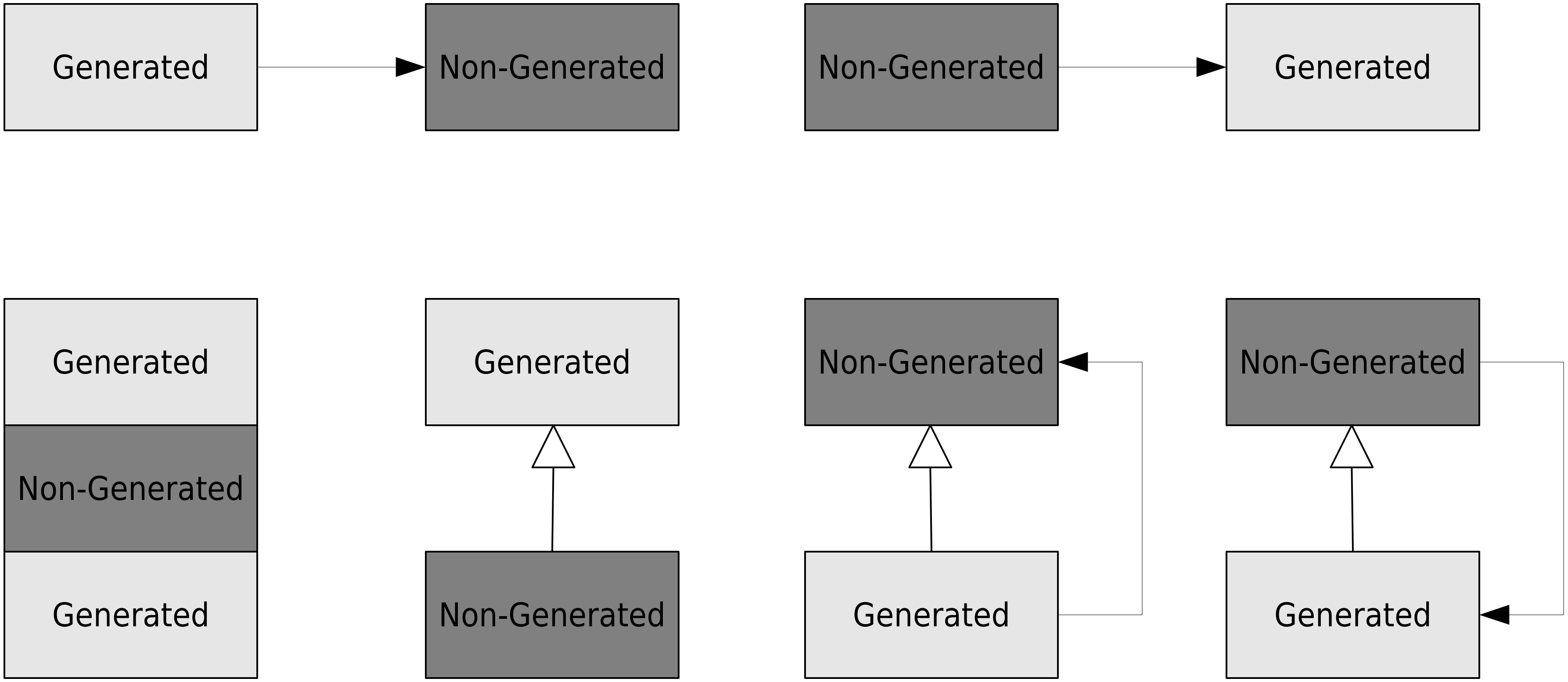
- Separate the Generated and Manually Created Code
  - protected regions (this requires checking generated code into the revision control system)
  - separate directory (e.g. src-gen)
  - language mechanisms (e.g. subclassing/inheritance, wrapping/containment, aspects, ...)
- Don't Manage Generated Code in Revision Control System
  - exception when using protected regions
  - exception when generator can't be integrated with build
- Integrate the Generator/Generation into the Build Process
  - generation phase must be added before the compilation phase

MDSD Best Practices $_2$

- Use the Native Techniques of Target Platform for Separating Generated and Manually Created Code
  - object languages
    - subclassing/inheritance (e.g. 3 levels for framework, generated and manual code)
    - wrapping/containment (delegation)
  - aspect languages
    - aspects/pointcuts (weaving)
  - procedural languages
    - preprocessing (e.g. includes)
    - libraries

- Generate Clean and Readable Code
  - code is primarily meant for humans
  - follow coding styles used for manually written code
  - generate comments that identify generated code and describe the used (parts of) source model
  - use code formatter

Combining Generated and Non-Generated Code



MDSD Best Practices $_3$

- Use the Compiler (to Guide the Developer)
  - let compiler check the constraints for manually written code (e.g. overriding of mandatory methods)
  - generate dummy code as example for manually written code
- Use Meta-Model as Ubiquitous Language
  - use consistent terminology that connects generated code with other parts of project
  - verify the adequacy of DSLs through constant usage of metamodel concepts
- Develop DSLs Incrementally
  - DSLs should be developed as understanding grows
  - DSLs are public interfces – should be developed and evolved like APIs
  - provide facilities for migrating old models to new metamodel (e.g. model transformation)

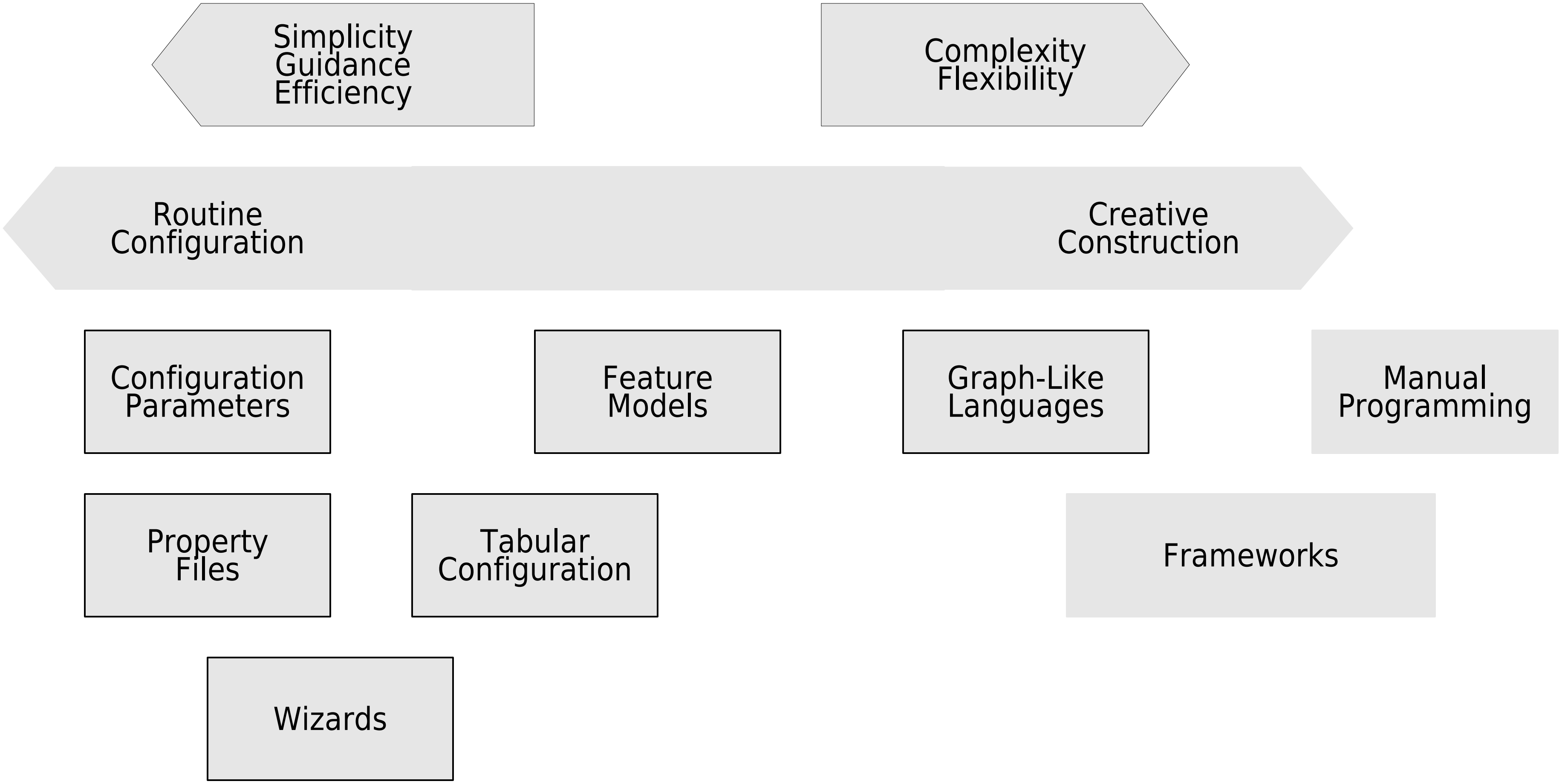
MDSD Best Practices 4

- Develop Model Validation (Iteratively)
  - semantics cannot be represented by metamodel alone (it describes only static aspects of model – structure)
  - constraints representing semantics should be added incrementally
  - integrate model validation into build process
- Test the Generator(s) (using Reference Model)
  - use reference (test) models as unit tests to test the generator
  - generate unit tests for combination of generated and manually created code
- Select Suitable Technology – Avoid too Complex Meta-Models
  - define core abstractions clearly and expandable
  - models should be quickly editable
  - turnaround (model ⟶ generate ⟶ execute) should be quick
  - avoid overly complex metamodels (like UML)

MDSD Best Practices $_5$

- Encapsulate UML (and other Complex Meta-Models)
  - transform complex metamodels into simpler metamodels targetted for specific domains
  - formulate domain specific constraints on simpler metamodels
- Use Graphical and Textual Syntax Correctly (to Support Modeller)
  - don't overburden model with details – use implicit knowledge
  - compromise between compactness and comprehensiveness
- Use Configuration by Exception
  - use defaults for normal configurations (e.g. only specify the exceptions)
  - remember, that defaults become the part of interface (API)

Configuration vs. Construction



Simplicity
Guidance
Efficiency

Complexity
Flexibility

Routine
Configuration

Creative
Construction

Configuration
Parameters

Feature
Models

Graph-Like
Languages

Manual
Programming

Property
Files

Tabular
Configuration

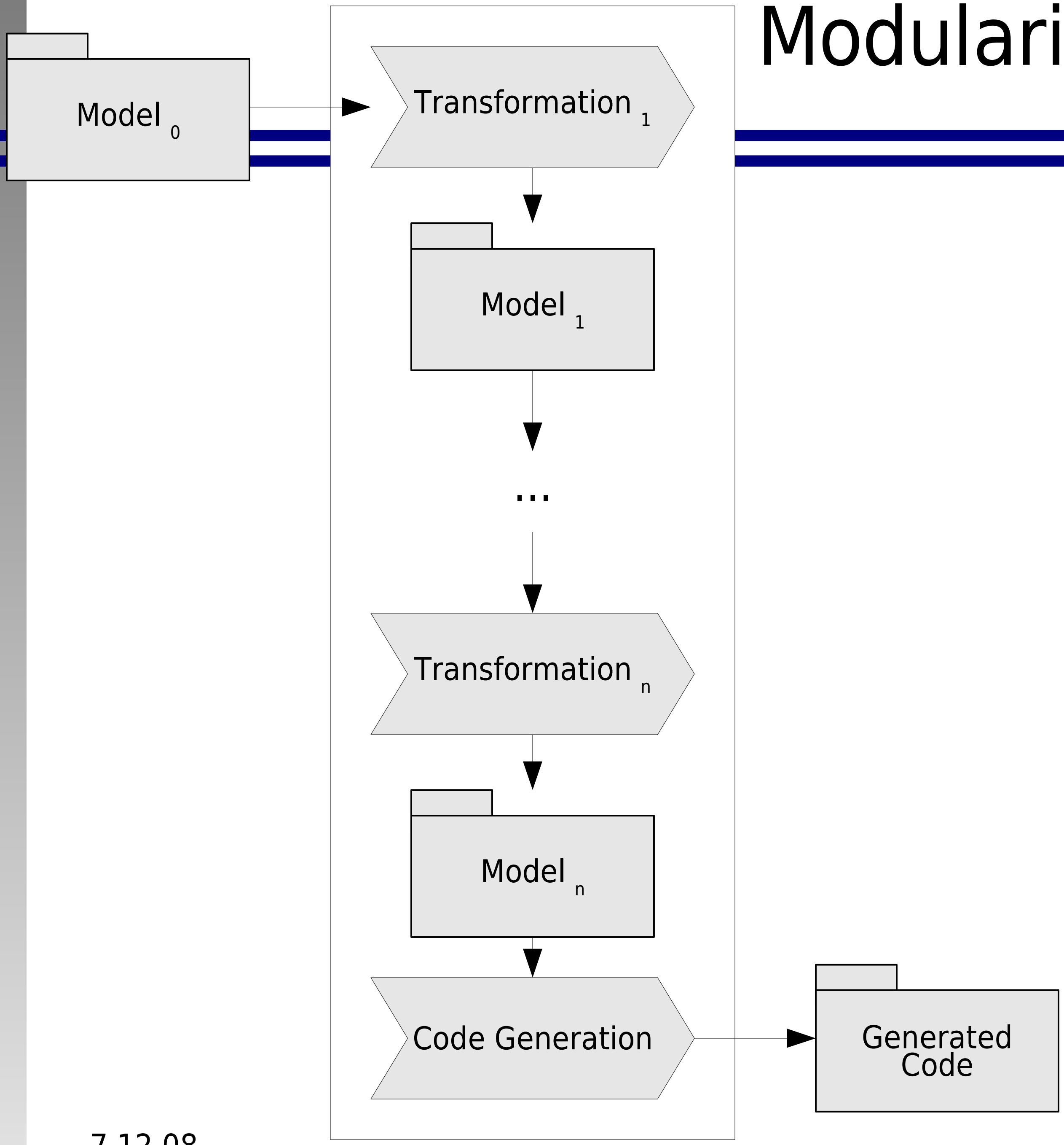
Frameworks

Wizards

MDSD Best Practices 6

- Teamwork Loves Textual DSLs
  - use exclusive locking for graphical models
  - if possible, use both textual and graphical DSL (both representations of same model)
- Use Model Transformations to Reduce Complexity
  - divide the step between source model and code into several transformation steps to fight complexity
- Generate towards a Comprehensive Platform – Keep Translation Steps as Small as Possible
  - develop domain specific platforms to reduce the complexity of generators

Modularize Generator



Model $_0$

Transformation $_1$

Model $_1$

...

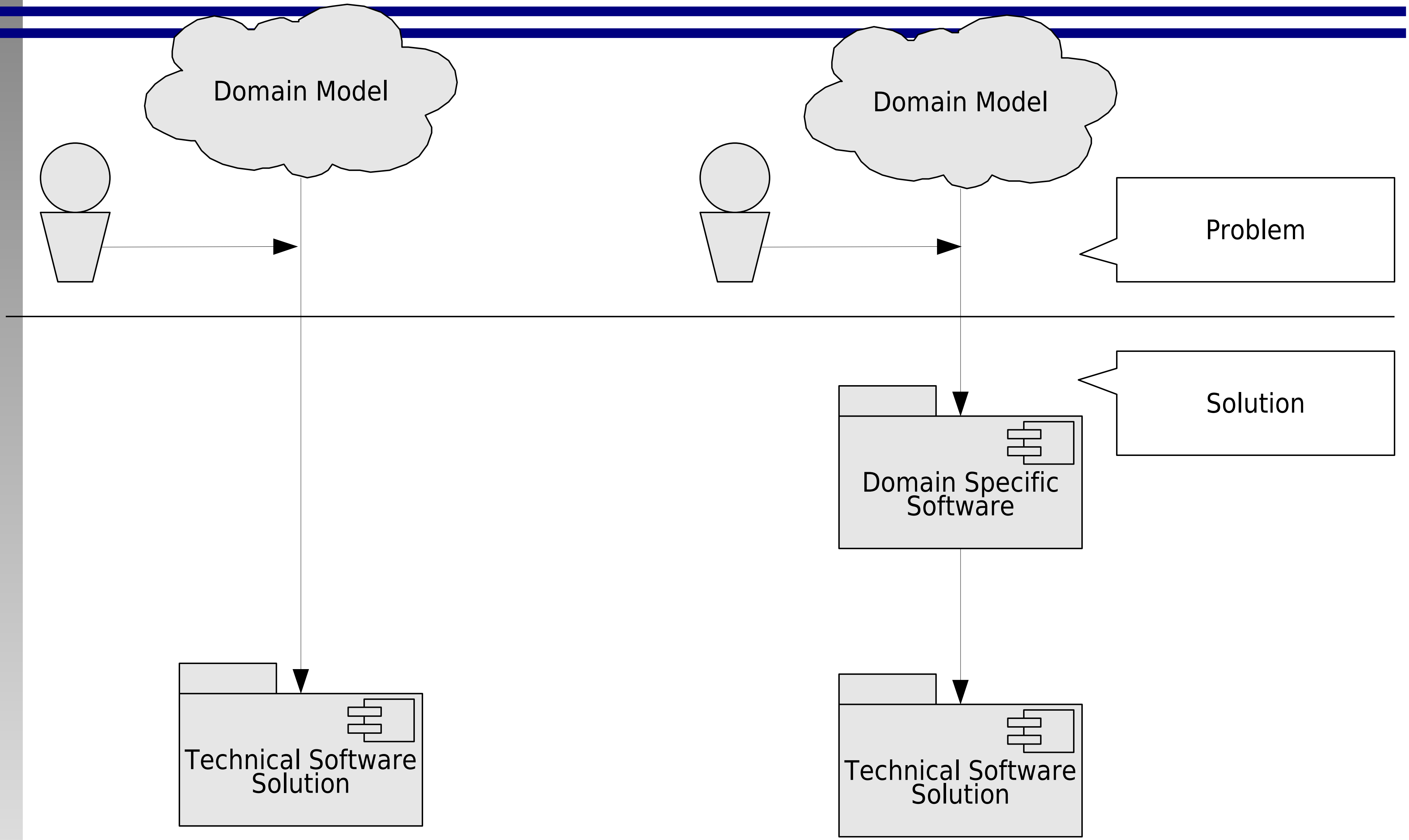
Transformation $_n$

Model $_n$

Code Generation

Generated Code

Use Rich Domain-Specific Platform



MDSD Best Practices 7

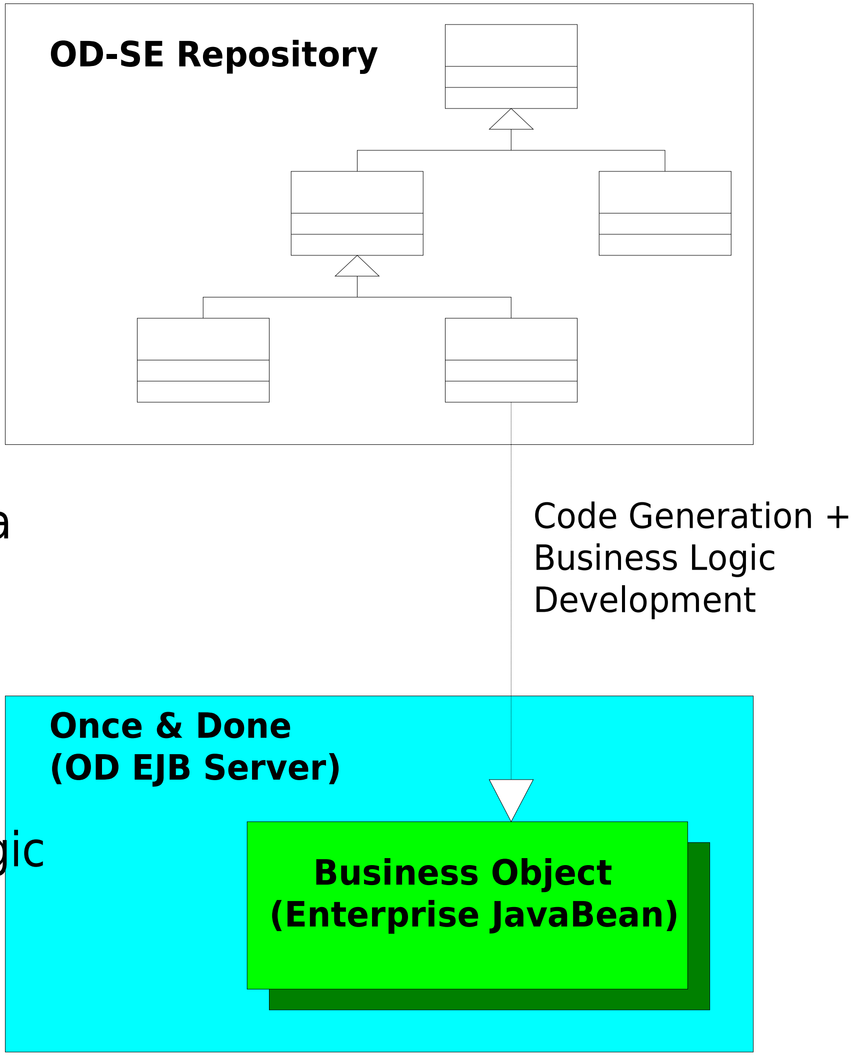
- Many Small DSLs – Concentrate on the Task
  - swiss army knife is nice as present, but specialised tools are used for serious work
  - *divide et impera* – models should be modular
- Don't Reverse Engineer – Model is Primary Artifact
  - all changes should be done in model, and then all derived artifacts should be regenerated
- Regenerate Frequently
  - include generation into continuous build process
  - frequent regeneration ensures compliance with model and architectural constraints (embedded into generator)

Examples

- Example of Model-Driven Development in Insurance
  - Once & Done – a model-driven technology for insurance systems product-line
- Example of Model-Driven Development in Banking
  - RISLA – a DSL for credit products
  - MLFi – a DSL for financial instruments and contracts

OD Process

- Beginning
- Anaysis
    - Business Domain Analysis
    - Modeling Domain Objects
    - Modeling Insurance Products
- Design
    - Refinement of Analysis Models
    - Design of the Database Schema
    - Design of the User Interface
    - Design of the Printouts
- Implementation
    - Generation of Code
    - Implementation of Business Logic
    - Installation of Business Objects into the Base System
- Finalisation



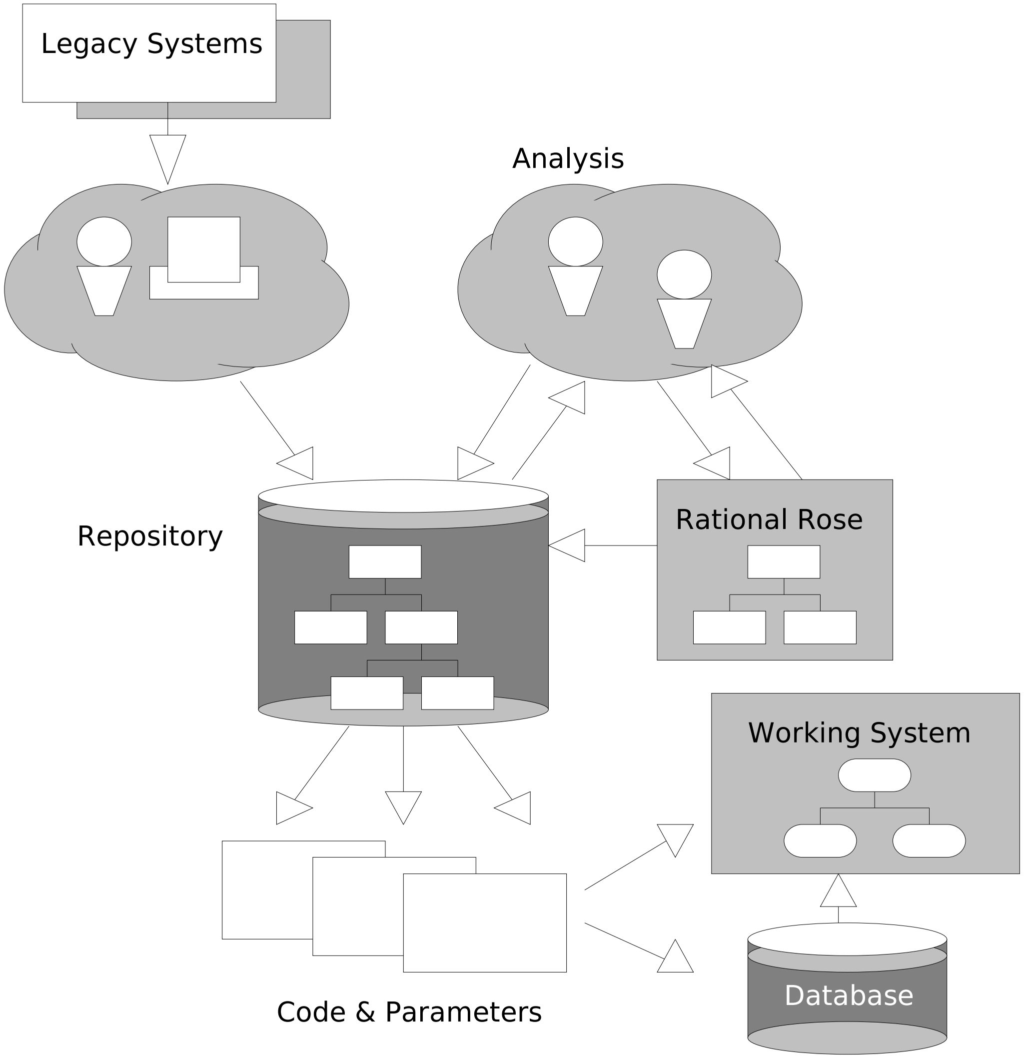
OD-SE Repository

Code Generation + Business Logic Development

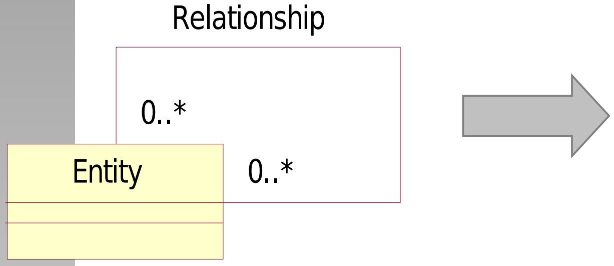
Once & Done (OD EJB Server)

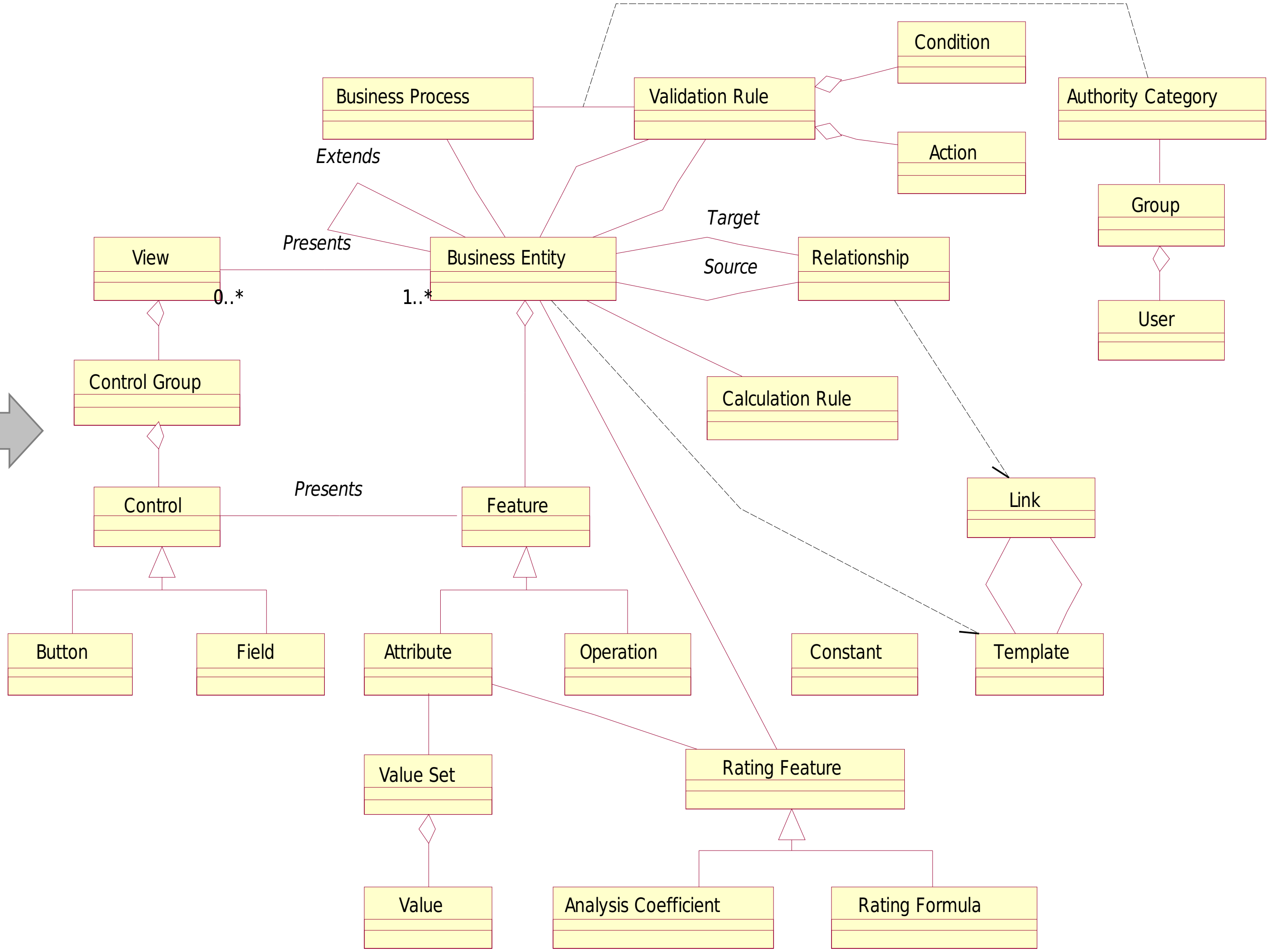
Business Object (Enterprise JavaBean)

Overview of OD Software Process

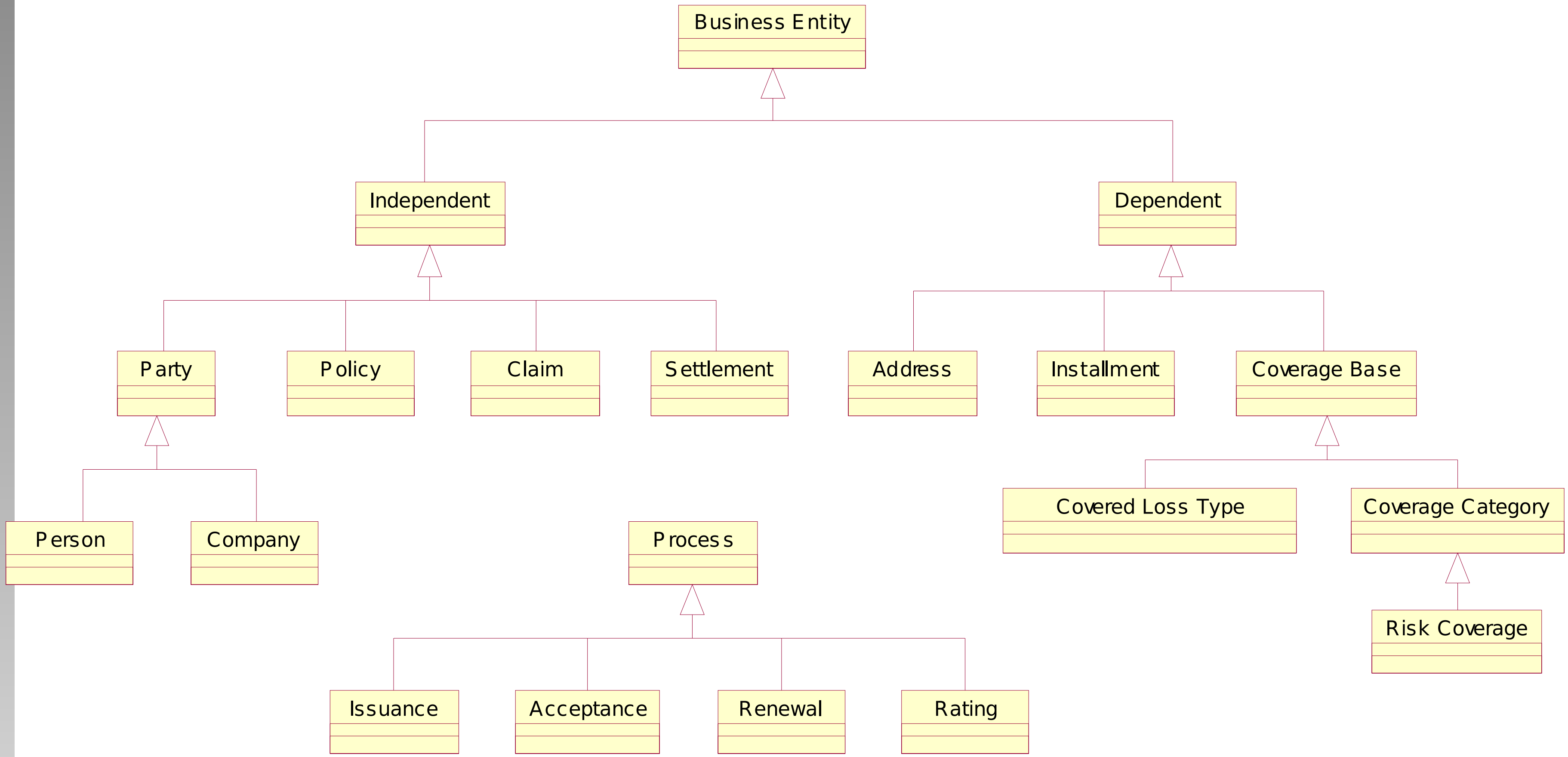


Extending the Meta-model

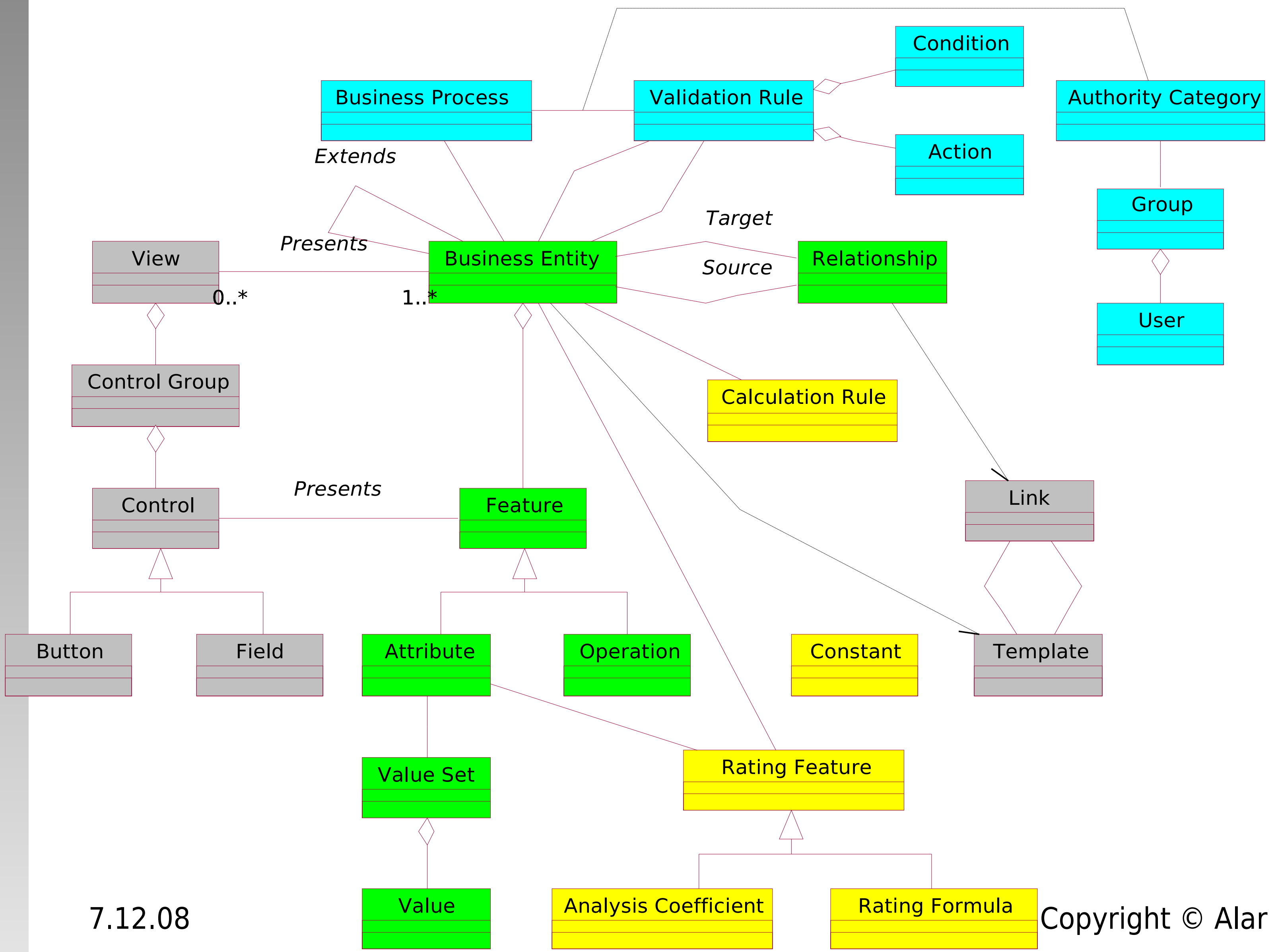




Insurance Domain Model

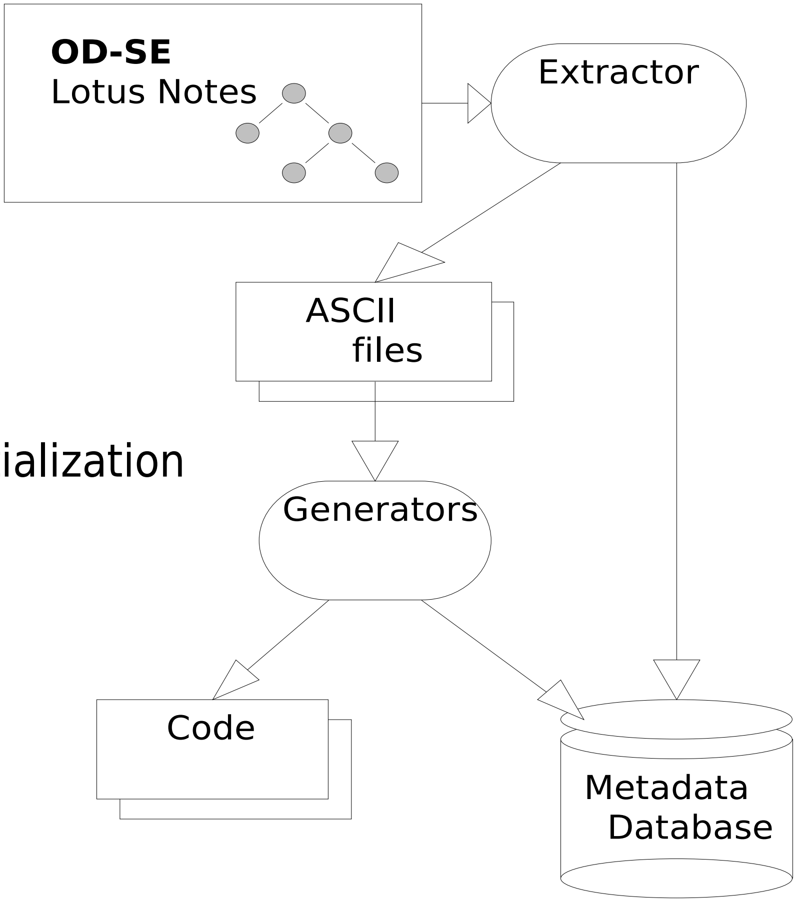


Extended OOA/OOD Meta-Model



Sample Generation Process (OD)

- Exported files
  - ➨ Entities, Attributes, DB Tables
- Entity specific files
  - ➨ Interface Definitions (.idl)
  - ➨ Class Implementations (.cpp)
  - ➨ Utility Macros
- Module specific files
  - ➨ Module Definition, Makefile, Module Initialization
- System-wide files
  - ➨ OD Metadata files
  - ➨ POS Metadata files
  - ➨ DB Tables Creation Script
  - ➨ OD Desktop Metadata files
  - ➨ Rating parameters



Example of Generation Template

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE ClassTemplate SYSTEM "class.dtd">
<ClassTemplate Type="Java">

package <PackageName Target="EJBServer"/>;
import javax.ejb.*;
<BasePackages>
import <PackageName Target="EJBServer"/>.*;
</BasePackages>

<Preserve>
// begin of user imports

// end of user imports
</Preserve>

...
```

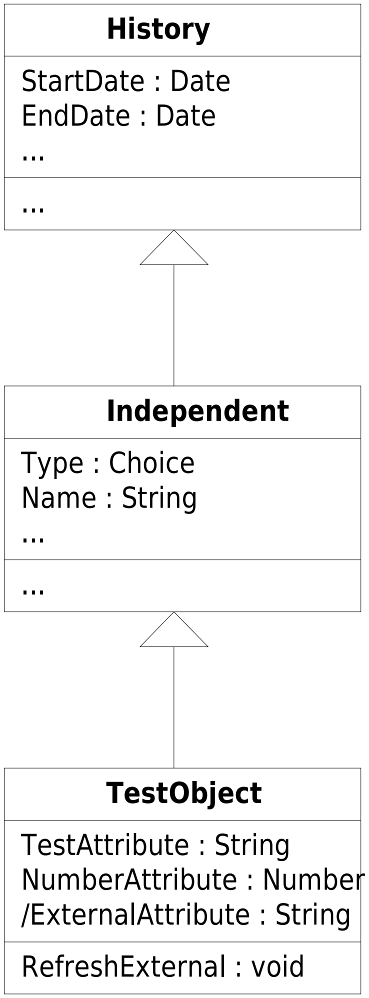
Example of Generation Template

```
public class <Name/>Bean
       extends <MainParent/>Bean {
<Preserve>
  // -=-=- Begin of user code
  // -=-=- End of user code
</Preserve>

<Attributes>
  public <Type/> <Name/>;
</Attributes>
  protected javax.ejb.EntityContext ctx;
<Methods>
  public <Type/> <Name/>(<Args Separator=", "><Type/> <Name/></Args>)
    throws java.rmi.RemoteException {
<Preserve>
    // Begin of code for <ClassName/> method <Name/>
    <MethodDummyReturns/>
    // End of code for <ClassName/> method <Name/>
</Preserve>
  }
</Methods>
...
```

Example of Generation Template

```
<Methods Override="Yes">
  public <Type/> <Name/>(<Args Separator=", "><Type/> <Name/></Args>)
    throws java.rmi.RemoteException {
<Preserve>
    // Begin of code for <ClassName/> method <Name/>
    <MethReturnUnlessVoid/> super.<Name/>(<Args Separator=",
"><Name/></Args>);
    // End of code for <ClassName/> method <Name/>
</Preserve>
  }

</Methods>

<Methods Inherited="Interface">
  public <Type/> <Name/>(<Args Separator=", "><Type/> <Name/></Args>)
    throws java.rmi.RemoteException {
    <MethReturnUnlessVoid/> super_<OriginalClassName/>.<Name/>(
      <Args Separator=", "><Name/></Args>);
  }
</Methods>
...
</ClassTemplate>
```

Content of Repository



History

StartDate : Date
EndDate : Date
...

...

Independent

Type : Choice
Name : String
...

...

TestObject

TestAttribute : String
NumberAttribute : Number
/ExternalAttribute : String

RefreshExternal : void

Result of Generation

```
package istest.server.ejb;

import javax.ejb.*;

// begin of user imports
// end of user imports
```

Extension Point

```
public class ISDTestObjectBean
       extends ISDIndependentBean {
  // -=-=- Begin of user code
  // -=-=- End of user code
```

Extension Point

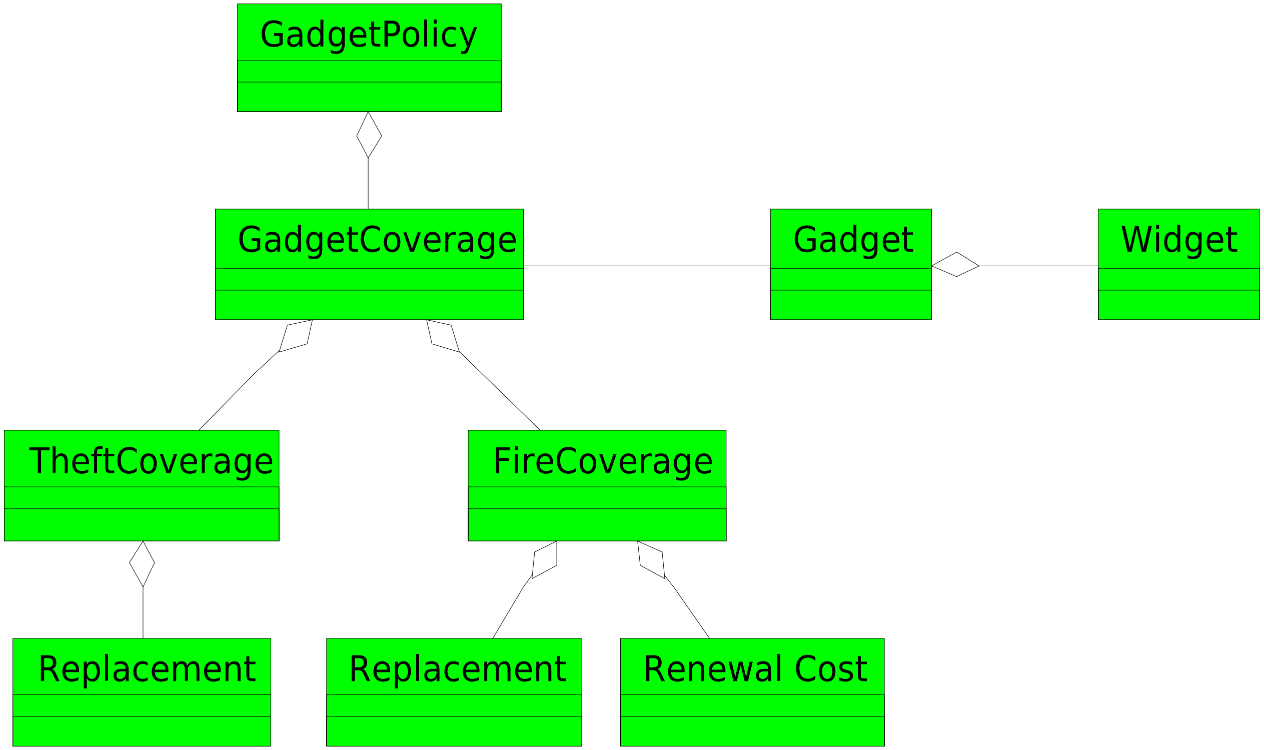
```
  public String extattribute;
  public String testattribute;
  public double numberattribute;
  protected javax.ejb.EntityContext ctx;
  public void CLRefExt()
    throws java.rmi.RemoteException {
    // Begin of code for ISDTestObject method CLRefExt
    // theDO.CLRefExt();
    // End of code for ISDTestObject method CLRefExt
  }
```

Extension Point

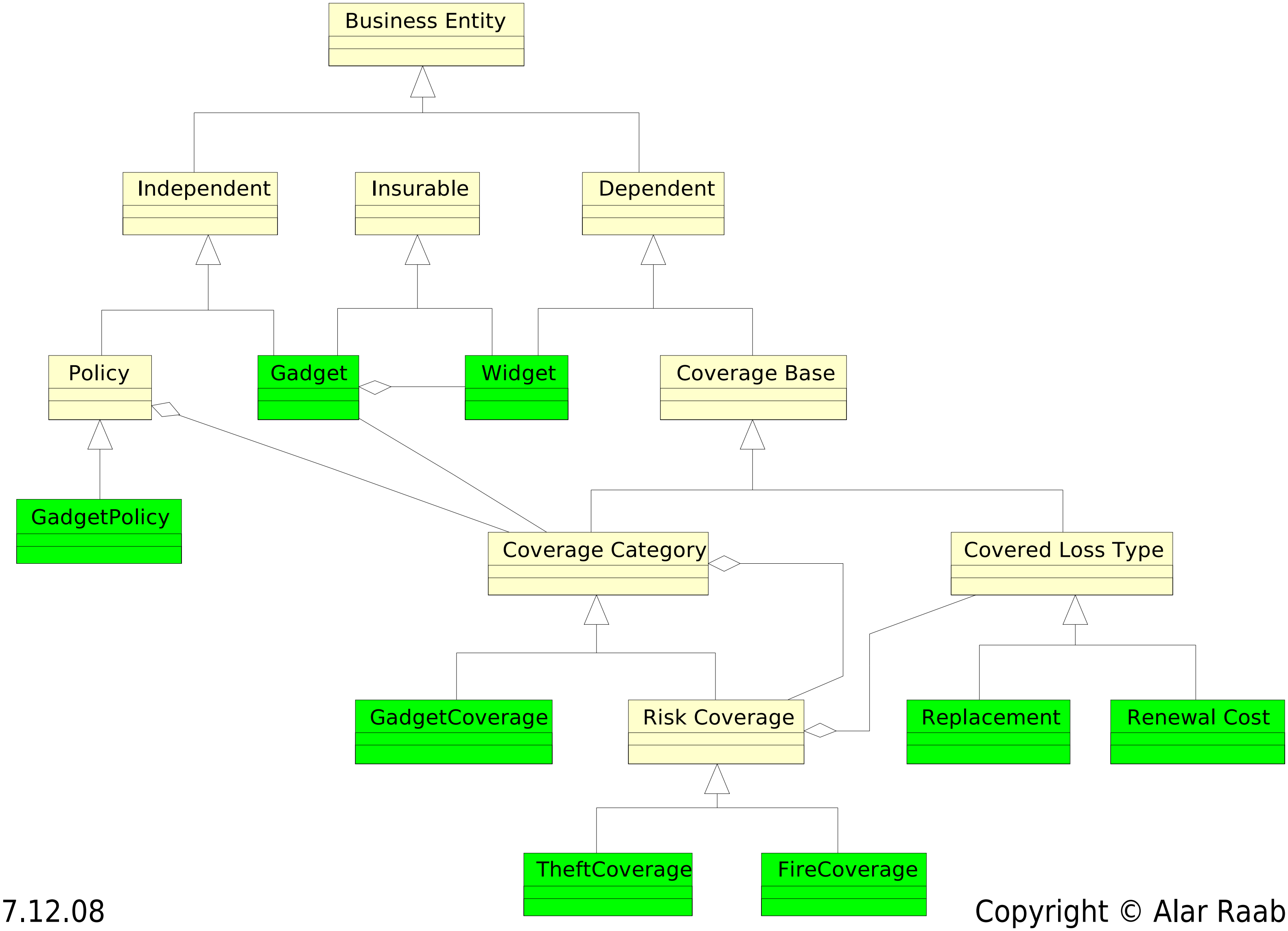
Example of Using Once&Done

- “Gadget Insurance”
  - Gadgets consist of Widgets
  - Gadgets can be insured against Fire and Theft
- Analysis model of “Gadget Insurance”
- Extending insurance domain model with “Gadget Insurance”
- “Gadget Insurance” product model
- Design model for “Gadget Insurance” policy management system

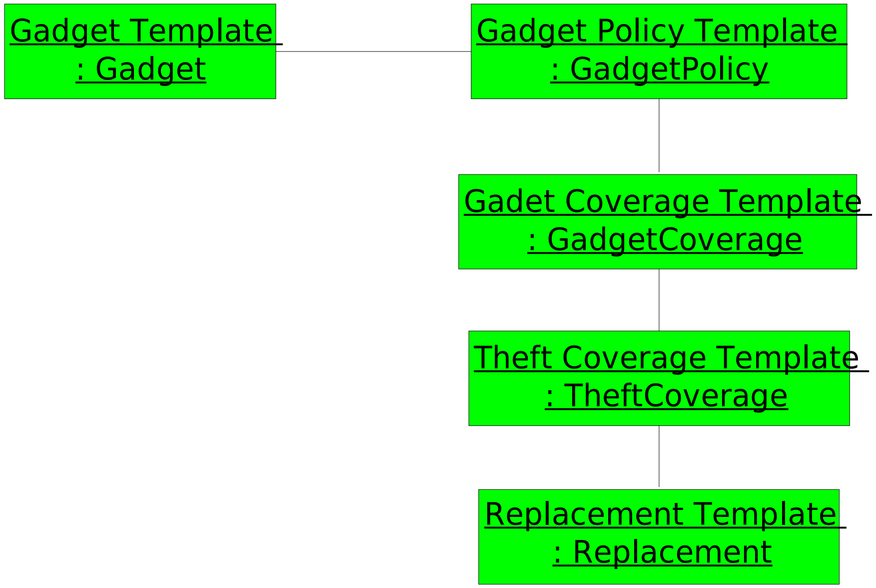
“Gadget Insurance” Analysis Model



“Gadget Insurance” Model as Extension to Insurance Domain Model



“Gadget Insurance” Product Model



Gadget Template
: Gadget

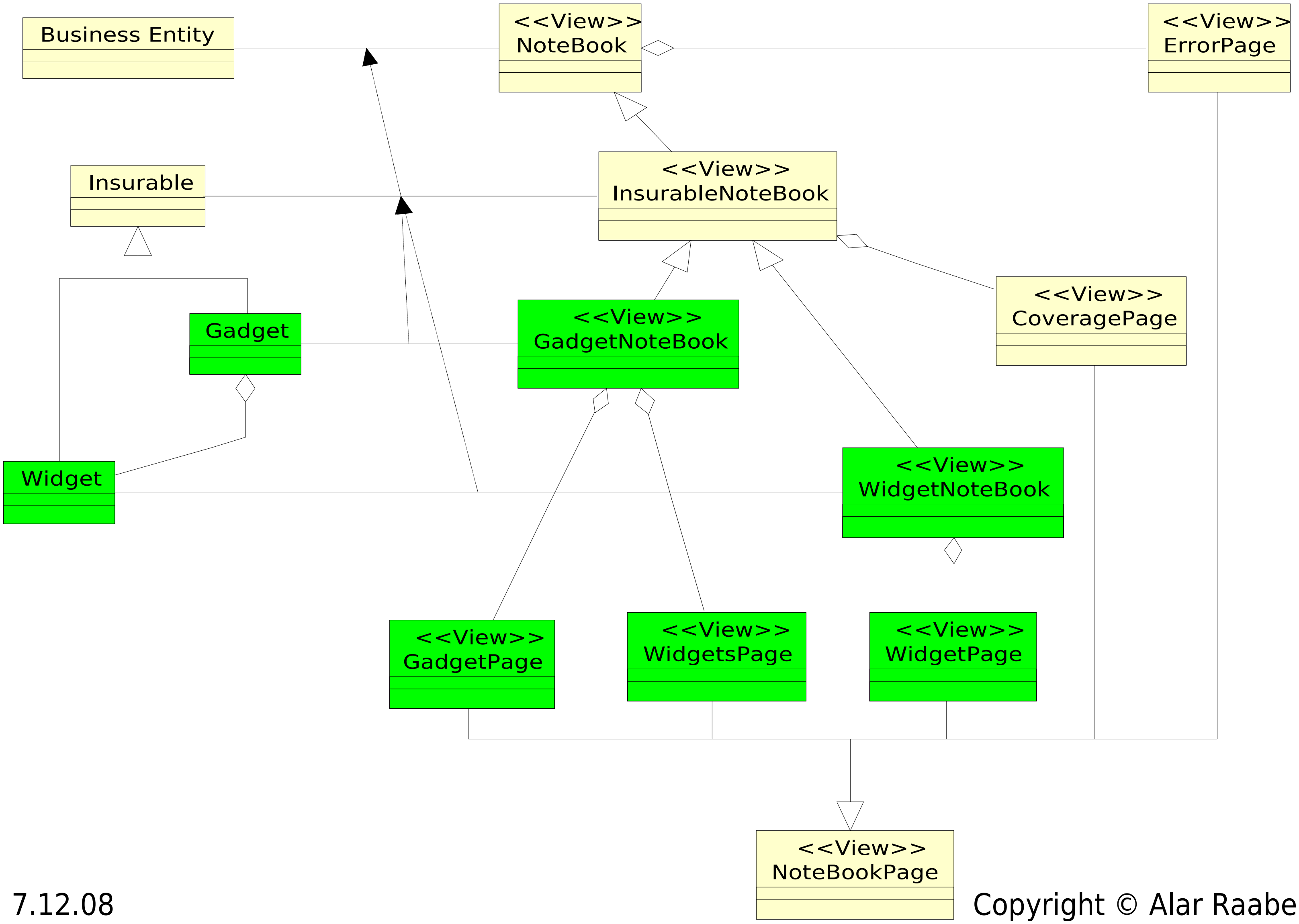
Gadget Policy Template
: GadgetPolicy

Gadet Coverage Template
: GadgetCoverage

Theft Coverage Template
: TheftCoverage

Replacement Template
: Replacement

“Gadget Insurance” Design Model



Once&Done – Results

- Reduction of development time
  - standard functionality generated from model
  - some parts of the model interpreted at run-time
- Quality of developed code
  - generated code had hints for developers
  - regeneration forced to conform to architecture
- Flexibility of resulting systems
  - business people were able to maintain parameters
- Technology independence of domain knowledge
  - easy transition from C/C++ client-server to Java based web-application

Comparing Model-Driven Method with Traditional

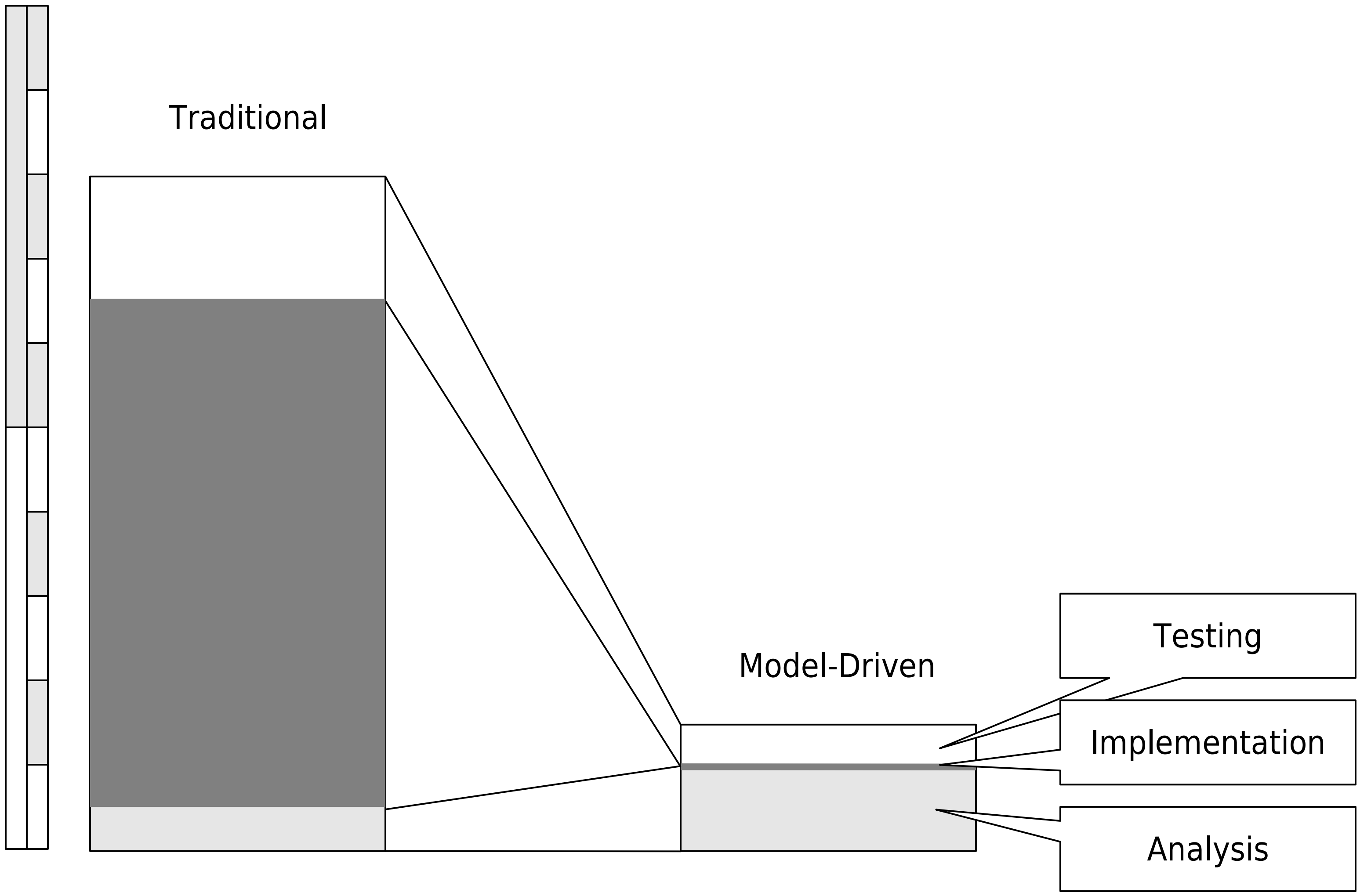
- Effort for First Iteration – Basically CRUD Application

- Manually coded Claims application
  - Volume
    - Domain Model: 30 entities, 30 relationships
    - Functionality: 10 use-cases (CRUD excl.)
    - User Interface: 34 screens
  - Effort: ~800 man-days (~50 analysis, ~550 implementation)

- Generated Claims application
  - Volume
    - Domain Model: 20 entities, 45 relationships
    - Functionality: 15 use-cases (CRUD excl.), 20 business rules
    - User Interface: 25 screens
  - Effort: ~130 man-days (~80 analysis, ~2 implementation)

- Generated Claims was regenerated on different platform

Comparing Model-Driven Method with Traditional



Lessons Learned

- Modeling is hard work and requires domain knowledge
- Project budget structure changes when using generation
- Repository is good for concurrent work, analysis and synthesis, model checking and transformations, but has problems with versioning and version management
- Textual models can be versioned as code, but this is not best for concurrent work
- Interpreters of meta-info (heavily parametric software components) are very difficult to debug – here generation/compilation is better

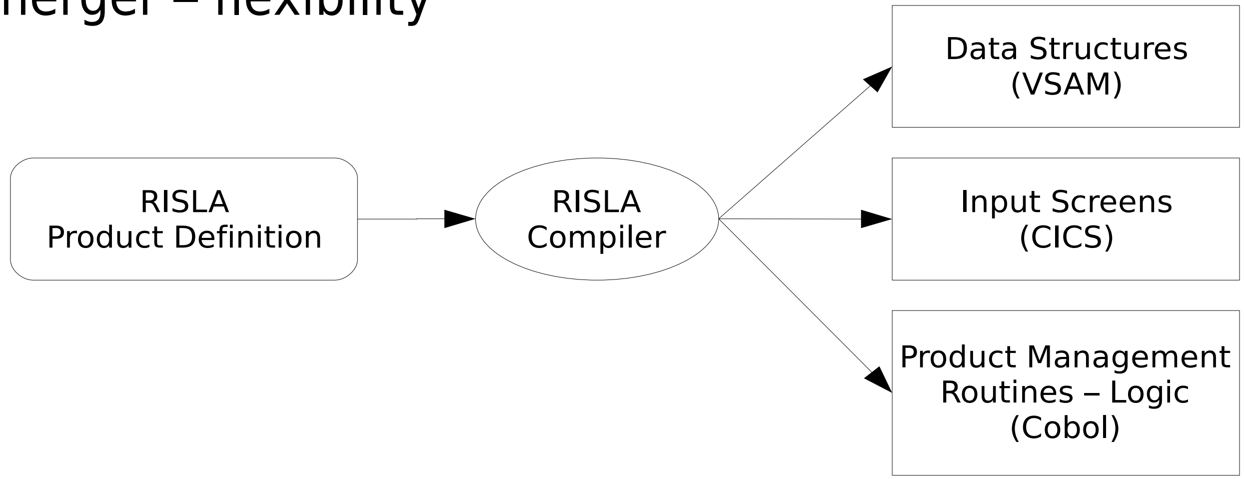
RISLA – Language for Product Models

- Started 1990 – CAP, MeesPierson, ING, CWI
- Describes interest rate products
  - Characterised by cash-flows
- Generates
  - Database
  - User Interface
  - Product Logic
- Example:
  - Loan

```
product LOAN

declaration
  contract data
    PAMOUNT : amount               %% Principal Amount
    STARTDATE : date               %% Starting date
    MATURDATE : date               %% Maturity data
    INTRATE : int-rate             %% Interest rate
    RDMLIST := [] : cashflow-list  %% List of redemptions.

  information
    PAF : cashflow-list            %% Principal Amount Flow
    IAF : cashflow-list            %% Interest Amount Flow

  registration
    %% Register one redemption.
    RDM(AMOUNT : amount, DATE : date)

...
```

RISLA – Result

- Success
  - Business people use – appropriate level of abstraction
  - Time to market decreased from 3 months to 3 weeks
  - Library of 100 components and 50 products
  - Survived merger – flexibility



RISLA Product Definition

RISLA Compiler

Data Structures (VSAM)

Input Screens (CICS)

Product Management Routines – Logic (Cobol)

Instrument Models in MLFi

- American Option

```
american : (date * date * contract) -> contract
american(t1, t2, u) =
         anytime({[t1, t2]}, zero, u)
```

- Zero Coupon

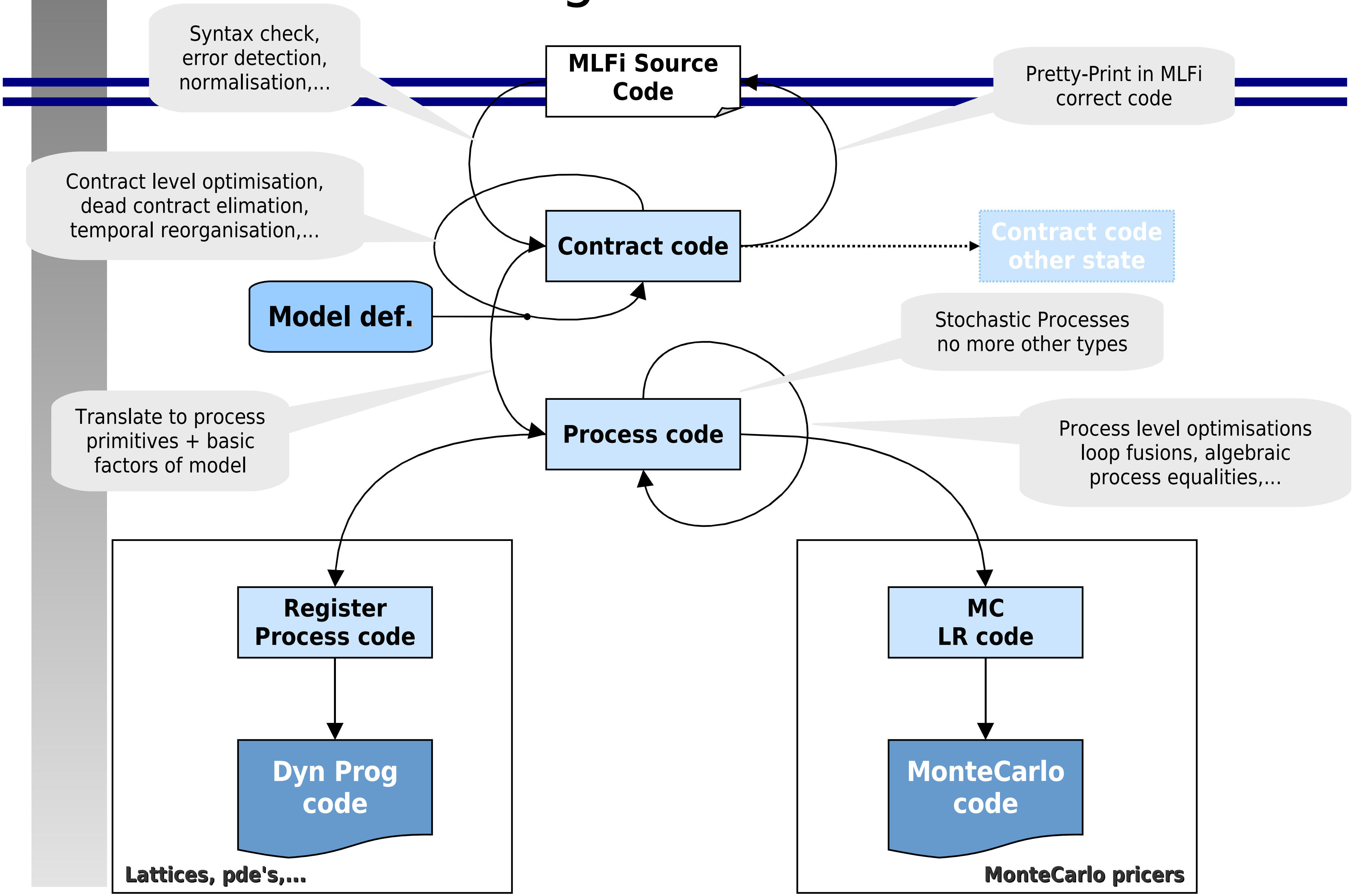
```
one : currency -> contract
(* if you acquire the contract (one k), then
 you acquire one unit of k. *)

scale : (observable * contract) -> contract
(* if you acquire scale(o, c), then you acquire
 c, but where all incoming and outgoing payments
 are multiplied by the value of o at acquisition
 date. *)

obs_from_float : float -> observable
(* obs_from_float k is an observable always equal to k *)
```

Contract Model in MLFi

- Custom-built Contracts

```
let option1 =

  let strike = cashflow(USD:2.00, 2001-12-27) in

  let option2 =

    let option3 =
      let t = 2001-12-18T15:00 in
      either
        ("--> GBP payment", cashflow(GBP:1.20, 2001-12-30))
        ("reinvest in EUR + receive cash later",
         (give(cashflow(EUR:1.00, t))) 'and' cashflow(EUR:3.20, 2001-12-29))
        t in

    either
      ("--> EUR payment", cashflow(EUR:2.20, 2001-12-28))
      ("wait for last option", option3)
      2001-12-11T15:00 in

  (either
     ("--> USD payment", cashflow(USD:1.95, 2001-12-29))
     ("wait for second option", option2)
     2001-12-04T15:00) 'and' (give (strike))
```

Generating Code for Valuation



Syntax check, error detection, normalisation,...

MLFi Source Code

Pretty-Print in MLFi correct code

Contract level optimisation, dead contract elimation, temporal reorganisation,...

Contract code

Contract code other state

Model def.

Stochastic Processes no more other types

Translate to process primitives + basic factors of model

Process code

Process level optimisations loop fusions, algebraic process equalities,...

Register Process code

Dyn Prog code

Lattices, pde's,...

MC LR code

MonteCarlo code

MonteCarlo pricers

Content

Conclusions

- **No Round-Trips**
  - when you are Model-Driven, **models are primary artifacts (models are your code)**
- **Model is Not the Picture**
  - model is a collection of structured information in the form, which is best fore given Domain (**pictures should be Model-Driven**)
- **Keep Focus, Don't Mix Domains**
  - to represent information about problems/solutions in different Domains **use several Models with different Meta-Models**
- **Let the Models drive the Analysis**
  - models are the ubiquitous language for stakeholders
- **This is not a Religion**
  - use Model-Driven Approaches only where it makes sense and brings value

References

- Some books to read
  - Krzysztof Czarnecki and Ulrich W. Eisenecker, Generative Programming - Methods, Tools, and Applications, 2000
    - http://www.generaative-programming.org/
  - Tom Stahl, Markus Völter , Model-Driven Software Development: Technology, Engineering, Management, 2006
    - http://www.voelter.de/publications/books-mdsd-en.html

- Some WWW sites to look
  - OMG MDA
    - http://www.omg.org/mda
  - Eclipse Modeling Framework
    - http://www.eclipse.org/modeling/emf/
  - Others
    - http://www.andromda.org/
    - http://www.openarchitectureware.org/
    - http://www.voelter.de/services/mdsd-tutorial.html
    - http://www.martinfowler.com/bliki/dsl.html
    - http://www.prakinf.tu-ilmenau.de/~czarn/gpsummerschool02/
    - …

Thank You!

Questions?

Domain analysis

- Domain
  - an area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area (UML)
- Domain Analysis
  - Domain scoping – select and define domain of focus (context)
  - Domain modelling – collect the relevant domain information and integrate it into a coherent domain model
- Domain model
  - A body of knowledge in a given domain represented in a given modelling language
    - Scope (boundary conditions of the domain)
    - Domain knowledge (elements that constitute the domain)
    - Generic and specific features of elements and configurations
    - Functionality and behaviour

Domain analysis methods

- Domain analysis methods
  - Language based
  - Algebraic (formal)
  - Object-oriented
  - Aspect-oriented
  - Feature-oriented
  - Combined approaches → feature-oriented + …

- Domain analysis methods based on features
  - Feature-Oriented Domain Analysis (FODA) – SEI
  - Feature-Oriented Reuse Method (FORM) – K. Kang
  - Domain Engineering Method for Reusable Algorithmic Libraries (DEMRAL) – Czarnecki, Eisenecker
  - …

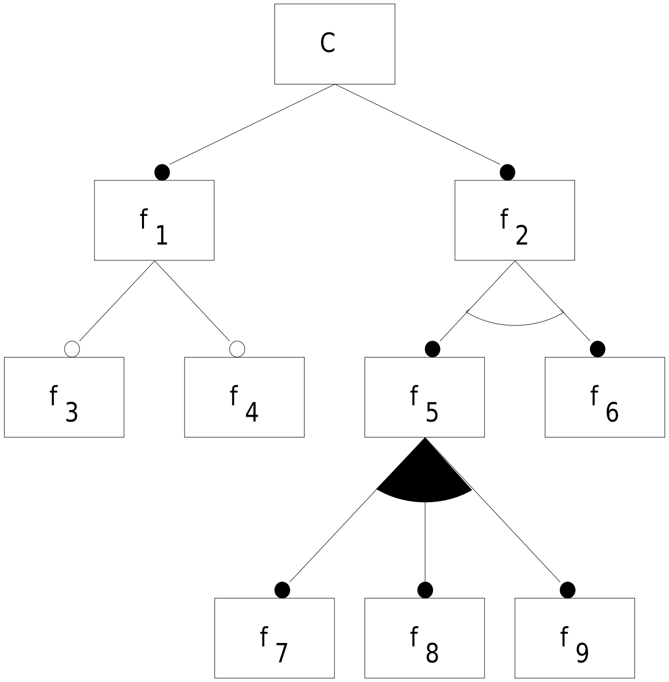
Feature modelling

- Feature modelling (a.k.a feature analysis)
  - is the activity of modelling the common and the variable properties of concepts and their interdependencies

- In feature modelling
  - ***Concepts*** are any elements and structures of the domain of interest
  - ***Features*** are qualitative properties of concepts
  - ***Feature model*** represents the common and variable features of concept instances and the dependencies between the variable features
  - Feature model consists of a ***feature diagram*** and additional information

Feature diagram

- Tree-like diagram where
  - The root node represents a concept, and
  - Other nodes represent features
- Feature types
  - Mandatory features (f_1, f_2, f_5, f_6)
  - Optional features (f_3, f_4)
  - Alternative features (f_5, f_6)
  - Or-features (f_7, f_8, f_9)
- Constraints between features
  - Composition rules (requires, excludes, …)



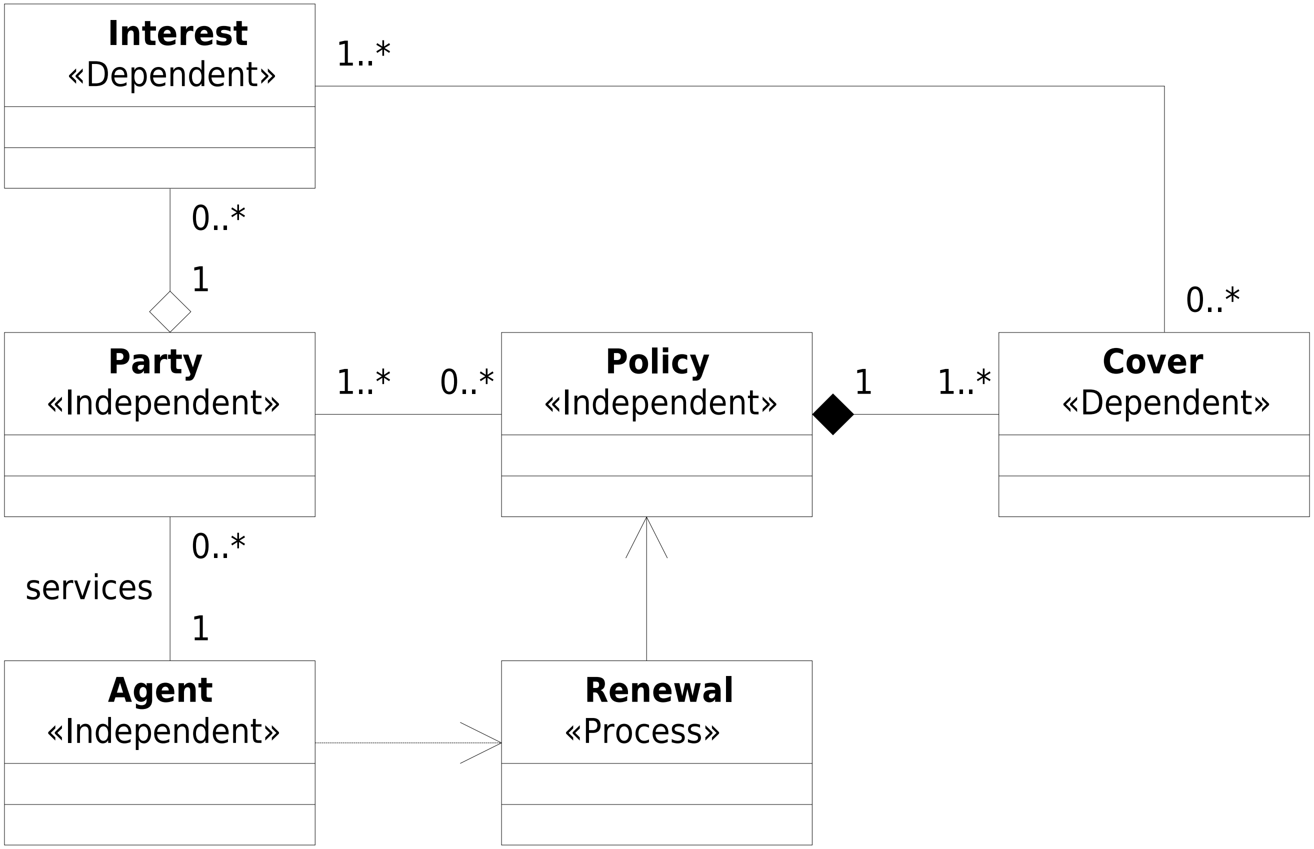
Feature types

- FODA feature types
  - Context features – performance, synchronization, …
  - Operational features – application functions
  - Representation features – visualization, externalization, …

- FORM feature types
  - Capabilities
  - Operating environment
  - Domain technologies
  - Implementation techniques (domain independent)

- Only some of the features depend on problem domain

Example problem domain model (Insurance)

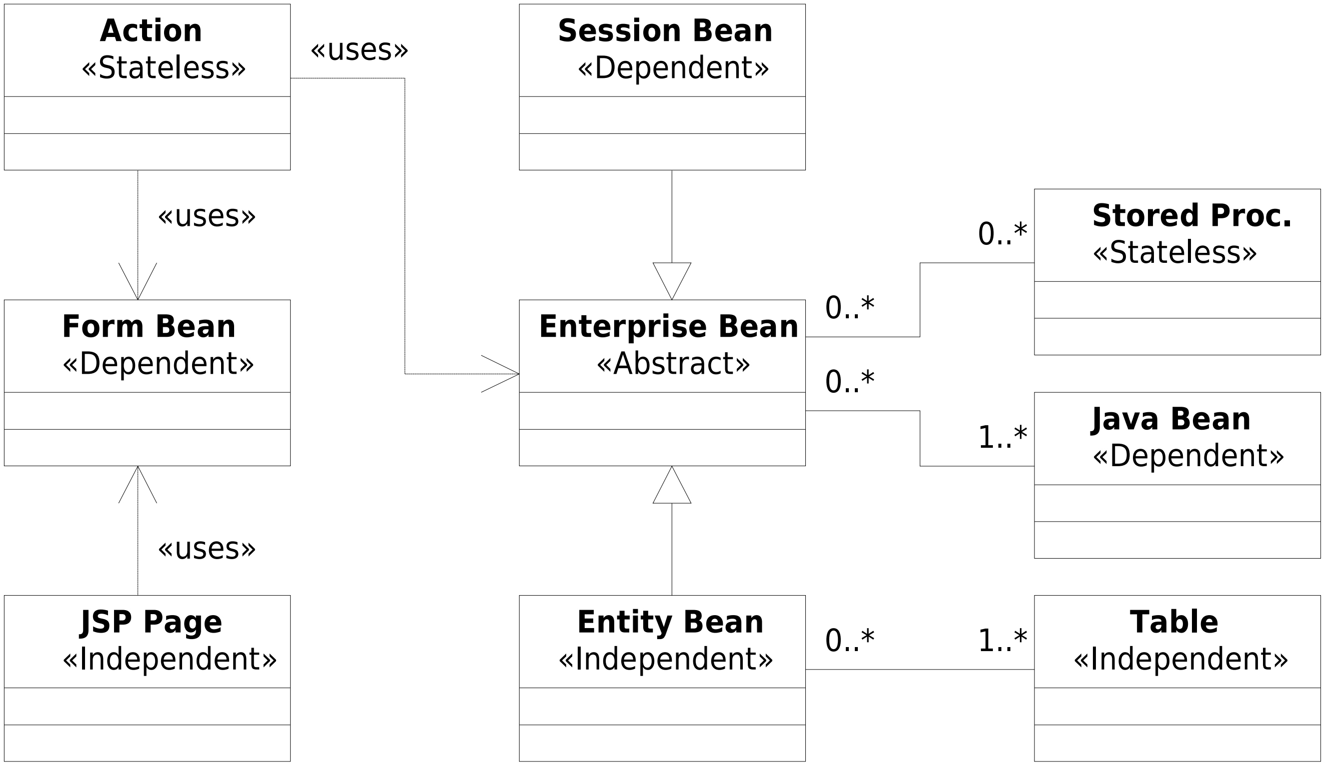


Example problem domain model – features independent of domain

- Concept “Policy” – independent business object
  - Features (domain independent)
    - Has identity
    - Independent
    - Has state
    - Persistent → Storable, Searchable
    - Viewable → Modifiable

- Concept “Renewal” – business process
  - Features (domain independent)
    - No identity
    - No state
    - Transient
    - Business behavior → Asynchronous

Example solution domain model (J2EE + Struts + RDB)



Example solution domain model – features independent of domain

- Concept “Entity Bean”
  - Features (independent of domain)
    - Identity
    - State
    - Persistent → Storable, Searchable
    - Behavior

- Concept “Session Bean”
  - Features (independent of domain)
    - No identity
    - State is optional
    - Transient
    - Behavior

Configurations

- Configuration
  - A set of concepts collectively providing required set of features
  - Feature set of configuration might be larger than sum of feature sets of all the concepts in the configuration

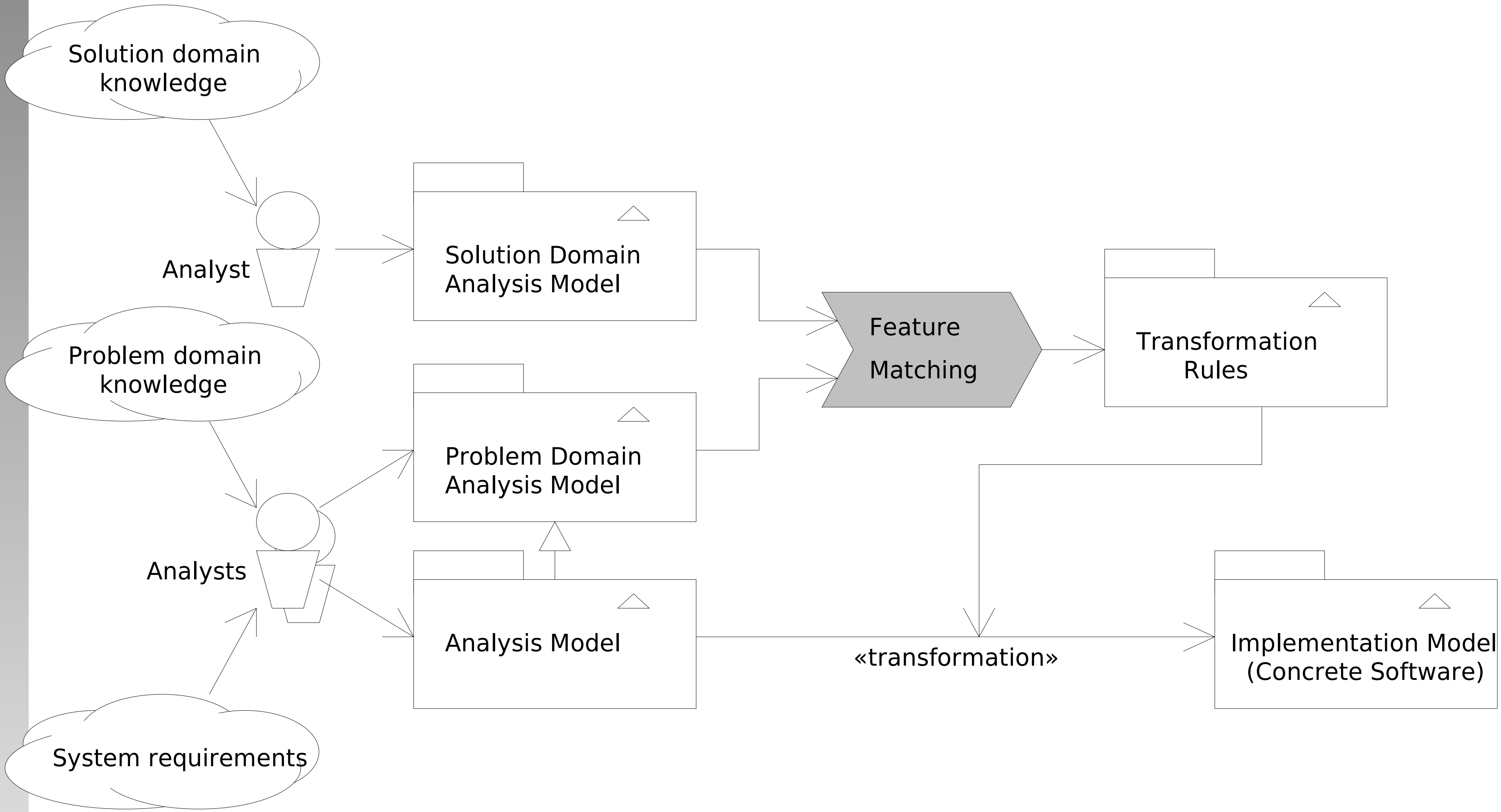
- Configurations of solution domain are identified during the solution domain analysis

Example solution domain model – features of configurations

- Configuration {“JSP Page”, “Form Bean”, “Action”, “Entity Bean”}
  - Features (independent of domain)
    - Identity
    - State
    - Persistent → Storable, Searchable
    - Behavior
    - Viewable → Modifiable

- Configuration {“JSP Page”, “Form Bean”, “Action”, “Session Bean”}
  - Features (independent of domain)
    - No identity
    - Transient
    - Behavior
    - Viewable

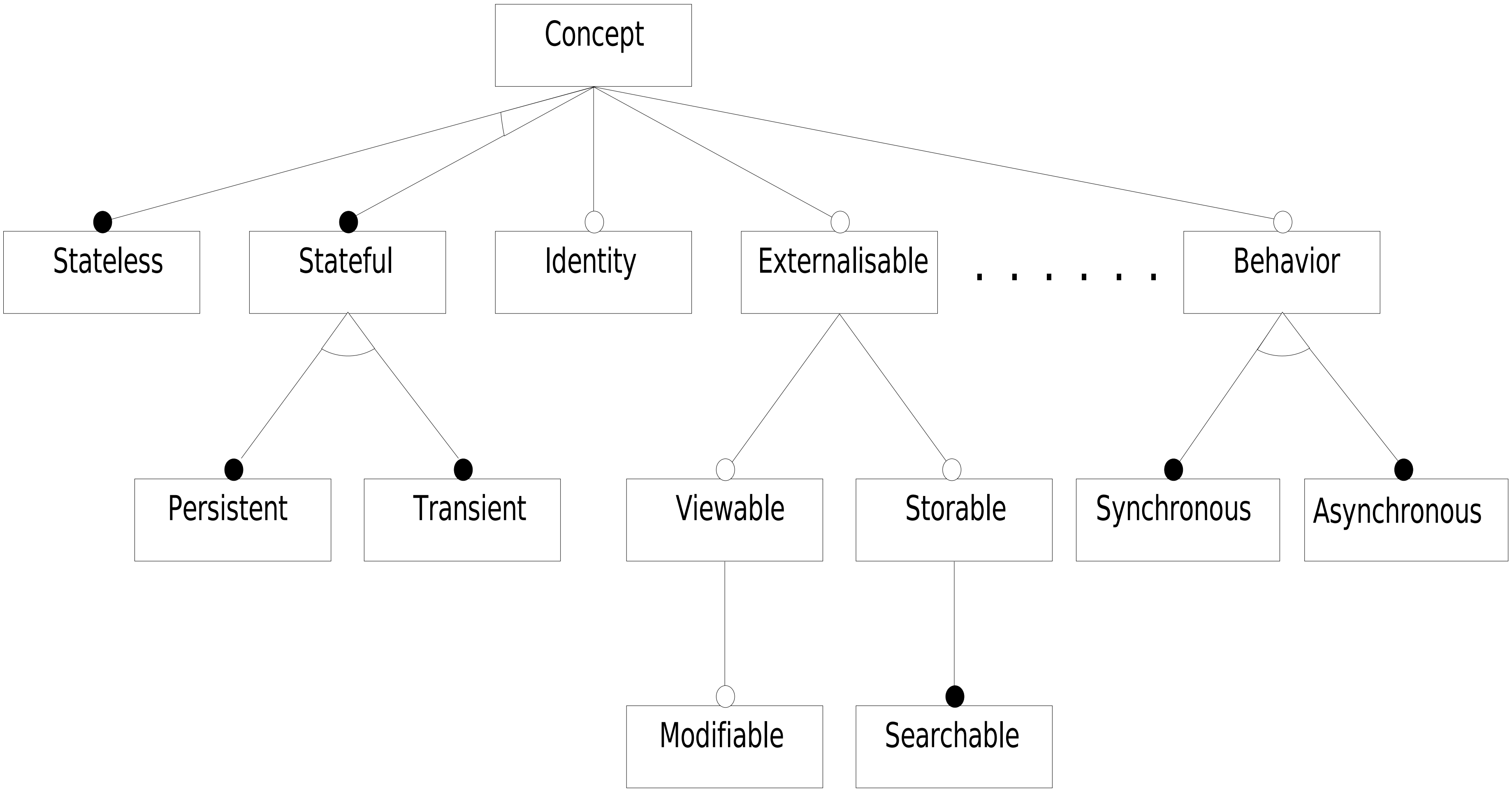
Feature matching in model-based software development



Common feature space

- Common features of concepts and configurations (identified for business information systems)
  - Functional features
    - May have identity
    - Independent | Dependent
    - Stateless | Stateful
    - Transient | Persistent → Storable, Searchable
    - Viewable → Modifiable
    - Business behavior → Asynchronous, Synchronous
    - …
  - Non-functional features
    - Efficiency → Speed, Space
    - Scalability
    - Modifiability
    - Portability
    - …

Feature diagram of common features of a concept



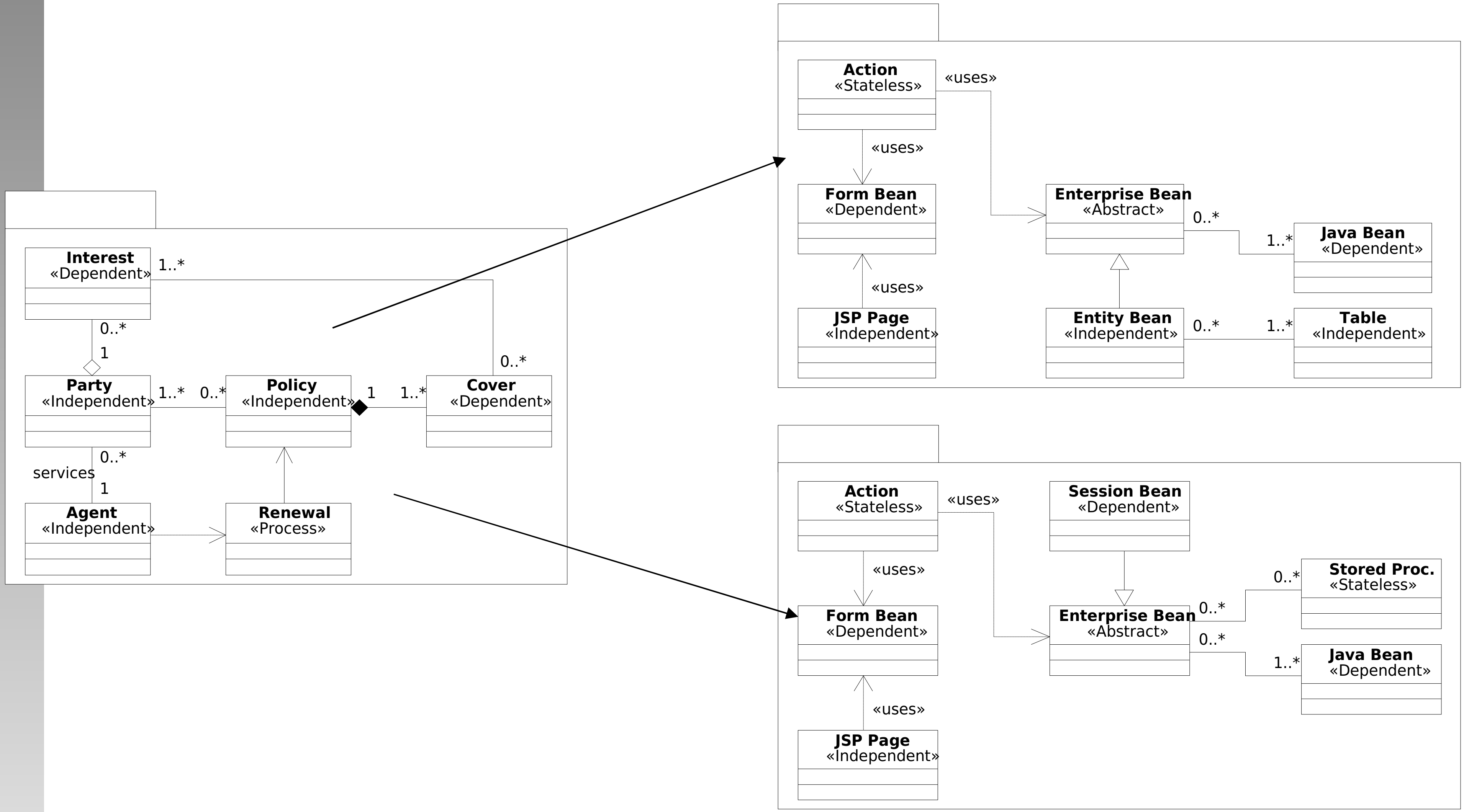
Solution domain and architecture selection

- Solution domain selection is based on the features offered by solution domain configurations
- Selecting the suitable architecture style
  - Based on functional features
    - Persistence
    - …
  - Based on non-functional features
    - Scalability
    - Modifiability
    - …
- Examples
  - Data-entry application → Central Repository
  - Signal processing application → Pipes and Filters
  - Decision Support → Blackboard

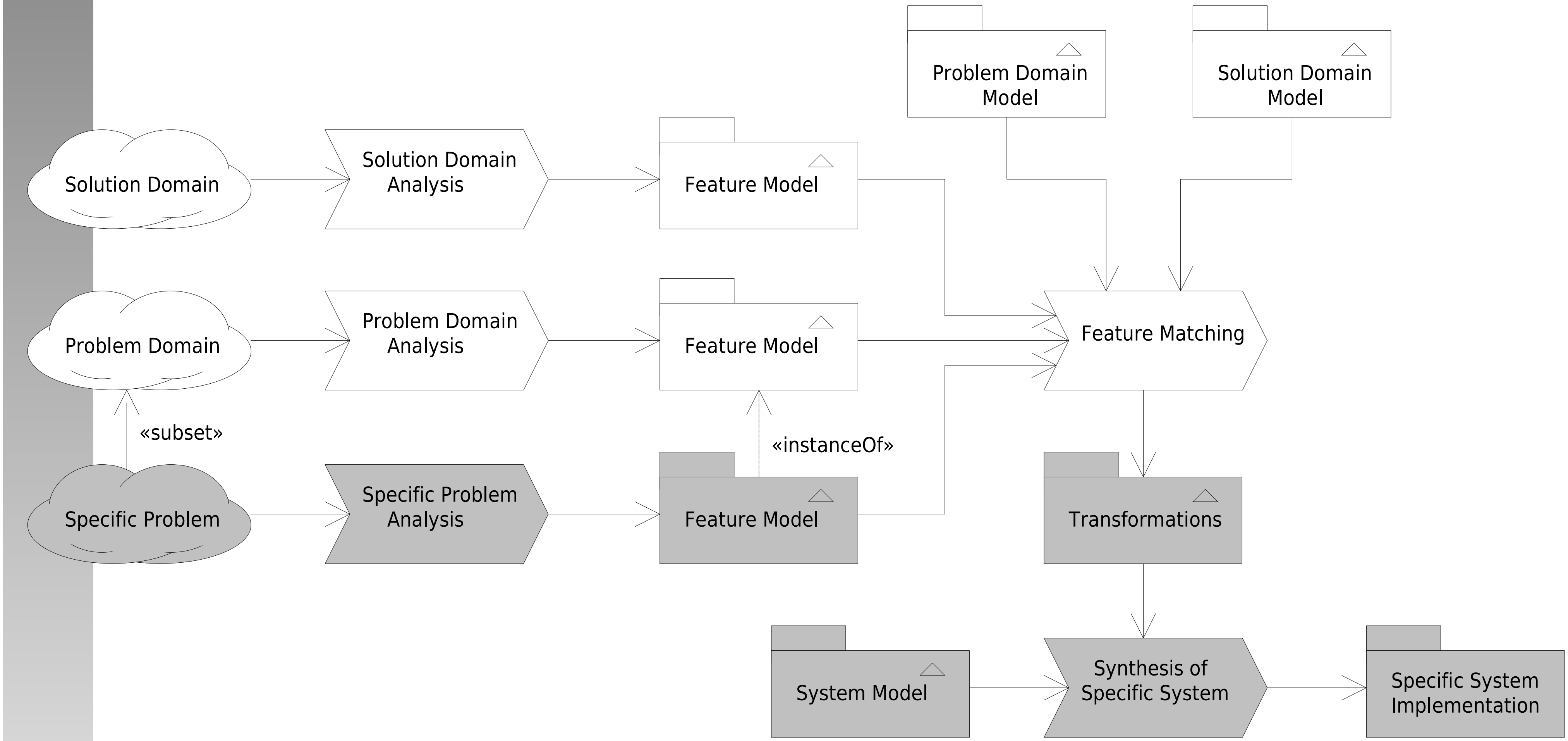
Implementation synthesis

- Feature Analysis
  - Problem domain feature analysis
    - starting from implicit features (external features)
  - Solution domain feature analysis
  - System feature analysis – explicit features
- Common Feature space
  - Normative set for implicit features
- Synthesis – transformation of business analysis model into implementation model
  - Selection of solution domain (architecture style)
  - Selection of solution domain elements and configurations (implementation)
    - Decided by feature matching

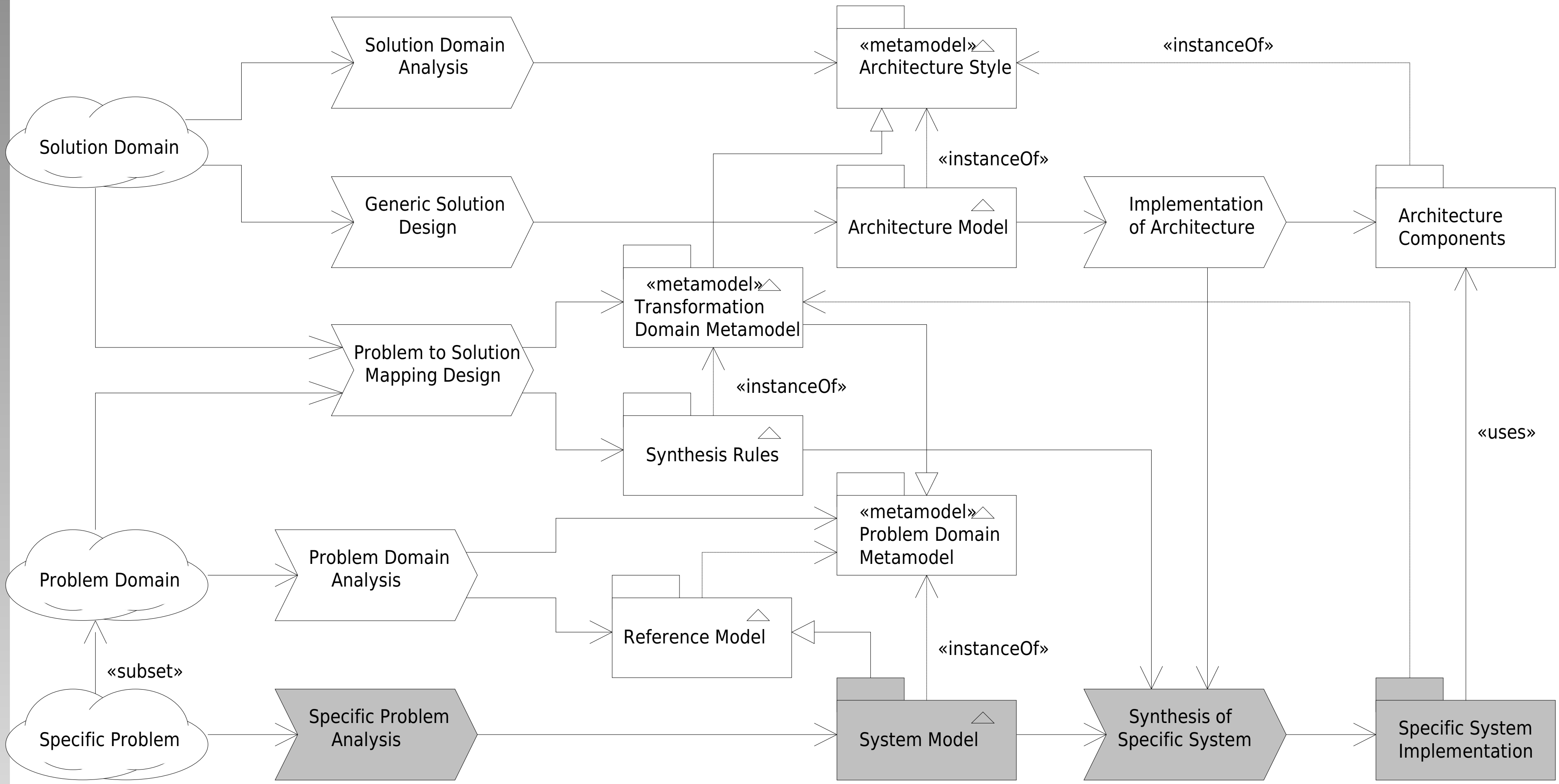
Example feature mapping (Insurance → J2EE)



Feature matching in model-based software development



Steps of Model-Oriented Software Development



MDD vs. “normal” way

- Analysis phase takes some more time, as model has to be developed to certain level before anything useful could be generated
- First iteration of development takes no time – code is generated – prototyping is extremely cheap and prototypes could be used to verify the model (by executing the business scenarios)
- With regular regeneration next iterations stay consistent with model and avoid architecture “decay”

Modeling Guidelines

- When combining metamodels – use
  - Complex queries for selection of elements
  - Massive renaming for name conflicts resolution
  - Overriding, replacing and deferring of elements
- When creating reference models – support model combinations via
  - Role-Oriented modeling
  - Clear identification of extension points
  - Separation of variable parts
  - Separation of functionality
  - Grouping (clustering) of model elements

Design Techniques

- Always have an escape plan
  - Example of business rules
    - simple parametric rule types
    - simple declarative rule language
    - dynamically bound code
  - Example of UI screens
    - generated screens
    - painted screens
  - Example of database interface
    - automatic object-relational mapping
    - dynamically bound mappers (externalizers/internalizers)

Programming Techniques

- Declarative Programming
- Intention-Revealing (Fluent) Interfaces
- Regenerate often → allow only such customizations, that follow the architecture conventions
    - reverse engineering is one-time tool for legacy integration, not for regular development
- Involve business people in development and maintenance through the model manipulations

Representations of Model

- Repository (RDBMS)
- Textual representation (XML)
- Code (e.g. Java + tags or annotations)
    - Java annotations are restricted to interpretative model (compilation could be achieved with generating code – “two pass” execution)