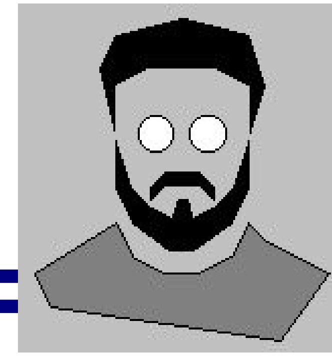
Model-Driven Development

Model-Driven Methods in Software Engineering

Alar Raabe

Alar Raabe



- Over 30 years in IT
  - held various roles from programmer to a software architect and to enterprise business architect
- 15 years in insurance and last 6 years in banking domain
  - developed model-driven technology for insurance applications product-line (incl. models, method/process, platform/framework and tools)
  - developing/implementing business architecture framework and methods for a banking group
- Interests
  - software engineering (tools and technologies)
  - software architectures
  - model-driven software development
  - industry reference models (e.g. IBM IAA, IFW)
  - domain specific languages

Content

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
    - Model-Driven Software Development
    - Generative Programming
    - Domain Specific Languages
- Practical Aspects
    - Model Management
    - Best Practices
    - Examples
- Conclusions
- References

Common Language – some Definitions 1

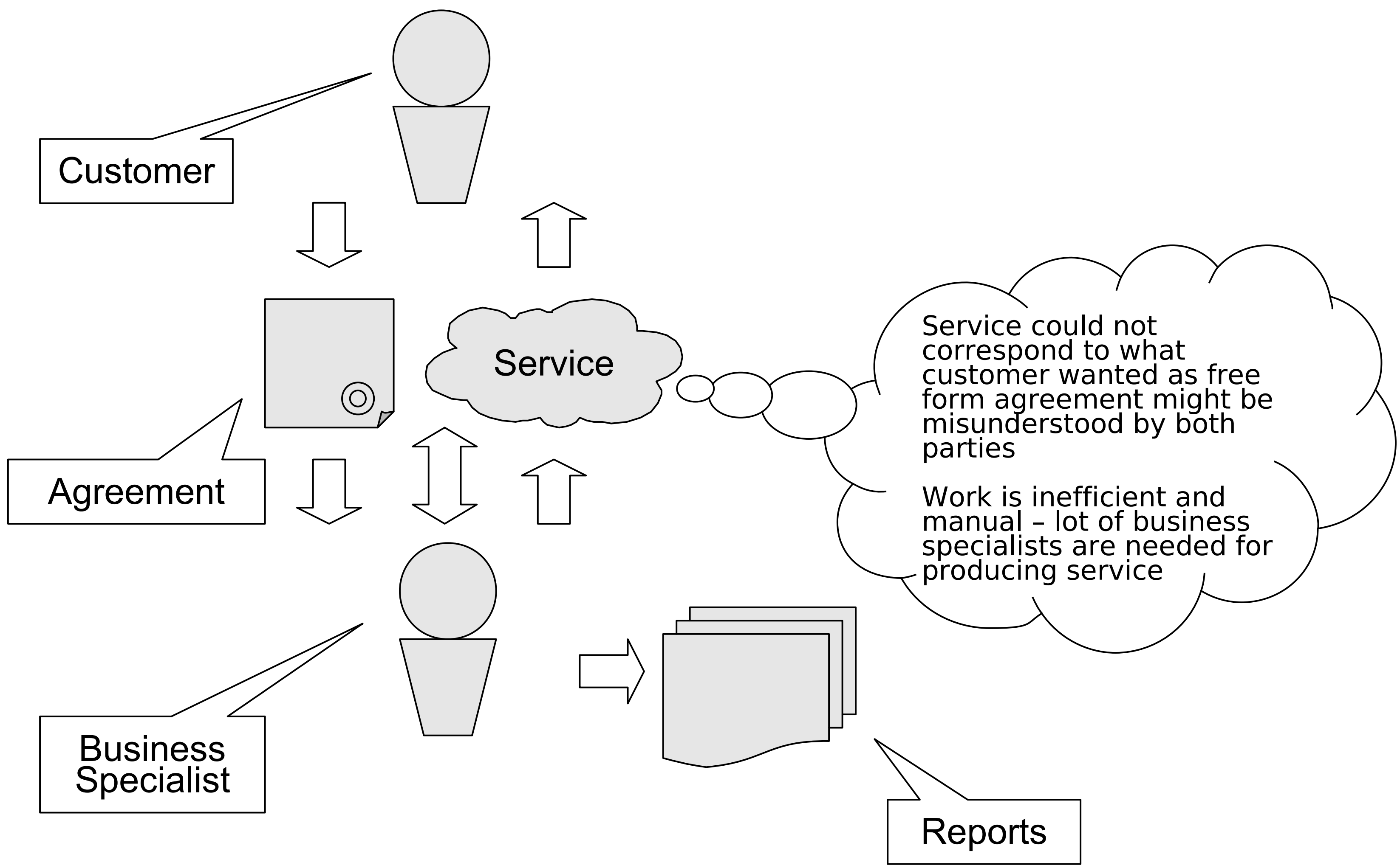
- Abstraction
  - a view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information
  - the process of formulating a view
- **Model**
  - an interpretation of a theory for which all the axioms of the theory are true
  - a semantically closed abstraction of a system or a complete description of a system from a particular perspective
  - **anything that can be used to answer questions about system**
    - Marvin Minsky & Doug Ross
- **Meta-model**
  - **a model of models** (or a language for models)
  - a logical information model that specifies the modelling elements used within another (or the same) modeling notation
  - model defining the concepts and their relations for some modelling notation

A set of structured information
NOT JUST A PICTURE !

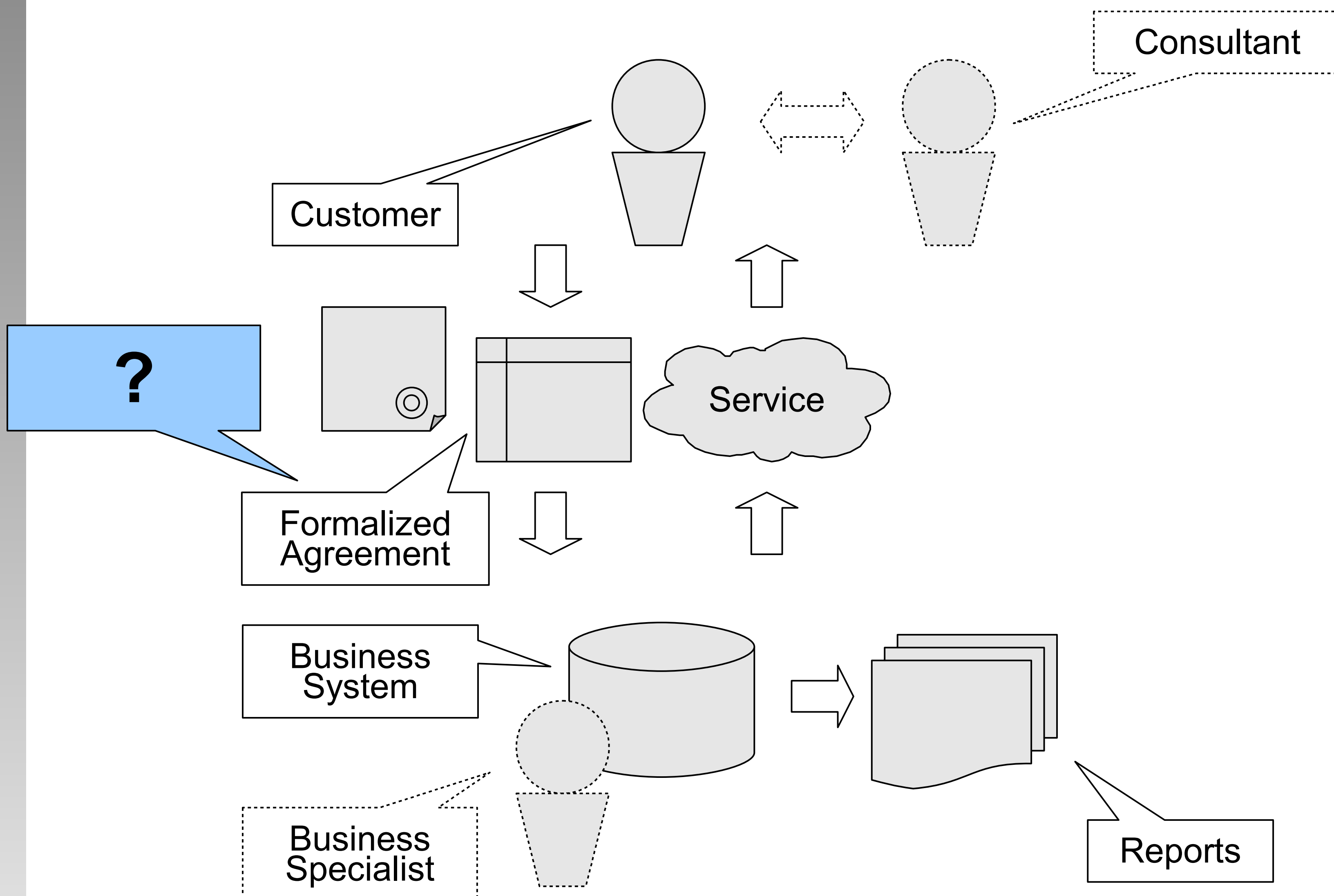
Common Language – Some Definitions $_2$

- 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

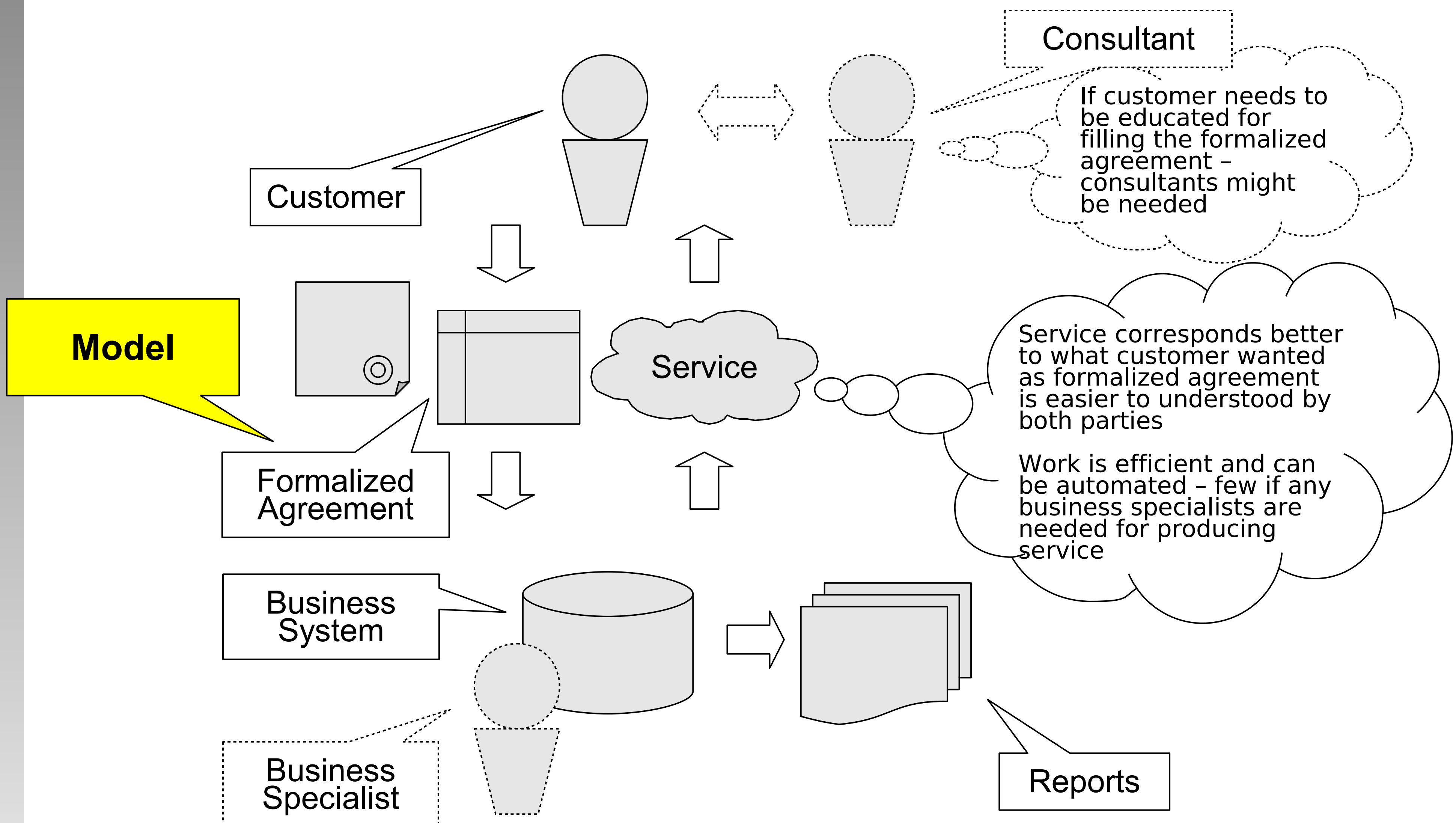
How we did Business Yesterday



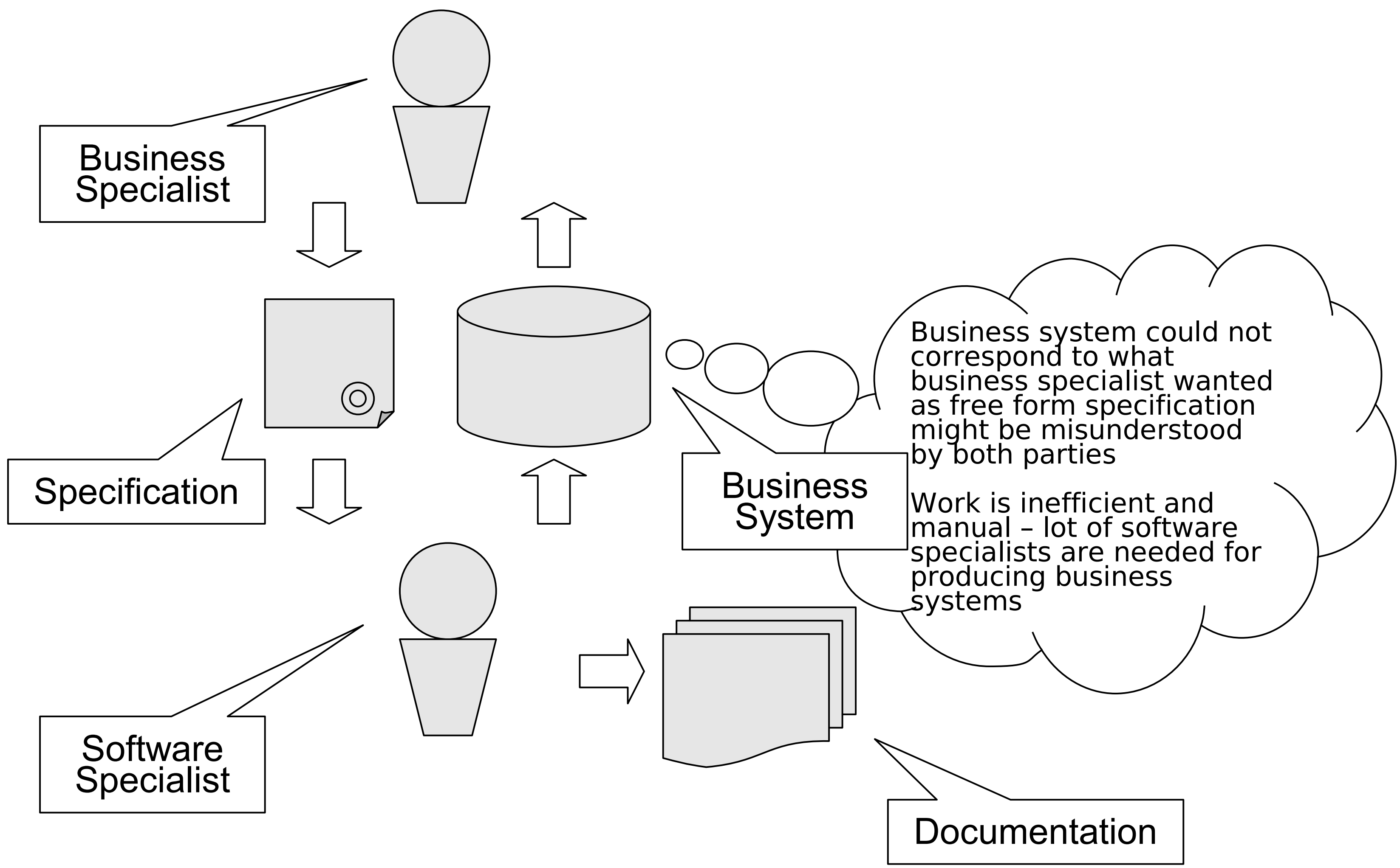
How we do Business Today/Tomorrow



How we do Business Today/Tomorrow

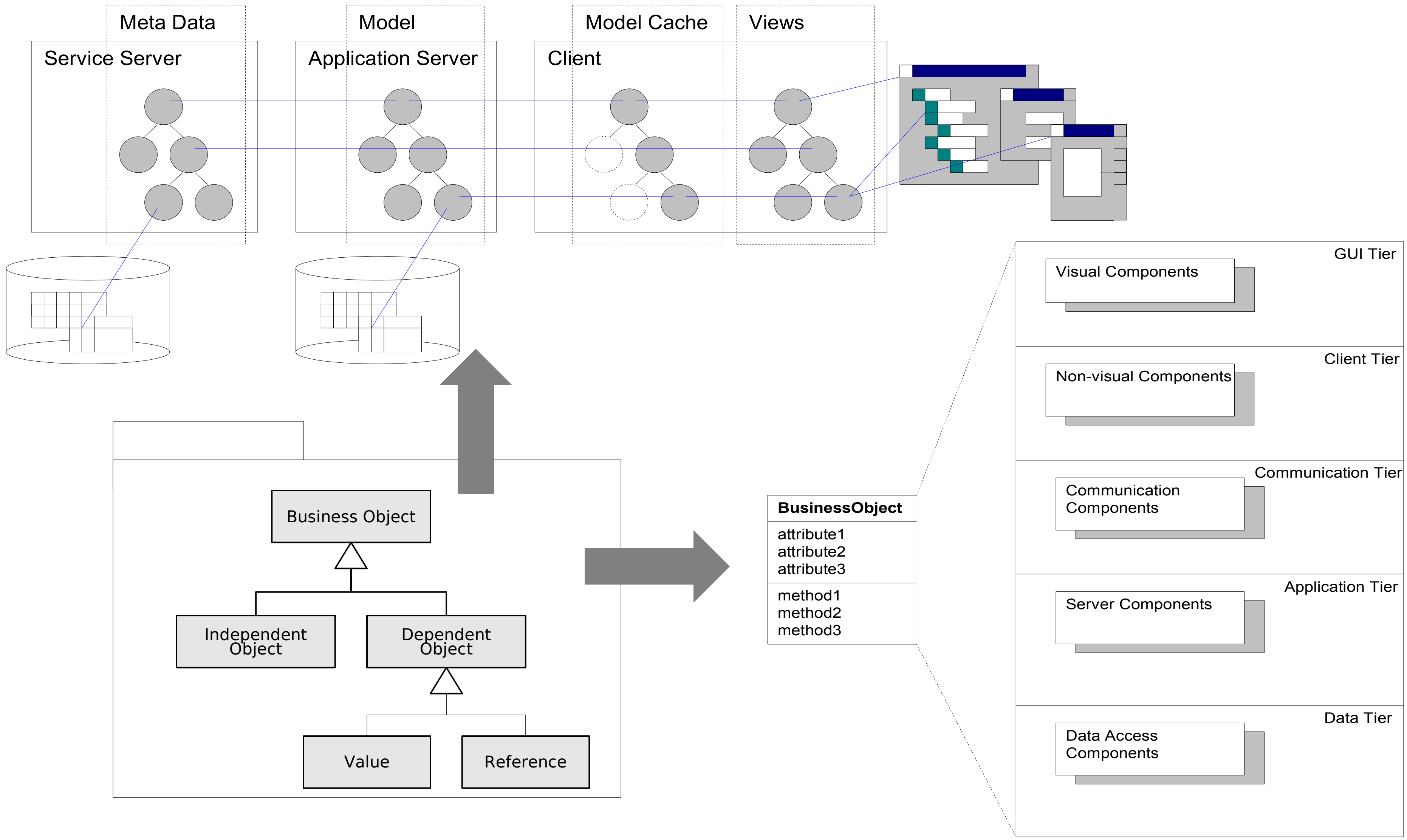


How we Develop Software Today

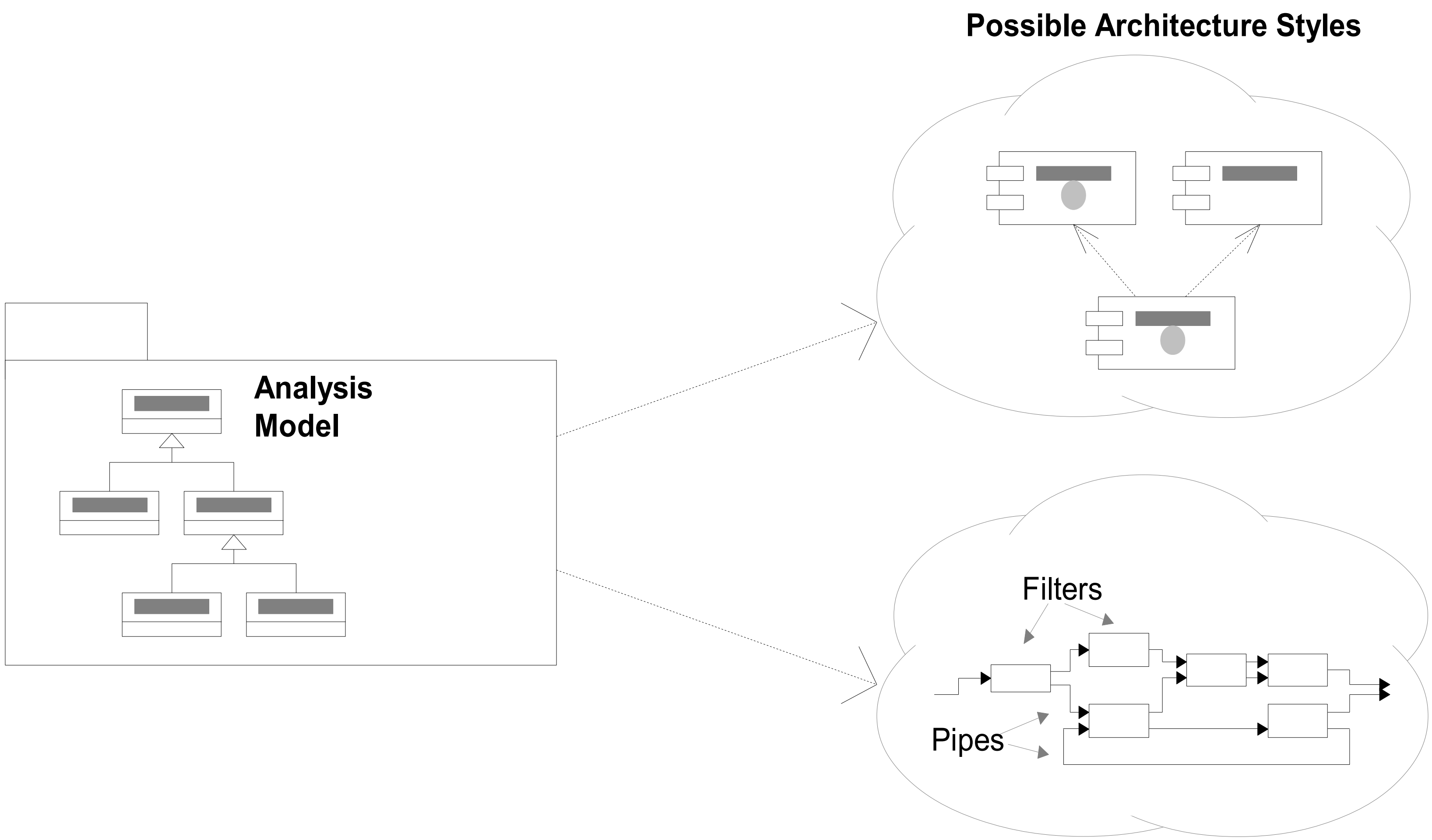


PROBLEM

Consistency of Implementation



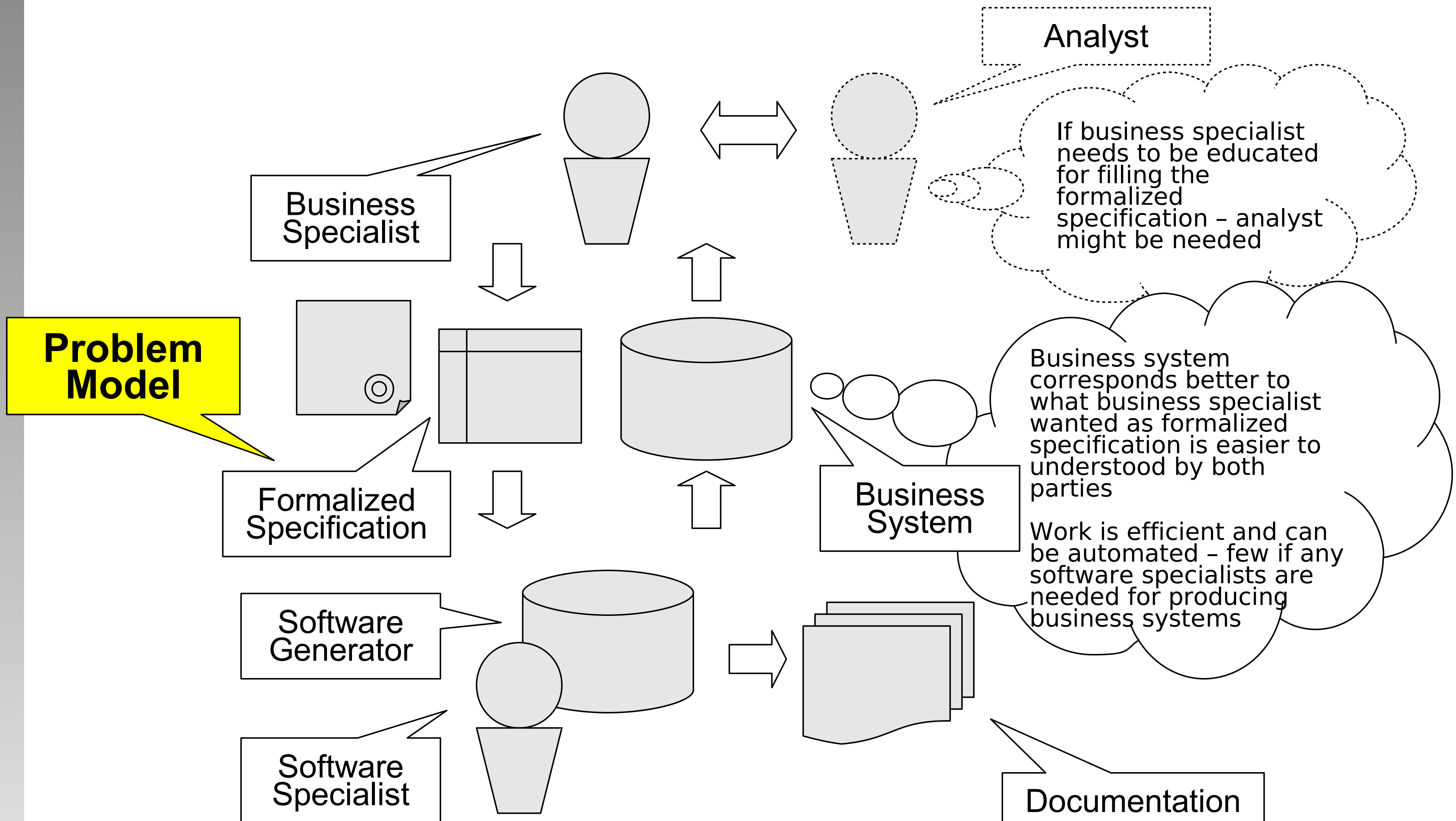
Mapping to Different Implementations



Problems → Solution

- 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 flexibility of systems (high cost of changes)
- Solution ⟶ **Automation**

How we should Develop Software



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.

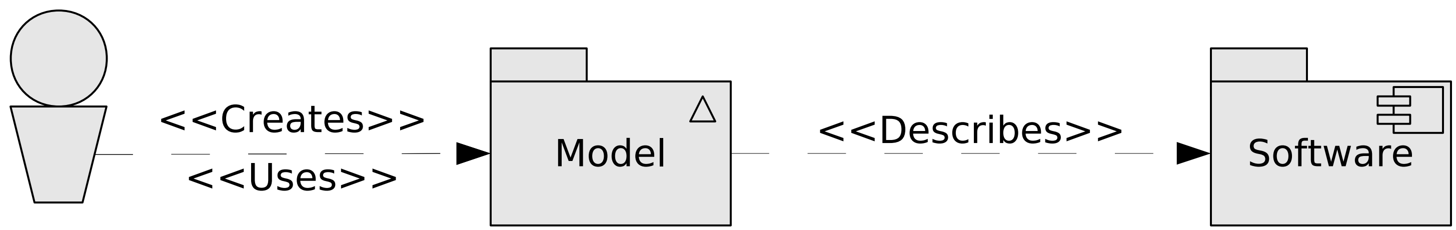
-- Ecclesiastes 1:9

- Programming Languages – to automate coding
  - FORTRAN (1954), Lisp (1956)
  - APT (MIT 1957) ← *First DSL!*
  - Algol (1958)
- Problem-Oriented Languages/Systems – to automate programming
  - ICES (MIT 1961) → COGO, STRUDL, BRIDGE, ...
  - PRIZ (ETA Kübl)
- Compiler Generators – generation of solution from model of problem
  - Yacc/Lex (1979)
- Application Generators
  - MetaTool & GENII/GENOA & ... (Bell Labs 1980s)
- CASE (Computer-Aided Software Engineering) Tools
  - GraphiText, DesignAid (Nastec 1982)

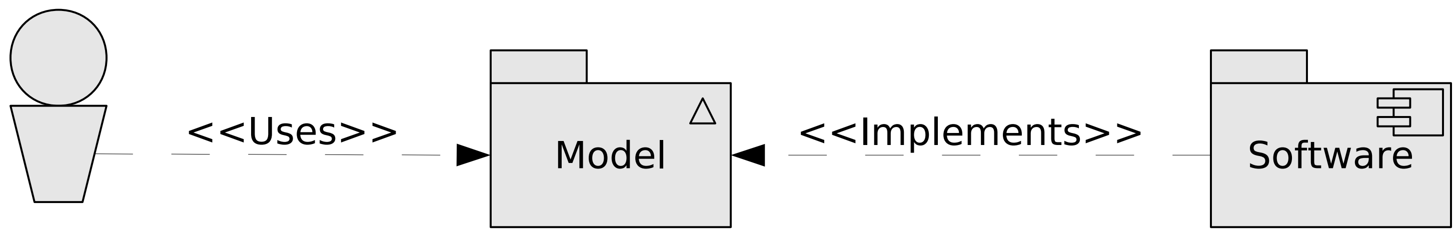
Using Models in Software Development

Most usual – we will not deal with this

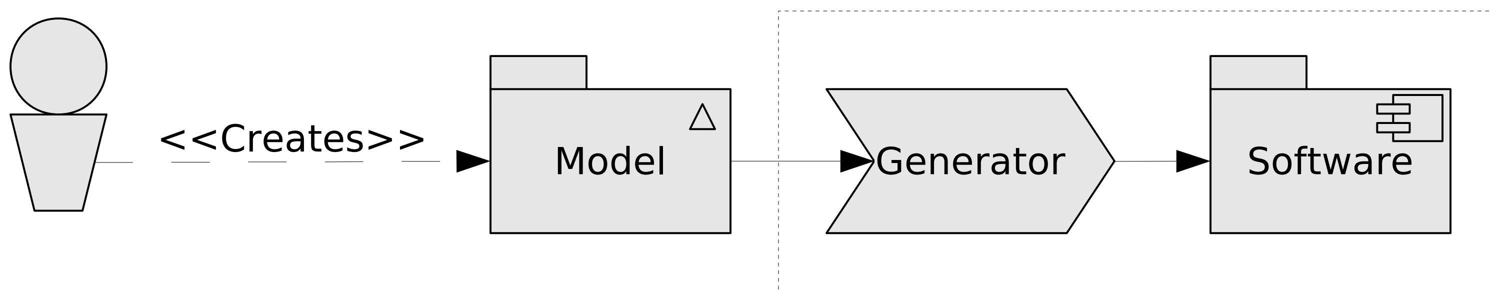
- Models as Descriptions and Illustrations (Documentation)



- Software as Model – Direct Modeling (of Domain)



- Models as Primary Artifacts (Models as Software)



Content

Convergent Engineering

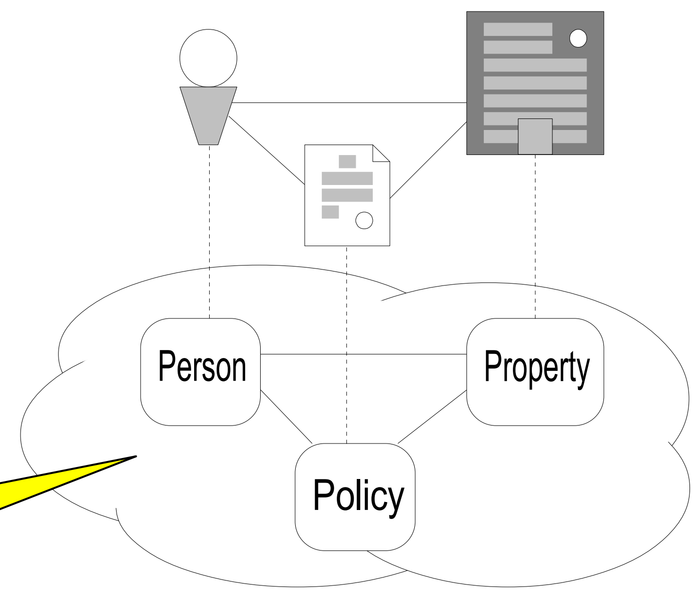
Structure of business and software should converge

- Structured Programming / Structured Design [Jackson 1975]
  - program structure should correspond to the structure of the problem
- 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,

    2) do it in many different ways, and

    3) plan/forecast (do “what-if” analysis)

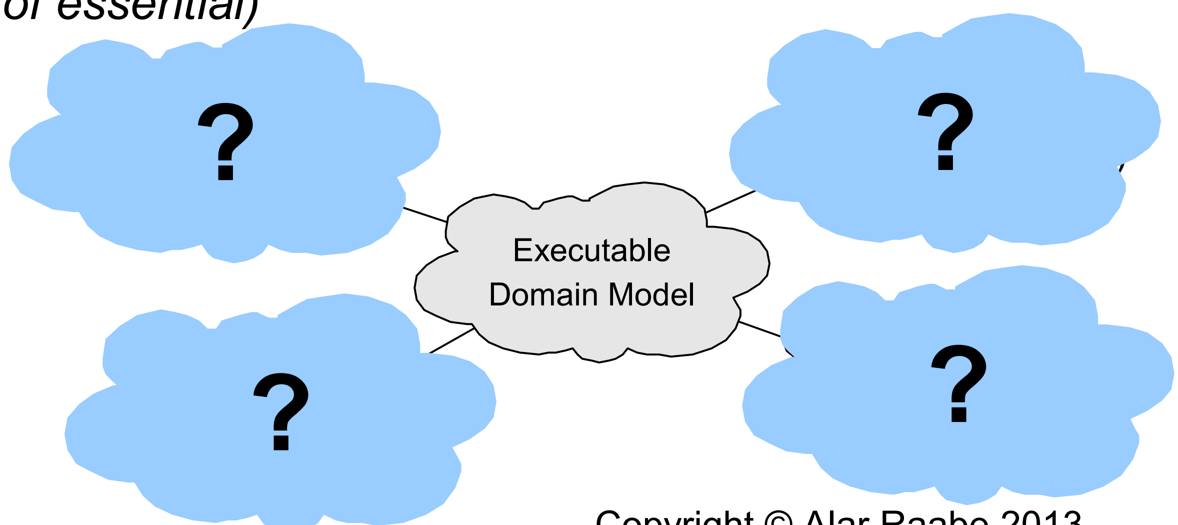


Software System is modelled according to relevant reality

Domain-Driven Design

Designing by building a domain model

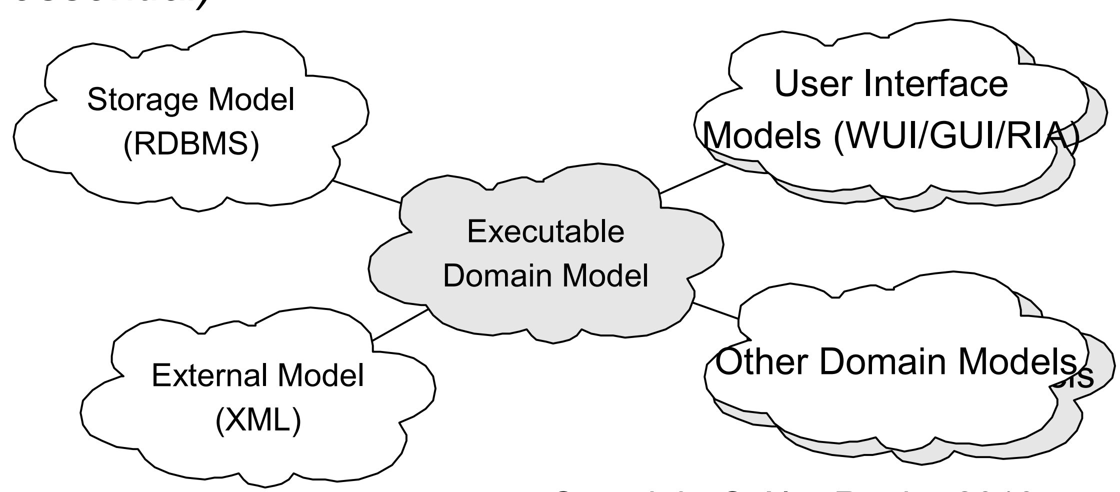
- Domain-Driven Design – a way of thinking and a set of priorities, for accelerating software projects, which deal with complicated domains [Evans]
  - the primary focus should be on the domain and domain logic
  - complex domain designs should be based on a model
- Some techniques and practices of Domain-Driven Design
  - *Declarative design (executable specification)*
  - *Conceptual contours (modules)*
  - *Distillation (separation of essential)*



Domain-Driven Design

Designing by building a domain model

- 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 and practices of Domain-Driven Design
  - *Declarative design (executable specification)*
  - *Conceptual contours (modules)*
  - *Distillation (separation of essential)*



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
        - Model-Driven Software Development
        - Generative Programming
        - Domain Specific Languages
- Practical Aspects
        - Model Management
        - Best Practices
        - Examples
- Conclusions
- References

Models as Primary Artifacts

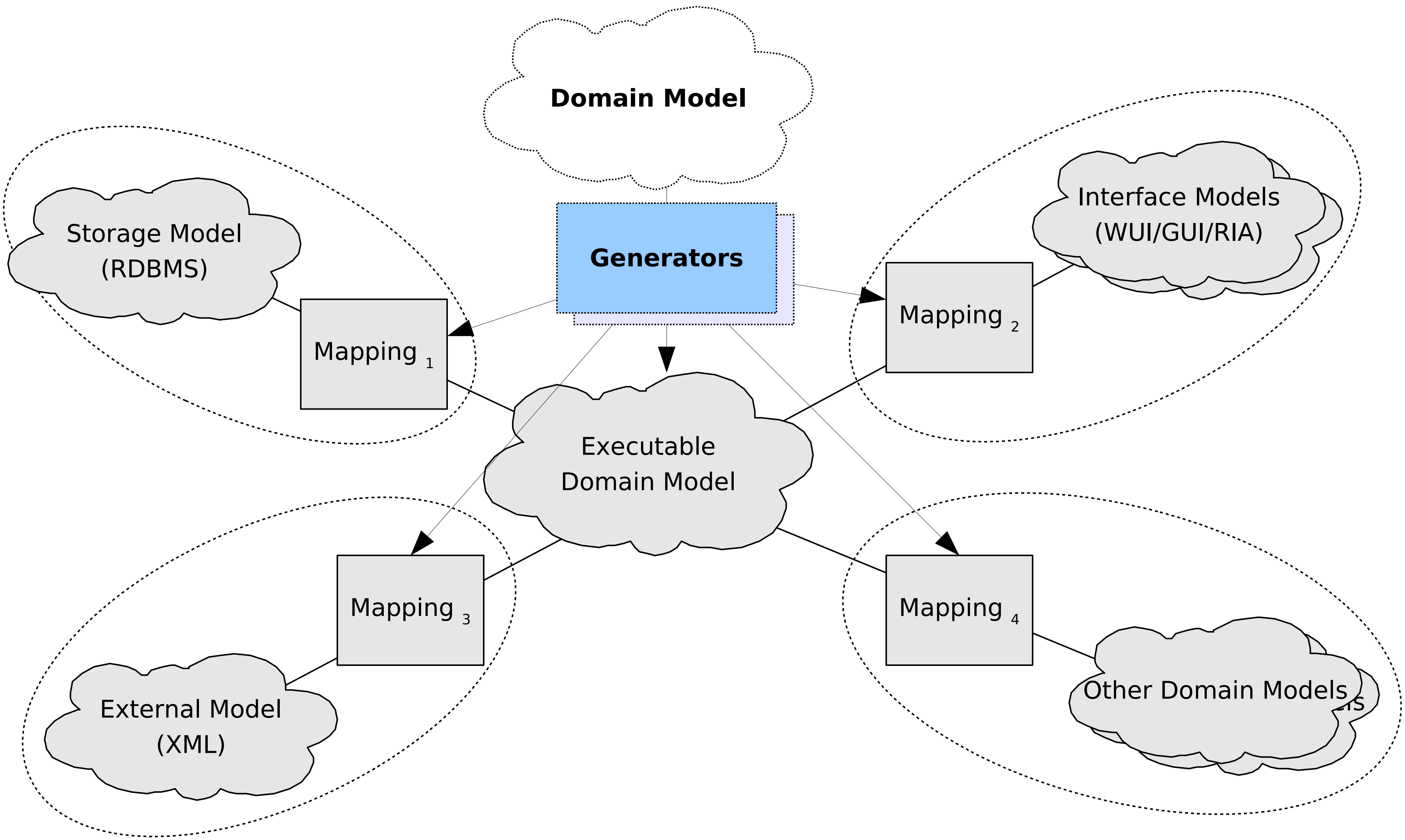
Using models to raise level of abstraction

- History
  - Shlaer-Mellor method → models with precise semantics

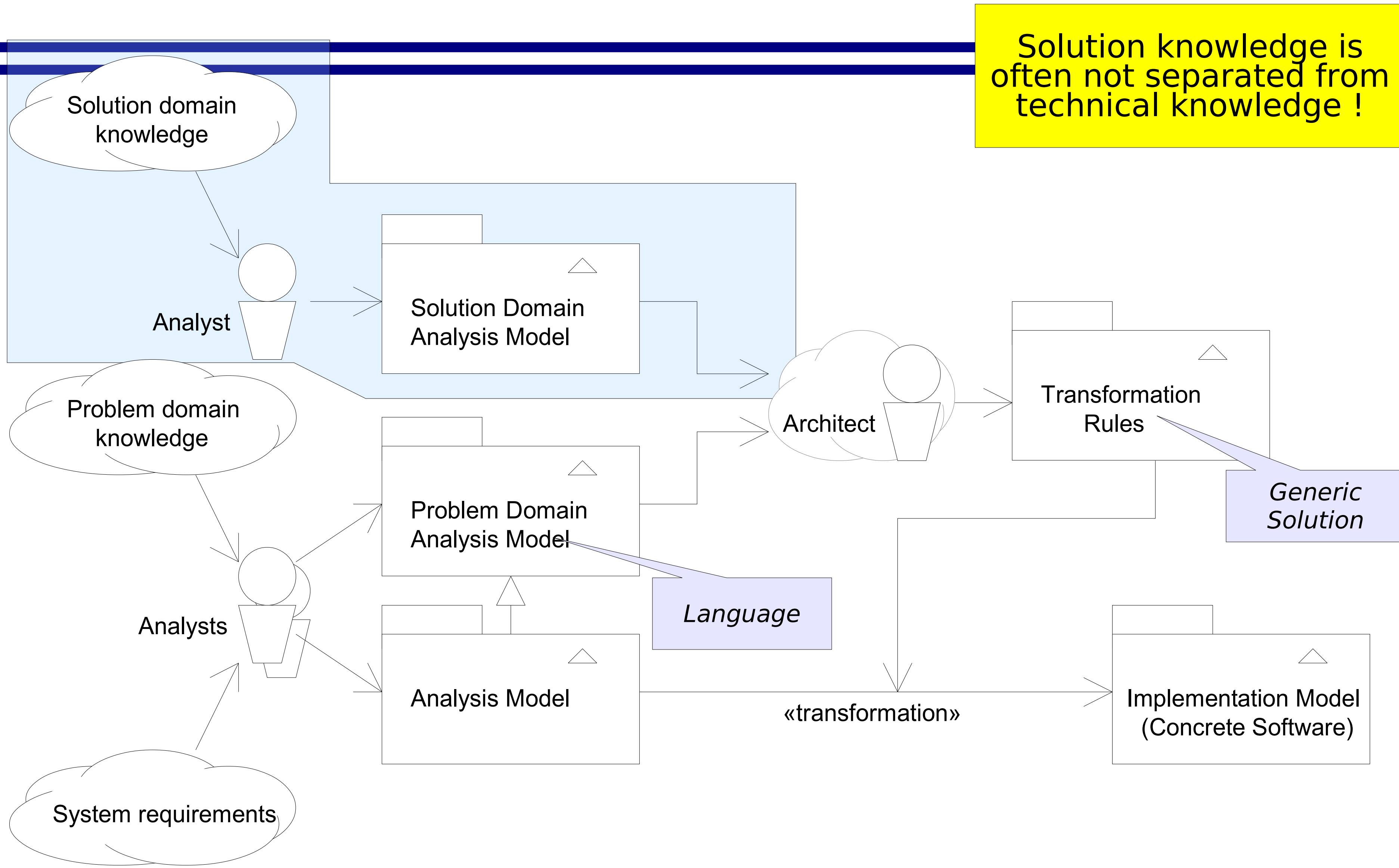
- Main Techniques
  - Model-Driven Software Development (MDSD)
  - Generative Programming
  - Domain Specific Languages (external & internal)

- Examples
  - Application Generators
  - CASE Tools
  - OMG MDA & Executable UML
    - fUML (Foundational Subset for Executable UML Models)
      - operational style description of structural and behavioral semantics
    - Alf (Action Language for fUML)
      - textual description of fine-grained behavior of the system (concrete syntax corresponding to fUML abstract syntax)

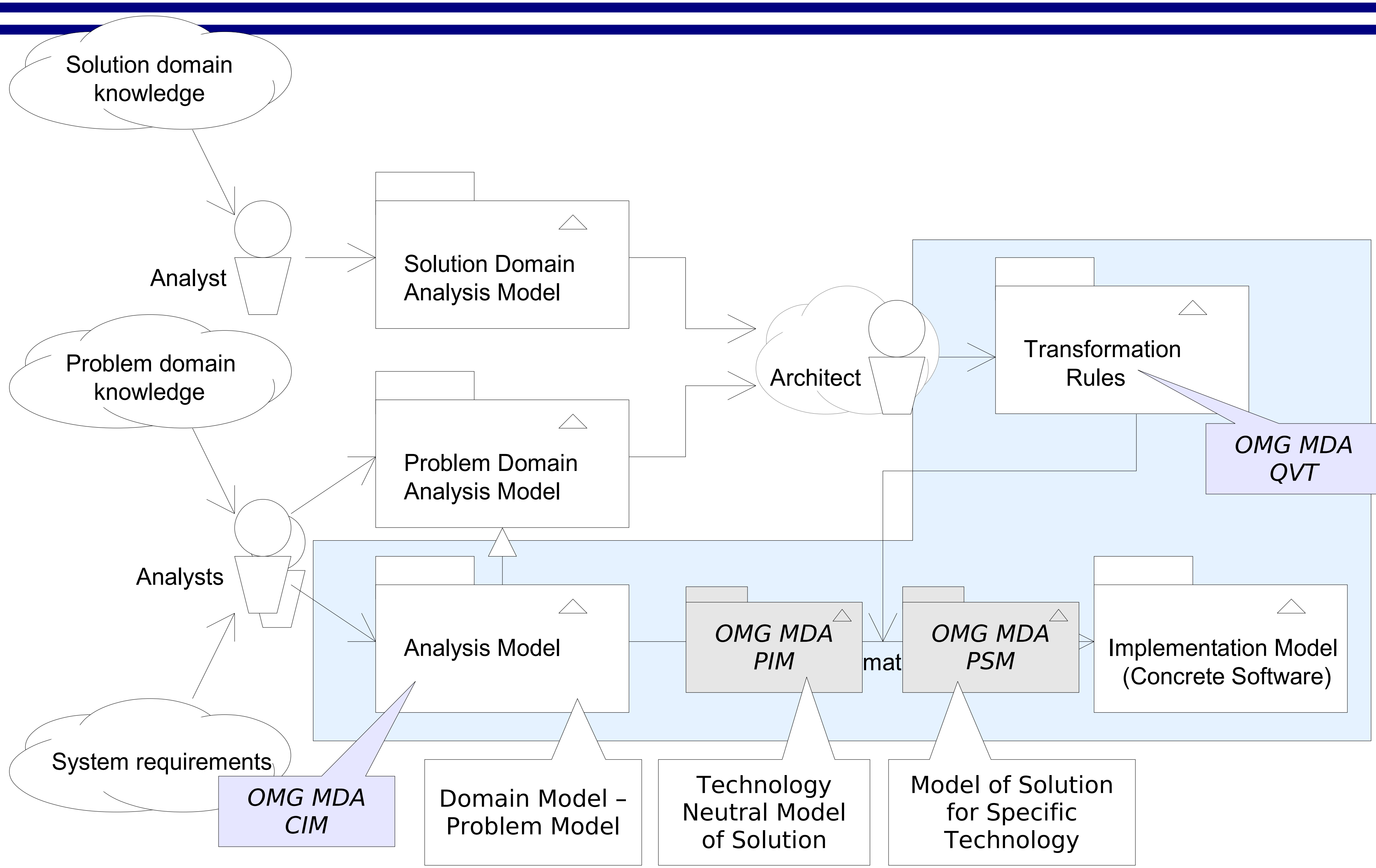
Domain Model → Source for Solution



MDSD Approach



OMG MDA Approach



Generative Programming

[Czarnecki, Eisenecker]

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

Generative Programming

[Czarnecki, Eisenecker]

Domain Specific Language (DSL)

Generator Reflection

Components + System Family Architecture

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 model traversal
- templates and frames
- transformation systems
- languages with meta-programming support
- extensible programming systems

Component Technologies
- generic components
- component models
- AOP approaches

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

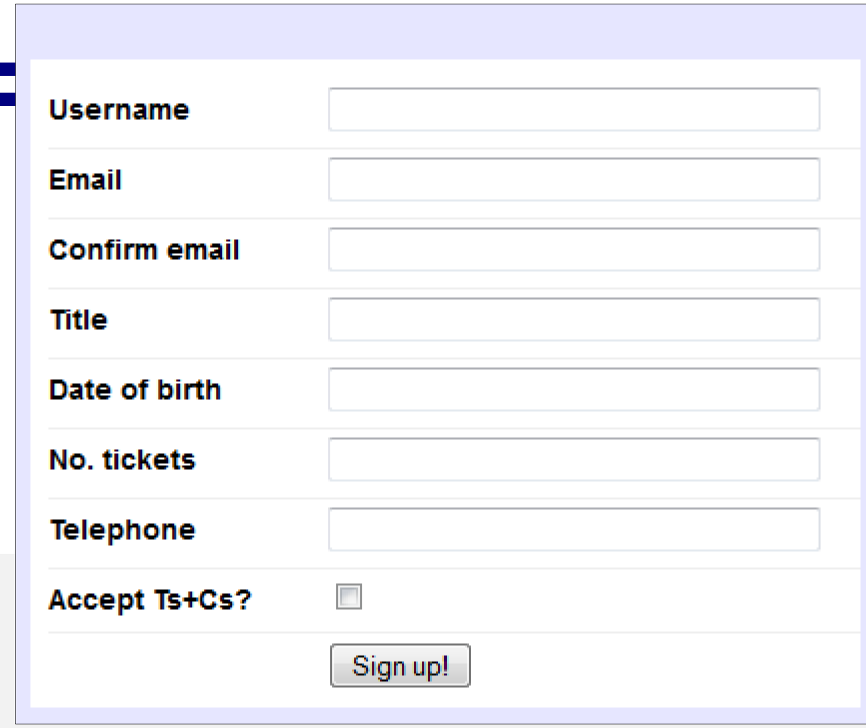
- Defining DSL
  - *concrete syntax* – representation of a DSL in a human-usable form
  - *abstract syntax* – elements + relationships without representation
  - *semantics* – meaning of the expressable phrases and sentences

- Technologies
  - Internal DSLs
    - Built-in features of languages (e.g. C++ templates, Lisp Macros, ...)
    - Extensible languages (e.g. Scala, Ruby, JavaScript, Seed7, XL, ...)
    - Well-Designed APIs
  - External DSLs
    - Textual languages (e.g. XML, xText, ...)
    - Graphical languages (e.g. UML, MetaCASE, ...)
    - Interactive wizards

WARNING:
Don't be too Clever !

Internal DSL

- 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');
...
```

External DSL

Model in EBNF

<entity>	::=	“entity” <name> [“extends” <name>] “{“ { <feature> } “}”
<feature>	::=	<attribute> | <reference>
<attribute>	::=	<type> <name> “;”
<reference>	::=	“ref” [“+”] <type> <name> [“<->” <name>] “;”

- xText (oAW)

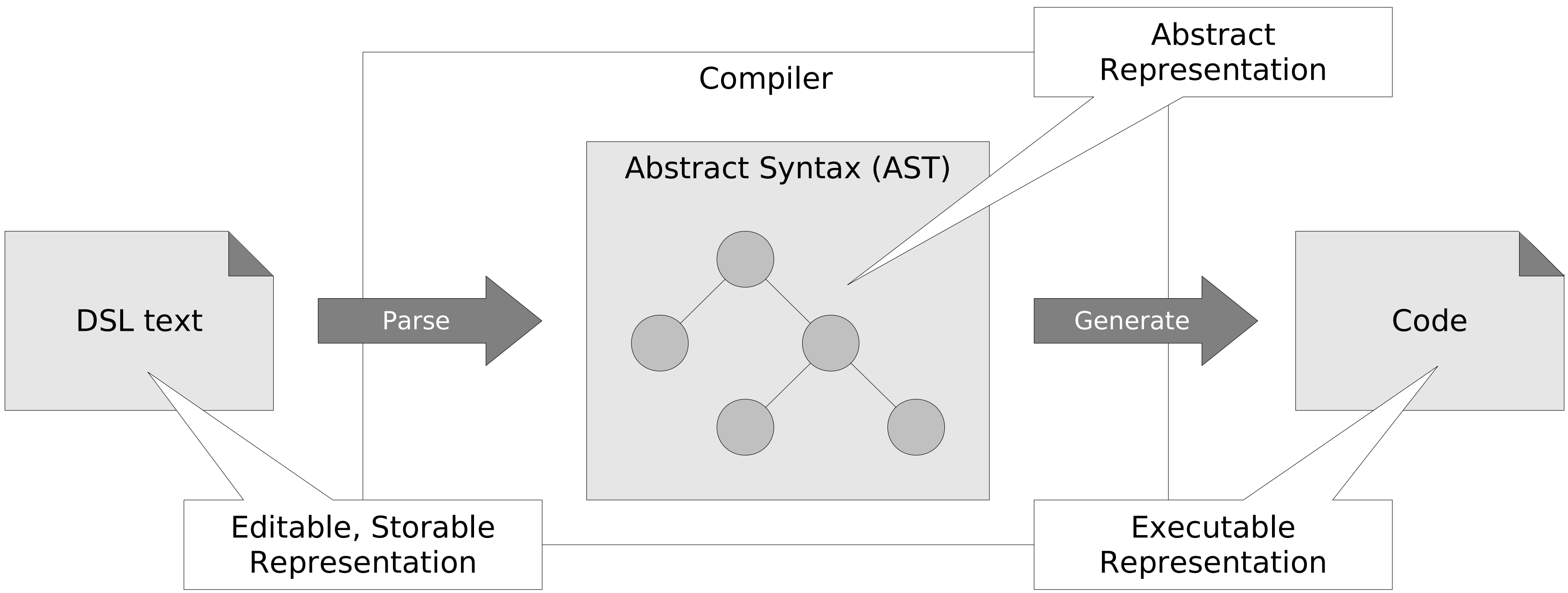
```
Entity :
  "entity" name=ID ("extends" superType=[Entity])?
  "{"
     (features+=Feature)*
  "}";
Feature :
  Attribute | Reference;
Attribute :
  type=ID name=ID ";";
Reference :
  "ref" (containment?"+")? type=ID name=ID ("<->" oppositeName=ID)? ";";
```

- Example

```
entity Customer {
  String  fullName;
  ref     +Address address <-> resident;
  Integer ageInFullYears;
  Boolean isPremiumCustomer;
}
```

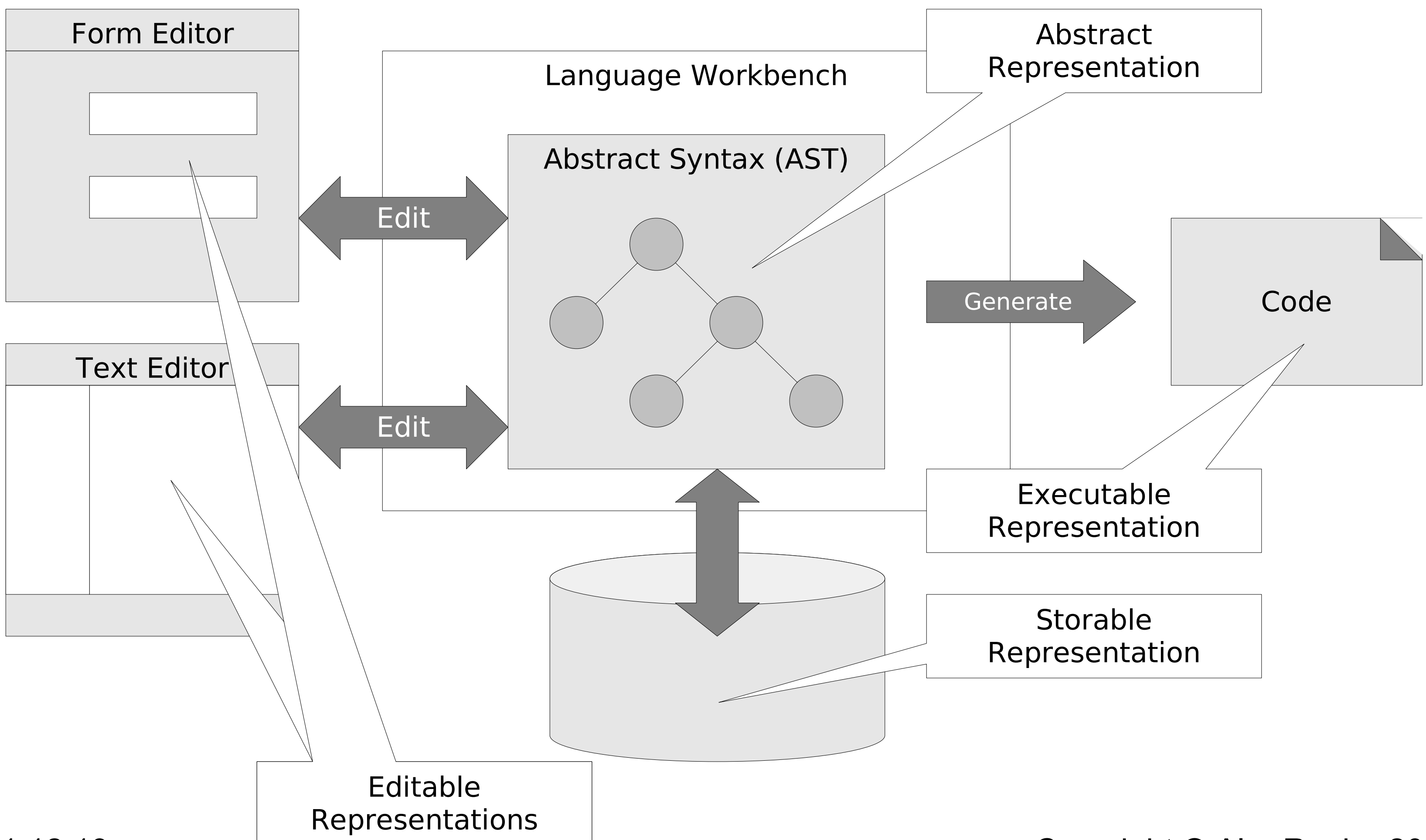
DSL Implementation 1

- Compiler-Based



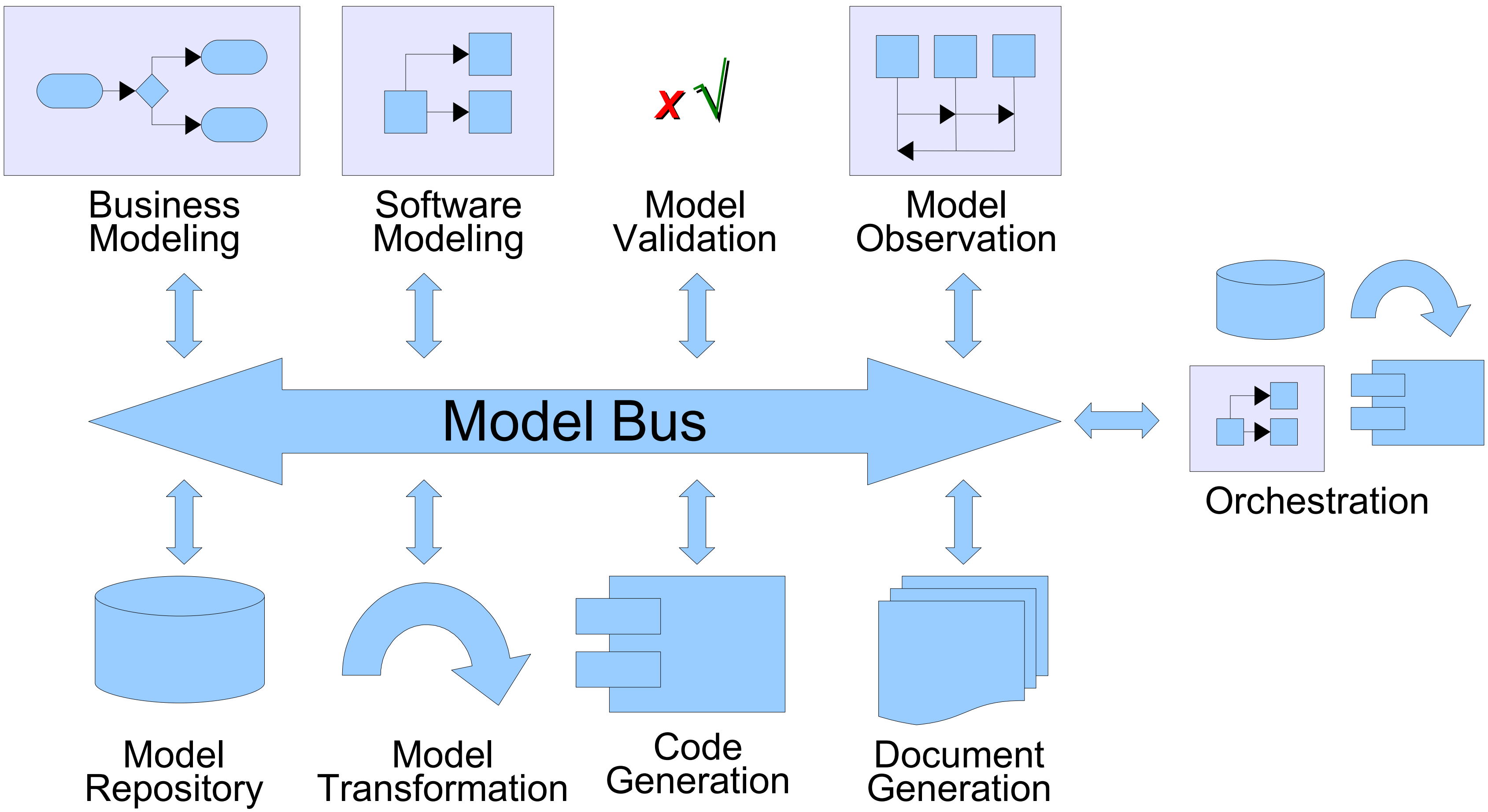
DSL Implementation $_2$

- Language Workbench



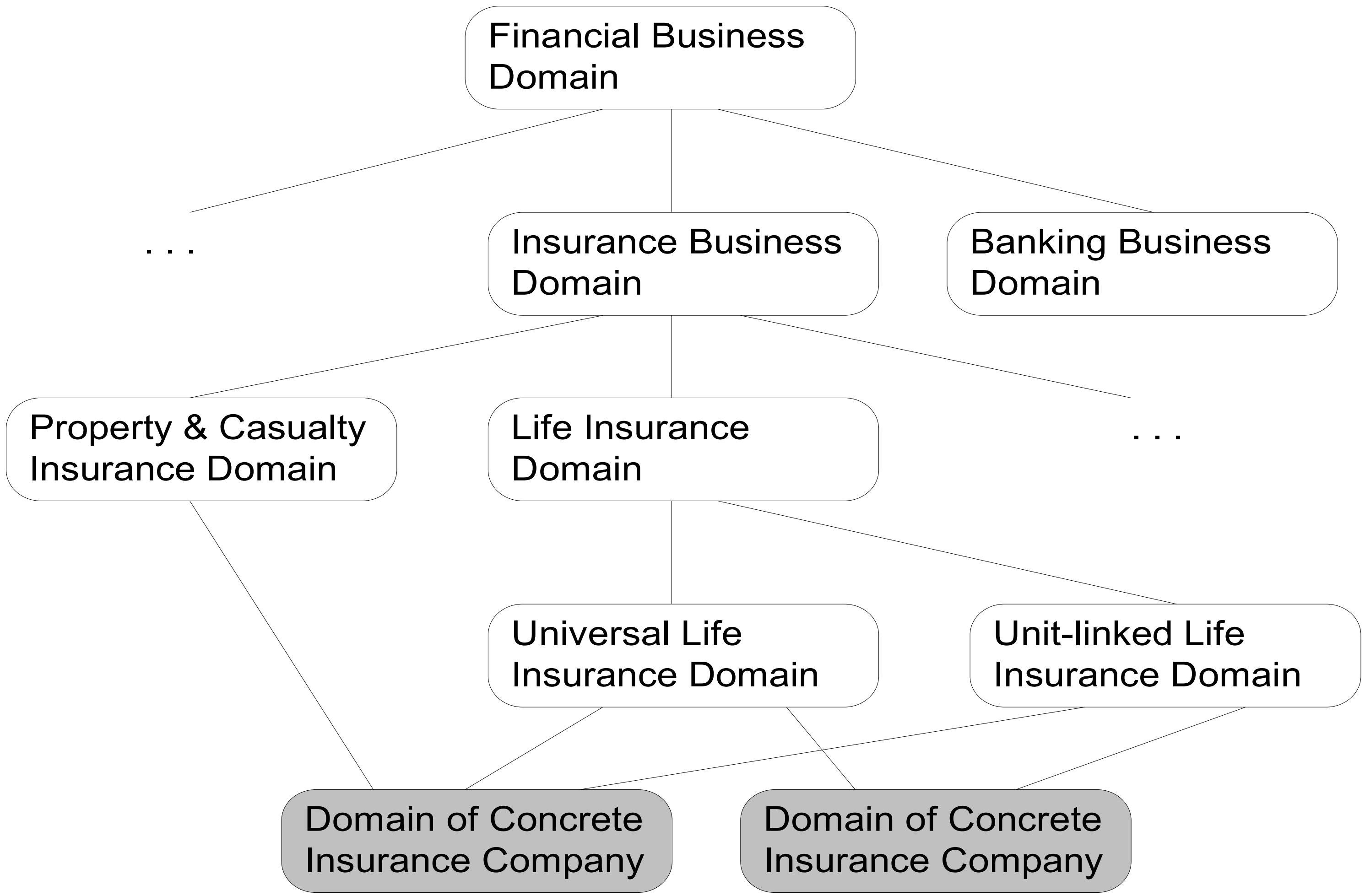
MDSD Implementation

- Model Bus (e.g. Eclipse MDDi)

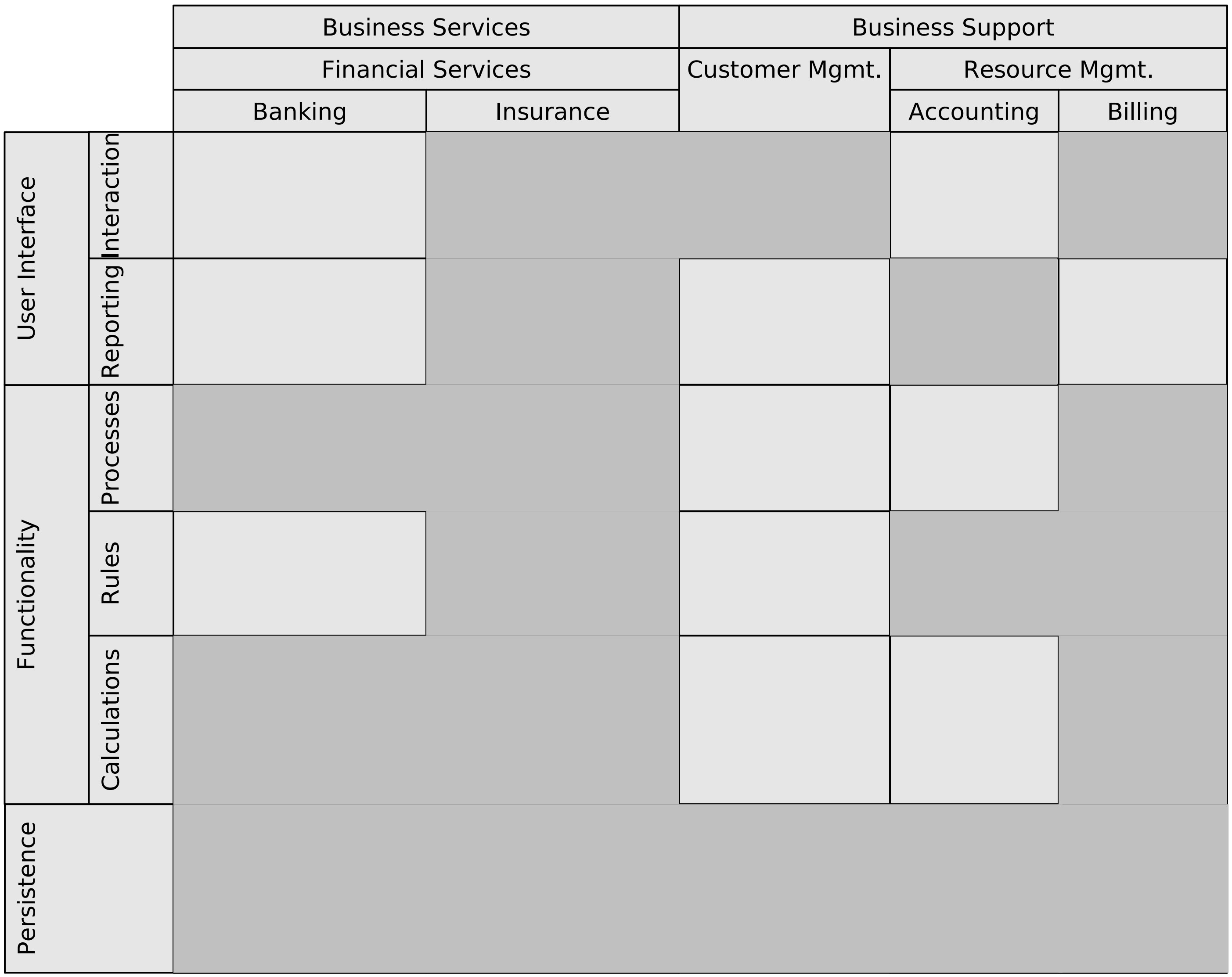


Content

Network of Problem Domains → Specific Domain is a Combination of Generic Domains



Different Problem and Solution Domains in a Specific System → Many Dimensions



Model Management

- Relationships between Models
  - “inheritance” – extension of models (package/model merge in UML2)
  - correspondence mappings between models
  - references to external models (package/model import in UML 2)
- Operations on Models (e.g. Epsilon & Atlas on Eclipse)
  - calculations on models
    - model validation
    - comparing models
    - transformations of models (to other models or to text)
  - editing models
    - graphical model editors
    - form-based model editors
    - text-based model editors
  - storing models
    - repository
    - source code control
    - embedding into code

Domain-Driven Design Best Practices

[Evans]

- Use the Domain Model as **Ubiquitous Language**
- Design to Reflect Domain Model –

  Avoid Divide between Analysis and Design
  - Domain Model should be constrained to support efficient implementation
- Express Domain Model in Code – Hands-On Modelling
  - with Services, Entities, Aggregates and Value Objects
- Isolate Domain with Layered Architecture
  - Presentation Layer
  - Application Layer
  - Domain Layer
  - Infrastructure Layer

MDSD Best Practices 1

[Voelter, ...]

- During the Software Development
  - **Don't Reverse Engineer – Model is Primary Artifact**
  - Don't Manage Generated Code in Revision Control System
  - Integrate the Generator/Generation into the Build Process
  - Regenerate Frequently
  - Use Meta-Model as Ubiquitous Language
  - Use Graphical and Textual Syntax to Support Modeller
  - Use Configuration by Exception – use implicit knowledge
- When Generating the Code
  - Generate Clean and Readable Code
  - Use the Compiler (to Guide the Developer)
  - Separate the Generated and Manually Created Code

MDSD Best Practices 2

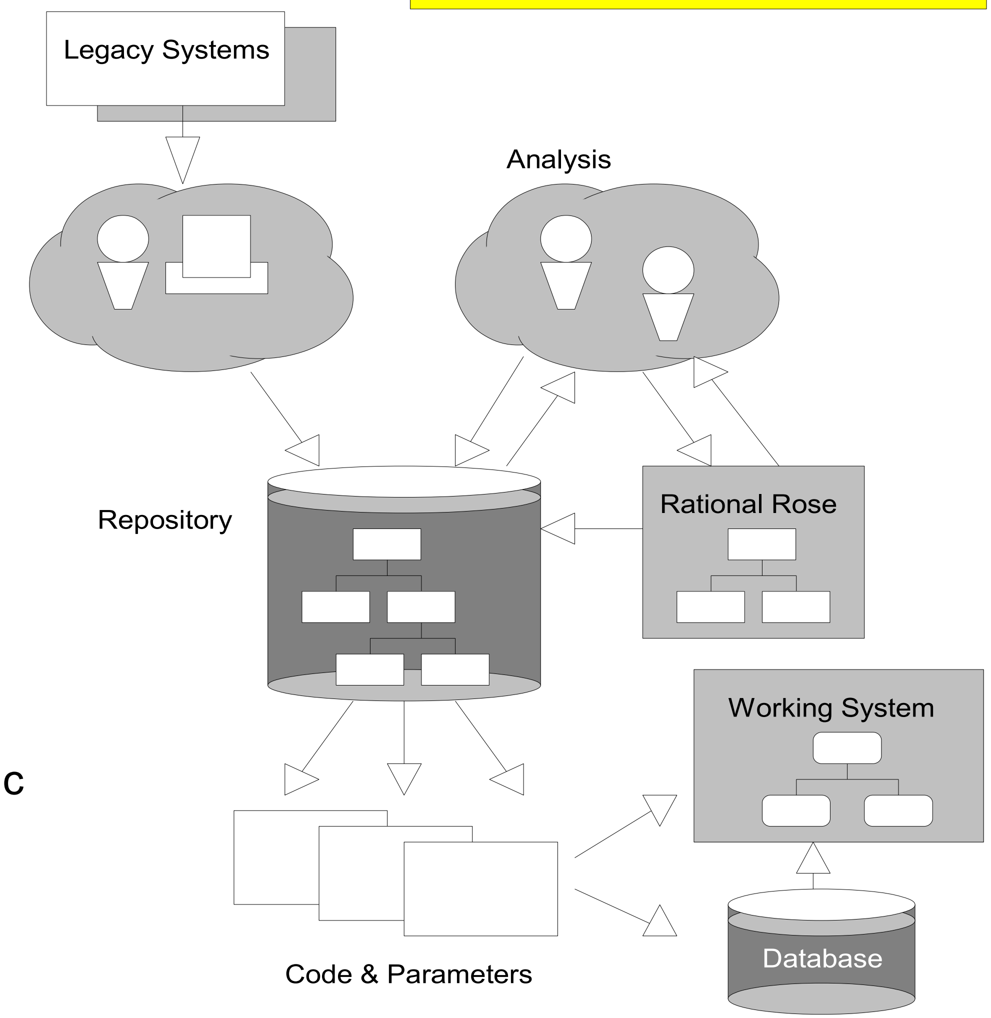
[Voelter, ...]

- During the Language and Tools Development
  - Develop DSLs Incrementally
  - Teamwork (Tools) Prefer(s) Textual DSLs
  - Many Small DSLs – Concentrate on the Task
  - Select Suitable Target – Avoid too Complex Meta-Models
- During the Tools Development
  - Test the Generator(s) (using Reference Model)
  - Develop Model Validation (Iteratively)
  - Use Model Transformations to Reduce Complexity
  - Keep Translation Steps as Small as Possible

Overview of Once&Done Software Process

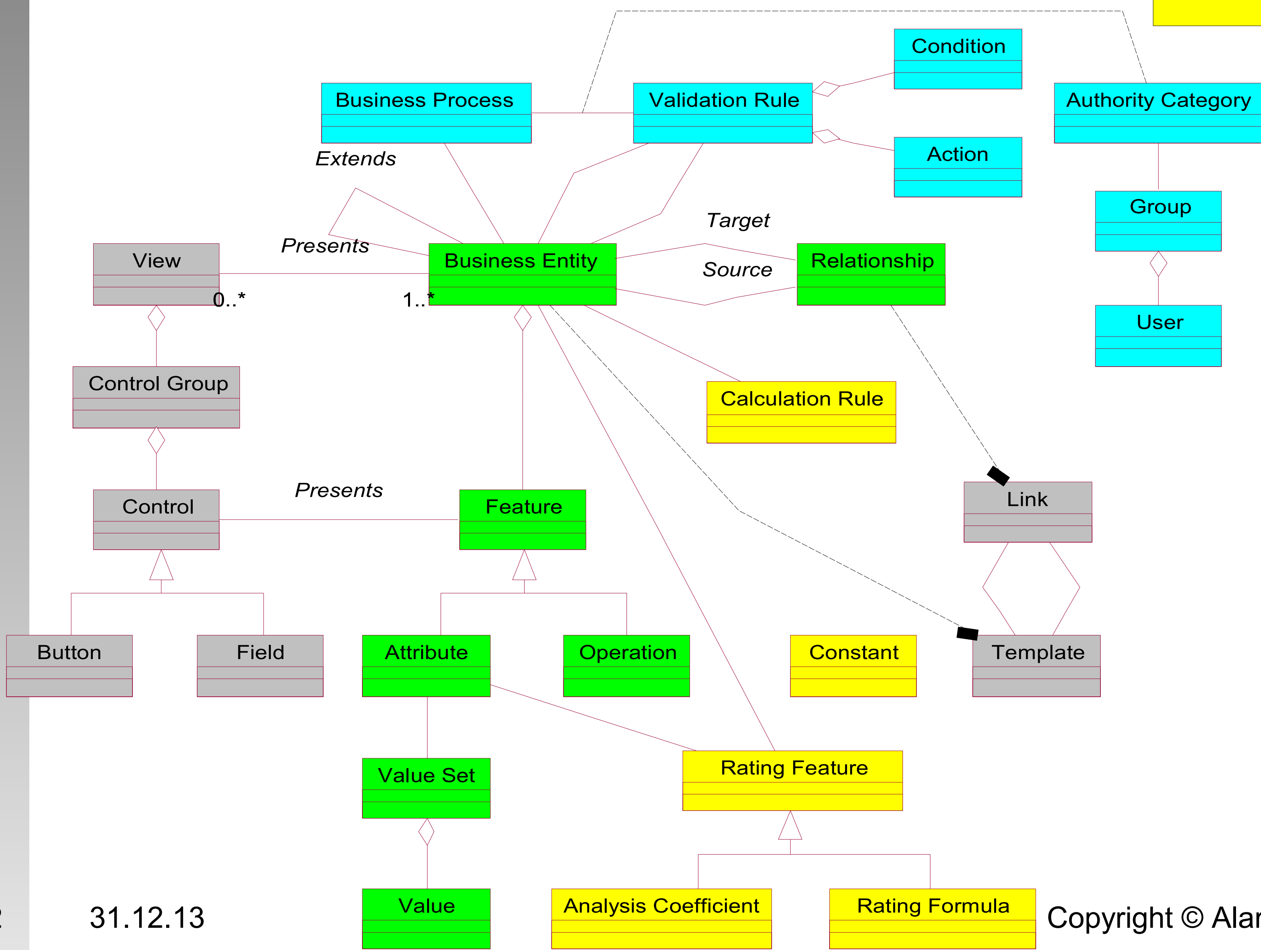
A model-driven technology for insurance systems product-line

- Beginning
- Analysis
    - Business Domain Analysis
    - Modelling Domain Objects
    - Modelling 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



Extended OOA/OOD Meta-Model

a DSL for Insurance Systems



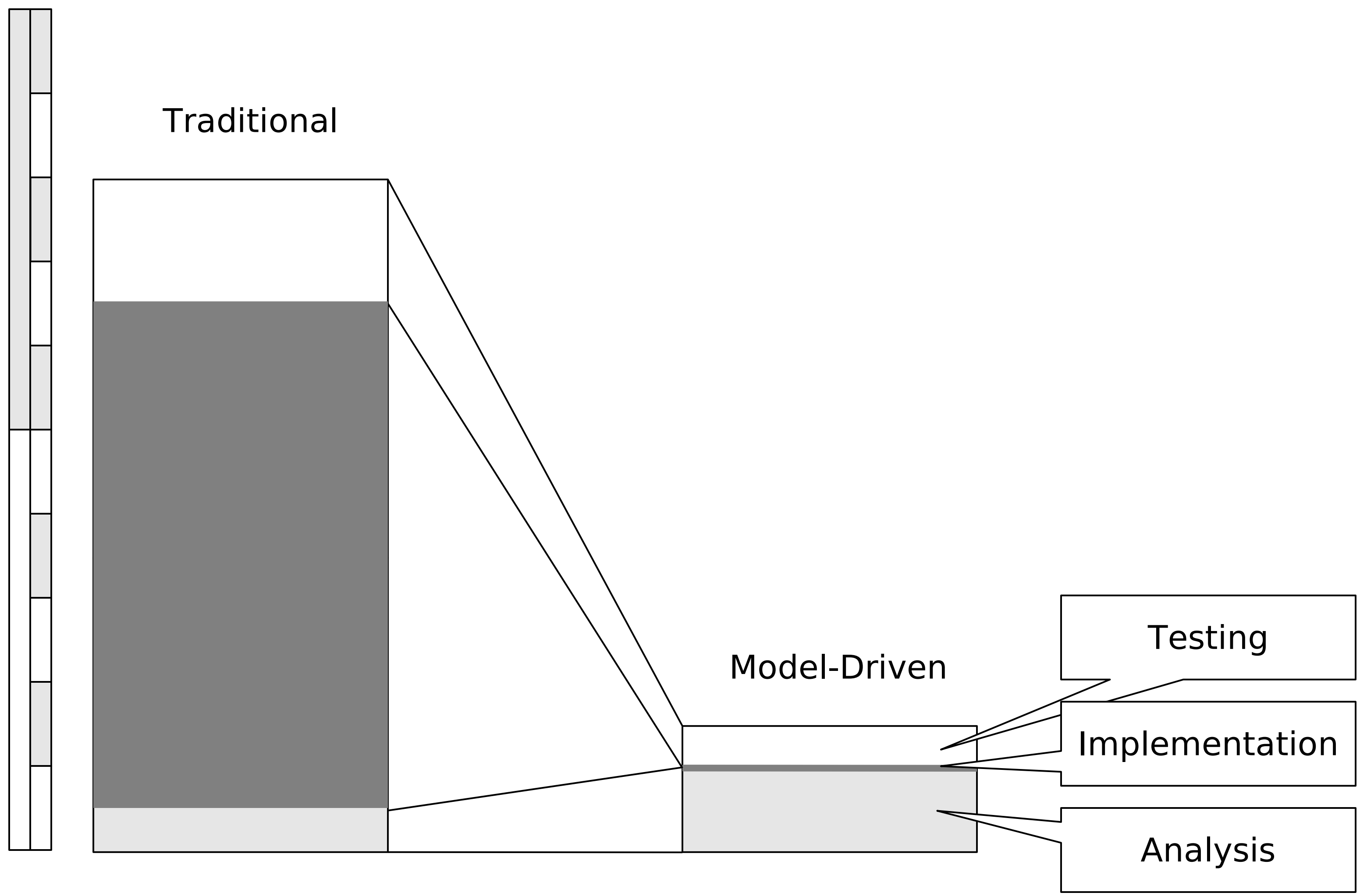
Once&Done – Results

- 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 Rich Client, further
    - HTML-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



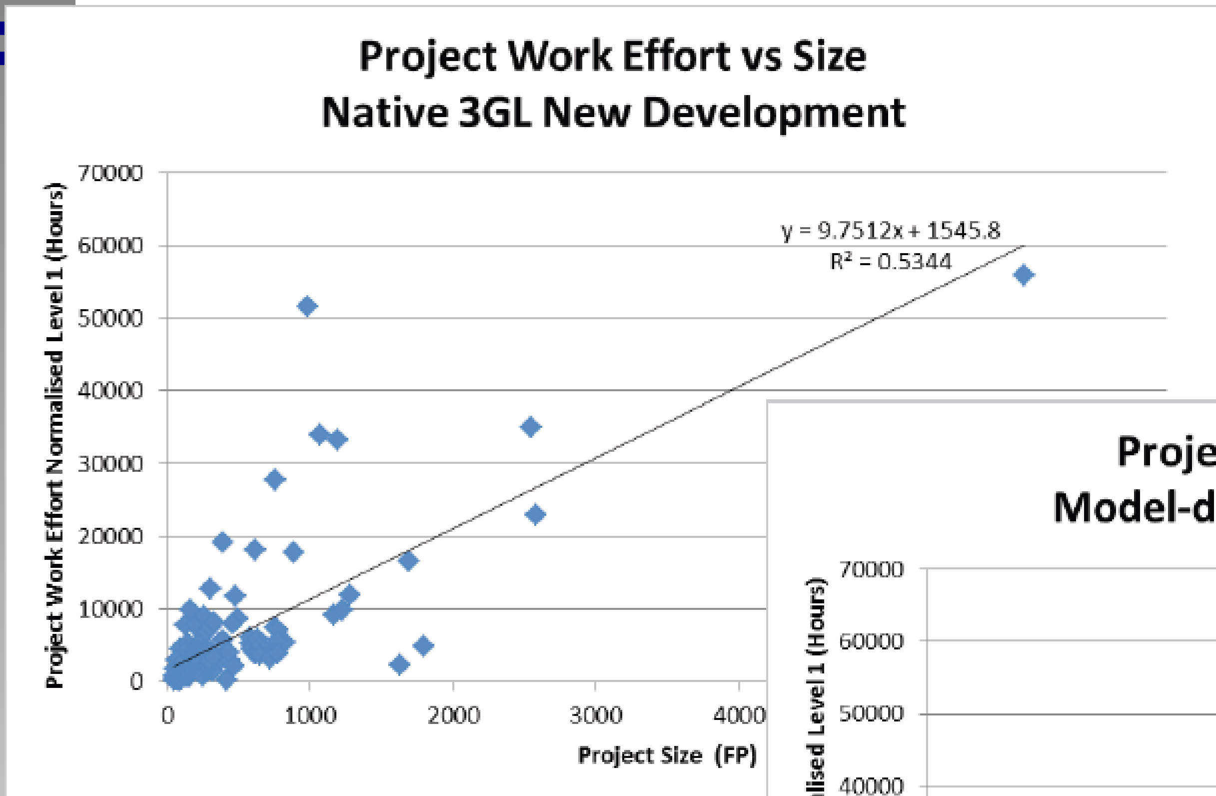
Lessons Learned

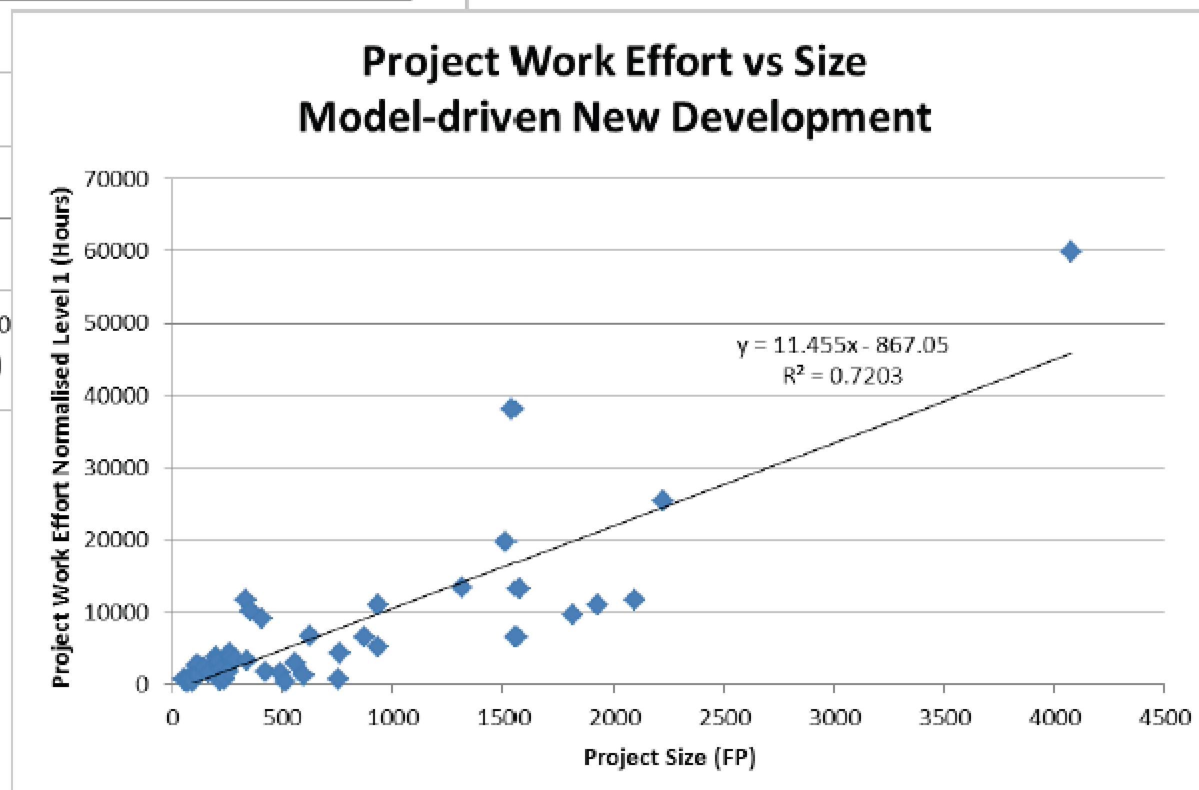
Too much time for solving the business problem !

- Modelling is hard work and requires domain knowledge
- Project budget structure changes when using generation
- Generated system can be used as analysis tool
- 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 with graphical models
- Interpreters of meta-info (heavily parametric software components) are very difficult to debug – here generation/compilation is better

Projects become more predictable

Statistics from
CA Technologies





RISLA – Language for Product Models

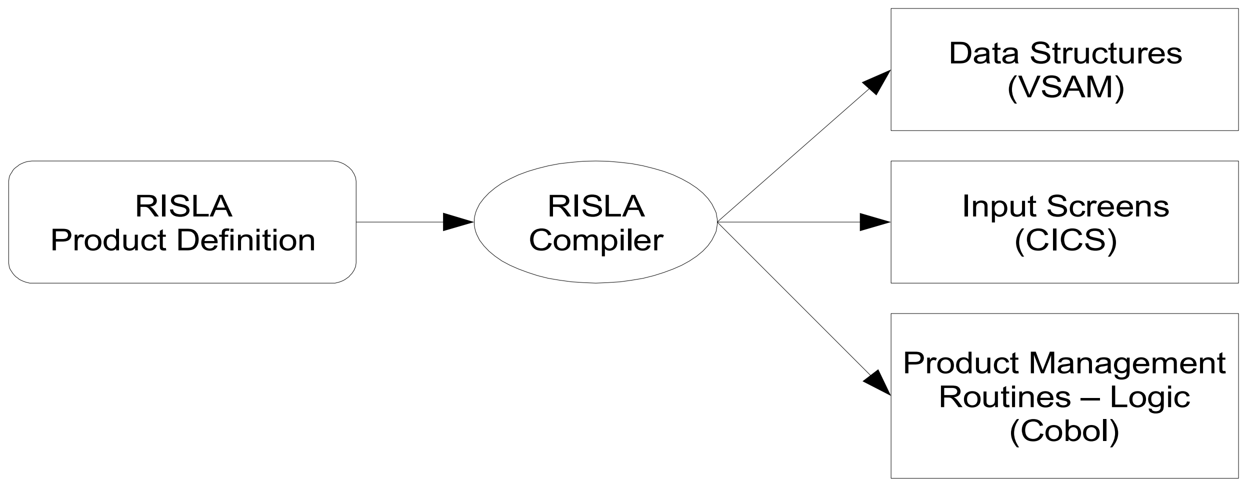
a DSL for credit products

- 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



MLFi – Language for Financial Instruments and Contracts

a DSL for financial instruments and contracts

- Financial Instrument (American Option)

```
american :: (Date,Date) -> Contract -> Contract
american (t1,t2) u
    = get (truncate t1 opt) `then` opt
where
    opt :: Contract
    opt = anytime (perhaps t2 u)
```

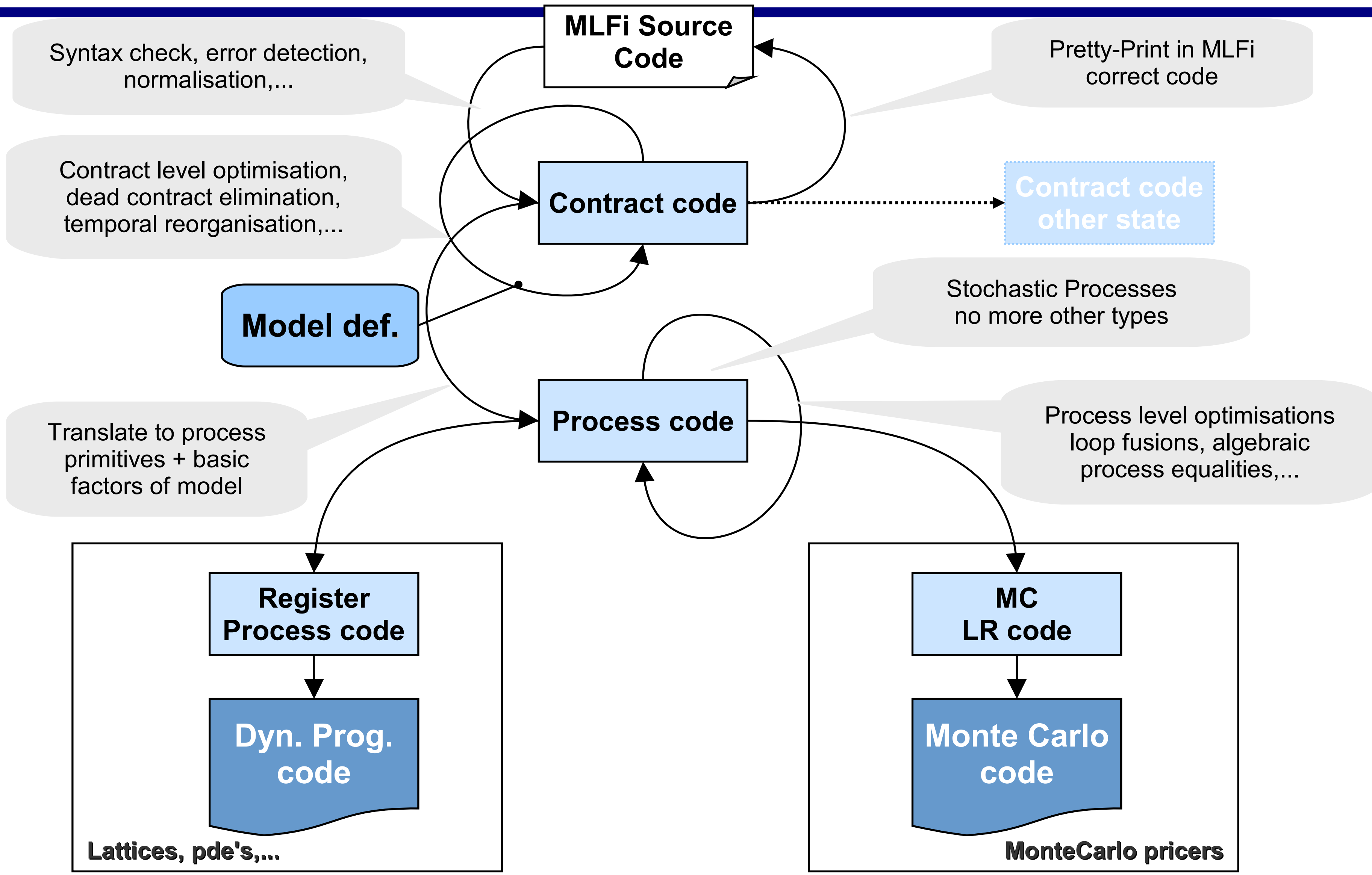
- Custom-built Contract

Against the promise to pay $2.00 on 27.12, the holder has the right, on 04.12, to choose between receiving $1.95 on 29.12, or having the right, on 11.12, to choose between receiving €2.20 on 28.12, or having the right, on 18.12, to choose between receiving £1.20 on 30.12, or paying immediately €1.0 and receiving €3.20 on 29.12.

```
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))
```

EXAMPLE

Generating Code for Financial Instrument Agreement Valuation

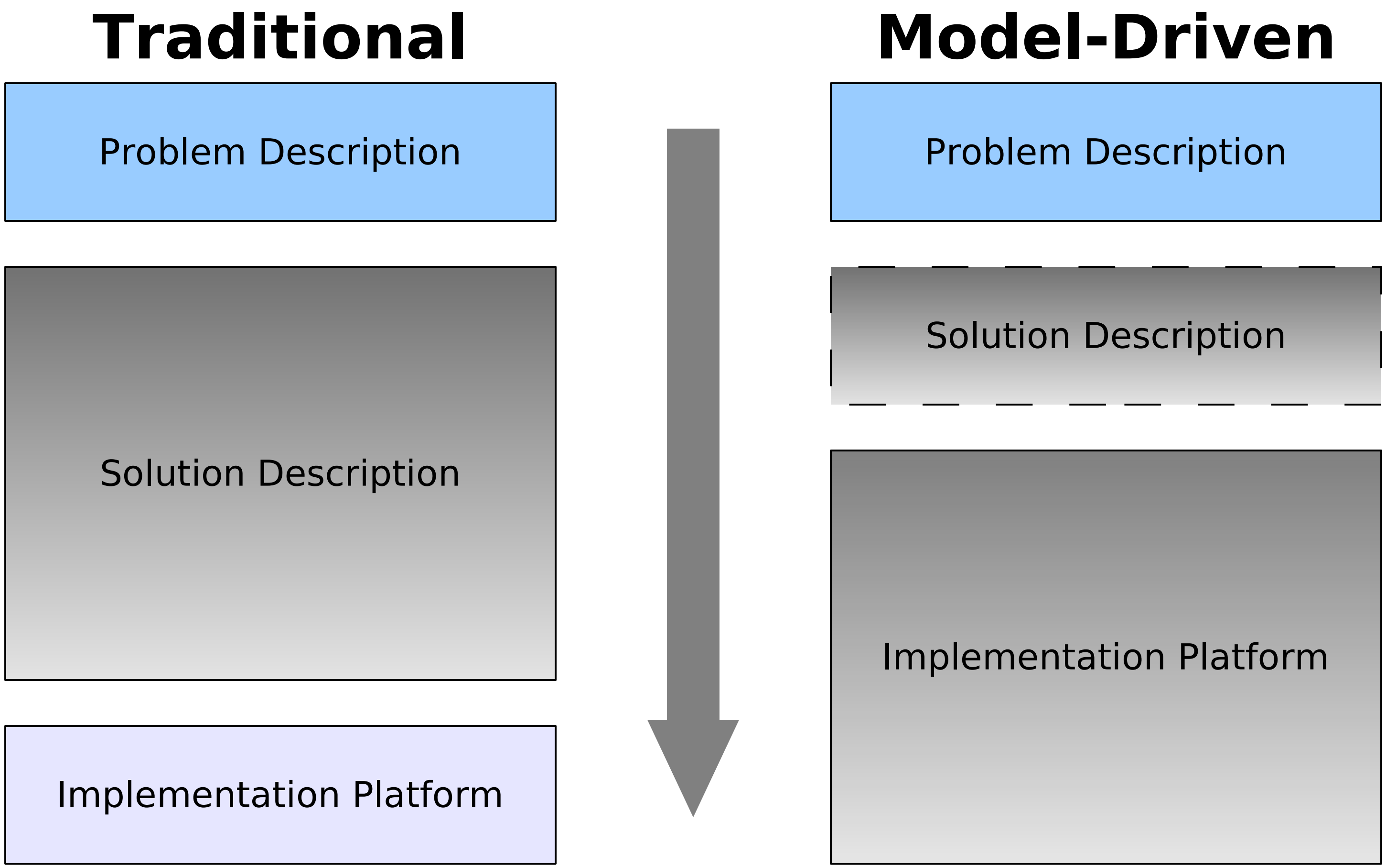


Content

- 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

Compared to the Traditional Development

Reducing the gap



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 (fight Complexity)**
  - to represent information about problems/solutions in different Domains **use several Models with different Meta-Models**
- **Let the Models drive the Analysis & Design**
  - 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
  - Eric Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, 2004
    - http://domaindrivendesign.org/

- Some WWW sites to look
    - http://www.omg.org/mda
    - http://www.eclipse.org/modeling/emf/
    - http://www.infoq.com/minibooks/domain-driven-design-quickly
    - 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!

LWC 2013 – QL (questionnaires)

```
form Box1HouseOwning {
     hasSoldHouse: "Did you sell a house in 2010?" boolean
     hasBoughtHouse: "Did you buy a house in 2010?" boolean
     hasMaintLoan: "Did you enter a loan for maintenance/reconstruction?" boolean
     if (hasSoldHouse) {
          sellingPrice: "Price the house was sold for:" money
          privateDebt: "Private debts for the sold house:" money
          valueResidue: "Value residue:" money(sellingPrice - privateDebt)
     }
}
```

1

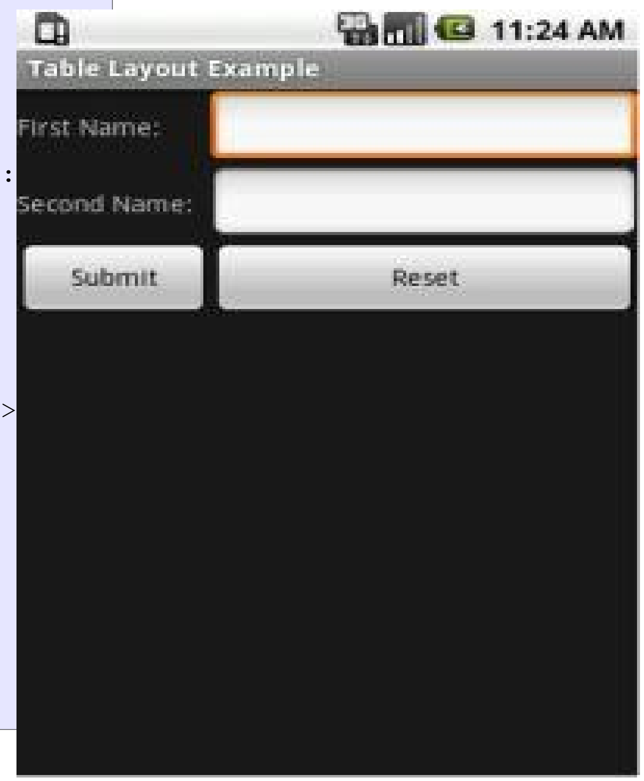
```
                 Did you sell a house in 2010? [X]
                  Did you buy a house in 2010? [ ]
Did you enter a loan for maintenance/reconstruction? [ ]
```

2

```
                 Did you sell a house in 2010? [X]
                  Did you buy a house in 2010? [ ]
Did you enter a loan for maintenance/reconstruction? [ ]
-------------------------------------------------------
               Price the house was sold for: [230000]
           Private debts for the sold house: [180000]
                              Value residue: [ 50000]
```

Android Layouts

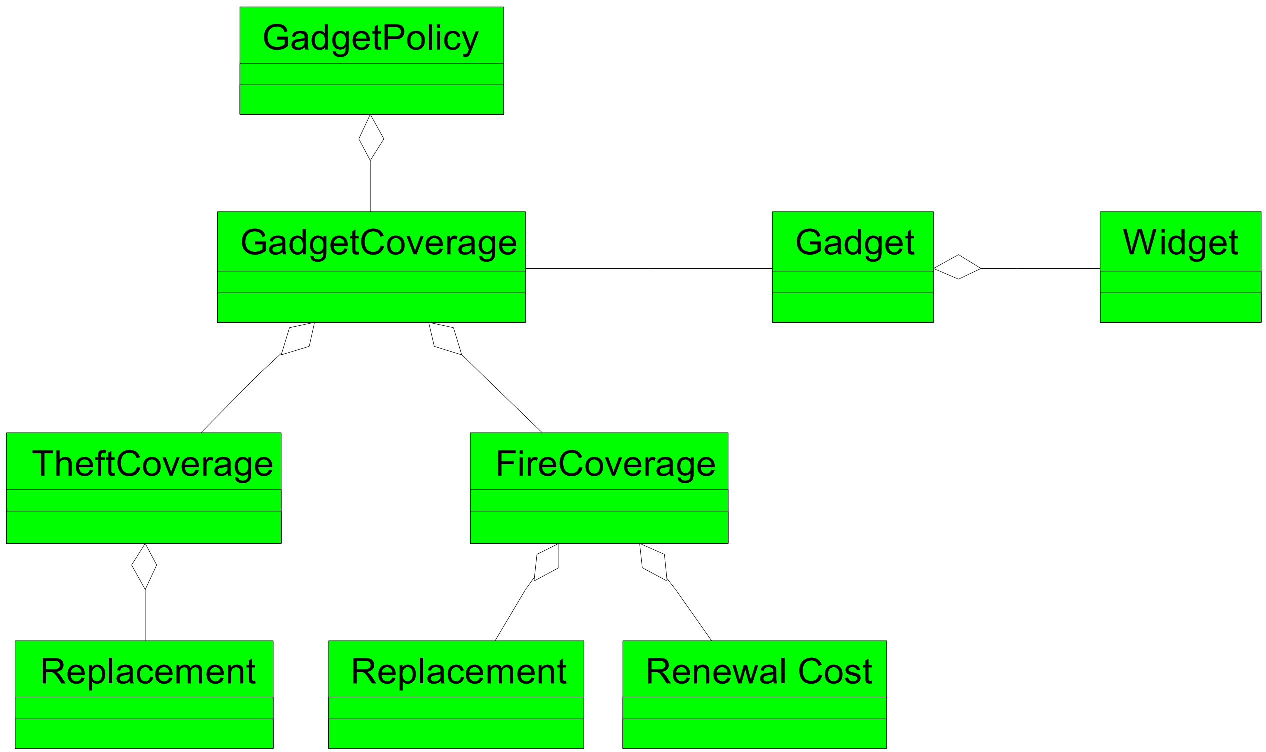
```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout android:id="@+id/TableLayout01"
    android:layout_width="fill_parent" android:layout_height="fill_parent"
    xmlns:android="http://schemas.android.com/apk/res/android">
    <TableRow android:id="@+id/TableRow01">
        <TextView android:id="@+id/TextView01" android:text="First Name:"
            android:width="100px" />
        <EditText android:id="@+id/EditText01" android:width="220px" />
    </TableRow>

    <TableRow android:id="@+id/TableRow02">
        <TextView android:id="@+id/TextView02" android:text="Second Name:
        <EditText android:id="@+id/EditText02" />
    </TableRow>

    <TableRow android:id="@+id/TableRow03">
        <Button android:id="@+id/Button01"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" android:text="Submit" />

        <Button android:id="@+id/Button02"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" android:text="Reset"
            android:width="100px" />
    </TableRow>
</TableLayout>
```



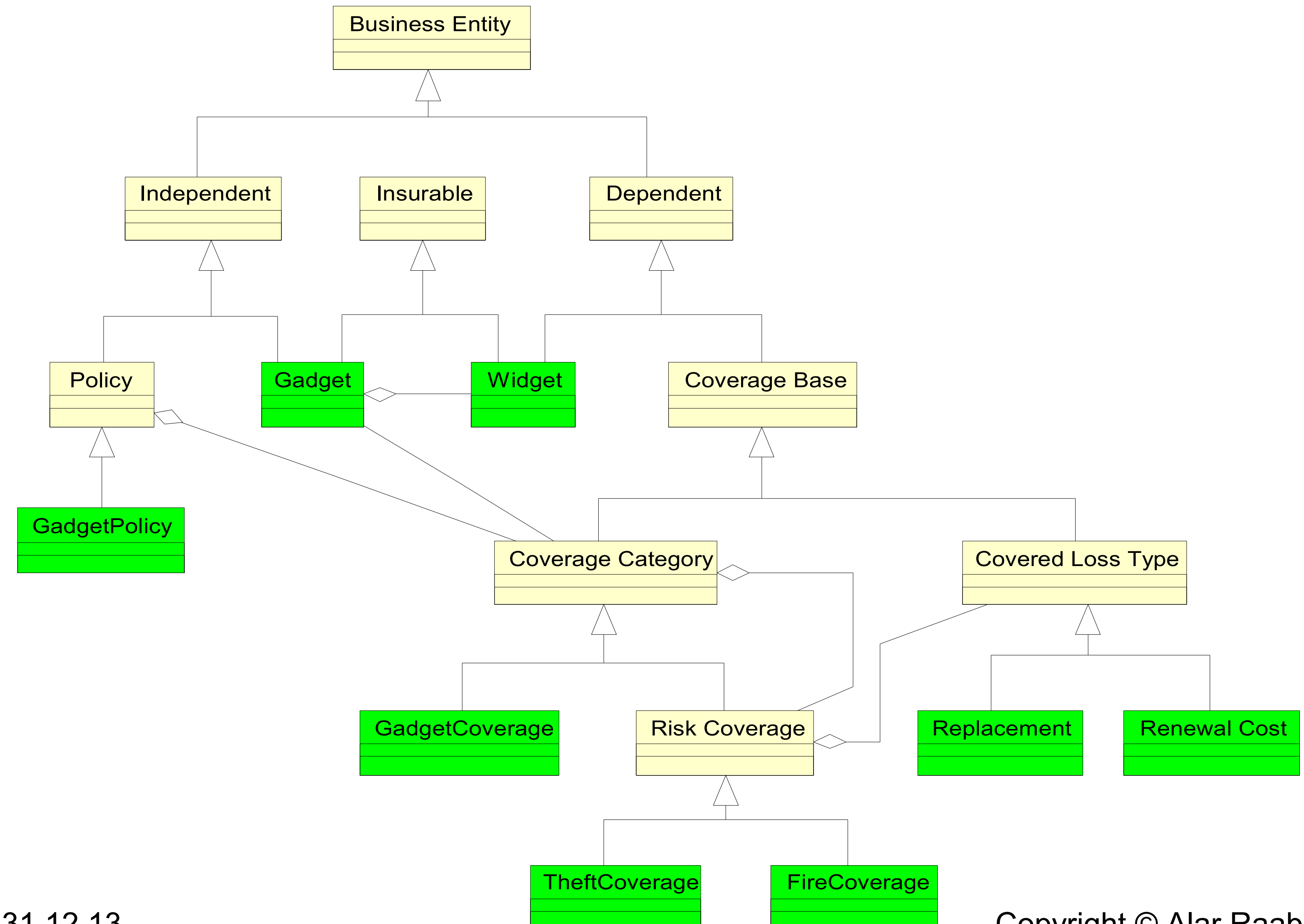
Example of Using Once&Done

- “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

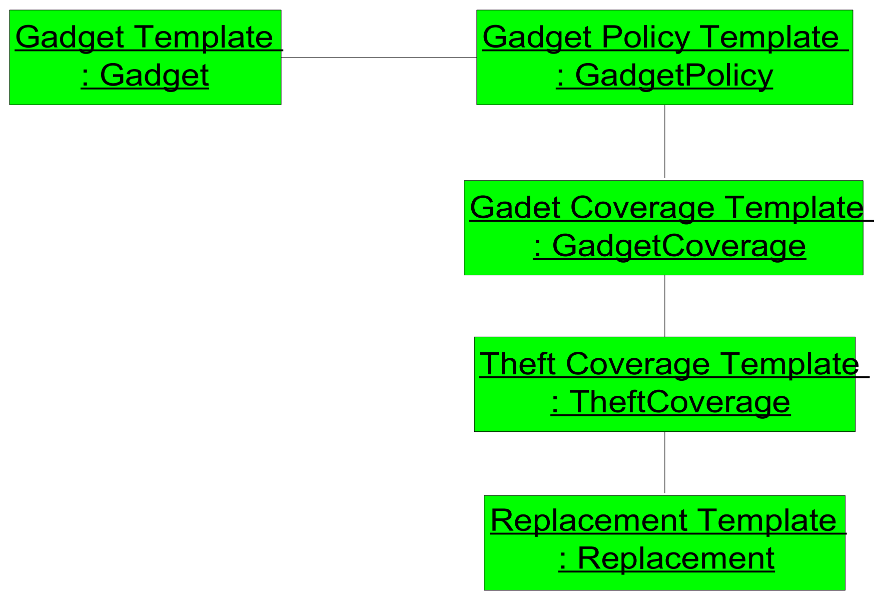
"Gadget Insurance" Analysis Model



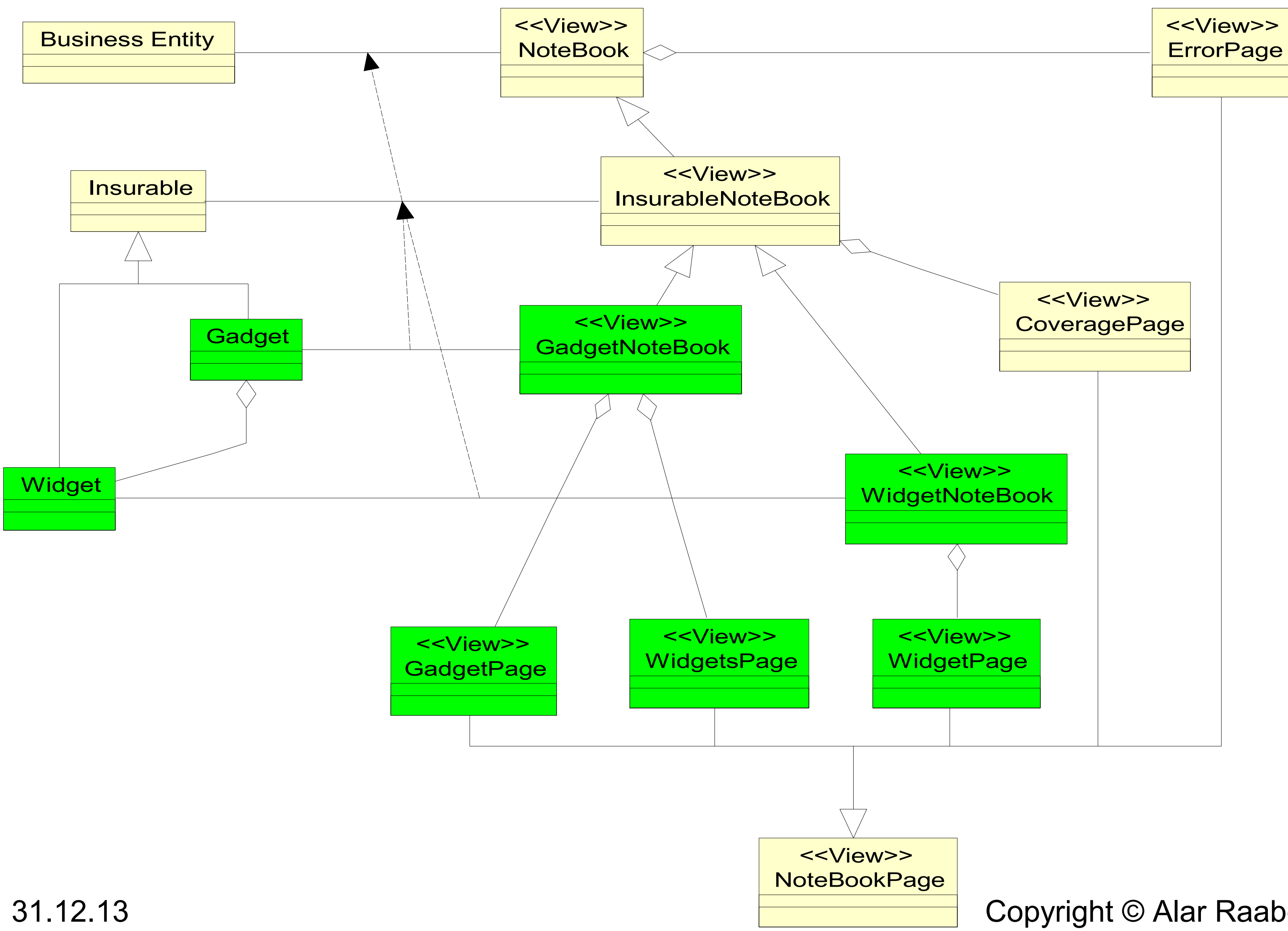
“Gadget Insurance” Model as Extension to Insurance Domain Model



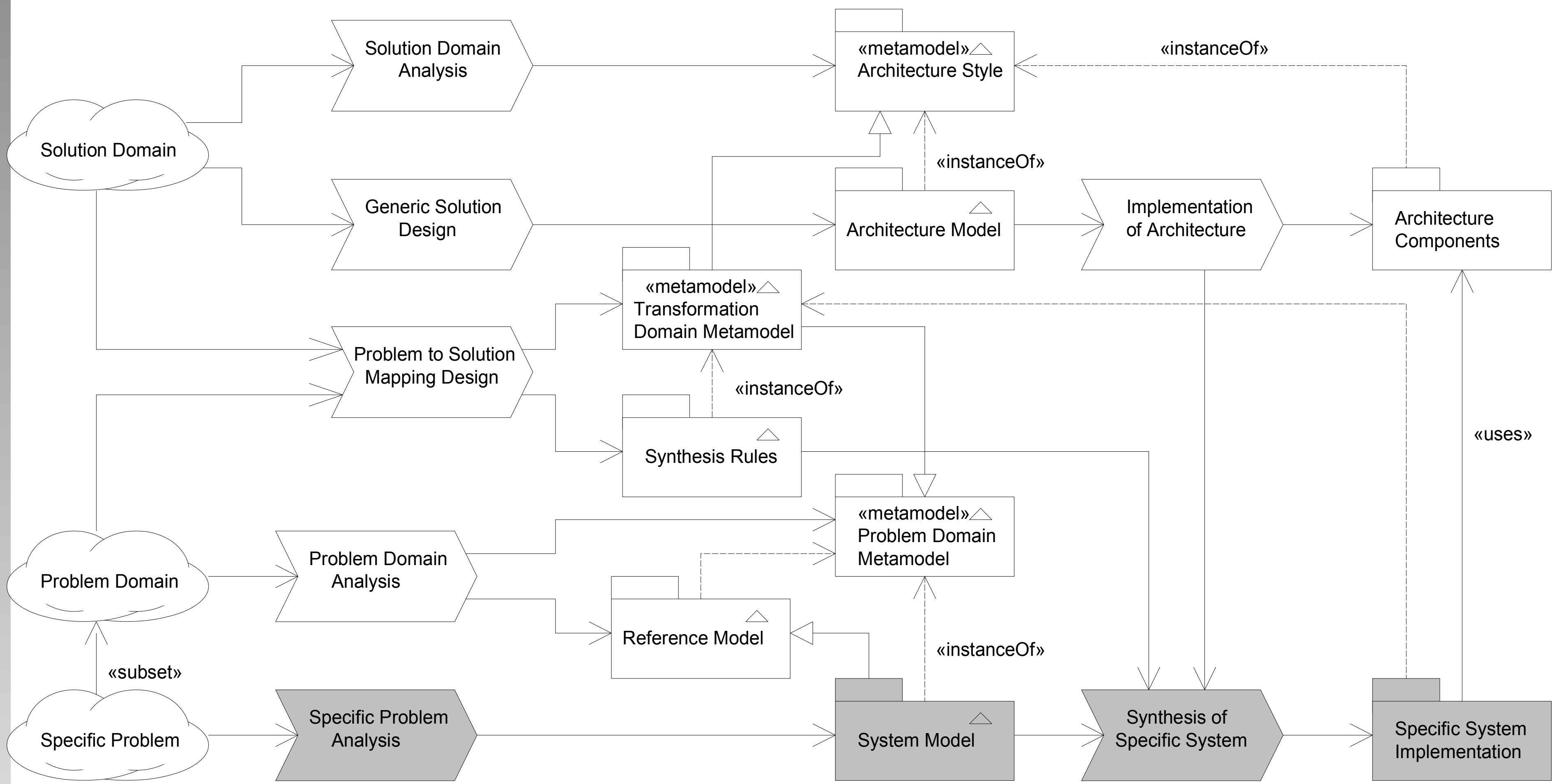
“Gadget Insurance” Product Model



“Gadget Insurance” Design Model



Steps of Model-Oriented Software Development



MDSD Benefits$_1$

- Reasons for MDSD – when to use
  - domain experts can formally specify their knowledge
  - need to provide different implementations of the same model
  - need to capture knowledge about the domains and their mapping
  - separate functionality from implementation details
  - same model is source for several targets (consistency)
  - domain specific product-lines and software system families
- Benefits MDSD – why to use
  - models directly represent domain knowledge – are free from implementation artifacts (separation of concerns)
  - generation for various platforms is possible
  - experts of different domains don't interfere
  - domain experts are directly involved in development
  - due to automation development is more efficient
  - enforcement of architectural constraints/rules/patterns
  - cross-cutting concerns are easily addressed by generators

MDSD Benefits$_2$

- Benefits for Quality
  - explicit, well-defined architecture is needed
  - transformations capture expert knowledge
  - architecture defines strict programming model for manually developed parts
  - generator doesn't produce accidental/random errors
  - documentation is always up-to-date
- You are forced to
  - do domain/application scoping
  - do variability management
  - create well-defined architecture
  - understand domain and target architecture

MDSD Costs

- You need additional skills
  - domain analysis
  - meta-modelling
  - generator development
  - architecture
- Development process is more complex
  - domain architecture development
  - application development