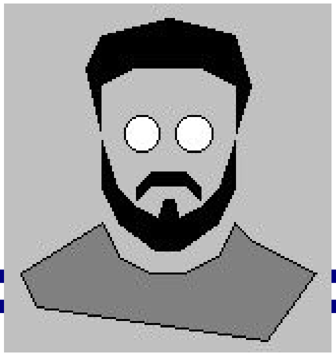
Model-Driven Development

Model-Driven Methods in Software Engineering

Alar Raabe

Content

Alar Raabe



- Over 30 years in IT
  - held various roles from programmer to a software architect
- 15 years in insurance and last 5 years in banking domain
  - developed model-driven technology for insurance applications product-line
    - models
    - method/process
    - tools and platform framework
  - 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

- 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 Development Methods
        - 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

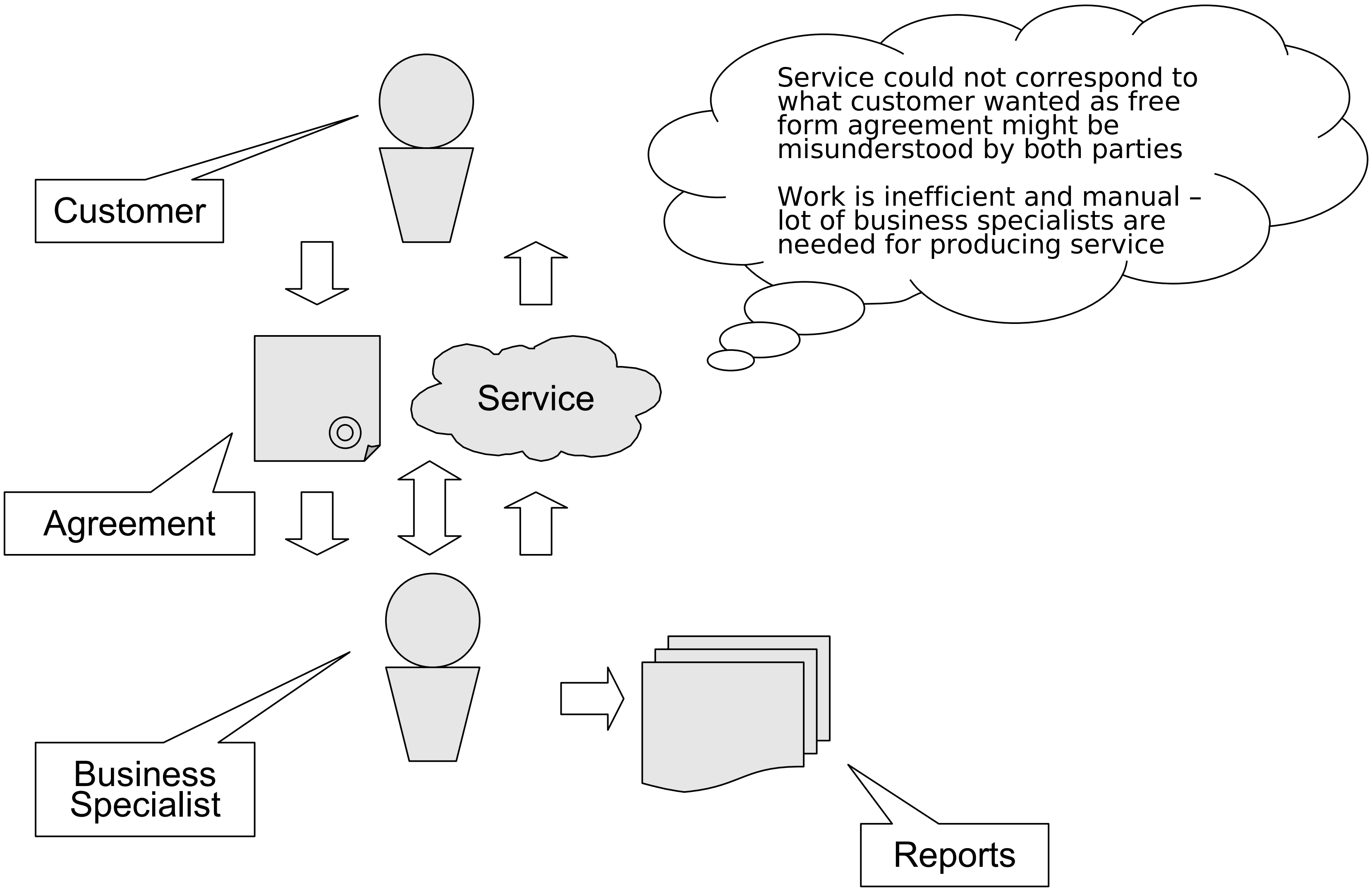
A set of structured information
NOT JUST A PICTURE !

- **Metamodel**
  - – **a model of models** (or a language for models)
  - – a logical information model that specifies the modeling elements used within another (or the same) modeling notation
  - – model defining the concepts and their relations for some modeling notation

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

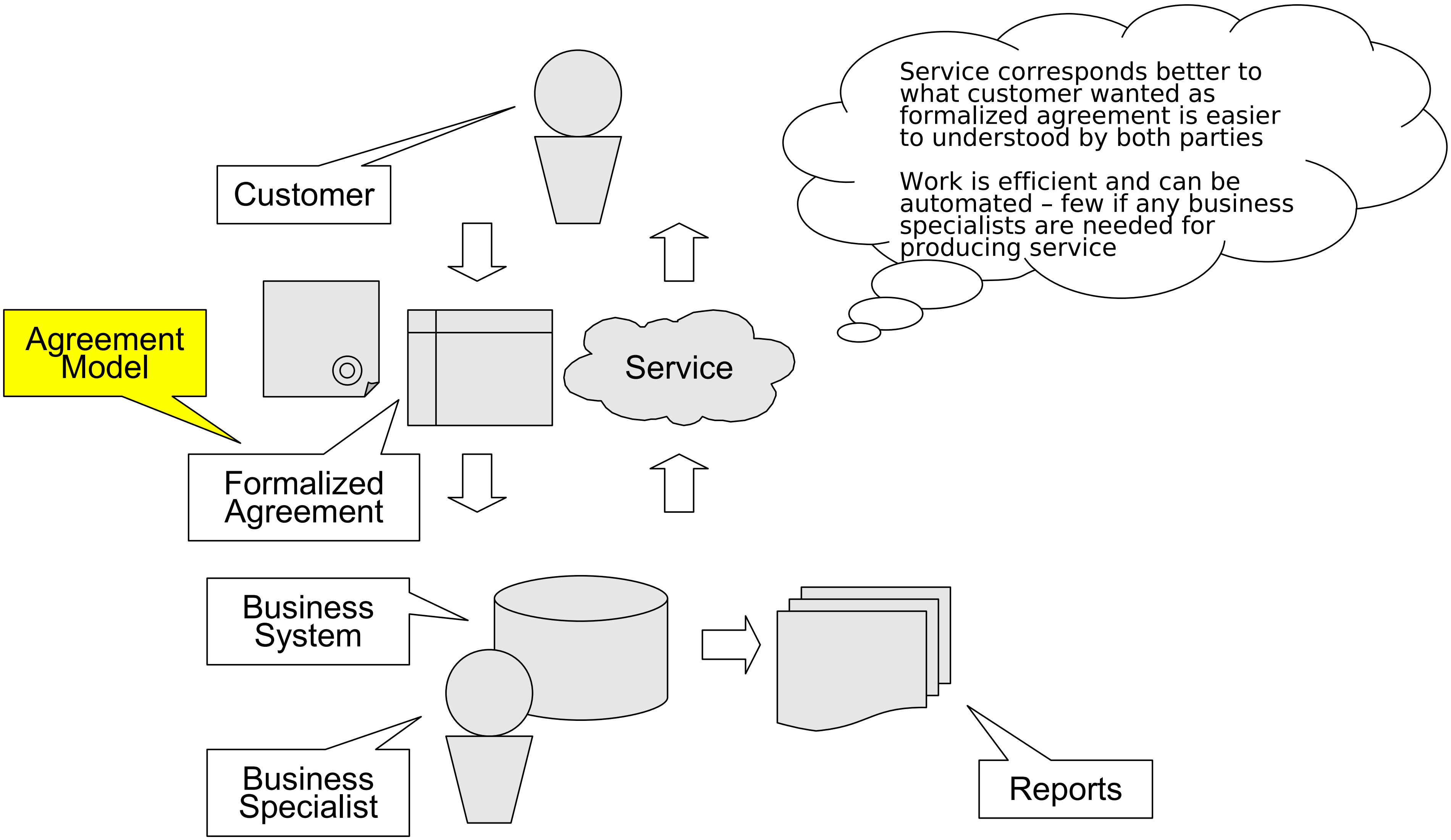
How we did Business Yesterday



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

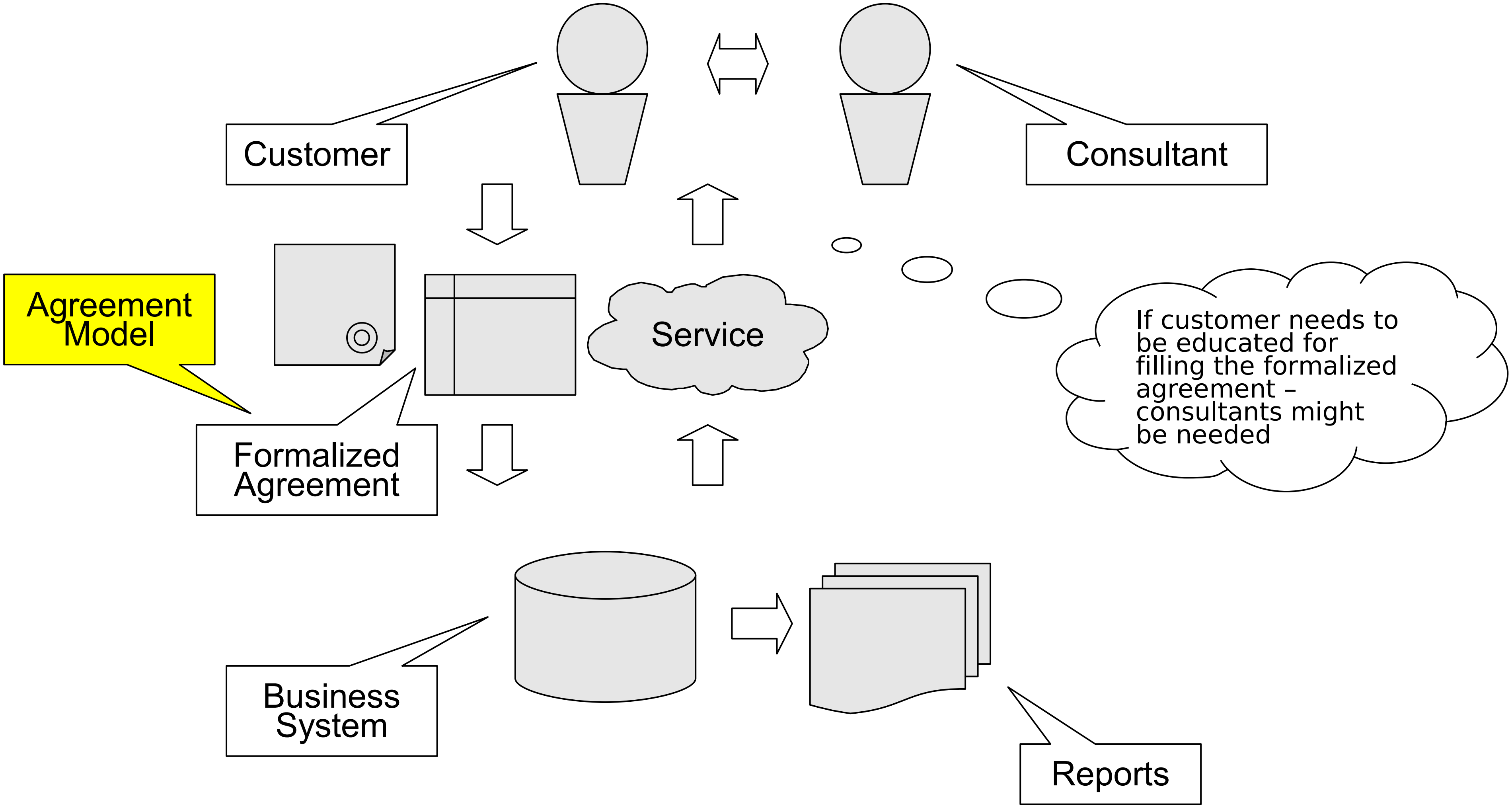
How we do Business Today



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

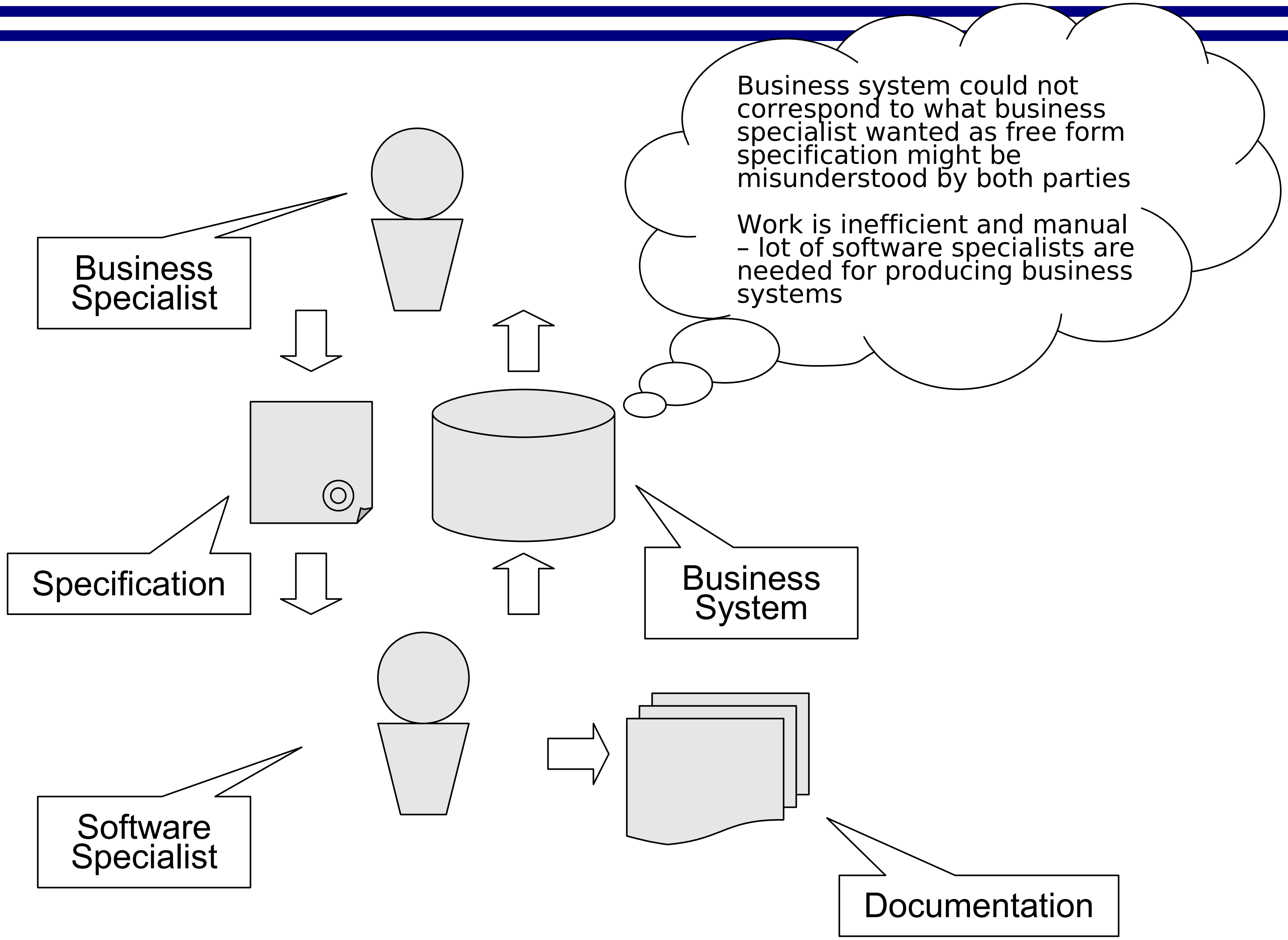
Work is efficient and can be automated – few if any business specialists are needed for producing service

How we do Business Tomorrow



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

How we Develop Software Today



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

Business Specialist

Specification

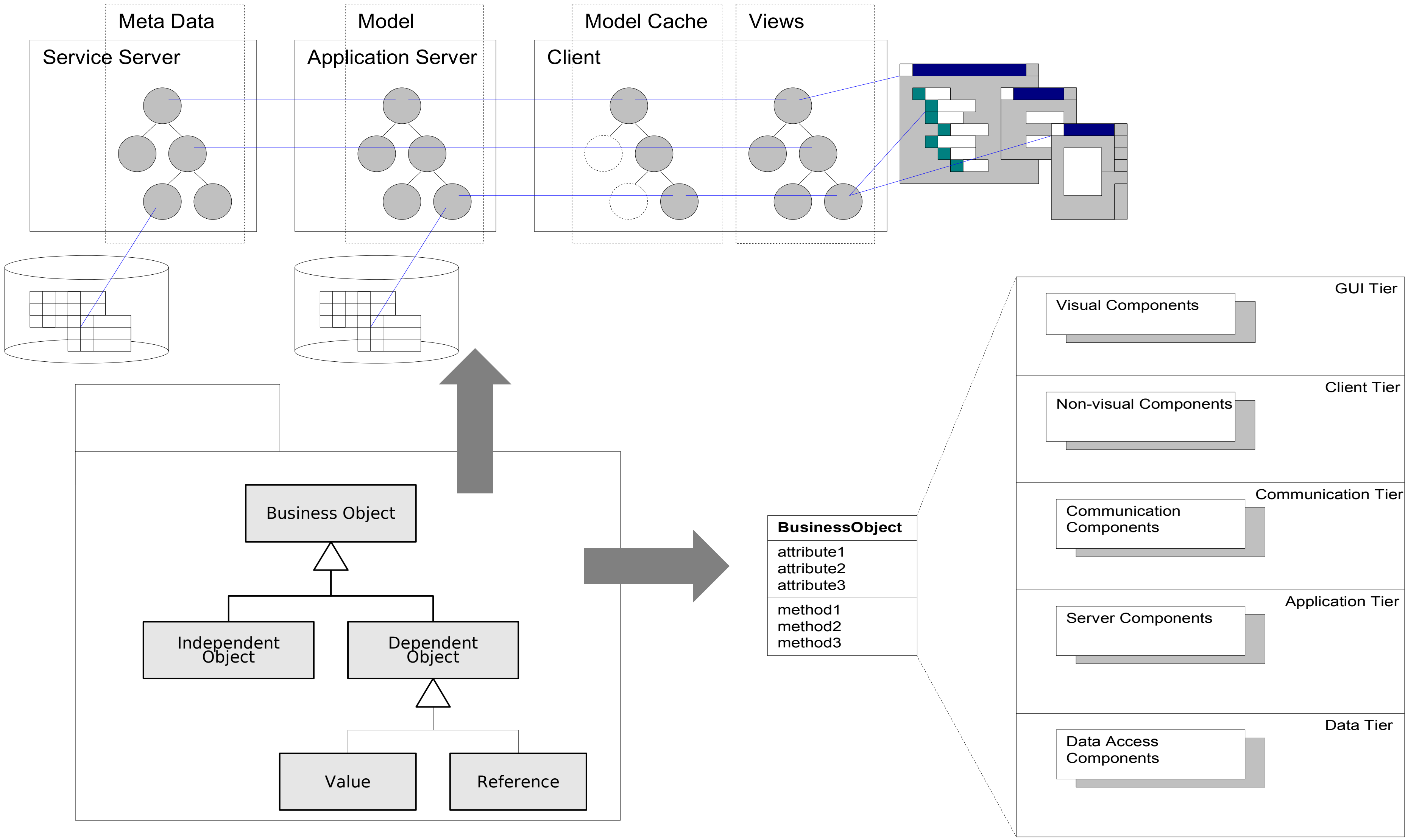
Business System

Software Specialist

Documentation

PROBLEM

Consistency of Implementation



Meta Data

Model

Model Cache

Views

Service Server

Application Server

Client

Business Object

Independent Object

Dependent Object

Value

Reference

BusinessObject

attribute1
attribute2
attribute3

method1
method2
method3

GUI Tier

Visual Components

Client Tier

Non-visual Components

Communication Tier

Communication Components

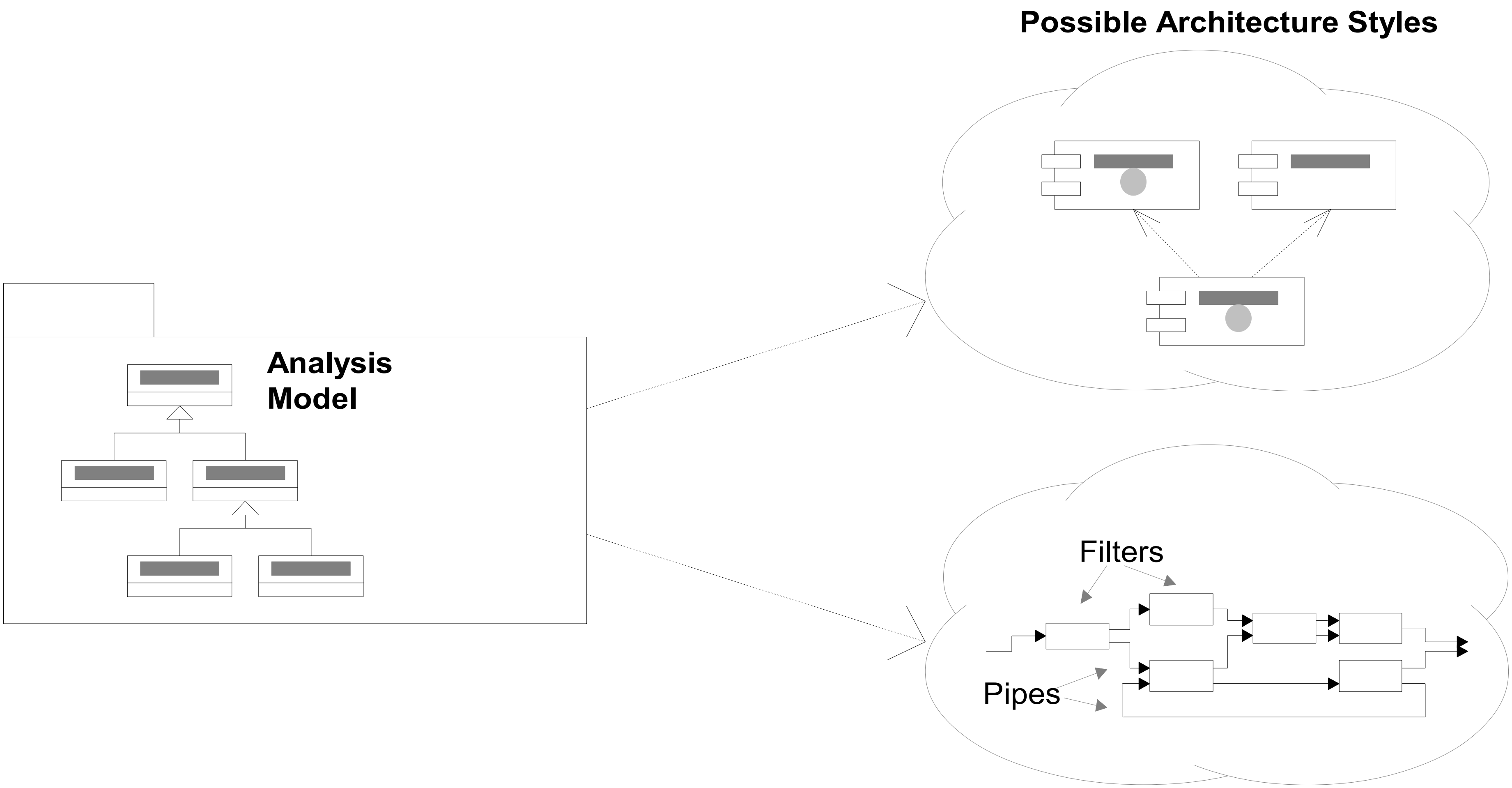
Application Tier

Server Components

Data Tier

Data Access Components

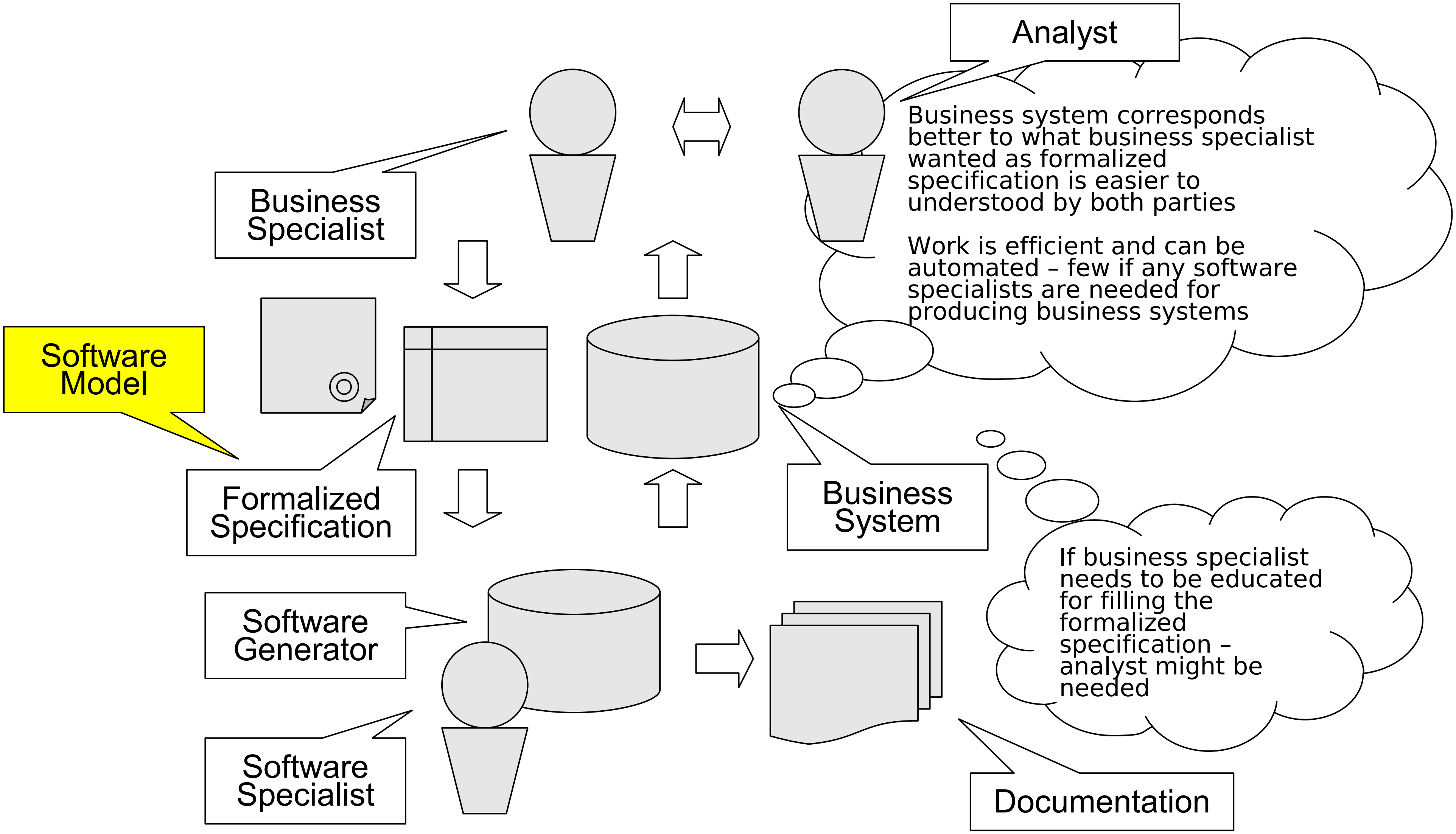
PROBLEM
Mapping to Different Implementations



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



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)
  - 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)

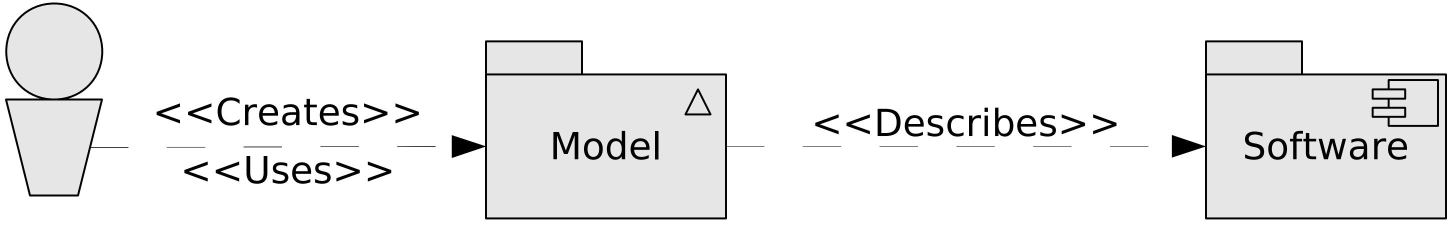
Content

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

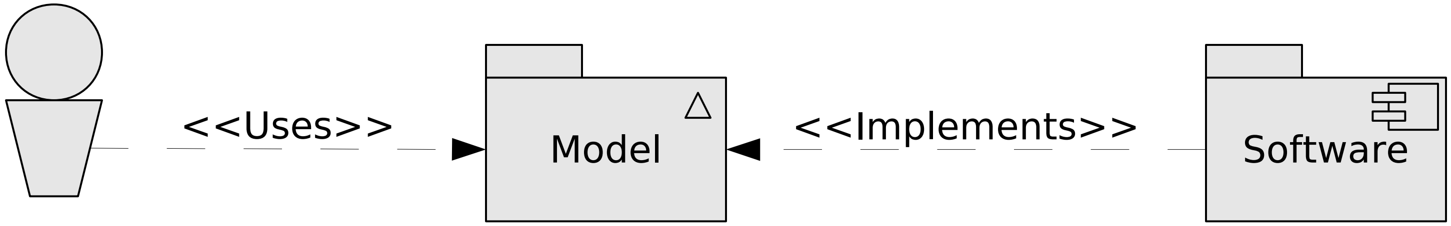
Using Models in Software Development

Most usual – we will not deal with this

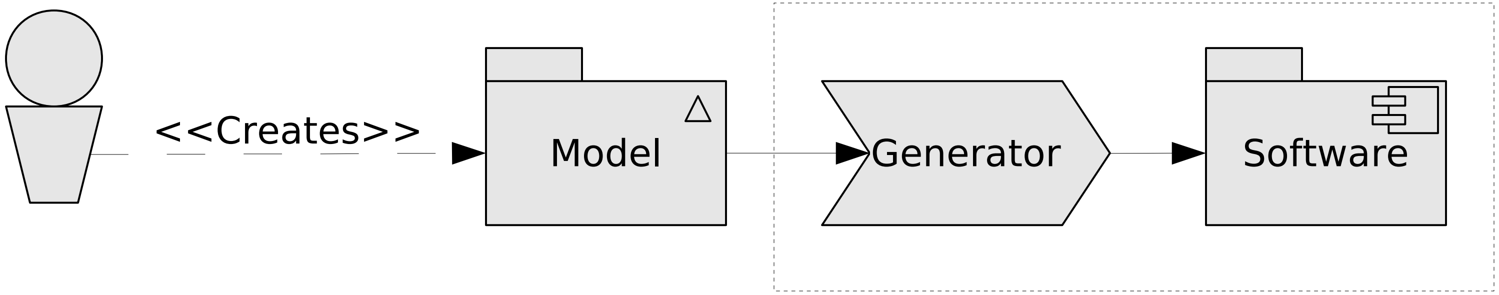
- Models as Descriptions and Illustrations (Documentation)



- Software as Model – Direct Modeling (of Domain)



- Models as Primary Artifacts (Executable Models)



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
  - – designing by building a domain model
- Examples
  - – Modeling programs – programs that directly model something
    - Recursive descent parser implements grammar (model) of language
  - – Meta-programs (or generating programs) – programs, which being models, generate other programs

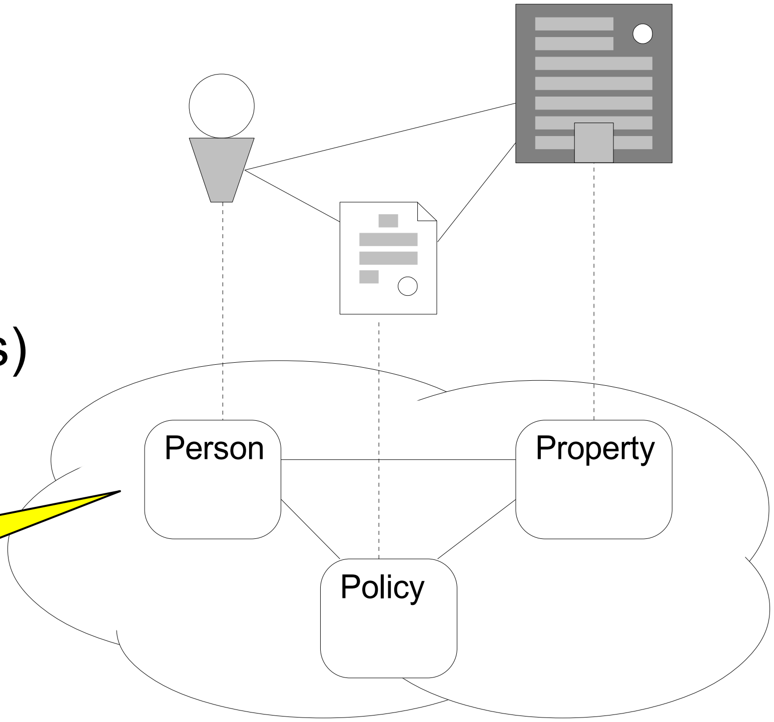
Convergent Engineering

- 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,

    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

- 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 (executable specification)*
      - *intention revealing interfaces (fluent interfaces)*
      - *side-effect-free functions & closure of operations (for value objects)*
      - *assertions (explicit constraints – contracts & invariants)*
    - *Conceptual contours (modules)*
      - *bounded context (explicit context – incl. description of boundary)*
      - *context map (connecting models)*
      - *shared kernel (common subset of models)*
      - *anticorruption layer (interface between models)*
    - *Distillation (separation of essential)*
      - *core domain + generic sub-domains*
      - *knowledge level (meta-level separation)*

Content

History of Model-Driven Software Development (MDSD)

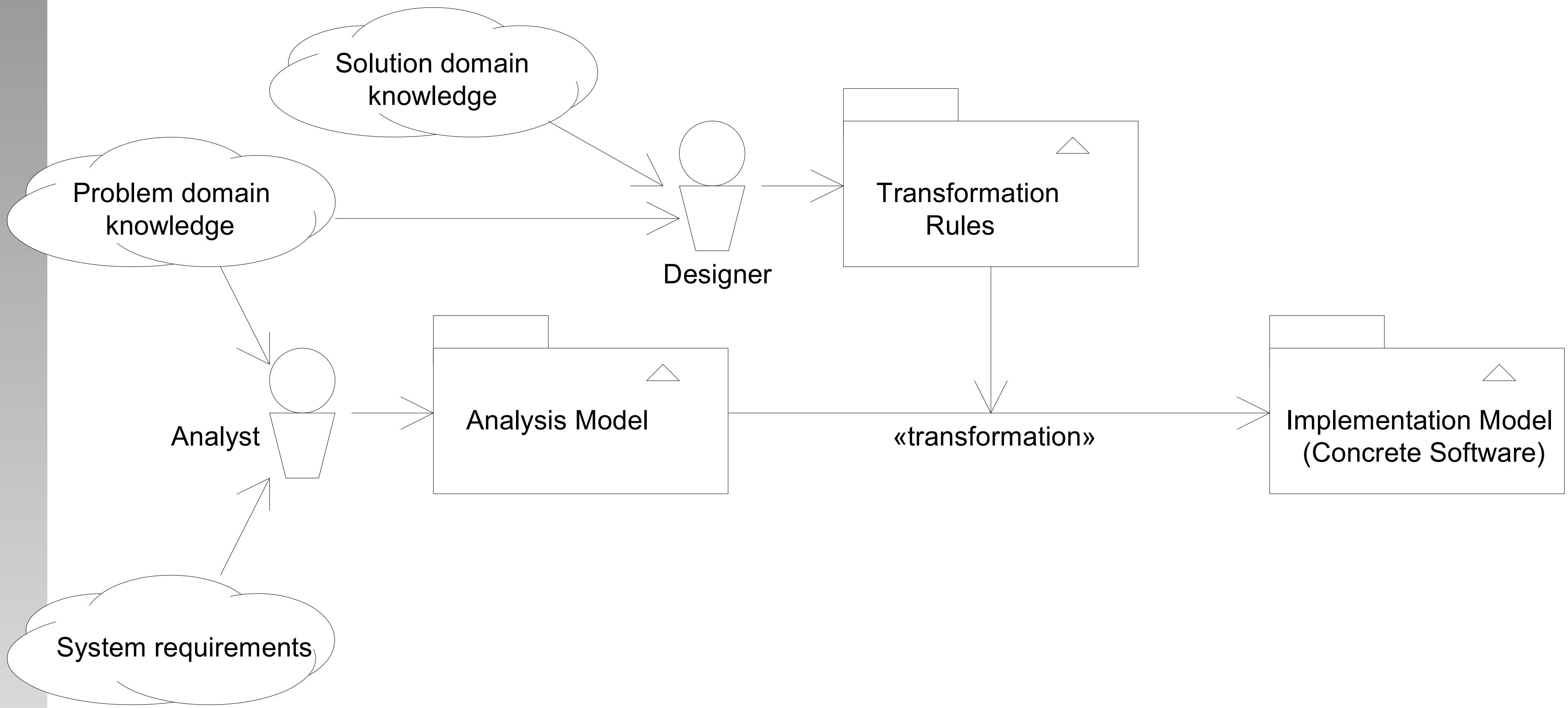
- Real-time and embedded systems
  - Model-Integrated Computing (MIC) and model-based software synthesis – (Vanderbilt Univ. (ISIS), 1993; Abbott et al., 1994)
  - Model-based development – (Mellor, 1995)
- Generative programming
  - GenVoca – (Batory, 1992)
  - Family-Oriented Abstraction, Specification, 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)
  - fUML & Alf – (OMG, 2011 ...)

Four Components of MDSD

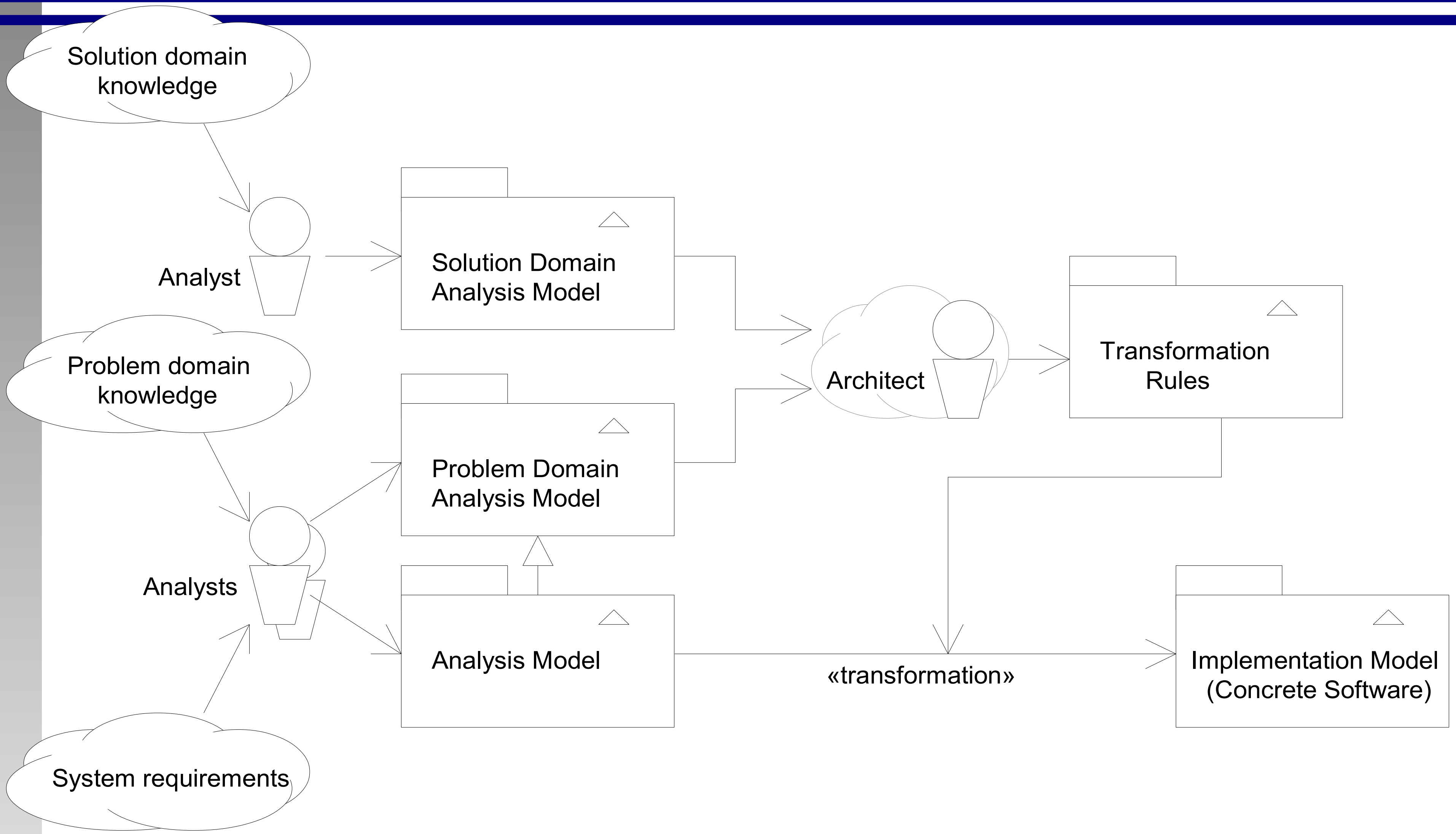
- Models
  - Analysis and design meta-models
  - Domain (reference) models
- Architecture
  - Architecture style(s)
  - Domain (reference) architecture (in selected architecture style(s))
- Process
  - Generation/transformation rules
  - Process of application of generation rules
- Tools
  - Model manipulation tools
  - Generators

Traditional MDSD Approach

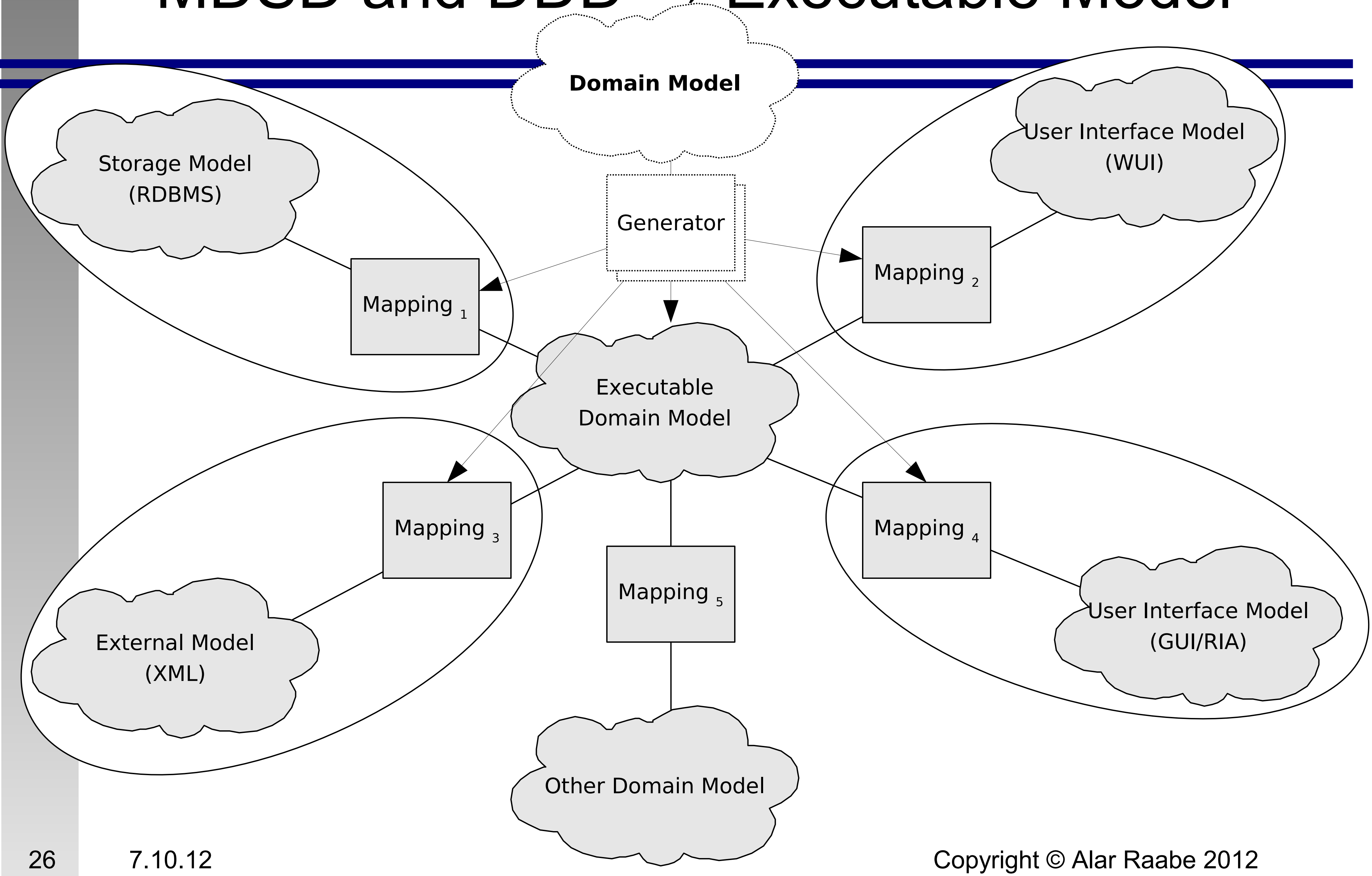
Solution knowledge is not separated from technical knowledge !



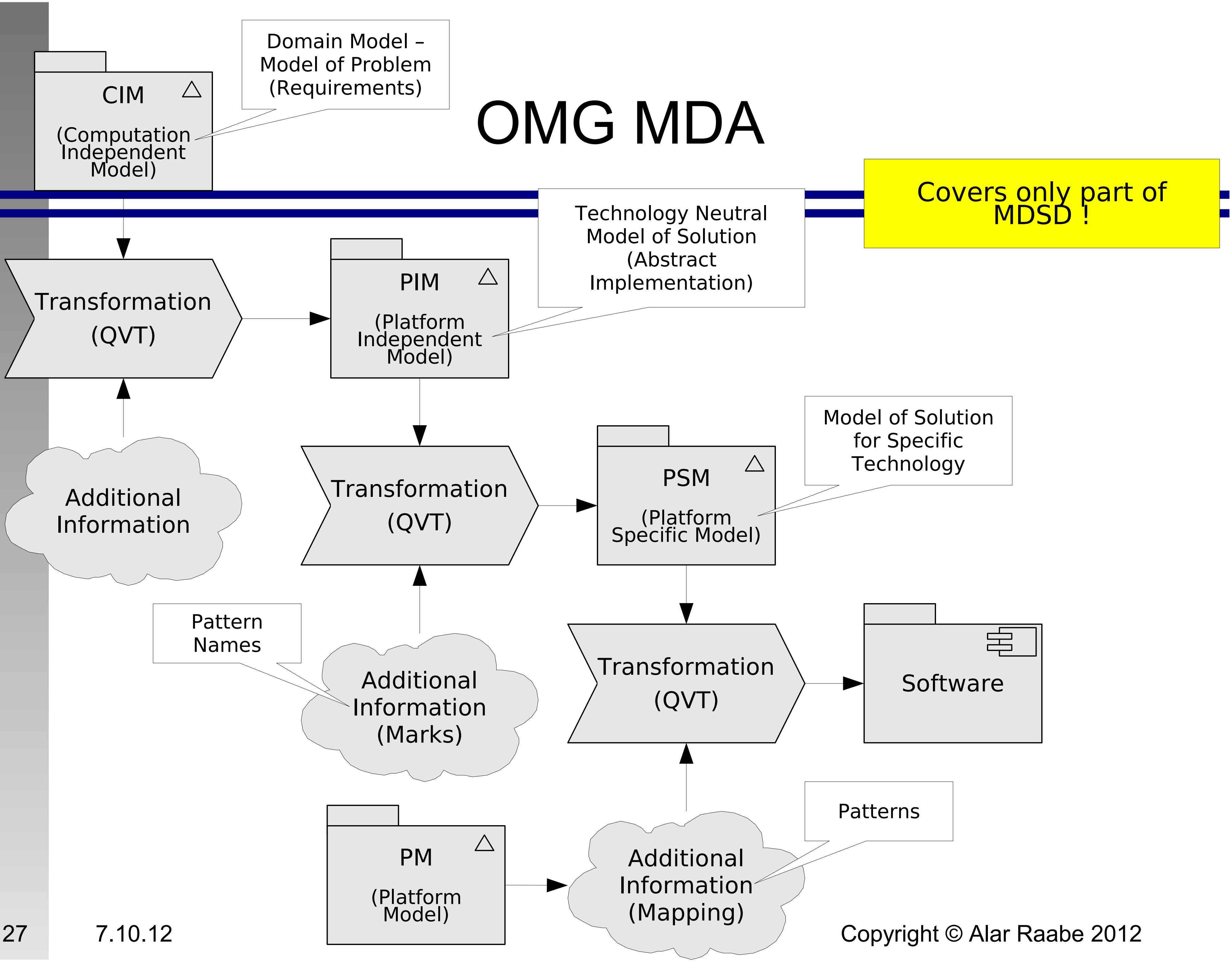
Extended MDSD Approach



MDSD and DDD → Executable Model



OMG MDA



CIM

(Computation Independent Model)

Domain Model – Model of Problem (Requirements)

Covers only part of MDSD !

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

OMG Executable Models

- Shlaer-Mellor method (models with precise semantics)

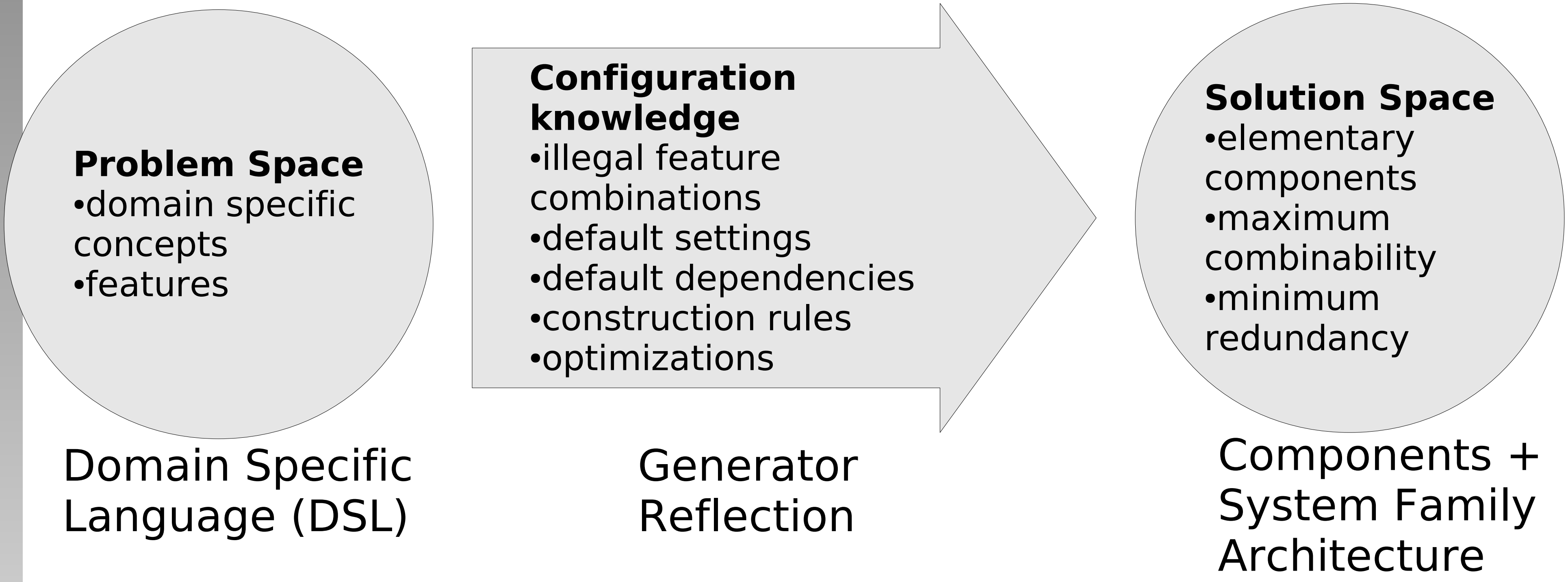
- OMG Standards for Executable UML
  - fUML (Foundational Subset for Executable UML Models)
    - For describing, in an operational style, structural and behavioral semantics of the system

  - Alf (Action Language for fUML)
    - For describing textually fine-grained behavior of the system – concrete syntax corresponding to fUML abtract syntax

- Domain model consists of
  - *Domain chart* – provides a view of the domain being modeled, and the dependencies it has on other domains
  - *Class diagram* – defines the classes and class associations for the domain
  - *Statechart diagram* – defines the states, events, and state transitions for a class or class instance
  - *Action language* defines the actions or operations that perform processing on model elements

Generative Programming

[Czarnecki, Eisenecker]



Problem Space
- domain specific concepts
- features

Domain Specific Language (DSL)

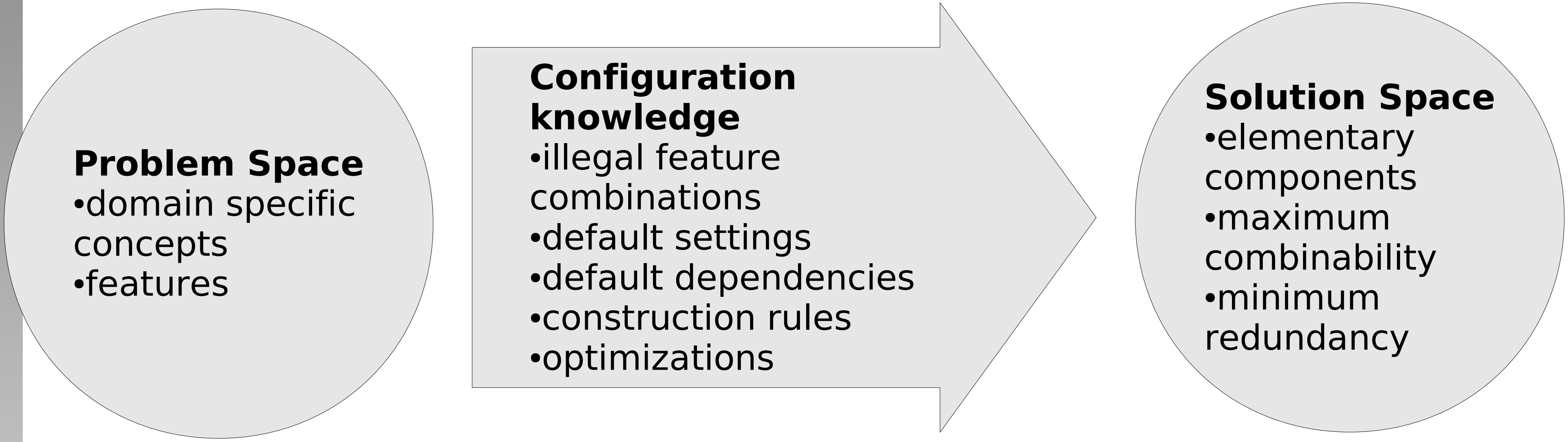
Configuration knowledge
- illegal feature combinations
- default settings
- default dependencies
- construction rules
- optimizations

Generator Reflection

Solution Space
- elementary components
- maximum combinability
- minimum redundancy

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, ...)

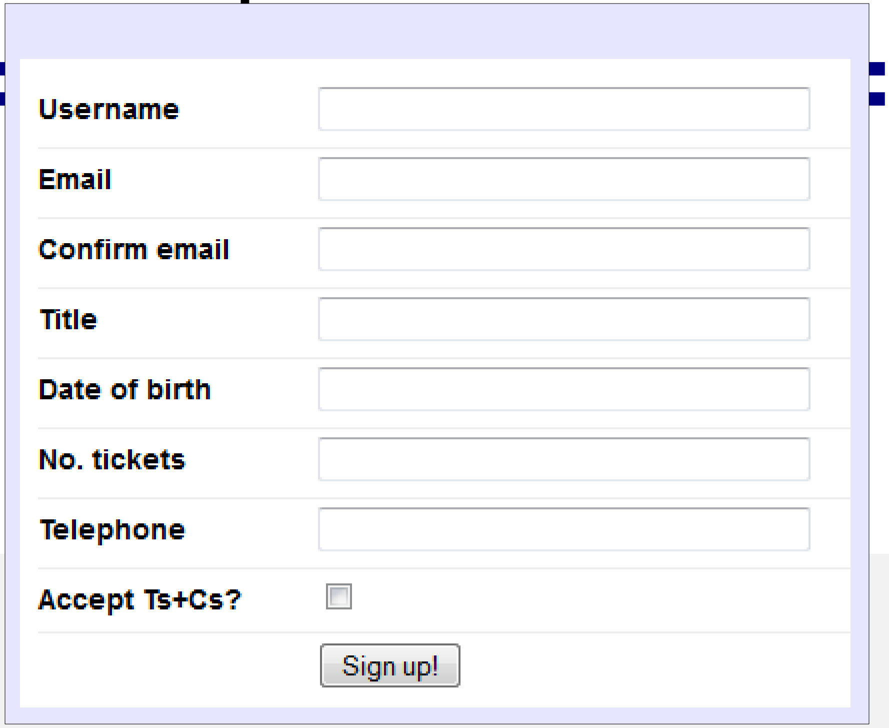
DSL Technologies

WARNING:
Don't be too Clever !

- Internal DSLs
  - Built-in features of programming languages
    - C++ templates
    - Lisp Macros
  - Extensible languages
    - Scala, Ruby, Groovy, JavaScript, ...
    - Seed7, XL, … (with extensible syntax)
  - Well-Designed APIs
- External DSLs
  - Textual languages
    - XML, xText, ...
  - Graphical languages
    - UML, MetaCASE, ...
  - Interactive wizards

Internal DSL Example

- 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 Example

Model in EBNF

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

- xText (oAW)

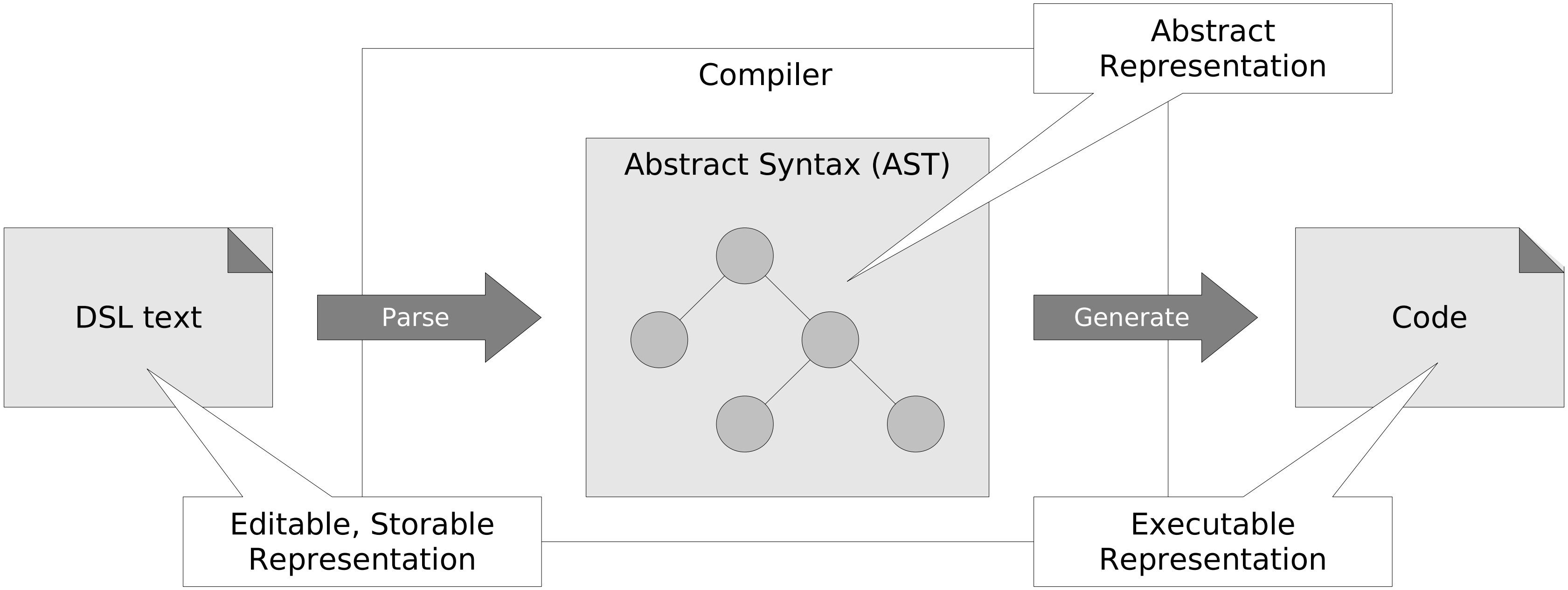
```
Entity :
  "entity" name=ID ("extends" superType=[Entity])?
  "{"
     (features+=Feature)*
  "}";
Feature :
  Attribute | Reference;
Attribute :
  type=ID name=ID ";";
Reference :
  "ref" (containment?"+")? type=ID name=ID ("<->" oppositeName=ID)? ";";
```

- Example

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

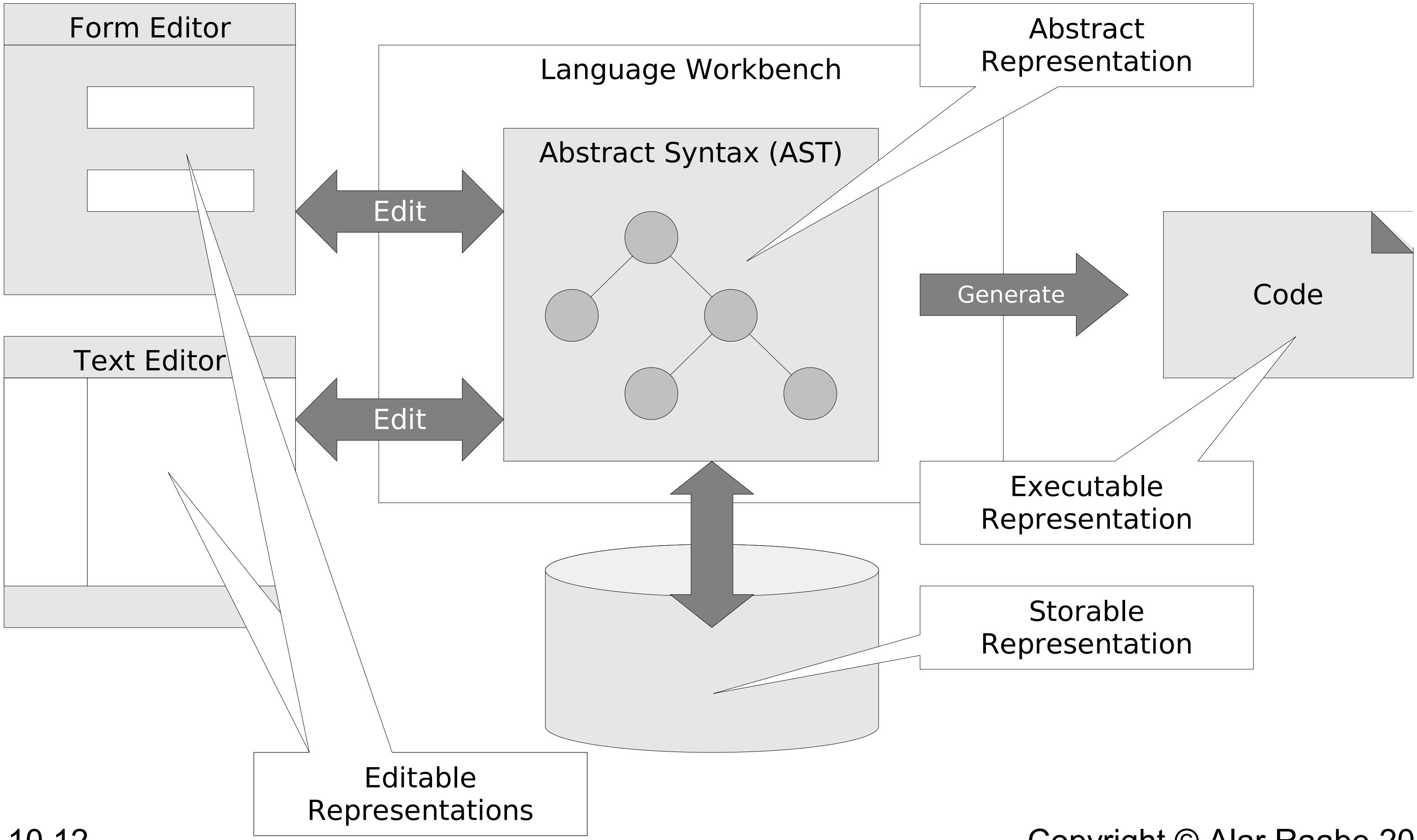
DSL Implementation 1

- Compiler-Based



DSL Implementation $_2$

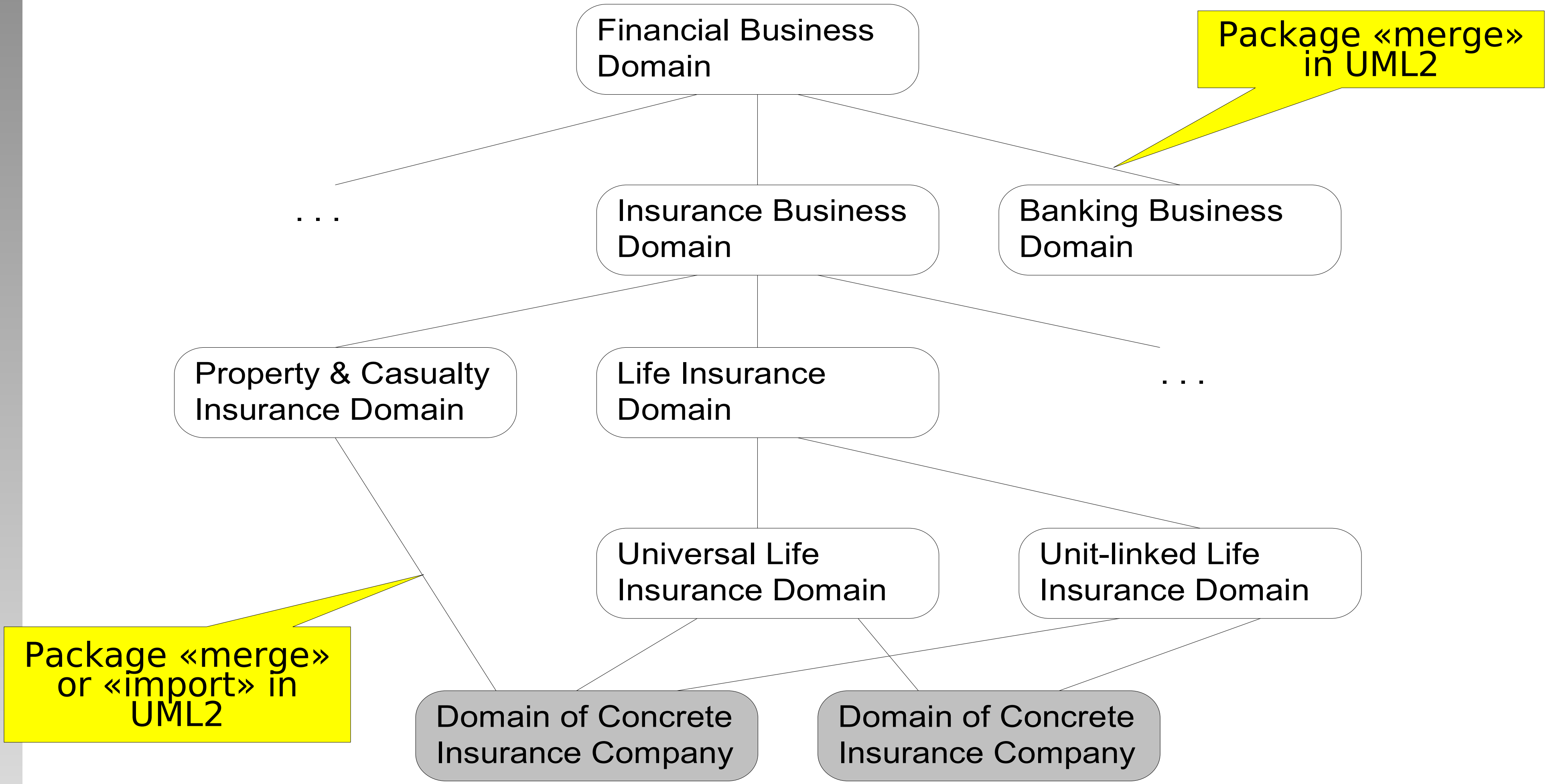
- Language Workbench



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

Network of Problem Domains



Financial Business Domain

. . .

Insurance Business Domain

Banking Business Domain

Package «merge» in UML2

Property & Casualty Insurance Domain

Life Insurance Domain

. . .

Universal Life Insurance Domain

Unit-linked Life Insurance Domain

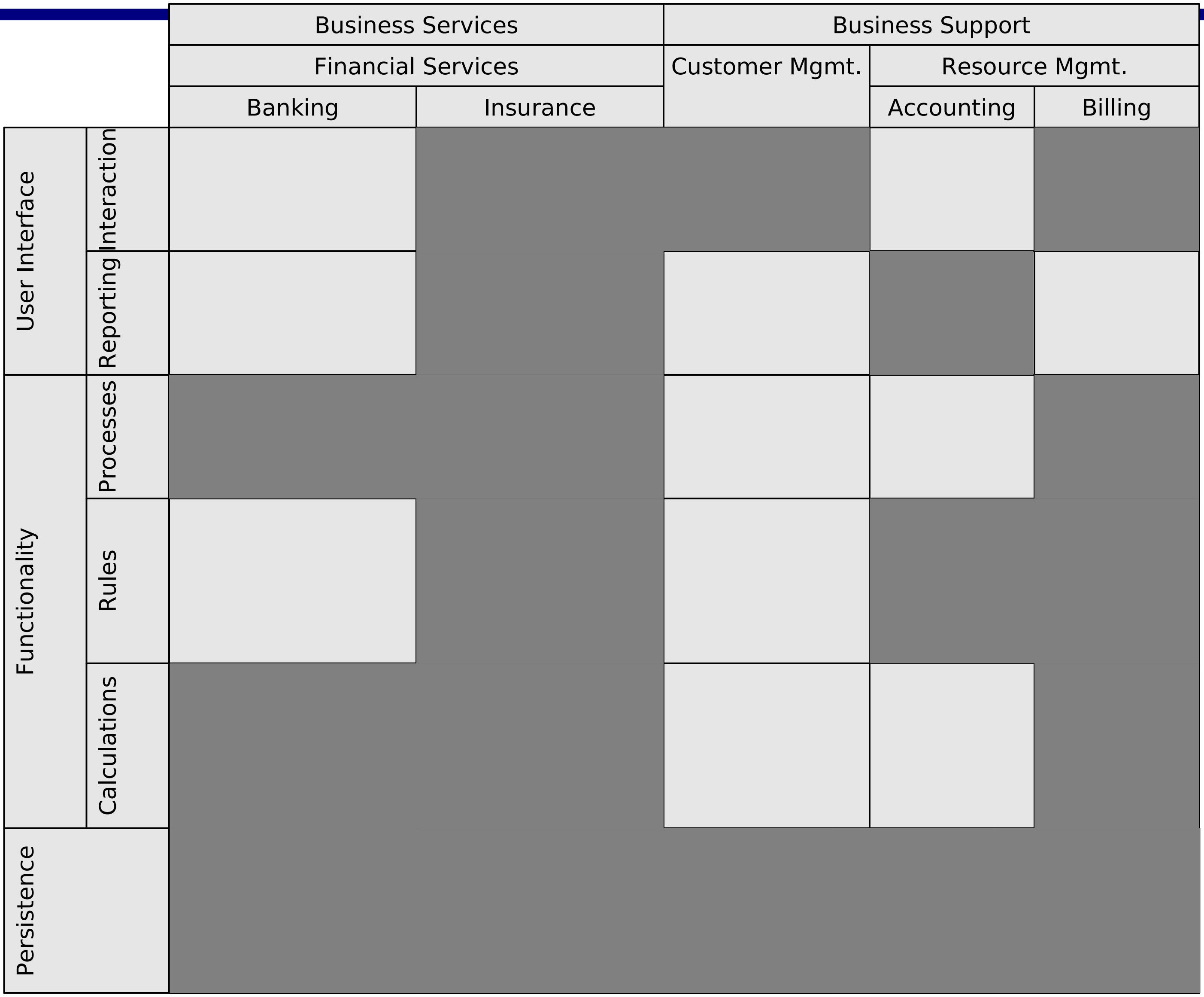
Package «merge» or «import» in UML2

Domain of Concrete Insurance Company

Domain of Concrete Insurance Company

PROBLEM

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



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

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

MDSD Best Practices 1

[Voelter, ...]

- Separate the Generated and Manually Created Code
  - protected regions (generated code must be revision controlled)
  - separate directory (e.g. src-gen)
  - language mechanisms (e.g. sub-classing/inheritance, wrapping/containment, aspects, ...)
- Don't Manage Generated Code in Revision Control System
  - exception when using protected regions or 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
- 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

MDSD Best Practices $_2$

[Voelter, ...]

- Use the Native Techniques of Target Platform for Separating Generated and Manually Created Code
  - object languages – sub-classing/inheritance, wrapping/containment
  - aspect languages – aspects/pointcuts (weaving)
  - procedural languages – preprocessing (e.g. includes), libraries
- 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

MDSD Best Practices 3

[Voelter, ...]

- Develop DSLs Incrementally
  - DSLs should be developed as understanding grows
  - DSLs are public interfaces – should be developed and evolved like APIs
  - provide facilities for migrating old models to new metamodel (e.g. model transformation)
- 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

MDSD Best Practices 4

[Voelter, ...]

- Select Suitable Technology – Avoid too Complex Meta-Models
  - define core abstractions clearly and expandable
  - models should be quickly editable and turnaround (model ⟶ generate ⟶ execute) should be quick
  - avoid overly complex metamodels (like UML) or encapsulate these
  - transform complex metamodels into simpler metamodels targeted 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)

MDSD Best Practices 5

[Voelter, ...]

- Teamwork Prefers 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
- 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

MDSD Best Practices 6

[Voelter, ...]

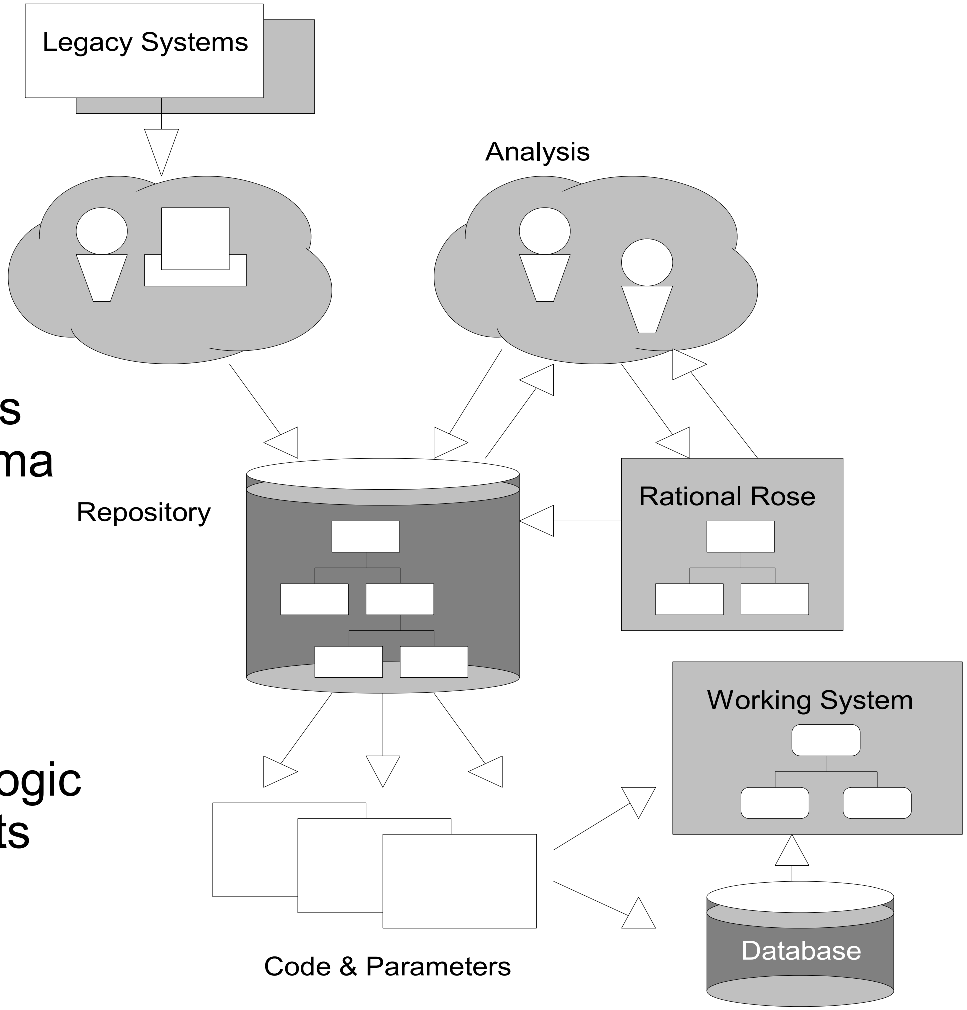
- Don't Reverse Engineer – Model is Primary Artifact
  - all changes should be done in model, and then all derived artifacts should be regenerated
- Regenerate Frequently
  - include generation into continuous build process
  - frequent regeneration ensures compliance with model and architectural constraints (embedded into generator)

Examples of MDSD

- 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

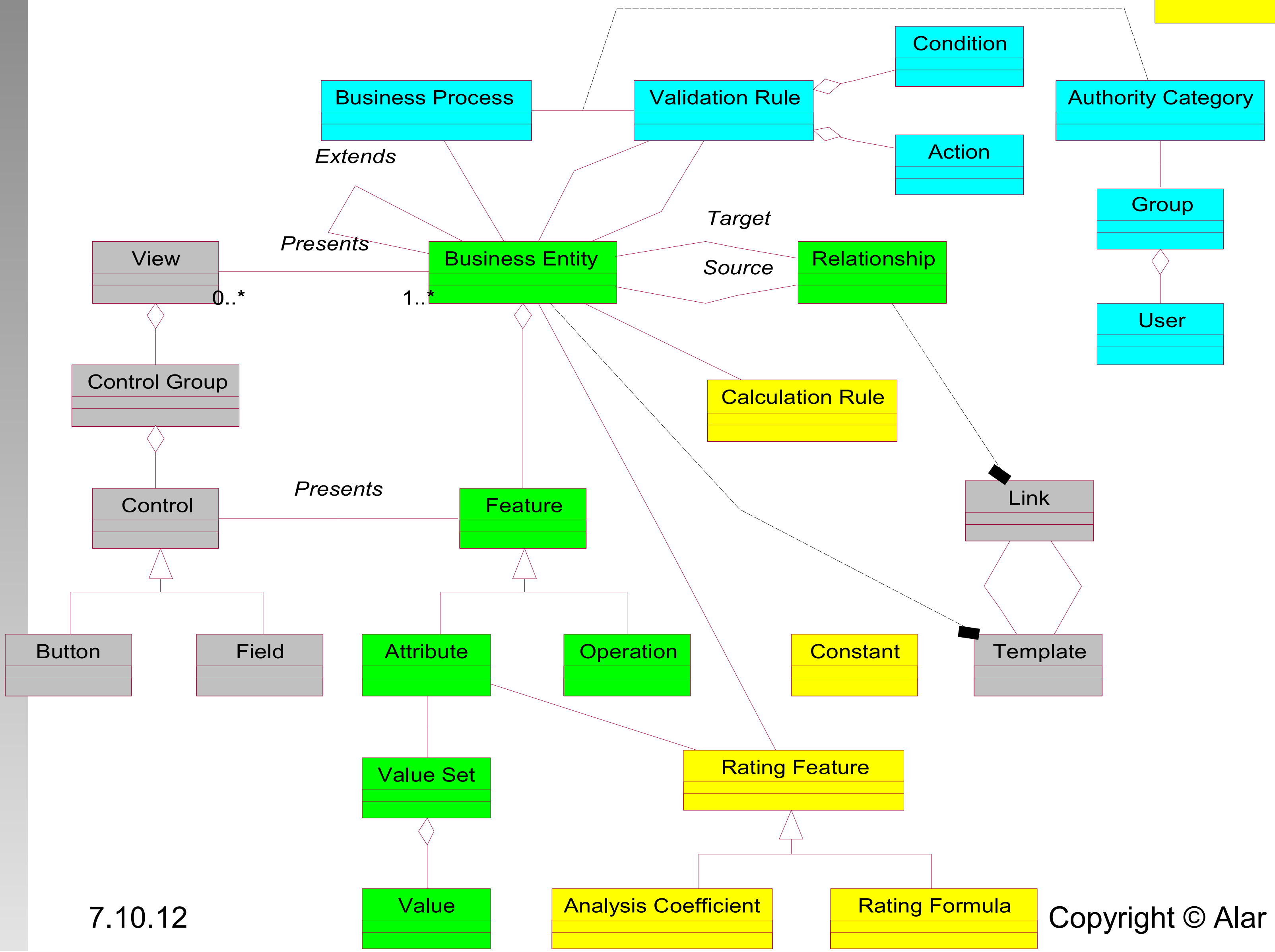
Overview of OD Software Process

- Beginning
- Analysis
  - 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



Extended OOA/OOD Meta-Model

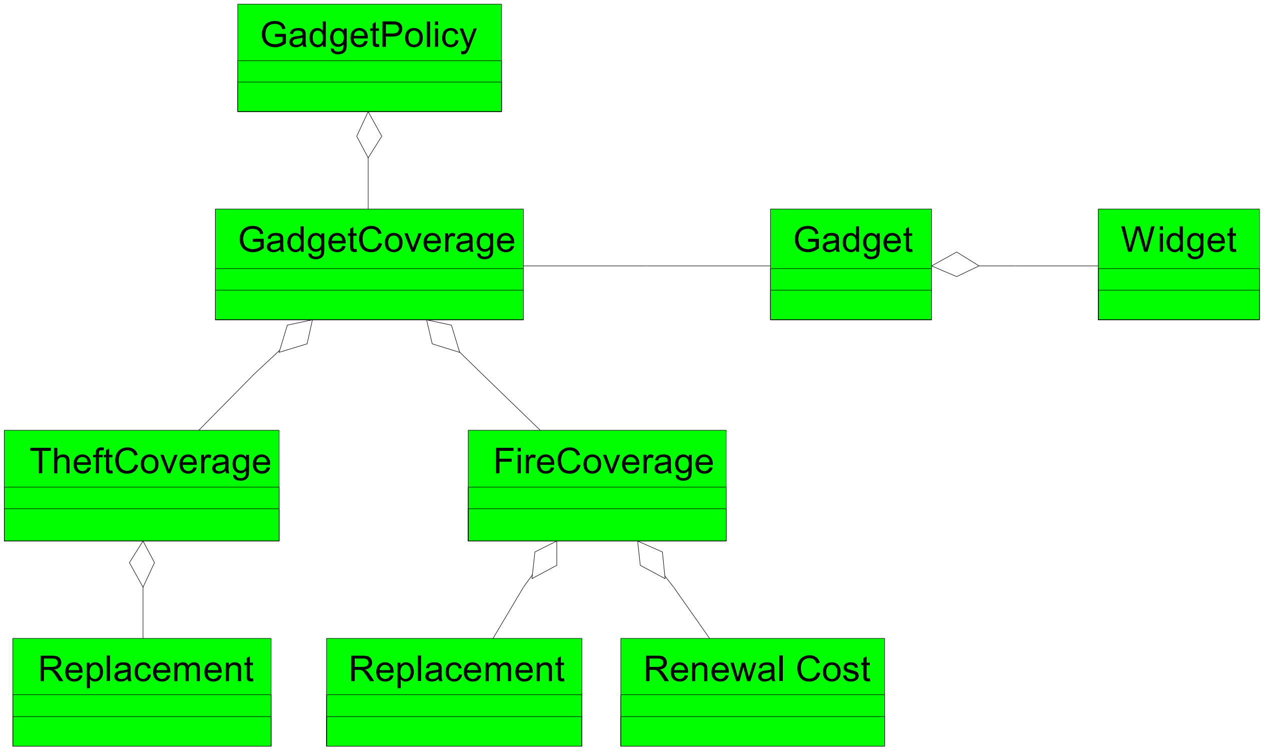
a DSL for Insurance Systems



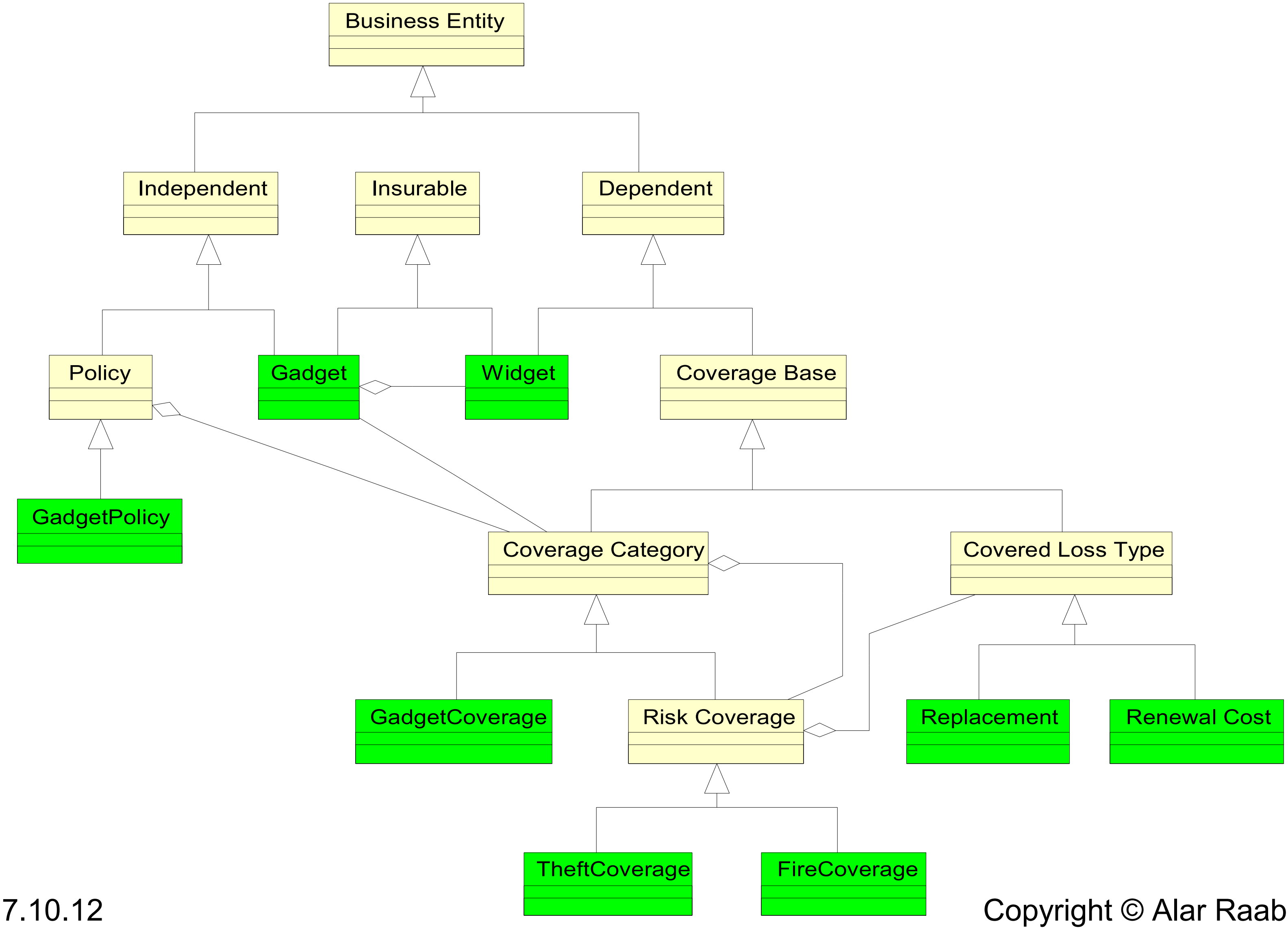
Example of Using Once&Done

- “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

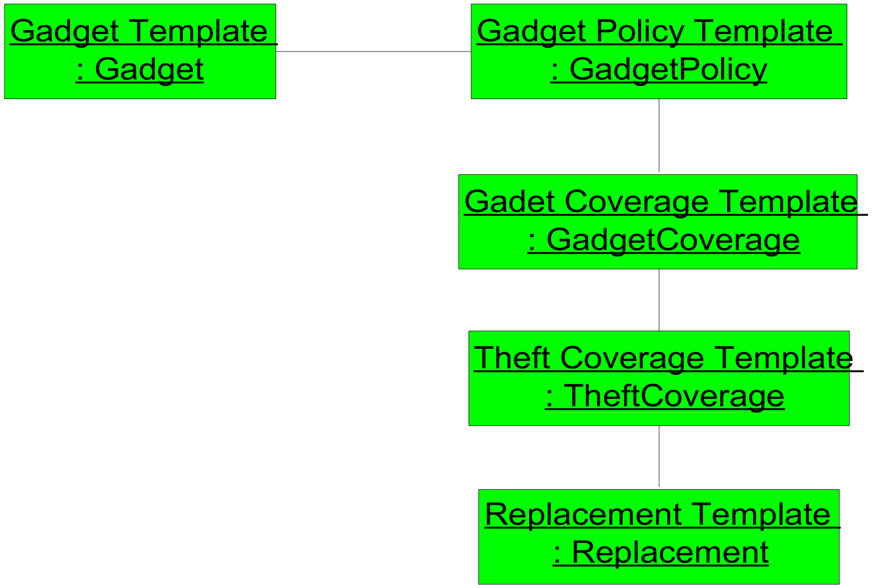
“Gadget Insurance” Analysis Model



“Gadget Insurance” Model as Extension to Insurance Domain Model



“Gadget Insurance” Product Model



Gadget Template : Gadget

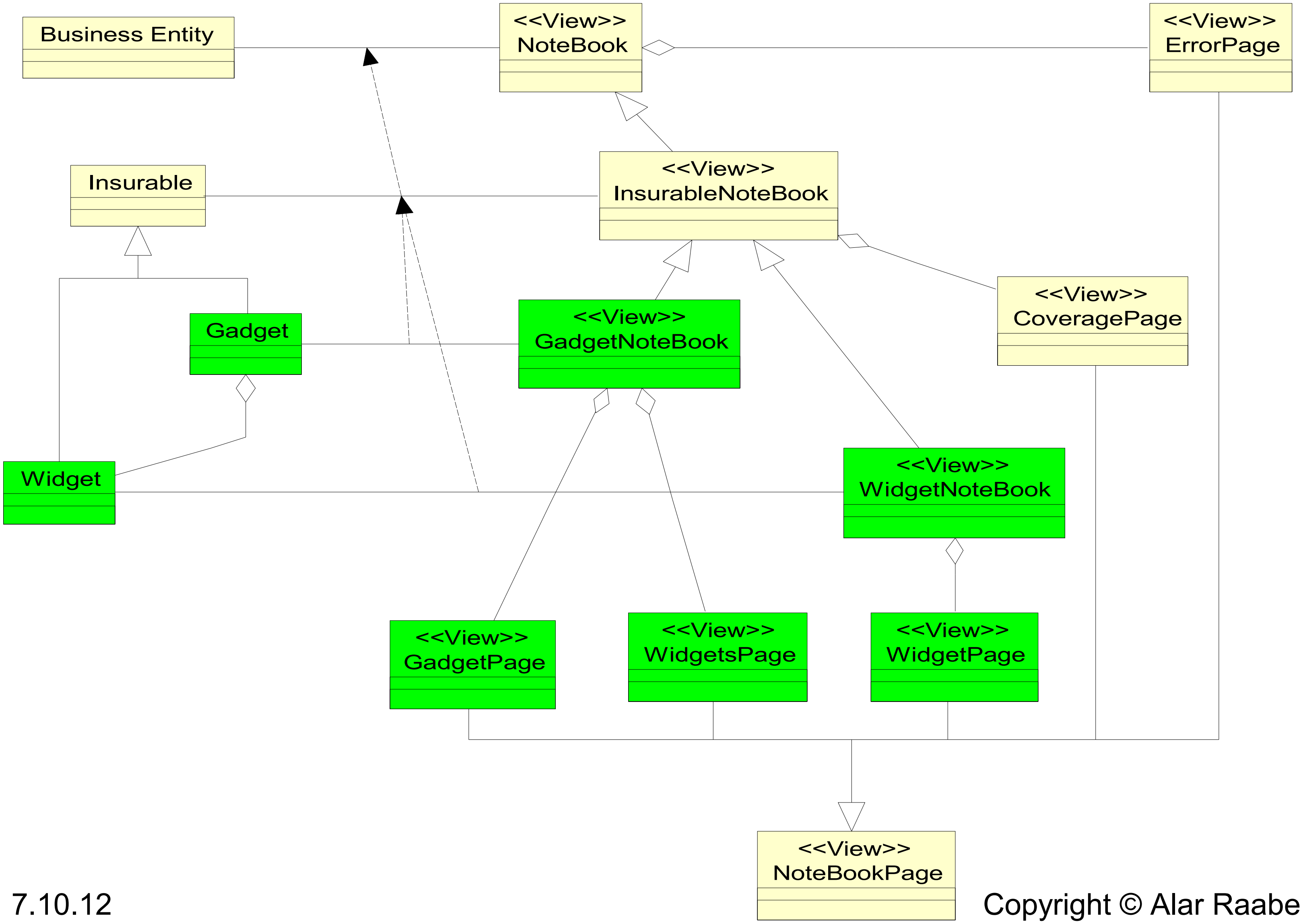
Gadget Policy Template : GadgetPolicy

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 Rich Client, further
    - HTML-based web-application

Comparing Model-Driven Method with Traditional

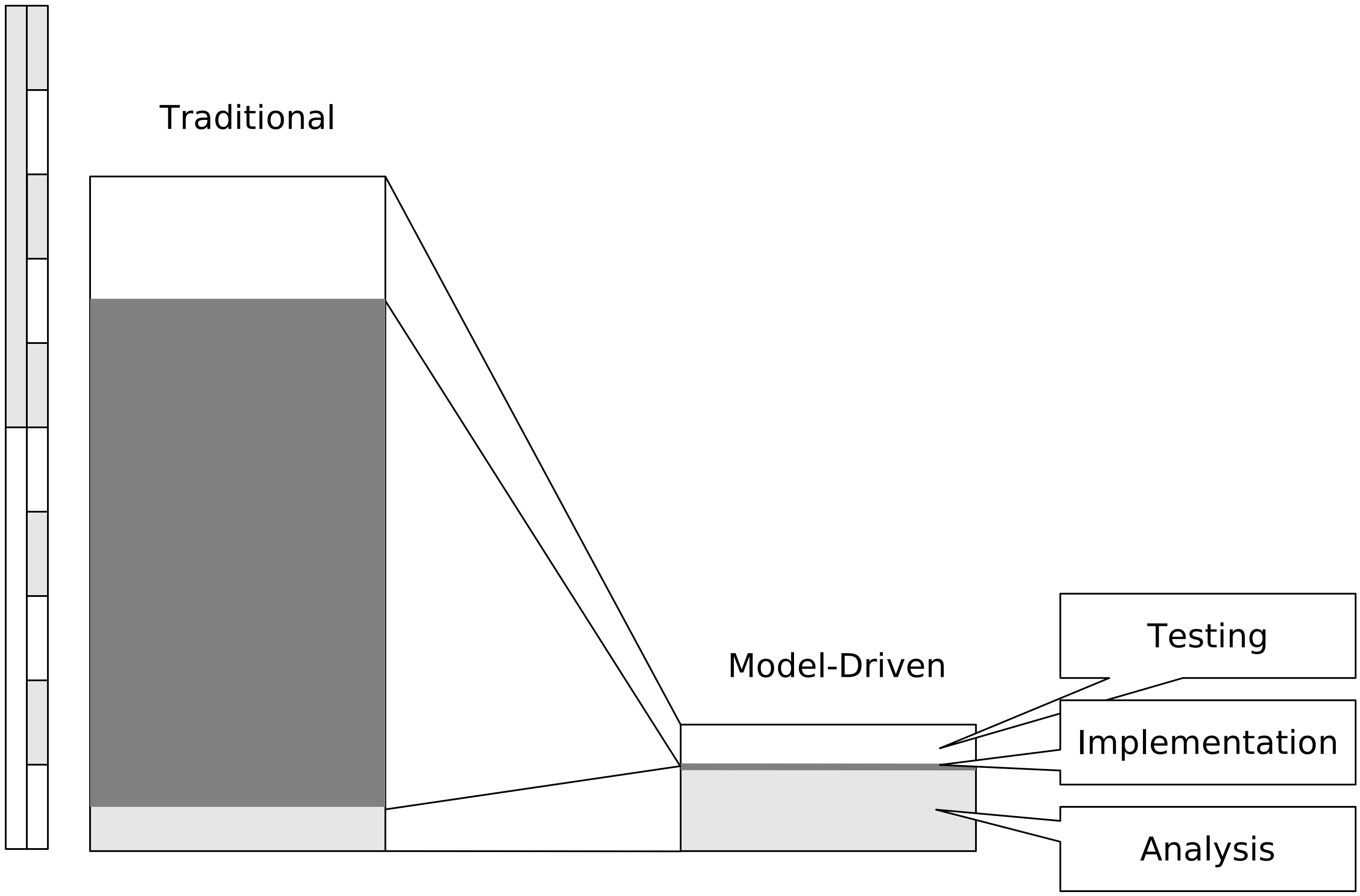
- Effort for First Iteration – Basically CRUD Application

- 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

- Modeling 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

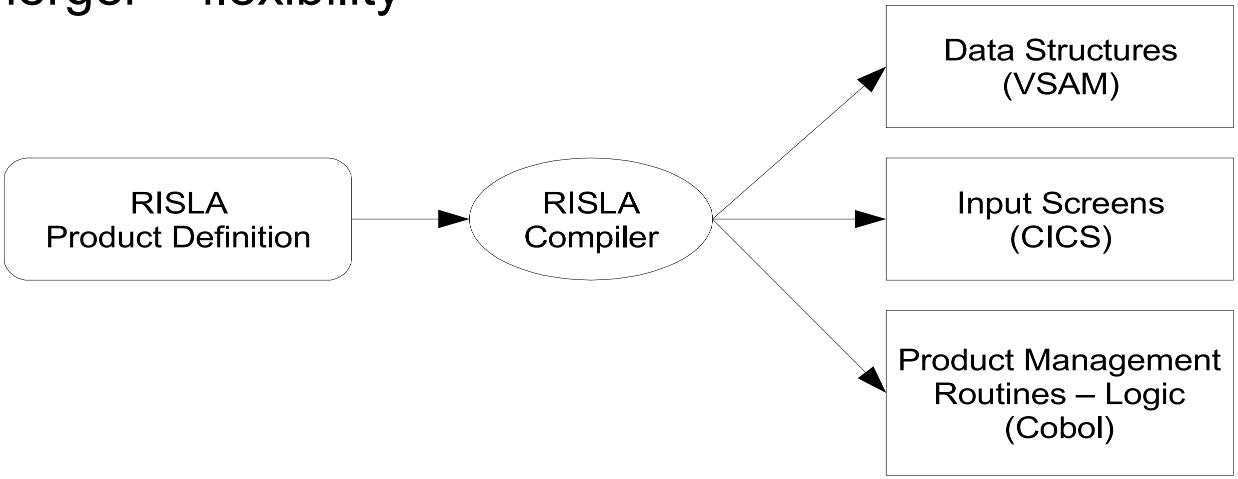
RISLA – Language for Product Models

- 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



MLFi – Language for Financial Instruments and Contracts

- Financial Instrument (American Option)

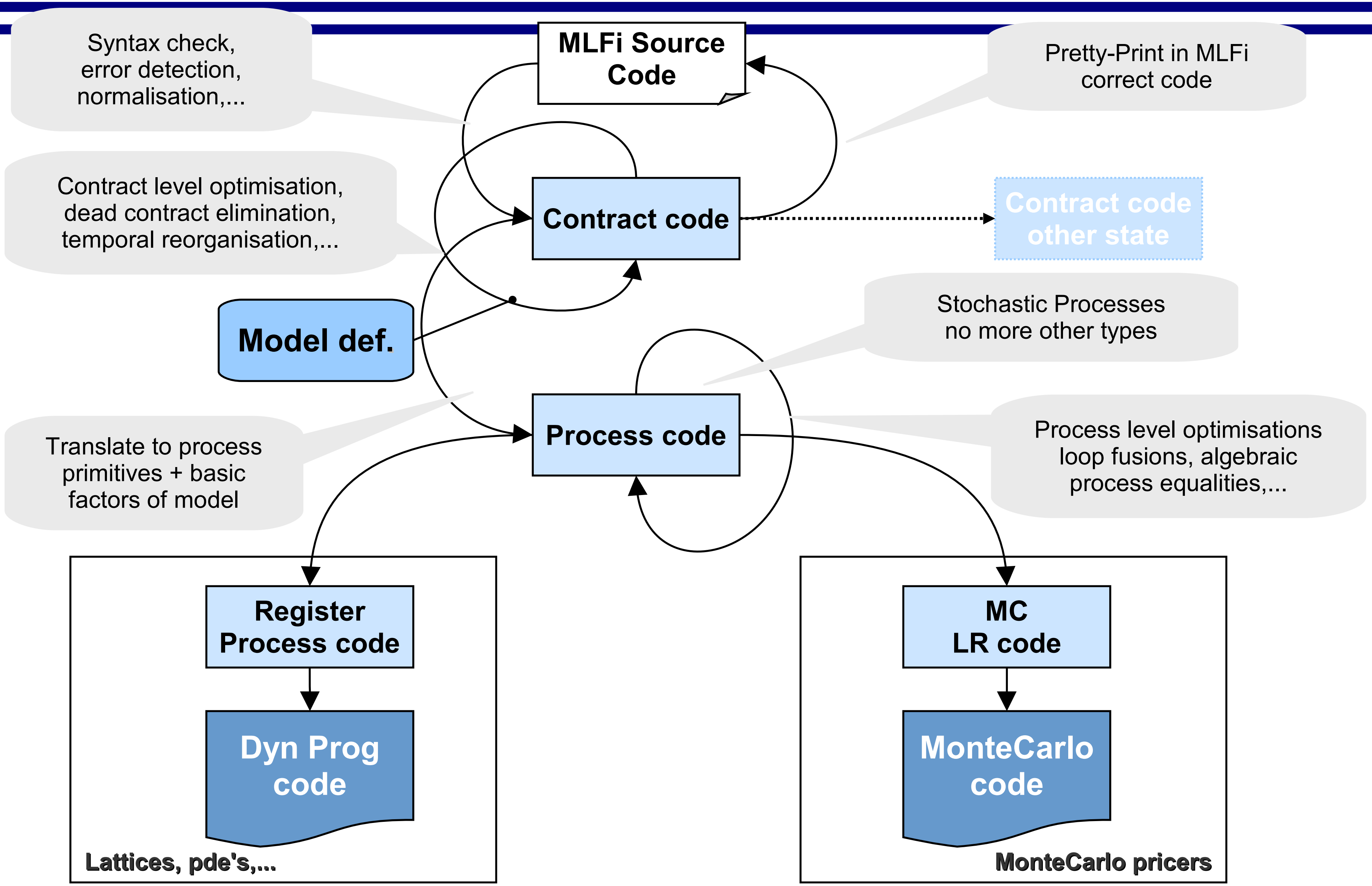
```
american :: (Date,Date) -> Contract -> Contract
american (t1,t2) u
    = get (truncate t1 opt) `then` opt
where
    opt :: Contract
    opt = anytime (perhaps t2 u)
```

- Custom-built Contract

```
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))
```

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.

Generating Code for Financial Instrument Agreement Valuation



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 (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

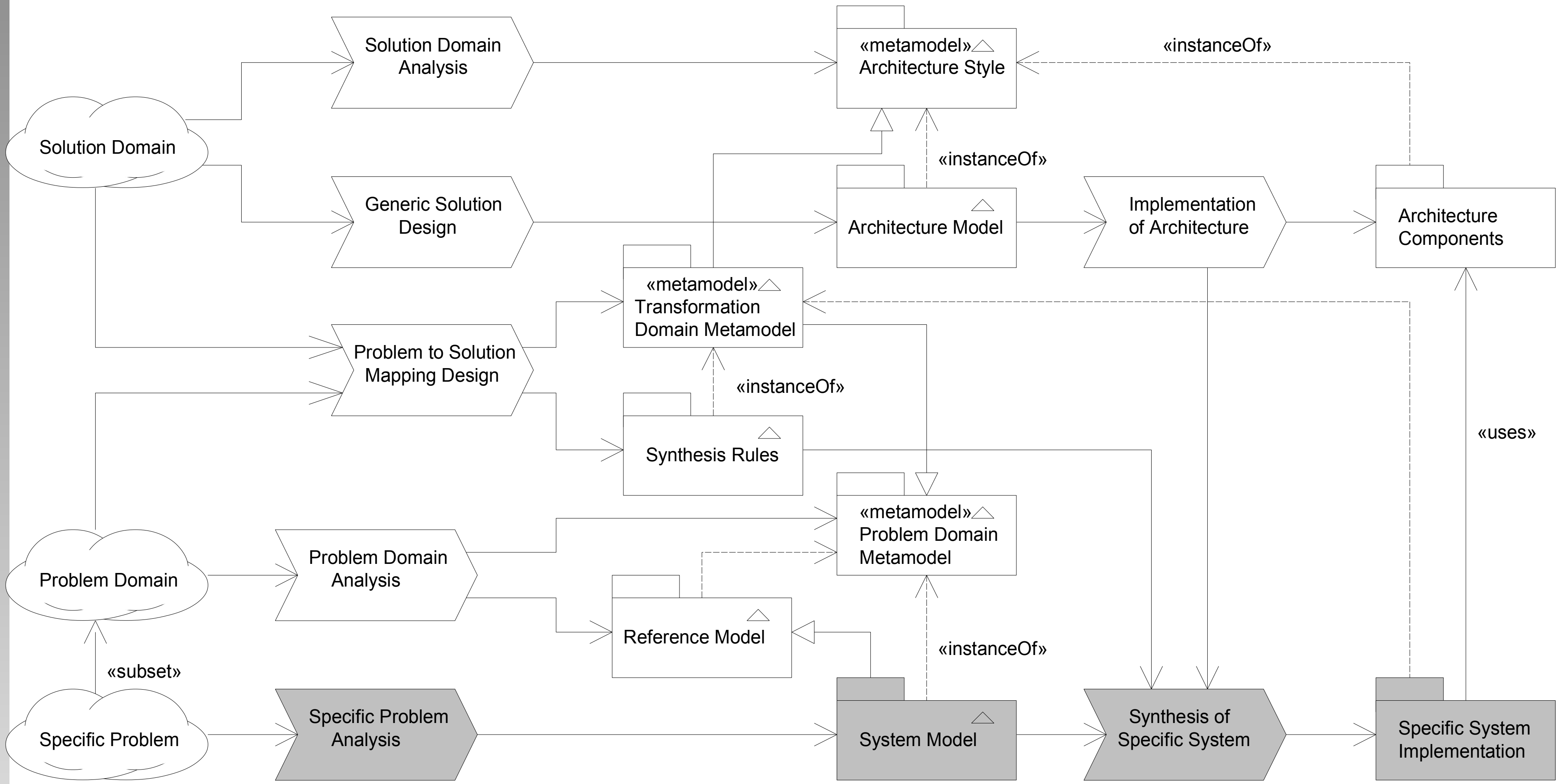
References

- 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!

Questions?

Steps of Model-Oriented Software Development



MDSD Benefits (1)

- Reasons for MDSD
  - 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
  - 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

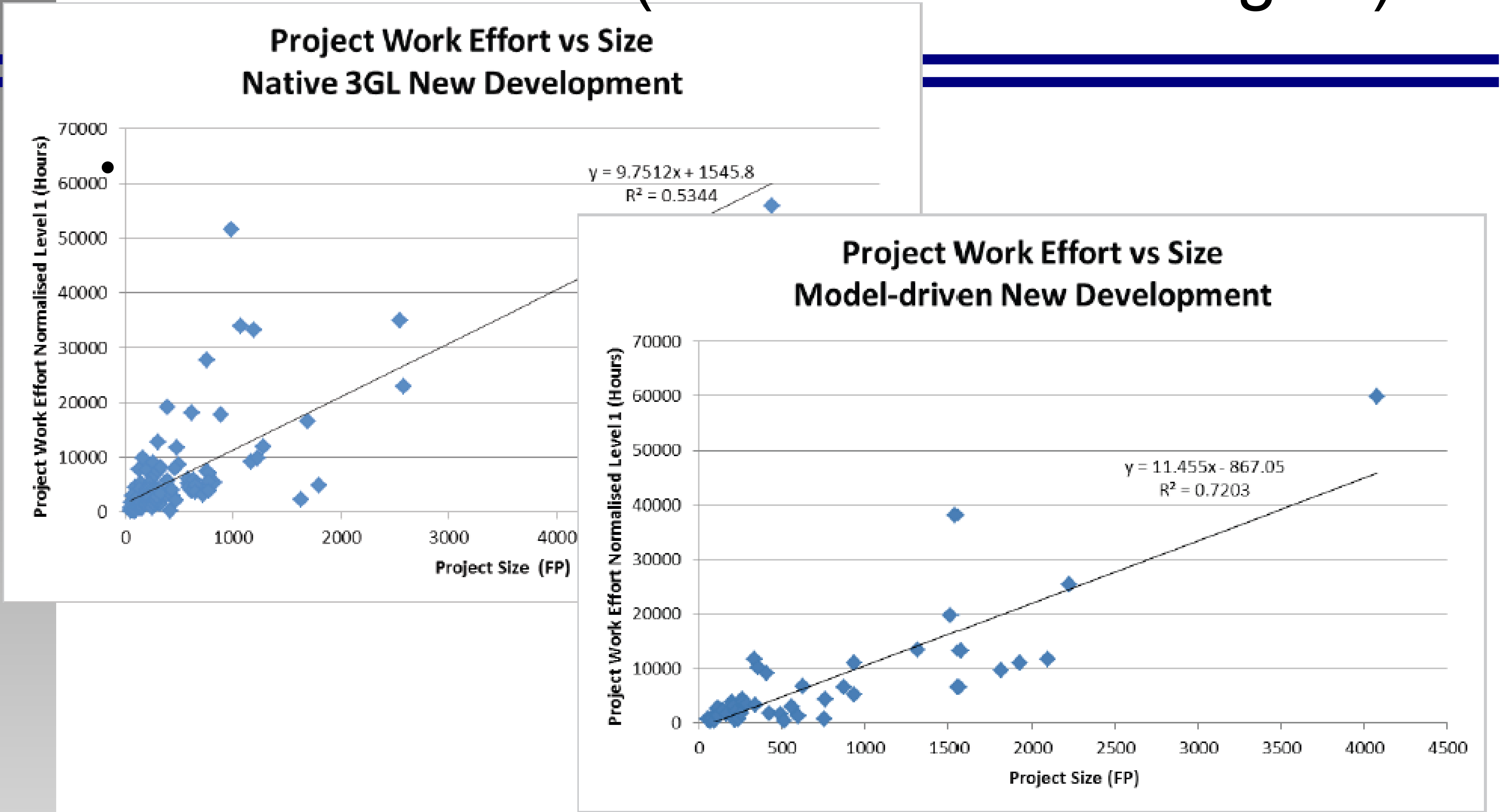
MDSD Benefits (2)

- 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
  - metamodeling
  - generator development
  - architecture
- Development process is more complex
  - domain architecture development
  - application development

Some Statistics (from CA Technologies)



- For MDD projects, project work effort becomes more predictable

Examples of 4 Layers of Models

- M_3 – meta-metamodel
  - a language for compilers – Yacc language syntax definition (maybe in Yacc or in EBNF)
  - XML definition in EBNF
- M_2 – metamodel
  - C language syntax definition in Yacc (“.y” file)
  - XSD schema definition in XSD (or in DTD)
- M_1 – model
  - program in C (“.c” file)
  - schema definition in XSD (or in DTD)
- M_0 – an instance of a model
  - executable code
  - XML file

Definitions 1

- System
  - a collection of interacting components organized to accomplish a specific function or set of functions within a specific environment
- Interface (Connection)
  - a shared boundary between two functional units, defined by various characteristics of the functions
  - component that connects two or more other components for the purpose of passing information from one to the other
- Module (Component)
  - a logically separable part of a system
- Encapsulation
  - isolating some parts of the system from the rest of the system
  - a module has an outside that is distinct from its inside (an external interface and an internal implementation)

Definitions $_2$

- Modularity
  - the degree to which a system is composed of discrete components such that a change to one component has minimal impact on other components
  - the extent to which a module is like a black box
- Coupling
  - the manner and degree of interdependence between modules
  - the strength of the relationships between modules
  - a measure of how closely connected two modules are
- Cohesion
  - the manner and degree to which the tasks performed by a single module are related to one another
  - a measure of the strength of association of the elements within a module

Definitions $_3$

- 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
    - to an observer B, an object M_A is a model of an object A to the extent that B can use M_A to answer questions that interest him about A

      Marvin Minsky

    - M is a model of A with respect to question set Q if and only if M may be used to answer questions about A in Q within tolerance T

      Doug Ross