Software (Systems) Architecture Foundations

Lecture #6

From One to Many Systems

Alar Raabe

Recap of Last Lecture

Strategy without tactics is the slowest route to victory Tactics without strategy is the noise before defeat

Sun Tzu

- It is important to **understand the larger context**
  - what is the super-system and how it changes
  - what are the other (peer) systems in the super-system
  - what are the constraints that super-system imposes
- System of Systems
  - a collection of systems, that pool their resources and capabilities
  - often with separate management and authority
  - **offer more** functionality and performance **than simply the sum of the constituent systems**
- Enterprise is a complex system – a System of Systems
- **Isolate your system** from the environment/context – build for
  - *interoperability* – reconcile differences between interfaces
  - *change* – detect changes and adapt to it
  - *failure* and *unexpected* – detect failures and recover from these
  - *security* – detect threats and neutralize these

Recap of Last Lecture

Strategy without tactics is the slowest route to victory Tactics without strategy is the noise before defeat

Sun Tzu

- When architecting System of Systems,
  - concentrate on *interfaces* (build platforms)
  - provide *collaboration incentives*
  - design so that value can be delivered even by incomplete System of Systems
- When analyzing/designing a complex system
  - ask WHY, WHAT, HOW, WHO, WHEN, WHERE
  - describe the answers for different stakeholders (interested parties)
- Enterprise Architecture is a
  - **holistic view** on whole enterprise – a description of the enterprise to provides a common understanding
  - **strategic planning tool** – a bridge between Strategy and Execution
- Reference architectures provide for a particular domain, a common vocabulary, reusable designs and industry best practices

Content

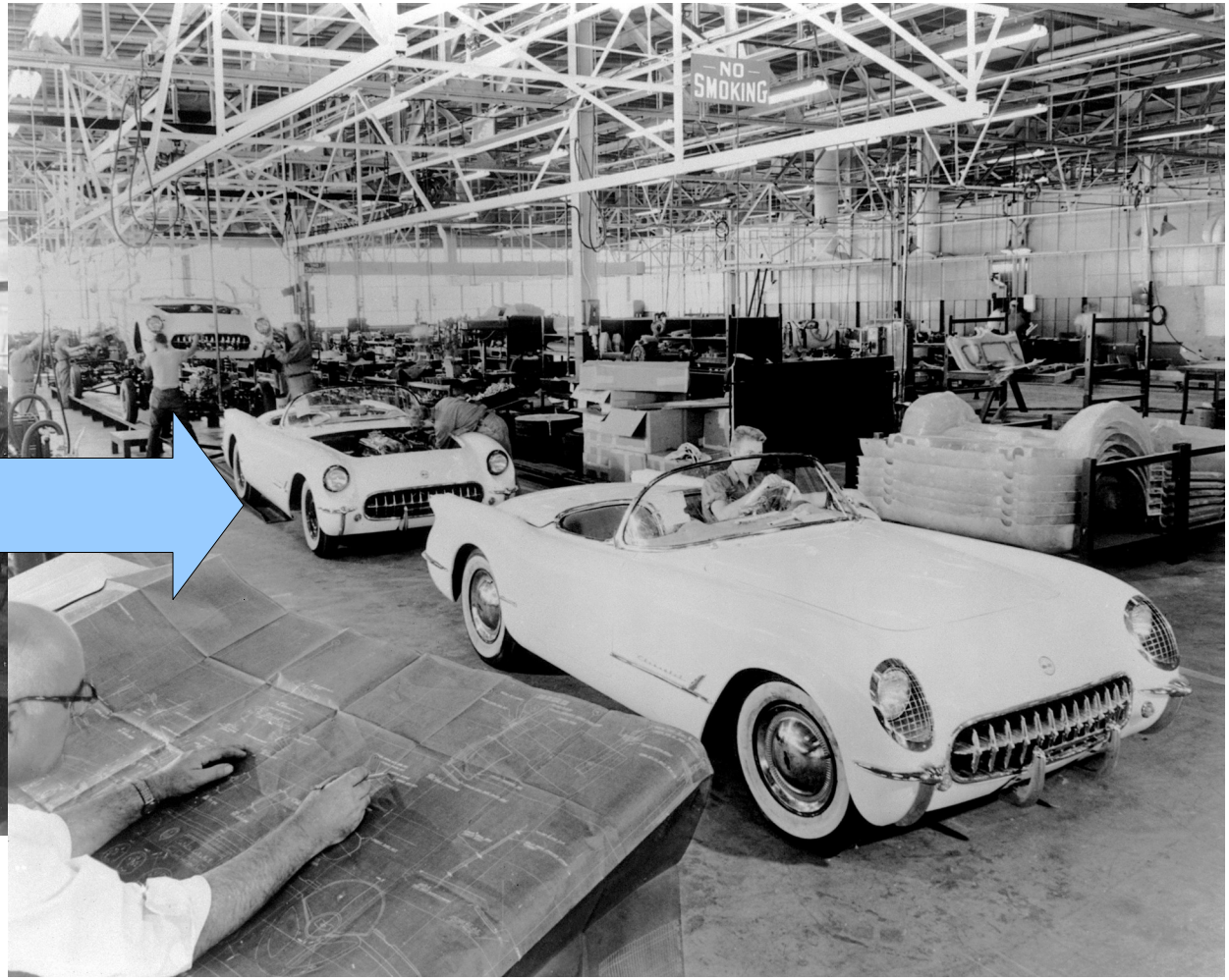
Any customer can have a car painted any color that he wants so long as it is black

Henry Ford

- From one system to many
- System Families
  - Product-line architectures
- Model-driven development
  - Model as Primary Artifact
  - Generative Programming
  - Feature Modeling
- Conclusions

From One to Many

- Standardization



From One to Many
Balancing Standardization with Variability

- Platform Architecture

Content

Any customer can have a car painted any color that he wants so long as it is black

Henry Ford

- From one system to many
- System Families
  - Product-line architectures
- Model-driven development
  - Model as Primary Artifact
  - Generative Programming
  - Feature Modeling
- Conclusions

System/Program Families

When developing a family of programs, develop, and document, the common/shared aspects first

D. L. Parnas

- To industrialize the software industry, **software components** (routines), widely applicable to different machines and users, should be available **in families** arranged according to precision, robustness, generality and time/space performance
(M. D. McIlroy, 1968)

- A program can be viewed as **an ordered set of progressively less abstract forms**, where in all next forms one or more concepts used above are explained (refined) – the **family of programs** becomes the set of forms from a given collection that can be strung into **a fitting chain of refinements**
(E. W. Dijkstra, 1969)

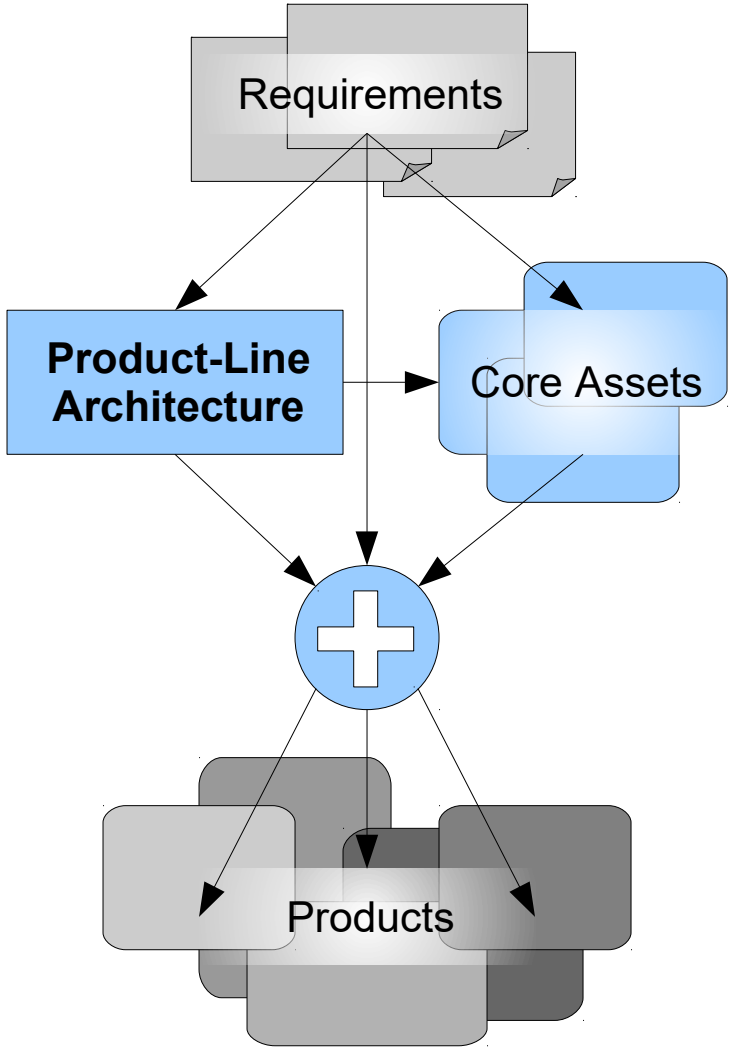
- We consider a set of programs to be a **program family** if they have so much in common that it pays to **study their common aspects** before looking at the aspects that differentiate them
(D. L. Parnas, 1976)

- The members of a program family can vary by
  - *implementation methods* used – a set of programs which meets the module specifications form the family
  - variation in the *external parameters* – programs differing in the values of those parameters form the family
  - *use of subsets* – programs consisting of a subset of the parts described by a set of module specifications form the family

Product-Line Architecture
a blue-print for creating a family of Systems

CMU SEI

- A software **product-line** is a set of
  - software systems that **share a common, managed set of features** satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way
  - **reusable assets** (called core assets, including designs and their documentation, user manuals, project management artifacts, test cases, etc.) **based on a common architecture** and the software elements that populate that architecture
- A **product-line architecture** is the architecture with **built-in variation points** – decisions, intentionally left open
  - Each product in the product line may have a slightly different architecture – these are instances of the product-line architecture
  - The architecture for a product ("instance" of product-line architecture) is created when builder exercises the variation mechanisms that the product-line architect has put in place



Product-Line Architecture Benefits

CMU SEI

- Overall benefits
  - *Large scale productivity gains* and more efficient use of human resources
  - *Decreased time to market* and increased market agility
  - *Increased product quality*, customer satisfaction and decreased product risk
  - Ability of mass customization, to maintain market presence and to sustain growth

- Directs cost savings *for each product* after core assets have been built
  - *Requirements* – product requirements are deltas to established common requirements base
  - *Architecture* – an architecture for a software system represents a large investment
  - *Components* – up to 100% of the components in the core asset base are used in each product
  - *Modeling and analysis* – models and the associated analyses are existing for core assets
  - *Testing* – generic test plans, processes, data, harnesses, etc. exist, and need only be tailored
  - *Planning* – budgets and schedules from previous projects provide basis for the planning
  - *Processes* – the overall software development process is in place and has been used before
  - *People* – fewer people are required to build products, and the people are more easily transferred

A software product line approach provides options to future market opportunities – permit low cost/low/risk experiments

Product-Line Architecture
Some Variation Mechanisms

CMU SEI

- *Inclusion* or *omission* of elements
  - can be reflected in the build procedures for different products, or the implementation of an element can be conditionally compiled based on some parameter indicating its presence or absence
- *Inclusion of a different number* of replicated elements
- *Selection/substitution of different versions* of elements that have the same interface but different behavioral or quality attribute characteristics
  - can occur at compile time, build time, or runtime (via static libraries, dynamic link libraries, or add-ons (e.g., plug-ins, extensions, and themes), which add or modify application functionality at runtime)
- *Extension points* – identified places in the architecture where additional behavior or functionality can be safely added
- *Reflection* – the ability of a program to manipulate data on itself or its execution environment or state (reflective programs can adjust their behavior based on their context)
- *Overloading* – reusing a named functionality to operate on different types (promotes code reuse, but at the cost of understandability and code complexity)

Example

Product-Line Architecture
Common Variation Mechanisms

CMU SEI

Variation Mechanism	Properties Relevant to Building the Core Assets	Properties Relevant to Exercising the Variation Mechanism When Building Products
Inheritance; specializing or generalizing a particular class	Cost: Medium Skills: Object-oriented languages	Stakeholder: Product developers Tools: Compiler Cost: Medium
Component substitution	Cost: Medium Skills: Interface definitions	Stakeholder: Product developer, system administrator Tools: Compiler Cost: Low
Add-ons, plugins	Cost: High Skills: Framework programming	Stakeholder: End user Tools: None Cost: Low
Templates	Cost: Medium Skills: Abstractions	Stakeholder: Product developer, system administrator Tools: None Cost: Medium
Parameters (including text pre-processors)	Cost: Medium Skills: No special skills required	Stakeholder: Product developer, system administrator, end user Tools: None Cost: Low
Generator	Cost: High Skills: Generative programming	Stakeholder: System administrator, end user Tools: Generator Cost: Low
Aspects	Cost: Medium Skills: Aspect-oriented programming	Stakeholder: Product developer Tools: Aspect-oriented language compiler Cost: Medium
Runtime conditionals	Cost: Medium Skills: No special skills required	Stakeholder: None Tools: None Cost: No development cost; some performance cost
Configurator	Cost: Medium Skills: No special skills required	Stakeholder: Product developer Tools: Configurator Cost: Low to medium

Content

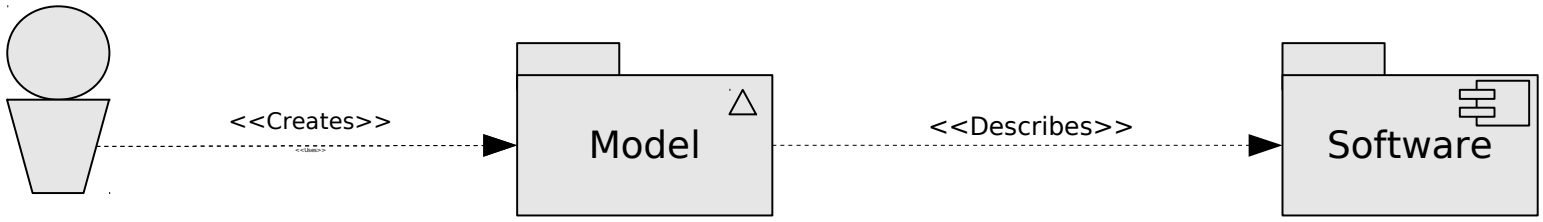
Any customer can have a car painted any color that he wants so long as it is black

Henry Ford

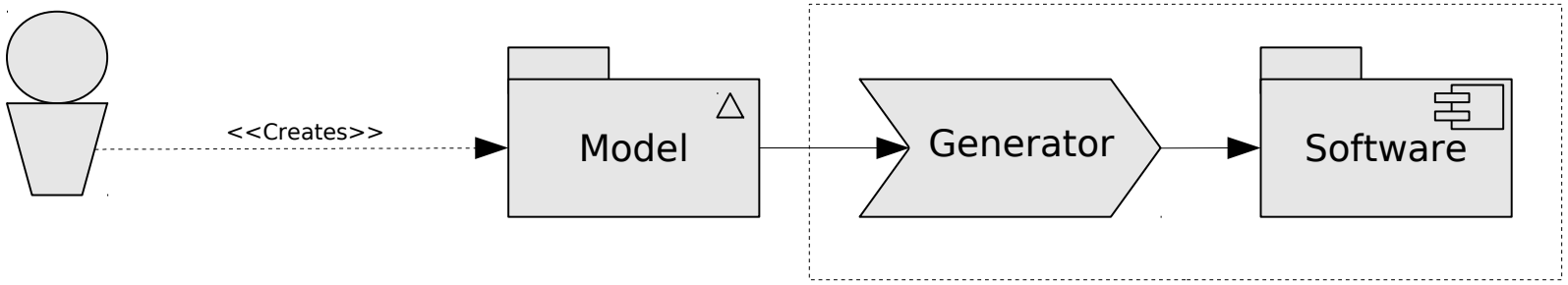
- From one system to many
- System Families
  - Product-line architectures
- Model-driven development
  - Model as Primary Artifact
  - Generative Programming
  - Feature Modeling
- Conclusions

Using Models in Software Development

- Models as Descriptions and Illustrations (Documentation)



- Models as Primary Artifacts (Models as Software)



Automation of Programming → Excursion into the History

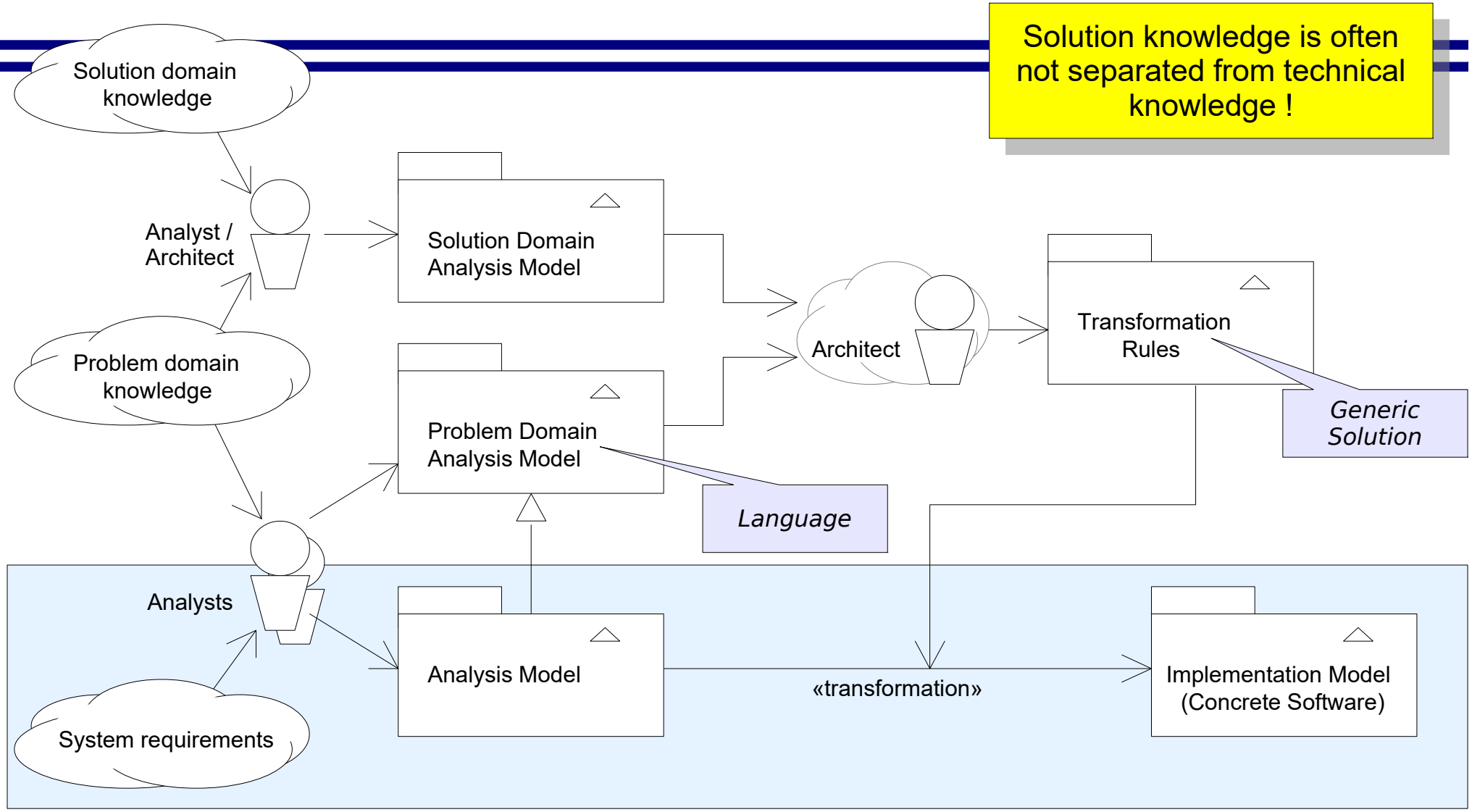
What has been will be again,
what has been done will be done again;
there is nothing new under the sun.

Ecclesiastes 1:9

- Automation of coding (*modelling solution*)
  - FORTRAN (1954), Lisp (1956)
  - APT (MIT 1957) ← *First DSL !*
  - Algol (1958)

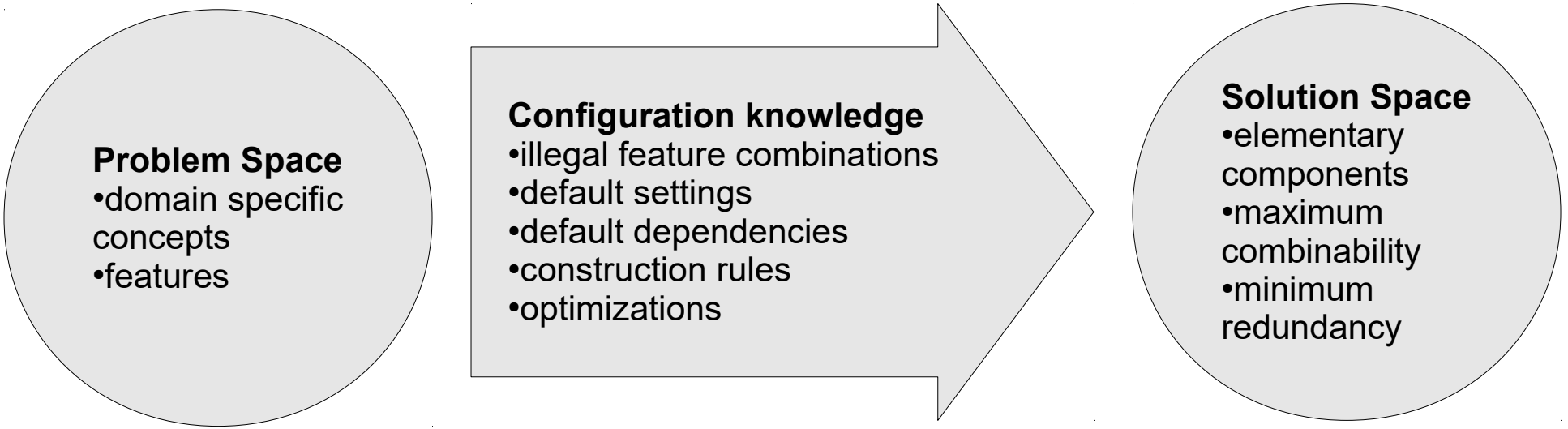
- Automation of programming (*modelling problem*)
  - Problem-Oriented Languages/Systems (program synthesis)
    - ICES (MIT 1961) → COGO, STRUDL, BRIDGE, ...
    - **PRIZ (ETA Kübl)**
  - Application Generators (program generation)
    - Compiler Generators – Yacc/Lex (1979)
    - Application Generators – MetaTool & GENII/GENOA & ... (Bell Labs 1980s)
  - CASE (Computer-Aided Software Engineering) tools (software generation)
    - GraphiText, DesignAid (Nastec 1982)

Model-Driven Software Development Approach



Generative Programming

Czarnecki, Eisenecker



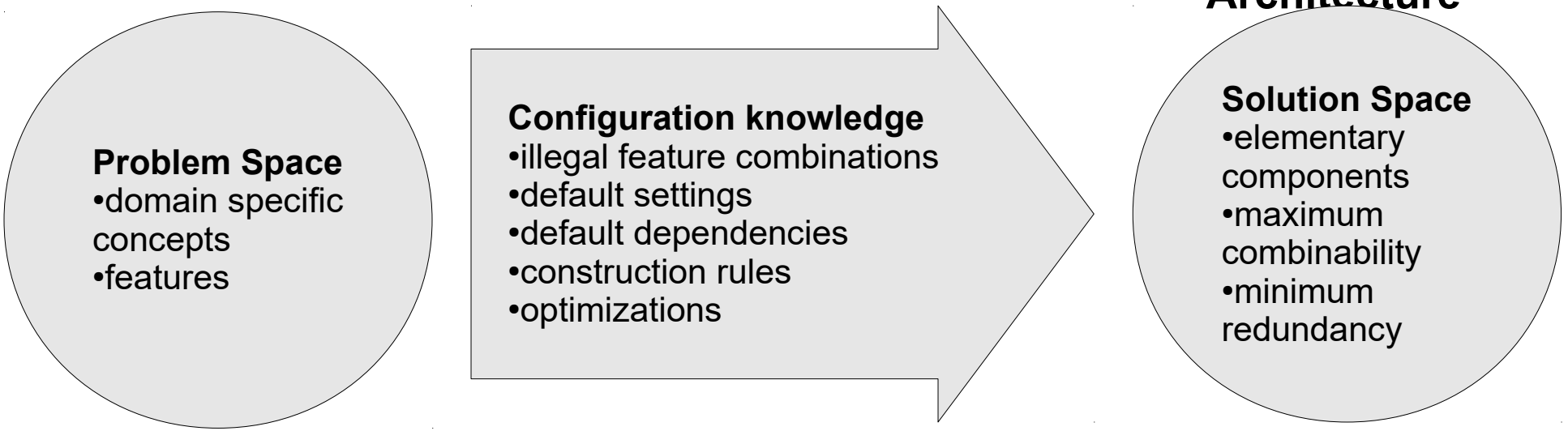
Generative Programming

Czarnecki, Eisenecker

Domain Specific Language (DSL)

Generator Reflection

Components + **System Family Architecture**



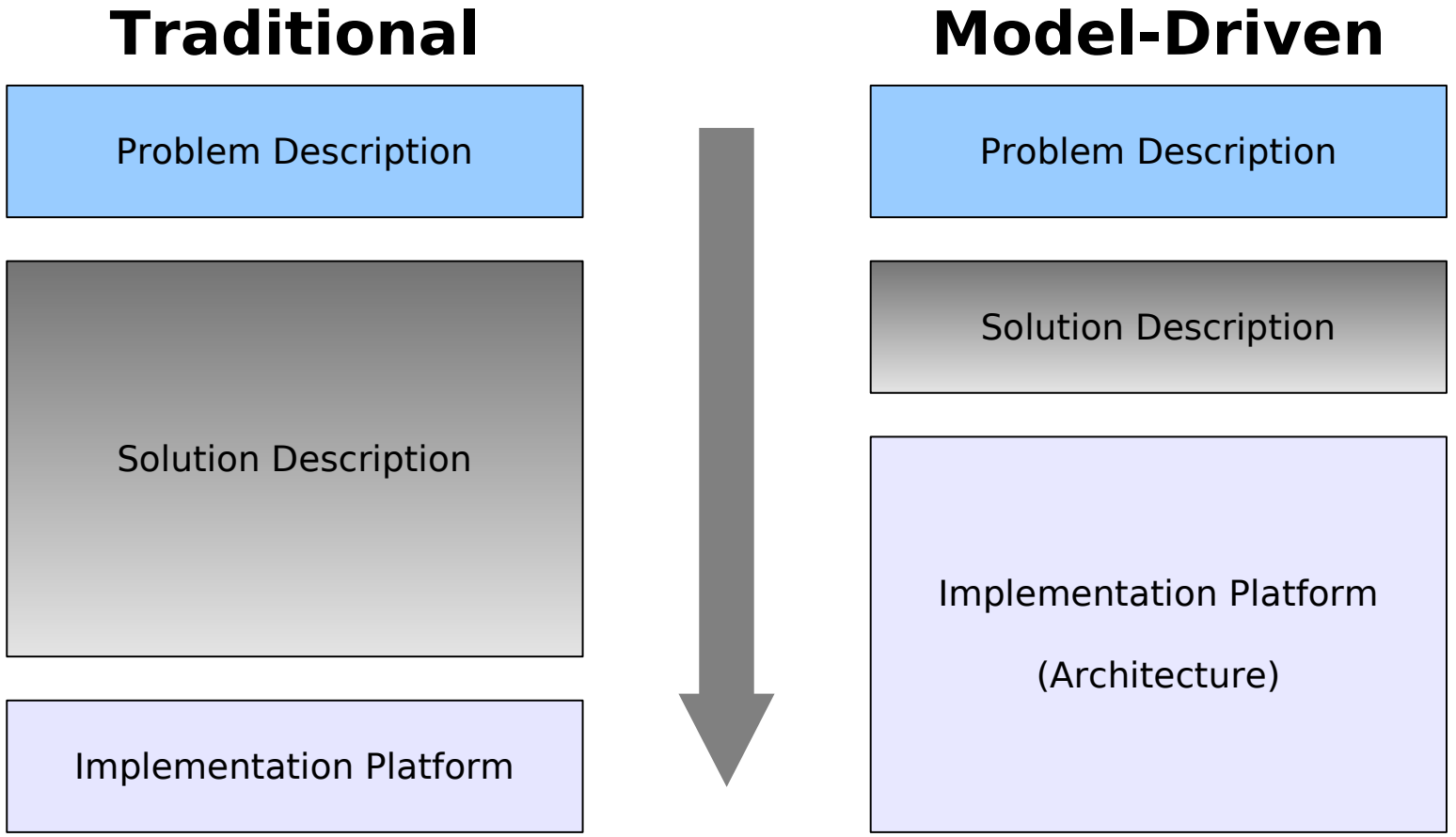
DSL Technologies
- •programming language
- •extensible languages
- •textual languages
- •graphical languages
- •interactive wizards
- •any mixture of above

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

Comparing to the Traditional Development

Reducing the gap



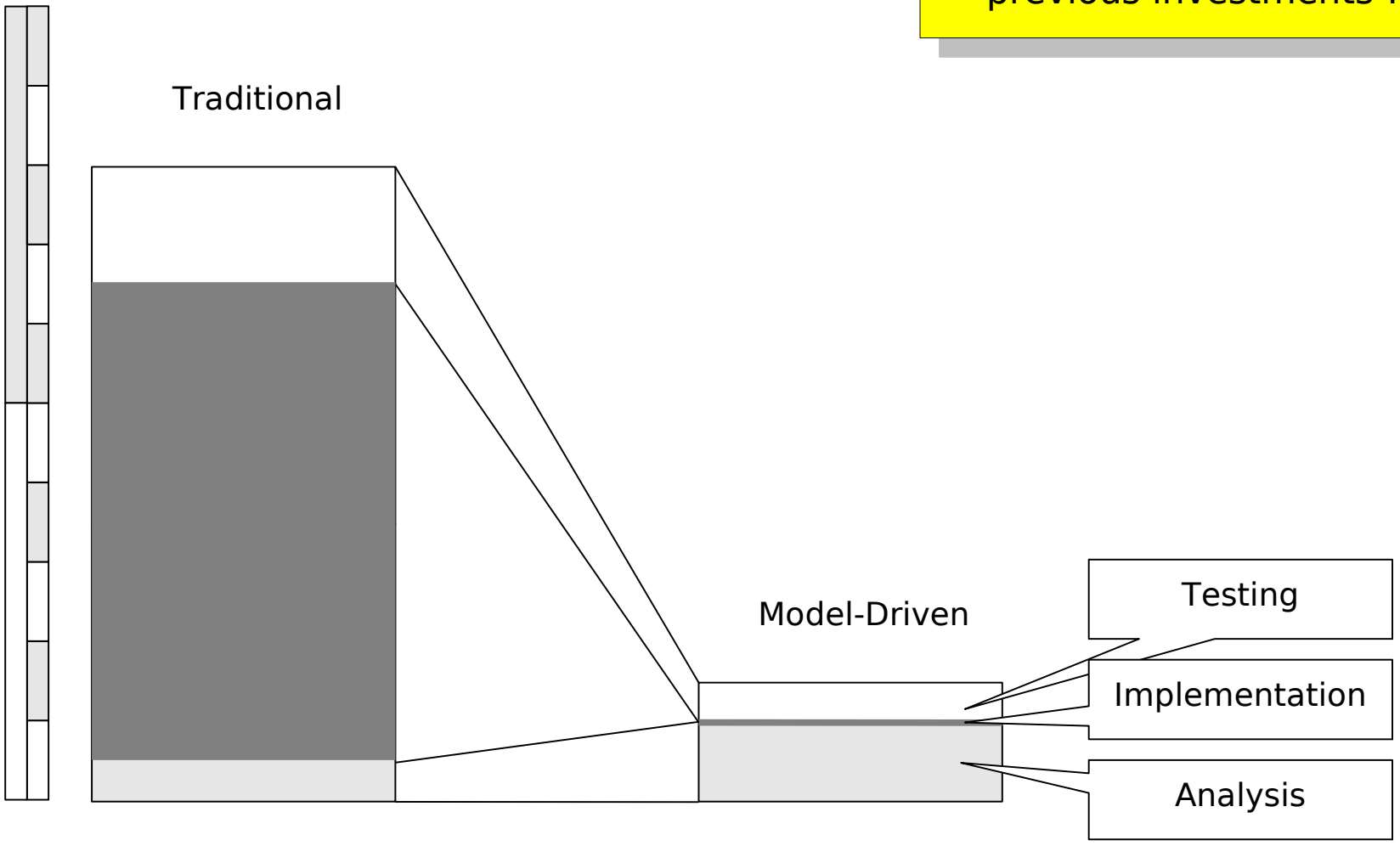
EXAMPLE

Comparing Model-Driven Method with Traditional

- Effort for First Iteration – basic 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 without additional effort*

Comparing Model-Driven Method with Traditional

Benefits are based on previous investments !



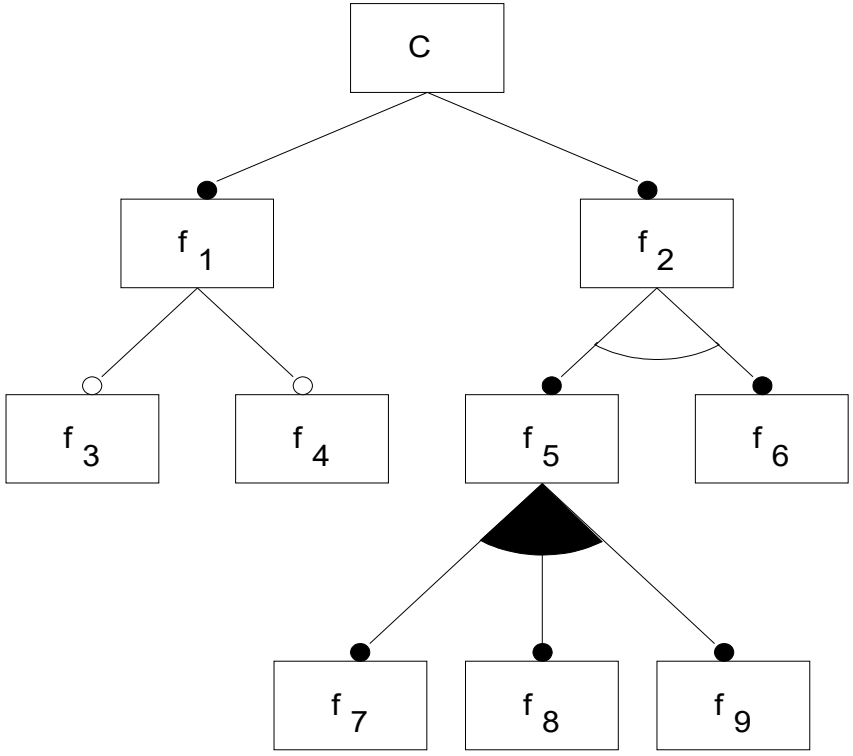
Feature Modeling

Configuration knowledge in Generative Programming

- Feature modeling (or feature analysis)
  - is the activity of modeling the common and the variable properties of concepts and their inter-dependencies
- In feature modeling
  - ***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, …)



Content

Any customer can have a car painted any color that he wants so long as it is black

Henry Ford

- From one system to many
- System Families
  - Product-line architectures
- Model-driven development
  - Model as Primary Artifact
  - Generative Programming
  - Feature Modeling
- Conclusions

Conclusions

The architect must be a prophet – if he can't see at least ten years ahead don't call him an architect.

Frank Lloyd Wright

- A software product line is a set of
  - software systems that **share a common, managed set of features** and that are developed from a common set of core assets in a prescribed way
  - **reusable assets** (called core assets) **based on a common architecture** and the software elements that populate that architecture
- Selected variation mechanisms must support
  - the variations reflected in the products (often manifested as different quality attributes)
  - the production strategy and production constraints (support the way the organization plans to build products)
  - efficient integration (a large number of products requires a smooth and easy process) – some degree of automation
- Primary architectural variation mechanisms are
  - *Inclusion or omission* of elements or inclusion of a different number of replicated elements
  - *Selection/substitution* of different versions of elements with the same interface but different behavioral or quality attribute characteristics (libraries or addons/plugins)
  - *Reflection* (ability to adjust the behavior based on the context)

53. The great way is easy, yet programmers prefer the side paths. Be aware when things are out of balance. Remain centered within the design.

Lao Tsu (by Philippe Kruchten)

Thank You!

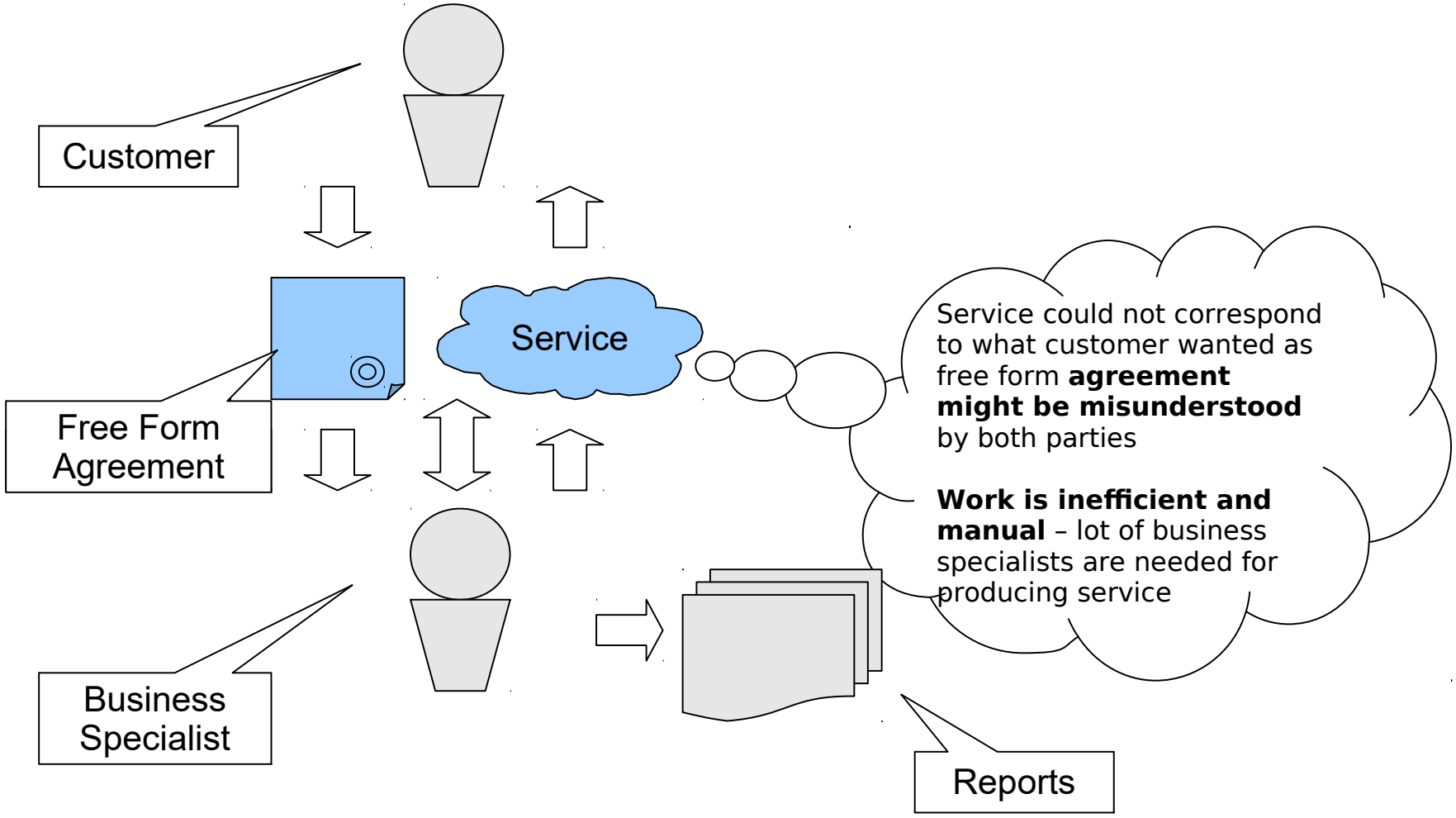
Questions

- What is program family?
- What is a software product-line and from what it consists of?
- What is a software product-line architecture?
- What are benefits of a software product-line?
- What is the role of variation mechanisms in software product-line architecture?
- List main variability mechanisms in product-line architectures?
- How models can be used in software development?
- What is model-driven software development?
- What is generative programming?
- What is a domain specific language?

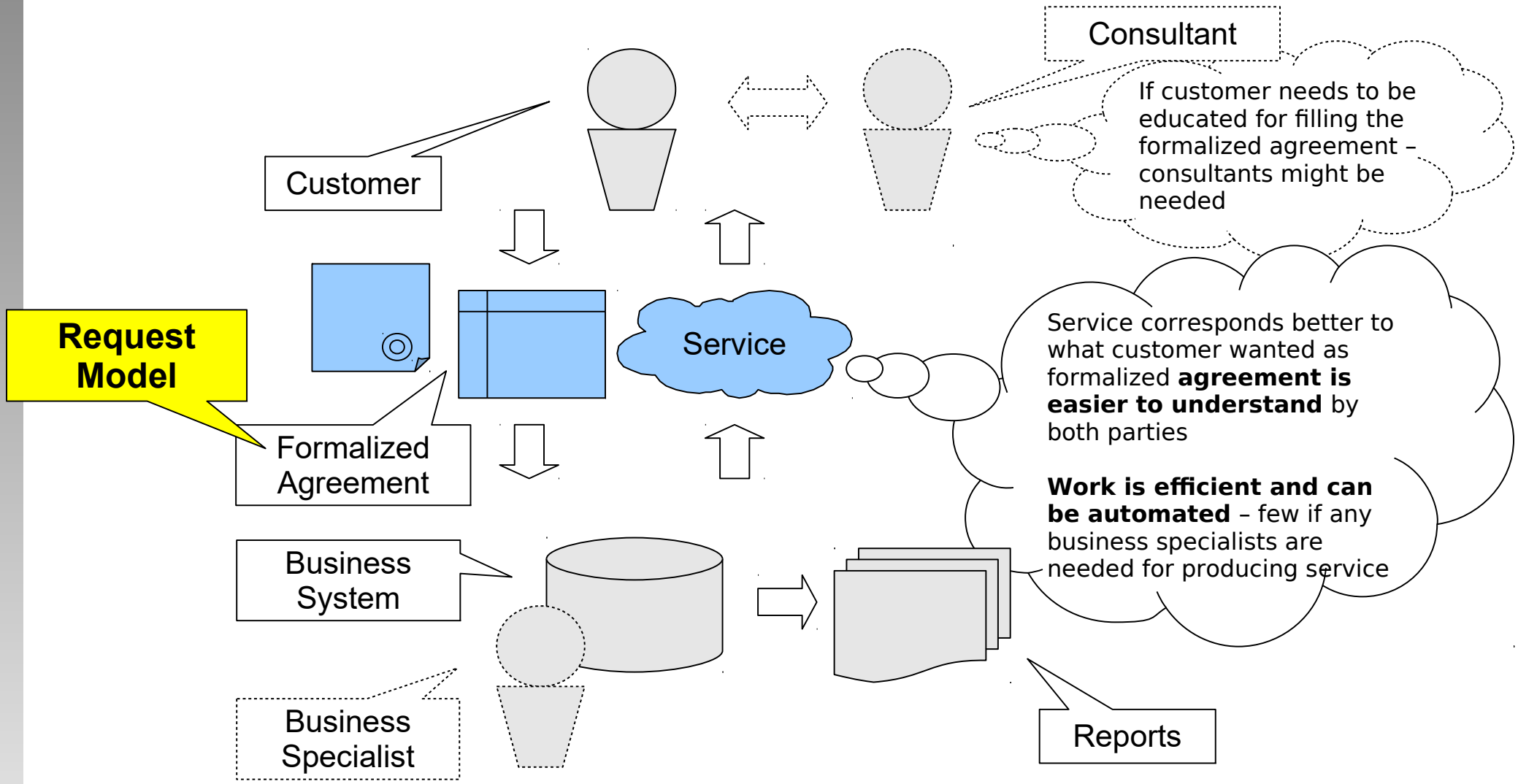
Literature

- http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF
- http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.136.2645&rep=rep1&type=pdf
- http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF
- https://pdfs.semanticscholar.org/4d3a/fd1b6cab4ed952137d2499f4468bb5da3670.pdf
-
- http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.136.2645&rep=rep1&type=pdf
-
- https://www.cs.utexas.edu/ftp/predator/stja.pdf
- https://www.janbosch.com/articles/pla-casestudy.pdf
- https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=495357
- https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=11231
-
- … Google “software product line” + “model-driven development” + “feature models” + “variability management”...

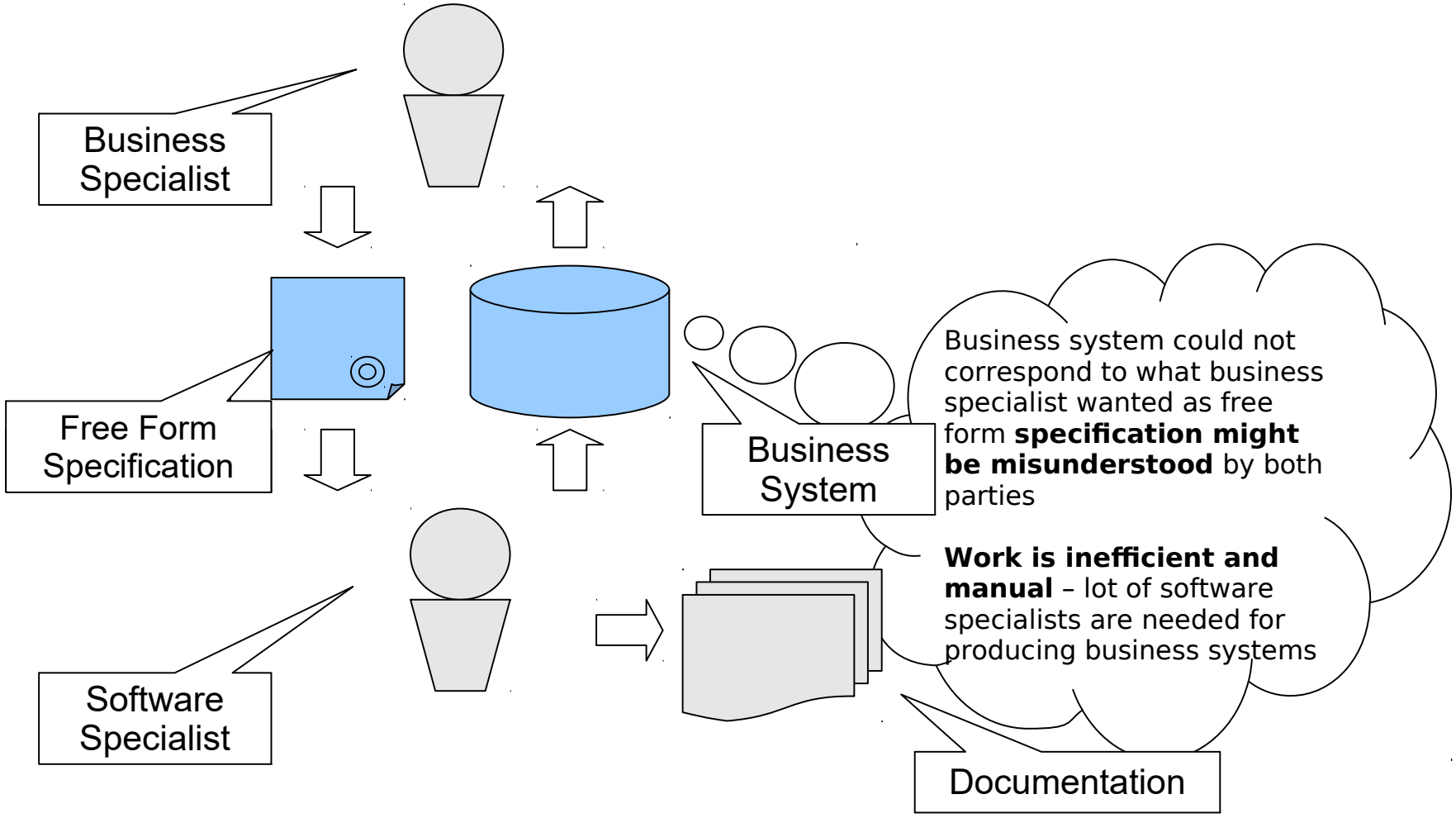
How we did Business Yesterday



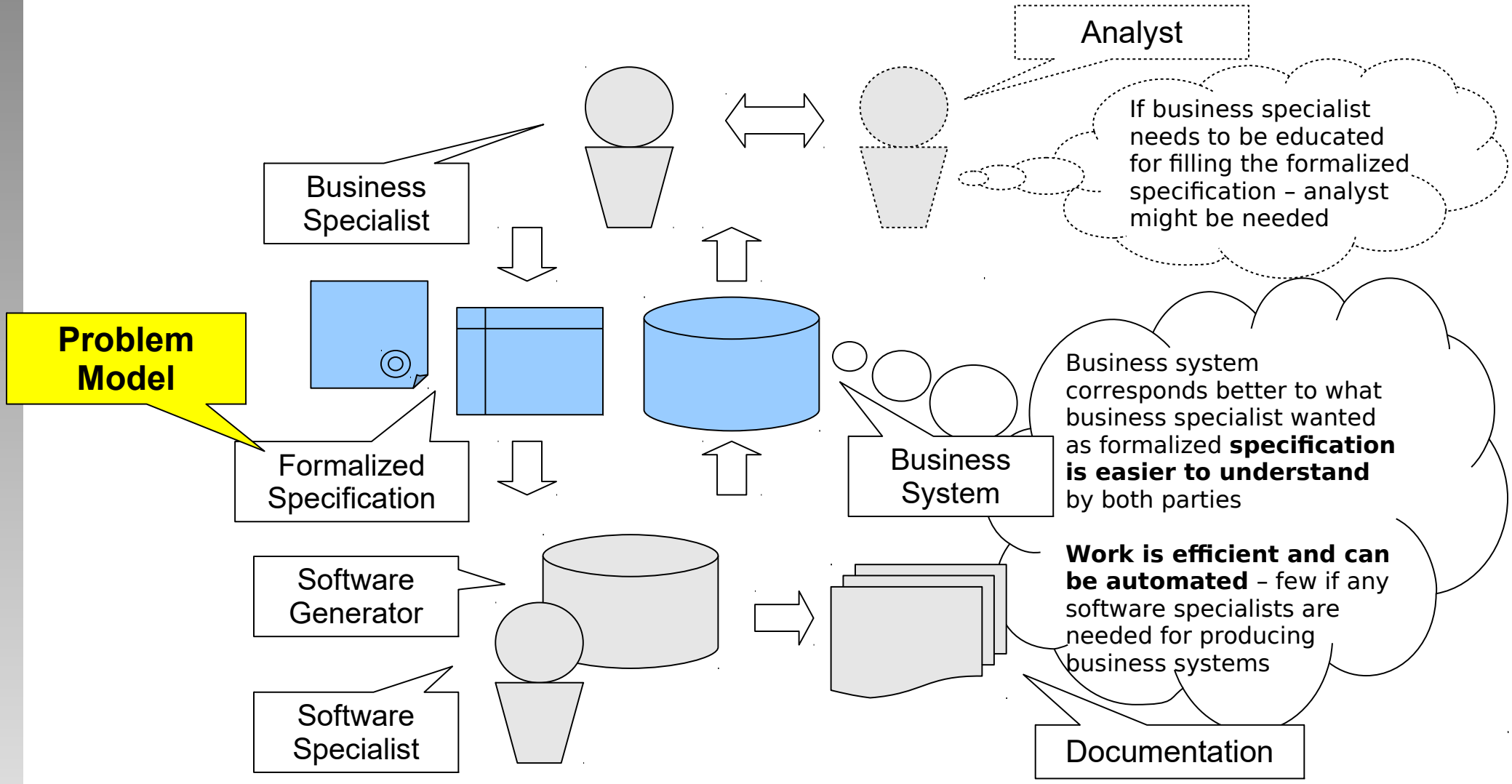
How we do Business Today/Tomorrow



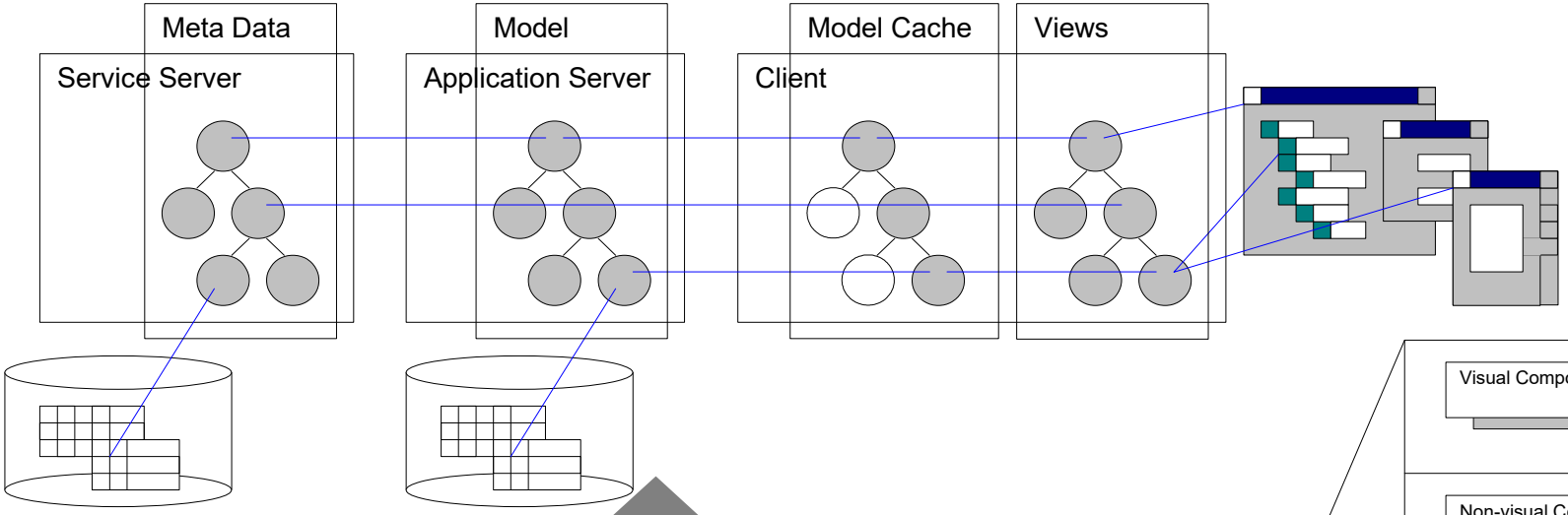
How we Develop Software Today

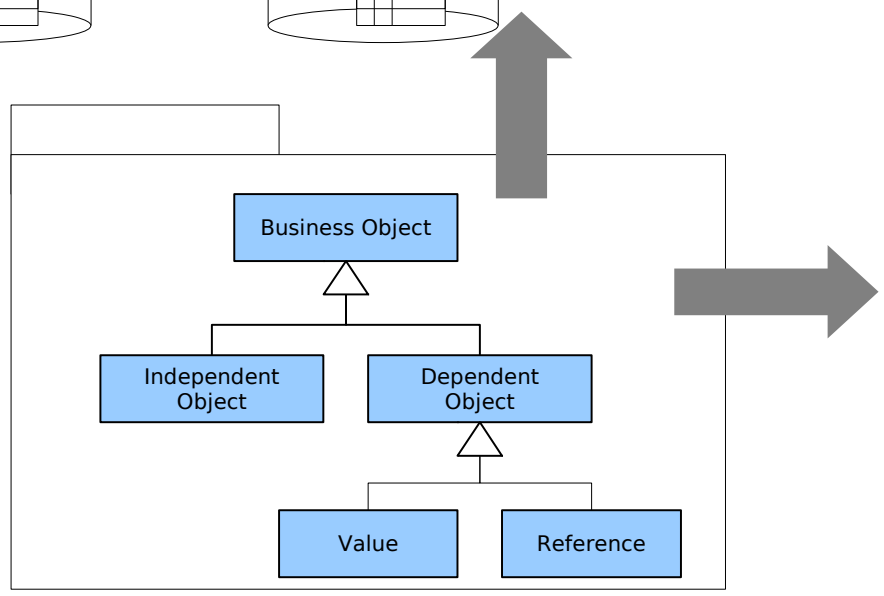


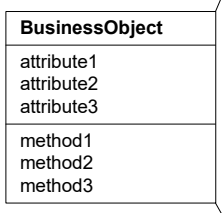
How we should Develop Software

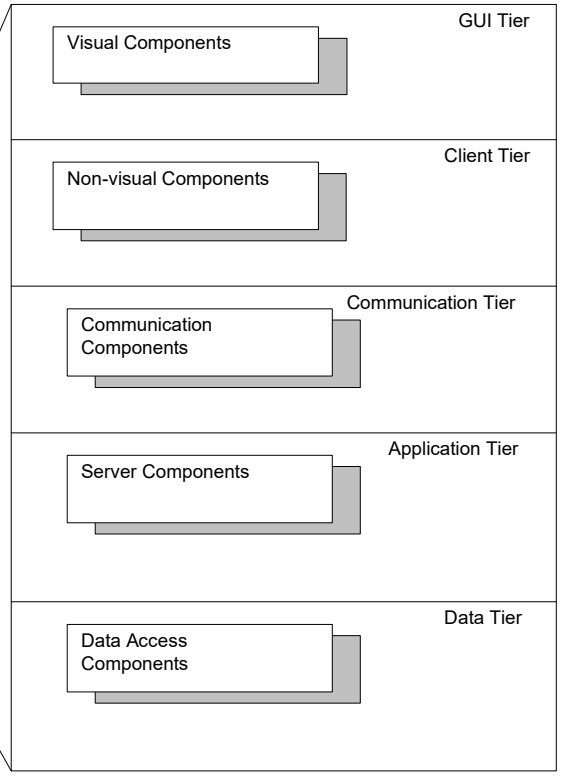


Consistency of Implementation



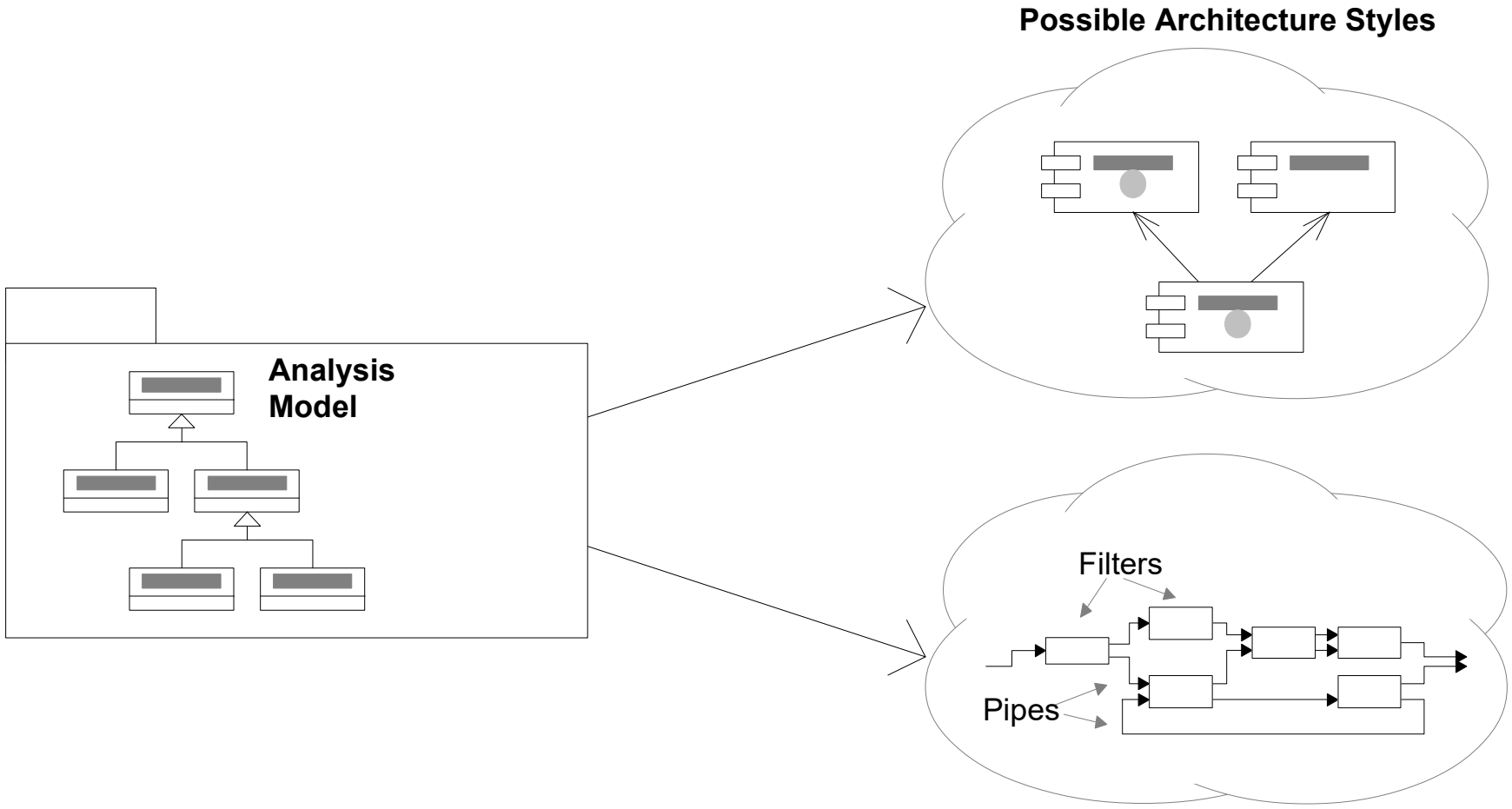




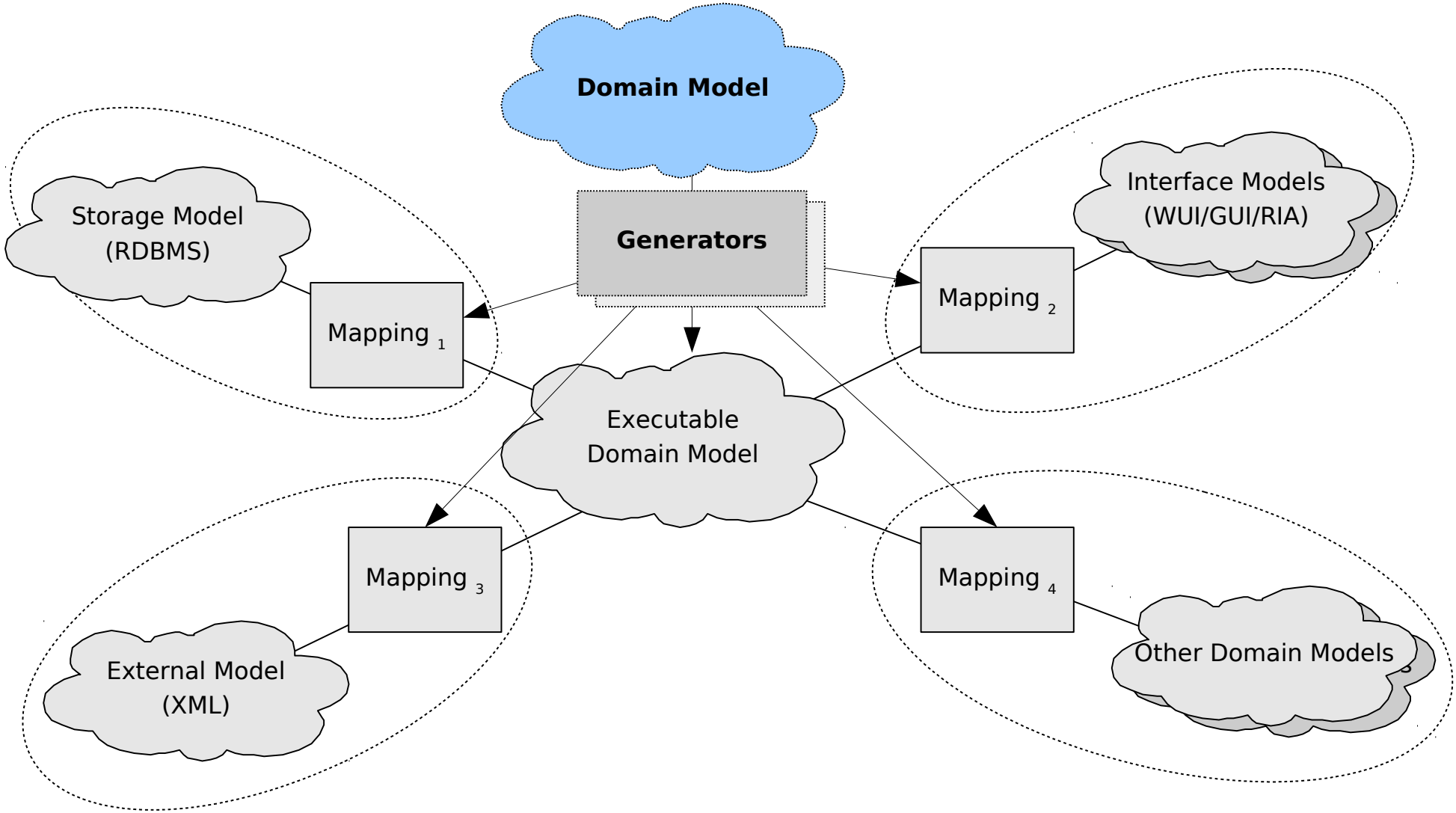


PROBLEM

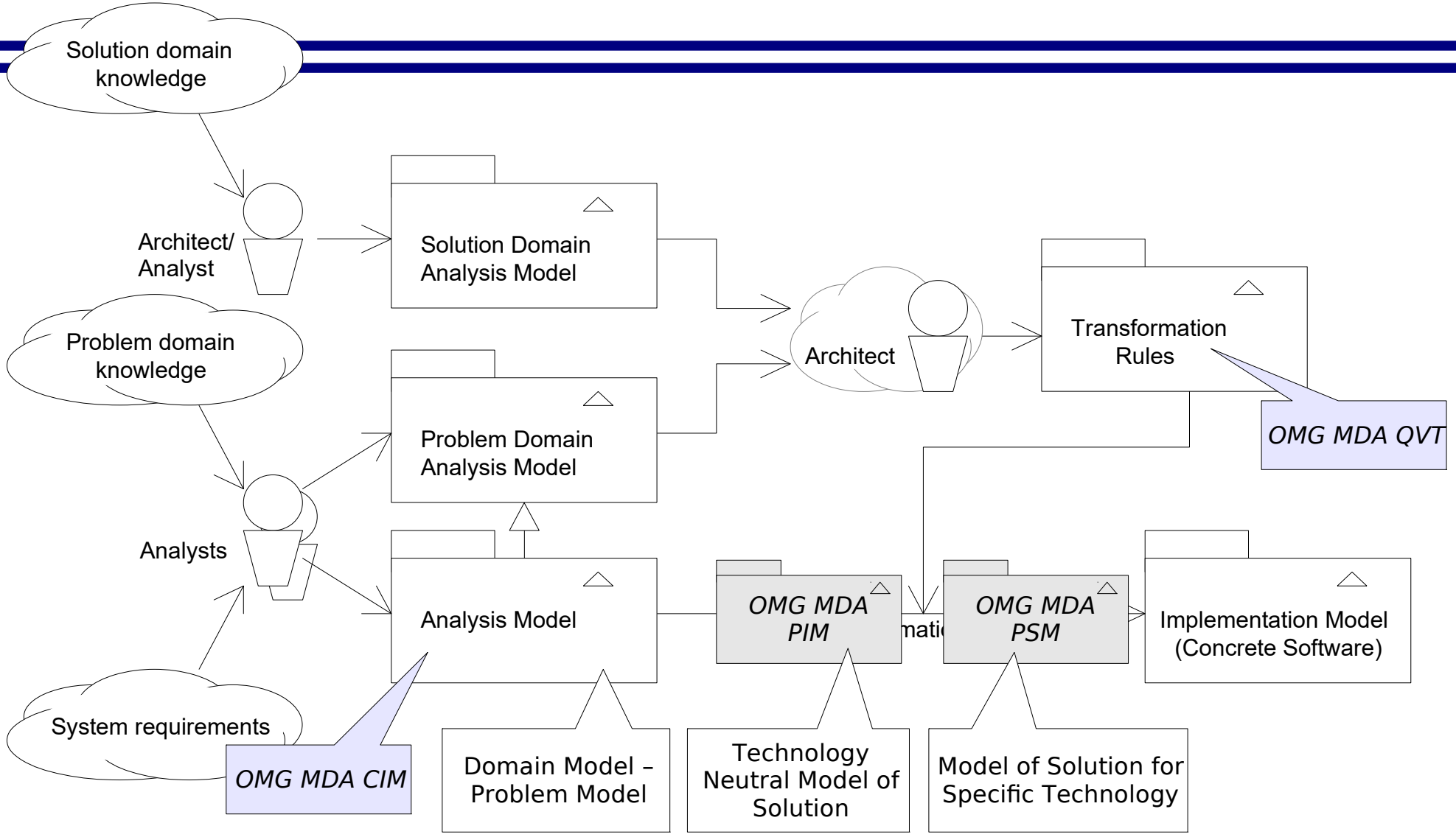
Mapping to Different Implementations



Domain Model → Source for Solution



OMG MDA Approach



Detailed Steps of Model-Driven Software Development