

A model-driven software factory to support enterprise application product lines

Modeling Wizards Master Class, 2 Oct '10, Oslo

Vinay Kulkarni

Tata Consultancy Services, Pune, India

Vinay.vkulkarni@tcs.com

Outline

- Background
- Business applications
- Model Driven Development
- Generating business applications from models
- MDD toolset
- A method for disciplined use of tools
- Generating complete application from its model - experience and lessons learnt
- 'Code is model' approach
- MDD tools product line – model-based generation of model-based generators
- Generating application families from its model
- Towards a model-driven software factory

Background

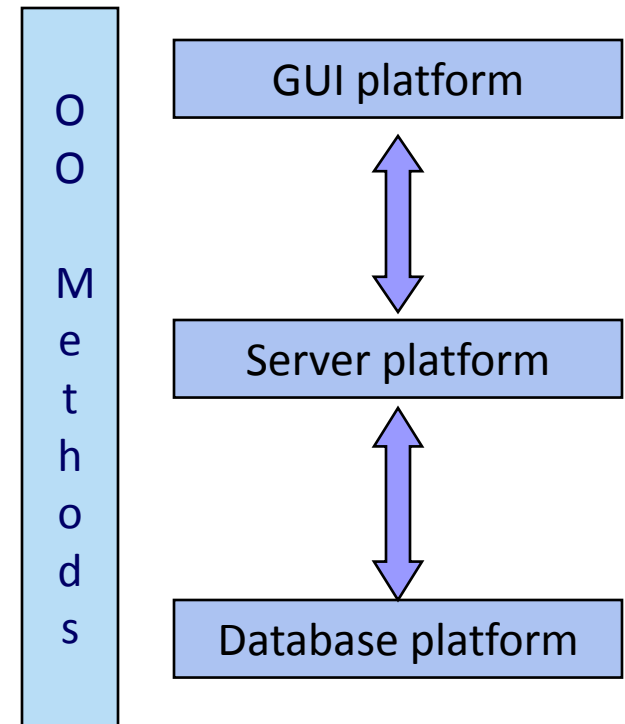
Back in '94

- A highly successful consultancy organization
 - Doubling every two years
- Enviably track record in delivering **high quality** large 'bespoke' business applications **on time**
 - SEGA Intersettle
- Thought of the 'unthinkable'
- Object orientation was taking root
 - Silver bullet
- Distributed computing was gaining over centralized processing a la mainframes
 - Lower TCO
- Technology was changing at increasingly rapid rate
 - Reduced time to market
 - Faster obsolescence

Decided to enter the products business

First step

- A Banking Product
- New Complex Platform
 - C++, PB
 - Tuxedo
- New Methods
 - OO, OMT

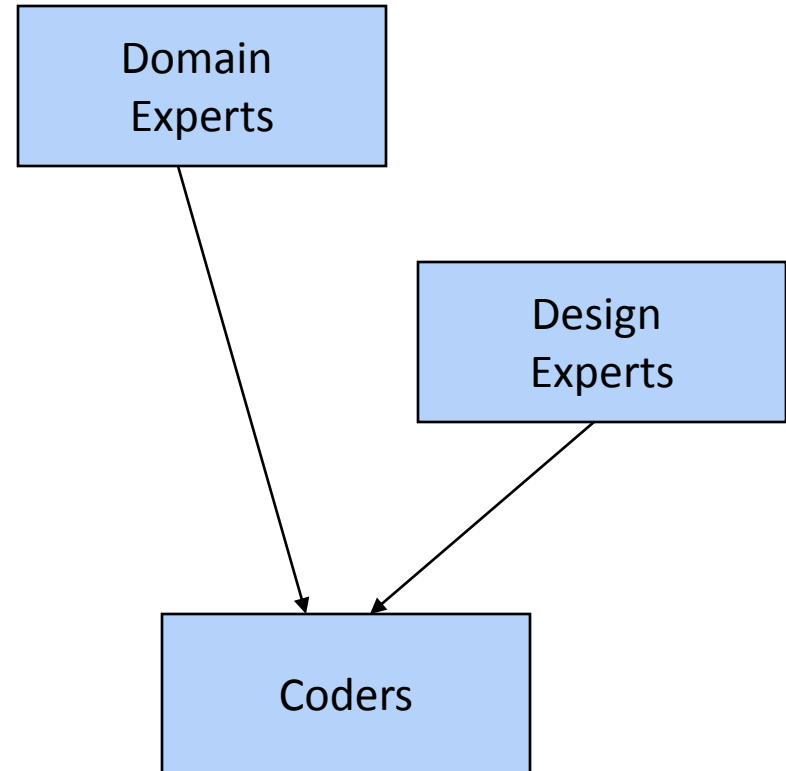


Too many things to manage simultaneously

Ground situation

- New Team
- Tight Schedules

- Crack Design Team
- Evolving Method
- Functional – Technical Gap



How to quickly make the coders productive?

Business applications

Business applications

Business-critical

- ? High performance, Availability, Maintainability
- Low algorithmic complexity, database-centric

Multitude of technologies

- ? Team of diverse skills

Long life

- ? Need to quickly respond to
 - Process change
 - Business change
 - Technology change

Large size

- ? Further exacerbates the above problems

Message round trip in a typical distributed architecture

Client

Form message

Send Message

Server

Receive Message

Process Message

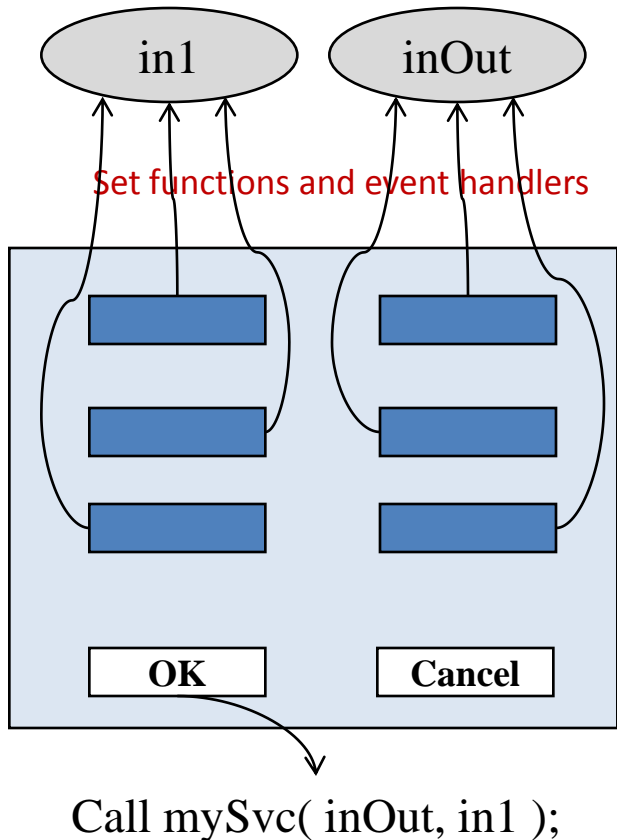
Return Message

Client

Receive Message

Display

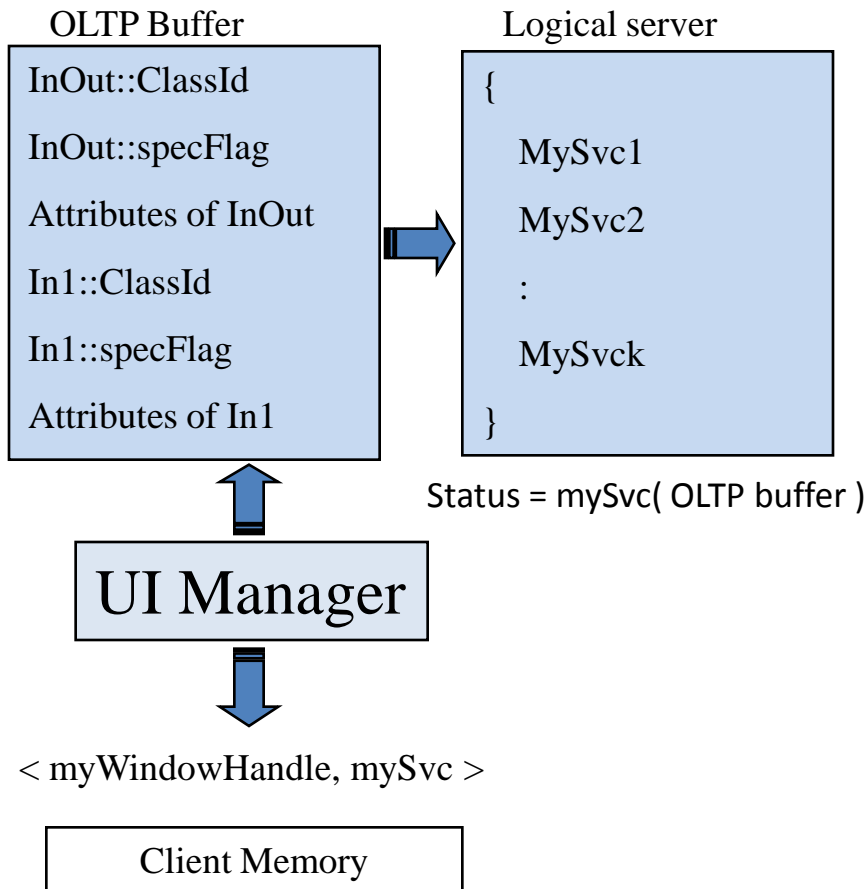
Form Message on Client



Message = < service name, input objects >

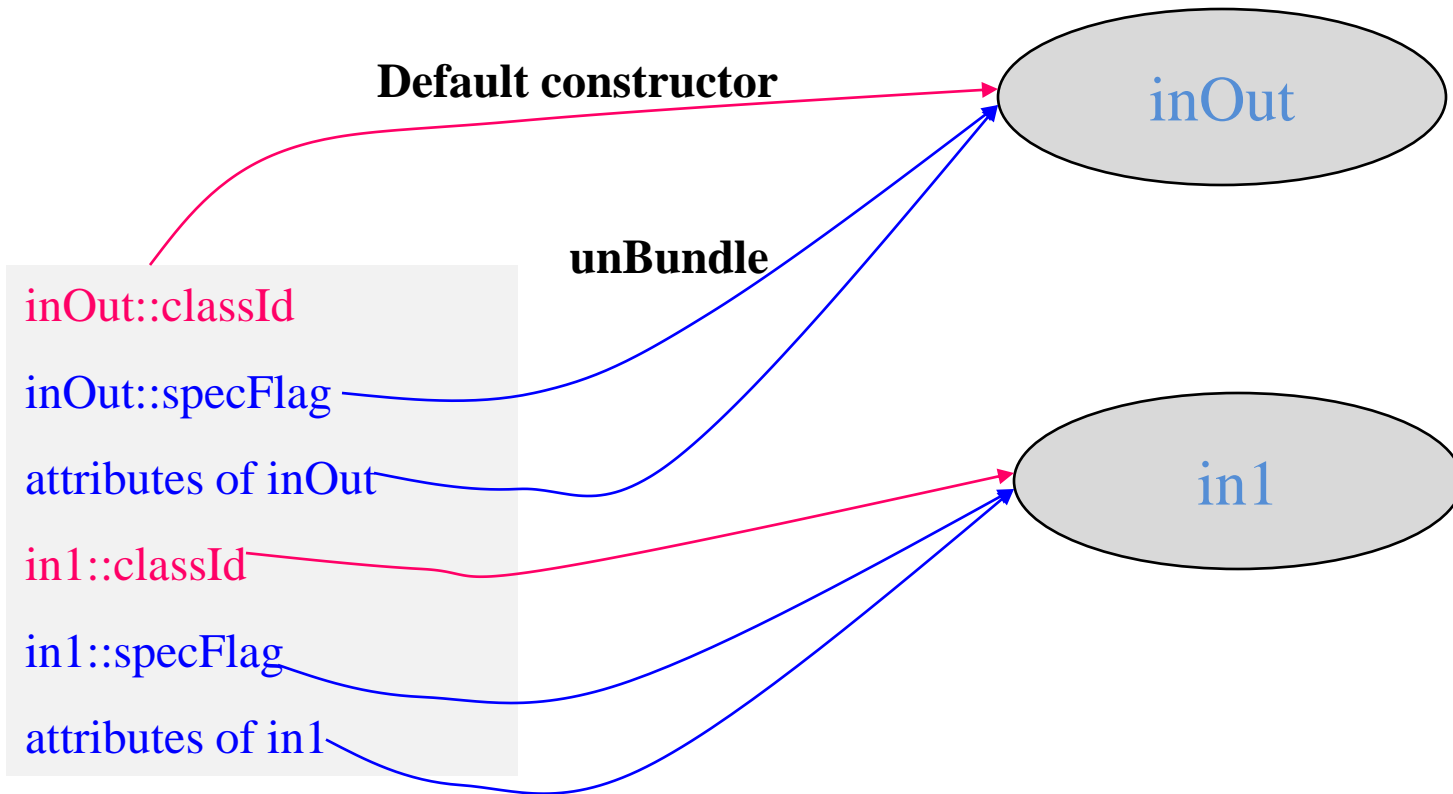
- GUI screen is one of the many ways to *form* a message
- GUI fields correspond to attributes of the input (and output) objects
- GUI buttons correspond to the service being invoked

Send message to the server for processing



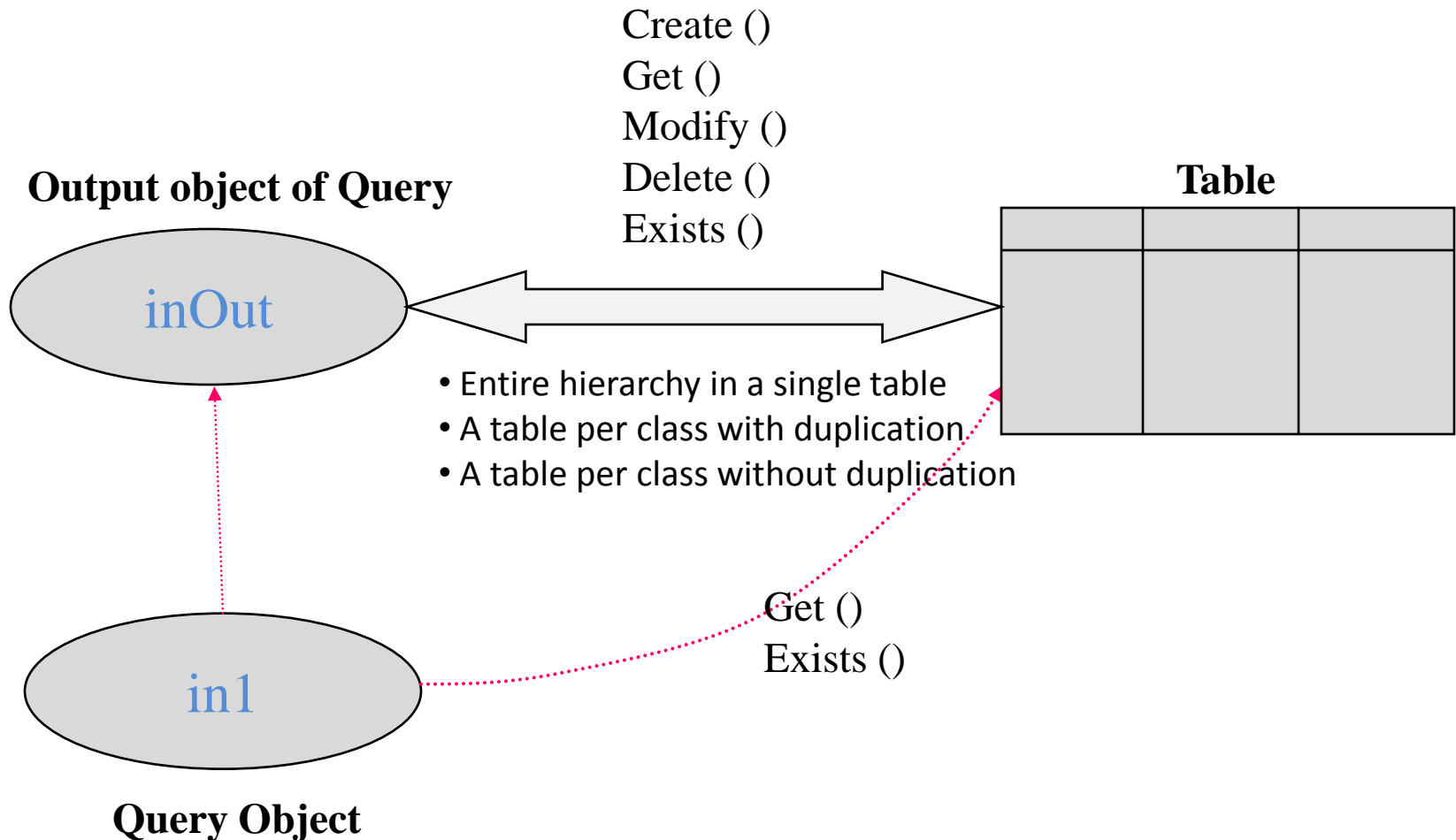
- Construct message objects from OLTP buffer
- Invoke actual function
- On Success
 - Commit Transaction
 - Put return value(s) into OLTP buffer
- On Failure
 - Abort Transaction
 - Put error context into OLTP buffer
- Return to UI manager with status

Construct message objects from OLTP buffer

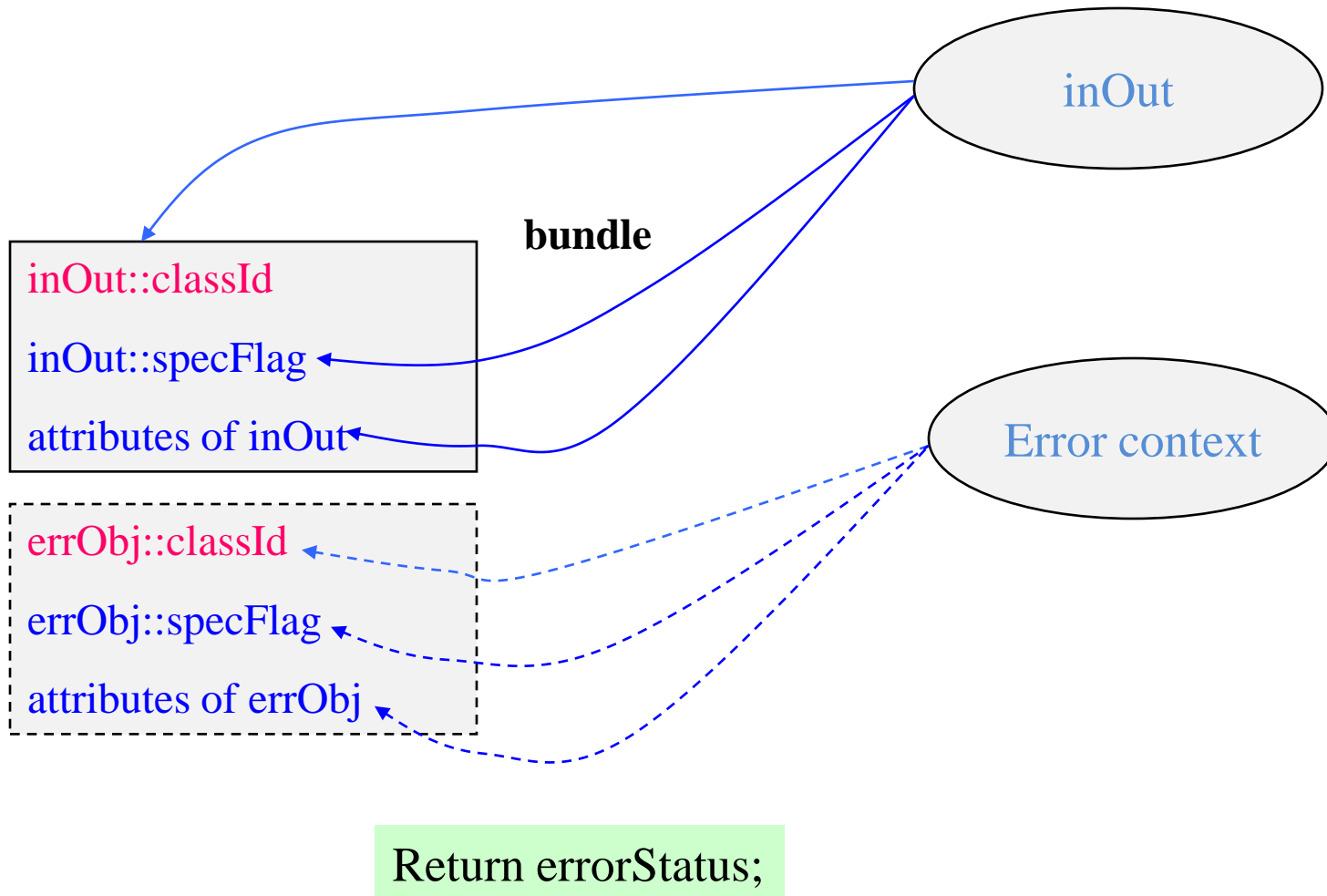


```
errorStatus = inOut->mySvc( in1 );
```

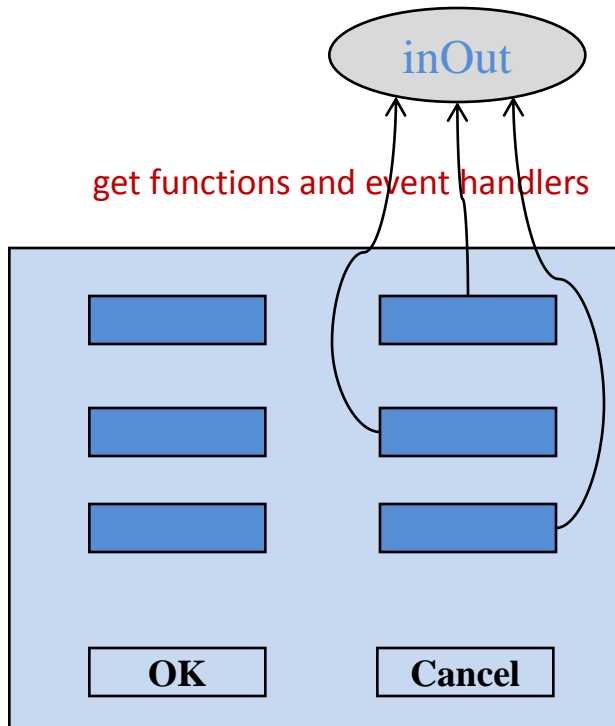
Server Side Processing



Put return value(s) into OLTP buffer



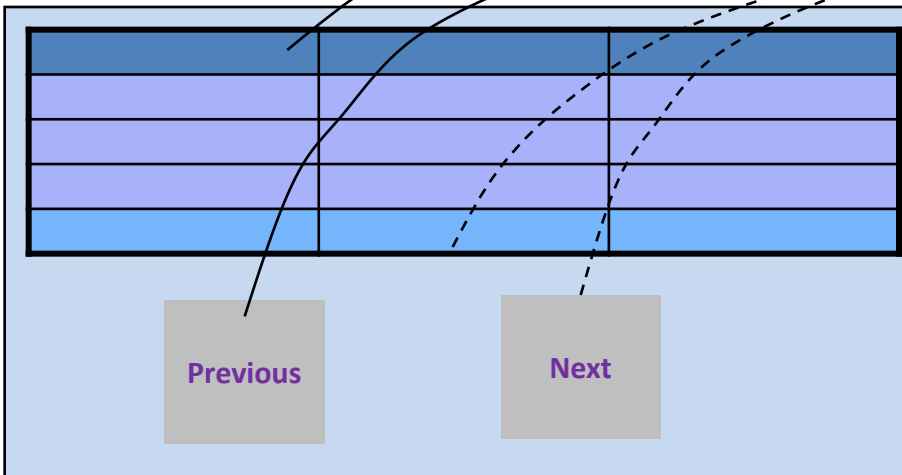
Display return value(s) onto Screen



Response = < service name, output objects >

- GUI screen is one of the many ways to *display* a response
- GUI fields correspond to attributes of the output objects

Pattern for handling of a set of objects



```
Service ( parameters, continuation key )  
{  
  :  
  db_access_query_func( continuation key )  
  :  
}
```

```
Db_access_query_func ( continuation key )  
{  
  :  
  continuation key contributes additional  
  where clause to the original SQL statement  
  :  
}
```

Interesting observations

Variety of patterns

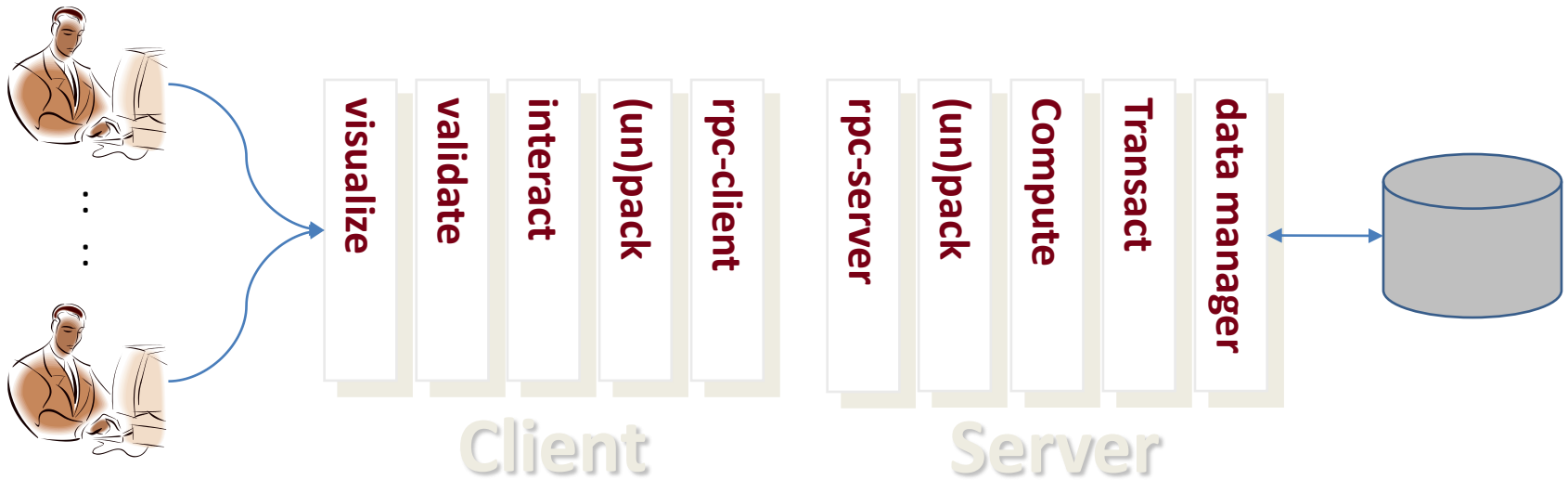
- Presentation
 - Form
 - List as a result of Search
 - Aggregation of parts
- Database access
 - **Key-based** object access (CRUD)
 - Association-based object sets
- Processing
 - Validate parameters – Raise Errors - Perform computations – Commit to database – Return
- Across architectural layers
 - Displaying large volume data in a 'paged' manner

- ☺ All patterns are data-centric
- ☺ Lends for easier specification
- ☺ Lends for framework definition
- ☺ Lends for automatic code generation for implementing the pattern
- ☺ Can be kept independent of implementation technology

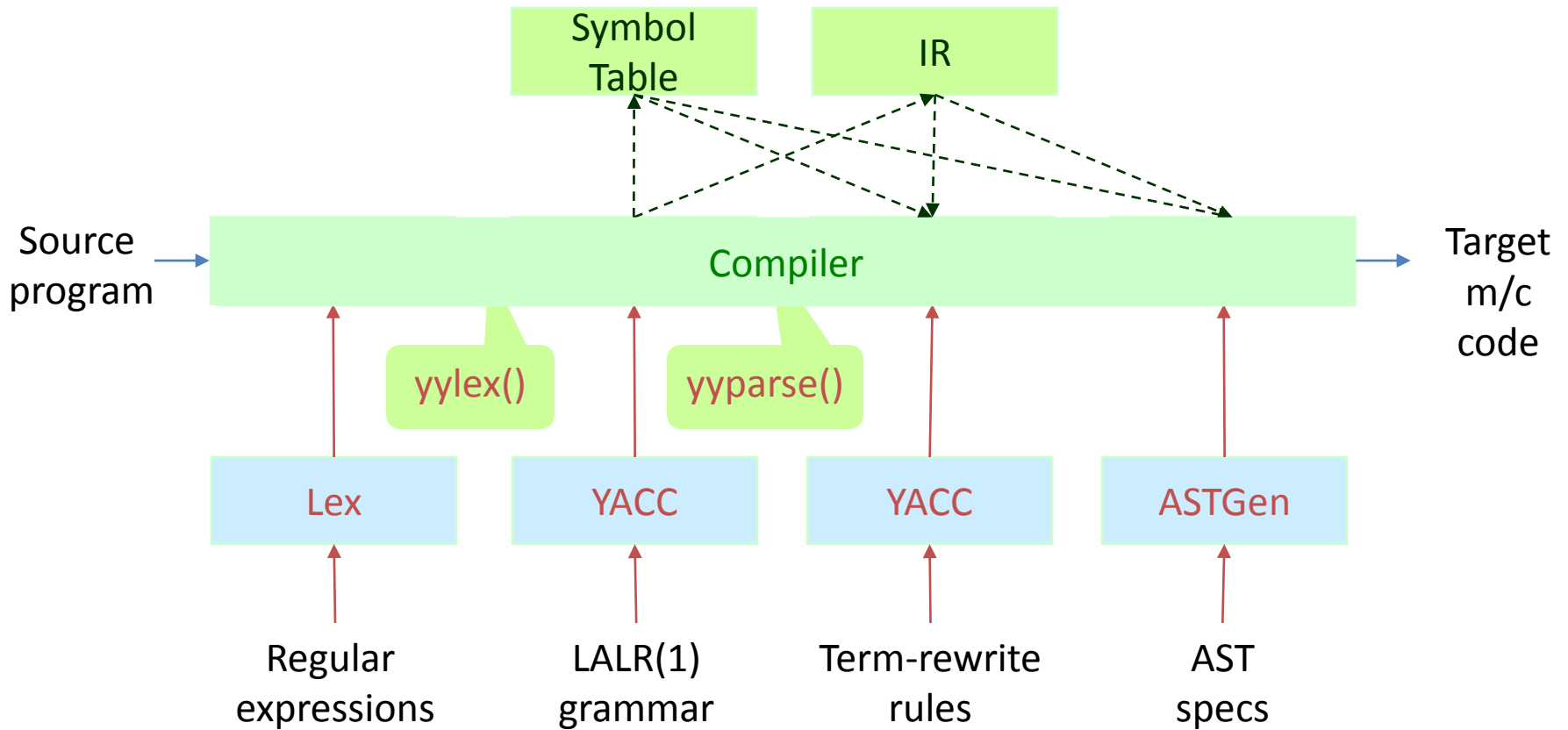
Strategies

- Audit – *What? When? How?*
- Locks – **Optimistic Vs Actual**
- Deletes – **Soft Vs Hard**
- Exception handling context – *What? How?*

It's an assembly line!



Serendipitous analogy

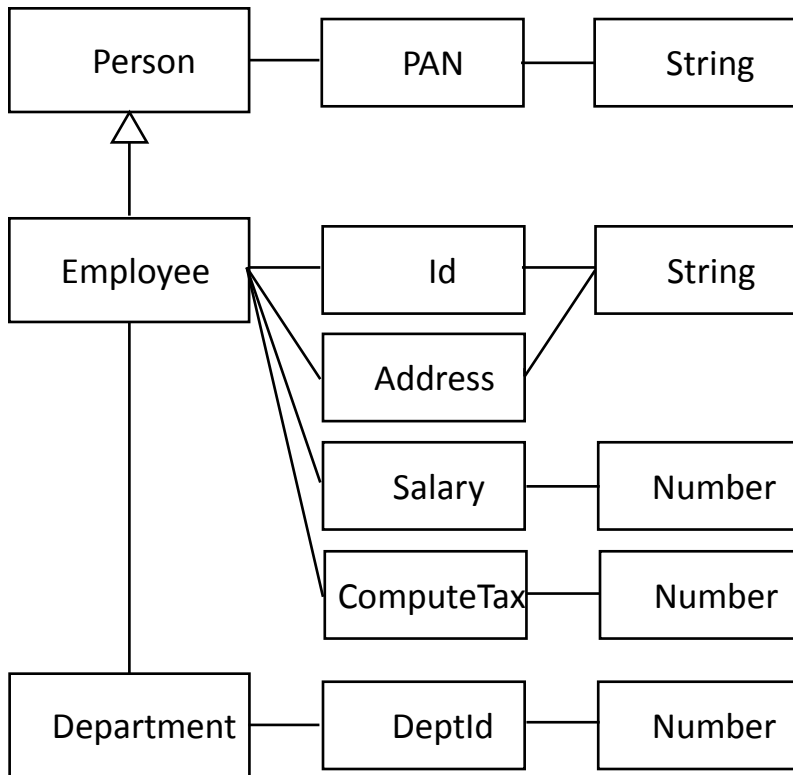


Model-driven development

Basic tenets

- Models are the primary artifacts in software development
- Separation of concerns
 - Various kinds of models to address various concerns of the system and the system development process
 - Separation between specification models and implementation models is the key emphasis of MDA
- Model-driven development process
 - A refinement based process
 - Abstract models are transformed in successive stages of refinement into more and more concrete models
 - Each refinement stage addresses a specific system concern

What is a model?



```
Class Employee extends Person
{
    String Id;
    String Address;
    Double Salary;

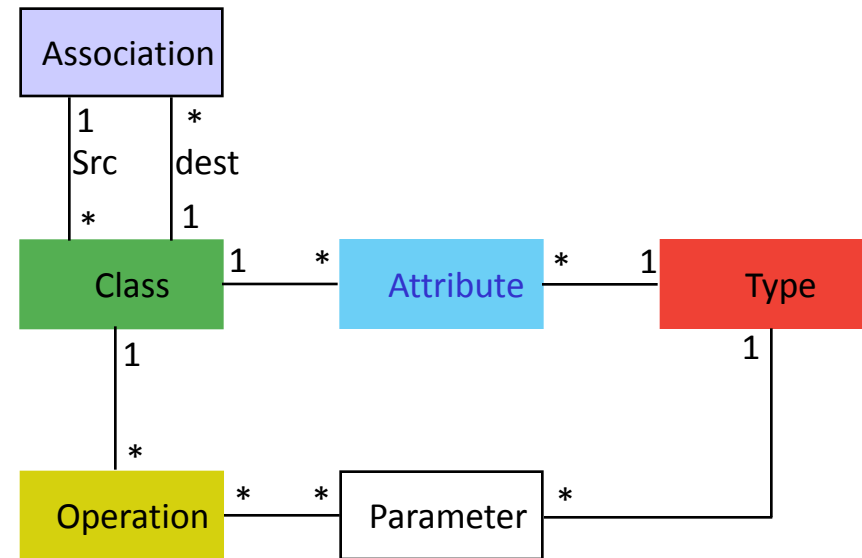
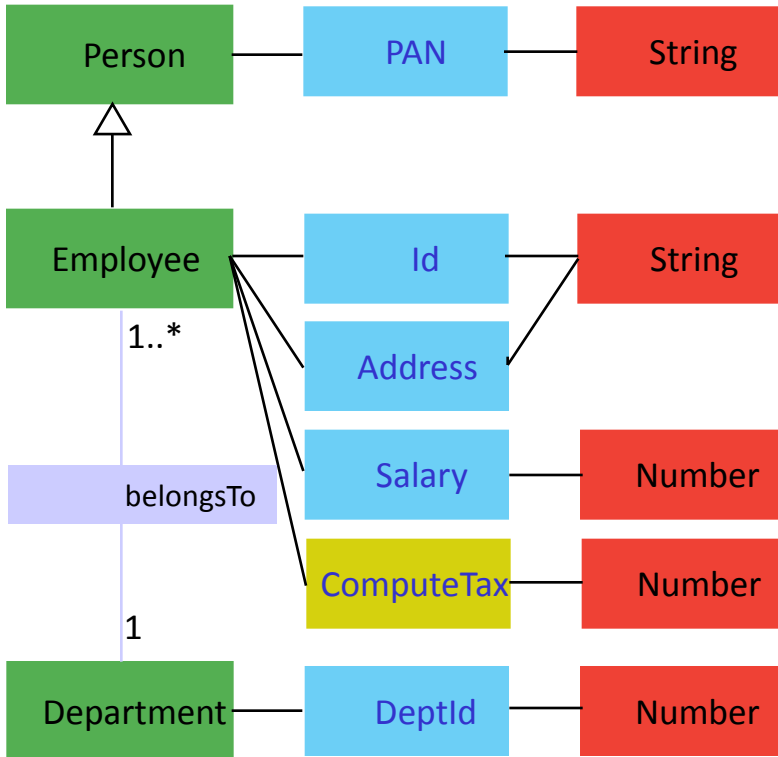
    public:
    Double ComputeTax ();
    Department getDepartment();
}
```

```
Class Employee : public Person
{
    String Id;
    String Address;
    Float Salary;
    Department *dept;

    public:
    Float ComputeTax ();
}
```

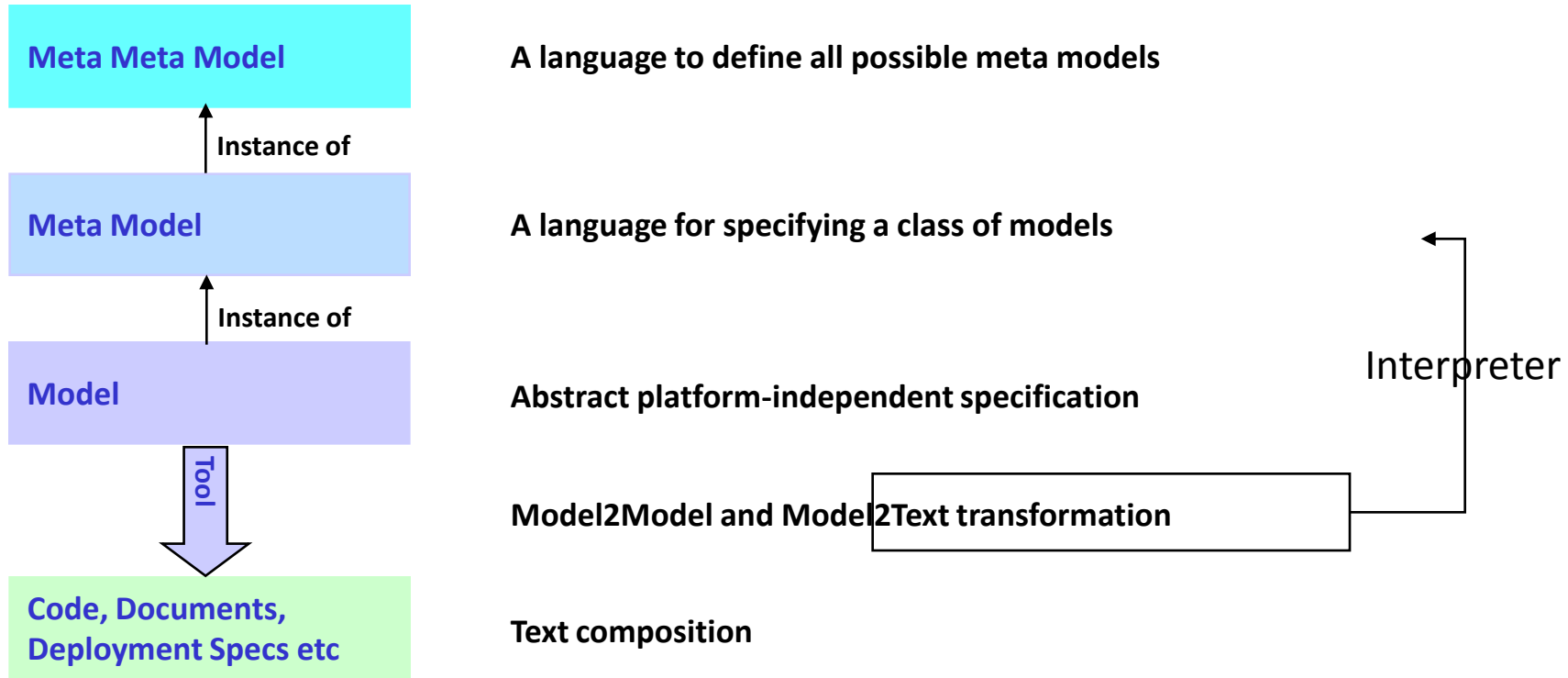
A model is an abstract representation of some concrete information

What is a meta model?



Structural correctness and consistency can be specified at meta model level

MDD basics



*Shifting the focus of software development from **how** to **what***

A code generation template

```
class Employee
{
    // Attribute declarations
    String id;
    Double salary;

    // Attribute set/get/isSpecified methods
    void Setid(String pid)
    {
        id = pid;
    }
    String Getid() {return id;}
    void Setsalary(Double psalary)
    {
        salary = psalary;
    }
    Double Getsalary() {return salary;}
}
```

```
class <c.name/>
{
    // Attribute declarations
    <for (a : Attribute | c.attribute)>
        <a.type.name/> <a.name/>;
    </for>

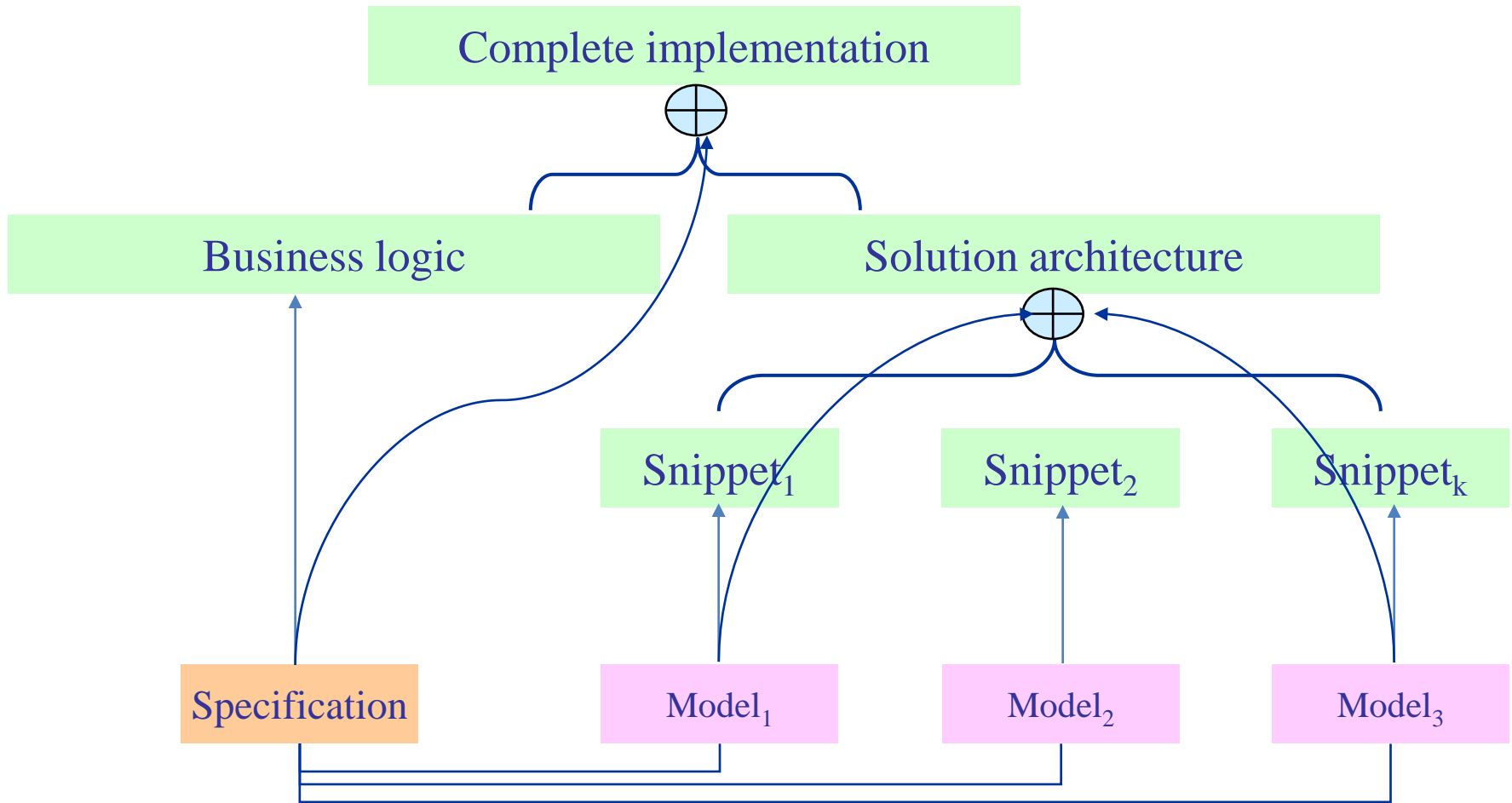
    // Attribute set/get/isSpecified methods
    <for (a : Attribute | c.attribute)>
        void Set<a.name/> (<a.type.name/>
            p<a.name/>)
        {
            <a.name/> = p<a.name/>;
        }

        <a.type.name/> Get<a.name/> () {return
            <a.name/>;}
    </for>
}
```

Of-repeating code patterns can be automatically generated

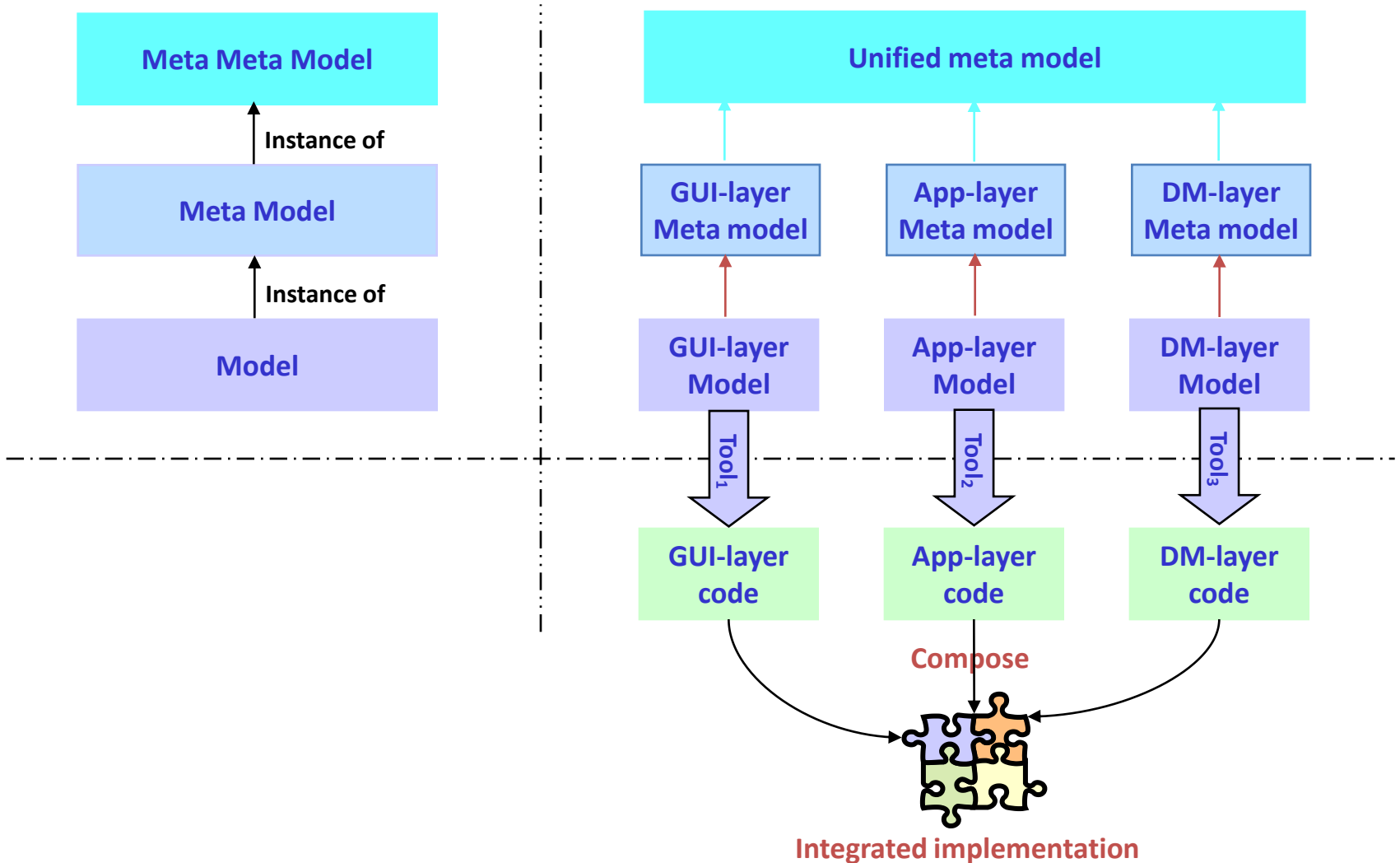
Generating business applications from models

Intuition



Generate solution architecture specific code from its declarative specifications

Architecture



Shifting the focus away from coding

Why?

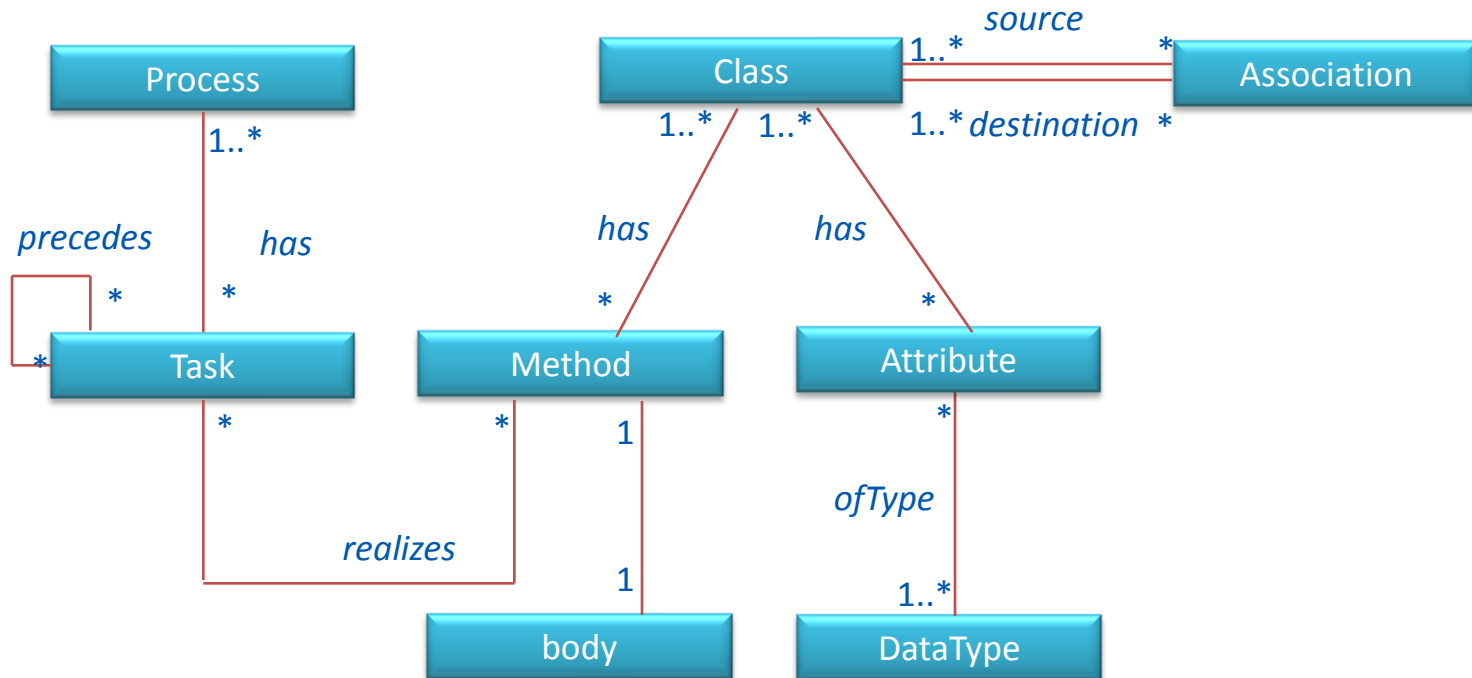
- Helps prevention and early detection of errors
- Can be automatically translated to code and other SDLC artefacts
- Can be retargeted to different technology platforms

How?

- Various code patterns recur in an implementation
- Capture a pattern using models and / or HLLs
- Implement a pattern-specific code generator to generate the corresponding code snippet
- Integrate code snippets in a consistent manner through an implementation architecture

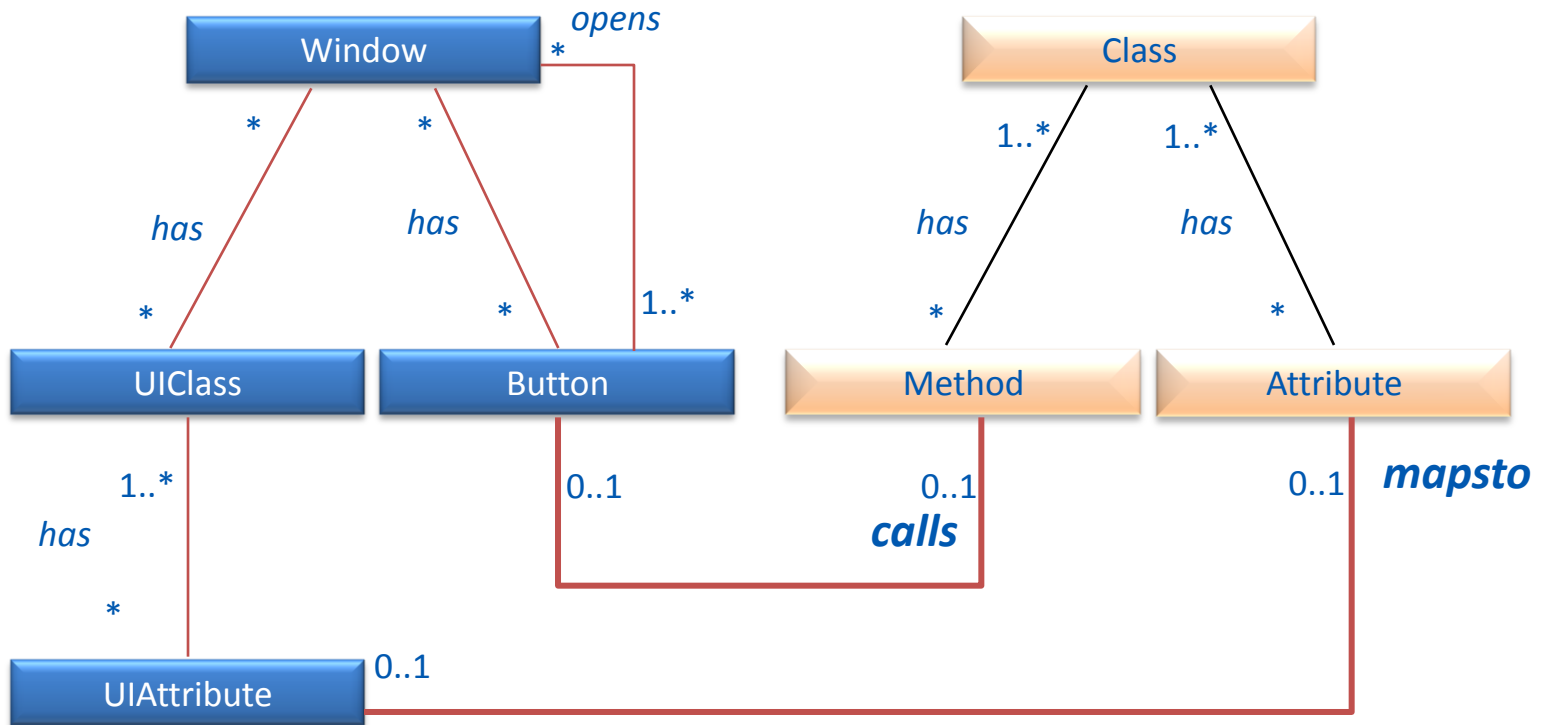
Illustrative example:
Client Server application

Application layer meta model

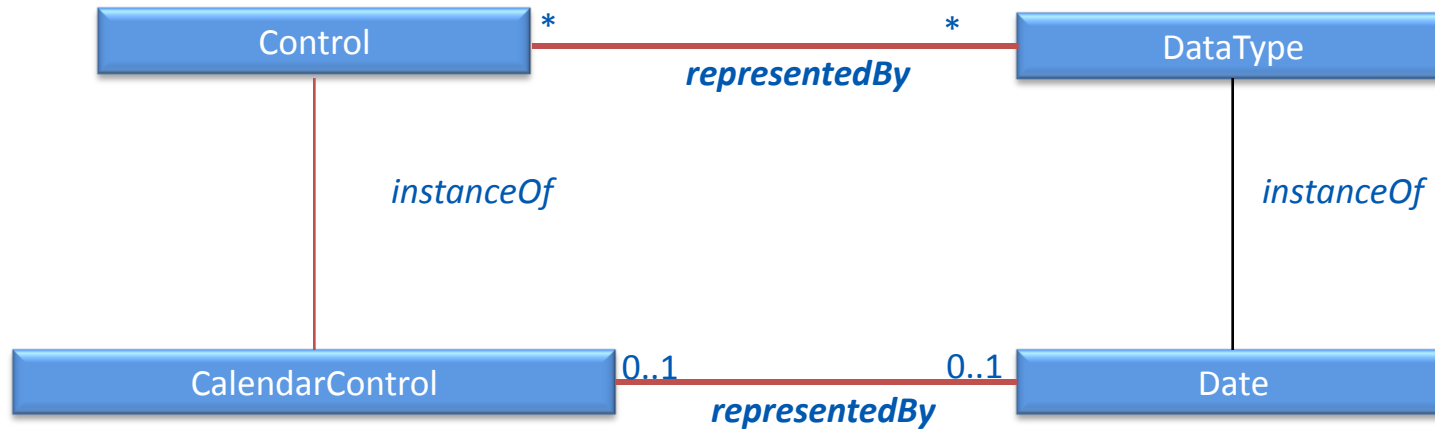


Behaviour specified using a domain specific language

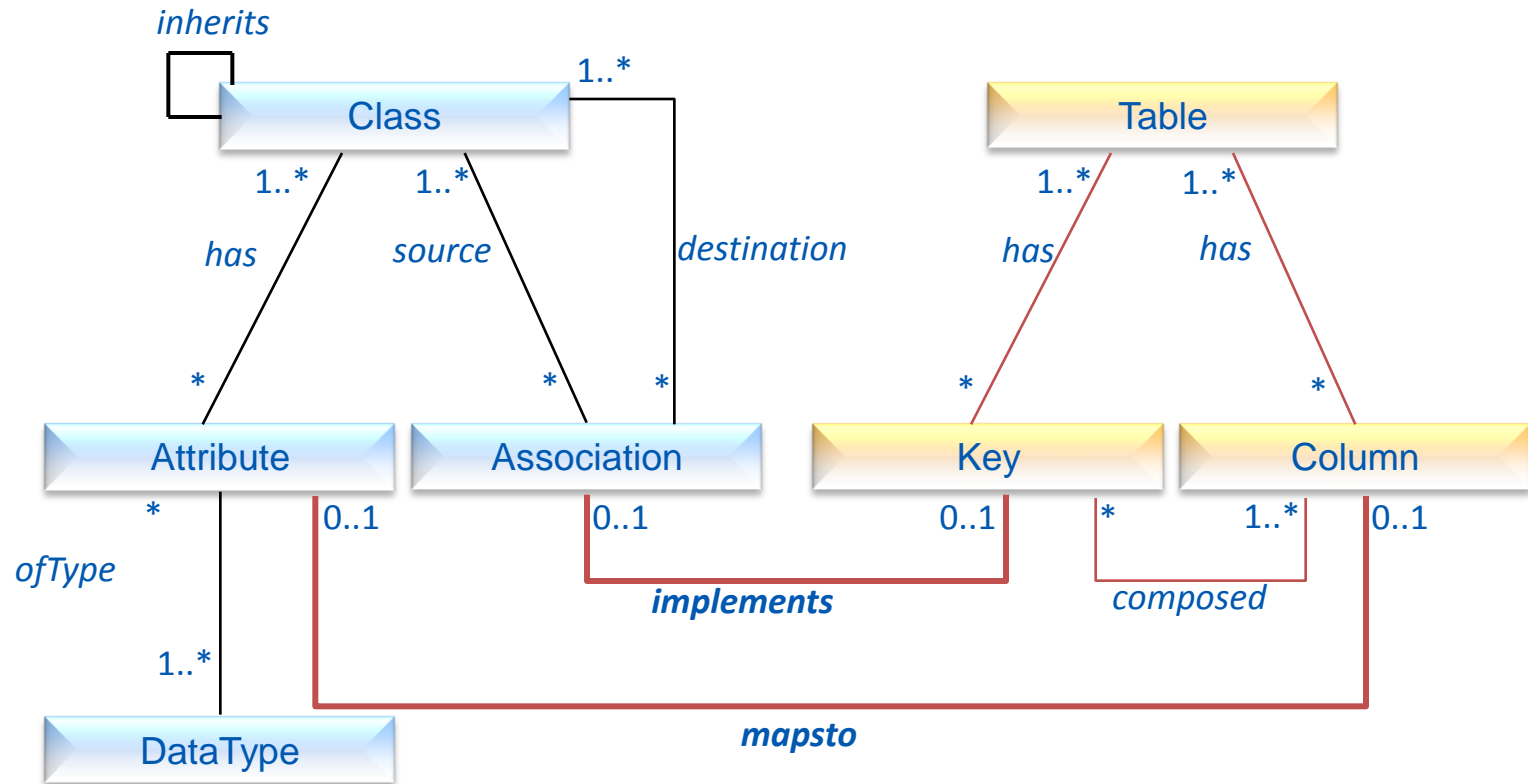
UI layer meta model



GUI standards meta model

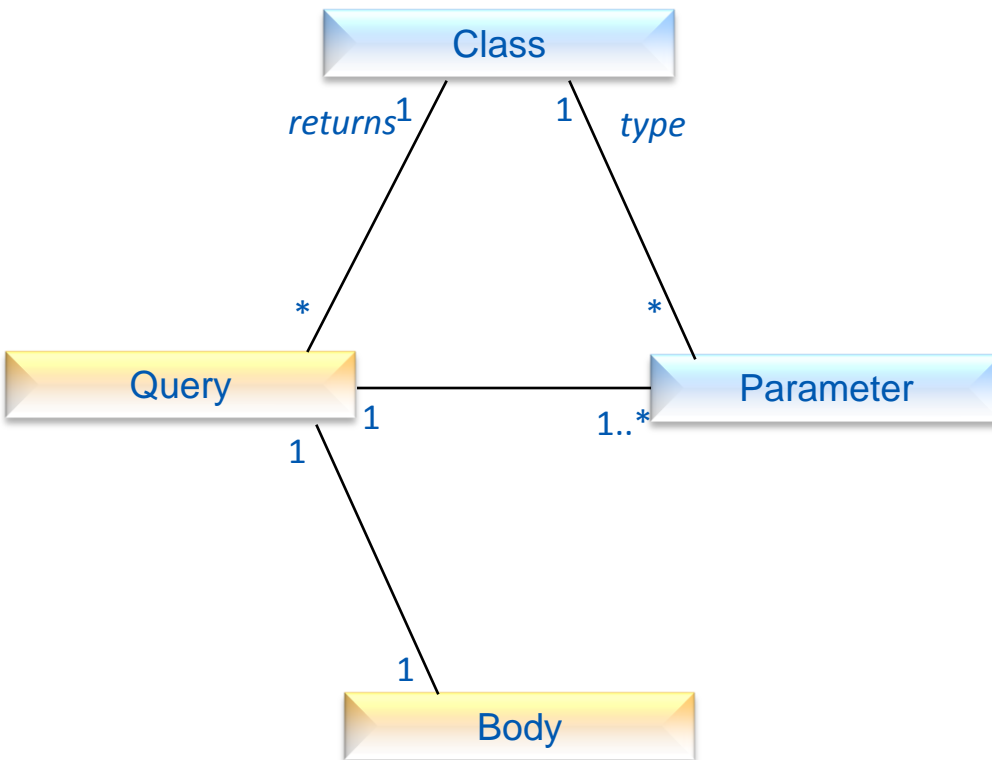


Database layer meta model



Query meta model

- Body is specified using extended SQL syntax
- Bindings to query parameters e.g.



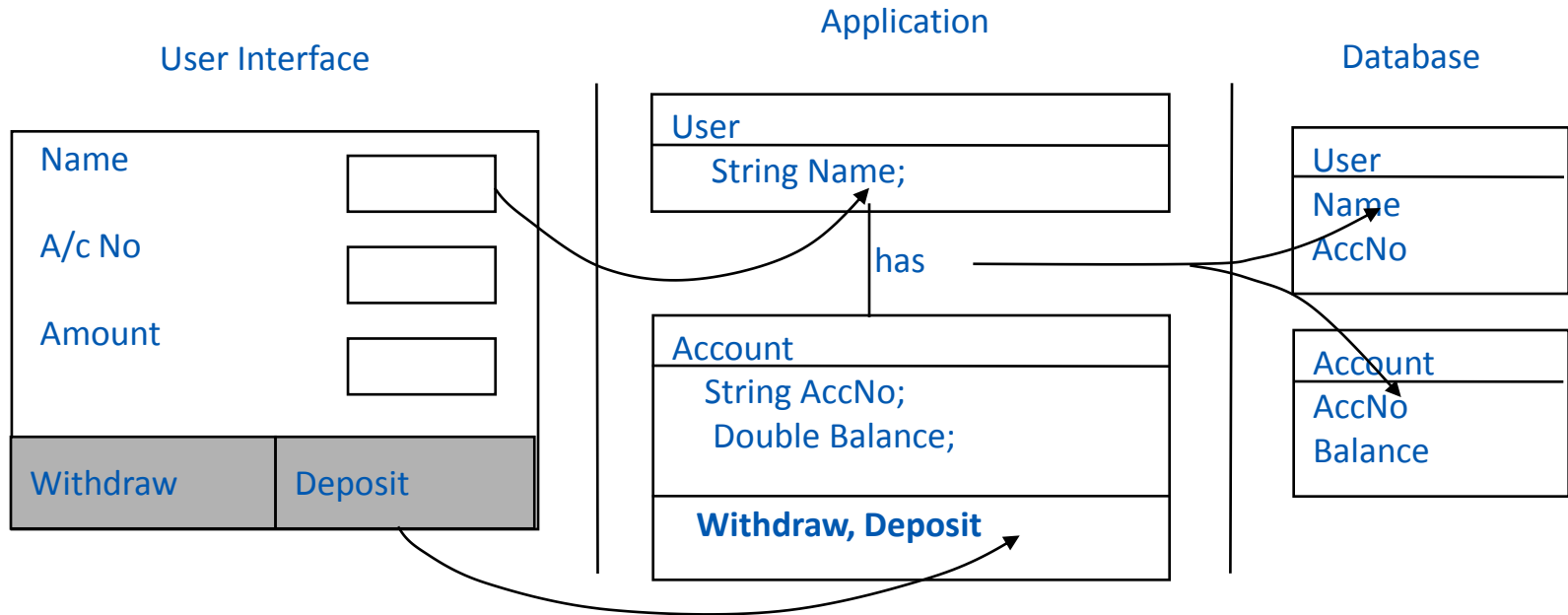
GetCustomerList(Bank b) :
Customer MULTIROW

```
SELECT name FROM customer  
WHERE bankId = :1.id
```

Some inter model consistency checks

- UI should allow specification of all **Tasks** in the business process and be consistent with *precedes* relationship between **Tasks**
- UI should display data that is consistent with respect to the **Parameters** being passed to the **Operations** invoked from the **Window**
- DB layer should ensure that *implements* association is implemented in a consistent manner

Integration across architectural layers



Generating complete application

Server side

- Class definition
 - Attribute handling logic
 - Getter/Setter
 - Default constructor and copy constructor
 - Persistence
 - Primary- and Alternate-key based CRUDE methods
 - Primary- and foreign-key based association handling
 - Design strategies
 - Soft locking, Soft delete, Auditing, Error handling, Trace logs
- Application services
- Object-wrapper for non primary-key based data accesses

Server side

- Component deployment
 - Bean class, Remote interface and Deployment descriptor
- Testing support
 - Unit test harness for a service,
 - Test data generation
- Build support
 - Makefiles

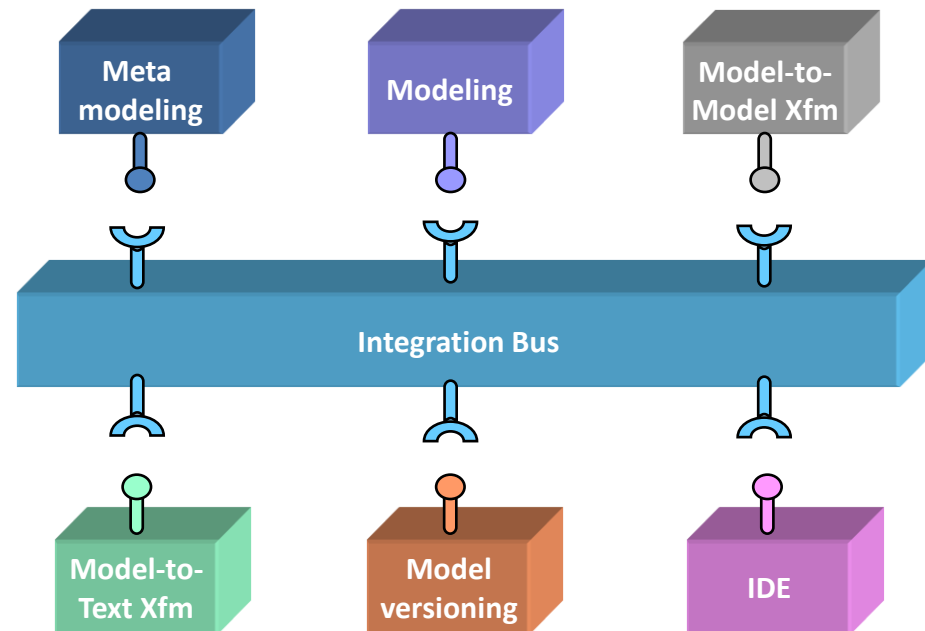
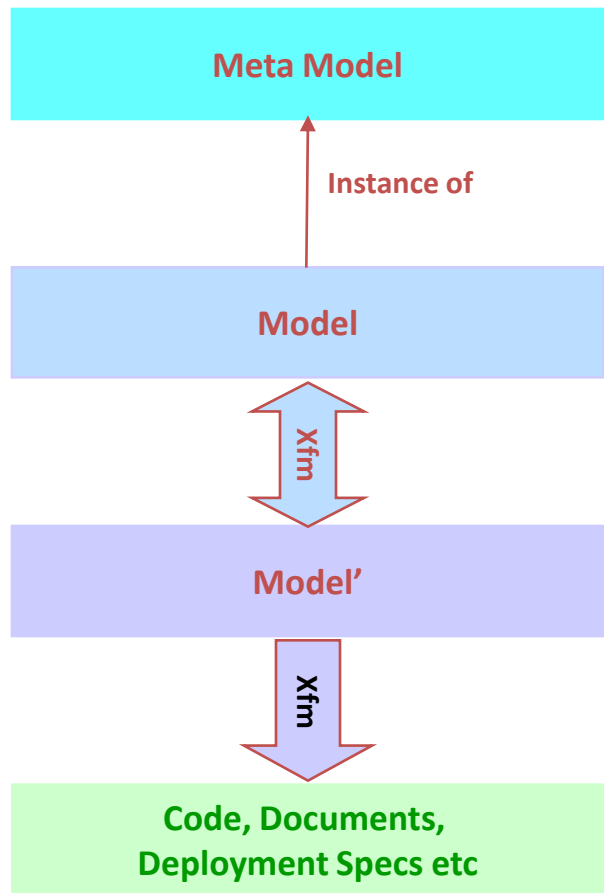
Client side

- Screen definition
 - Layout
 - Server-side interface
 - Placeholder for Event logic
- Testing support
 - Look-n-feel
 - Interaction with server-side

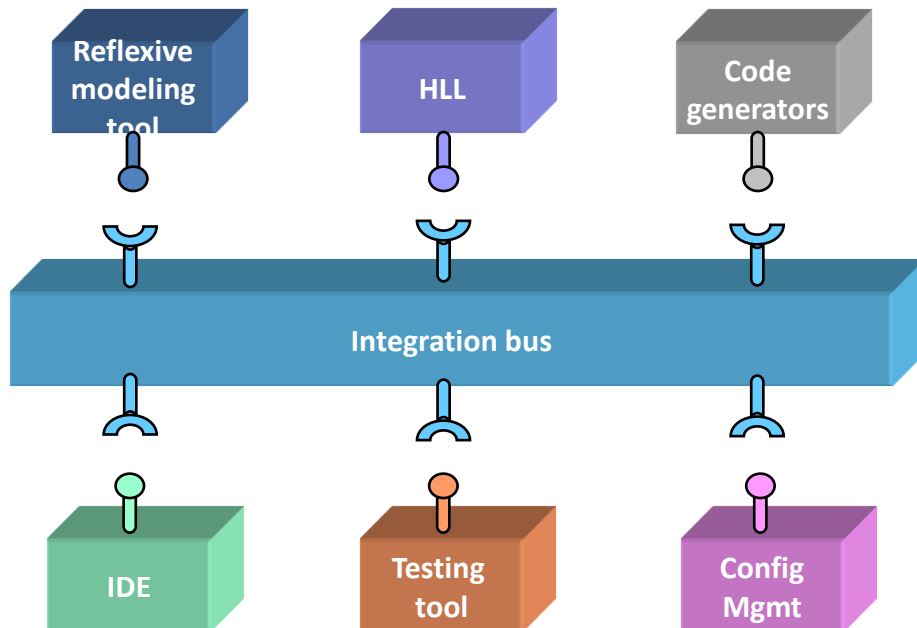
Guaranteed integration of the various architectural layers

MDD toolset

Requirements of MDD toolset

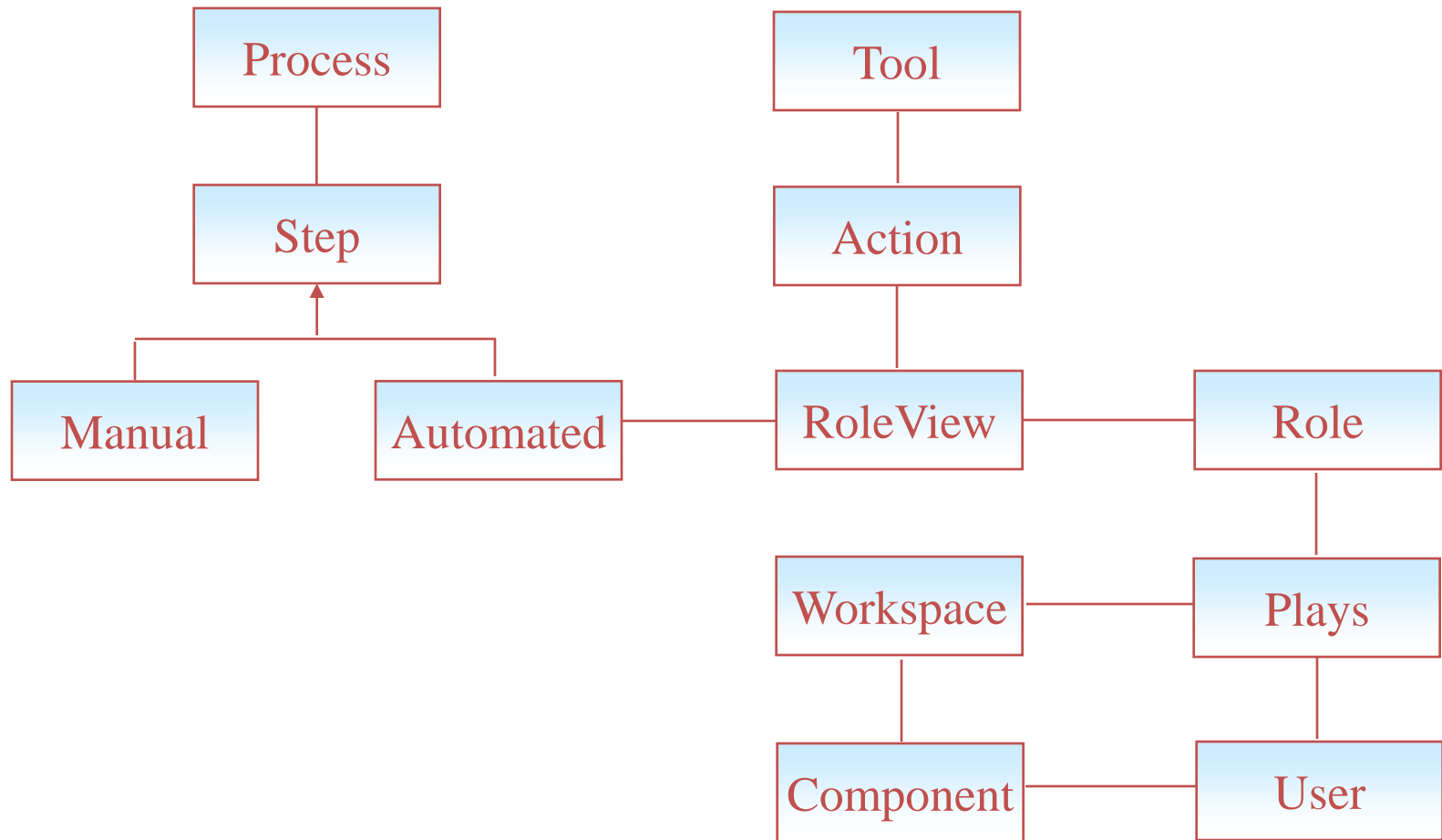


Our MDD toolset

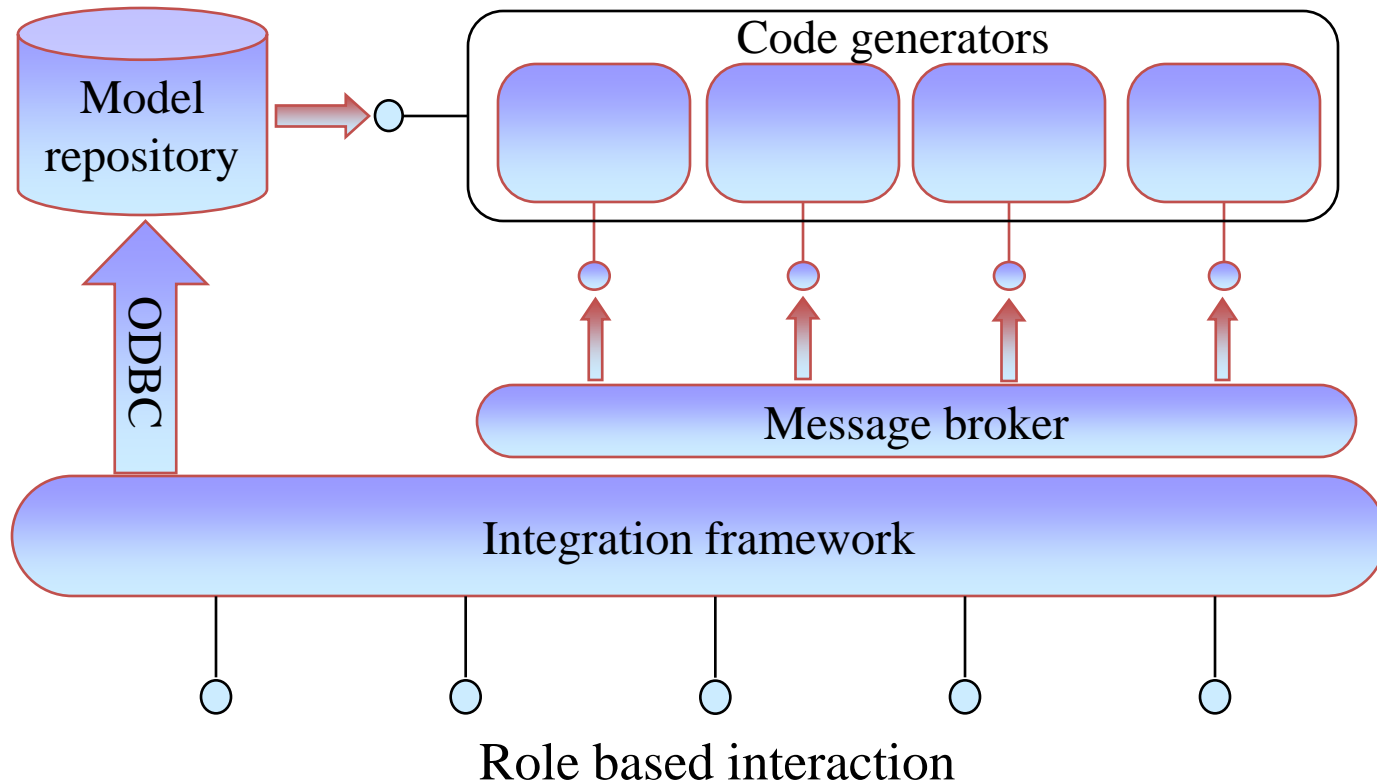


- Reflexive modeling tool
 - Meta modeling as well as Modeling tool
 - Extensible
 - Architected for distributed workspaces
- Code generators
 - Model-to-model and Model-to-text transformers
 - Extensible
- HLL
 - Simplified Business Oriented OO language
 - Translator(s) to popular OO language(s)
- IDE
 - Uniform Role-based interaction

Meta model for IDE



IDE - Architecture

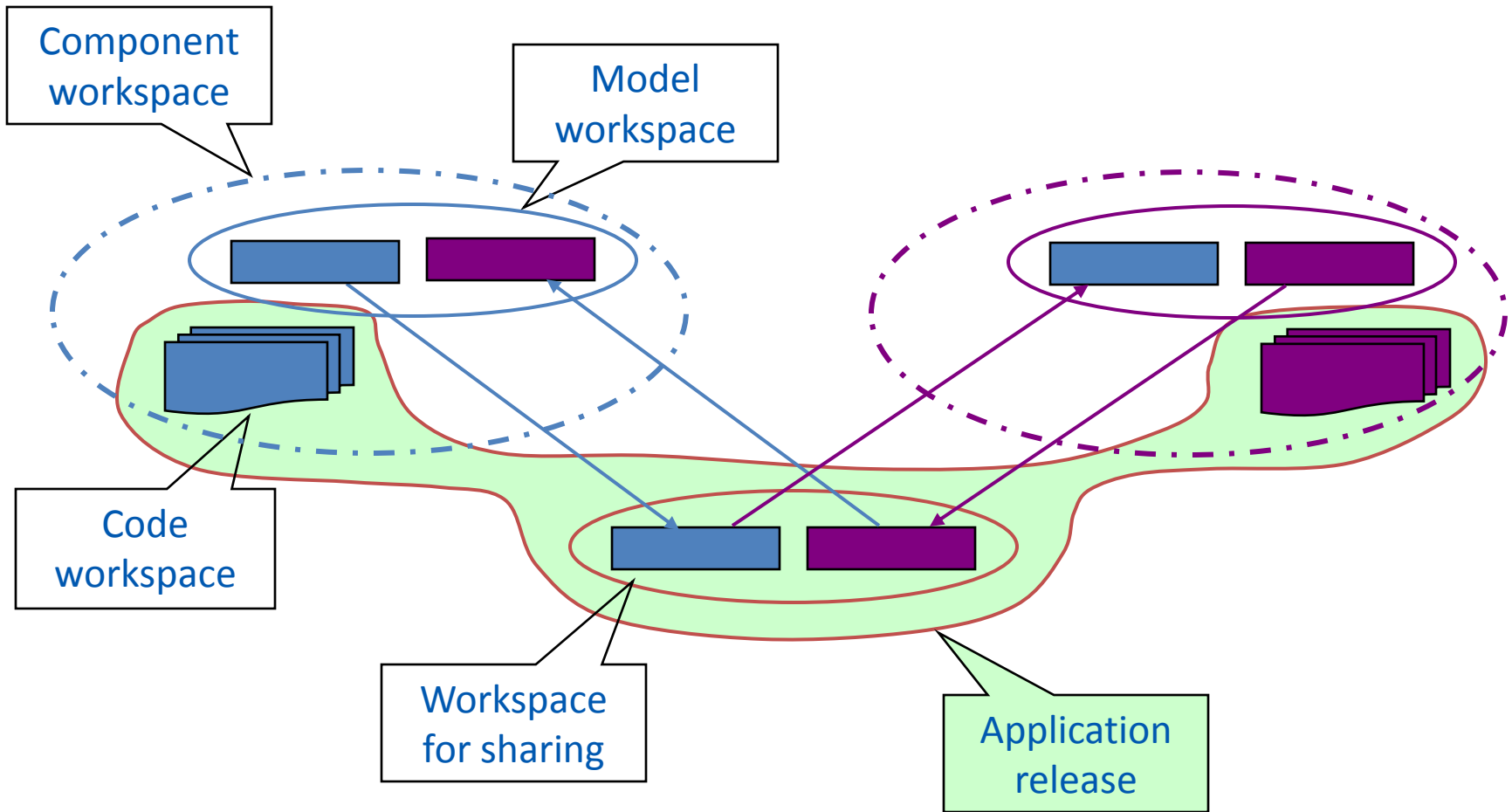


A method for disciplined use of tools

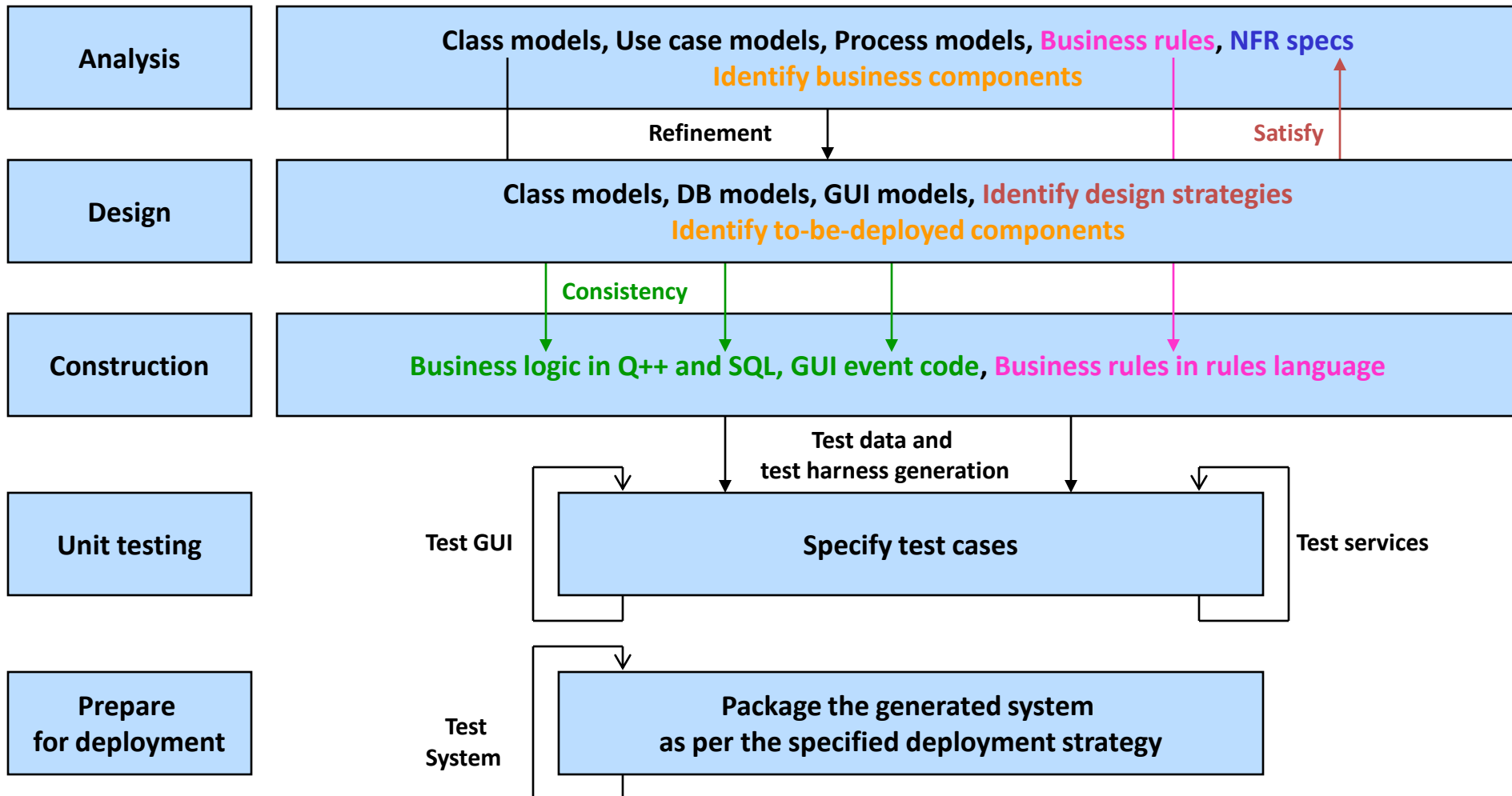
Component abstraction

- Application as a set of related components
- Component
 - Implementation : owned functionality
 - Interface : exposed Services
 - Dependence : used Interfaces
- Unit of development
 - Implementation : Models and files created
 - Interface : Model elements and DLLs exposed (Visibility)
 - Dependence : Model elements used and DLLs linked to (Completeness)
- Unit of deployment
 - Implementation : Jar file of owned functionality
 - Interface : Bean class and remote interface (External invocability)
 - Dependence : Jar files used (Peer-to-peer communication)
- By default, unit of deployment is same as the unit of development

Component based development



Well-defined method ensures repeatability



Model based code generation

- Requirements
- Analysis phase: [use cases](#), [process diagram](#), [analysis document](#)
- Design phase: [class diagram](#), [sequence diagram](#), [design document](#)
- Construction phase
 - Code generated from model
 - Data manager layer: [DM](#) and [DDL](#)
 - [Class definition](#)
 - Screens: [JSP](#)
 - [Q++](#) and [generated Java](#)
 - [Query](#) and [generated Java class](#)
 - [Service wrapper](#)
- Testing phase
 - Unit testing harness: input data and driver
- Deployment
 - Remote interface, it's implementor class and deployment descriptor

Generating complete application from
its model

Enthusiastic adoption by large projects

Project	Specifications			Generated code		Technology Platforms
	Domain model (no of classes / screens)	Size (kloc)	Kind	Size (kloc)	Kind	
Streight Through Processing system	334 / 0	183	Business logic, Business rules, Queries	3271	Application layer, Database layer, Architectural glue	IBM S/390, Sun Solaris, Win NT, C++, Java, ICS, MQ Series, WebSphere, DB2
Negotiated dealing system	303 / 0	46	Business logic, Queries	627	Application layer, Database layer, Architectural glue	IBM S/390, Win NT, C++, CICS, MQ Series, COM+, DB2
Distributor management system	250 / 213	380	Business logic, Business rules, Queries, GUI	2670	Application layer, Database layer, GUI layer, Architectural glue	HP-UX, Java, JSP, Weblogic, Oracle, EJB
Insurance system	105 / 0	357	Business logic, Business rules, Queries	2700	Application layer, Database layer, Architectural glue	IBM S/390, Sun Solaris, C++, Java, CICS, DB2, CORBA

Experience and lessons learnt

Over the past 14 years have delivered 70+ large business applications on a variety of technology platforms

Technical

- Increased productivity, uniformly high quality and platform independence
- Limited success on reuse front and that too in very recent past
- MOF and MTT sufficed majority of our common needs
- Need ready-to-use models of various NFRs

Managerial

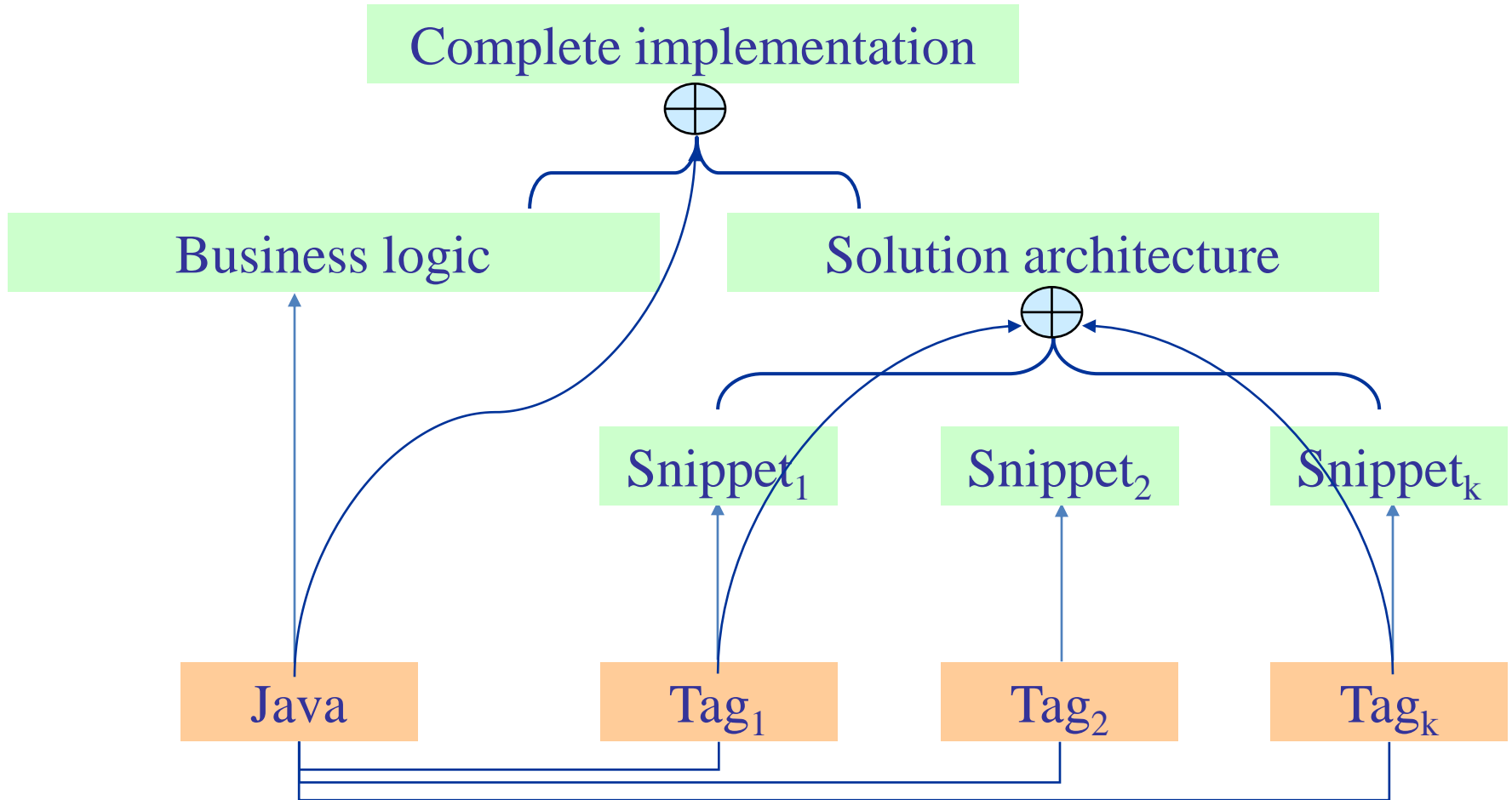
- Have found it easy to convince product-owners and managers of large projects
- OMG standardization process has helped
- Large cycle-times for changes and steep learning curve are still the major hurdles
- Small projects still prefer code-centric development

Social

- MDD not yet established as a 'domain' – skills are still platform-specific ☹️
- Needs greater involvement of academia
- Community focus needs to expand beyond 'D' to Analysis, Testing, Integration, Maintenance and Re-engineering

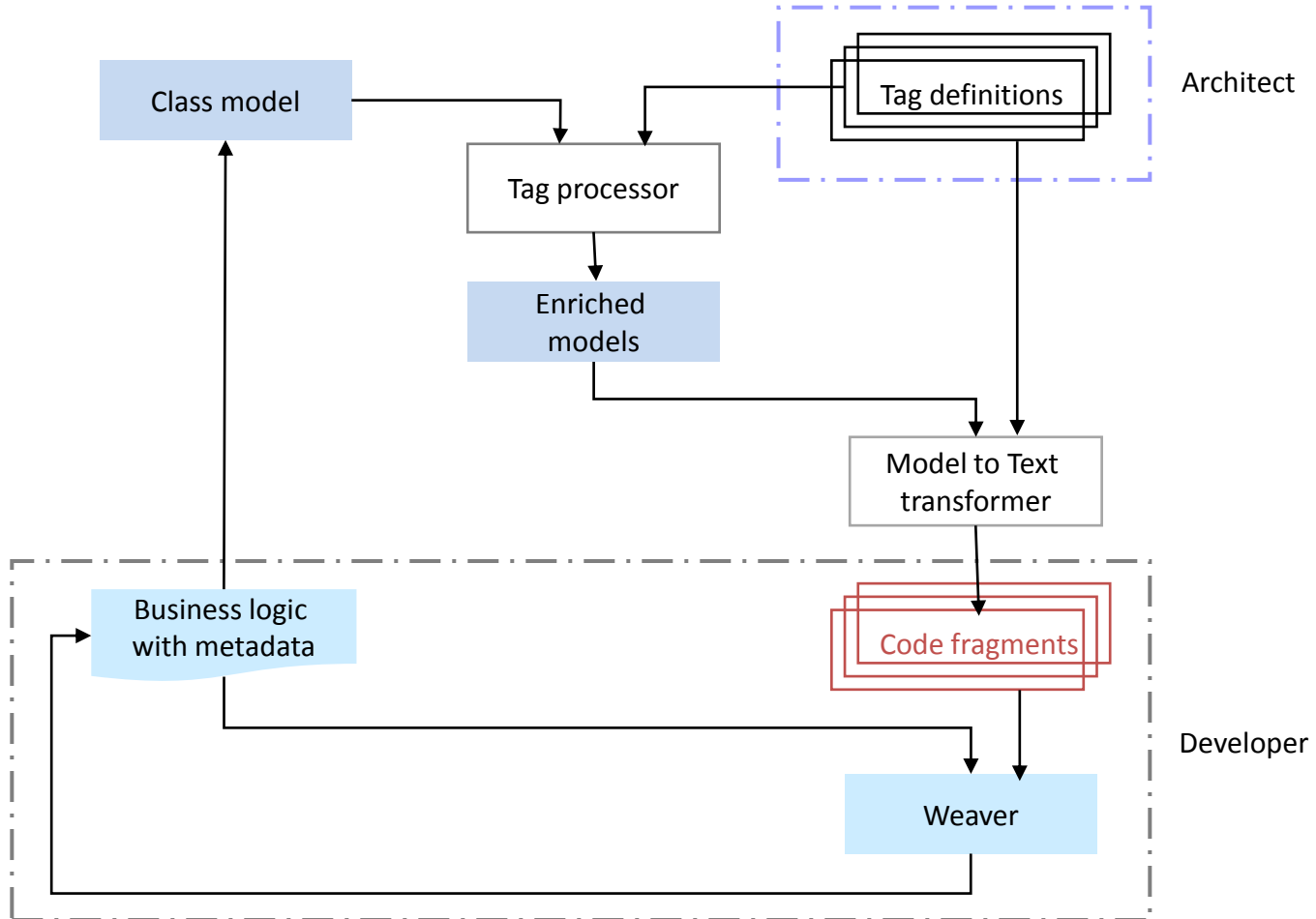
‘Code is model’ approach

Intuition

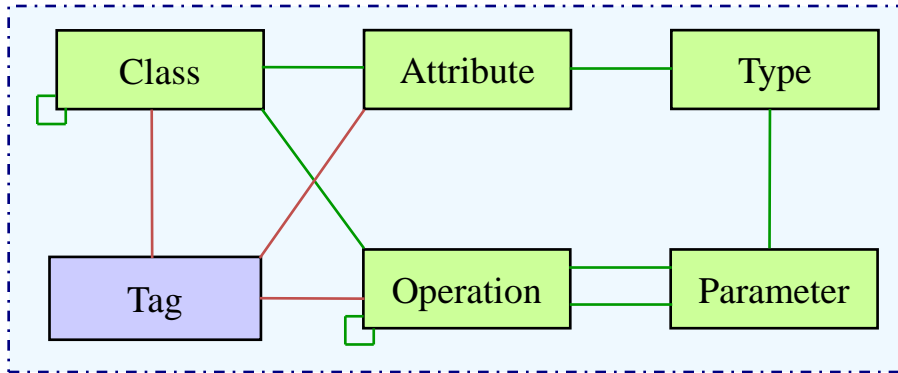


Generate solution architecture from its declarative specification

Code generation



Tag abstraction



Encapsulates non-functional characteristics

- Strategies: Persistence, Auditing, Archival, Logging, Error handling etc
- Platforms: RDBMSs, Presentation managers, EJB servers etc
- Architectures: JMI, EJB etc

Unifying abstraction to specify

- Data contribution i.e. attributes
- Method contribution i.e. code fragments
- Composition of code fragments

Specified using

- Extensible Java meta data
- Model-to-text transformation language

Enables

- Realization of a solution architecture as a composition of tags
- A repository of reusable tags
- J2EE development as a component factory

Tag is manifestation of building block abstraction in code

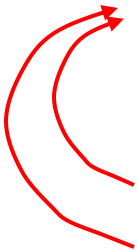
Operational view

Tool building

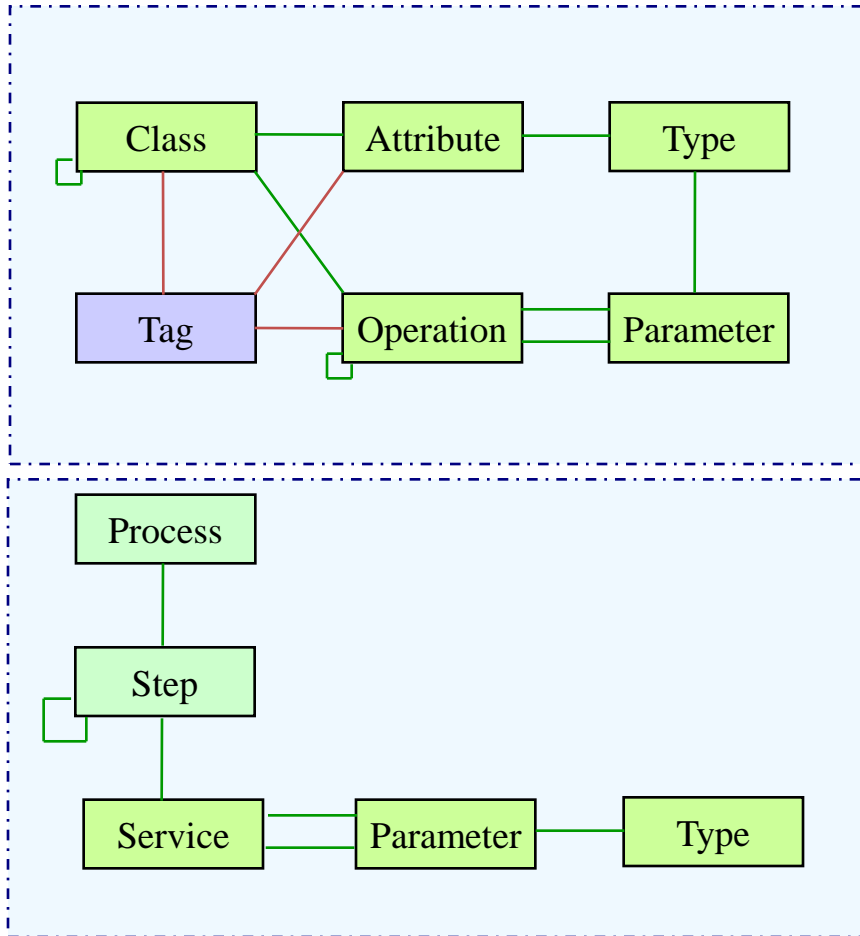
- Define solution architecture
- Identify tags to support realization of the solution architecture
- Convert code snippets into tag definitions
- Test tags for correctness w.r.t. solution architecture
- Release tags and their processor as a plugin

Tool usage

- Write business logic in Java
- Annotate appropriately with tags
- Generate complete class implementation
- Unit Test and release tested class
- System build + test
- Deploy



Extensibility

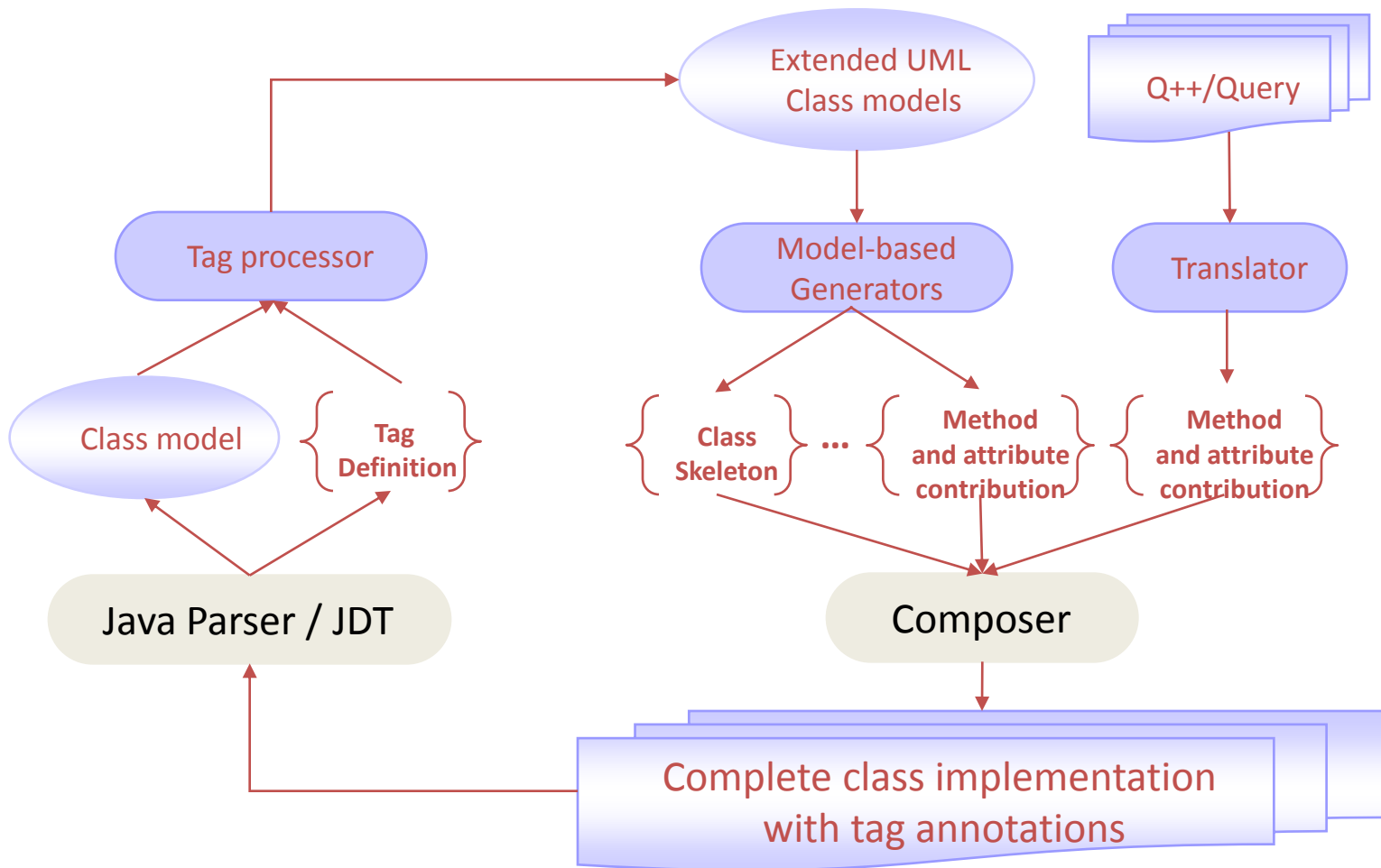


All *services* that are part of *secure process* steps need to be audited in a different manner

Purpose-specific meta models can be defined

- Enhanced code generation
- Sophisticated weaving

Interoperable with model-centric approach

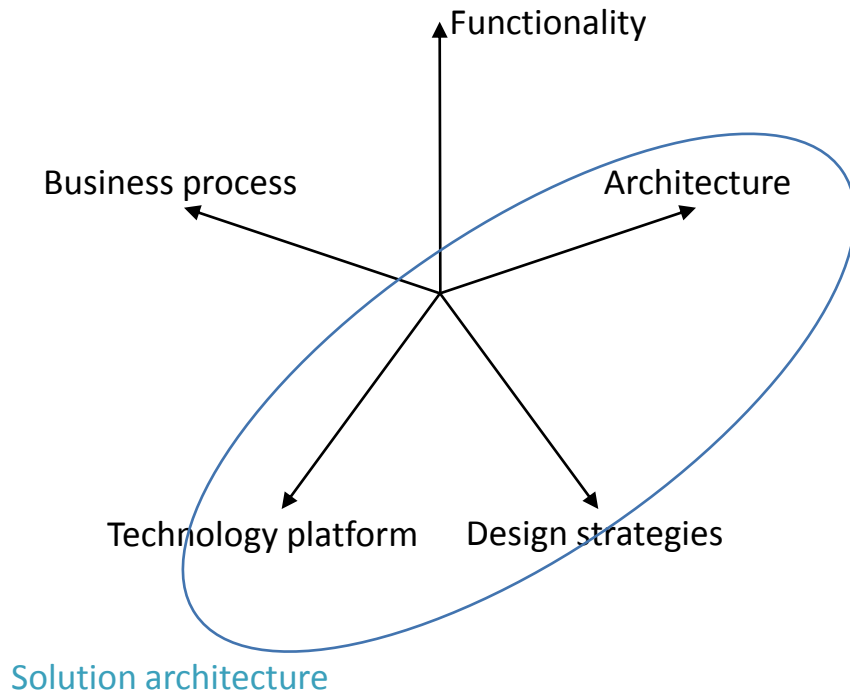


Benefits and pain points

- 😊 Uses industry standard technologies
- 😊 Quick turn-around time
- 😊 No runtime footprint
- 😊 No vendor lock-in
- 😊 Generated application can be maintained using only JDK
- 😊 Lightweight loosely coupled development process
- 😊 Quick development of purpose-specific tools
- 😊 Use of toolset can be easily switched off any time
- 😊 Leads to a repository of reusable software artefacts
- 😞 Limitations due to restricted meta model

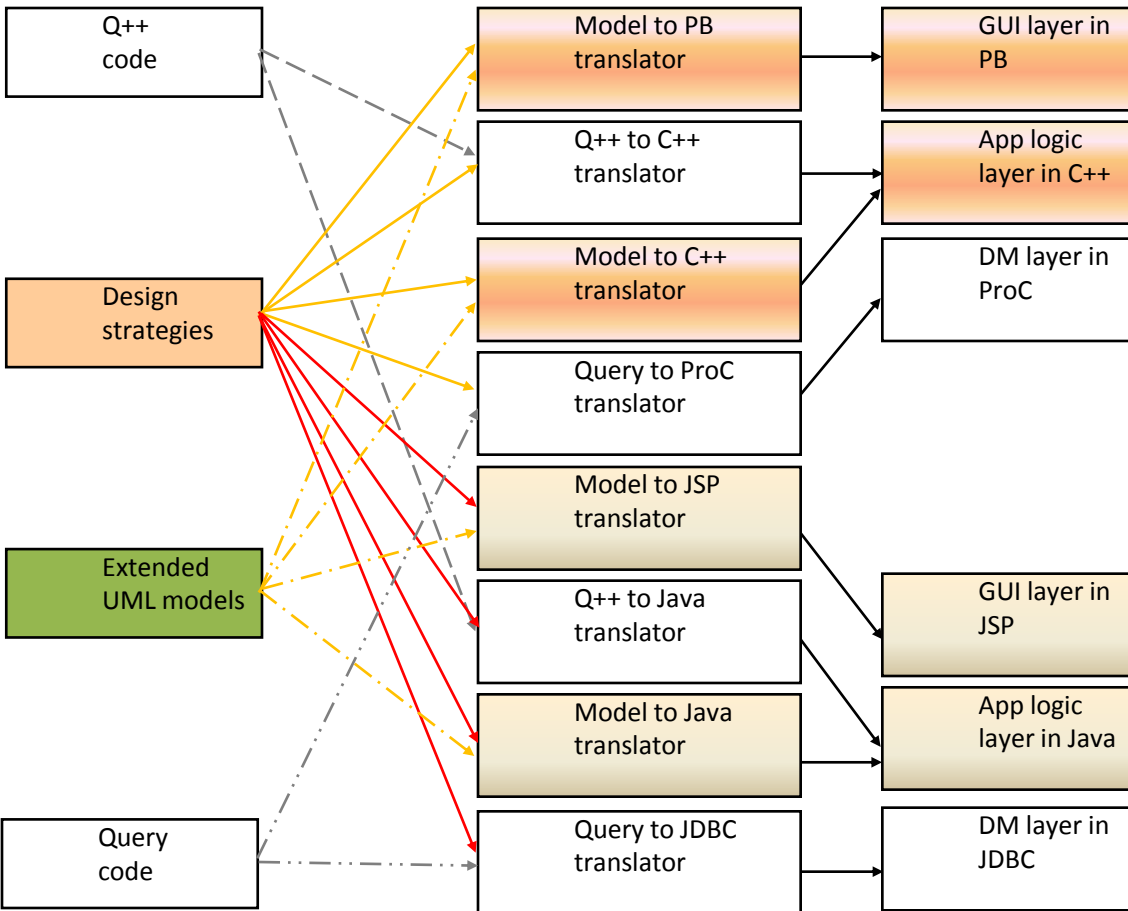
MDD tools product line

Issues in model-based code generation of business applications



- Enterprise applications have similar solution architectures but not exactly the same
 - Considerable commonality with well-defined variability
- Model-based code generation addresses A, D and T dimensions
- Choices along A, D and T dimensions cross-cut leading to scattering and tangling
- Ensuring consistent implementation of choices for a purpose-specific solution architecture is difficult
- Implementing an interpretation from scratch is effort intensive and can be error-prone
- Supporting a family of purpose-specific solution architectures even more so

Example



- A design decision impacts at multiple places

e.g.

optimistic locking strategy

- A table corresponding to a persistent class should have an additional column
- Create() and Update() methods should increment the column value

∴ impacted generation

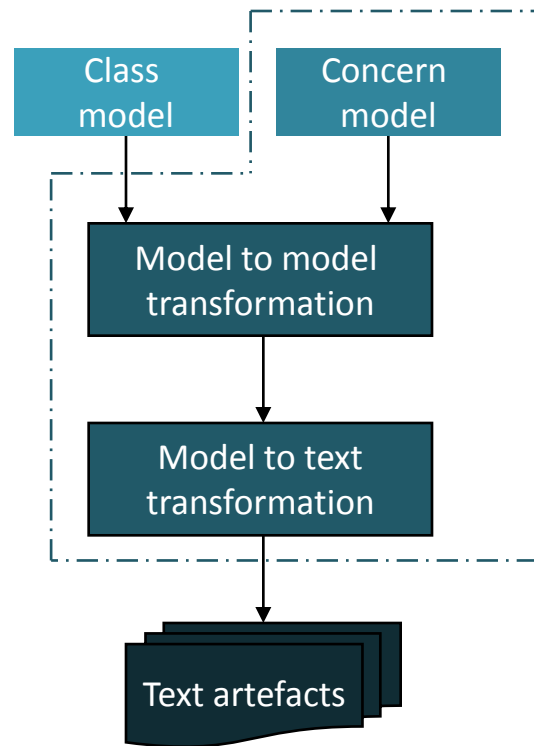
- DDL
- Primary-key based CRUD methods

For which user-defined class model suffices

A purpose-specific variation of implementing the strategy

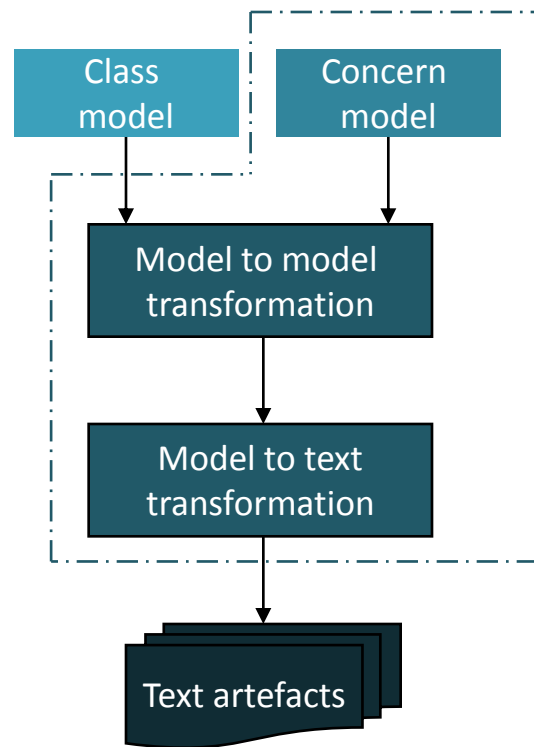
- Type of the column is 'Date' instead of 'integer'

Addressing scattering and tangling - intuition



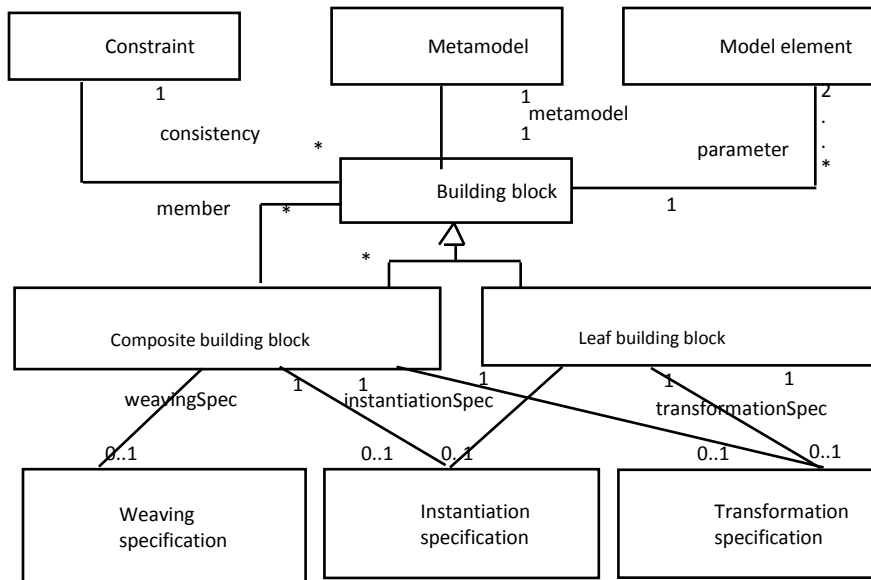
- Identify the variation points and alternatives thereof, e.g.
 - Optimistic locking – additional column, integer or date
 - Auditing – audit info, per class / service / component / application
 - User specified model (usually class model) is *base* and the variation points are *aspects*
- ∴ Therefore
- Final Model (for code generation) = Transformation(Transformation (base model, aspect model1), aspect model2) ...
 - Final code = Weaving (base code, aspect1 code, aspect2 code) ...

Addressing scattering and tangling - solution



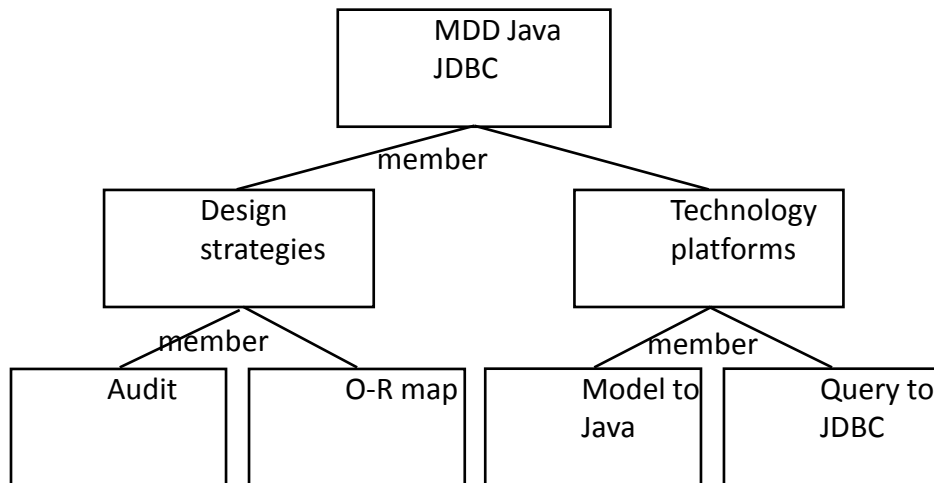
- Generate a purpose-specific code generator (or a set leading to the purpose specific solution architecture) from its specifications in model form
- Separation of concerns
 - Architecture,
 - Design strategies
 - Technology platform
- [de]composition architecture
- Weaving
 - Variety of text artefacts to be generated
 - Variety of concrete syntaxes
 - Variety of join points
 - ∴ Would necessitate a variety of AOP implementations

Building block abstraction



- Enables specification of a choice along A, D and T dimension
- Unifying abstraction to specify
 - Model contribution
 - Model 2 model transformation
 - Model 2 text transformation
 - Text composition
- Composable artefact
 - Leaf building block specifies the code to be generated
 - Composite building block specifies how the code generated by its child building block is to be composed
- Reusable artefact
 - Inheritance
 - Extension
 - Overloading

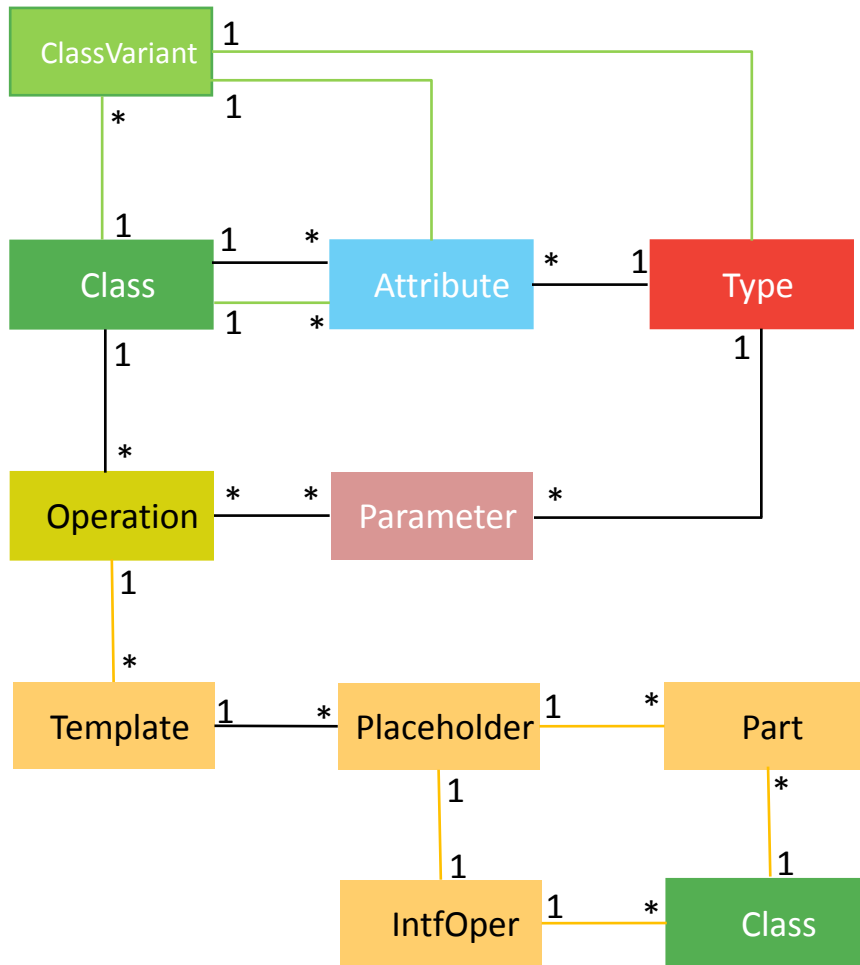
MBCG as a composition of building blocks



- [de]composition architecture
- Model-based code generation is tree traversal
 - Phase 1
 - @ Leaf: Collect model contribution
 - @ Composite: Transform the model as specified
 - Phase 2
 - @ Leaf: Transform model to text
 - @ Composite: Generate text composition specification
 - Phase 3
 - Execute text composition specification on the generated text fragments

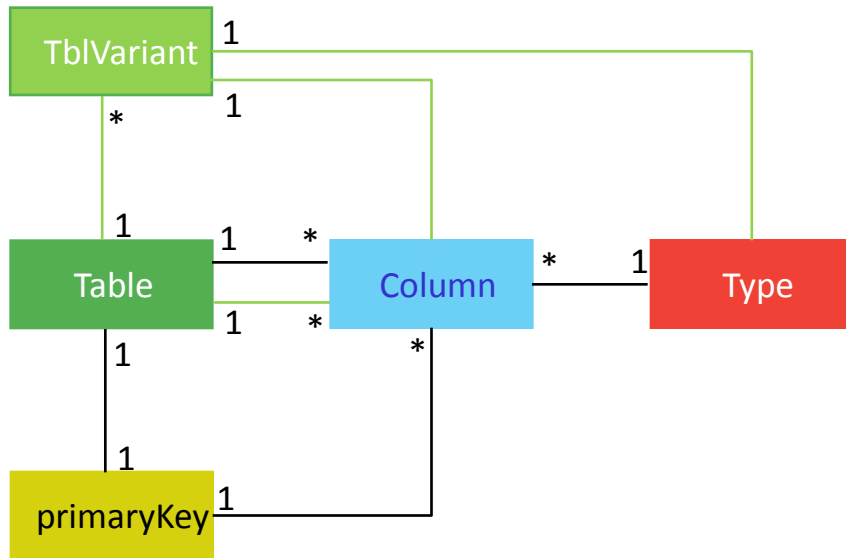
Generating application families from its model

Model of class family



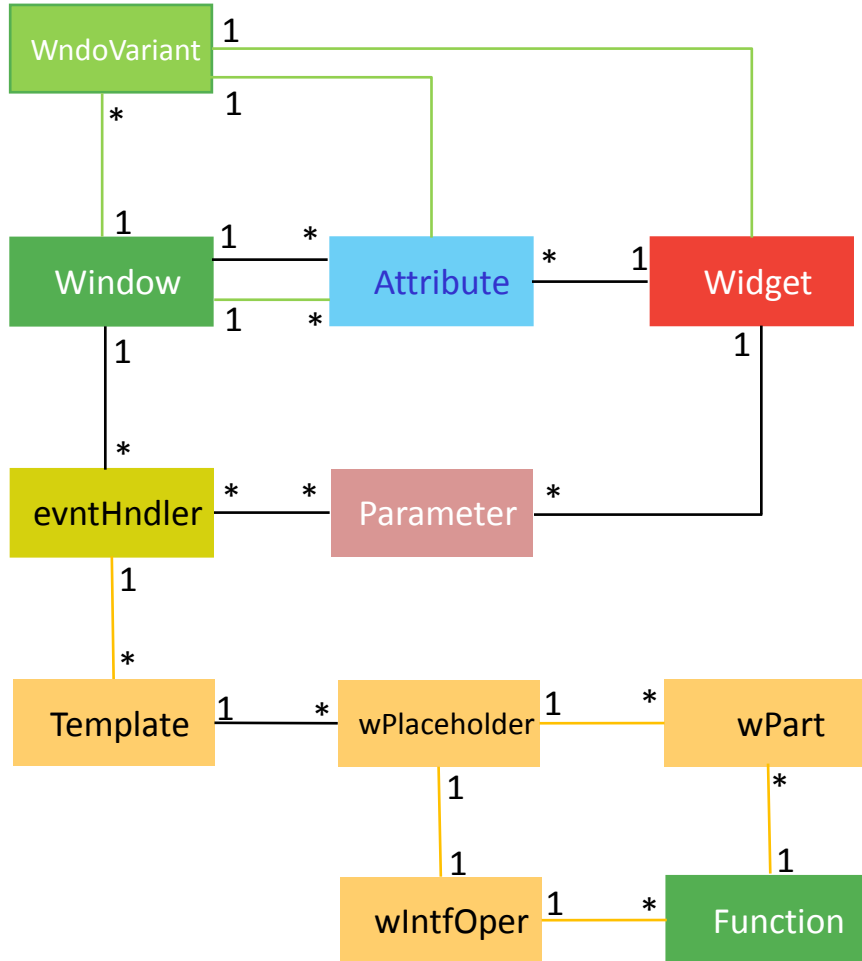
- Generic class definition
 - All attributes are present
- Generic operation bodies make extension points explicit
- Design time extensibility
 - Add a *Placeholder* for a given *Operation*
 - Add a *Part* for a given *Placeholder* for a given *Operation*
- Run time configurability
 - Choice amongst available *Parts* for each *Placeholder* of an *Operation*
- *Class* with chosen *Part* for its every *Placeholder* defines a *ClassVariant*
- User of a class knows which variant is being referred to

Model of db schema family



- Single DDL for every table
- Separate metadata table to store variation related information
- Situation specific DDL = Family DDL + interpretation of metadata
- Run time configurability
 - Generic implementation of CRUD and Query methods
 - Interpretation of metadata encoded therein
- Design time fix
 - Implementation of CRUD and Query methods specific for the DDL
 - No metadata and hence no run time interpretation

Model of GUI screens family

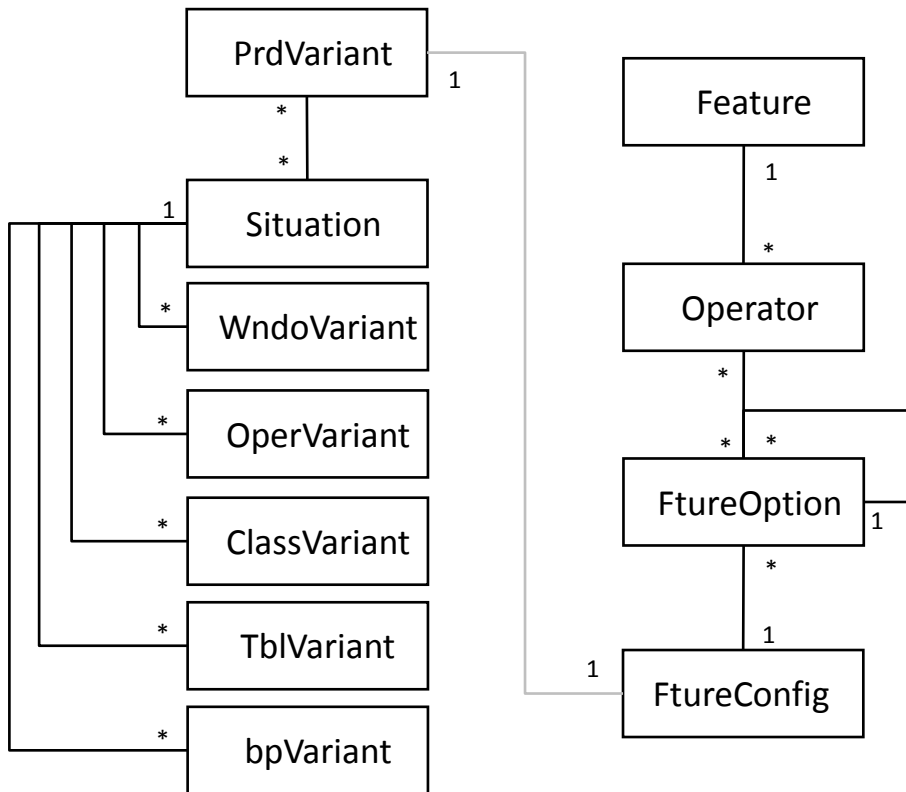


- Personalization options
 - Design time
 - Choice of widgets for a field
 - Choice of templates for event handling functions
 - Choice of event handling functions for a template
 - Run time
 - Choice of skins
 - Hide / unhide fields

Putting together the variants correctly

Solution space

Problem space



- Situation
 - A set of variants that are together valid for some purpose
- Product variant
 - A set of situations
- Feature model
 - A declarative mechanism
 - Variations and constraints
 - Purely structural – no semantics
 - No link to implementation either
- Feature configuration
 - A set of features that collectively denote some purpose
 - A well-formed projection of feature model
- Still at the implementation level only

Summary

What

- Family caters to a set of related requirements that exhibit commonality and well defined variability
 - Common part + the desired choice delivered as a ‘bespoke’ solution
 - Flexibility to choose one of the many alternatives at application run time

Why is it needed

- Identification of common part leads to greater reuse
- Identification of where and how members differ from each other leads to work commensurate with the desired change
- Well-defined operations such as *add a variant*, *customize existing parts* and *assemble from existing parts* lead to managed evolution

How to raise one

- Essentially, divide and conquer approach
 - Identify decomposition structure for the solution thus identifying parts
 - Identify which parts change over time and which don’t – and how they change
 - Implement parts
 - Implement the composition
- Model-driven approach scores over code-centric approach in many ways

Towards a model-driven software factory

Less *how* and more *what*

Abstraction

Models

Abstract away the implementation platform

Predefined functionality with placeholders

Freedom from *repetitive mundane* tasks

Automation

Programs

Abstract away the hardware

Application generators

Frameworks

Architecturally complete application

Machine code

Libraries

Oft required computations

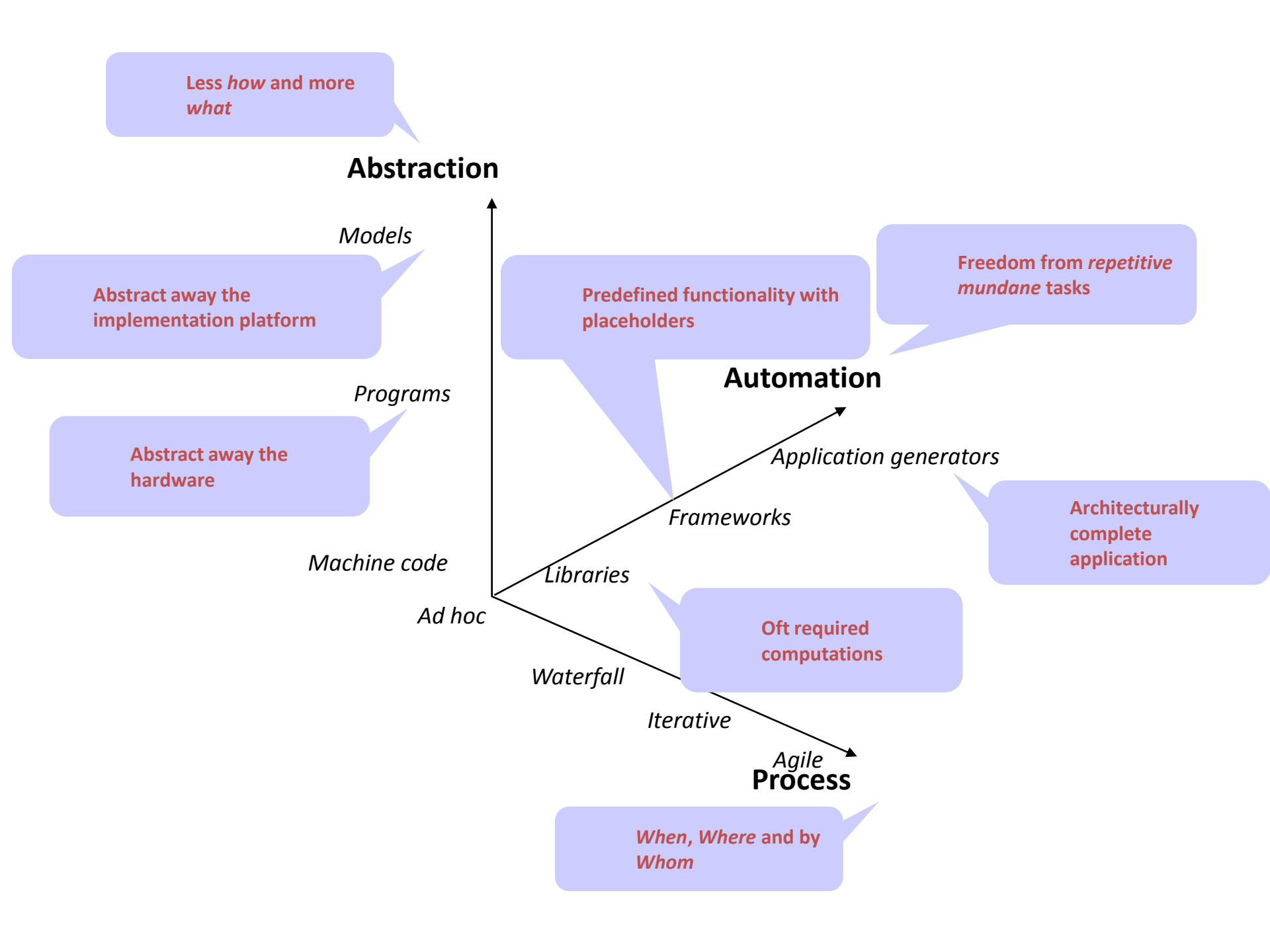
Ad hoc

Waterfall

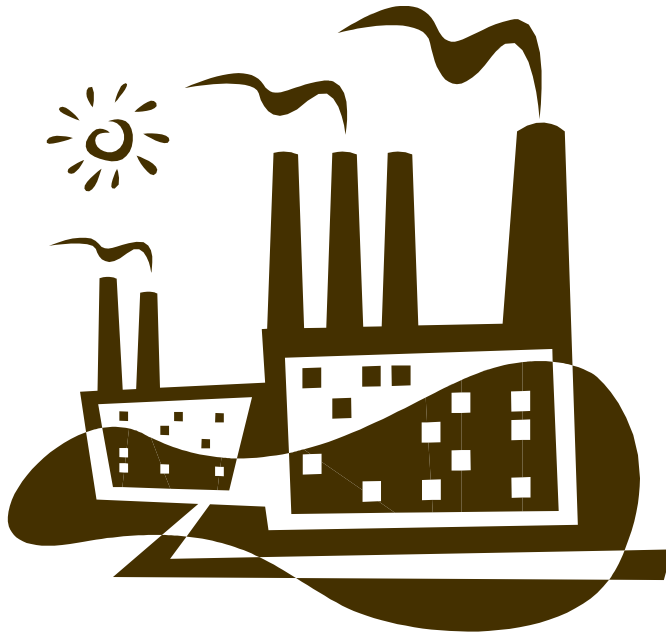
Iterative

Agile Process

When, Where and by Whom



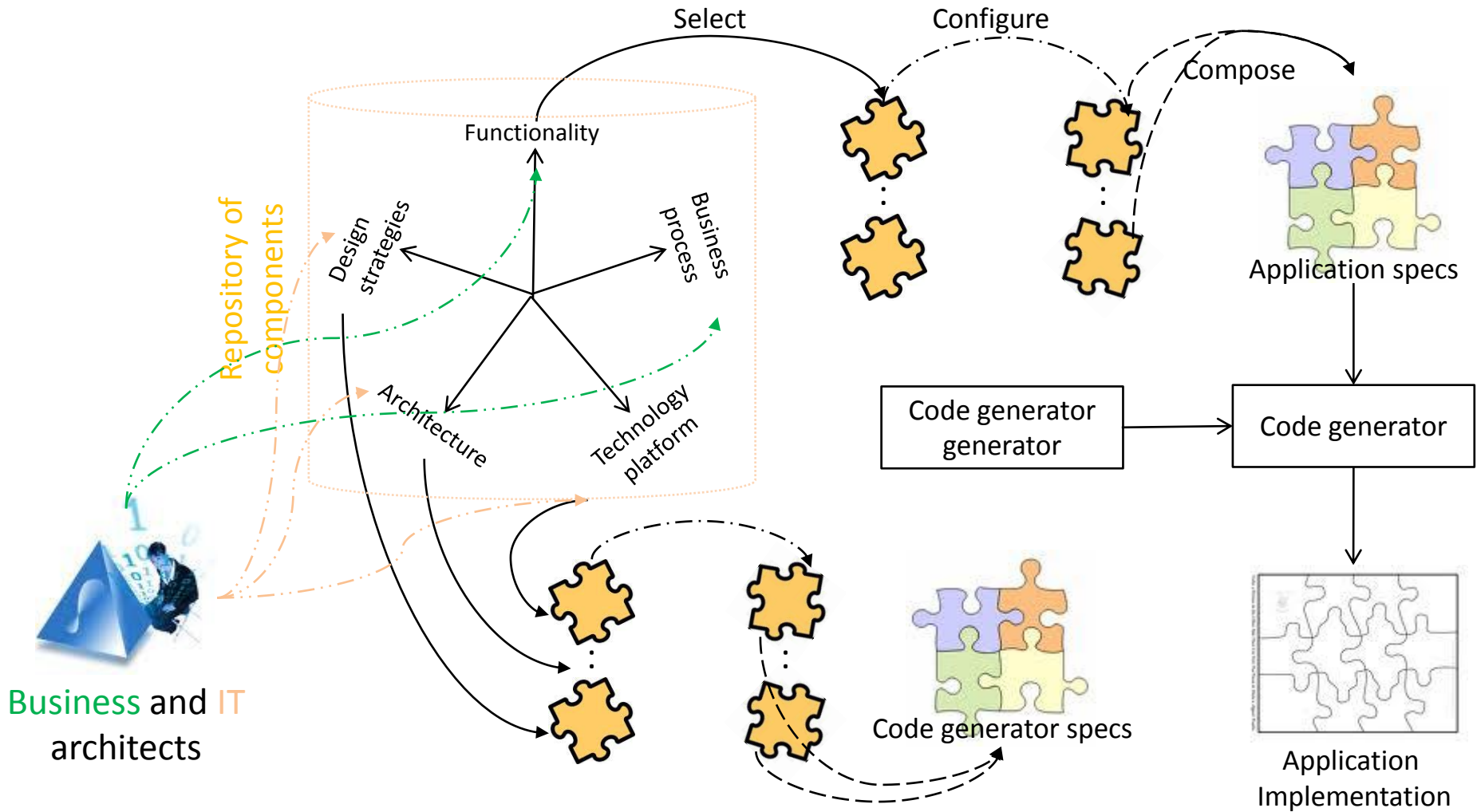
Factory approach



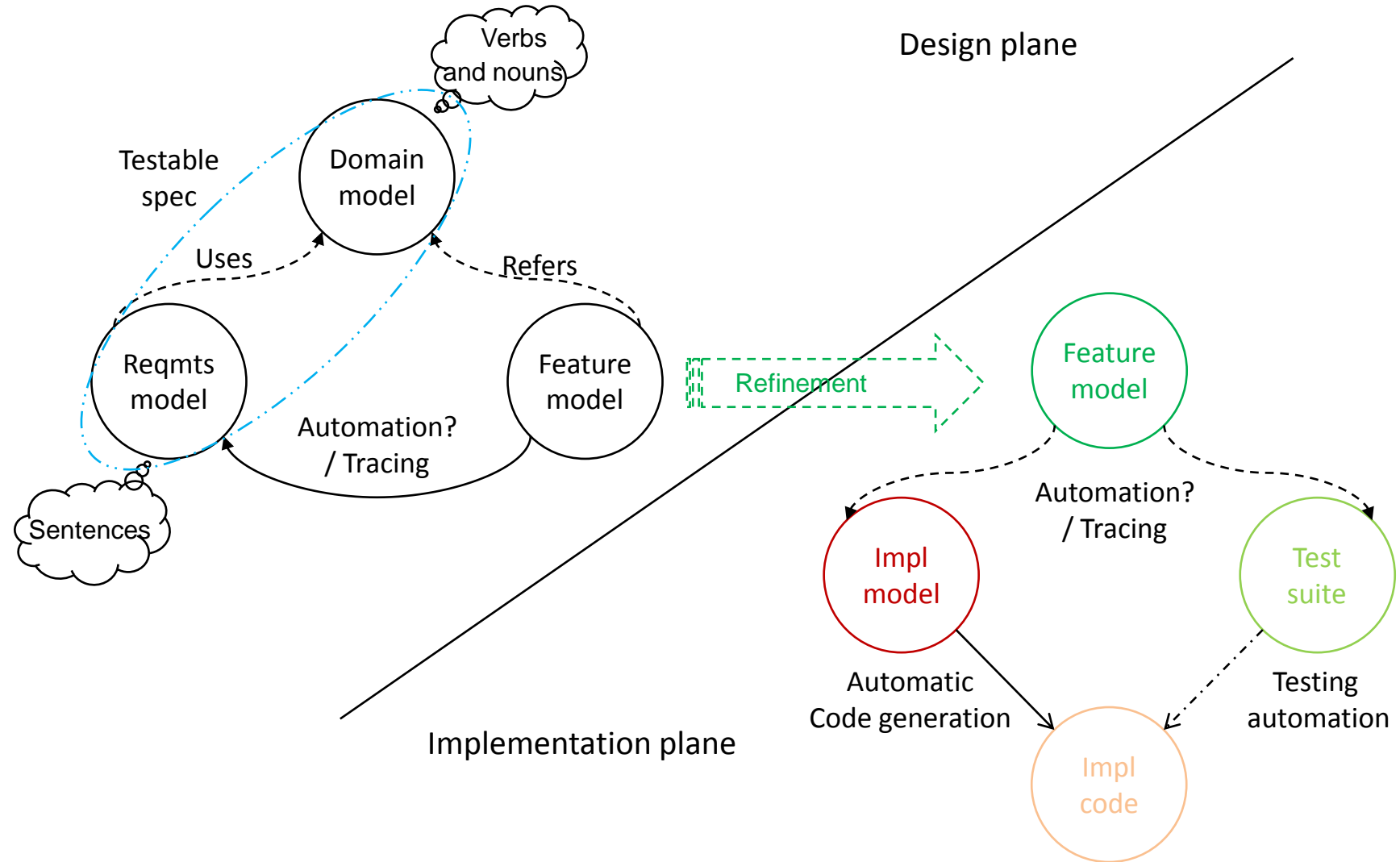
- Characterized by
 - Tools replacing artisans for repetitive tasks
 - A well-defined method
 - Standards
 - Flexible assembly-line
- So as to deliver
 - Higher productivity
 - Uniformly high quality

in a **person independent** manner

Our factory



Future work



Credits

All associated with development of *MasterCraft*

