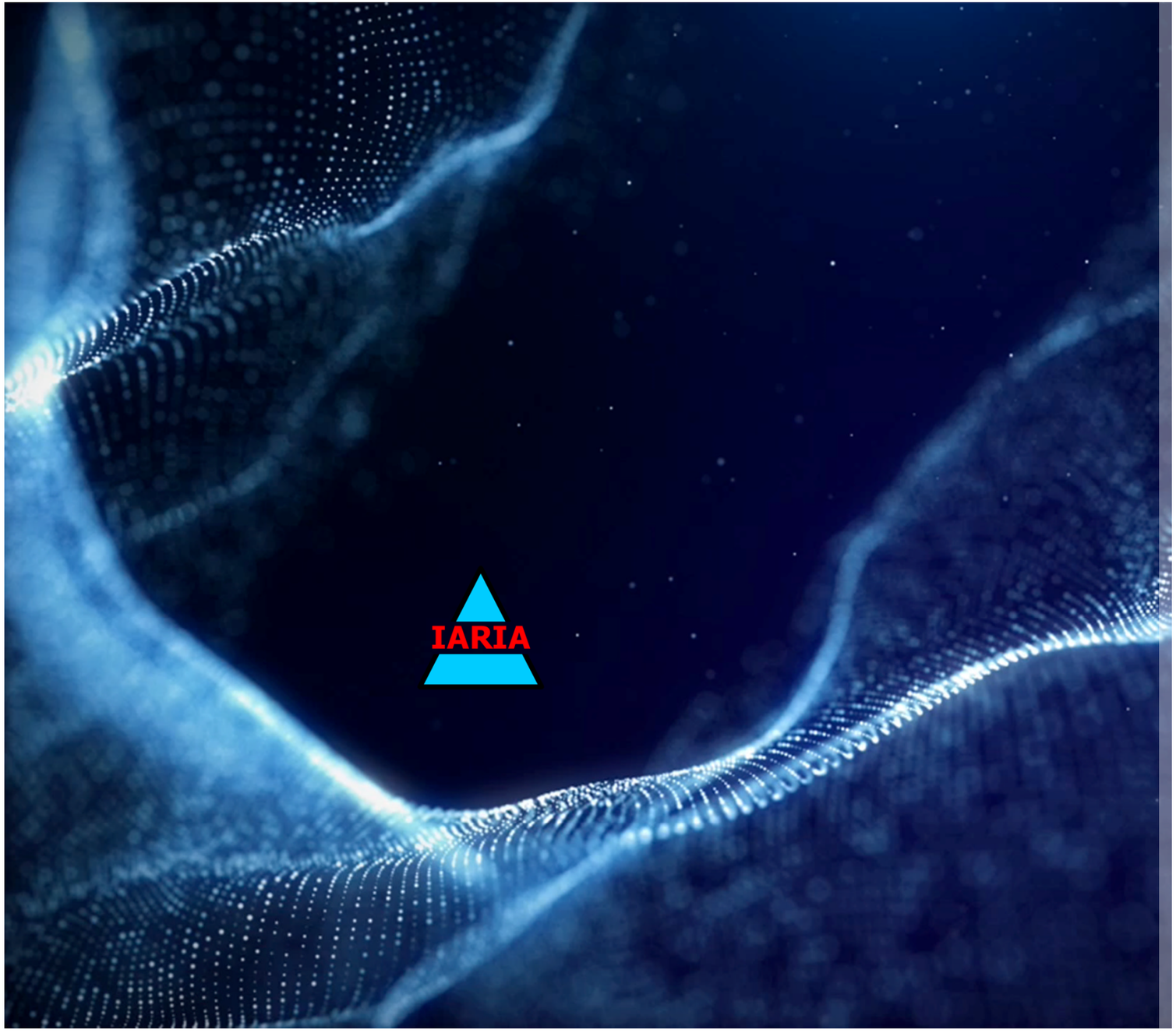



SOFTNET 2019

# Advancing Automatic Programming: Regeneration, Meta-circularity, and Two-Sided Interfaces

HERWIG MANNAERT

NOVEMBER 27, 2019

Universiteit Antwerpen

ADVANCING AUTOMATIC PROGRAMMING

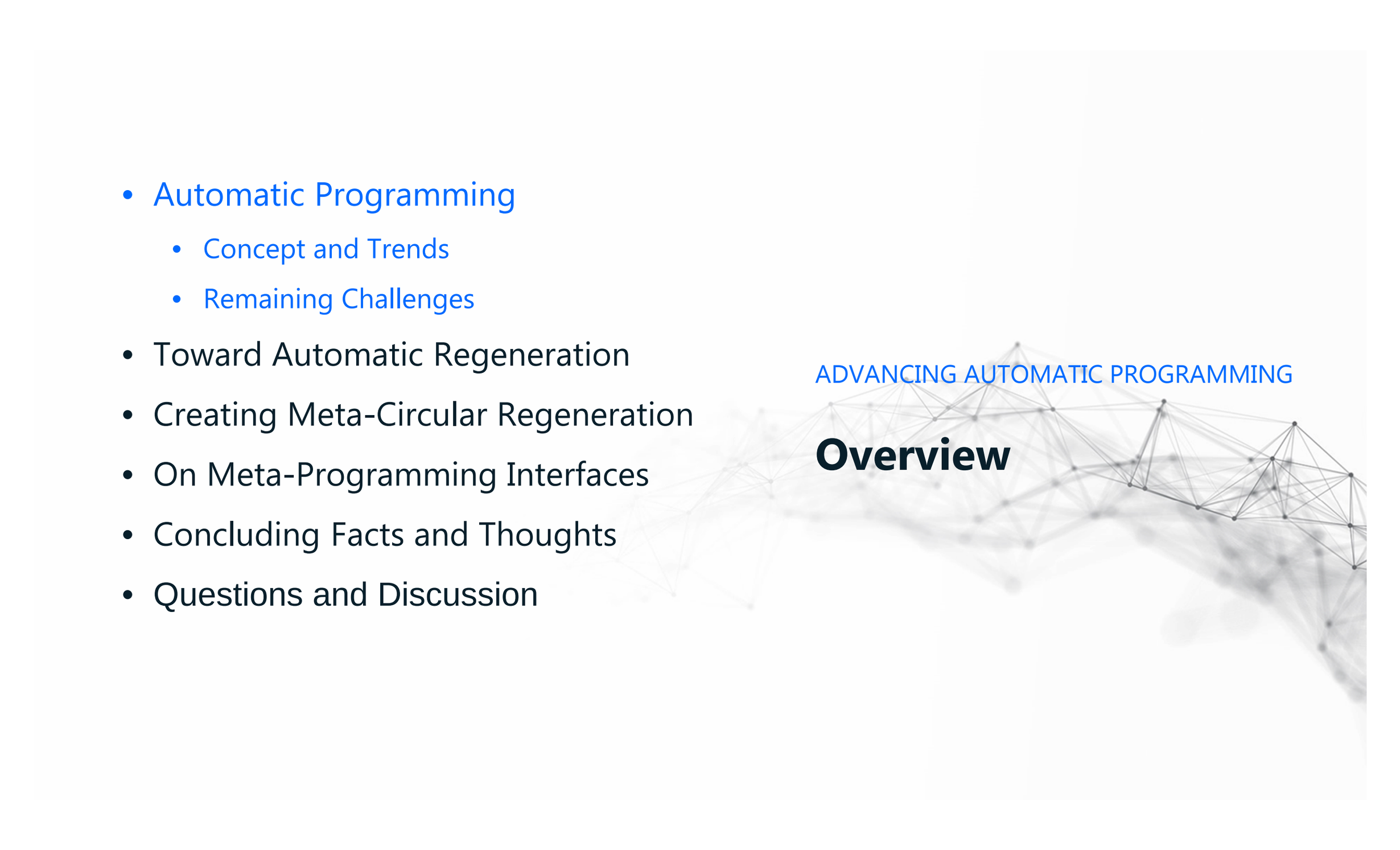
# Overview

- Automatic Programming
- Toward Automatic Regeneration
- Creating Meta-Circular Regeneration
- On Meta-Programming Interfaces
- Concluding Facts and Thoughts
- Questions and Discussion

- Automatic Programming
  - Concept and Trends
  - Remaining Challenges
- Toward Automatic Regeneration
- Creating Meta-Circular Regeneration
- On Meta-Programming Interfaces
- Concluding Facts and Thoughts
- Questions and Discussion



# Overview



# Automatic Programming

- Automatic programming:
  - The act of automatically generating source code from a model or template
  - Sometimes distinguished from *code generation*, as performed by a compiler
  - Has always been a euphemism for programming in a higher-level language than was then available to the programmer [David Parnas]
- Also referred to a generative or meta-programming
  - To manufacture software components in an automated way
- It is as old as programming itself:

```
System.out.println(“Hello world.”);
```

```
System.out.println(“System.out.println(\“Hello world.\”);”);
```

# The Need for Automatic Programming

- Goal is and has always been to improve programmer productivity
- In general, to manufacture software components in an automated way, in the same way *as automation in the industrial revolution*, would:
  - Increase programming productivity
  - Consolidate programming knowledge
  - Eliminate human programming errors
- Such an approach is likely to address many long-term issues:
  - Growing amount of software
  - Shortage of computer programmers
  - Increasing amount of software bugs and defects
  - Ever rising IT development and maintenance budgets

# The Field of Automatic Programming

- Better known through names/trends like:
  - Model-Driven Architecture (MDA)
  - Model-Driven Engineering (MDE)
  - Model-Driven Software Development (MDSD)
  - Low-Code Development Programs (LCDP)
- The various trends share the *use of models to structure requirements* and/or *to represent domain knowledge*:
- The field is still evolving and facing challenges and criticisms:
  - Suitability for large-scale and mission-critical enterprise systems
  - Lack of intermediate representation, pervasive concepts for DSL reuse
  - Either a *conceptual gap* toward code, *or tied to* a technological solution

# Relevance of Automatic Programming

- The issues that automatic programming is supposed to address/solve are as relevant and acute as ever:
  - Software is growing in size and importance
  - Shortage of tens of thousands of programmers
  - Multi-trillion lines of code with billions of defects
  - Gigantic IT development and maintenance budgets
- Automatic programming has not yet delivered on its promises, because we believe at least *these fundamental issues need to be addressed*:
  - Regeneration with support for additive manual code
  - Meta-circularity to include the automatic programming code
  - Two-sided programming interfaces to support scalable collaboration

- Automatic Programming
- Toward Automatic Regeneration
  - The NS Elements Model
  - The Need for Regeneration
  - Exploring an Implementation
- Creating Meta-Circular Regeneration
- On Meta-Programming Interfaces
- Concluding Facts and Thoughts
- Questions and Discussion



# Overview

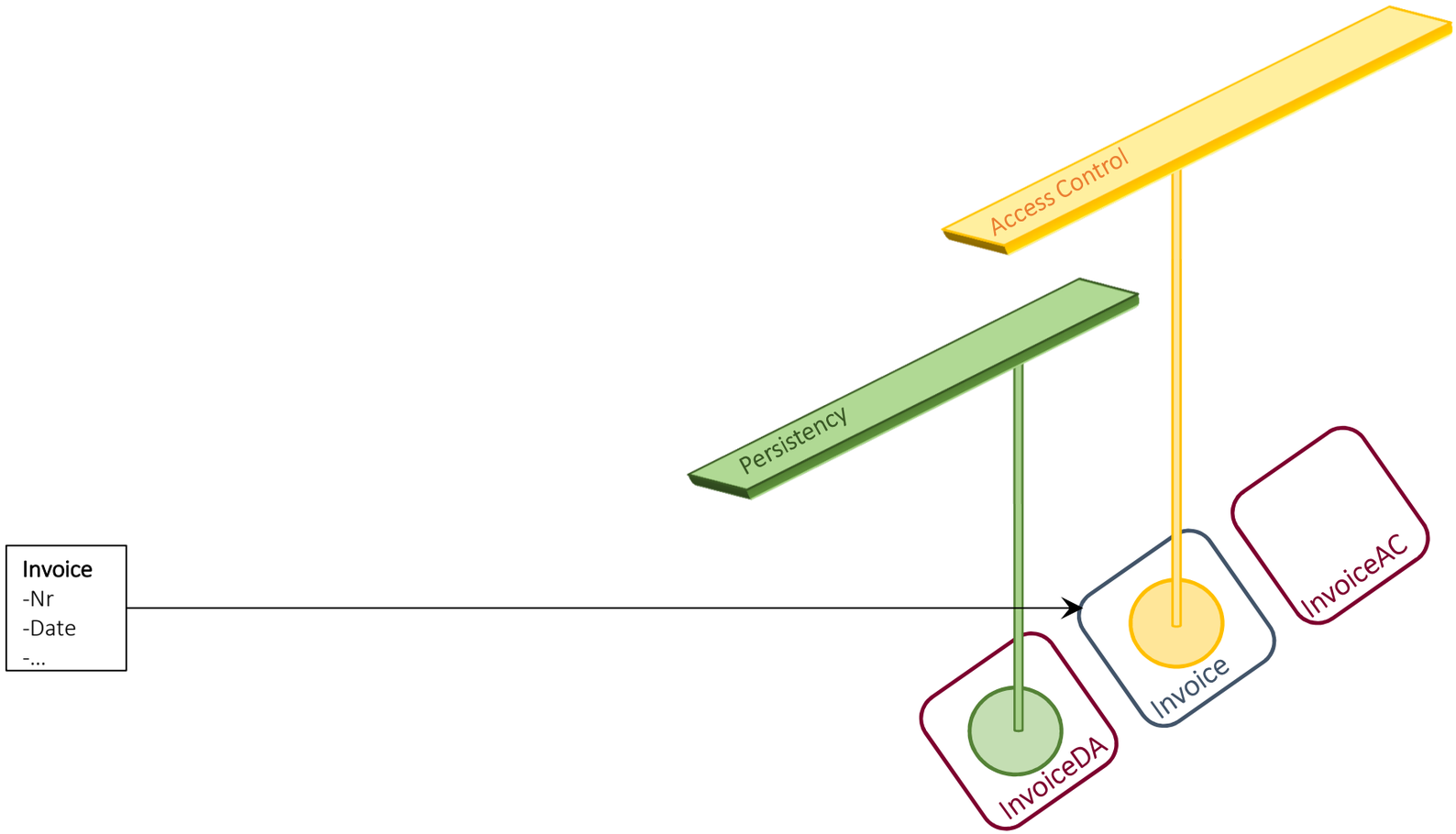
# The NS Elements Model

- Automatic programming uses *models that are transformed into code*
- *Code transformation* is based on Normalized Systems Theory:
  - Seeks to provide ex-ante proven approach to build evolvable software
  - Founded on systems theoretic stability (BIBO), for the impact of changes
- NST proves a set of principles, that are necessary conditions to avoid *instabilities or combinatorial effects*:
  - Separation of Concerns
  - Action Version Transparency
  - Data Version Transparency
  - Separation of States

# The NS Elements Model

- Automatic programming uses *models to represent domain knowledge*
- Our *intermediate NS Elements model* is a *General Purpose Language*:
  - At the low end of the modelling abstraction, but an executable model
  - Very basic and close to traditional software implementation concepts
  - Aligned with the fundamental primitives of a Von Neumann processor
- NS Elements model defines 5 types of elements:
  - Data element
  - Task element
  - Flow element
  - Connector element
  - Trigger element

# Separating Cross-Cutting Concerns



# Separating Cross-Cutting Concerns

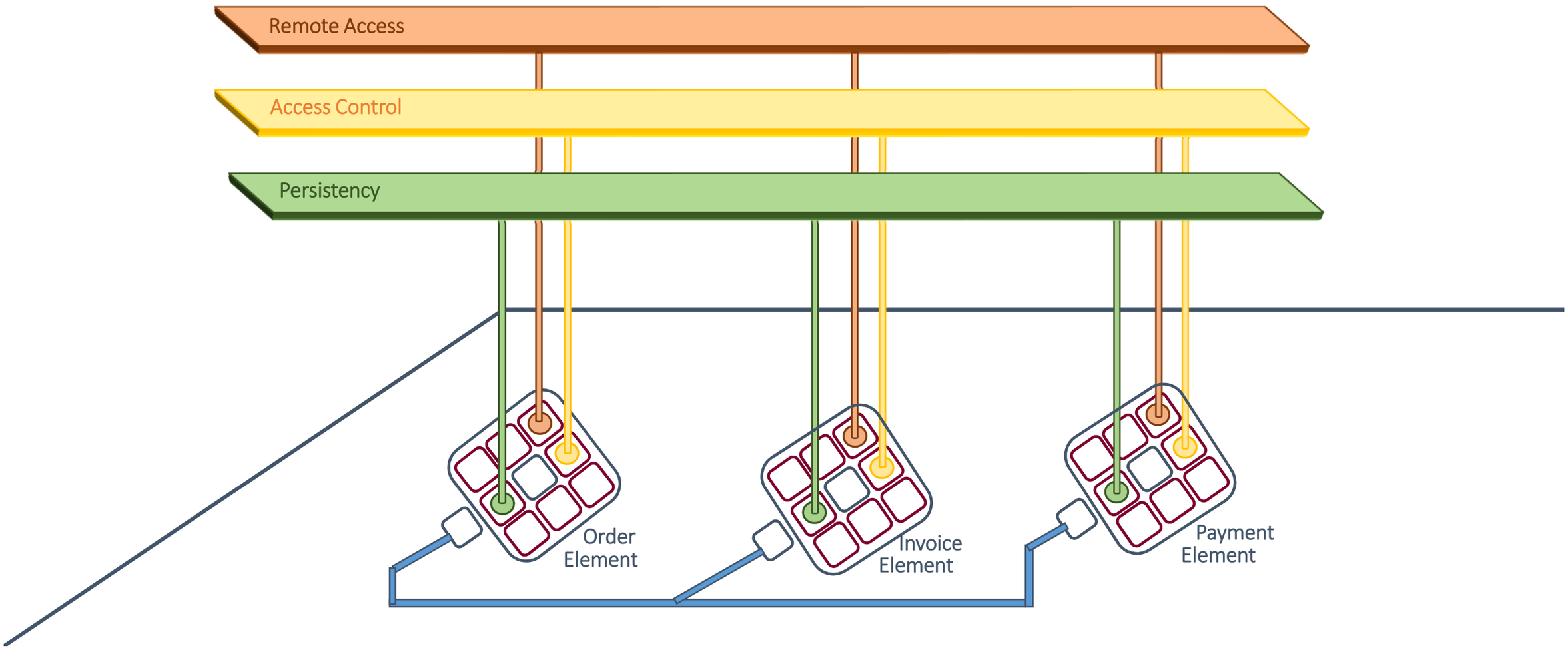
```java
@Entity(name="be.uantwerpen.ea.Person")
@Table(name="TUTO_TUTO_Person")
@NamedQueries({
  @NamedQuery(name="be.uantwerpen.ea.Person.findAll",
  query="select o FROM be.uantwerpen.ea.Person o")
  // anchor:custom-queries:start
  // anchor:custom-queries:end
})
public class PersonData implements java.io.Serializable {

    private Long mId;
    // anchor:member-fields:start
    private String mName;
    private Integer mAge;
    private String mFunction;
    // anchor:member-fields:end

    /*========== Default constructor ==========*/
    public PersonData() {
        // this.mId = new Long(0);
    }

    /*========== Detailed constructor ==========*/
    public PersonData(Long id
        // anchor:constructor-parameters:start
        , String name
        , Integer age
        , String function
        // anchor:constructor-parameters:end
        ) {
        this.mId = id;
        // anchor:constructor-assign:start
        this.mName = name;
        this.mAge = age;
        this.mFunction = function;
        // anchor:constructor-assign:end
    }

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Long getId() {
        return this.mId;
    }

    public void setId(Long id) {
        this.mId = id;
    }
```

Persistency

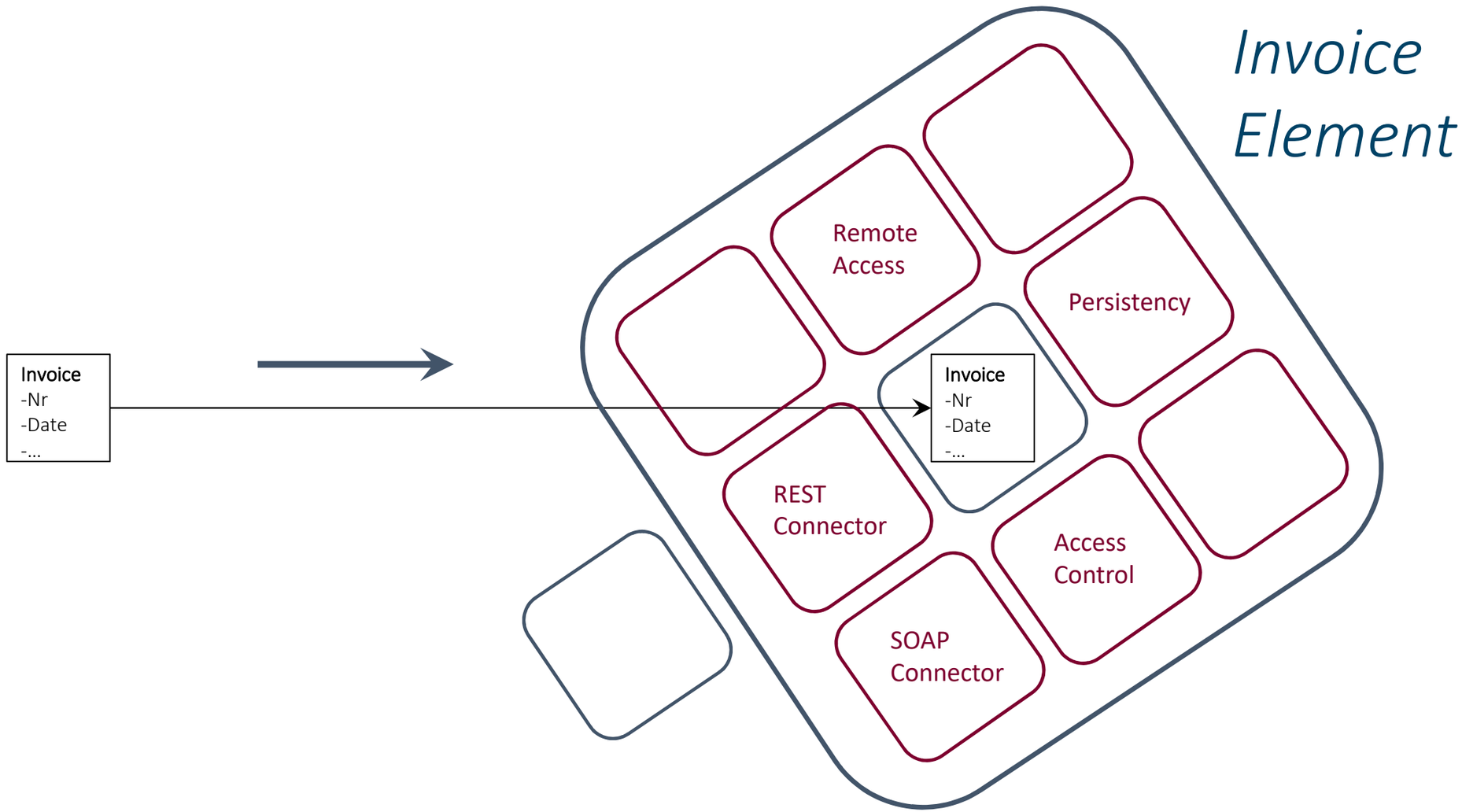
```java
@Stateless()
@Remote(PersonRemote.class)
@Local(PersonLocal.class)
@DeclareRoles({
// anchor:custom-declare-roles:start
// anchor:custom-declare-roles:end
})
public class PersonBean implements PersonRemote, PersonLocal {

    private static Logger logger = LoggerFactory.getLogger("be.uantwerpen.ea.
        PersonBean");

    protected Context initialContext;

    @Resource
    private SessionContext sessionContext;

    @EJB
    PersonCrudsLocal personCrudsLocal;

    // anchor:context-variables:start
    // anchor:context-variables:end

    // anchor:link-variables:start
    // anchor:link-variables:end

    // anchor:custom-variables:start
    // anchor:custom-variables:end

    /*========== Bean lifecycle methods ==========*/

    // anchor:custom-create-annotations:start
    // anchor:custom-create-annotations:end
    public CrudsResult<DataRef> create(ParameterContext<PersonDetails>
        detailsParameter) {
        if (sessionContext.getRollbackOnly()) {
            return getDiagnosticHelper().createCrudsError(CREATE_ERROR_MSG_KEY);
        }

        PersonDetails details = detailsParameter.getValue();
        UserContext userContext = detailsParameter.getUserContext();
```

Transaction

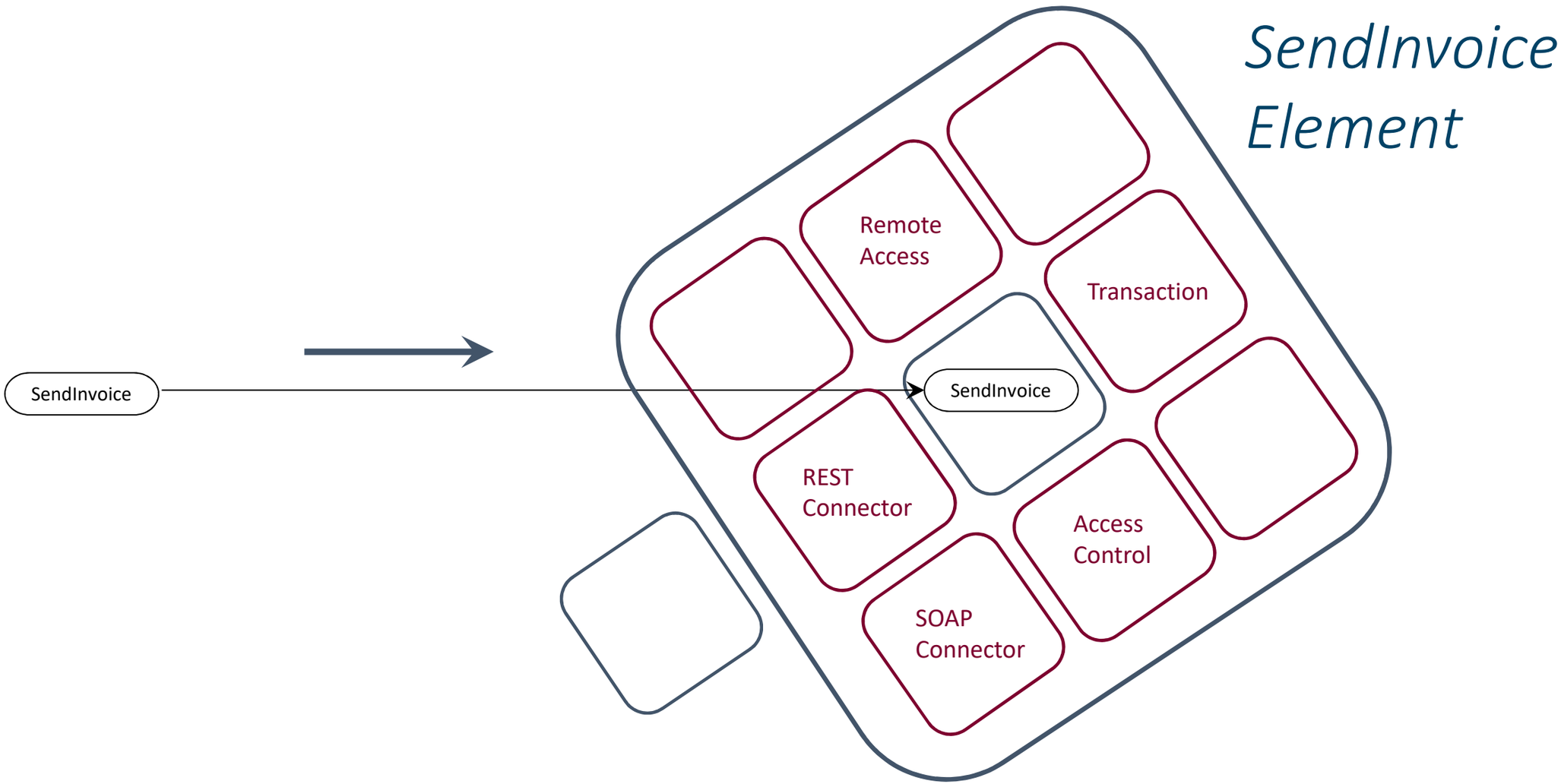
# The Emergence of Elements



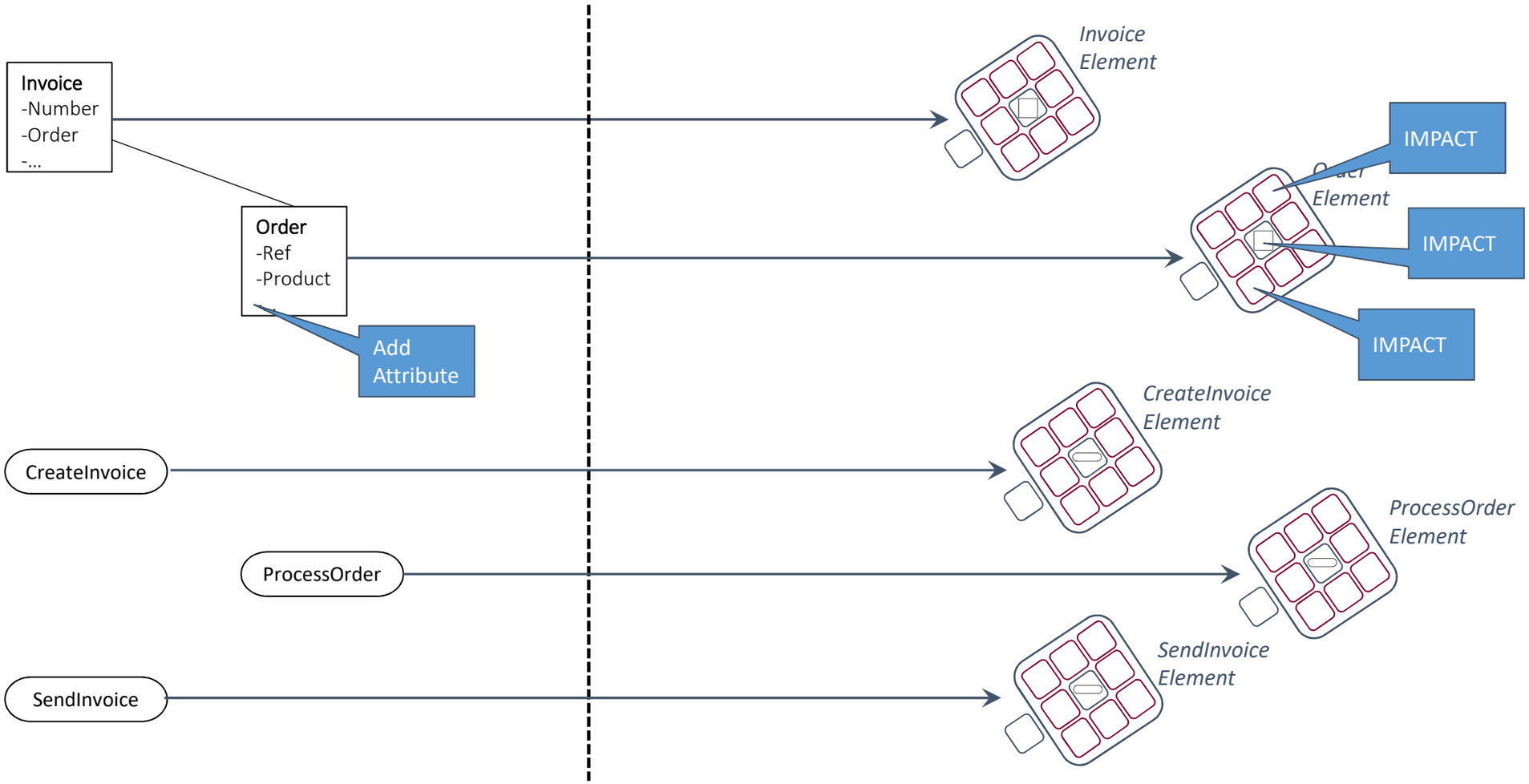
# Model Transformation: Data Element



# Model Transformation: Task Element



# Model Transformation: Recurring Structure



# Generating Recurring Structure: Catch 22

- Structure should be recurring, as variations:
  - increase complexity of codebase
  - decrease consistency in behaviour
- Recurring structure may need to vary over time:
  - new insights
  - discovery of flaws
  - changes in technologies

N=100

| K | Total |
|---|---|
| 100 | 100 |
| 50 | 150 |
| 20 | 300 |
| 10 | 550 |
| 5 | 1050 |
| 2 | 2550 |
| 1 | 5050 |

*Structural changes may need to be applied with retroactive effect, but the efforts increase with the frequency of change.*

*N instances, update every K* ➔ ***#updates*** = $\frac{N(N+K)}{2K}$

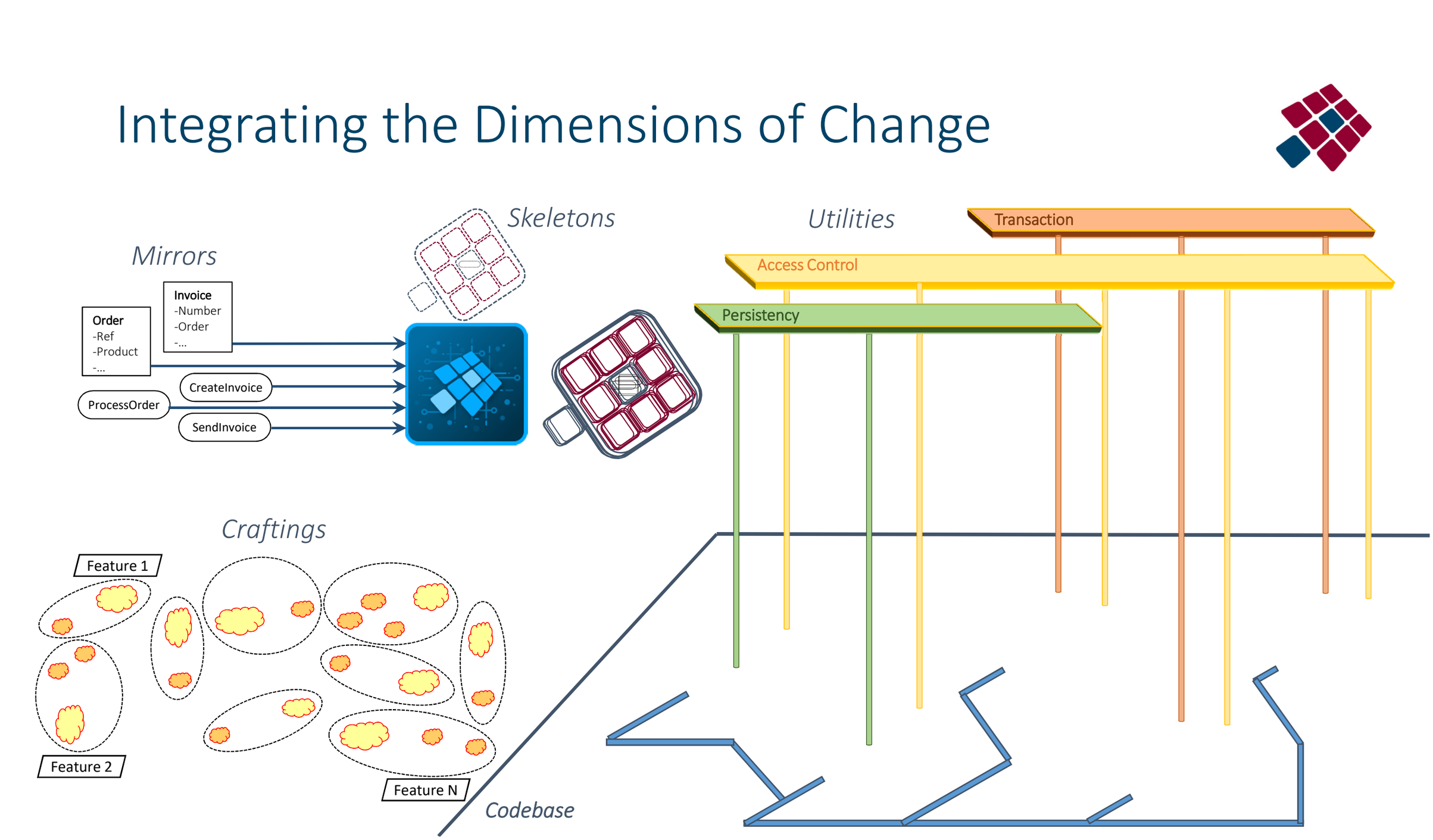
| 150 | 300 | 550 | 1050 |
|---|---|---|---|
| | | | 5 |
| | | 10 | 10 |
| | | | 15 |
| | 20 | 20 | 20 |
| | | | 25 |
| | | 30 | 30 |
| | | | 35 |
| | 40 | 40 | 40 |
| | | | 45 |
| 50 | | 50 | 50 |
| | | | 55 |
| | 60 | 60 | 60 |
| | | | 65 |
| | | 70 | 70 |
| | | | 75 |
| | 80 | 80 | 80 |
| | | | 85 |
| | | 90 | 90 |
| | | | 95 |
| 100 | 100 | 100 | 100 |

# Catch 22: The Only Way Out

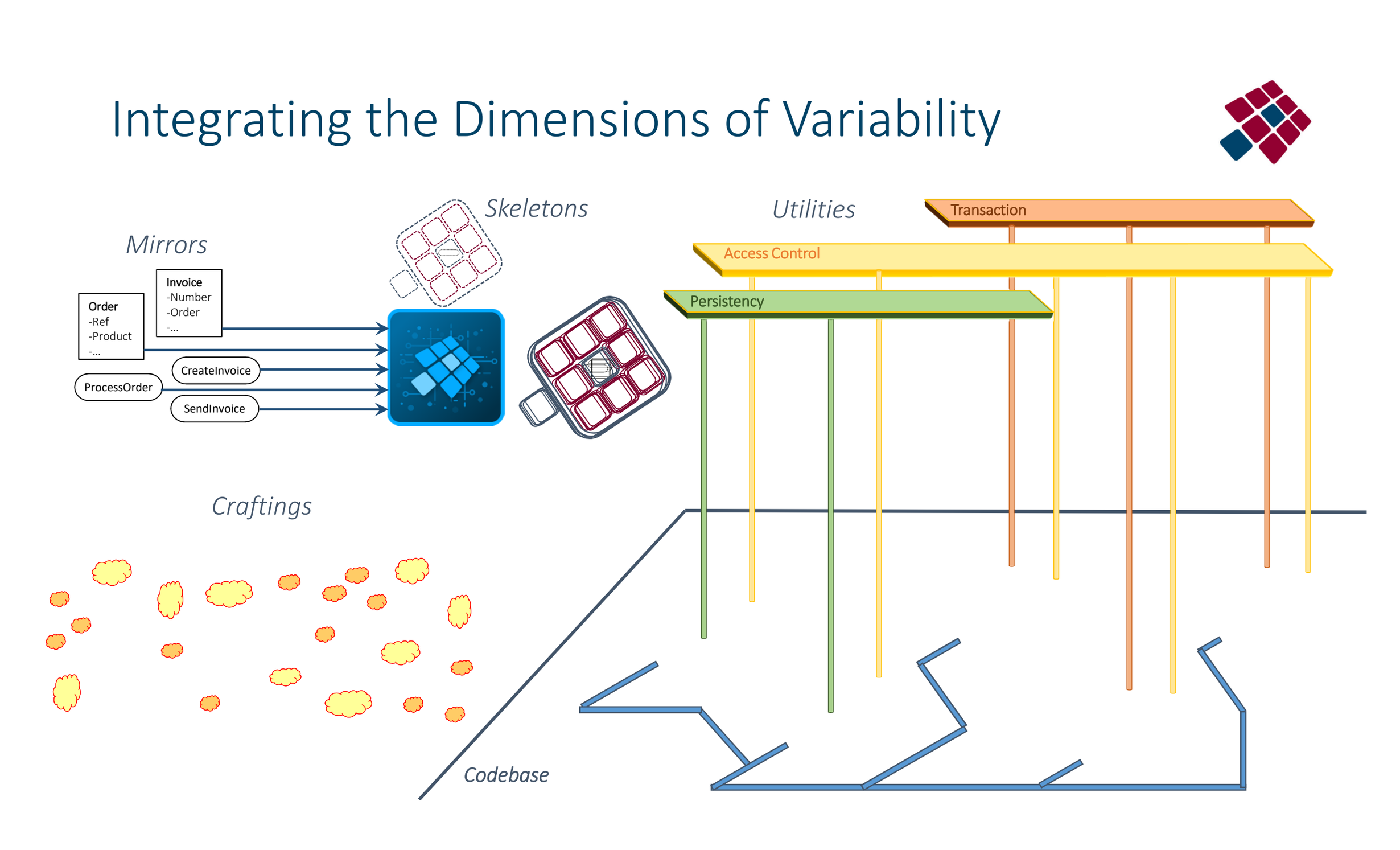
- Recurrent stable structures are required to limit complexity and to guarantee consistency
- Recurrent stable structures need to be able to adapt over time, to overcome flaws and technology changes
- Additional custom code is inevitable and needs to be maintained across updated stable structures

*An automated mechanism is required,*
*providing both code generation or expansion,*
*and regeneration with harvesting and injection.*

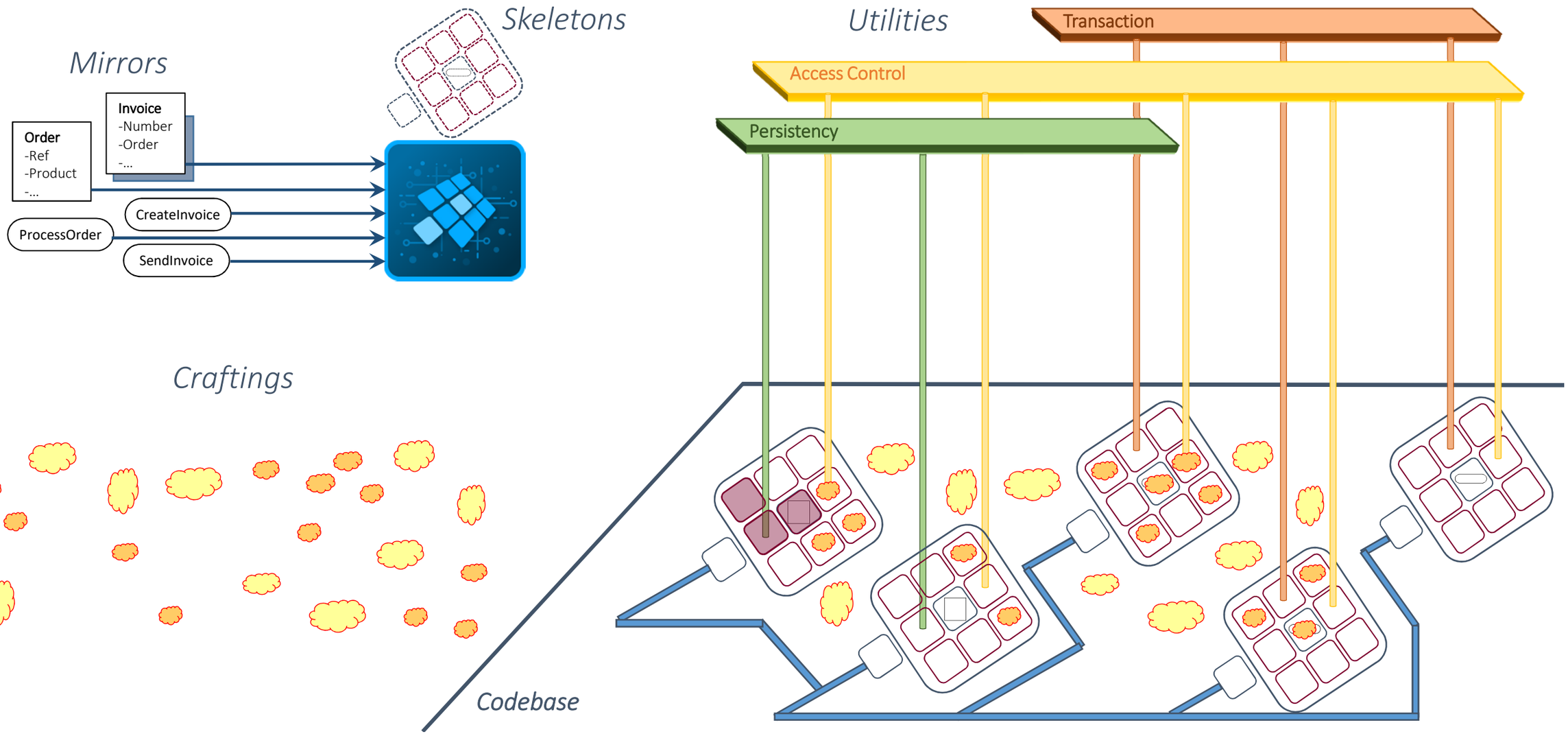
# Integrating the Dimensions of Change



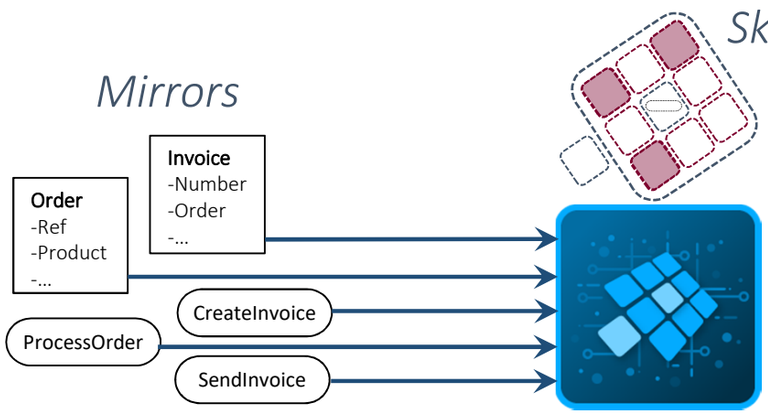
# Integrating the Dimensions of Variability

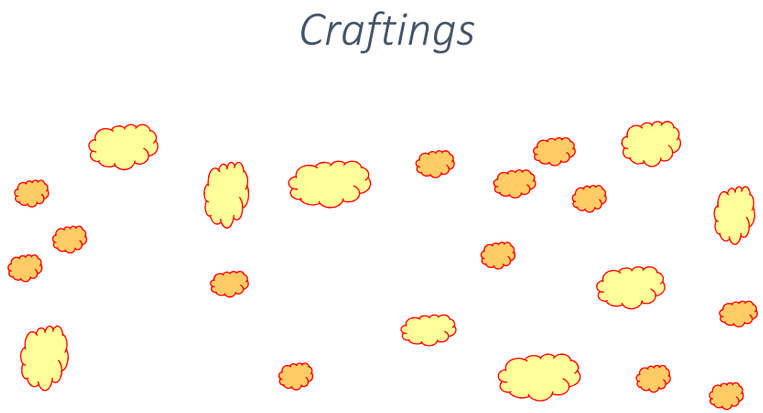


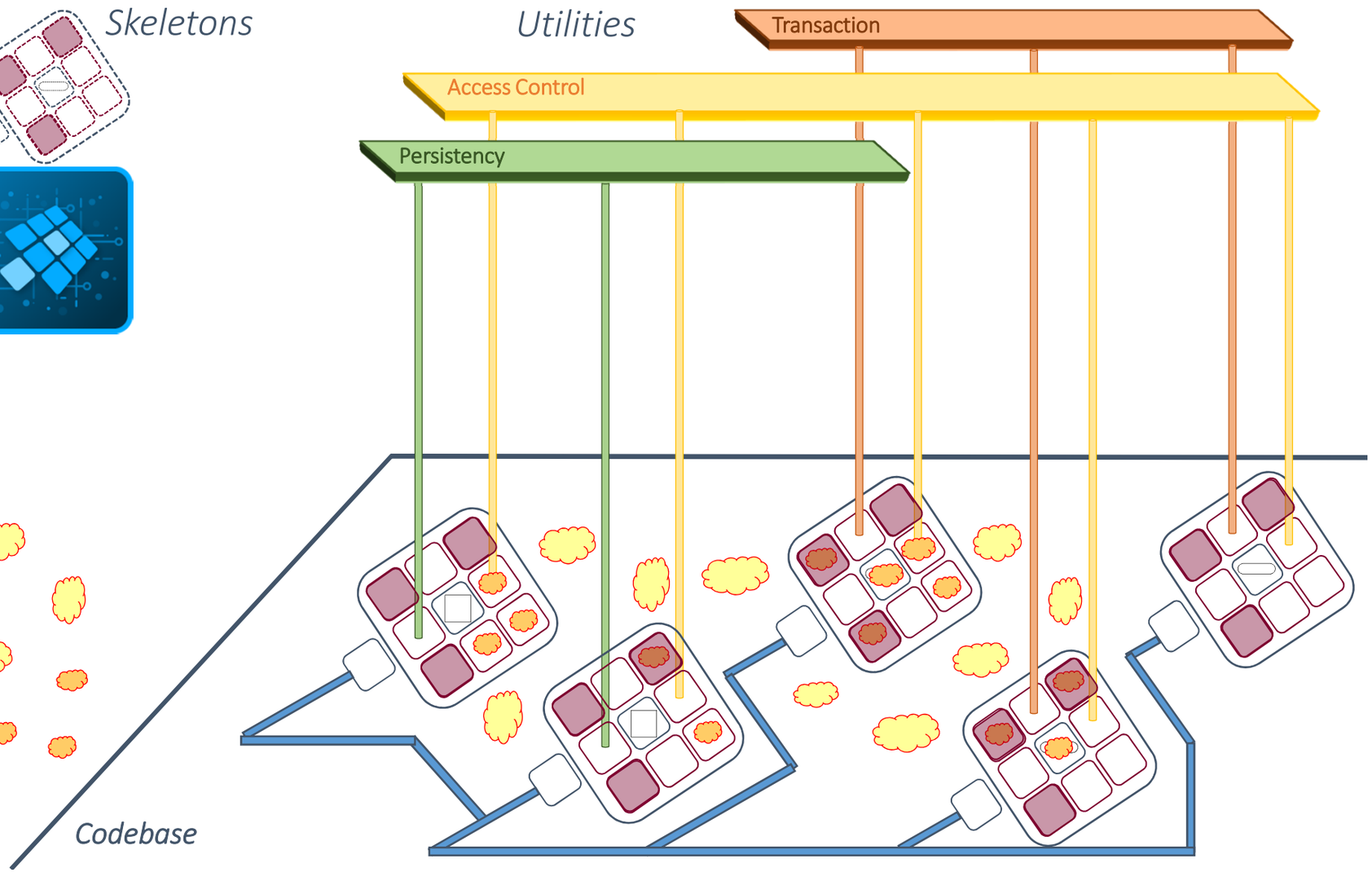
# Change Dimension 1: The Mirrors



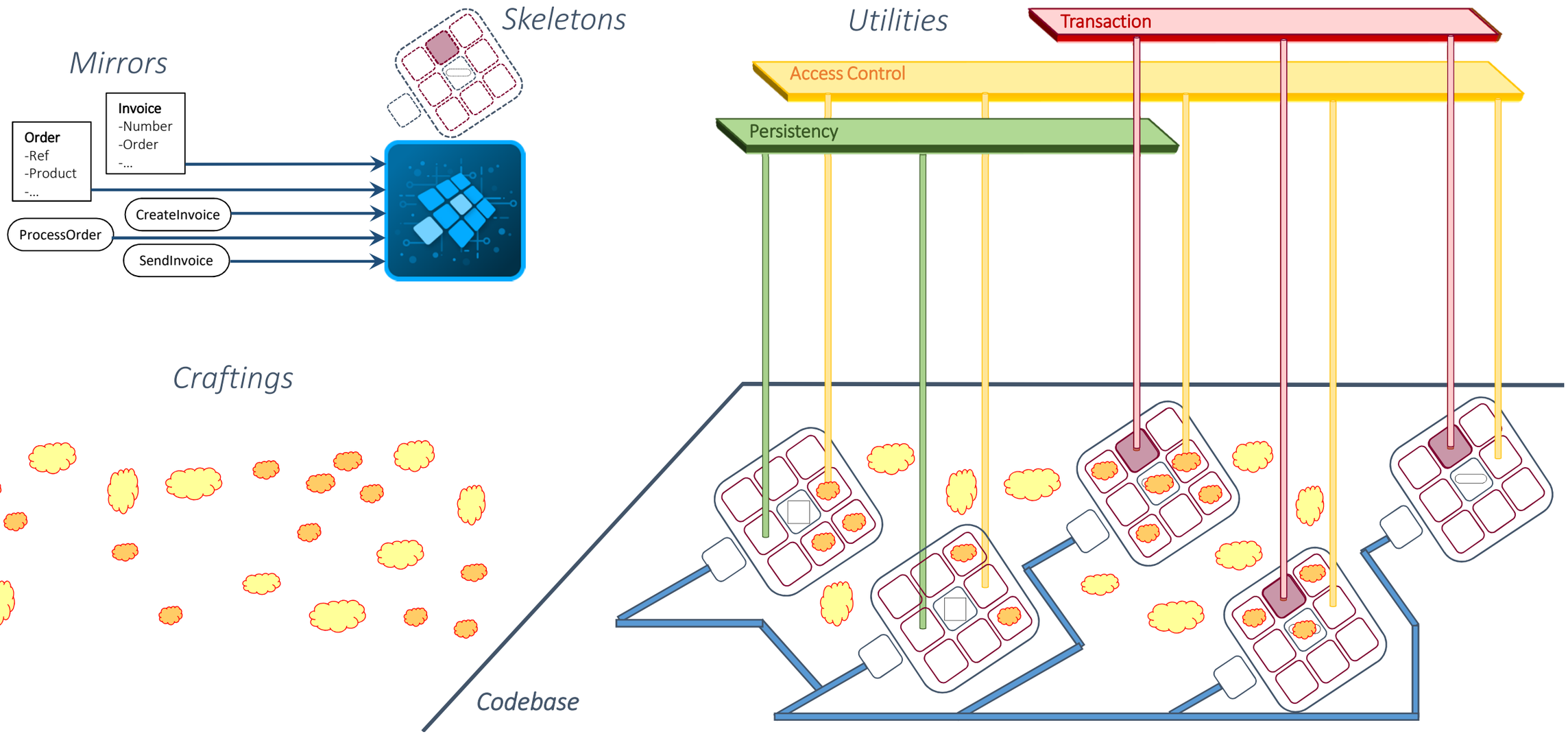
# Change Dimension 2: The Skeletons



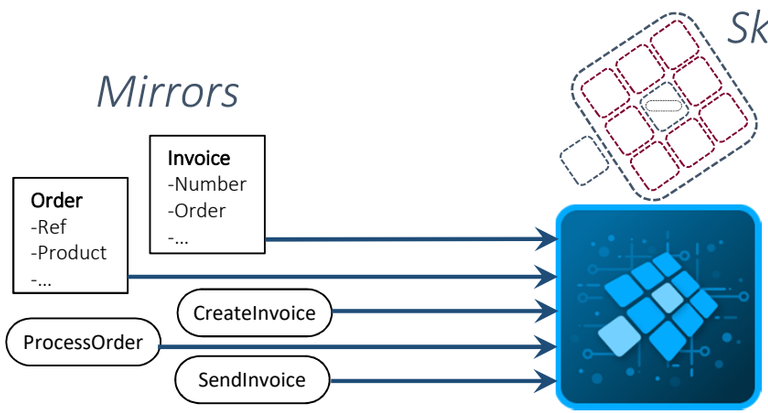



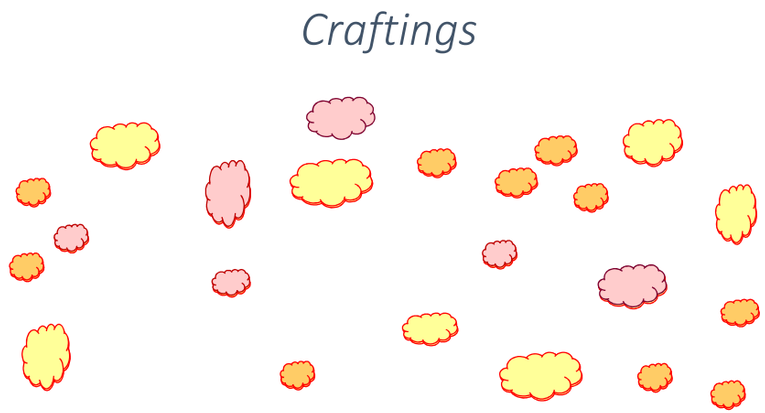



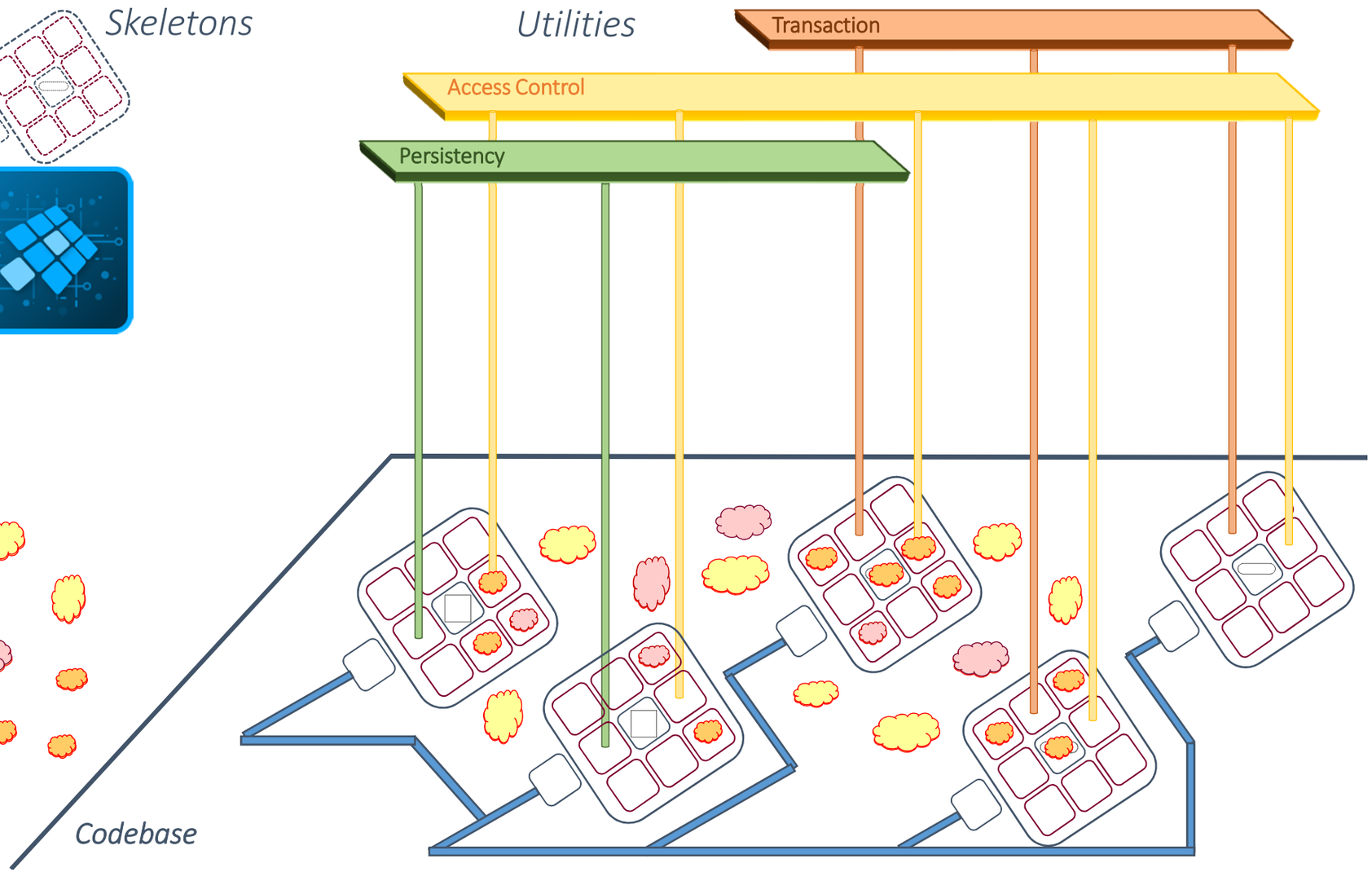
# Change Dimension 3: The Utilities



# Change Dimension 4: The Craftings



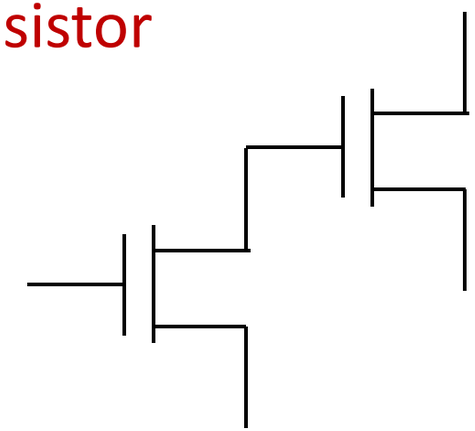
- Automatic Programming
- Toward Automatic Regeneration
- Creating Meta-Circular Regeneration
  - The Need for Meta-Circularity
  - Closing the Automatic Meta-circle
  - Exploring an Implementation
- On Meta-Programming Interfaces
- Concluding Facts and Thoughts
- Questions and Discussion



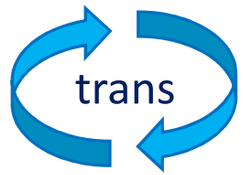
# Overview

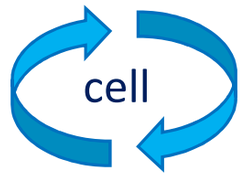
# The Power of Circularity

- A transistor is switched by a transistor
- A cell is produced by a cell



- Enables rapid evolution
  - Single point of progress
    - Better transistor → better circuits
    - Improved cell → improved life forms
  - Collapses/shortcuts the design cycle
    - Even *positive feedback* or *resonance*



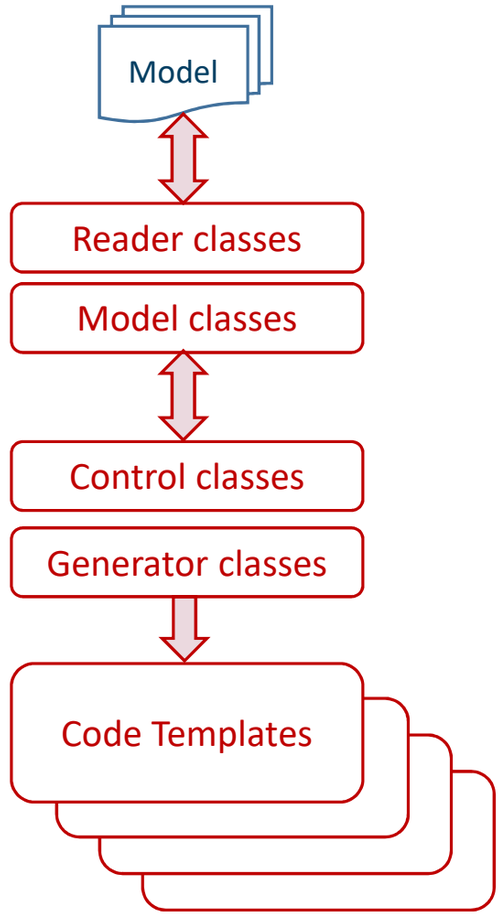



# Meta-Circularity in Software Engineering

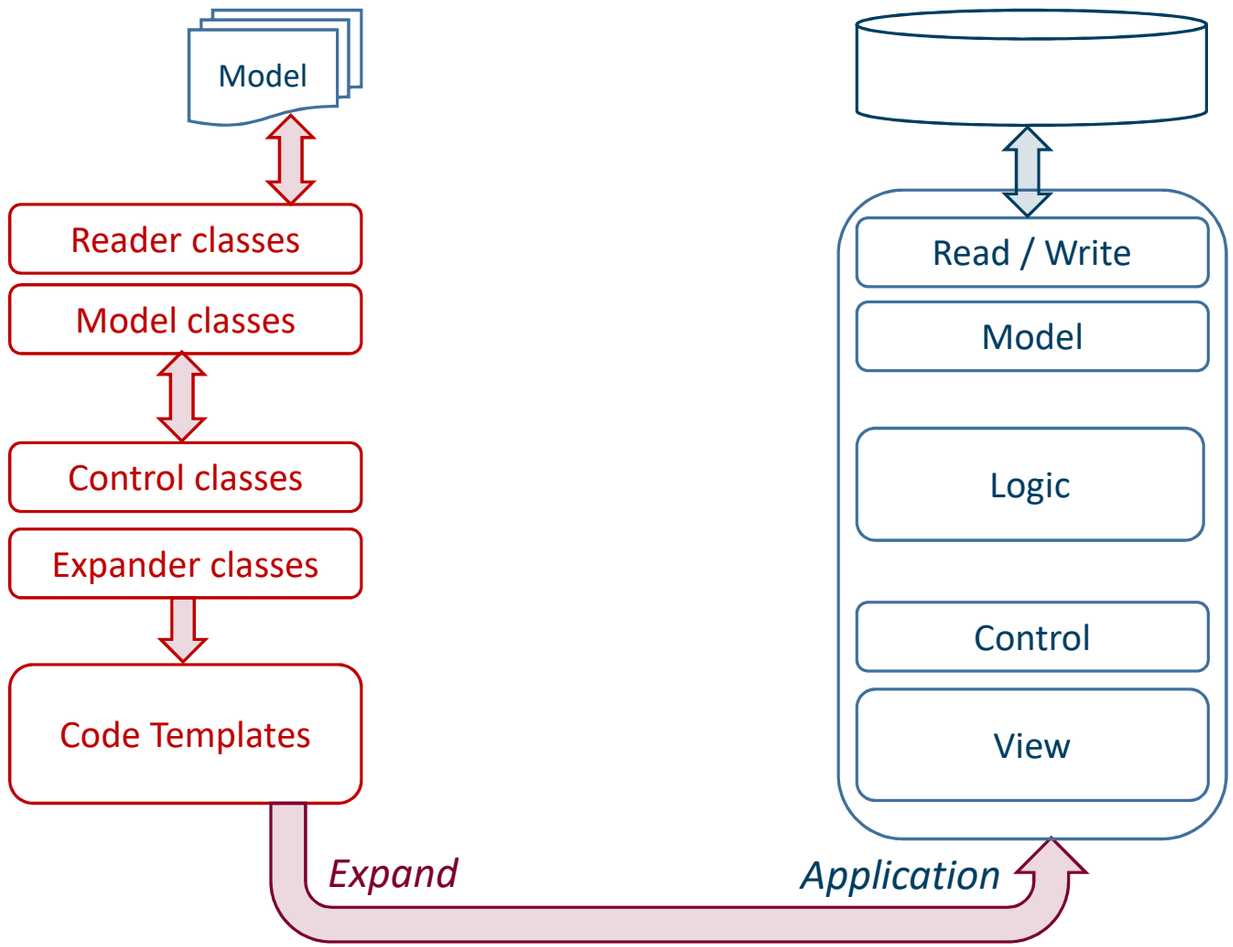
- Associated with *Homoiconicity*:
  - Was coined in 1965, by Mooers & Deutsch (TRAC), and traces back to McIlroy
  - Definitions use concepts like *"code as data"* and *"program structure similar to its syntax"*, and are often considered to be a bit vague and controversial
  - The concept is often associated with LISP
- The term *Meta-Circular Evaluator*:
  - Was coined by John Reynolds in 1972 for an interpreter
  - It defines each feature of the defined language by using the corresponding feature of the defining language.
- There is nevertheless a widespread belief that this kind of properties *increase the abstraction level and therefore the productivity*

# Why Meta-Circularity in Meta-Programming ?

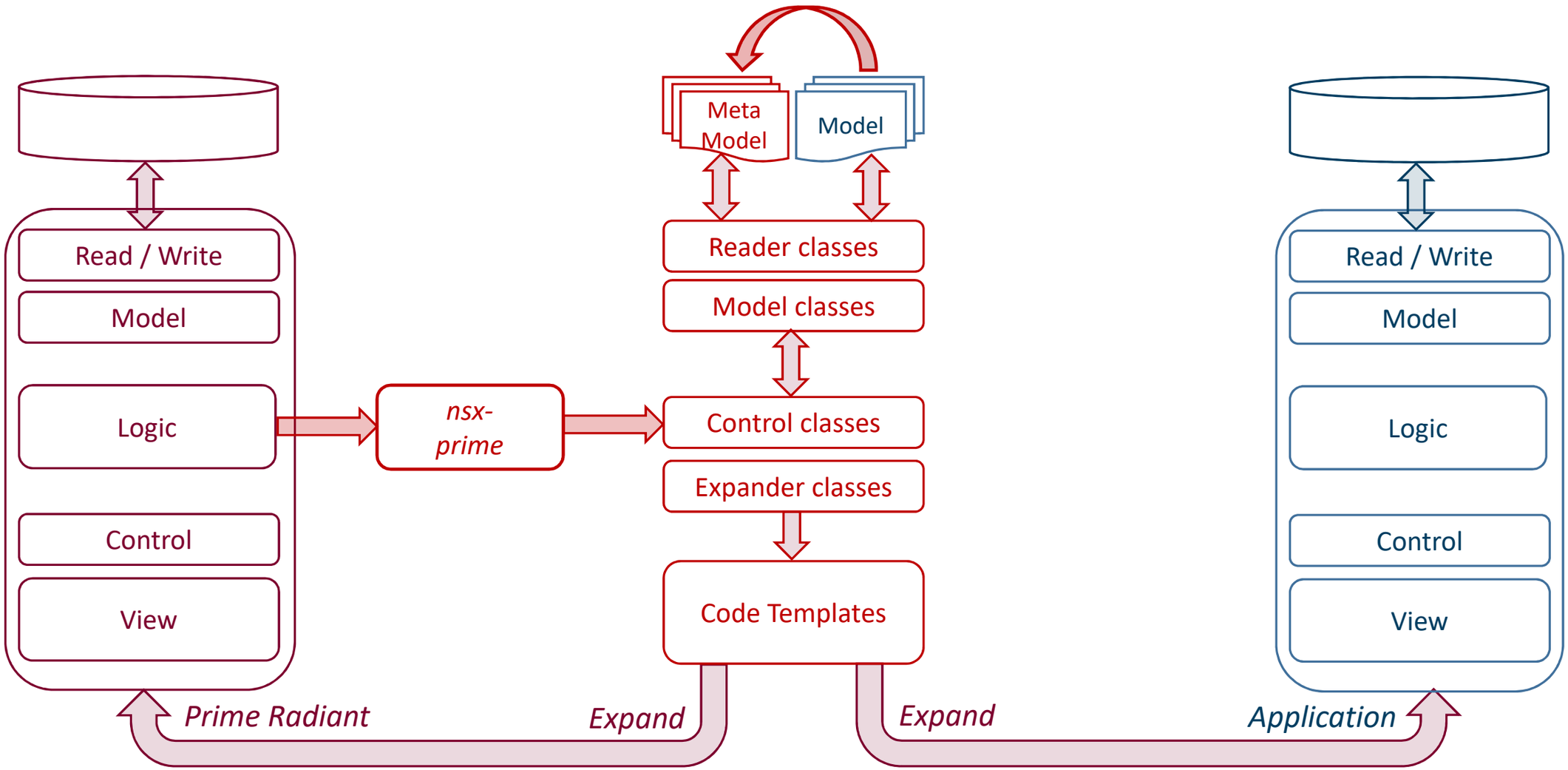
- You also have to maintain the meta-code
  - Consists of several modules
  - Is in general not trivial to write
- Will face growing number of implementations:
  - Different versions
  - Multiple variants
  - Various technology stacks
- Will have to adapt itself to:
  - Evolutions of its underlying technology
    - Which even may become obsolete
- *Meta-Circularity: meta-code that (re)generates itself*



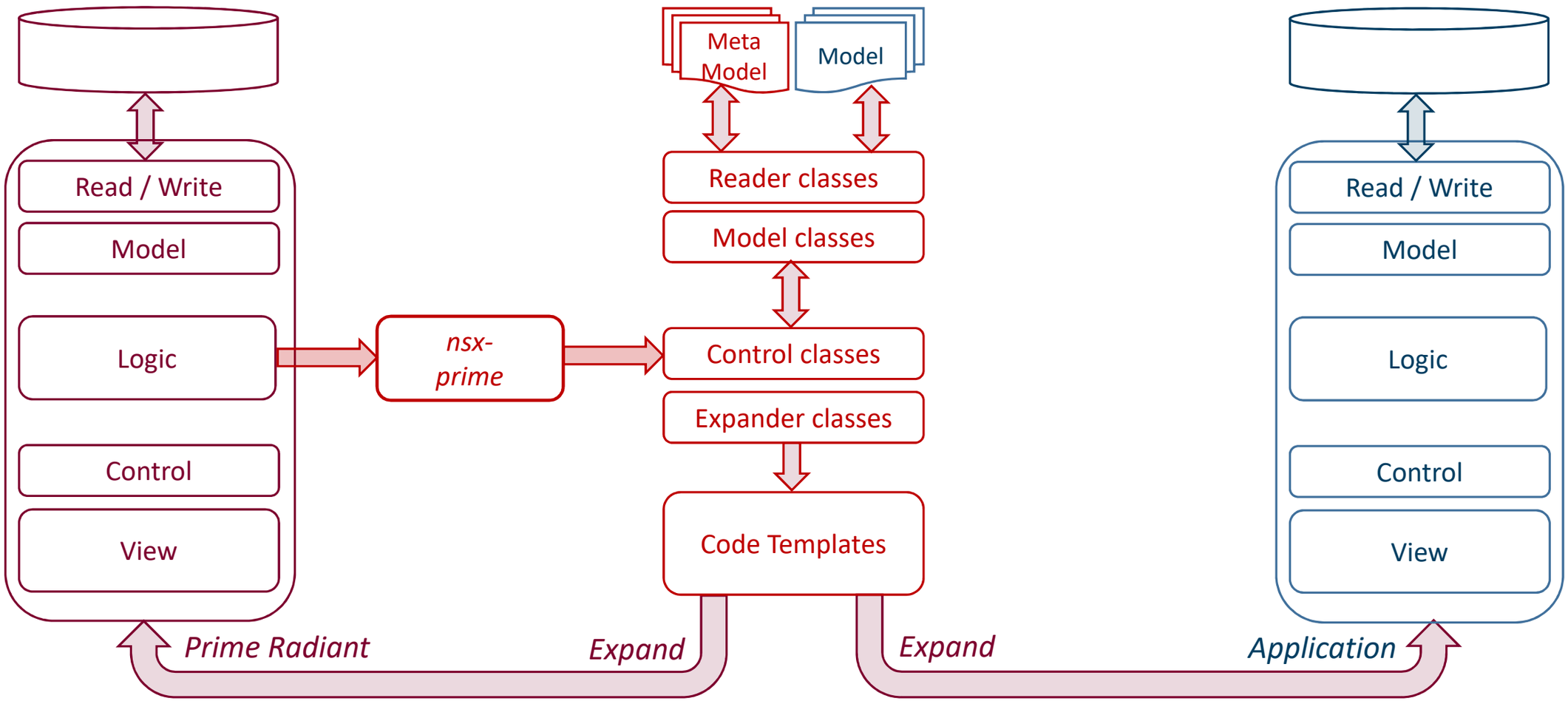
# Closing the Meta-Circle : Phase 1



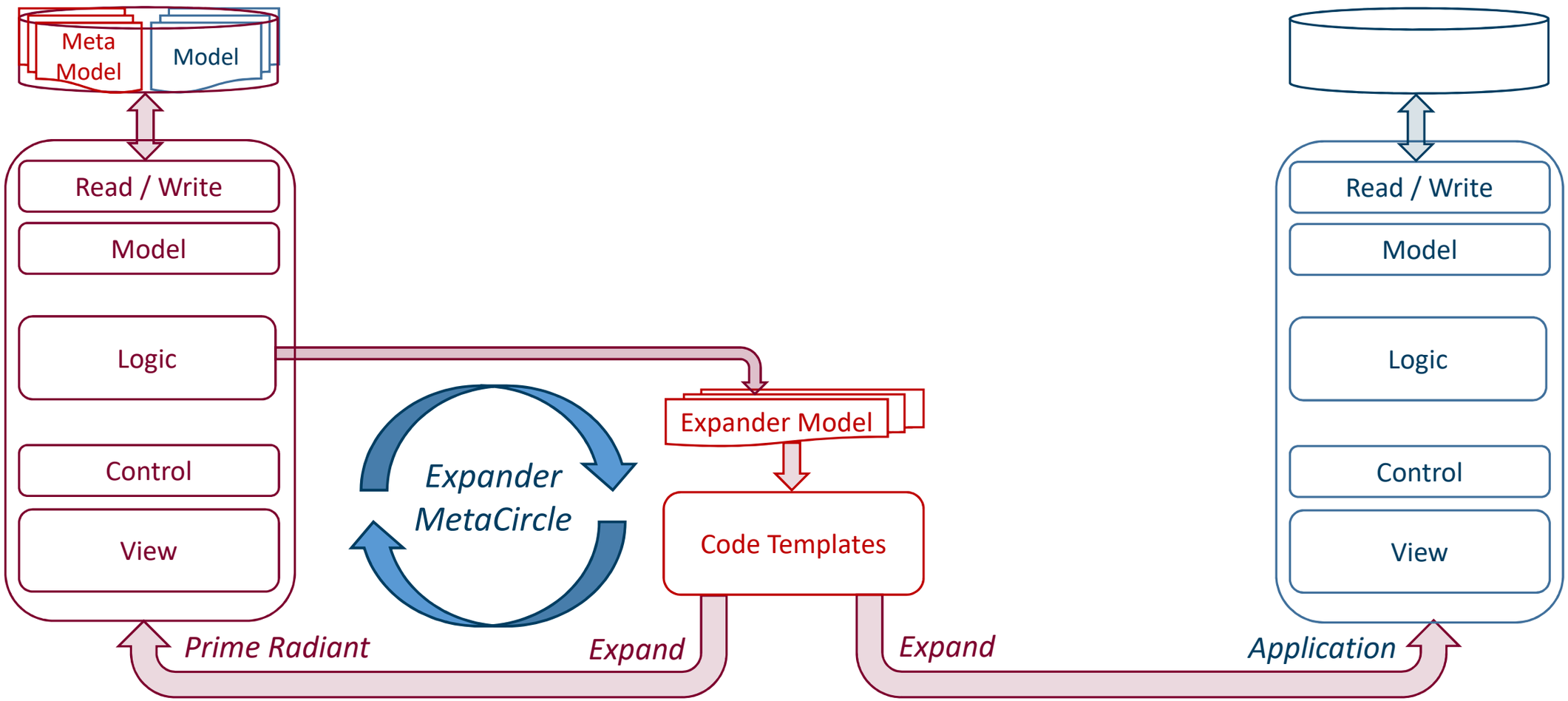
# Closing the Meta-Circle : Phase 2



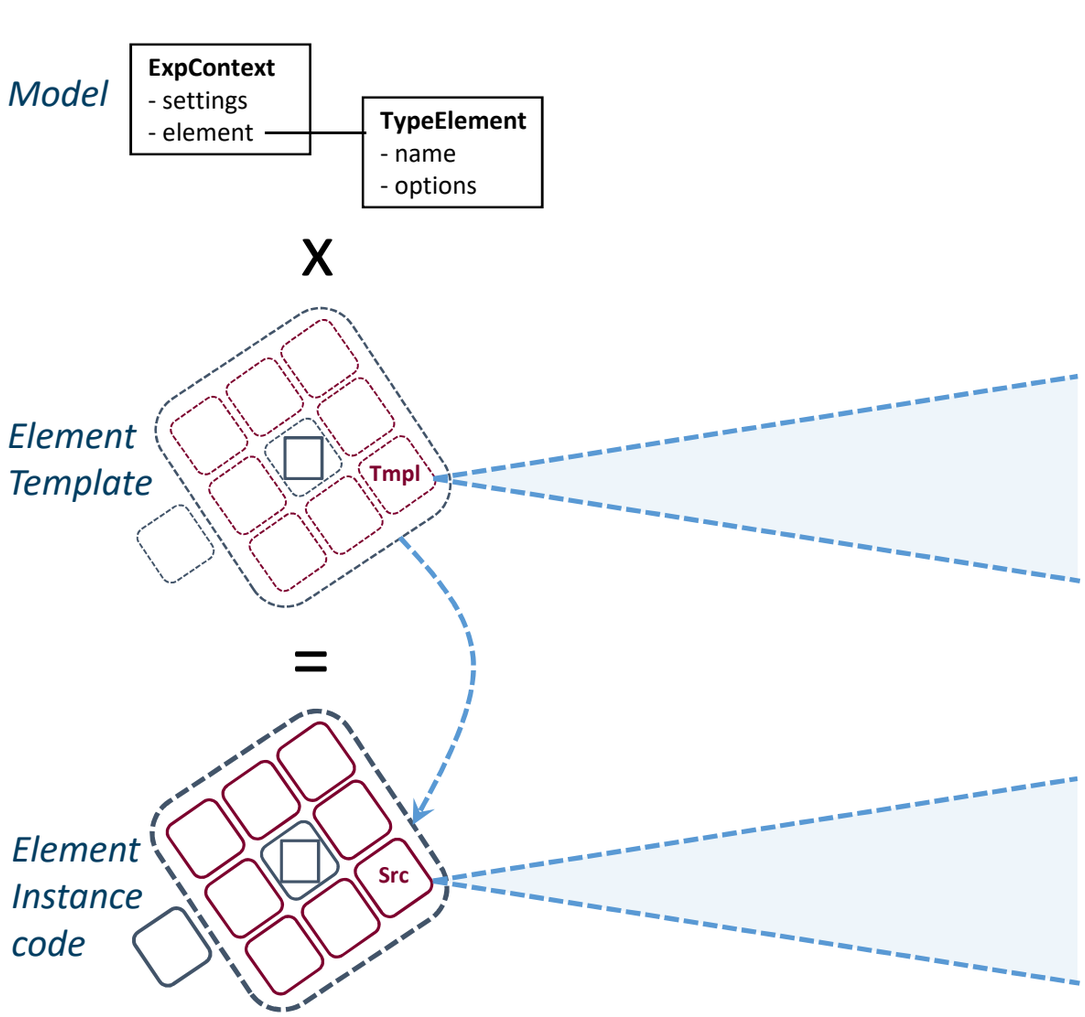
# Closing the MetaCircle : Phase 3

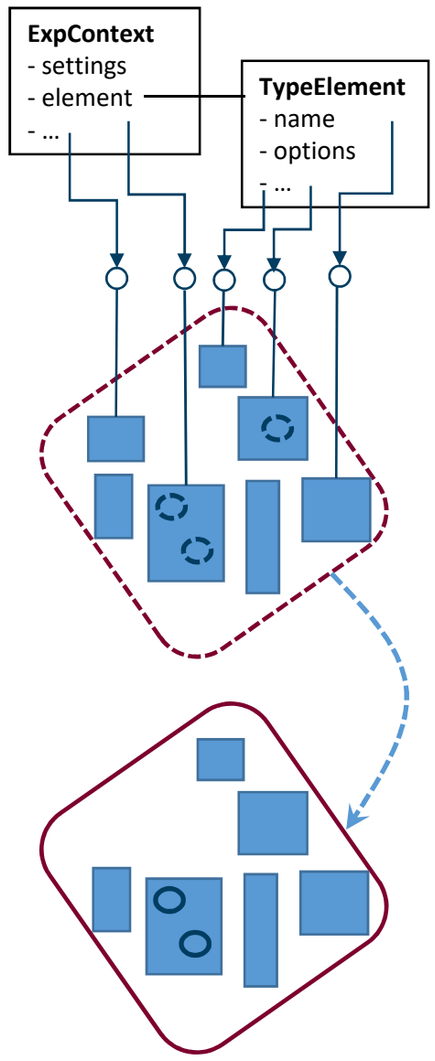


# Closing the MetaCircle : Phase 3



# Artifact = Expansion(Template, Model)




*Settings.xml*
*TypeElement.xml*

*ArtifactExpander*
*Mapping.xml*

*ArtifactExpander.xml*
*ArtifactExpander.stg*

*Artifact.source*

```
/*========== Default constructor ==========*/

public EmployeeDetails() {
    this.mId = new Long();
    // anchor:default-constructor-initialization:start
    this.mName = "";
    this.mContractType = "";
    this.mSourcer = new DataRef();
    this.mEmployeeType = new DataRef();
    this.mDateOfBirth = new Date();
    this.mEmail = new Email();
    this.mMobile = "";
    this.mPrivateContact = new DataRef();
    // anchor:default-constructor-initialization:end

    // anchor:custom-default-constructor:start
    // anchor:custom-default-constructor:end
}
```

# Artifact = Expansion(Template, Model)

*TaskElement.xml*

```
<expander name="TaskInterfaceExpander" xmlns="http://nsx.
  <packageName>net.democritus.expander.common.taskElement
  <layerType name="SHARED_LAYER"/>
  <technology name="COMMON"/>
  <sourceType name="JAVA"/>
  <elementTypeName>TaskElement</elementTypeName>
  <artifactName>$taskElement.name$.java</artifactName>
  <artifactPath>$componentRoot.directory$/$artifactSubFol
  <isApplicable>true</isApplicable>
  <active value="true"/>
  <anchors/>
  <customAnchors>
    <customAnchor name="custom-imports"/>
    <customAnchor name="custom-methods"/>
  </customAnchors>
</expander>
```

*ArtifactExpander.xml*

```
<taskElement name="PrimeRadiantUpdater">
  <packageName>net.democritus.settings</packageName>
  <targetClass>net.democritus.settings.NsfBaseDetails</targetClass>
  <targetElement component="elements" name="NsfBase"/>
  <paramClass/>
  <description/>
  <taskElementType name="Updater"/>
  <transactionType name="noTransaction"/>
  <taskOptions>
    <taskOption name="PrimeRadiantUpdater:includeDelegation">
      <value/>
      <taskOptionType name="includeDelegation"/>
    </taskOption>
    <taskOption name="PrimeRadiantUpdater:includePerform">
      <value/>
      <taskOptionType name="includePerform"/>
    </taskOption>
    <taskOption name="PrimeRadiantUpdater:includeRemoteAccess">
      <value/>
      <taskOptionType name="includeRemoteAccess"/>
    </taskOption>
  </taskOptions>
</taskElement>
```

# Artifact = Expansion(Template, Model)

*ArtifactExpanderMapping.xml*

```
<?xml version="1.0" encoding="UTF-8" ?>
<mapping xmlns="http://nsx.normalizedsystems.org/201806/expanders/mapping'
  <let name="helper" eval="new net.democritus.expander.common.taskElement

  <!--EXPANSION CONTEXT-->
  <value name="componentName" eval="expansionContext.componentExpansionCo
  <value name="logicSecurity" eval="taskElement.component.getOption('useL
  <value name="useLocalEjbOnly" eval="globalOptionSettings.beanInterfaceP
  <value name="useRemoteEJB" eval="not globalOptionSettings.beanInterface

  <!--TASK ELEMENT-->
  <value name="class" eval="classBuilder.from(taskElement)"/>
  <value name="includePerform" eval="taskElement.getOption('includePerfor
  <value name="includeDelegation" eval="taskElement.getOption('includeDel
  <value name="hasResultClass" eval="taskElement.getOption('hasResultClas
  <value name="targetClass" eval="classBuilder.from(taskElement.targetCla
  <value name="targetElementClass" eval="classBuilder.from(targetDataElem

  <value name="includePerformOnTarget"
         eval="taskElement.getOption('includePerform').defined
               and !isDetailsOnly
               and targetProjection.defined"/>

  <conditionalValue name="resultClass">
    <option if="taskElement.getOption('isBranchingTask').defined"
            eval="classBuilder.from(taskElement.targetElement, 'State')"
    <option if="taskElement.getOption('hasResultClass').defined"
            eval="classBuilder.from(taskElement.getOption('hasResultClass
    <defaultOption eval="classBuilder.from('Void')"/>
  </conditionalValue>
```

*ArtifactExpander.stg*

```
base() ::=<<
package <class.packageName>;

// <expanderComment>

import net.democritus.sys.TaskPerformer;
<if(paramClass&&!paramClass.packageName.empty)>
import <paramClass.qualifiedName>;
<endif>
<if(resultClass&&!resultClass.packageName.empty)>
import <resultClass.qualifiedName>;
<endif>
import <targetClass.qualifiedName>;

// anchor:custom-imports:start
// anchor:custom-imports:end

/**
 * Interface to access the implementation of the task element <class.className>.
 */

public interface <class.className> extends TaskPerformer\<<resultClass.className>,<targetClas

<if(includeParameters)>
    public void setParameters(<paramClass.className>Details <paramClass.varName>Details);
<endif>
   // anchor:custom-methods:start
   // anchor:custom-methods:end

}
>>
```

- Automatic Programming
- Toward Automatic Regeneration
- Creating Meta-Circular Regeneration
- On Meta-Programming Interfaces
  - Need for Meta-Level Interfaces
  - Exploring Two-Sided Interfaces
- Conclusions and Discussion
- Questions



# Overview

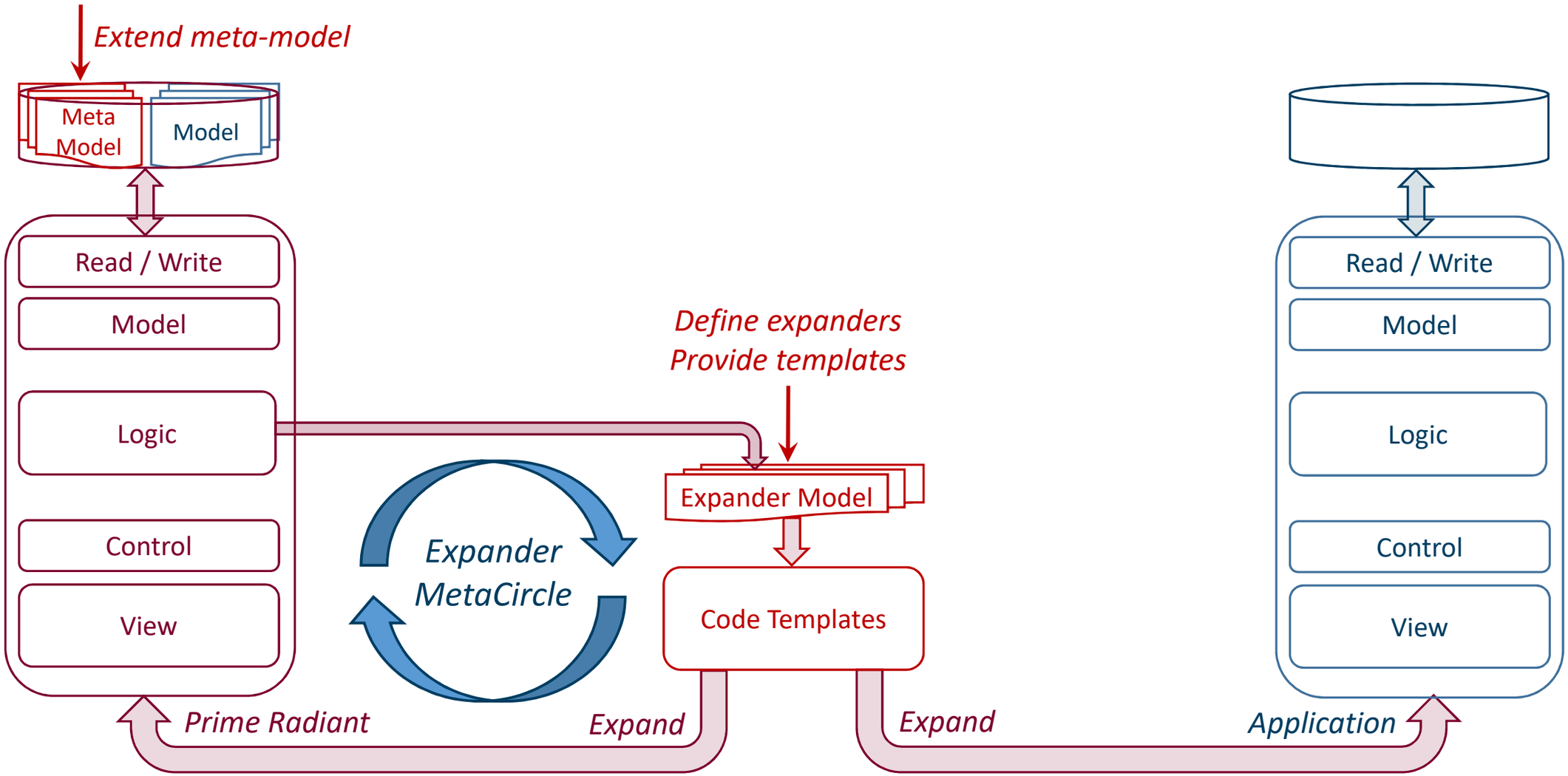
# Need for Meta-Level Interfaces

- Programming interfaces enable *scalable collaboration*:
  - Within companies
  - Across companies
  - In open source communities
- Scalable collaboration is needed in software for:
  - Rich application offering (desktop applications, apps, …)
  - Convenient hardware support (drivers for modems, screens, disks, …)
- Defining *meta-level interfaces is still a subject of research* in 2019:
  - Novel conceptual model for the systematic reuse of textual DSLs [Wortmann]
  - An intermediate representation to be used for code generation [Gusarovs]

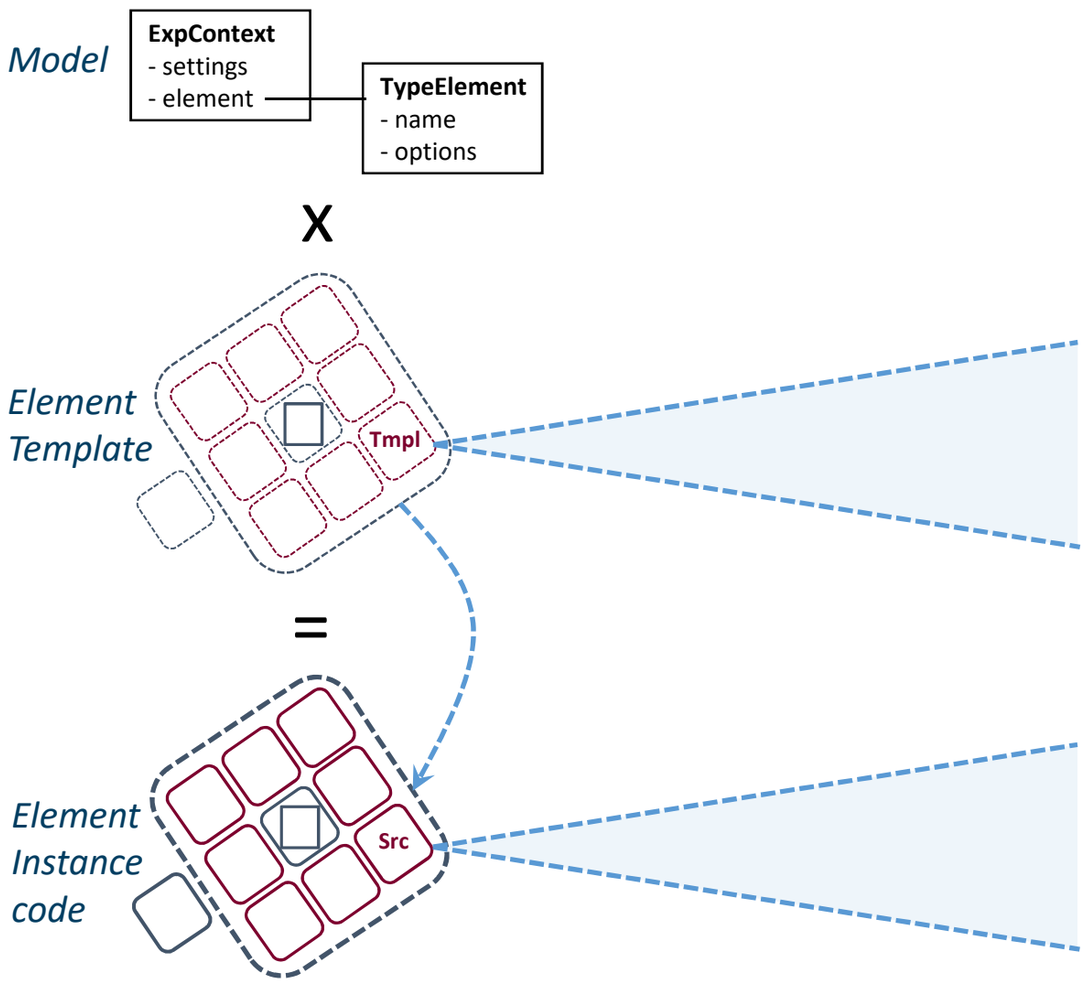
# Two-Sided Meta-Level Interfaces

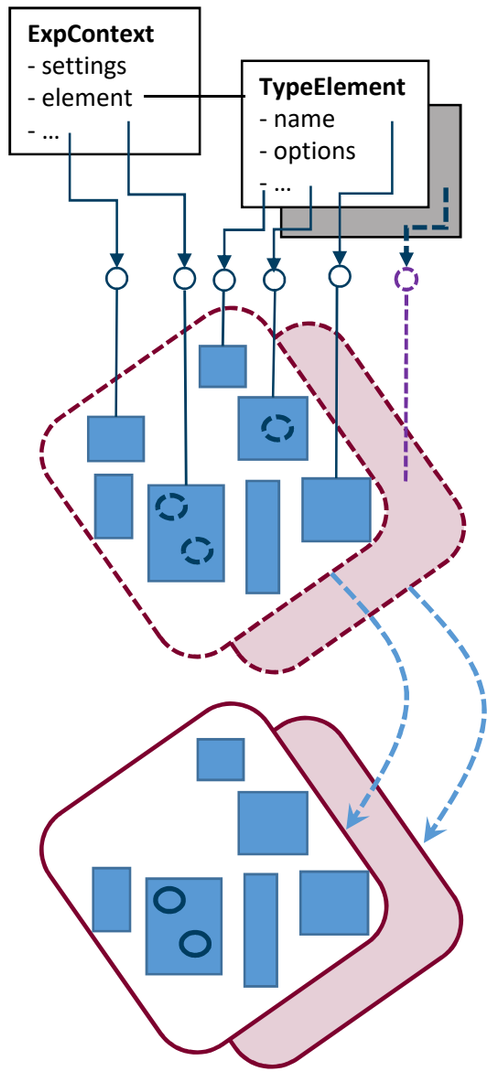
- Automatic programming performs a *transformation*
  - From domain and/or intermediate models
  - To code generators and programming code
- Need to define open *interfaces at both ends*
  - To add or extend domain models
  - To add or replace code generators
- The proposed meta-circular structure
  - Simplifies the definition of the interfaces
  - Avoids the *non-scalable burden* on the meta-code
    - To integrate, or at least accommodate, ever more extensions at both ends

# Closing the Meta-Circle : Expander API



# Expander API: Model/Mapping/Template




*Settings.xml*
*TypeElement.xml*

*Artifact*
*ExpanderMapping.xml*

*ArtifactExpander.xml*
*ArtifactExpander.stg*

*Artifact.source*

```
/*========== Default constructor ==========*/
public EmployeeDetails() {
    this.mId = new Long();
    // anchor:default-constructor-initialization:start
    this.mName = "";
    this.mContractType = "";
    this.mSourcer = new DataRef();
    this.mEmployeeType = new DataRef();
    this.mDateOfBirth = new Date();
    this.mEmail = new Email();
    this.mMobile = "";
    this.mPrivateContact = new DataRef();
    // anchor:default-constructor-initialization:end

    // anchor:custom-default-constructor:start
    // anchor:custom-default-constructor:end
}
```

- Automatic Programming
- Toward Automatic Regeneration
- Creating Meta-Circular Regeneration
- On Meta-Programming Interfaces
- Concluding Facts and Thoughts
- Questions and Discussion



# Overview

# (Re)Generated Code in Production *(or ½)*

- Enterprise applications (JEE)
  - Budget follow-up tool
  - Master thesis evaluation
  - Diplomatic card services
  - Data centre management
  - Solar panels monitoring
  - Beverage product lifecycle
  - Energy datahub management
  - Real estate estimation tool
  - IoT data inflow engine
  - Privacy and digital vault
  - ….

| Applications | ± 20 |
|---|---|
| Components | 43 |
| Data elements | 1546 |
| Attributes | 7094 |
| Task elements | 535 |
| Flow elements | 133 |

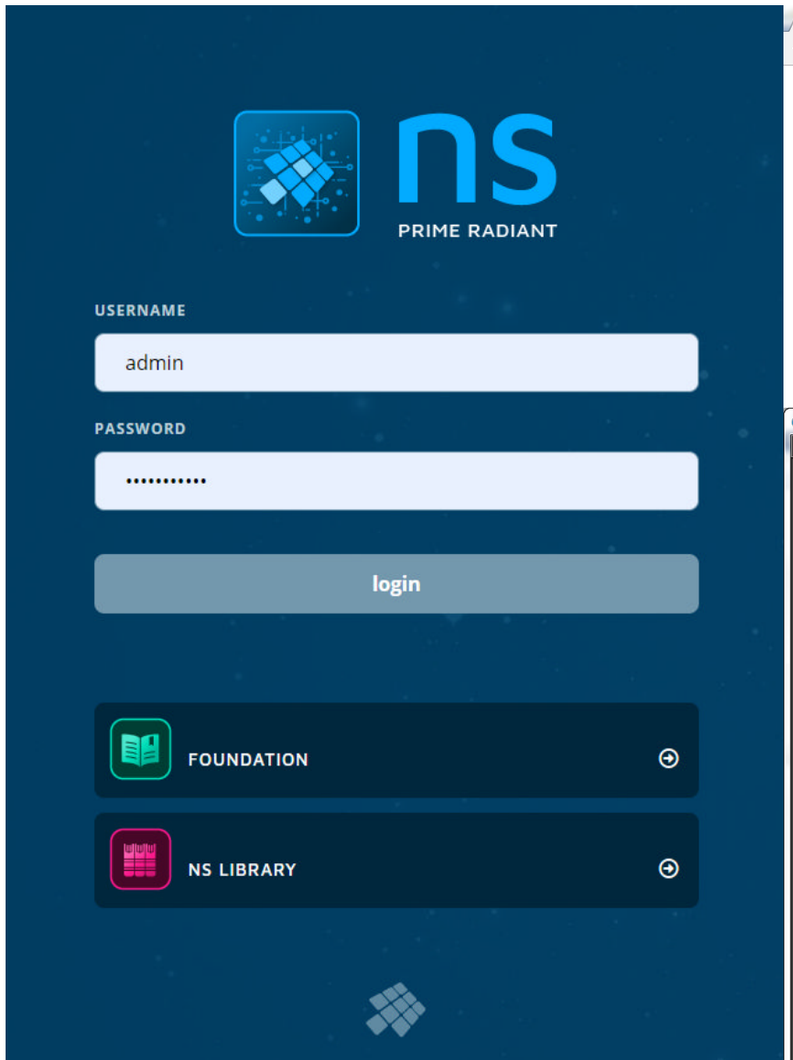
| Extensions | Total |
|---|---|
| Data layer | 6 |
| Logic layer | 1731 |
| Shared layer | 250 |
| Proxy layer | 5 |
| Control layer | 218 |
| View layer | 1186 |

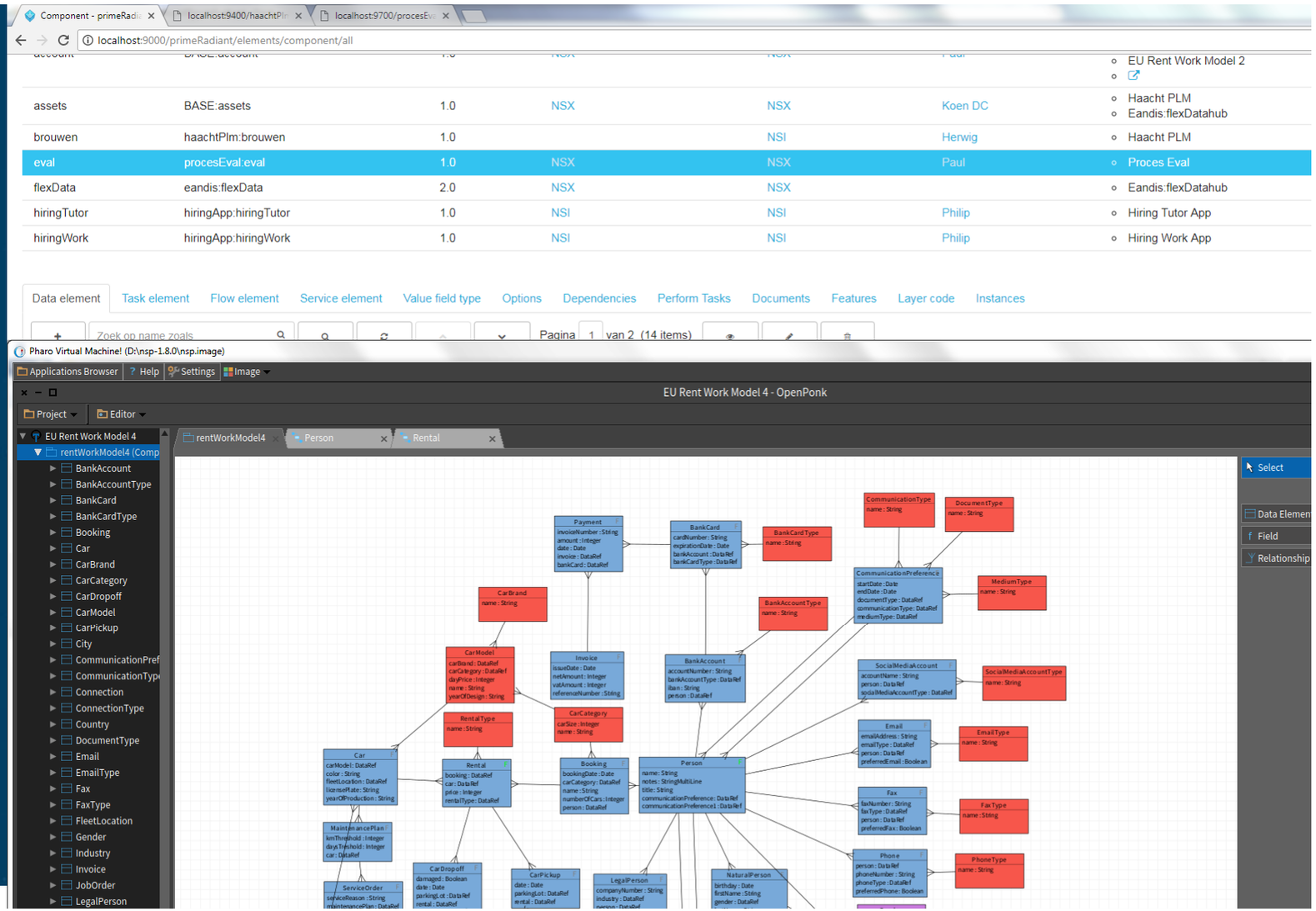
| Skeletons | Total |
|---|---|
| Classes | ± 40.000 |

| Insertions | Total |
|---|---|
| Data element | 1436 |
| User connector | 146 |
| Task element | 401 |
| Flow element | 0 |

*Data 2018*

# (Re)Generated Meta-Application in Use

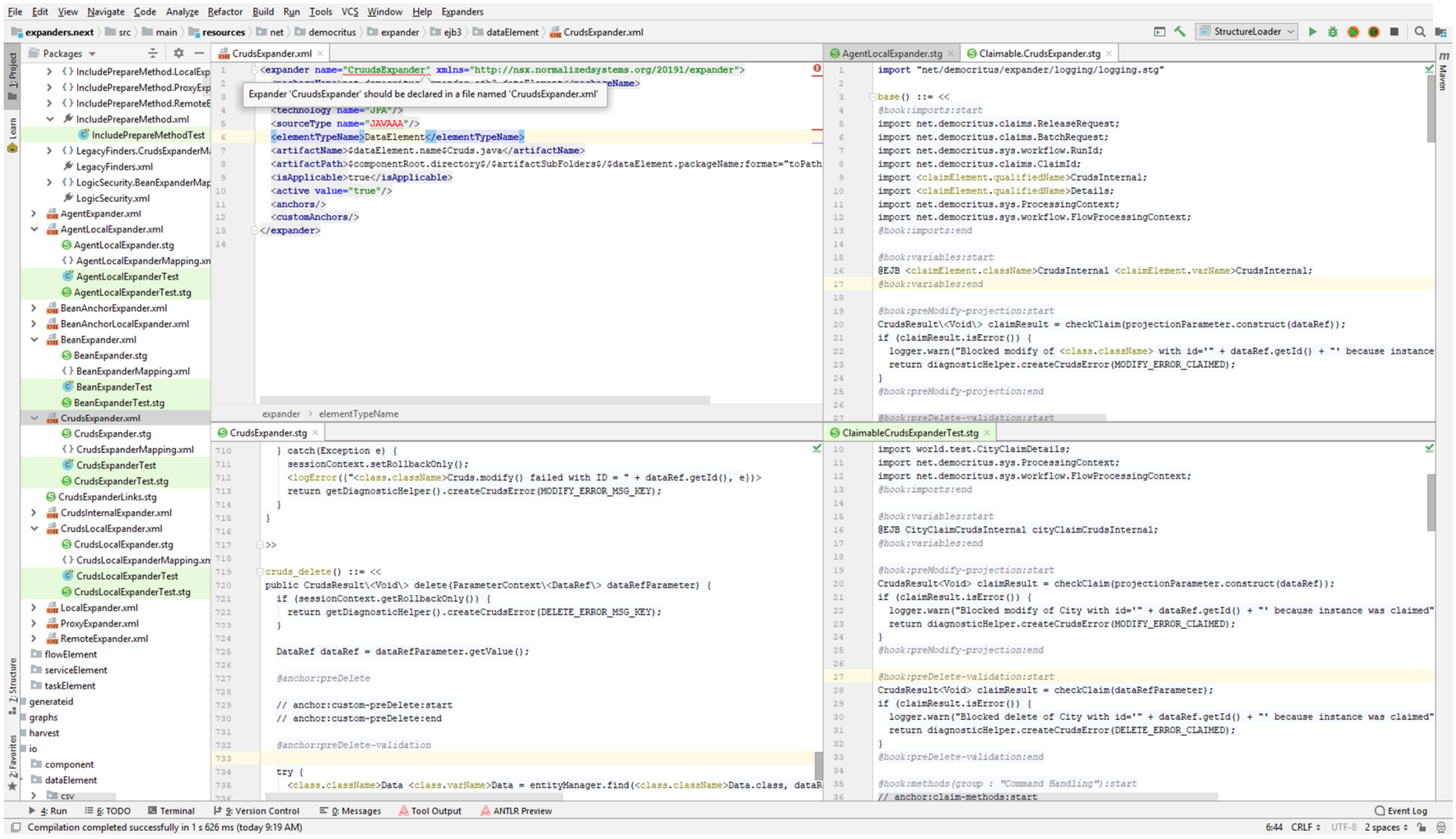


# (Re)Generated Meta-Application in Use

- Prime Radiant, supporting:
  - CRUDS for *87 data elements* of the meta-model
  - Invocation of operations using *31 task elements*
  - Expansion, build, and deployment of applications
- Expander or code generation software:
  - Has been available for 8 years
  - Has been refactored to meta-circular architecture in 2019
  - Contains *181 expanders* or code generators for JEE application stack
- New expander bundles are being released:
  - REST/Swagger bundle of *39 expanders* by internal developer
  - *Additional bundles by one of the early customers*

# Releasing a Expander Developer Kit



# Starting exchange.stars-end.net


## NS Foundation Expanders

The foundational expander bundle containing the complete set of expanders to create application skeletons based on Normalized Systems Theory.

Download

Reference manual | Feature manual

## CPaaS REST Expanders

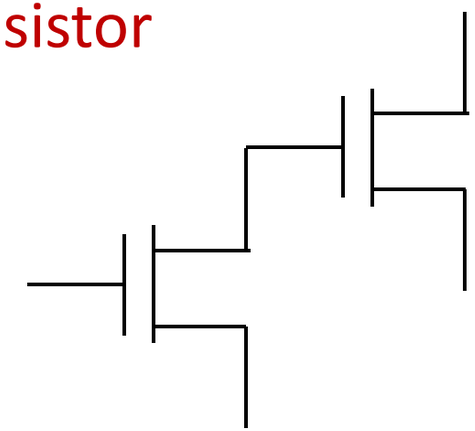
An expander bundle providing additional functionality for REST interfaces using Swagger technology, developed within the Digipolis CPaaS project.

Download

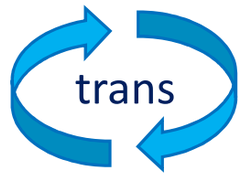
Reference manual | Feature manual

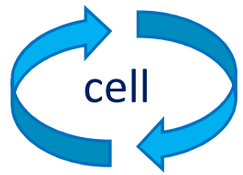
# *Remember:* The Power of Circularity

- A transistor is switched by a transistor
- A cell is produced by a cell

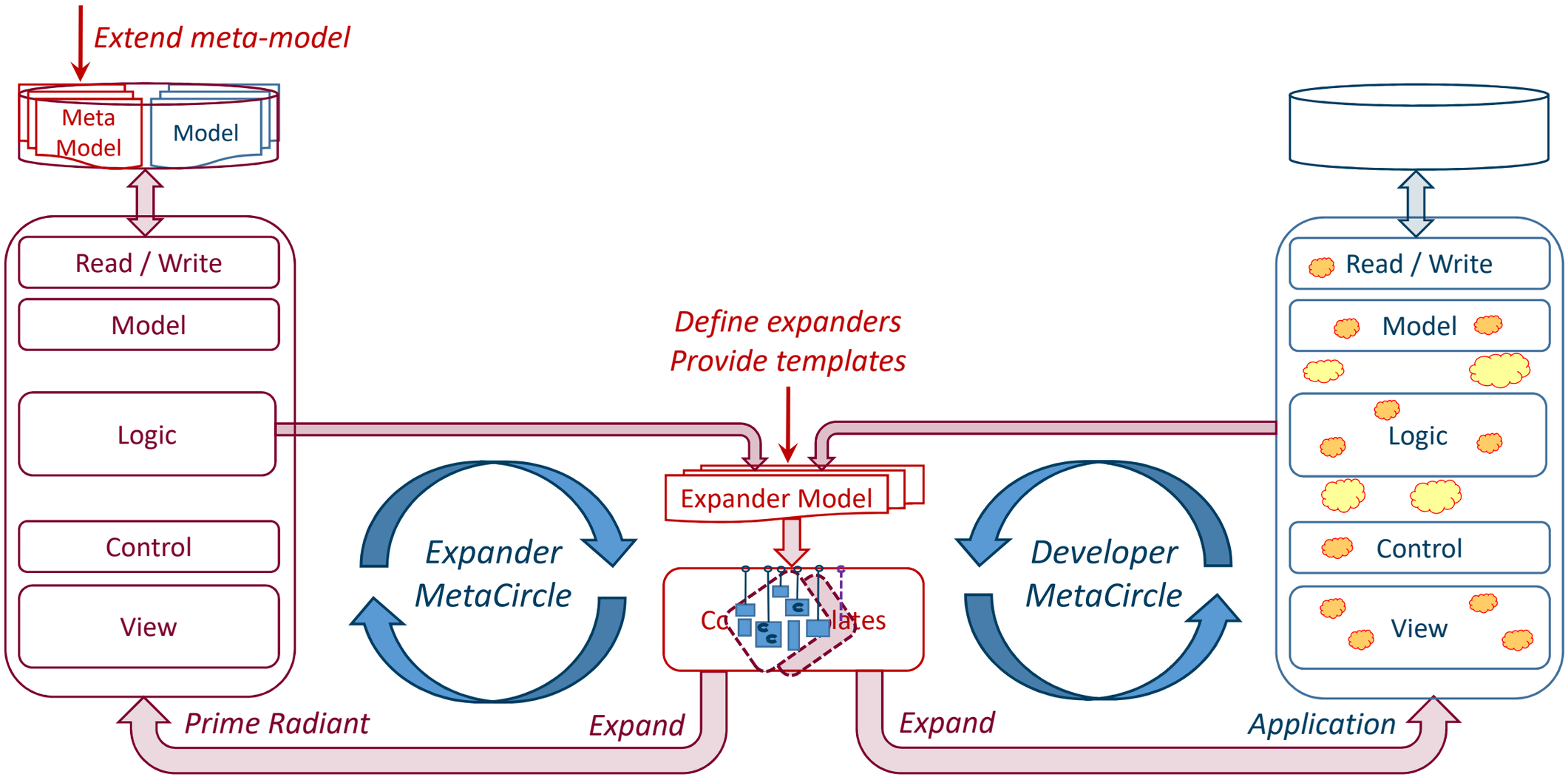


- Enables rapid evolution
  - Single point of progress
    - Better transistor → better circuits
    - Improved cell → improved life forms
  - Collapses/shortcuts the design cycle
    - Even *positive feedback* or *resonance*

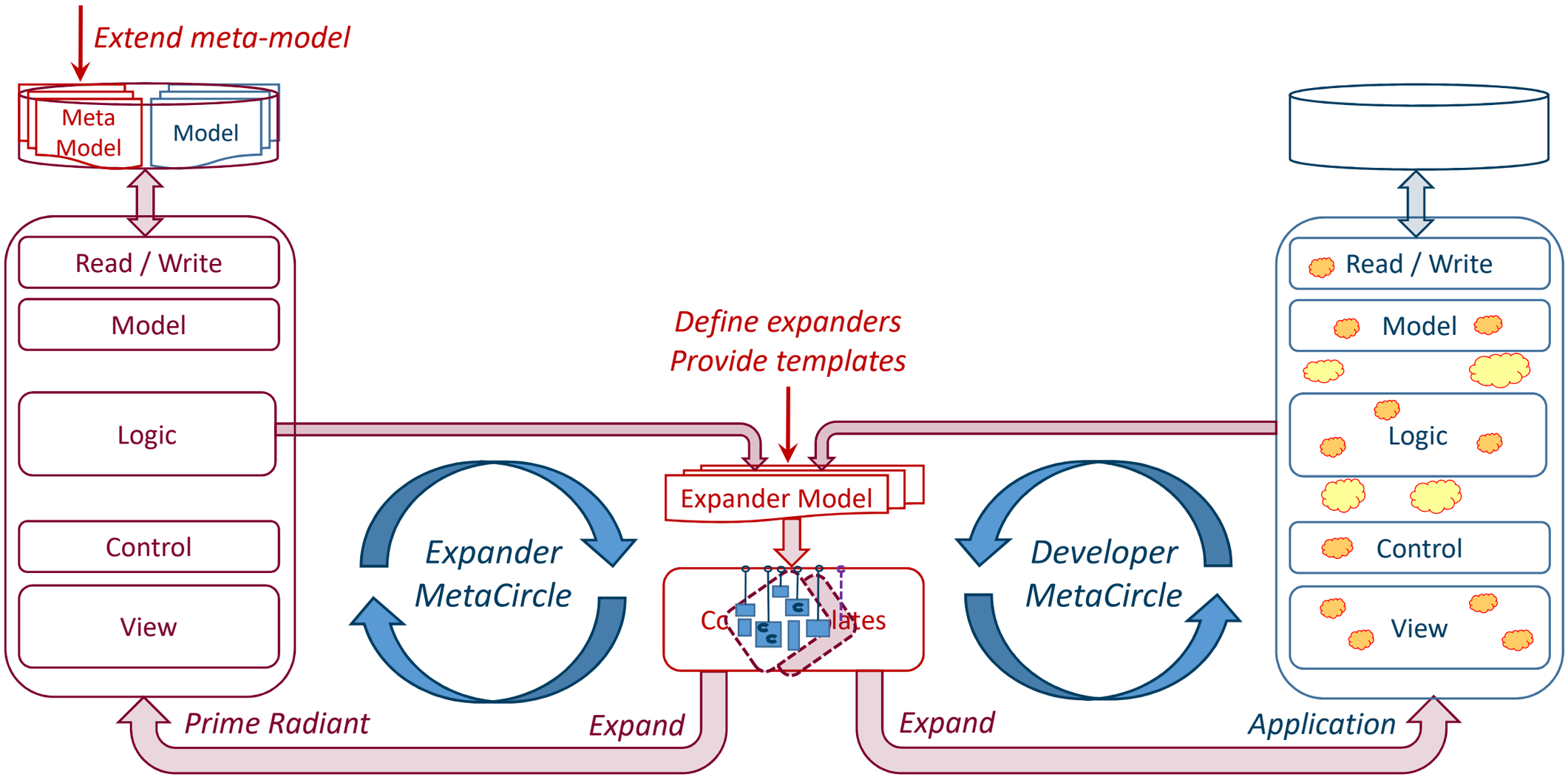


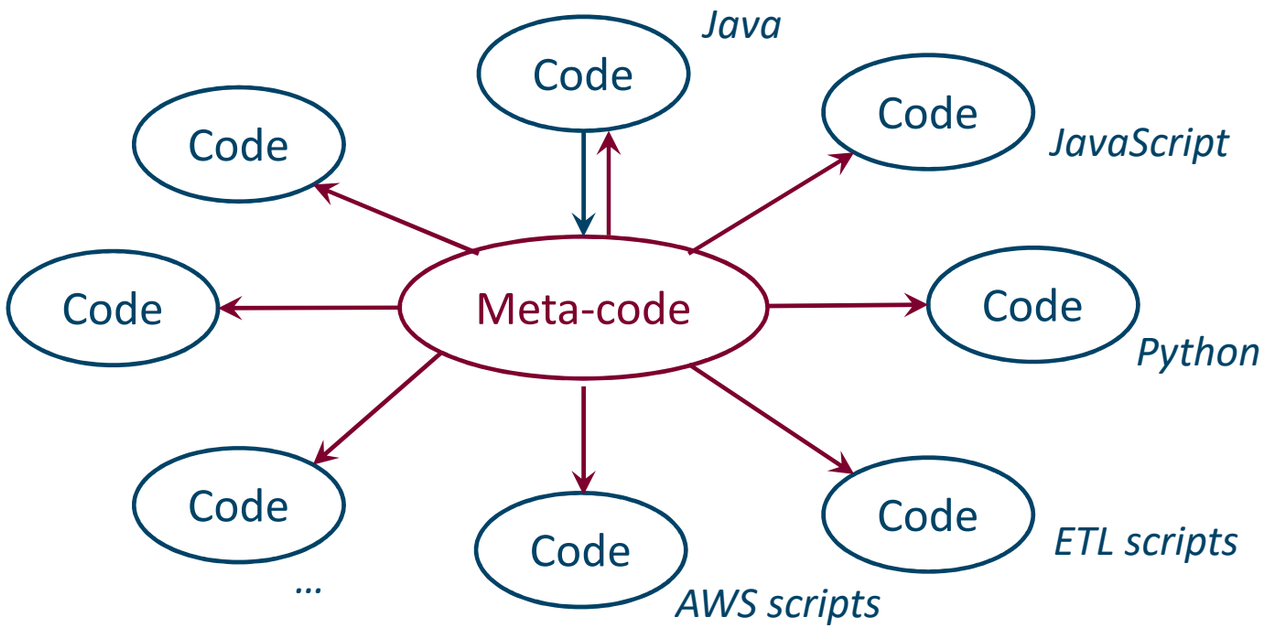
# Closing the Meta-Circle : Resonance



# Closing the Meta-Circle : Resonance



# Closing the Meta-Circle : Resonance




*"Let us turn every programmer into a meta-programmer, and create a meta-circular instability or resonance effect."*

ADVANCING AUTOMATIC PROGRAMMING

# Overview

- Automatic Programming
- Toward Automatic Regeneration
- Creating Meta-Circular Regeneration
- On Meta-Programming Interfaces
- Concluding Facts and Thoughts
- Questions and Discussion


QUESTIONS ?
herwig.mannaert@uantwerp.be