# Model-Driven Development (MDD) of Distributed Systems

## OOPSLA 2006, Thursday, Oct 26

**Dr Douglas C Schmidt**
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt
Institute for Software Integrated Systems
Vanderbilt University Nashville, Tennessee

**Markus Völter**
voelter@acm.org
www.voelter.de
Independent Consultant for Software Engineering & Technology
Heidenheim, Germany


OOPSLA'06, October 22–26, 2006, Portland, Oregon, USA.
2006 ACM 06/0010.n

## What We Want You to Learn Today

- Key MDD concepts & what kinds of domains & problems they address
- What are some popular MDD tools & how they work
- How MDD relates to other software tools & (heterogeneous) platform technologies
- What types of projects are using MDD today & what are their experiences
- What are the open issues in MDD R&D & adoption
- Where you can find more information





## CONTENTS







## CONTENTS

## The Road Ahead

CPU & network performance has increased by 3-8 orders of magnitude in past decades

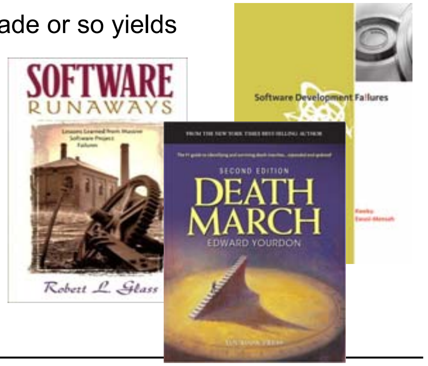

10 Megahertz to 3+ Gigahertz

1,200 bits/sec to 10+ Gigabits/sec

Extrapolating these trends another decade or so yields

- ~100 Gigahertz desktops
- ~100 Gigabits/sec LANs
- ~100 Megabits/sec wireless
- ~10 Terabits/sec Internet backbone

Unfortunately, software quality & productivity hasn't improved as rapidly or predictably as hardware






## Why Hardware Improves So Consistently

Advances in hardware & networks stem largely from R&D on *standardized* & *reusable* APIs & protocols

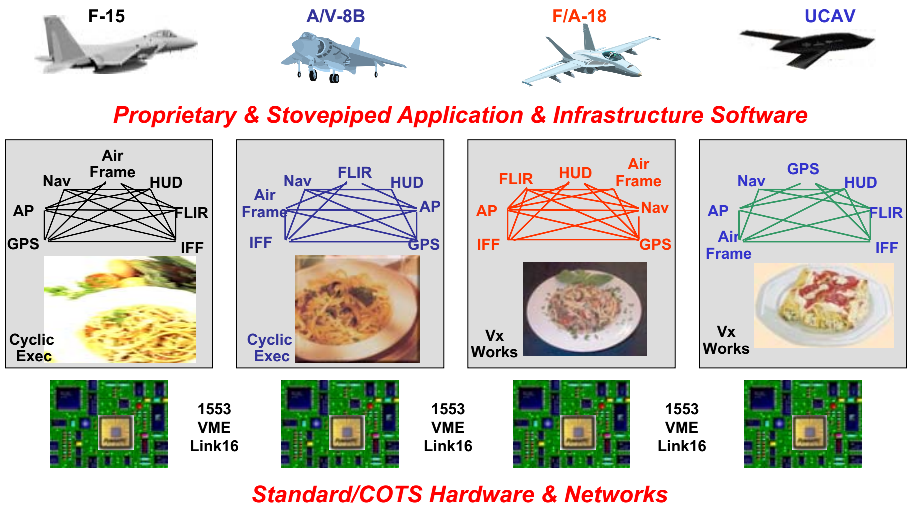

**x86 & Power PC chipsets**

**TCP/IP**



## Why Software Fails to Improve as Consistently

In general, software has not been as standardized or reusable as hardware

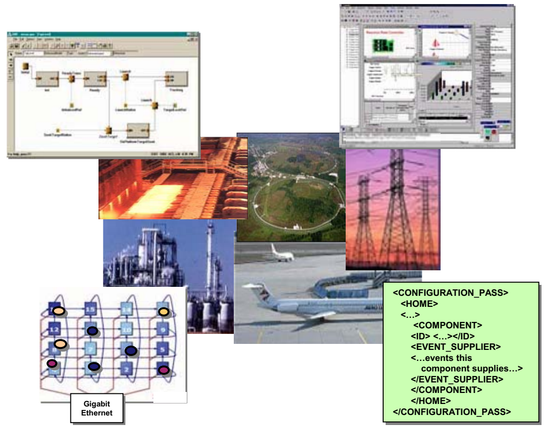


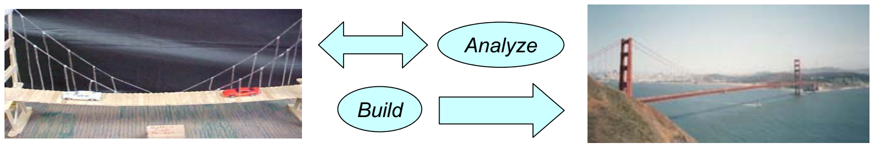



## The Promise

- Develop standardize technologies that:
  1. ***Model***
  2. ***Analyze***
  3. ***Synthesize &***
  4. ***Provision***

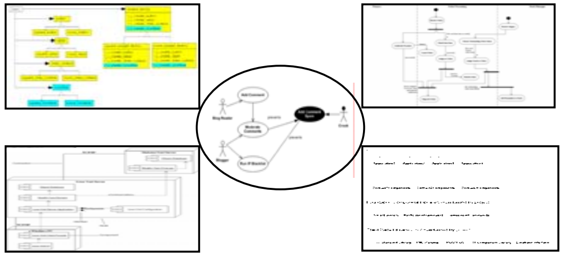
  complex software systems



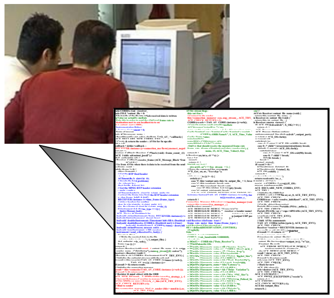



# The Reality

- Architects (sometimes) use UML to express software designs at a high-level



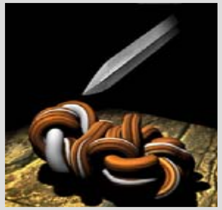
- Developers write & evolve code manually



**We ought/need to be able to do much better than this!**

# Sources of the Problems

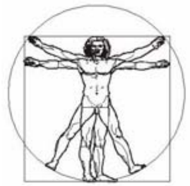
**Technical Challenges**



**Inherent & accidental complexities**

- More automated specification & synthesis of
  - Broader range of target domain capabilities
  - Model interpreters & transformations
  - Static & dynamic quality of service (QoS) properties
- Round-trip engineering from models ↔ source
- Poor support for debugging *at the model level*
- Version control of models *at the model level*

**Non-Technical Challenges**



**Impediments of human nature**

- Organizational, economic, administrative, political, & psychological barriers

**Ineffective technology transition strategies**

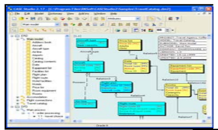
- Disconnects between methodologies & production software development realities
- Lack of incremental, integrated, & *triaged* transitions

www.cs.wustl.edu/~schmidt/reuse-lessons.html

# Key Challenges for Software Developers

**Developers & users of software face challenges in multiple dimensions**

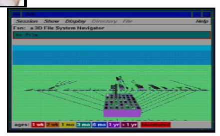
**Logical View**



**Process View**

**Use Case View**

**Physical View**



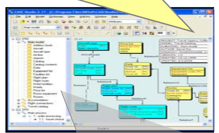
**Development View**



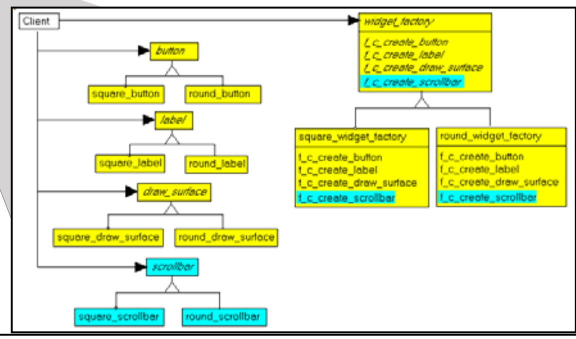
# Key Challenges for Software Developers

Determining units of abstraction for system (de)composition, reuse, & validation

**Logical View**

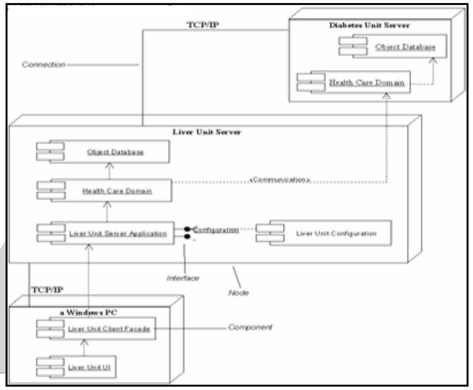


- Popular technologies & tools provide inadequate support for
  - Checking pre-/post-conditions & invariants
  - Specifying & analyzing dependencies
  - Expressing design intent more clearly using domain concepts



# Key Challenges for Software Developers

- Popular technologies & tools provide inadequate support for
  - Configuring & customizing components for application requirements & run-time environments
  - Automated mapping of components onto nodes in target environments




**Physical View**

Integrating/deploying diverse new & reusable application components in a networked environment to ensure end-to-end QoS requirements

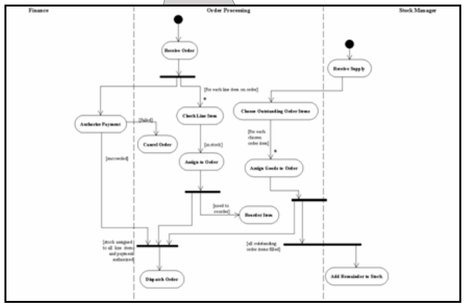


# Key Challenges for Software Developers

- Popular technologies & tools provide inadequate support for
  - Identifying & reducing performance & robustness risks earlier in system lifecycle
  - Satisfying multiple (often conflicting) QoS demands
    - e.g., secure, real-time, reliable
  - Satisfying QoS demands in face of fluctuating/insufficient resources
    - e.g., mobile ad hoc networks (MANETs)

Devising execution architectures, concurrency models, & communication styles that ensure multi-dimensional QoS & correctness of new/reusable components

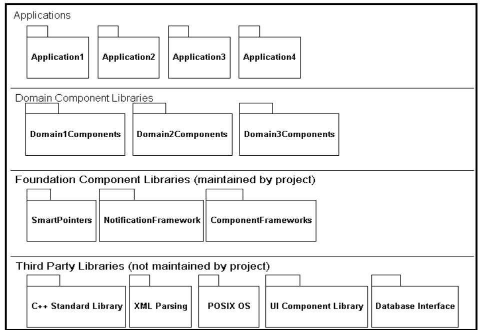
**Process View**

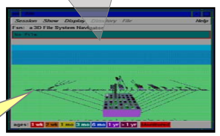




# Key Challenges for Software Developers

- Popular technologies & tools provide inadequate support for avoiding "bloatware", i.e.:
  - *Cyclic dependencies*, which make unit testing & reuse hard
  - *Excessive link-time dependencies*, which bloat the size of executables
  - *Excessive compile-time dependencies*, where small changes trigger massive recompiles



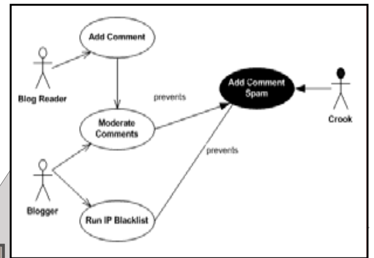
**Development View**

(De)composing systems into reusable modules (e.g., packages, subsystems, libraries) that achieve/preserve QoS properties



# Key Challenges for Software Developers

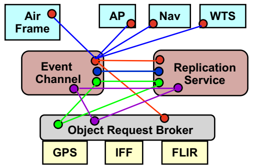
Capturing functional & QoS requirements of systems & reconciling them with other views during evolution




**Use Case View**

- Popular technologies & tools provide inadequate support for
  - Ensuring semantic consistency & traceability between requirements & software artifacts
  - Visualizing software architectures from multiple views

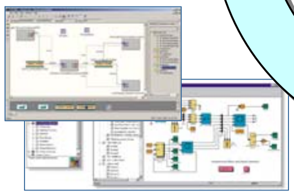
# Promising Solution Approaches

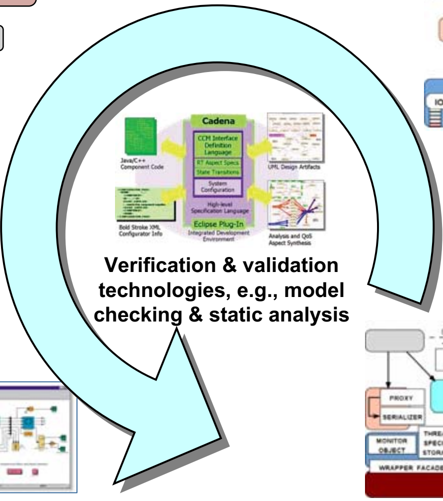



**Multi-faceted Software Development**

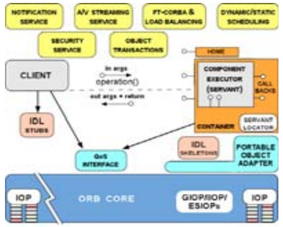
**Cross-cutting Concerns**
- Synchronization
- Persistence
- Memory Management
- Fault Tolerance

**Model-driven development & domain-specific languages**



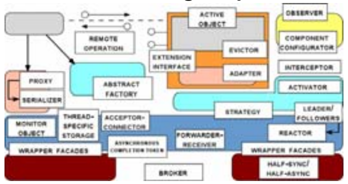



**Verification & validation technologies, e.g., model checking & static analysis**




**Middleware frameworks that integrate multiple QoS properties**

**Formalizing best practices & design expertise**

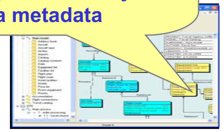


There is no single "silver bullet" technology that resolves all software problems!

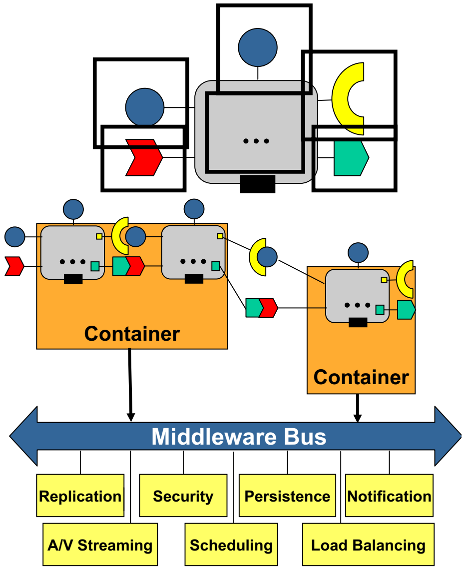
# Promising Solution Approaches

Devising composable abstractions whose interfaces & QoS properties can be specified/analyzed via metadata

**Logical View**

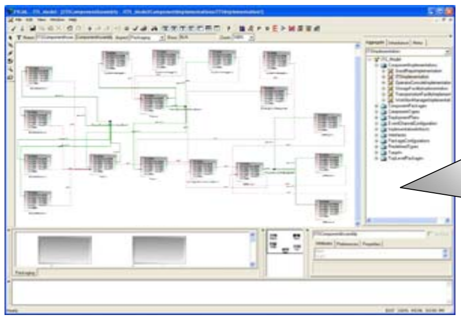


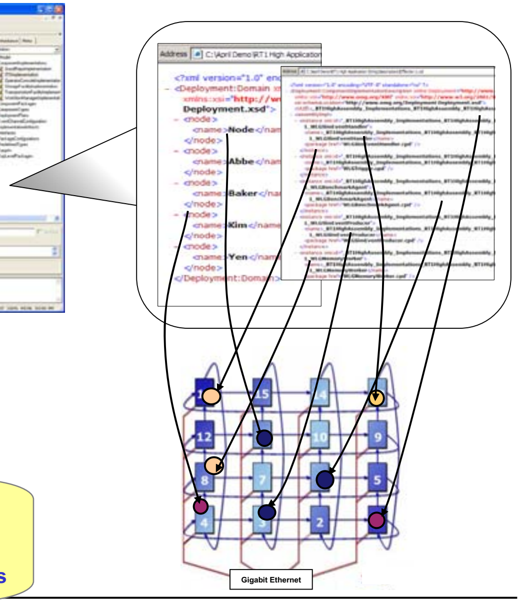
- *Components* encapsulate "business" logic
- Components interact via *ports*
  - *Provided ports*, e.g.,facets
  - *Required ports*, e.g., receptacles
  - *Event sink & source ports*
- *Containers* provide execution environment Components/containers can also
  - Communicate via a *middleware bus* &
  - Reuse *common middleware services*
- Aspect-oriented techniques can help with integration






# Promising Solution Approaches






**Physical View**

Model-driven development & analysis techniques for optimizing, verifying, & automating the deployment & configuration process

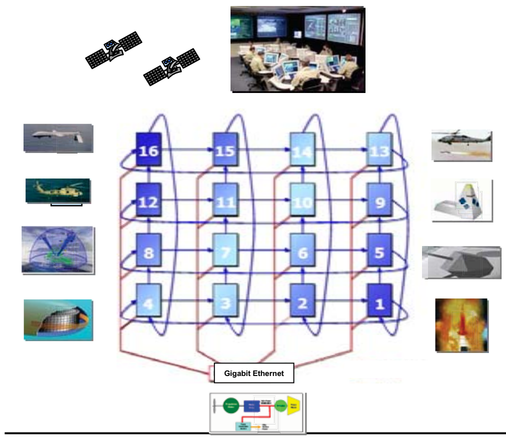


# Promising Solution Approaches

- Synthetic workload & emulated components
- Replaced incrementally with actual applications & components

Software execution modeling & emulation techniques & tools; distributed continuous quality assurance

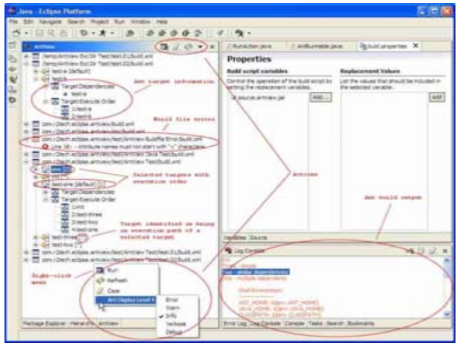
**Process View**

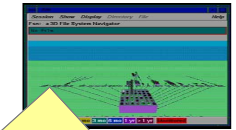



- Automate QA processes

## Promising Solution Approaches

- *Packages view* – shows element tree defined by project's build class path
- *Type hierarchy view* – shows the sub- & super-type hierarchies
- *Outline view* – shows the structure of a compilation unit or class file
- *Browsing perspective* – allows navigating models using separate views for projects, packages, types & members
- *Wizards for creating elements* – e.g., project, package, class, interface
- *Editors* – syntax coloring, content specific code assist, code resolve, method level edit, import assistance, quick fix & quick assist





**Development View**

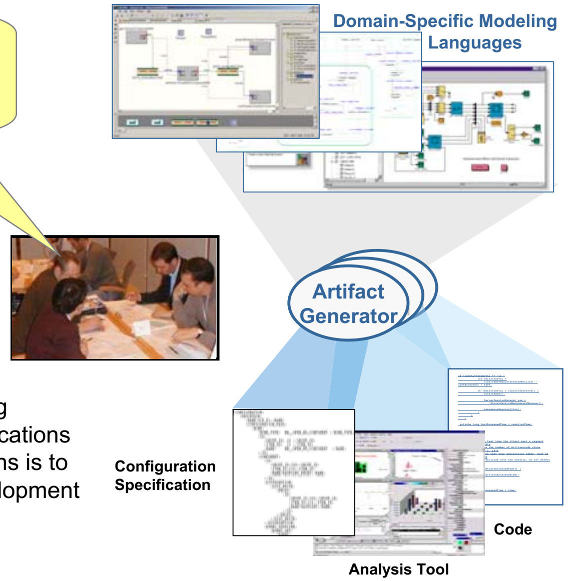
Development environments that provide multiple views & minimize dependencies between large-scale software artifacts to optimize development & test cycles



## Promising Solution Approaches

**Automated tracing of (in)consistency between requirement specifications & associated software artifacts**

**Use Case View**



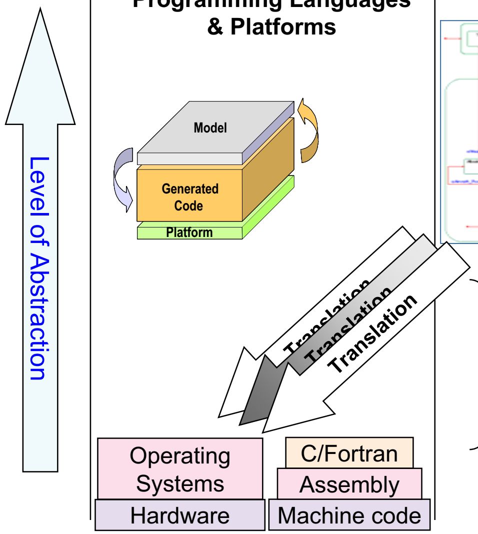

- One way to automate tracing between higher-level specifications & lower-level implementations is to leverage model-driven development techniques & tools

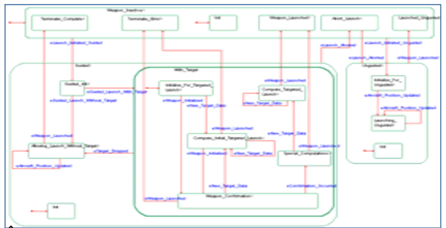


## Technology Evolution (1/4)

**Programming Languages & Platforms**

**Model-Driven Engineering (MDE)**



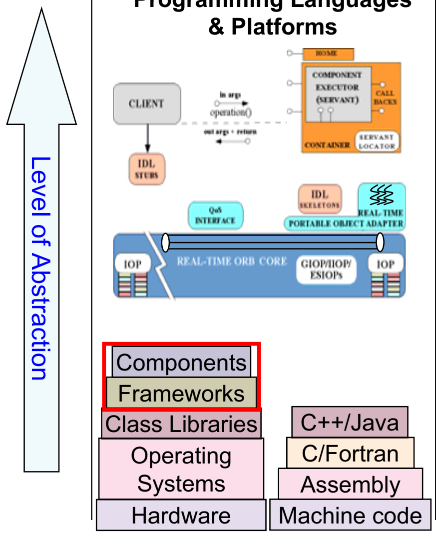
- State chart
- Data & process flow
- Petri Nets

Large Semantic Gap

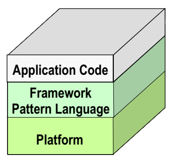


## Technology Evolution (2/4)

**Programming Languages & Platforms**




- Newer 3rd-generation languages & platforms have raised abstraction level significantly
  - "Horizontal" platform reuse alleviates the need to redevelop common services




- There are two problems, however:
  - Platform complexity evolved faster than 3rd-generation languages
  - Much application/platform code still (unnecessarily) written manually

# Technology Evolution (3/4)

Level of Abstraction

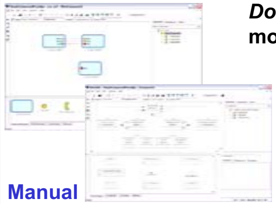
**Programming Languages & Platforms**

Saturation!!!!

Components
Frameworks
Class Libraries
Operating Systems
Hardware

C++/Java
C/Fortran
Assembly
Machine code

**Model-Driven Development (MDD)**



***Domain-specific* modeling languages**
- ESML
- PICML
- Mathematica
- Excel
- *Metamodels*

**Manual translation**


**Domain-independent modeling languages**
- State Charts
- Interaction Diagrams
- Activity Diagrams

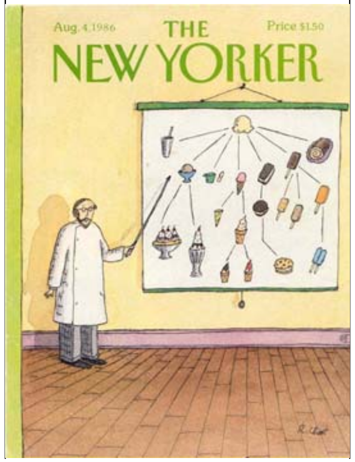
**Semi-automated**



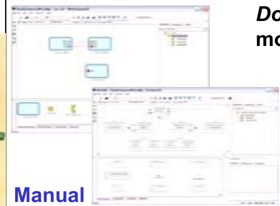
# Technology Evolution (3/4)

Level of Abstraction

**Programming Languages & Platforms**




**Model-Driven Development (MDD)**



***Domain-specific* modeling languages**
- ESML
- PICML
- Mathematica
- Excel
- *Metamodels*

**Manual translation**


**Domain-independent modeling languages**
- State Charts
- Interaction Diagrams
- Activity Diagrams

**Semi-automated**

- OMG is evaluating MDE via MIC PSIG
  - mic.omg.org

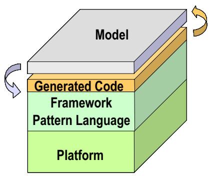
Model Integrated Computing PSIG



# Technology Evolution (3/4)

Level of Abstraction

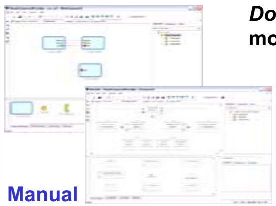
**Programming Languages & Platforms**




Components
Frameworks
Class Libraries
Operating Systems
Hardware

C++/Java
C/Fortran
Assembly
Machine code

**Model-Driven Development (MDD)**



***Domain-specific* modeling languages**
- ESML
- PICML
- Mathematica
- Excel
- *Metamodels*

**Manual translation**


**Domain-independent modeling languages**
- State Charts
- Interaction Diagrams
- Activity Diagrams

**Semi-automated**

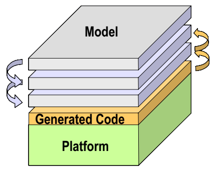
- OMG is evaluating MDE via MIC PSIG
  - mic.omg.org



# Technology Evolution (4/4)

Level of Abstraction

**Programming Languages & Platforms**

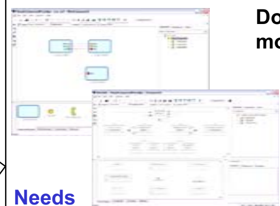



**Needs Automation**

Components
Frameworks
Class Libraries
Operating Systems
Hardware

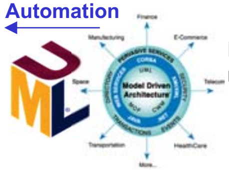
C++/Java
C/Fortran
Assembly
Machine code

**Model-Driven Development (MDD)**



**Domain-specific modeling languages**
- ESML
- PICML
- Mathematica
- Excel
- *Metamodels*

**Needs Automation**




**Domain-independent modeling languages**
- State Charts
- Interaction Diagrams
- Activity Diagrams

**Needs Automation**

Research is needed to automate DSMLs & model translators

See February 2006 IEEE Computer special issue on MDE techniques & tools

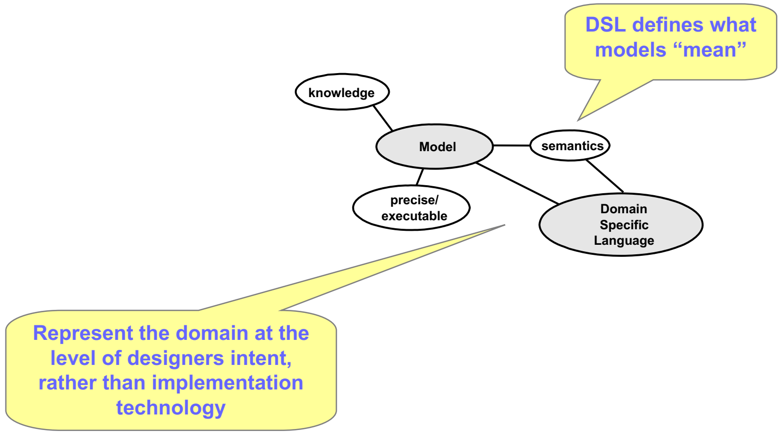
Model-Driven Development
of Distributed Systems

# CONTENTS



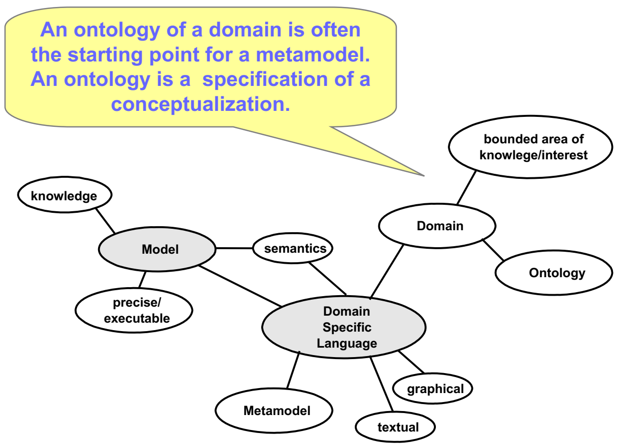


# Overview of Important Terms



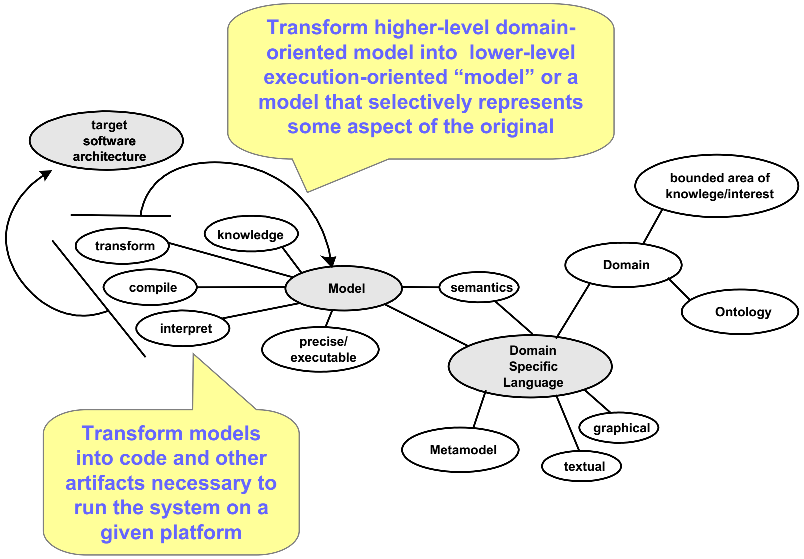



# Overview of Important Terms

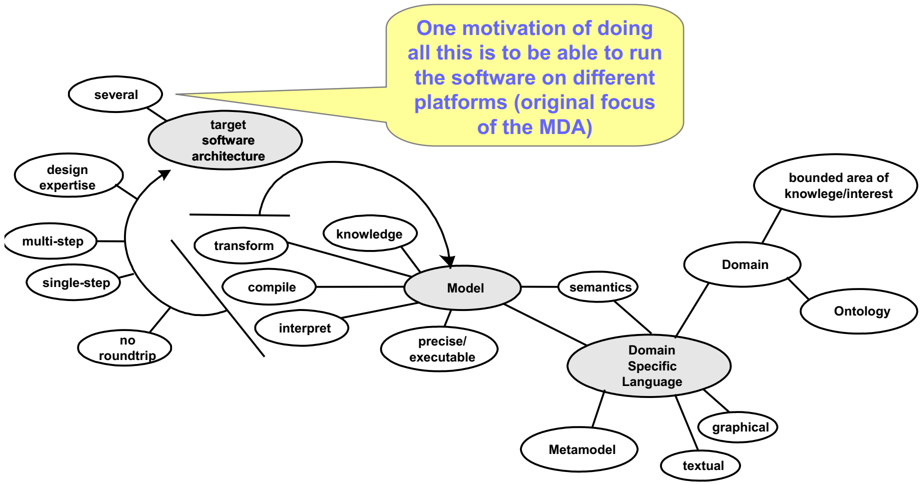





# Overview of Important Terms

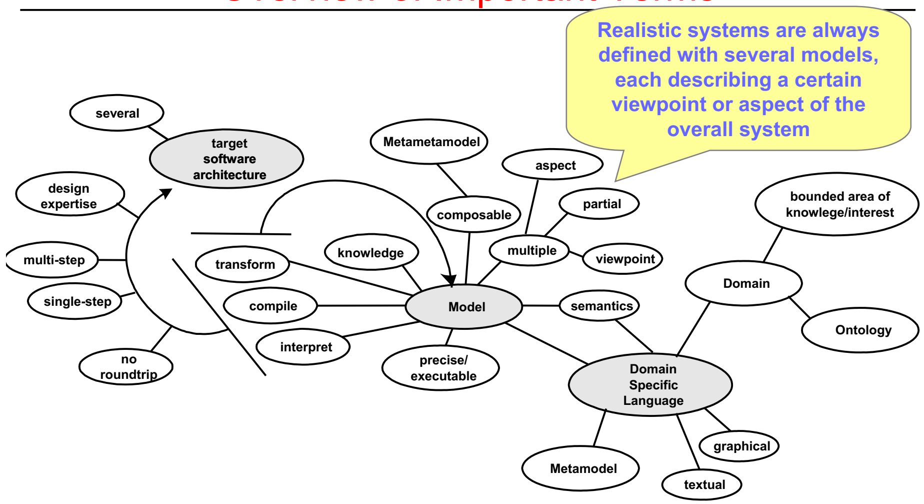


# Overview of Important Terms






# Overview of Important Terms






**Model-Driven Development** of Distributed Systems

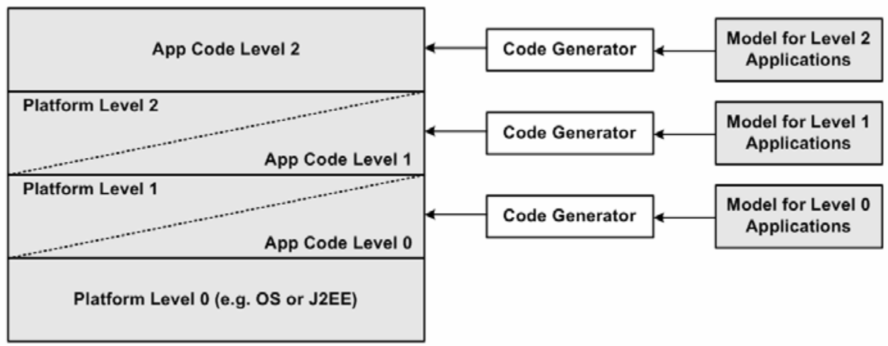
# CONTENTS





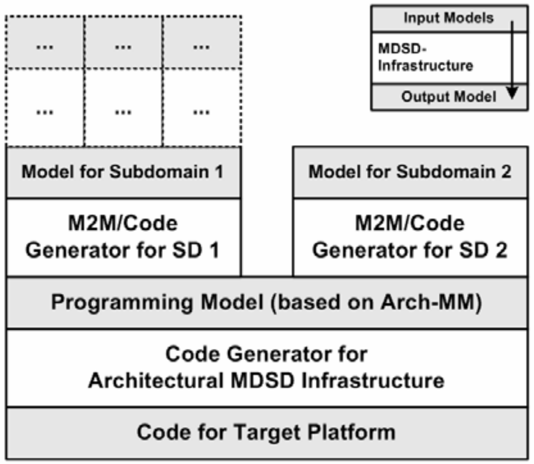
# Cascading MDD Using Platform Stacking

- The **generated code** of the lower layer **serves as the platform** for the next higher level
- A **sequence of generation steps** is used, whereas each of the generates code on which the next step builds



## Cascading MDD Using M2M

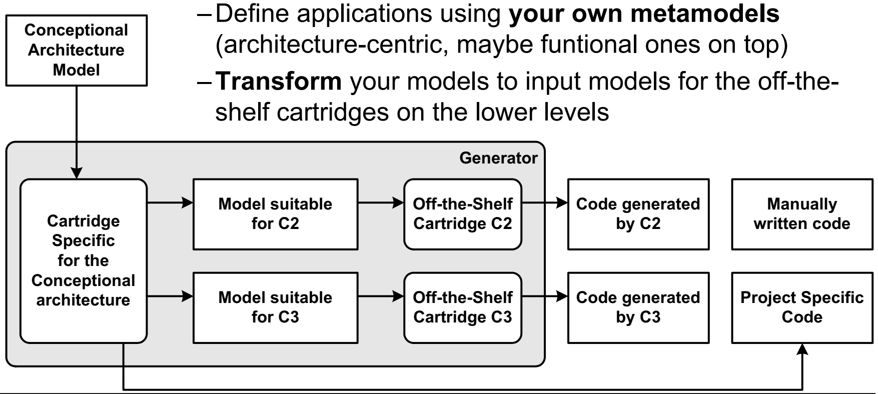
- Here the higher level models are **transformed** into lower-level models that serve as input for the lower level generators Model-to-Model Transformations are used
- Typically, **higher level models are more specific** to a certain (sub-)domain






## DYI vs 3rd Party Cartridges

- Do you build **your own generator** for your specific architecture?
  – This is good, because it's tailored to *your* architecture
- Or do you want to (re)use **off-the-shelf cartridges** for certain standard technologies (such as J2EE, Hibernate, Spring)?
  - You can do the **best of both worlds:**
    – Define applications using **your own metamodels** (architecture-centric, maybe funtional ones on top)
    – **Transform** your models to input models for the off-the-shelf cartridges on the lower levels






**Model-Driven Development** of Distributed Systems

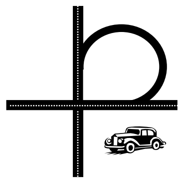
## CONTENTS





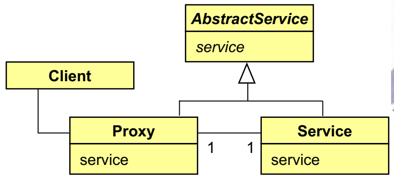
## Overview of Patterns

- Present *solutions* to common software *problems* arising within a certain *context*



- Help resolve key software design forces
  - *Flexibility*
  - *Extensibility*
  - *Dependability*
  - *Predictability*
  - *Scalability*
  - *Efficiency*

- Capture recurring structures & dynamics among software participants to facilitate reuse of successful designs




*The Proxy Pattern*

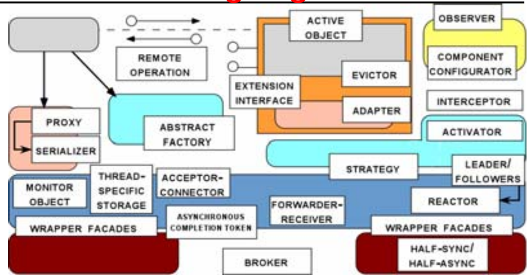
- Generally codify expert knowledge of design strategies, constraints & "best practices"


MDD tools codify & automate many (but by no means all) aspects of patterns

# Overview of Pattern Languages

**Motivation**

- Individual patterns & pattern catalogs are insufficient
- Software modeling methods & tools largely just illustrate ***what/how*** – not ***why*** – systems are designed




**Benefits of Pattern Languages**

- Define a *vocabulary* for talking about software development problems
- Provide a *process* for the orderly resolution of these problems, eg:
  - What are key problems to be resolved & in what order
  - What alternatives exist for resolving a given problem
  - How should mutual dependencies between the problems be handled
  - How to resolve each individual problem most effectively in its context
- Help to generate & reuse software architectures

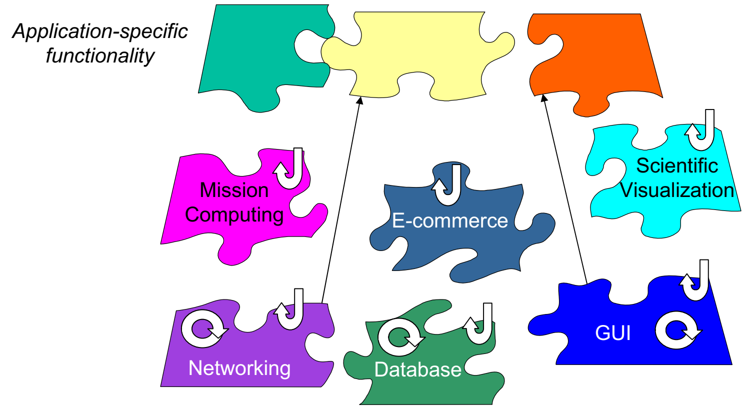


Pattern languages are crucial for DSLs & frameworks

# Overview of Frameworks

## Framework Characteristics

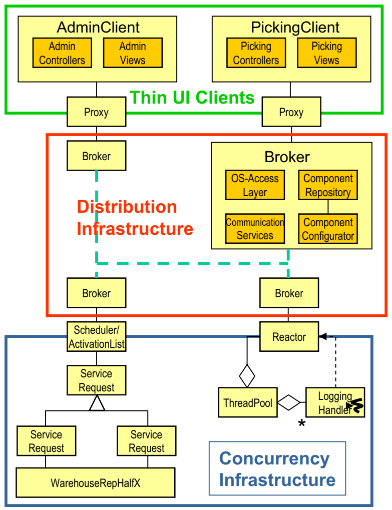
- Frameworks exhibit "inversion of control" at runtime via callbacks
- Frameworks provide integrated domain-specific structures & functionality
- Frameworks are "semi-complete" applications






# Benefits of Frameworks

- **Design reuse**
  - e.g., by guiding application developers through the steps necessary to ensure successful creation & deployment of software





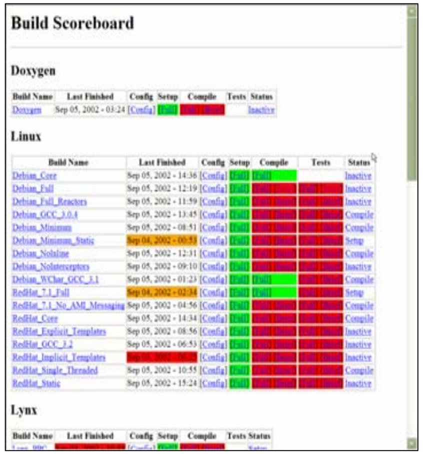

# Benefits of Frameworks

- **Design reuse**
  - e.g., by guiding application developers through the steps necessary to ensure successful creation & deployment of software
- **Implementation reuse**
  - e.g., by amortizing software lifecycle costs & leveraging previous development & optimization efforts

```
package orgapachetomcatsession;

import orgapachetomcatcore*;
import orgapachetomcatutilStringManager;
import javaio*;
import javanet*;
import javautil*;
import javaxservlet*;
import javaxservlethttp*;

/**
 * Core implementation of a server session
 *
 * @author James Duncan Davidson [duncan@engsuncom]
 * @author James Todd [gonzo@engsuncom]
 */

public class ServerSession {

    private StringManager sm =
        StringManagergetManager("orgapachetomcatsession");
    private Hashtable values = new Hashtable();
    private Hashtable appSessions = new Hashtable();
    private String id;
    private long creationTime = SystemcurrentTimeMillis();;
    private long thisAccessTime = creationTime;
private int inactiveInterval = -1;

    ServerSession(String id) {
        thisid = id;
    }

    public String getId() {
        return id;
    }

    public long getCreationTime() {
        return creationTime;
    }


    public ApplicationSession getApplicationSession(Context context,
        boolean create) {
        ApplicationSession appSession =
            (ApplicationSession)appSessionsget(context);

        if (appSession == null && create) {

            // XXX
            // sync to ensure valid?

            appSession = new ApplicationSession(id, this, context);
            appSessionsput(context, appSession);
        }

        // XXX
        // make sure that we haven't gone over the end of our
        // inactive interval -- if so, invalidate & create
        // a new appSession

        return appSession;
    }

    void removeApplicationSession(Context context) {
        appSessionsremove(context);
    }
```

# Benefits of Frameworks

- Design reuse
  - e.g., by guiding application developers through the steps necessary to ensure successful creation & deployment of software
- Implementation reuse
  - e.g., by amortizing software lifecycle costs & leveraging previous development & optimization efforts
- **Validation reuse**
  - e.g., by amortizing the efforts of validating application- & platform-independent portions of software, thereby enhancing software reliability & scalability

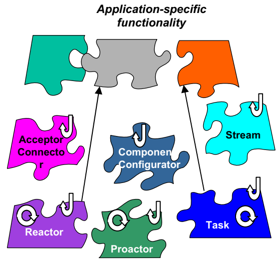


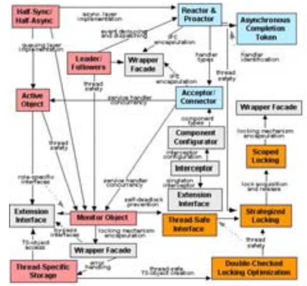


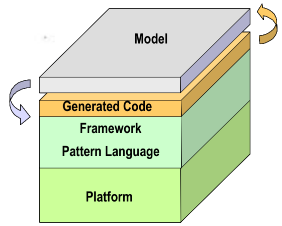

# Summary of Pattern, Framework, & MDD Synergies

**These technologies codify expertise of domain experts & developers**

- Frameworks codify expertise in the form of reusable algorithms, component & service implementations, & extensible architectures
- Patterns codify expertise in the form of reusable architecture design themes & styles, which can be reused event when algorithms, components implementations, or frameworks cannot
- MDD tools codify expertise by automating key aspects of pattern languages & providing developers with domain-specific modeling languages to access the powerful (& complex) capabilities of frameworks






**There are now powerful feedback loops advancing these technologies**

## Model-Driven Development of Distributed Systems

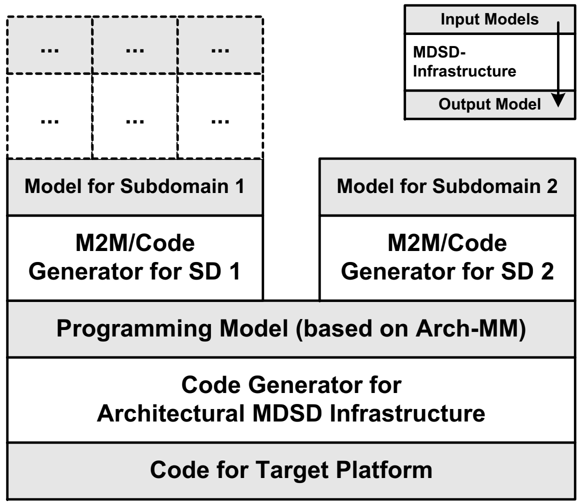
# CONTENTS





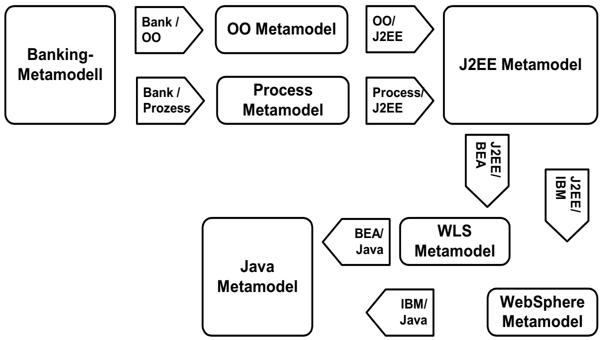
# Why You Need M2M

- As explained earlier, **cascading MDD** requires model-to-model transformations



# Modular, Automated Transformations

- To more easily reuse parts of a transformation, it is a good idea to **modularize a transformation**
- Note that in contrast to the OMG, we do not recommend **looking at, changing, or marking** the intermediate models
- They are merely a **standardized format for exchanging data** among transformations
- Example: Multi-Step transformation from a banking-specific DSL to Java via J2EE

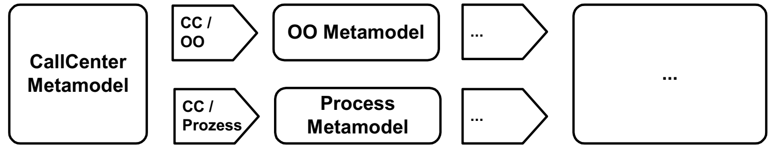




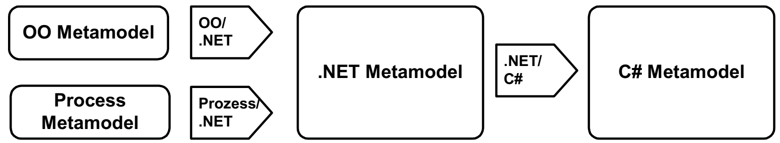

# Modular, Automated Transformations II

- Example cont'd:

  Now consider a Call-Center application; only the first step needs to be adapted

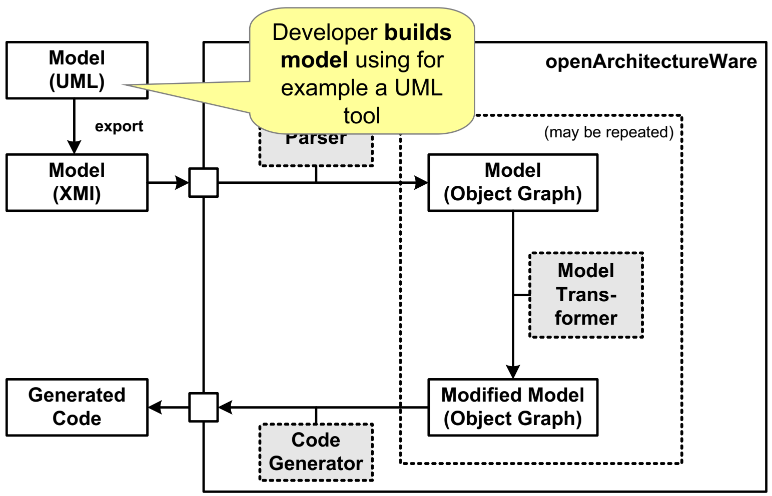



- If both should be transformed to NET, only the backend needs to be exchanged



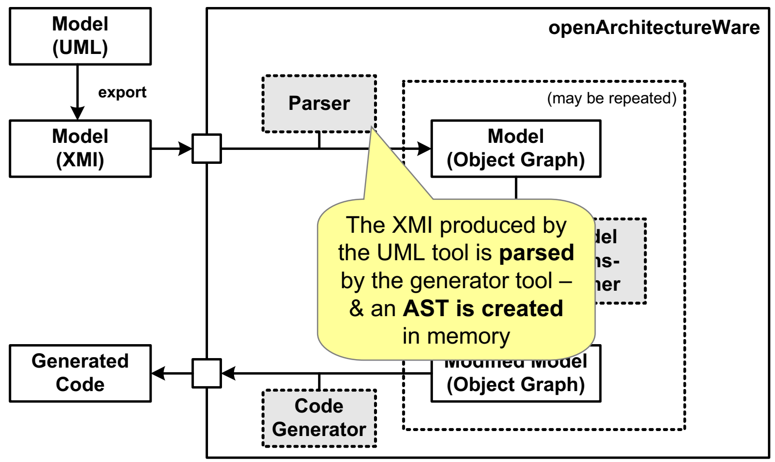



# Transforming "in the Tool"

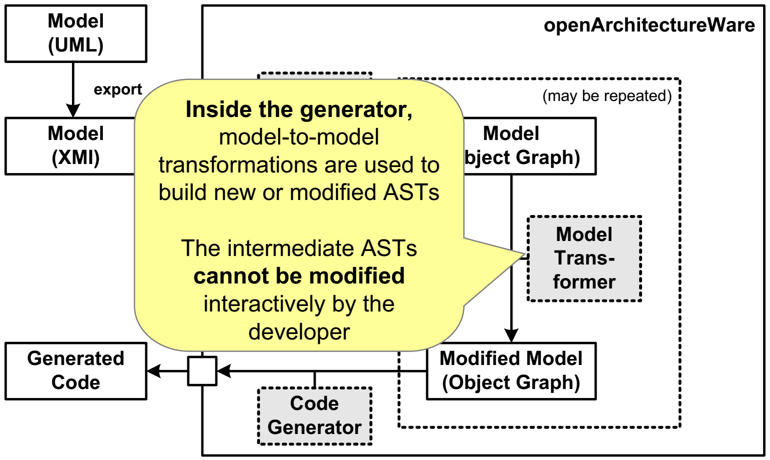





# Transforming "in the Tool"

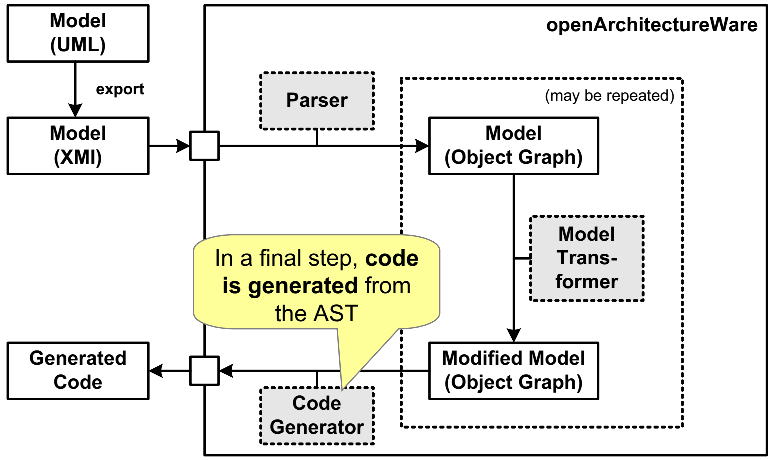


# Transforming “in the Tool”






# Transforming “in the Tool”






# External Model Markings (AO–Modeling)

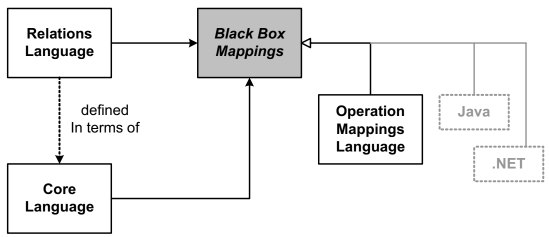
- To allow the transformation of a source model into a target model (or to generate code) it is sometimes necessary to **provide “support” information that is specific to the target meta model**
  - Example: Entity Bean vs Type Manager
- Adding these to the source model “**pollutes**” **the source model** with concepts specific to the target model
- MDA proposes to add **“model markings,”** but this currently supported only by a few tools
- Instead, we recommend keeping this information **outside of the model** (e.g., in an XML file)
  - The transformation engine would use this auxiliary information when executing the transformations

This is an example of “aspect-oriented programming/modeling”



# Model–to–Model Transformations: QVT

- Most of the transformations built thus far have been constructed with **Java code**
  - If the metaclasses have a **well-designed API** (repository API) then this “procedural transformations” does indeed work well
- However, more **dedicated model transformation languages** are becoming available:
  - e.g., ATL, MOLA, Wombat (oAW), etc
- The **QVT standard** is becoming a reality
  - It will be finalized by the end of 2006
- QVT actually comprises **three languages**:



## Model–to–Model Transformations: QVT Relational

```
top relation EntityKeyToTableKey {

  checkonly domain alma entity:Entity {
    key = entityKeyField:Field {}
  };

  enforce domain db table:Table {
    key = tableKey:Key {}
  };

  when {
    EntityToTable(entity, table
  }

  where {
    KeyRecordToKeyColumns(entit
  }

}
```

```
relation PhysicalQuantityTypeToColumn {

  pqName, pqUnit, fieldName : String;

  checkonly domain alma field:Field {
    name = fieldName,
    type = pq:PhysicalQuantityType {
      name = pqName,
      units = pqUnit
    }
  };

  enforce domain db table:Table {
    columns = column:Column {
      name = prefix + fieldName + '_as_' +
             pqName + '_in_' + pqUnit,
      type = AlmaPhysicalQuantityTypeToDbType(pq)
    }
  };

  primitive domain prefix:String;
}
```



## M2M–Transformations: QVT Operational

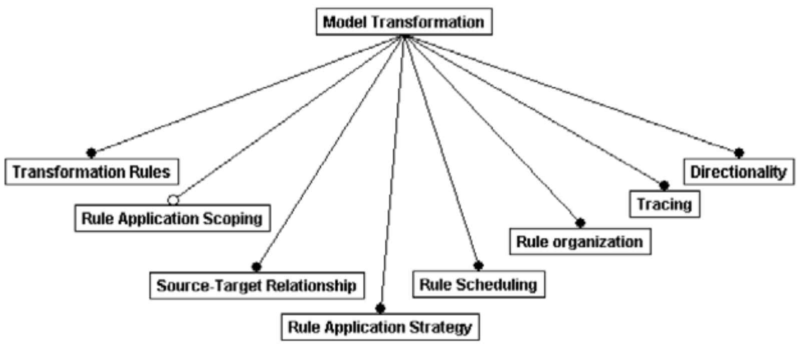
```
mapping DependentPart::part2table(in prefix : String) : Table
inherits fieldColumns {

  var dpTableName := prefix + recordName;
  name := dpTableName;
  columns := mainColumns +
             object Column {
               name := 'key_' + dpTableName;
               type := 'INTEGER';
               inKey := true;
             }

  end { self.parts->map part2columns(result, dpTableName + '_'); }
}
```

```
query PrimitiveType::convertPrimitiveType() : String =

  if self.name = "int" then 'INTEGER'
  else if self.name = "float" then 'FLOAT'
  else if self.name = "long" then 'BIGINT'
  else 'DOUBLE'
endif endif endif;
```



## Many Means of Transformations

- Today, many means of transformations are used:
  - Plain old Java
  - Eclipse GMT ATL
  - IBM MTF
  - ISIS GReAT
  - Several partial QVT implementations
  - UMLX
- A paper by Czarnecki/Helsen gives a very good overview: www.swen.uwaterloo.ca/~kczarnec/ECE750T7/czarnecki_helsen.pdf






**Model-Driven Development**
of Distributed Systems

## CONTENTS

## Architectural Case Study

- PHASE 1: Elaborate!
  - Technology-Independent Architecture
  - Programming Model
  - Technology Mapping
  - Mock Platform
  - Vertical Prototype
- PHASE 2: Iterate!
- PHASE 3: Automate!
  - Architecture Metamodel
  - Glue Code Generation
  - DSL-based Programming Model
  - Model-based Architecture Validation



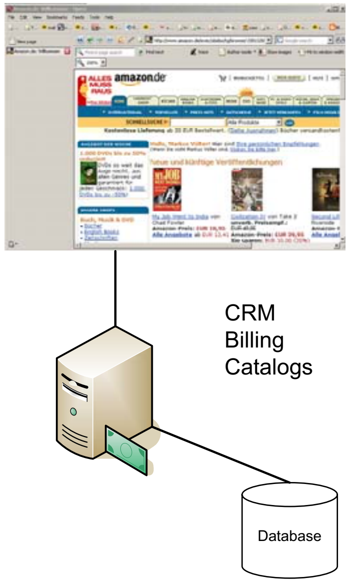
## Architectural Case Study

- **PHASE 1: Elaborate!**
  - Technology-Independent Architecture
  - Programming Model
  - Technology Mapping
  - Mock Platform
  - Vertical Prototype
- PHASE 2: Iterate!
- PHASE 3: Automate!
  - Architecture Metamodel
  - Glue Code Generation
  - DSL-based Programming Model
  - Model-based Architecture Validation



## Phase 1: Elaborate!

- This first elaboration phase should be handled by a **small team**, before the architecture is rolled out to the whole team
- We want to build an **enterprise system** that contains various subsystems such as customer management, billing & catalogs
- In addition to managing the data using a database, forms & the like, we also have to manage the associated **long-running business processes**
- We will look at how we can attack this problem below






## Architectural Case Study

- PHASE 1: Elaborate!
  - **Technology-Independent Architecture**
  - Programming Model
  - Technology Mapping
  - Mock Platform
  - Vertical Prototype
- PHASE 2: Iterate!
- PHASE 3: Automate!
  - Architecture Metamodel
  - Glue Code Generation
  - DSL-based Programming Model
  - Model-based Architecture Validation

## Technology–Independent Architecture

- We decide that our system will be built from **components**
  - Each component can **provide** a number of **interfaces**
  - It can also **use** a number of **interfaces** (provided by other components)
  - Communication is **synchronous**, Communication is also restricted to be **local**
  - We design components to be **stateless**
- In addition to components, we also explicitly support **business processes**
  - These are modeled as a **state machine**
  - Components can trigger the state machine by supplying **events** to them
  - Other components can be triggered by the state machine, resulting in the **invocation of certain operations**
  - Communication to/from processes is **asynchronous**, **remote** communication is supported



## Technology–Independent Architecture

- We decide that our system will be built from **components**
  - Each component can **provide** a number of **interfaces**
  - It can also **use** a number of **interfaces** (provided by other components)
  - Communication is **synchronous**, Communication is also restricted to be **local**
- In ... **pr**...
  - Other components can be triggered by the state machine, resulting in the **invocation of certain operations**
  - Communication to/from processes is **asynchronous**, **remote** communication is supported

> - Use well-known **architectural styles & patterns** here
> - Typically these are best practices for architecting certain kinds of systems independent of a particular technology
> - They provide a reasonable starting point for defining (aspects of) your systems's architecture



## Architectural Case Study

- PHASE 1: Elaborate!
  - Technology-Independent Architecture
  - **Programming Model**
  - Technology Mapping
  - Mock Platform
  - Vertical Prototype
- PHASE 2: Iterate!
- PHASE 3: Automate!
  - Architecture Metamodel
  - Glue Code Generation
  - DSL-based Programming Model
  - Model-based Architecture Validation



## Programming Model

- The programming model uses a **simple Dependency Injection approach** à la Spring to define component dependencies on an interface level
  - Spring is a modular framework for Java enterprise applications (see www.springframework.org)
- An **external XML file** is responsible for configuring the instances

```
<beans>
  <bean id="proc" class="somePackage.SomeProcess">
    <property name="resource"><ref bean="hello"/></property>
  </bean>
  <bean id="hello" class="somePackage.ExampleComponent">
    <property name="console"><ref bean="cons"/></property>
  </bean>
  <bean id="cons" class="someFramework.StdOutConsole">
</beans>
```

## Programming Model

- The following piece of code shows the **implementation of a simple example component** (note the use of Java 5 annotations)

```
public @component class ExampleComponent
  implements HelloWorld {        // provides HelloWorld

  private IConsole console;

  public @resource void setConsole( IConsole c ) {
    this.console = c;            // setter for console
  }                              // component

  public void sayHello( String s ) {
    console.write( s );
  }
}
```

- Processes **engines** are components like any other
- For **triggers**, they provide an interface w/ void operations
- They also define interfaces with the actions that those components can implement that want to be notified of state changes



## Programming Model

- Process Component Implementation Example

```
public @process class SomeProcess
                implements ISomeProcessTrigger {

  private IHelloWorld resource;

  public @resource void setResource( IHelloWorld w ) {
    this.resource = w;
  }

  public @trigger void T1( int procID ) {
    SomeProcessInstance i = loadProcess( procID );
    if ( guardG1() ) {
      // advance to another state…
    }
  }

  public @trigger void T2( int procID ) {
    SomeProcessInstance i = loadProcess( procID );
    // …
    resource.sayHello( "hello" );
 }
}
```



## Architectural Case Study

- PHASE 1: Elaborate!
  - Technology-Independent Architecture
  - Programming Model
  - **Technology Mapping**
  - Mock Platform
  - Vertical Prototype
- PHASE 2: Iterate!
- PHASE 3: Automate!
  - Architecture Metamodel
  - Glue Code Generation
  - DSL-based Programming Model
  - Model-based Architecture Validation



## Technology Mapping

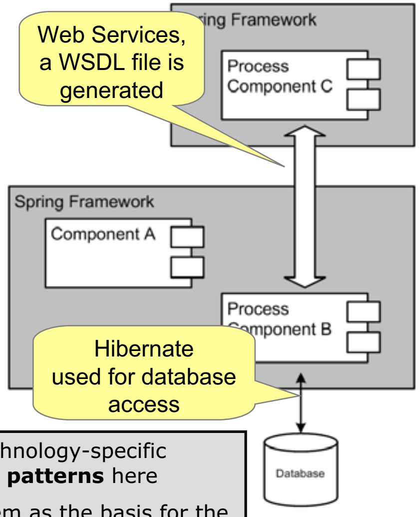
- For the remote communication between business processes we will use **web services**
  - From the interfaces such as `IHelloWorld`, we **generate a WSDL file**, & the necessary endpoint implementation We use on of the many available web service frameworks
- **Spring will be used** as long as no advanced load balancing & transaction policies are required

```
<beans>
  <bean id="proc" class="somePackage.SomeProcess">
    <property name="resource"><ref bean="hello"/></property>
  </bean>
  <bean id="hello" class="somePackage.ExampleComponent">
    <property name="console"><ref bean="cons"/></property>
  </bean>
  <bean id="cons" class="someFramework.StdOutConsole">
</beans>
```

- Once this becomes necessary, we will use **Stateless Session EJBs** The necessary code to wrap our components inside beans is easy to write

# Technology Mapping

- **Persistence** for the process instances – like any other persistent data – is managed using **Hibernate**
  - To make this possible, we create a **data class for each process**
  - Since this is a normal value object, using Hibernate to make it persistent is straight forward




Decide about **standards usage** here, not earlier

But keep in mind: ***First*** solve the problem, ***then*** look for a standard – Not vice versa

Use technology-specific **design patterns** here

Use them as the basis for the TECHNOLOGY MAPPING



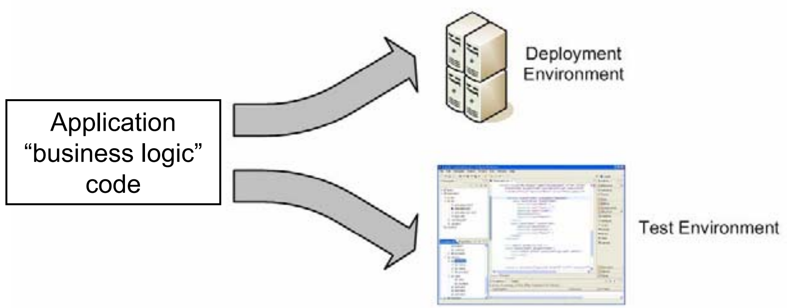
# Architectural Case Study

- PHASE 1: Elaborate!
  - Technology-Independent Architecture
  - Programming Model
  - Technology Mapping
  - **Mock Platform**
  - Vertical Prototype
- PHASE 2: Iterate!
- PHASE 3: Automate!
  - Architecture Metamodel
  - Glue Code Generation
  - DSL-based Programming Model
  - Model-based Architecture Validation



# Mock Platform

- Since we are already using a PROGRAMMING MODEL that resembles Spring, we use the Spring container to run the application components locally
- Stubbing out parts is easy based on Springs XML configuration file
- Since persistence is something that Hibernate takes care of for us, the MOCK PLATFORM simply ignores the persistence aspect






# Architectural Case Study

- PHASE 1: Elaborate!
  - Technology-Independent Architecture
  - Programming Model
  - Technology Mapping
  - Mock Platform
  - **Vertical Prototype**
- PHASE 2: Iterate!
- PHASE 3: Automate!
  - Architecture Metamodel
  - Glue Code Generation
  - DSL-based Programming Model
  - Model-based Architecture Validation

# Vertical Prototype

- The **vertical prototype** includes parts of the customer & billing systems
  - For creating an invoice, the billing system uses **normal interfaces** to query the customer subsystem for customer details
  - The invoicing process is based on a **long-running process**
- A **scalability test** was executed & resulted in two problems:
  - For short running processes, the repeated loading & saving of persistent process state had become a problem
    - A **caching layer** was added
  - Second, web-service based communication with process components was a problem
    - **Communication was changed to CORBA** for remote cases that were inside the company



# Vertical Prototype


- The **vertical prototype** includes parts of the customer & billing systems
  - For creating an invoice, the billing system uses **normal interfaces** to query the customer subsystem for customer details
  - The invoicing process is based on a **long-running process**
- A **scalability test** was executed & resulted in two problems:


- Work on **performance improvements** here, not earlier
- It is **bad practice** to optimize design for performance from the beginning, since this often destroys good architectural practice
- In certain domains, there are **patterns to realize certain QoS properties** (such as stateless design for large-scale business systems)
  - Don't ignore these intentionally at the beginning!



# Architectural Case Study

- PHASE 1: Elaborate!
  - Technology-Independent Architecture
  - Programming Model
  - Technology Mapping
  - Mock Platform
  - Vertical Prototype
- **PHASE 2: Iterate!**
- PHASE 3: Automate!
  - Architecture Metamodel
  - Glue Code Generation
  - DSL-based Programming Model
  - Model-based Architecture Validation



# Phase 2: Iterate!

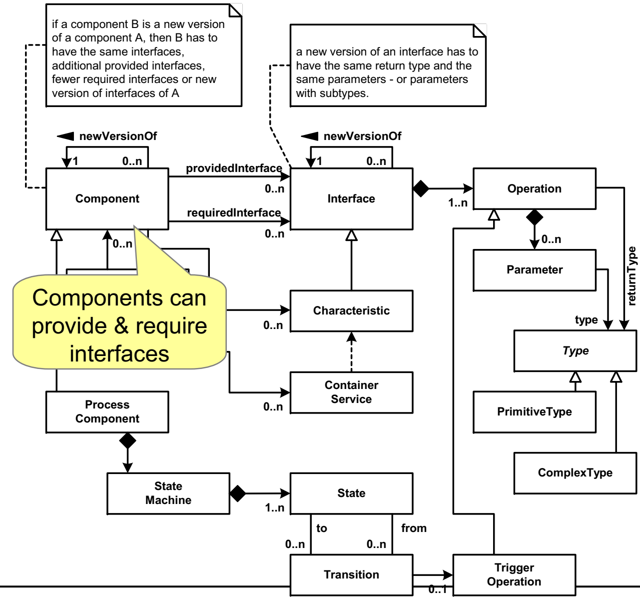
- Spring was intended for the production environment
- **New requirements** (versioning!) have made this infeasible
  - Spring does not support two important features
    1. **Dynamic installation/de-installation** of components &
    2. **isolations of components** from each other(classloaders)
- Eclipse has been chosen as the new execution framework
  - The PROGRAMMING MODEL **did not change**
  - The TECHNOLOGY MAPPING, however, had to be adapted

# Architectural Case Study

- PHASE 1: Elaborate!
  - Technology-Independent Architecture
  - Programming Model
  - Technology Mapping
  - Mock Platform
  - Vertical Prototype
- PHASE 2: Iterate!
- PHASE 3: Automate!
  - **Architecture Metamodel**
  - Glue Code Generation
  - DSL-based Programming Model
  - Model-based Architecture Validation

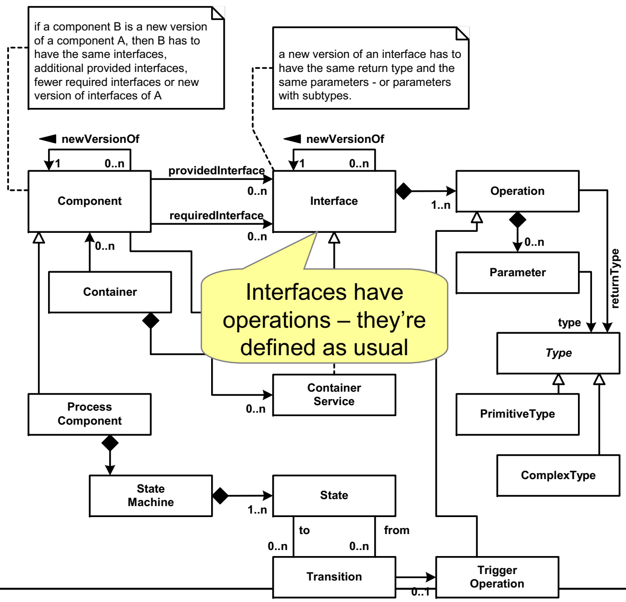


# Architecture Metamodel



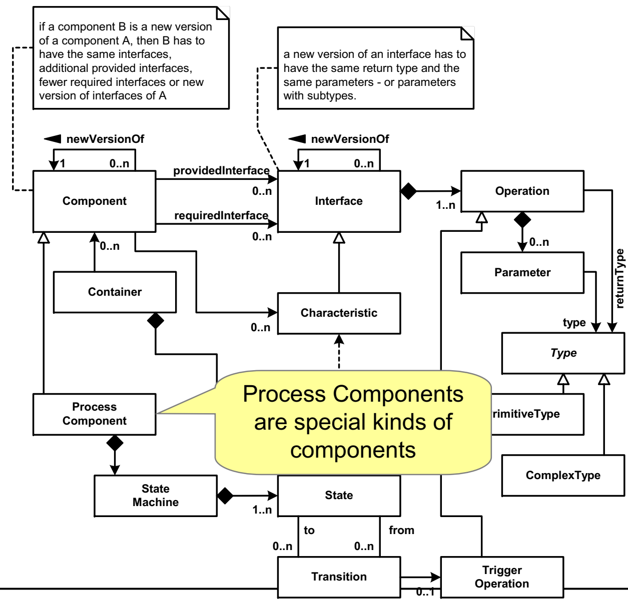



# Architecture Metamodel

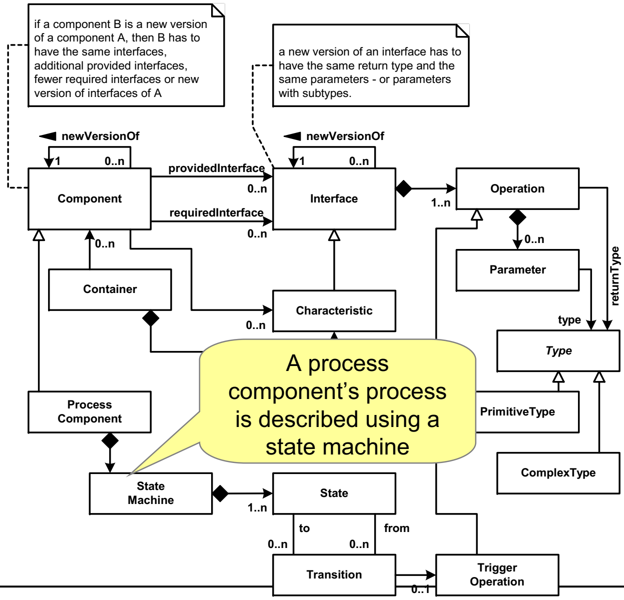





# Architecture Metamodel

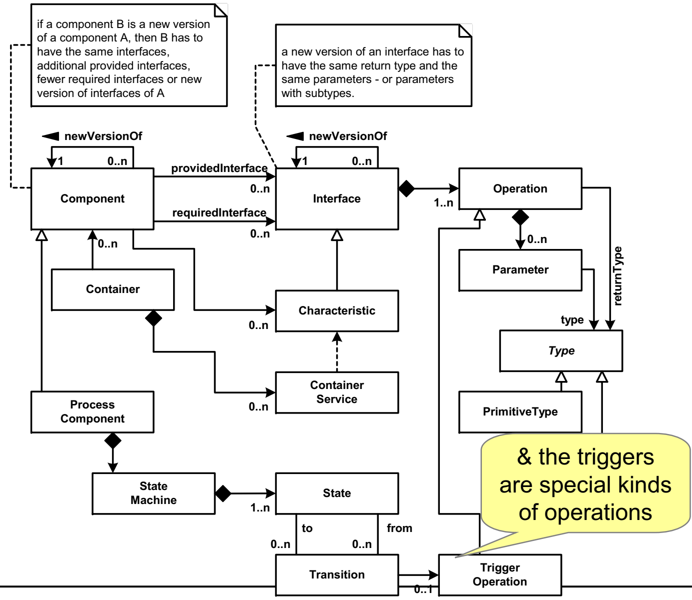


# Architecture Metamodel



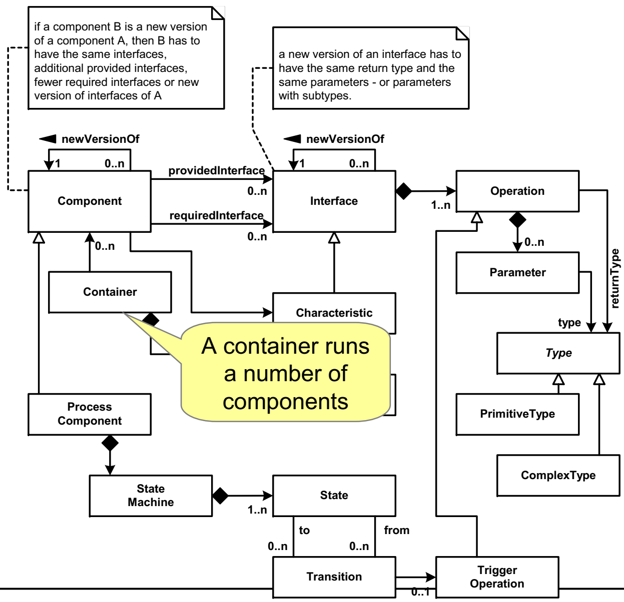



# Architecture Metamodel



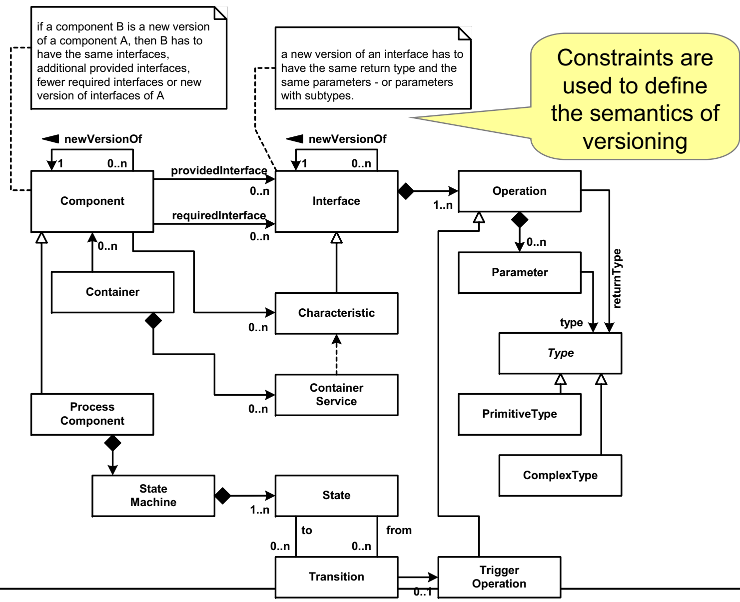



# Architecture Metamodel






# Architecture Metamodel



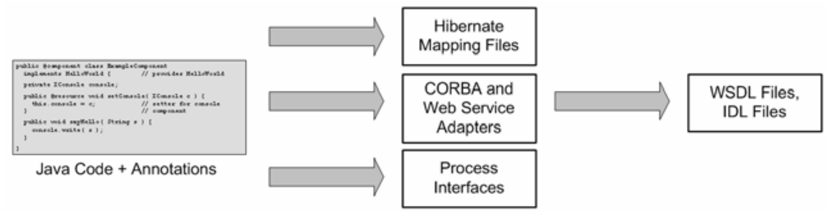
# Architectural Case Study

- PHASE 1: Elaborate!
  – Technology-Independent Architecture
  – Programming Model
  – Technology Mapping
  – Mock Platform
  – Vertical Prototype
- PHASE 2: Iterate!
- PHASE 3: Automate!
  – Architecture Metamodel
  – **Glue Code Generation**
  – DSL-based Programming Model
  – Model-based Architecture Validation



# Glue Code Generation

- Our scenario has several useful locations for glue code generation
  – We generate the **Hibernate** mapping files
  – We generate the **web service & CORBA adapters** based on the interfaces & data types that are used for communication The generator uses reflection to obtain the necessary type information
  – Finally, we generate the **process interfaces** from the state machine implementations




- In the programming model, we use **Java 5 annotations** to **mark up** those aspects that cannot be derived by using reflection alone
- Annotations can help a code generator to “know what to generate” **without making the programming model overly ugly**



# Architectural Case Study

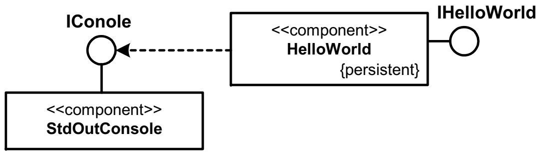
- PHASE 1: Elaborate!
  – Technology-Independent Architecture
  – Programming Model
  – Technology Mapping
  – Mock Platform
  – Vertical Prototype
- PHASE 2: Iterate!
- PHASE 3: Automate!
  – Architecture Metamodel
  – Glue Code Generation
  – **DSL-based Programming Model**
  – Model-based Architecture Validation



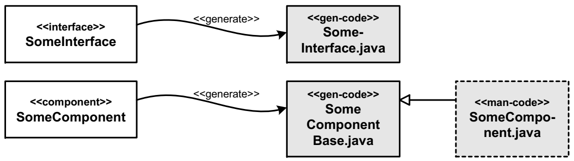
# DSL–based Programming Model

- We use DSLs for **components, interfaces & dependencies** Describing this aspect in a model has two benefits:
  – First, the GLUE CODE GENERATION can use a **more semantically rich model** as its input &
  – The model allows for very powerful MODEL-BASED ARCHITECTURE **VALIDATION** (see below)

# DSL–based Programming Model

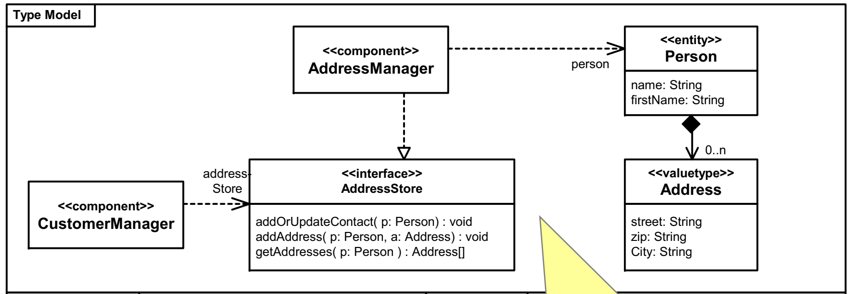



- From these diagrams:
  - We can generate a **skeleton component class**
  - All the **necessary interfaces**
- Developers simply **inherit from the generated skeleton** & implement the operations defined by the provided interfaces






# DSL–based Programming Model




**Composition Model**

```
<configurations>
  <configuration name="addressStuff">
    <deployment name="am" type="AddressManager">
      <wire name="personDAO" target="personDAO"/>
    </deployment>
    <deployment name="personDAO" type="PersonDAO"/>
  </configuration>
  <configuration name="customerStuff">
    <deployment name="cm" type="CustomerManager">
      <wire name="addressStore" target=":addressStuff:am"/>
    </deployment>
  </configuration>
  <configuration name="test" includes="addressStuff, customerStuff"/>
</configurations>
```

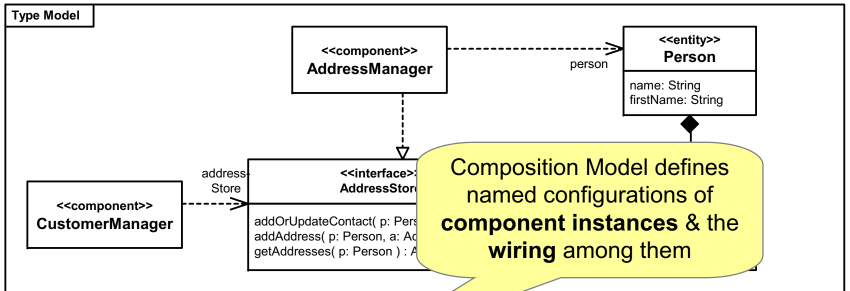
**System Model**

```
<systems
  <system
    <node
    <node
  <system
  <system
    <node
  <system
</systems
```

Type model defines **components** (which are instantiatable types), **interfaces** & **data types**, as well as the **depdendencies** among them



# DSL–based Programming Model




Composition Model defines named configurations of **component instances** & the **wiring** among them

**Composition Model**

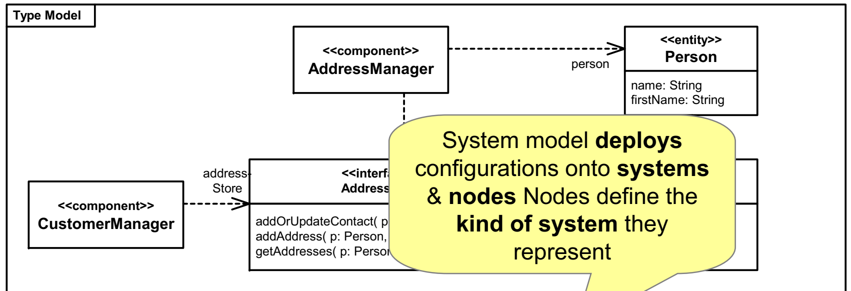
```
<configurations>
  <configuration name="addressStuff">
    <deployment name="am" type="AddressManager">
      <wire name="personDAO" target="personDAO"/>
    </deployment>
    <deployment name="personDAO" type="PersonDAO"/>
  </configuration>
  <configuration name="customerStuff">
    <deployment name="cm" type="CustomerManager">
      <wire name="addressStore" target=":addressStuff:am"/>
    </deployment>
  </configuration>
  <configuration name="test" includes="addressStuff, customerStuff"/>
</configurations>
```

**S...odel**

```
<systems>
  <system name="production">
    <node name="server" type="spring" configuration="addressStuff"/>
    <node name="client" type="eclipse" configuration="customerStuff"/>
  <system>
  <system name="test">
    <node name="test" type="spring" configuration="test"/>
  <system>
</systems>
```



# DSL–based Programming Model




System model **deploys** configurations onto **systems** & **nodes** Nodes define the **kind of system** they represent

**Composition Model**

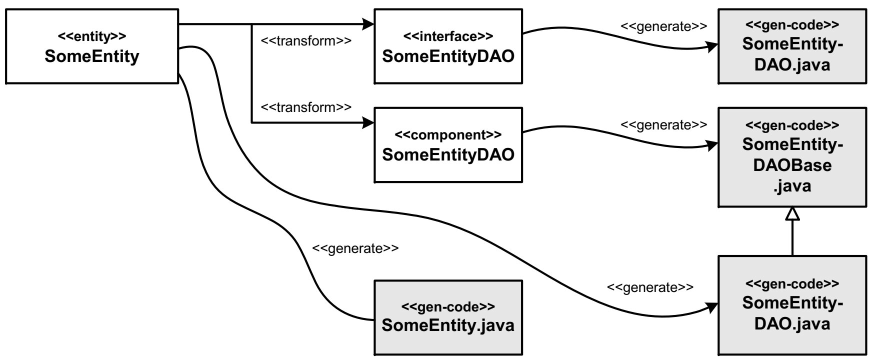
```
<configurations>
  <configuration name="addressStuff">
    <deployment name="am" type="AddressManager">
      <wire name="personDAO" target="personDAO"/>
    </deployment>
    <deployment name="personDAO" type="PersonDAO"/>
  </configuration>
  <configuration name="customerStuff">
    <deployment name="cm" type="CustomerManager">
      <wire name="addressStore" target=":addressStuff:am"/>
    </deployment>
  </configuration>
  <configuration name="test" includes="addressStuff, customerStuff"/>
</configurations>
```

**System Model**

```
<systems>
  <system name="production">
    <node name="server" type="spring" configuration="addressStuff"/>
    <node name="client" type="eclipse" configuration="customerStuff"/>
  <system>
  <system name="test">
    <node name="test" type="spring" configuration="test"/>
  <system>
</systems>
```

# DSL–based Programming Model

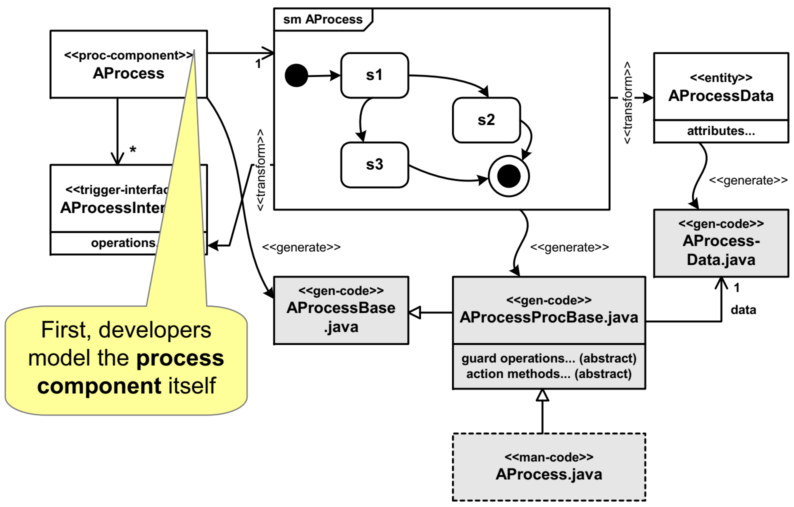
- Using **Cascaded MDD**, we generate
  - **DAO Components** for Entities from the Entities in the model
  - An **interface** for the DAO component,
  - As well as the **implementation** code for the DAO & the Entity itself






# DSL–based Programming Model

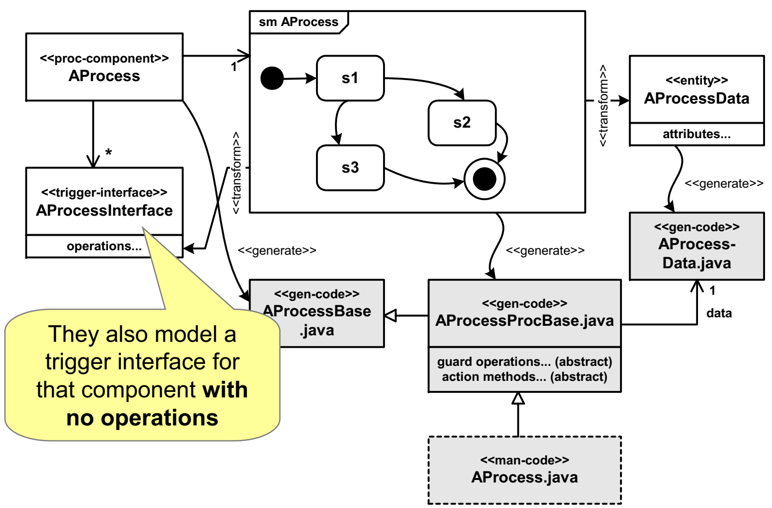
- We also use **cascading** for the Process Components






# DSL–based Programming Model

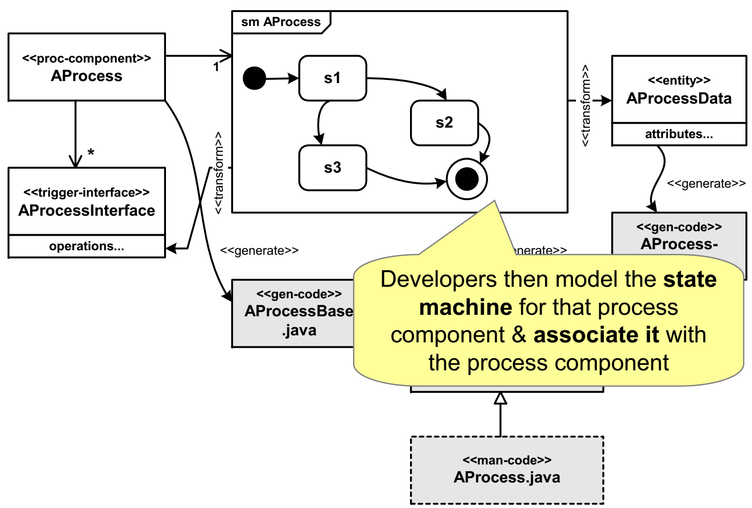
- We also use **cascading** for the Process Components





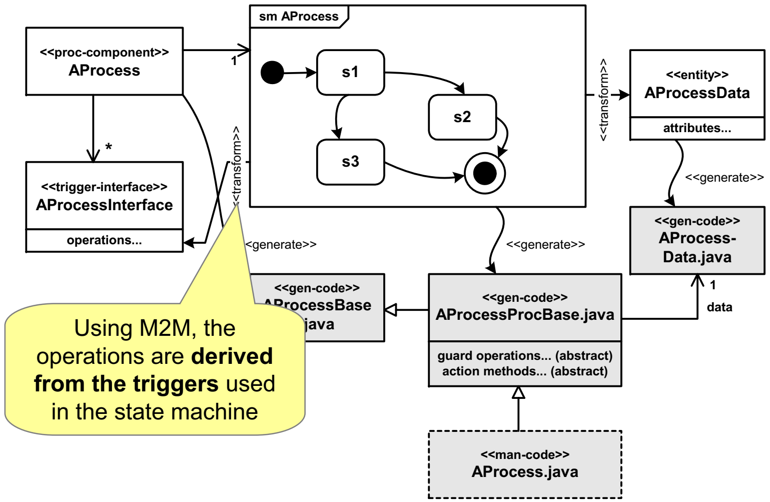

# DSL–based Programming Model

- We also use **cascading** for the Process Components



# DSL–based Programming Model

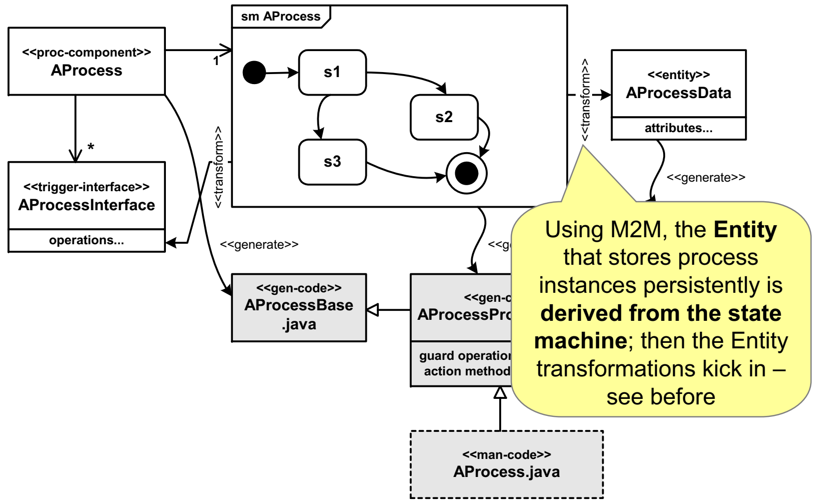
- We also use **cascading** for the Process Components






# DSL–based Programming Model

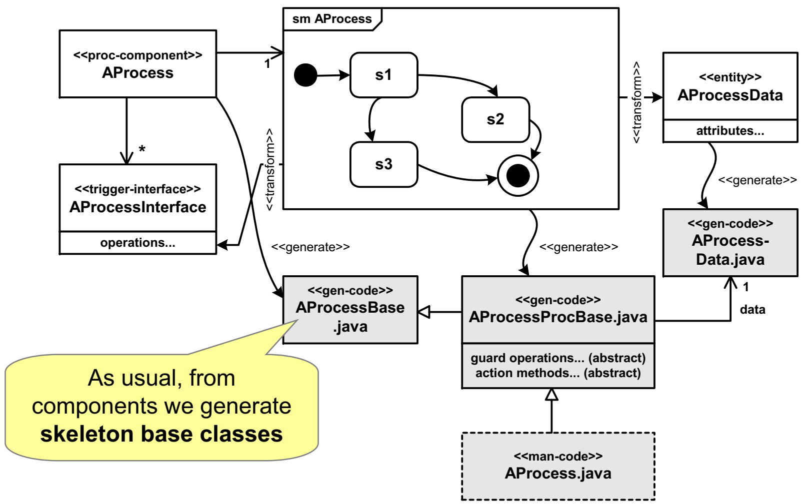
- We also use **cascading** for the Process Components






# DSL–based Programming Model

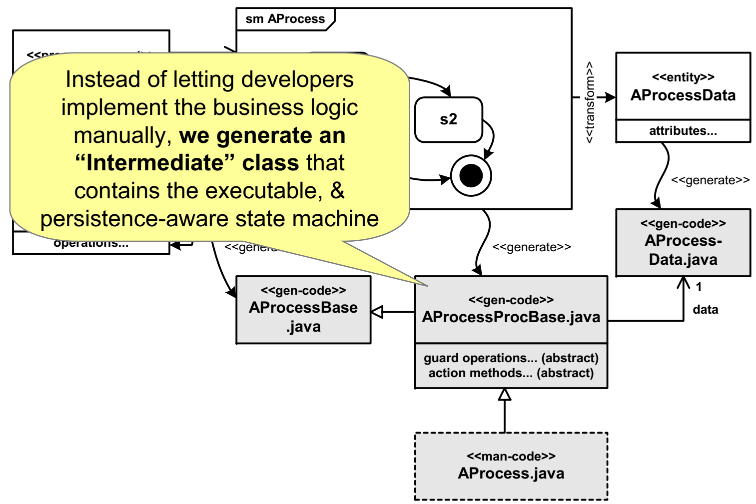
- We also use **cascading** for the Process Components





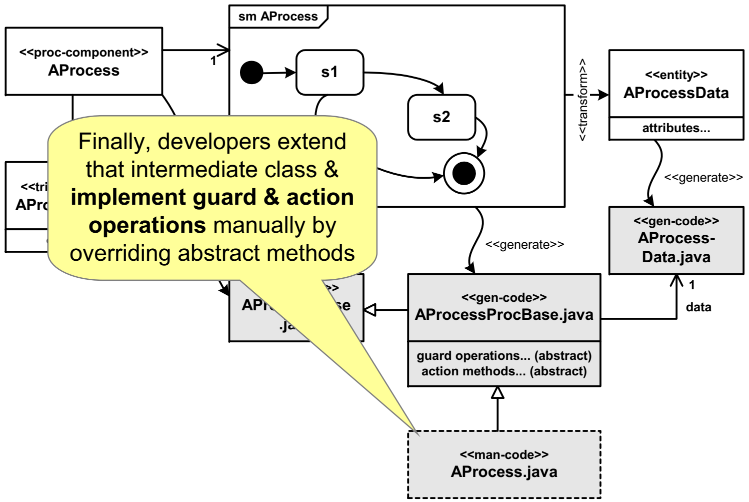

# DSL–based Programming Model

- We also use **cascading** for the Process Components



# DSL–based Programming Model

- We also use **cascading** for the Process Components






# Architectural Case Study

- PHASE 1: Elaborate!
  - Technology-Independent Architecture
  - Programming Model
  - Technology Mapping
  - Mock Platform
  - Vertical Prototype
- PHASE 2: Iterate!
- PHASE 3: Automate!
  - Architecture Metamodel
  - Glue Code Generation
  - DSL-based Programming Model
  - **Model-based Architecture Validation**



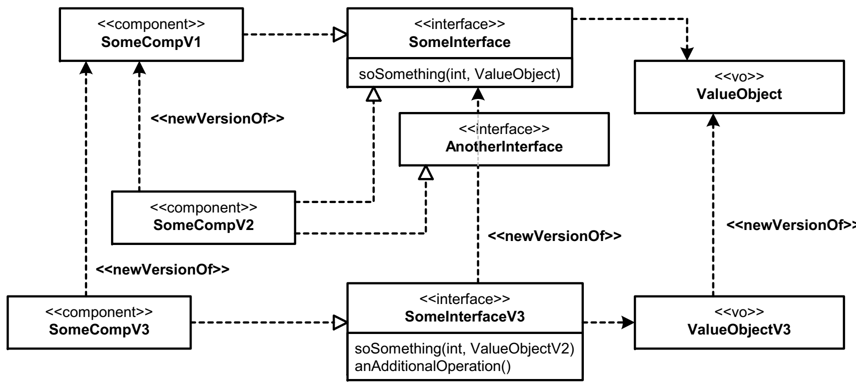
# Model–Based Architecture Validation

- We can use automated model checking to verify that
  - For **triggers** in processes there is a component that calls the trigger
  - **Dependency** management: It is easy to detect circular dependencies among components
  - Components are assigned to **layers** (app, service, base) & dependencies are only allowed in certain directions
- The component signature generated from the model prevents developers from creating dependencies to components that are not described in the model



# Model–Based Architecture Validation

- Another really important aspect in our example system is **evolution of interfaces**:



**Model-Driven Development** of Distributed Systems

# CONTENTS



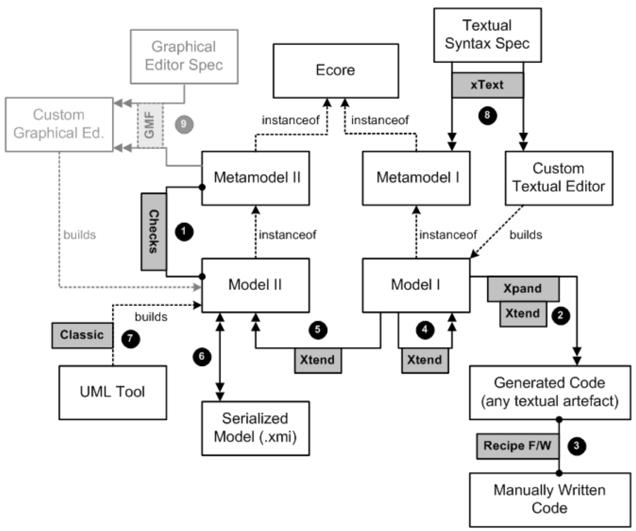


# Status / Track Record

- **Open Source**
- **Version 4.1 is current**
- **Proven track record in various domains & project contexts**
  - e.g., telcos, internet, enterprise, embedded realtime, finance, …
- **www.openarchitectureware.org**
- **IDE-portions based on Eclipse**
- (Optional) **Integration with Eclipse Modelling** facilities (such as EMF)



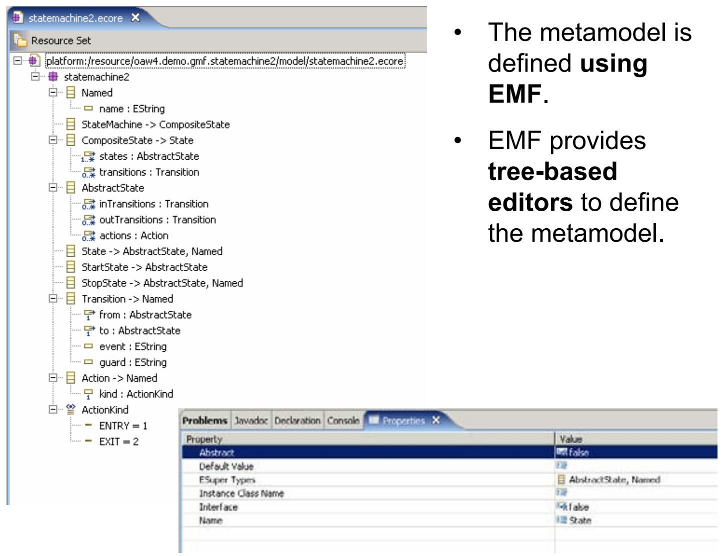
# Overview



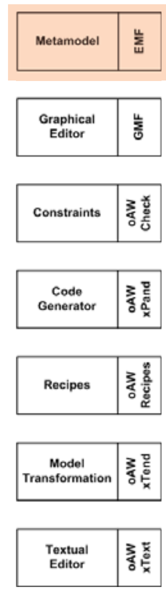

1. Model Verification using Constraints
2. Integrating Generating Code with Manually Written Code
3. Code Generation
4. Model Modification/Completion
5. Model-to-Model Transformation
6. Loading/Storing Models
7. Model Editing based on Custom-built Textual Editors
8. Model Editing using UML Tools
9. Model Editing based on Custom-built Graphical Editors (based on Eclipse GMF, not on openArchitectureWare)



# Defining the Metamodel




- The metamodel is defined using **EMF**.
- EMF provides **tree-based editors** to define the metamodel.



# Building the Graphical Editor

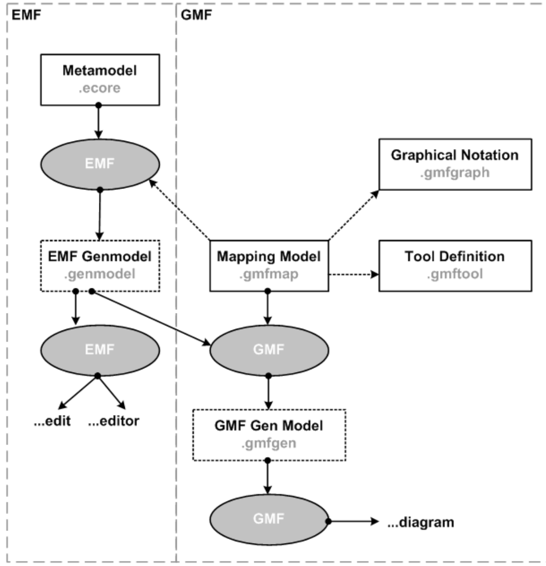
- The editor is **based on the metamodel** defined before.
- A number of additional models has to be defined:
  - A model defining the **graphical notation**
  - A model for the editor's **pallette &** other tooling
  - A **mapping model** that binds these two models to the domain metamodel
- A **generator** generates the concrete editor based on these models.

- The editor is build with the Eclipse GMF, the **Graphical Modelling Framework.**

Metamodel | EMF
Graphical Editor | GMF
Constraints | oAW Check
Code Generator | oAW xPand
Recipes | oAW Recipes
Model Transformation | oAW xTend
Textual Editor | oAW xText



# Building the Graphical Editor II

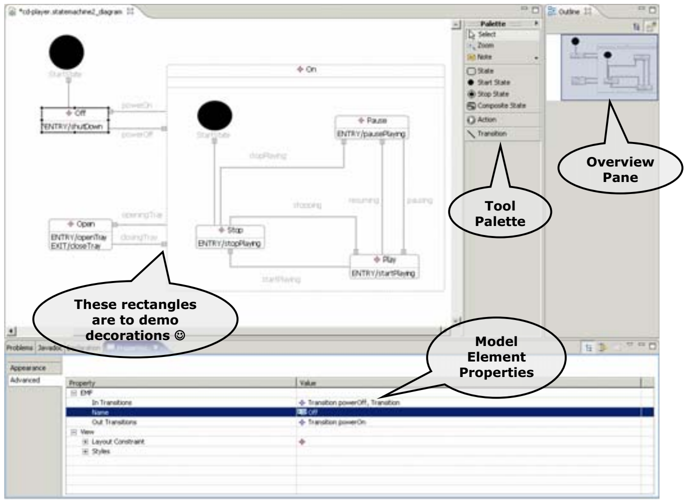



Metamodel | EMF
Graphical Editor | GMF
Constraints | oAW Check
Code Generator | oAW xPand
Recipes | oAW Recipes
Model Transformation | oAW xTend
Textual Editor | oAW xText



# Building the Graphical Editor III

- Here is the **editor**, started in the runtime workbench, with our CD Player example.




Metamodel | EMF
Graphical Editor | GMF
Constraints | oAW Check
Code Generator | oAW xPand
Recipes | oAW Recipes
Model Transformation | oAW xTend
Textual Editor | oAW xText



# Constraints

- Constraints are **rules that models must conform to** in order to be valid. These are in addition to the structures that the metamodel defines.
- A constraint is a **boolean expression (a.k.a predicate)** that must be true for a model to conform to a metamodel.
- Constraint Evaluation should be available
  - in **batch mode** (when processing the model)
  - as well as **interactively**, during the modelling phase in the editor

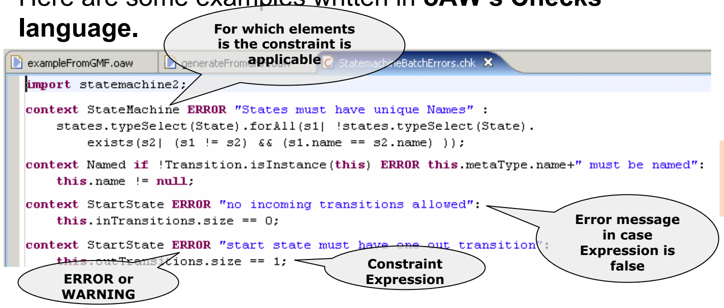
  ... & **we don't want to implement constraints twice** to have them available in both places!

- **Functional languages** are often used here.
  - UML's OCL (Object Constraint Language) is a good example,
  - We use **oAW's check language**, which is alike OCL

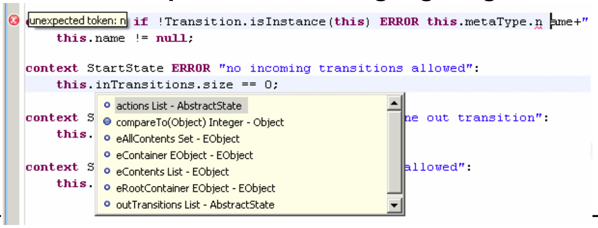
Metamodel | EMF
Graphical Editor | GMF
Constraints | oAW Check
Code Generator | oAW xPand
Recipes | oAW Recipes
Model Transformation | oAW xTend
Textual Editor | oAW xText

# Constraints II

- Here are some examples written in **oAW's Checks language.**




- Note the **code completion** & **error highlighting** ☺

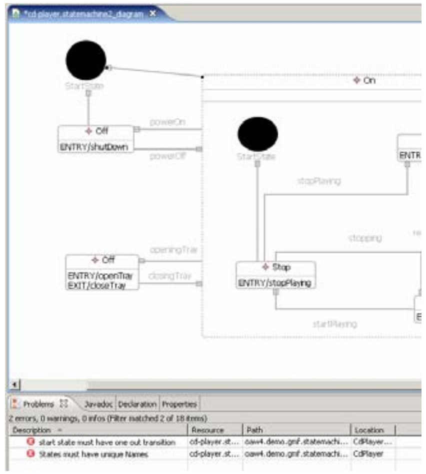







# Constraints III

- In this model there are **two errors**
  - There are two states with the same name (Off)
  - The start state has more than one out-Transition
- The validation is executed automatically
- Clicking the error message **selects** the respective "broken" **model element** in the diagram.

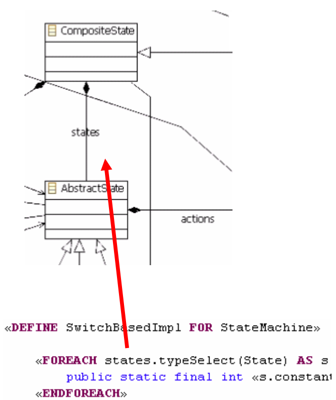






# Code Generation

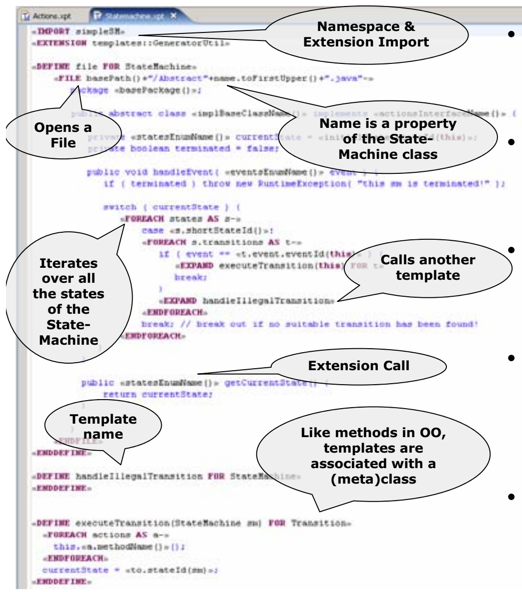
- Code Generation is used to **generate executable code** from models.
- Code Generation is **based on the metamodel** & uses **templates** to attach to-be-generated source code.
- In openArchitectureWare, we use a **template language** called **xPand.**
- It provides a number of **advanced features** such as polymorphism, AO support and a powerful integrated expression language.
- Templates can access **metamodel properties** seamlessly







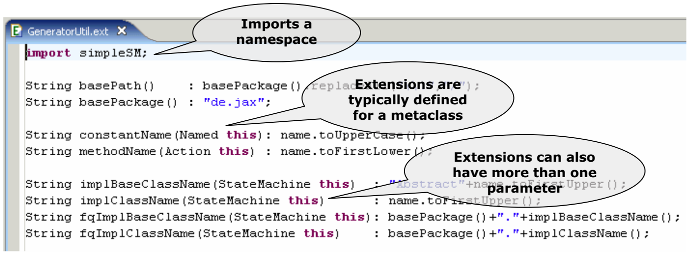

# Code Generation II



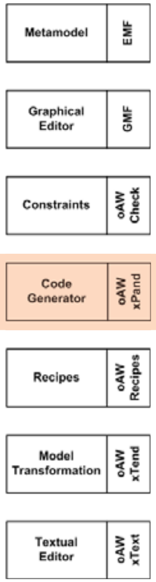

- The **blue text** is generated into the target file.
- The **capitalized words** are xPand keywords
- **Black text** are metamodel properties
- DEFINE...END-DEFINE blocks are called **templates**.
- The whole thing is called a **template file.**

# Code Generation III

- One can **add behaviour to existing metaclasses** using oAW's **Xtend** language.



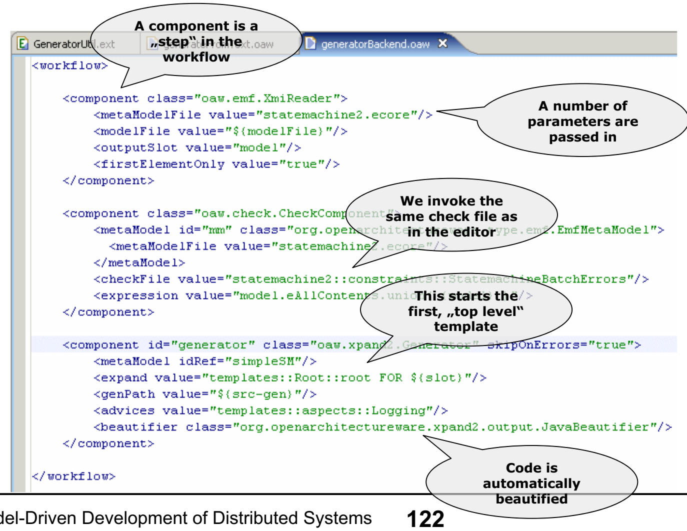

- Extensions can be called using **member-style syntax**: *myAction.methodName()*
- Extensions can be used in **Xpand templates, Check files** as well as in other **Extension files**.
- They are imported into template files using the **EXTENSION** keyword

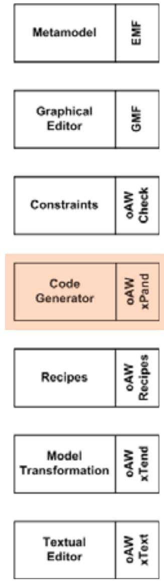





# Code Generation IV

- Workflow **loads** the model, **checks** it (same constraints as in Editor!) & then **generates** code.

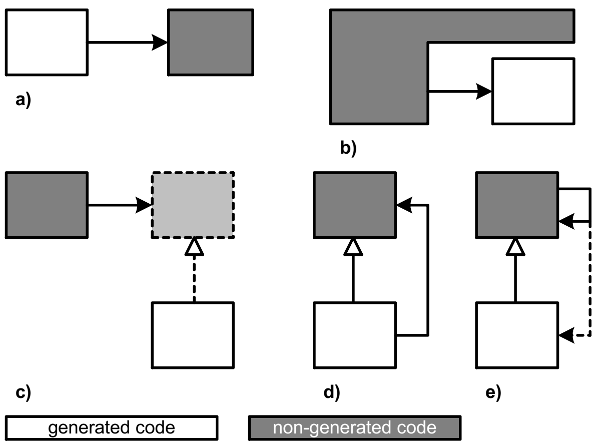


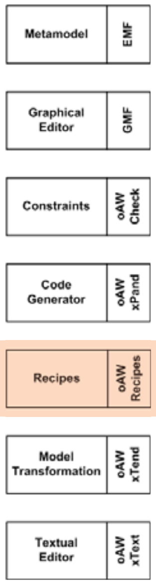






# Recipes I

- There are various ways of integrating generated code with non-generated code:

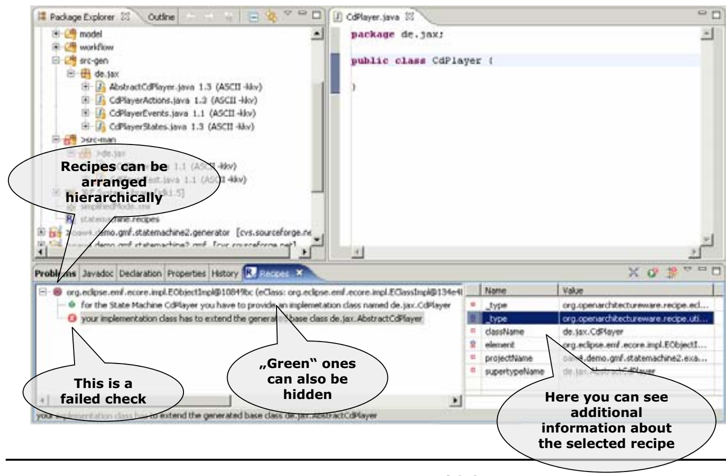


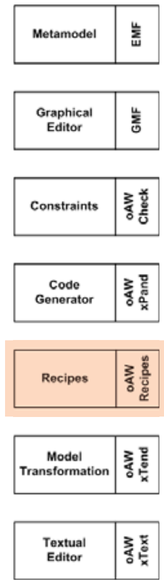






# Recipes II

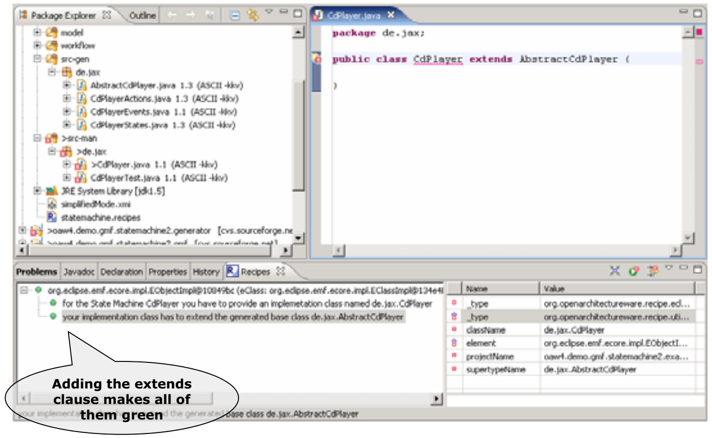
- Here's an error that suggests that I **extend** my manually written class **from the generated base class:**

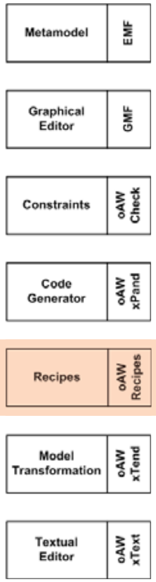


# Recipes III

- I now **add the respective *extends* clause**, & the message goes away – automatically.

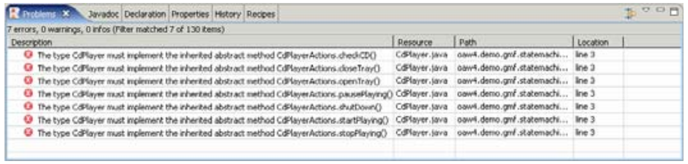


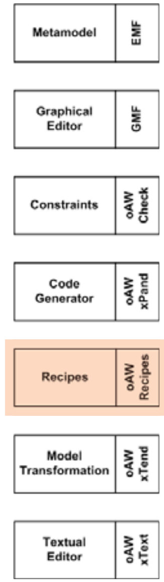



# Recipes IV

- Now I get a number of compile errors because I have to **implement the abstract methods** defined in the super class:



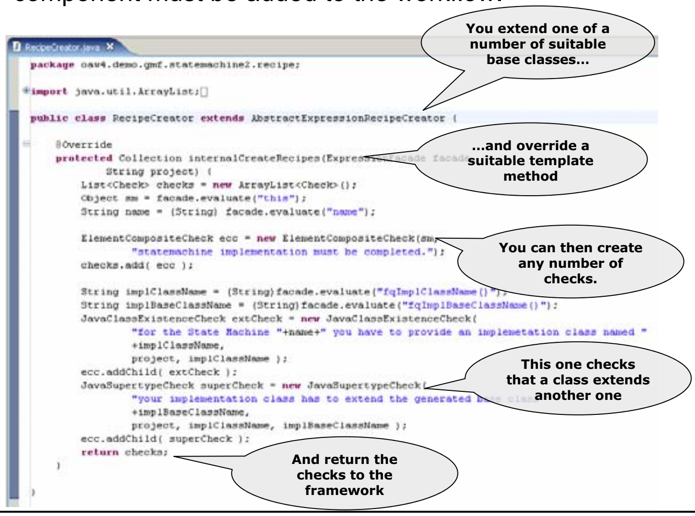
- I finally implement them sensibly, & everything is ok.
- The Recipe Framework & the Compiler have **guided me through the manual implementation steps.**
  - If I didn't like the compiler errors, we could also add recipe tasks for the individual operations.
  - oAW comes with a number of **predefined recipe checks for Java**. But you can also define your own checks, e.g. to verify C++ code.

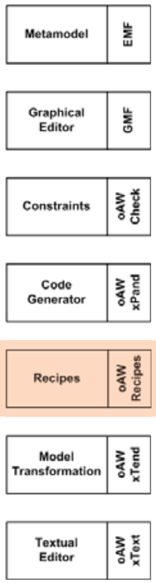





# Recipes V

- Here's the **implementation of the Recipes**. This workflow component must be added to the workflow.

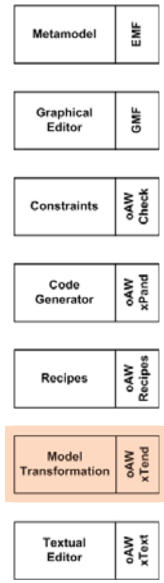





# Model Transformations I

- **Model Transformations** create one or more new models from one or more input models. The input models are left unchanged.
  - Often used for stepwise refinement of models & modularizing generators
  - Input/Output Metamodels are different
- **Model Modifications** are used to alter or complete an existing model
- For both kinds, we use the **xTend language**, an extension of the openArchitectureWare expression language.
- **Alternative languages** are available such as Wombat, ATL, MTF or Tefkat (soon: various QVT implementations)



# Model Transformation II

- The **model modification** shows how to add an additional state & some transitions to an existing state machine (emergency shutdown)

AddEmergencyShutdown.ext

```
import statemachine2;

extension statemachine2::constraints::Statemachine;

StateMachine modify(StateMachine sm) :
   sm.transitions.addAll(sm.allConcreteStates().createTransition()) ->
   sm.states.add(createShutDown()) ->
   sm;

private create State this createShutDown() :
   setName("EmergencyShutDown");

private create Transition this createTransition(State s) :
   setEvent("Error")->
   setName("Aborting") ->
   setFrom(s) ->
   setTo(createShutDown());
```

**Extensions can import other extensions**

**The main function**

**„create extensions" guarantee that for each set of parameters the *identical* result will be returned.**

**Therefore createShutDown() will always return the same element.**

| | |
|---|---|
| Metamodel | EMF |
| Graphical Editor | GMF |
| Constraints | oAW Check |
| Code Generator | oAW xPand |
| Recipes | oAW Recipes |
| **Model Transformation** | oAW xTend |
| Textual Editor | oAW xText |



# Model Transformation III

- The generator is based on an **implementation-specific metamodel** without the concept of composite states.
- This makes the **templates simple**, because we don't have to bridge the whole abstraction gap (from model to code) in the templates.
- Additionally, the **generator is more reusable**, because the abstractions are more general.
- We will show a transformation which transforms models described with our GMF editor into models expected by the generator.


| | |
|---|---|
| Metamodel | EMF |
| Graphical Editor | GMF |
| Constraints | oAW Check |
| Code Generator | oAW xPand |
| Recipes | oAW Recipes |
| **Model Transformation** | oAW xTend |
| Textual Editor | oAW xText |




# Model Transformation IV

- We want to transform from the editor's metamodel '**statemachine2**' to the generator's metamodel '**simpleSM**'

GMF2SimpleSM.ext

```
import statemachine2;

extension statemachine2::constraints::Statemachine;
extension org::openarchitectureware::util::IO;

create simpleSM::StateMachine createStatemachine(StateMachine sm) :
  setName(sm.name) ->
  setInitialState(sm.concreteState().createState() ) ->
  states.addAll(sm.allConcreteStates().createState()) ->
  actions.addAll(sm.eAllContents.typeSelect(Action).name.createAction()) ->
  events.addAll(sm.eAllContents.typeSelect(Transition).event.without({null}).createEvent());

private create simpleSM::State createState(State s) :
  setName(s.name) ->
  transitions.addAll(s.allOutTransitions().createTransition(s));

private create simpleSM::Action createAction(String n) :
  setName(n);

private create simpleSM::Event createEvent(String n) :
  setName(n);

private create simpleSM::Transition createTransition(Transition t,State s) :
  actions.addAll(allActions(s,t.to.concreteState()).name.createAction()) ->
  setEvent(t.event.createEvent()) ->
  setTo(t.to.concreteState().createState());
```

- **We need to 'normalize' composite states.**
- **States inherit outgoing transitions from their parent states**
- **For those transitions the exit actions are inherited, too**
- **Unify action & event elements with the same name**


| | |
|---|---|
| Metamodel | EMF |
| Graphical Editor | GMF |
| Constraints | oAW Check |
| Code Generator | oAW xPand |
| Recipes | oAW Recipes |
| **Model Transformation** | oAW xTend |
| Textual Editor | oAW xText |




# Textual Editor I

- A graphical notation is not always the best syntax for DSLs.
- So, while GMF provides a means to generate editors for graphical notations, we also need to be able to come up with **editors for textual syntaxes**.
- These **editors need to include** at least
  - Syntax hightlighting
  - Syntax error checking
  - Semantic constraint checking

| | |
|---|---|
| Metamodel | EMF |
| Graphical Editor | GMF |
| Constraints | oAW Check |
| Code Generator | oAW xPand |
| Recipes | oAW Recipes |
| Model Transformation | oAW xTend |
| **Textual Editor** | oAW xText |

# Textual Editor II

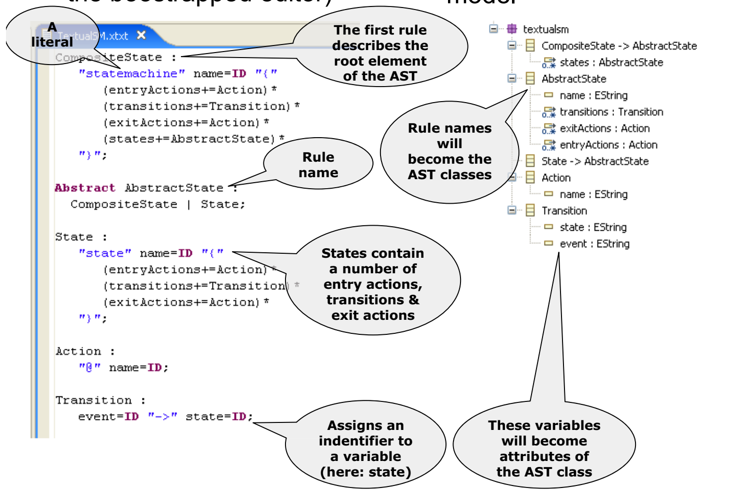
- We use oAW's textual DSL generator framework **xText**
- Based on a BNF-like language it provides:
  - An **EMF-based metamodel** (representing the AST)
  - An **Antlr parser** instantiating **dynamic** EMF-models
  - An **Eclipse text editor plugin** providing
    - **syntax highlighting**
    - An **outline view**,
    - **syntax checking**
    - as well as **constraints checking** based on a *Check* file, as always oAW





# Textual Editor III

- The **grammar** (shown in the boostrapped editor)
- The **generated** eCore AST model

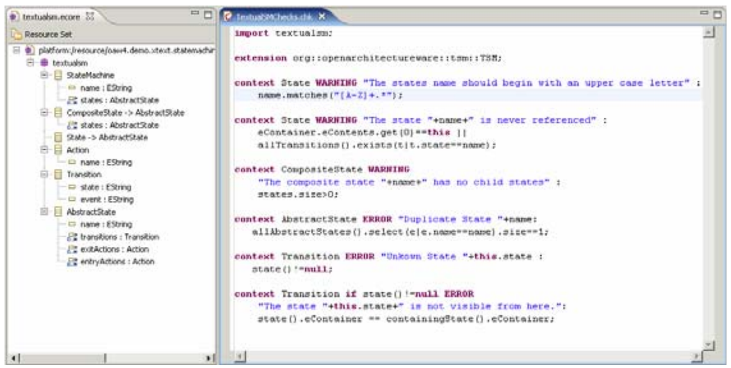



Metamodel | EMF
Graphical Editor | GMF
Constraints | oAW Check
Code Generator | oAW xPand
Recipes | oAW Recipes
Model Transformation | oAW xTend
Textual Editor | oAW xText



# Textual Editor IV

- You can define **additioal constraints** that should be validated in the generated editor.
- This is based on oAW's *Check* language
  - i.e. These are constraints like all the others you've already come across

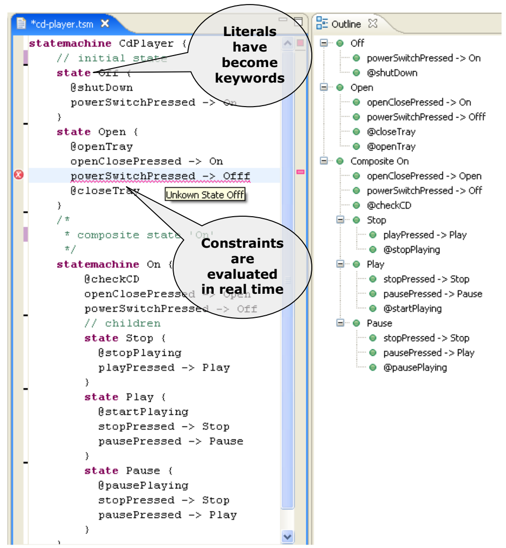



Metamodel | EMF
Graphical Editor | GMF
Constraints | oAW Check
Code Generator | oAW xPand
Recipes | oAW Recipes
Model Transformation | oAW xTend
Textual Editor | oAW xText



# Textual Editor V

- **The generated editor** & it's **outline view**




Metamodel | EMF
Graphical Editor | GMF
Constraints | oAW Check
Code Generator | oAW xPand
Recipes | oAW Recipes
Model Transformation | oAW xTend
Textual Editor | oAW xText

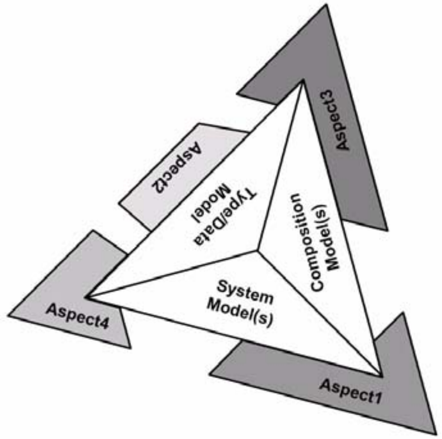
# CONTENTS





# Why?

- Based on our experience, the core "asset" in model-driven component based development is **not** a generator that generated some J2EE code, rather, the **"right" selection of models & viewpoints** is essential

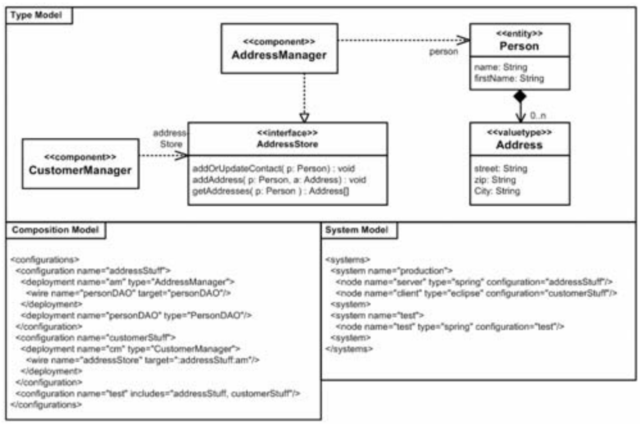



- So these slides contain exactly this: **a reference metamodel** that has been used in many, many different projects



# Three Basic Viewpoints

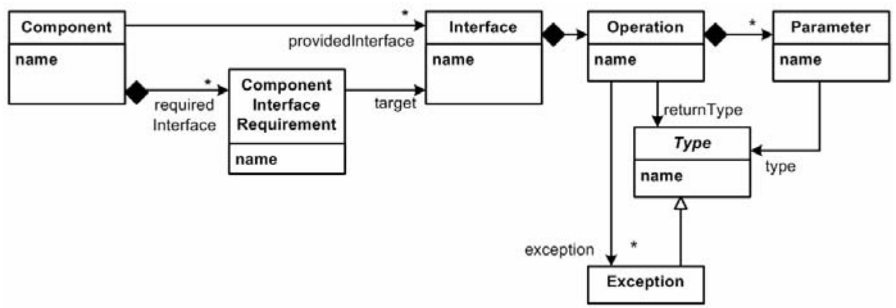
- **Type Model**: Components, Interfaces, Data Types
- **Composition Model**: Instances, "Wirings"
- **System Model**: Nodes, Channels, Deployments





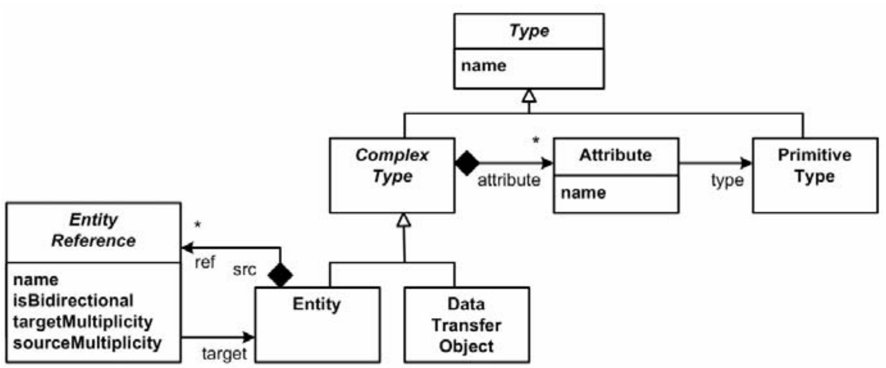

# Type Metamodel

- Components
- Interfaces
- Operations



# Type Metamodel II (Data)

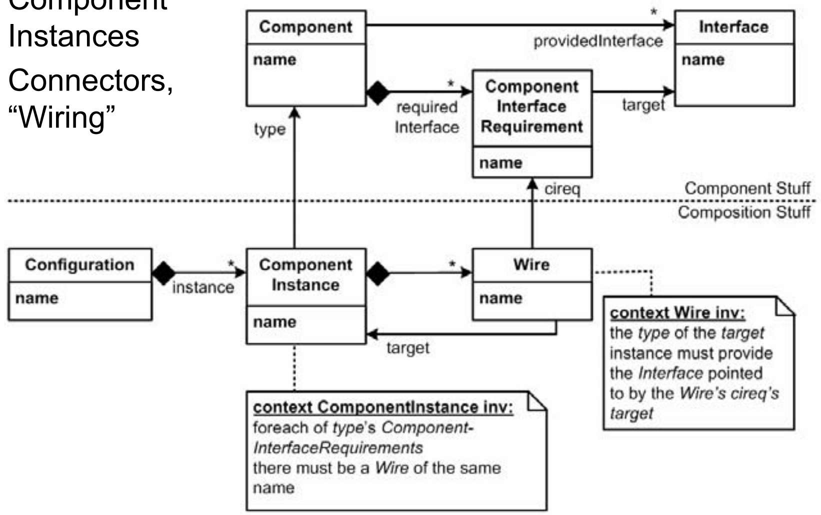
- Data Types
- Cross-References






# Composition Metamodel

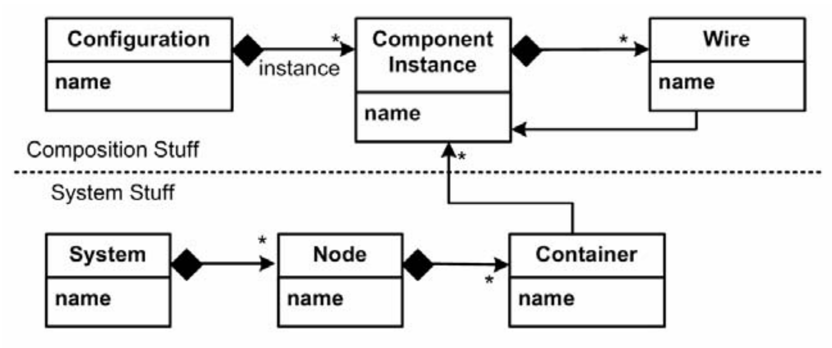
- Component Instances
- Connectors, "Wiring"






# System Metamodel

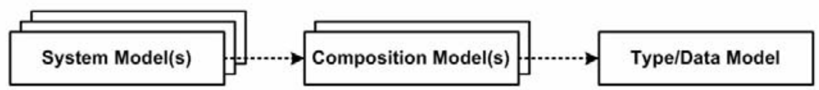
- Hardware
- Deployment





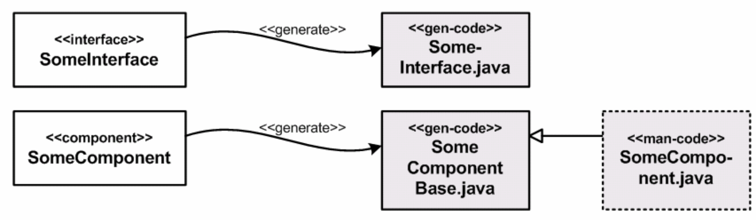

# Viewpoint Dependencies

- Dependencies between Viewpoint Models **are only allowed in the way shown below** in order to
  - Be able to have several compositions per type model
  - And several system models per composition
- This is important to be able to have **several "systems",**
  - Several deployed locally for testing, using only a subset of the defined components,
  - And "the real system"



# Component Implementation

- We have not yet talked about the **implementation code** that needs to go along with components.
  - As a default, you will provide the implementation by a **manually written subclass**

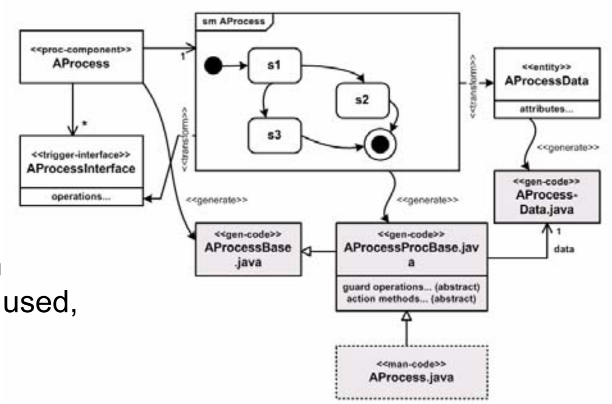



- However, for **special kinds of components** ("component kind" will be defined later) can use different implementation strategies -> **Cascading!**



# Component Implementation II

- Remember the **example of the process components** from before:

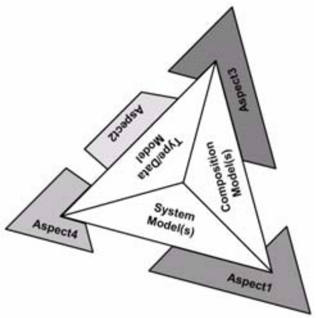



- Various other **implementation stragies** can be used, such as:
  - Rule-Engines
  - "Procedural" DSLs or action semantics
- Note that, here, **interpreters** can often be used sensibly instead of generating code!



# Aspect Models

- Often, the described three viewpoints are not enough, **additional aspects** need to be described.
- These go into **separate aspect models**, each describing a well-defined aspect of the system.
  - Each of them uses a suitable DSL/syntax
  - The generator acts as a weaver
- Typical **Examples** are
  - Persistence
  - Security
  - Forms, Layout, Pageflow
  - Timing, QoS in General
  - Packaging & Deployment
  - Diagnostics & Monitoring



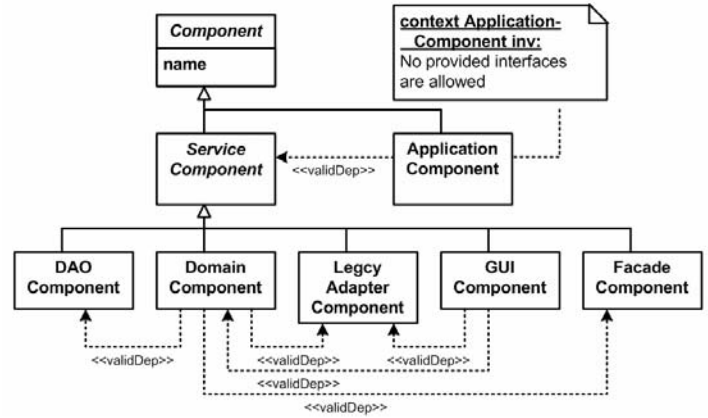



# Separate Interfaces

- You might **not need separate Interfaces**
  - Operations could be annotated directly to components
  - Dependencies would be to components, not to interfaces
- **Relationships between interfaces** are often needed,
  - "if you require this interface, you also have to provide that one"

# Component Types

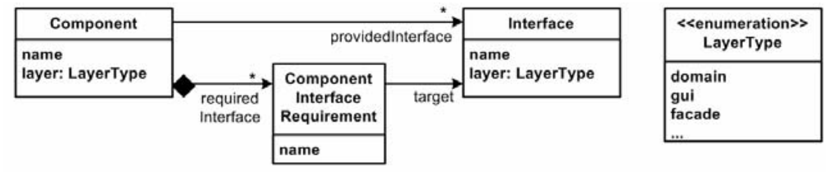
- Often different **"kinds" of Components** are needed.
  - To **manage dependencies**,
  - And to define **implementation strategies**






# Component Layering

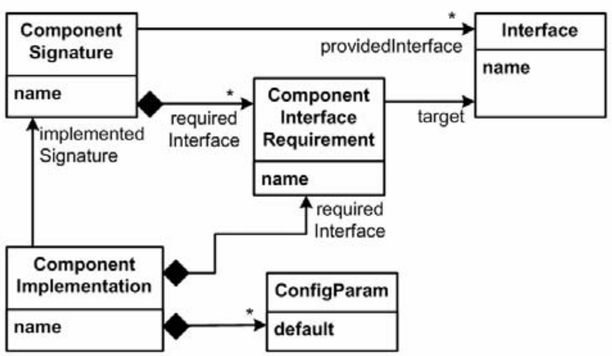
- Alternatively you can simply **annotate each component with a layer**






# Component Signatures

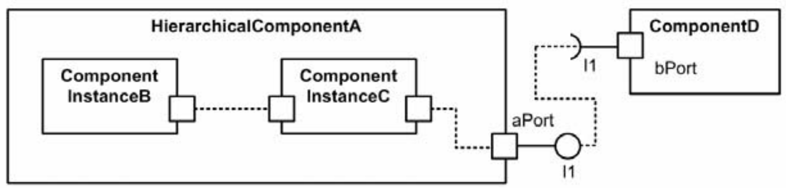
- You might need to **provide several implementations** (i.e. components) for the same signature (i.e. provided/required interfaces).
  - So you need to separate implementation from signature



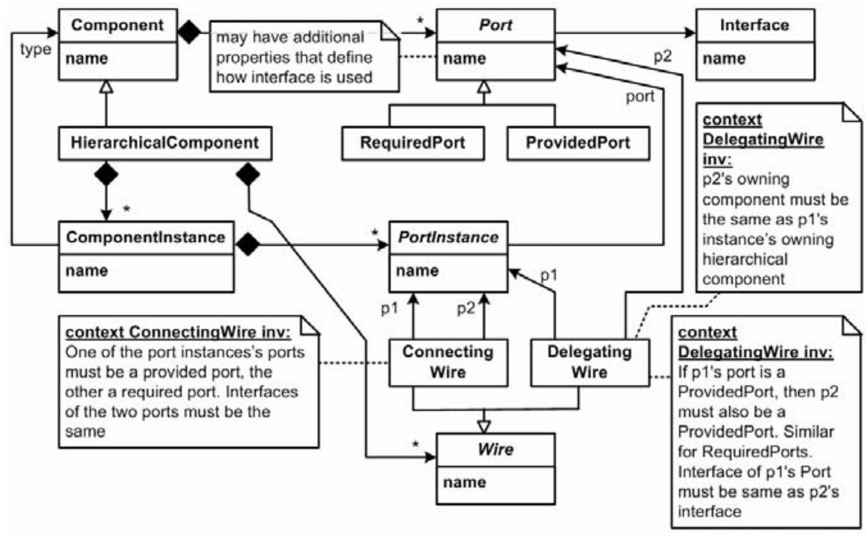



# Hierarchical Components I

- This allows an **infinite nesting** of component structures
- It requires the concept of **ports**
- Note that the clear **boundaries** between type & composition models **are blurted** (which makes this approach a bit more advanced!)
- Example:



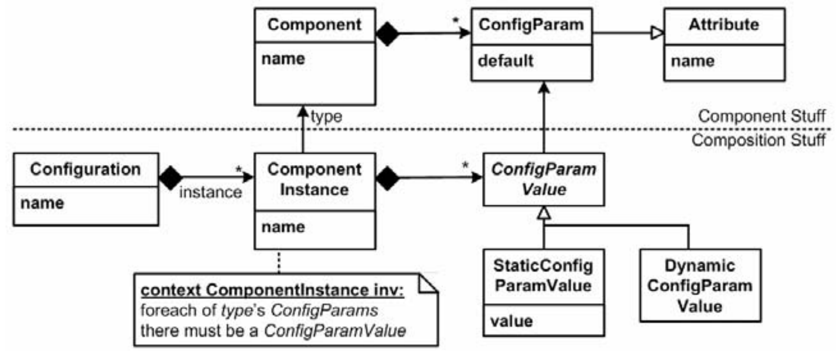
# Hierarchical Components II






# Configuration Parameters

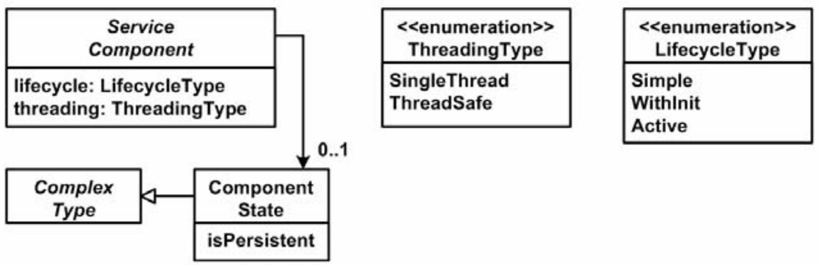
- Parameters allow for **dynamic configuration** of components.
- There is a wide **variety of potential value definition** scopes






# Behaviour

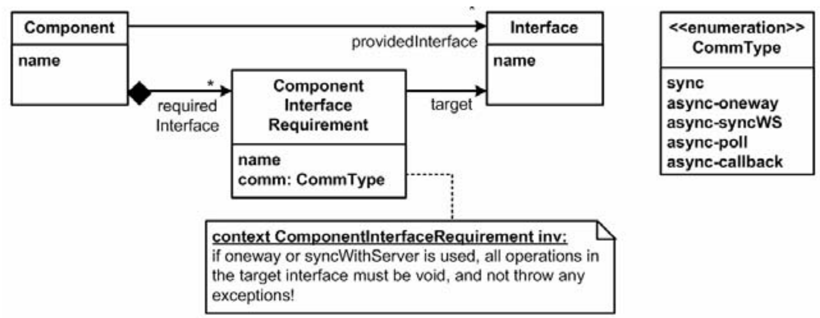
- Different (types of) Components typically have different **lifecycles**
- The **threading model** is typically different, too.
- Also, some components might be **stateless**, while others are **stateful** (with persistent state, or not)





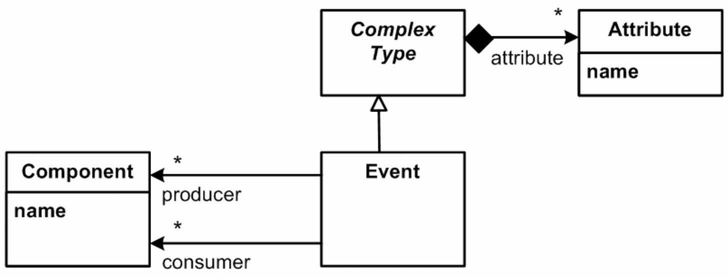

# Asynchronous Communication

- Some components might need **asynchronous communication** with others
  - Note that this has to be **specified in the type model** – since it affects the API!



# Events

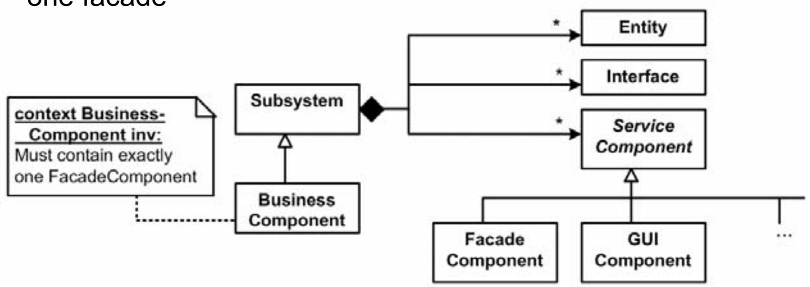
- Events are a way to **signal information** from a component to another, **asynchronously**.
  - Sometimes it is useful to allow for violations of the (otherwise rigidly enforced) dependency rules






# Subsystems & Business Components

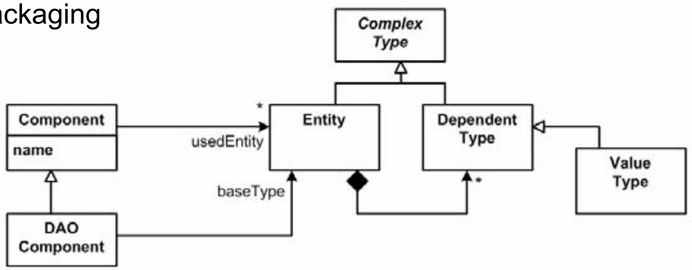
- If the number of components grows, **additional means to organize them** are required.
- The **internal structure** of subsystems or business components can be defined by enforcing certain policies wrt. Component types
  - For example, each business component must have exactly one facade






# Data

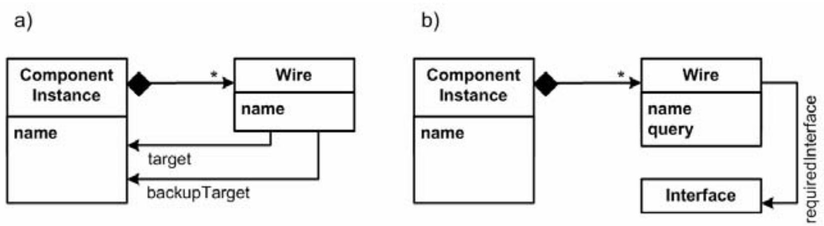
- More **elaborate data** structures are often required
  - Typical example is based on entities & dependent types
- DAOComponents are used to **manage the entities &** their associated dependent types
- **Ownership & Scope** of data types is essential
  - Indirect dependency management
  - packaging





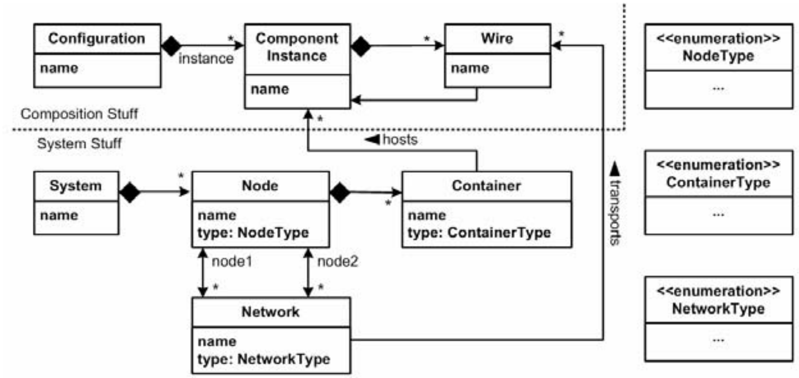

# Wiring

- **Optional** wires might be useful
- **Dynamic Wires** don't specify the target instance, but rather a set of properties based on which at runtime, the target can be found
  - Important for dynamic systems, e.g. P2P



# Container Types & Networks

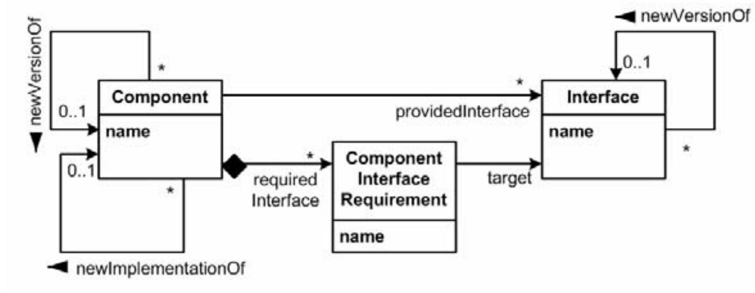
- This allows for **more specific description of hardware**,
  - **Networks & network types** describe means to communicate
  - Whereas **container types** are important to distinguish various execution environments (server, local, …)






# Versioning

- Capturing versioning & type evolution information explicitly in the model allows for definitive statements about **component compatibility & system evolution.**






**Model-Driven Development** of Distributed Systems

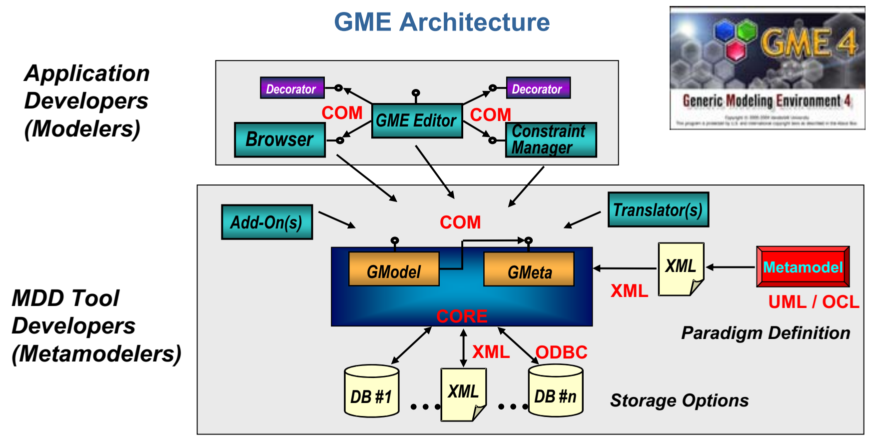
# CONTENTS





# Generic Modeling Environment (GME)

**"Write Code That Writes Code That Writes Code!"**

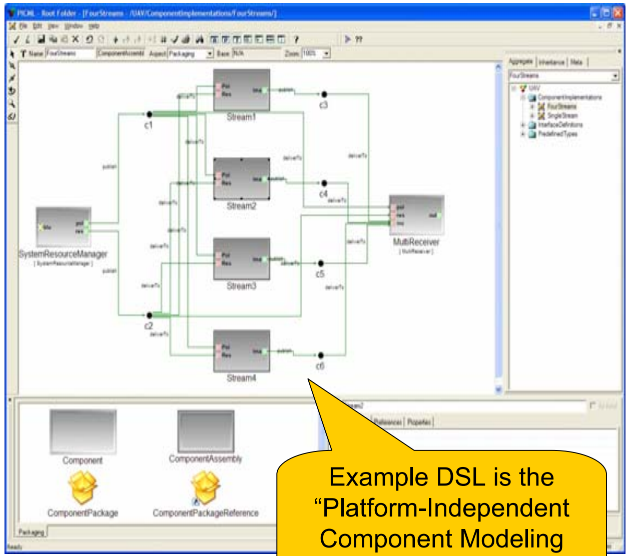



Supports "correct-by-construction" of software systems

GME is open-source: www.isis.vanderbilt.edu/Projects/gme/default.htm

# MDD Application Development with GME

- **Application developers** use modeling environments created w/MetaGME to build *applications*
  - Capture elements & dependencies visually

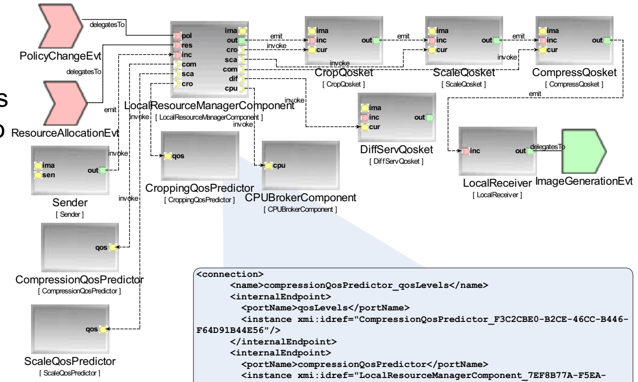



Example DSL is the "Platform-Independent Component Modeling Language" (PICML) tool



# MDD Application Development with GME

- **Application developers** use modeling environments created w/MetaGME to build *applications*
  - Capture elements & dependencies visually
  - Model interpreter produces something useful from the models
    - e.g., 3rd generation code, simulations, deployment descriptions & configurations





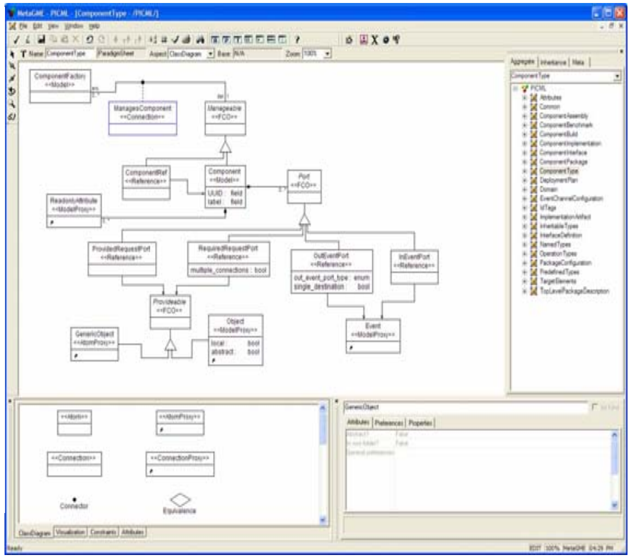
```
<connection>
      <name>compressionQosPredictor_qosLevels</name>
      <internalEndpoint>
        <portName>qosLevels</portName>
        <instance xmi:idref="CompressionQosPredictor_F3C2CBE0-B2CE-46CC-B446-F64D91B44E56"/>
      </internalEndpoint>
      <internalEndpoint>
        <portName>compressionQosPredictor</portName>
        <instance xmi:idref="LocalResourceManagerComponent_7EF8B77A-F5EA-4D1A-942E-13AE7CFED30A"/>
      </internalEndpoint>
    </connection>
    <connection>
      <name>scalingQosPredictor_qosLevels</name>
      <internalEndpoint>
        <portName>qosLevels</portName>
        <instance xmi:idref="ScaleQosPredictor_F3024...-F6E8-4B9A-BD56-A2E802C33E32"/>
      </internalEndpoint>
      <internalEndpoint>
        <portName>sca...</portName>
        <inst...ResourceManagerComponent_7EF8B77A-F5EA-
```


PICML generates XML descriptors corresponding to OMG Deployment & Configuration (D&C) specification



# MDD Tool Development in GME

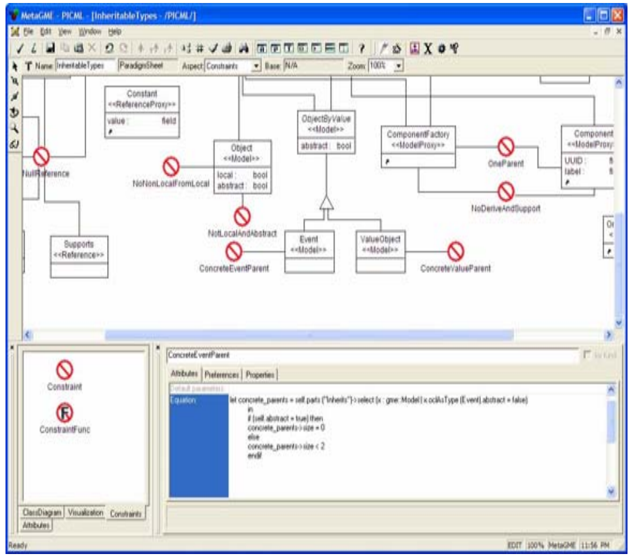
- **Tool developers** use MetaGME to develop a *domain-specific graphical modeling environment*
  - Define syntax & visualization of the environment via *metamodeling*





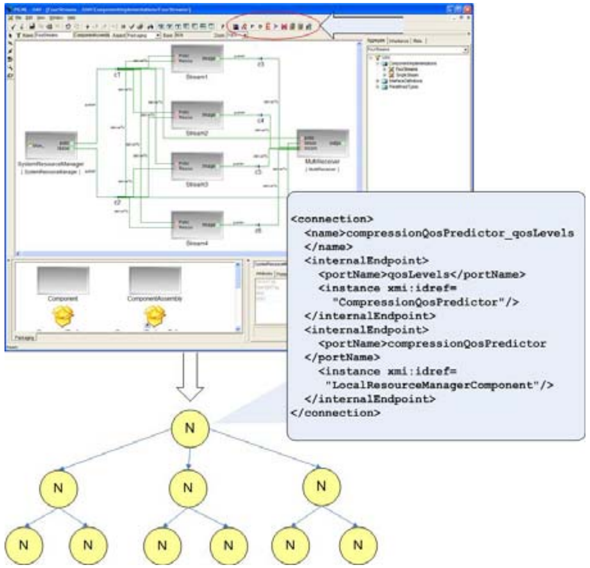
# MDD Tool Development in GME

- **Tool developers** use MetaGME to develop a *domain-specific graphical modeling environment*
  - Define syntax & visualization of the environment via *metamodeling*
  - Define static semantics via *Object Constraint Language (OCL)*



# MDD Tool Development in GME

- **Tool developers** use MetaGME to develop a *domain-specific graphical modeling environment*
  - –Define syntax & visualization of the environment via *metamodeling*
  - –Define static semantics via *Object Constraint Language (OCL)*
  - –Dynamic semantics implemented via *model interpreters*

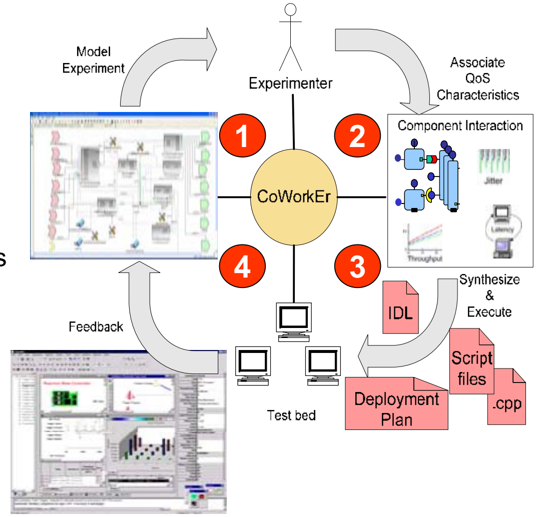




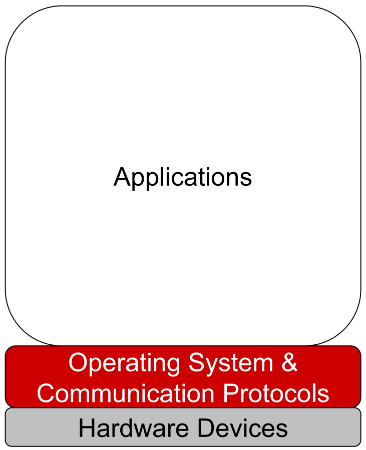

# Applying GME to System Execution Modeling

System Execution Modeling Workflow

1. Compose scenarios to exercise critical system paths/layers
2. Associate performance properties with scenarios & assign properties to components specific to paths/layers
3. Configure workload generators to run experiments, generate path-/layer-specific deployment plans, & measure performance along critical paths/layers
4. Feedback results into models to verify if deployment plan & configurations meet performance requirements



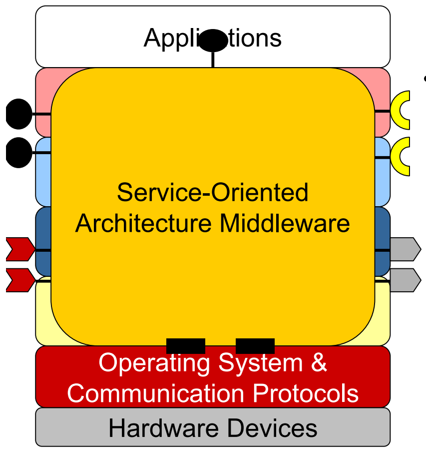

# Context: Service–Oriented Architectures




- Historically, distributed real-time & embedded (DRE) systems were built directly atop OS & protocols



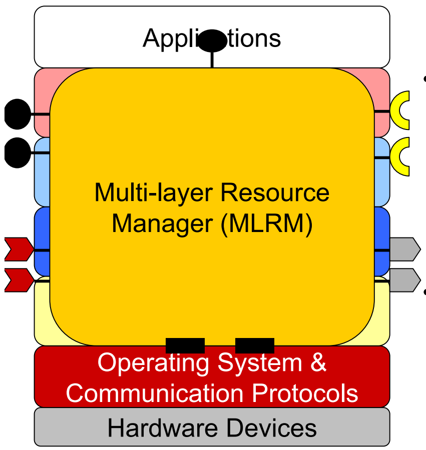
# Context: Service–Oriented Architectures




- Historically, distributed real-time & embedded (DRE) systems were built directly atop OS & protocols
- Traditional methods of development have been replaced by middleware layers to reuse architectures & code for *enterprise* DRE systems
  - Viewed externally as *Service-Oriented Architecture (SOA) Middleware*

Note: our techniques also apply to conventional enterprise distributed systems

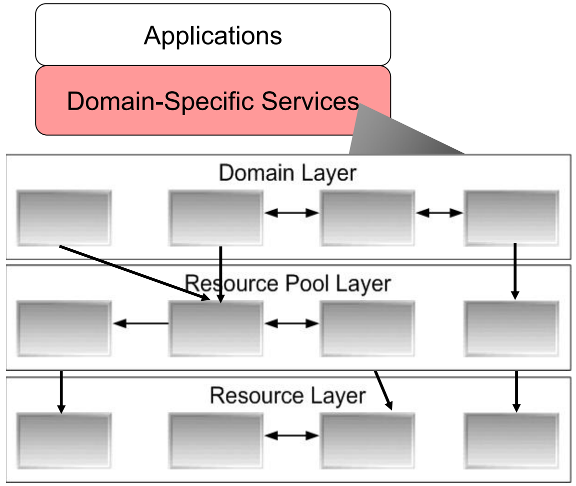
## Context: Service–Oriented Architectures




- Historically, distributed real-time & embedded (DRE) systems were built directly atop OS & protocols
- Traditional methods of development have been replaced by middleware layers to reuse architectures & code for *enterprise* DRE systems
  - Viewed externally as *Service-Oriented Architecture (SOA) Middleware*
- e.g., DARPA Adaptive & Reflective Management System (ARMS) program's *Multi-layer Resource Manager* (MLRM)
  - MLRM leverages standards-based SOA middleware to manage resources for shipboard computing environments


dtsn.darpa.mil/ixodarpatech/ixo_FeatureDetail.asp?id=6

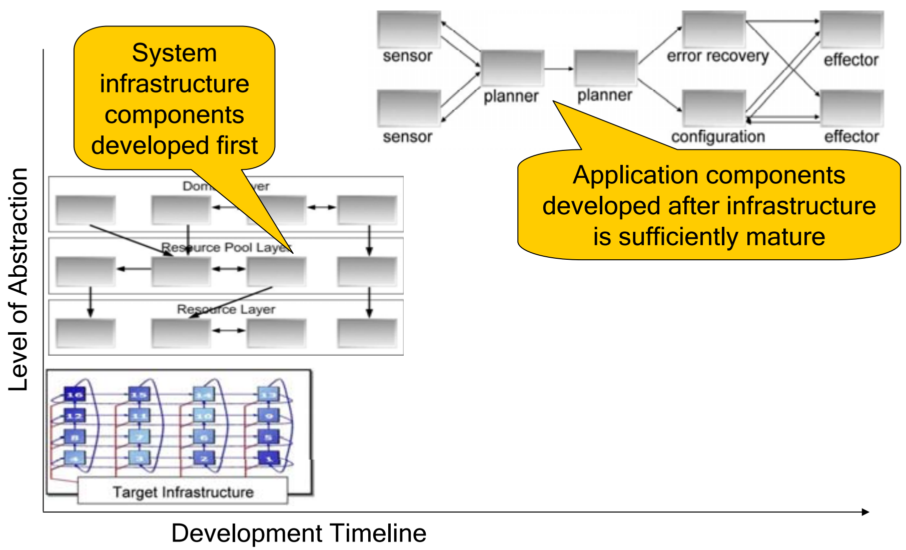
## ARMS Multi–Layer Resource Manager (MLRM)




- ARMS MLRM architecture includes
  - Top *domain layer* containing components that interact with the ship *mission manager*
  - Middle *resource pool layer* is an abstraction for a set of computer nodes managed by a *pool manager*
  - Bottom *resource layer* managers the actual resource computing components, i.e., CPUs, memory, networks, etc.

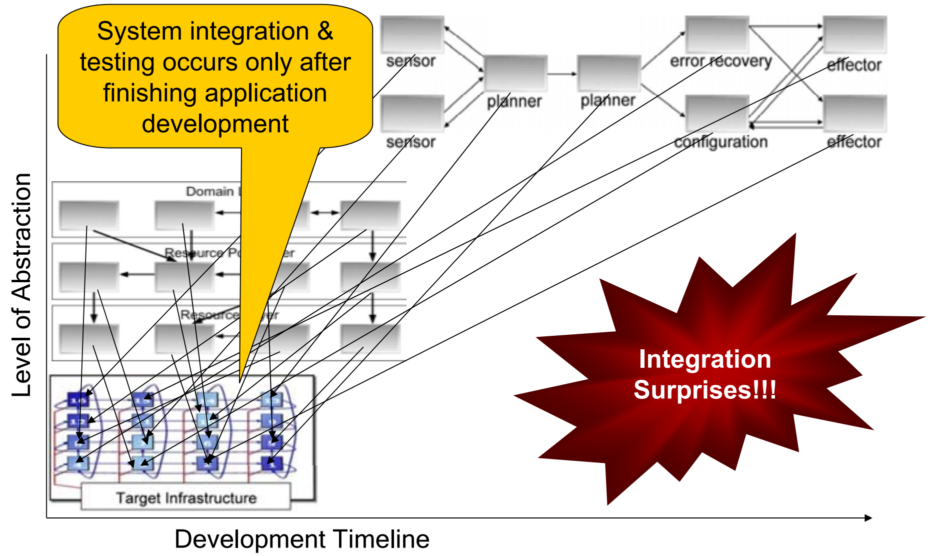

www.cs.wustl.edu/~schmidt/PDF/JSS-2006.pdf

## Serialized Phasing is Common in Enterprise DRE Systems

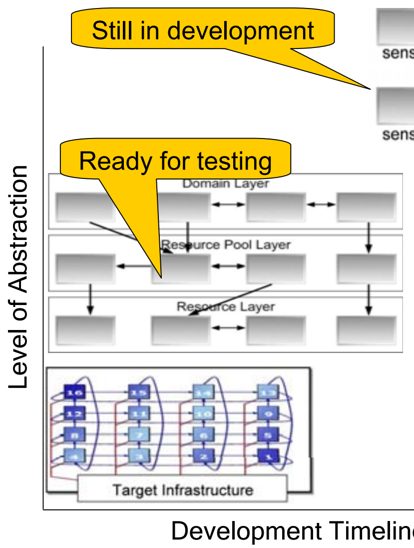


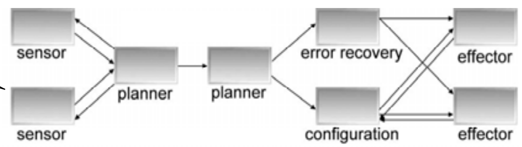



## Serialized Phasing is Common in Enterprise DRE Systems



## Complexities of Serialized Phasing



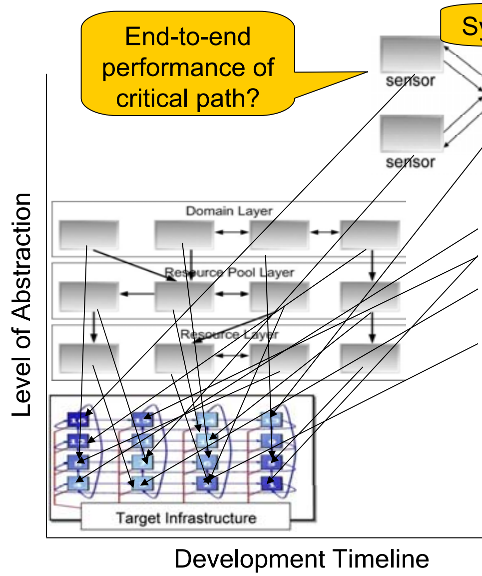



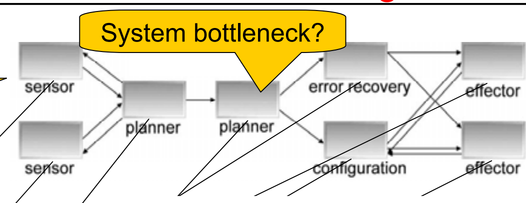

**Complexities**

- System infrastructure cannot be tested adequately until applications are done



## Complexities of Serialized Phasing



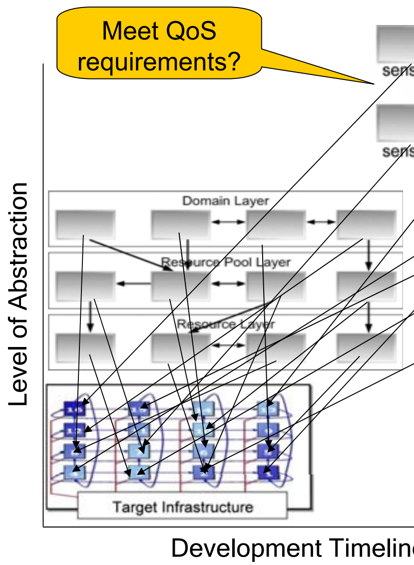



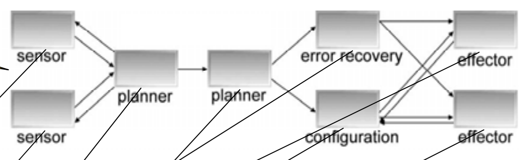

**Complexities**

- System infrastructure cannot be tested adequately until applications are done
- Entire system must be deployed & configured (D&C) properly to meet QoS requirements
- Existing evaluation tools do not support "what if" evaluation

Often, QoS requirements of components aren't known until late in the lifecycle

## Unresolved QoS Concerns with Serialized Phasing



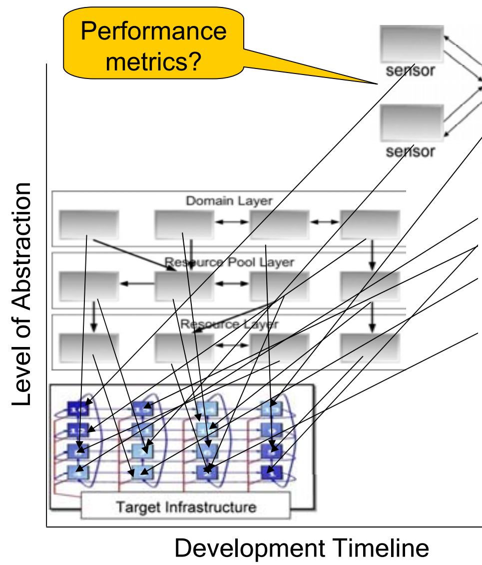



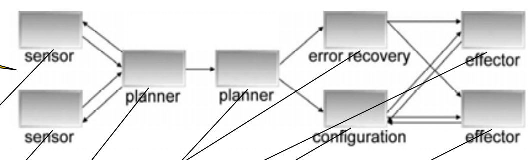

**Key QoS concerns**

- Which D&C's meet the QoS requirements?



## Unresolved QoS Concerns with Serialized Phasing

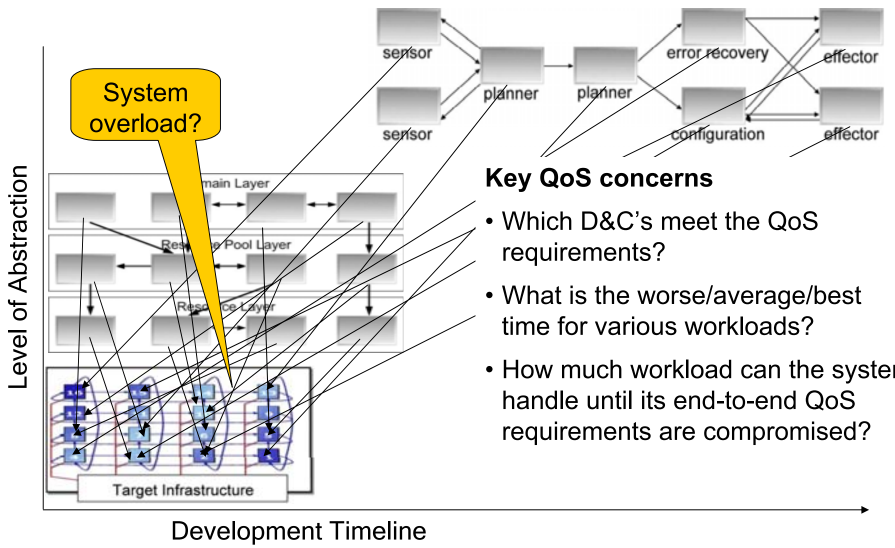



**Key QoS concerns**

- Which D&C's meet the QoS requirements?
- What is the worse/average/best time for various workloads?

# Unresolved QoS Concerns with Serialized Phasing

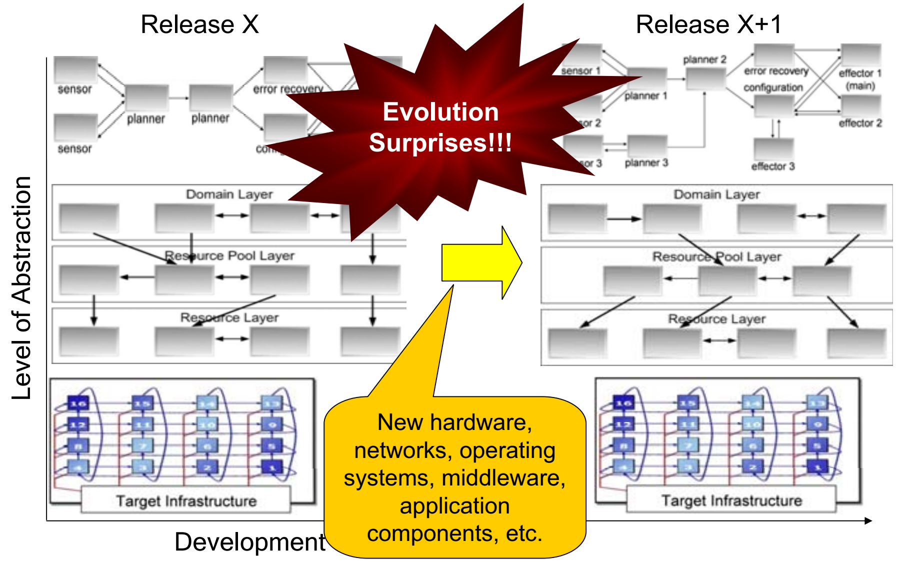



**Key QoS concerns**

- Which D&C's meet the QoS requirements?
- What is the worse/average/best time for various workloads?
- How much workload can the system handle until its end-to-end QoS requirements are compromised?

It can take a long time to address these concerns using serialized phasing!!

# Related Large-Scale System Development Problems






# Promising Solution Approach: New Generation of System Execution Modeling (SEM) Tools

**Tools to express & validate design rules**

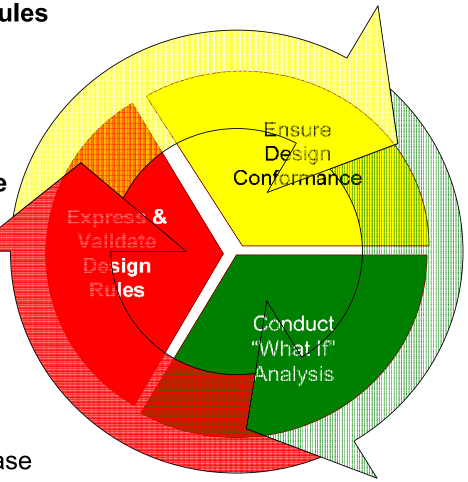
- Help applications adhere to system specifications at design-time
  - "Correct-by-construction"

**Tools to ensure design conformance**

- Help properly deploy & configure applications to enforce system *design rules* at run-time

**Tools to conduct "what if" analysis**

- Help analyze QoS concerns *prior* to completing the entire system
  - e.g., before system integration phase




The cycle is repeated when developing application & infrastructure components

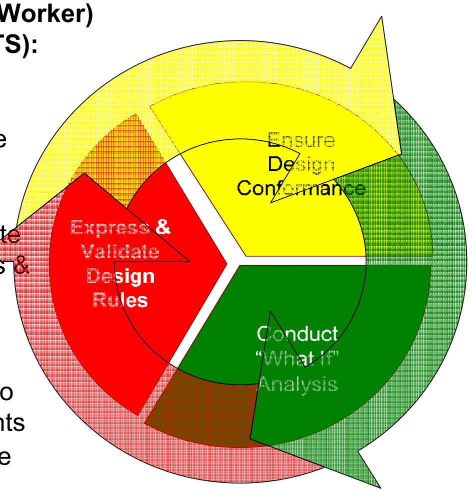


# Our Approach: Emulate Application Behavior via QoS-enabled SOA Middleware & MDD Tools

**Component Workload Emulator (CoWorker) Utilization Test Suite Workflow (CUTS):**

While creating target infrastructure

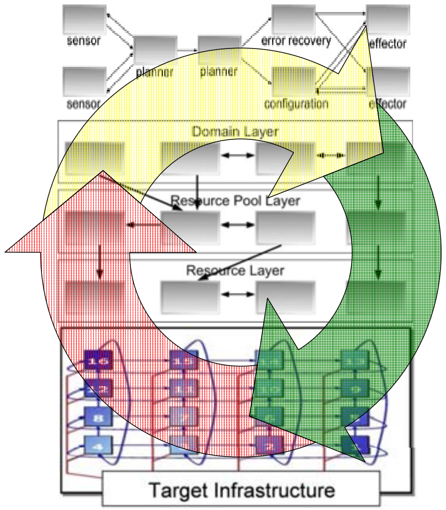
1. Use the PICML *domain-specific language* (DSL) to define & validate infrastructure specifications & requirements
2. Use PICML & WML DSLs to emulate & validate application specifications & requirements
3. Use CIAO & DAnCE middleware & PICML DSL to generate D&C metadata to ensure apps conform to system specifications & requirements
4. Use BMW analysis tools to evaluate & verify QoS performance
5. Redefine system D&C & repeat




Enable "application" testing to evaluate target infrastructure earlier in lifecycle

## Motivation for Using Emulation

- Can use actual target infrastructure
  - Rather than less precise simulations that abstract out key QoS properties
- Many artifacts can be used directly in the final production system
  - e.g., models of application component relationships & D&C plans
- Early feedback to developers, architects & systems engineers
  - Instead of waiting to complete application components before conducting performance experiments

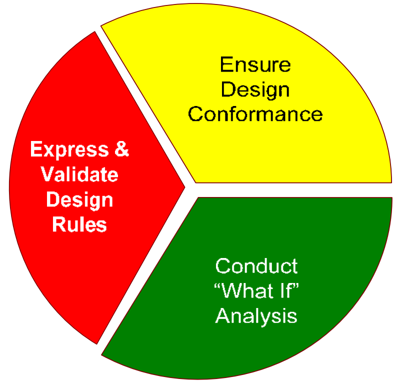





## Our SOA Middleware & MDD Tool Infrastructure

- **System Design & Specification Tools**
  - Define & validate system specification & requirements
- **System Assembly & Packaging Tools**
  - Compose implementation & configuration information into deployable assemblies
- **System Deployment Tools**
  - Automates the deployment of system components & assemblies to component servers
- **Component Implementation Framework**
  - Automates the implementation of many system component features

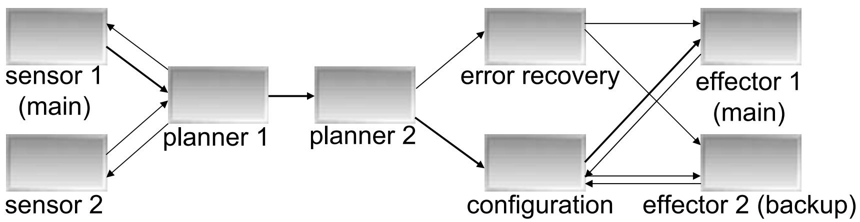
**PICML & CIAO & DAnCE**



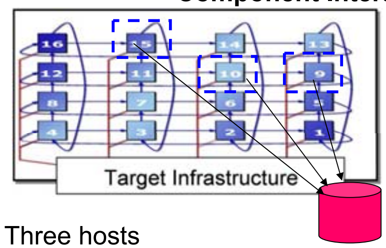

**CUTS & BMW**

www.dre.vanderbilt.edu/CIAO & www.dre.vanderbilt.edu/cosmic

## ARMS MLRM Case Study: SLICE Scenario (1/2)




**Component Interaction for SLICE Scenario**




- Three hosts
- One database is shared between all hosts (used largely offline)

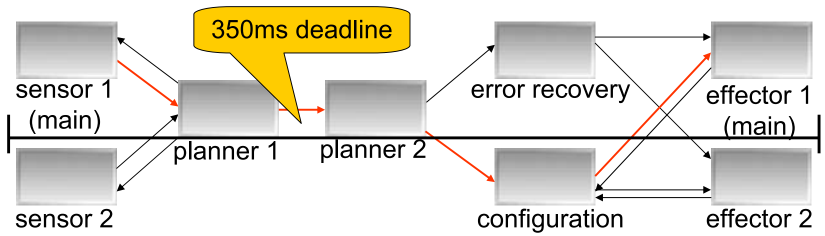
**D&C & Performance Requirements & Constraints**

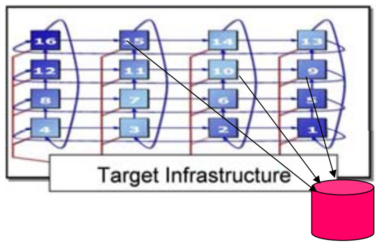
- *Critical path* deadline is 350 ms
  - Main sensor to main effector through configuration
- To ensure availability, components in critical paths should not be collocated
- Main sensor & main effector must be deployed on separate hosts

Example design rules



## ARMS MLRM Case Study: SLICE Scenario (2/2)

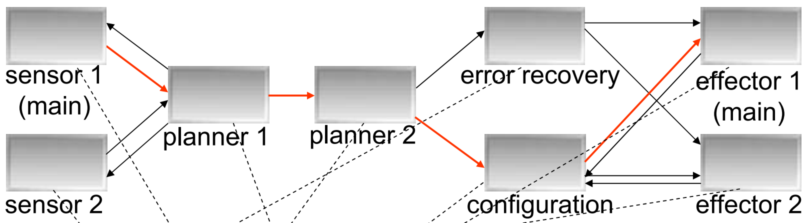


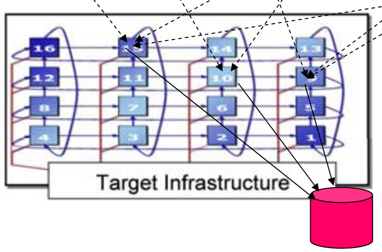




**Some questions we'd like to answer**

1. Can a particular D&C meet end-to-end performance requirements?

## ARMS MLRM Case Study: SLICE Scenario (2/2)



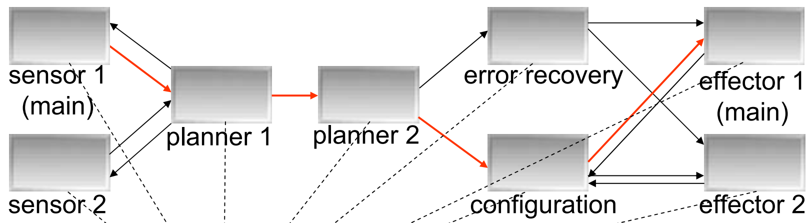



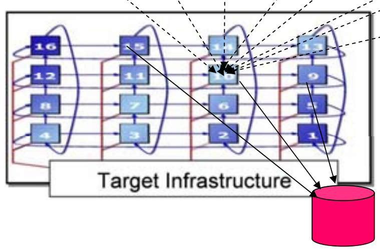

**Some questions we'd like to answer**

1. Can a particular D&C meet end-to-end performance requirements?
2. Are there multiple deployments that meet the 350ms critical path deadline?
   - e.g., which yields most headroom?



## ARMS MLRM Case Study: SLICE Scenario (2/2)



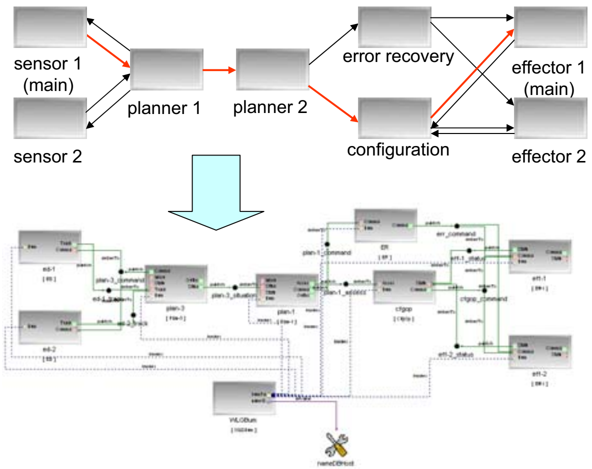

**Some questions we'd like to answer**

1. Can a particular D&C meet end-to-end performance requirements?
2. Are there multiple deployments that meet the 350ms critical path deadline?
   - e.g., which yields most headroom?
3. Can we meet the 350ms critical path deadline with all component deployed on a single host?



## Representing SLICE Scenario in PICML

**Conceptual model**

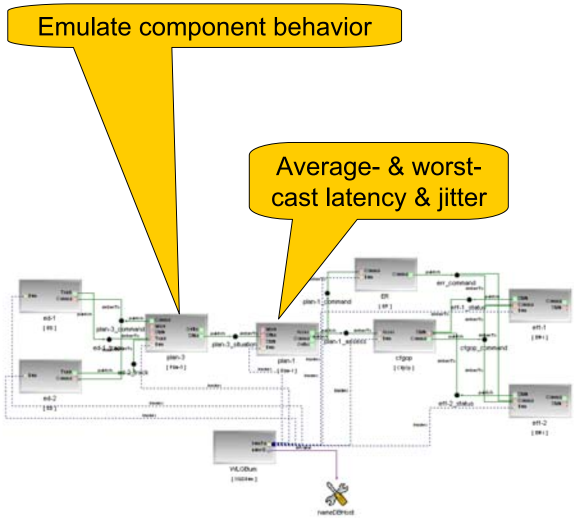



**PICML Model of SLICE Scenario**

- Conceptual models can be helpful at certain design phases
- But they are also imprecise & non-automated
- PICML model provides detailed representation of component properties & interconnections
- They are also precise & automated



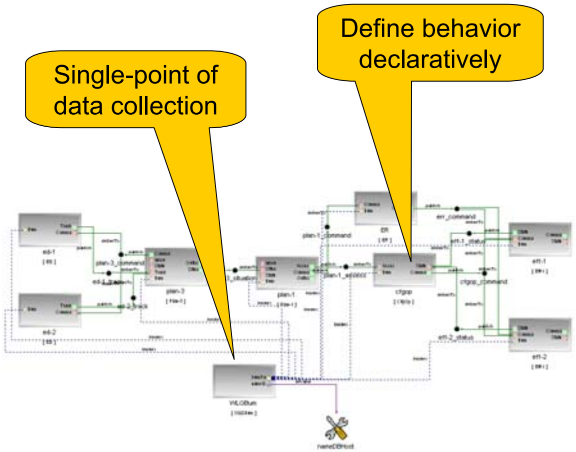
## Summary of CUTS Challenges




**PICML Model of SLICE Scenario**

1. Evaluate QoS characteristics of DRE systems
2. Emulate QoS characteristics of DRE systems

## Summary of CUTS Challenges

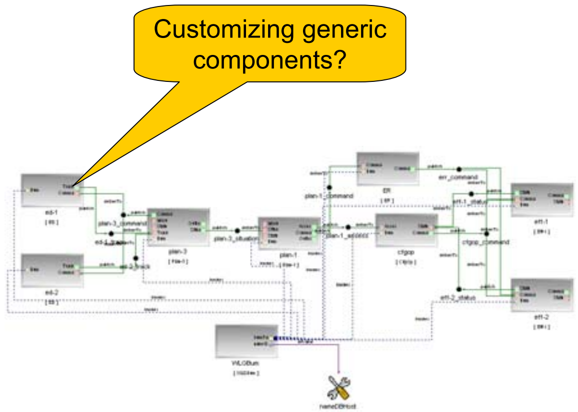



1. Evaluate QoS characteristics of DRE systems
2. Emulate QoS characteristics of DRE systems
3. Non-intrusive benchmarking & evaluation
4. Simplifying component behavior specification

**PICML Model of SLICE Scenario**



## Summary of CUTS Challenges

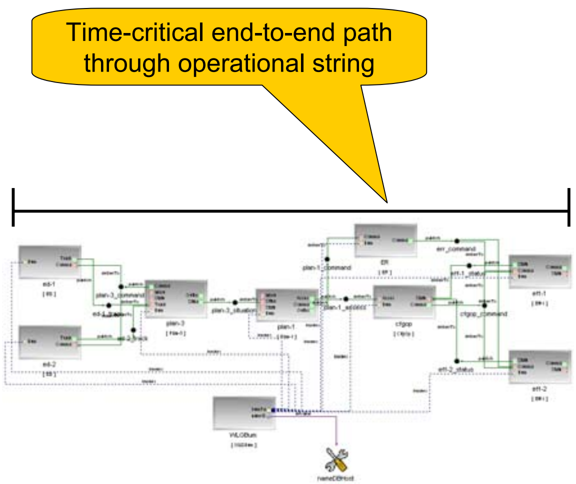



1. Evaluate QoS characteristics of DRE systems
2. Emulate QoS characteristics of DRE systems
3. Non-intrusive benchmarking & evaluation
4. Simplifying component behavior specification
5. Simplify component customization

**PICML Model of SLICE Scenario**



## Summary of CUTS Challenges

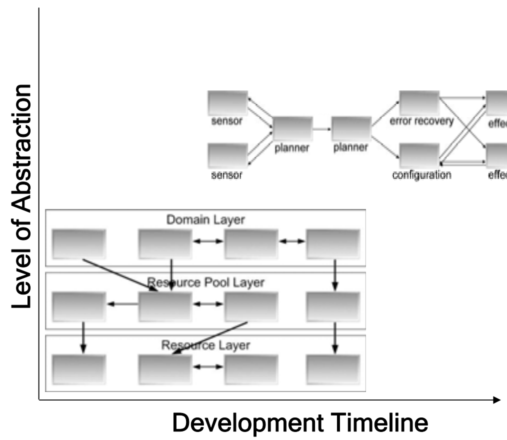



1. Evaluate QoS characteristics of DRE systems
2. Emulate QoS characteristics of DRE systems
3. Non-intrusive benchmarking & evaluation
4. Simplifying component behavior specification
5. Simplify component customization
6. Informative analysis of performance

**PICML Model of SLICE Scenario**



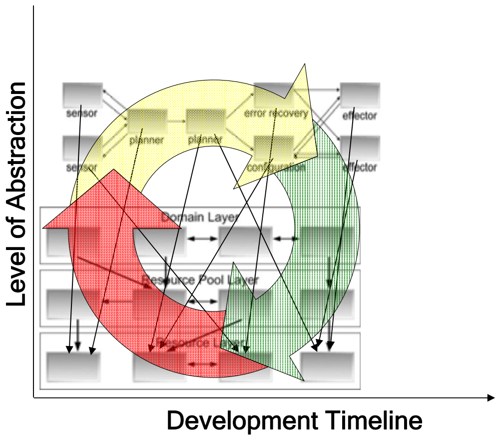
## Challenge 1: Evaluating QoS Characteristics of Enterprise DRE Systems Early in Life-cycle




**Context**

- In phase 1 of ARMS, QoS evaluation was not done until *application* integration
  - –Prolonged project development & QA
- In phase 2 of ARMS, MLRM is implemented using Real-time CCM (via CIAO & DAnCE)
- Software components & challenges are similar in both phases

# Challenge 1: Evaluating QoS Characteristics of Enterprise DRE Systems Early in Life-cycle




**Context**

- In phase 1 of ARMS, QoS evaluation was not done until *application* integration
  - Prolonged project development & QA
- In phase 2 of ARMS, MLRM is implemented using Real-time CCM (via CIAO & DAnCE)
- Software components & challenges are similar in both phases

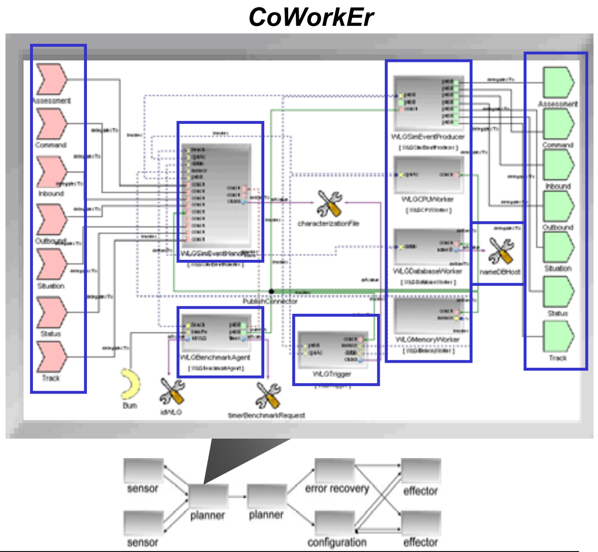
**Problem**

- How to evaluate MLRM QoS earlier in lifecycle?
  - i.e., *prior* to integration



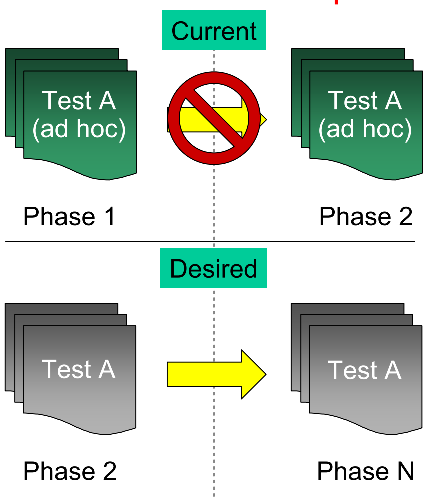
# Solution: Evaluate Component QoS & Behavior using Component–based Emulators

- System components are represented as *Component Workload Emulators* (*CoWorkErs*)
- Each *CoWorkEr* is a CCM *assembly* component constructed from CCM monolithic components
- Each *CoWorkEr* has an optional database
  - Can be local or remote
- *CoWorkErs* can be interconnected to form *operational strings*
  - Basically a "work flow" abstraction






# Challenge 2: Emulating Behavior & QoS of Enterprise DRE Systems




**Context**

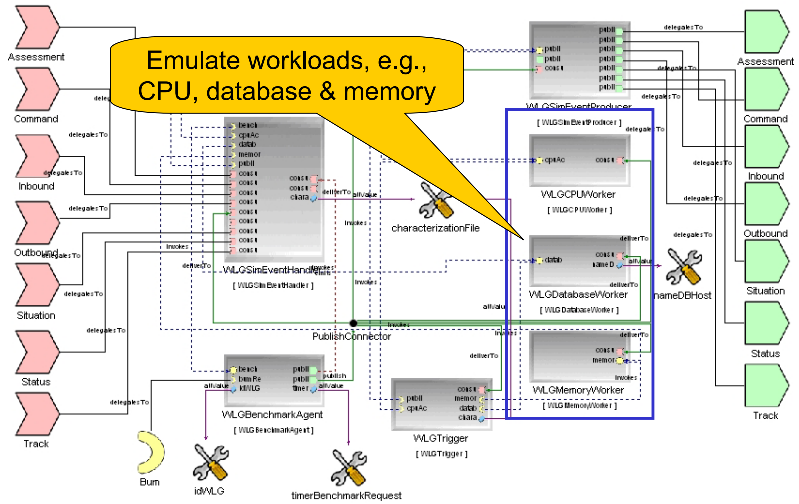
- In phase 1 of ARMS, QoS evaluation was not done until integration
- QoS testing was done using *ad hoc* techniques
  - e.g., creating non-reusable artifacts & tests that do not fully exercise the infrastructure

**Problem**

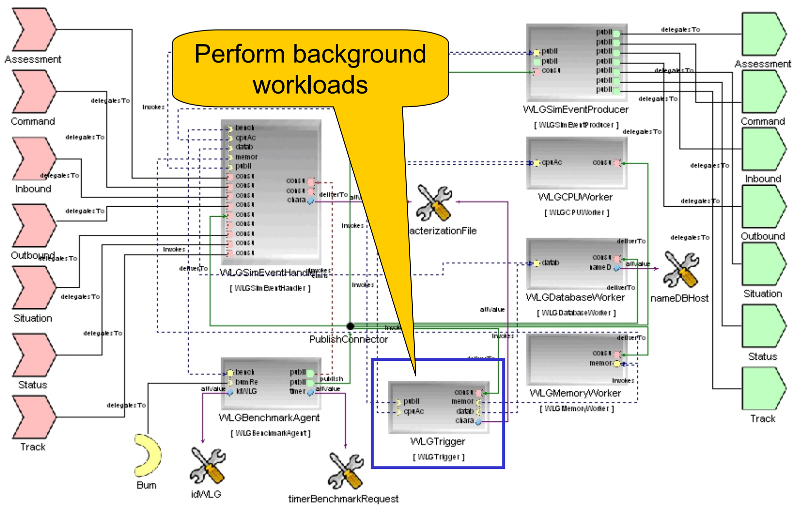
- How to emulate behavior & QoS in a reusable manner to evaluate the complete infrastructure & apply tests in different contexts



# Solution: Emulate Component Behavior & QoS Using Configurable CoWorkErs

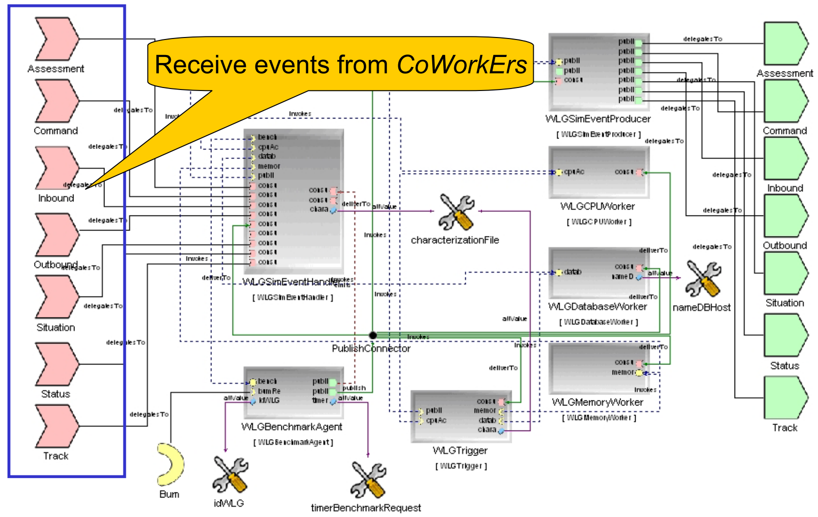


## Solution: Emulate Component Behavior & QoS Using Configurable CoWorkErs



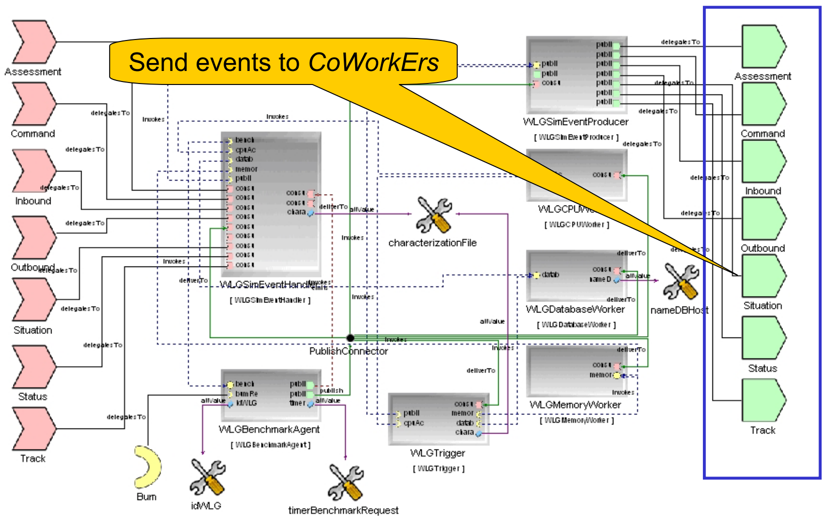



## Solution: Emulate Component Behavior & QoS Using Configurable CoWorkErs



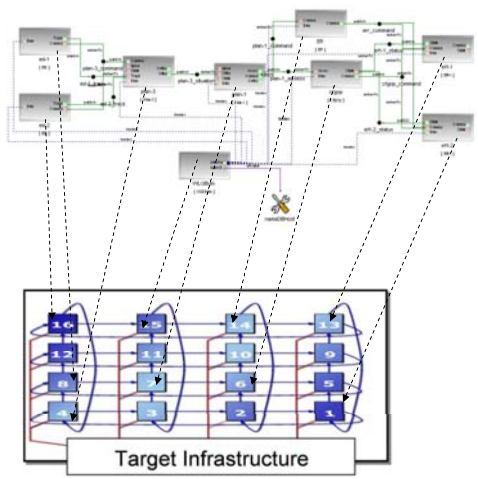



## Solution: Emulate Component Behavior & QoS Using Configurable CoWorkErs





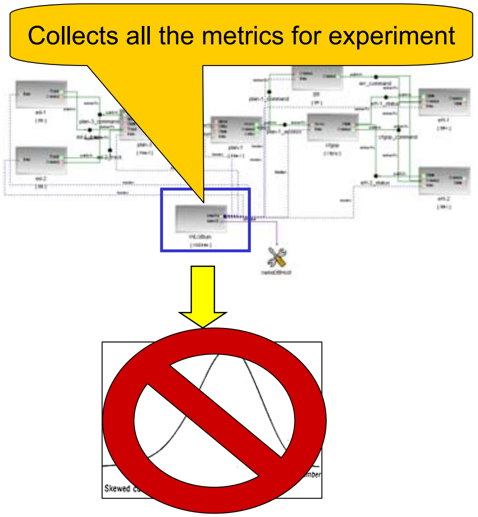

## Challenge 3: Non-Intrusive Benchmarking & Evaluation




**Context**

- The SLICE scenario of MLRM is composed of multiple components deployed over multiple nodes
- Each component, including components in assemblies, must be monitored & evaluated

## Challenge 3: Non-Intrusive Benchmarking & Evaluation




**Context**

- The SLICE scenario of MLRM is composed of multiple components deployed over multiple nodes
- Each component, including components in assemblies, must be monitored & evaluated

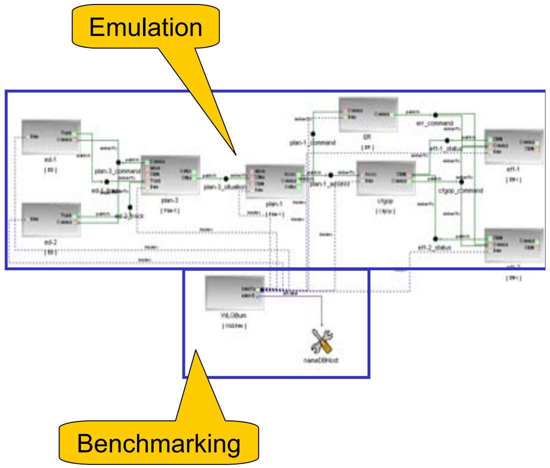
**Problem**

- Collecting data from each component without interfering with emulation
- Collecting data without unduly perturbing operational performance measures



## Solution: Decouple Emulation & Benchmarking

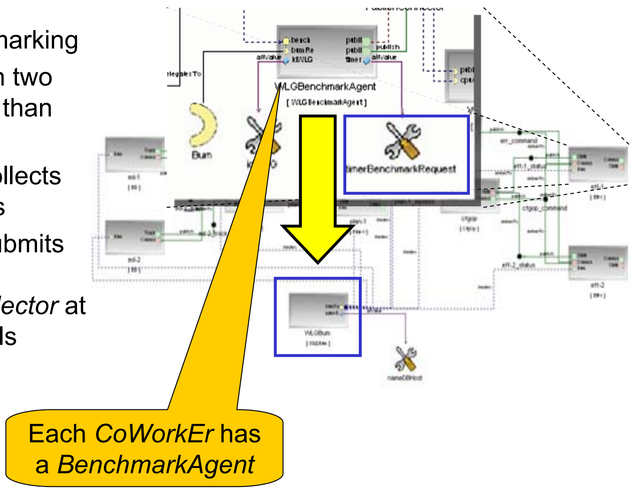
- CUTS environment is decoupled into two sections
  - Emulation & benchmarking






## Solution: Decouple Emulation & Benchmarking

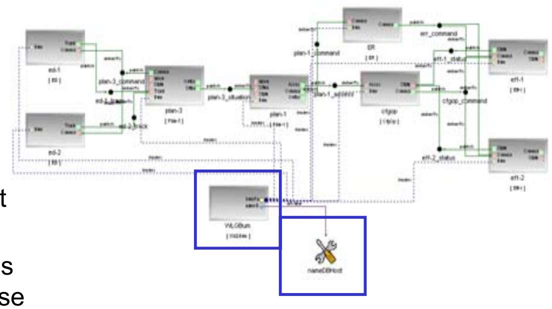
- CUTS environment is decoupled into two sections
  - Emulation & benchmarking
- Data acquisition done in two phases at lower priority than emulation
  1. *BenchmarkAgent* collects performance metrics
  2. *BenchmarkAgent* submits data to *BenchmarkDataCollector* at user-defined intervals



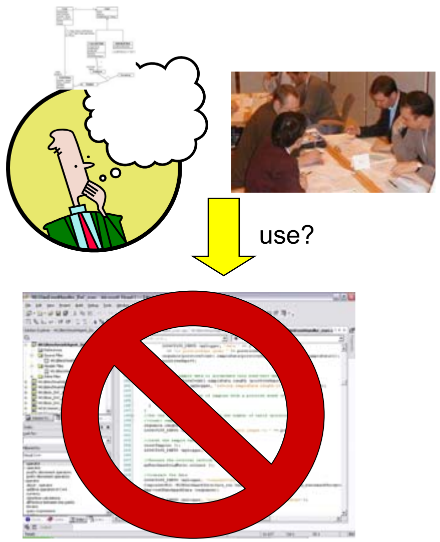



## Solution: Decouple Emulation & Benchmarking

- CUTS environment is decoupled into two sections
  - Emulation & benchmarking
- Data acquisition done in two phases at lower priority than emulation
  1. *BenchmarkAgent* collects performance metrics
  2. *BenchmarkAgent* submits data to *BenchmarkDataCollector* at user-defined intervals
- *BenchmarkDataCollector* stores performance metrics in database for offline analysis
- Separate networks are used for *CoWorkEr* communication & data acquisition



# Challenge 4: Simplify Characterization of Workload




**Context**

- People developing & using the SLICE scenario with CUTS come from different disciplines
  - –e.g., software architects, software developers, & systems engineers
- Many CUTS users may not be familiar with 3rd generation or configuration languages
  - –e.g., C++ & Java or XML, respectively

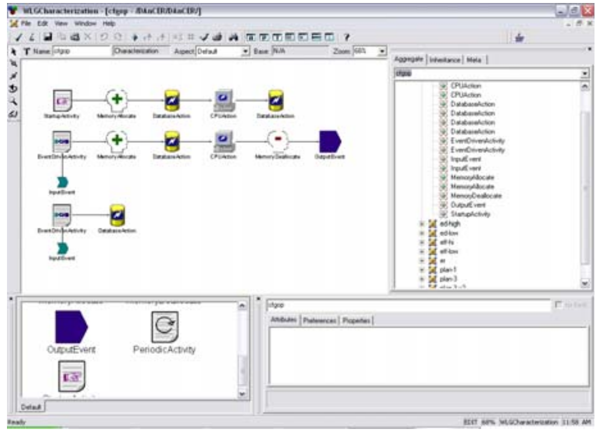
**Problem**

- Avoiding tedious & error-prone manual programming of *CoWorkEr* behavior using 3rd generation languages or configuration files

The harder it is to program CoWorkErs, the less useful CUTS emulation is…

# Solution: Use Domain–Specific Modeling Language to Program CoWorkEr Behavior

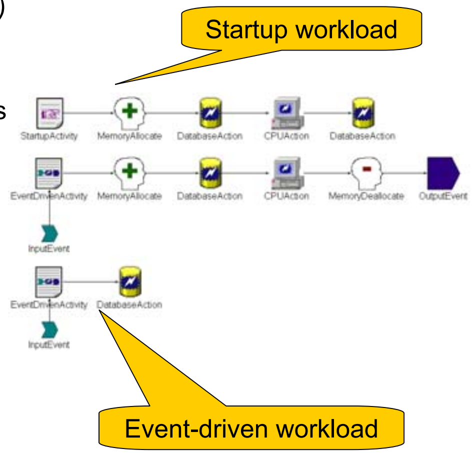
- *Workload Modeling Language (WML)* is used to define the behavior of *CoWorkEr* components





# Solution: Use Domain–Specific Modeling Language to Program CoWorkEr Behavior

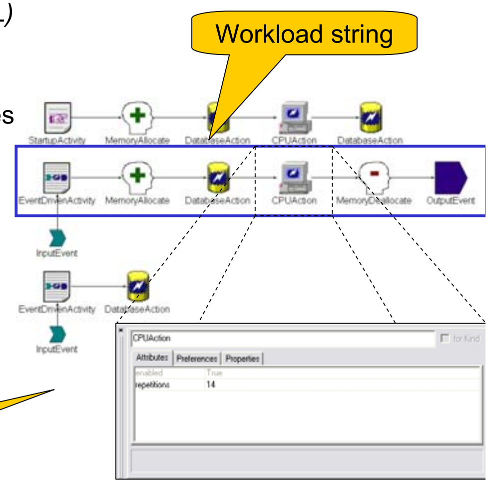
- *Workload Modeling Language (WML)* is used to define the behavior of *CoWorkEr* components
- WML events represent different types of workloads in *CoWorkEr*

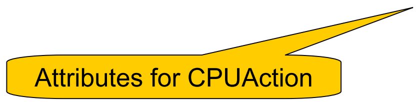





# Solution: Use Domain–Specific Modeling Language to Program CoWorkEr Behavior

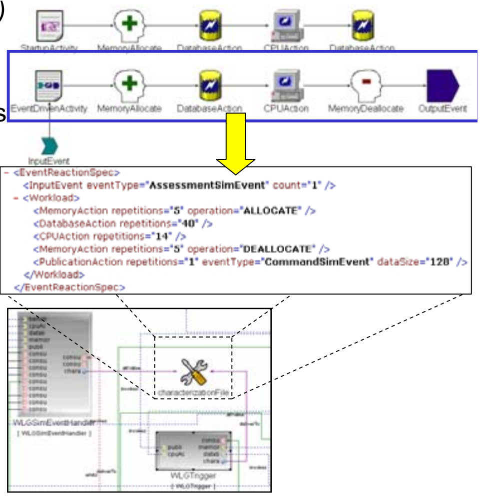
- *Workload Modeling Language (WML)* is used to define the behavior of *CoWorkEr* components
- WML events represent different types of workloads in *CoWorkEr*
- Actions can be attached to events & specified in order of execution to define “work sequences”
  - –Each action has attributes, e.g., number of repetitions, amount of memory to allocate & etc



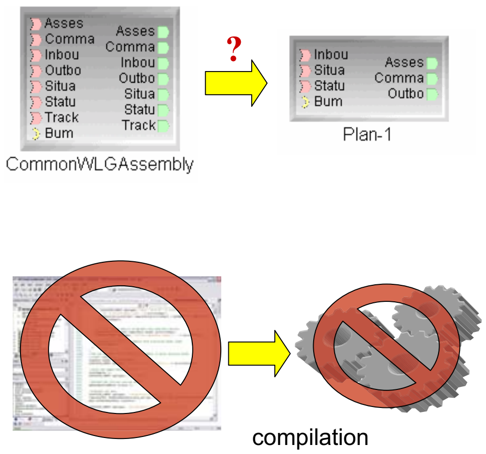
# Solution: Use Domain–Specific Modeling Language to Program CoWorkEr Behavior

- *Workload Modeling Language (WML)* is used to define the behavior of *CoWorkEr* components
- WML events represent different types of workloads in *CoWorkEr*
- Actions can be attached to events & specified in order of execution to define “work sequences”
  - –Each action has attributes, e.g., number of repetitions, amount of memory to allocate & etc
- WML programs are translated into XML characterization files
- Characterization specified in *CoWorkEr* & used to configure its behavior






# Challenge 5: Simplify Component Customization




**Context**

- By default a *CoWorkEr* can send & receive every type of event
- The SLICE components are all different, however, & do not send/receive the same types of events
  - –i.e., each contains a different composition pertaining to its specific workload(s)

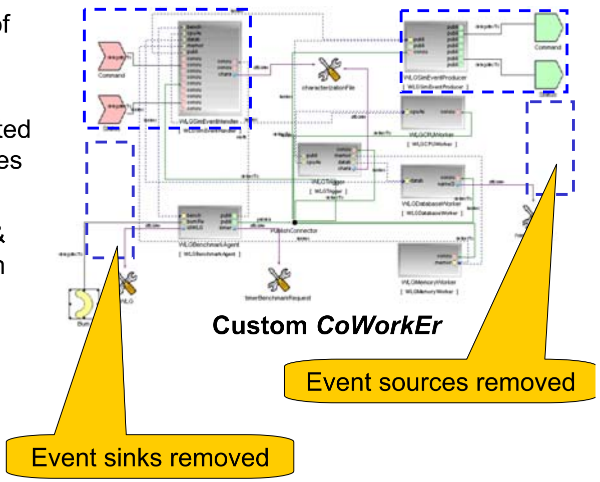
**Problem**

- How can we customize *CoWorkEr* components to enforce strong type-checking without requiring time-consuming modification & recompilation of components?



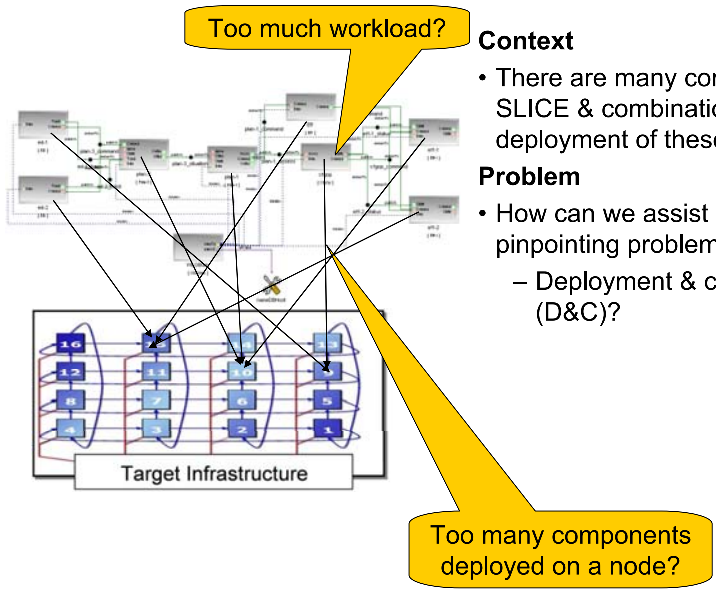
# Solution: Customize CoWorkErs at System Modeling Level

- Event sinks of a *CoWorkEr* are delegated to the respective event sources of the *EventHandler*
- Events produced by the *EventProducer* are delegated to respective events sources for a *CoWorkEr*
- Delegated event sources & sinks can be removed from *CoWorkEr*
  - –Does not require recompilation of components




This technique leverages key properties of CCM assemblies, i.e., virtual APIs

# Challenge 6: Informative Analysis of QoS Performance



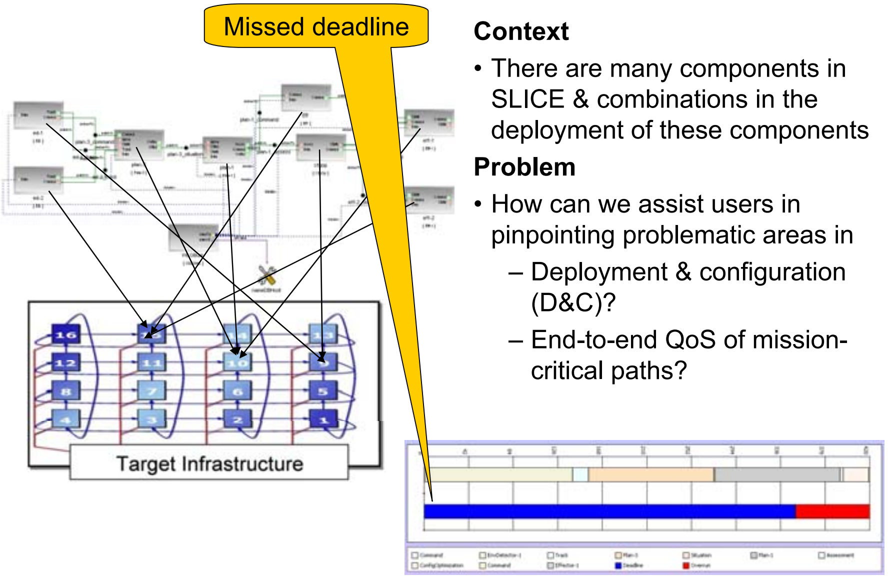

**Context**

- There are many components in SLICE & combinations in the deployment of these components

**Problem**

- How can we assist users in pinpointing problematic areas in
  - – Deployment & configuration (D&C)?

## Challenge 6: Informative Analysis of QoS Performance




**Context**

- There are many components in SLICE & combinations in the deployment of these components

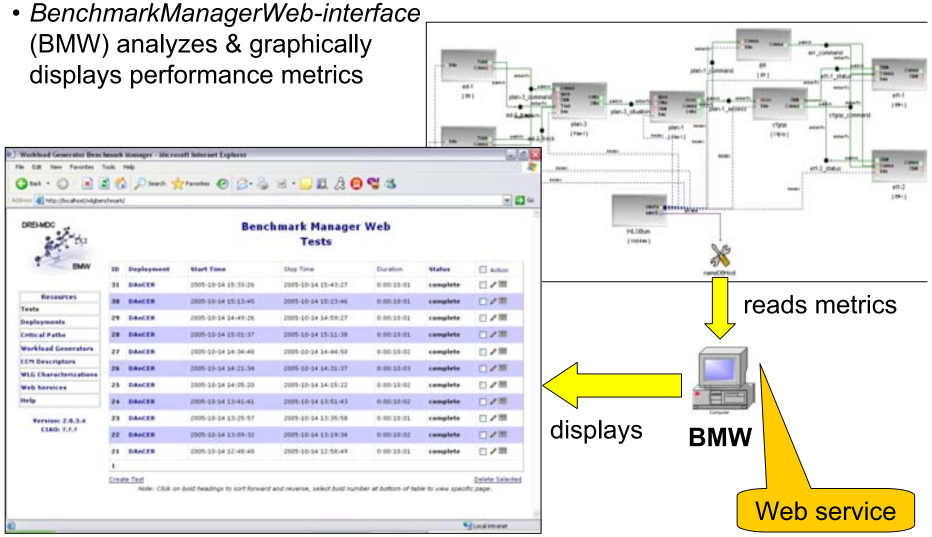
**Problem**

- How can we assist users in pinpointing problematic areas in
  - Deployment & configuration (D&C)?
  - End-to-end QoS of mission-critical paths?



## Solution: Present Metrics Graphically in Layers to Support General & Detailed Information

- *BenchmarkManagerWeb-interface* (BMW) analyzes & graphically displays performance metrics






## Solution: Present Metrics Graphically in Layers to Support General & Detailed Information

- *BenchmarkManagerWeb-interface* (BMW) analyzes & graphically displays performance metrics
- General analysis shows users overall performance of each *CoWorkEr*
  - e.g., transmisssion delay & processing

General analysis of actions

Host

CoWorkEr

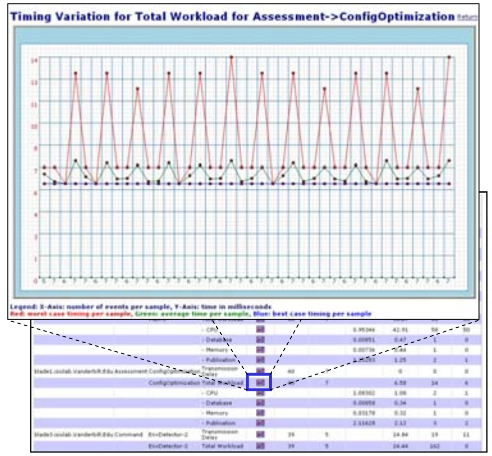
**BMW General Time Data**

| Host | Event -> | WLG | Workload | Timeline | Snapshots | Avg Samples* | Avg/Rep (ms)* | Average (ms) | Worst Case (ms) | Best Case (ms) |
|---|---|---|---|---|---|---|---|---|---|---|
| blade1.isislab.Vanderbilt.Edu | Status | ConfigOptimization | Transmission Delay | | 40 | 10 | | 0 | 0 | 0 |
| | | ConfigOptimization | Total Workload | | 40 | 10 | | 1.6 | 3 | 1 |
| | | | - Database | | | | 0.40365 | 0.4 | 1 | 0 |
| blade1.isislab.Vanderbilt.Edu | Command | EnvDetector-1 | Transmission Delay | | 40 | 5 | | 1.9 | 4 | 1 |
| | | EnvDetector-1 | Total Workload | | 40 | 5 | | 54.79 | 58 | 53 |
| | | | - CPU | | | | 0.81753 | 36.79 | 34 | 16 |
| | | | - Memory | | | | 0.00589 | 0.41 | 1 | 0 |
| | | | - Publication | | | | 0.88442 | 0.88 | 2 | 0 |
| blade1.isislab.Vanderbilt.Edu | Track | Plan-3 | Transmission Delay | | 40 | 7 | | 11.32 | 57 | 0 |
| | | Plan-3 | Total Workload | | 40 | 7 | | 55.27 | 63 | 53 |
| | | | - CPU | | | | 0.95346 | 42.91 | 58 | 50 |
| | | | - Database | | | | 0.00851 | 0.47 | 1 | 0 |
| | | | - Memory | | | | 0.00736 | 0.44 | 1 | 0 |
| | | | - Publication | | | | 1.25283 | 1.25 | 2 | 1 |
| blade1.isislab.Vanderbilt.Edu | Assessment | ConfigOptimization | Transmission Delay | | 40 | 7 | | 0 | 0 | 0 |
| | | ConfigOptimization | Total Workload | | 40 | 7 | | 6.58 | 14 | 6 |
| | | | - CPU | | | | 1.08302 | 1.08 | 2 | 1 |
| | | | - Database | | | | 0.00858 | 0.34 | 1 | 0 |
| | | | - Memory | | | | 0.03178 | 0.32 | 1 | 0 |
| | | | - Publication | | | | 2.11628 | 2.12 | 3 | 2 |
| blade3.isislab.Vanderbilt.Edu | Command | EnvDetector-2 | Transmission Delay | | 39 | 5 | | 14.84 | 19 | 11 |
| | | EnvDetector-2 | Total Workload | | 39 | 5 | | 24.44 | 162 | 0 |



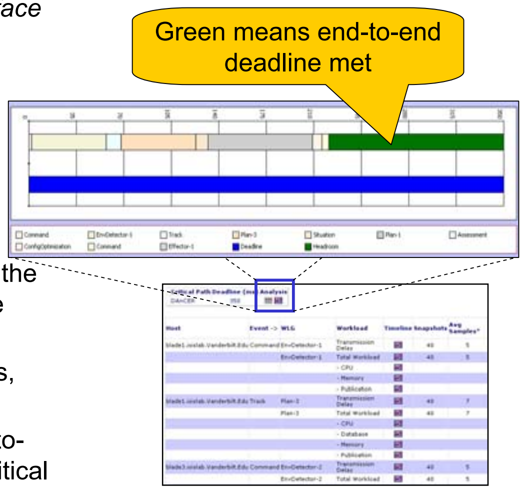
## Solution: Present Metrics Graphically in Layers to Support General & Detailed Information

- *BenchmarkManagerWeb-interface* (BMW) analyzes & graphically displays performance metrics
- General analysis shows users overall performance of each *CoWorkEr*
  - e.g., transmisssion delay & processing
- Detailed analysis shows users the performance of an action in the respective *CoWorkEr*
  - e.g., memory & CPU actions, event handling & etc



# Solution: Present Metrics Graphically in Layers to Support General & Detailed Information

- *BenchmarkManagerWeb-interface* (BMW) analyzes & graphically displays performance metrics
- General analysis shows users overall performance of each *CoWorkEr*
  - e.g., transmisssion delay & processing
- Detailed analysis shows users the performance of an action in the respective *CoWorkEr*
  - e.g., memory & CPU actions, event handling & etc
- Critical paths show users end-to-end performance of mission-critical operational strings

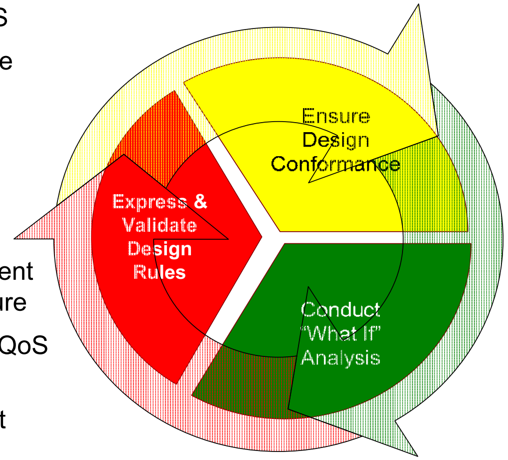





# Applying CUTS to the SLICE Scenario

Using ISISLab as our target infrastructure in conjunction with CUTS

1. Use *PICML* to define & validate infrastructure specifications & requirements
2. Use *WML* to define & validate application specifications & requirements
3. Use *DAnCE* to deploy component emulators on target infrastructure
4. Use *BMW* to evaluate & verify QoS performance
5. Redefine system D&C & repeat

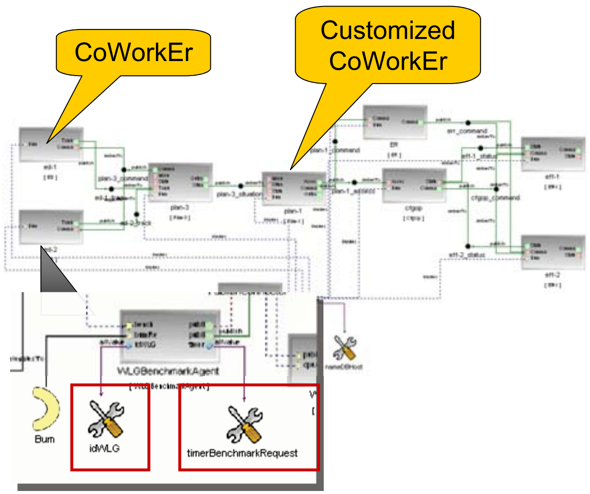



www.dre.vanderbilt.edu/ISISlab/



# Defining Components of SLICE Scenario in PICML for CUTS

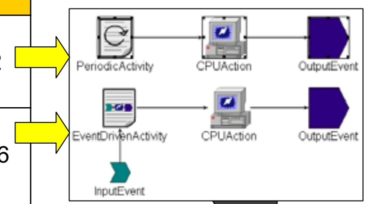
- Each component in SLICE is defined as a *CoWorkEr*
- The default *CoWorkEr* is customized to handle events specific to its representative SLICE component
- Each *CoWorkEr* is assigned a unique user-defined ID number
- The benchmark data submission rate is set to 15 seconds

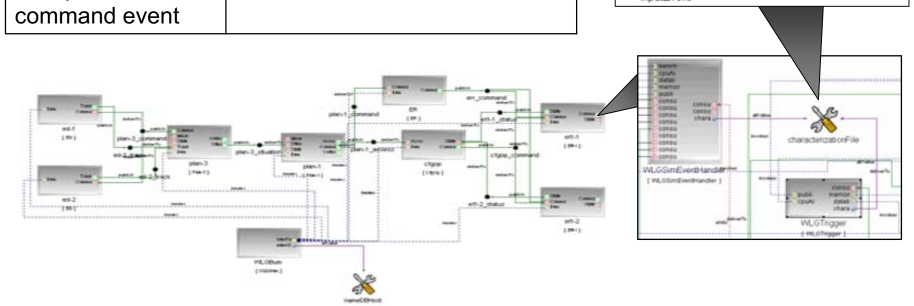




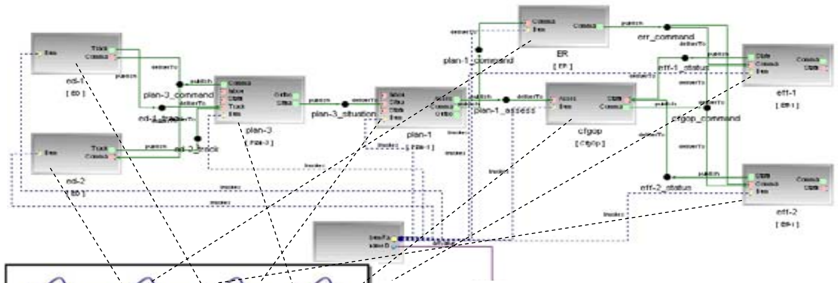

# Defining Behavior of SLICE Scenario Components using WML

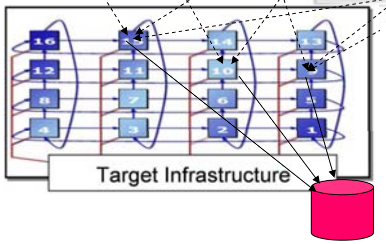
| Effector 1 & Effector 2 | |
|---|---|
| Workload performed every second | CPU: 25 reps<br>PUBLISH: STATUS – SIZE 32 |
| Workload performed after receipt of command event | CPU: 25 reps<br>PUBLISH: STATUS - SIZE 256 |



# Recap of Questions We Wanted to Answer





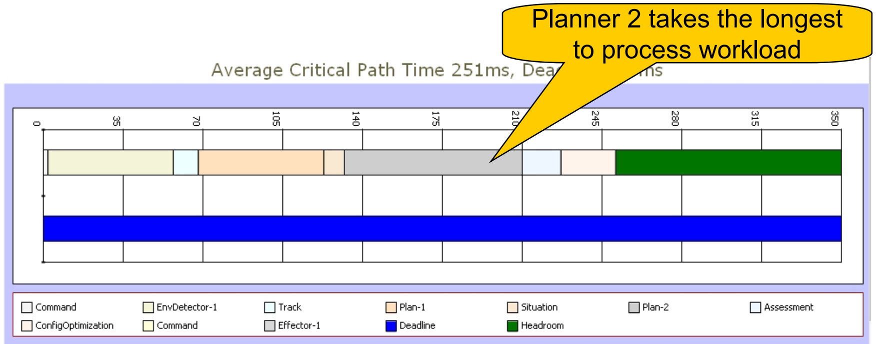

1. Can we meet the D&C & performance requirements?
2. Are there multiple deployments that meet the 350ms critical path deadline?
   - e.g., which yields most headroom?
3. Can we meet D&C & performance requirements using a single host?

To answer these questions we ran 11 tests using different CoWorkEr D&C's



# SLICE Scenario Results: Meeting D&C & QoS Requirements

| Deployment Table | | | |
|---|---|---|---|
| **Test** | **Node 1** | **Node 2** | **Node 3** |
| 9 | sensor 1 & planner 1 | planner 2, configuration, & effector 1 | sensor 2, error recovery & effector 2 |



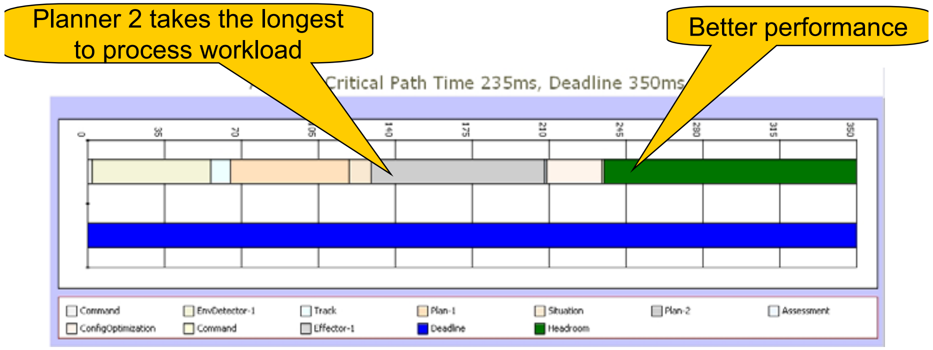

Critical Path Timing Information for Test 9

"What if" planner-2 is put on Node 3, which has no critical path components?



# SLICE Scenario Results: Meeting D&C & QoS Requirements

| Deployment Table | | | |
|---|---|---|---|
| **Test** | **Node 1** | **Node 2** | **Node 3** |
| 10 | sensor 1 & planner 1 | configuration & effector 1 | planner 2, sensor 2, error recovery & effector 2 |



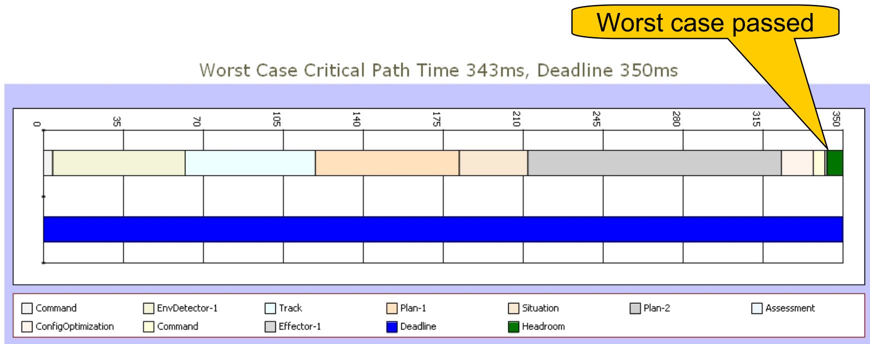

Critical path timing information for Test 10

Node 3 is "saturated" with non-critical path components, "what if" we evenly distribute *critical path* workload on collocated components?



# SLICE Scenario Results: Meeting D&C & QoS Requirements

| Deployment Table | | | |
|---|---|---|---|
| **Test** | **Node 1** | **Node 2** | **Node 3** |
| 11 | sensor 1, planner 1 & configuration | planner 2 & effector 1 | sensor 2, error recovery & effector 2 |



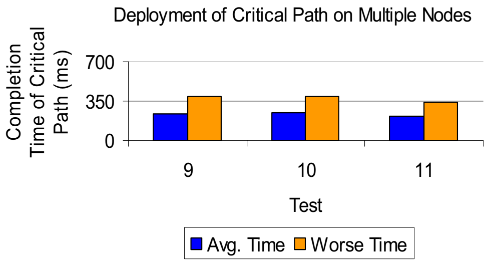

Critical path timing information for Test 11

We were able to answer the critical path & deployment questions

# SLICE Scenario Results: Meeting D&C & QoS Requirements

| Deployment Table | | | |
|---|---|---|---|
| Test | Node 1 | Node 2 | Node 3 |
| 9 | sensor 1 & planner 1 | planner 2, configuration, & effector 1 | Sensor 2, error re recovery & effector 2 |
| 10 | sensor 1 & planner 1 | configuration & effector 1 | planner 2, sensor 2, error recovery & effector 2 |
| 11 | sensor 1, planner 1 & configuration | planner 2 & effector 1 | sensor 2, error recovery & effector 2 |

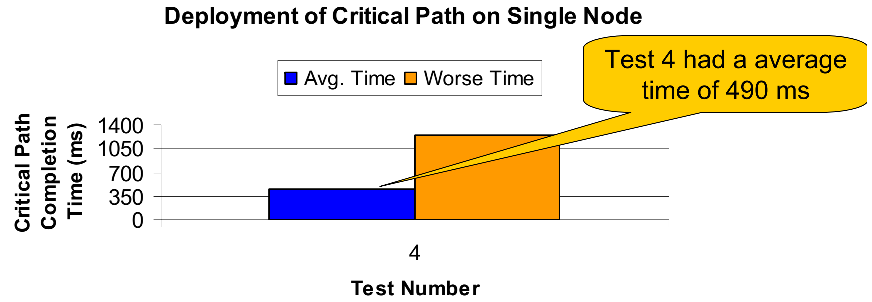
- Test 9, 10 & 11 meet the performance requirements for the average execution time of the critical path
- Test 11 meet the performance requirements for worst execution time
- We did not exhaustively test all D&C's, but that could be done also






# SLICE Scenario Results: Single Host Deployment

| Deployment Table | | | |
|---|---|---|---|
| Test | Node 1 | Node 2 | Node 3 |
| 4 | All components | (nothing) | (nothing) |




We were able to answer the question about deploying on a single node

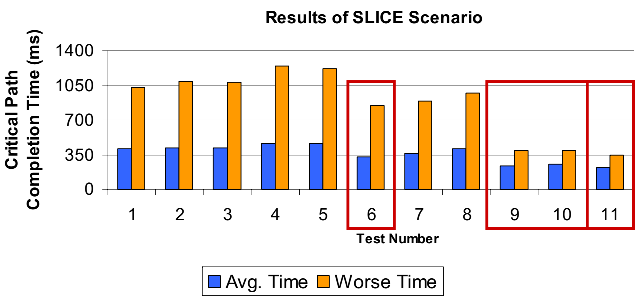


# Overall Results of SLICE Scenario


Population size of 11 tests

Test 11 produced the best results
- Average case: 221 ms
- Worse case: 343 ms

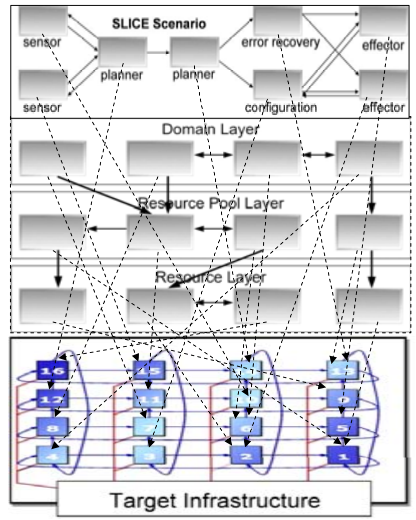
- Only 4 of 11 deployments met the 350 ms critical path deadline for average-case time
- Test 11 only test to meet critical path deadline for worst-case time





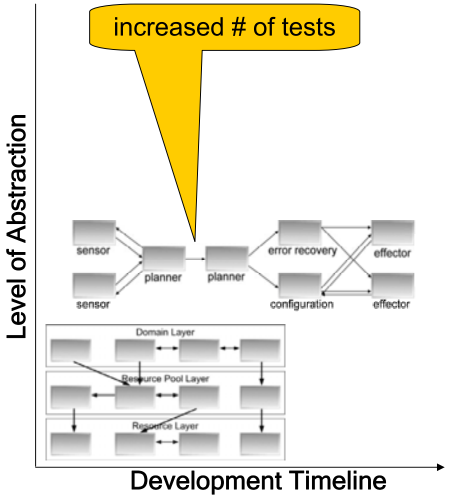

# Lessons Learned

- SOA middleware technologies allowed us to leverage the behavior & functionality of target architecture for realistic emulations
- SOA technologies allowed us to focus on the "business" logic of *CoWorkErs*
  - –e.g., D&C handled by underlying MDD & middleware technology



# Lessons Learned

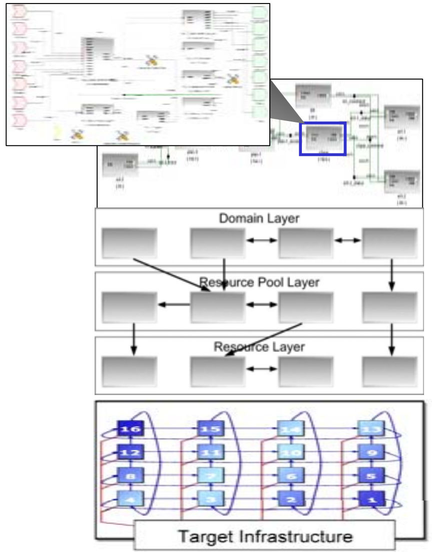
- SOA middleware technologies allowed us to leverage the behavior & functionality of target architecture for realistic emulations
- SOA technologies allowed us to focus on the “business” logic of *CoWorkErs*
  - –e.g., D&C handled by underlying MDD & middleware technology
- CUTS allowed us to test deployments *before* full system integration testing
- CUTS allowed us to rapidly test deployments that would have take *much* longer using *ad hoc* techniques
  - –e.g., hand-coding the D&C of components






# Summary

- We motivated the need for the *Component Workload Emulator (CoWorkEr) Utilization Test Suite (CUTS)*
- We presented a large-scale DRE system example that used CUTS to evaluate component D&C *before* complete integration
- We presented the design & implementation of CUTS, along with the design challenges we faced
- CUTS is being integrated into the open-source CoSMIC MDD toolchain
  - – www.dre.vanderbilt.edu/cosmic




www.cs.wustl.edu/~schmidt/PDF/CUTS.pdf
www.cs.wustl.edu/~schmidt/PDF/QoSPML-WML.pdf

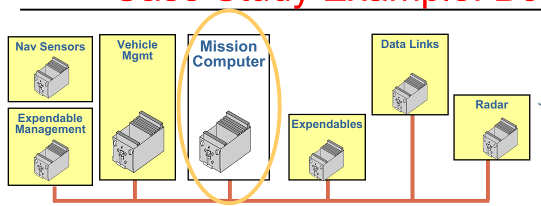
Model-Driven Development of Distributed Systems

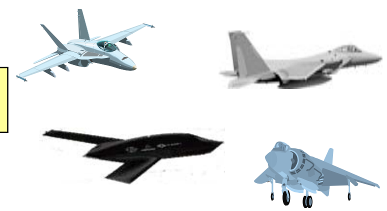
# CONTENTS





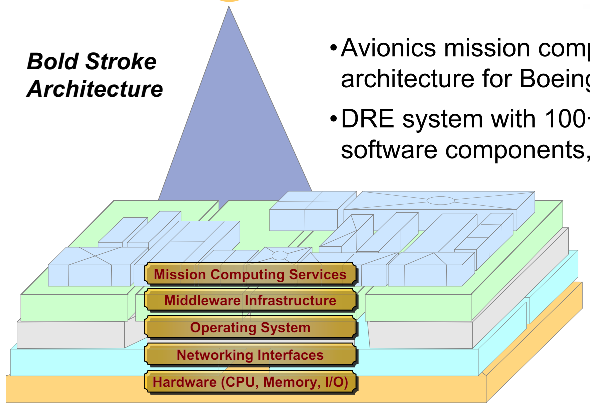
# Case Study Example: Boeing Bold Stroke



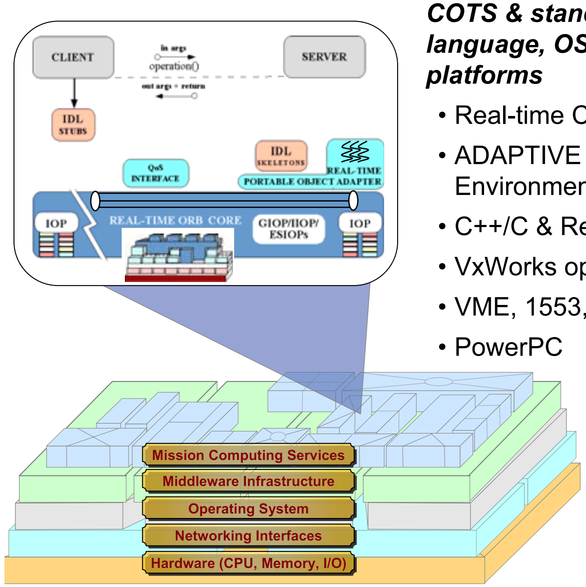
***Bold Stroke Architecture***

- Avionics mission computing product-line architecture for Boeing aircraft
- DRE system with 100+ developers, 3,000+ software components, 3-5 million lines of C++




- Based on COTS hardware, networks, operating systems, languages, & middleware

## Applying COTS to Boeing Bold Stroke



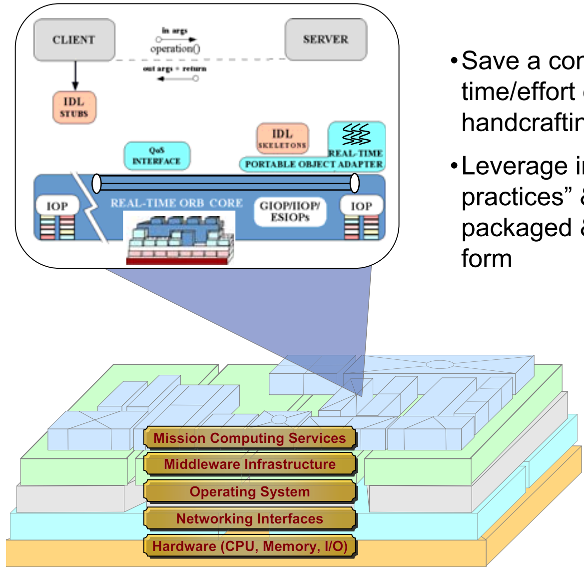

***COTS & standards-based middleware, language, OS, network, & hardware platforms***

- Real-time CORBA middleware services
- ADAPTIVE Communication Environment (ACE)
- C++/C & Real-time Java
- VxWorks operating system
- VME, 1553, & Link16
- PowerPC

www.cs.wustl.edu/~schmidt/TAO.html



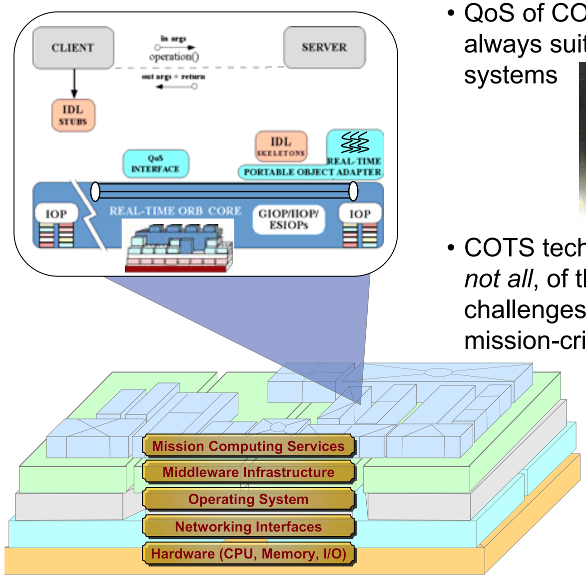
## Benefits of Using COTS



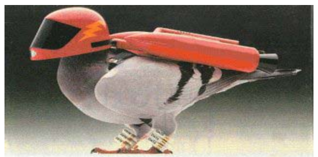

- Save a considerable amount of time/effort compared with handcrafting capabilities
- Leverage industry "best practices" & patterns in pre-packaged & ideally standardized form



## Limitations of Using COTS




- QoS of COTS components is not always suitable for mission-critical systems

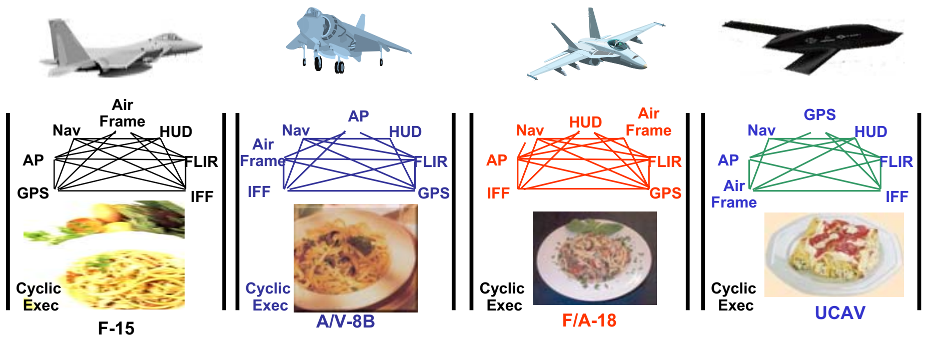


- COTS technologies address some, *but not all*, of the ***domain-specific*** challenges associated with developing mission-critical DRE systems

What we need is a reuse technology for organizing & automating key roles & responsibilities in an application domain



## Motivation for Product-line Architectures (PLAs)

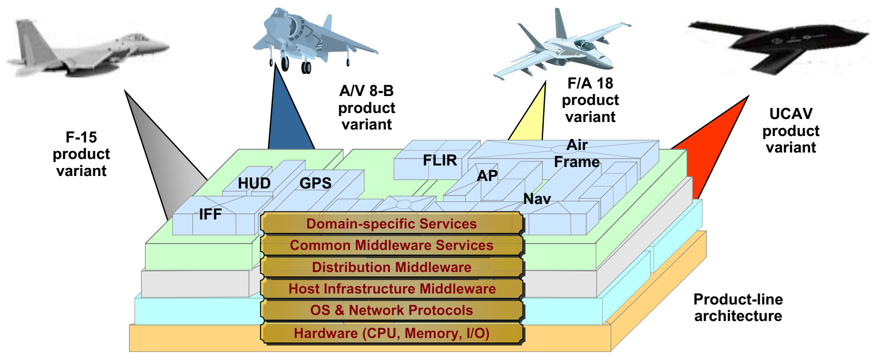



Legacy DRE systems have historically been:

- Stovepiped
- Proprietary
- Brittle & non-adaptive
- Expensive
- Vulnerable

*Consequence: Small HW/SW changes have big (negative) impact on DRE system QoS & maintenance*

# Motivation for Product-line Architectures (PLAs)

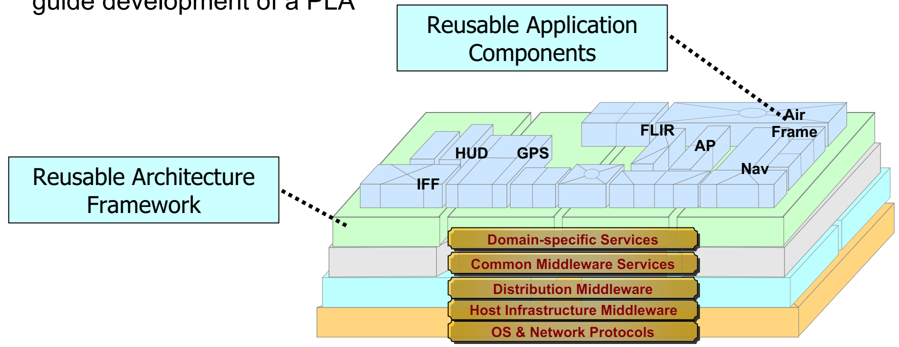



- ***Frameworks*** factors out many reusable general-purpose & domain-specific services from traditional DRE application responsibility
- Essential for ***product-line architectures (PLAs)***
- Product-lines & frameworks offer many configuration opportunities
  - e.g., component distribution & deployment, user interfaces & operating systems, algorithms & data structures, etc



# Overview of Product-line Architectures (PLAs)

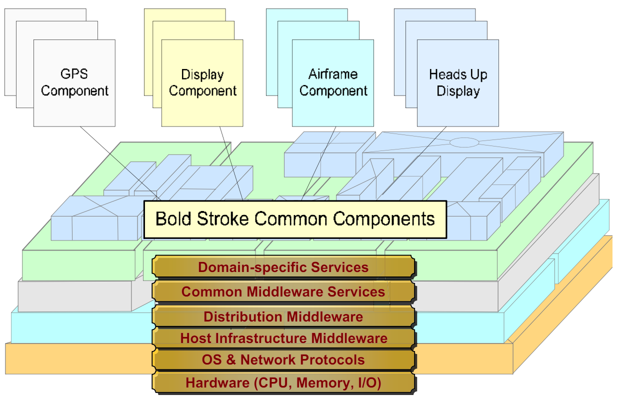
- **PLA characteristics are captured via *Scope, Commonalities, & Variabilities (SCV) analysis***
  - This process can be applied to identify commonalities & variabilities in a domain to guide development of a PLA
- **Applying SCV to Bold Stroke**
  - **Scope defines the domain & context of the PLA**
    - Bold Stroke component architecture, object-oriented application frameworks, & associated components, e.g., GPS, Airframe, & Display






# Applying SCV to the Bold Stroke PLA

- ***Commonalities*** describe the attributes that are common across all members of the PLA family
  - Common object-oriented frameworks & set of component types
    - e.g., GPS, Airframe, Navigation, & Display components
  - Common middleware infrastructure
    - e.g., Real-time CORBA & a variant of Lightweight CORBA Component Model (CCM) called Prism

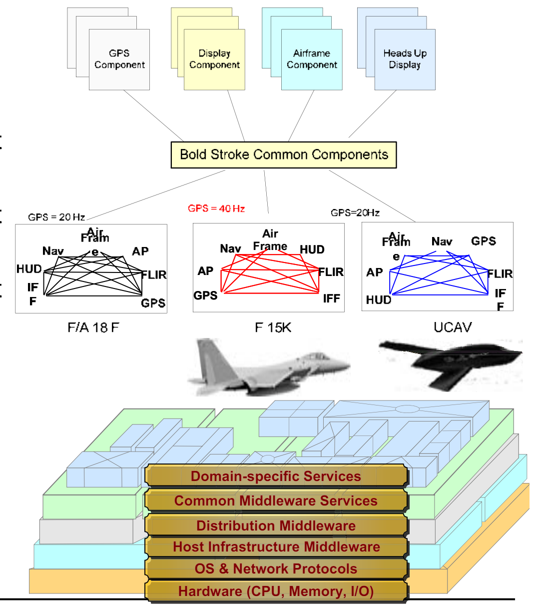




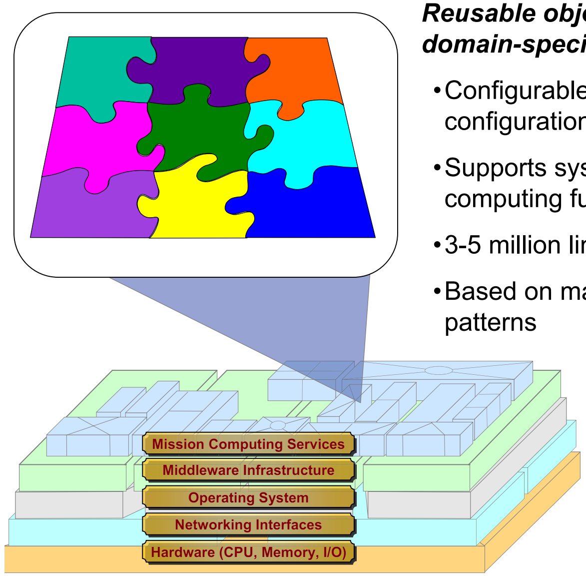

# Applying SCV to the Bold Stroke PLA

- ***Variabilities*** describe the attributes unique to the different members of the family
  - Product-dependent component implementations (GPS/INS)
  - Product-dependent component connections
  - Product-dependent component assemblies (e.g., different weapons systems for different customers/countries)
  - Different hardware, OS, & network/bus configurations

***Patterns*** & ***frameworks*** are essential for developing reusable PLAs



# Applying Patterns & Frameworks to Bold Stroke




***Reusable object-oriented application domain-specific middleware framework***

- Configurable to variable infrastructure configurations
- Supports systematic reuse of mission computing functionality
- 3-5 million lines of C++
- Based on many architecture & design patterns

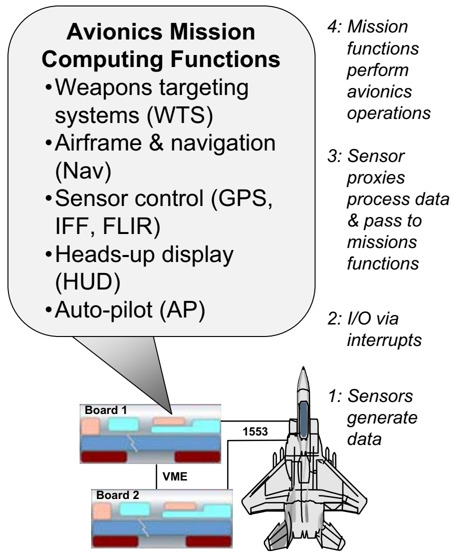
Patterns & frameworks are also used throughout COTS software infrastructure



# Legacy Avionics Architectures

**Key System Characteristics**

- Hard & soft real-time deadlines
  - ~20-40 Hz
- Low latency & jitter between boards
  - ~100 *u*secs
- Periodic & aperiodic processing
- Complex dependencies
- Continuous platform upgrades





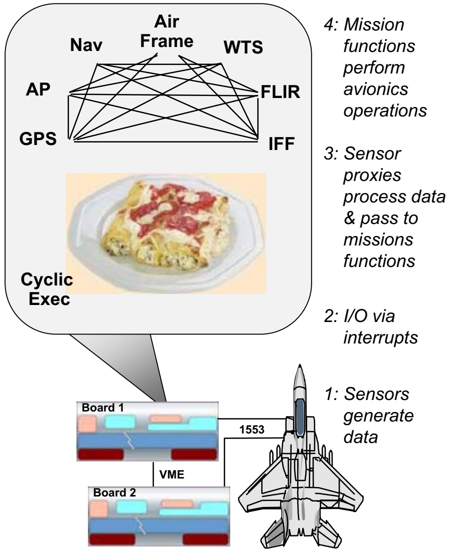

# Legacy Avionics Architectures

**Key System Characteristics**

- Hard & soft real-time deadlines
  - ~20-40 Hz
- Low latency & jitter between boards
  - ~100 *u*secs
- Periodic & aperiodic processing
- Complex dependencies
- Continuous platform upgrades

**Limitations with Legacy Avionics Architectures**

- Stovepiped
- Proprietary
- Expensive
- Vulnerable
- *Tightly coupled*
- *Hard to schedule*
- *Brittle & non-adaptive*

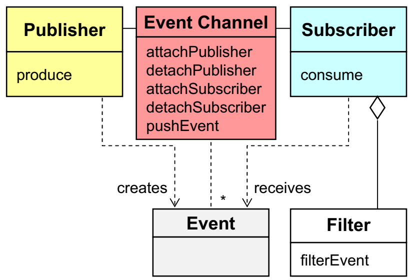




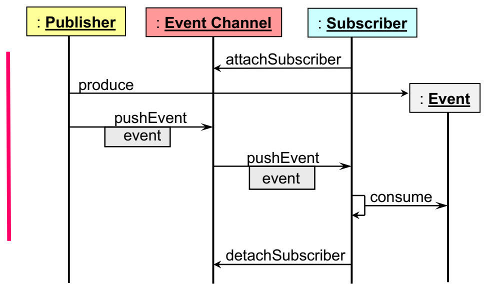

# Decoupling Avionics Components

| Context | Problems | Solution |
|---|---|---|
| • I/O driven DRE application<br>• Complex dependencies<br>• Real-time constraints | • Tightly coupled components<br>• Hard to schedule<br>• Expensive to evolve | • Apply the *Publisher-Subscriber* architectural pattern to distribute periodic, I/O-driven data from a single point of source to a collection of consumers |

**Structure**




**Dynamics**



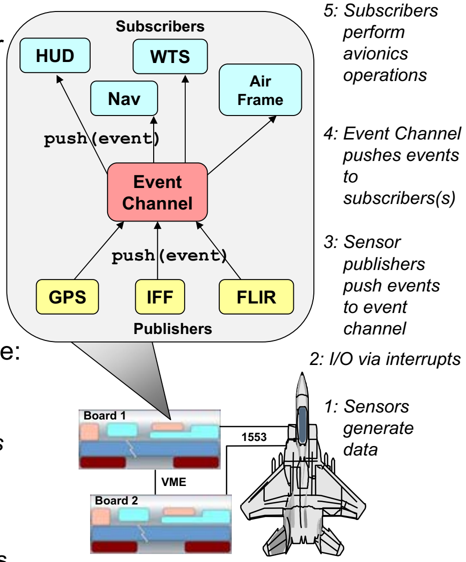
# Applying the Publisher-Subscriber Pattern to Bold Stroke

Bold Stroke uses the *Publisher-Subscriber* pattern to decouple sensor processing from mission computing operations

- Anonymous publisher & subscriber relationships
- Group communication
- Asynchrony

Considerations for implementing the *Publisher-Subscriber* pattern for mission computing applications include:

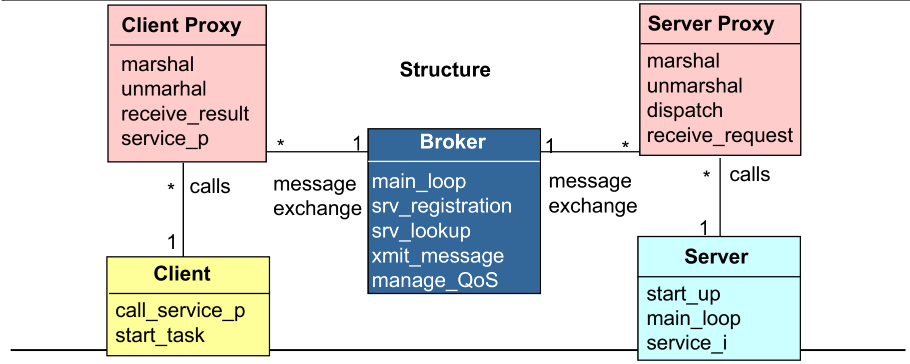
- *Event notification model*
  - Push control vs pull data interactions
- *Scheduling & synchronization strategies*
  - e.g., priority-based dispatching & preemption
- *Event dependency management*
  - e.g.,filtering & correlation mechanisms






# Ensuring Platform-neutral Inter-process Communication

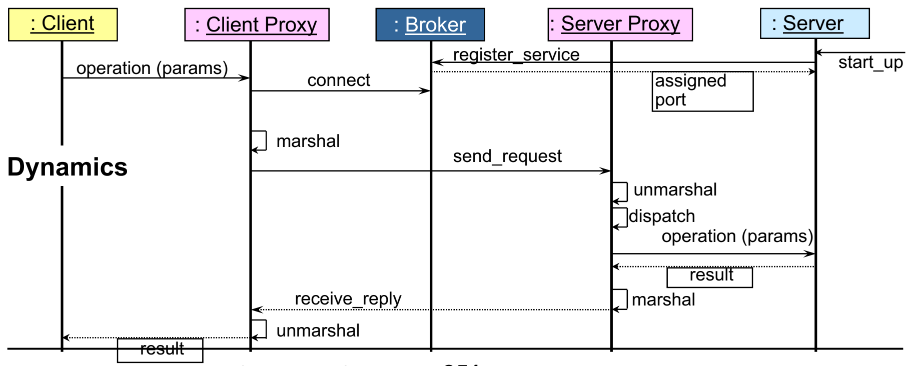
| Context | Problems | Solution |
|---|---|---|
| • Mission computing requires remote IPC<br>• Stringent DRE requirements | • Applications need capabilities to:<br>• Support remote communication<br>• Provide location transparency<br>• Handle faults<br>• Manage end-to-end QoS<br>• Encapsulate low-level system details | • Apply the *Broker* architectural pattern to provide platform-neutral comms between mission computing boards |






# Ensuring Platform-neutral Inter-process Communication

| Context | Problems | Solution |
|---|---|---|
| • Mission computing requires remote IPC<br>• Stringent DRE requirements | • Applications need capabilities to:<br>• Support remote communication<br>• Provide location transparency<br>• Handle faults<br>• Manage end-to-end QoS<br>• Encapsulate low-level system details | • Apply the *Broker* architectural pattern to provide platform-neutral comms between mission computing boards |





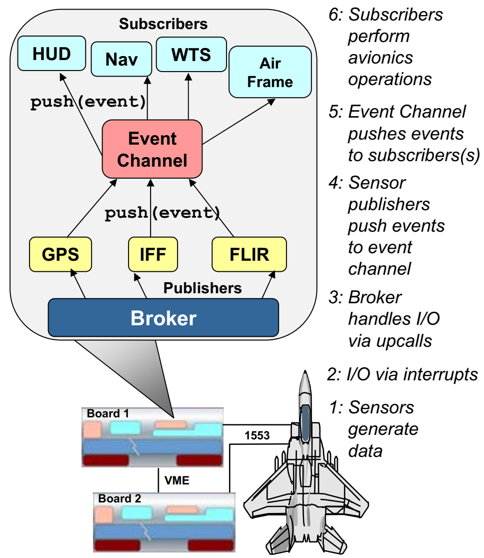

# Applying the Broker Pattern to Bold Stroke

Bold Stroke uses the *Broker* pattern to shield distributed applications from environment heterogeneity, *e.g.,*

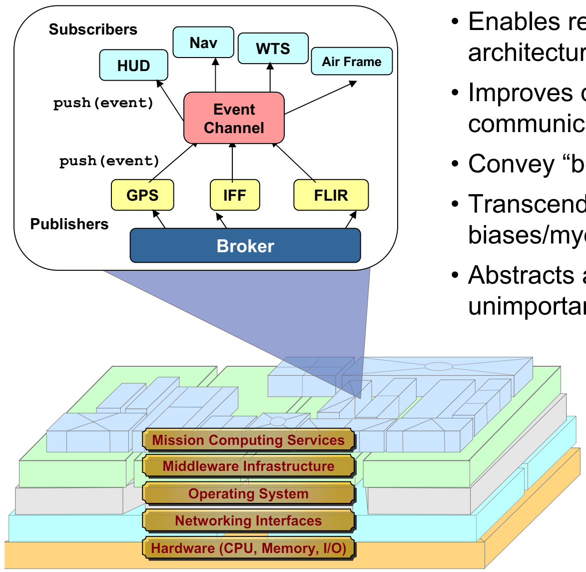
- Programming languages
- Operating systems
- Networking protocols
- Hardware

A key consideration for implementing the *Broker* pattern for mission computing applications is QoS support

- e.g., latency, jitter, priority preservation, dependability, security, etc



# Benefits of Patterns

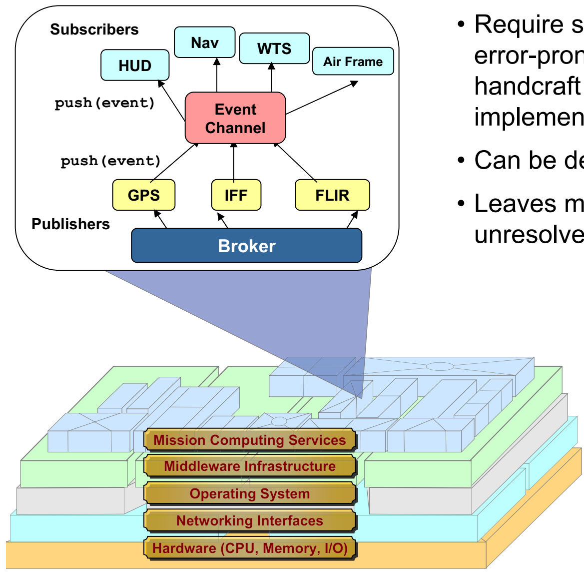



- Enables reuse of software architectures & designs
- Improves development team communication
- Convey "best practices" intuitively
- Transcends language-centric biases/myopia
- Abstracts away from many unimportant details

www.cs.wustl.edu/~schmidt/patterns.html



# Limitations of Patterns



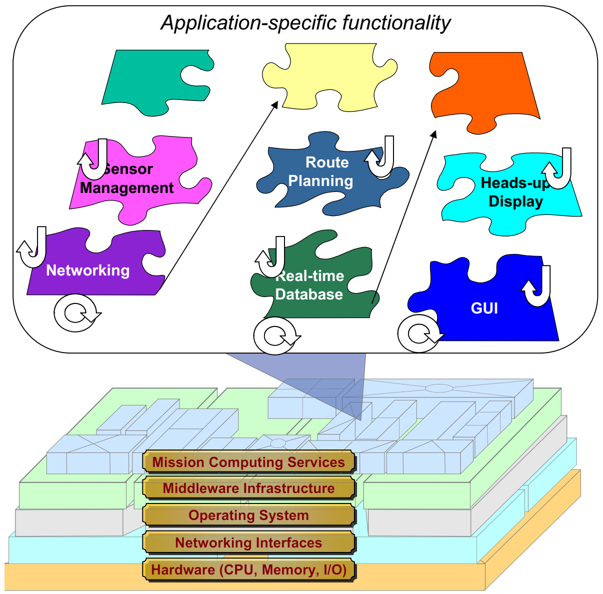

- Require significant tedious & error-prone human effort to handcraft pattern implementations
- Can be deceptively simple
- Leaves many important details unresolved

We therefore need more than just patterns to achieve systematic reuse

www.cs.wustl.edu/~schmidt/patterns.html



# Applying Frameworks to Bold Stroke



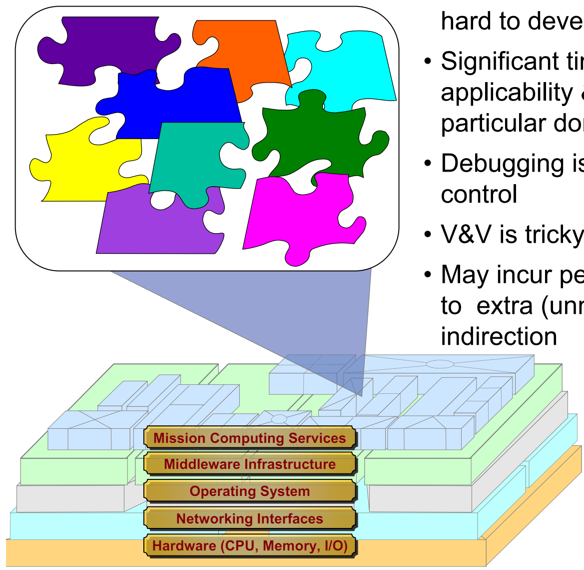

**Framework benefits & characteristics**

- Frameworks exhibit "inversion of control" at runtime via callbacks
- Frameworks provide integrated domain-specific structures & functionality
- Frameworks are "semi- complete" applications

www.cs.wustl.edu/~schmidt/ACE.html



# Limitations of Frameworks

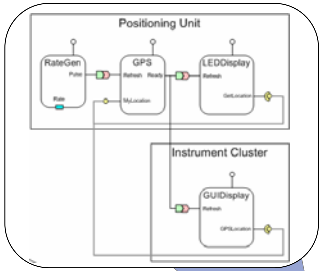



- Frameworks are powerful, but can be hard to develop & use effectively
- Significant time required to evaluate applicability & quality of a framework for a particular domain
- Debugging is tricky due to inversion of control
- V&V is tricky due to "late binding"
- May incur performance degradations due to extra (unnecessary) levels of indirection

We therefore need something simpler than frameworks to achieve systematic reuse

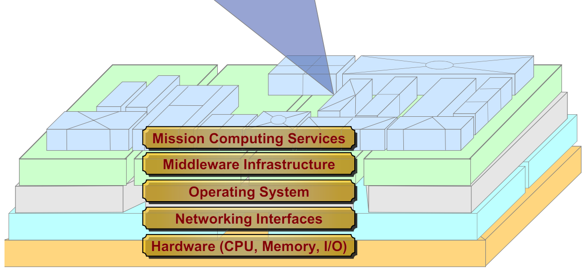
www.cs.wustl.edu/~schmidt/PDF/Queue-04.pdf

# Applying Component Middleware to Bold Stroke




***Product-line component model***

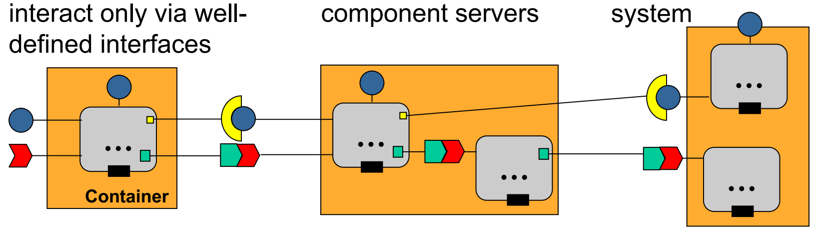
- Configurable for product-specific functionality & execution environment
- Single component development policies
- Standard component packaging mechanisms
- 3,000+ software components

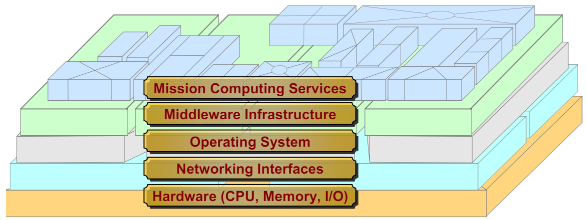





# Benefits of Component Middleware

- Creates a standard “virtual boundary” around application component implementations that interact only via well-defined interfaces
- Define standard container mechanisms needed to execute components in generic component servers
- Specify the infrastructure needed to configure & deploy components throughout a distributed system



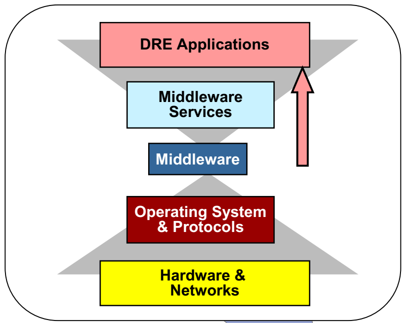





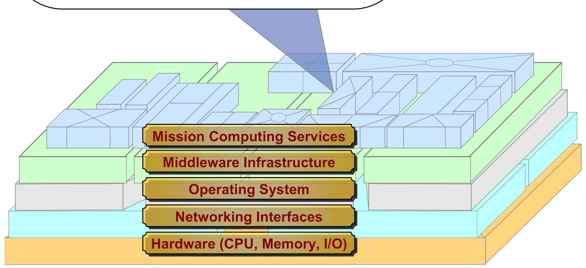
```
<ComponentAssemblyDescription id="a_HUDDisplay">
 <connection>
   <name>GPS-RateGen</name>
<internalEndPoint><portName>Refresh</portName><instance>a_GPS</
instance>
   </internalEndPoint>
   <internalEndPoint>
      <portName>Pulse</portName><instance>a_RateGen</instance>
   </internalEndPoint>
  </connection>
  <connection>
    <name>NavDisplay-GPS</name>
<internalEndPoint><portName>Refresh</portName><instance>a_NavDi
splay</instance>
    </internalEndPoint>
<internalEndPoint><portName>Ready</portName><instance>a_GPS</in
stance>
    </internalEndPoint>
  </connection>
</ComponentAssemblyDescription>
```




# Limitations of Component Middleware

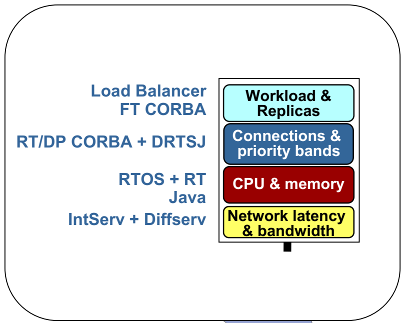



- Limit to how much application functionality can be refactored into reusable COTS component middleware

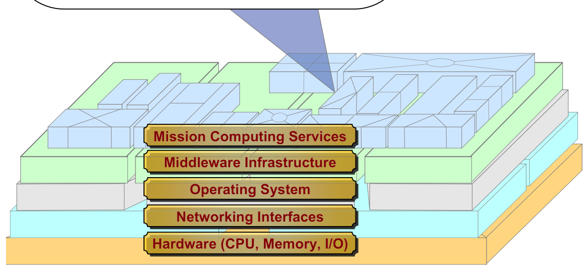




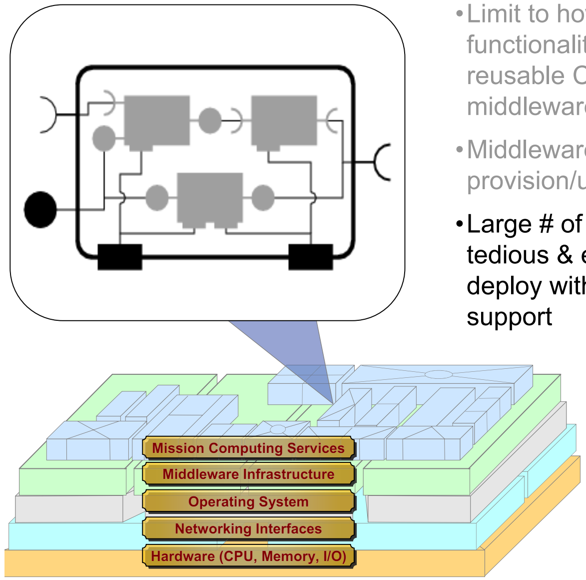

# Limitations of Component Middleware




- Limit to how much application functionality can be refactored into reusable COTS component middleware
- Middleware itself has become hard to provision/use



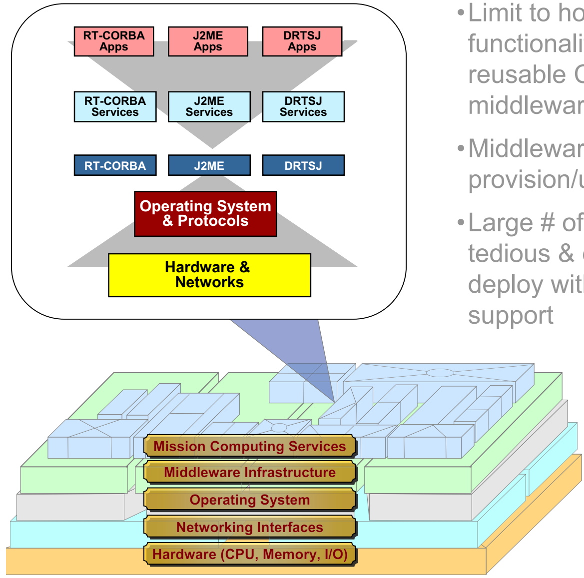
# Limitations of Component Middleware




- Limit to how much application functionality can be refactored into reusable COTS component middleware
- Middleware itself has become hard to provision/use
- Large # of components can be tedious & error-prone to configure & deploy without proper integration tool support



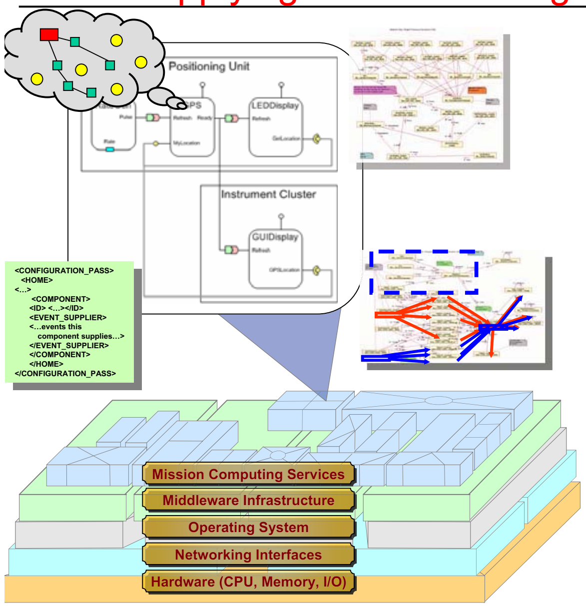
# Limitations of Component Middleware




- Limit to how much application functionality can be refactored into reusable COTS component middleware
- Middleware itself has become hard to provision/use
- Large # of components can be tedious & error-prone to configure & deploy without proper integration tool support
- There are many middleware technologies to choose from



# Applying MDD to Boeing Bold Stroke




***Model-driven development (MDD)***

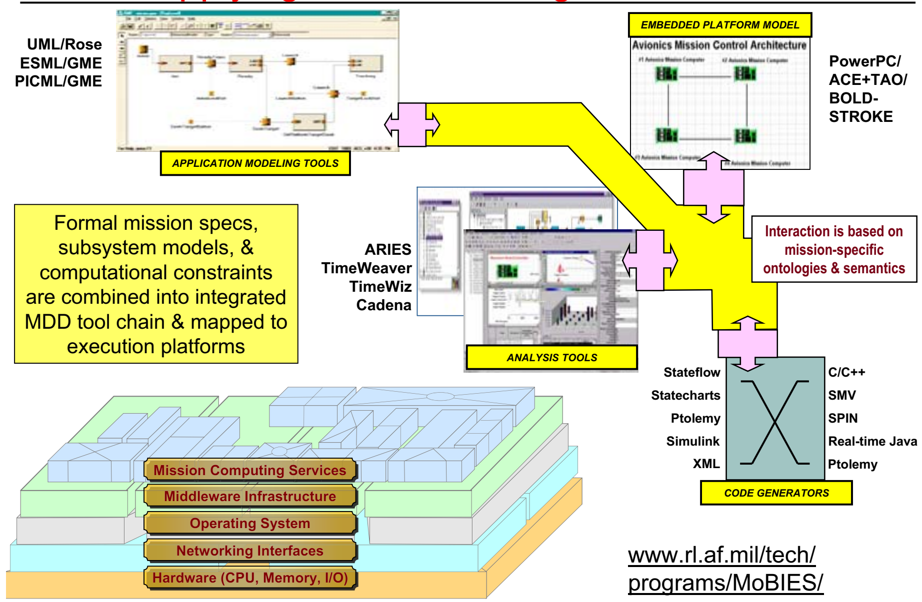
- Apply MDD tools to
  - Model
  - Analyze
  - Synthesize
  - Provision

  middleware & application components
- Configure product-specific component assembly & deployment environments
- Model-based component integration policies

www.isis.vanderbilt.edu/projects/mobies

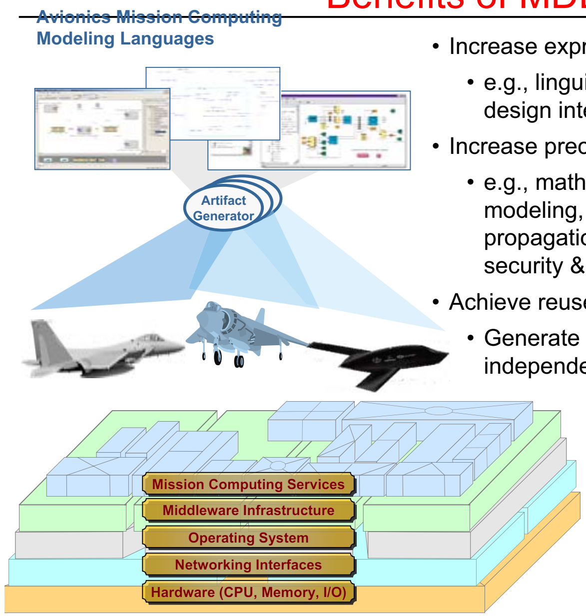


# Applying MDD to Boeing Bold Stroke




www.rl.af.mil/tech/programs/MoBIES/

## Benefits of MDD




- Increase expressivity
  - e.g., linguistic support to better capture design intent
- Increase precision
  - e.g., mathematical tools for cross-domain modeling, synchronizing models, change propagation across models, modeling security & other QoS aspects
- Achieve reuse of domain semantics
  - Generate code that's more "platform-independent" (or not)!
    - Support product-line architecture development & evolution



## Limitations of MDD



- Modeling technologies are still maturing & evolving
  - i.e., non-standard tools
- Magic (& magicians) are still necessary for success



**Model-Driven Development** of Distributed Systems

## CONTENTS





## Open MDD R&D Issues

- **Accidental Complexities**
- Round-trip engineering from models ↔ source
- Mismatched abstraction levels for development vs debugging
- Tool chain vs monolithic tools
- Backward compatibility of modeling tools
- Standard metamodeling languages & tools

- **Inherent Complexities**
- Capturing specificity of target domain
- Automated specification & synthesis of
  - Model interpreters
  - Model transformations
  - Broader range of application capabilities
  - Static & dynamic QoS properties
- Migration & version control of models
- Scaling & performance
- Verification of the DSLs

Solutions require validation on large-scale, real-world systems

## Current Status & Available Tools

- **Today's MDD tools can be used productively** – although sometimes some "magic" is necessary
  - Today's problem is not really that we need better tools, per se, **we rather need more experience with existing tools!**
- **Standardization efforts** are slowly coming to fruition: EMF/GMF, QVT, MIC, etc.

Start today – it will make you more productive

- **CoSMIC** & **CUTS** is available from www.dre.vanderbilt.edu/cosmic
- **GME** is available from www.isis.vanderbilt.edu/Projects/gme/default.htm
- **openArchitectureWare** is available from www.openarchitectureware.org



## What We Hope You Learned Today!

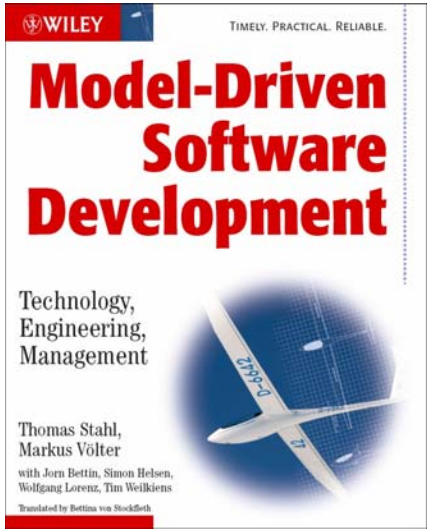
- Key MDD concepts & what kinds of domains & problems they address
- What are some popular MDD tools & how they work
- How MDD relates to other software tools & (heterogeneous) platform technologies
- What types of projects are using MDD today & what are their experiences
- What are the open issues in MDD R&D & adoption
- Where you can find more information



## Some Advertisements ☺

- Thomas Stahl, Markus Völter
- **Model-Driven Software Development**, Wiley, 2006

  www.mdsd-book.org






## Questions?