



**CSSE 490 Model-Based
Software Engineering:
Play it again Sam...
more Software Factories**



Shawn Bohner

Office: Moench Room F212

Phone: (812) 877-8685

Email: bohner@rose-hulman.edu

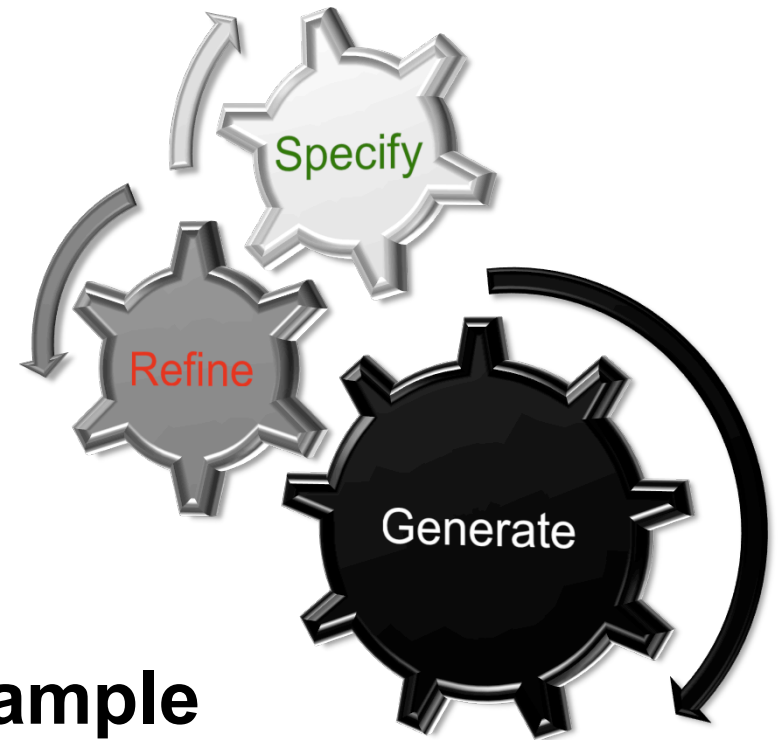


ROSE-HULMAN
INSTITUTE OF TECHNOLOGY

Learning Outcomes: MBE Discipline

Relate Model-Based Engineering as an engineering discipline.

- Demo Final Project
- Finish Software Factories
- Examine Executable UML
- Short Action Language Example
- Recipe Framework for Manual Code Inclusion
- Review Term Paper Assignment



Recall: Software Factories

- 1990's Software Factories emerged as the new automated programming
- Faced an untrained community coupled with limitations in computing capabilities
 - The Virtual Software Factory
 - Software Templates
 - Software Refinery
- Devolved into IDEs configured for efficient development of Domain applications (led by Microsoft these days)

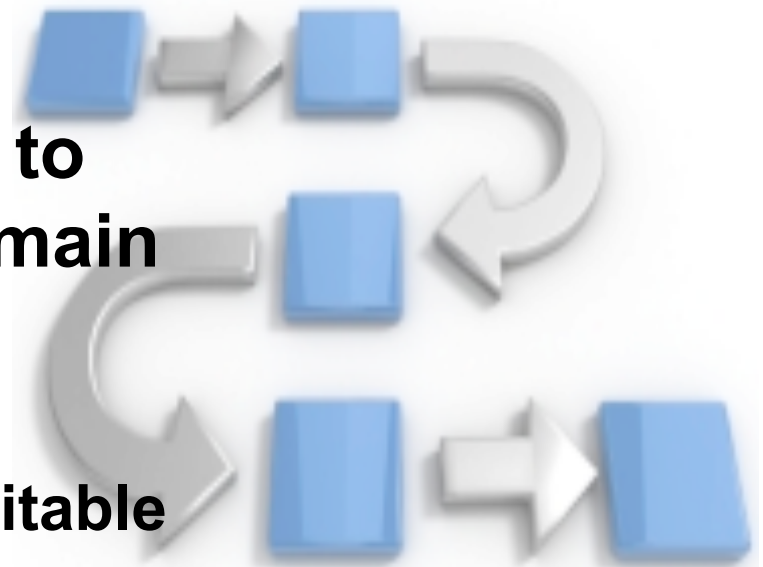


Recall: Software Factories Schema

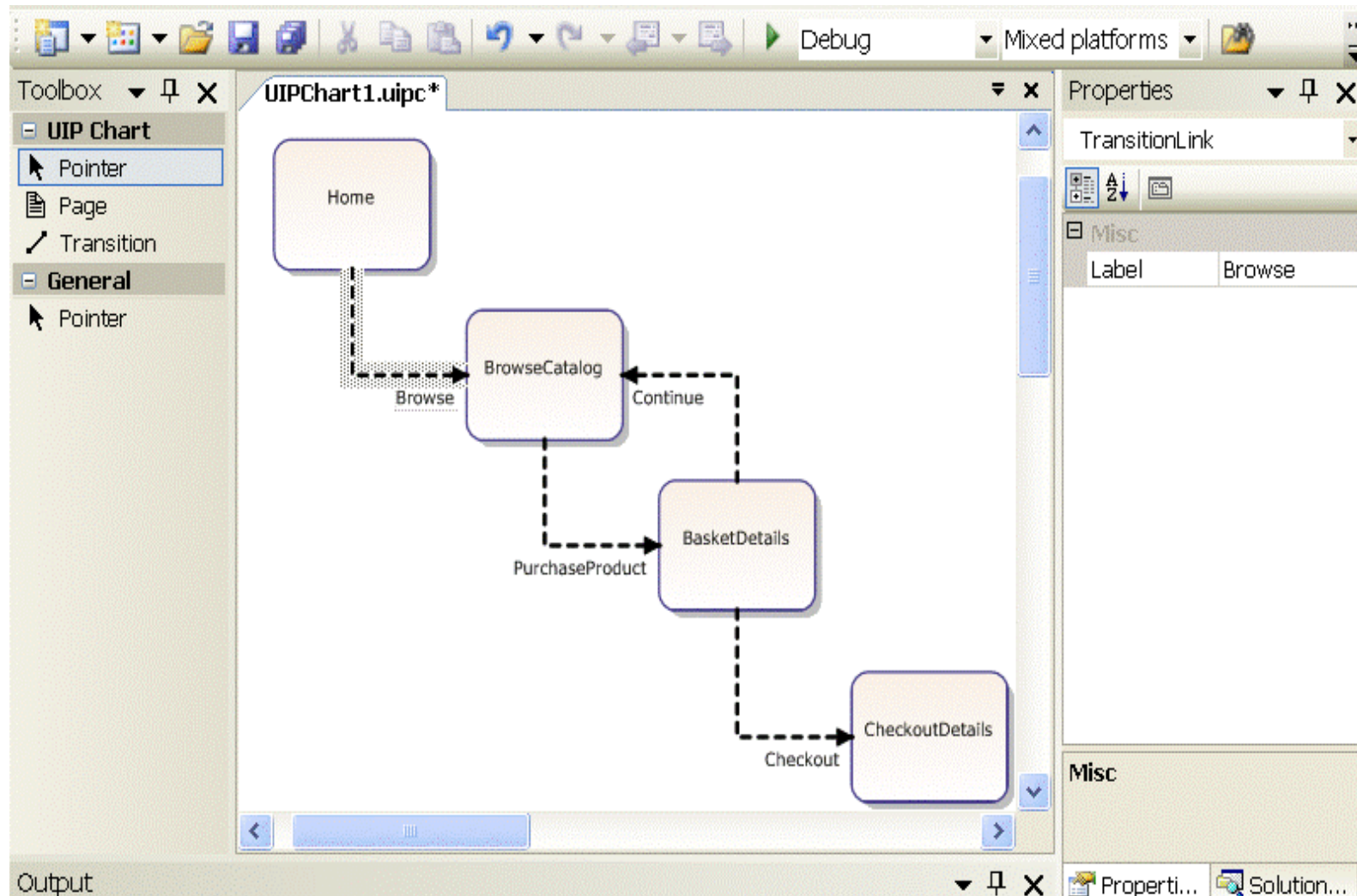
- Schema defines viewpoints for modeling and building a system (e.g., enterprise system):
 - Presentation, form layout and workflow
 - Component structure and business data model
 - Persistence mapping, Deployment, ...
- Schema identifies core artifacts as well as the most efficient way of producing them
 - DSLs, frameworks, patterns, manual programming
- Schema identifies commonalities and differences among applications in the domain

Software Factories Templates

- **Makes the Schema usable**
- **Load SF Template into IDE to configure it for specific domain**
 - Provides the necessary frameworks or libraries
 - Contributes project types suitable for the factory
 - Delivers build scripts
 - Extends IDE with DSL editors and transformations



MS DSL Tools Example



Defining a Metamodel

Toolbox: Domain Mo...
Pointer
Class
Value Prop...
Embedding
Reference
Inheritance
General
Pointer

ResealdePetri.dd ResealdePetri.dmd

ElementHasElement
PlaceHasJeton
PlaceHasTransition
TransitionHasPlace
RDP
Elts
Element
ReferredE
Element
Jeton
Place
Transition
Jeton
Transition
TPlace
Place

Solution Explorer - Solution 'ResealdePetri' (3 projects)

Resources
BLinkBitmap.bmp
JetonShapeBitmap.bmp
PlaceShapeBitmap.bmp
TransitionShapeBitmap.bmp
Shell
CommandSet.cs
DesignerCallbacks.cs
EditorFactory.mdfddt
Guids.cs
PkgCmdID.cs
ResealdePetriDocData.mdfddt
ResealdePetriDocView.cs
ResealdePetriPackage.cs
ResealdePetriPropertyDescriptionRedirector
Templates
rdp.rdp
ResealdePetri_cs.vstemplate
ResealdePetri_vb.vstemplate
ResealdePetri.dd
DesignerUIT
ObjectModel
Properties
References
app.config
Enumerations.mdfomt
ObjectModelExtras.mdfomt
PreBuild.bat
ResealdePetri.dmd
ResealdePetri.dmd.mdfomt
ResealdePetri.Resource.resx

Properties
Enumerations.mdfomt File Properties

Advanced
Build Action: None
Copy to Output Directory: False
Custom Tool: DmdFileCodeGenerator
Custom Tool Namespace:
File Name: Enumerations.mdfomt
Full Path: C:\Documents and Settings\...

Output
Show output from: Build
ToolWindow: Example.ResealdePetri.Designer.ResealdePetriModelExplorer, {c5bd4cf5-1704-4ca7-9b70-8b946a763781}
Editor Extension: .rdp, {a29fab9e-fa92-4582-8450-8cd792d85bab}
LoadKey: ResealdePetriPackageName
Version 1.0.0.0
Edition Required: Standard
SUCCEEDED: Example.ResealdePetri.Designer
==== Build: 1 succeeded, 0 failed, 2 up-to-date, 0 skipped =====



Software Factory and MBSE's

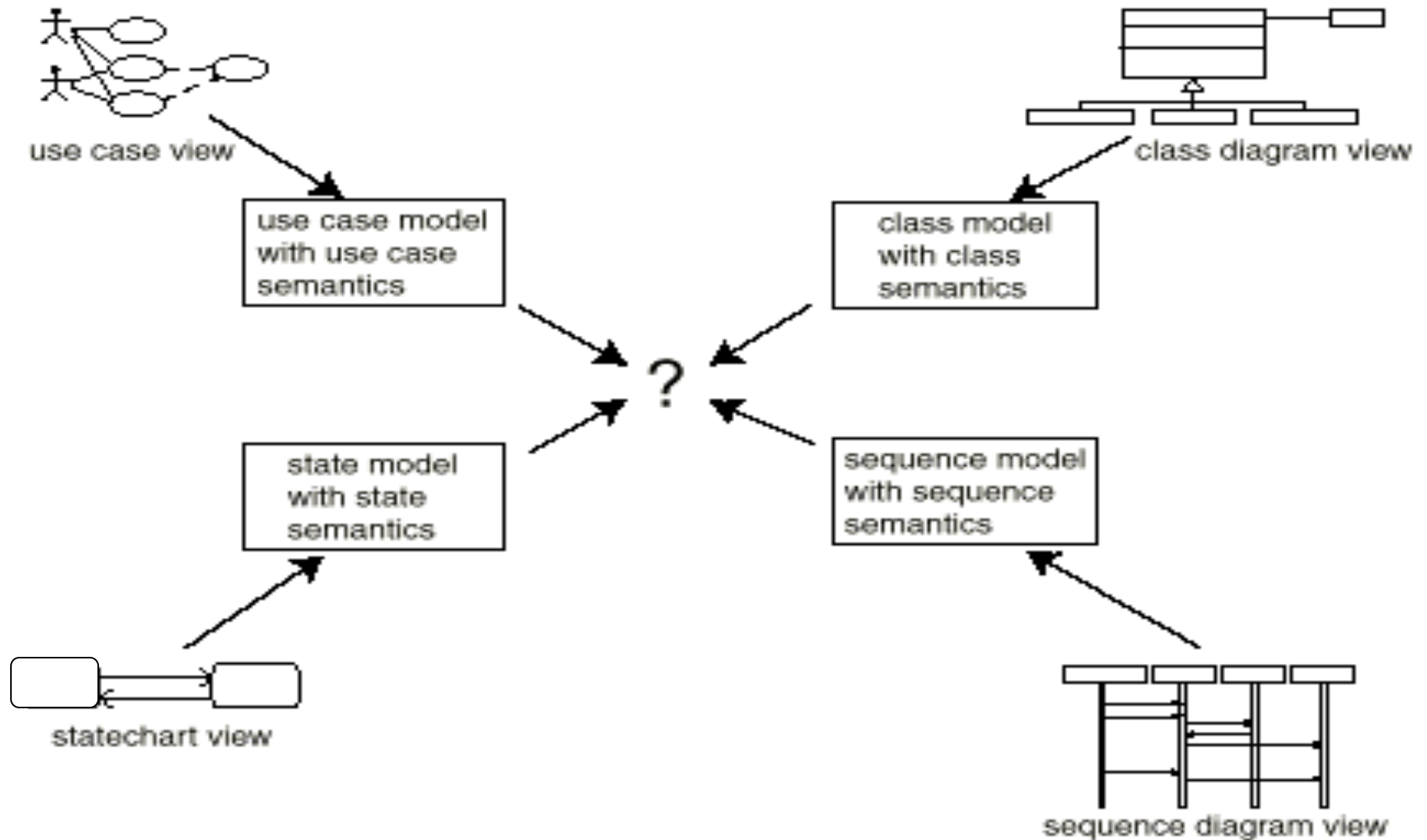
- **SFs use model-based concepts without major changes**
 - DSLs are used to build models, Languages often graphical
 - Some provide tooling to define the metamodels as well as concrete syntax and editors

- **SFs seldom use OMG standards for their infrastructure**
 - DSLs are not UML based
 - Metamodels are not based on the MOF, and not QVT

- **Application developer's perspective**
 - Models are first class artifacts in development projects
 - Editors and transformations integrate seamlessly with the IDE

- **Infrastructure developer's perspective**
 - Metamodels, editor definitions and transformations are first class artifacts
 - Tools to build them are seamlessly integrated into the IDE

Same Semantics for Different Views

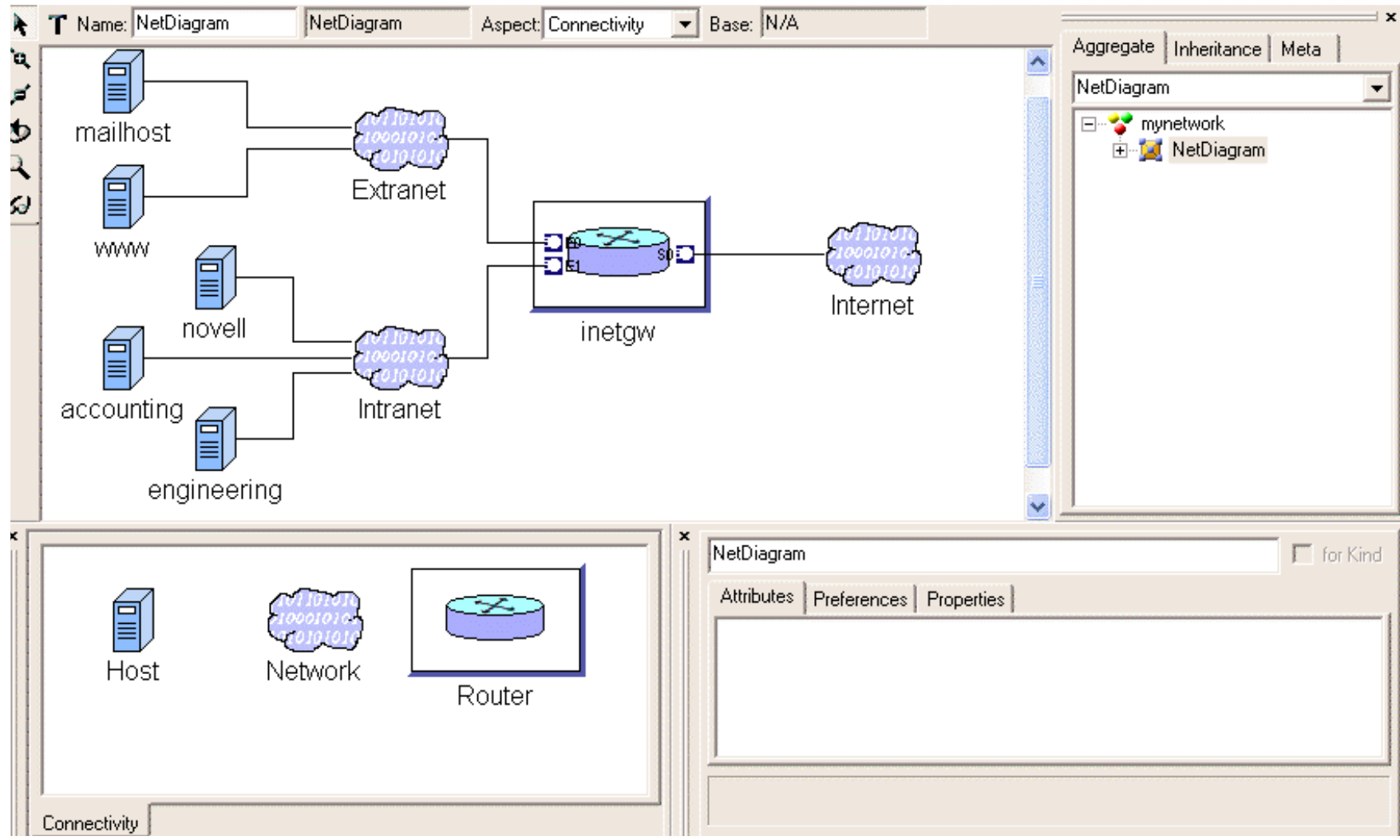


Graphical Model Editor

The screenshot displays a graphical model editor interface with several key components labeled:

- Menubar:** Located at the top left, containing menu items like File, Edit, View, Window, and Help.
- Toolbar:** A row of icons below the menubar for various editing functions.
- Modebar:** A vertical bar on the left side with icons for different editing modes.
- Model Editing Windows:** The main workspace showing a hierarchical model. The top window, titled "System", shows a flow from "PreProcessing" to "Processing" to "PostProcessing". The "Processing" window is open, showing a detailed view of the "Processing" component with inputs "In" and "Time", and outputs "Branch0" and "Branch1".
- Model Browser:** A tree view on the right side showing the project structure, including folders like "System", "PostProcessing", "PreProcessing", "Processing", and "Branch0".
- Partbrowser:** A panel at the bottom left showing available components: "CompoundParts", "InputSignals", "OutputSignals", and "PrimitiveParts".
- Attribute Browser:** A panel at the bottom right showing the properties of the selected "Branch0" component, including "Firing" (IFALL), "Script" (ComputeTime), and "Priority" (10).
- Statusbar:** Located at the bottom of the window, showing the current state as "Ready" and the window title "EDIT | 100% | SF2000 | 11:39 AM".

GME: Modeling based on previously defined Metamodel



GME: OCL Constraint Validation

T Name: SF2000 | ParadigmSheet | Aspect: Constraints | Base: N/A

```

classDiagram
    class Folder["Folder <<Folder>>"]
    class Processing["Processing <<Model>>"]
    class Primitive["Primitive <<Model>>"]
    class Compound["Compound <<Model>>"]
    class Signal["Signal <<Atom>>"]
    class InputSignal["InputSignal <<Atom>>"]
    class OutputSignal["OutputSignal <<Atom>>"]
    class DataflowConn["DataflowConn <<Connection>>"]
    class ParameterConn["ParameterConn <<Connection>>"]

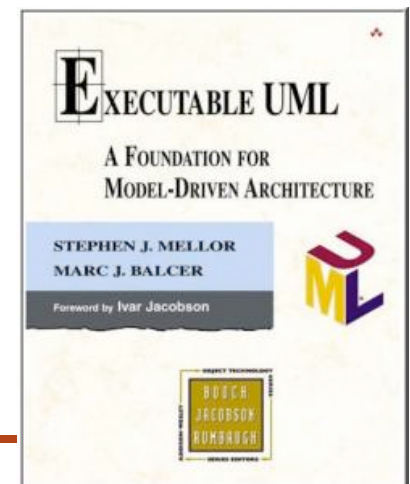
    Folder <|-- Primitive
    Processing <|-- Primitive
    Processing <|-- Compound
    Signal <|-- InputSignal
    Signal <|-- OutputSignal
    
```

AtLeastOnePart for Kind

Attributes	Preferences	Properties
Description:	Compounds must have parts	
Default parameters:		
Equation:	self.parts()->size > 0	
Priority (1=High):	2	
Depth:	1	
On close model	False	
On create	False	
On delete	False	
On new child	False	
On lost child	False	
On move	False	
On derive	False	
On connect	False	
On disconnect	False	
On change attribute	False	
On change property	False	
On change assoc.	False	
On refer	False	
On unrefer	False	
On include in set	False	
On exclude from set	False	

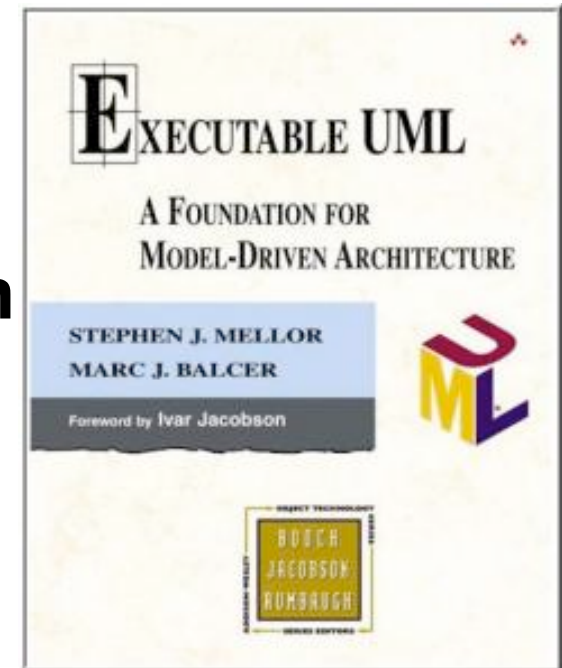
Executable UML (xUML) Concepts

- Executable UML is not a formal standard, but a goal for a UML-based programming language
- Must eliminate redundancy and ambiguities, to increase executability of UML
- Action language needed to define complete implementations of software systems
- Not a DSL, but rather a universal, UML-based programming language



Executable UML → Action Semantics

- Hard to model a complete system today via UML or even MOF-based languages
- Action semantics do not contain structural constructs (classes, attributes & relationships)
 - Already defined in the structural part of the model
 - Merely define behavioral building blocks

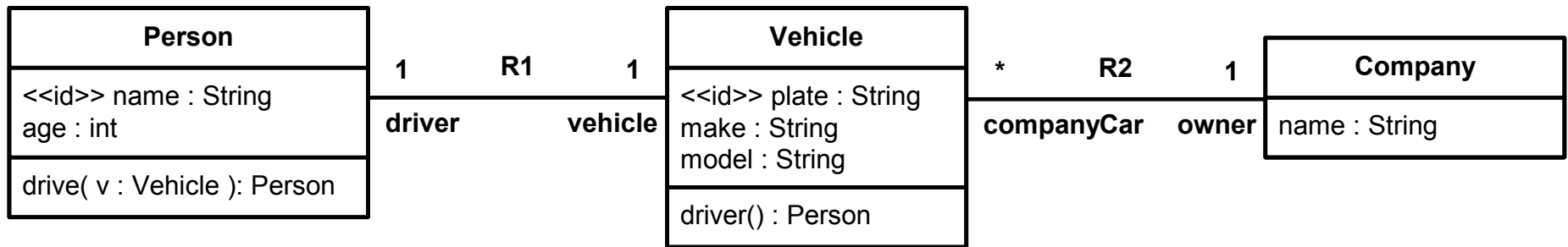




Action Semantics in UML 2.0

- Models **procedural behavior** via abstract syntax
- **Variables** for assigning/reading (sets, bags...)
- Arithmetic and logical operations
- Control flow (if-then-else, case, block...)
- Class extents that be queried (SQL-like)
- Creation, deletion, and navigation of associations
- Generation of signals and timers
- Definition of functions

Action Languages Example 1/3



```
myJeep = create Vehicle with plate = "IYQ2"  
myJeep.make = "Chrysler Jeep"  
myJeep.model = "Liberty CRD"
```




Action Languages Example 2/3

```
shawn = create Person with name = "Shawn"
```

We can now call the operation drive() to let the driver drive the vehicle.

```
[actualDriver] = drive[aVehicle] on shawn
```

What is still missing, of course, is the implementation of the operation drive(). The least it must do is to instantiate the association R1 (that is, to create a link between the two concerned objects).

```
link this R1 aVehicle
```



Action Languages Example 3/3

```
theCurrentDriver = this.R1.“driver“
```

Let's assume we want to find all people in the system:

```
{allPersons} = find-all Person
```

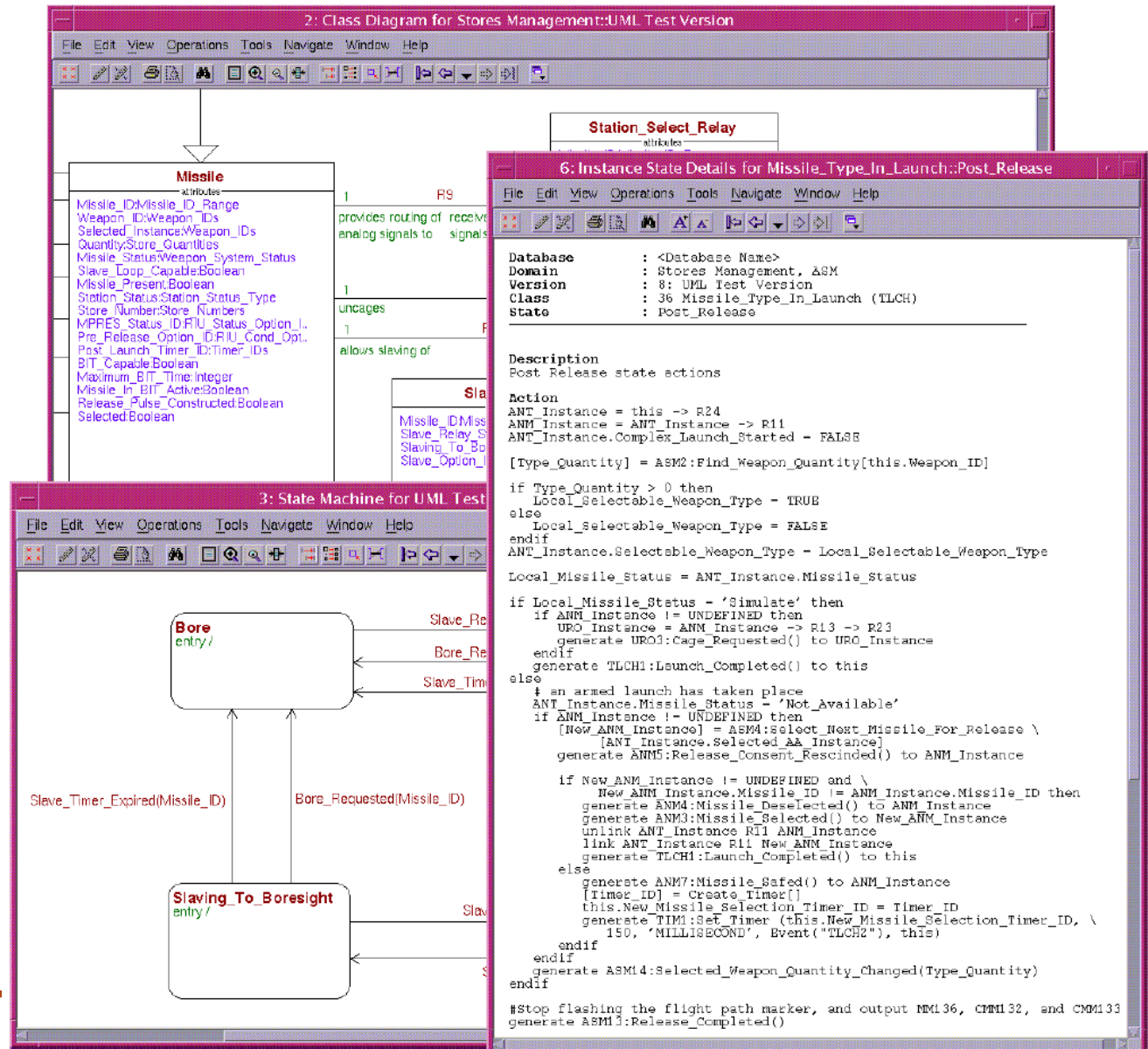
The braces state that allPersons is a set of objects instead of just one.

One can also limit such a search. For example, all vehicles of the brand Jeep can be looked for.

```
{Jeeps} = find Vehicle where make = “Jeep”
```

Example: Kennedy Carter's iUML

- Model Diagrams
- Code
- Integration



Recipe Framework for Integrating Manually Developed Code

The screenshot illustrates the Recipe Framework integration process in an IDE. The Package Explorer on the left shows the project structure, including folders like `>example`, `>BC1.impl`, `>BC2.impl`, `>tests`, `>src-gen`, `>BC1`, `>impl`, `>CannotAdd.java`, `>IC1.java`, `>ICalculator.java`, `>BC2`, `>exsys.client`, `model`, `build.properties`, `build.xml`, `>dump.dump`, and `>helloWorld.recipes`. The Ant console at the bottom shows build tasks such as `oAW - antlrInstantiator - build`, `oAW - antSupport - build`, `oAW - core - build`, `oAW - metamodelGenerator - build`, `oAW - recipe.ant - build`, `oAW - recipe.core - build`, `oAW - recipe.plugin - update`, `oAW - recipe.simpleChecks - build`, and `oAW - umlMetamodel - 1) generate`.

The main editor displays a recipe file with annotations 1-7. The Recipes view shows a table of recipe parameters and their values:

Name	Value
<code>_type</code>	<code>org.open</code>
<code>_type</code>	<code>org.open</code>
<code>className</code>	<code>example.BC1.impl.C1Implementation</code>
<code>element</code>	<code>:example:BC1:C1</code>
<code>projectName</code>	<code>scmHelloWorld</code>
<code>srcPath</code>	<code>src</code>

The generator generates a base class for components, in this case `example.BC1.impl.C1ImplementationBase`. From this base class you have to extend your own class that has to be called `example.BC1.impl.C1Implementation`.



Write and Present a Term Paper

- Use IEEE/ACM format for the paper (template provided on Angel)
- Include abstract, introduction, background/related work, analysis, and conclusion (along with references)
- Target 5-7 pages
 - If you are not a strong writer, use a lot of tables and figures to organize your work
 - Use your own words - copied elements without reference are considered plagiarism
- Paper due 11:55pm Tuesday, May 17th, 2011 (Angel dropbox)
- Presentation due by class time on May 19th, 2011 (pres. Delivered to Angel dropbox)



Homework and Milestone Reminders

- **Term Paper**

- Paper Due by 11:55pm, Tuesday, May 17th, 2011.

- **Term Paper and Presentation**

- Presentation Due by 1:35pm, Thursday, May 19th, 2011.