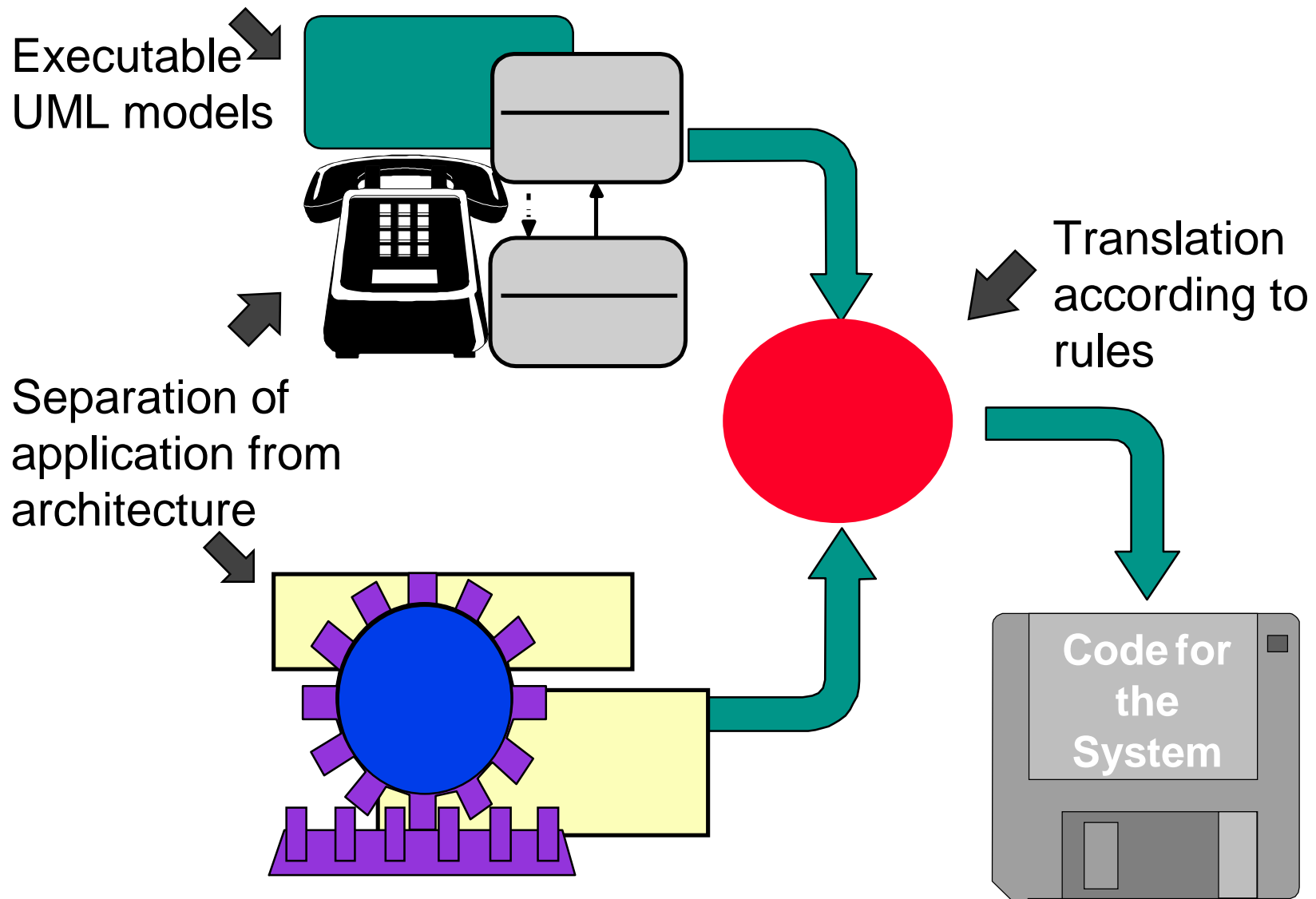


# **System Design: Architectures and Archetypes**

**Stephen J. Mellor  
Project Technology, Inc.  
<http://www.projtech.com>**

**PROJECT TECHNOLOGY** INC.  
|||||

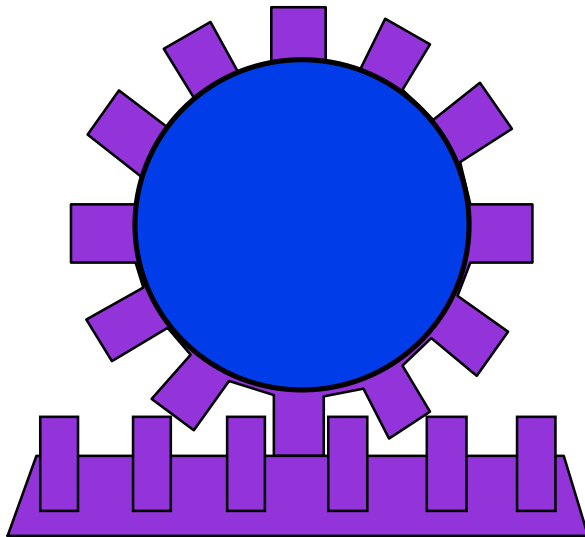
# Properties



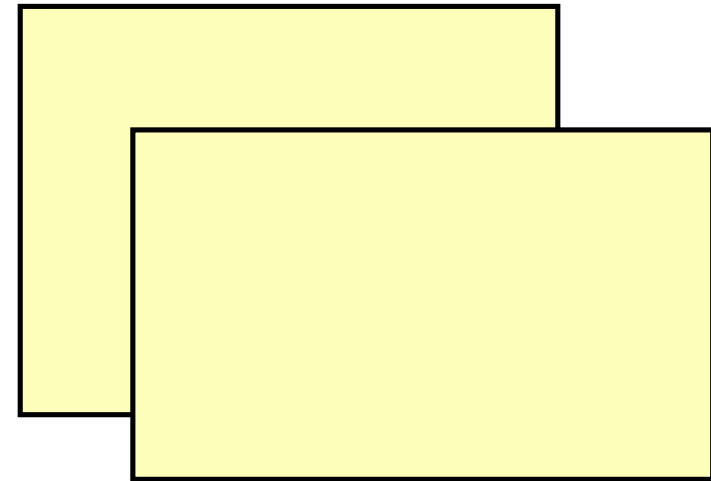
# What's in the Architecture?

The architecture comprises:

- ◆ an execution engine plus
- ◆ a set of archetypes.



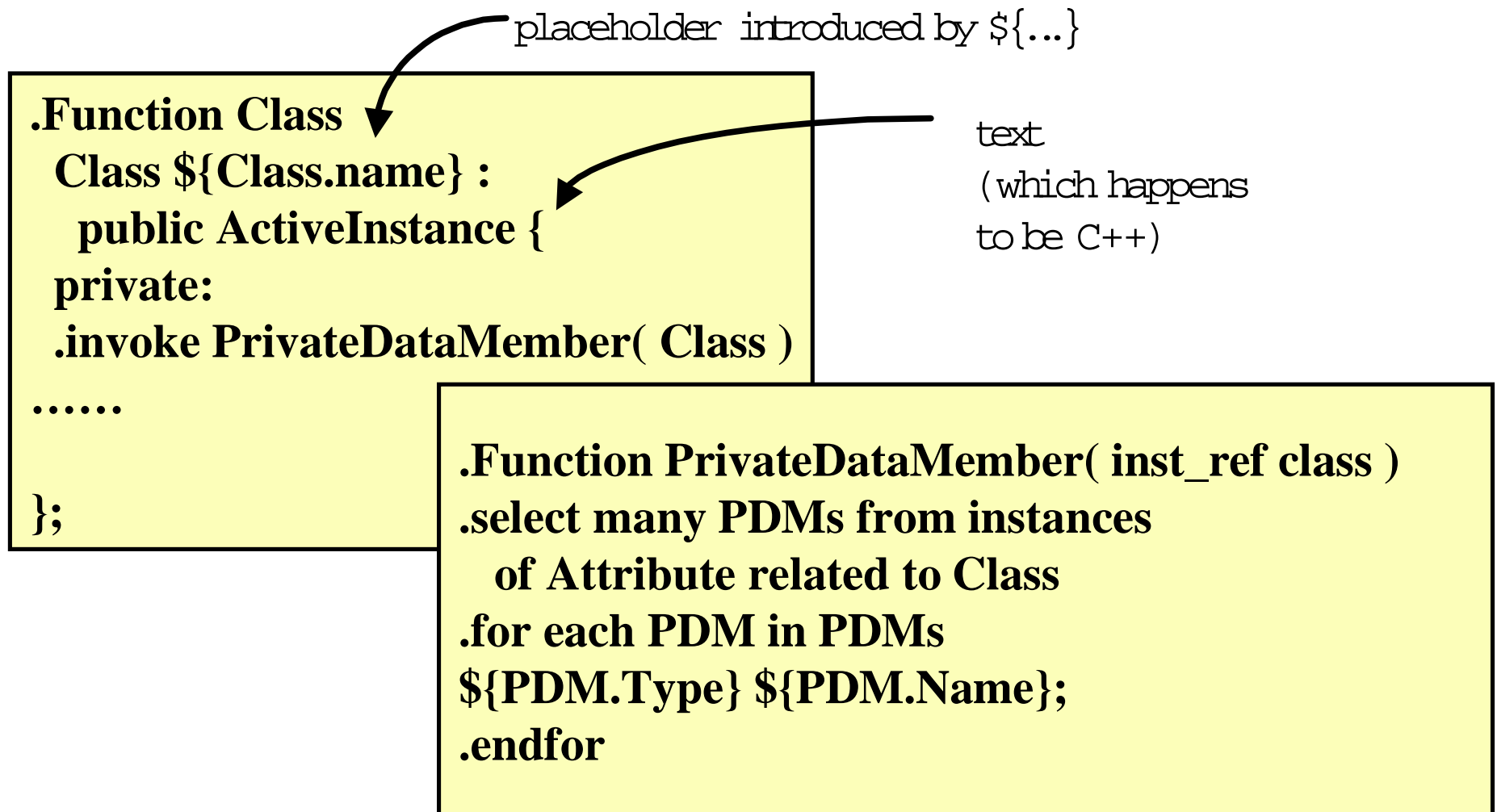
Execution Engine



Archetypes

# Archetypes

Archetypes define the rules for translating the application into a particular implementation.



# Application-Independent Software Architecture

PROJECT TECHNOLOGY, INC.

The software architecture is independent of the semantics of the application.

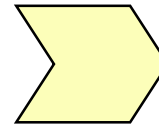
This offers:

- ◆ reuse of the architecture
- ◆ faster performance tuning
- ◆ faster integration
- ◆ faster, cheaper retargeting
- ◆ centralized diagnostics and instrumentation
- ◆ centralized error-detection and recovery
- ◆ specialization of skills

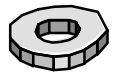
# Table of Contents



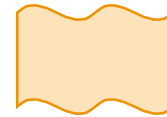
Executable Domain  
Models



A Direct Translation



Model Execution



An Indirect  
Translation



Model Capture



Model-Driven  
Architecture



Archetype Language

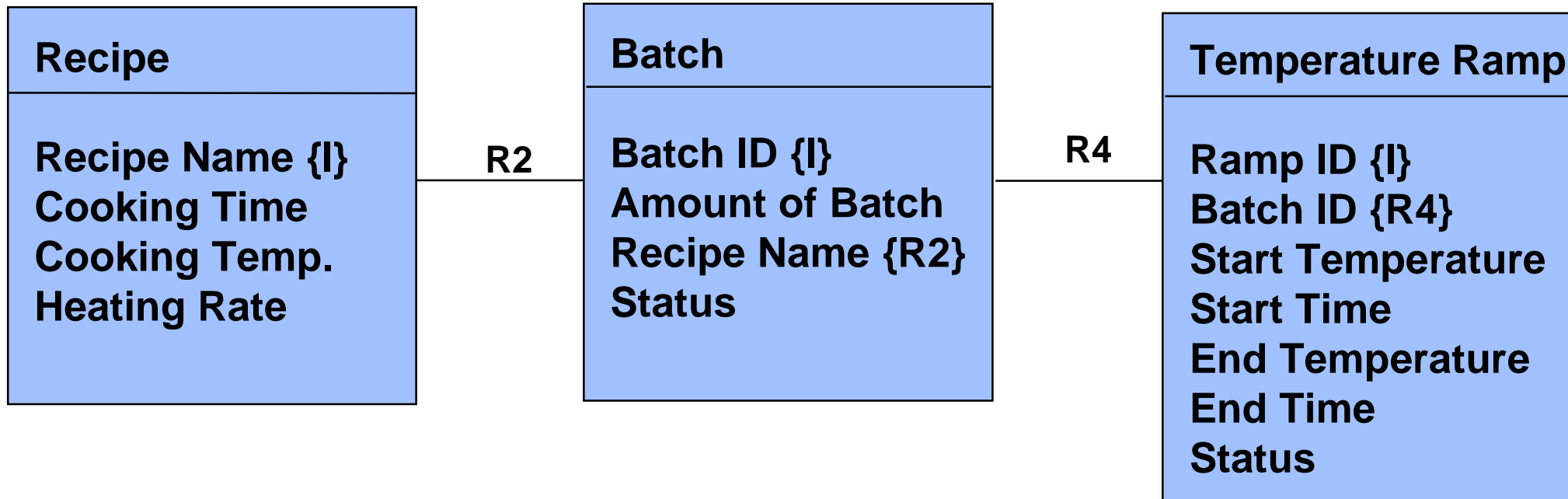
# Executable Domain Models



# Class Diagram

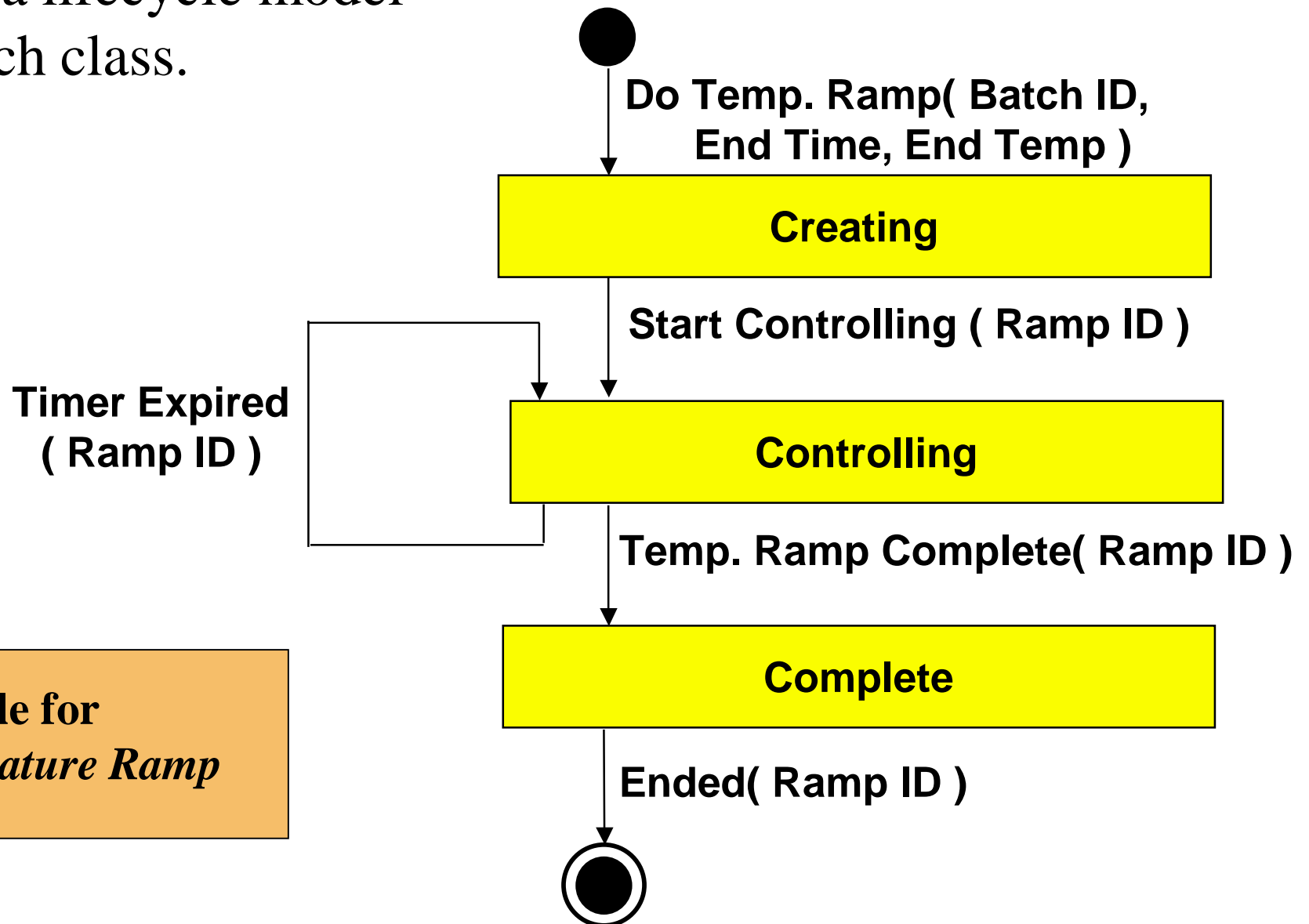
Abstract classes based on both:

- ◆ data, and
- ◆ behavior



# Lifecycles

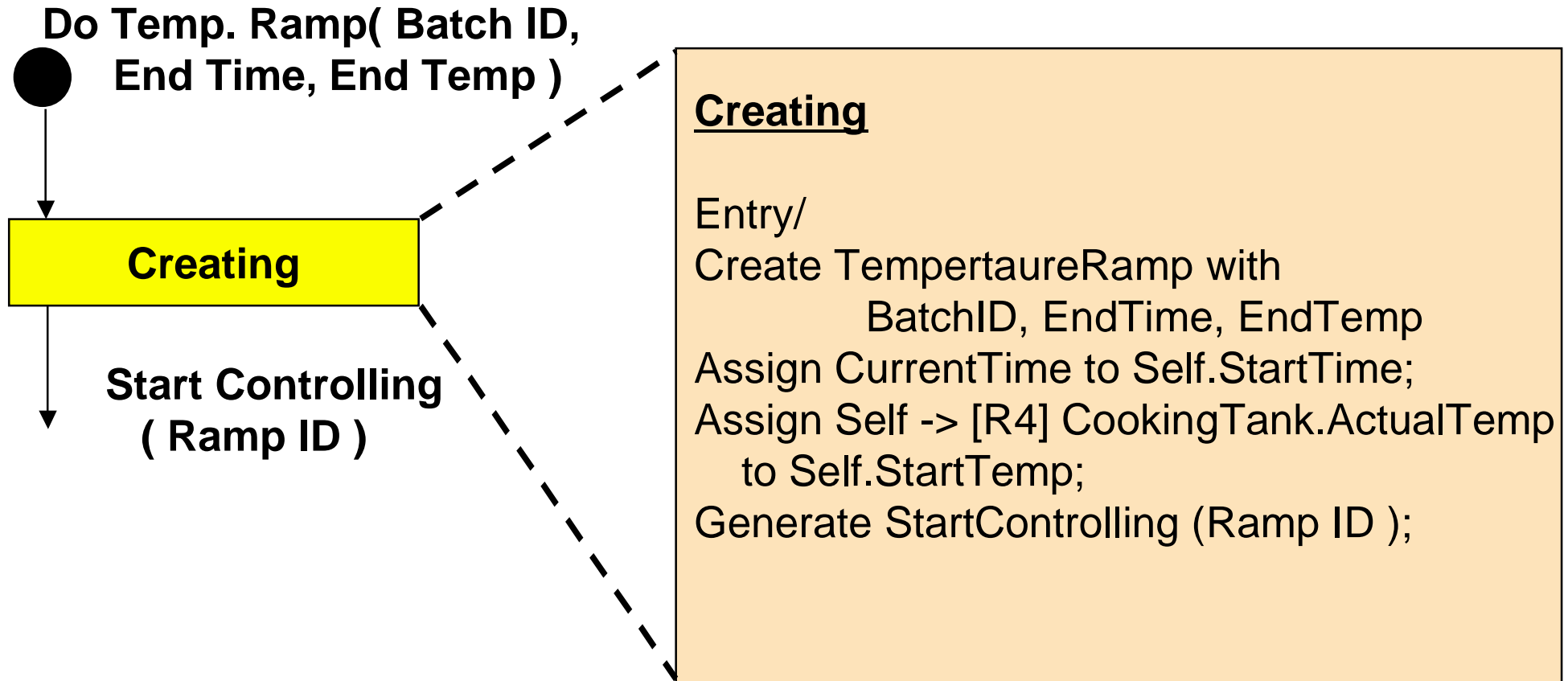
Build a lifecycle model for each class.



Lifecycle for  
*Temperature Ramp*

# Actions

Specify the logic for each state's action.



# Action Semantics

The action semantics should:

- ◆ not over-constrain sequencing
  - i.e concurrency & data flow
- ◆ separate computations from data access
  - to make decisions about data access without affecting algorithm specification
- ◆ manipulate only UML elements
  - to restrict the generality and so make a specification language

## Creating

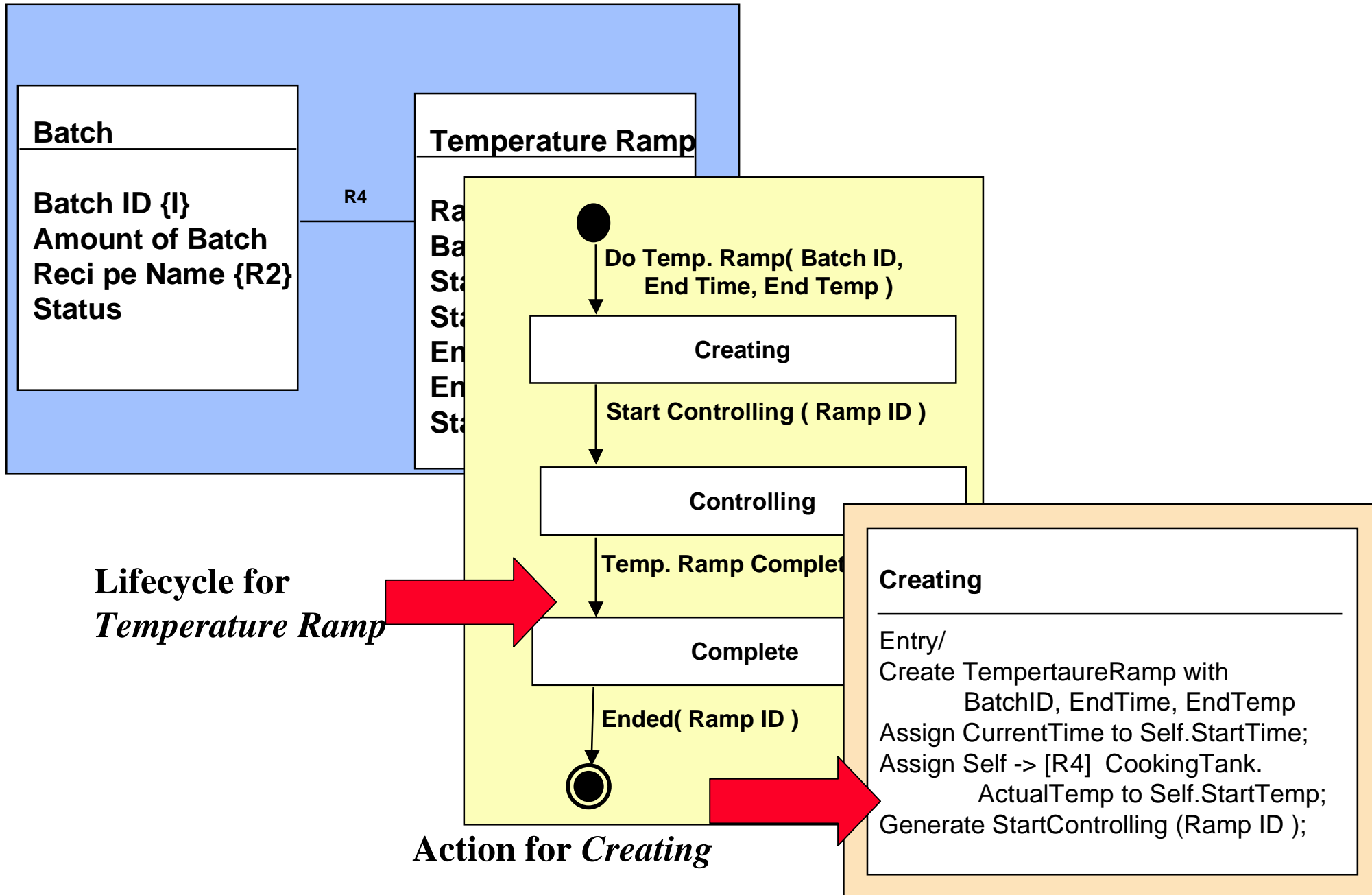
Entry/

Create TempertaureRamp with  
BatchID, EndTime, EndTemp

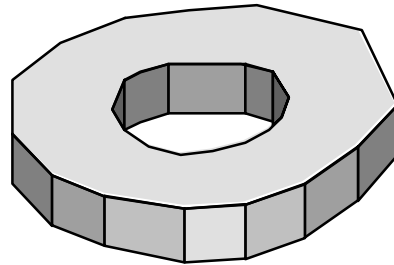
Assign CurrentTime to Self.StartTime;

Assign Self -> [R4] CookingTank.ActualTemp  
to Self.StartTemp;

# An Executable Model

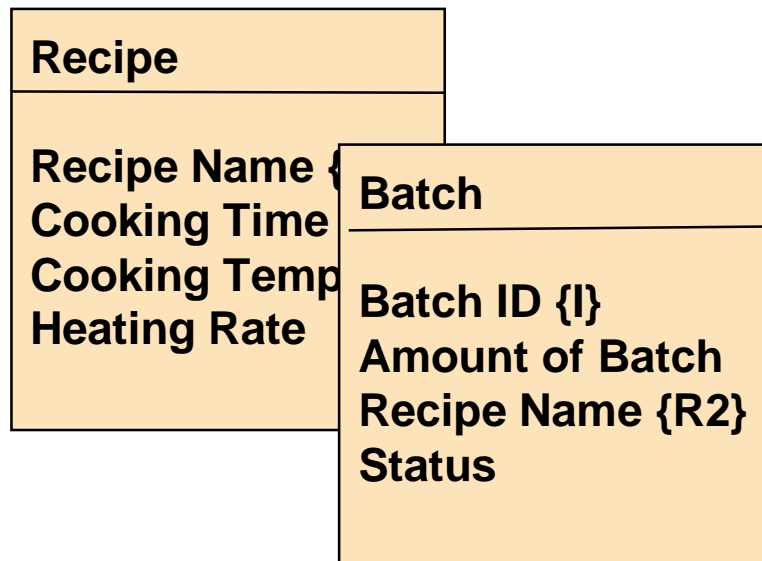


# Model Execution



# Instances

An executable model operates on data about instances.



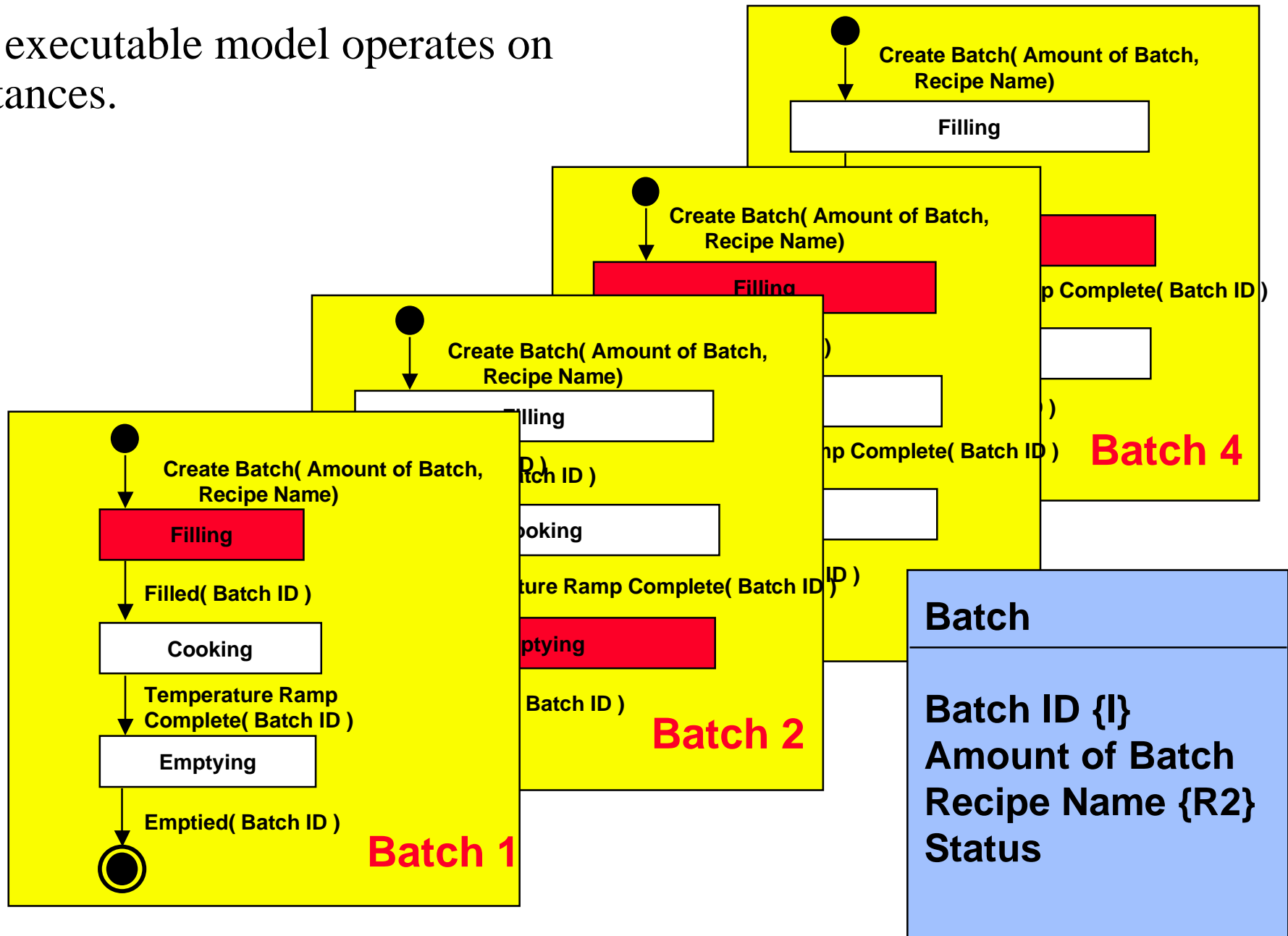
Recipe			
Recipe Name	Cooking Time	Cooking Temp	Heating Rate
Nylon	22	200	2.22
Kevlar			
Stuff			

Batch			
Batch ID	Amount of Batch	Recipe Name	Status
1	100	Nylon	Filling
2	127	Kevlar	Emptying
3	93	Nylon	Filling
4	123	Stuff	Cooking

# Instances

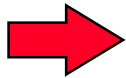
An executable model operates on instances.



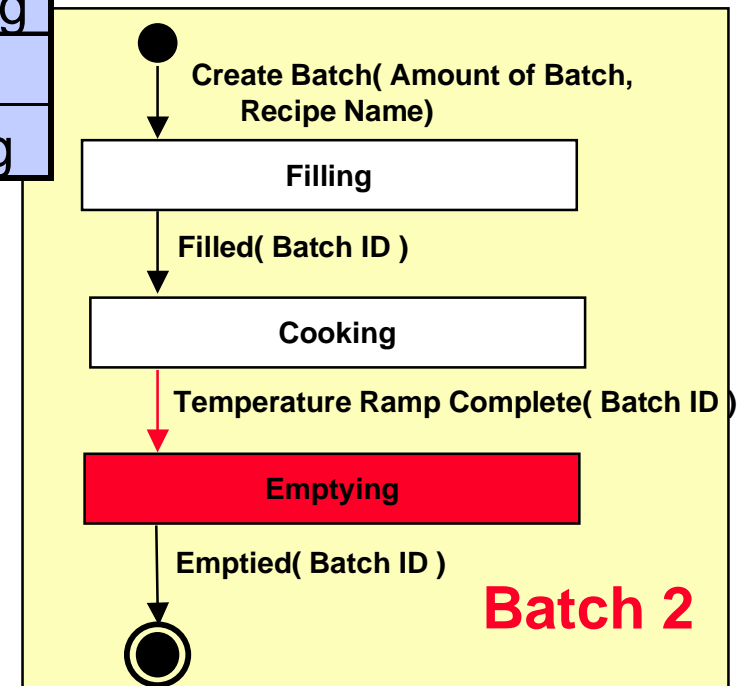
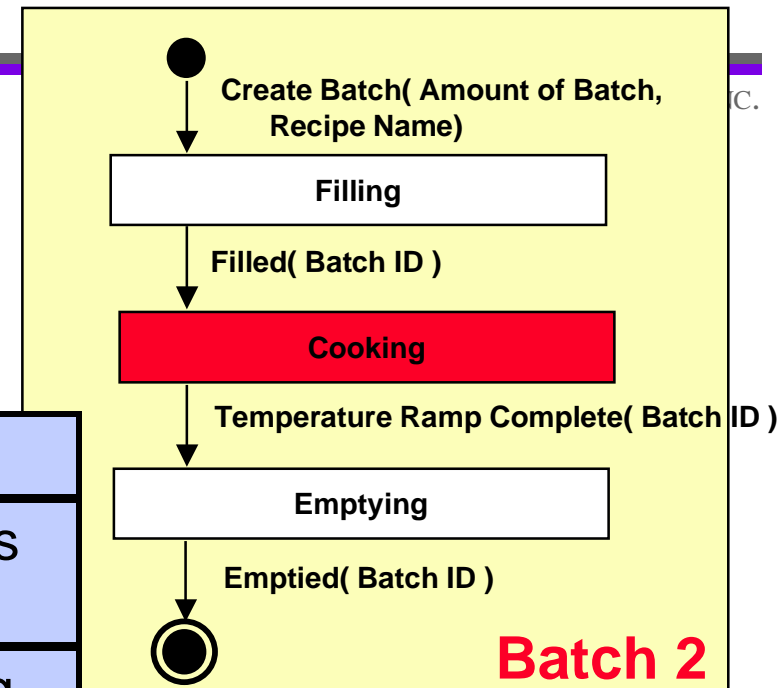
# Execution

The lifecycle model prescribes execution.

Batch			
Batch ID	Amount of Batch	Recipe Name	Status
1	100	Nylon	Filling
2	127	Kevlar	Emptying
3	93	Nylon	Filling
4	123	Stuff	Cooking

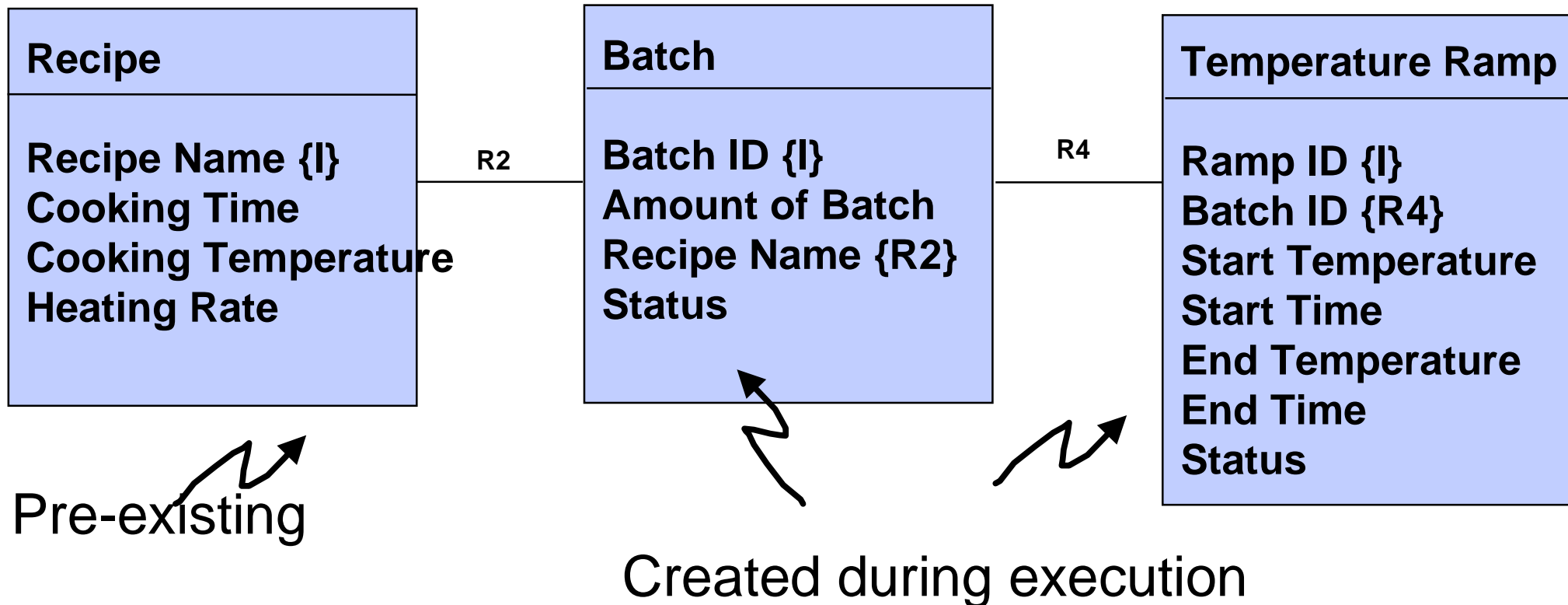


When the Temperature Ramp is complete, the instance moves to the next state....and executes actions.



# Pre-existing Instances

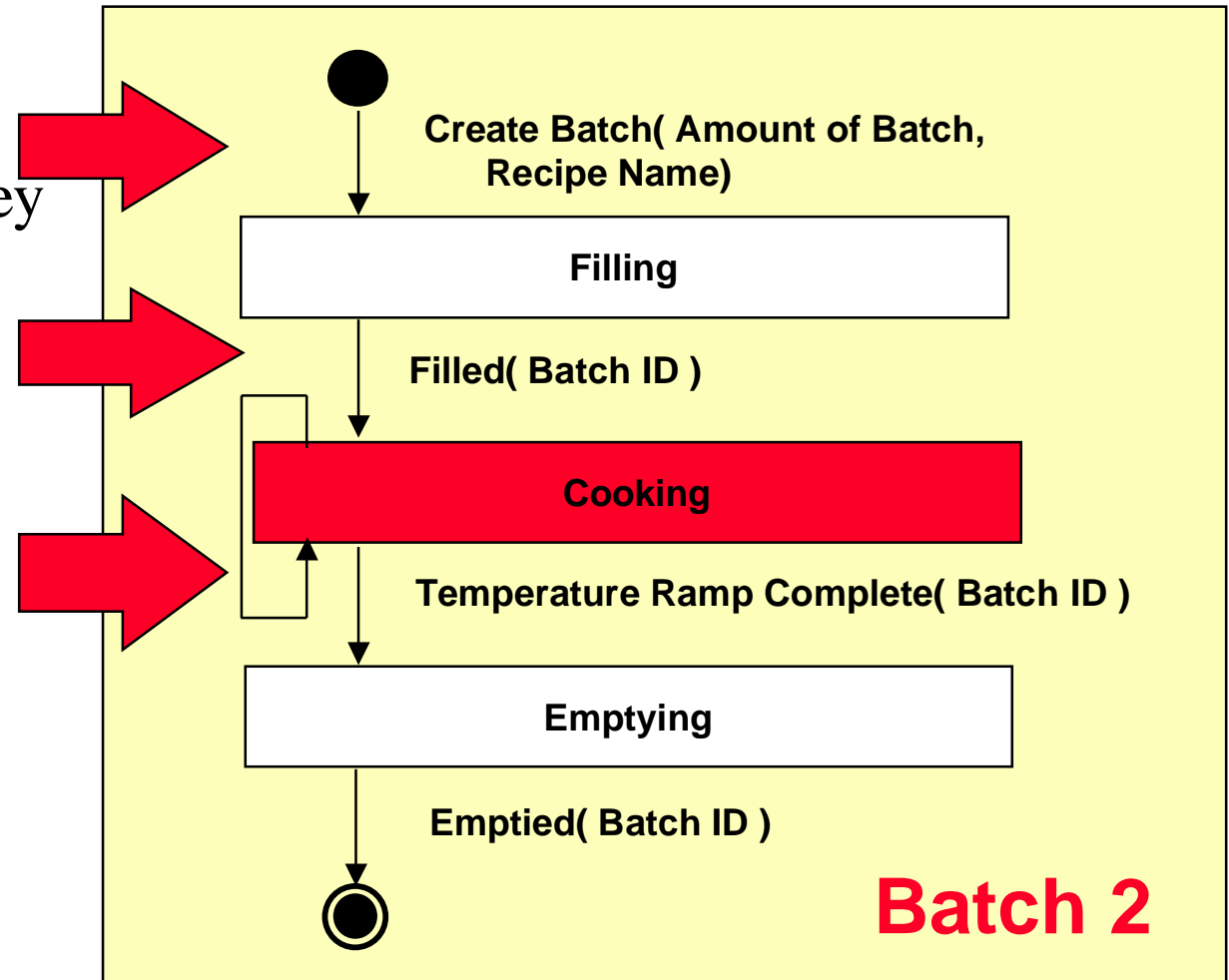
Some instances exist before the model begins to execute...



# Executing the Model

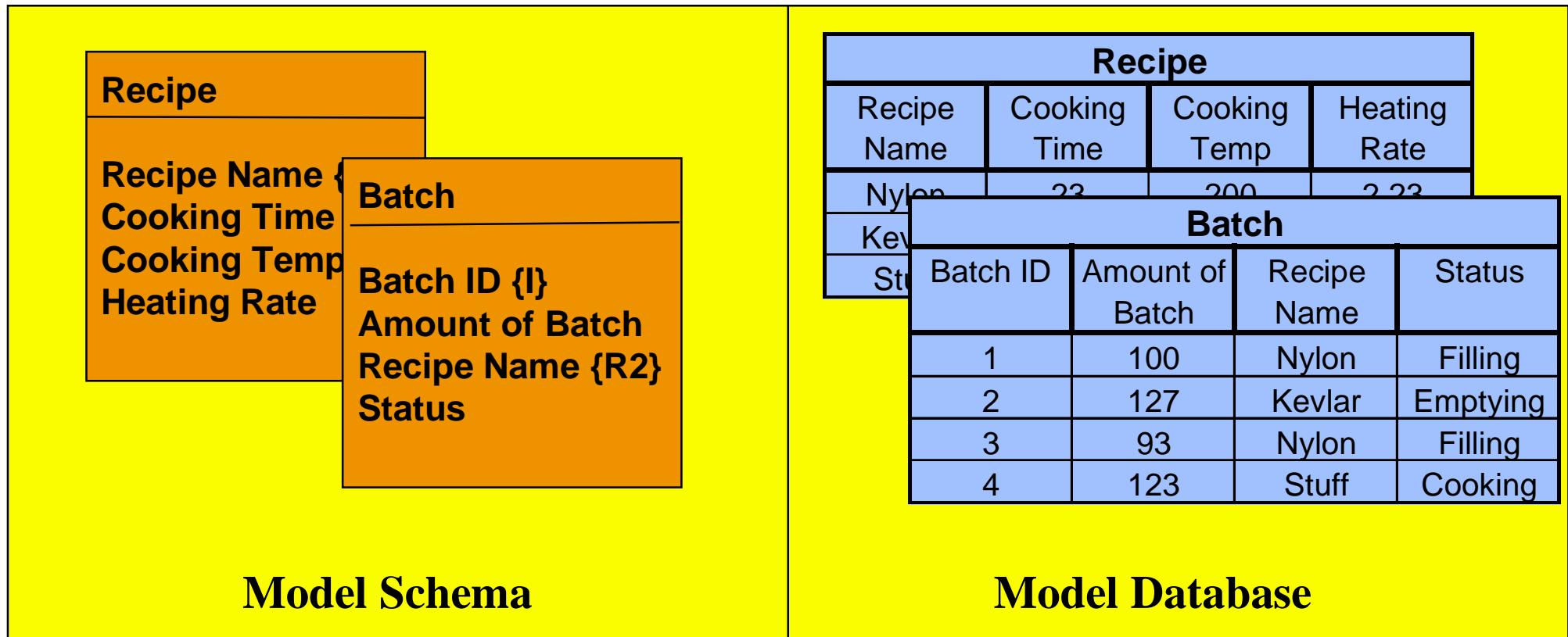
The model executes in response to signals from:

- ◆ the outside,
- ◆ other instances as they execute
- ◆ timers

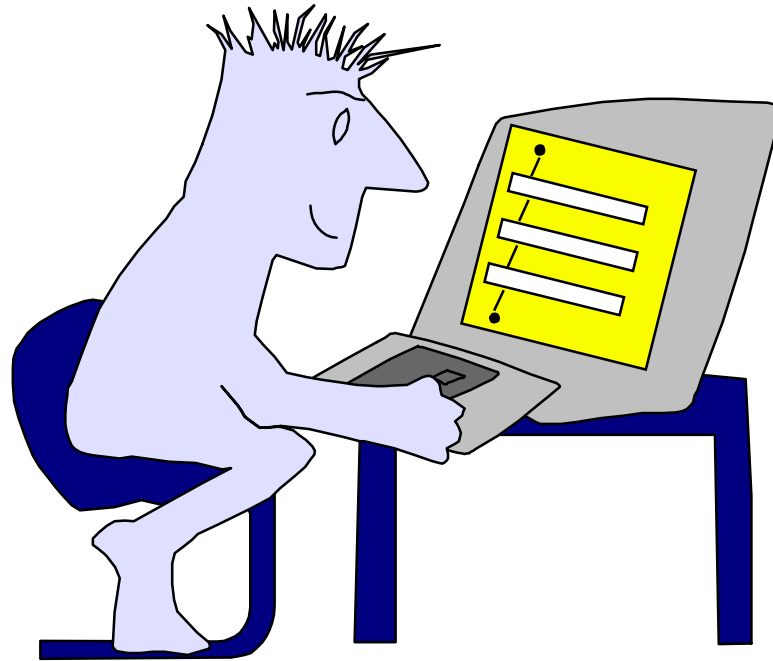


# Model Database

Each schema has a corresponding database for instances.



# Model Capture



# Model Repository

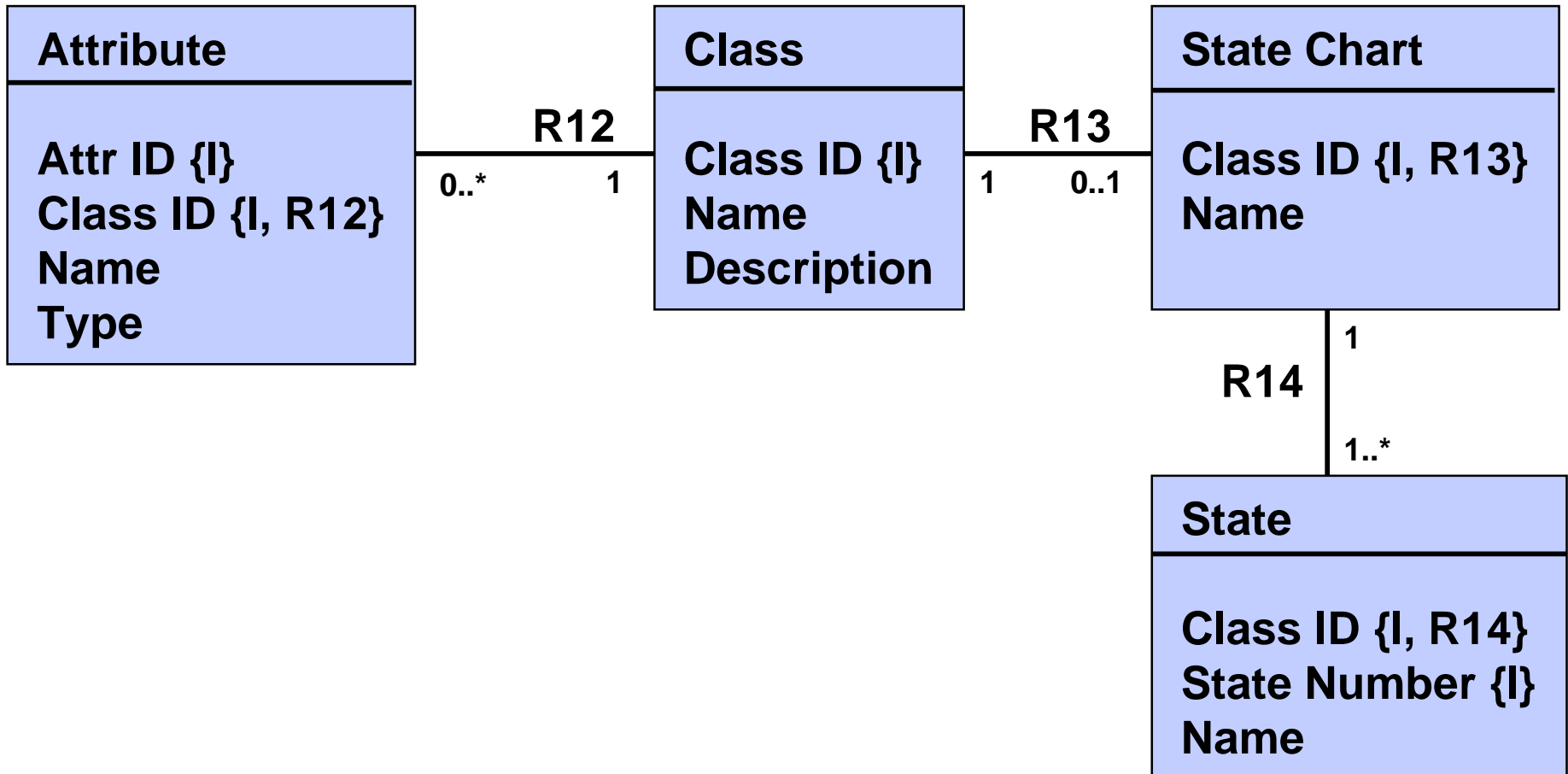
Capture the model in a model repository.

What is the structure of the repository?



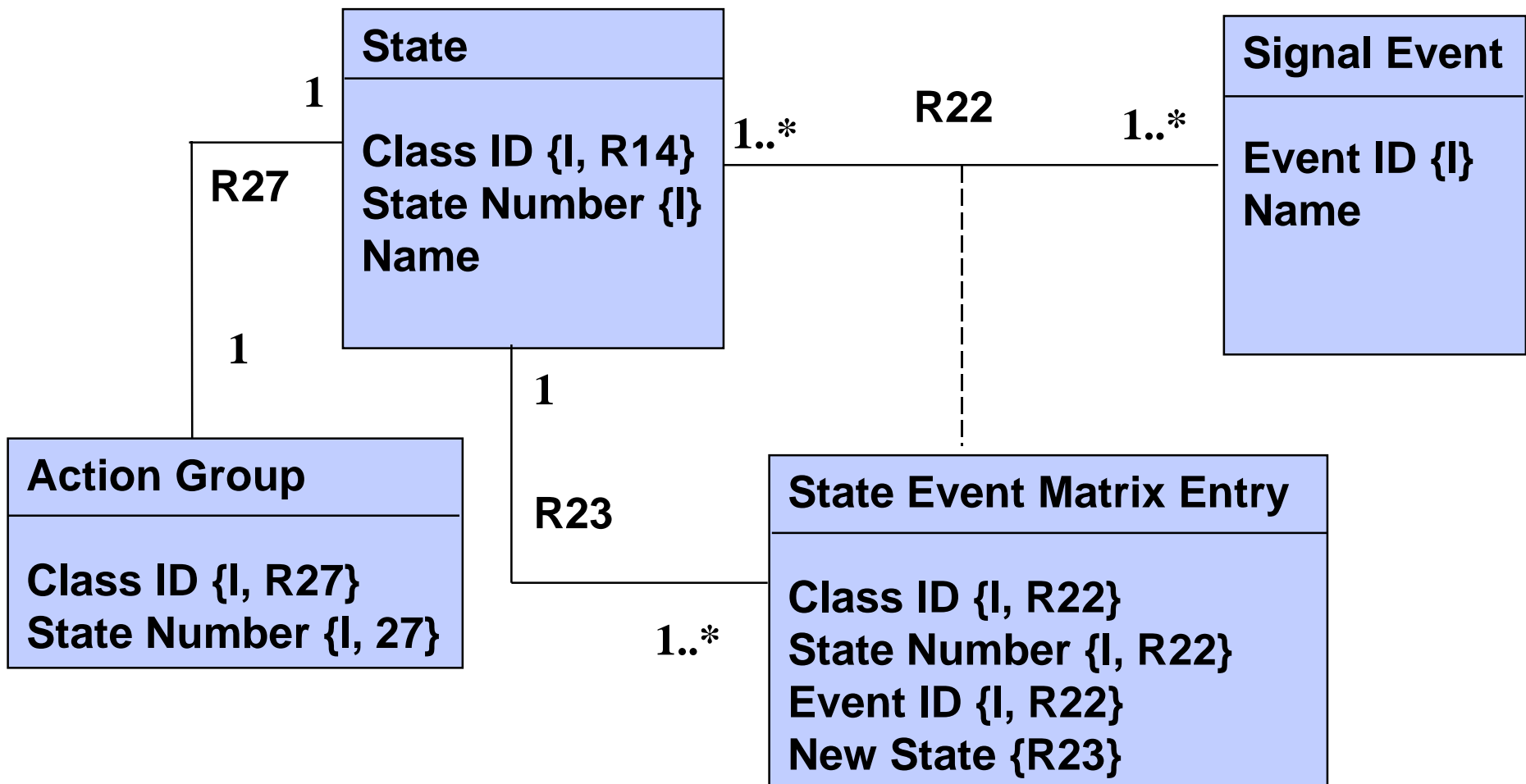
# Model Structure

*A meta-model* defines the structure of the repository.



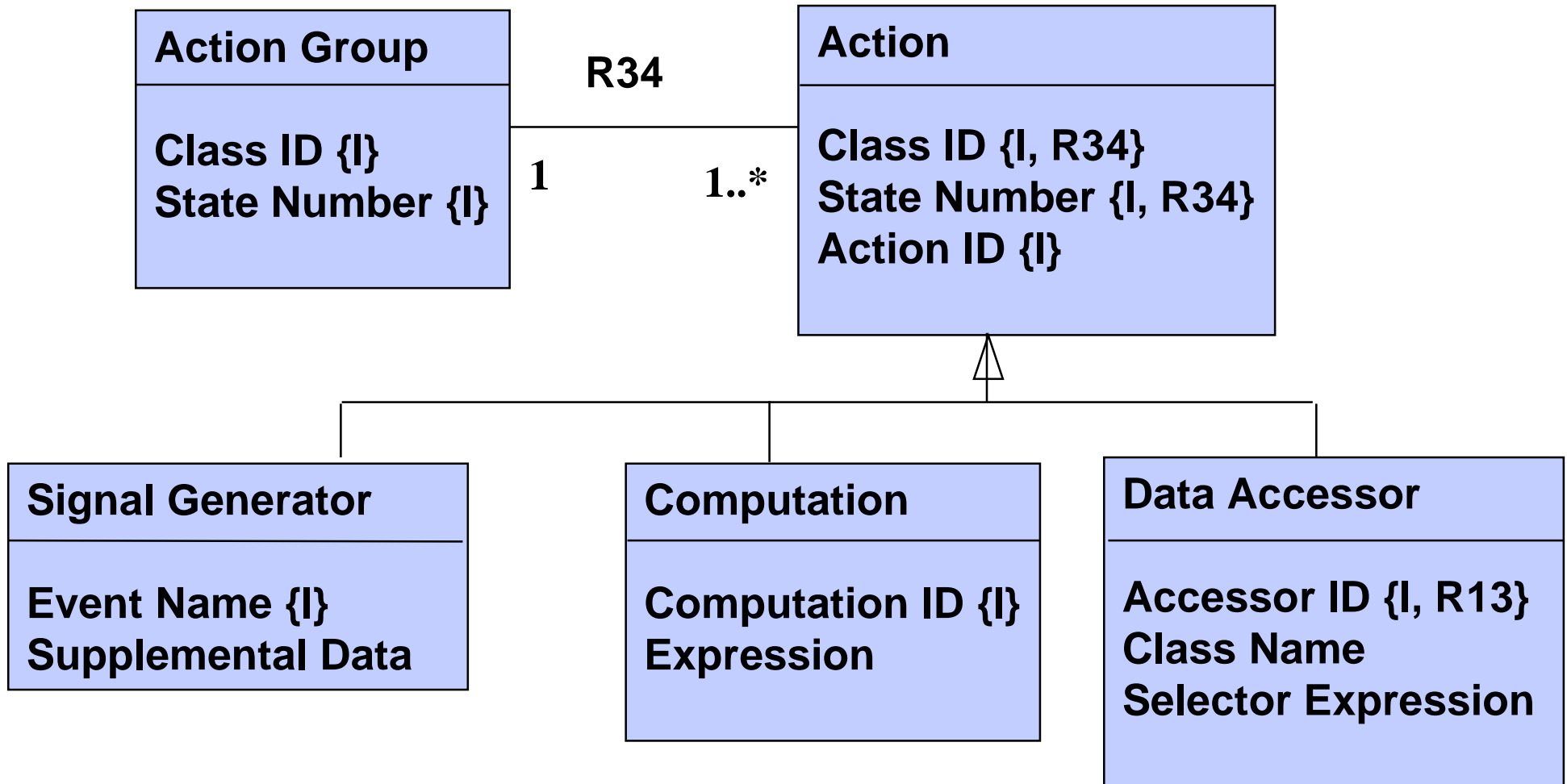
# Model Structure

*A meta-model defines the structure of the repository.*



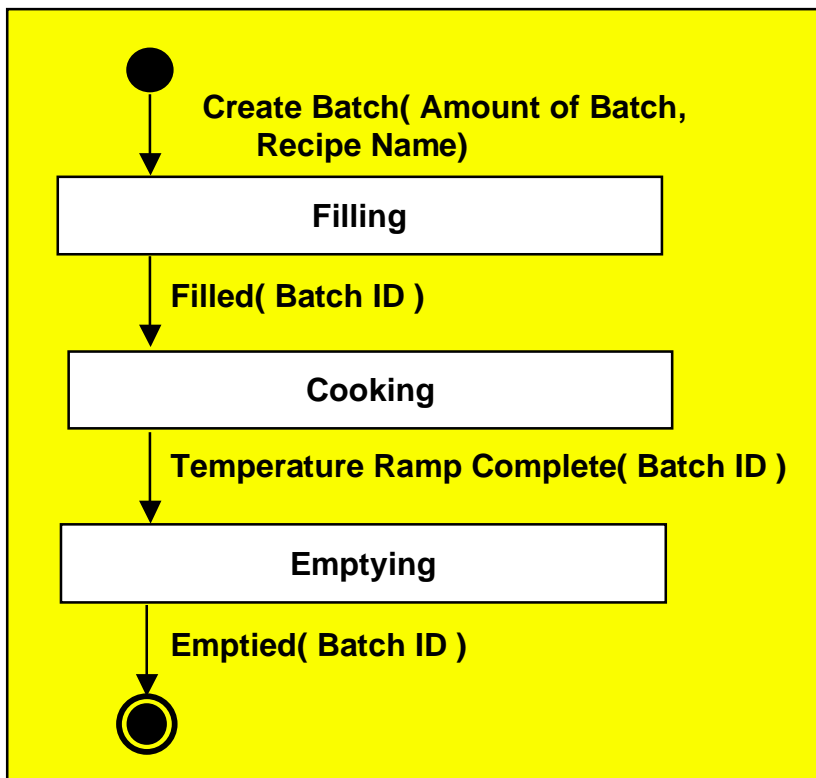
# Model Structure

*A meta-model* defines the structure of the repository.



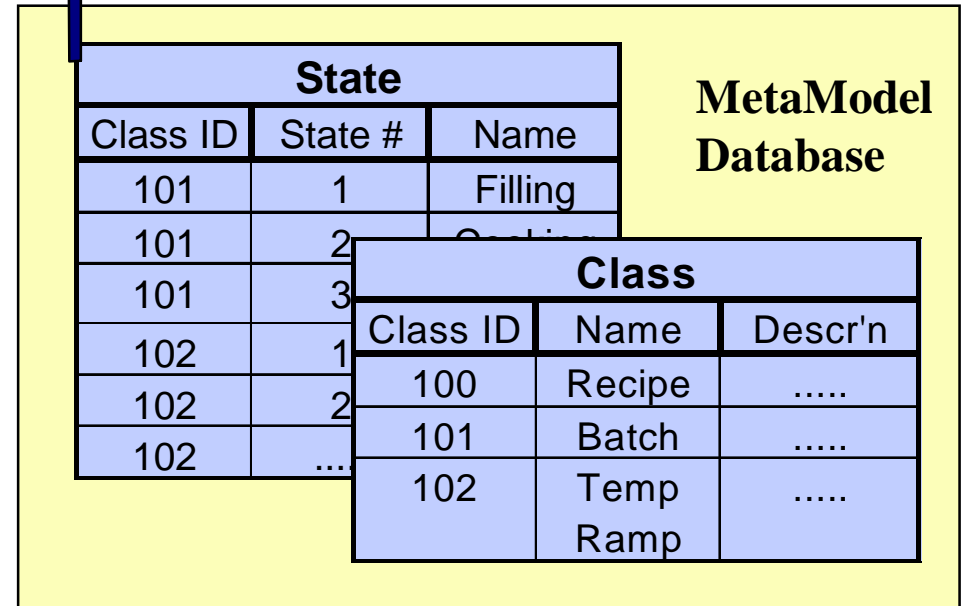
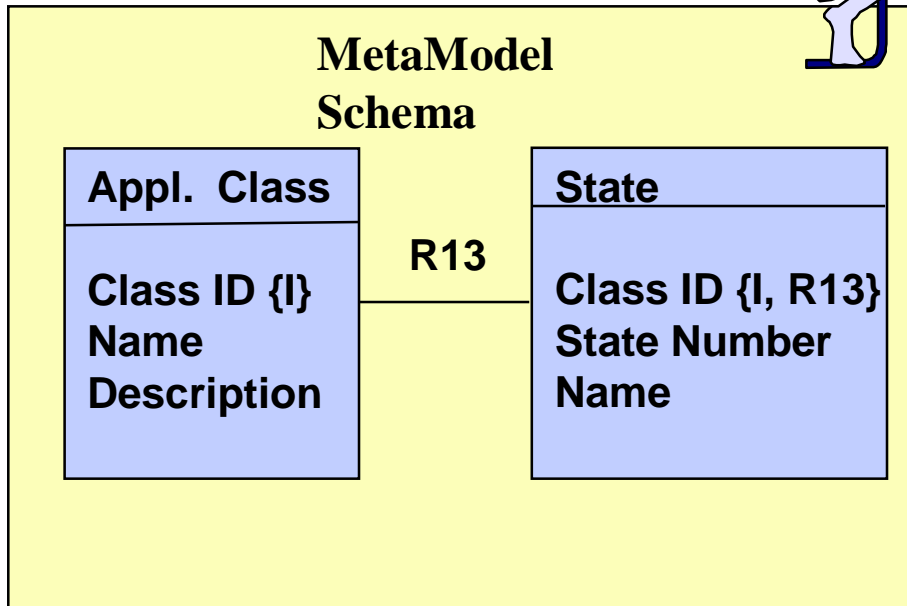
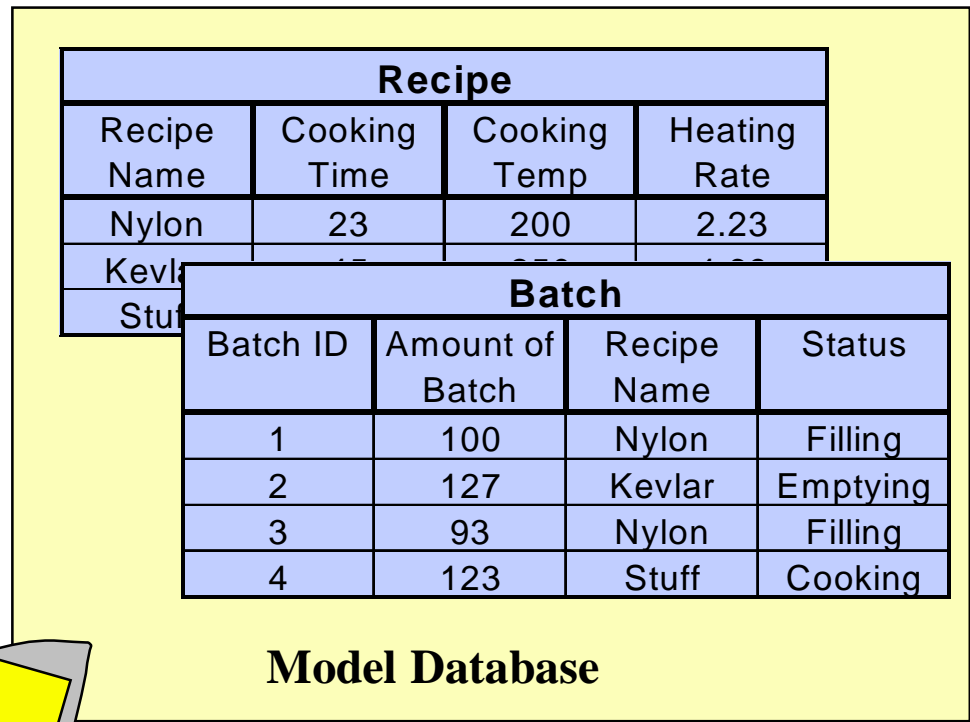
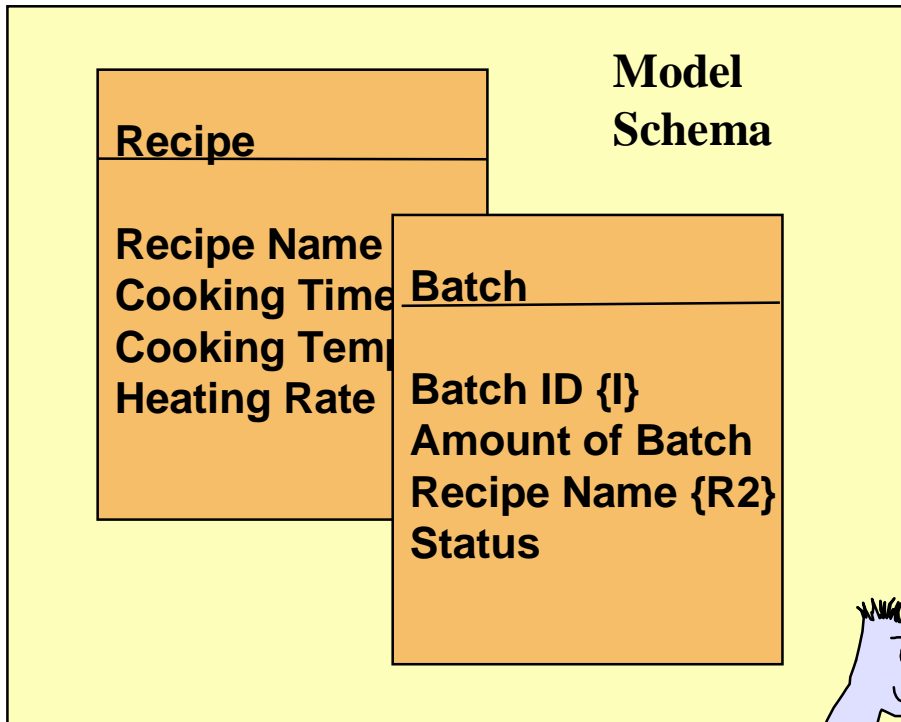
# Meta-Model Instances

Just like an application model, the meta-model has instances.

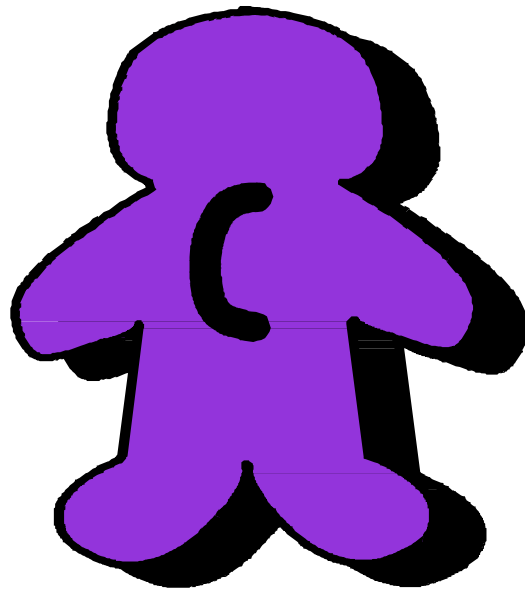


Class		
Class ID	Name	Descr'n
100	Recipe	.....
101	Batch	.....
102	Temp Ramp	.....

State		
Class ID	State #	Name
101	1	Filling
101	2	Cooking
101	3	Emptying
102	1	.....
102	2	.....
102	.....	.....



# Archetype Language



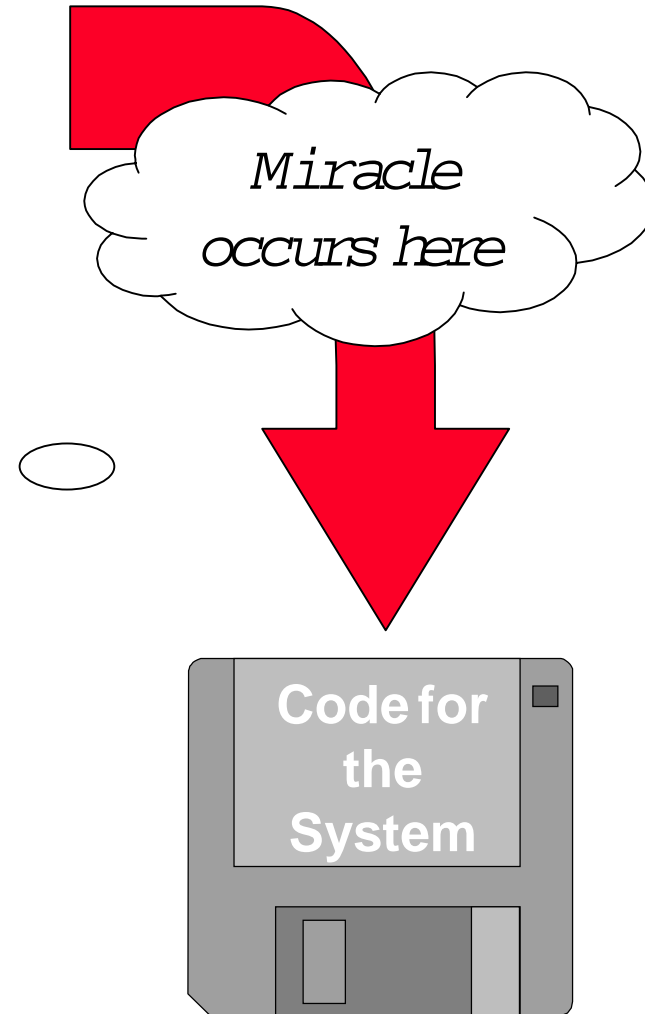
# Purpose

To generate code....

State		
Class ID	State #	Name
101	1	Filling
101	2	...
101	3	...

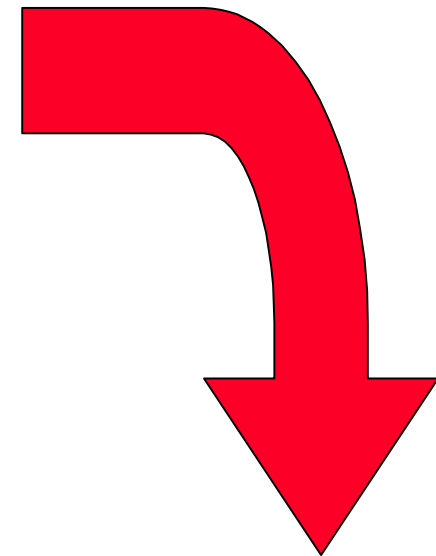
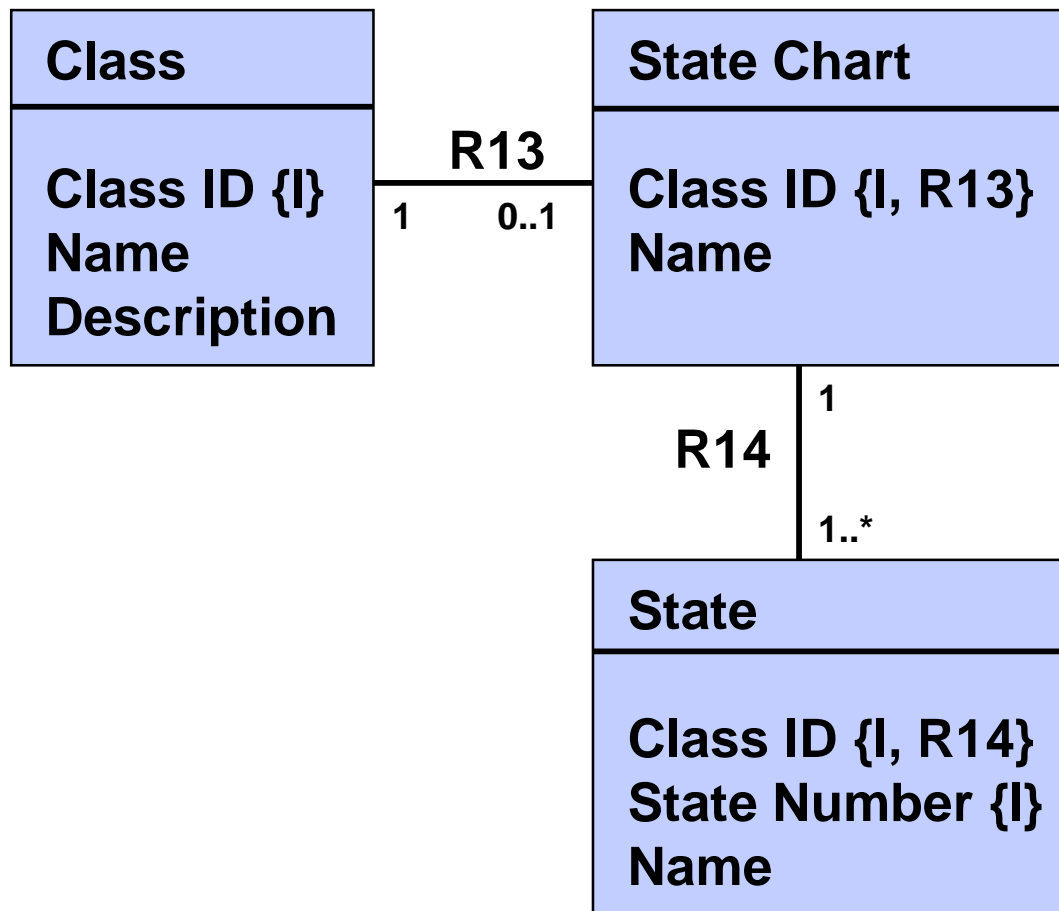
Class		
Class ID	Name	Descr'n
100	Recipe	.....
101	Batch	.....
102	Temp Ramp	.....

**MetaModel Database**



# Purpose

....traverse the repository and...



... output text.

# Example

The archetype language produces text.

```
.select many stateS related to instances of  
class->[R13]StateChart ->[R14]State  
where (isFinal == False)
```

```
public:
```

```
enum states_e
```

```
{ NO_STATE = 0 ,
```

```
.for each state in stateS
```

```
.if ( not last stateS )
```

```
  state.Name ,
```

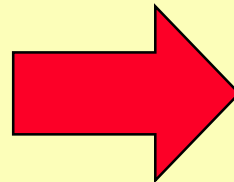
```
.else
```

```
  NUM_STATES = state.Name
```

```
.endif
```

```
.endfor
```

```
};
```



```
public:
```

```
enum states_e
```

```
{ NO_STATE = 0 ,
```

```
  Filling ,
```

```
  Cooking ,
```

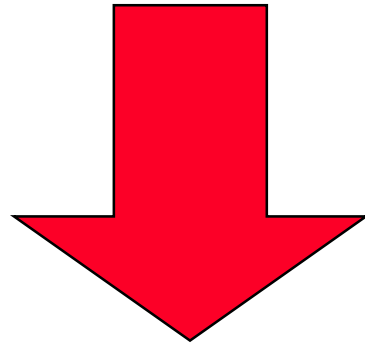
```
  NUM_STATES = Emptying
```

```
};
```

# Text

To generate text:

**The quick brown fox jumped over the lazy dog.**



The quick brown fox jumped over the lazy dog.

# Data Access

To select any instance from the repository:

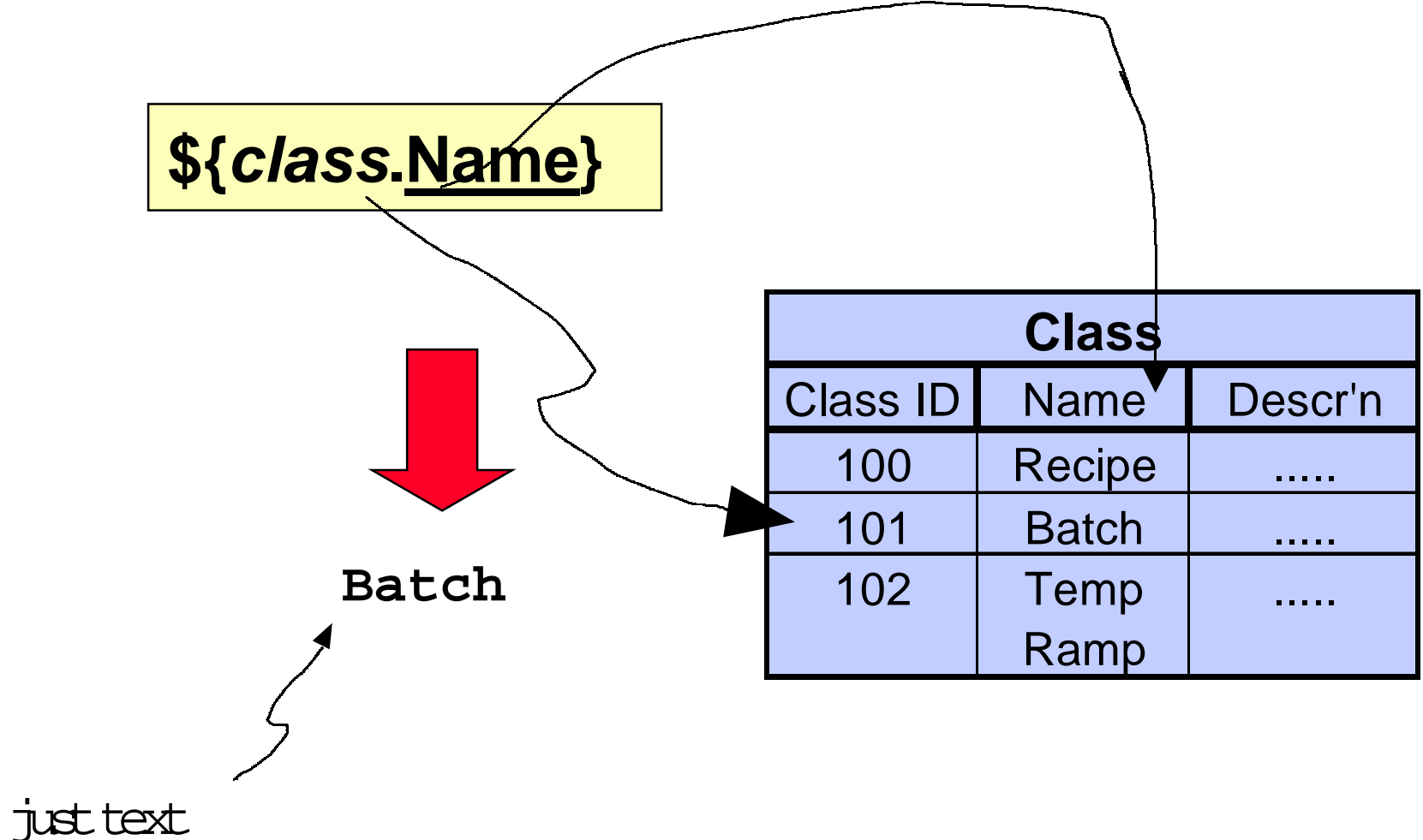
**.select any *class* from instances of Class**

Instance reference

Class		
Class ID	Name	Descr'n
100	Recipe	.....
101	Batch	.....
102	Temp Ramp	.....

# Substitution

To access attributes of the selected instance....

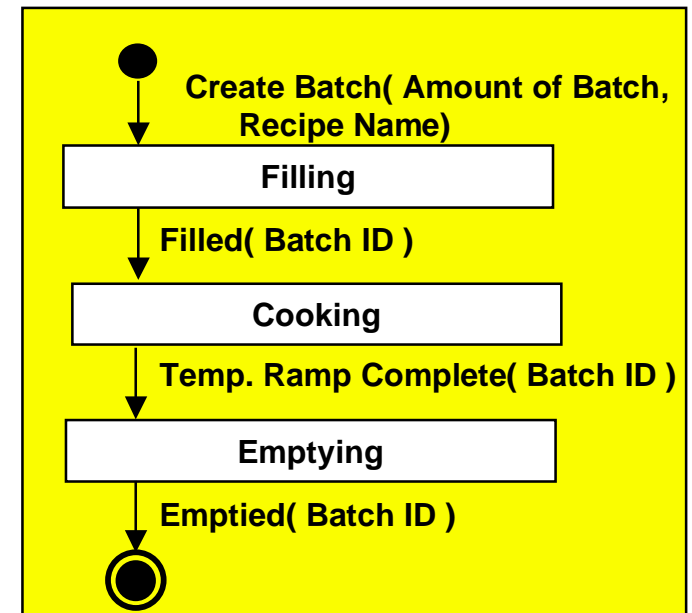
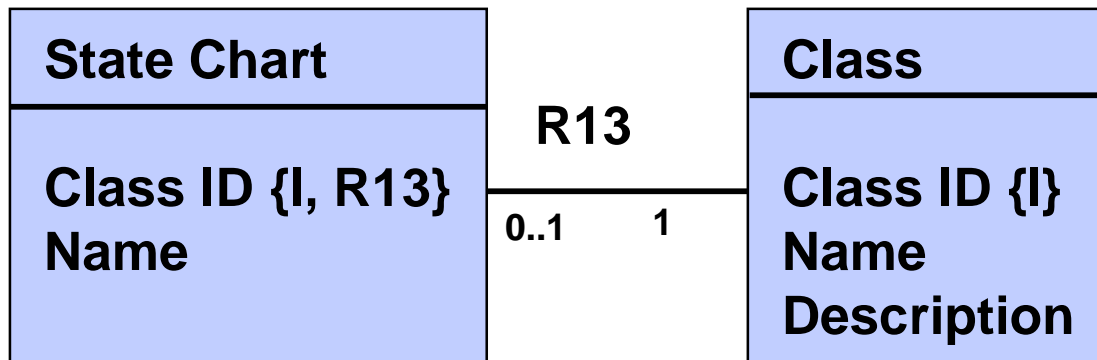


# Association Traversal

To traverse an association.....

Not just any one—  
the one that's associated

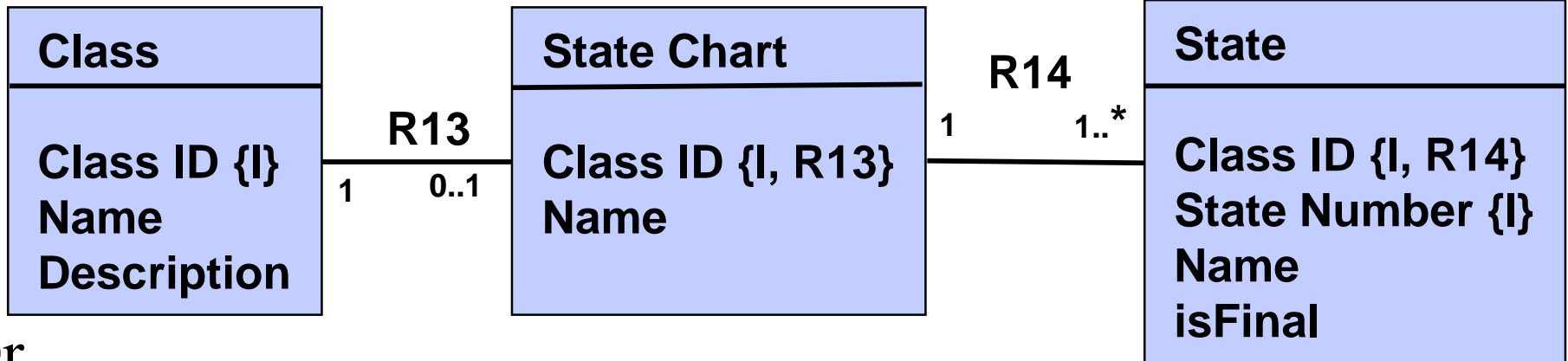
**.select one *StateChart* related to instances of *class*->[R13]StateChart**



# Arbitrary Instance

To select an arbitrary one....

**.select any *state* related to instances of *StateChart*->[R14]State**



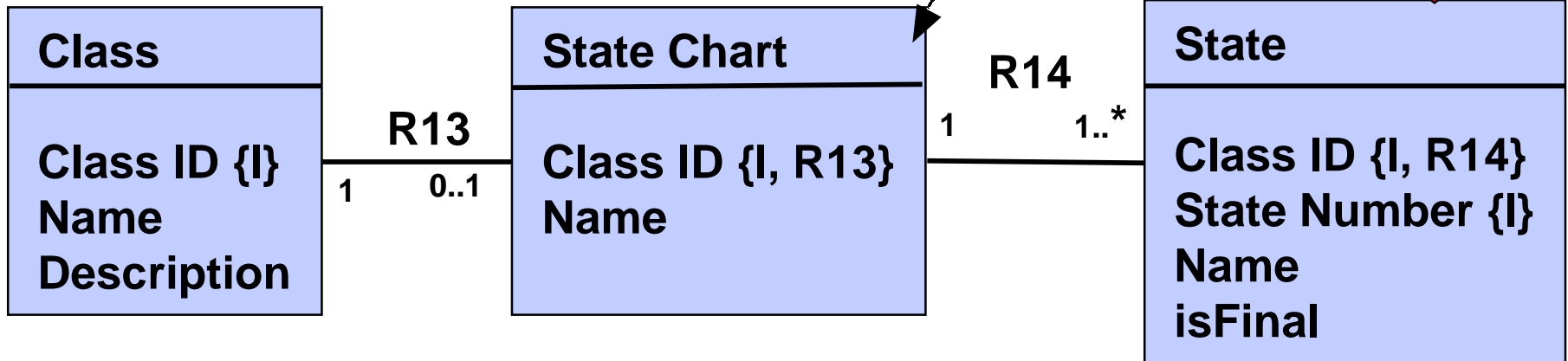
Or...

**.select any *state* related to instances of *Class*->[R13]StateChart->[R14]State**

# Complex Traversals

To qualify the selection...

**.select any *state* related to instances of *StateChart*->[R14]State  
where (isFinal == False)**

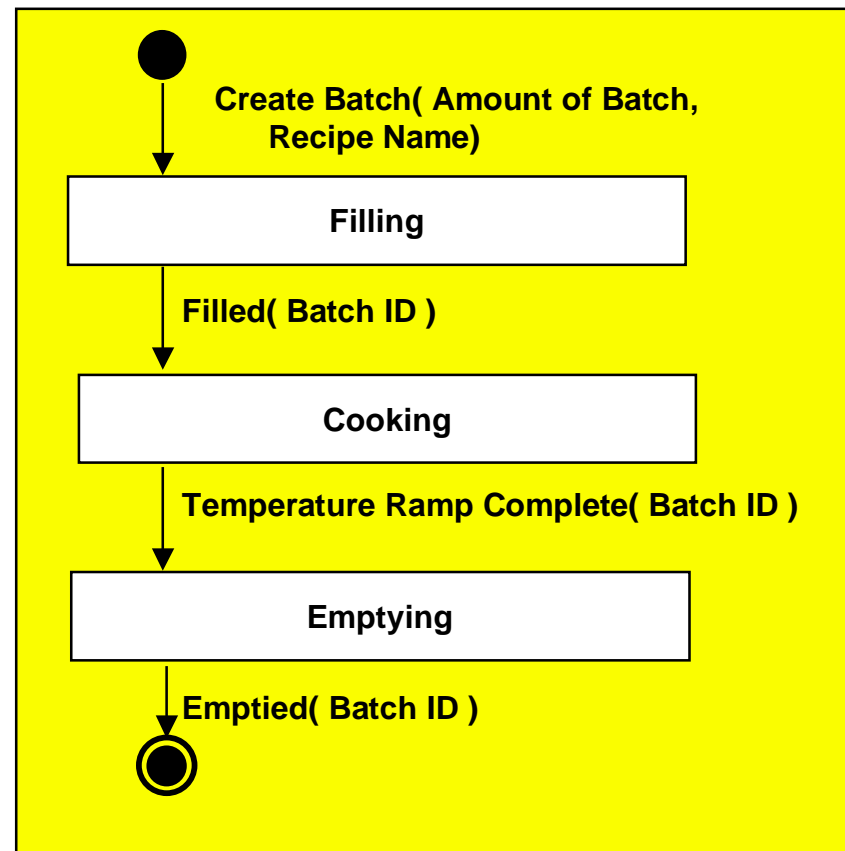
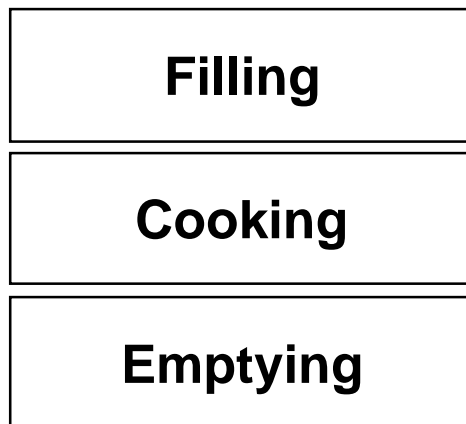


# Instance Sets

To select many instances:

**.select many *stateS* related to instances of *Class*->  
[R13]StateChart ->[R14]State where (isFinal==False)**

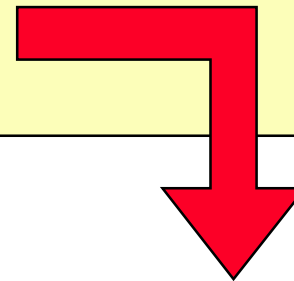
StateS =



# Iteration

To iterate over instances...

```
.select many stateS related to instances of  
  Class->[R13]StateChart -> [R14]State  
  where (isFinal == False)  
.for each state in stateS  
  state.Name ,  
.endfor
```



```
Filling,  
Cooking,  
Emptying,
```

# Putting It Together

We may combine these techniques....

```
.select many stateS related to instances of  
  class->[R13]StateChart->[R14] State  
  where (isFinal == False)
```

```
public:
```

```
  enum states_e
```

```
    { NO_STATE = 0 ,
```

```
  .for each state in statesS
```

```
    .if ( not last stateS )
```

```
      state.Name ,
```

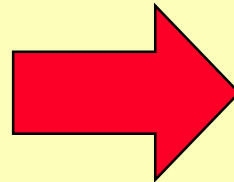
```
    .else
```

```
      NUM_STATES = state.Name
```

```
    .endif
```

```
  .endfor
```

```
};
```



```
public:
```

```
  enum states_e
```

```
    { NO_STATE = 0 ,
```

```
      Filling ,
```

```
      Cooking ,
```

```
      NUM_STATES = Emptying
```

```
    };
```

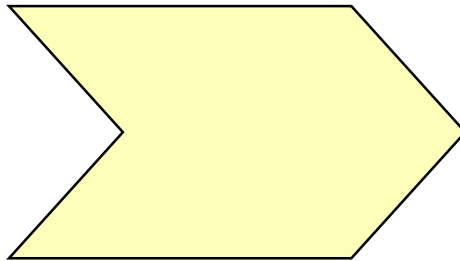
# Application Semantics

An archetype language gives access to

- ◆ the semantics of the application
- ◆ as stored in the repository.

We may use the archetype language to generate code.

**A**  
**Direct**  
**Translation**



# Application Classes

Each application class becomes an implementation class.

```
.select many classES from instances of class  
.for each class in classES  
class `${class.Name} : public ActiveInstance {  
  .invoke addPDMDDecl( inst_ref class)  
  ...  
};  
.endfor
```

# Application Attributes

Each attribute becomes a private data member:

```
.function addPDMDecl( inst_ref class )  
private:  
  .select many attrS related to class->[R12]Attribute  
  .for each attr in attrS  
    attr.Type {attr.Name} ;  
  .endfor  
.end function
```

# State Chart Declaration

To declare a state chart: (i.e. all the actions in the state chart)

```
.function addProtectedActions( inst_ref class )  
.select one statechart related by class-> [R13]StateChart  
protected:  
// state action member functions  
  .select many stateS related by statechart->[R14]State  
  .for each state in stateS  
    .invoke addActionFunctionDecl( inst_ref state )  
  .endfor  
.end function
```

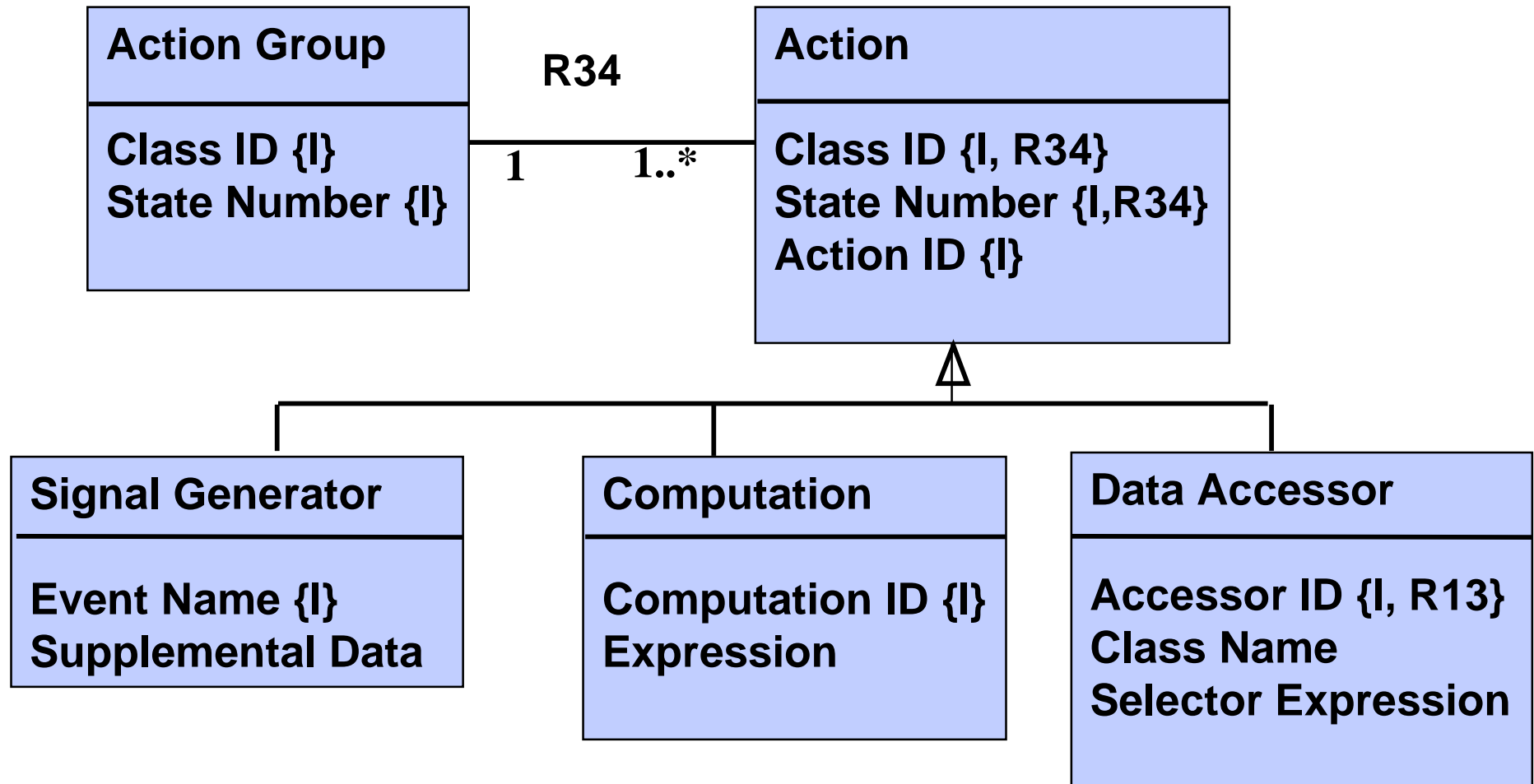
# State Action Declaration

To generate the state action declaration:

```
.function addActionFunctionDecl( inst_ref state )  
// State action: state.Name  
static void sAsyncAction $\{state.Name\}$ (  
    stda_eventMsg_c *eventPtr, int nextState);  
    void  $\{state.Name\}$ (stda_eventMsg_c *p_evt );  
void asyncAction $\{state.Name\}$  ( );  
.endfor
```

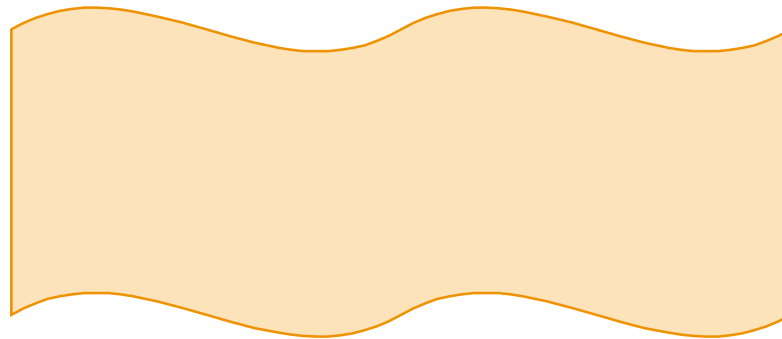
# State Action Definition

To *define* the state action function...



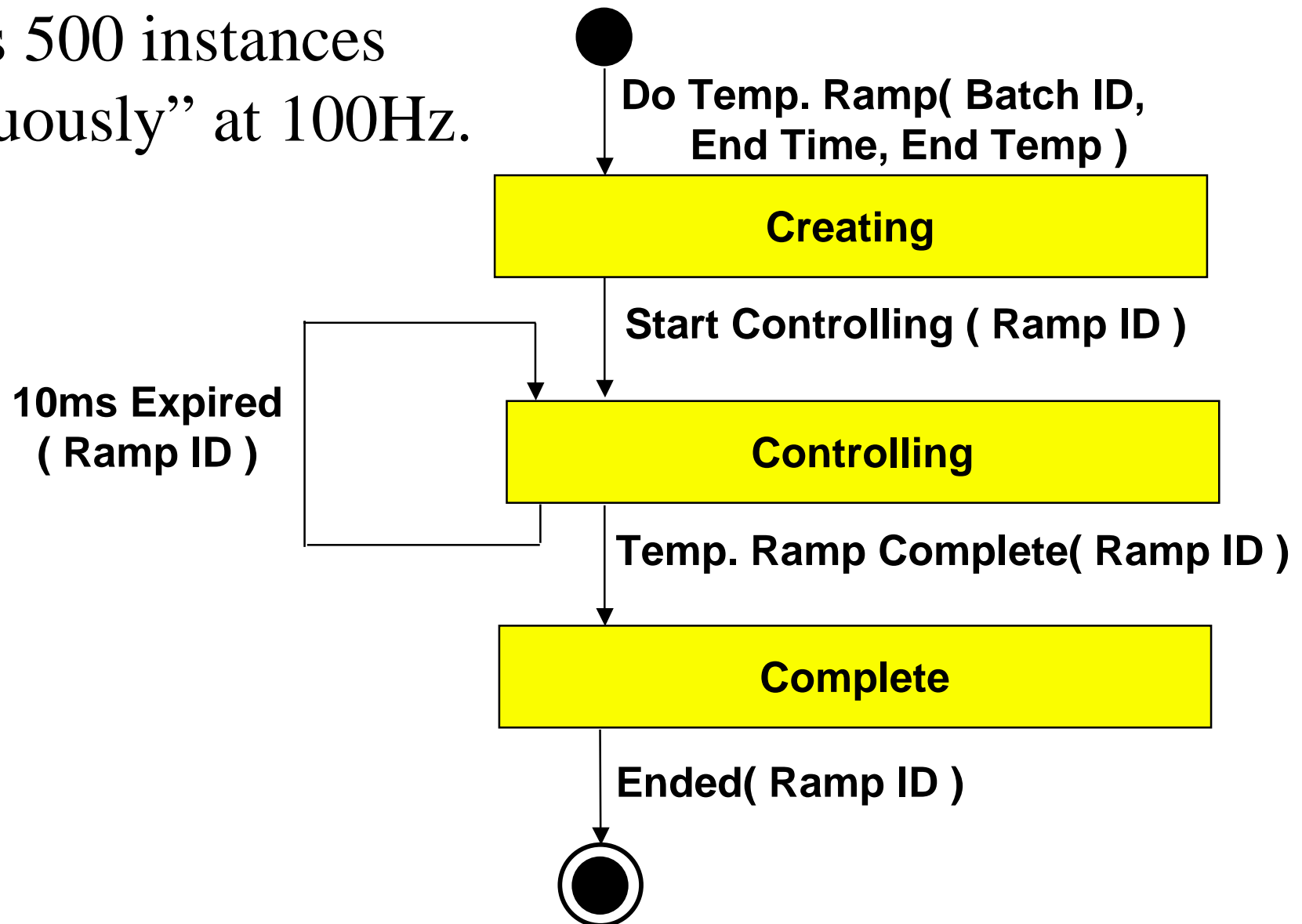
...traverse the repository in the same manner.

# An Indirect Architecture



# Where Have All The Cycles Gone?

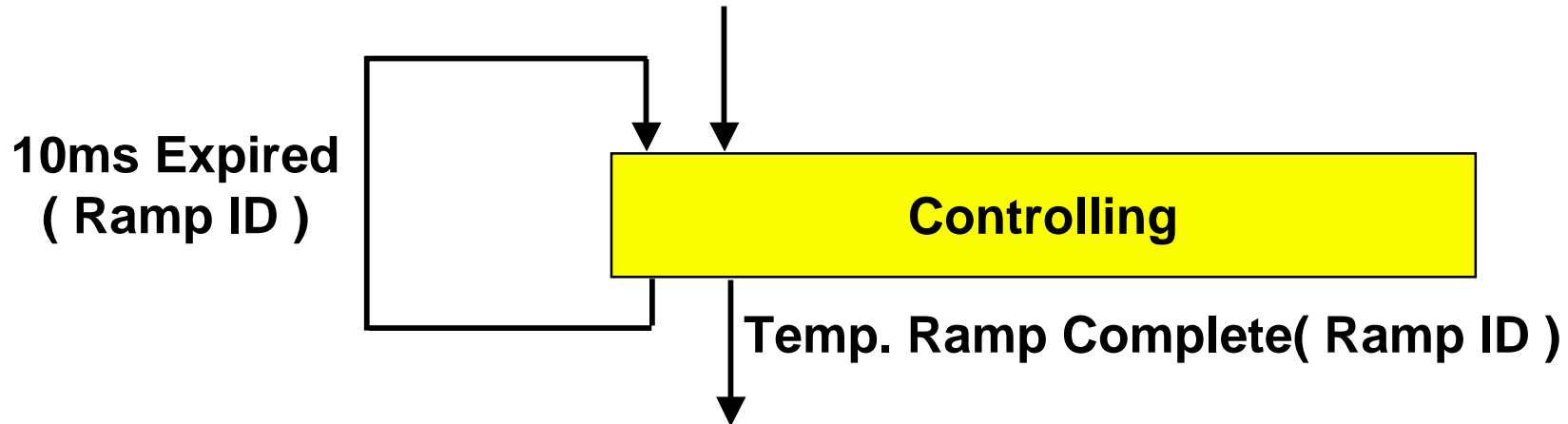
Consider a system that controls 500 instances “continuously” at 100Hz.



# Where Have All The Cycles Gone?

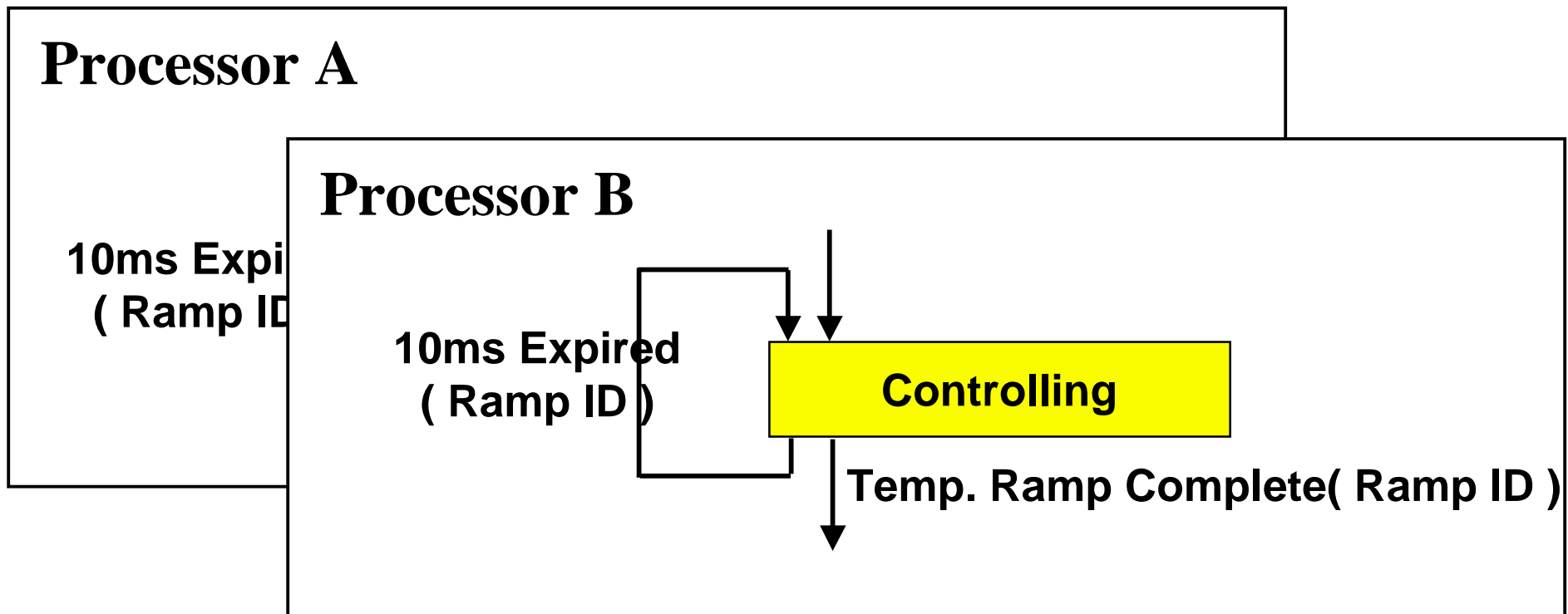
Model the controller so that each :

- ◆ 500 instances \* 100 cycles transitions per second, or
- ◆ 20 microseconds per transition including the actions



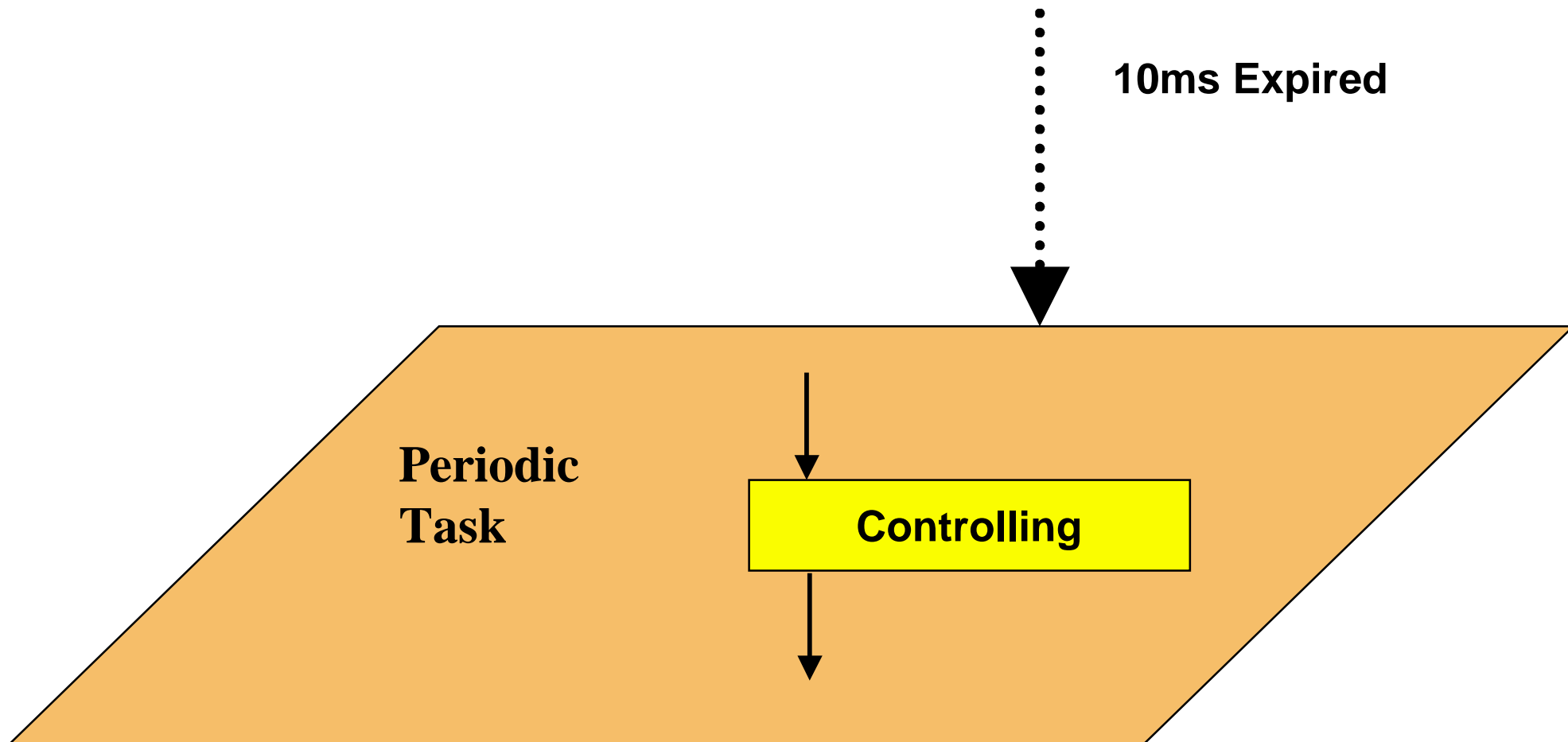
# Where Have All The Cycles Gone?

Or, we could buy several cheaper processors, each controlling different instances.



# Where Have All The Cycles Gone?

Or, we could use a single task that executes periodically.

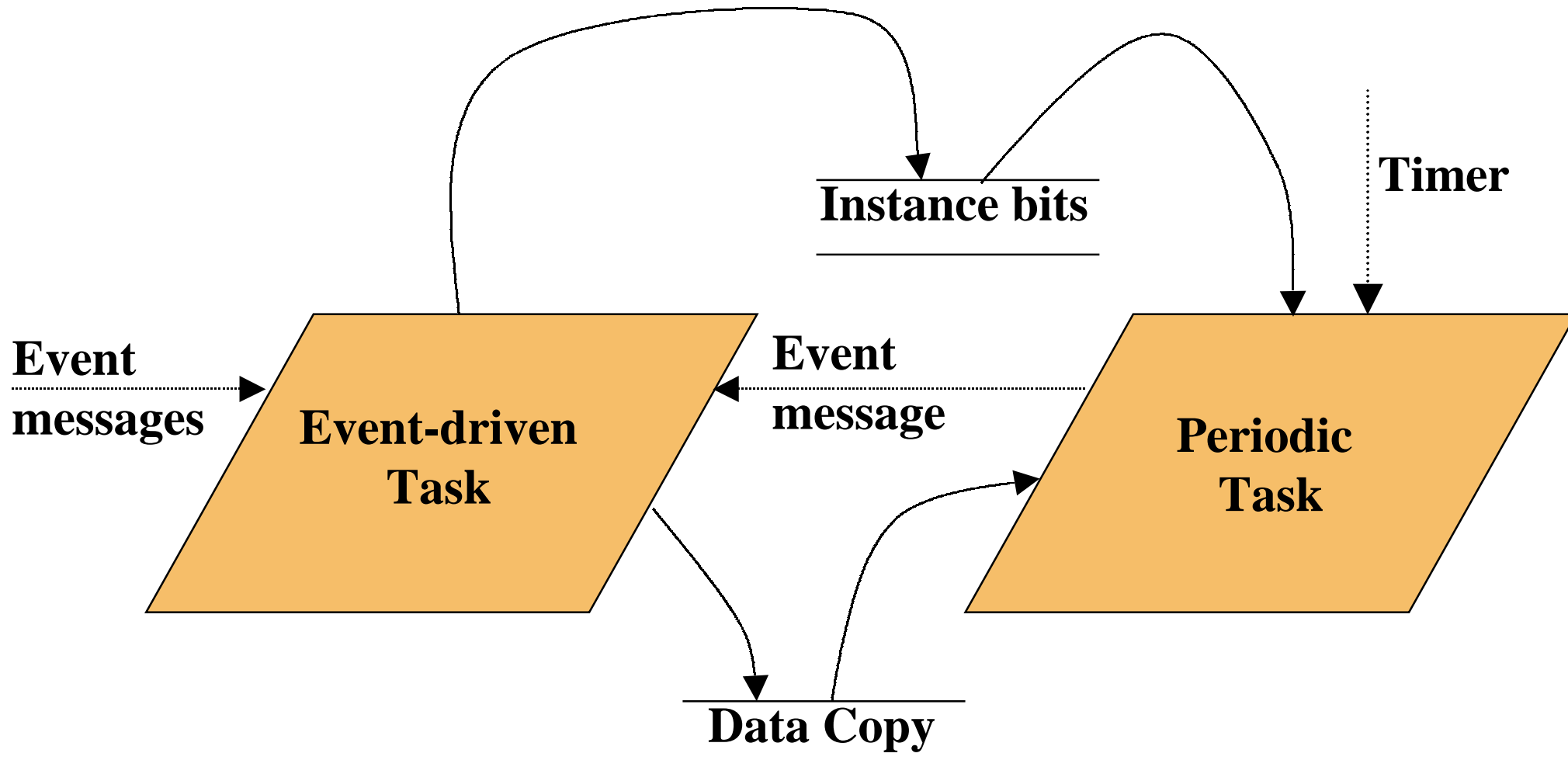


# Description of Architecture

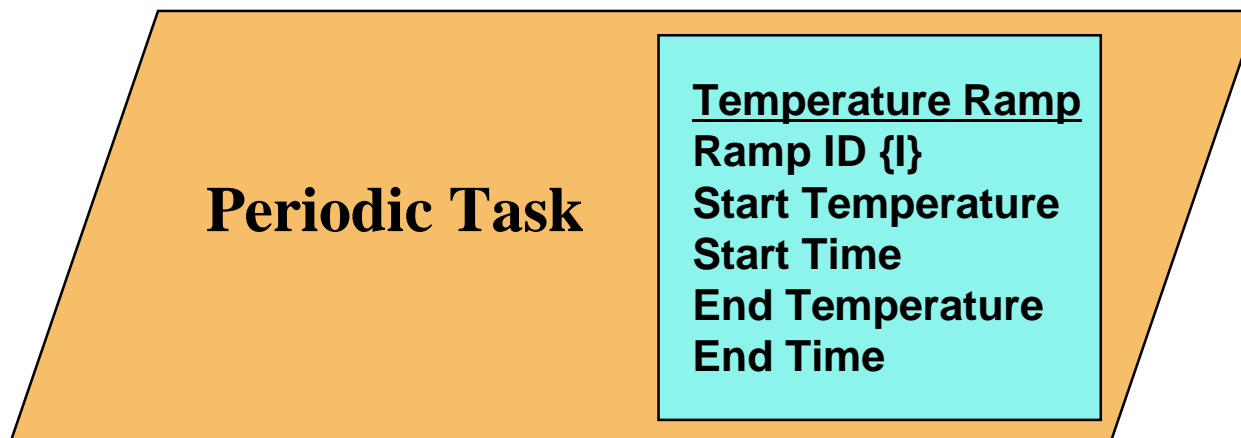
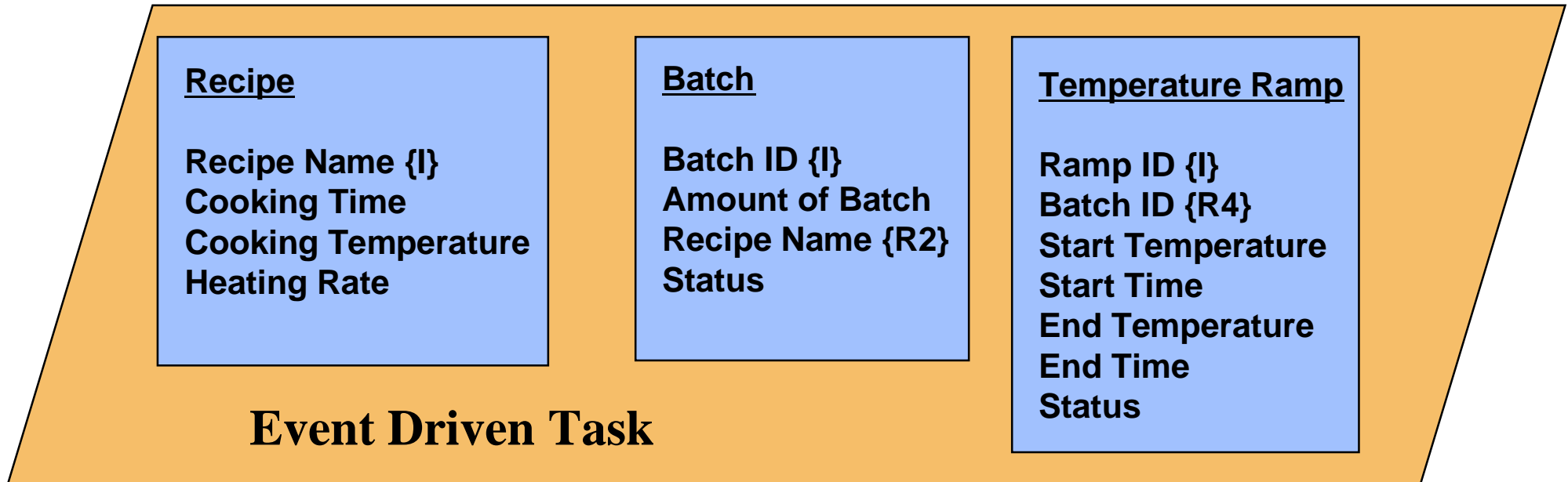
This last architecture introduces new issues:

- ◆ how to indicate transition into the periodic state (use one bit per instance), and
- ◆ how to indicate transition out of the periodic state
  
- ◆ the periodic task has to be able to execute when “it’s time,” so it needs higher priority----
- ◆ which raises issues of inconsistent data, so we should duplicate data needed for the control loop----
- ◆ and copy it over by the periodic task when required

# Description of Architecture



# Application Mapping



---

Start Temperature  
Start Time  
End Temperature  
End Time

---

---

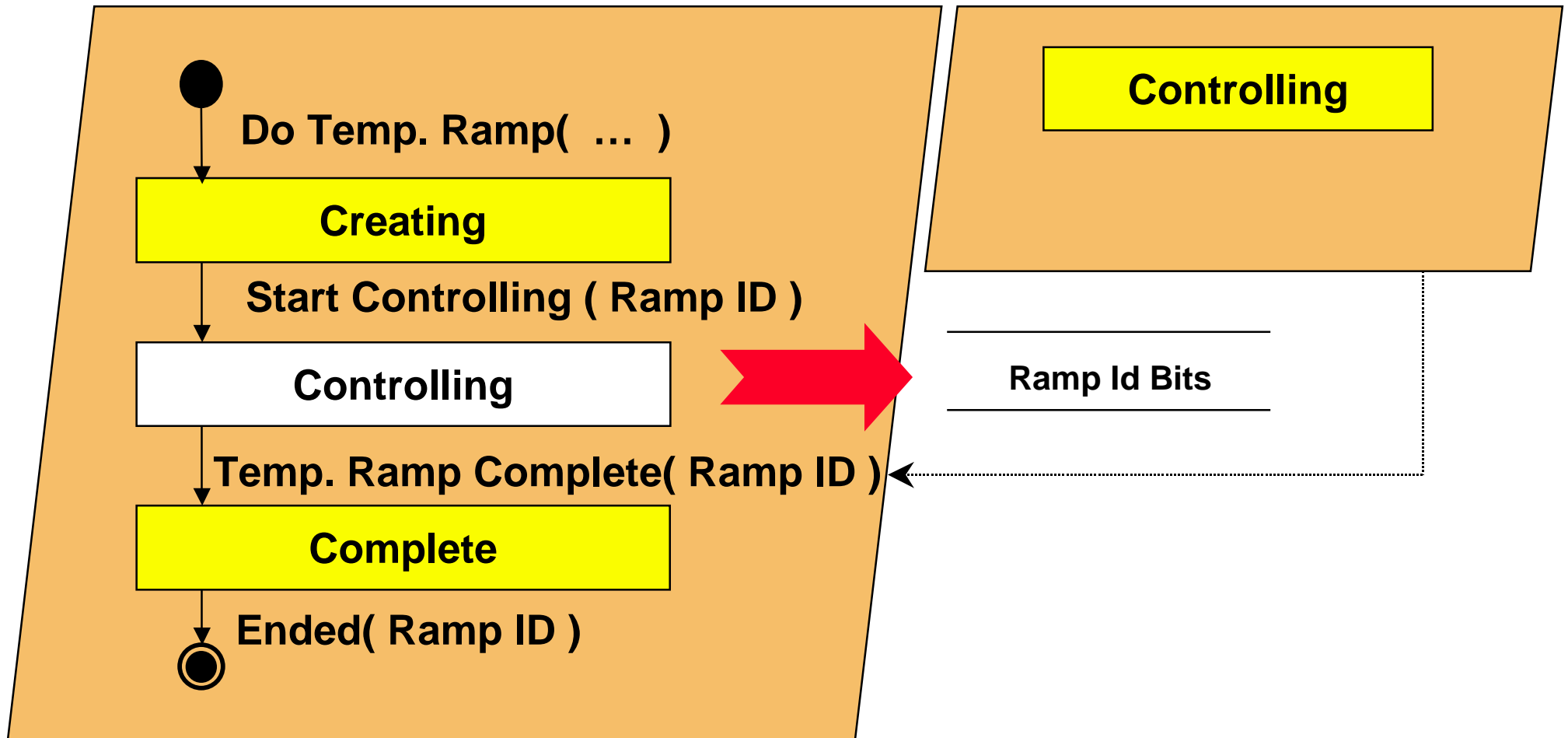
Ramp Id Bits

---

# Application Mapping

## Event Driven Task

## Periodic Task



# Extended Properties

To make certain distinctions, we need to tag elements of the meta-model.

```
.function addPeriodicStateAction  
...  
RampIDbits[insNumber].activateActions();
```

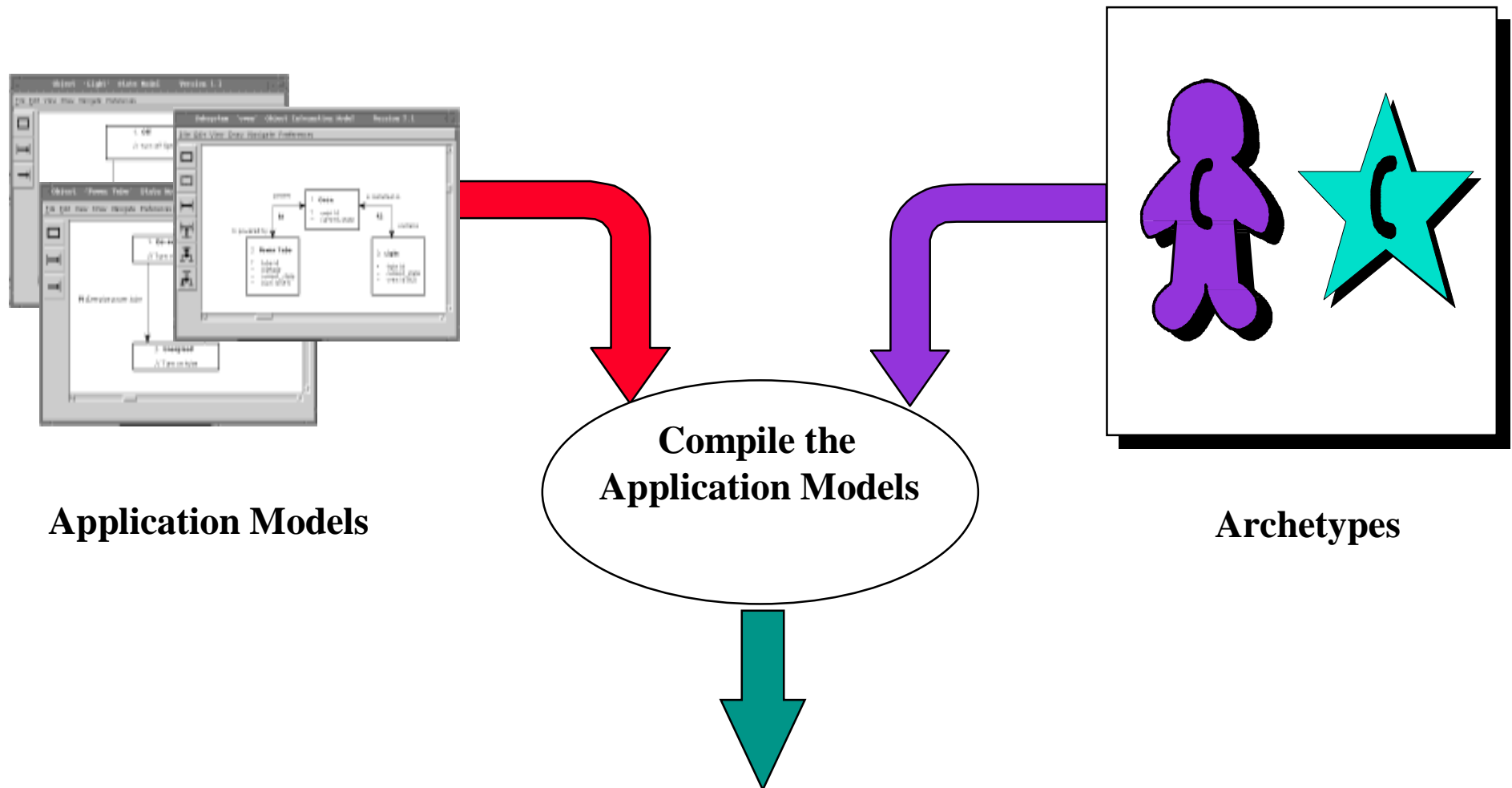
State
<b>Class ID {I, R14}</b>
<b>State Number {I}</b>
<b>Name</b>
<b>isFinal</b>
<b><i>isPeriodic</i></b>

# Model-Driven Architecture



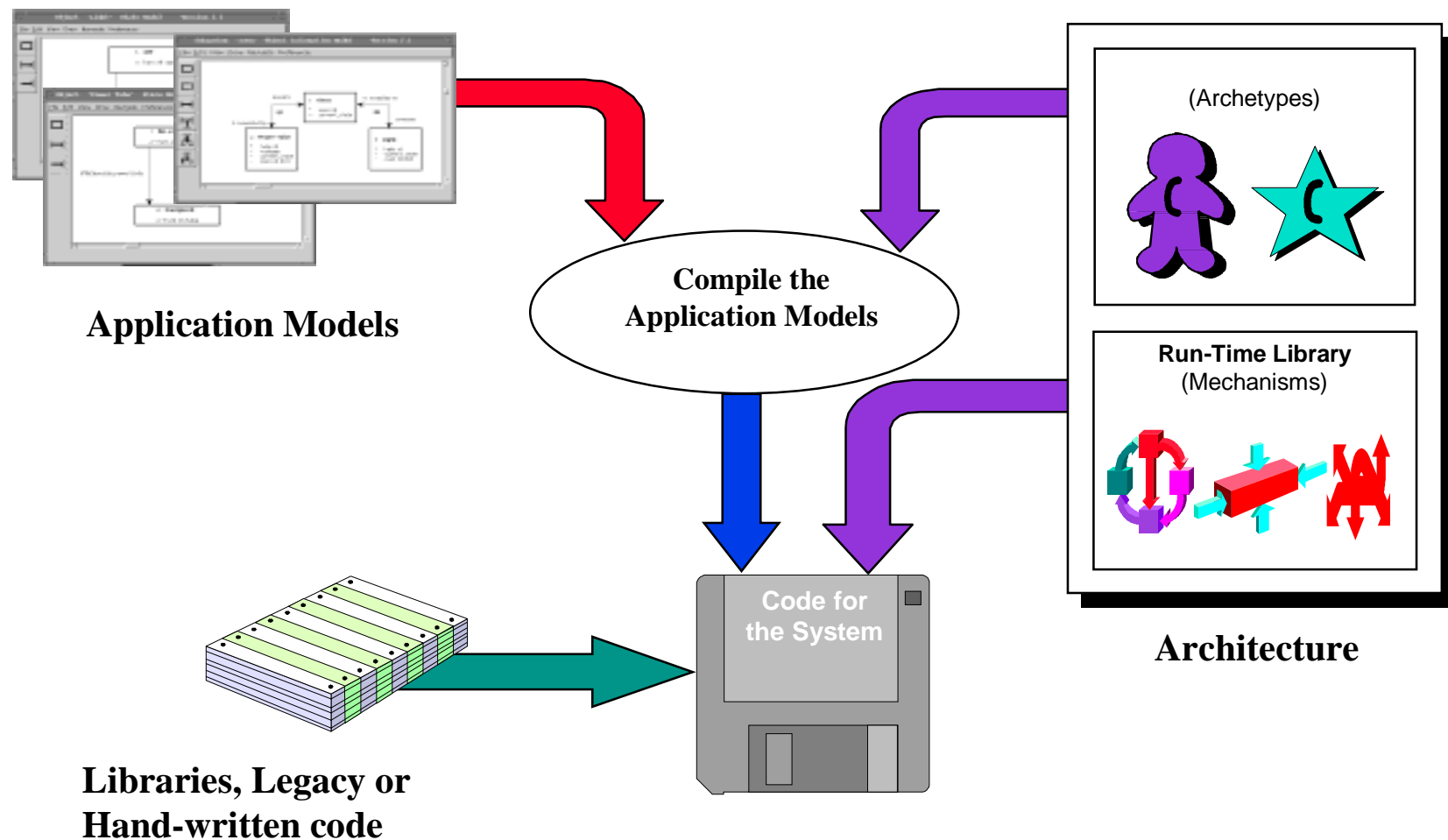
# Generating the Production Code

Invoke the archetypes and iterate over instances of the corresponding architecture objects to generate the complete source code for the system.



# Production Code

Compile the source code and include initialization data files (if any) to generate the deliverable production code.

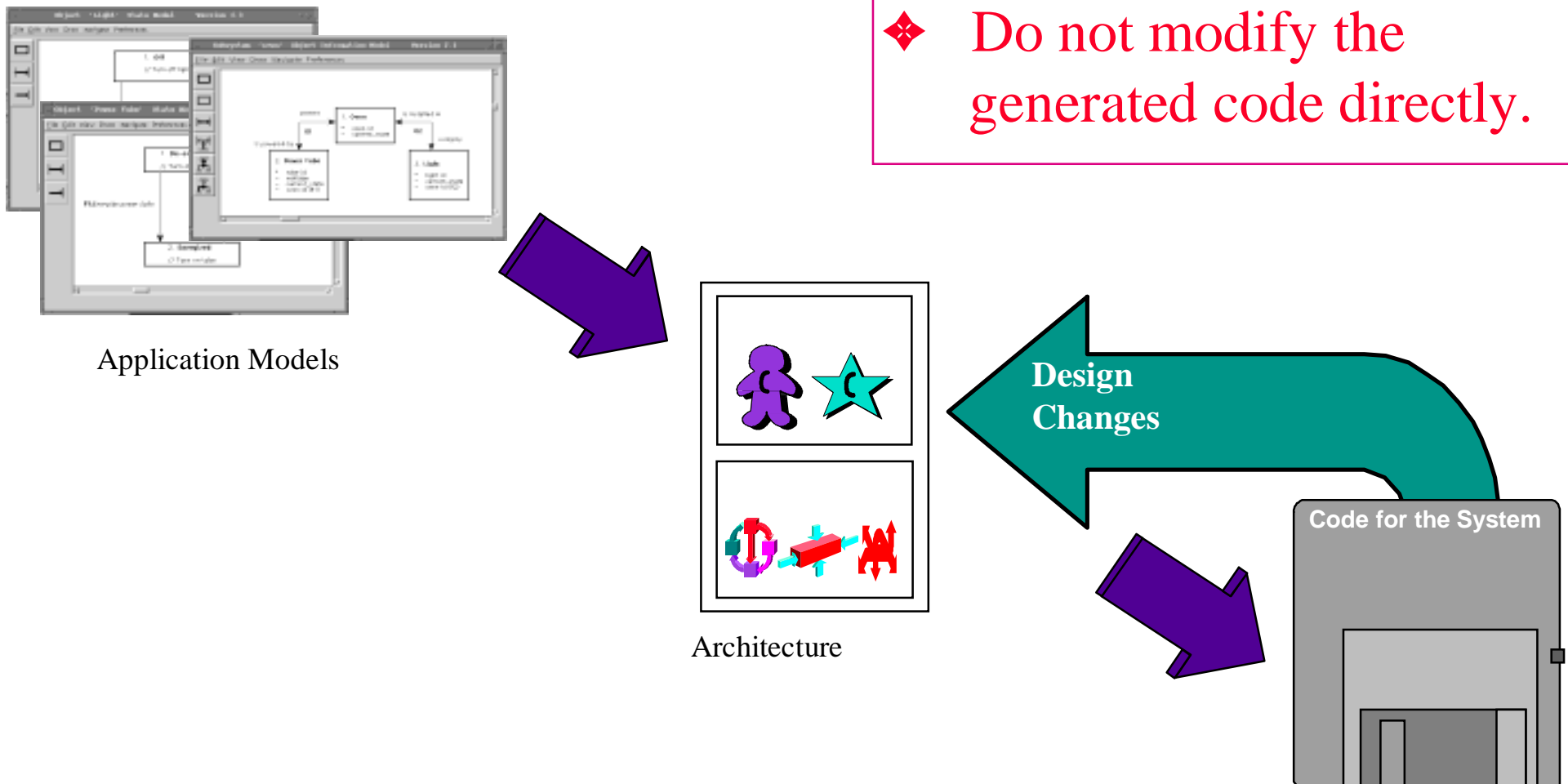


# Model-Based Maintenance

To address performance-based issues:

- ◆ modify the architecture models, and
- ◆ and regenerate the system.

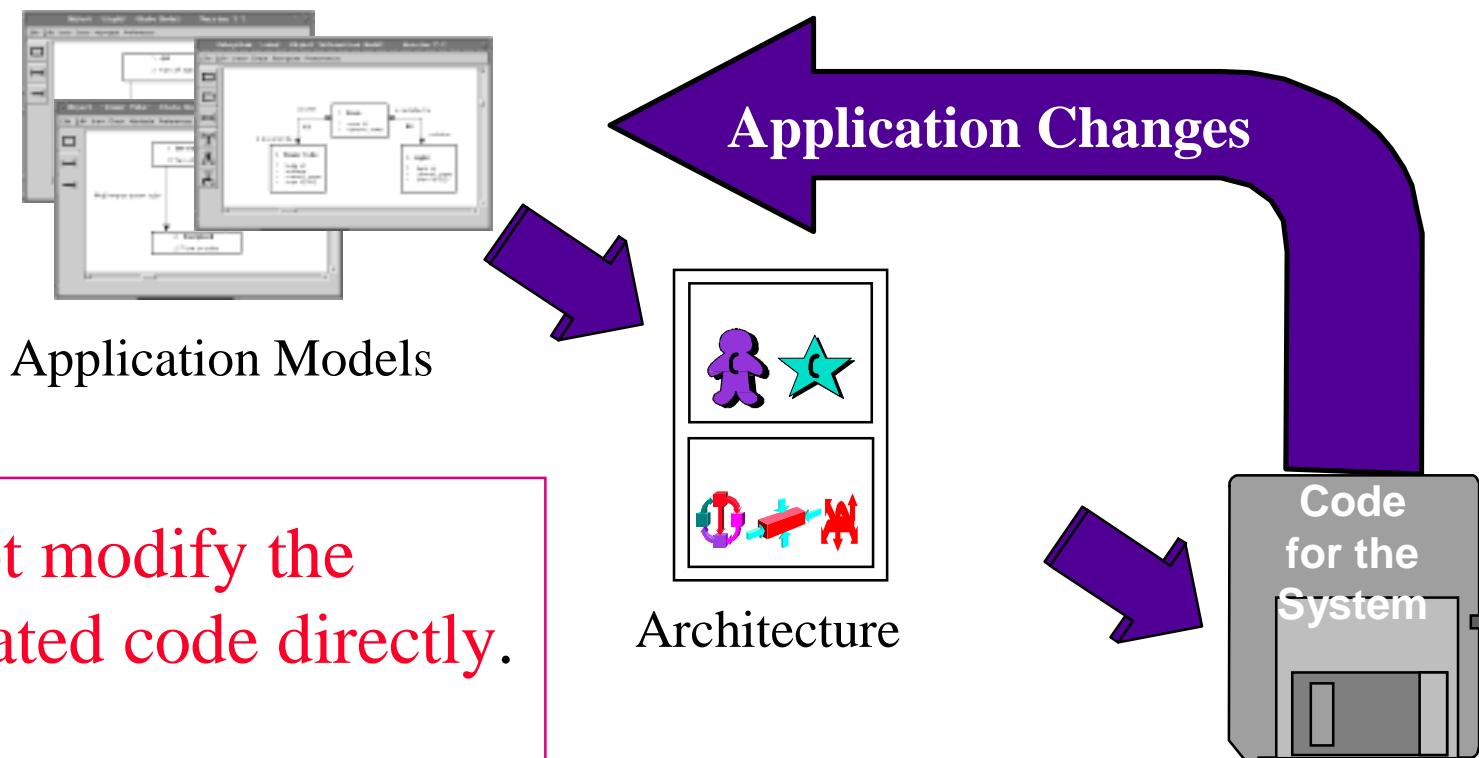
◆ Do not modify the generated code directly.



# Model-Based Maintenance

To address application behavior issues,

- ◆ modify the relevant application model, and
- ◆ regenerate the system.



**Do not modify the generated code directly.**

# Model Compiler

An architecture is an *Executable UML model compiler*.

It translates a system specified in executable UML into the target programming language incorporating decisions made by the architect about:

- ◆ data,
- ◆ control,
- ◆ structures, and
- ◆ time.

**Architectures, like programming language compilers, can be bought.**

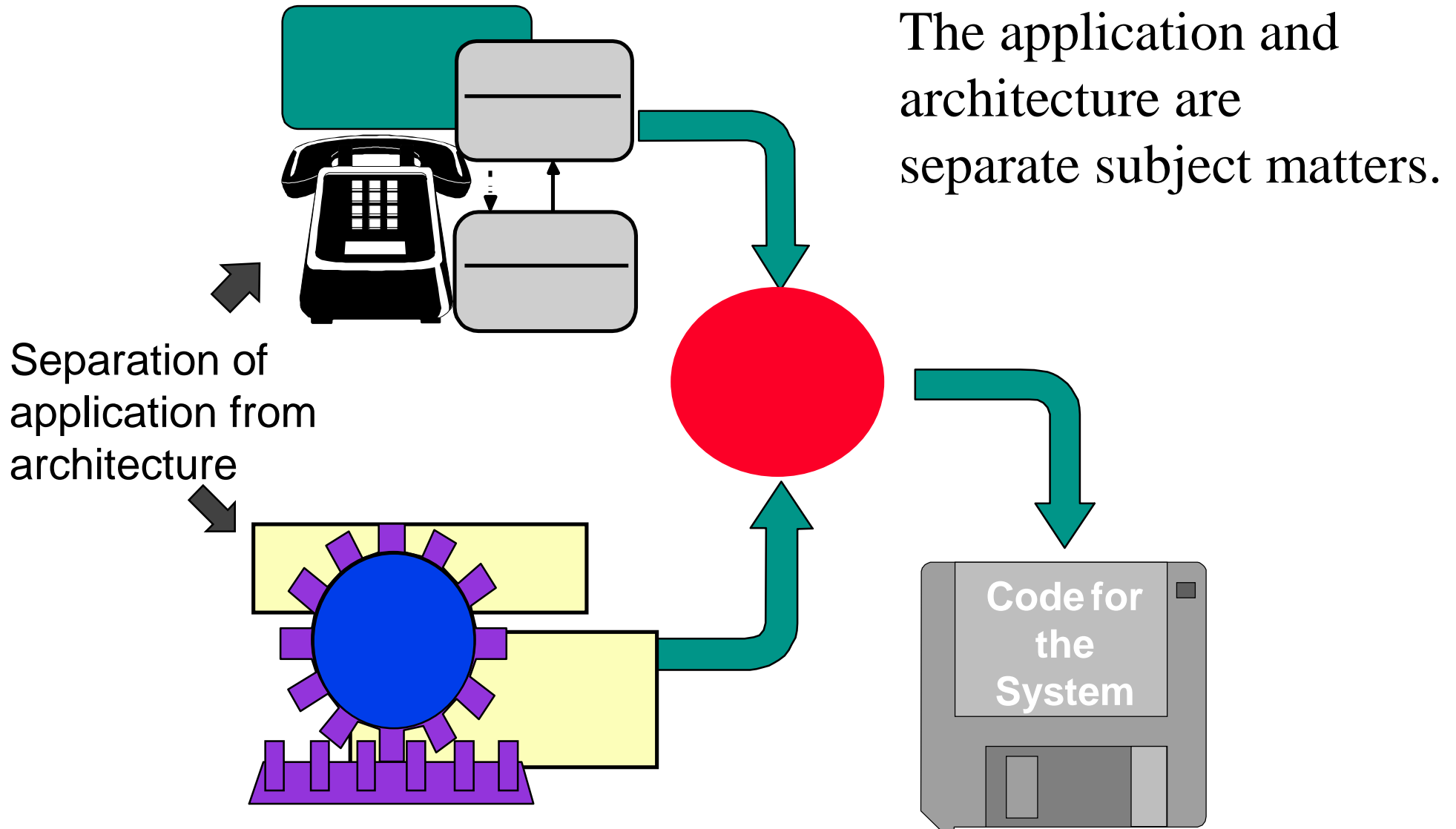
# Translative System Generation

PROJECT TECHNOLOGY, INC.

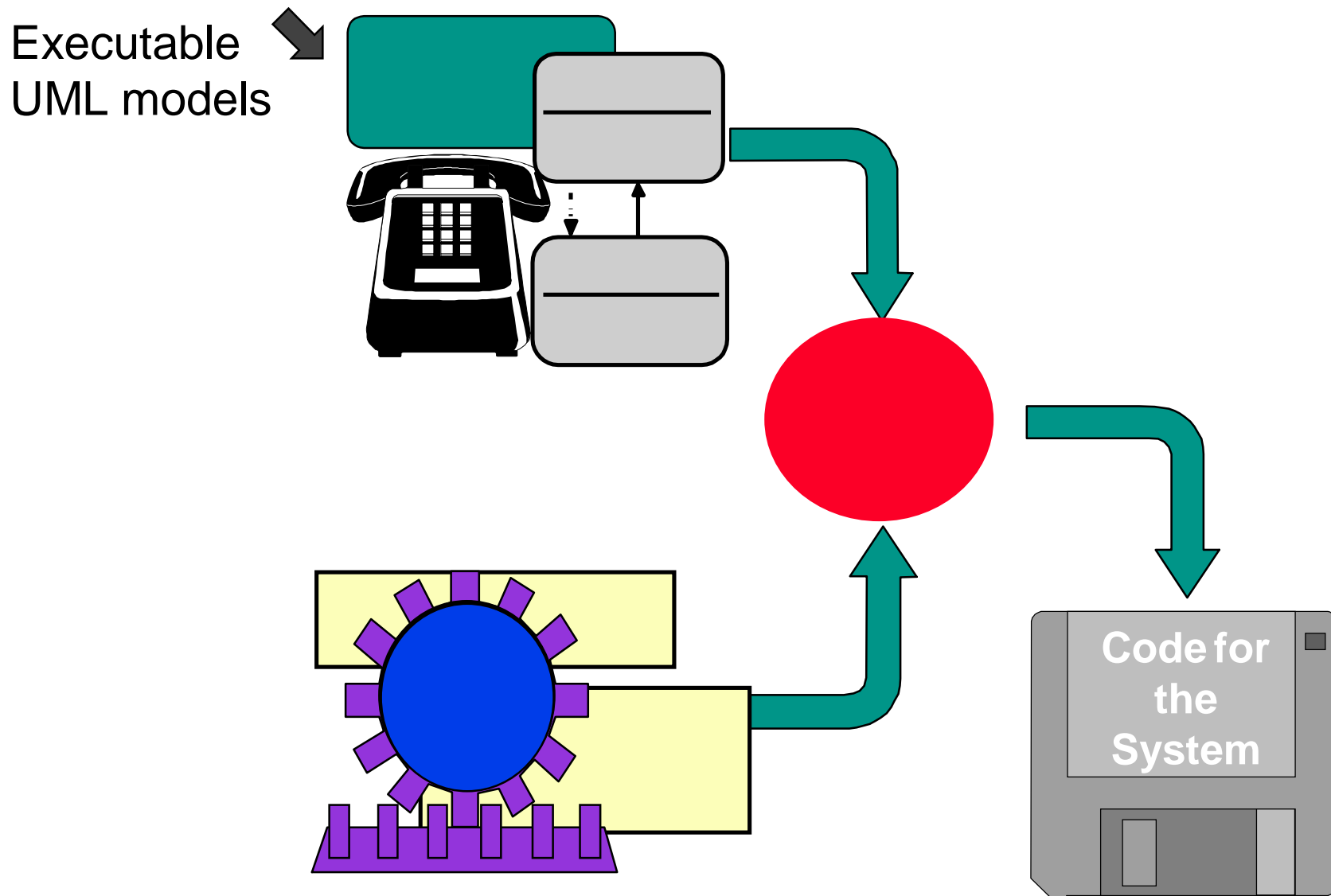
Translative System Generation is a system-construction method based on:

- ❖ separating systems into subject matters (domains)
- ❖ specifying each domain with an executable UML model
- ❖ translating the models

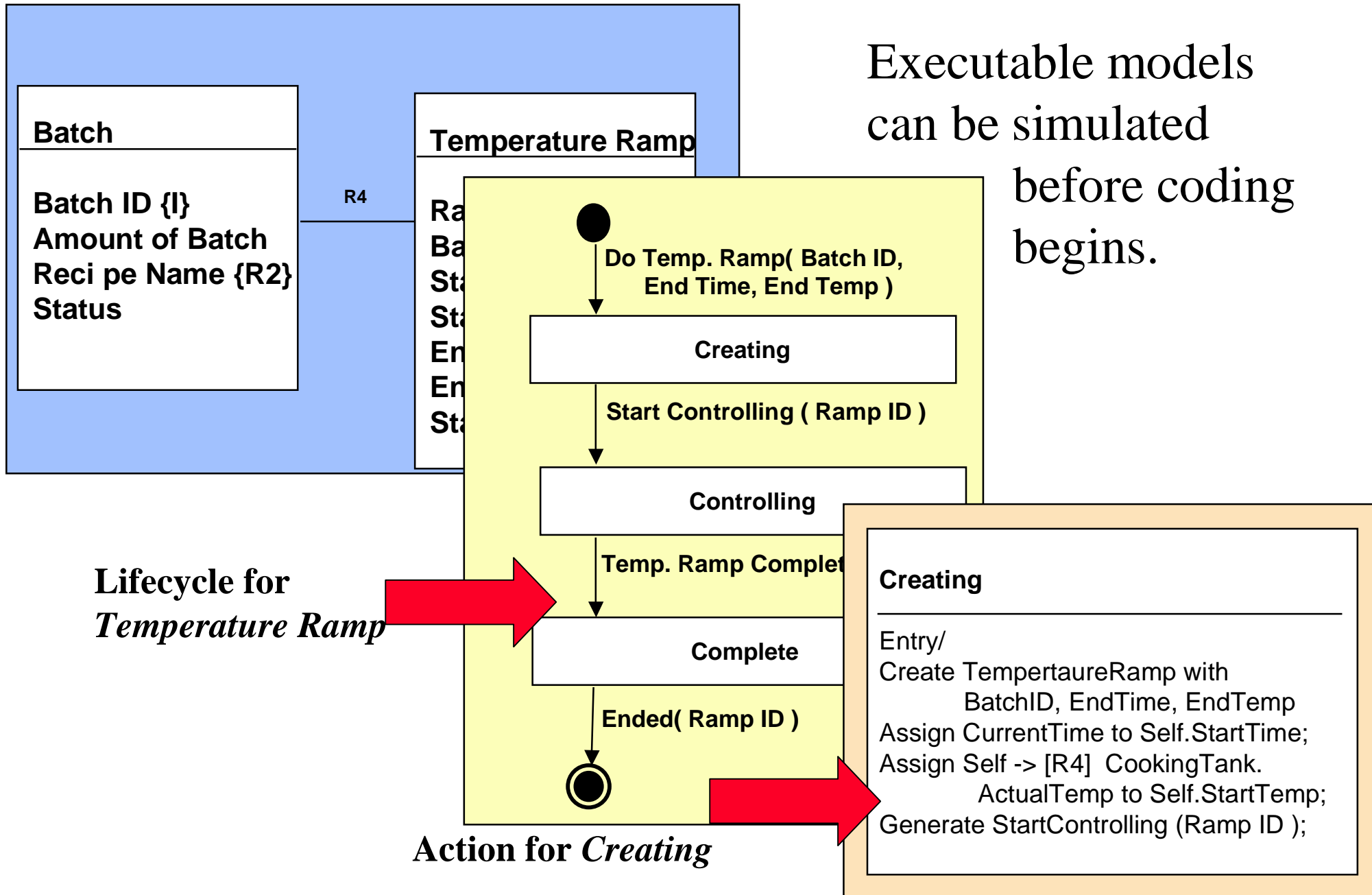
# Subject Matter Separation



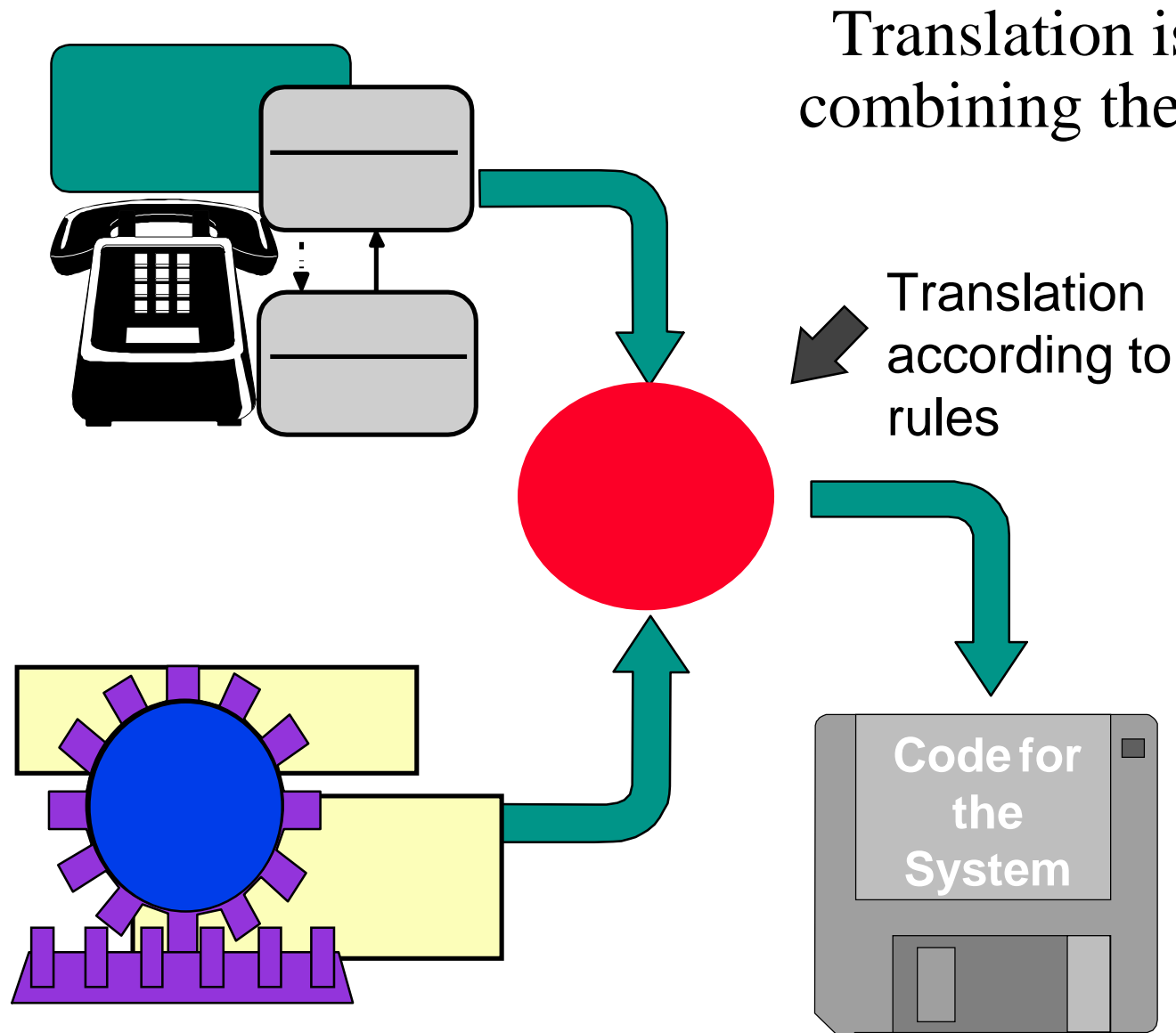
# Executable UML Models



# Executable UML Models



# Translation



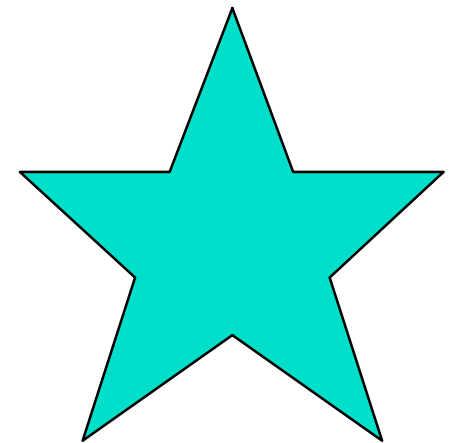
# Translation

Translating the application domain models generates:

- ◆ highly systematic
- ◆ uniform
- ◆ reproducible
- ◆ *understandable application code*

and minimizes:

- ◆ *coding and code inspection effort*
- ◆ *coding errors*
- ◆ component integration issues





**Brought to you by...**



PROJECT TECHNOLOGY, INC.

**PROJECT TECHNOLOGY** INC.



**Translating Inspiration Into Products.**

**Makers of BridgePoint <sup>®</sup> and DesignPoint <sup>®</sup>**

**Stephen J. Mellor**  
**Project Technology, Inc.**  
**<http://www.projtech.com>**