System Design: Architectures and Archetypes

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

PROJECT TECHNOLOGY INC.

Shlaer-Mellor Method

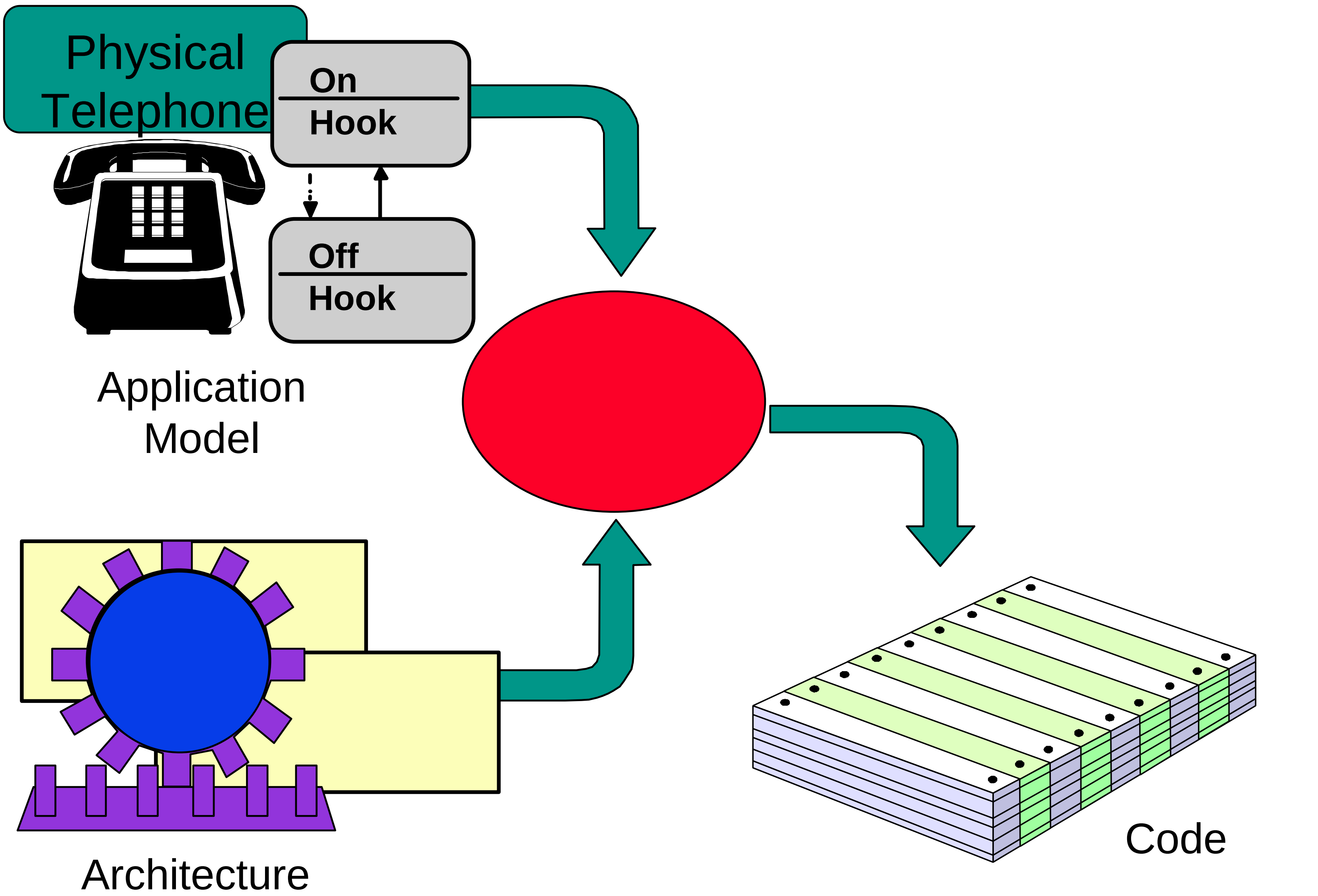
System Design: Architectures and Archetypes

This tutorial shows you how to:

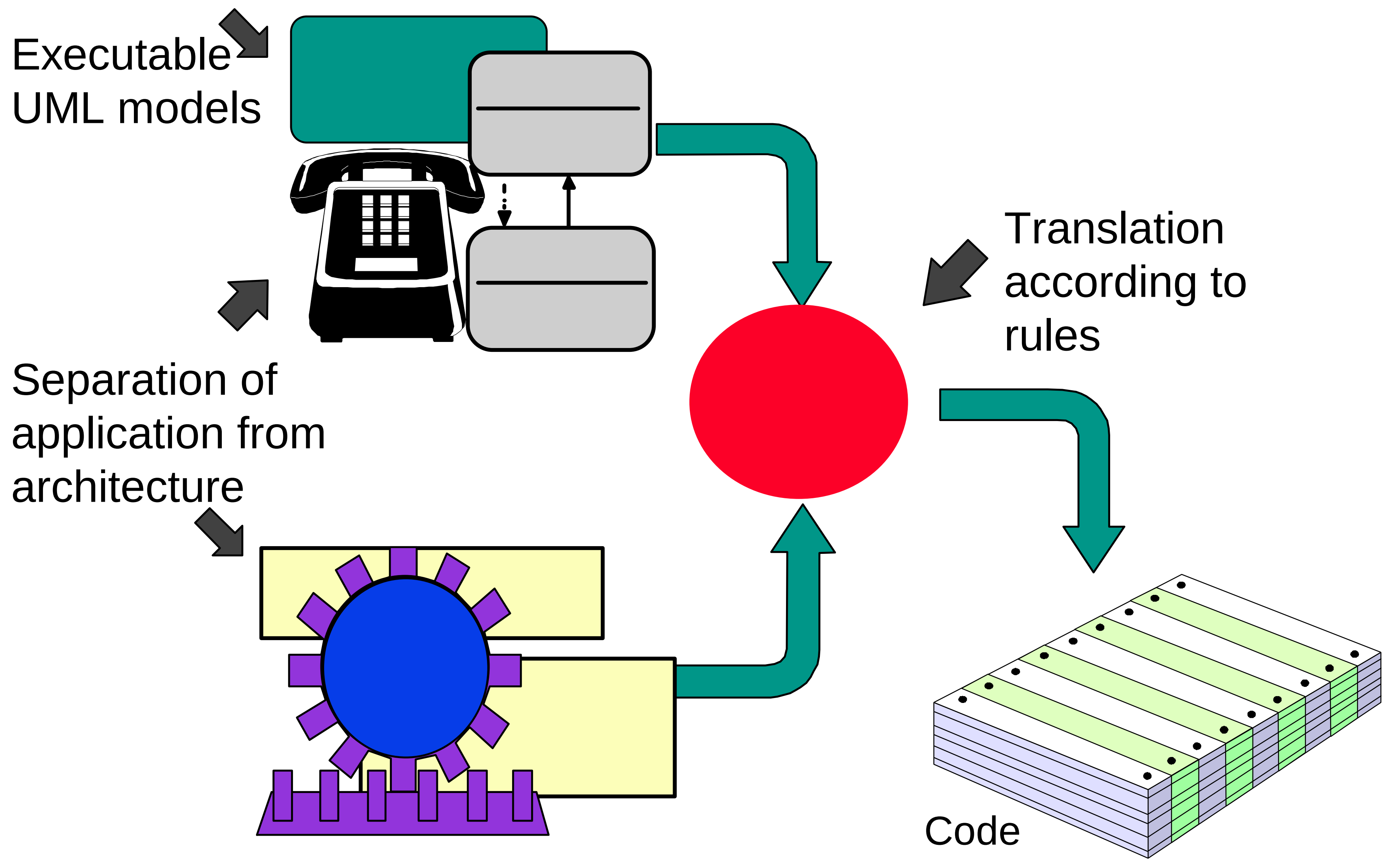
- identify the characteristics that determine the system design;
- engineer the system-wide design to meet performance constraints;
- model the system-wide design—the software *architecture*;
- build *archetypes* to produce efficient code.

Application-Independent Software Architecture

PROJECT TECHNOLOGY, INC.



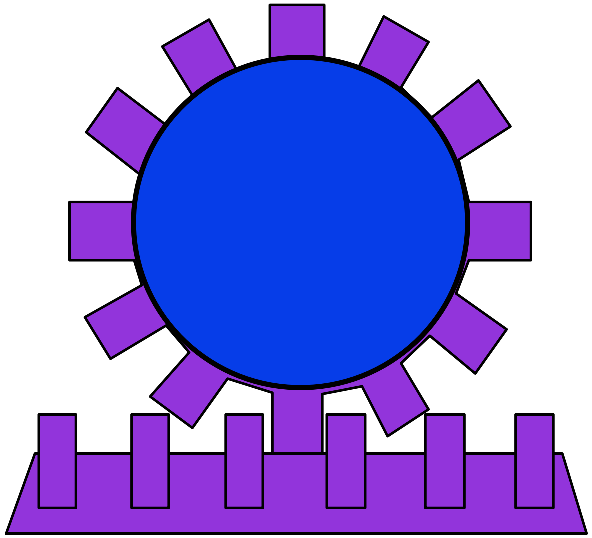
Properties


Executable
UML models
Separation of
application from
architecture
Translation
according to
rules
Code

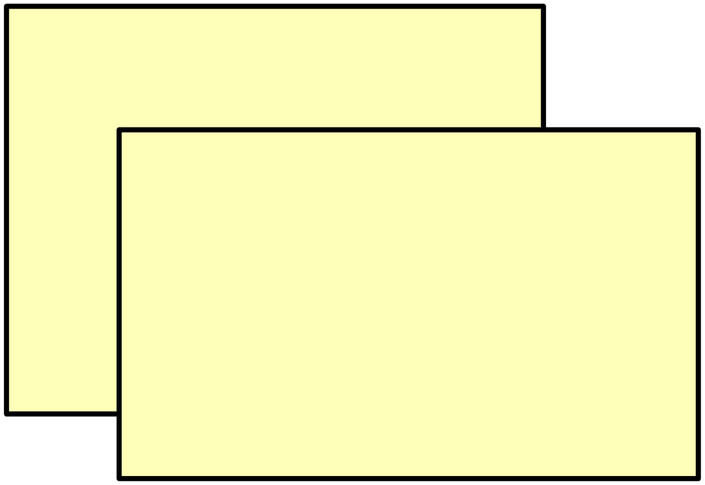
What's in the Architecture?

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.

placeholder introduced by ${...}

```
.Function Class
  Class ${Class.name} :
    public ActiveInstance {
  private:
  .invoke PrivateDataMember( Class )
……

};
```

text
(which
happens to be
C++)

```
.Function PrivateDataMember( inst_ref class )
.select many PDMs from instances
   of Attribute related to Class
.for each PDM in PDMs
${PDM.Type} ${PDM.Name};
.endfor
```

Application-Independent Software Architecture

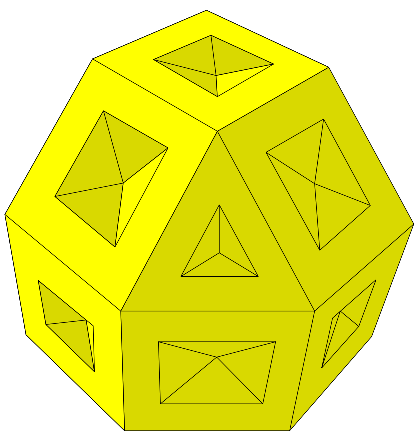
The software architecture is independent of the semantics of the application.

This offers:

- early error detection through verification
- reuse of the architecture
- faster performance tuning
- faster integration
- faster, cheaper retargeting

Table of Contents

The Software Architecture



Challenges of Real-Time Development

How can we both:

- provide required functionality

and

- meet real-time performance constraints?

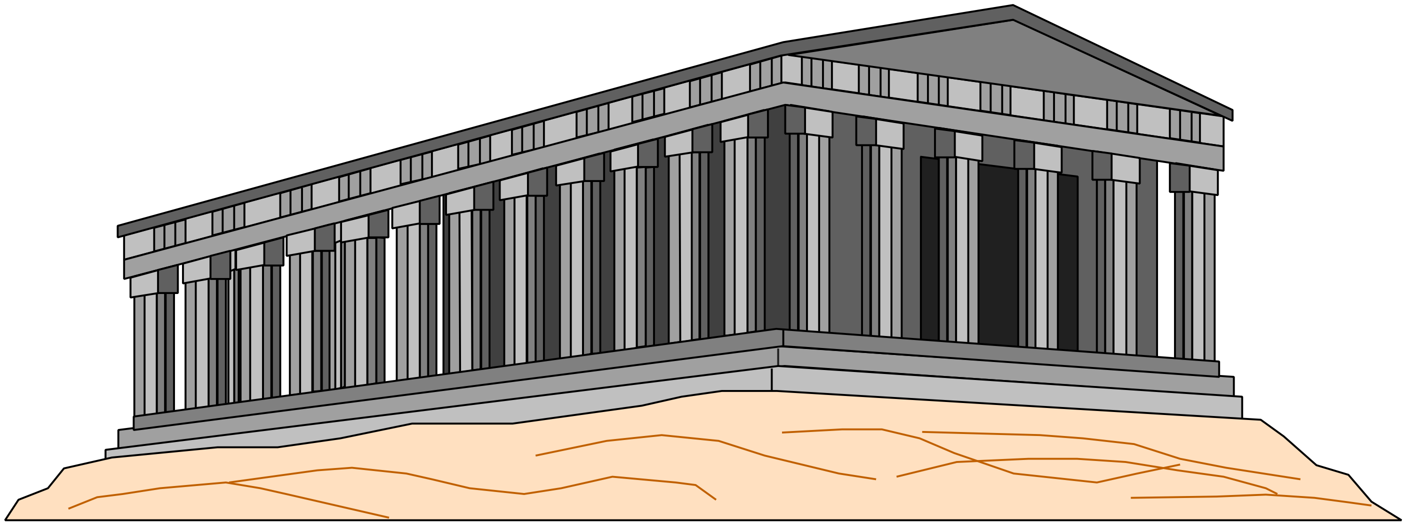
☞ (Re-)organize the software.

Software Architecture

The abstract organization of software is called the *software architecture*.

It proclaims and enforces system-wide rules for the organization of:

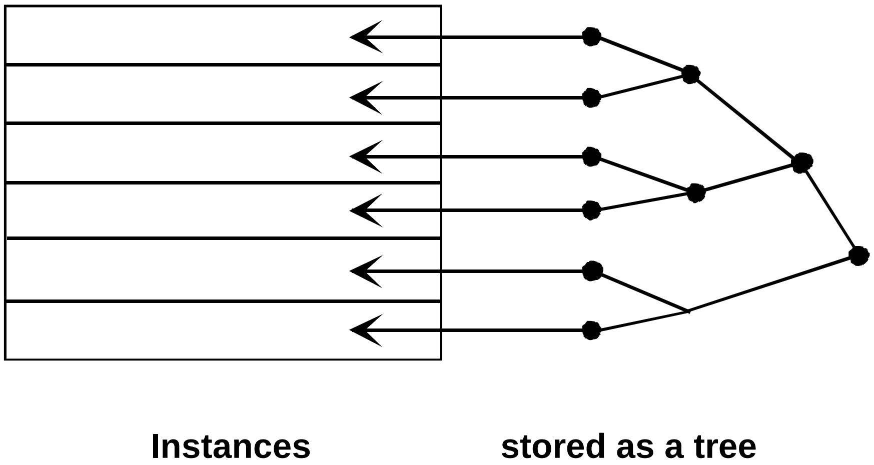
- data
- control
- structures
- time



Data

The architect prescribes the *storage scheme* for data elements:

- tables or arrays?
- special purpose structures such as trees, linked lists?
- independent?



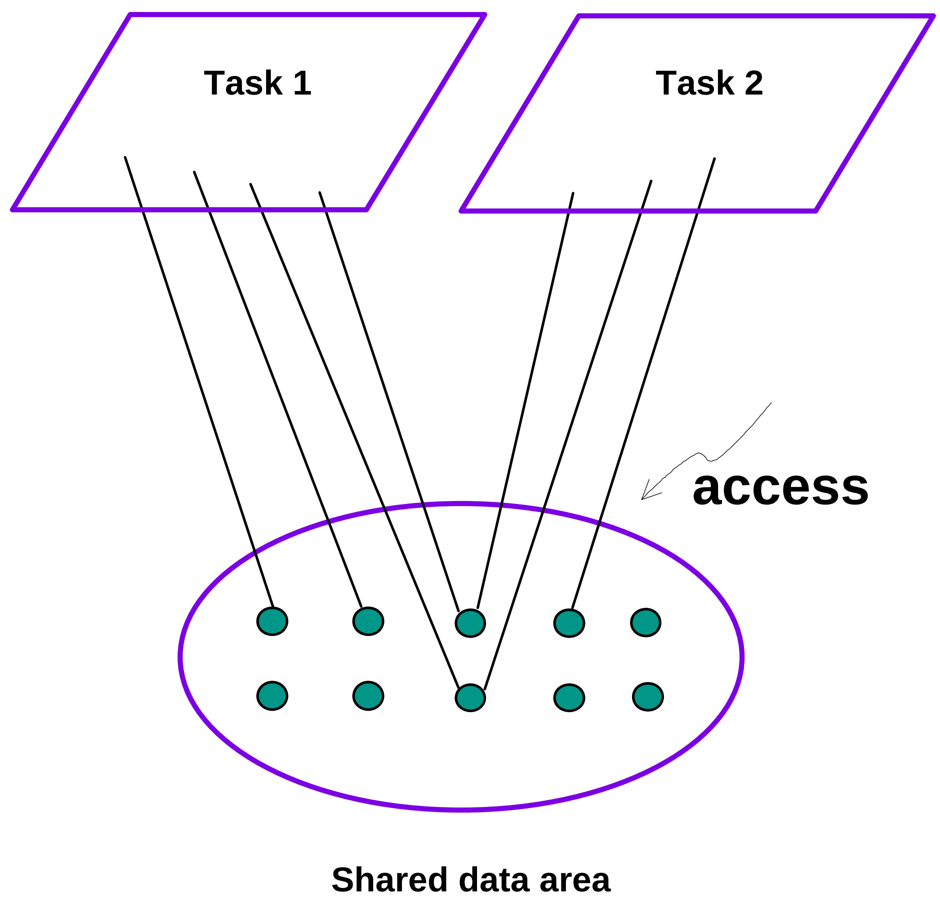
and *access* to them:

- direct access by name or pointer?
- indirect access through functions that encapsulate the data structure?

Control

The architect prescribes control:

- what causes a task to execute?
- what causes a task to relinquish control?
- what is the next function to execute within a task?
- how to coordinate multiple tasks accessing common data to ensure data consistency?

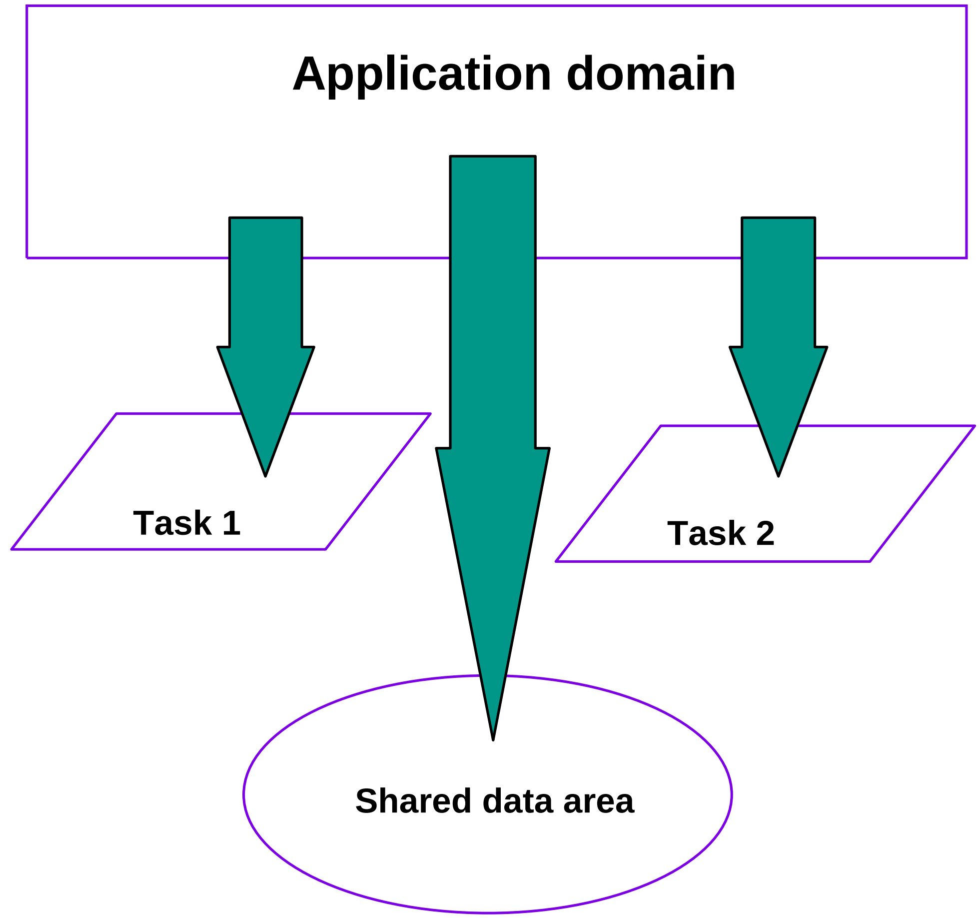


Structures

The architect prescribes how *to package* code and data in:

- tasks?
- functions?
- shared data areas?
- classes?

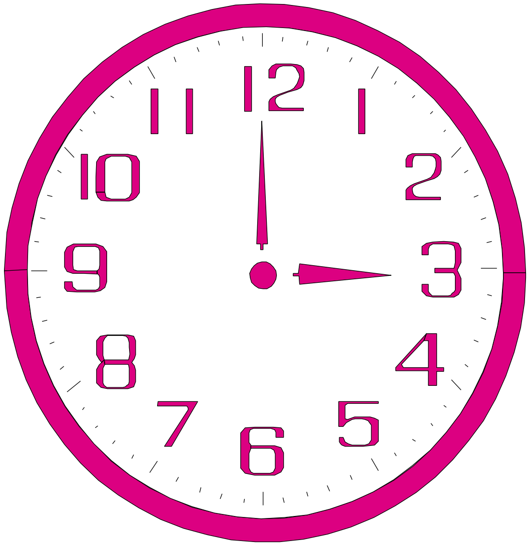
and the *allocation criteria* for allocating parts of the application to these structures.

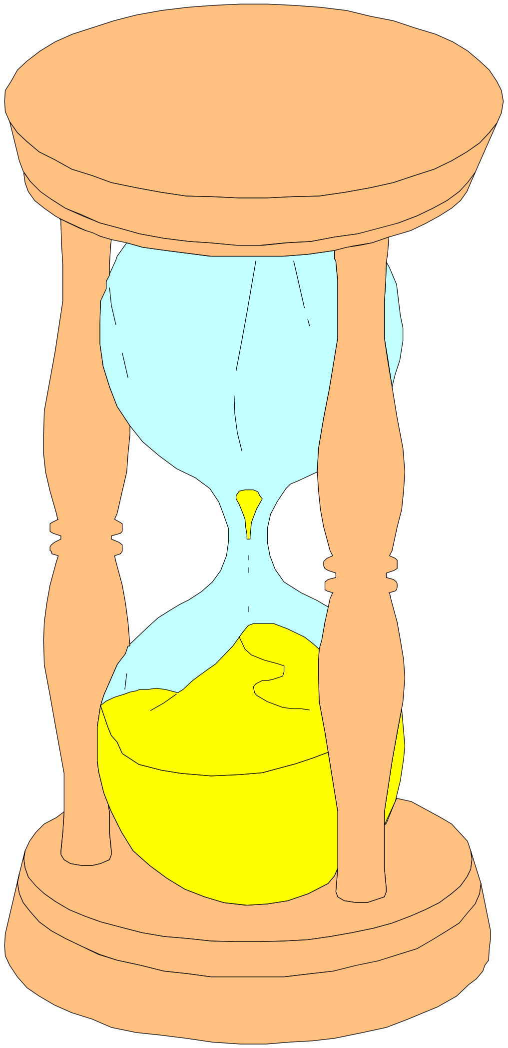


Time

The software architect prescribes how to provide time-related services:

- absolute time
- relative time

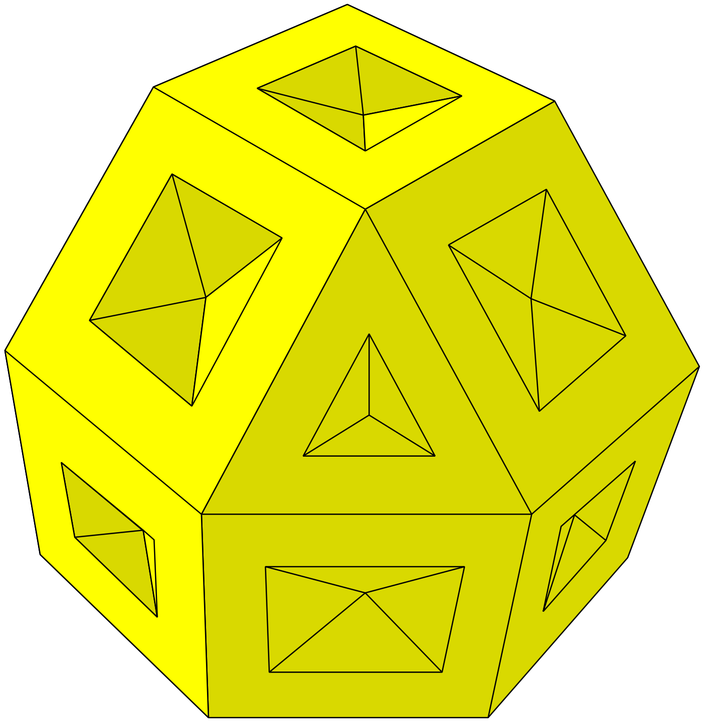




Uniformity

A minimal, uniform set of organization rules:

- reduces cost of understanding, building, and maintaining the software
- decreases integration effort
- leads to smaller, more robust code

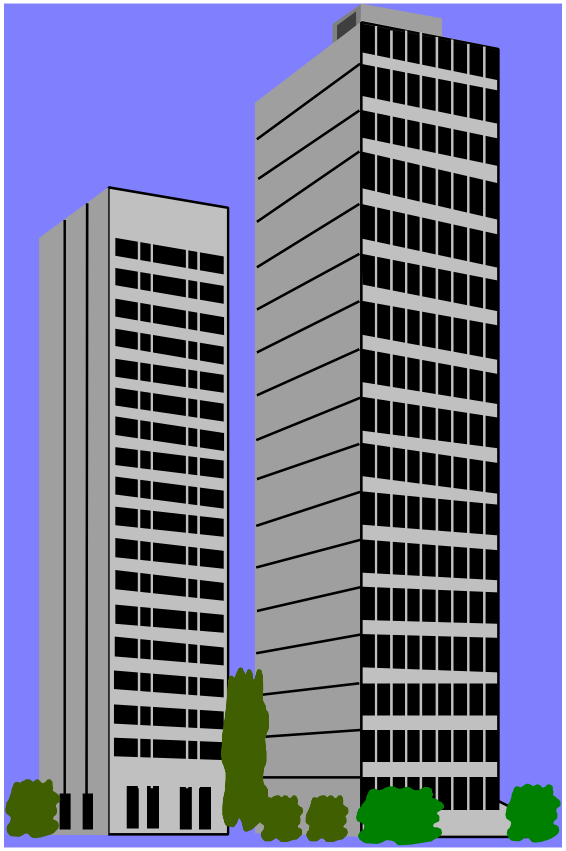


Architectural Styles

Architectural Styles

Real-time and embedded systems commonly employ (parts of) three major architectural styles:

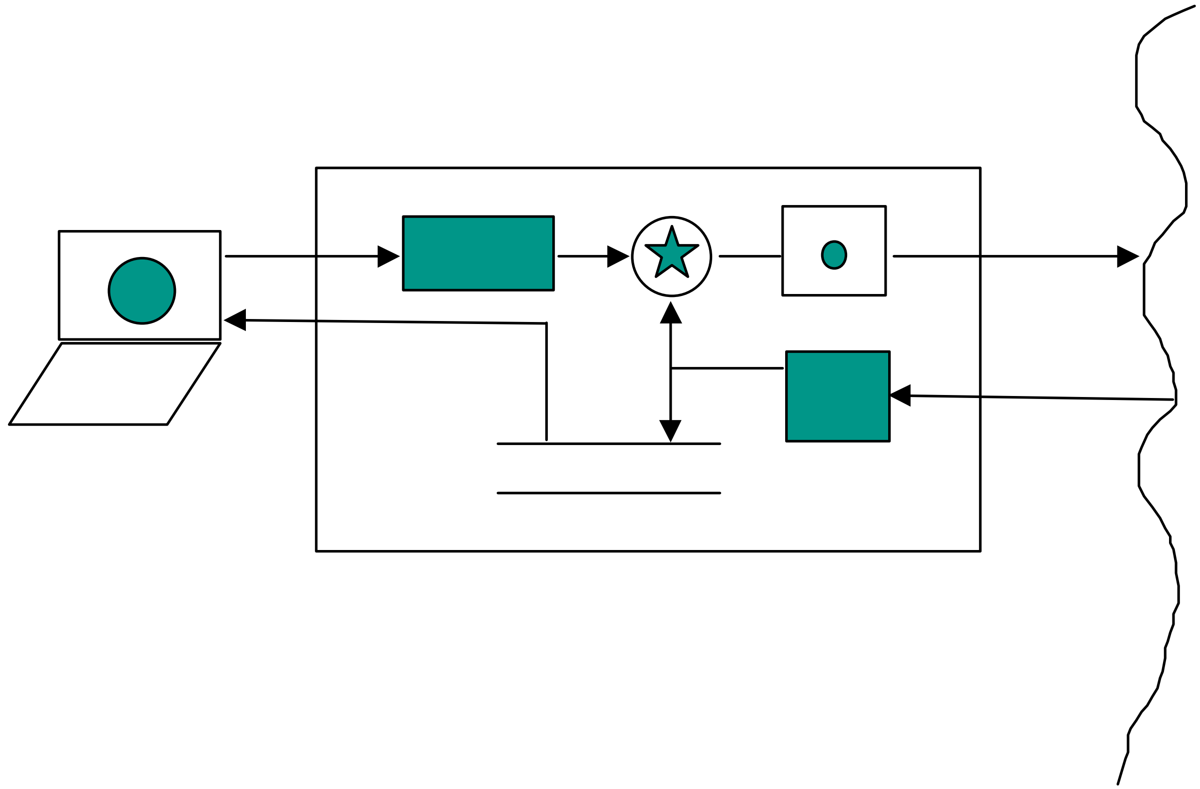
- Monitor and control
- Transporters
- Transactions



Monitor and Control

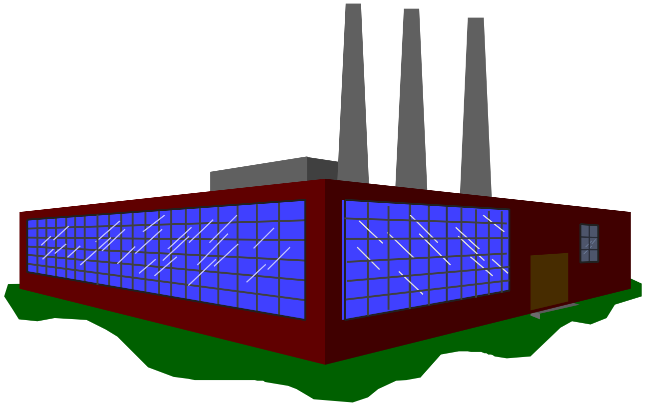
This style comprises a collection of related control loops that:

- set control points in the hardware with desired values
- read values from hardware for comparison or display

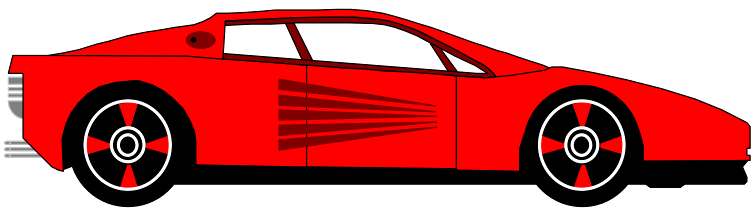


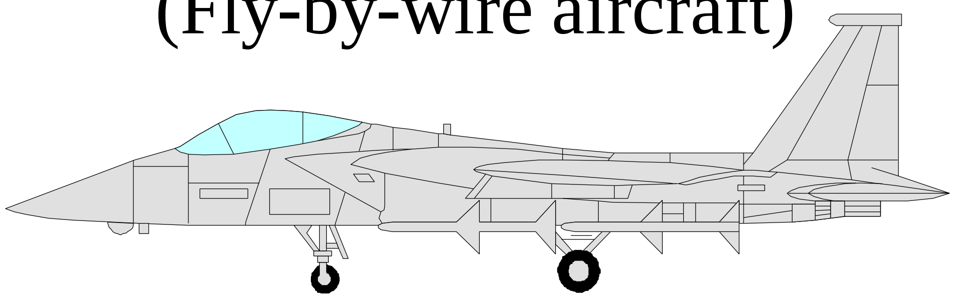
Monitor and Control

- Manufacturing systems (Aluminum rolling mill)
- Embedded microprocessor control systems (automobiles)



•Real-time control systems (Fly-by-wire aircraft)





- Household microprocessor (temperature control)

Monitor and Control

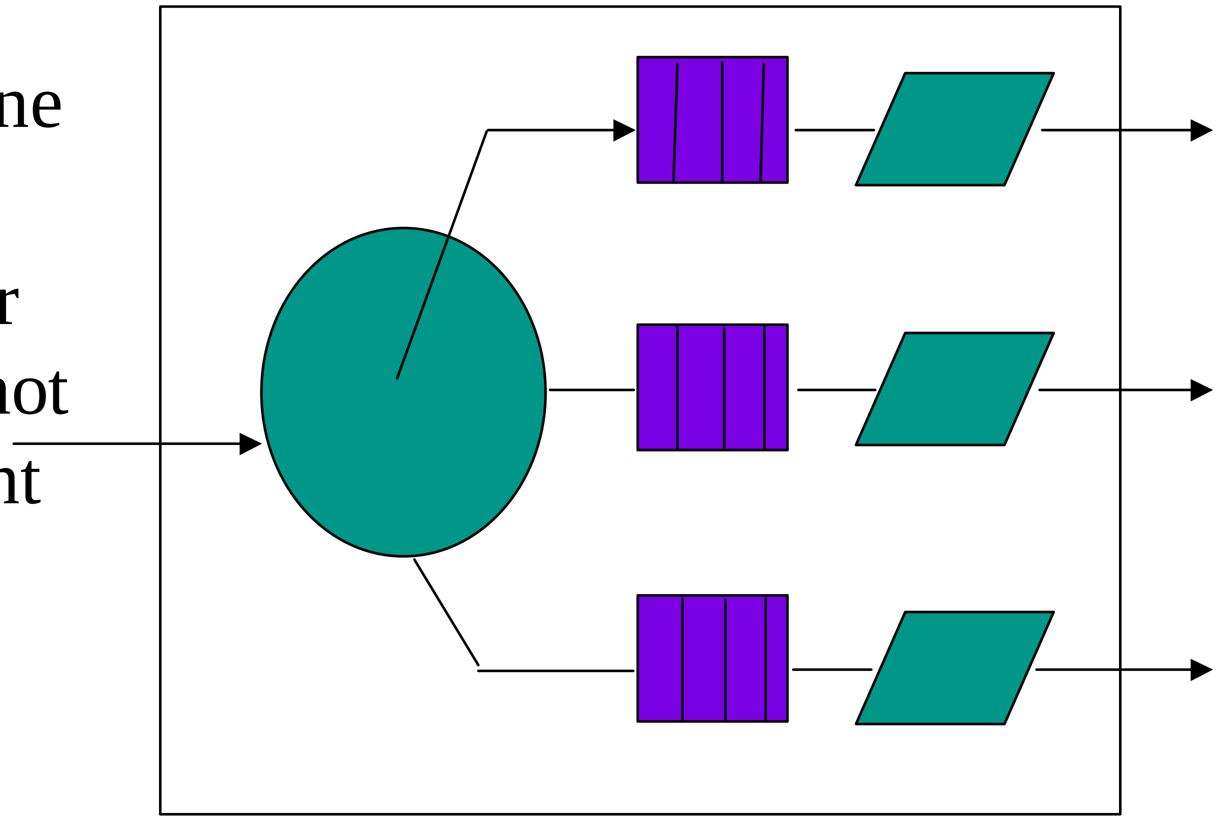
This style tends to have:

- hard response deadlines
- data that must have current values
- significant computation on the data

Transporters

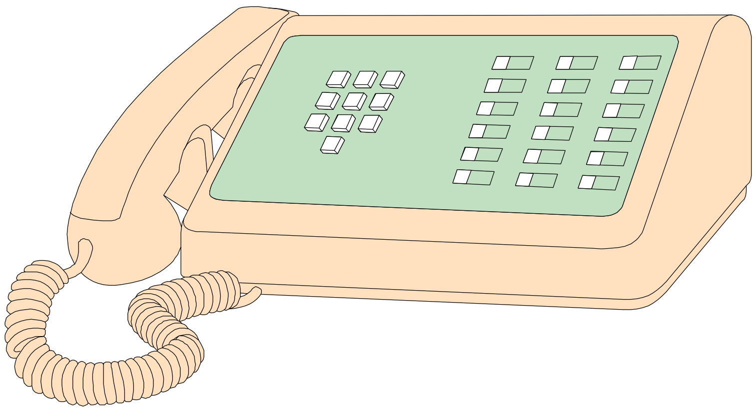
Transporters:

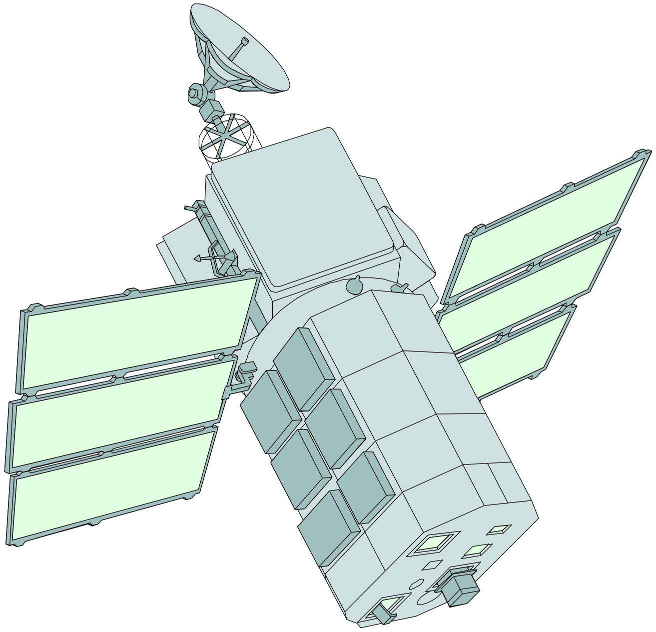
- move data from one place to another
- are responsible for routing data, but not for the data content
- may split or re-assemble the data packets



Transporters

- telephony
- telemetry



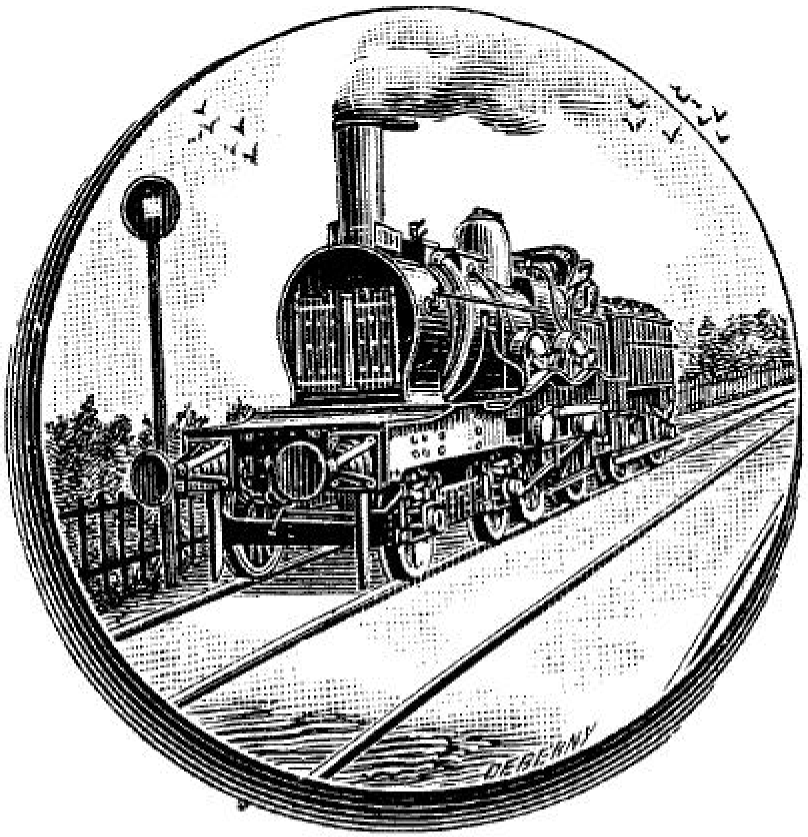


- off-line transaction processing (credit card processing)
- data collection and archiving systems

Transporters

Transporters:

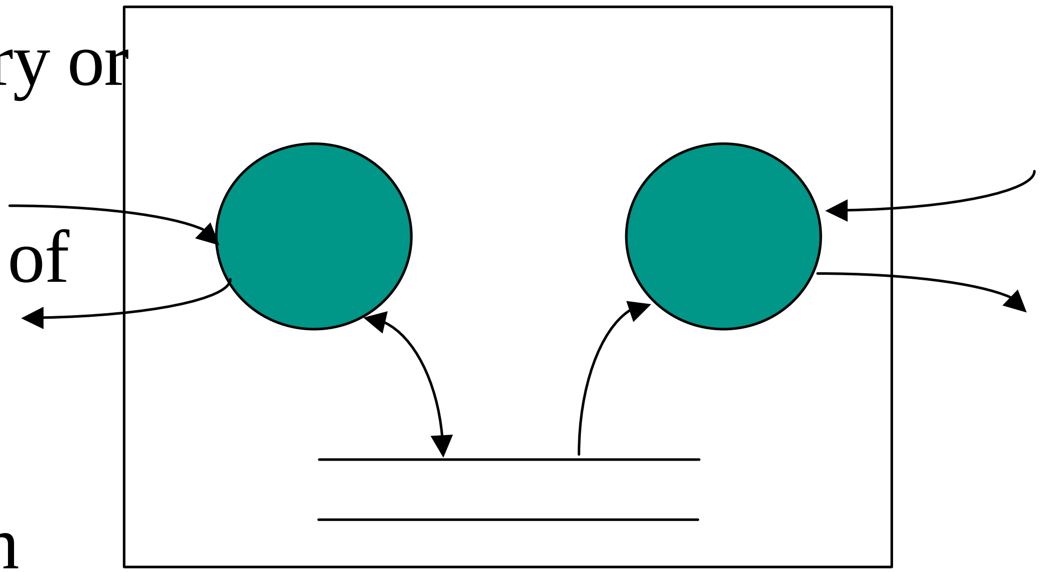
- must meet throughput requirements
- may have response time requirements on some streams
- have persistent application data describing routing
- must manage buffers containing application packets



Transactions

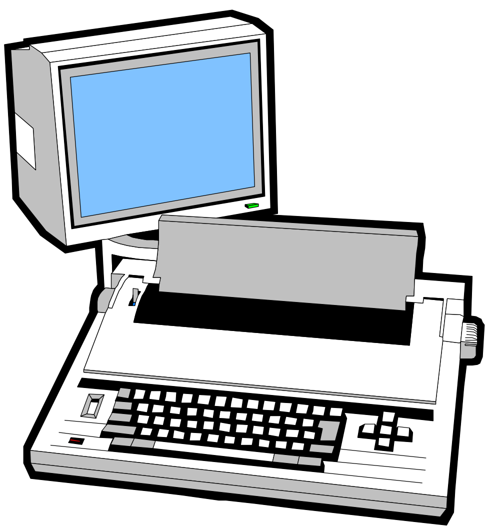
Transactions:

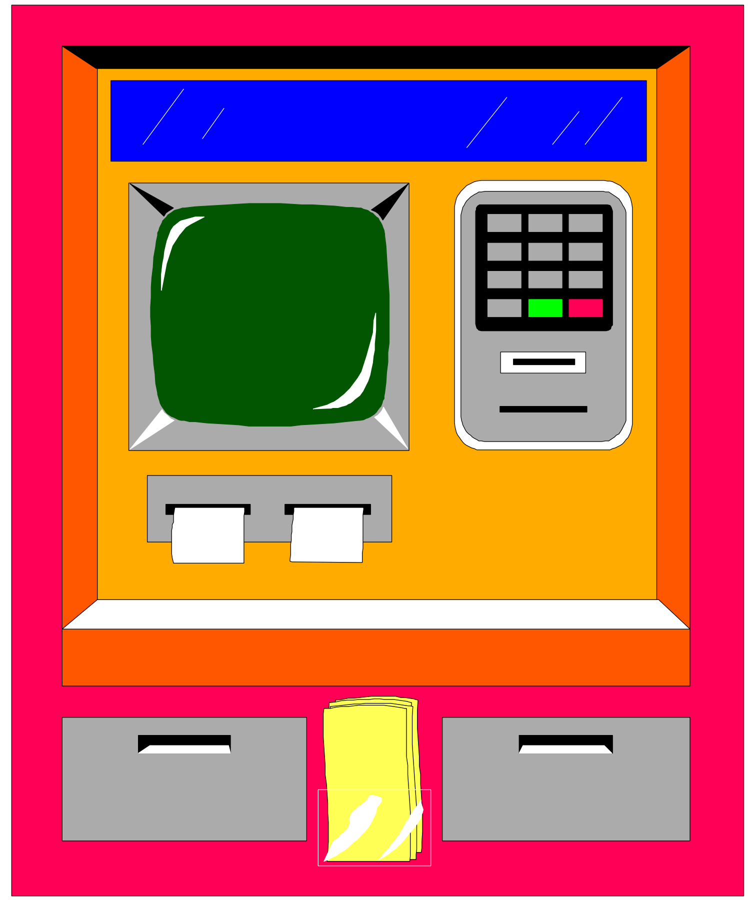
- maintain a picture of a real or hypothetical world
- accept requests to query or update the picture
- perform some amount of computation
- send responses to the outside world based on the computation



Transactions

- on-line banking
- reservation systems
- simulators
- desktop applications (word processors, spreadsheets)

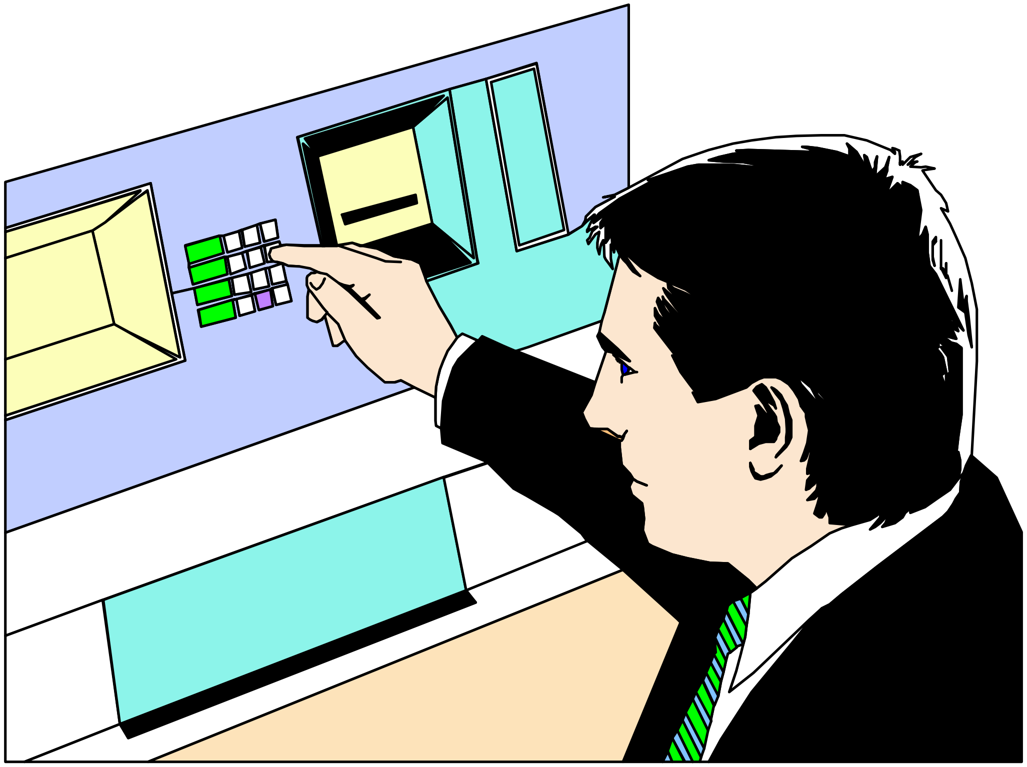




Transactions

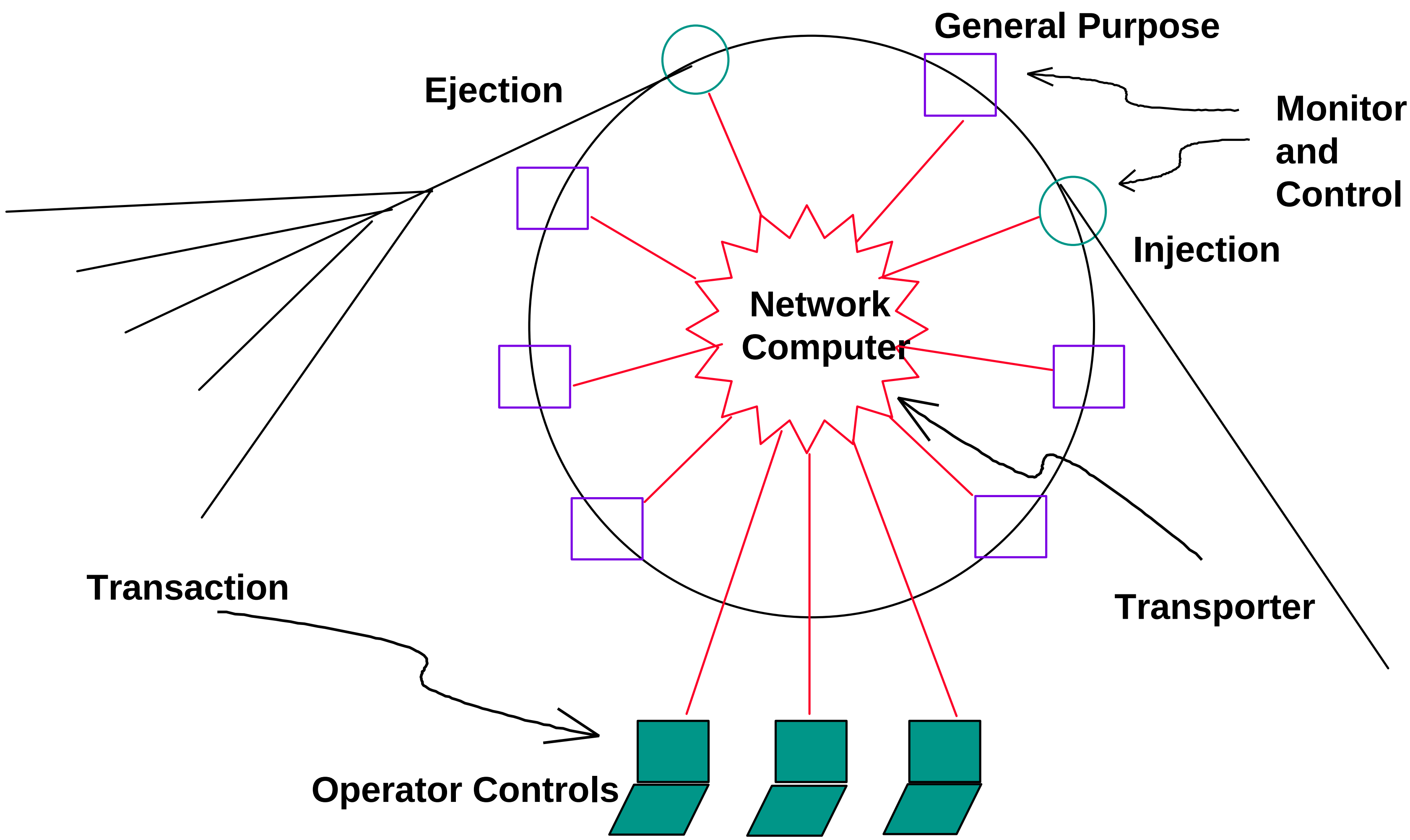
This style tends to have:

- considerable persistent application data
- variable response times
- significant throughput requirements



Hybrids

Many systems use several styles.



Selecting an Architecture

Characterize the System

"[E]very design problem begins with an effort to achieve a fitness between two entities: the form in question and its context. The form is the solution to the problem; the context defines the problem. In other words, when we speak of design, the real object of discussion is not the form alone, but the ensemble comprising the form and its context. Good fit is a desired property of this ensemble which relates to some particular division of the ensemble into form and context."

Notes on the Synthesis of Form

Christopher Alexander

The External World

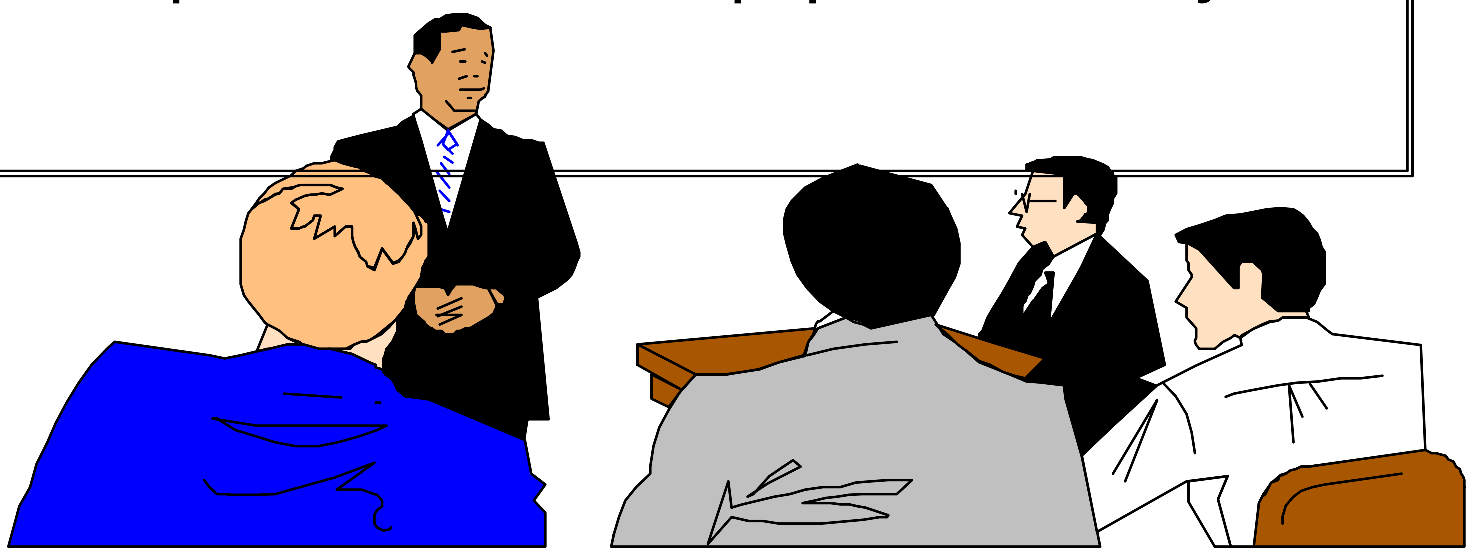
Understand and quantify the external world in terms of:

- rate and volume of events originating in the external world
  - normal quiescent rates
  - burst rates in periods of unusual demand
- its natural periodicities
- how frequently data elements change values

Non-Localized Requirements

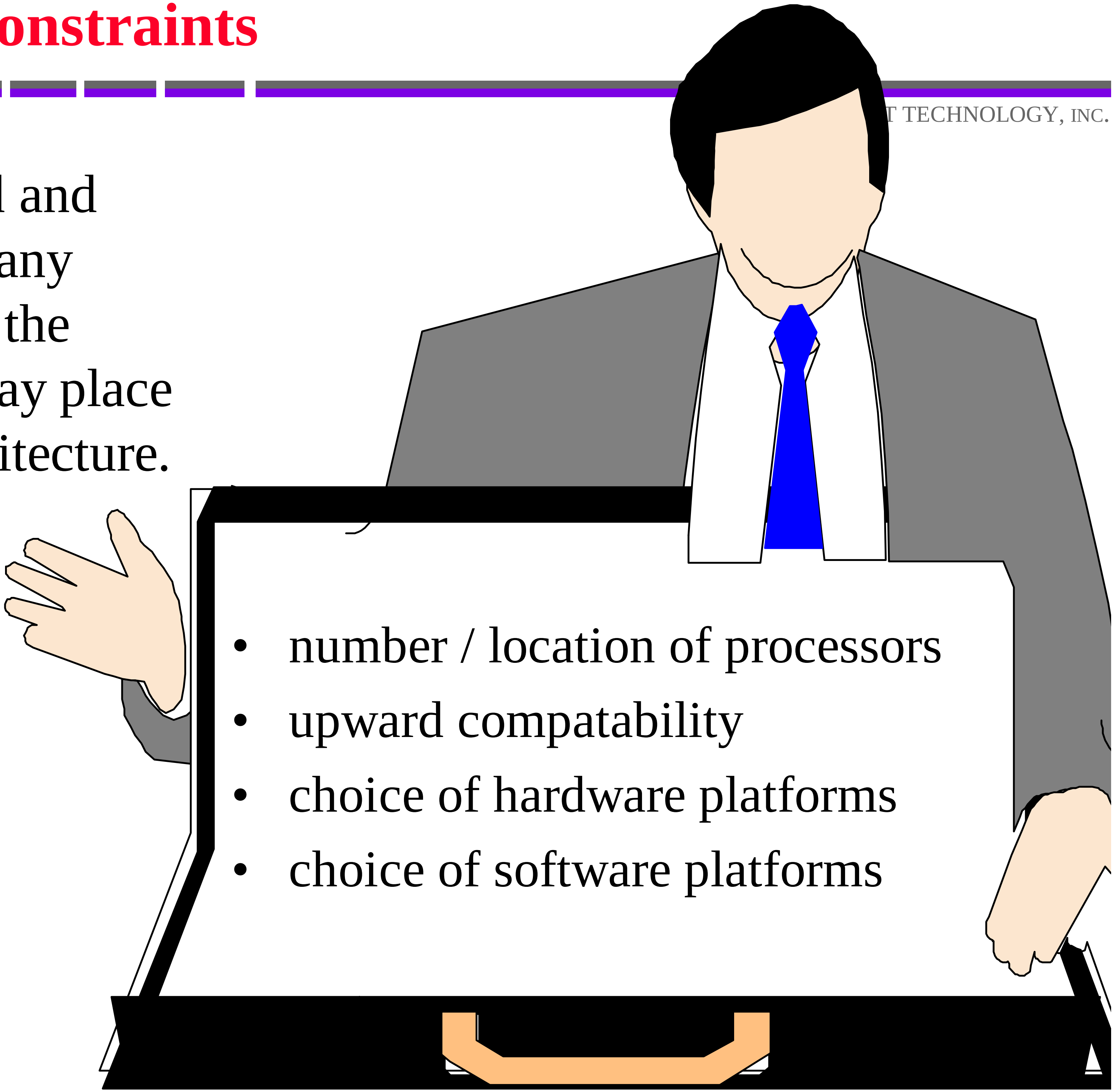
Requirements Meeting

- continuous 24 x 7 operation
- fault tolerance and recovery
- personnel and equipment safety



Business Constraints

Understand and enumerate any constraints the business may place on the architecture.

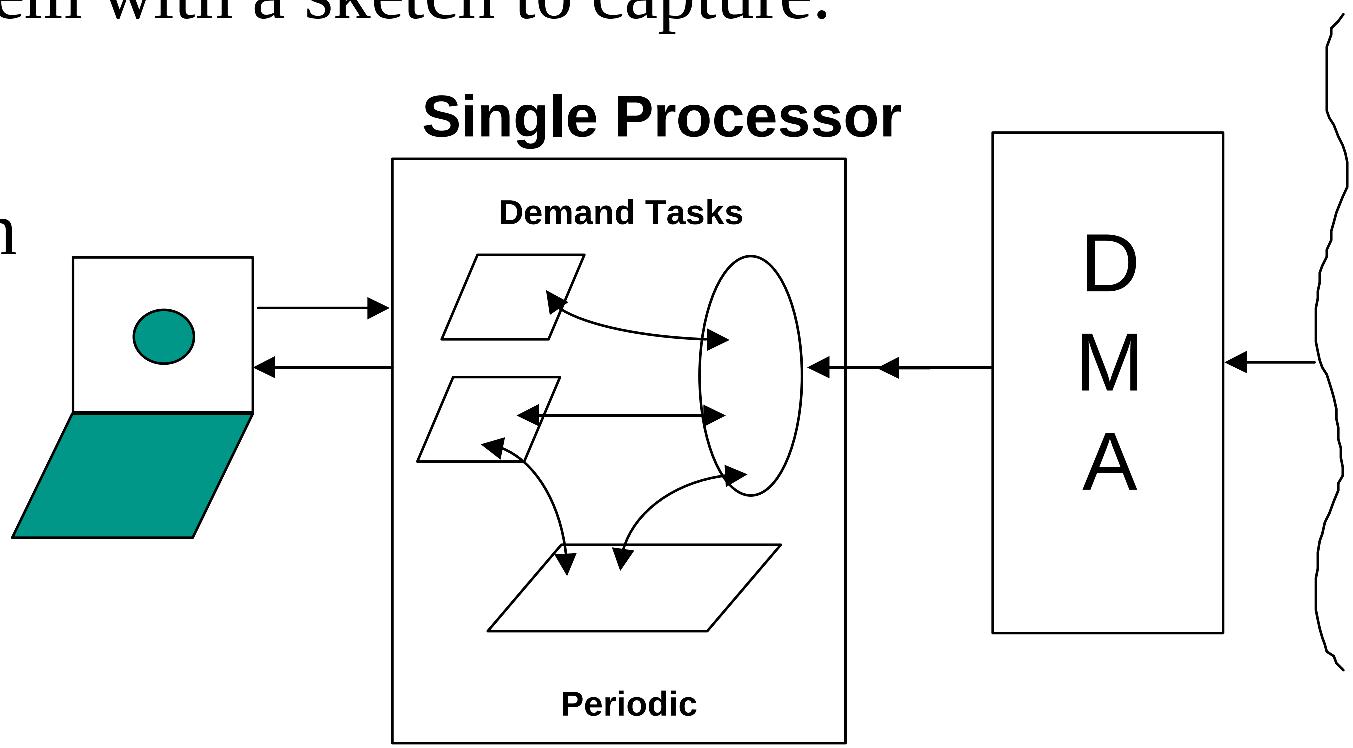


- number / location of processors
- upward compatability
- choice of hardware platforms
- choice of software platforms

System Sketch

Document the system with a sketch to capture:

- processors
- communication channels
- bandwidth
- external actors
- protocols



to provide a reference basis for both the client and the architect.

Performance Requirements

The High Spots

“It is common practice in engineering, if we wish to make a metal face perfectly smooth, to fit it against the surface of a metal block which is level within finer limits than we are aiming at, by inking the surface of this standard block and rubbing our metal face against the inked surface. If our metal face is not quite level, ink marks appear on it at those points that are higher than the rest. We grind away at these high spots...”

Notes on the Synthesis of Form

Christopher Alexander

Monitor and Control

Determine the sampling time(s).

The external process may be:

- naturally periodic

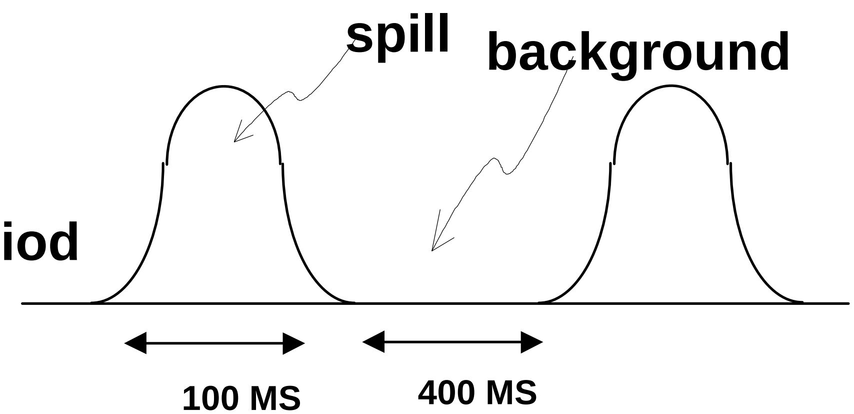
  use natural period

- continuous

  impose period based on fastest data

- loosely coupled

  impose period based on longest acceptable delay



Presence of beam

A naturally periodic system may require sampling at several points in the waveform.

Transporters

Streams* may have packets* that can be:

- state-dependent, or
- throttled, or
- ignored with impunity

For the worst case, figure:

- throughput requirements
- response-time requirements

for each stream.

*** A <u>stream</u> is a source of packets.**
*** A <u>packet</u> is some piece of information (control or data).**

Transactions

Threads* may be either:

- time-critical
- at operator speeds
- at will

Throughput, then, is:

- subordinate to the critical threads
- important on the average
- the design goal

*** A <u>thread</u> is all the work done as a result of some stimulus.**

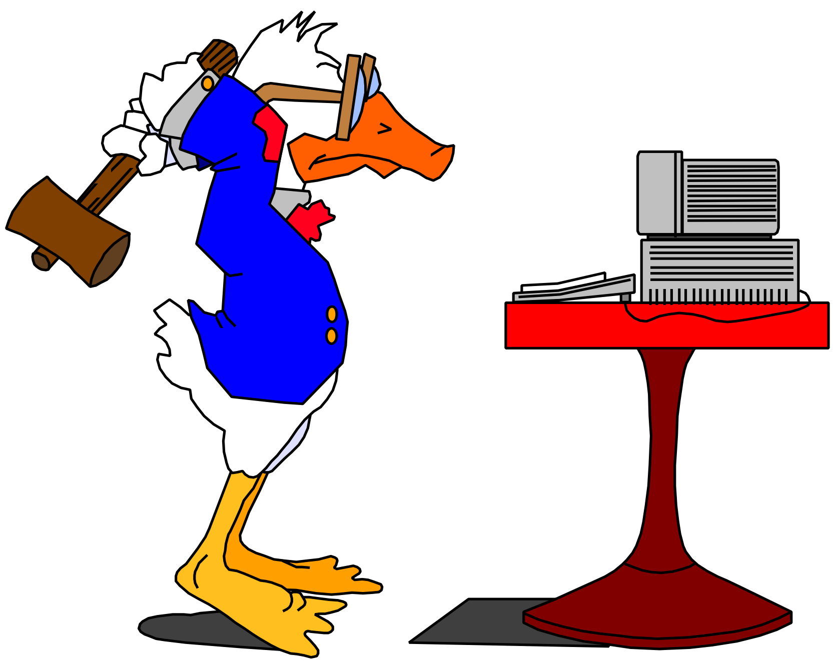
Performance Quantification

To quantify performance requirements in an analysis:

- Identify critical threads
- Identify worst bursts
- Identify the required processing for each

Only do this for the "high spots!"

Executable Domain Models



Unified Modeling Language

"The Unified Modeling Language is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system."

The UML Summary

UML Models

Unified Modeling Language (UML) addresses the following development tasks:

Requirements — requirements analysis (external usage)

Analysis — system modeling (data, control, algorithm)

Design — system deployment (allocation to processors)

UML Model Notation

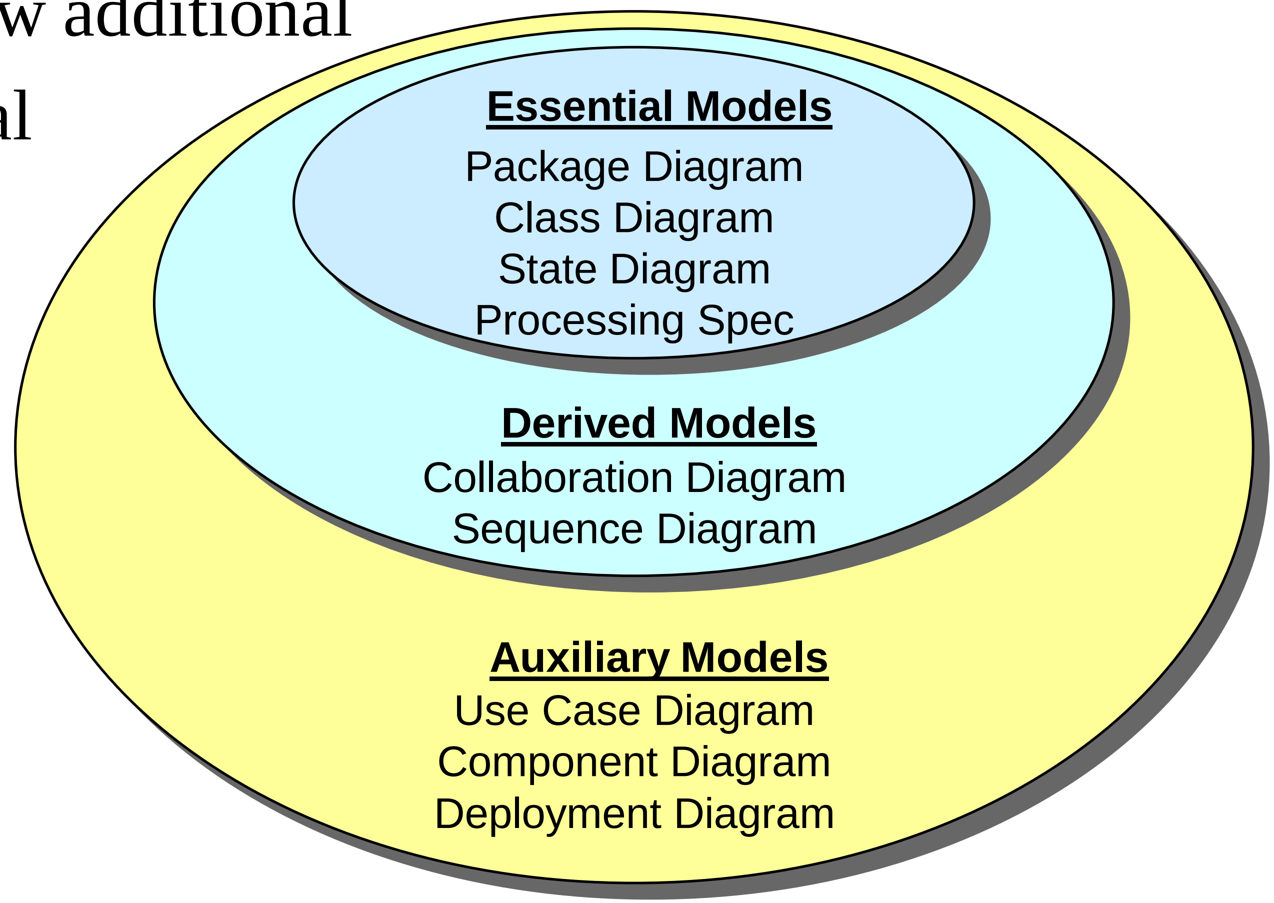
UML defines a notation for the following models.

- Use Case Diagram: system stimulus-response model
- Static Structure Diagram: package, class, and object models
- State Diagram: control for dynamic behavior
- Activity Diagram: workflow of activities
- Sequence Diagram: dynamic interactions with time
- Collaboration Diagram: dynamic interactions without time
- Component Diagram: software components
- Deployment Diagram: allocation of components to processing elements

Use of UML Models

- *Essential Models* capture the complete scope and behavior of the system and support model translation to code.
- *Derived Models* show additional views of the essential models.
- *Auxiliary Models* augment the essential models.

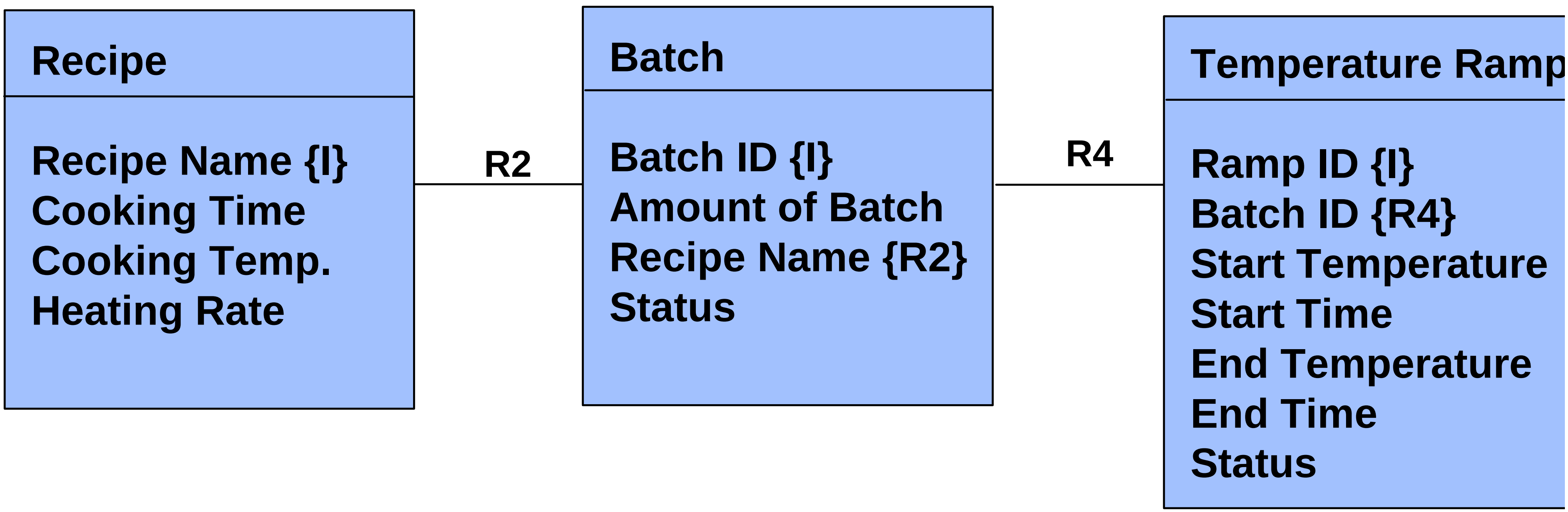
UML Models



Class Diagram

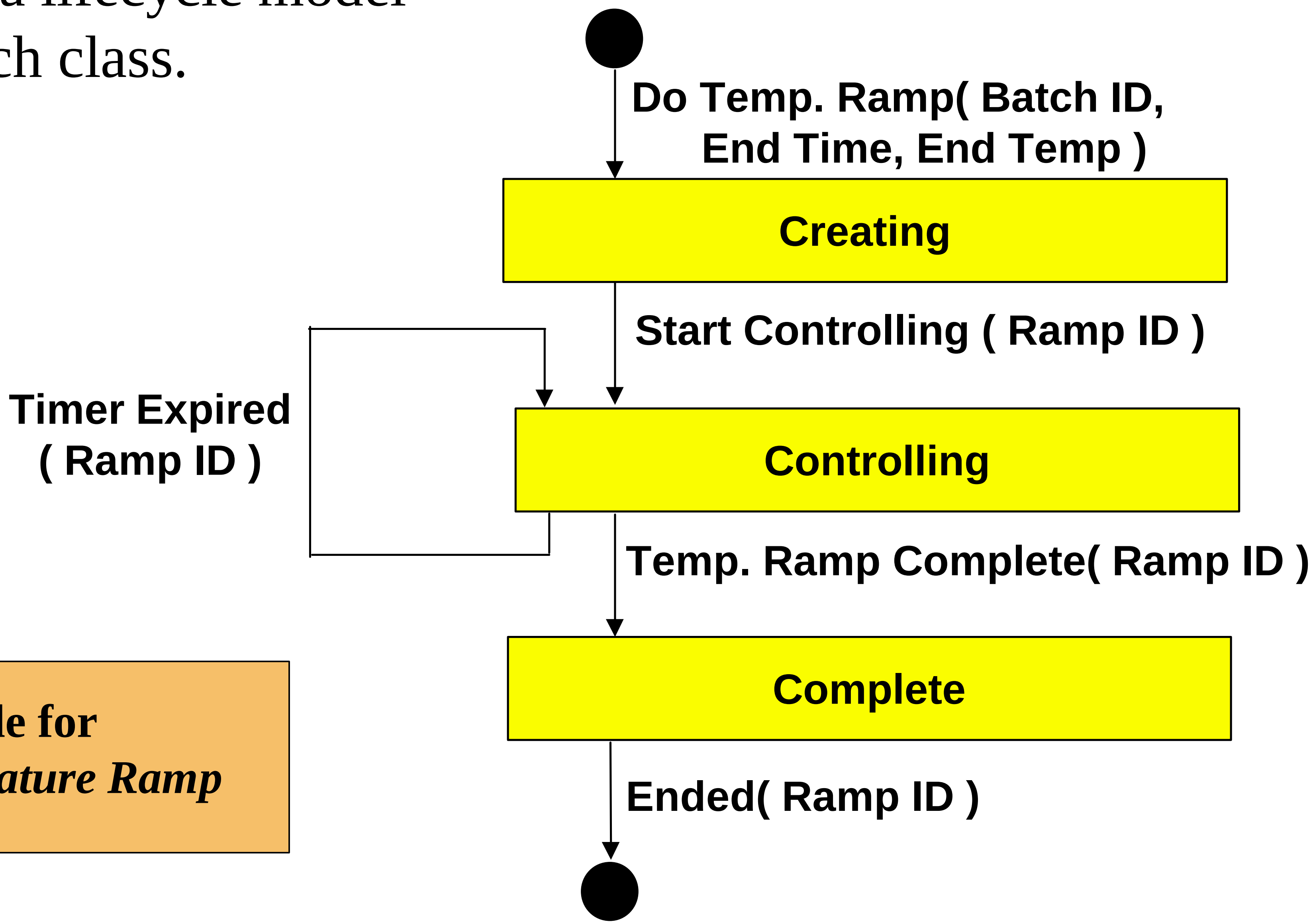
Abstract classes based on both:

- data, and
- behavior



Lifecycles

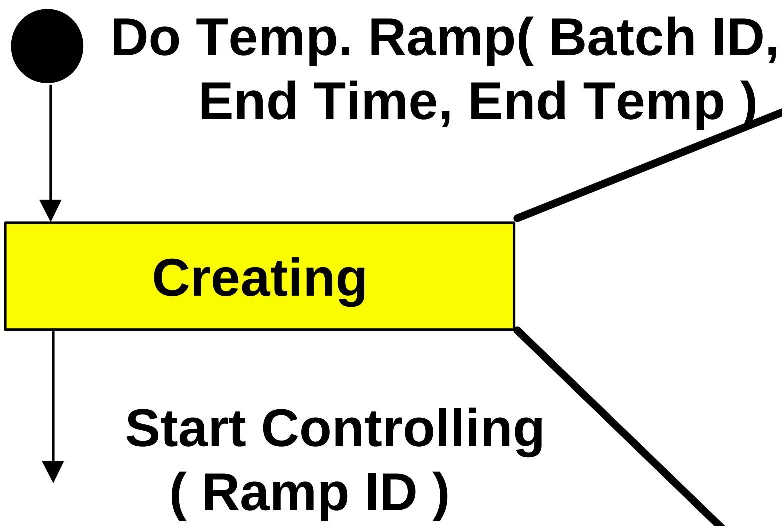
Build a lifecycle model
for each class.



Lifecycle for
Temperature Ramp

Actions

Specify the logic for each state's action.



Creating

Entry/
BatchID, EndTime, EndTemp
    >> TempRamp;
CurrentTime > Self.StartTime;
Self -> [R4] CookingTank.ActualTemp
    > Self.StartTemp;
Signal Start Controlling (Ramp ID);

Action Semantics

The action semantics should:

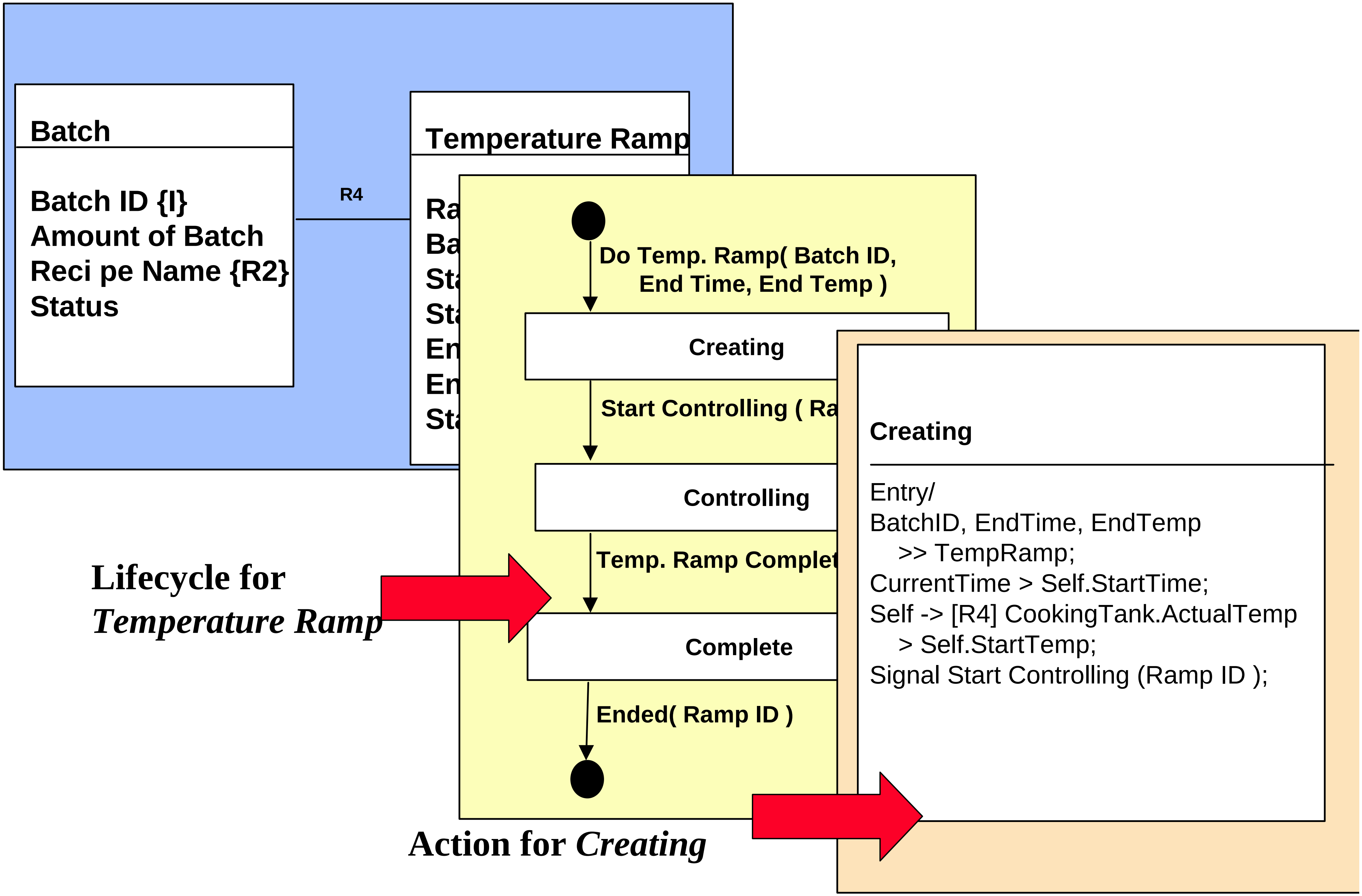
- not over-constrain sequencing
  - i.e concurrency & data flow
- separate computations from data access
  - to make decisions about data access without affecting algorithm specification
- manipulate only UML elements
  - to restrict the generality and so make a specification language

<u>Creating</u>

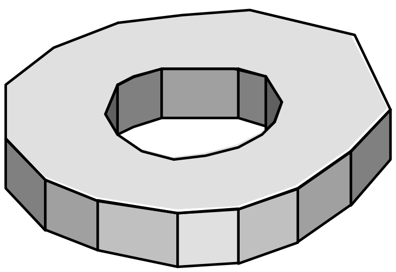
```
Entry/
BatchID, EndTime, EndTemp
   >> TempRamp;
CurrentTime > Self.StartTime;
Self -> [R4] CookingTank.ActualTemp
   > Self.StartTemp;
Signal Start Controlling (Ramp ID );
```

An Executable Model

PROJECT TECHNOLOGY, INC.

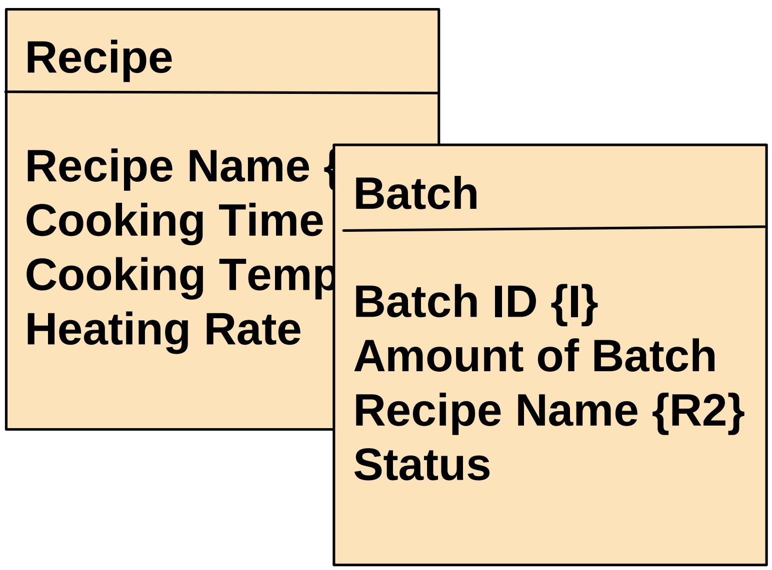


Model Execution



Instances

An executable model operates on data about instances.



Recipe

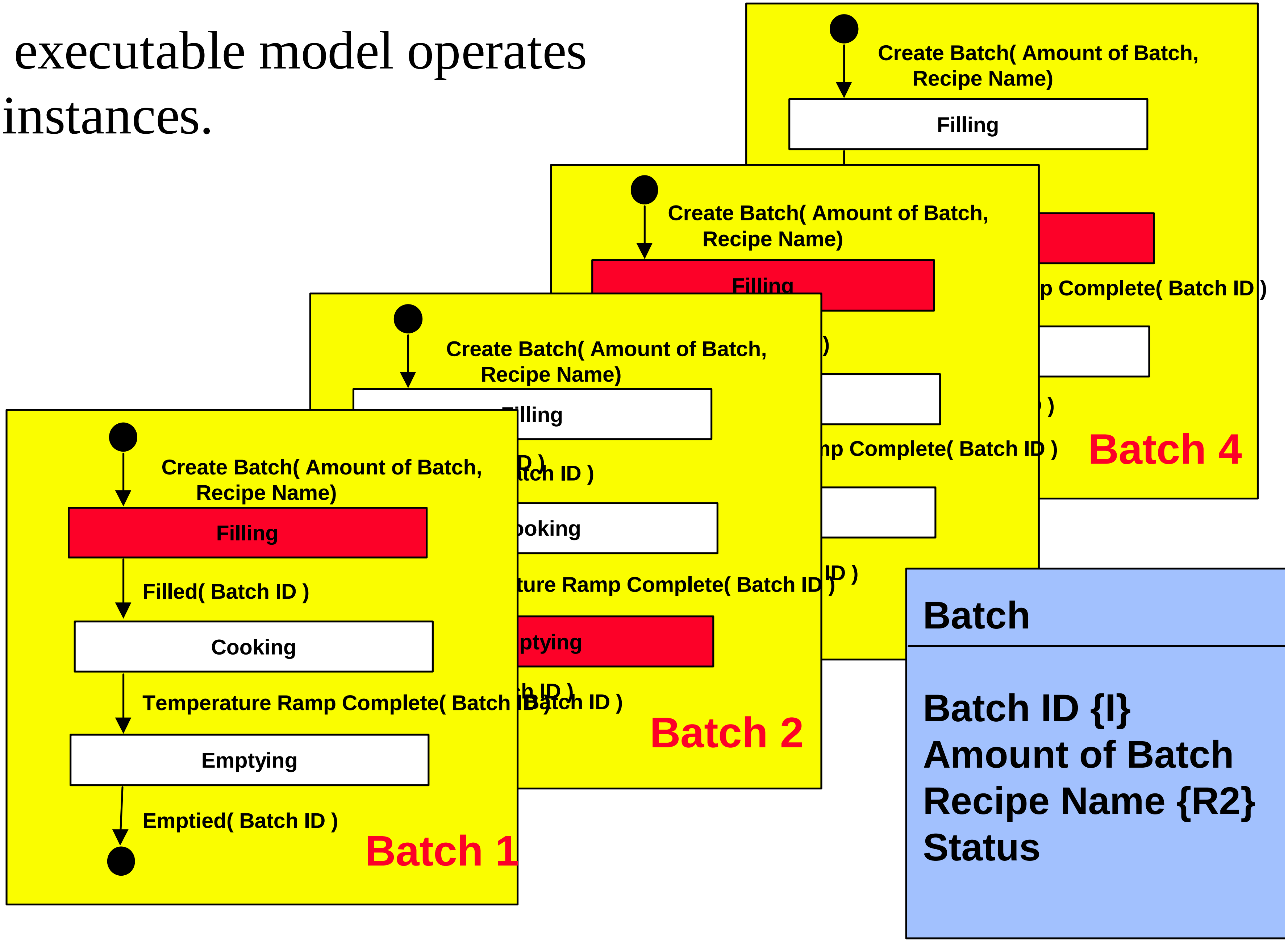
Recipe Name	Cooking Time	Cooking Temp	Heating Rate
Nylon	23	200	2.23
Kev			
St			

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

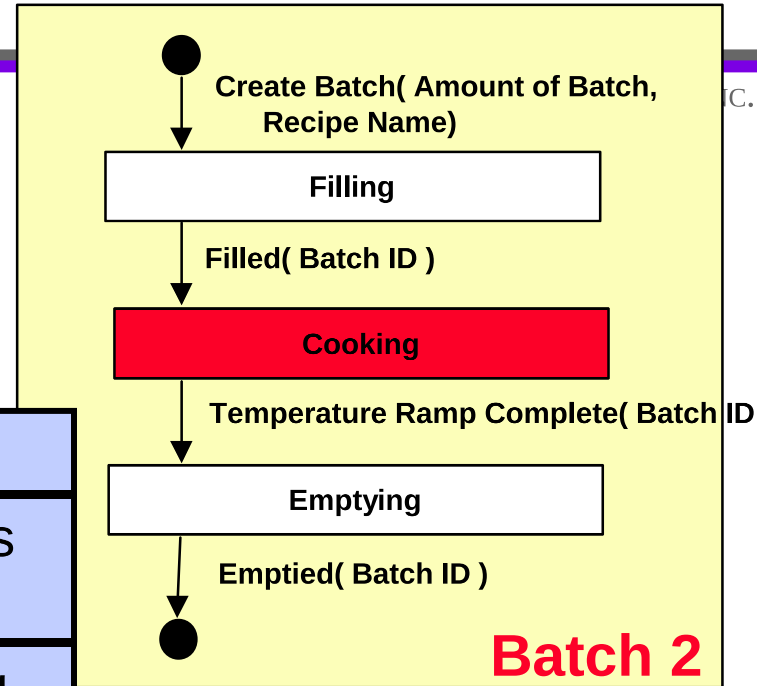
An executable model operates on instances.

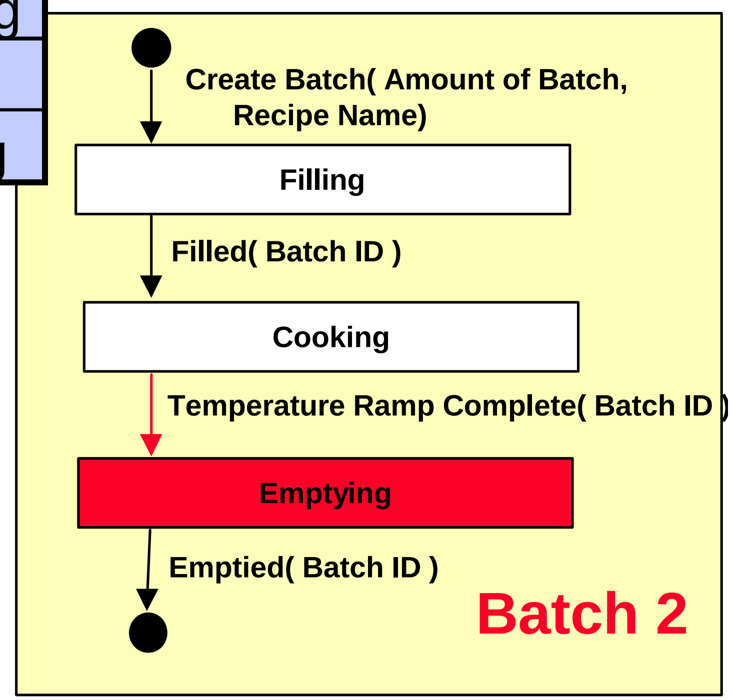


Execution

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



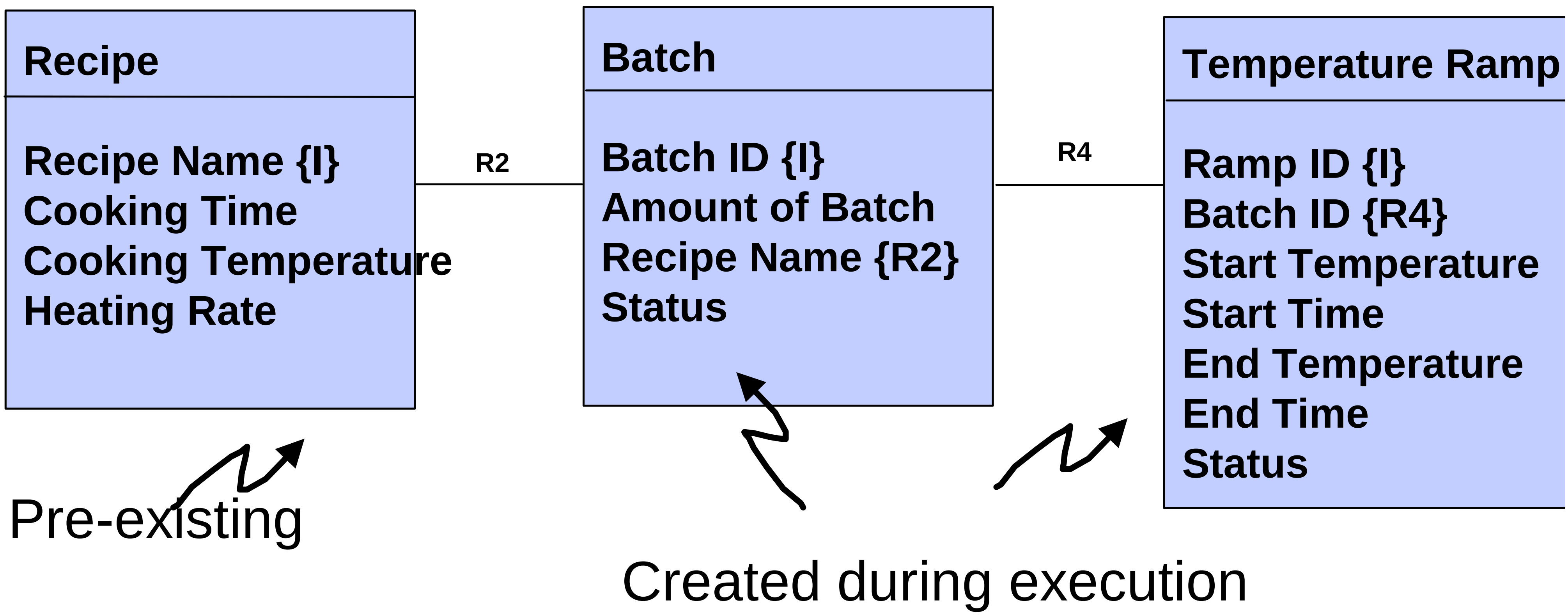


When the Temperature Ramp is complete, the instance moves to the next state....and executes actions.

Pre-existing Instances

PROJECT TECHNOLOGY, INC.

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



Initialization

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

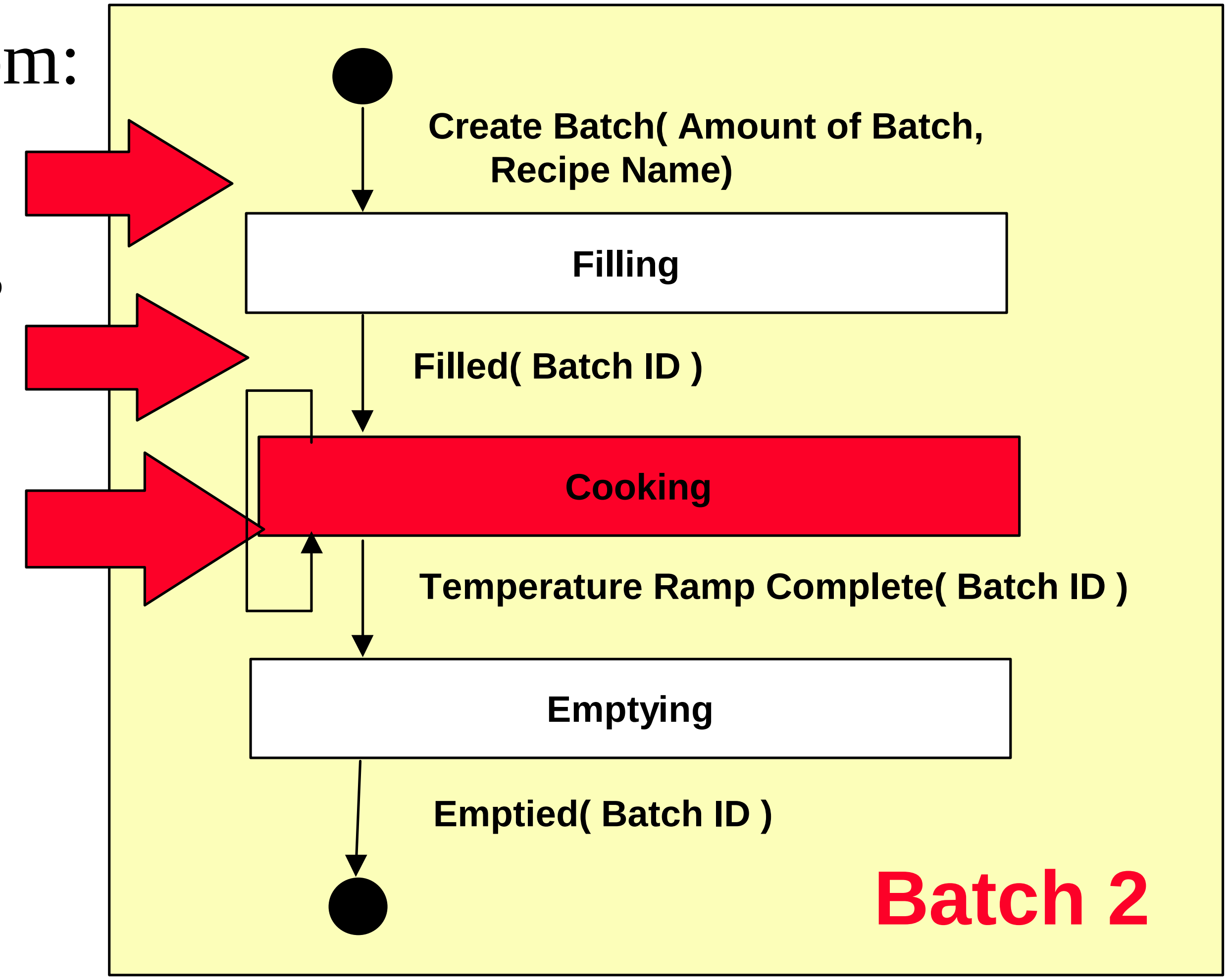
...and so require initialization.

Recipe			
Recipe Name	Cooking Time	Cooking Temp	Heating Rate
Nylon	23	200	2.23
Kevlar	45	250	4.69
Stuff	67	280	1.82

Executing the Model

The model executes in response to signals from:

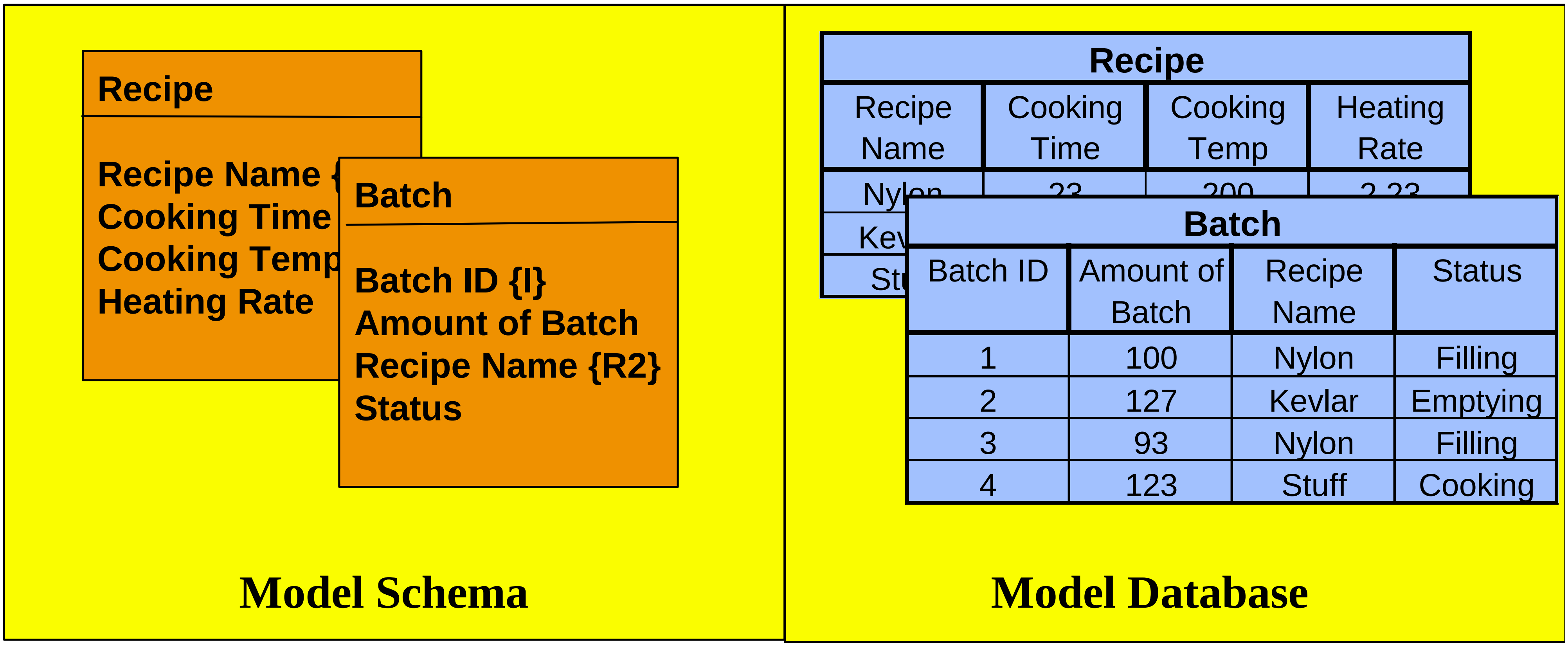
- the outside,
- other instances as they execute
- timers



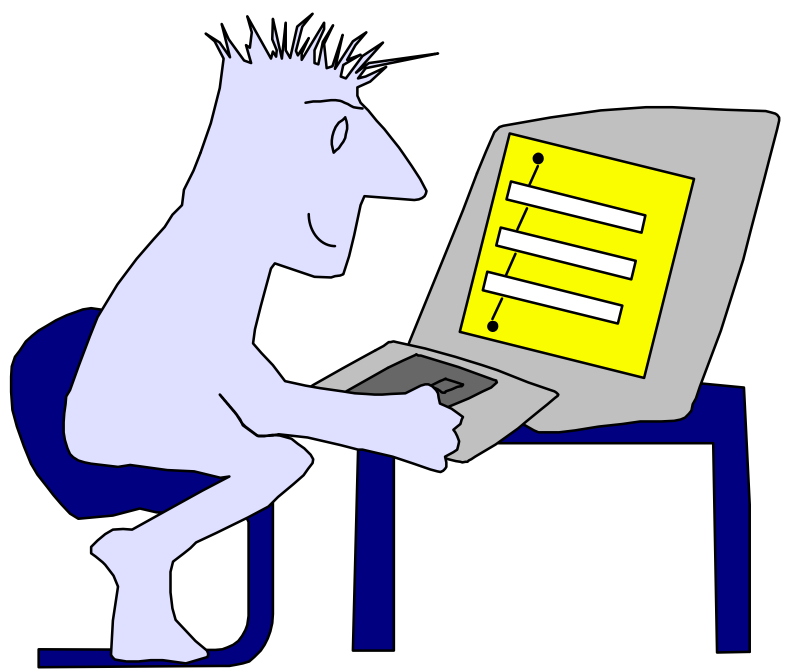
Model Database

PROJECT TECHNOLOGY, INC.

Each schema has a corresponding database for instances.



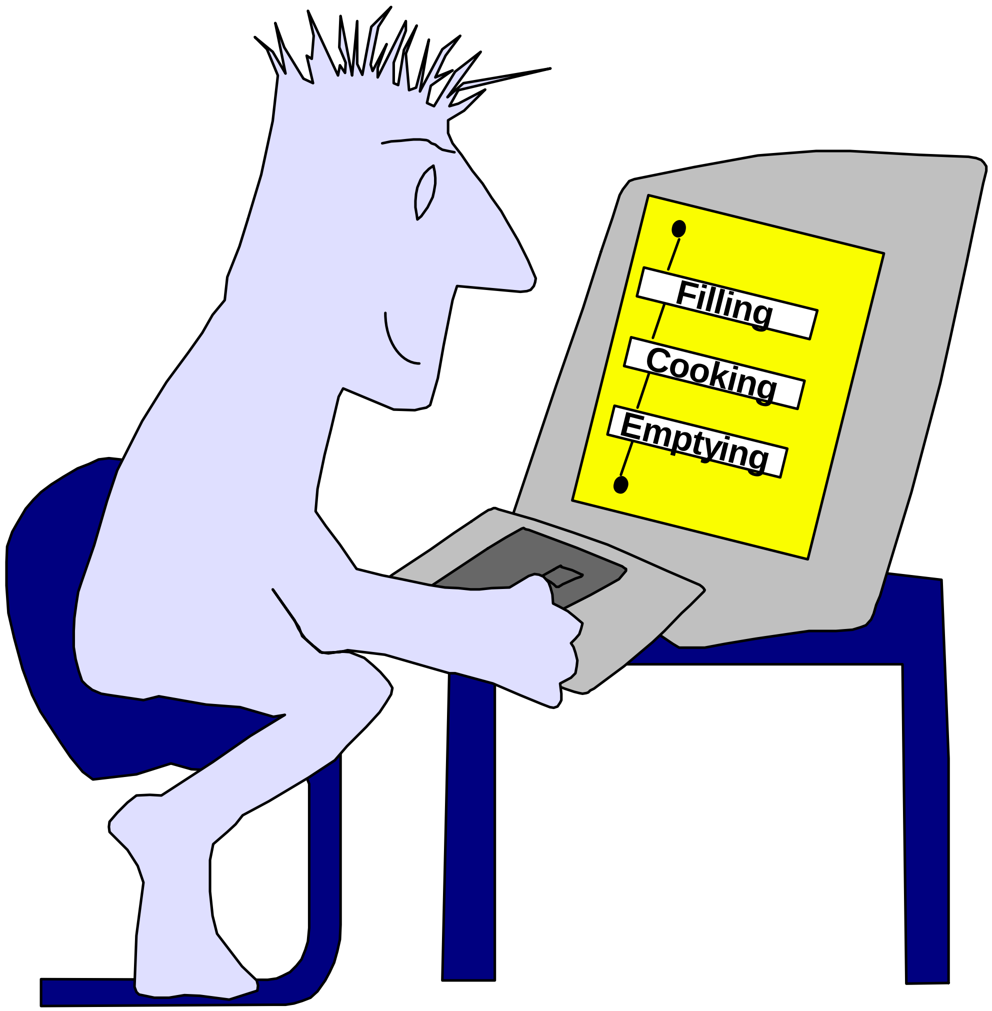
Capturing The Models



Model Repository

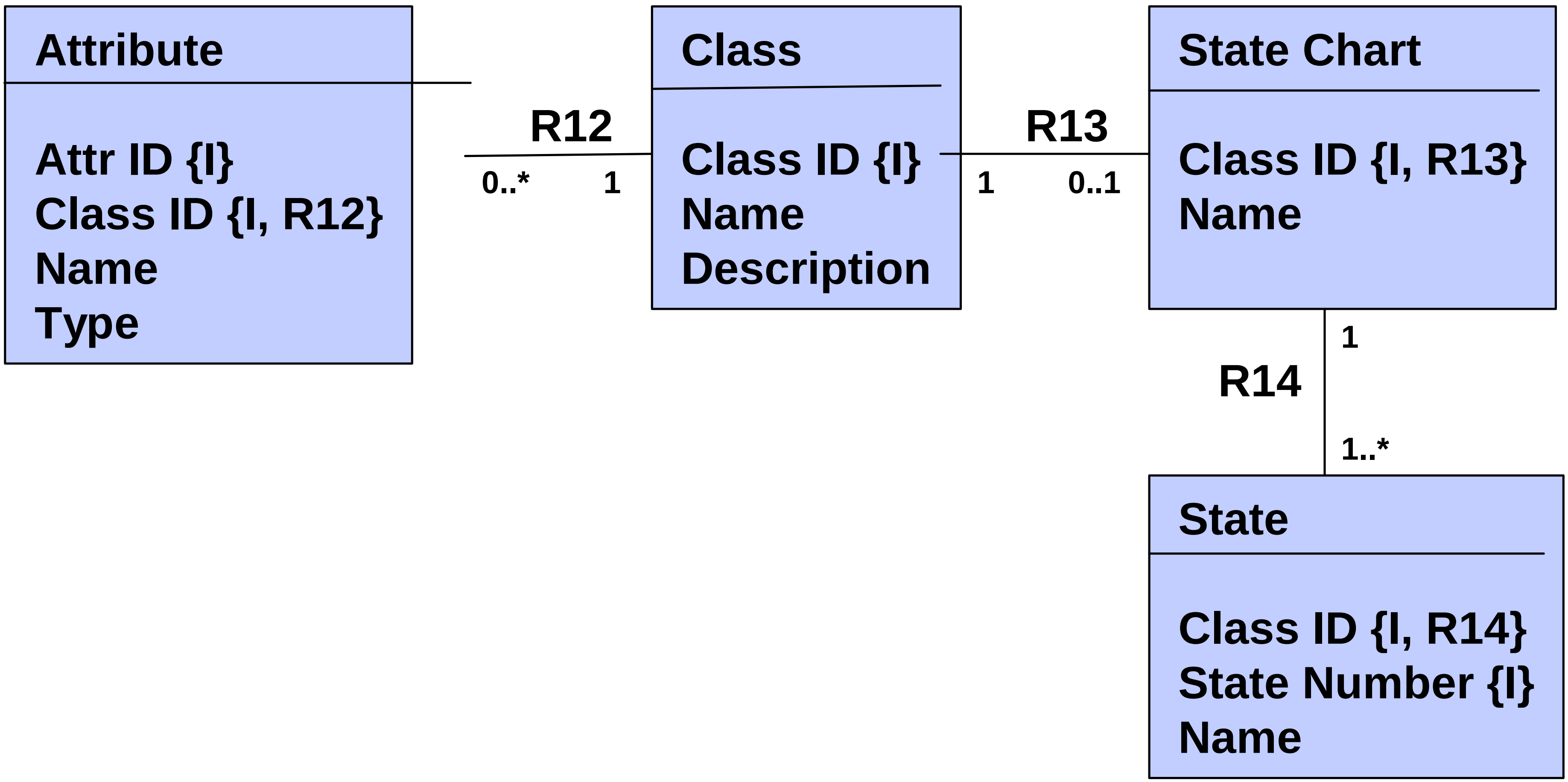
Capture the model in a model repository.

What is the structure of the repository?



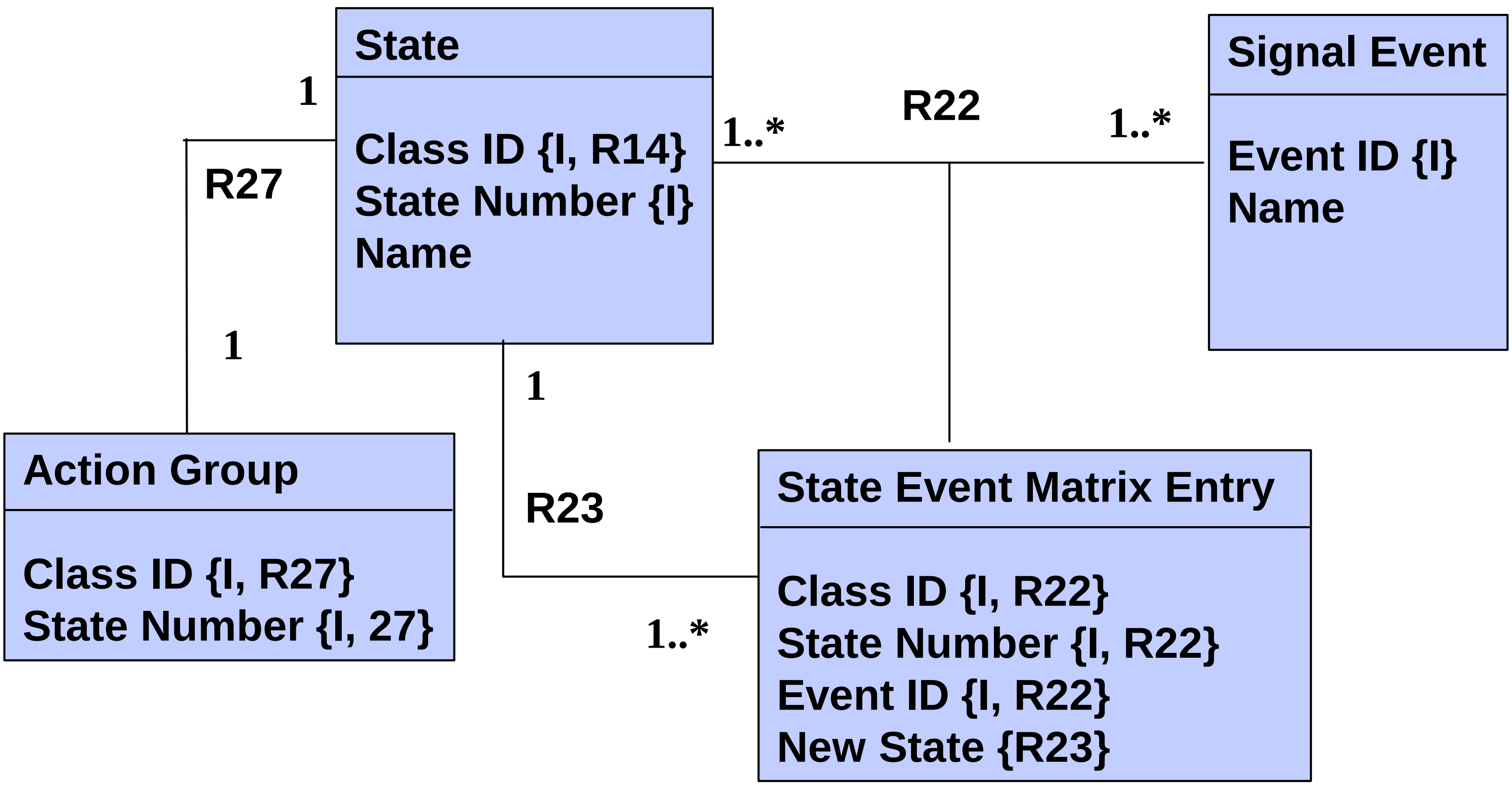
Model Structure

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



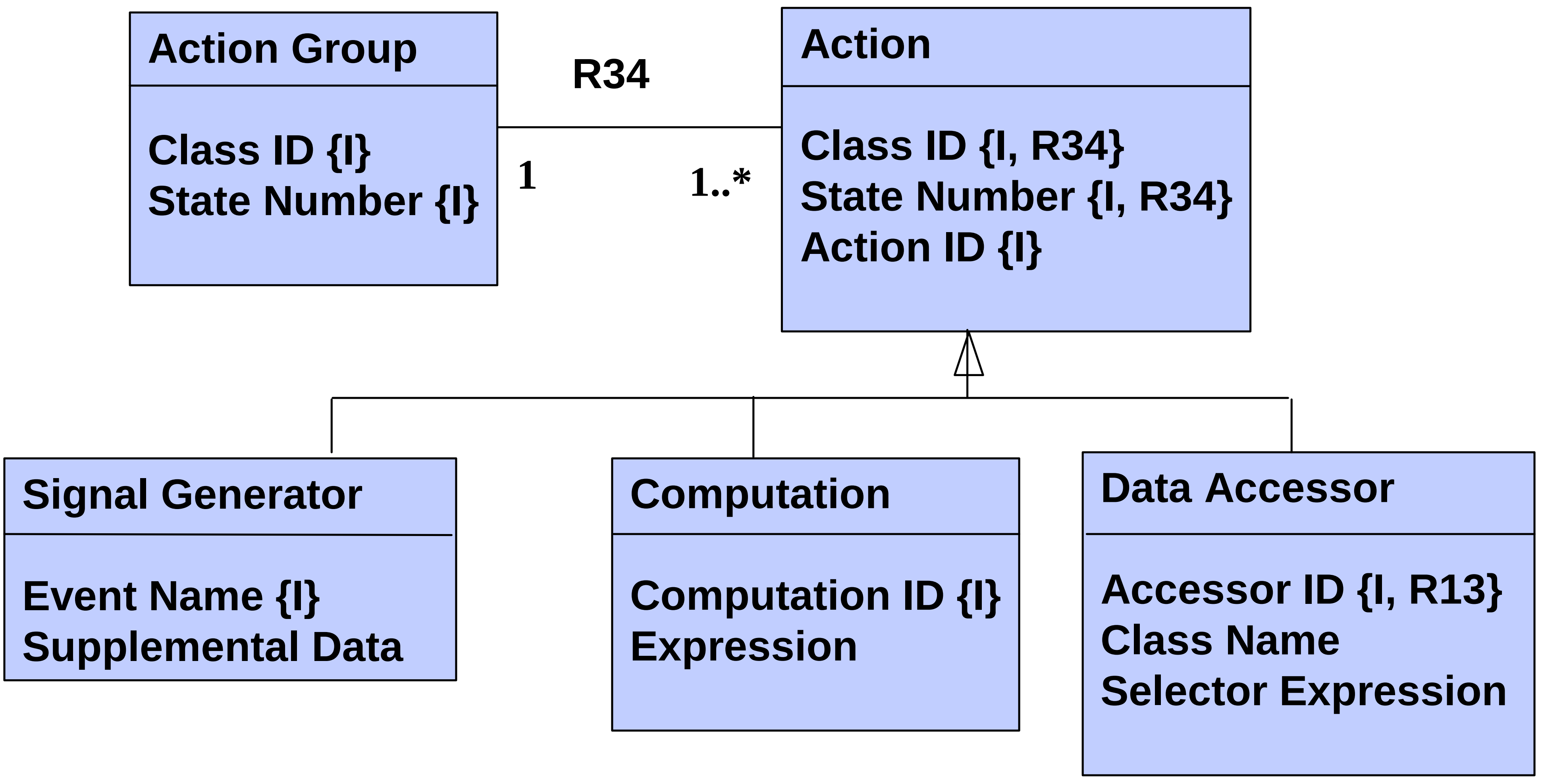
Model Structure

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



Model Structure

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

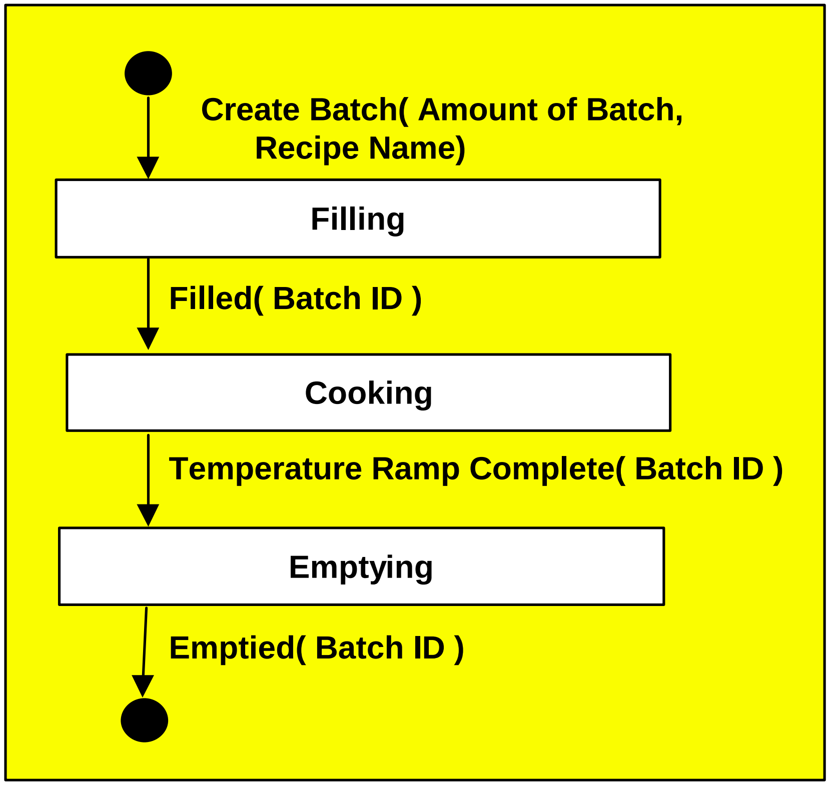


Meta-Model Instances

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

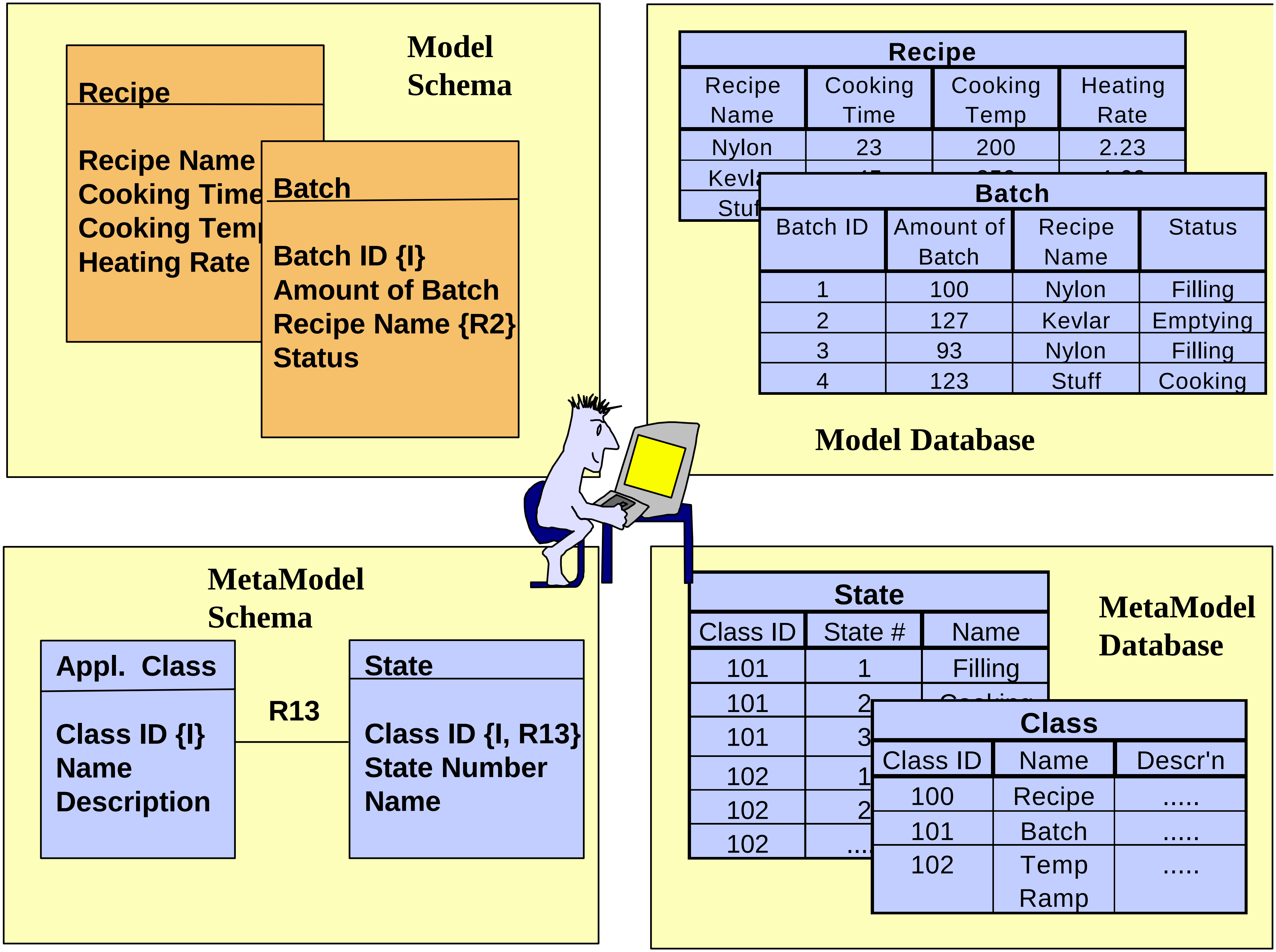
Class

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



State

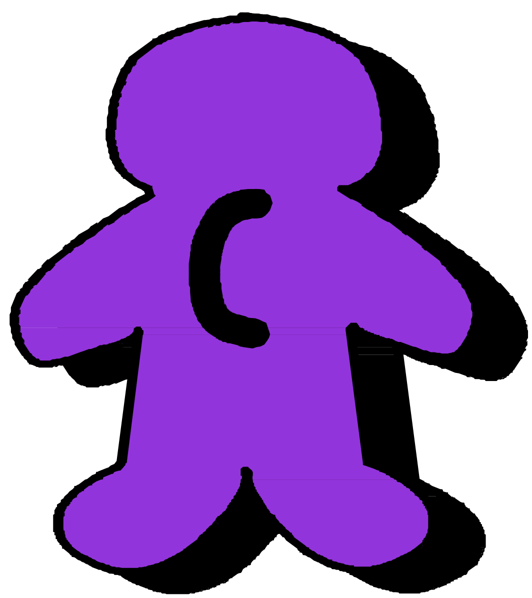
Class ID	State #	Name
101	1	Filling
101	2	Cooking
101	3	Emptying
102	1	....
102	2	.....
102	.....	.....


Model Schema
Recipe
Recipe Name
Cooking Time
Cooking Temp
Heating Rate
Batch
Batch ID {I}
Amount of Batch
Recipe Name {R2}
Status
Recipe
Recipe Name
Cooking Time
Cooking Temp
Heating Rate
Nylon
23
200
2.23
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
Model Database
MetaModel Schema
Appl. Class
Class ID {I}
Name
Description
R13
State
Class ID {I, R13}
State Number
Name
State
Class ID
State #
Name
101
1
Filling
101
101
102
102
102
MetaModel Database
Class
Class ID
Name
Descr'n
100
Recipe
.....
101
Batch
.....
102
Temp Ramp
.....

PROJECT TECHNOLOGY INC.

Shlaer–Mellor Method ▪ BridgePoint

Archetype Language



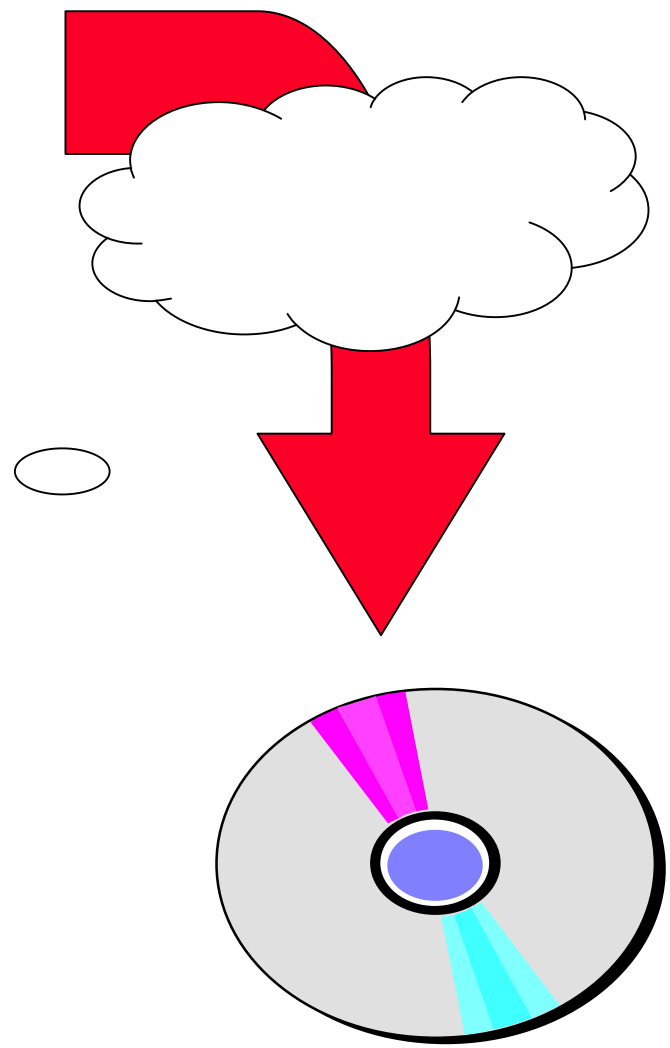
Purpose

To generate code….

MetaModel Database

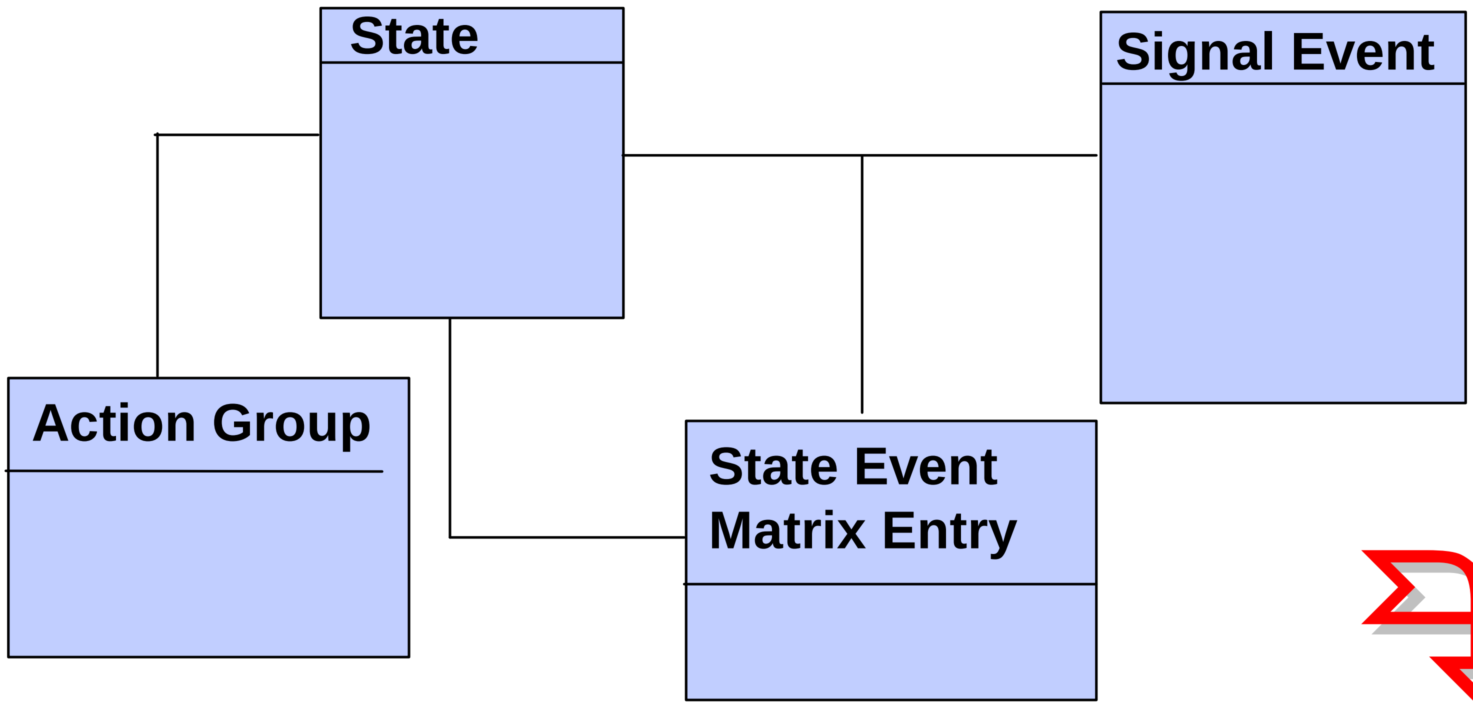
State		
Class ID	State #	Name
101	1	Filling
101	2	Cooking
101	3	
102	1	
102	2	
102	...	

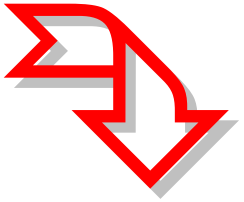
Class		
Class ID	Name	Descr'n
100	Recipe	.....
101	Batch	.....
102	Temp Ramp	.....



Purpose

….traverse the repository and...





… output text.

Example

The archetype language produces text.

```
.select many stateS related to instances of
      class->State->StateChart
           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 <u>Class</u>

Instance reference

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

Substitution

To access attributes of the selected instance….

${*class*.Name}

Batch

just text

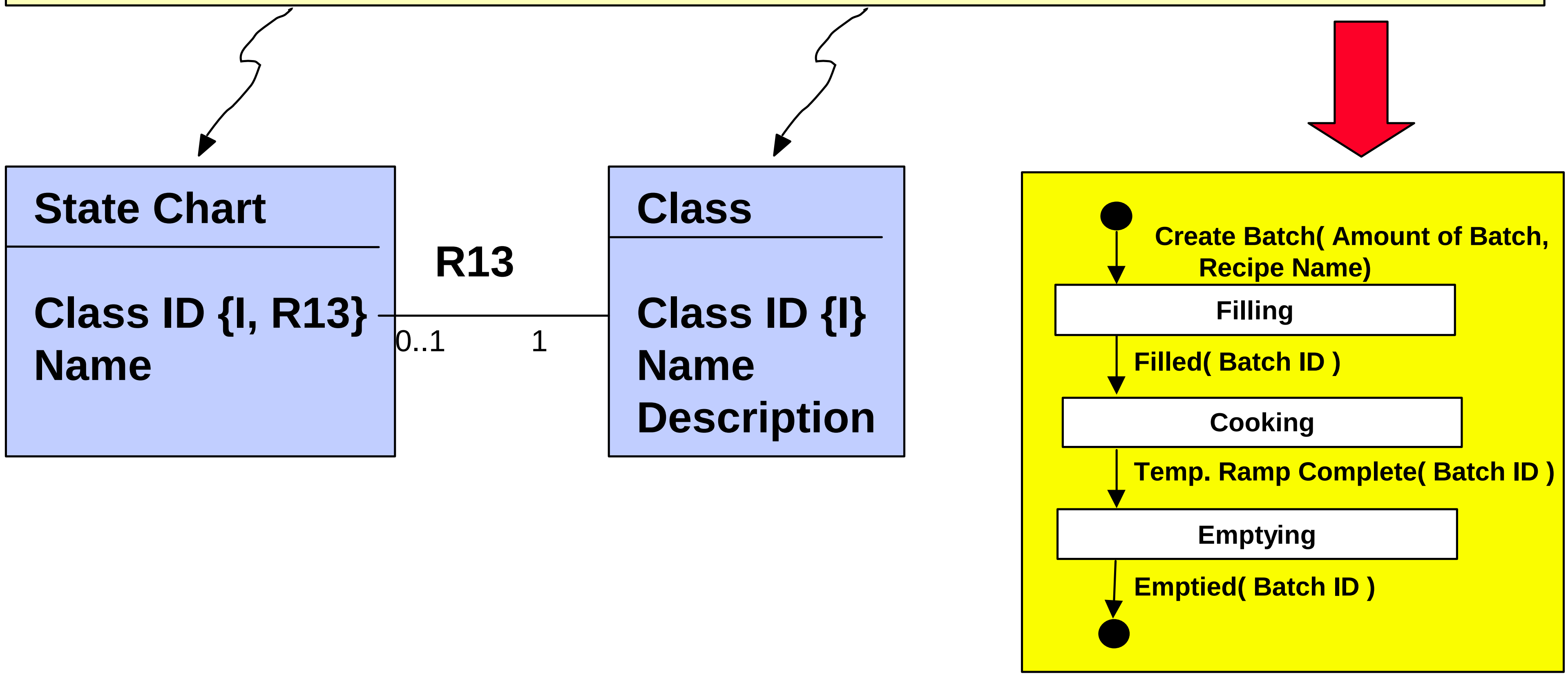
Class		
Class ID	Name	Descr'n
100	Recipe	.....
101	Batch	.....
102	Temp Ramp	.....

Association Traversal

To traverse an association…..

Not just any one--
the one that’s associated

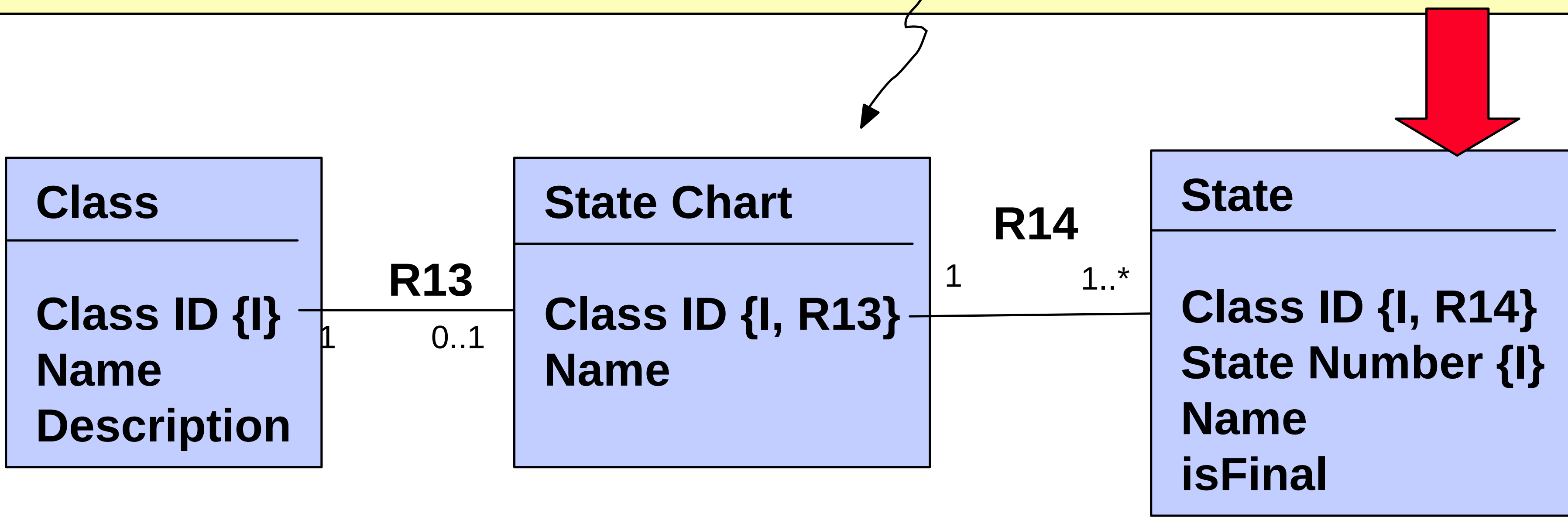
.select one *StateChart* related to instances of *class*->StateChart



Arbitrary Instance

To select an arbitrary one….

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



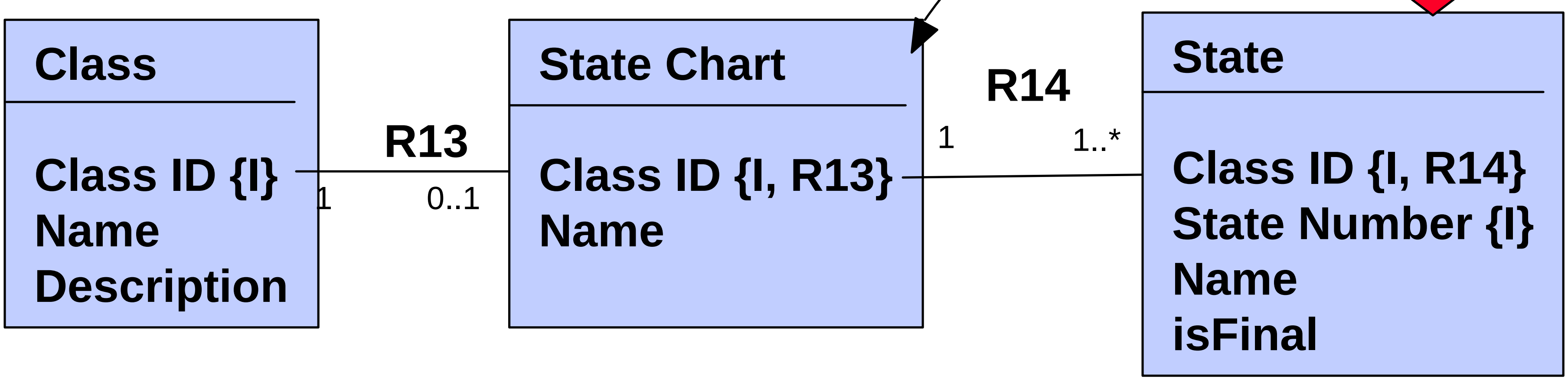
Or...

.select any *state* related to instances of *Class*->StateChart->State

Complex Traversals

To qualify the selection...

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



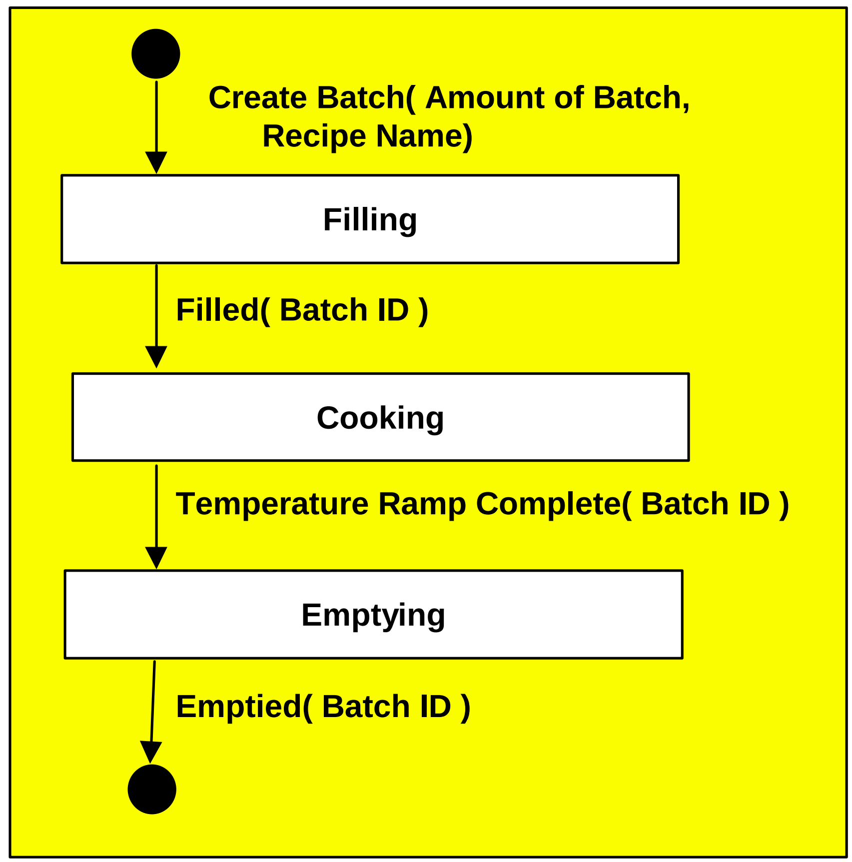
Instance Sets

To select many instances:

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

StateS =

Filling
Cooking
Emptying



Iteration

To iterate over instances…

.select many *stateS* related to instances of *Class*->StateChart -> 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->StateChart-> 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
    };
```

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 **<u>code</u>**.

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 addPDMDecl( 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->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->StateChart
protected:
// state action member functions
    .select many stateS related by statechart->State
    .for each state in stateS
        .invoke addActionFunctionDecl( inst_ref state )
    .endfor
  .end function
```

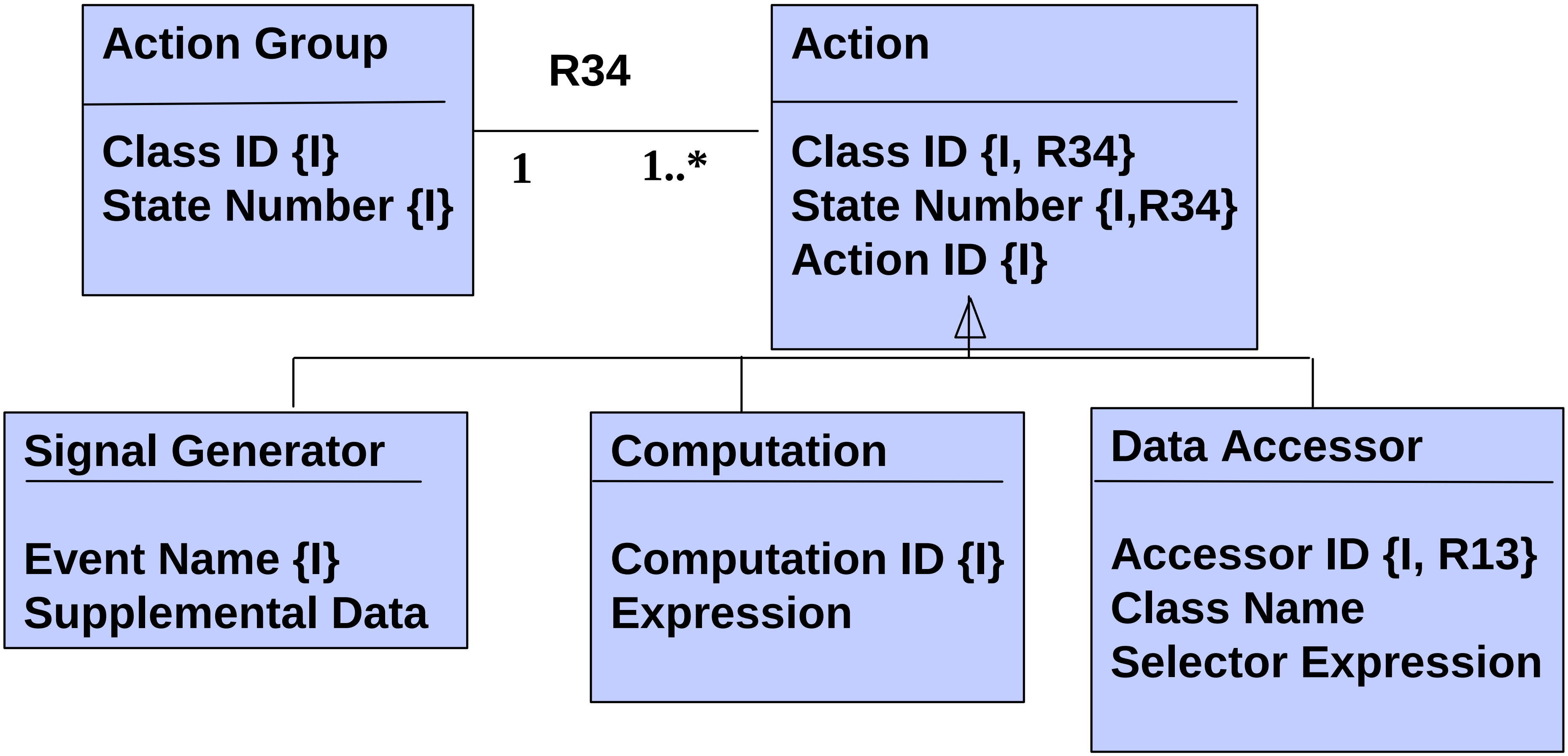
State Action Declaration

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
```

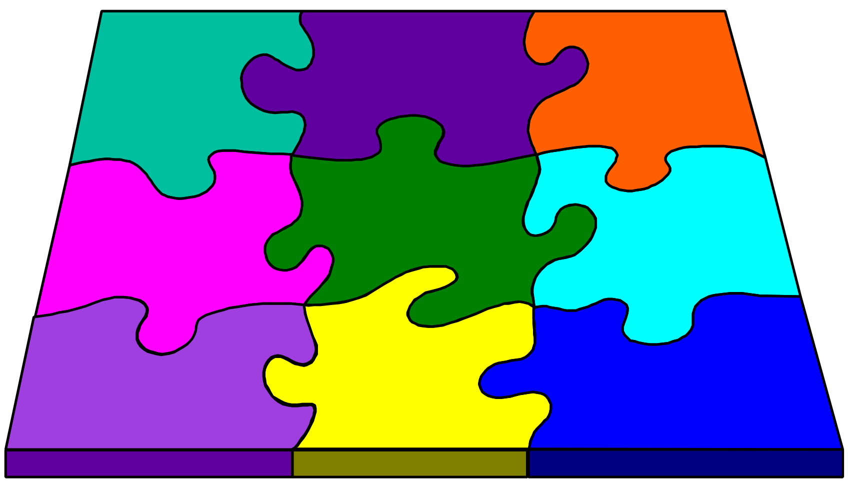
State Action Definition

To *define* the state action function….



…traverse the repository in the same manner.

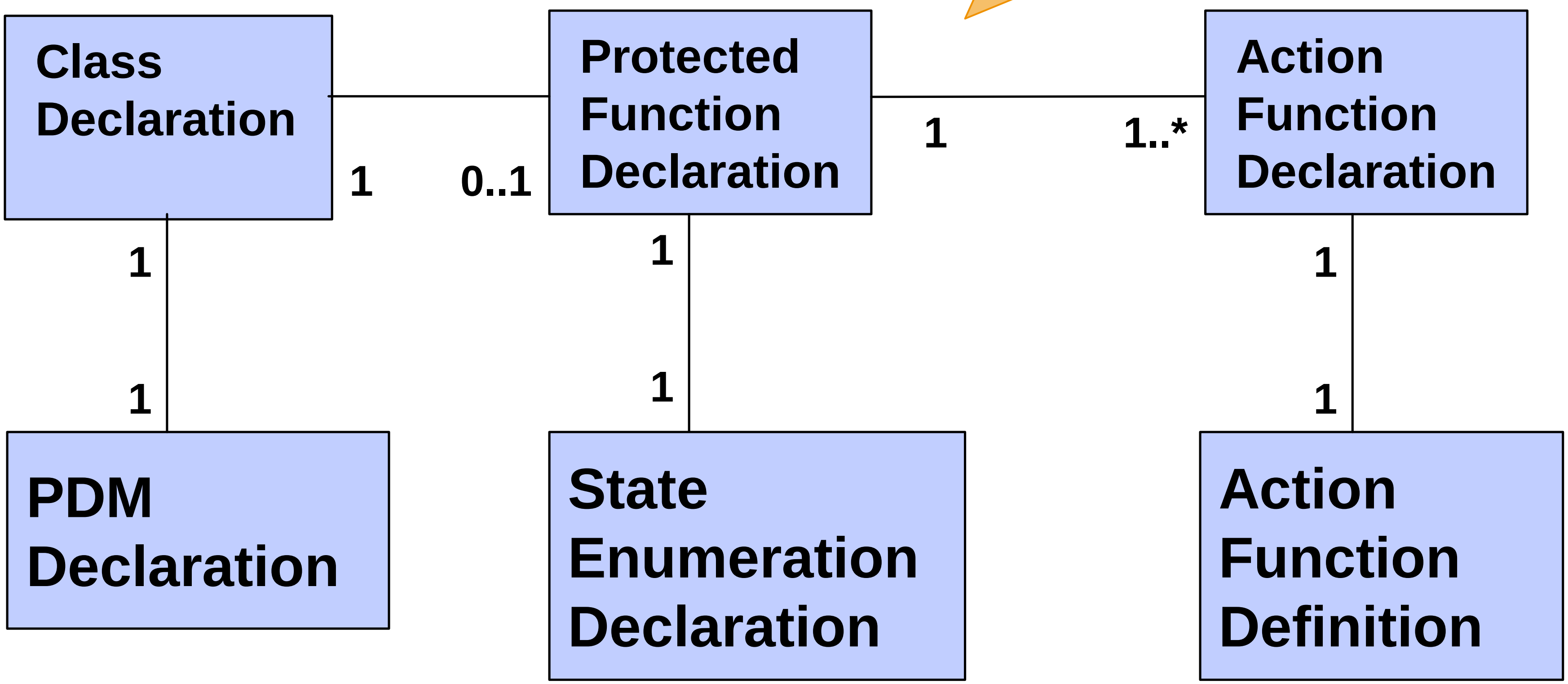
Specifying the Architecture



Model the Architecture

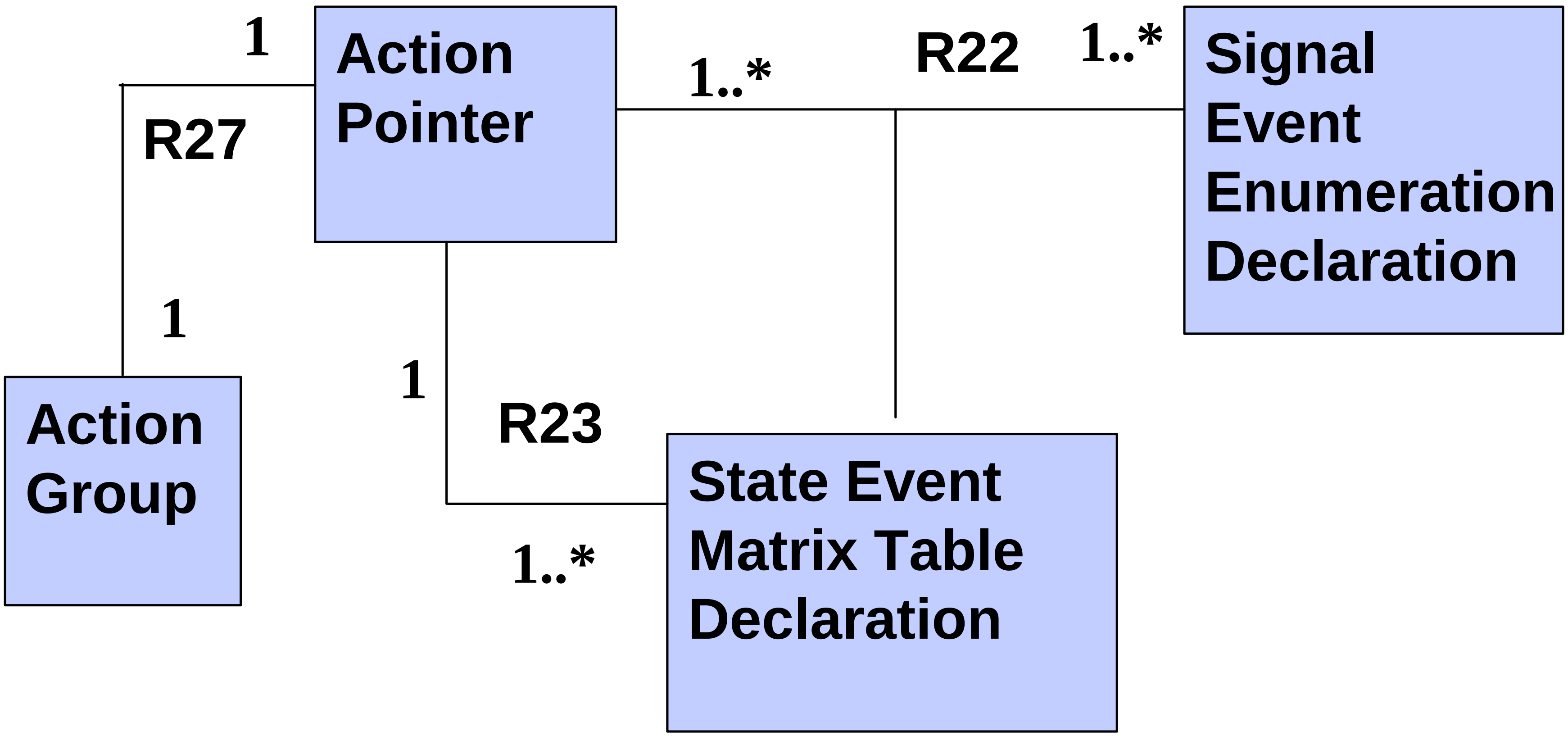
To specify the architecture, build a model of its conceptual entities.

Use the same approach to modeling the “design.”



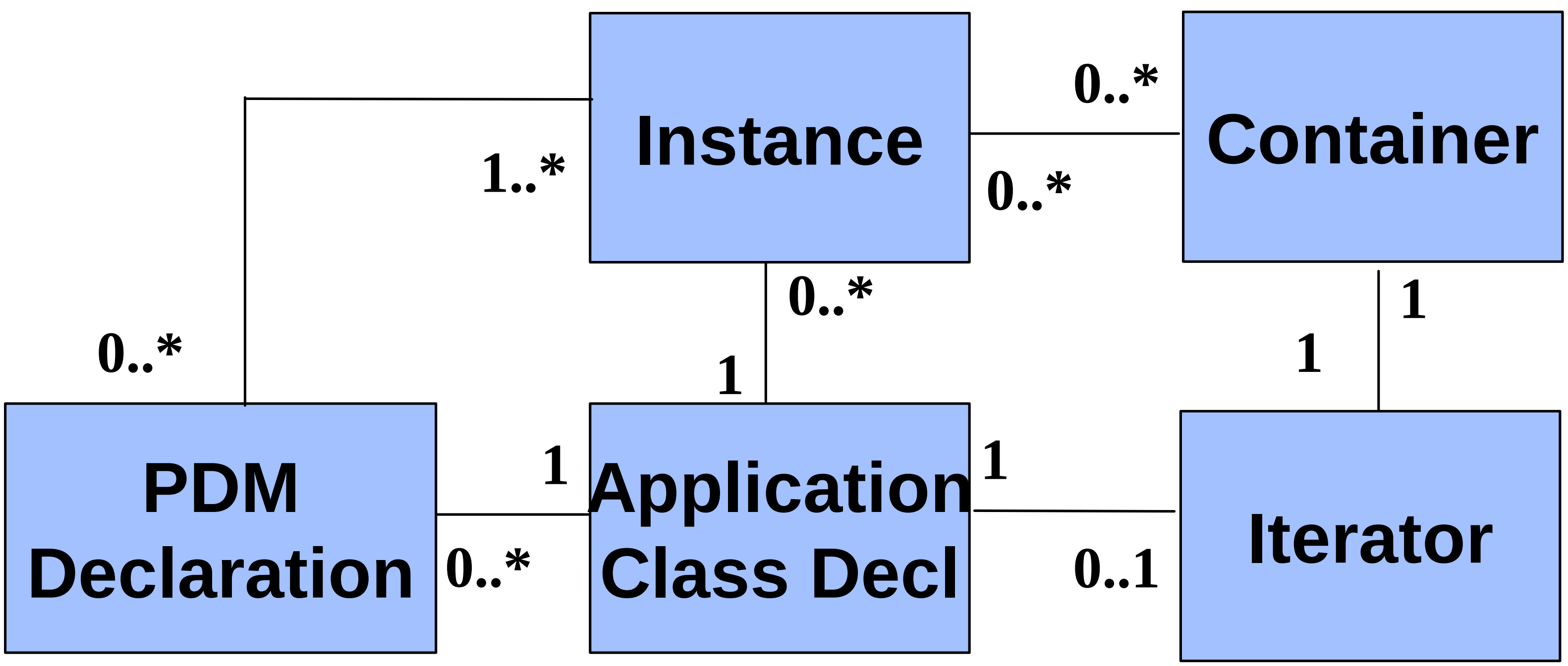
Model the Architecture

To specify the architecture, build a model of its conceptual entities.



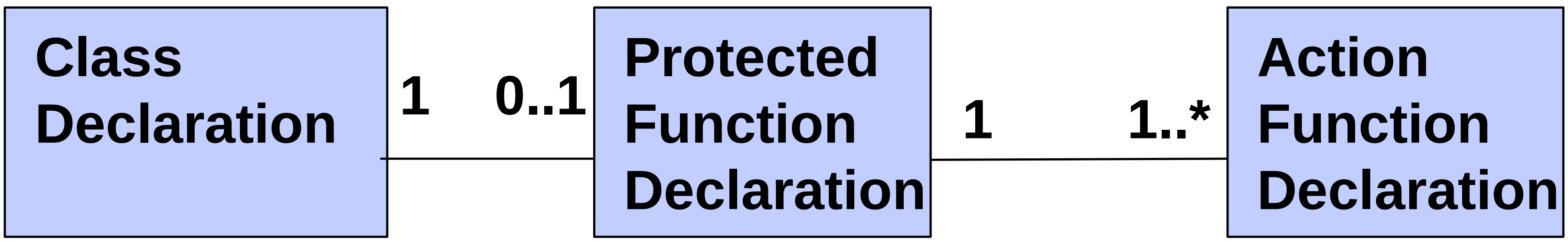
Example Architecture

The architecture specification should be very detailed--as well as “high-level.”



Archetypes

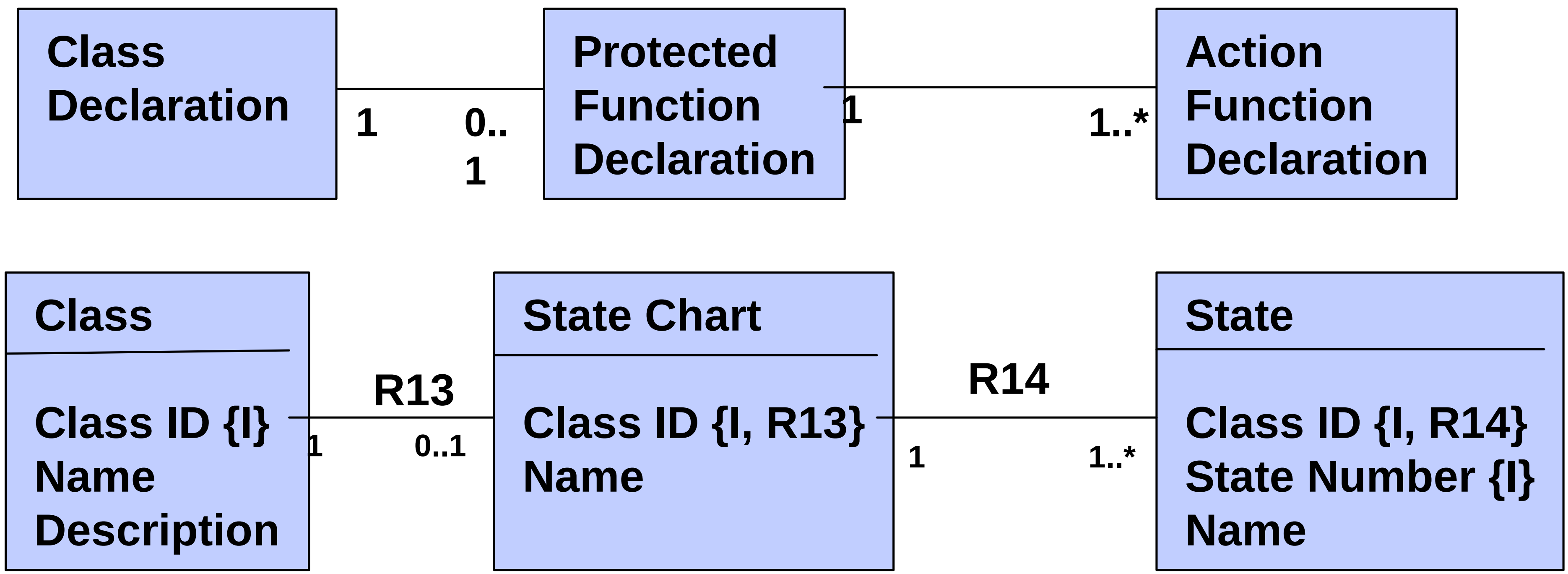
Build an archetype for each conceptual entity in the architecture.



.Function addClassDeclaration

.Function addProtectedFunctionDecl

Metamodel and Architecture Model



The models are similar because the architecture is a direct translation.

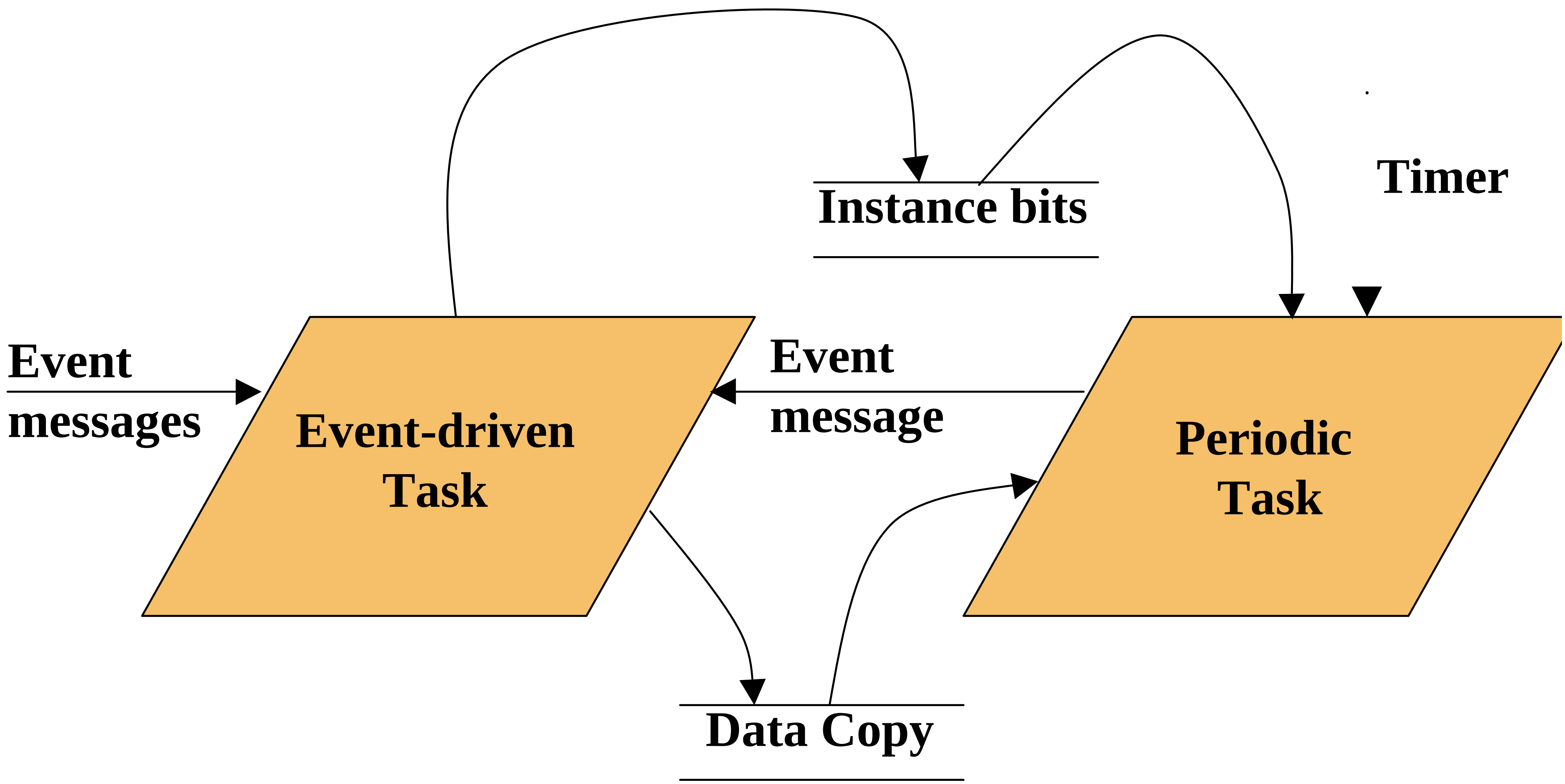
An Indirect Architecture

Description of Architecture

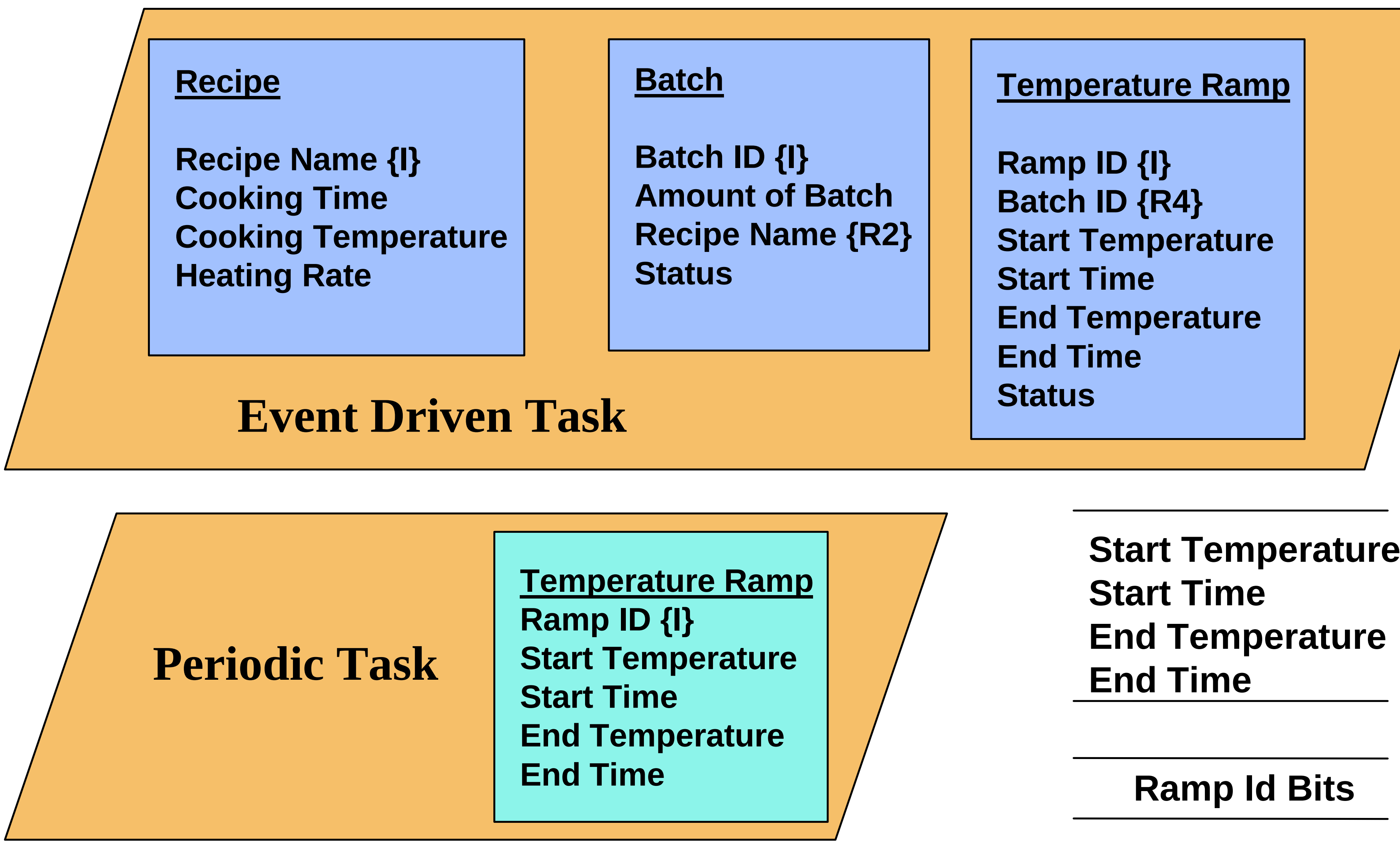
Because of the periodic nature of the system, we can build:

- two tasks,
- one of which is periodic and higher priority
- one bit per instance to indicate presence in the periodic state
- duplicated data needed for the control loop, and
- copied over by the periodic task when required by it

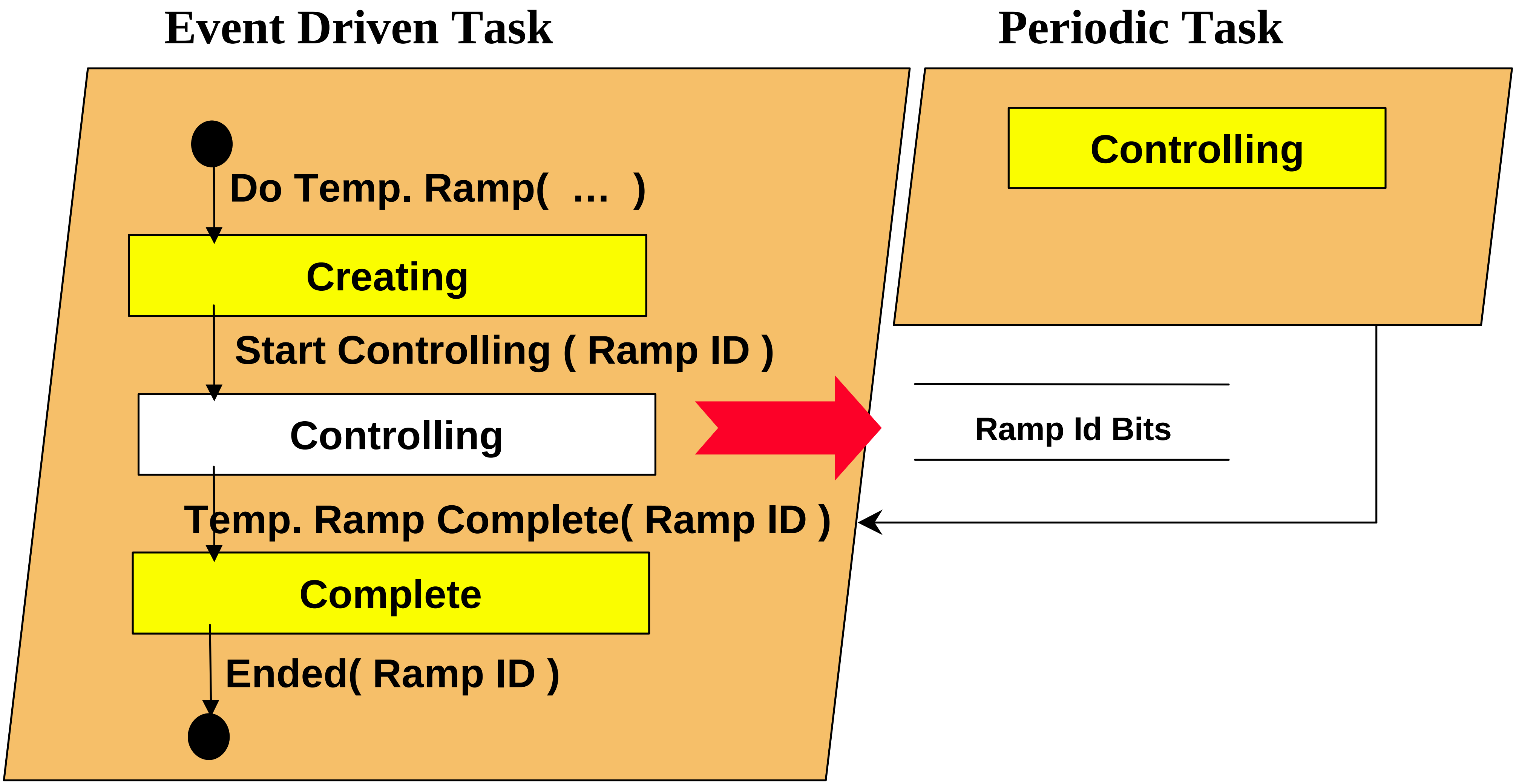
Description of Architecture



Application Mapping



Application Mapping



Extended Properties

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

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

State

Class ID {I, R14}
State Number {I}
Name
isFinal
isPeriodic

System Construction

Recap

At this point we have:

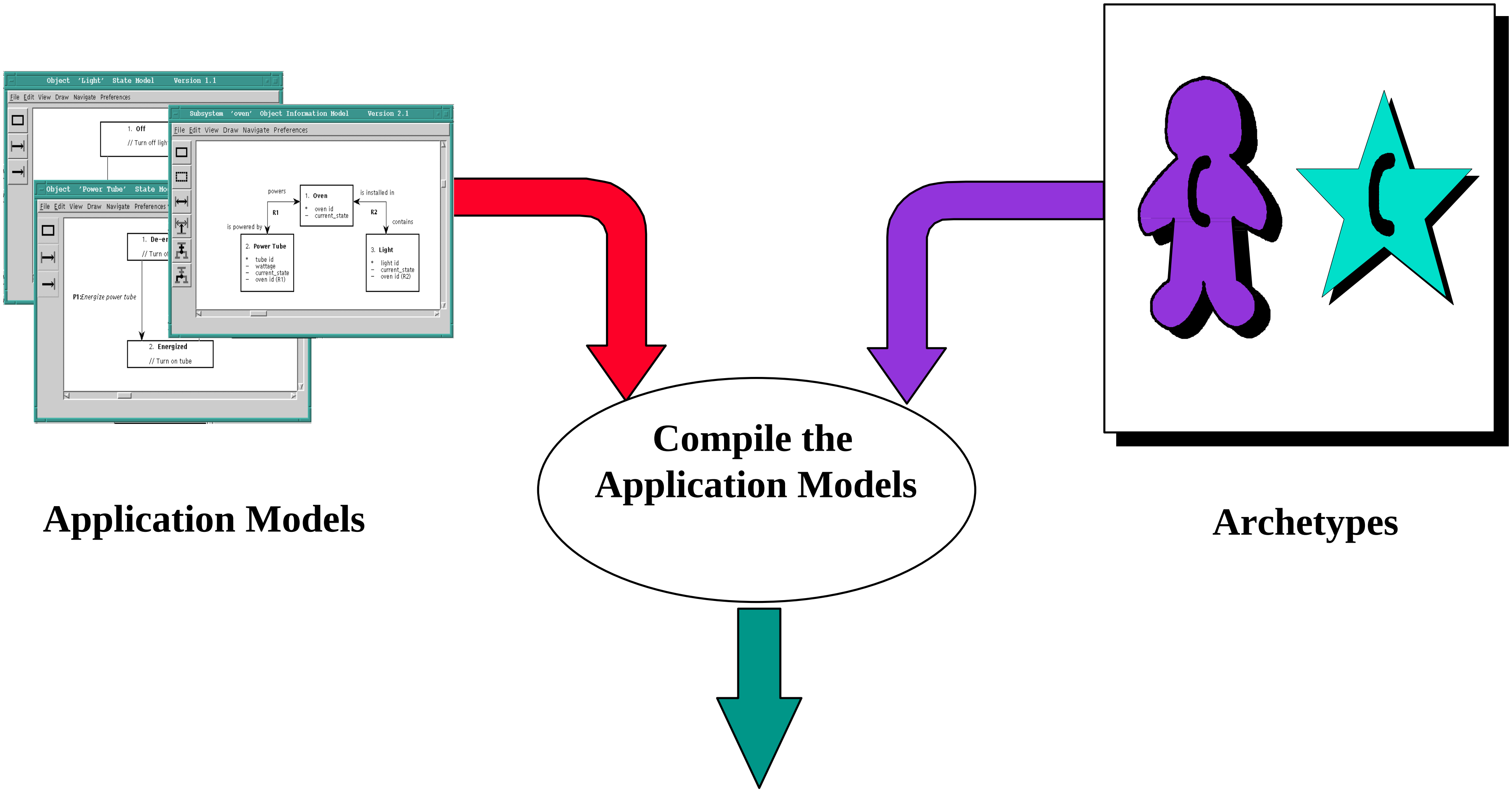
- a populated instance database for the application describing the system to be built
- archetypes for objects in the OOA of the architecture

What's next?

☞ Producing the executable code.

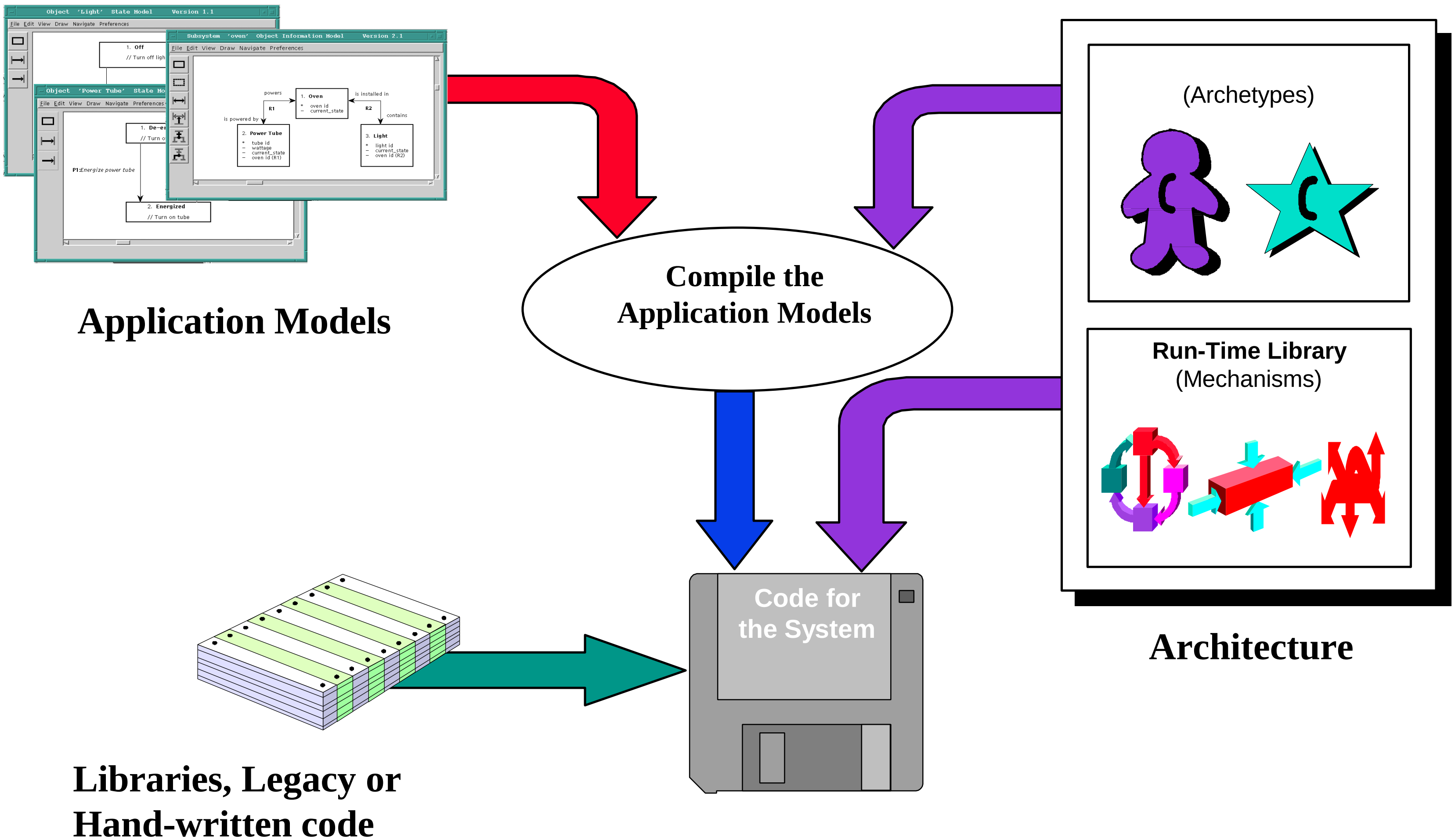
Generating the Production Code

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.

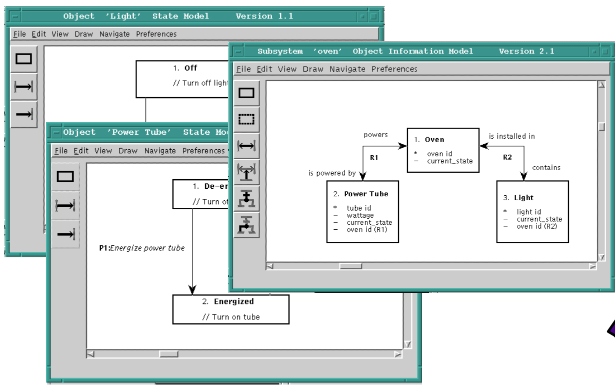


Model-Based Maintenance

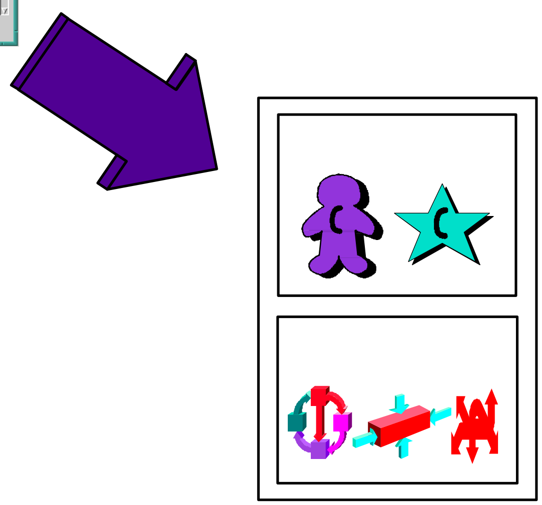
To address performance-based issues:

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

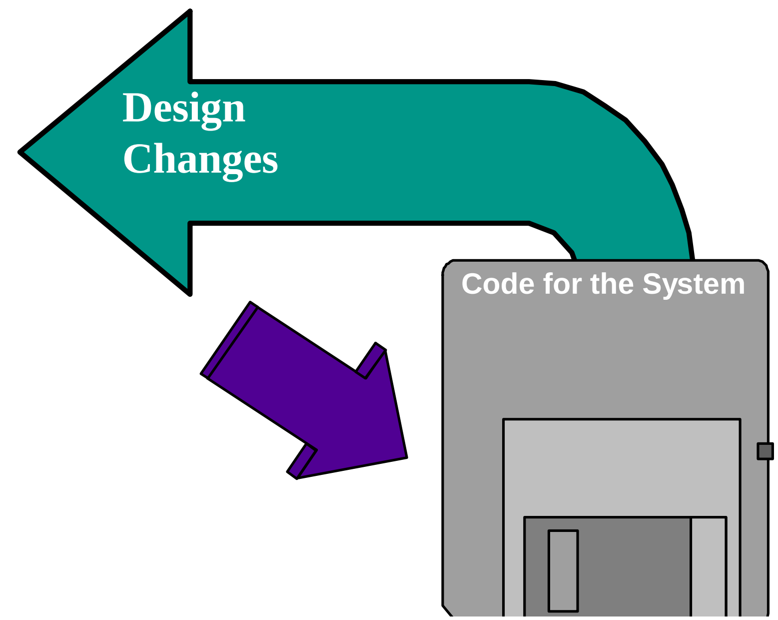
☞ Do not modify the generated code directly.



Application Models



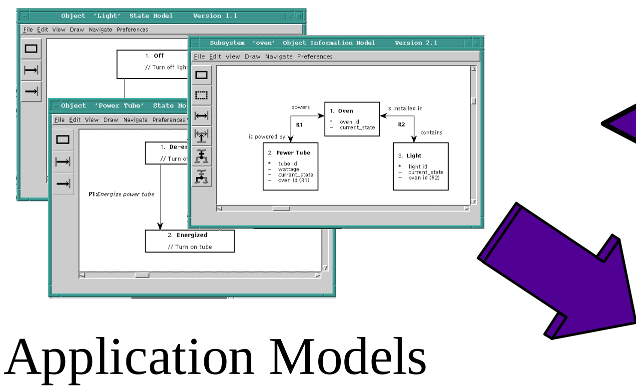
Architecture

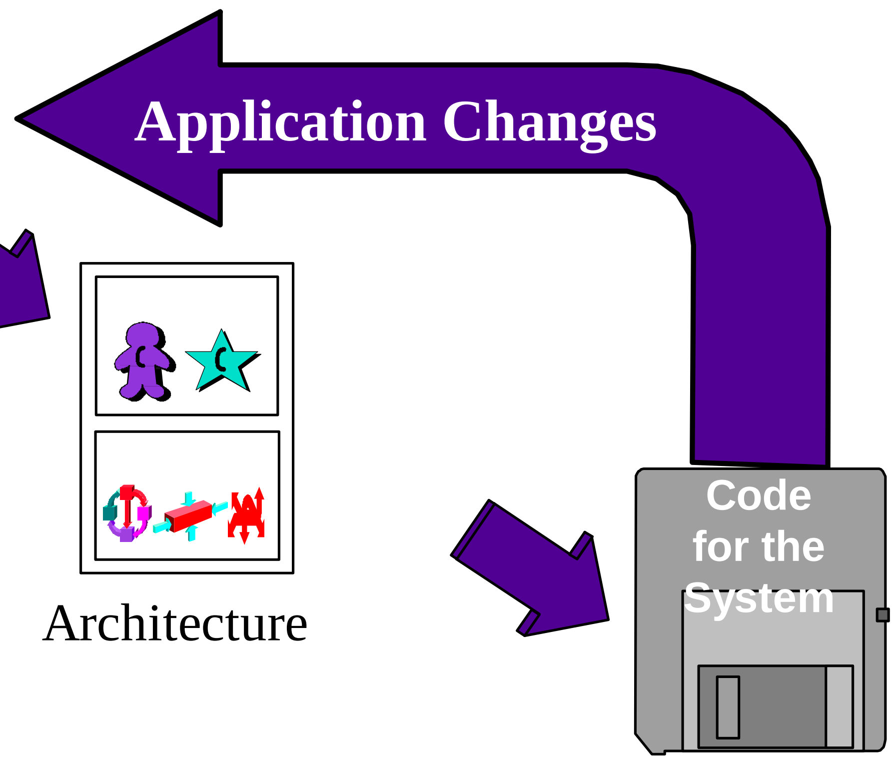


Model-Based Maintenance

To address application behavior issues,

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



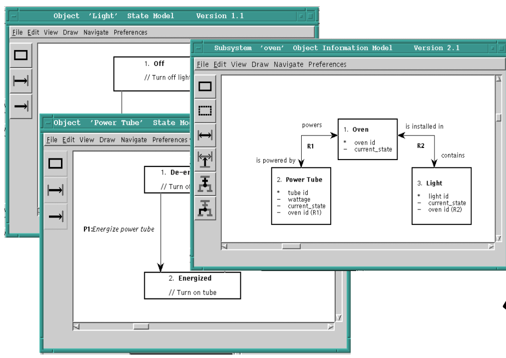


☞ Do not modify the generated code directly.

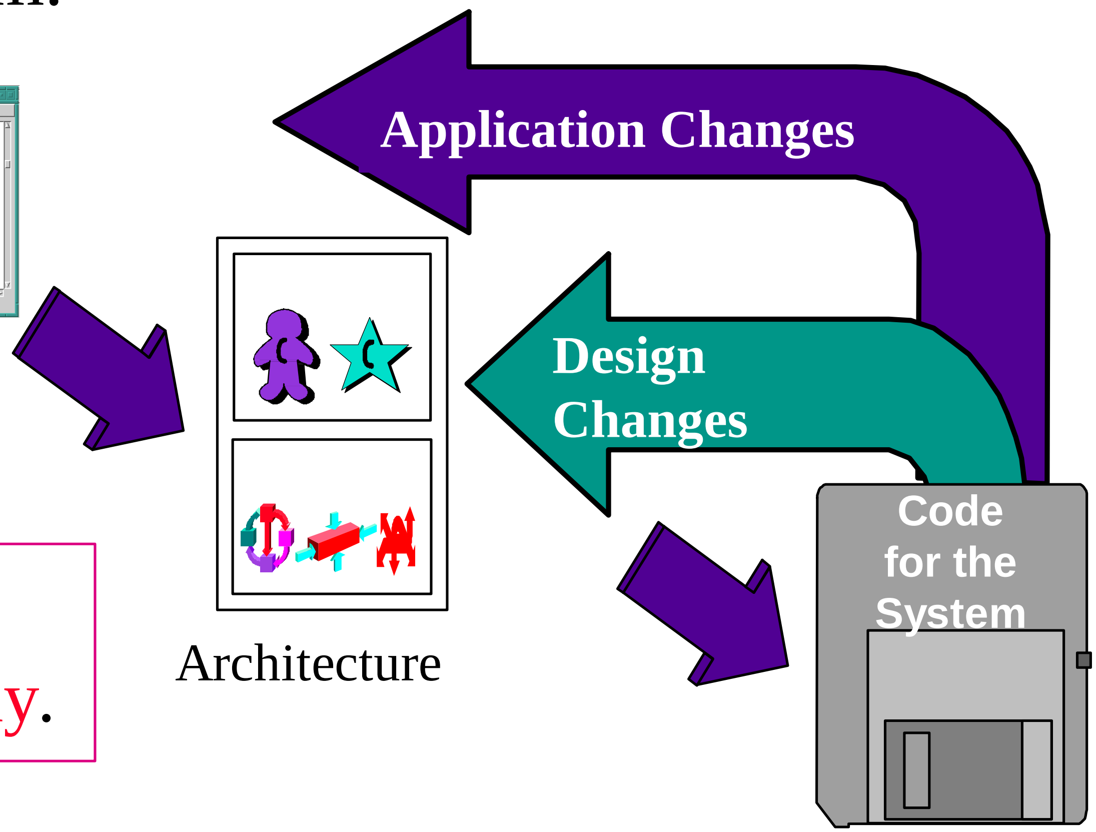
Model-Based Maintenance

For subsequent product enhancements,

- modify or replace the domain in question, and
- regenerate the system.



Application Models



☞ Do not modify the generated code directly.

Model Compiler

An architecture is an *OOA-model compiler*.

It translates a system specified in OOA 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.

The Shlaer-Mellor Method

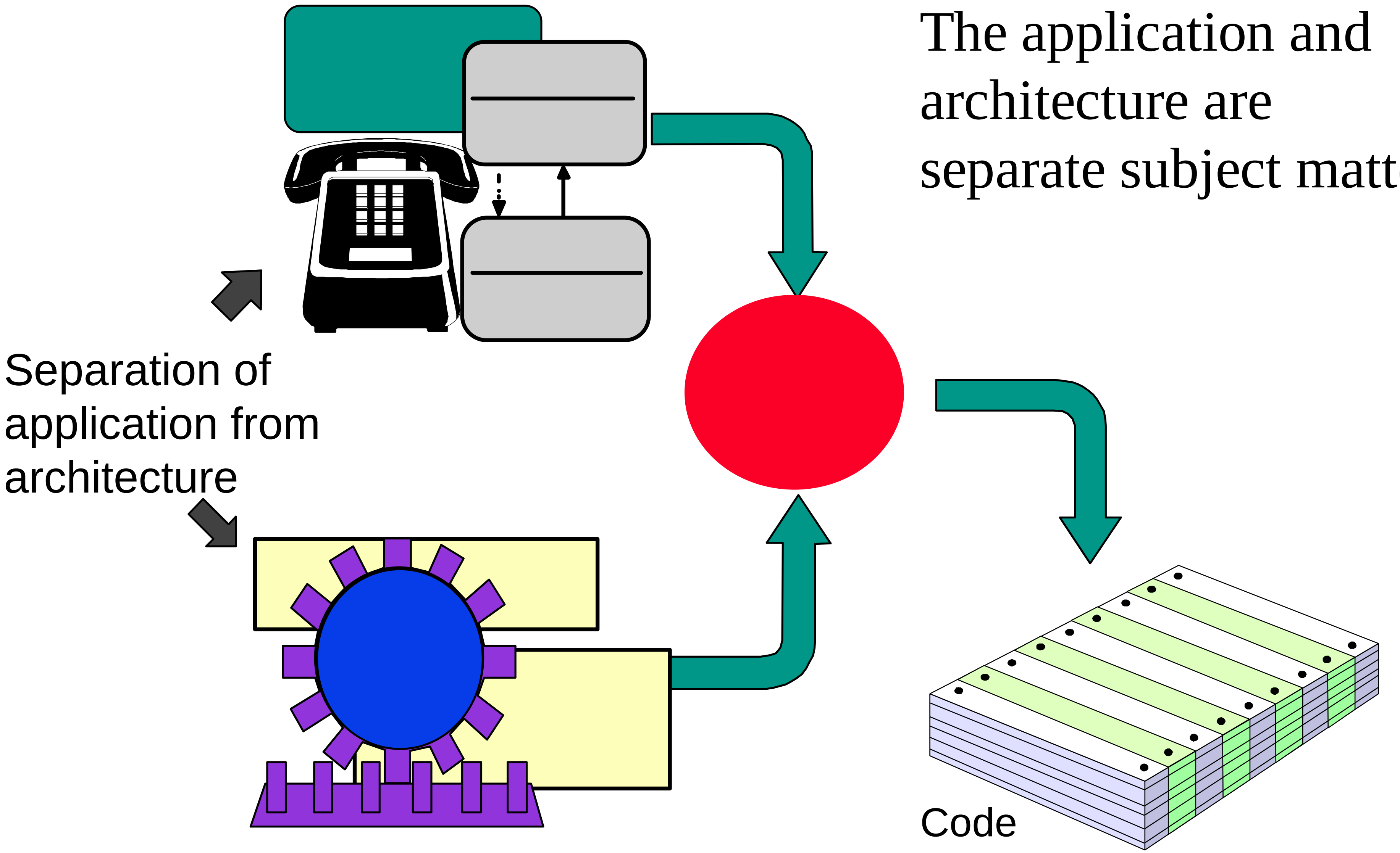
The Shlaer-Mellor Method

The Shlaer-Mellor Method is a software-construction method based on:

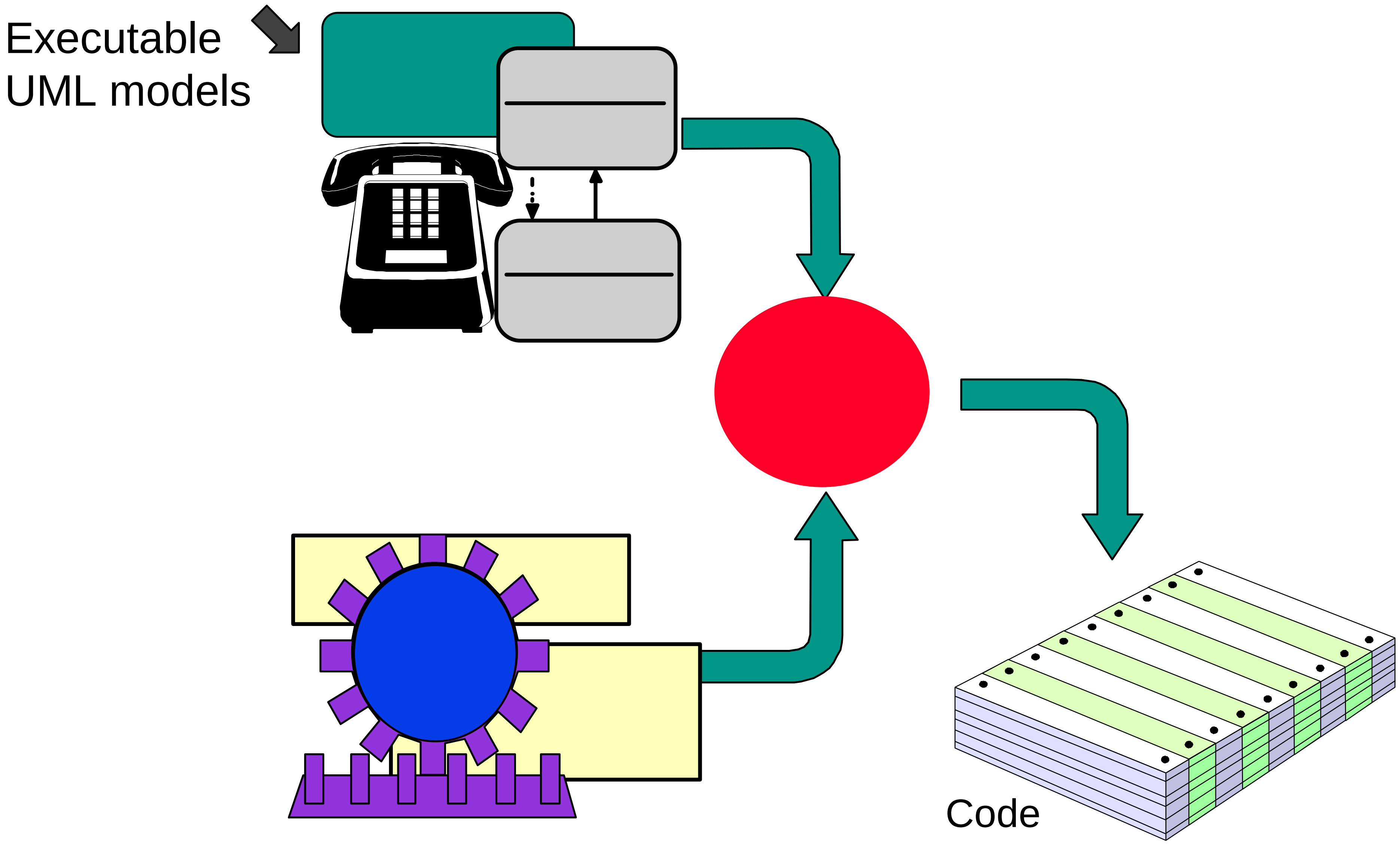
- separating systems into subject matters (domains)
- specifying each domain with an executable OOA model
- translating the models

Subject Matter Separation

The application and architecture are separate subject matters.

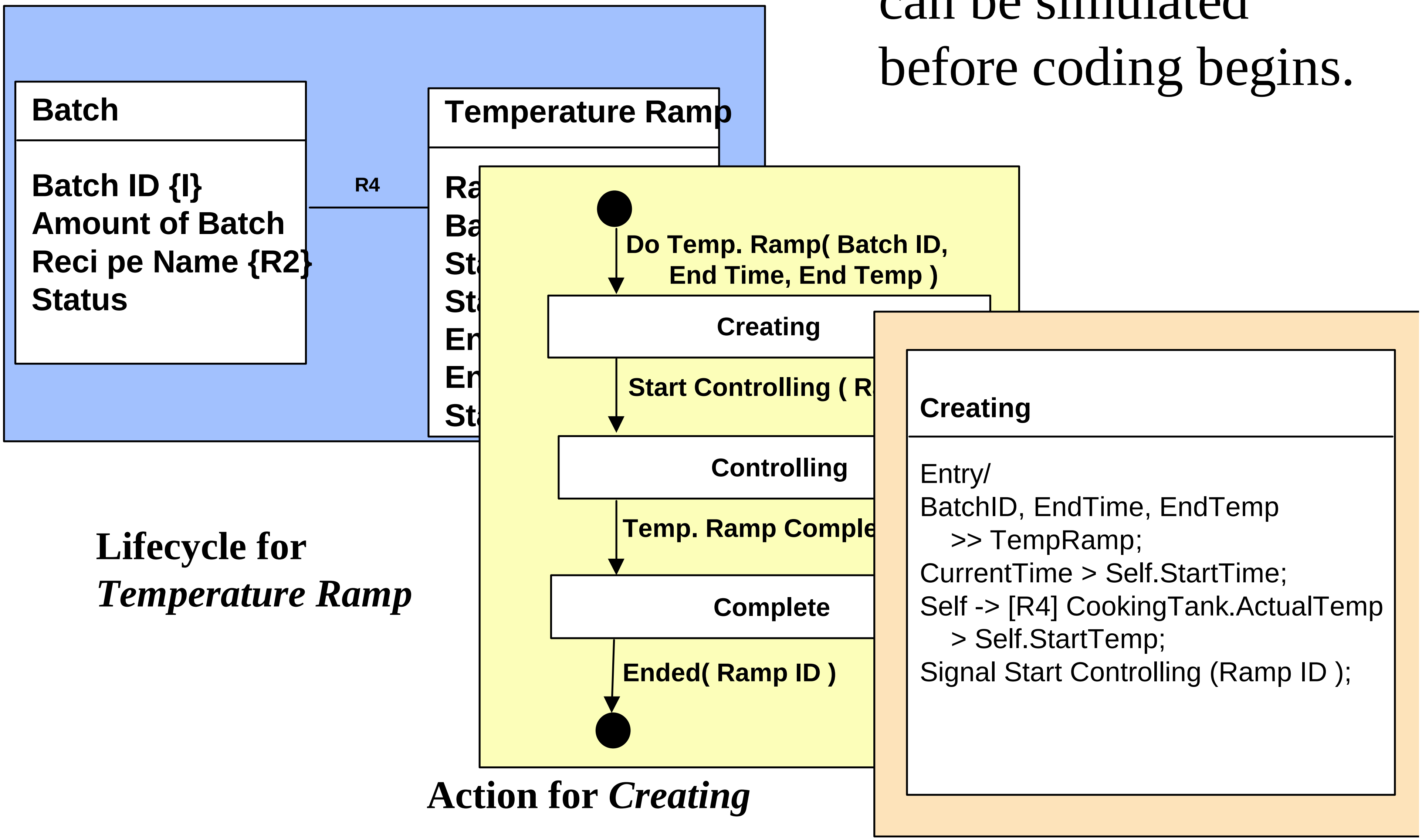


Executable UML Models



Executable UML Models

Executable models can be simulated before coding begins.

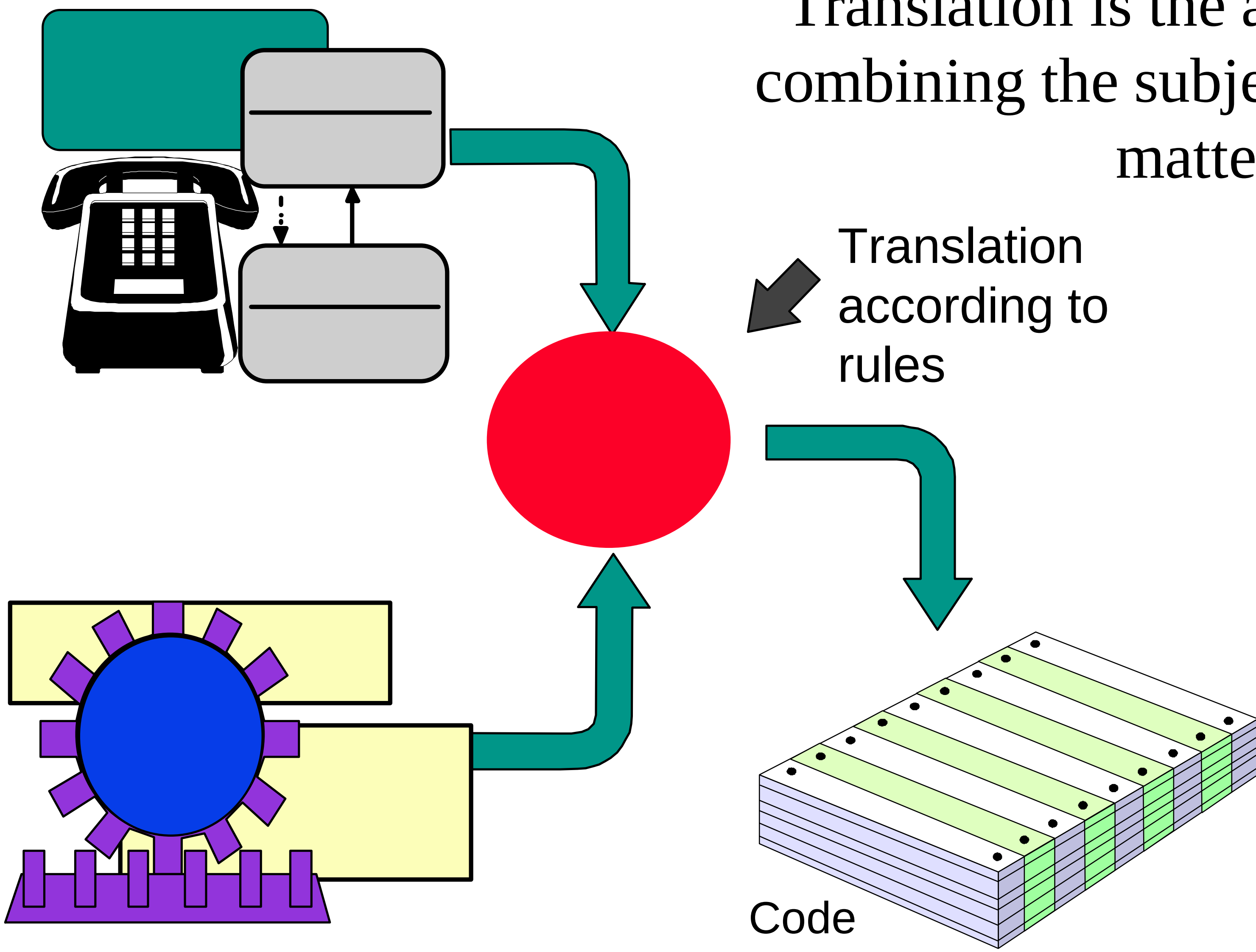


Lifecycle for *Temperature Ramp*

Action for *Creating*

Translation

Translation is the act of combining the subject matters.



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

Shlaer-Mellor Method

The Shlaer-Mellor Method meets the challenges of real-time software development by:

- localizing critical software design issues to the software architecture domain
- ensuring that the design decisions are incorporated uniformly and systematically
- providing a framework to modify system performance without affecting system behavior

System Design: Architectures and Archetypes

This tutorial showed you how to:

- identify the characteristics of the problem that determine the system design;
- engineer the system-wide design to meet performance constraints;
- model the system-wide design—the software architecture;
- build archetypes to produce efficient code.