System Design: Architectures and Archetypes

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

Shlaer-Mellor Method
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
Properties

Separation of application from architecture

Executable UML models

Translation according to rules

Code
What’s in the Architecture?

The architecture comprises:

- an execution engine plus
- a set of archetypes.
Archetypes define the rules for translating the application into a particular implementation.

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

```
.Function PrivateDataMember( inst_ref class )
    .select many PDMs from instances of Attribute related to Class
    .for each PDM in PDMs
        $\{\text{PDM.Type}\} $\{\text{PDM.Name}\};
    .endfor
```
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>
<thead>
<tr>
<th>The Software Architecture</th>
<th>Archetype Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architectural Styles</td>
<td>A Direct Translation</td>
</tr>
<tr>
<td>Selecting an Architecture</td>
<td>Specifying the Architecture</td>
</tr>
<tr>
<td>Performance Requirements</td>
<td>An Indirect Translation</td>
</tr>
<tr>
<td>Executable Domain Models</td>
<td>System Construction</td>
</tr>
<tr>
<td>Model Execution</td>
<td>The Shlaer-Mellor Method</td>
</tr>
<tr>
<td>Capturing the Models</td>
<td></td>
</tr>
</tbody>
</table>
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.
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?
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?
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.
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
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)
- Household microprocessor (temperature control)

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

This style tends to have:

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

- 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:

- 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
“[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
“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
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 stream is a source of packets.
* A packet 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 thread 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
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)

© 1998 Model Integration, LLC
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 Diagram]

© 1998 Model Integration, LLC
Abstract classes based on both:

- data, and
- behavior

<table>
<thead>
<tr>
<th>Recipe</th>
<th>Batch</th>
<th>Temperature Ramp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recipe Name {I}</td>
<td>Batch ID {I}</td>
<td>Ramp ID {I}</td>
</tr>
<tr>
<td>Cooking Time</td>
<td>Amount of Batch</td>
<td>Batch ID {R4}</td>
</tr>
<tr>
<td>Cooking Temp.</td>
<td>Recipe Name {R2}</td>
<td>Start Temperature</td>
</tr>
<tr>
<td>Heating Rate</td>
<td>Status</td>
<td>Start Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>End Temperature</td>
</tr>
<tr>
<td></td>
<td></td>
<td>End Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Status</td>
</tr>
</tbody>
</table>
Lifecycles

Build a lifecycle model for each class.

Lifecycle for Temperature Ramp

- **Creating**
  - Do Temp. Ramp( Batch ID, End Time, End Temp )
  - Start Controlling ( Ramp ID )
  - Temp. Ramp Complete( Ramp ID )
  - Timer Expired ( Ramp ID )

- **Controlling**
  - Temp. Ramp Complete( Ramp ID )

- **Complete**
  - Ended( Ramp ID )

- **End**
Specify the logic for each state’s action.

Actions

Do Temp. Ramp( Batch ID, End Time, End Temp )

Creating

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

Lifecycle for Temperature Ramp

Batch
Batch ID {I}
Amount of Batch
Recipe Name {R2}
Status

Temperature Ramp
Ramp ID {I}
Batch ID {R4}
Start Temperature
Start Time
End Temperature
End Time
Status

Action for Creating

Creating

Controlling

Complete

Ended(Ramp ID)

Do Temp. Ramp(Batch ID, End Time, End Temp)

Creating

Entry/
BatchID, EndTime, EndTemp

>> TempRamp;

CurrentTime > Self.StartTime;

Self -> [R4] CookingTank.ActualTemp

> Self.StartTemp;

Signal Start Controlling(Ramp ID);
Model Execution
An executable model operates on data about instances.
Instances

An executable model operates on instances.
Execution

The lifecycle model prescribes execution.

<table>
<thead>
<tr>
<th>Batch ID</th>
<th>Amount of Batch</th>
<th>Recipe Name</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>Nylon</td>
<td>Filling</td>
</tr>
<tr>
<td>2</td>
<td>127</td>
<td>Kevlar</td>
<td>Emptying</td>
</tr>
<tr>
<td>3</td>
<td>93</td>
<td>Nylon</td>
<td>Filling</td>
</tr>
<tr>
<td>4</td>
<td>123</td>
<td>Stuff</td>
<td>Cooking</td>
</tr>
</tbody>
</table>

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...

Recipe
- Recipe Name {I}
- Cooking Time
- Cooking Temperature
- Heating Rate

Batch
- Batch ID {I}
- Amount of Batch
- Recipe Name {R2}
- Status

Temperature Ramp
- Ramp ID {I}
- Batch ID {R4}
- Start Temperature
- Start Time
- End Temperature
- End Time
- Status

Pre-existing

Created during execution
### Initialization

Some instances exist before the model begins to execute...  
...and so require initialization.

<table>
<thead>
<tr>
<th>Recipe Name</th>
<th>Cooking Time</th>
<th>Cooking Temp</th>
<th>Heating Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nylon</td>
<td>23</td>
<td>200</td>
<td>2.23</td>
</tr>
<tr>
<td>Kevlar</td>
<td>45</td>
<td>250</td>
<td>4.69</td>
</tr>
<tr>
<td>Stuff</td>
<td>67</td>
<td>280</td>
<td>1.82</td>
</tr>
</tbody>
</table>
Executing the Model

The model executes in response to signals from:

- the outside,
- other instances as they execute
- timers
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?
A *meta-model* defines the structure of the repository.
A *meta-model* defines the structure of the repository.
A meta-model defines the structure of the repository.
Meta-Model Instances

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

<table>
<thead>
<tr>
<th>Class ID</th>
<th>Name</th>
<th>Descr'n</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Recipe</td>
<td>.....</td>
</tr>
<tr>
<td>101</td>
<td>Batch</td>
<td>.....</td>
</tr>
<tr>
<td>102</td>
<td>Temp Ramp</td>
<td>.....</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class ID</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>102</td>
</tr>
<tr>
<td>102</td>
</tr>
<tr>
<td>102</td>
</tr>
</tbody>
</table>
Archetype
Language
To generate code....

State

<table>
<thead>
<tr>
<th>Class ID</th>
<th>State #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1</td>
<td>Filling</td>
</tr>
<tr>
<td>101</td>
<td>2</td>
<td>Cooking</td>
</tr>
<tr>
<td>101</td>
<td>3</td>
<td>Emptying</td>
</tr>
</tbody>
</table>
| 102      | 1       | .......
| 102      | 2       | .......
| 102      | ....    | .......

Class

<table>
<thead>
<tr>
<th>Class ID</th>
<th>Name</th>
<th>Descr'n</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Recipe</td>
<td>.....</td>
</tr>
<tr>
<td>101</td>
<td>Batch</td>
<td>.....</td>
</tr>
<tr>
<td>102</td>
<td>Temp Ramp</td>
<td>.....</td>
</tr>
</tbody>
</table>
....traverse the repository and...

... output text.
The archetype language produces text.

```cpp
.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
};
```

```cpp
public:
  enum states_e
  { NO_STATE = 0,
    Filling,
    Cooking,
    NUM_STATES = Emptying
  };
```

Example
To generate text:

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**

<table>
<thead>
<tr>
<th>Class ID</th>
<th>Name</th>
<th>Descr'n</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Recipe</td>
<td>.....</td>
</tr>
<tr>
<td>101</td>
<td>Batch</td>
<td>.....</td>
</tr>
<tr>
<td>102</td>
<td>Temp Ramp</td>
<td>.....</td>
</tr>
</tbody>
</table>

Instance reference
To access attributes of the selected instance….

```java
${class.Name}
```

<table>
<thead>
<tr>
<th>Class ID</th>
<th>Name</th>
<th>Descr'n</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Recipe</td>
<td>.....</td>
</tr>
<tr>
<td>101</td>
<td>Batch</td>
<td>.....</td>
</tr>
<tr>
<td>102</td>
<td>Temp</td>
<td>Ramp</td>
</tr>
</tbody>
</table>
Association Traversal

To traverse an association…..

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

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

State Chart
Class ID {I, R13}
Name
R13
0..1 1

Class
Class ID {I}
Name
Description

Create Batch( Amount of Batch, Recipe Name)
Filling
Filled( Batch ID )
Cooking
Temp. Ramp Complete( Batch ID )
Emptying
Emptied( Batch ID )
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,
We may combine these techniques…. 

```cpp
.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
};
```
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 addPDMDecl( inst_ref class)
  ...
};
.endfor
```
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
```
To declare a state chart: (i.e. all the actions in the state chart)

```plaintext
.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
```
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.
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

.Action Function Declaration

.Protected Function Declaration
The models are similar because the architecture is a direct translation.
An Indirect 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

Event-driven Task

- Event messages

Periodic Task

- Event message
- Data Copy
- Instance bits
- Timer
Application Mapping

Recipe
Recipe Name {I}
Cooking Time
Cooking Temperature
Heating Rate

Batch
Batch ID {I}
Amount of Batch
Recipe Name {R2}
Status

Temperature Ramp
Ramp ID {I}
Batch ID {R4}
Start Temperature
Start Time
End Temperature
End Time
Status

Event Driven Task

Periodic Task
Temperature Ramp
Ramp ID {I}
Start Temperature
Start Time
End Temperature
End Time

Ramp Id Bits

Start Temperature
End Time
Application Mapping

Event Driven Task

- Do Temp. Ramp( ... )
- Creating
- Start Controlling( Ramp ID )
- Controlling
- Temp. Ramp Complete( Ramp ID )
- Complete
- Ended( Ramp ID )

Periodic Task

- Controlling

Ramp Id Bits
Extended Properties

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

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

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class ID {I, R14}</td>
</tr>
<tr>
<td>State Number {I}</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>isFinal</td>
</tr>
<tr>
<td>isPeriodic</td>
</tr>
</tbody>
</table>

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.
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.
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.

Do not modify the generated code directly.
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 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.

Separation of application from architecture

Code
Executable UML Models
Executable UML Models

Executable models can be simulated before coding begins.

Lifecycle for Temperature Ramp

Action for Creating

Batch

Batch ID {I}
Amount of Batch
Recipe Name {R2}
Status

Temperature Ramp

Ramp ID {I}
Batch ID {R4}
Start Temperature
Start Time
End Temperature
End Time
Status

Do Temp. Ramp(Batch ID, End Time, End Temp )

Creating

Start Controlling(Ramp ID)

Controlling

Temp. Ramp Complete

Complete

Ended(Ramp ID )

Creating

Entry/
BatchID, EndTime, EndTemp
>>> TempRamp;
CurrentTime > Self.StartTime;
Self -> [R4] CookingTank.ActualTemp
> Self.StartTemp;
Signal Start Controlling(Ramp ID );
Translation is the act of combining the subject matters.

Translation according to rules
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
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
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.