A Model-Driven Development Approach to Creating Instantiable and Reusable Software Frameworks

Peter Keppler
Raytheon
Network Centric Systems
Version 1
Presentation Purpose

Part 1:
- Define the UML 2.0 diagrams and elements that support the creation of software subsystems and framework construction
- Show how a SW system architecture is constructed using UML 2.0

Part 2:
- Demonstrate an approach to create software frameworks that are instantiable and where product specific behaviors are “plugged” into the framework at runtime
  - Define what a framework is
  - Where to use frameworks
  - Framework properties
  - MDD framework properties
  - How to design frameworks
  - UML 2.0 framework construction and initialization

Purpose of presentation is to show how to construct a SW system using UML 2.0 to maximize code generation and reusability
UML Elements That Generate Code And Are Used For Framework Construction
## Composition

- Two ways to model composition, aggregation and composite class. Composition is implemented as an embedded object.

```
class Whole
{
    Part itsPart; // Class Part is contained by class Whole
    Part itsPart1;
};
```
Aggregation

- Aggregation is implemented as a pointer
  - (in Rhapsody C++ version)

```cpp
class Whole
{
    Part   * itsPart; // pointer to Part
};
```
Inheritance

- The derived classes (Missile and Laser) inherit common behaviors (and data) from the parent Armament class.
- The derived classes may specialize this behavior to suit their own specific needs.
Active Classes and Concurrency

- An active class is a class whose instances are active objects.
- An active object is an instance of an active class that owns its own thread and can initiate control activity.
- Active objects are graphically portrayed with thicker borders.
- For the purpose of this presentation, active classes and state machines are used to transfer control from one thread to another to implement concurrency.
State Charts Used to Implement Concurrency

- A simple state machine receives an event from a port which is used to extract the payload (i.e., data) that the event carries.
- The state forwards processing to an operation within the active class that implements the behavior.

```
processMessage(msg: Message): void
```

State machines contained within an active class are used to implement concurrency.
Systems, Subsystems, and Components

- Defined during architectural design
  - Identifies major components that form the top-level structure
  - Defines the behavior of each component
  - Defines the interfaces for the components

- System definition
  - A class that encompasses the largest-scale entity in a project

- A system is organized into subsystems
  - A class that is the first-level compositional unit of a system

- Component: A modular, self-sufficient, replaceable unit that works like a black box
  - A subsystem can be considered a large component
  - A complex class with internal parts can also be a component
  - Makes system more flexible, maintainable, scalable, and reusable

MDD espouses component-based development philosophy
Abstract interface class supplies public access to subsystem behavior (ITrackAccess)

Concrete class inherits from interface class and supplies the behavior implementation (TrackController)

Concrete class is implemented as a singleton pattern

Subsystem behavior is abstracted by interface
UML 2.0 Subsystem Construction

- Abstract interface class supplies public access to subsystem behavior (ITrackAccess)
- Concrete class inherits from interface class and supplies the behavior implementation (TrackController)
- Singleton is created from the composite subsystem class

Model compiler provides code generation for subsystem
Ports

- A point of interaction between the inside and outside of a component
- Provided and required interfaces are connected to a port
- Interfaces specify the services that the component provides or requires

Ports provide access points to a component, class, or subsystem
Required and Provided Interfaces

- Interface services that this subsystem provides to clients are supplied by “provided” interfaces.
- Interface services that this subsystem requires from service providers are accessed by “required” interfaces.

Services accessed by subsystem are also encapsulated.
A link connects two objects that communicate
A link is an instance of an association
A link generates the code necessary to connect two objects
System Example
System Example: Combat System

- This presentation will construct a simple Combat System to demonstrate MDD development using UML 2.0
- Combat System Mission: To track, identify, and engage threats
We will be investigating the subsystem below in the following slides:

Track Database Subsystem
Mission: Keeps an inventory of all objects currently being tracked by the combat system and a record of their creation and deletion.

- **TrackController**
  - createTrack():void
  - deleteTrack():void
  - getTrack():void

- **TrackFactory**
  - Reports Creation of Tracks

- **TrackDeletionManager**
  - Reports Deletion of Tracks

- **Track**
  - pTrackAccess
  - pTrackStatus

- **TrackHistoryRecord**
  - reportNewTrack():void
  - reportTrackUpdate():void

- **TrackRecord**
  - trackNumber_:unsigned long
UML 2.0 allows you to place a diagram inside a class
We can build our subsystems using this approach
Each “part” is instantiated when the composite class is created

Composite structure diagram is used to create subsystems
Links Connect the Parts of a Subsystem

- A link connects two objects that communicate
- A link is an instance of an association
- A link generates the code necessary to connect two objects
Links: continued

Links connect the instantiated parts of a subsystem
Combat System Overview

- We have just completed construction of one of the subsystems of the Combat System as a reusable component.
- The next several slides give an overview of the Combat System domain subsystems and classes.
- We will combine this subsystem with others to create the Combat System in the following slides.
- The Combat System is composed of:
  - Track Database Subsystem
  - Track Management Subsystem
  - Display Subsystem
Track Management Subsystem

Definition of Track Management subsystem used in this presentation to build a Combat System

Track Management Subsystem
Mission:
This subsystem is responsible for the creation and maintenance of tracks.

«Subsystem»
TrackManagementSubsystem

1
itsTracker:Tracker

pTrackAccess
ITrackAccess

pTrackAccess
ITrackAccess
Display Subsystem

Definition of Display subsystem used in this presentation to build a Combat System

Display Subsystem
Mission:
This subsystem is responsible for generating and outputting track information on the combat system display.
The subsystem components become parts of the system whose ports are connected by links.

A loosely coupled system is created by port linkages.
New Subsystems May Be Easily Added

- Any component that exposes an interface on a port may be added or changed with any component having a compatible interface.

Components may be added or changed at will
Framework-Based Model-Driven Development
The purpose of this part of the presentation is to demonstrate an approach to create SW frameworks that are instantiable and where product specific behaviors are “plugged” in to the framework at runtime.

Topics covered:
- Define what a framework is
- Where to use frameworks
- Framework properties
- MDD framework properties
- How to design frameworks
- UML 2.0 framework construction and initialization
A software framework is a reusable design for a software system (or subsystem).

They are expressed as a set of abstract classes and the way their instances collaborate for a specific type of software.

Software frameworks are usually implemented with object-oriented designs and in an OO language.

A software framework may include support programs, code libraries, a scripting language, or other software to help develop and glue together the different components of a software project.

Various parts of the framework may be exposed through an application programming interface (API).
Framework Importance

- specifies system architecture
- enforces architectural conformance
- reusable for families of products
- architecture is pretested for subsequent use
Motivation To Use Frameworks

Family of Products

Constrains Choices

Concentrates On Business Logic

Reusable Structure Allow Focus on Requirements

Product Line Model

Increases Productivity

Framework

Concentrates On Business Logic

Reusable Structure Allow Focus on Requirements

Product Line Model

Increases Productivity

Family of Products

Constrains Choices
Framework Properties

Software Systems contain “frozen” and “hot” spots. Hot spots are areas that are subject to change.
Framework Requirements

Frameworks need to:

- Use a common notation to describe its design
- Need an architecture that defines the system parts
- Define communication paths
- Strongly types the communication paths
- Define the threading model
- Allow different levels of reuse (parts can be frameworks)
- Allow extensibility (statically and/or dynamically)
- Allow behaviors to be added to framework
MDD Framework Properties

MDD Frameworks (arguably):
- Are based on the Unified Modeling Language (UML)
- Are constructed with Object-Oriented concepts
- Their behavior is abstracted with formal interfaces
- Are created using UML composite classes
- The subsystem interaction points are encapsulated with ports
- The subsystems ports are connected with links
- Are instantiatable (i.e., constructable at run time)
- Their code is generated by a model compiler
## Mapping of MDD Framework Properties to Framework Requirements

<table>
<thead>
<tr>
<th>Framework Requirements</th>
<th>MDD Framework Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>■ Common design notation used                                                          ■ Based on the UML</td>
<td></td>
</tr>
<tr>
<td>■ Architecture defines system parts                                                     ■ UML composite classes define parts</td>
<td></td>
</tr>
<tr>
<td>■ Define communication paths                                                           ■ Interaction points use UML ports</td>
<td></td>
</tr>
<tr>
<td>■ Strongly types communication paths                                                   ■ Ports are connected with UML links</td>
<td></td>
</tr>
<tr>
<td>■ Defines the threading model                                                          ■ Behavior abstracted with interfaces</td>
<td></td>
</tr>
<tr>
<td>■ Allows different levels of reuse                                                     ■ Defined with UML active classes</td>
<td></td>
</tr>
<tr>
<td>■ Allows extensibility                                                                 ■ Designed using OO patterns</td>
<td></td>
</tr>
<tr>
<td>■ Allows behaviors to be added                                                          ■ Designed using OO patterns</td>
<td></td>
</tr>
<tr>
<td>■ MDD added properties:                                                                  ■ Constructed using OO patterns</td>
<td></td>
</tr>
<tr>
<td>- System is instantiatable                                                              ■ MDD added properties:</td>
<td></td>
</tr>
<tr>
<td>- Model compiler generates code                                                         ■ MDD added properties:</td>
<td></td>
</tr>
<tr>
<td>- System is instantiatable</td>
<td></td>
</tr>
</tbody>
</table>
|                                                                                      - Model compiler generates code
A Pattern-Based Approach

- **Interface/Controller** – Provide an interface that abstracts the behavioral implementation of an operation from client view.

- **Strategy** - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms vary independently from clients that use it.

- **Builder** – Separate the construction of a complex object from its representation so that the same construction process can create different representations.

- **Decorator** - Attach additional responsibilities to an object dynamically.

- **Command** - Encapsulate a request as an object, thereby letting you parameterize clients with different requests.
The Product Line Model (PLM) is the framework which only contains behavior that is common to all the product instantiations of the framework.

The Product Specific Model (PSM) is a specific instance of the framework that contains the behaviors specific to a particular product (e.g., a sensor platform).
PLM/PSM Example Data Flow

Product Line Model

Common Processing

Product Specific Model

Sensor Specific Message Format

Sensor

Save Data

Combat System Specific Message Format

Combat System
PLM to PSM Data/Control Flow

Product Line Model

Events in

Subsystem

Events out

Procedure in

Procedure out

Product Specific Model
Framework Construction

A UML composite class is used to create the framework and its parts (the subsystems).
Each Framework Subsystem Defines a Design Pattern That Can Be Instantiated

The Product Specific behavior is modeled using the Gamma "Strategy" Design Pattern:

“Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.”

Class Diagram for the Framework Subsystem in the Product Line Model

```
FrameworkController

pReceive

IlInitiateBehavior

~FrameworkController()

processMessage(message: MessageClass): void

frameworkController_ 1

frameworkStrategy_ 1

FrameworkStrategy

~FrameworkStrategy()

processMessage(message: MessageClass): void
```
Product Specific Behavior

Class Diagram for the Subsystem the in the Product Specific Model that corresponds to the framework subsystem.

The Product Specific Behavior is Defined by the Product Specific UML Model.

This class is part of the Platform Independent Model Framework.

This diagram also shows an additional port is added in the PSM.
A UML Model Defines the Abstract Instantiable Framework (the PLM)

Name: Framework Subsystem Class Diagram

Mission: This subsystem describes the design patterns used to form the subsystems of an abstract framework that is instantiated by a platform specific model that supplies the platform specific behaviors via the Strategy design pattern. This diagram describes the collaboration relationships between the classes that comprise the subsystem.

- FrameworkController
  - pReceive
  - InitiateBehavior
  - pSend

- frameworkController
  - frameworkStrategy

- FrameworkStrategy
  - processMessage(message: MessageClass): void
A UML Model Defines Each Product Specific Implementation of the Framework (a PSM)
**PLM and PSM Summary**

**Product Line Model**

- **FrameworkSubsystem**
  - 1: itsFrameworkController
  - pReceive
  - pSend
  - pReceiveSS
  - pSendSS
  - IInitiateBehavior

**FrameworkController**
- processMessage(message:MessageClass):void

**FrameworkStrategy**
- processMessage(message:MessageClass):void

**SpecificBehavior**
- processMessage(message:MessageClass):void

**Product Specific Model**
The PSM subsystem can be modeled with any level of complexity needed.
Adding Threads and Behaviors to the Static Framework

A state chart in the PLM can be extended in the PSM.
The thread is “inherited”

A state chart and thread can be added to the PSM derived class.

Product Line Model

Product Specific Model

Inherited State Machine

Additional Active Class
Passing Product Specific Data Through the Product Line Framework

1. A product specific data type is created by the PSM.
2. The data is passed from the PSM derived class to the corresponding PLM subsystem,
3. and then as an abstract data type to the PLM subsystem that is responsible for handling the data.
4. The receiving PLM subsystem passes the data down to its corresponding PSM derived class to be handled as the product specific data type.

**Product Line Model**

```
MessageBase
- messageType_: PsmMsgType

PimFrameworkSubsystem
- pSend
  - IExtendBehavior

PimFrameworkSubsystem
- pReceive
  - IExtendBehavior

PsmFrameworkSubsystem
- Creates
  - PlatformSpecificMessage
    - messageType_: PsmMsgType
    - payload_: PsmSpecificData

Product Specific Model
```

**Diagram:**

1. A product specific data type is created by the PSM.
2. The data is passed from the PSM derived class to the corresponding PLM subsystem.
3. The data is then passed as an abstract data type to the PLM subsystem responsible for handling the data.
4. The receiving PLM subsystem passes the data down to its corresponding PSM derived class to be handled as the product specific data type.
Framework is Modifiable at Runtime

- Parts may be instantiated and added to the system
- The ports of the new parts are connected with “links” at runtime
- New ports can be created and added to existing composite classes in the framework
- Use a generic interface as the port contract
Framework Instantiation and Initialization

1. Factory creates Behavior
2. PSM passes behavior to base class
3. Subsystem adds PSM behavior to strategy aggregation relationship
4. Controller
5. UML aggregation relationship

- **UML Inheritance**
- **Product Line Model**
- **Product Specific Model**
You can develop design patterns to indicate common behavior
Reusability Quantified

Non-behavioral (constructional) size

- PLM (white): 8 classes, 500 reusable SLOC
- PSM (blue): 1 class, 20 SLOC
- PLM test reusability: 1200 SLOC
Program Metrics

- Two programs built from the Product Line model
- Both shared the costs of developing the Product Line model
- The costs of creating the Product Line model were greater than expected
- The next program will have dramatically less development costs

<table>
<thead>
<tr>
<th></th>
<th>Project A</th>
<th>Project B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Line SLOC</td>
<td>33K</td>
<td>33K</td>
</tr>
<tr>
<td>Product Specific SLOC</td>
<td>67K</td>
<td>48K</td>
</tr>
<tr>
<td>Reuse factor</td>
<td>49%</td>
<td>69%</td>
</tr>
</tbody>
</table>
Summary

At this time you should have an understanding of one proven approach to creating reusable software frameworks in a UML modeling tool and how to use the tool to generate the software system from the UML model.
QUESTIONS?
Biography

Peter Keppler has been with Raytheon NCS at St. Petersburg for 7 years and has a tremendous amount of software development experience in OOA/OOD. He is currently working in the Joint Sensor Networking (JSN) business area in pioneering the use of using MDD to construct software systems. He has been instrumental in advancing Model Driven Architecture and the Rhapsody MDD environment within the NCS community and is a co-chair of the Model-Driven Software Development TIG. Peter has also worked in the Joint Sensor Networking business area as part of the Cooperative Engagement Capability Program involved with network connectivity management and communications scheduling protocol software.

Previously at a major Unisys software development center, Peter worked on operating system development. His activities included analyzing new software development methodologies and processes for use in the development of a high performance, transaction based, operating system used on Unisys’ commercial enterprise servers.