Real-Time UML

Bruce Powel Douglass, Ph.D.
Chief Evangelist, I-Logix

UML is a trademark or registered trademark of Object Management Group, Inc. in the U.S. and other countries.
Basic UML
What is UML?

• UML is a 3rd generation object-oriented modeling language

• UML unifies the best practice from
  – Booch (Grady Booch)
  – OMT (Jim Rumbaugh)
  – OOSE (Ivar Jacobson)
  – Statecharts (David Harel)
  – Several dozen others

• UML has been accepted by the Object Management Group as a standard (1997)

• Current UML standard is 1.4
  – Available at
    • www.omg.org
    • www.ilogix.com
Why Should I Care About UML?

• UML is *better*
  – Support for latest advances in object modeling
  – Supports both elaborative and translative development processes

• UML has broad tool support
  – I-Logix
  – Rational
  – Paradigm

• UML supports real-time systems today without extensions or add-ons
  – UML *does* have a profile for schedulability, timeliness and performance modeling we will discuss later in the day
What is Real-Time UML?

• Real-Time UML is **standard UML**
  – “UML is adequate for real-time systems” Grady Booch 1997
  – “Although there have been a number of calls to extend UML for the real-time domain … experience had proven this is not necessary.” Bran Selic, 1999 (Communications of the ACM, Oct 1999)

• Real-time and embedded applications
  – Special concerns about quality of service (QoS)
  – Special concerns about low-level programming
  – Special concerns about safety and reliability

• Real-Time UML is about applying the UML to meet the specialized concerns of the real-time and embedded domains
UML Object Model
Major UML Features

- Well-defined object model
- Richly defined *executable* object semantics
- Language independence
- Richly defined behavioral model using statecharts
- Support for
  - object-oriented patterns
  - Software architectures
  - Component-based development
- Does *not* specify a development process
  - The UML can be used in ANY reasonable development process
Objects, Classes and Interfaces

- An **object** is a run-time entity that occupies memory at some specific point in time
  - Has *behavior* (methods)
  - Has *data* (attributes)

- A **class** is a design-time concept that defines the structure and behavior for a set of objects to be created at run-time.
  - Specifies *behavior* (methods)
  - Specifies *data* (attributes)

- An **interface** is a design time concept that specifies the messages a class receives
  - Has *behavior only* (operations)
Structural Diagrams

• Diagrams serve many purposes
  – System model-capture & specification
  – View aspects of system design
  – Provide basis for communication and review

• Diagrams bring 2 things to the design process
  – Represent different aspects of design, eg
    • Functional
    • Structural
    • Behavioral
    • Quality of Service
  – Show aspects at different levels of abstraction
    • System
    • Subsystem
    • Component
    • “Primitive” class
Classes

class Sensor

value: int
calibrationConstant: long

getValue()
zero()

value = 7680
calibrationConstant = 45

intakeSensor: Sensor

value = 7680
calibrationConstant = 45

getValue()
zero()

motorSensor: Sensor

value = -65
calibrationConstant = 6780

getValue()
zero()
UML Classes

• Classes are specifications of objects that exist at design time
• Shown as a rectangle with a class name

• (Optionally) shown as a 3-compartment rectangle:
  – Class name
  – Attribute list
  – Operation list

<table>
<thead>
<tr>
<th>Sensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
</tr>
<tr>
<td>Alarm Limit</td>
</tr>
<tr>
<td>GetValue</td>
</tr>
<tr>
<td>SetAlarmLimit</td>
</tr>
<tr>
<td>EnableAlarm</td>
</tr>
</tbody>
</table>
UML Classes specify...

- **Methods**
  - Implementation of operations
  - Typically manipulate class attributes

- **Attributes**
  - Typed values “known” by the class

- **Statechart**
  - States
  - Transitions
  - Actions

- **Interfaces**
  - Operations realized by the class methods
UML Objects

• Objects are *instances* of classes that exist at some point in time in some memory location

• Shown just like a class except name is underlined

• May optionally show object class as well:
  – role name: class name

  TempSensor: Sensor
Interfaces

- An interface is a named collection of operations
  - May not have attributes or methods
  - Not instantiable
- An interface provides the means to request services of (e.g. send messages to) an object
- Not supported directly in all source languages
  - Java yes!!!
  - C++ no!!!
Interfaces

- UML interfaces specify operations
- UML Interfaces have no implementation
  - No methods
  - No attributes
- Classes realize an interface by providing a method (implementation) of an operation
  - A class that realizes an interface is said to be compliant with that interface
  - Classes may realize any number of interfaces
  - Interfaces may be realized by any number of classes
Interfaces

Sensor

iSensor

sensorClient

canonical form, or “lollipop”

Detailed form

Sensor

Value: int
Get(): int
Set(int a): void;
Config(): void;
Filter(corner: long): void

<<interface>>

iSensor

Get(): int;
Set(int a): void;
Config(): void;

<<realizes>>

sensorClient

<<interface>>
UML Relationships

- Relation means that two entities are somehow related
- Classes are little things and need to collaborate to achieve system-wide behaviors

```
Relation
  ├── Generalization
  │    └── Aggregation
  └── Association
      └── Dependency
          └── Composition
```
UML Relations

- Primary Relations in UML
  - Associations
    - Normal association
    - Aggregation
    - Composition
  - Generalization
  - Dependency
    - <<bind>> for templates
    - <<friend>> for classes
    - <<includes>> and <<extends>> for use cases
    - Etc, etc etc
Class Relations

[Diagram showing class relationships between mouse, keyboard, inputDevice, window, scrollbar, clientArea, dialogBox, SDIWindow, MDIWindow, and key.]
Associations

• Associations between classes mean that there are *links* between objects at run-time that allow the objects to exchange messages.

• Associations may have
  – Role names at each end of the association
  – Association labels
  – Multiplicity identifying the number of participant objects
  – Navigation direction as indicated by open arrow on association
Normal Association

• Used when objects communicate but have independent lifecycles
• UML normal associations are assumed to be bidirectionally navigable
• Client-Server is a normal association that is navigated in a single direction
  – Client knows about the server, but server is ignorant of the client
• Navigation implies a pointer or a reference
• Shown with a simple line

Sensor ——— Controller
Aggregation

- Used when one object is physically or logically composed of others
- Whole-part relationship
- “Whole” end is marked with a diamond
- AKA “catalog aggregation” because a part may belong to multiple “wholes”
  - No strong statement about responsibility for creation of parts is made
Composition

- Used when one object is composed of one or more others
- The whole object primarily delegates its responsibilities to its parts
- Strong form of aggregation
  - Implies the “whole” both creates and destroys the “part”
- Implies multiplicity of “1” with owner (it is unshared)
Composition

- PRIMARY USE: show systems at different levels of abstraction
  - Composite is at a higher level of abstraction than the part
  - Composite achieves its behavioral goals primarily through delegation
- Additional use: Populate simple classes into «active» classes for concurrency
- Additional use: Use composites to aid in system boot and object construction
Composition Example (abstract)
Composition Example (details)
Using Various Associations
Generalization

• Generalization means two things in the UML
  – *Inheritance*
    • The child acquires things from the parent(s). In UML, a child class acquires the attributes and operations (including statecharts) from the parent class(es).
  – *Substitutability*
    • It’s satisfactory to use the substitute instead of the intended item. In UML, a substituted object (instance) performs satisfactorily.
Generalization

• Subclass *is-a-type-of* the superclass

• UML generalization means:
  – Subclass inherits structure and behavior from the superclass
    • attributes
    • operations
    • relations
    • statechart
  – Instances of subclass are freely substitutable for instances of the superclass
Generalization: Example

Inheritance

Sensor
- value : int
- location : int

getValue()
getLocation()

TempSensor
- maxLimits : int
- minLimits : int
- calConstant : int
- calibrate()
- setLimits()
Generalization

- Subclasses may be **extended** by adding
  - new attributes
  - new behaviors

- Subclasses may be **specialized** by
  - redefining existing behaviors
Example: Generalization

- **new attribute**
- **inherited operation**
- **new operation**
- **specialized operation**
- **inherited attributes**
Packages

- Booch has said “Classes provide a necessary, but insufficient means of decomposition”

- Large-Scale Logical Design
  - Packages capture larger scale decompositions that exist at design time (cannot be instantiated)
  - Packages organize your model
  - Packages define a namespace for managing large numbers of model elements

- Large-Scale Physical Design
  - Subsystems capture larger scale decompositions that exist at run-time
  - Contains objects that collaborate to realize common function purpose (use case) of the subsystem
Class Packages
Deployment Diagrams

- Show the physical entities making up the systems that exist during run-time
- Use stereotypes for common node types:
  - Processor
  - Sensor
  - Disk drive
  - Display
  - Knobs and buttons
Deployment Diagrams

- Especially useful in embedded systems
- Can specify where tasks and system functions are located
- 3D boxes are nodes (devices)

- Components are physical packaging of logical elements
  - compile-time (e.g. files)
  - run-time (e.g. libraries) software entities
Deployment Diagram
Behavioral Diagrams
Sequence Diagrams

- Sequence diagrams show the behavior of a group of instances (e.g. objects) over time. Instances may be
  - Objects (most common)
  - Use case instance
  - System
  - Subsystem
  - Actor

- Useful for
  - Capturing typical or exceptional interactions for requirements
  - Understanding collaborative behavior
  - Testing collaborative behavior
Sequence Diagrams
Sequence Diagrams and States

- States can be shown on sequence diagrams. The states hold in time until a change of state occurs.
- Shows correspondence between collaboration and object state change.
Statecharts and Scenarios
Statecharts
Statecharts

- Statecharts are a behavioral diagram
- They show the specification of behavior of a single classifier, e.g.
  - Class
  - Object
  - Component
  - Subsystem
  - Use case
UML State Models

- Specifies the behavior for reactive classes
- Not all classes have state behavior
- Notation and semantics are Harel Statecharts
  - Nested States
  - Actions on
    - Transitions
    - State Entry
    - State Exit
  - Activities performed within a state

**Actions are run-to-completion behaviors**

**Activities are interruptible behaviors**
Statecharts

• What’s a state?

A state is a distinguishable, disjoint, orthogonal ontological condition of an objects that persists for a significant period of time

• What’s a transition?

A transition is a response to an event of interest moving the object from one state to another
Statecharts

• What’s an action?

An **action** is a run-to-completion behavior. The object will not accept or process any new events until the actions associated with the current event are complete.

• What’s the order of action execution?

(1) exit actions of current state
(2) transition actions
(3) entry actions of next state
Basic Syntax 1

A
entry / g(x), h(y)
exit / m(a), n(b)
do / act(a,y,z)
defer / e1, e2
   e3 / p(x,y), q(z)

B

T1(int r)[r < 0] / f(r)

- event parameters
- event name
- guard
- action list
- entry actions
- exit actions
- activities
- deferred events
- internal transition
- state name
- state
UML State Events

• An *event* is a noteworthy occurrence:
  – A designated condition coming true (guard)
  – Receipt of an explicit signal from another object
  – Receipt of a call from an object
  – Timeout (passage of a specified interval)

• Events can cause *state transitions*
Types of Events

- UML defines 4 kinds of events
  - Signal Event
    - Asynchronous signal received
    - e.g. evFlameOn
  - Call Event
    - operation call received
    - e.g. op(a,b,c)
  - Change Event
    - change in value occurred
  - Time Event
    - Relative time elapse
    - Absolute time arrived
    - e.g. tm(PulseWidthTime)
And-States

- fork
- and-state separator
- join
Pseudostates

- Initial pseudostate
- History pseudostate
- Diagram connector
- Conditional pseudostate
- Junction pseudostate
- Fork pseudostate
- Join pseudostate
- Terminal pseudostate
Activity Diagrams

• A kind of specialized state machine
• Statecharts transition *primarily* when they receive an event
• Activity diagrams transition *primarily* when the work done in an activity or action state completes
  – Used for algorithmic design
• May be thought of as a “concurrent flowchart”
Activity Diagram

- Initial action state
- Action state
- Conditional pseudostate
- Fork
- Join

[Diagram with nodes and edges representing the activity diagram, including states like 'waiting', 'initialising', and events like 'evGo/count = params->count;', 'evBase', 'evShoulder', 'evElbow', 'evWrist', 'tm(3000)', 'evGrip', and 'gripReady']
Representing Algorithms in the UML

• Normally algorithms are an operation description which may include multiple operation calls and actions. These can be modeled in the UML in a variety of ways:
  – Statecharts
    • Excel at representing event driven algorithms, and the parallelism permitted by the algorithm
  – Activity Diagrams
    • For algorithms whose program flow is determined by the completion of processing steps
    • MOST COMMON
  – Sequence and collaboration diagrams
    • Weaker than activity diagrams
Algorithmic Example: Run-Time Data Interpolation

UML Use Case Model
Use Cases

- *Use cases* are a representation of sets of system uses
- Use cases model external visible behavior without revealing internal implementation
- Help capture requirements during analysis
- Fundamentally a behavioral (functional) decomposition
- Provide test cases for the system during validation
A Use Case is used to

- Capture requirements of a system
  - Functional requirements
    - *What* the system does
  - Quality of service (QoS)
    - *How well* the system does it
      - speed
      - throughput
      - predictability
      - capacity
      - Safety, reliability, reusability, maintainability...

Functional requirements are modeled as use case, message sequences, or statecharts

QoS Requirements are modeled as *constraints*
Use Cases

• Capture a system’s functional requirements
• Provide an external view of the system’s functionality
• Class and object models provide an internal view of the system’s structure
• Use cases can provide sets of test cases
• Use cases are detailed by
  – Providing example scenarios
  – Specification (e.g. statecharts)
Example: Cardionada

- Cardiac pacemaker can pace the heart in different operational modes, Off, AAI, VVI, AAT, VVT, and AVI
  - ([pace chamber][sense chamber][action on sense])
- Pacemaker can be commanded by the external programmer to:
  - Set pacing mode
  - Set pacing parameters (rate, pulse width, pulse amplitude)
  - Return pacemaker status
Example: Cardionada QoS

- **Pacing**
  - Pacing rate may be set in units of 1 ppm from 30..120 inclusive. Accuracy of pacing shall be <= 100 ms.
  - Pulse width may be set in units of ms from 1..15. Accuracy of pulse width shall be within 0.25ms.
  - Pulse amplitude may be set in units of mV from 10..100. Accuracy shall be within 2 mV.

- **Communications**
  - between the programmer and Cardionada shall be reliable with an MTBF of 0.01 in an electrically noisy environment.
  - Single and dual bit errors shall be caught with an reliability of at least 0.0001%. Errors detected by the pacemaker shall be reported to the programmer. The programmer shall not assume successful transmission without explicit acknowledgement.
Example: Cardionada

- Programmer
- Set Pacing Parameters
- Set Operational Mode
- Report Pacemaker Status
- Pace the Heart
- Cardionada
- Rate set in units of ppm from 30..120.
- Pulse width set in units of ms from 1..15.
- Pulse amplitude set in units of mV from 10..100
- Modes: Off, AAI, VVI, AAT, VVT, AVI.
- See "Cardionada Comm Reliability.HTML" for communications reliability
- Rate accuracy +/- 100ms
- Pulse width accuracy +/- 0.25ms
- Pulse amp. accuracy +/- 2mV
Detailing Use Cases

• By example
  – Use scenarios captured via sequence diagrams
    • Each scenario capture a specific path through the use case
  – Partially constructive
  – Infinite set, so you must select which are interestingly different
  – Non-technical users can understand scenarios

• By specification
  – Use a statechart to represent all possible paths
  – Fully constructive
  – Non-technical readers may not understand it
Use Cases → Object Analysis and Design

- Use cases define requirements
  - functional
  - QoS
- Use case are detailed
  - sequence diagrams
  - text and text annotations
  - statecharts
- Use cases are “black box” (i.e. do not reveal design or implementation)
- Use cases are realized by collaborations.
- Collaborations are shown by
  - Sequence Diagrams
  - Collaboration Diagrams

A collaboration is a set of object roles working together to achieve a higher-level function.
Use Cases → Object Analysis and Design

Object model realizing the use case

Identify Functional and QoS Requirements

Deliver Anesthesia

Patient

Use case

Collaboration

Apply object identification strategies
Defining objects in collaborations

- Defines domains
- Selects use cases
- Applies object identification strategies
- Constructs collaboration from objects
- Refines scenarios
- Adds generalization
- Reifies objects into classes
- Refines domains

Involves iteration

- Includes defining operations and attributes
Object Identification Strategies

• Strategies
  – Nouns in problem statement
  – Actors or Causal Agents
  – Services to be performed
  – Transactions
  – Physical devices
  – Key concepts
  – Persistent Data

• Test via execution
  – Players in scenarios
Advanced UML Stuff
Constraints

• Constraints are *user-defined well-formedness rules that apply to one or more model element*
  – Constraints may be structural, behavioral, or refer to any kind of quality of service

• Constraints are usually specified within curly braces:
  • \{ b - a < 10 s \}
  • \{ nextPtr.Prev == prevPtr.Next \}

• Constraints are often placed within Notes boxes for clarity
Constraints are most commonly used to capture:

- Performance of use cases, operations, state transitions (i.e. *quality of service properties* or requirements)
  - Worst case execution time
  - Average execution time
  - Blocking Time
  - Deadline
  - Throughout
  - Bandwidth

- Scheduling Information
  - Priority (of active objects)
  - Period
  - Minimum interarrival time
Tagged Values

• An explicitly named property-value pair
• Predefined UML tagged values:
  – documentation
  – location (component, node)
  – persistence (instance, attribute)
  – postcondition (operation)
  – precondition (operation)
  – responsibility
  – semantics (type, operation)
  – space semantics (type, operation)
  – time semantics (type, operation)
Example

Constraints are shown in curly braces { }.

Stereotype icon for active object is a class with a heavy boarder.

Task_A

Task_B

Task_C

DataClass

mutex

{ resource locked for 10ms}

{ resource locked for 40ms }

{ Priority = 10
Period = 100ms
Deadline = 80 ms
Execute time = 10 ms } { Priority = 15
Period = 250ms
Deadline = 250ms
Execute time = 200 ms } { Priority = 20
Period = 500ms
Deadline = 500ms
Execute time = 50 ms }
Architecture in UML
Physical Architectural Views

- Subsystem and component view
- Distribution View
- Resource and Concurrency View
- Safety and Reliability View
- Deployment View
Physical Architectural Views

- Construct architectural design models
  - Subsystem Model
  - Concurrency Model
  - Distribution Model
  - Safety and Reliability Model
  - Deployment Model

- Capture with
  - Class Diagrams
  - Package Diagrams
  - Subsystem Diagrams
  - Task Diagrams
  - Deployment Diagrams
System Level Architecture

[Diagram showing the System Level Architecture with components such as Aircraft, Controller, Primary RADAR, Secondary RADAR, National Airspace System, and Flight Planning.]
Subsystem Architecture
Component Architecture
Subsystem and Component View

• **A subsystem**
  – is a large object that provides opaque interfaces to its clients and achieves its functionality through delegation to objects that it owns internally
  – contains components and objects on the basis of *common run-time functional purpose*
  – a metasubtype of both *Classifier* and *Package*

• **A component**
  – is the basic reusable element of software
  – organizes objects together into cohesive run-time units that are replaced together.
  – provides language-independent opaque interfaces
Collaboration Architecture
Distribution Architecture

• Distribution model refers to
  – Policies for distribution objects among multiple processors and communication links, e.g.
    • Asymmetric distribution (dedicated links to objects with *a priori* known location)
    • Publish-Subscribe
    • CORBA and Broker symmetric distribution
  – Policies for managing communication links
    • Communication protocols
    • Communication quality of service management
Distribution Architecture
Safety and Reliability Model

- Safety and reliability model refers to the structures and policies in place to ensure
  - Safety
    - Freedom from accidents or losses
  - Reliability
    - High MTBF
    - Fault tolerance
- Safety and fault tolerance *always* require some level of redundancy

Safety and reliability of object models is described more completely in *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*.
Safety and Reliability Architecture

Actuation Channel 1

input processing → data transformation → output processing

data integrity checks

monitoring data source → Actuation Channel 2

input processing → data transformation → output processing

data integrity checks

control signal → Actuator
Concurrency Architecture

- Refers to
  - Identification of task threads and their properties
  - Mapping of passive classes to task threads
  - Identification of synchronization policies
  - Task scheduling policies

- Unit of concurrency in UML is the «active» object
  - «active» objects are added in architectural design to organize passive objects into threads
  - «active» objects contain passive semantic objects via composition and delegate asynchronous messages to them
## Task Identification

<table>
<thead>
<tr>
<th>Task Identification Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single event groups</td>
<td>for simple systems, you may define a thread for each event type</td>
</tr>
<tr>
<td>Event source</td>
<td>group all events from a single source together for a thread</td>
</tr>
<tr>
<td>Related information</td>
<td>For example, all numeric heart data</td>
</tr>
<tr>
<td>Interface device</td>
<td>For example, a bus interface</td>
</tr>
</tbody>
</table>

### Event properties
- Events with the same period, or aperiodic events

### Target object
- For example, waveform queue or trend database

### Safety Level
- For example, BIT, redundant thread processing, watchdog tasks

Best for schedulability
Defining the Concurrency Model

Apply Task Identification Strategies

Construct <<active>> object for each

Add passive semantic objects into <<active>> objects via composition

Refine collaboration

Refine scenarios

{for all identified tasks}

Test by scenario execution

Apply schedulability analysis

[remaining tasks?]

[else]
Concurrency Model

• Active object is a stereotype of an object which owns the root of a thread
• Active objects normally aggregate passive objects via composition relations
• Standard icon is a class box with heavy line
Task Diagram

- A task diagram is a class diagram that shows only model elements related to the concurrency model
  - Active objects
  - Semaphore objects
  - Message and data queues
  - Constraints and tagged values

- May use opaque or transparent interfaces
Elements of Task Diagram

Mutex semaphore

Message queue

«active» object as basis for threads

Schedulability properties

RobotArm_Task

CmdQueue:Queue

(period = 10
priority = 3
worst case execution time = 5
deadline = 10)
Deployment Architecture

- Maps software components and subsystems to hardware
  - Represents a device as a node
  - Iconic stereotypes are common
- Identifies physical connections among devices
  - May be buses, networks, serial lines, etc
- Two primary strategies
  - Asymmetric
    - Design-time mapping of software elements to HW
  - Symmetric
    - Dynamic run-time mapping of software elements to HW
Deployment Architecture
Testing
How do we know it’s RIGHT?

• Ans: *We test it*

• The purpose of testing is
  – To demonstrate the correctness and adequacy of the design and implementation
  – Identify defects so they can be corrected

• Strategic Defects
  – Requirements or Architecture flaws
  – Up to 10,000x the cost of coding errors
Testing Concepts

• A test suite consists of a potentially large set of test vectors

• A test vector is a known sequence of input, causal events and messages with an expected unambiguous output

• A test plan is a written set of test vector specifications

• Test procedures are documented step-by-step approach for executing the Test Plan
Testing Goals

- **Goals**
  - Demonstrate that the system is correct both in terms of functionality and QoS
  - Identify strategic defects as early as possible so as not to let them persist and fester
  - Always construct the system with design aspects that demonstrably work
  - Identify no more than 10% of the defects in the last 10% of the project time

- **Policies**
  - *Continuous* testing not test-at-the-end
  - *Requirements-based* testing to show that the design pieces always work – i.e. prototype-based spiral development
Early Architectural Testing

Prototype 1

Name: 'Hello World'
Misson:
• Subsystem Architecture
• Data Acquisition
• Basic UI for monitoring

Prototype 2

Name: 'Revision 1'
Misson:
• Basic Distribution Architecture
• Data Waveform Display
• User setting of control values
• Data Logging

Prototype 3

Name: 'Customer Review Prototype'
Misson:
• Reliable Distribution Architecture
• Reliable transport protocol
• Sockets
• Closed Loop Control
• Built in Test
Deriving Test Vectors

- A scenario is an example execution of a system
- A test vector is an example execution of a system with a known expected result
- Scenario → Test Vector
  - Capture preconditions
  - Determine test procedure
    - Identify causal messages and events
    - Instrument test fixtures to insert causal messages
  - Remove optional or “incidental” messages
  - Define pass/fail criteria
    - Identify effect messages / states
    - Postconditions
    - Required QoS
What do we want to automate?

• Capturing of scenarios during execution
• Conversion of requirements scenarios to test vectors
• Application of test vectors to system
• Gathering of pass/fail statistics
• Identification of points of failure in the model not the code
Model Execution for Test/Debug

- Test and debug at the same level of abstraction as your design!
- Test either on your desktop or on your target hardware
Automated SB Testing

Specification of test inputs

Requirements Sequence-Diagrams as test vectors

Stimulate...

...Monitor

Scenario-based Test Execution Environment

% Completion

Pass / Fail

Specification of test inputs

Requirements Sequence-Diagrams as test vectors

Stimulate...

...Monitor

Scenario-based Test Execution Environment

% Completion

Pass / Fail
Automated Regression Testing

- Requirements Sequence-Diagrams as test vectors
- Specification of test inputs
- Scenario-based Test Execution Environment
- Stimulate...
- ...Monitor
- Automatically Generated Sequence-Diagrams
- % Completion
- Pass/Fail
Stimulating the System Under Test

Test Environment binds actual instances to instance parameters as necessary

Test Environment or user plays the “Collaboration Boundary”

Test Environment binds actual values to passed operation parameters as necessary
Test Results at the Model Level

Test passed

Test failed
ROPES Process
The ROPES process is described more completely in *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*.
Why Process?

• A good process
  – Improves the product quality
  – Saves calendar and work time
  – Makes development efforts more predictable, schedulable, and reliable
  – Enables rapid response to looming risks
Keys to Effective Development

- Iterative Development
- Real-Time Frameworks
- Visual Modeling
- Automated Requirements-Based Testing
- Model-Code Associativity
- Model Execution
Executable Models

- **You can’t test what you can’t execute**
- The best way to avoid having bugs is to not put them in the system
- Rapid execution allows you to incrementally construct your system and remove bugs easily and quickly
- It’s important to execute at the design (model) level even more than at the code level during test and debug
Model-Code Associativity

- When code deviates from your visual models, what do you do?
- The model is a coherent set of integrated abstractions that represents your system.
- Solution: think of the code as one of several possible views of the model.
- Change one view, the others change automatically.
Requirements-Based Testing

- Testing is difficult, very expensive, time-consuming, and essential
- Testing should not come at the end but should be done throughout development
- Test vectors should be automatically generated from the requirements specification
- Test execution and gathering of pass/fail statistics can and should be automated as much as possible
Requirements-Based Testing

- Requirements Sequence-Diagrams as test vectors
- Specification of test inputs
- Stimulate...
- Monitor...
- Scenario-based Test Execution Environment
- % Completion
  - Pass
  - Fail
Real-Time Frameworks

- Every application is 60-90% “housekeeping code” which is redeveloped every time we build a system.
- A framework is a partially completed application which you customize and specialize for your particular application project.
- A real-time framework is a “vertical framework” providing an integrated set of design patterns selected and optimized for real-time and embedded applications.

Application

specializes ➤ extends ➤ uses ➤ executes
Iterative Development

- Iterative development *incrementally* constructs your system.
  - Each increment is called a *prototype*
- It allows the early execution of your architecture
  - Uncovers architectural defects early while they are inexpensive to fix
  - Allows early identification and reduction of project risks
- To be maximally effective, relies on executability of models
Key Process Concepts

Worker Role performs Activity

- Process guidelines and standards
- Work guidelines and standards
- Performs
- Creates, manages, or uses
- Works on
- Results in

Artifact

- Template
- Guideline
- Defect Report
- Schedule
- Tool
- Configuration Management System
ROPES Spiral Lifecycle

- **Macro**: Focus on key concepts
- **Micro**: Focus on refinement of concepts
- **Nano**: Focus on design and implementation
- **Nano**: Focus on optimization & deployment

**Rev of a process step within a phase of the microcycle.**
*Typically 30 minutes to 1 day.*

**Project concept to final delivery.**
*Typically 1 yr to several years.*

**Single iteration resulting in a single prototype.**
*Typically 4-6 weeks.*
## Alternative Macrocycle: The Semi-spiral Macrocycle

<table>
<thead>
<tr>
<th>Requirements Analysis</th>
<th>Systems Engineering</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Electronic Systems Analysis</strong></td>
<td><strong>Mechanical Systems Analysis</strong></td>
</tr>
<tr>
<td><strong>Electronic Systems Design</strong></td>
<td><strong>Mechanical Systems Design</strong></td>
</tr>
<tr>
<td><strong>Electronic Systems Translation</strong></td>
<td><strong>Mechanical Systems Translation</strong></td>
</tr>
</tbody>
</table>

### Integration and Test

### Validation Testing
ROPES Microcycle Flow

- Analysis: defines the properties of all acceptable solutions
- Design: specifies a particular optimal solution
- Translation: translates model into executable code and constructs working architectural pieces of prototype
- Test: Integrates architectural pieces into prototype and validates
- Party: Project planning and on-going assessment and replanning
Analysis in ROPES

- Systems Engineering
- Object Analysis
- Architectural Design
- Detailed Design
- Mechanistic Design
- Coding
- Testing
- Integration And Test
- Validation Testing
- Iterative Prototypes
- Party!

Analysis

Requirements Analysis

Systems Engineering

Object Analysis

Translation

Coding

Unit Testing

Integration And Test

Validation Testing

Iterative Prototypes

Party!
Design in ROPES

Design

Mechanistic Design

Detailed Design

Architectural Design

Analysis

Object Analysis

Systems Engineering

Requirements Analysis

Iterative Prototypes

Validation Testing

Integration And Test

Unit Testing

Coding

Translation

Testing

Party!

I-Logix
Translation in ROPES

Translation

Coding

Testing

Unit Testing

Integration And Test

Validation Testing

Iterative Prototypes

Design

Mechanistic Design

Detailed Design

Architectural Design

Object Analysis

Analysis

Systems Engineering

Requirements Analysis

Coding Party!...

Page 126
Get More Information

www.ilogix.com