Model-Based Agility for Real-Time Systems Development

Dr. Bruce Powel Douglass, Ph.D.
Chief Evangelist
IBM Rational
Bruce.Douglass@us.ibm.com
Twitter: @BruceDouglass
http://tech.groups.yahoo.com/group/RT-UML/

What’s wrong with code?

- Answer:
  - Code is a necessary but insufficient structural model of the system
  - Code is a 1-dimensional, very detailed structural view
  - Every other view must be inferred
    - High level structure
    - Dynamic behavior
  - With large systems, code-based systems are *unmanageable!*

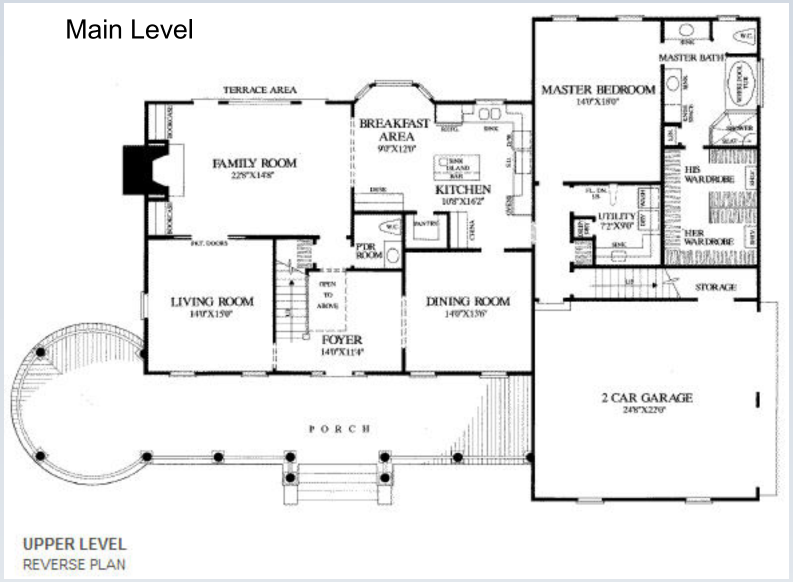
- Why?
  Because to understand complex systems you need to understand
  - How pieces *of different scale and abstraction* work together
  - How *different aspects (structural, behavioral) of the systems work*

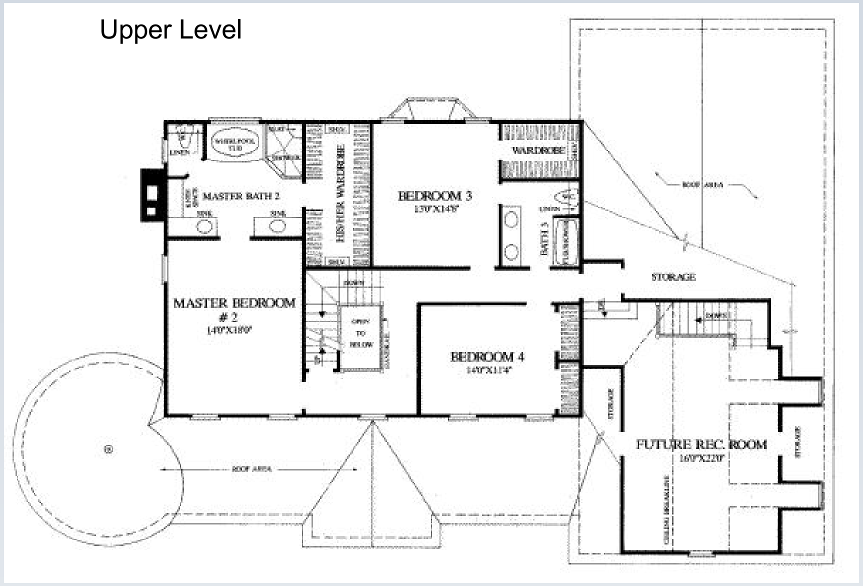
But Why Modeling???

- Assume you got a great bonus at work…
- You and your wife want to build a new home…
- You meet with one architect…
- Two months later he comes back with a 647 pages document:
  - *… indented by 7 meters from the west border of the premises, there is a left corner of the house*
  - *… The entrance door is indented by another 3.57 meters*
  - *… 2.30 meters wide and 2.20 meters high, left-hand hinge, opening to the inside*
  - *… If you come in, there are two light switches and a socket on your right, at a height of 1.30 meters*
  - *…*
- Is it right? Are the requirements correct? Accurate? Consistent? How can you tell?
- You are NOT happy… ☹ ☹ ☹

Yes… Modeling!

- Then you call another architect… And two weeks later he comes with this:

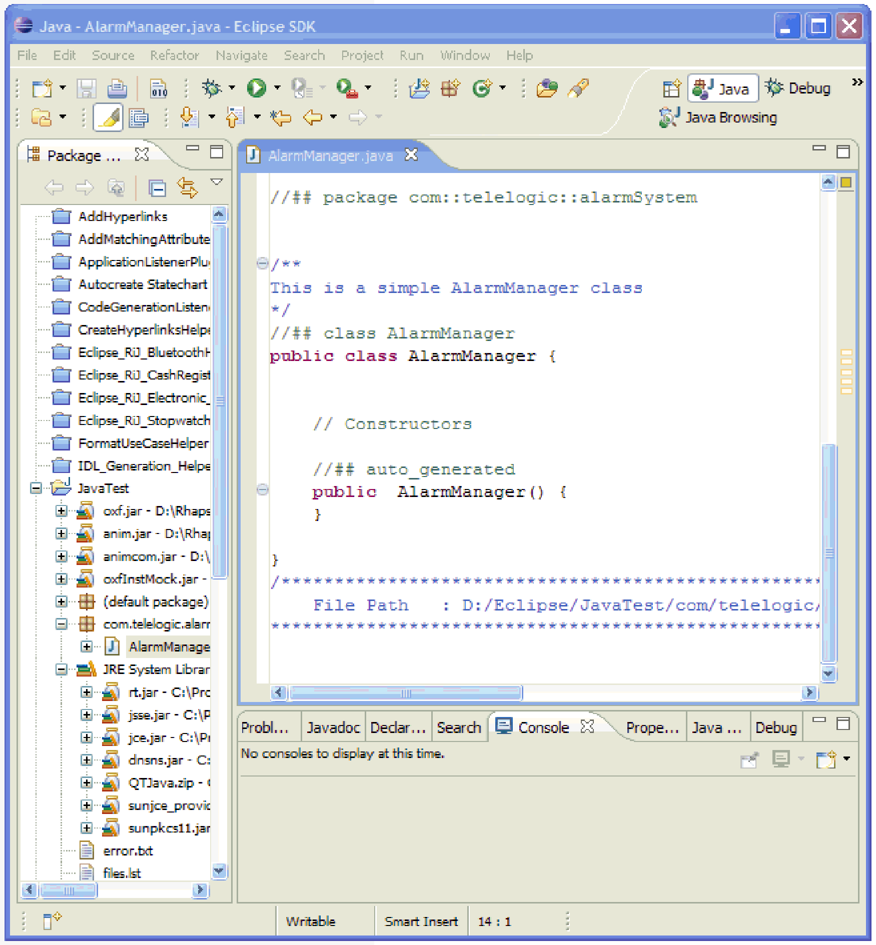




- You are very happy! ☺ ☺ ☺
- The second architect used ***Modeling*** to show different ***Views*** of the house!
  - Structural
  - Floor layout
  - Electrical
  - Plumbing
  - Heating

Hand Coding and Modeling Co-Existence

- Dynamic Model Code Associativity
- Change one view, the others *change automatically!*

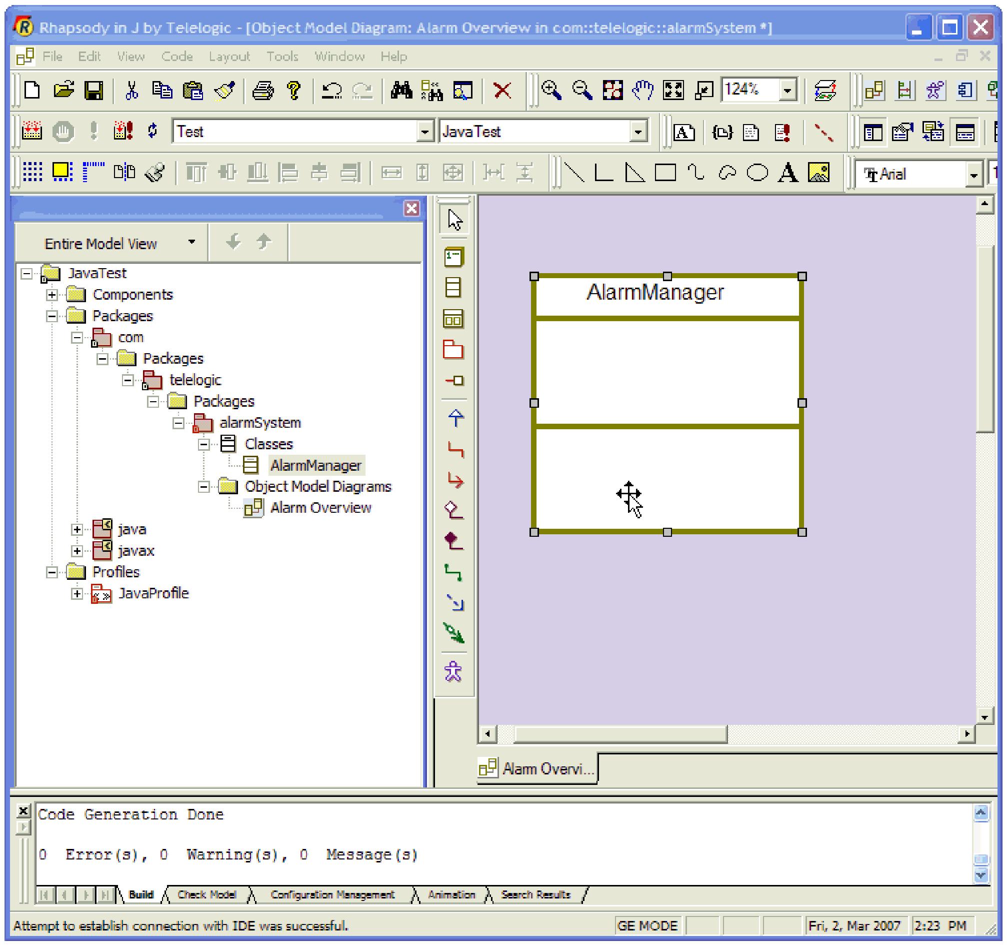


Java - AlarmManager.java - Eclipse SDK

```
//## package com::telelogic::alarmSystem


/**
This is a simple AlarmManager class
*/
//## class AlarmManager
public class AlarmManager {


    // Constructors

    //## auto_generated
    public AlarmManager() {
    }

}
/*****************************************************
    File Path   : D:/Eclipse/JavaTest/com/telelogic/
*****************************************************
```

No consoles to display at this time.

Writable | Smart Insert | 14 : 1



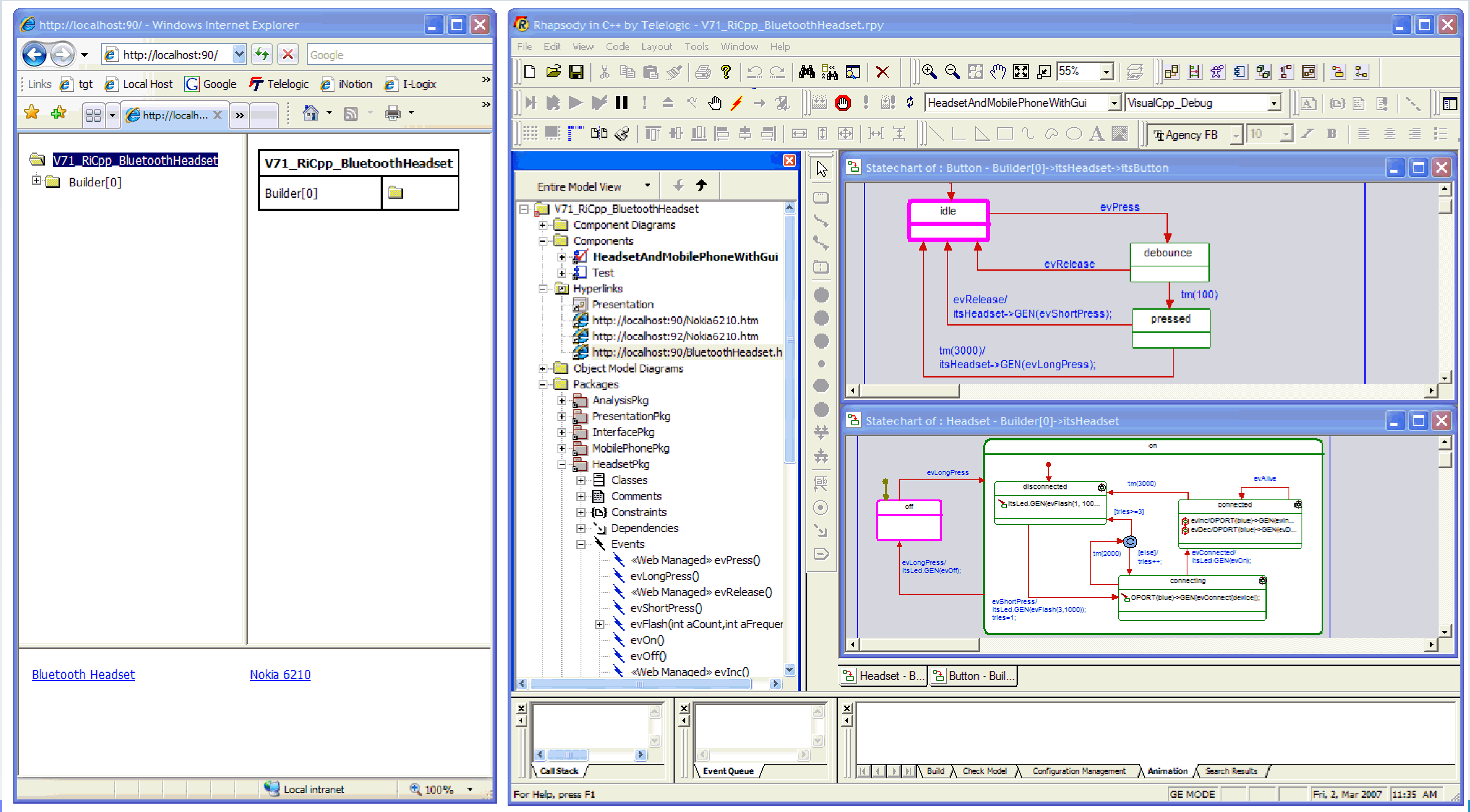
Rhapsody in J by Telelogic - [Object Model Diagram: Alarm Overview in com::telelogic::alarmSystem *]

```
Code Generation Done

0  Error(s), 0  Warning(s), 0  Message(s)
```

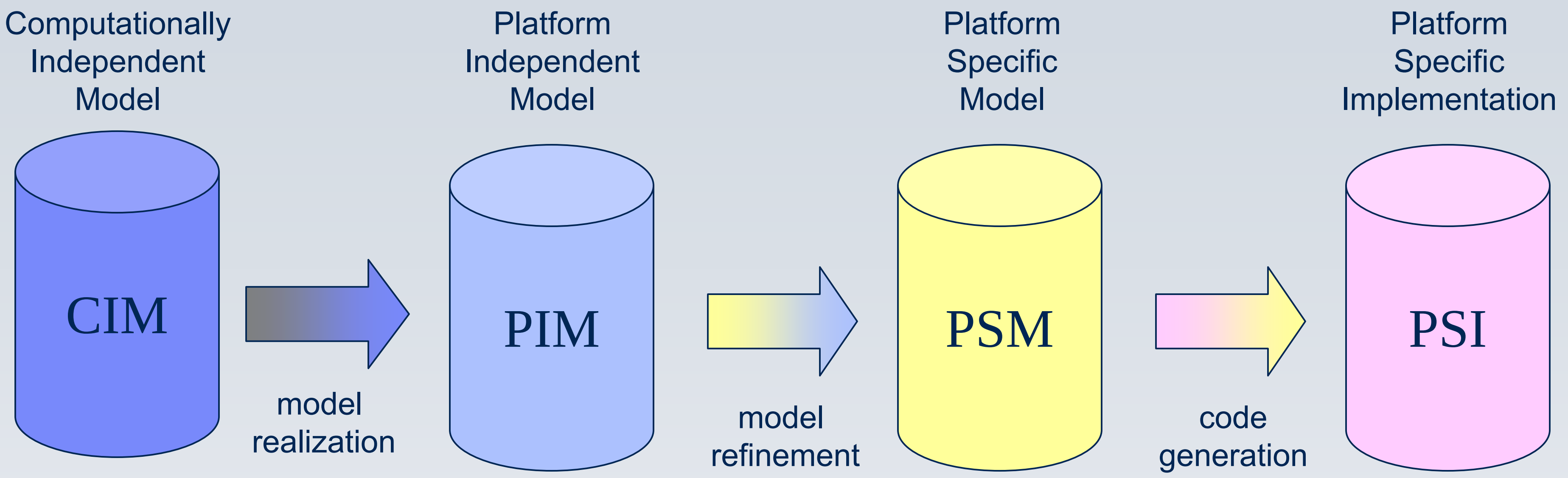
Attempt to establish connection with IDE was successful. | GE MODE | Fri, 2, Mar 2007 | 2:23 PM

Design Level Debugging with Rapid GUIs

- Virtual prototype / Panel graphics support
  - Ideal communications aid for design reviews and general sharing of information.



UML and MDA


Computationally Independent Model
CIM
model realization
Platform Independent Model
PIM
model refinement
Platform Specific Model
PSM
code generation
Platform Specific Implementation
PSI

Use cases clustering requirements and detailed with sequence, state and activity diagrams

Formal specifications (in UML) of the structure and function of the system that abstracts away technological details.

Refines PIM by adding technical details, such as
- middleware
- operating system
- network
- CPU characteristics

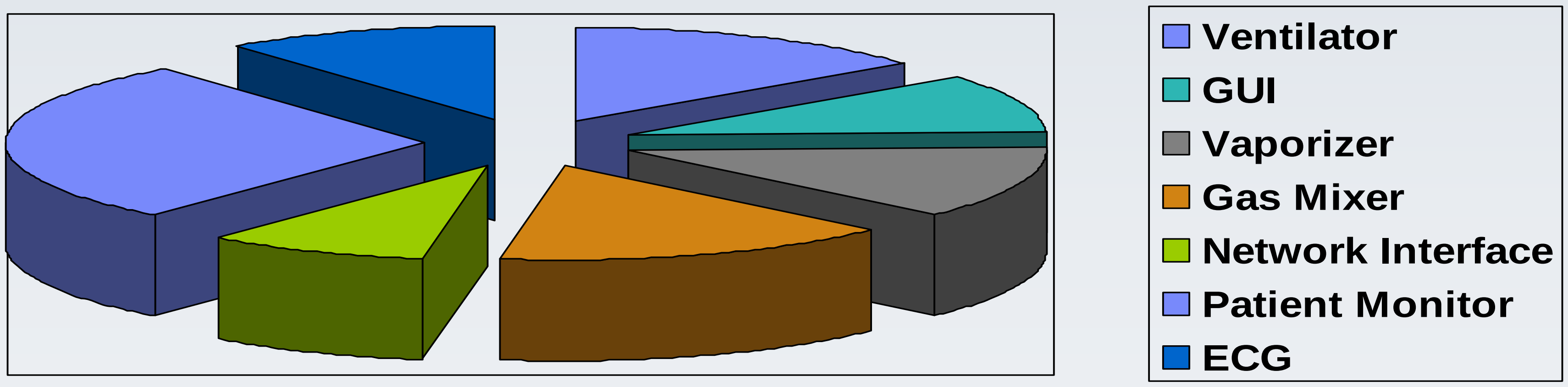
Source (and compiled) code produced from the model, running on the desired target platforms

How is MDD different than CDD?

Aspect	Code-Driven Development	Model-Driven Development
Primary design view	Source code	Model
Primary architecture view	Text	Model
Creation of source code	Manually	Auto-generated from models
Configuration managed artifacts	Source code	Models (and optionally source code)
Primary artifact tested by developer during unit test and debugging	Source Code	Models
Representation for requirements	Text	Text + model
Representation for test vectors	Text	Models + text
Primary artifact tested by validation test	Compiled code	Compiled code
Artifacts reviewed	Source code	Model
Porting application to new platform, OS, or middleware	Review and rewrite all appropriate code	Change PIM transformational rules

Moving From Code to Models

- Don’t move all the code at once in a “big-bang integration”
- DO – migrate the code incrementally:
  - Take one subsystem or component
  - Make a model of it either with reverse engineering or by replacing it
  - Generate code from modeled subsystem
  - Demonstrate (test) correctness of the system with model in place
  - Repeat until done



UMMI – UML Maturity Model Index

- Level 0 - Code
  - Emacs, vi, Visual Studio, or Eclipse code pounding
- Level 1 – Code Visualization (5% of possible benefit)
  - Reverse engineering of code intro structural views but retaining code-based concepts of Files, Functions, and Variables
- Level 2 – Structural Modeling (15% of possible benefit)
  - Forward Engineering of code frames
  - Use of use cases with sequence diagrams and requirements tracing
- Level 3 – Behavioral Modeling (30% of possible benefit)
  - Use of state machines and activity diagrams for classes
  - Formal specifications of use cases with state machines
- Level 4 – Model Based Execution & Debugging (70% of possible benefit)
  - Executable modeling with model-level debugging
  - Executable requirements models with use cases
- Level 5 – Optimizing Model Usage (100% of possible benefit)
  - Architectural modeling with UML – subsystems, concurrency, distribution, safety and reliability, and deployment
  - Continuous execution with constant feedback, periodic assessment of risks, model organization

Harmony™ for Embedded RealTime Agile Principles

- Your primary goal: develop working software
- Your primary measure of progress is *working software*
- Continuous feedback is crucial
- Five Key views of architecture define your architecture
- Secondary architectural views supplement your architecture
- Plan, Track, and Adapt
- Leading cause of project failure is ignoring risk
- Continuous attention to quality is essential
- Modeling is crucial
- Optimize the right things

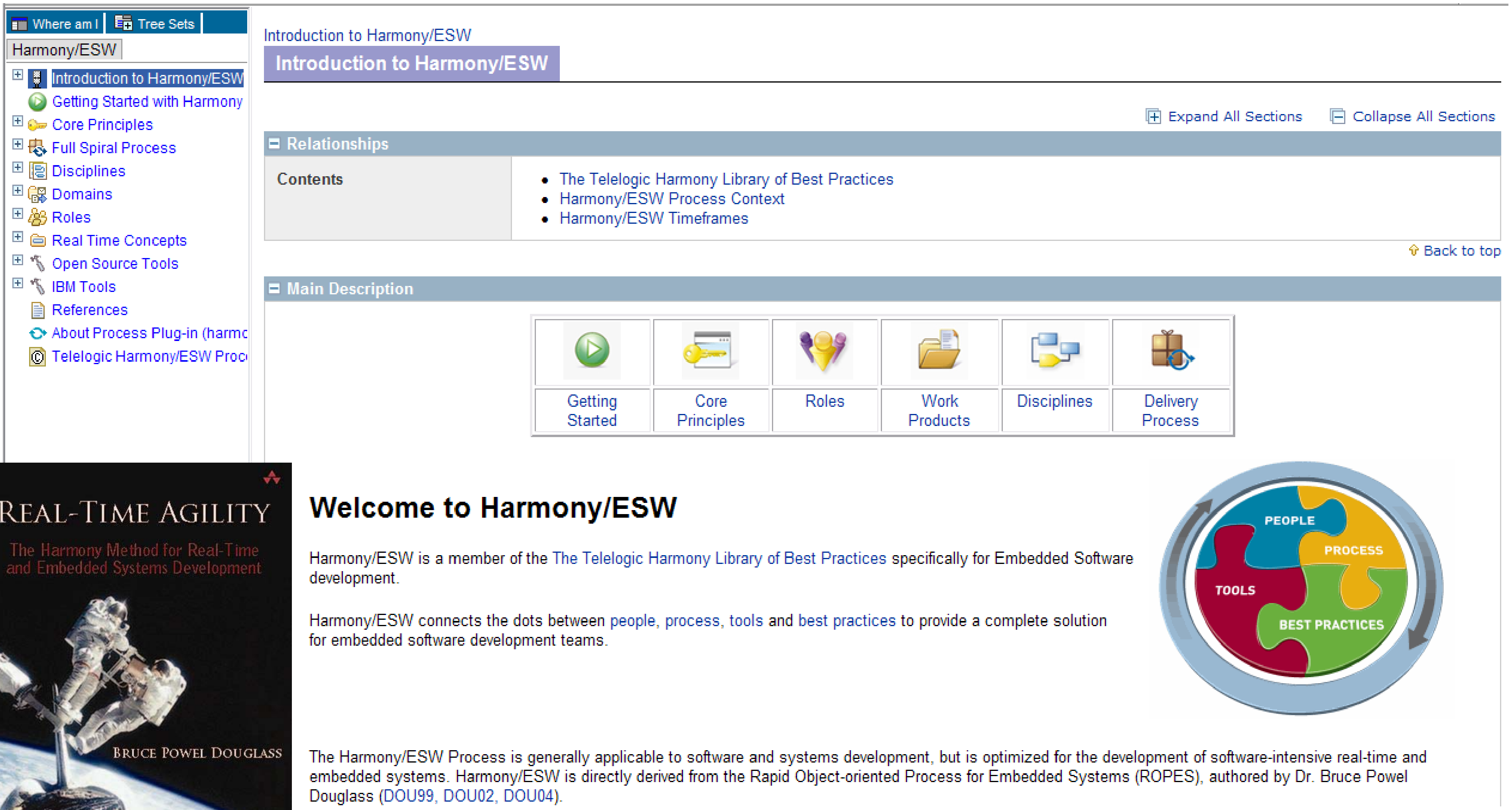
Agile methods are a very disciplined approach to software and systems development emphasizing these principles.

Harmony™ for Embedded RealTime Agile Practices

- Use dynamic 2-level planning
- Incrementally construct several times per day
- Minimize overall complexity
- Model with a purpose
- Develop and apply tests as your develop your software
- Prove it's correct – continually
- Avoid defects with defensive development
- Apply design patterns Intelligently
- Manage interfaces to ease integration
- Use model-code associativity

Practices are workflows that produce and consume work products to achieve the goals based on principles and concepts

The Harmony™ Process



Where am I | Tree Sets

Harmony/ESW

- Introduction to Harmony/ESW
- Getting Started with Harmony
- Core Principles
- Full Spiral Process
- Disciplines
- Domains
- Roles
- Real Time Concepts
- Open Source Tools
- IBM Tools
- References
- About Process Plug-in (harmc
- Telelogic Harmony/ESW Proc

Introduction to Harmony/ESW

Introduction to Harmony/ESW

Expand All Sections   Collapse All Sections

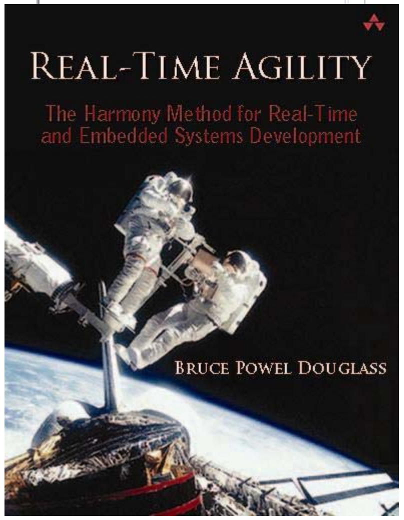
Relationships

Contents	• The Telelogic Harmony Library of Best Practices • Harmony/ESW Process Context • Harmony/ESW Timeframes

Back to top

Main Description

Getting Started	Core Principles	Roles	Work Products	Disciplines	Delivery Process


REAL-TIME AGILITY
The Harmony Method for Real-Time and Embedded Systems Development
BRUCE POWEL DOUGLASS

Welcome to Harmony/ESW

Harmony/ESW is a member of the The Telelogic Harmony Library of Best Practices specifically for Embedded Software development.

Harmony/ESW connects the dots between people, process, tools and best practices to provide a complete solution for embedded software development teams.

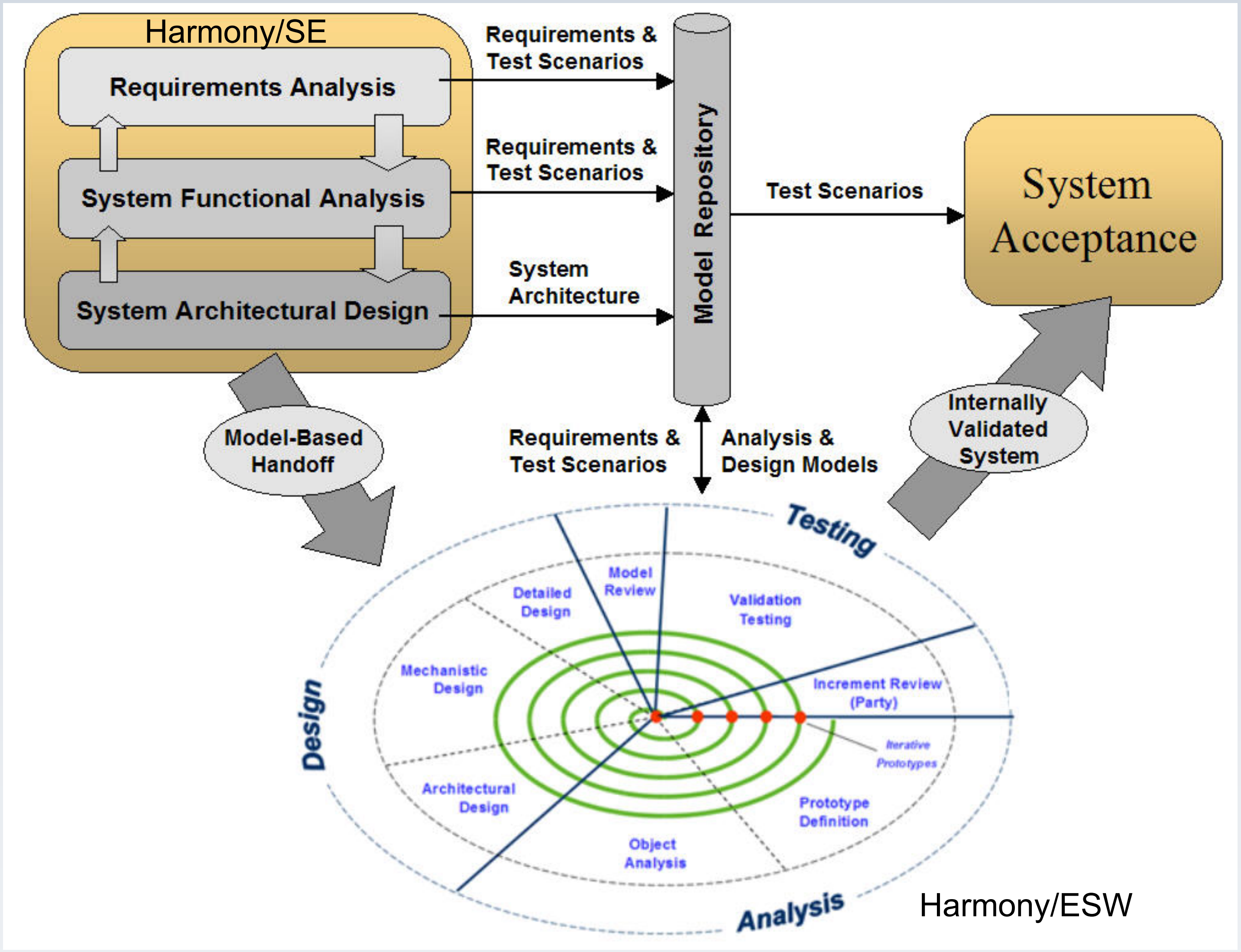
PEOPLE
PROCESS
TOOLS
BEST PRACTICES

The Harmony/ESW Process is generally applicable to software and systems development, but is optimized for the development of software-intensive real-time and embedded systems. Harmony/ESW is directly derived from the Rapid Object-oriented Process for Embedded Systems (ROPES), authored by Dr. Bruce Powel Douglass (DOU99, DOU02, DOU04).

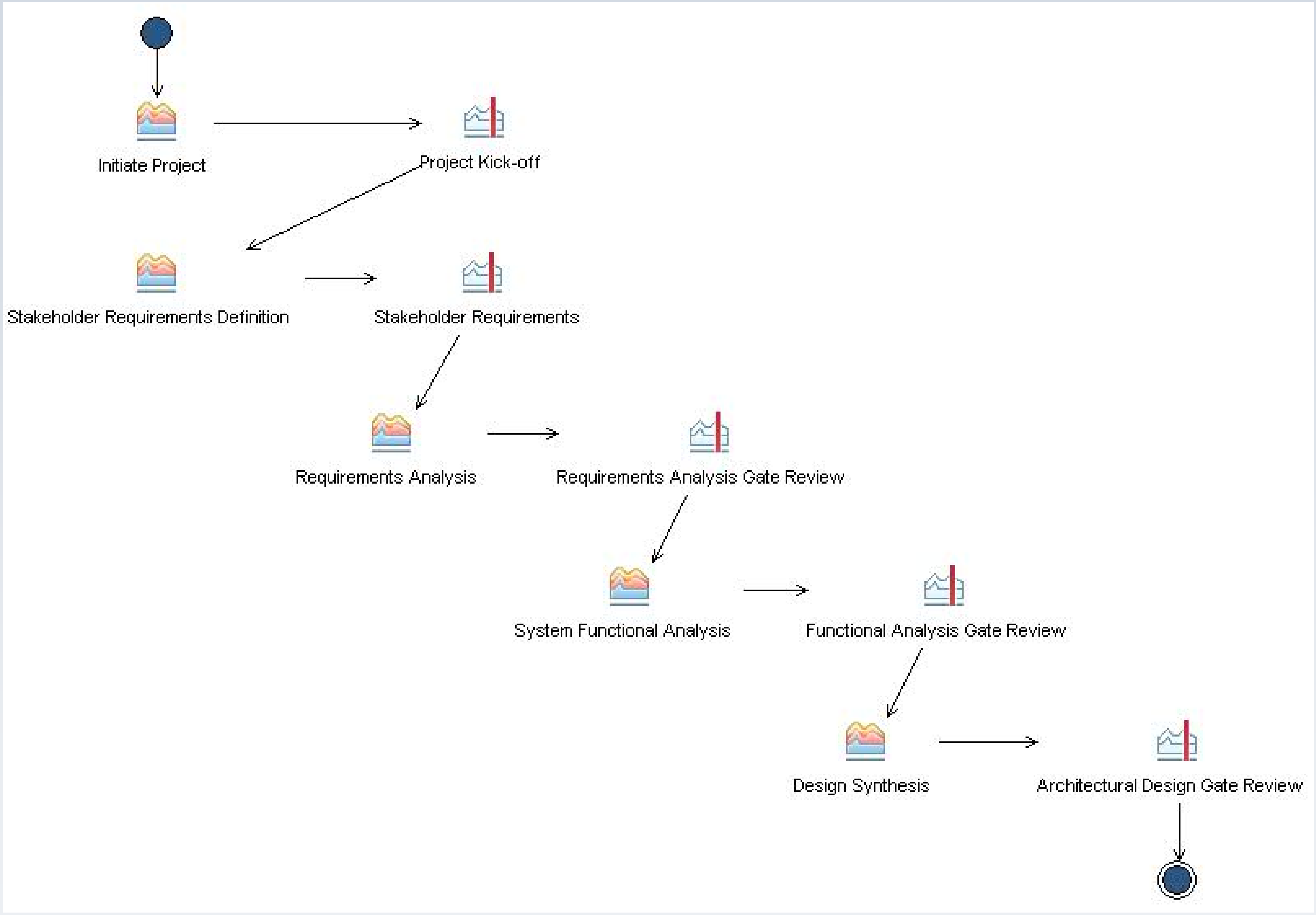
A Tale of Two Delivery Processes

- Harmony provides different delivery processes for different project types
  - A *delivery process* is a specific aggregation of various best practices (called *capability patterns*)
- ***Harmony/SE*** is a delivery process for performing model-based system engineering with SysML/UML
  - Clarifies requirements with use cases, scenarios, and state machines
  - Specifies architecture with SysML/UML block diagrams, interfaces, and constraints
- ***Harmony/ESW*** is a delivery process without an up-front system engineering activity, used for projects
  - That use existing or COTS hardware
  - That are primarily focused on the development of embedded software
- ***Harmony/Hybrid*** is a delivery process with an up-front system engineering phases, used for projects
  - That have significant hw/sw co-development
  - That have a significant system engineering effort
  - That must “hand-off” work products from system engineering to the development of embedded software

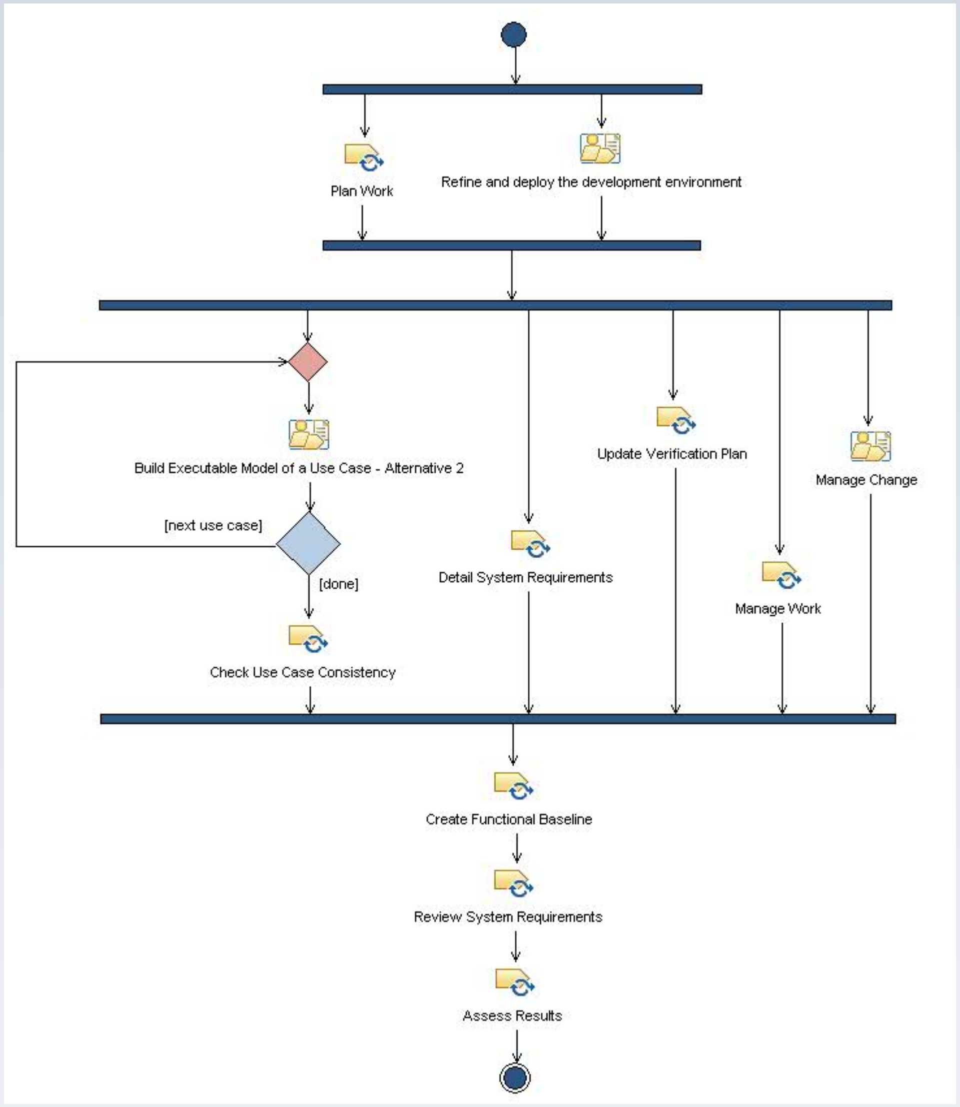
Harmony/Hybrid Delivery Process



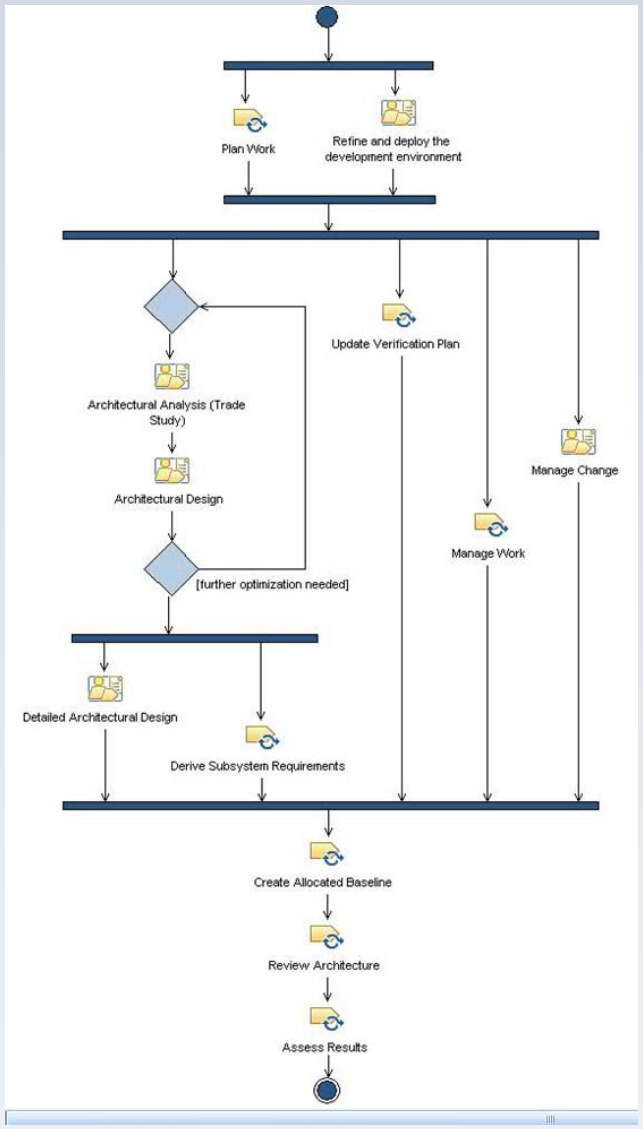
Harmony-SE Delivery Process Overview


Initiate Project
Project Kick-off
Stakeholder Requirements Definition
Stakeholder Requirements
Requirements Analysis
Requirements Analysis Gate Review
System Functional Analysis
Functional Analysis Gate Review
Design Synthesis
Architectural Design Gate Review

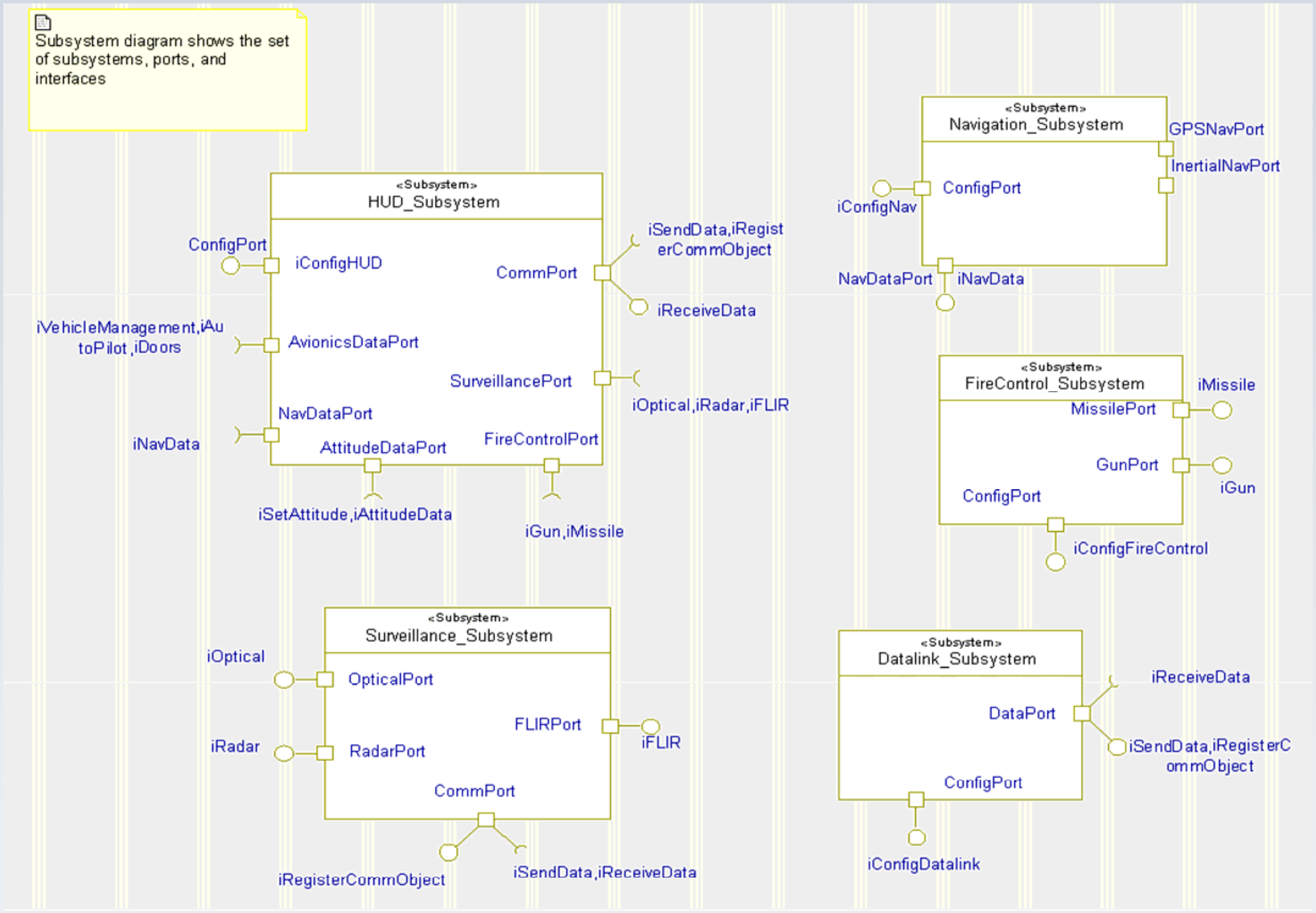
Harmony/SE: System Functional Analysis


Plan Work
Refine and deploy the development environment
Build Executable Model of a Use Case - Alternative 2
[next use case]
[done]
Check Use Case Consistency
Detail System Requirements
Update Verification Plan
Manage Work
Manage Change
Create Functional Baseline
Review System Requirements
Assess Results

Harmony/SE: Design Synthesis

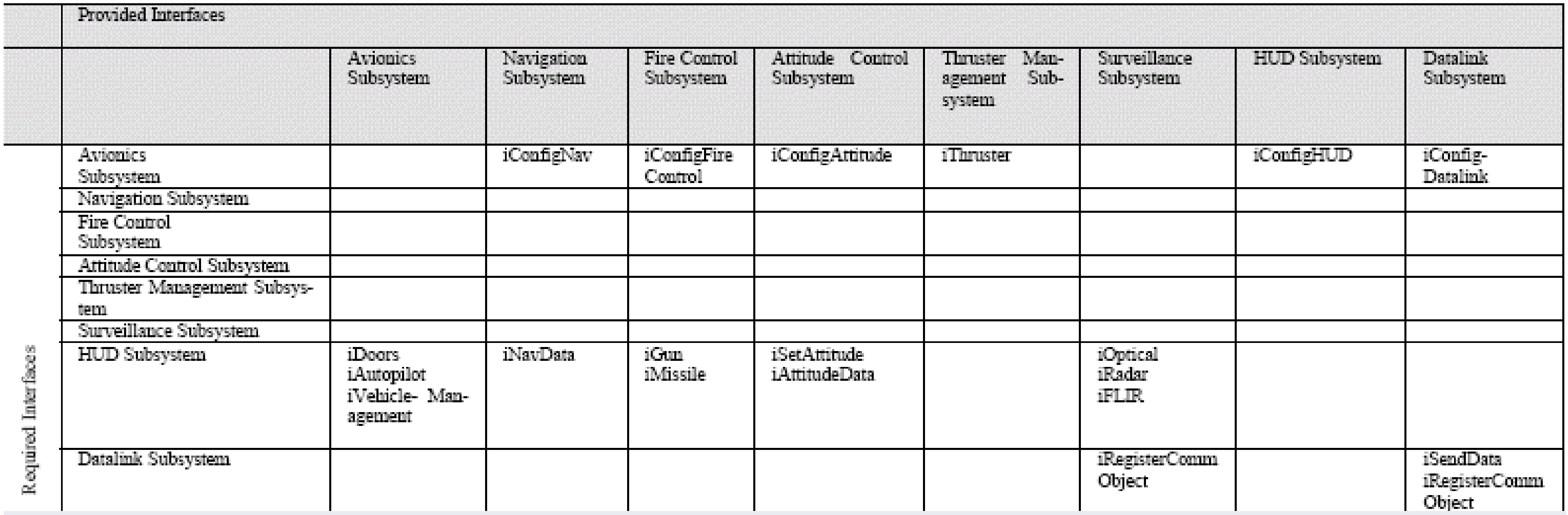

Plan Work
Refine and deploy the development environment
Update Verification Plan
Architectural Analysis (Trade Study)
Architectural Design
Manage Change
Manage Work
[further optimization needed]
Detailed Architectural Design
Derive Subsystem Requirements
Create Allocated Baseline
Review Architecture
Assess Results

Subsystem ports and Interfaces


Subsystem diagram shows the set of subsystems, ports, and interfaces

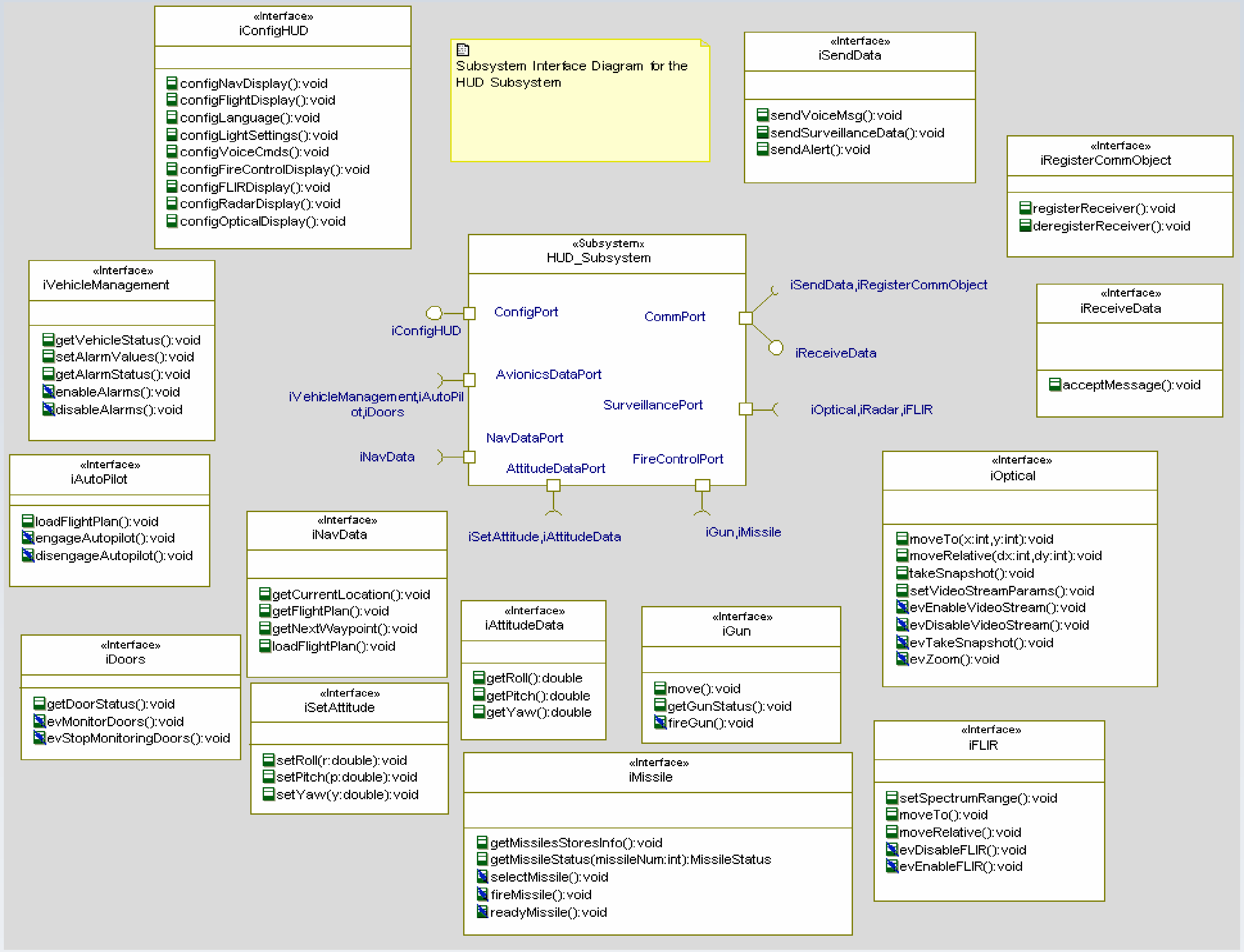
Information as a Matrix – Subsystem N^2 Diagram

- Can also be shown in **spreadsheet format**, referred to as an N^2 diagram. This shows the **provided and required interfaces** between the various subsystems. If desired, it also shows the details of the specific services within the interfaces as well.



	Provided Interfaces								
		Avionics Subsystem	Navigation Subsystem	Fire Control Subsystem	Attitude Control Subsystem	Thruster Management Subsystem	Surveillance Subsystem	HUD Subsystem	Datalink Subsystem
Required Interfaces	Avionics Subsystem		iConfigNav	iConfigFire Control	iConfigAttitude	iThruster		iConfigHUD	iConfig-Datalink
	Navigation Subsystem								
	Fire Control Subsystem								
	Attitude Control Subsystem								
	Thruster Management Subsystem								
	Surveillance Subsystem								
	HUD Subsystem	iDoors iAutopilot iVehicle-Management	iNavData	iGun iMissile	iSetAttitude iAttitudeData		iOptical iRadar iFLIR		
	Datalink Subsystem						iRegisterComm Object		iSendData iRegisterComm Object

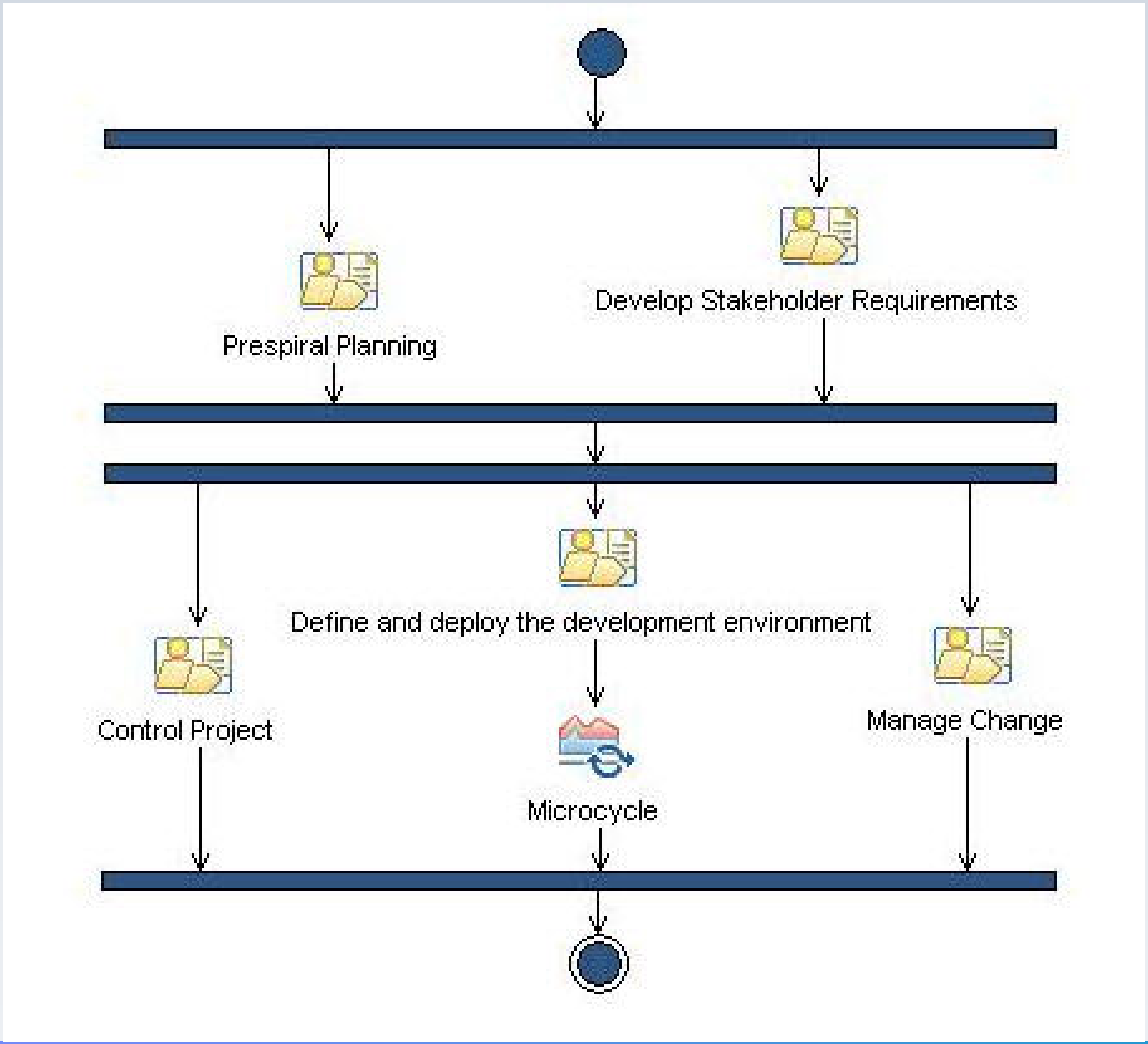
Example: HUD subsystem Interface Diagram



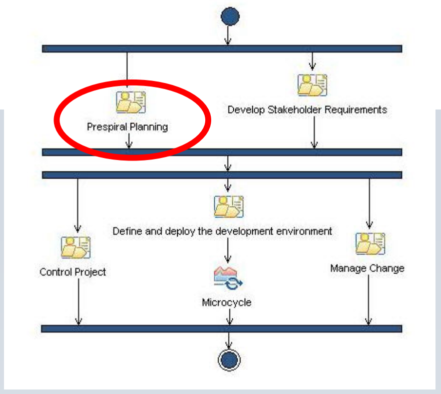
Harmony/ESW Delivery Process

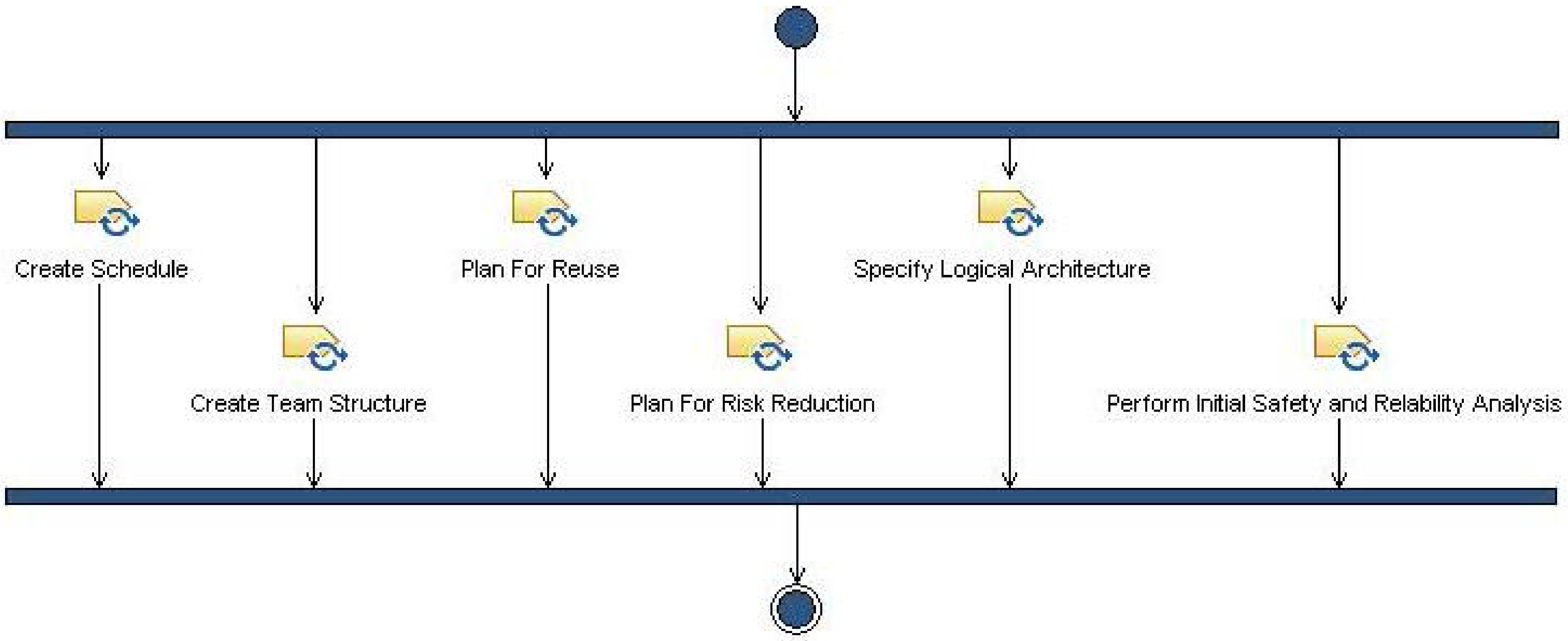
- Can be used as a part of the Harmony/Hybrid V process or as a stand-alone software-centric development process
- As a part of the Hybrid-V process
  - Systems engineering does much but not all of the requirements work
  - Used when there is significant systems engineering and/or hw/sw co-development
- Harmony/ESW stand-alone delivery process
  - As a stand-alone process, optimizations can be taken for projects which
    - Are software-only
    - Have requirements which are linearly-separable (i.e. can be put into approximately-independent use cases)
    - No significant hw/sw co-design
  - In initial planning, use cases are identified but not detailed
  - In each microcycle, one or more use cases is added to the prototype
    - Requirements are detailed for that set *at this time*
    - Deployment proceeds via incremental software design

Harmony/ESW Full Spiral


Prespiral Planning
Develop Stakeholder Requirements
Define and deploy the development environment
Control Project
Manage Change
Microcycle

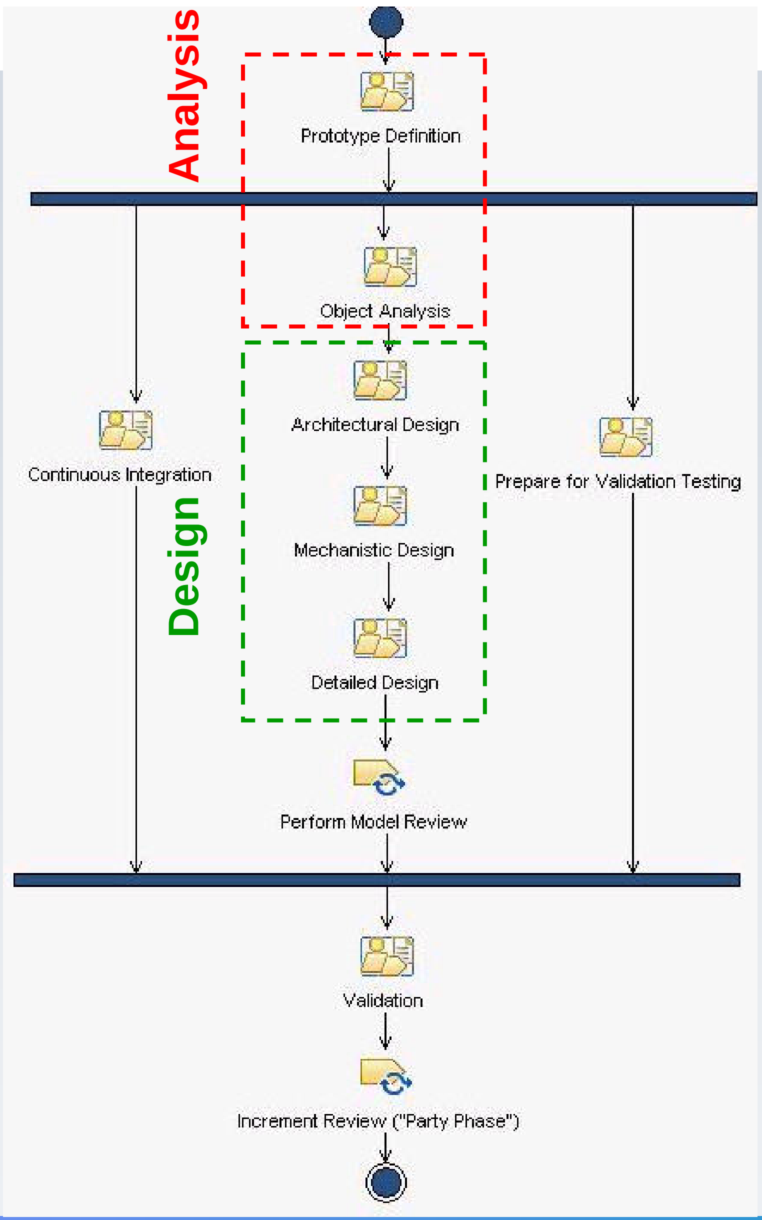
Prespiral Planning

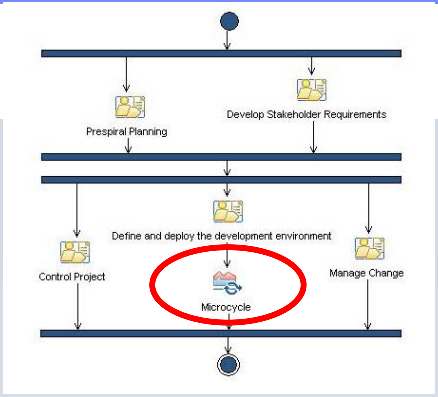

Prespiral Planning
Develop Stakeholder Requirements
Define and deploy the development environment
Control Project
Manage Change
Microcycle


Create Schedule
Create Team Structure
Plan For Reuse
Plan For Risk Reduction
Specify Logical Architecture
Perform Initial Safety and Relability Analysis

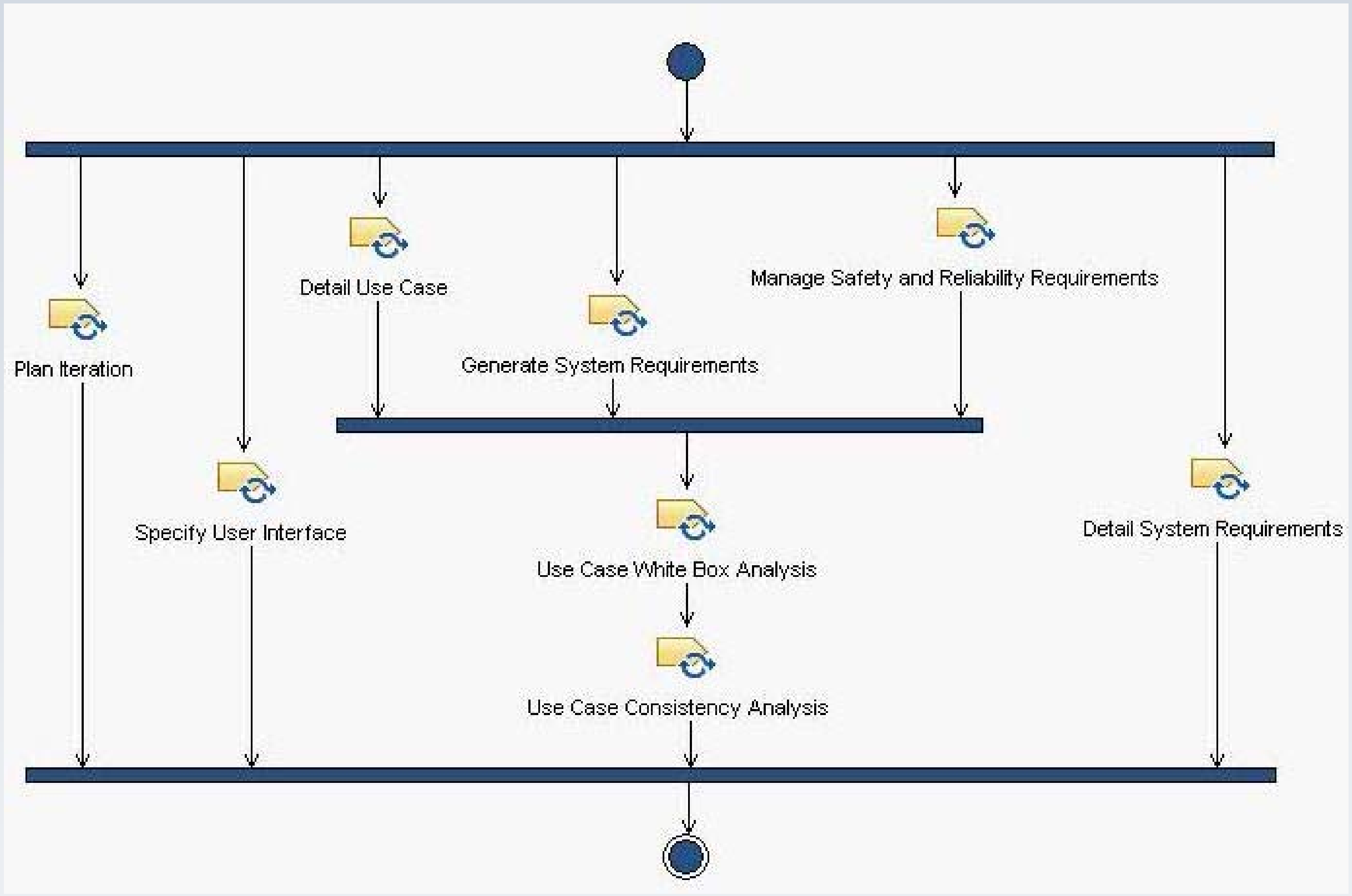
Microcycle Flow

- Analysis focuses on identifying the *essential required properties of the system,* resulting in
  - CIM (use case model)
  - PIM (object analysis model)
- Design optimizes the PIM at 3 levels of abstraction to produce the PSM
- Code is continuously generated and refined throughout analysis and design, producing the PSI
- The Party Phase provides a periodic project retrospective for process and project improvement and refinement

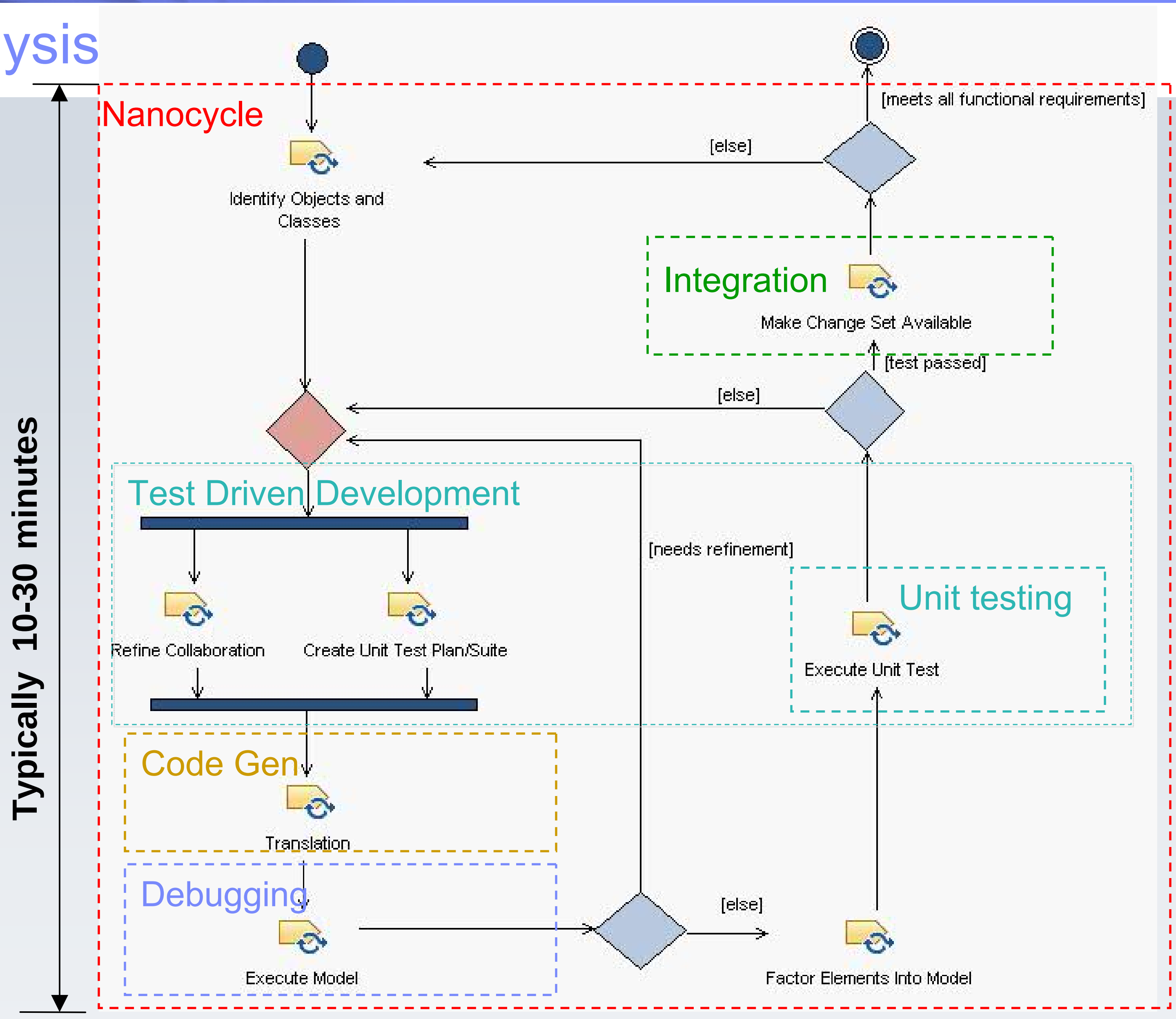



Prespiral Planning
Develop Stakeholder Requirements
Define and deploy the development environment
Control Project
Microcycle
Manage Change

Prototype Definition


Plan Iteration
Specify User Interface
Detail Use Case
Generate System Requirements
Manage Safety and Reliability Requirements
Use Case White Box Analysis
Use Case Consistency Analysis
Detail System Requirements

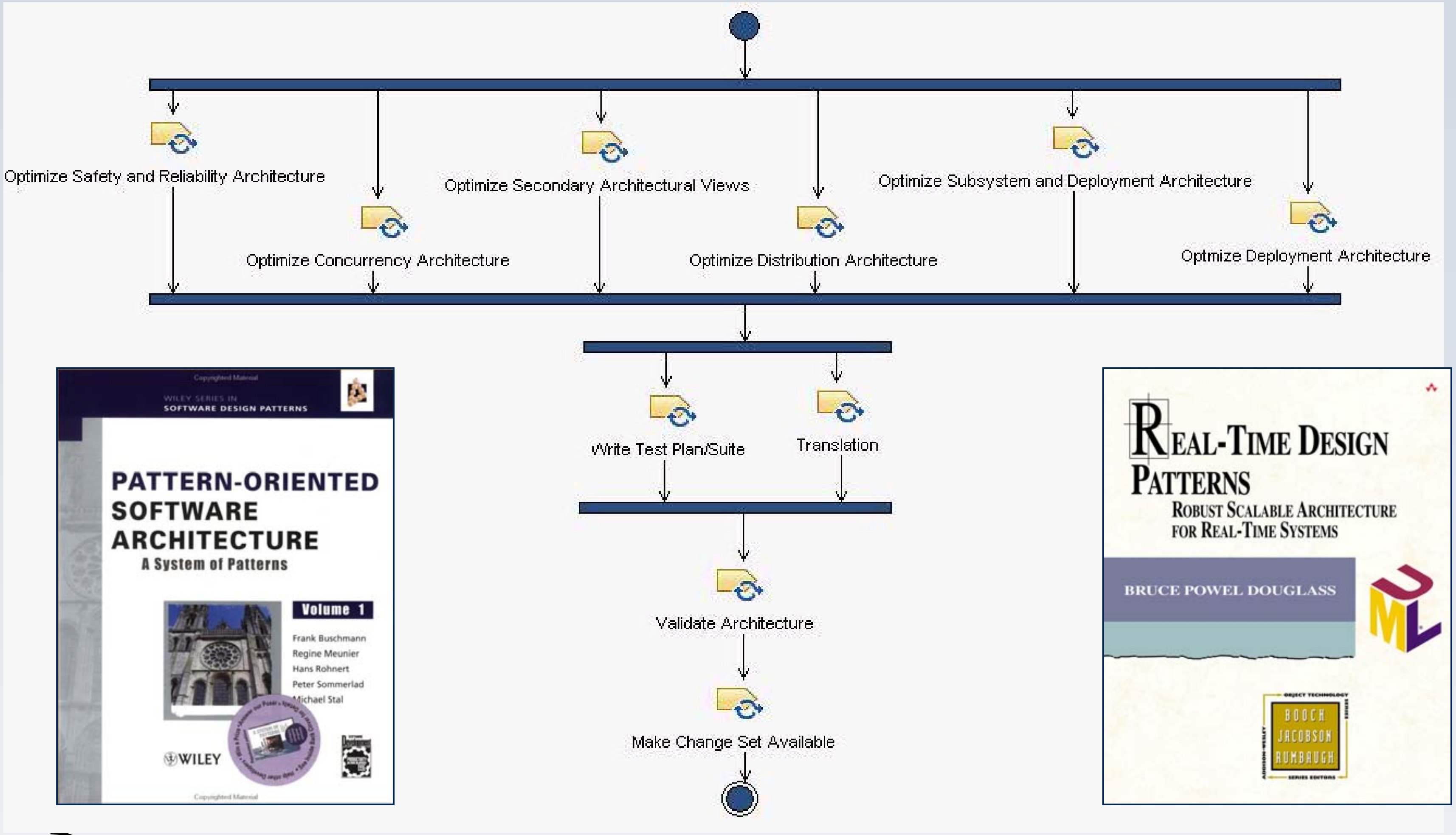
Object Analysis

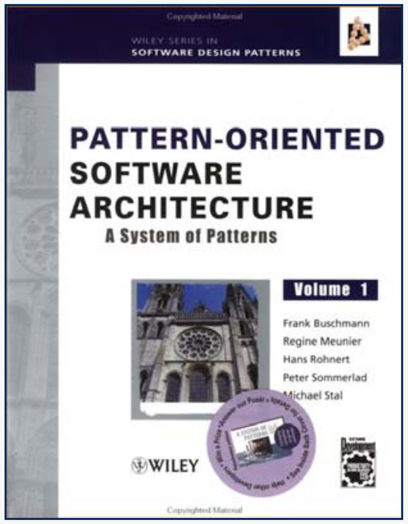


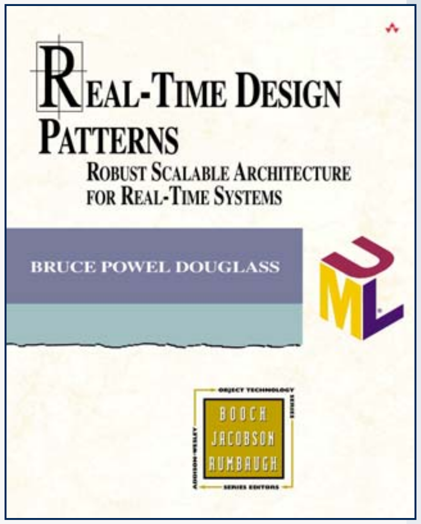
Harmony/ESW Design

- Design takes place at 3 levels of abstraction
  - Architectural (system scope)
  - Mechanistic (use case collaboration scope)
  - Detailed (class/function/data structure scope)
- All design activities in the Harmony process are *optimization-related activities* with the same basic workflow:
  - Identify the design optimization criteria
  - Rank the criteria in order of criticality
  - Identify design solutions that optimize the most important of these criteria at the expense of the least
    - Design patterns
    - Design technologies
  - Apply the design solutions into your model
  - Test
    - Ensure that you haven't broken the existing functionality
    - Ensure you have achieved your optimization goals

Architectural Design Workflow

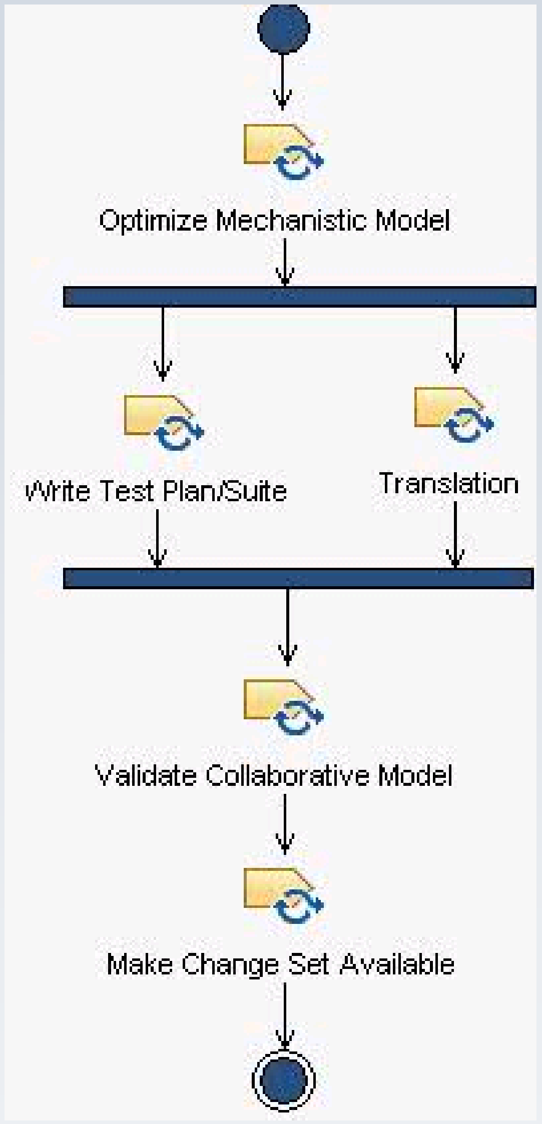

Optimize Safety and Reliability Architecture
Optimize Concurrency Architecture
Optimize Secondary Architectural Views
Optimize Distribution Architecture
Optimize Subsystem and Deployment Architecture
Optmize Deployment Architecture
Write Test Plan/Suite
Translation
Validate Architecture
Make Change Set Available

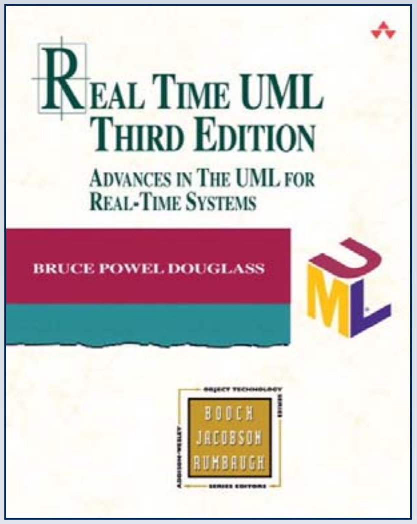


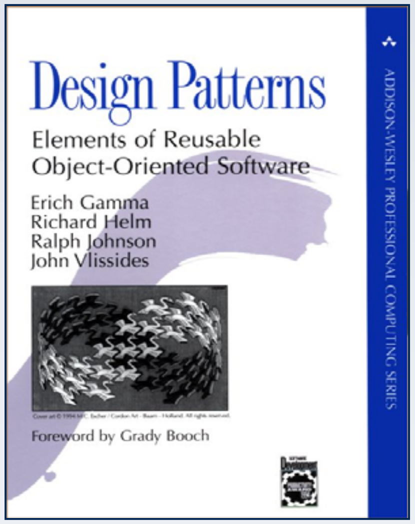

REAL-TIME DESIGN PATTERNS
ROBUST SCALABLE ARCHITECTURE FOR REAL-TIME SYSTEMS
BRUCE POWEL DOUGLASS
BOOCH JACOBSON RUMBAUGH

Not all views may be relevant within the mission of a given prototype, but all will be addressed before the project is completed.

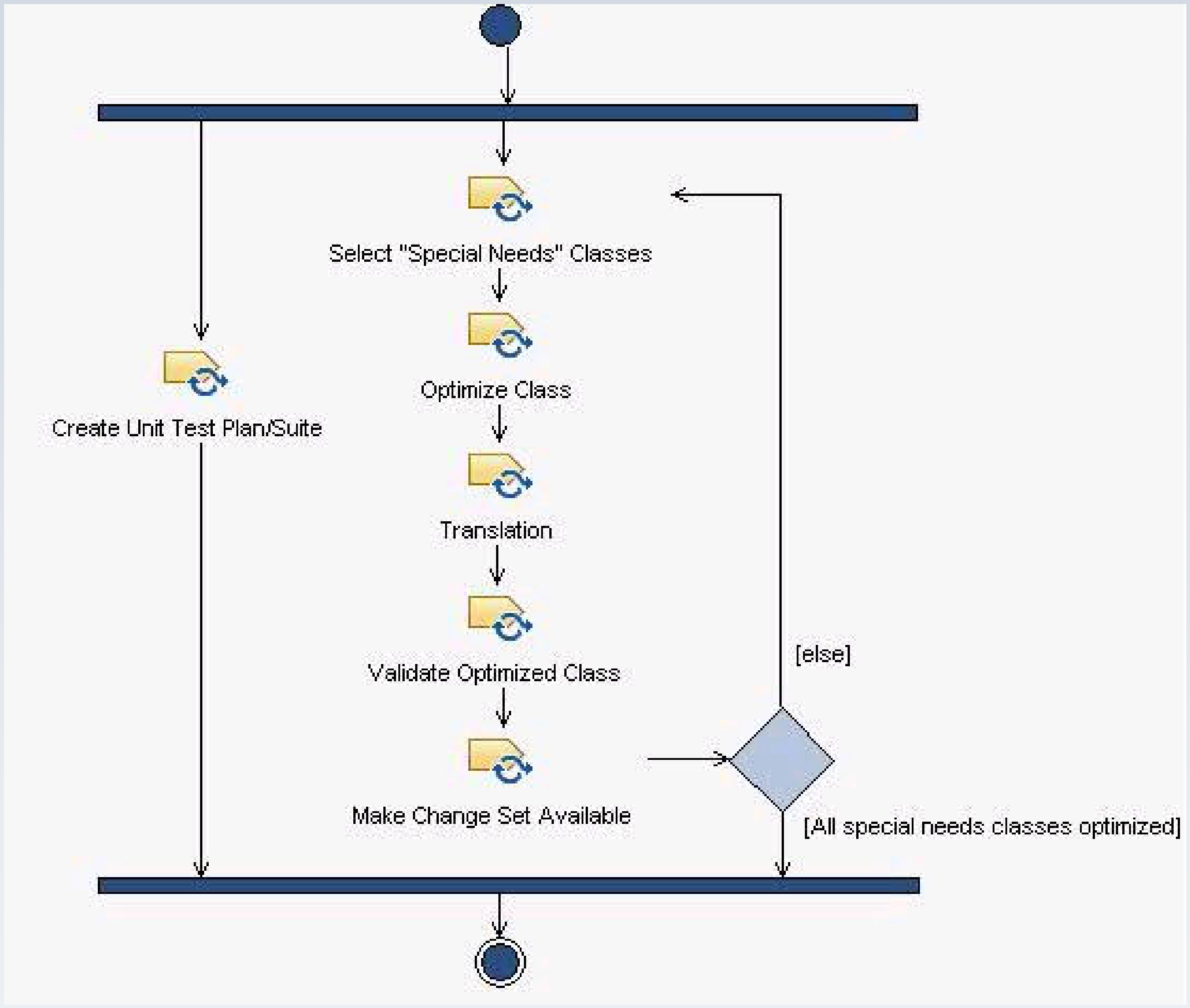
Mechanistic Design Workflow

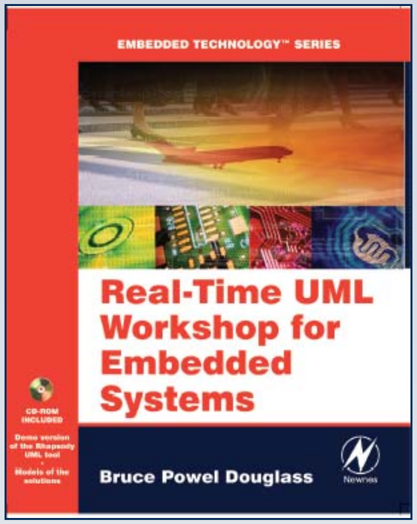

Optimize Mechanistic Model
Write Test Plan/Suite
Translation
Validate Collaborative Model
Make Change Set Available

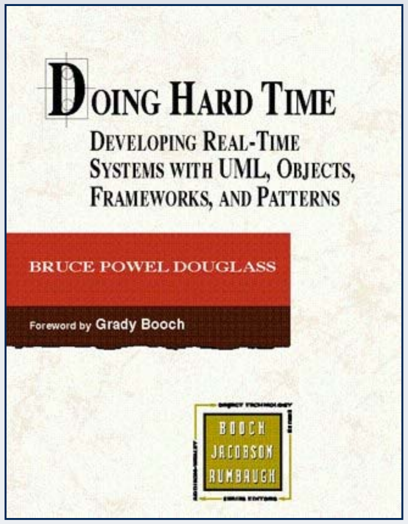



Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
Addison-Wesley Professional Computing Series

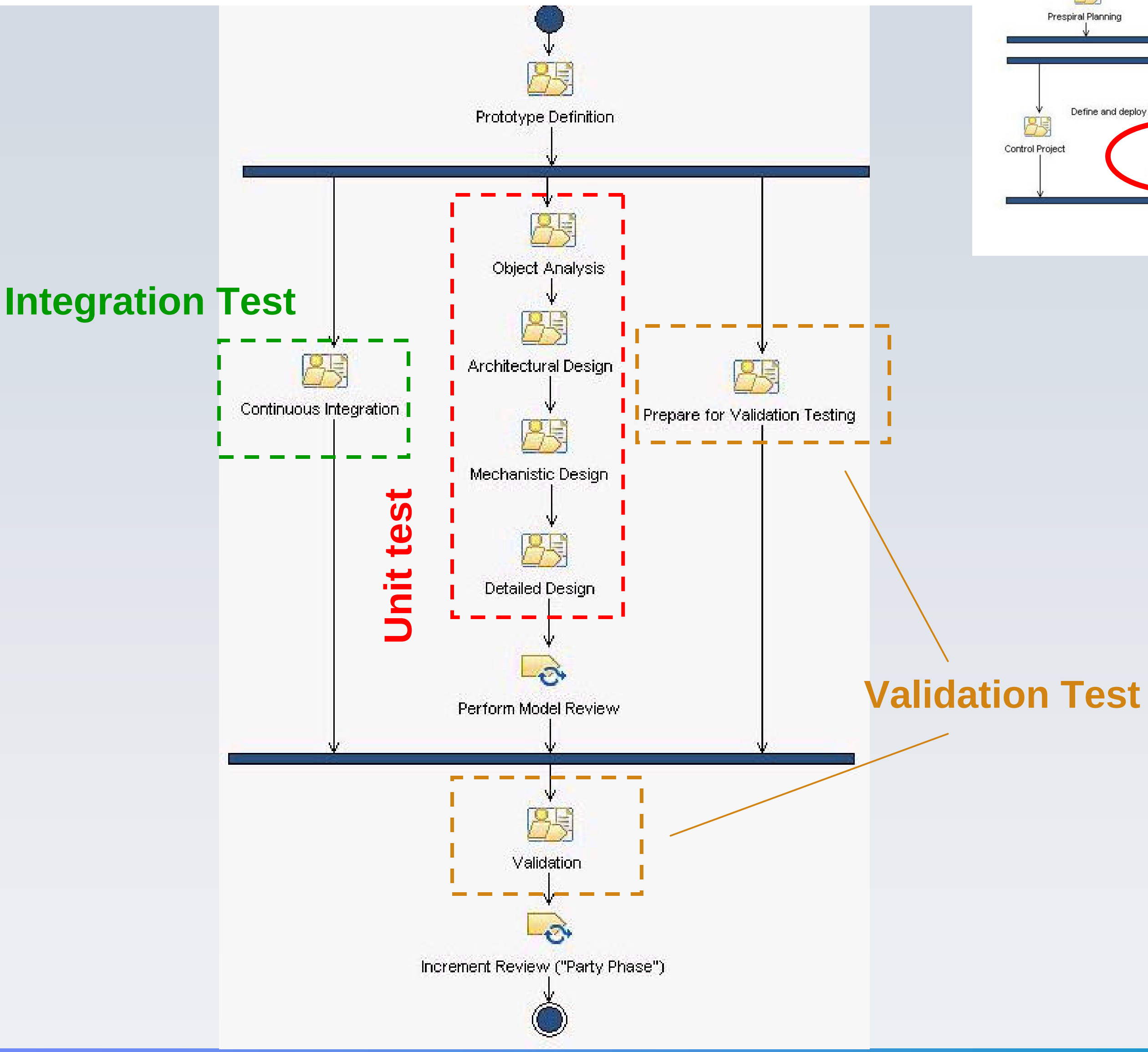
Detailed Design

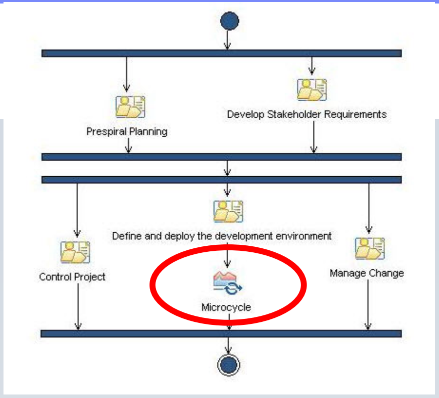

Select "Special Needs" Classes
Optimize Class
Translation
Validate Optimized Class
Make Change Set Available
Create Unit Test Plan/Suite
[else]
[All special needs classes optimized]


EMBEDDED TECHNOLOGY™ SERIES
Real-Time UML Workshop for Embedded Systems
Bruce Powel Douglass


DOING HARD TIME
DEVELOPING REAL-TIME SYSTEMS WITH UML, OBJECTS, FRAMEWORKS, AND PATTERNS
BRUCE POWEL DOUGLASS
Foreword by Grady Booch

Harmony/ESW Activities related to Testing

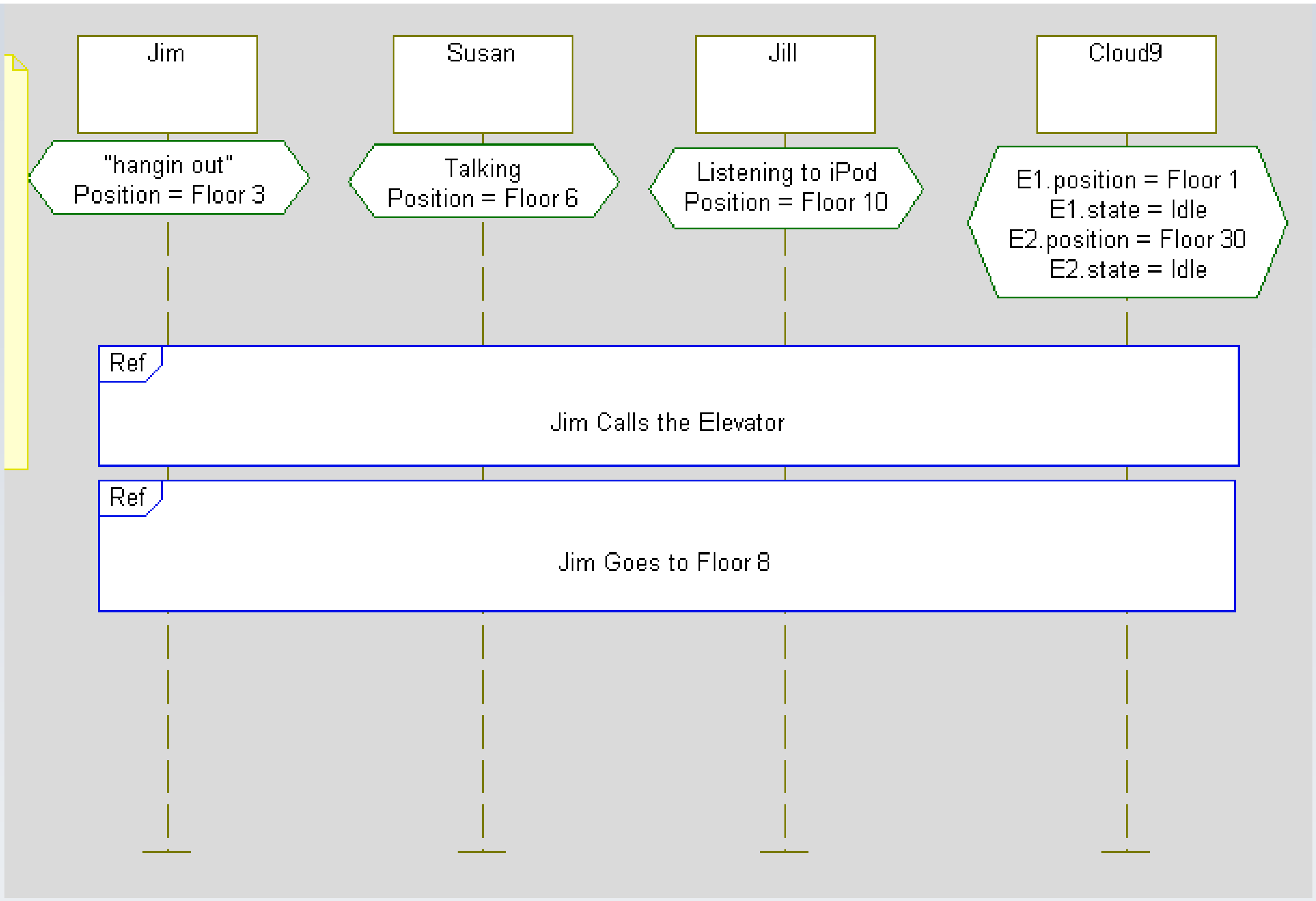



Prespiral Planning
Develop Stakeholder Requirements
Define and deploy the development environment
Control Project
Manage Change
Microcycle

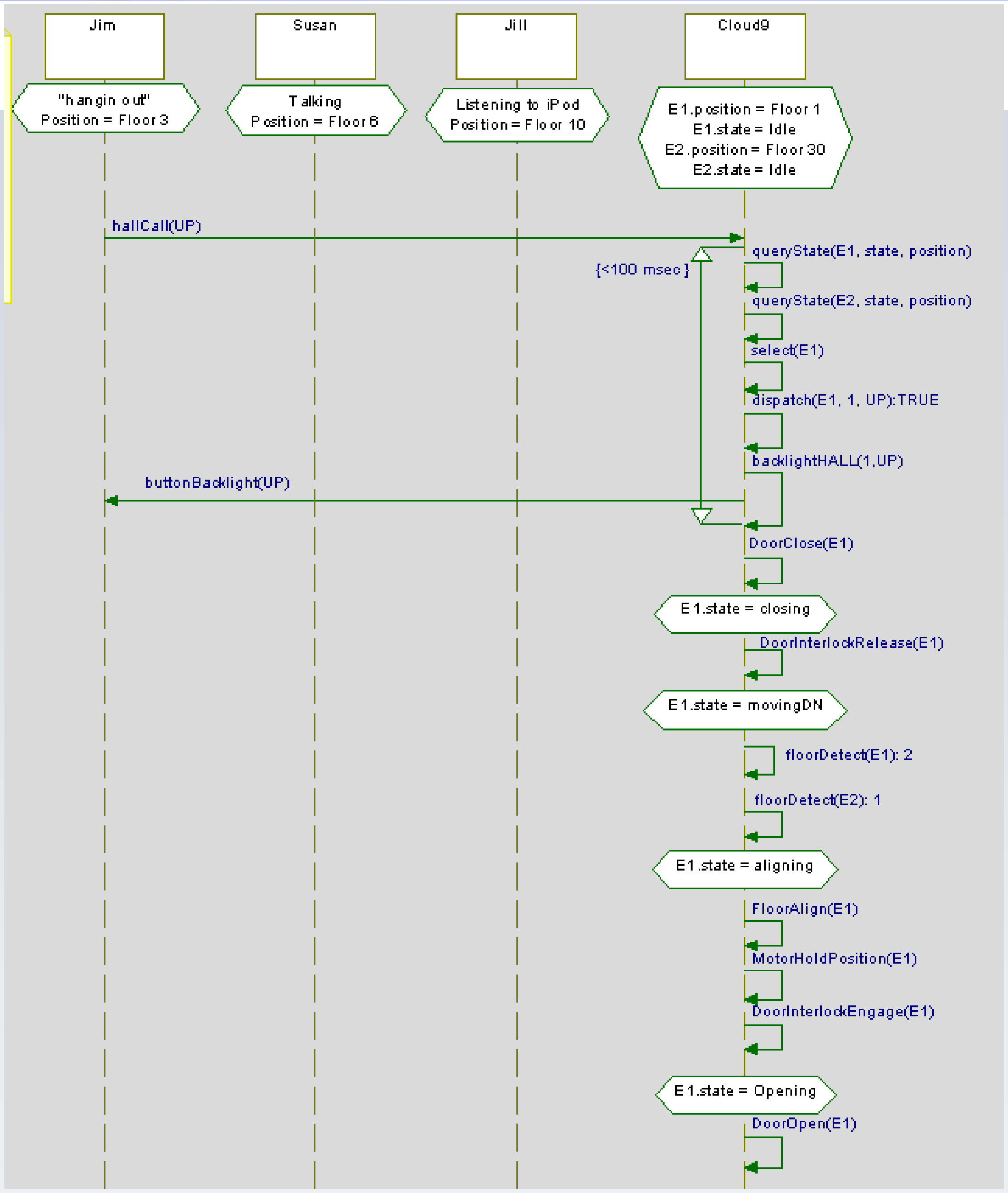
A Look at the Harmony/ESW Nanocycle

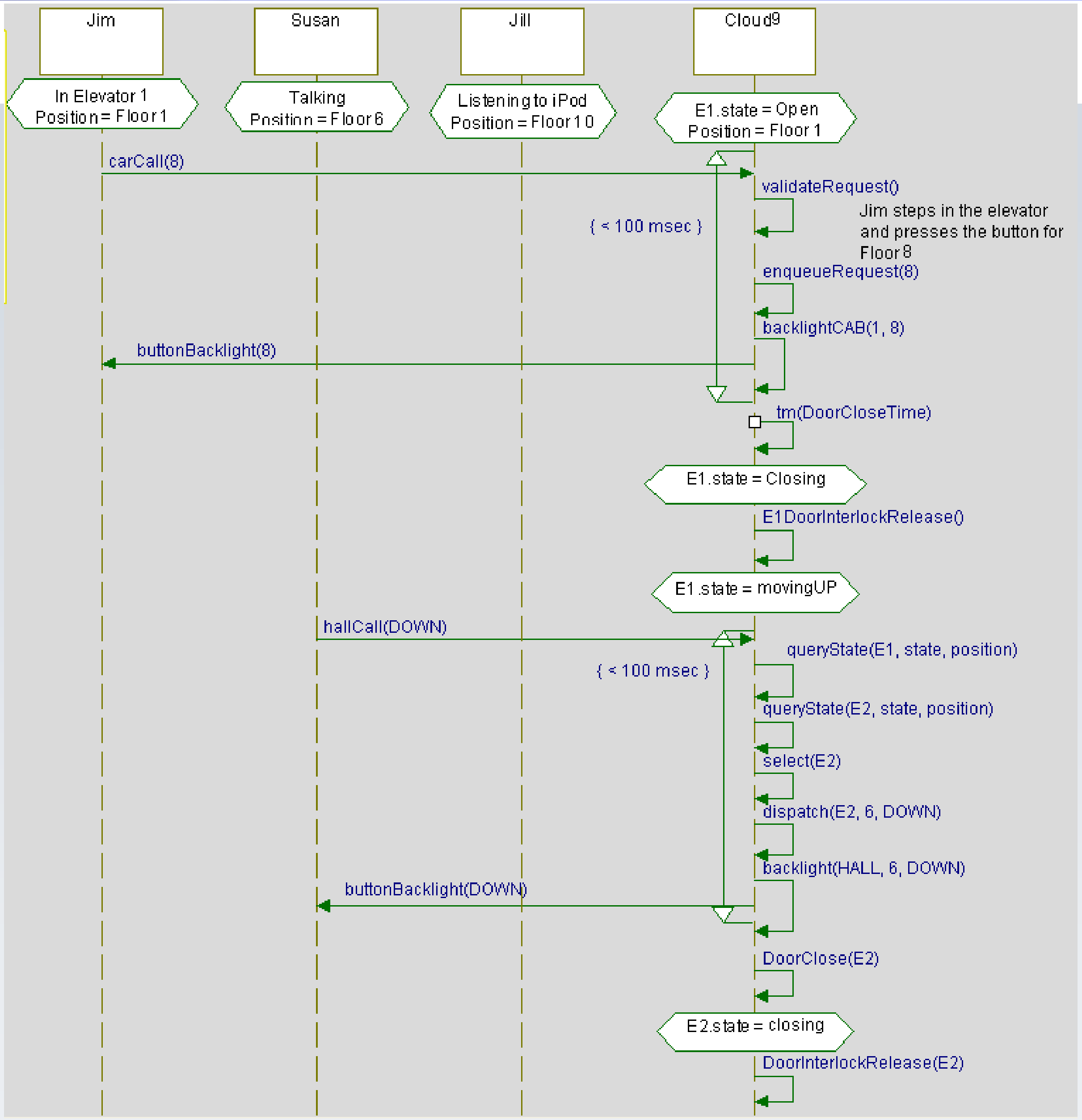
- Throughout analysis and design, developers produces partial models that are executed and debugged on a highly frequent basis - typically 10 - 30 minutes
- Based on the concept of *continual verification*
  - After each small incremental change, the portion of the model is reexecuted to make sure that it is right.
  - Best when debugging & unit testing can be at the modeling, rather than the code, level of abstraction
- Primary Activities
  - **Code Generation**: Source code is generated from the model frequently (<1 hr)
  - **Debugging**: Often informal tests are constructed and applied either with tools (e.g. Test Conductor) or by building “test buddy classes” to drive and check execution
  - **Unit testing**: Unit tests are created along with the software and unit tests are applied within the nanocycle
  - **Make changes available**: Tested changes are submitted to the CM manager for integration/test and update to baseline at least daily

Nanocycle Example: Elevator Scenario (1 of 3)

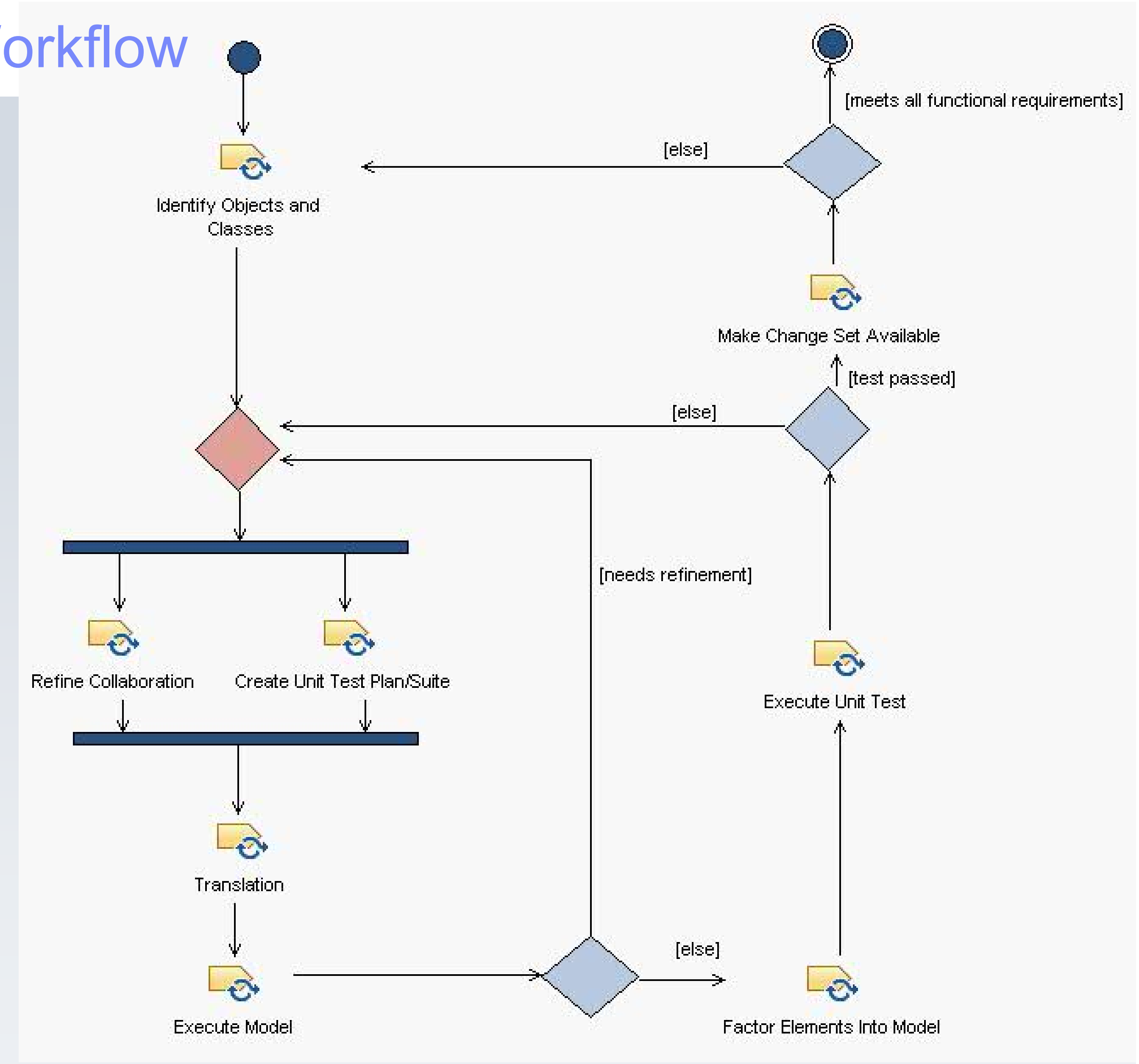


Remember: this is just *one* of possibly dozens of scenarios for this one use case. Features will be added as they are needed to fulfill the scenarios.


Jim
Susan
Jill
Cloud9
"hangin out" Position = Floor 3
Talking Position = Floor 6
Listening to iPod Position = Floor 10
E1.position = Floor 1
E1.state = Idle
E2.position = Floor 30
E2.state = Idle
hallCall(UP)
queryState(E1, state, position)
{<100 msec}
queryState(E2, state, position)
select(E1)
dispatch(E1, 1, UP):TRUE
backlightHALL(1,UP)
buttonBacklight(UP)
DoorClose(E1)
E1.state = closing
DoorInterlockRelease(E1)
E1.state = movingDN
floorDetect(E1): 2
floorDetect(E2): 1
E1.state = aligning
FloorAlign(E1)
MotorHoldPosition(E1)
DoorInterlockEngage(E1)
E1.state = Opening
DoorOpen(E1)

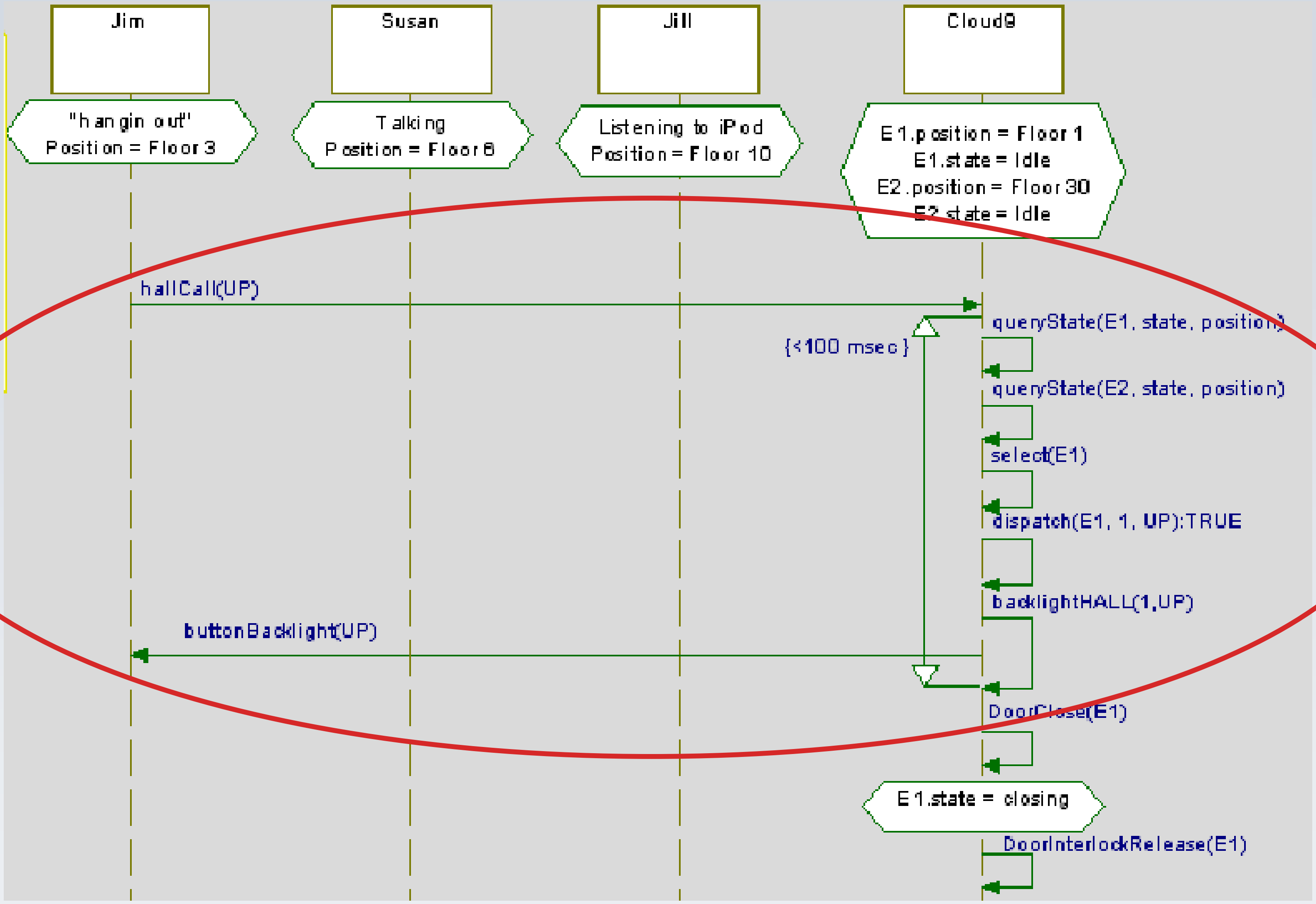

Jim
Susan
Jill
Cloud9
In Elevator 1
Position = Floor 1
Talking
Position = Floor 6
Listening to iPod
Position = Floor 10
E1.state = Open
Position = Floor 1
carCall(8)
validateRequest()
{ < 100 msec }
Jim steps in the elevator and presses the button for Floor 8
enqueueRequest(8)
backlightCAB(1, 8)
buttonBacklight(8)
tm(DoorCloseTime)
E1.state = Closing
E1DoorInterlockRelease()
E1.state = movingUP
hallCall(DOWN)
queryState(E1, state, position)
{ < 100 msec }
queryState(E2, state, position)
select(E2)
dispatch(E2, 6, DOWN)
backlight(HALL, 6, DOWN)
buttonBacklight(DOWN)
DoorClose(E2)
E2.state = closing
DoorInterlockRelease(E2)

Object Analysis Workflow



Identify Objects and Classes
[else]
[meets all functional requirements]
Make Change Set Available
[test passed]
[else]
[needs refinement]
Refine Collaboration
Create Unit Test Plan/Suite
Execute Unit Test
Translation
Execute Model
[else]
Factor Elements Into Model

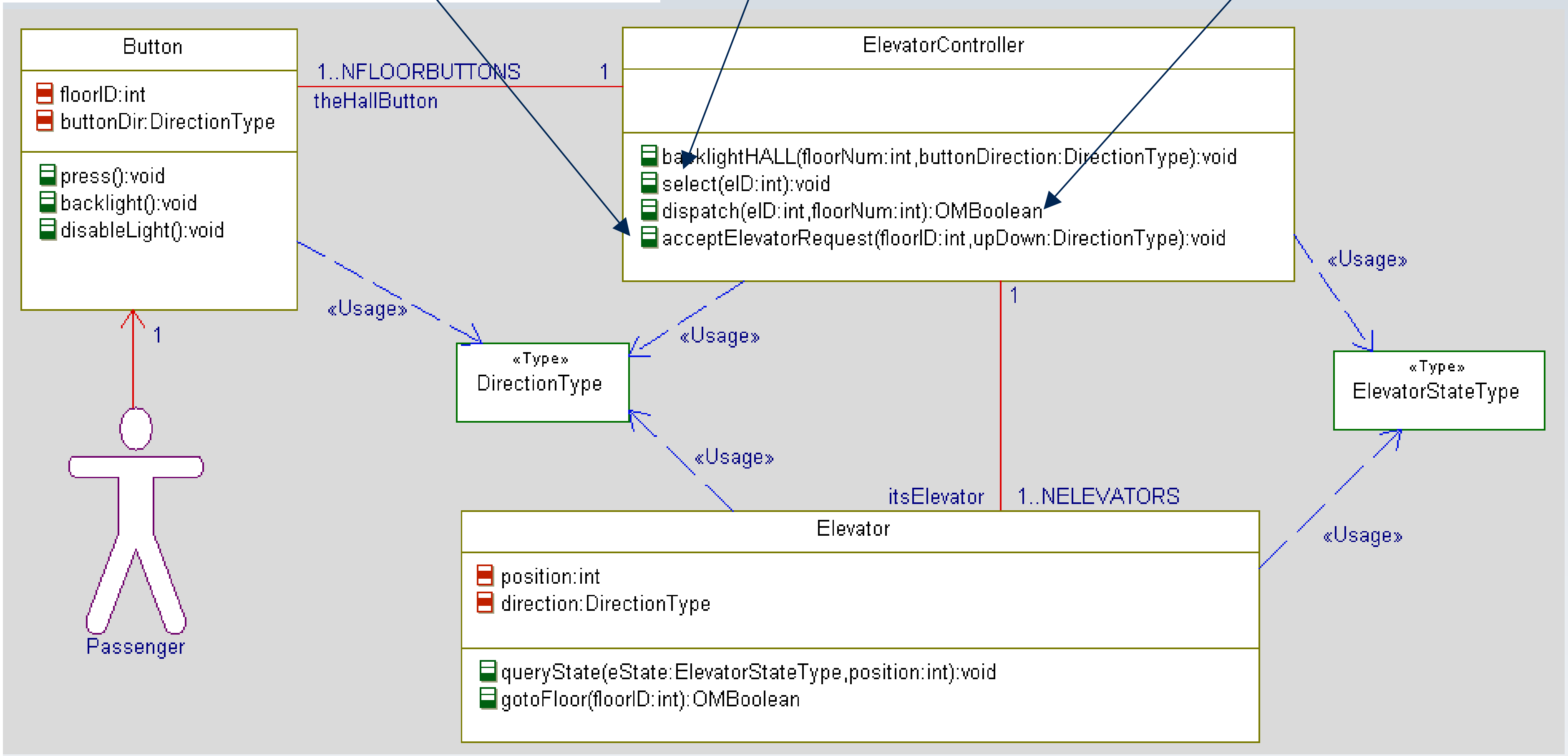
Step 1: Requesting the elevator



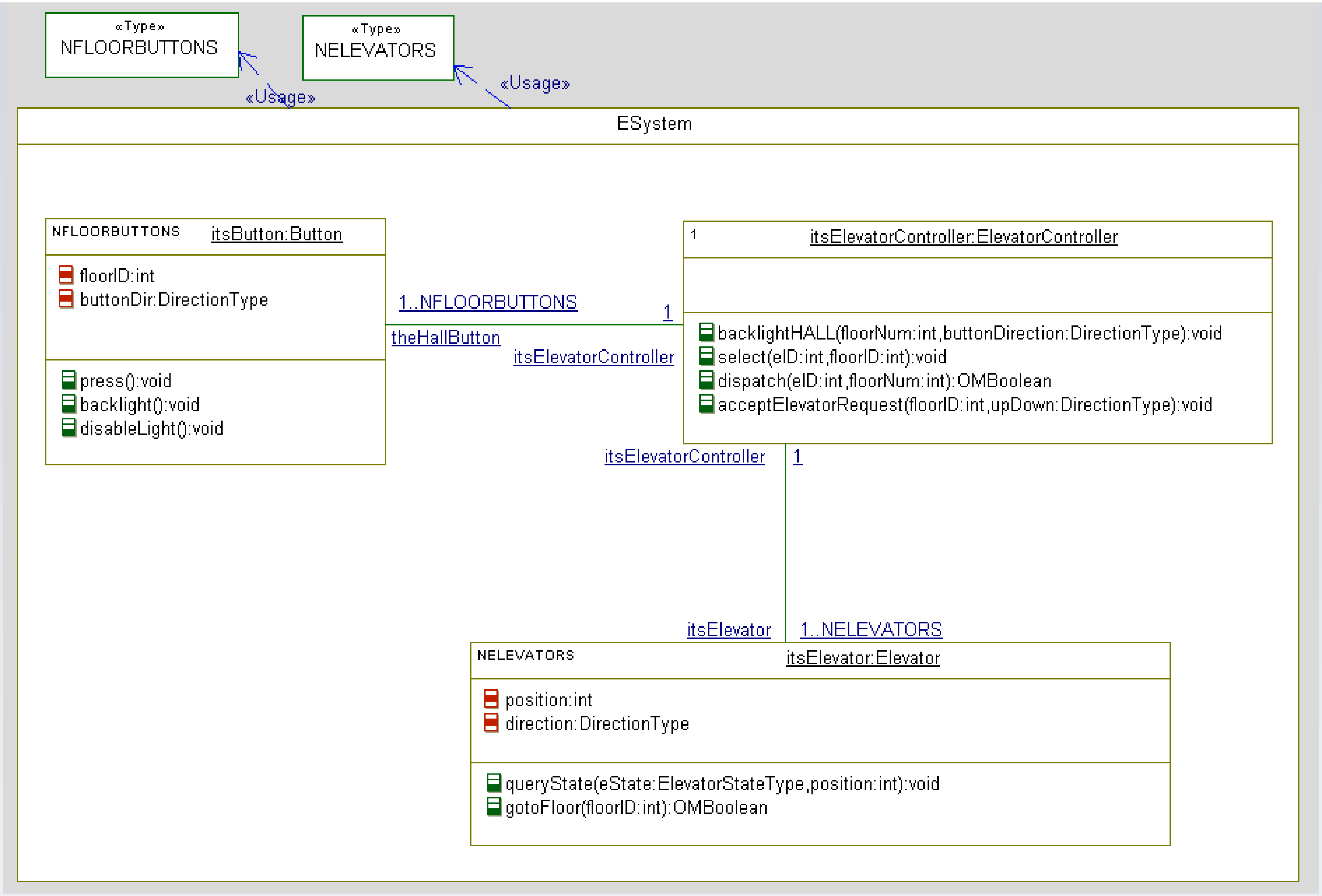
```
ElevatorStateType elevState;
int j;
int pos;

for (j=0;j<NELEVATORS;j++) {
    itsElevator[j]->queryState(elevState, pos);
    if (elevState = Idle) {
        select(j, floorID);
    }; // end if
}; //end for
```

```
OMBoolean success;
success = dispatch(eID, floorID);
if (success) {
    theHallButton[floorID]->backlight();
};
```

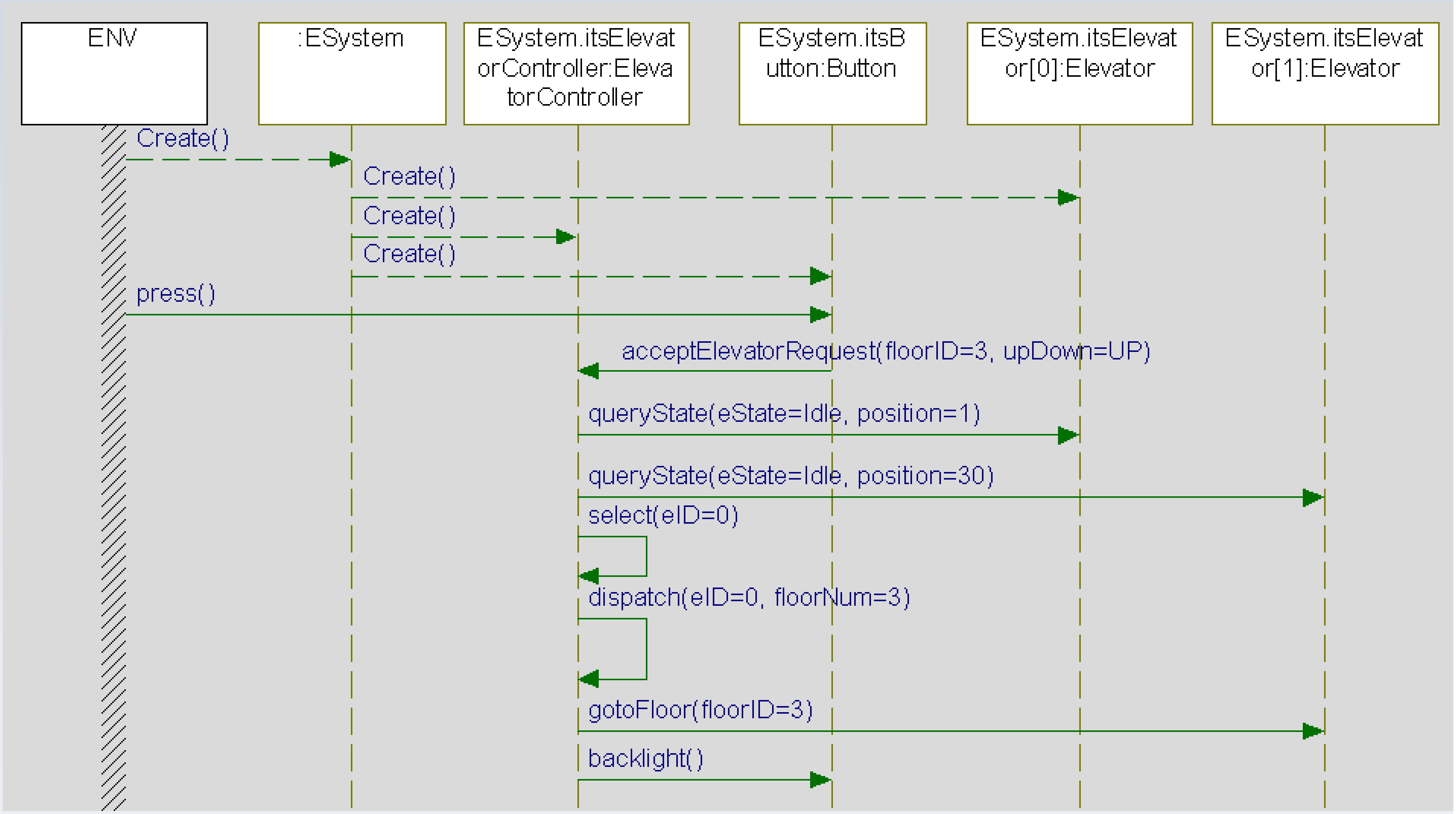
```
itsElevator[eID]->gotoFloor(floorNum);
return TRUE;
```



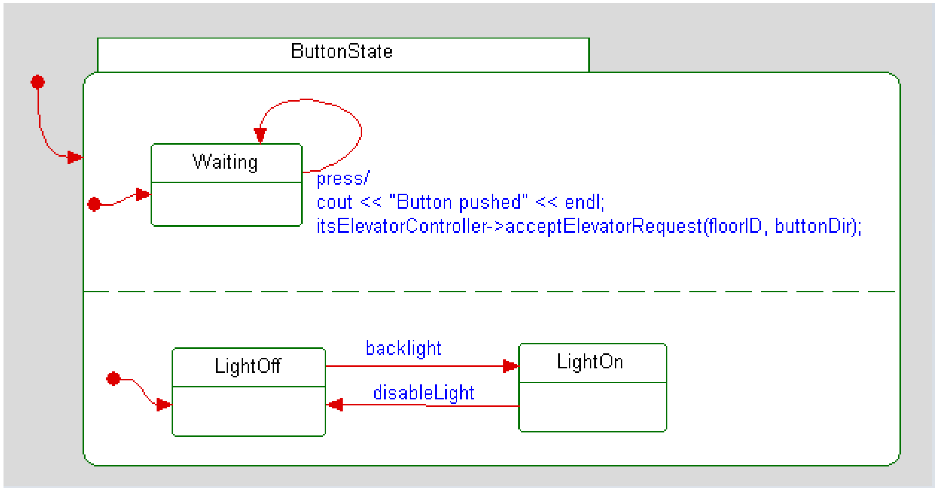
Executing Step 1



Executing Step 1

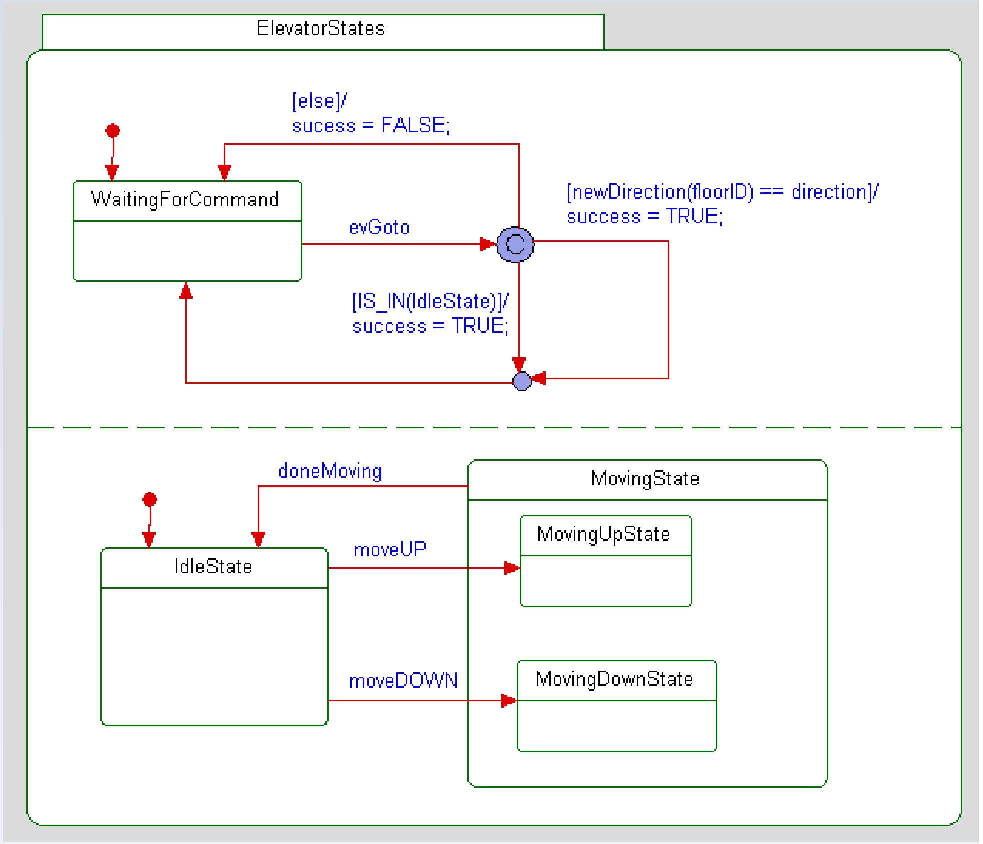

ENV
:ESystem
ESystem.itsElevatorController:ElevatorController
ESystem.itsButton:Button
ESystem.itsElevator[0]:Elevator
ESystem.itsElevator[1]:Elevator
Create()
Create()
Create()
Create()
press()
acceptElevatorRequest(floorID=3, upDown=UP)
queryState(eState=Idle, position=1)
queryState(eState=Idle, position=30)
select(eID=0)
dispatch(eID=0, floorNum=3)
gotoFloor(floorID=3)
backlight()

Step 2: Make classes reactive

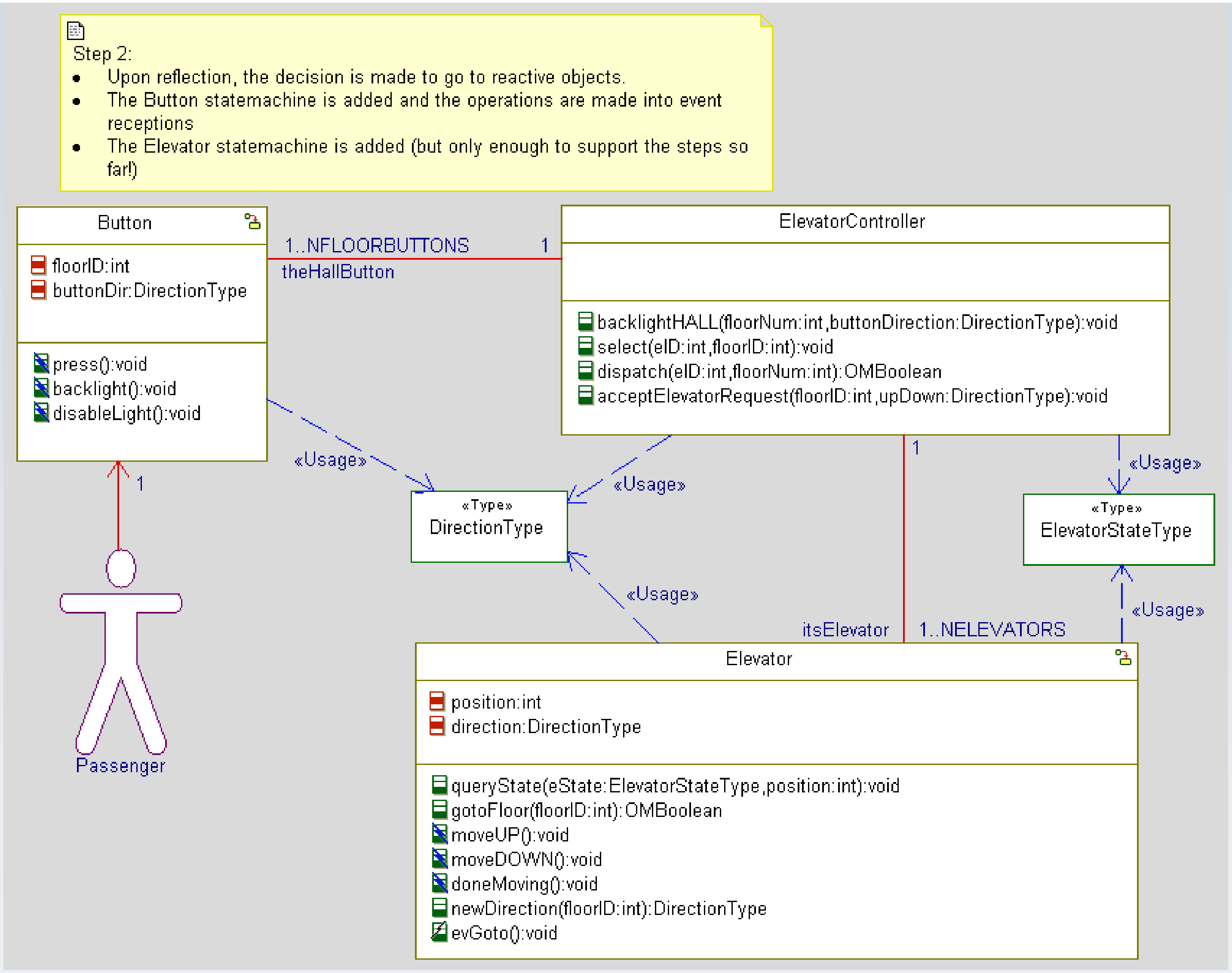

ButtonState
Waiting
press/
cout << "Button pushed" << endl;
itsElevatorController->acceptElevatorRequest(floorID, buttonDir);
LightOff
backlight
disableLight
LightOn

Button state machine

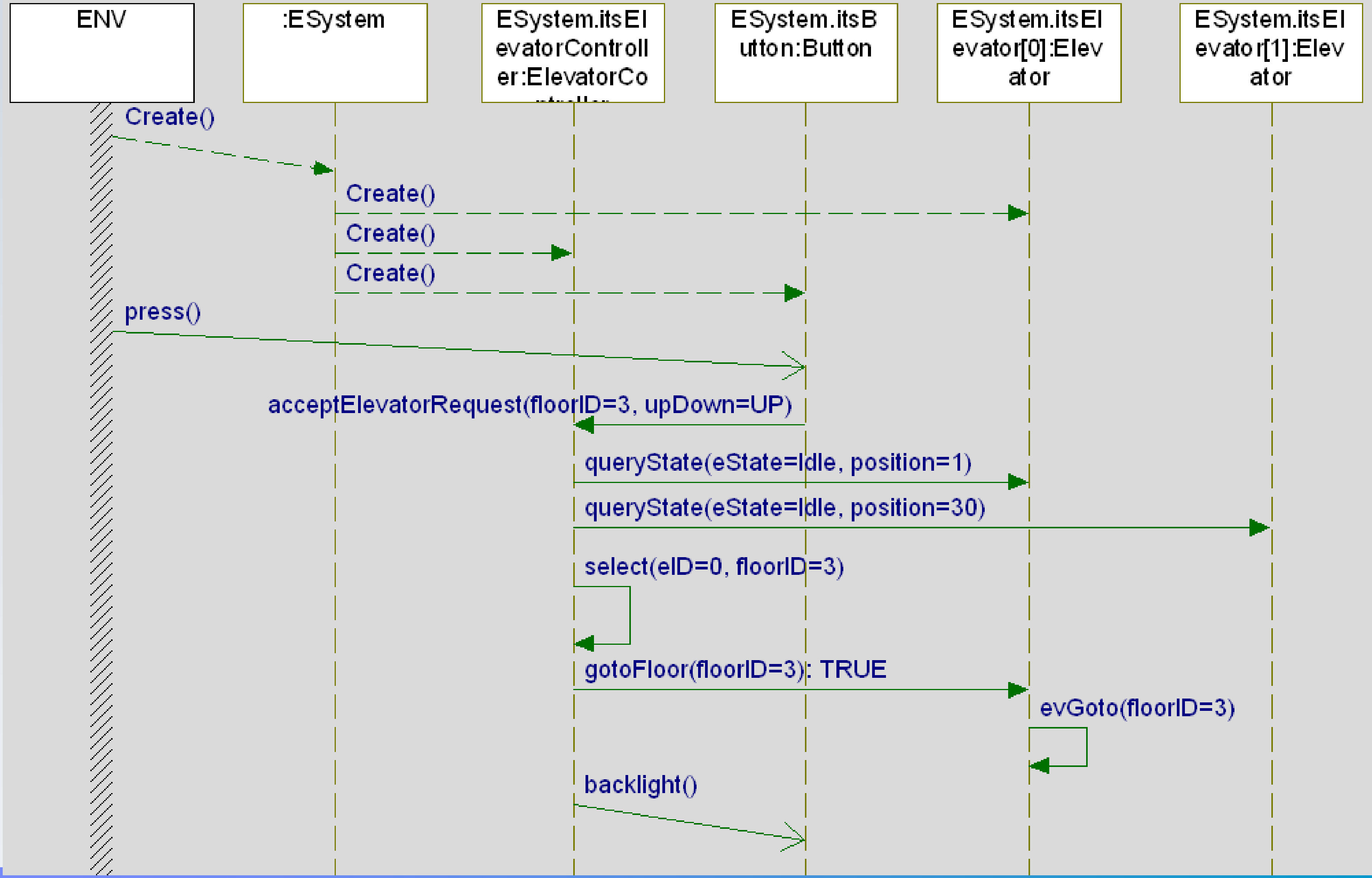
Elevator state machine



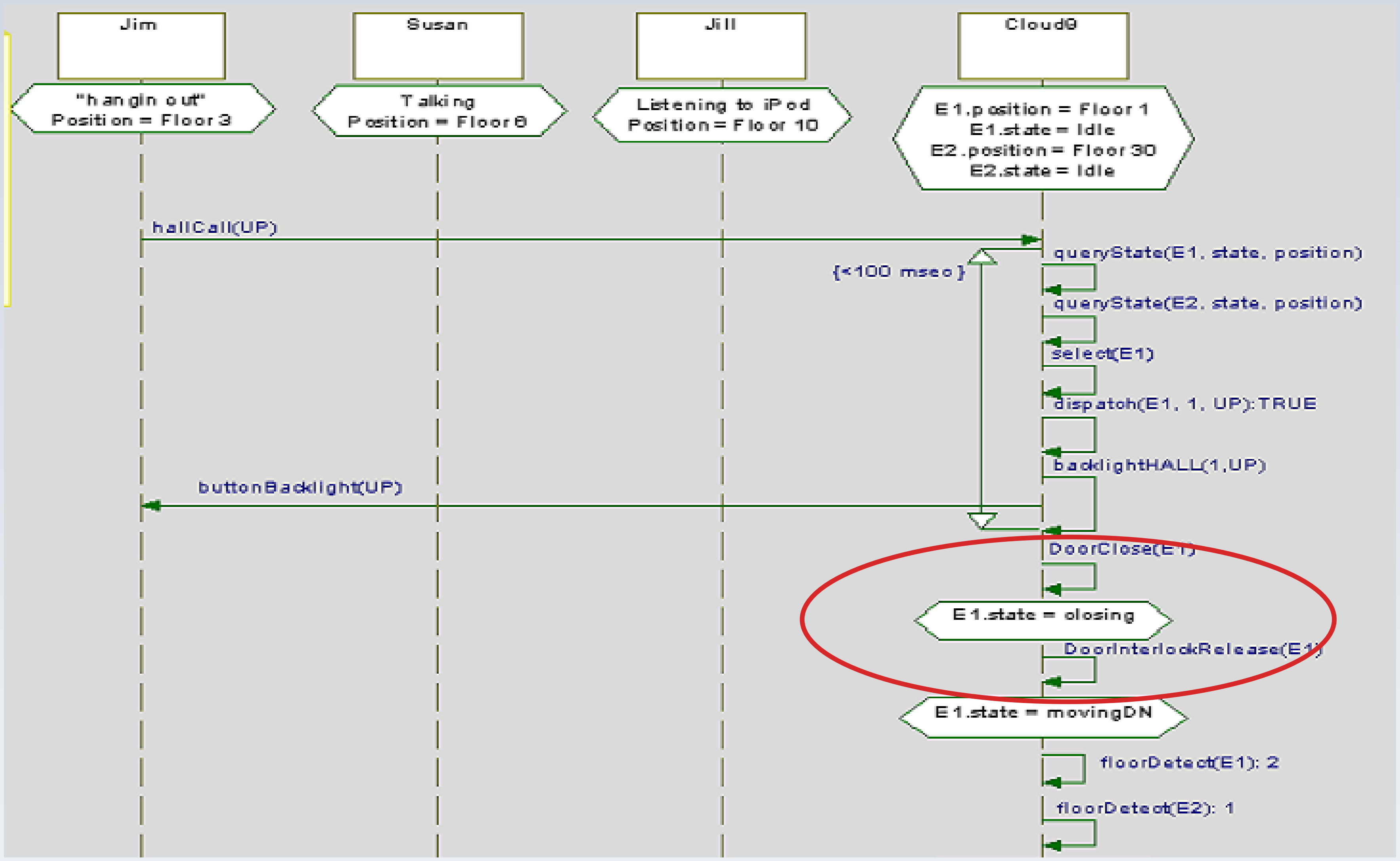
Step 2: Class Diagram


Step 2:
- Upon reflection, the decision is made to go to reactive objects.
- The Button statemachine is added and the operations are made into event receptions
- The Elevator statemachine is added (but only enough to support the steps so far!)

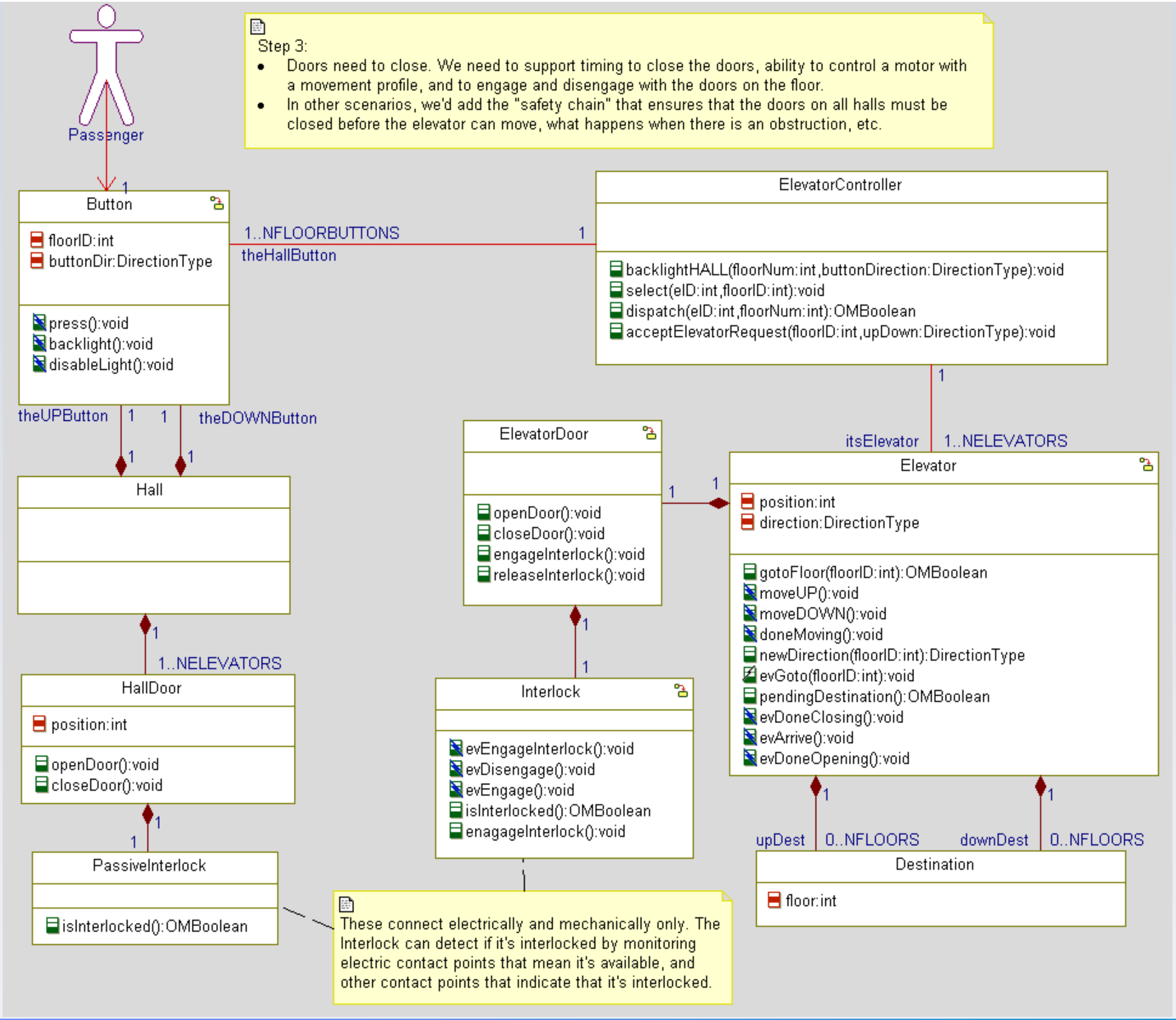
Executing Step 2


ENV
:ESystem
ESystem.itsElevatorController:ElevatorCo
ESystem.itsButton:Button
ESystem.itsElevator[0]:Elevator
ESystem.itsElevator[1]:Elevator
Create()
Create()
Create()
Create()
press()
acceptElevatorRequest(floorID=3, upDown=UP)
queryState(eState=Idle, position=1)
queryState(eState=Idle, position=30)
select(eID=0, floorID=3)
gotoFloor(floorID=3): TRUE
evGoto(floorID=3)
backlight()

Step 3: Closing the Doors



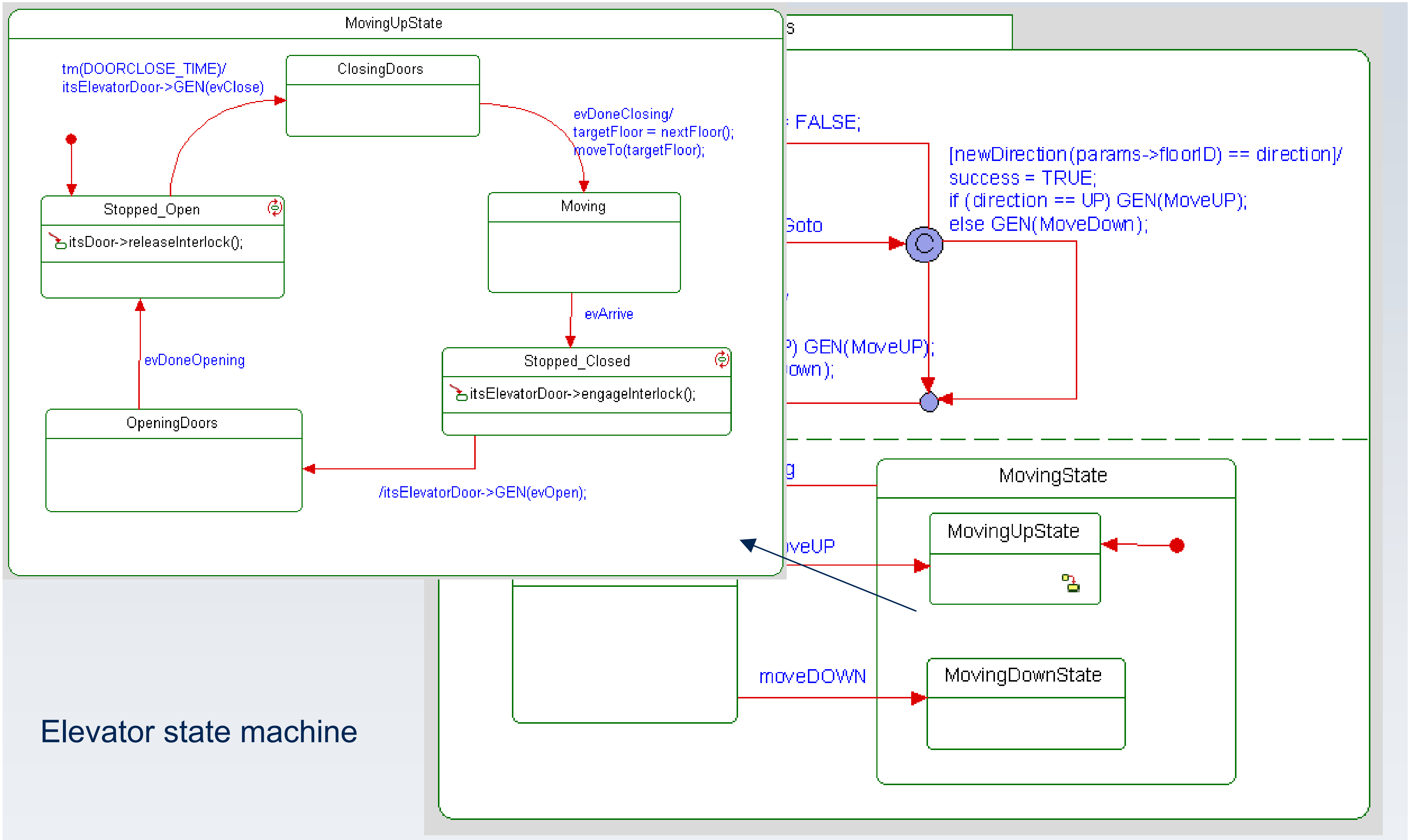
Step 3: Closing the Doors



Step 3:
- Doors need to close. We need to support timing to close the doors, ability to control a motor with a movement profile, and to engage and disengage with the doors on the floor.
- In other scenarios, we'd add the "safety chain" that ensures that the doors on all halls must be closed before the elevator can move, what happens when there is an obstruction, etc.

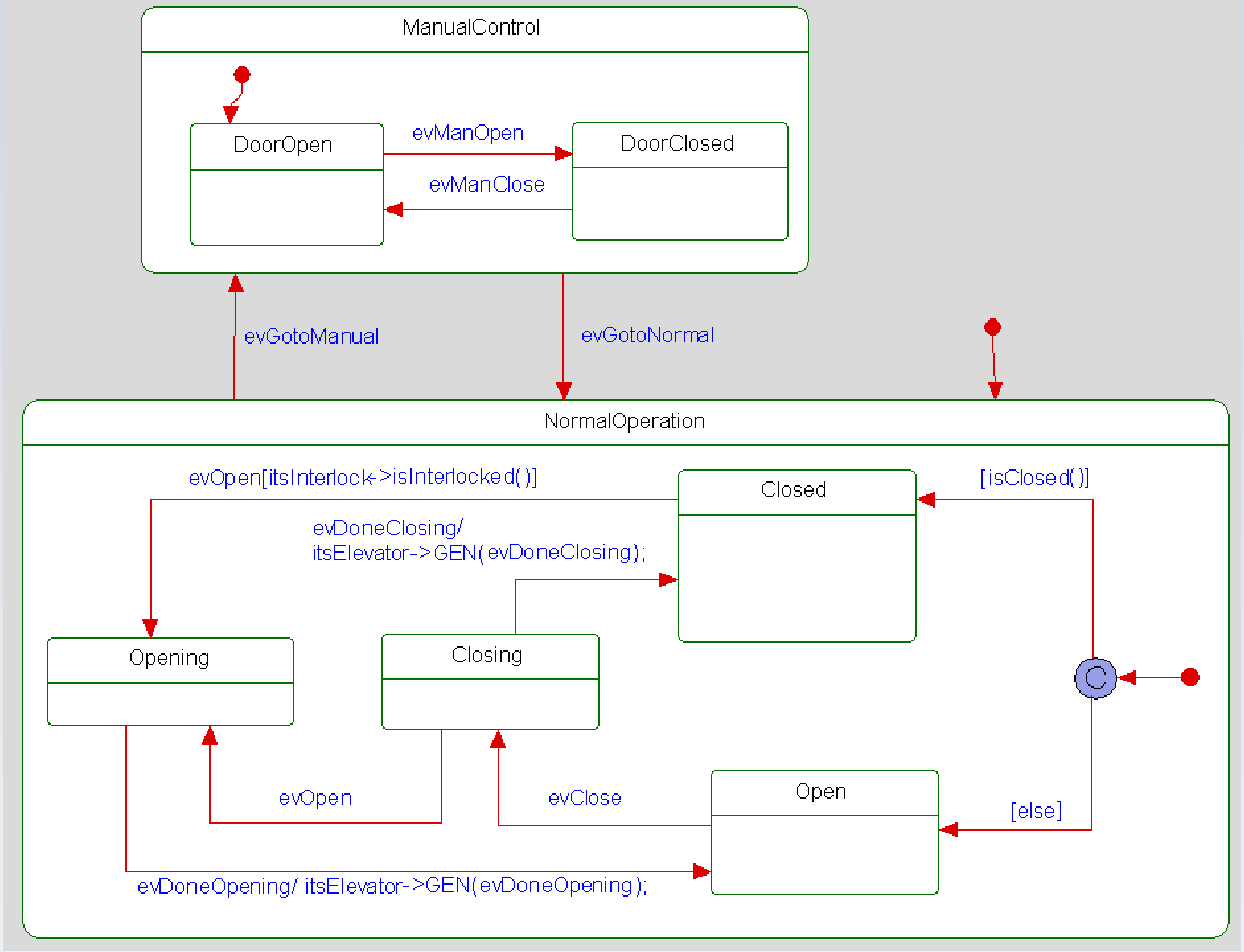
These connect electrically and mechanically only. The Interlock can detect if it's interlocked by monitoring electric contact points that mean it's available, and other contact points that indicate that it's interlocked.

Step 3: Closing the Doors



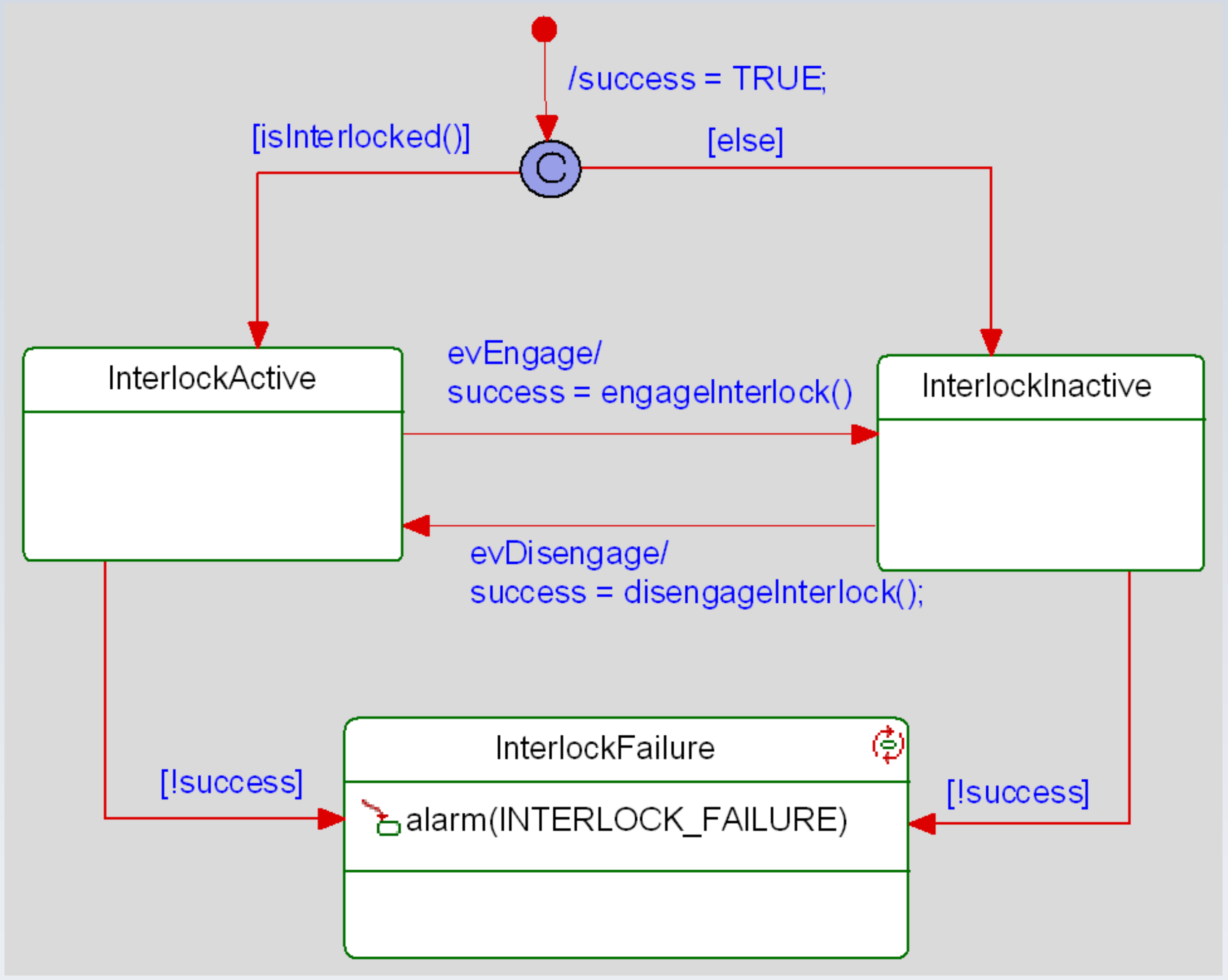
Elevator state machine

Step 3: Closing the Doors



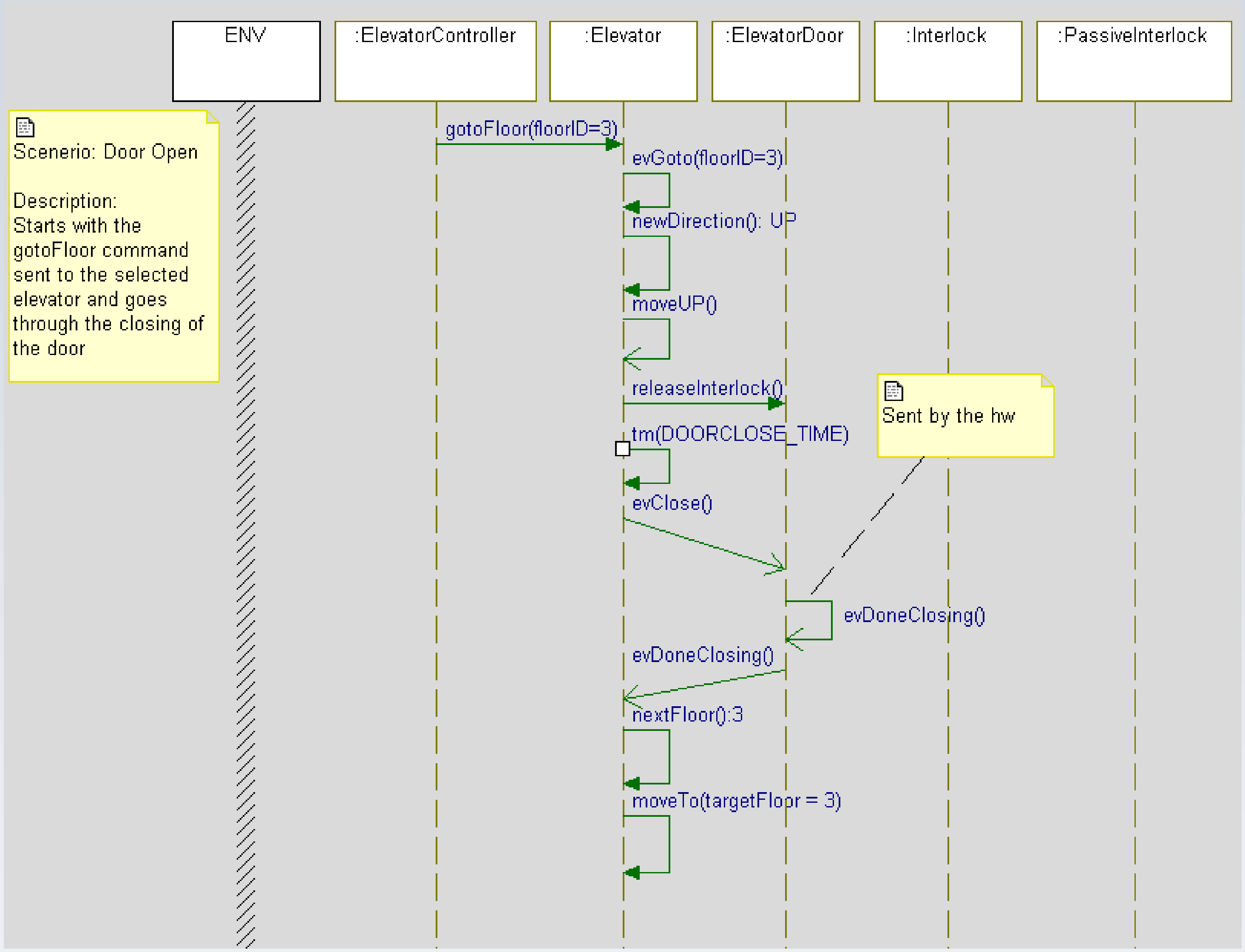
ElevatorDoor state machine

Step 3: Closing the Doors



Interlock state machine

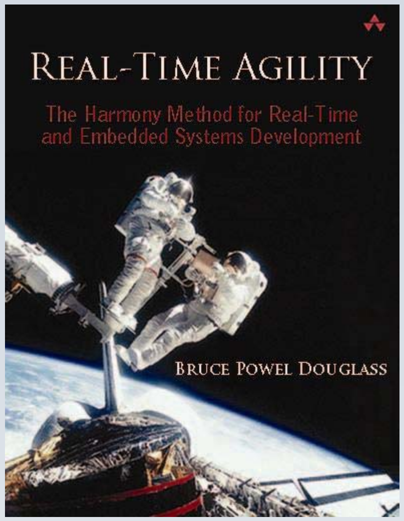
Step 3: Closing the Doors

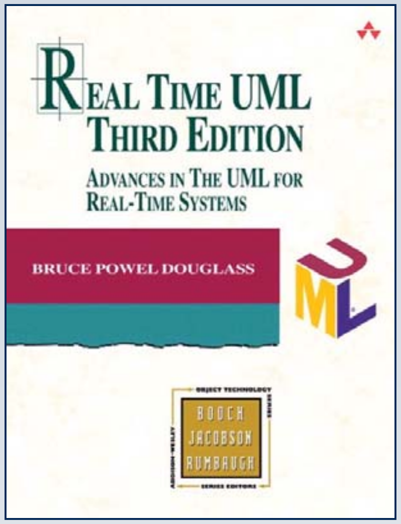


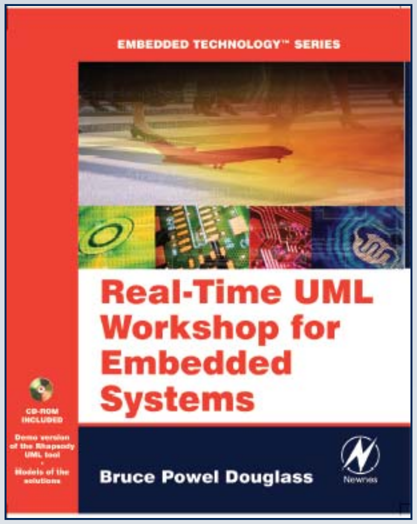
Summary

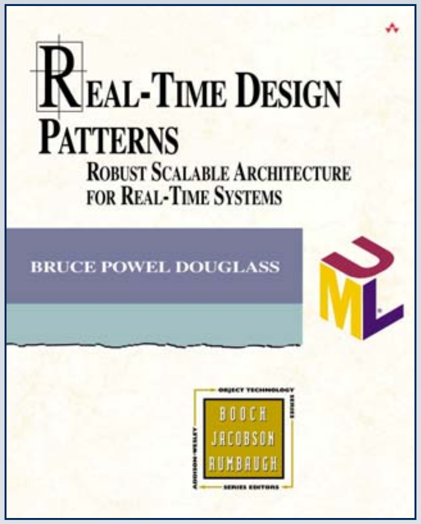
- Harmony/ESW is an agile process emphasizing
  - Use of UML Modeling to capture application behavior and structure
  - Not allowing defects to infect the design or implementation via
    - Continual code generation (many times/day)
    - Continual debugging and unit testing (many times/day)
    - Continuous integration (reestablish baseline >=1/day)
  - Frequently plan updates based on "truth on the ground"
- Harmony/ESW is
  - Requirements driven
  - Architecture-centric
  - Optimized for real-time and embedded systems, with guidance for
    - Design optimization with design patterns
    - Use of concurrency and OS features
    - Hardware/software co-development
    - Safety and reliability in analysis and design

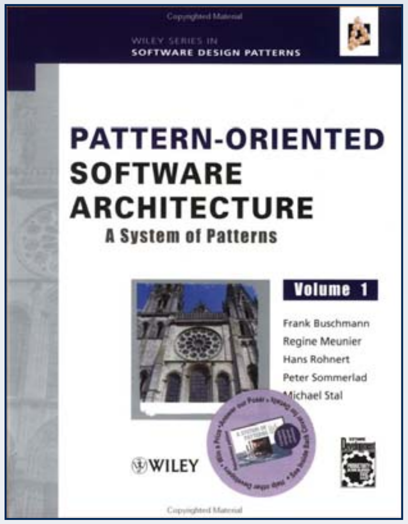
References

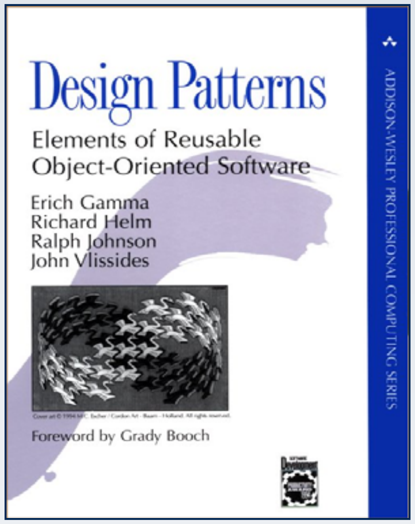

REAL-TIME AGILITY
The Harmony Method for Real-Time and Embedded Systems Development
BRUCE POWEL DOUGLASS

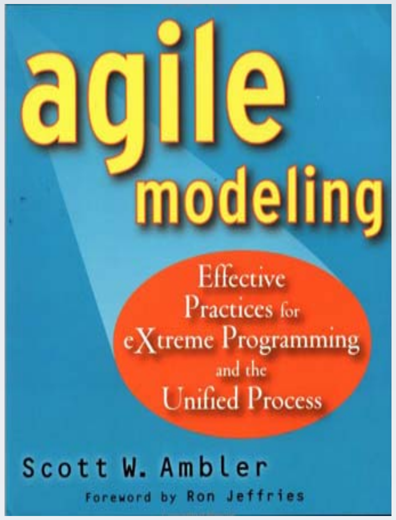

REAL TIME UML
THIRD EDITION
ADVANCES IN THE UML FOR REAL-TIME SYSTEMS
BRUCE POWEL DOUGLASS

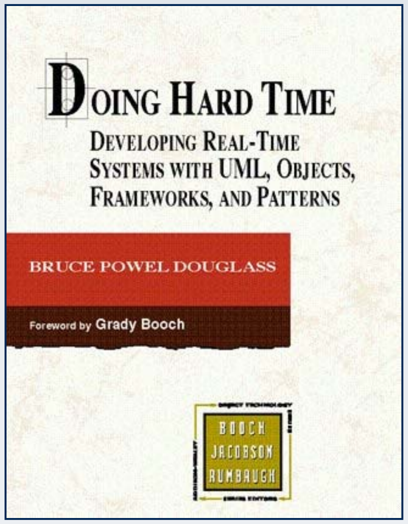

EMBEDDED TECHNOLOGY™ SERIES
Real-Time UML Workshop for Embedded Systems
Bruce Powel Douglass


REAL-TIME DESIGN PATTERNS
ROBUST SCALABLE ARCHITECTURE FOR REAL-TIME SYSTEMS
BRUCE POWEL DOUGLASS


WILEY SERIES IN SOFTWARE DESIGN PATTERNS
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
A System of Patterns
Volume 1
Frank Buschmann
Regine Meunier
Hans Rohnert
Peter Sommerlad
Michael Stal
WILEY


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


agile modeling
Effective Practices for eXtreme Programming and the Unified Process
Scott W. Ambler
Foreword by Ron Jeffries


DOING HARD TIME
DEVELOPING REAL-TIME SYSTEMS WITH UML, OBJECTS, FRAMEWORKS, AND PATTERNS
BRUCE POWEL DOUGLASS
Foreword by Grady Booch