Agile Model-Driven Development for Real-Time and Embedded Systems with Rational Harmony

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

Innovate2010
The Rational Software Conference

Let's build a smarter planet.

The premiere software and product delivery event.

IBM® Rational® Harmony™ for Embedded RealTime Development

- Rational Harmony for Embedded RealTime Development is a coherent set of best practices meant to effectively realize a set of core principles that define
  - Roles
  - Work Products
  - Tasks
- Rational Harmony for Embedded RealTime Development is
  - Agile
    - “Just enough” ceremony to meet the project governance needs but not more
    - “Test Driven Development”
    - Constant execution nanocycle
    - Continuous Integration
  - Requirements-driven
  - Architecture-centric
  - Optimized for
    - Systems and Software Engineering projects
    - Real-time (also safety-critical and high-reliability, optionally)
    - Embedded
    - Hardware-software co-design
    - Scalability from small to large systems projects

PEOPLE
PROCESS
TOOLS
BEST PRACTICES

Harmony 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
- Plan, Track, and Adapt
- Leading cause of project failure is ignoring risk
- Continuous attention to quality is essential
- Modeling is crucial

These principles guide the approaches, practices, work products, and workflows in the Harmony Process

Principle: Your primary goal: develop working software

- This implies that activities that do not directly assist in developing working software are necessarily *secondary*
  - Even if you're signing off documents faster than planned, you're not making real progress if the software doesn't work
- Practice: Daily activities should focus on
  - Understanding what needs to be done now
  - Getting execution immediately
  - (informally) Validating after each small incremental change that the software is doing the right thing
  - Avoid spending time doing tasks that don't contribute to execution
- This is not to say writing documents and other tasks are inappropriate, just that they are not your primary goal

Too often, managers and developers lose sight of this principle, but it is the most important single thing to remember!

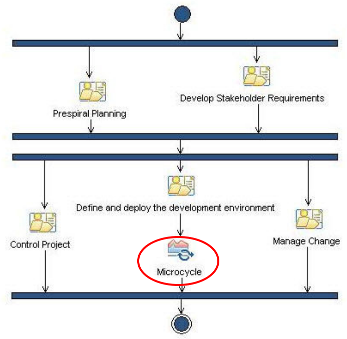
Principle: Continuous feedback is crucial

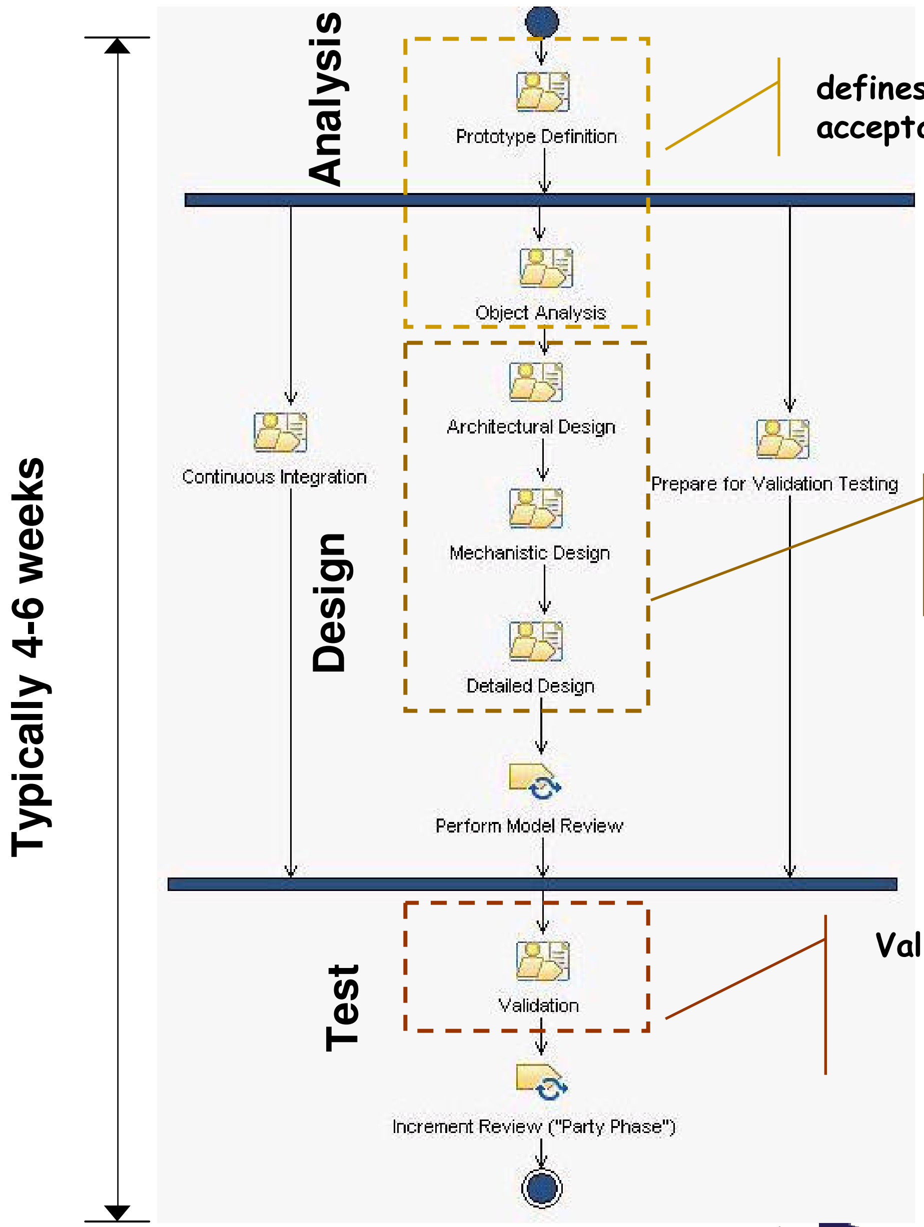
- "It ain't right if it don't run" – Law of Douglass
- "Optimism is the enemy of realism" – Law of Douglass
- "Optimism is a disease – feedback is the cure" – Kent Beck

- A key premise of the waterfall approach is that it is possible to go from phase to phase without making significant mistakes
  - We know now that *this is not true!*
- Practice: As we develop software over days, weeks, and months, we need assurance that we're doing the right thing:
  - Continual execution of the software provides feedback that the software at least runs
  - Comparing execution against expectations provides assurance that our expectations and actuality match
  - Debugging informally identifies and repairs either the expectation or the actuality
  - Testing formally assurances compliance between expectations and actuality
  - This execution should occur many times per day

The *nanocycle* in the Harmony process focuses on the continual execution of the software against its requirements

Practice: Harmony Microcycle


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



defines the properties of all acceptable solutions

specifies a particular "optimal" solution

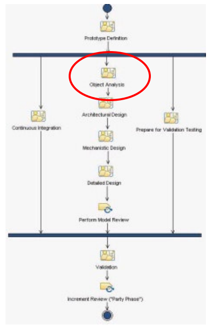
Validates against prototype mission

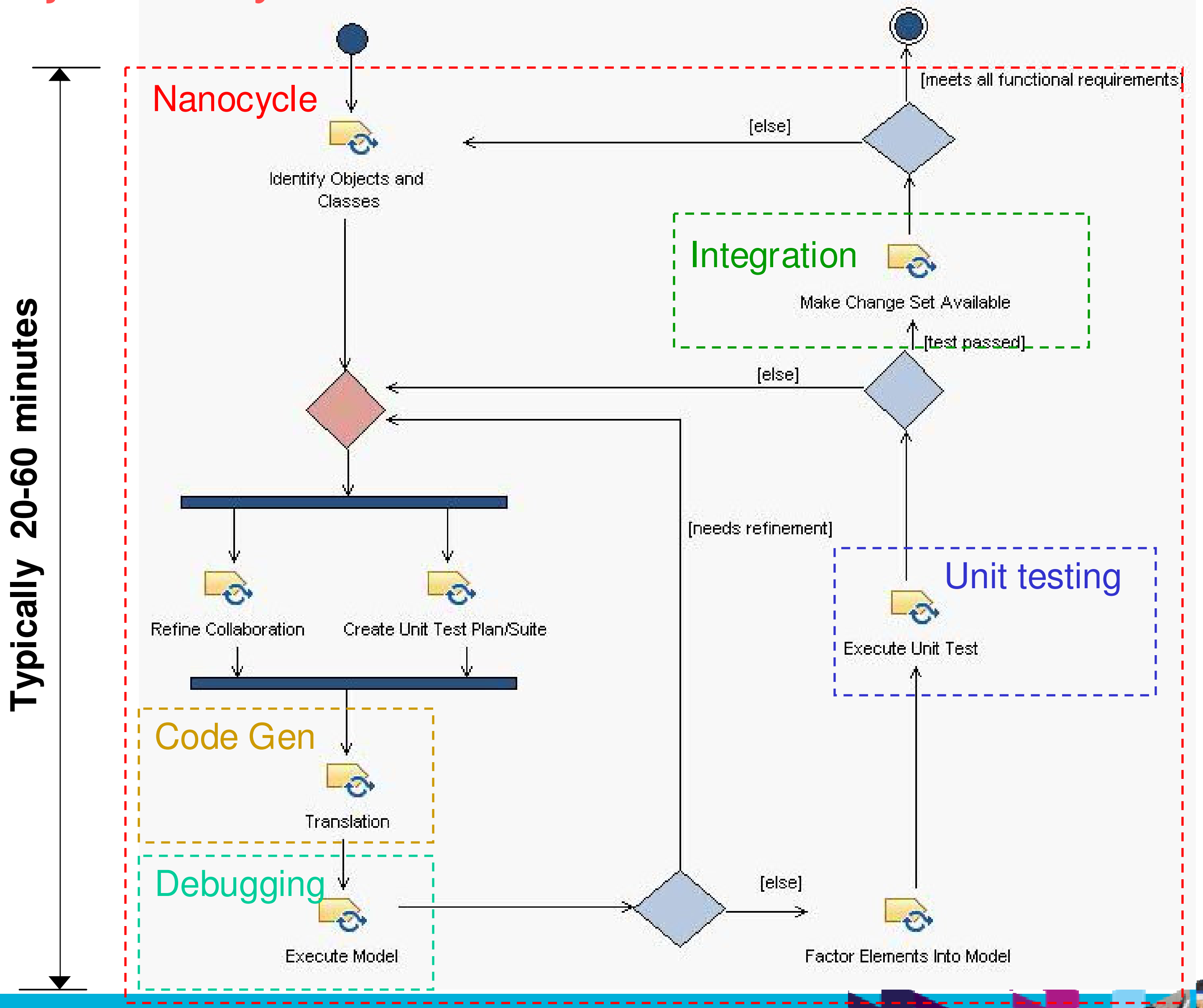
Practice: Nanocycle Incremental Construction

Produces partial models that are executed and debugged on a highly frequent basis - typically 10 - 60 minutes

- Based on the concept of *continual (informal) validation*
  - 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

Practice: Object Analysis Workflow



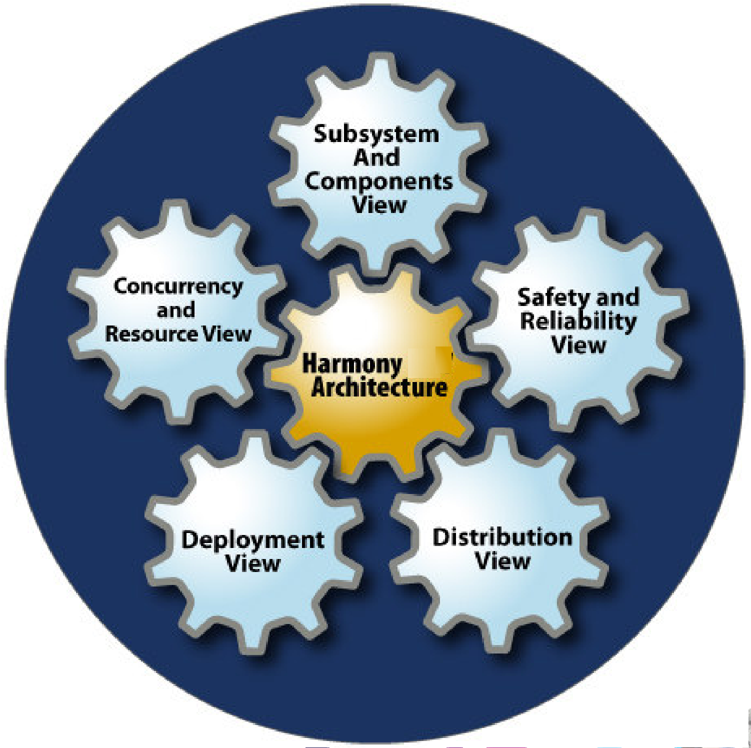


Principle: Orthogonal sets of strategic design decisions define your architecture

- Architecture consists of a *coherent set of design and technology decisions that affect most or all of the system*
  - *Design decisions are not about functionality – they are about optimizing the functionality of the system in various ways*
  - *A strong architecture is important because it optimizes the system according to a small set of consistent criteria within which all developers must work*

Practice: Harmony's 5 Key Views of Architecture

- Harmony identifies 3 levels of design optimization
  - Architectural
  - Mechanistic
  - Detailed
- Architecture is divided into 5 primary views
  - Each view is characterized by its own set of design patterns, approaches, and technologies
  - Secondary architecture views include
    - Information Assurance
    - Data Management
    - Quality of Service Management
    - Error and exception Management


Subsystem And Components View
Concurrency and Resource View
Harmony Architecture
Safety and Reliability View
Deployment View
Distribution View

Practice: Apply Patterns Intelligently for System Architecture

- "Apply no design before its time" - Law of Douglass

- Use design at the right time
  - Analysis focuses on *essential properties* - and executing the correct functionality
  - Design focuses on optimizing the analysis model against the weighted set of design criteria - therefore *optimality*
  - Don't supply decision or technology decisions until the functionality is correct and complete for the prototype at hand
- Harmony Agile Design is primarily a design-pattern focused activity
  - Design patterns are generalized solutions to recurring problems
  - Each design has benefits and costs
  - To select the best design pattern you must understand the optimization criteria in detail first
  - Don't optimize too early. Optimize only after:
    - Functionality is in place and working correctly
    - You understand the quality of service and other design criteria
    - You've identified and ranked the design criteria
    - You understand the costs as well as the benefits of the proposed design solutions

Principle: Plan, Track, and Adapt

▪“The more you know, ***the more you know!”*** - Law of Douglass
▪“Plan to Replan” – Law of Douglass

- Far too many managers use “Ballistic Planning”
  - that is, they identify a plan and pray* that some number of years later, the project will arrive on the target date and at the target cost, despite not tracking intermediate progress and identifying and taking corrective action when appropriate
- An agile methods key principle is “Dynamic Planning”
  - Plans are always made in the face of incomplete knowledge
  - Understanding improves as the project processes
  - In addition, things change, such as
    - Requirements
    - Customer needs
    - Environments
    - Capabilities
  - Thus, plans provide a blueprint but must be frequently updated to reflect the deeper understanding and address changes

***Hope** is not a plan!*

Principle: Leading cause of project failure is ignoring risk

▪"Leading cause of failure is ignoring risk" - Law of Douglass

- Most projects go awry because of *predictable* problems that were never addressed, e.g.:
  - Schedules were unrealistic from the outset
  - Project is under/over manned
  - Project scope is too ambitious
  - Requirements were inadequate or don't meet stakeholder needs
  - Obvious process inefficiencies were never addressed
  - Staff is overtasked due to unscheduled "wedge" projects and tasks
  - New technology is poorly understood by relevant team members
- Practice: The best way to reduce risk is to *manage it:*
  - Identify the risks
  - Define risk mitigation activities (RMAs)
  - Plan RMA execution in schedule
  - Heed the results of your RMAs
  - Frequently look for new risks

Continuous attention to quality is essential

▪“The best way not to have defects in your software is to not put them in!” - Law of Douglass

- Quality cannot be effectively added later into developed software
- *Practice: Continual attention to technical excellence* is the best way to get quality software, using practices such as
  - Constant execution
    - Execute after small incremental changes, typically no less than once per hour
  - Explicitly state pre- and post-conditions
    - State assumptions for correct execution (e.g. memory needs, parameter value ranges, etc)
  - Defensive design
    - Plan for your software to be robust in the presence of precondition violations
  - Test-Driven Development
    - Develop your tests prior to, or in conjunction with, the software elements
  - Peer modeling to give another set of eyes over the design or implementation

Being *agile* not not an excuse to hack or create low-quality software. It is an opportunity to focus on the things that make your software better!

Principle: Modeling is crucial

- "Modeling is next to Godliness" - Law of Douglass

- Modeling, *properly performed*, can greatly improve your design in terms of correctness, functionality, understandability, portability, and reusability
- Modeling, *improperly performed*, provides marginal benefit
- Practice: The Harmony process provides strong guidance over the best ways to model, for example:
  - Each diagram should have a stated mission (purpose) and include all elements that contribute to that mission, but no elements that do not. Example class diagram missions include:
    - Collaboration of classes realize a single use case
    - Provide an architectural view (see Five Key Architectural Views principle), such as a task diagram for the concurrency architecture
    - Show a generalization taxonomy
    - Show a class structure (aka "structure diagram")
    - Show contents of a package

Practice: Use model-code associativity

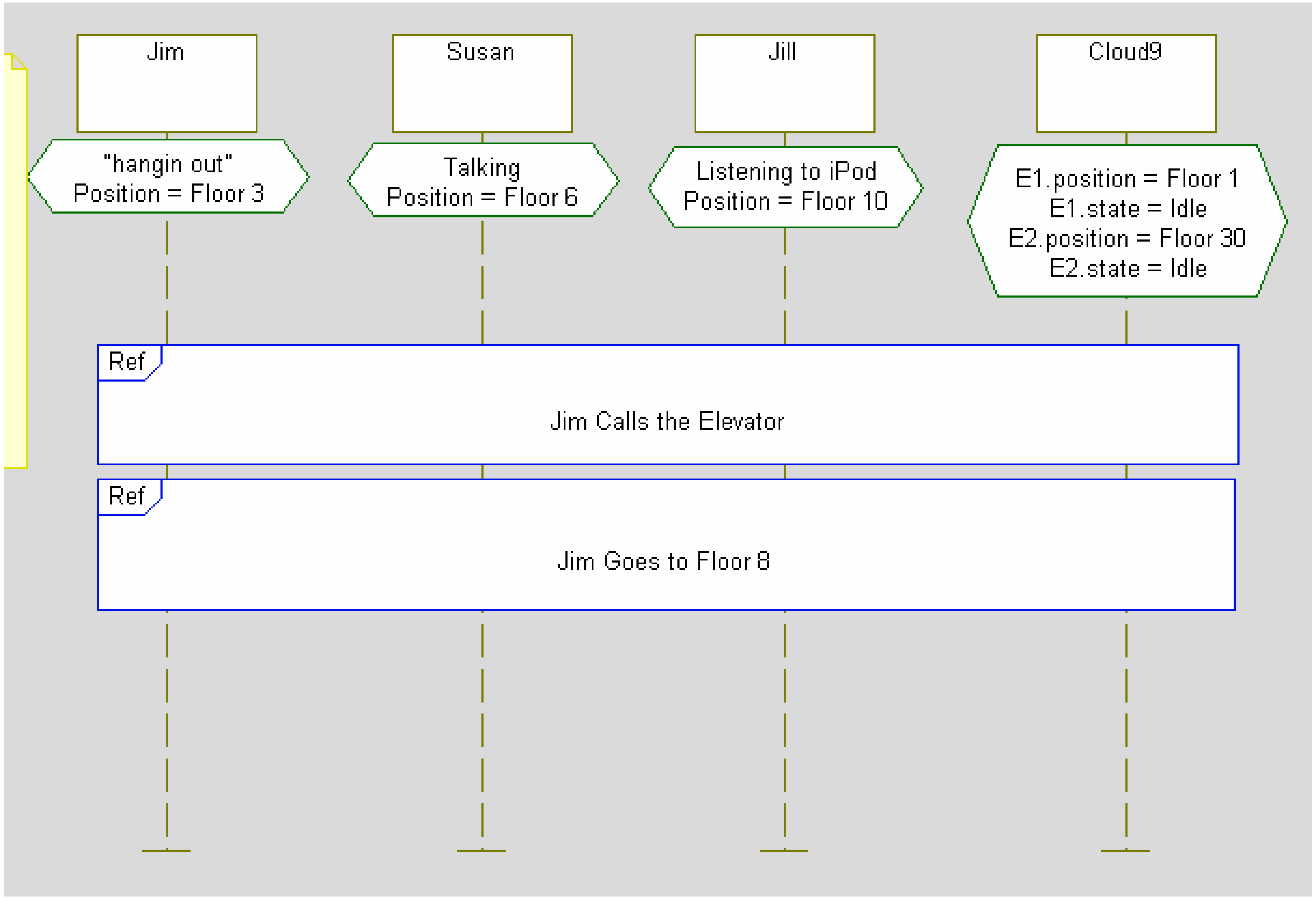
▪"The model *is* the code!" - Law of Douglass

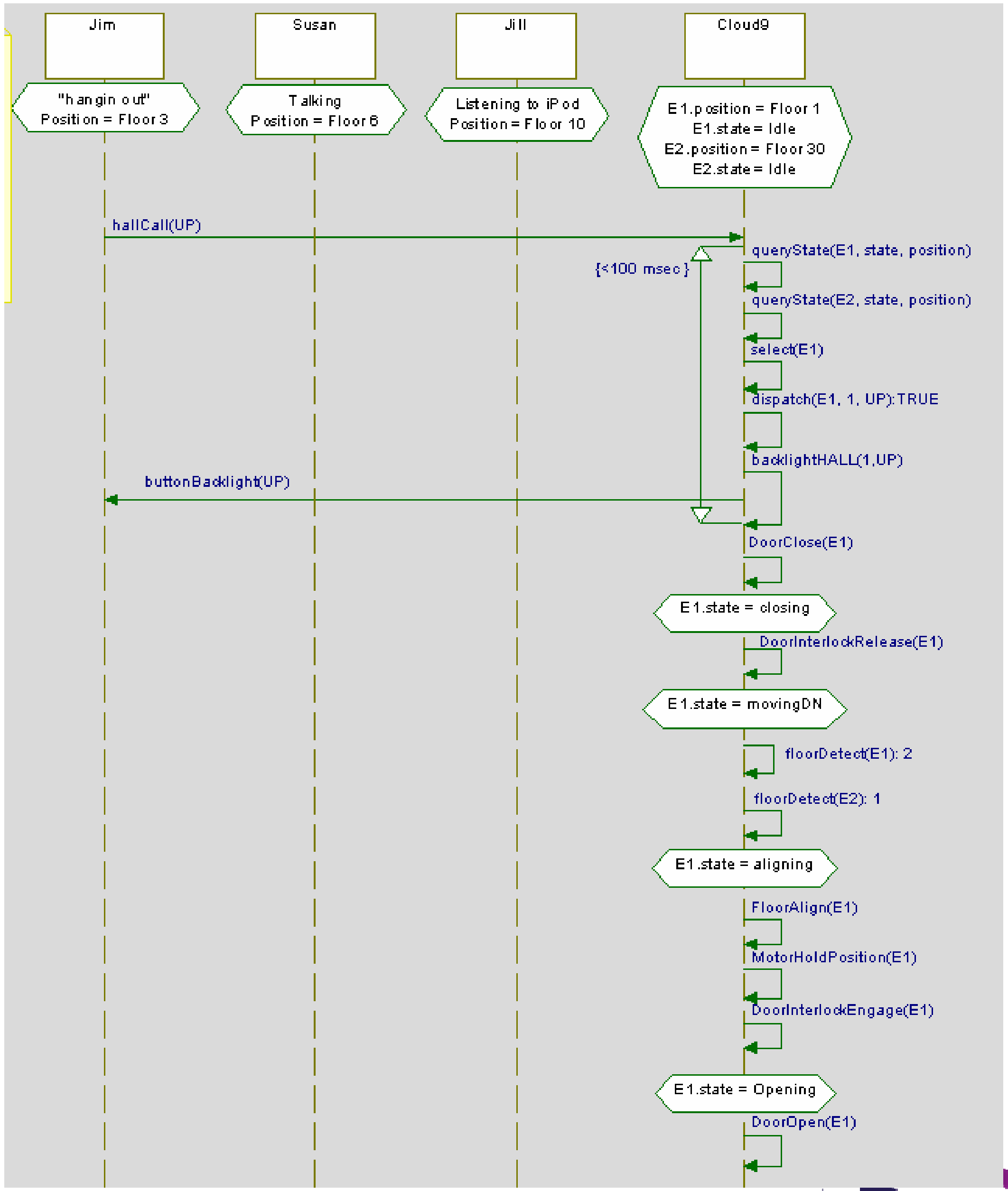
- MCA allows tools to automatically maintain synchronicity between the source code and the model
  - MCA assumes you have a model already. To construct a new model from an existing code base, we "reverse engineer" the design
- Developers should work at any level of abstraction if it is appropriate
  - Mostly, developers work in the model level and "forward-generate" code but sometimes it is more convenient to work at the code level
- Not all code must be modeled!
  - It is easy to use hand-written code with model-based code
  - If some code is highly stable, known to be of high quality, and won't undergo much maintenance, then making a model of it make not be an effective use of time

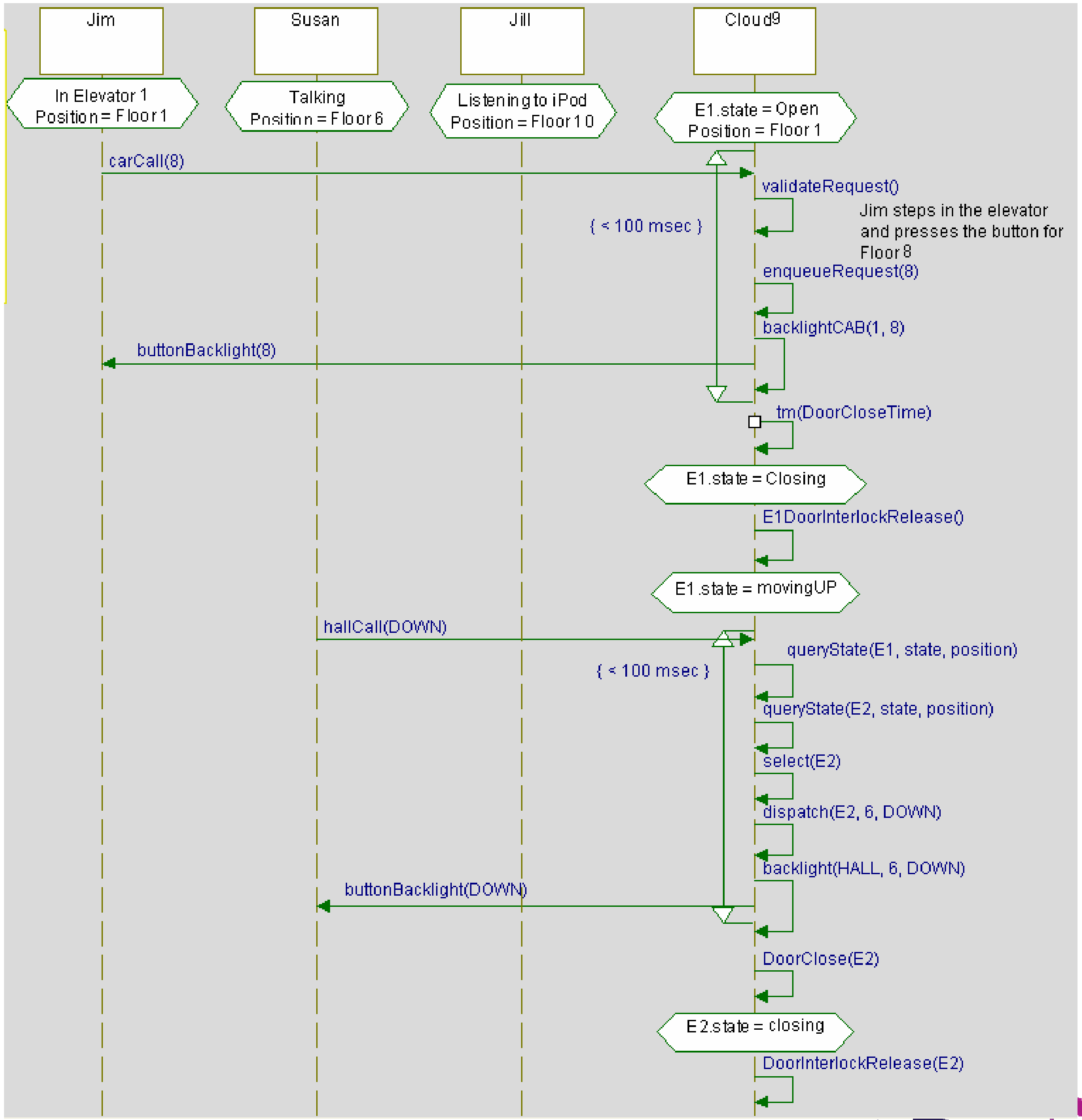
A Look at the Harmony for Embedded RealTime Development Nanocycle

- Throughout analysis and design, developers produces partial models that are executed and debugged on a highly frequent basis - typically 10 - 60 minutes
- Based on the concept of *continual (informal) validation*
  - 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

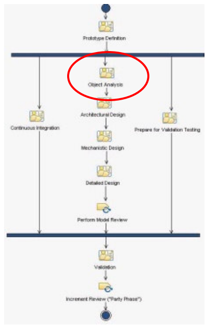
Nanocycle Example: Elevator Scenario (1 of 3)

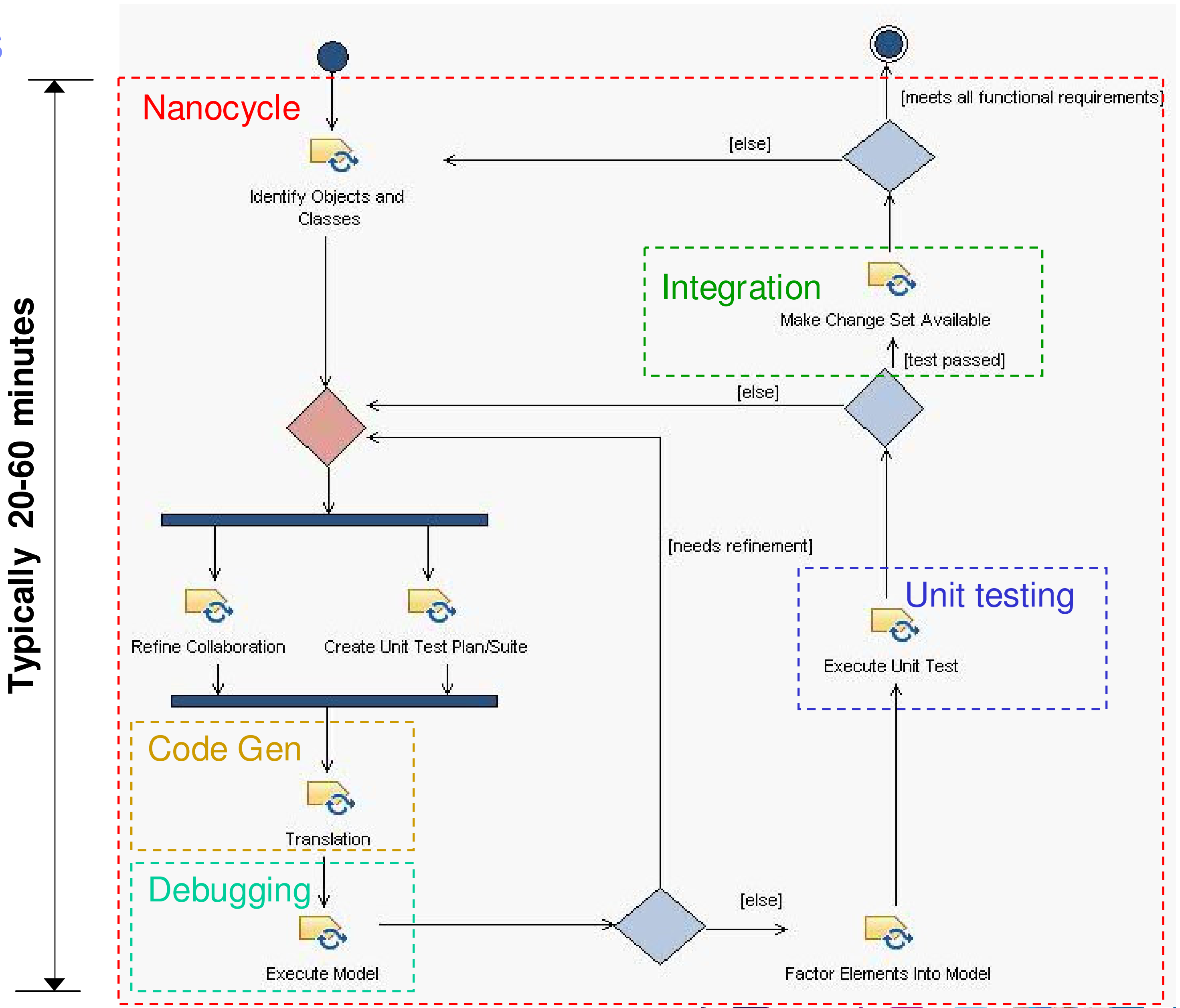



Jim
Susan
Jill
Cloud9
"hangin out" Position = Floor 3
Talking Position = Floor 6
Listening to iPod Position = Floor 10
E1.position = Floor 1
E1.state = Idle
E2.position = Floor 30
E2.state = Idle
hallCall(UP)
queryState(E1, state, position)
{<100 msec}
queryState(E2, state, position)
select(E1)
dispatch(E1, 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)

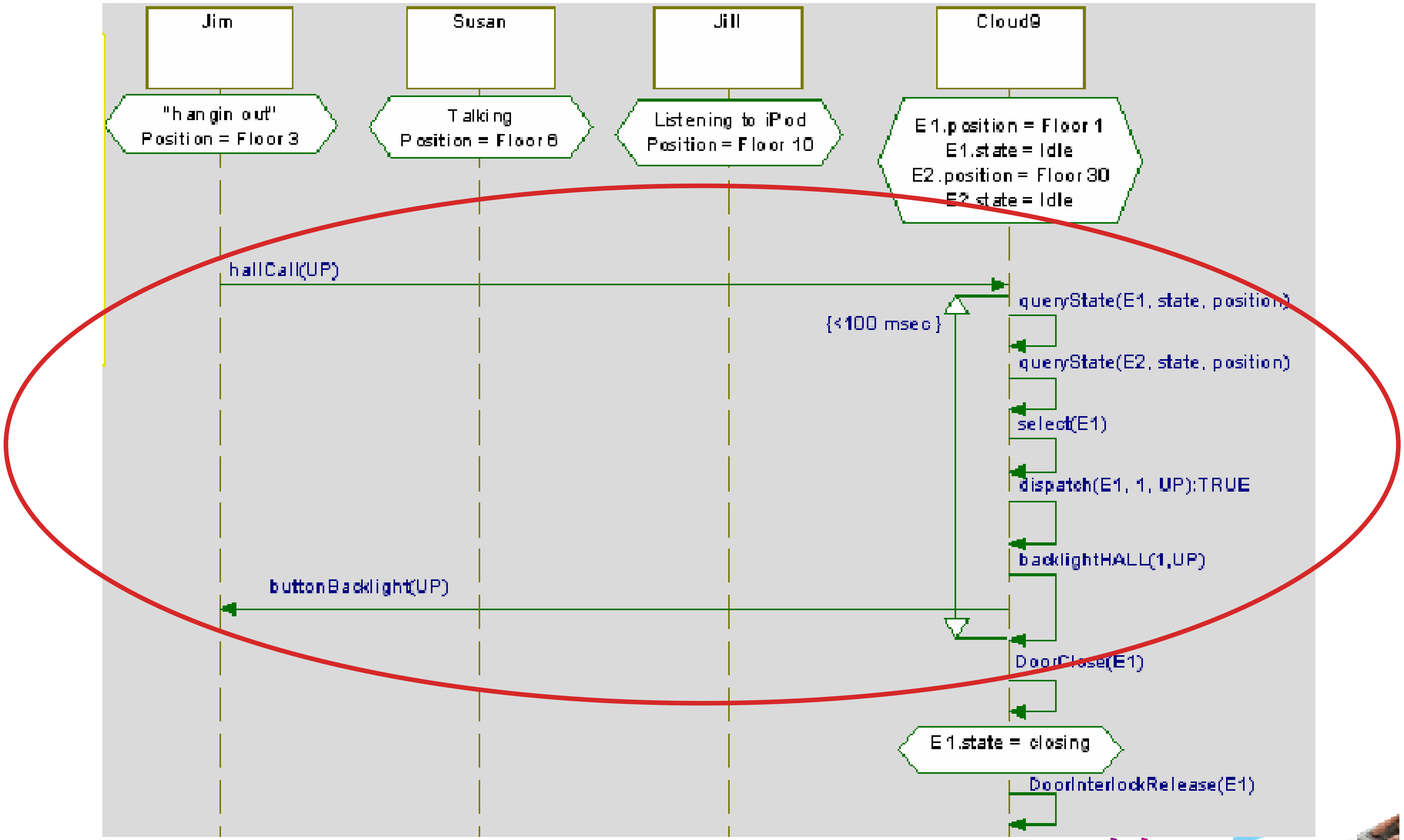

Jim
Susan
Jill
Cloud9
In Elevator 1 Position = Floor 1
Talking Position = Floor 6
Listening to iPod Position = Floor 10
E1.state = Open Position = Floor 1
carCall(8)
validateRequest()
Jim steps in the elevator and presses the button for Floor 8
{ < 100 msec }
enqueueRequest(8)
backlightCAB(1, 8)
buttonBacklight(8)
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





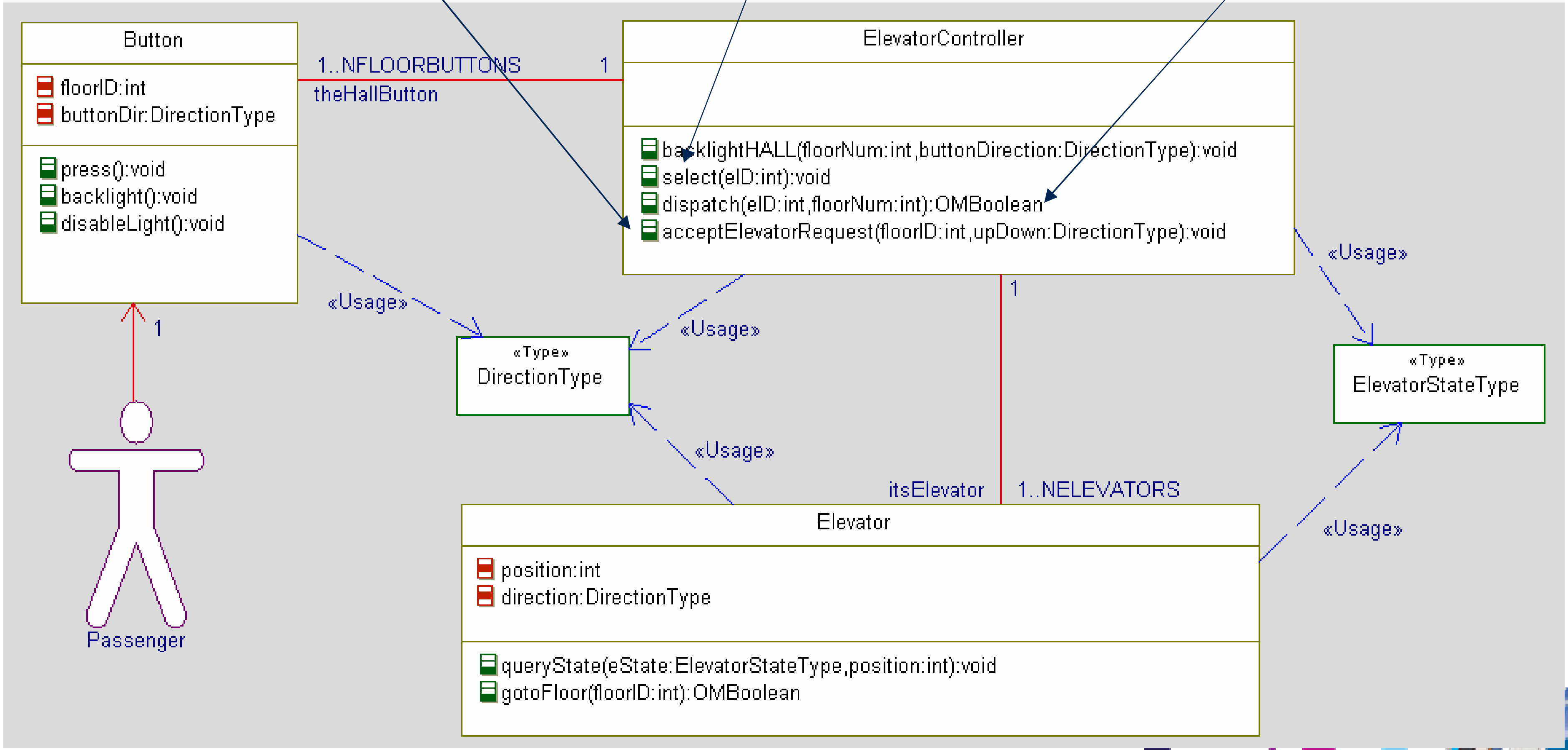
Step 1: Requesting the elevator



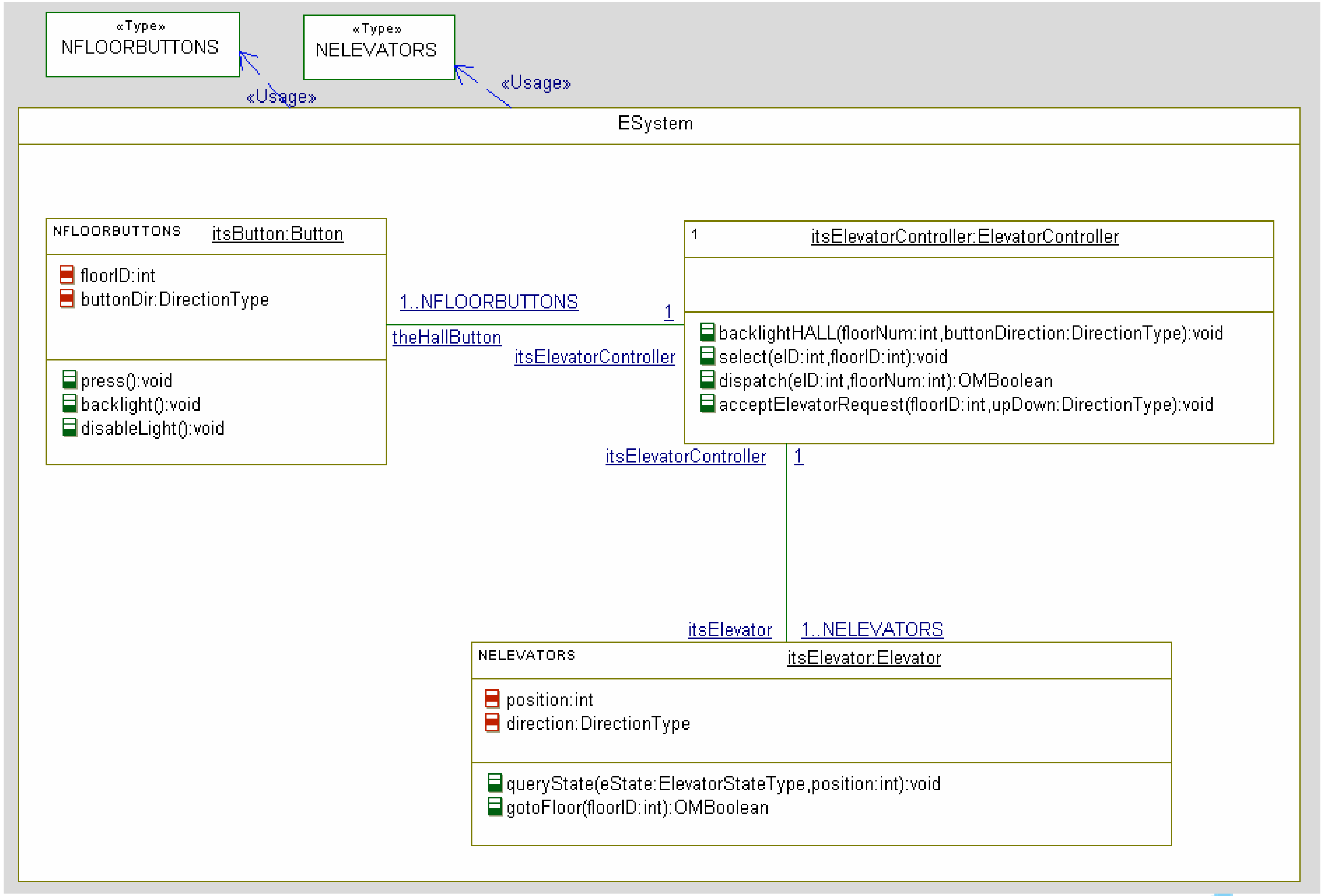
```
ElevatorStateType elevState;
int j;
int pos;

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();
};
```

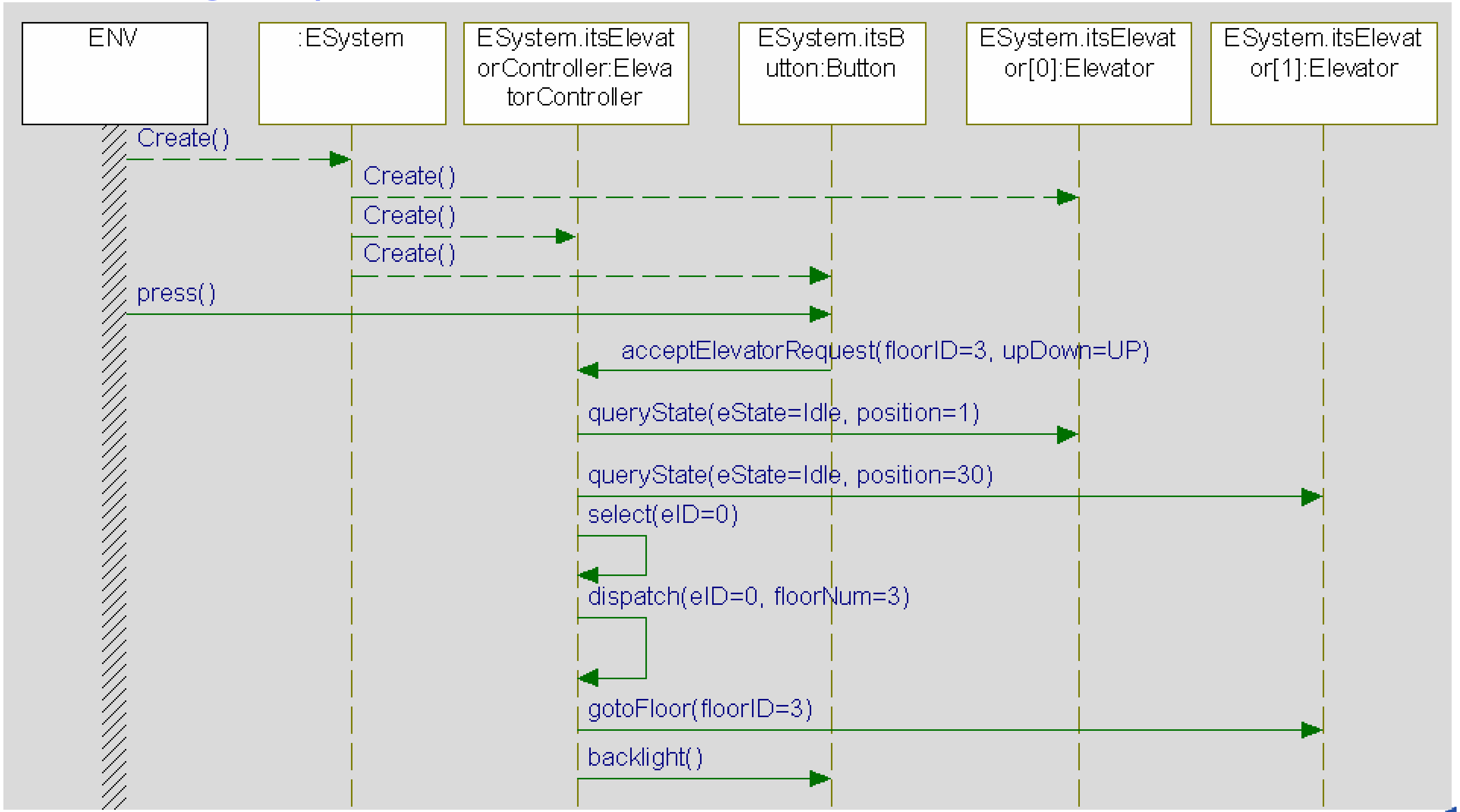
```
itsElevator[eID]->gotoFloor(floorNum);
return TRUE;
```



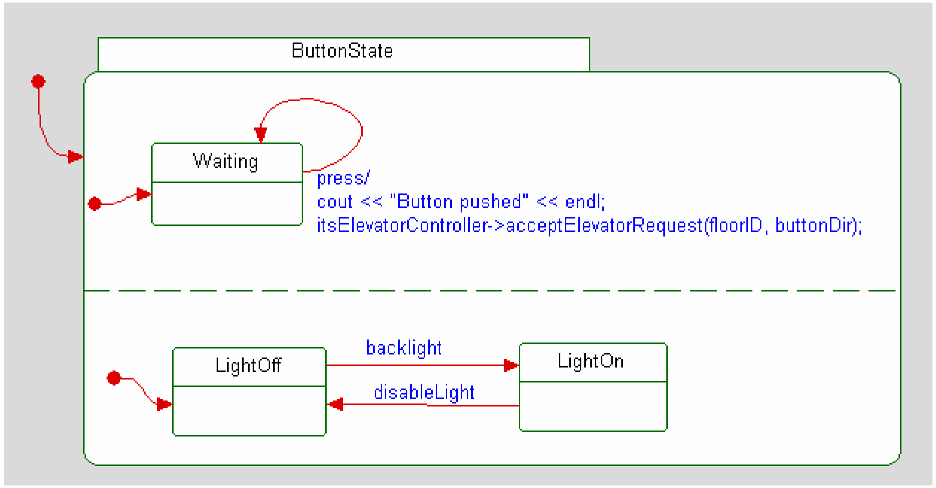
Executing Step 1



Executing Step 1

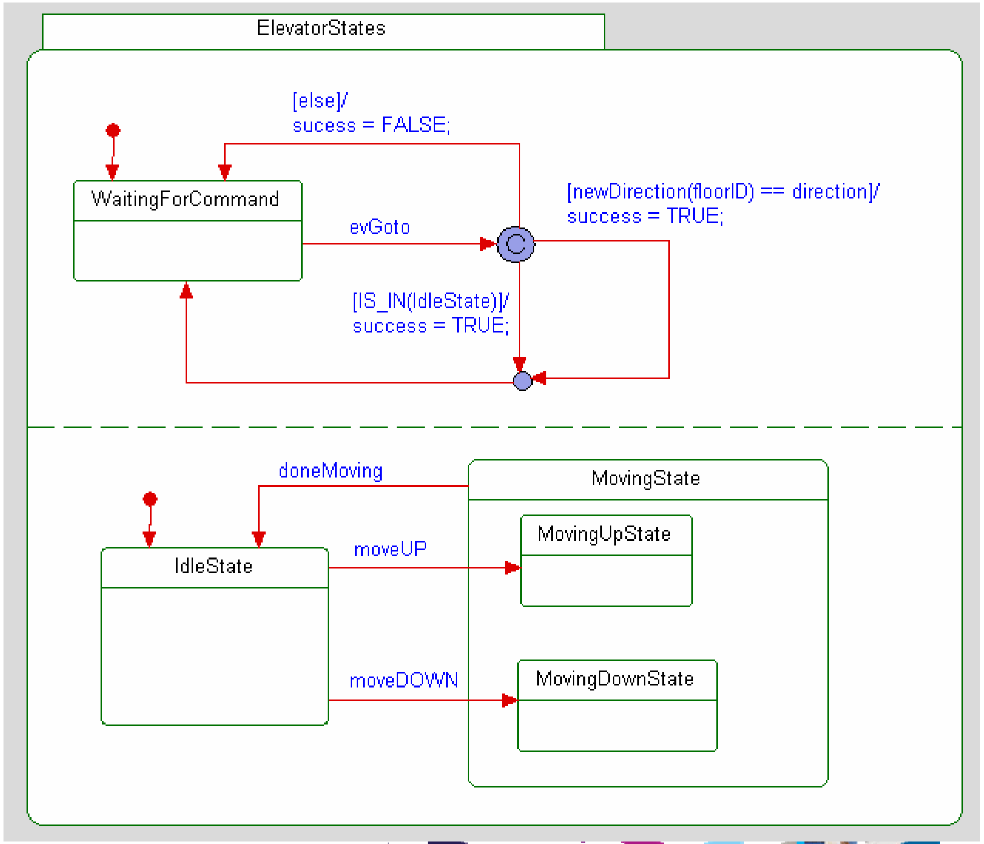

ENV
:ESystem
ESystem.itsElevatorController:ElevatorController
ESystem.itsButton:Button
ESystem.itsElevator[0]:Elevator
ESystem.itsElevator[1]:Elevator
Create()
Create()
Create()
Create()
press()
acceptElevatorRequest(floorID=3, upDown=UP)
queryState(eState=Idle, position=1)
queryState(eState=Idle, position=30)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

Step 2: Make classes reactive

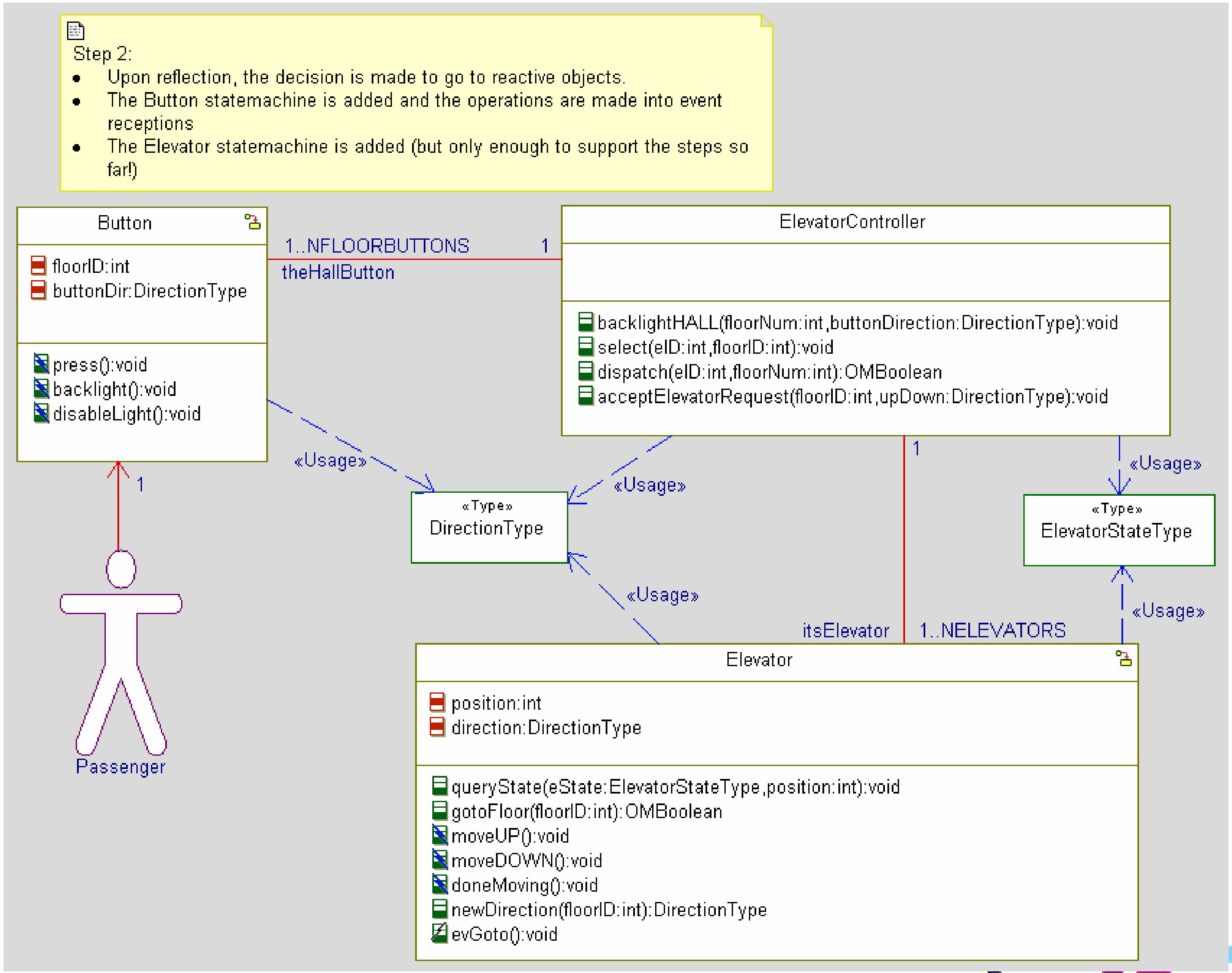

ButtonState
Waiting
press/
cout << "Button pushed" << endl;
itsElevatorController->acceptElevatorRequest(floorID, buttonDir);
LightOff
backlight
disableLight
LightOn

Button state machine

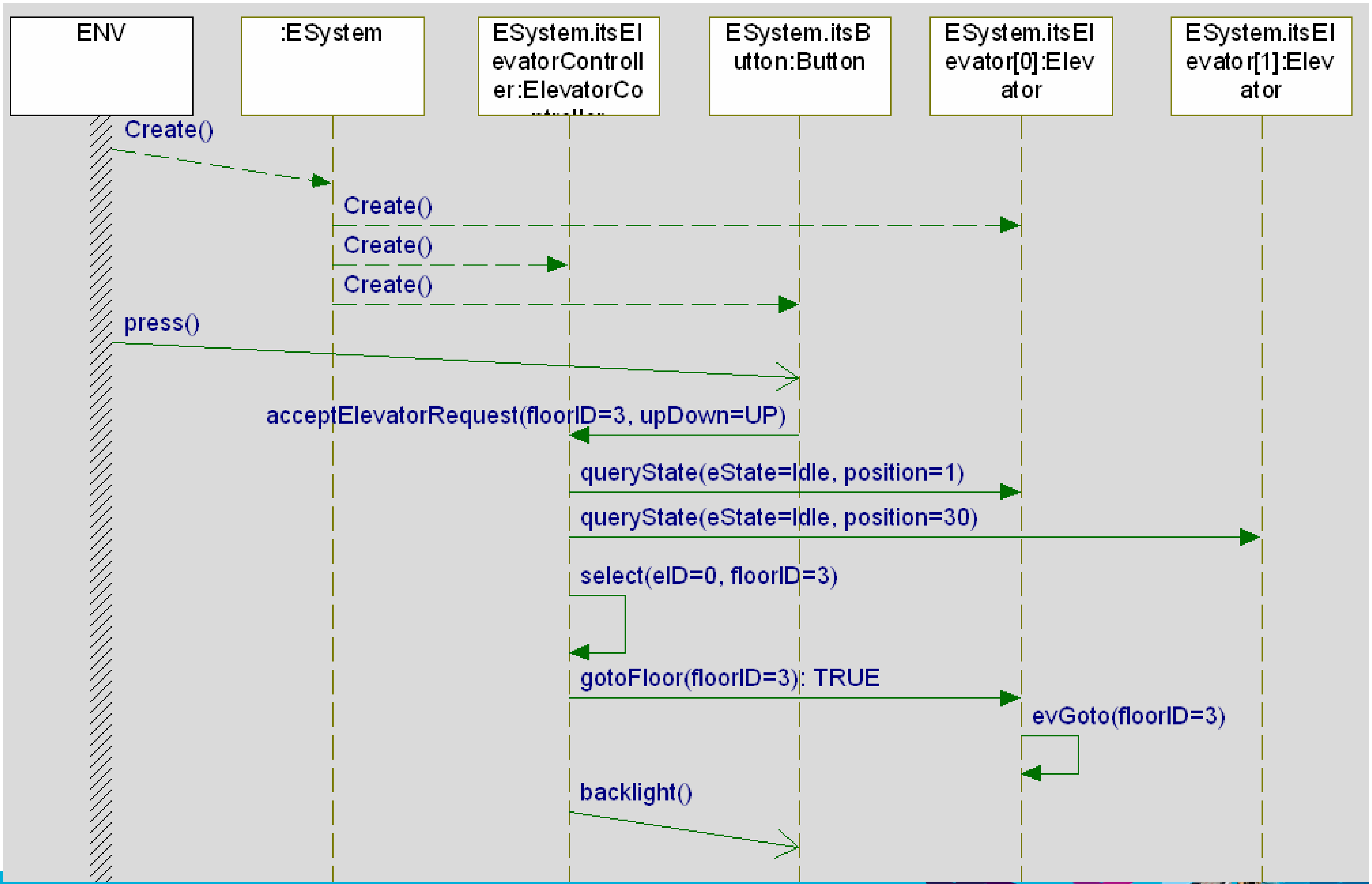
Elevator state machine



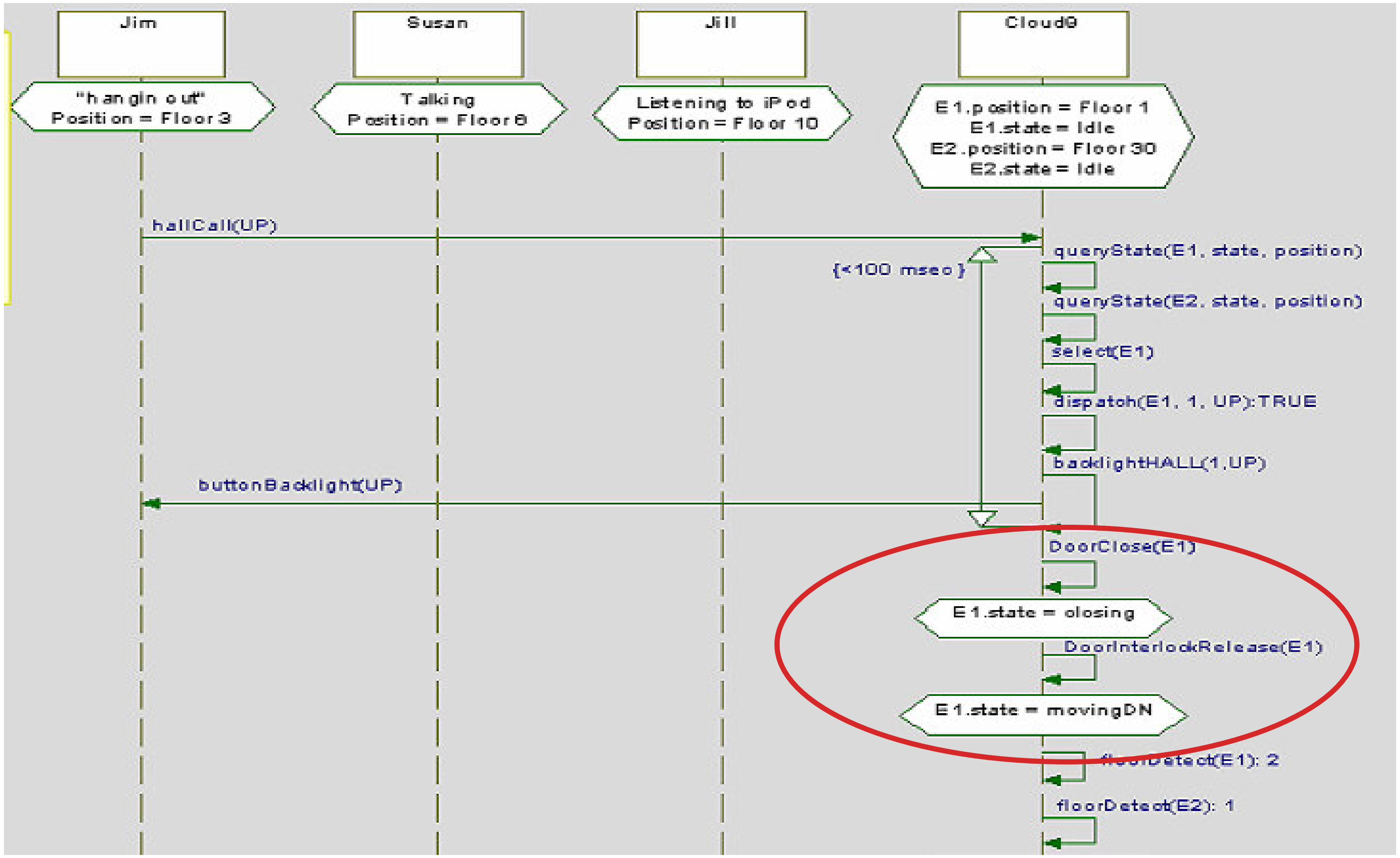
Step 2: Class Diagram


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!)
Button
floorID:int
buttonDir:DirectionType
press():void
backlight():void
disableLight():void
1..NFLOORBUTTONS
1
theHallButton
ElevatorController
backlightHALL(floorNum:int,buttonDirection:DirectionType):void
select(elD:int,floorID:int):void
dispatch(elD:int,floorNum:int):OMBoolean
acceptElevatorRequest(floorID:int,upDown:DirectionType):void
«Usage»
«Type» DirectionType
«Usage»
«Usage»
«Type» ElevatorStateType
«Usage»
«Usage»
1
1
Passenger
itsElevator
1..NELEVATORS
Elevator
position:int
direction:DirectionType
queryState(eState:ElevatorStateType,position:int):void
gotoFloor(floorID:int):OMBoolean
moveUP():void
moveDOWN():void
doneMoving():void
newDirection(floorID:int):DirectionType
evGoto():void

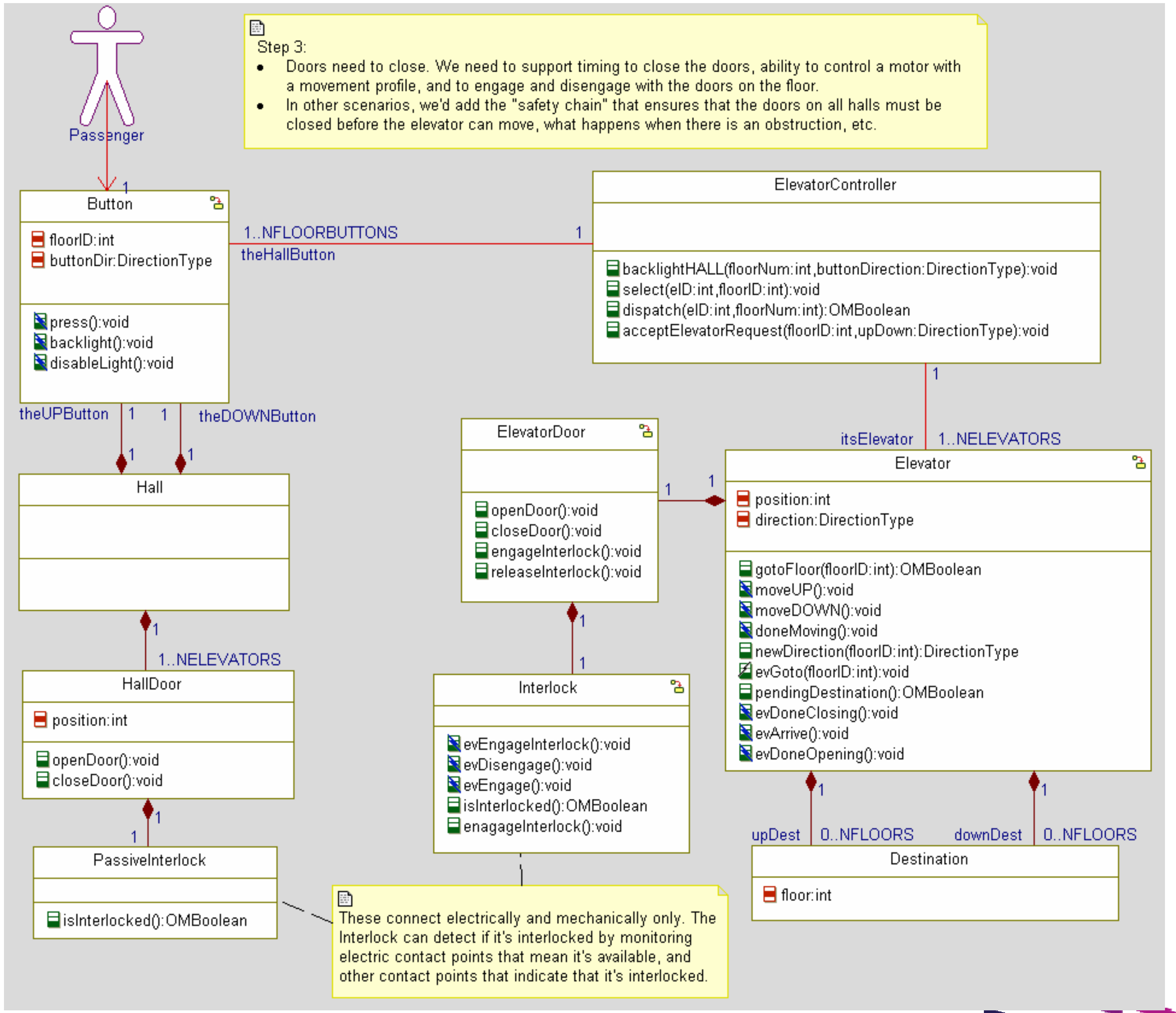
Executing Step 2


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(elD=0, floorID=3)
gotoFloor(floorID=3): TRUE
evGoto(floorID=3)
backlight()

Step 3: Closing the Doors

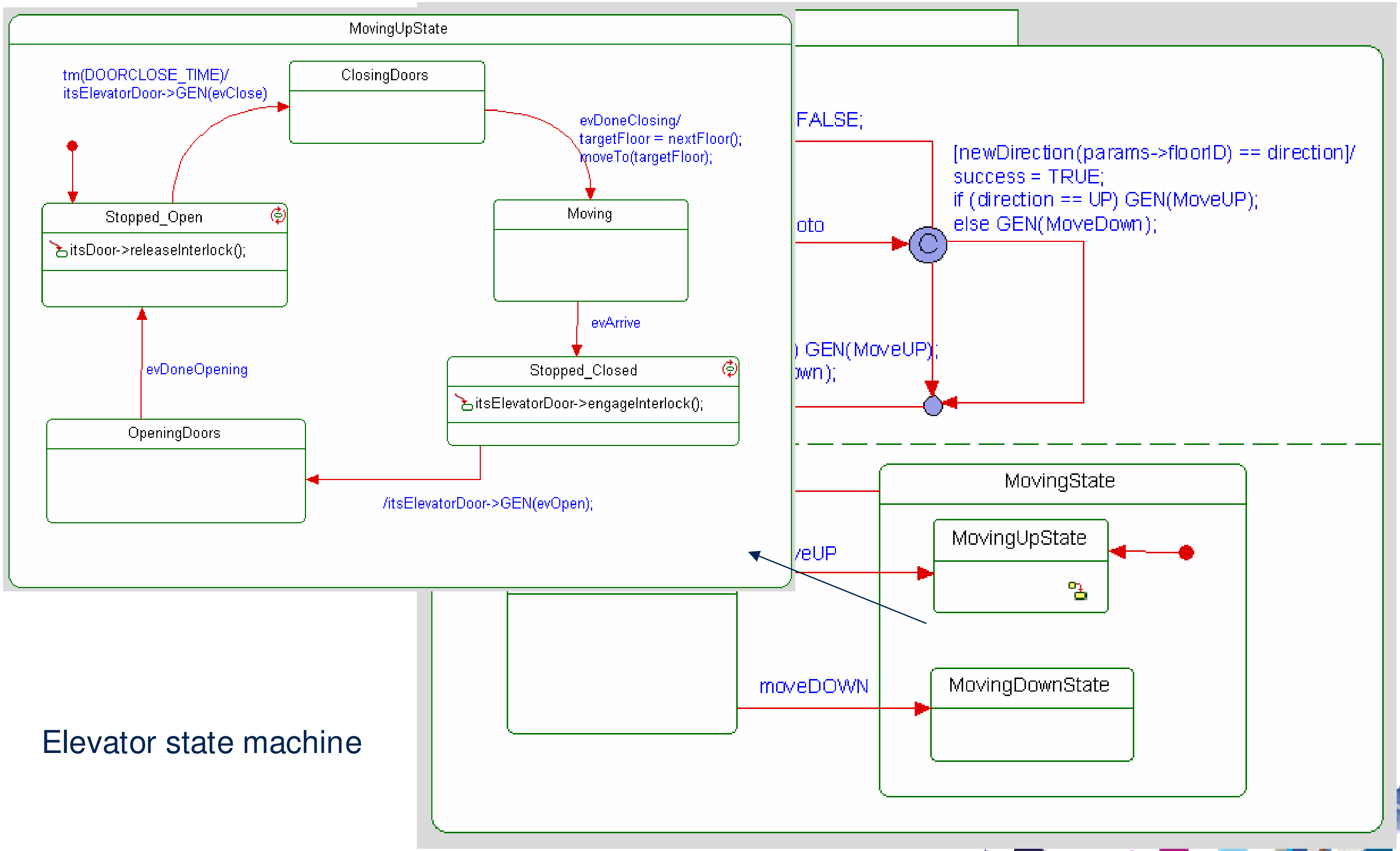


Step 3: Closing the Doors


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.

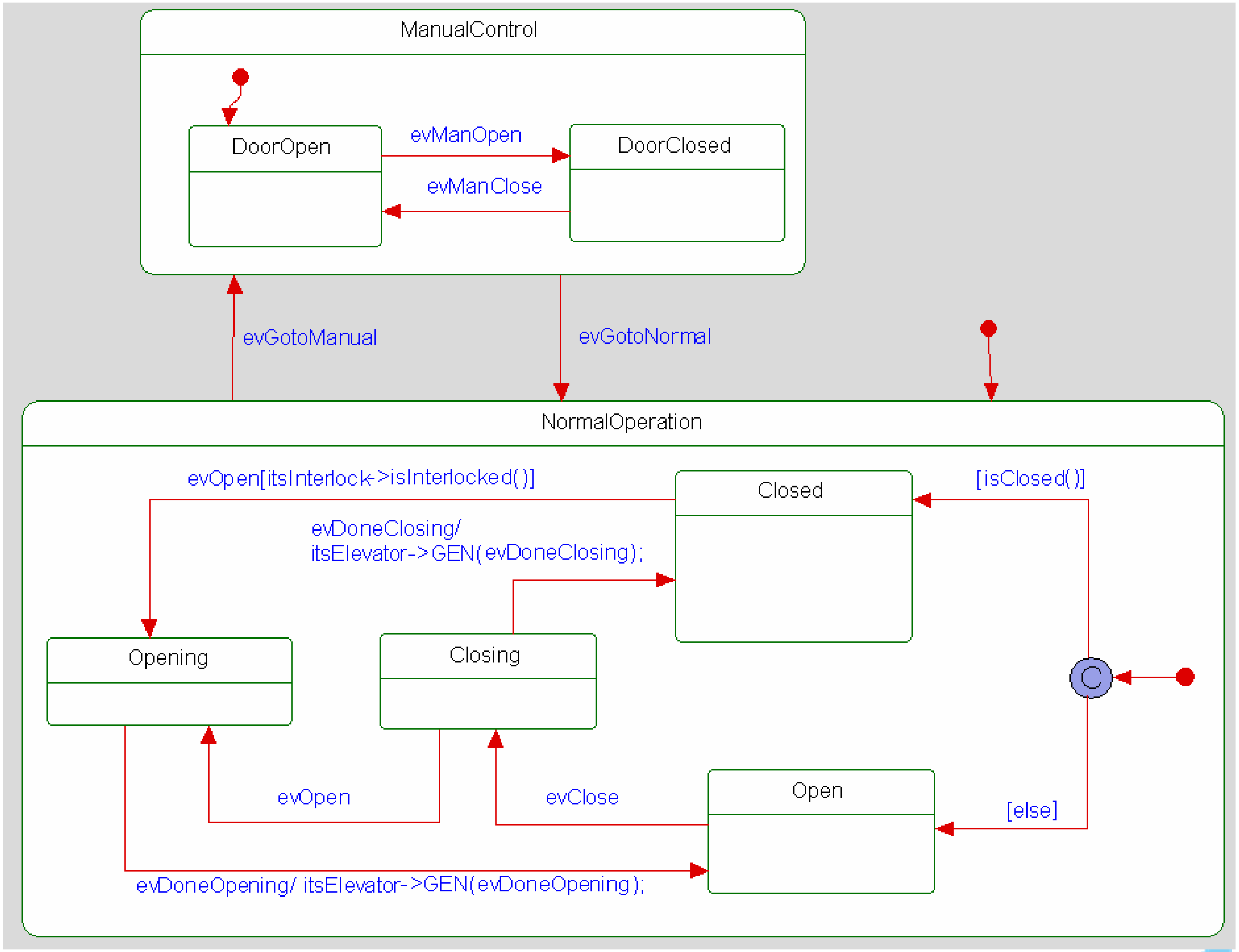
These connect electrically and mechanically only. The Interlock can detect if it's interlocked by monitoring electric contact points that mean it's available, and other contact points that indicate that it's interlocked.

Step 3: Closing the Doors



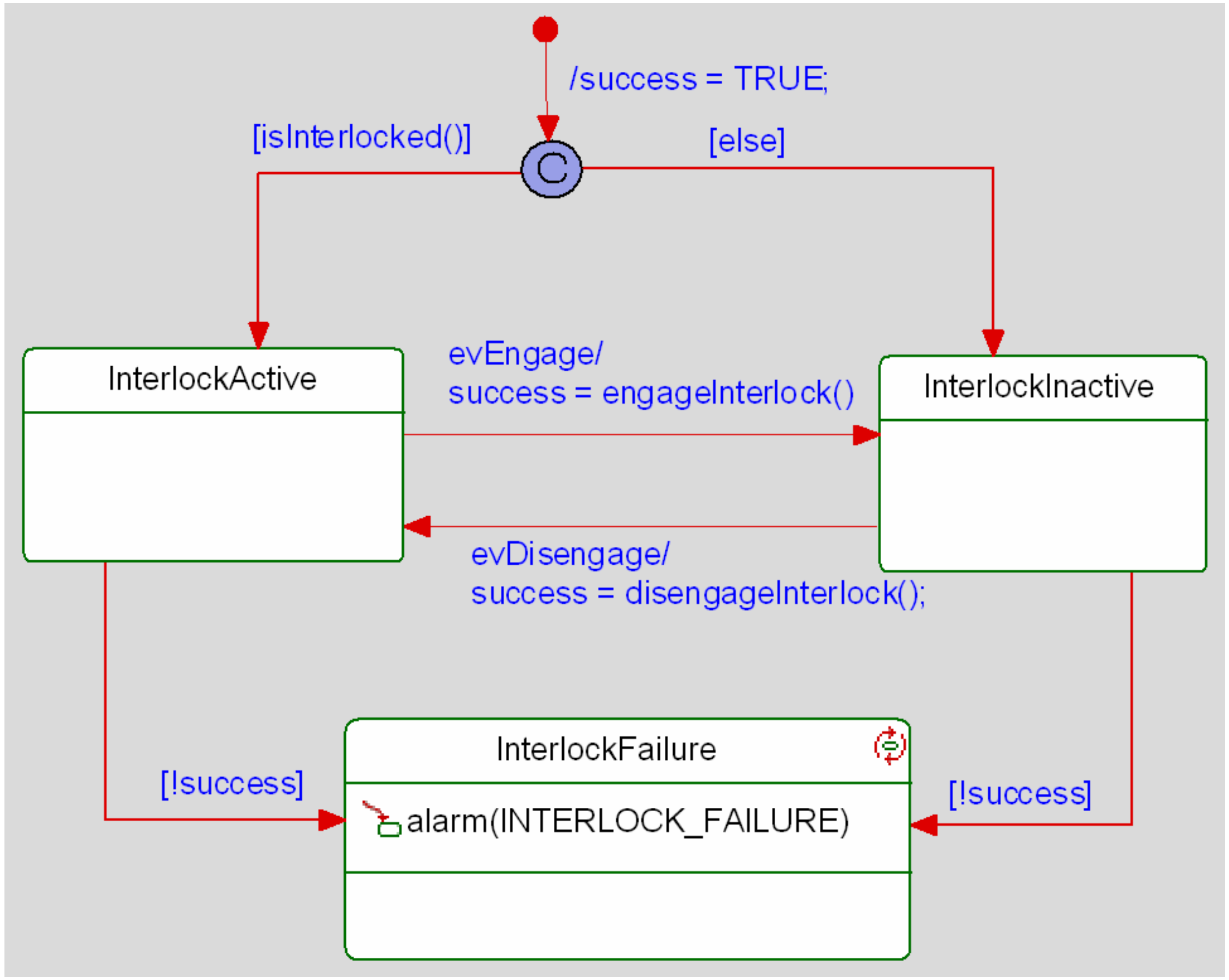
Elevator state machine

Step 3: Closing the Doors



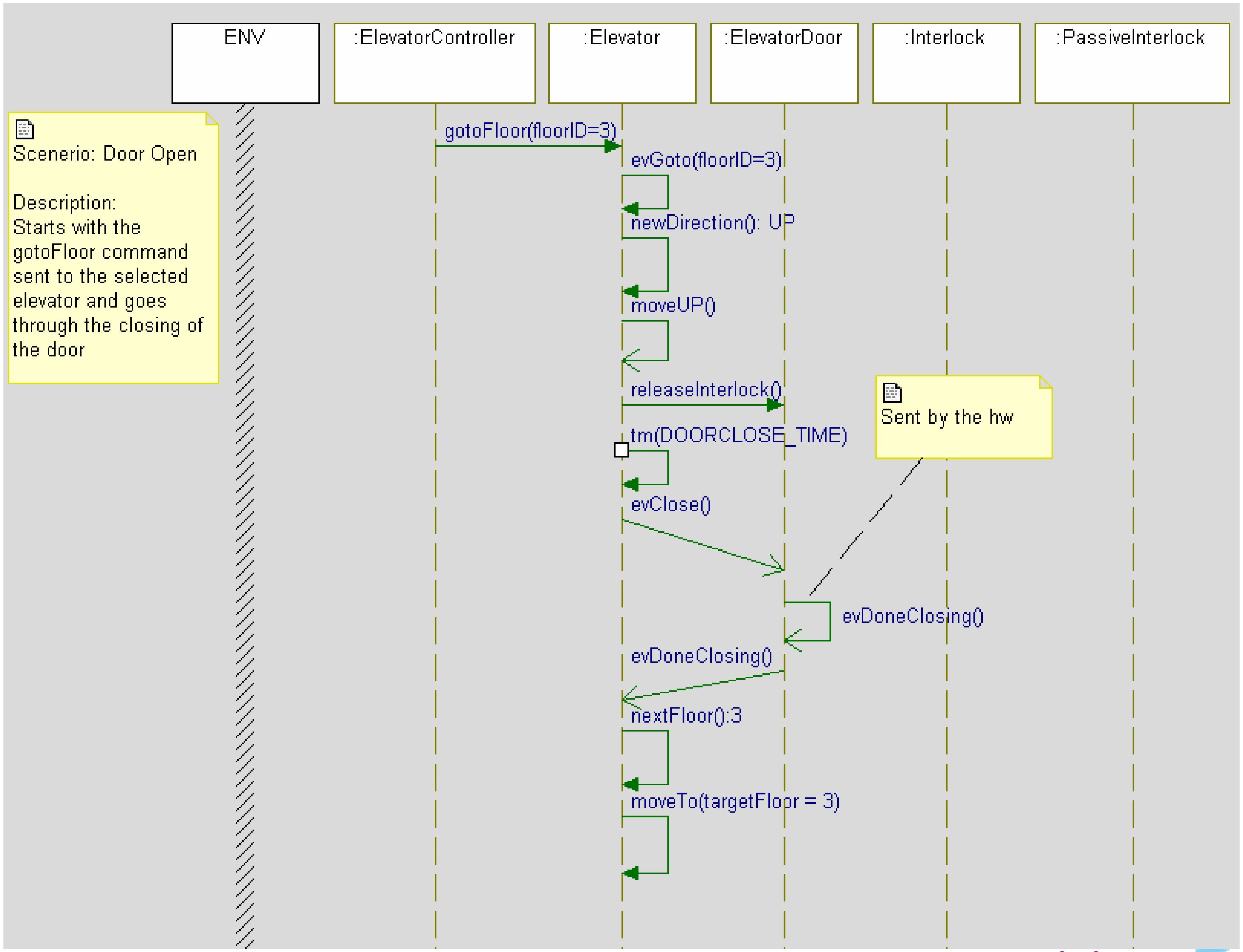
ElevatorDoor state machine

Step 3: Closing the Doors



Interlock state machine

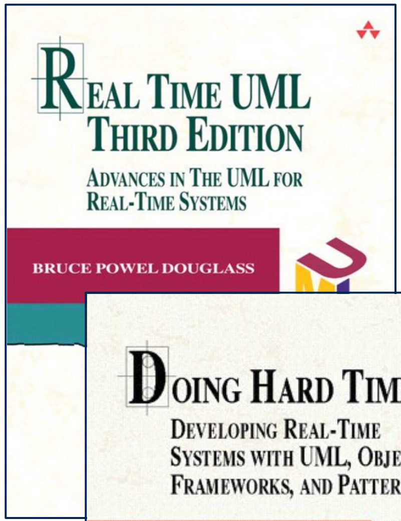
Step 3: Closing the Doors



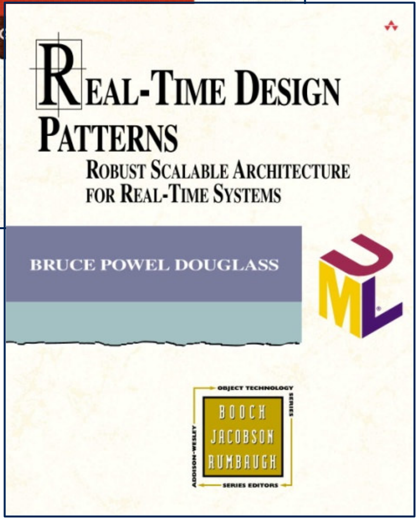
Summary

- Harmony for Embedded Realtime Development is a very 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 for Embedded Realtime Development 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

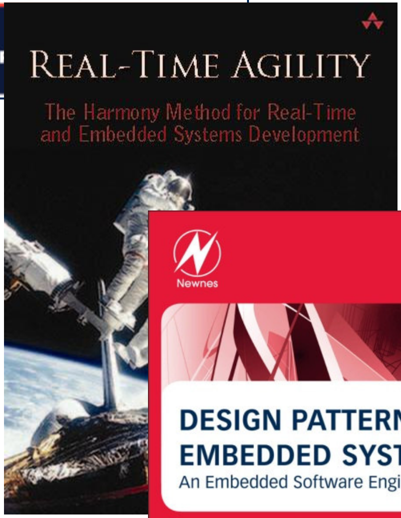
References

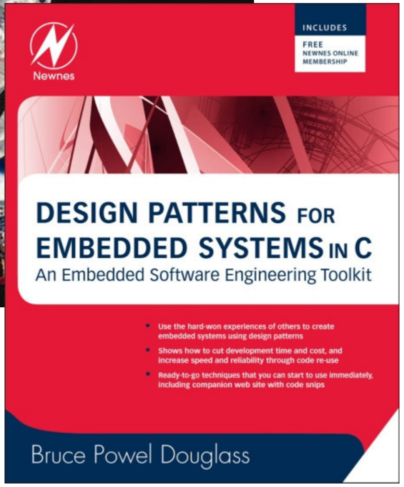

REAL TIME UML THIRD EDITION
ADVANCES IN THE UML FOR REAL-TIME SYSTEMS
BRUCE POWEL DOUGLASS

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

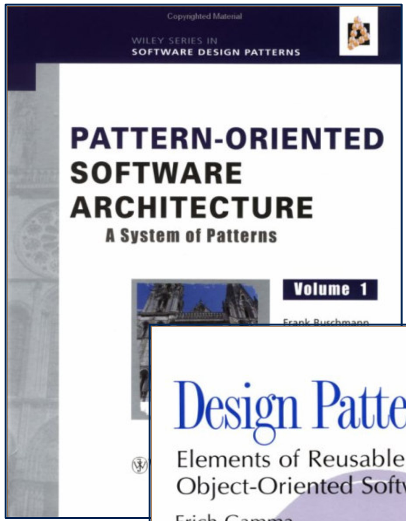

REAL-TIME DESIGN PATTERNS
ROBUST SCALABLE ARCHITECTURE FOR REAL-TIME SYSTEMS
BRUCE POWEL DOUGLASS
OBJECT TECHNOLOGY SERIES
BOOCH JACOBSON RUMBAUGH
SERIES EDITORS
ADDISON-WESLEY

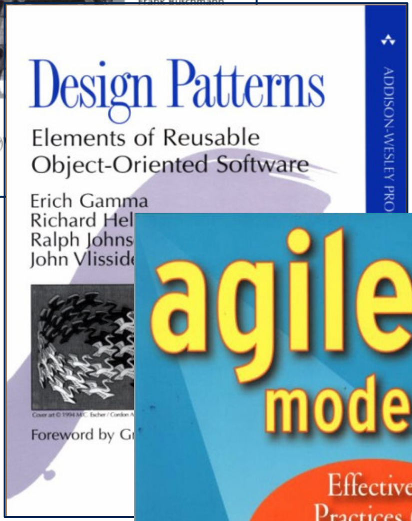
EMBEDDED TECHNOLOGY™ SERIES
Real-Time UML Workshop for Embedded Systems
CD-ROM INCLUDED
Demo version of the Rhapsody UML tool
Models of the solutions
Bruce

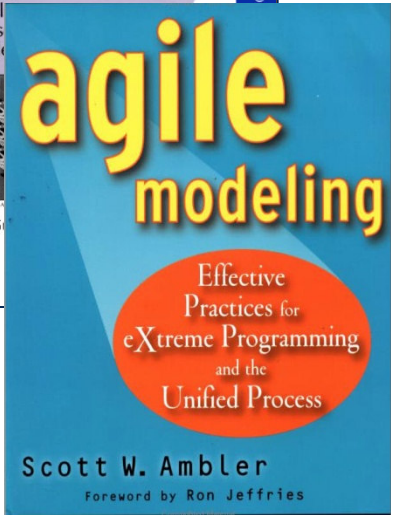

REAL-TIME AGILITY
The Harmony Method for Real-Time and Embedded Systems Development


Newnes
INCLUDES FREE NEWNES ONLINE MEMBERSHIP
DESIGN PATTERNS FOR EMBEDDED SYSTEMS IN C
An Embedded Software Engineering Toolkit
- Use the hard-won experiences of others to create embedded systems using design patterns
- Shows how to cut development time and cost, and increase speed and reliability through code re-use
- Ready-to-go techniques that you can start to use immediately, including companion web site with code snips

Bruce Powel Douglass


Copyrighted Material
WILEY SERIES IN SOFTWARE DESIGN PATTERNS
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
A System of Patterns
Volume 1
Frank Buschmann


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Hel
Ralph Johns
John Vlissid
Foreword by G
ADDISON-WESLEY PRO


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

www.ibm/software/rational