5
End-To-End Design Example

Distributed Embedded Systems
Philip Koopman
September 14, 2015

© Copyright 2010-2014, Philip Koopman
Where Are We Now?

◆ Where we’ve been:
  • General UML techniques

◆ Where we’re going today:
  • An end-to-end distributed system design example similar to course project
  • Importance of *traceability*

◆ Where we’re going next:
  • Distributed + Embedded
  • Design reviews & inspections
Example System: Soda Vending Machine

- **High Level Requirements:**
  Make it work like a real vending machine

- **Simplification:**
  - Sodas cost some number of quarters
  - All other coins are rejected (invisible to your control system)

- **Assume a Distributed System** per given class diagram
  - Processor for each button, coin return controller, vending controller
  - You get the message dictionary and most of the requirements specification (the “Architecture”)

- Complete worked out example available on course project web pages
General Approach  (Hybrid UML + Text)

**“Requirements”**
- Use cases (which are exemplary, but not necessarily coherent/definitive)
- System-level text requirements

**“Architecture”** (really just some parts of architecture)
- Class Diagrams – model “nouns” in system as classes & “architecture diagram”
- Define network variables that define architectural interfaces (message dictionary)
- Sensors, actuators, software objects

**Software Requirements**
- Scenarios – details inside use cases
- Sequence Diagrams

**Design**
- Detailed text behavioral requirements
- State Charts (state transitions)
- Test Design

**Implementation**
- Write the code
- Module testing

**Integration**
- Integration tests; acceptance tests
A Word On Traceability

- Traceability is checking to ensure that steps of the process fit together

- **Forward Traceability:**
  - Next step in process has everything in current step
  - “Nothing got left out”

- **Backward Traceability**
  - Previous step in process provoked everything in current step
  - “Nothing spurious included/no gold plating”

- **Lightweight traceability uses spreadsheets**
  - Examples in this talk
UML Use Cases For Requirements Development

- Actor is a person
- Actor initiates a Use Case
- Represents the system from the actor’s point of view
- Use cases are independent (“transactions”)
Adapting Use Cases For Distributed+Embedded

◆ Actors might not be people
  • Other computer systems can be actors
  • Sensors can be actor “proxies”
  • Timers, counters, monitors can be actors (e.g., close doors)

◆ Sometimes use cases form sequences
  • Example: can’t exit an elevator if you haven’t entered it
  • Shows up as preconditions for use case applicability
Solution: Use Cases

**Soda Machine**

- **U1.** Customer inserts a quarter
- **U2.** Customer pushes a soda button
- **U3.** Customer pushes coin return button
- **U4.** Observe soda availability

**Notes:**
- “Purchase a Soda” is not a use case – too complex
- Cooling, coin box full, other aspects ignored
System-Level Text Requirements

- **Goal: implement a soda vending machine**

  R1. Pushing a button **shall** vend a soda of the type corresponding to that button.
  R2. The machine **shall** permanently retain exactly SODACOST coins for each can of soda vended.
  R3. Coin return **shall** return all deposited coins since the last vend cycle.
  R4. The machine **shall** return all deposited money in excess of SODACOST coins before a vend cycle.
  R5. The machine **shall** flash the light for a selected item while vending is in progress to indicate acceptance of a selection to the buyer.
  R6. The machine **shall** illuminate the light for any out-of-stock item

- **Assume a Fully Distributed System**
  - Processor for each button, coin return controller, vending controller
## Traceability: UML and Text Requirements

- Put an “X” in every box with a related Use Case and Requirements

<table>
<thead>
<tr>
<th>Use Cases</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
</tr>
</thead>
<tbody>
<tr>
<td>U1. Customer inserts a quarter</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U2. Customer pushes a soda button</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U3. Customer pushes coin return button</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U4. Observe soda availability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
UML To Requirements Traceability Notes

- **Lack of backward traceability for R2**
  - There is a missing actor on the Use Case diagram – the soda delivery person
  - Could add “U5. Collect Money”
  - Possibly add “U6. Refill Machine”

- **Requirements must address off-nominal behaviors that are not apparent in use cases**
  - U1 – too many quarters inserted
  - U2 – soda button pressed without a quarter
  - U2 – two soda buttons pressed concurrently
  - U3 – coin return pressed with no money inserted

- **UML (as we are doing it) gradually eases from requirements to design**
  - Details of the use case become apparent as requirements are elaborated
  - Scenarios and sequence diagrams are partway between requirements and design
Revised Use Cases

Soda Machine

U1. Customer inserts a quarter

U2. Customer pushes a soda button

U3. Customer pushes coin return button

U4. Observe soda availability

U5. Collect Money

U6. Refill Machine

Customer

Vendor
## Revised Traceability: UML & Text Requirements

- Put an “X” in every box with a related Use Case and Requirements

<table>
<thead>
<tr>
<th>Use Cases</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
</tr>
</thead>
<tbody>
<tr>
<td>U1. Customer inserts a quarter</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U2. Customer pushes a soda button</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>U3. Customer pushes coin return button</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U4. Observe soda availability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>U5. Collect money</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U6. Refill machine</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Architecture

One definition of architecture is:
Architecture = Objects + Interfaces
Architecture: UML Class Diagrams

- Used to show system in terms of objects, attributes, and relationships
  - Objects are “nouns” in the system; Attributes are local state data within objects
  - Implicit, trivial controllers are assumed built in to uncontrolled components
    - (This is a simplified class diagram – VendMotor and VendPosition not there)
Architecture Diagram

This isn’t a terribly formal diagram, but it helps keep things straight
System Sensors

- **Button[s](v): Soda selection button** -- Physical state sensor
  - $v=\{\text{True, False}\}$.
  - One button per type of soda. All are False at initialization. $S$ is an integer $1..8$.
  - $\text{Button}[s](\text{True})$ is sent when button $s$ is depressed; $\text{Button}[s](\text{False})$ is sent when button $s$ is released.
  - The button sensors have a physical interlock that prevents more than one being pressed at a time.

- **Empty[s](v): Item empty sensor** -- Smart Sensor
  - $v=\{\text{True, False}\}$.
  - One empty sensor per type of soda vended. True when out of stock. $S$ is an integer $[1]..[8]$.
  - One per type of soda. Initialized to be False.
  - This is a smart sensor, so its implicit function is:
    transmit $\text{mEmpty}[s](v) = \text{Empty}[s](v)$
    (i.e., broadcast state to rest of system)
Environment-Only System State

- **SodaCount[s](n):** The number of sodas in each chute
  - Each count is set to 50 at startup

- **What does “environment-only” mean?**
  - We have a simulator in Java that simulates the entire system
    - Computing nodes
    - Network
    - Sensors & actuators
    - Physical world
  - The physical world model keeps track of how many sodas are in a chute
  - **The embedded computers do not know how many sodas are in the chute**
    - They only can infer it from sensors and build a model of the physical world
    - In this system, they only know if a chute is empty or not empty
    - In some other, fancier system the delivery person might program in number and the controllers could keep count – but they still wouldn’t “know” the actual value of SodaCount – they would be inferring it from external information.
System Actuators

- **ButtonLight[s](v):** Soda selection light.
  - $v=\{\text{True, False}\}$.
  - One per type of soda. When set to True turns on the light in the button for soda; when set to False turns that light off. $S$ is an integer 1..8.
  - All lights set to False at initialization.

- **Note:** *soda refill & money collection is done manually*

- **Note** – there are more sensors and actuators in the full example.
Software Control Objects

- **ButtonControl[s]**
  - One per soda selection (S is an integer \([1..8]\))
  - Controls button lights
  - Controls sending button selections to VendControl

- **CoinControl**
  - Controls coin return dispenser

- **VendControl**
  - Controls dispensing the soda cans

- **VendPositionControl**
  - Controls the movement of the VC
    (this is a mechanical device that moves across chutes to select a soda)
Message Dictionary

Notation:
- $s$ is button index number: $s=1..8$

Environmental Object Messages

These messages are sent by environmental objects and smart sensors provided in the system.

<table>
<thead>
<tr>
<th>Source Node Name</th>
<th>Message Name</th>
<th>Replication</th>
<th>Number of fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>mEmpty</td>
<td>$s$</td>
<td>1</td>
<td>See Object Description</td>
</tr>
<tr>
<td>Coin_Return</td>
<td>mCoinReturn</td>
<td>none</td>
<td>1</td>
<td>See Object Description</td>
</tr>
<tr>
<td>VendPosition</td>
<td>mVendPosition</td>
<td>$s$</td>
<td>1</td>
<td>See Object Description</td>
</tr>
</tbody>
</table>

Controller Messages

These messages are sent by the controllers that you will design. In the later projects, you will be allowed to modify the message dictionary for the controllers in a limited way, but for the time being, you must implement the message dictionary given below:

<table>
<thead>
<tr>
<th>Source Node Name</th>
<th>Message Name</th>
<th>Replication</th>
<th>Number of Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ButtonControl</td>
<td>mButton</td>
<td>$s$</td>
<td>1</td>
<td>State of the soda selection button</td>
</tr>
<tr>
<td>VendControl</td>
<td>mVend</td>
<td>none</td>
<td>1</td>
<td>True when vending a soda</td>
</tr>
<tr>
<td>CoinControl</td>
<td>mCoinCount</td>
<td>none</td>
<td>1</td>
<td>Integer number of coins received.</td>
</tr>
<tr>
<td>VendPositionControl</td>
<td>mVendMotor</td>
<td>none</td>
<td>1</td>
<td>State of the vend motor.</td>
</tr>
</tbody>
</table>
Software Requirements

- Structured representation of control objects
- Scenarios
- UML Sequence Diagrams
- A stylized detailed requirements template
2. ButtonControl[s]

◆ Replication:
  - There is one button controller per Button/Button_Light pair (8 total).

◆ Instantiation:
  - ButtonControl[s] commands Button_Light[s] to False at initialization.

◆ Assumptions:
  - Only one Button[s] is sent as True at a time to VendControl.
  - Each ButtonControl[s] has a physical interface to exactly one Button[s] and ButtonLight[s].

◆ Input Interface:
  - Button[s](v)
  - mEmpty[s](v)
  - mVend[s](v) (assume that any Vend message received indicates an actual vend event)

◆ Output Interface:
  - mButton[s](v)
  - ButtonLight[s](v)
Continued 2. ButtonControl[s]

- **Constants:**
  - FlashLimit (integer): determines the rate that the light flashes during vend.

- **State:**
  - IsEmpty (True, False); initialized to False; indicates when selection has no soda cans left.
  - ButtonState (True, False); initialized to False; indicates whether the button has been pressed.
  - FlashCounter: used to keep track of time while flashing the light during Vend

- **Constraints:**
  - None
Use Case 2: Customer pushes a soda button

**Scenario 2A:** Customer pushes a soda button when the correct amount has been paid

**Pre-Conditions:**
- The soda machine is not vending.
- No button is pressed.
- The system has received the correct number of coins for the cost of a soda since the last vend cycle.
- The VendCarriage is parked in front of chute r, r < s.

**Scenario:**
1. The Customer pushes soda button s.
2. The light on the soda button s begins flashing.
3. The VendPositionControl aligns with soda chute s.
4. The soda is vended.
5. The light on the soda button s stops flashing.

**Post-Conditions:**
- The system retains the cost of the soda and has one less soda of type s
- The system is out of soda of type s
These fonts are too small – don’t do this on your presentations!
This is better, but you still need to zoom in to see things
Sequence Diagram 2A:

1a. Button s pressed
1b. Button[s](true)
2a. ButtonLight[s](true)
2b. ButtonLight[s](false)

2a and 2b repeat until 5a.

1c. mButton[s](true)
1d. mCoinCount(2)

3b. mVendMotor(RIGHT)
3f. mVendMotor(STOP)

4a. Vend(true)
4b. mVend(true)
4c. Soda Vended
4d. Vend(false)
4e. mVend(false)

5a. ButtonLight[s](false)
5b. mButton[s](false)
Critique of Preceding Sequence Diagram

◆ Pro: Everything is there
  • You can see all the components of the system interacting

◆ Con: It is complex
  • If it is difficult to show in powerpoint, it is difficult to understand
    (the “Powerpoint Engineering” principle)
  • It is a very specific case (e.g., what if it wasn’t the last soda?)

◆ Possible ways to improve
  • Break it up vertically into multiple steps
  • Break it up vertically by not showing every piece interacting
  • There is no perfect, “best” way to do this – these are just ideas

◆ Project grading note
  • Not graded on whether your SDs are complex or simple or “best”
  • You are graded on whether your SDs trace properly
  • You are graded on whether the final project passes acceptance tests
How Many Sequence Diagrams?

◆ Examples:
  • Scenario 2A: Customer pushes a soda button when the correct amount has been paid
  • Scenario 2B: Customer pushes a soda button when the correct amount has NOT been paid

  ...

  • Scenario 1C: Customer pushes a soda button, holds it, and then deposits a coin
    – This is a combination of Use Case 1 & Use Case 2 – no clean distinction

◆ Most Use Cases have more than one scenario for use
  • And therefore more than one sequence diagram

◆ Keep making scenarios until you cover all the functions that matter
  • There is no single right way to do it …
    … but in general, simpler and fewer scenarios are better than many complex ones
Sequence Diagram Traceability

◆ Sequence Diagrams to Use Cases
  • Is there at least one sequence diagram for each Use Case number?
  • If so, you’ve satisfied traceability

◆ Sequence Diagrams to objects
  • Are all objects in at least one sequence diagram?

◆ Sequence Diagrams to messages
  • Are all messages in at least one sequence diagram?

◆ Traceability doesn’t prove you have everything; but it helps you avoid “stupid” mistake gaps
  • For example, if there were no scenario 4A, then Use Case 4 isn’t covered
Design

- “Design requirements” – has proven to be a useful step
- UML Statecharts
Two Step Design Process

(Attempts to reduce the size of the “miracle” in that process step)

1. **Write down constraints & behaviors**
   - Constraints are assumptions that other components can make
   - Behaviors are functions designed in to the component

2. **Synthesize a statechart**
   - Transitions have to account for all behavior triggers
   - Transitions have to account for all behaviors
     (alternately, states could account for all behaviors; depends on approach)

◆ **Alternate Approaches**
   - Tools can synthesize statecharts from *a complete* set of sequence diagrams
   - People can do that too, even if sequence diagrams are incomplete
Formula for Event-Driven Systems

◆ Behavioral Requirements:

- `<ID> <message received>` shall result in `<message transmitted>` …
  and/or `<variable value assigned>` …

- **OR**

- `<message received>` and `<variable value test(s)>` shall result in `<message transmitted>` …
  and/or `<variable value assigned>` …

- Account for all possible messages received; OK to restrict by value
  – E.g., `<message received>` with value V shall result in …
- Account for all possible messages that need to be transmitted outbound
- Make sure all variables are set as required in right hand sides
- **EXACTLY ONE** received message per requirement (network serializes messages; simultaneous reception of multiple messages is impossible)
- OK to have: multiple messages transmitted; multiple variables assigned
**Sequence Diagram To Behavioral Requirements**

- **For each object in system**
  - Consider every sequence diagram
  - Create rules that explain behaviors of arcs *for that object*

- **What matters for an object?**
  - All ovals with conditions/assignments
  - All arrows exiting the object
  - All arrows entering the object
**ButtonControl[s] Event Triggered Requirements**

- **ER2.1.** If `mEmpty[s]` is received as `v`, then `IsEmpty` shall be set to `v`.
- **ER2.2.** If `mEmpty[s]` is received True and `ButtonState ← False`, then
  - ER2.2a. `ButtonLight[s](v)` shall be commanded to False.
  - ER2.2b. `mButton[s]` shall be set to False.
- **ER2.3.** If `mEmpty[s]` is received False and `ButtonState ← False`, then
  - ER2.3a. `ButtonLight[s](v)` shall be commanded to True.
  - ER2.3b. `mButton[s]` shall be set to False.
- **ER 2.4.** If `Button[s]` is received True and `IsEmpty` is False, then
  - ER2.4a. `ButtonState` shall be set to True.
  - ER2.4b. `ButtonLight[s]` shall be commanded to blink with a period of 0.25s.
  - ER2.4c. `mButton[s]` shall be set to True.
- **ER 2.5.** If `mVend[s]` is received True and `IsEmpty` is False, `ButtonLight` shall be commanded to True.
- **ER 2.6.** If `mVend[s]` is received True and `IsEmpty` is True, `ButtonLight` shall be commanded to False.
- **ER 2.7.** If `mVend[s]` is received True, then
  - ER 2.7a `mButton[s]` shall be set to False.
  - ER 2.7b `ButtonState` shall be set to False.
Statechart Design

- We now have a (we think) complete behavioral requirements specification
  - Really you can just call these “behaviors”, but we use the word requirements to remind you that “shall” and “should” are mandatory words.

- Design Statecharts for each software object
  - Design states for each object
  - Behavior requirements become conditions for state transitions
  - Cover every behavior requirement in state chart

- Traceability
  - Every behavior requirement should map to a state transition arc

- Note: we’re not covering control loop design with these
  - Statecharts sometimes implement sequential logic
  - But, sometimes they cause mode transitions for control loops
ButtonControl Time Triggered Statechart

<table>
<thead>
<tr>
<th>Transition #</th>
<th>Guard</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2.1</td>
<td>mButton[s] ← True AND mEmpty[s] ← False</td>
</tr>
<tr>
<td>T2.2</td>
<td>mVend ← True AND mEmpty[s] ← False</td>
</tr>
<tr>
<td>T2.3</td>
<td>mVend ← True AND mEmpty[s] ← True</td>
</tr>
<tr>
<td>T2.4</td>
<td>FlashCounter &gt; FlashLimit</td>
</tr>
<tr>
<td>T2.5</td>
<td>FlashCounterLimit ← 0</td>
</tr>
<tr>
<td>T2.6</td>
<td>mEmpty ← True</td>
</tr>
<tr>
<td>T2.7</td>
<td>mEmpty ← False</td>
</tr>
</tbody>
</table>

Important – show guard conditions with statechart diagram!

Use 12 point+ font
Event Triggered vs. Time Triggered?

- **Event triggered**
  - Exactly one message on left hand side of “shall”
  - Each message arrival is an “event” which triggers a statechart transition
  - Networks deliver only one message at a time, so that’s the way it is
  - “Asynchronous state machines” from hardware design

- **Time triggered**
  - Arriving message values put into memory buffers
  - State chart transitions based on most recent message value
  - “Synchronous state machines” from hardware design

- **Project sequence is**
  - Event triggered project 3
  - Convert to time triggered project 4
  - Why? Because every time we skipped event triggered half the class got lost
    - Once you see it, it’s not too bad, but it is not easy to “see” if you skip this step
    - You’ll see more about this as we go
## Traceability

- Does every requirement map to at least one state or transition?
- Does every state or transition map to at least one requirement?

### Requirements-to-Statecharts Traceability

<table>
<thead>
<tr>
<th>States</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R2.1</td>
</tr>
<tr>
<td>IDLE</td>
<td>x</td>
</tr>
<tr>
<td>EMPTY</td>
<td>x</td>
</tr>
<tr>
<td>VEND</td>
<td>x</td>
</tr>
<tr>
<td>FLASH_OFF</td>
<td>x</td>
</tr>
<tr>
<td>FLASH_ON</td>
<td>x</td>
</tr>
<tr>
<td>Transitions</td>
<td></td>
</tr>
<tr>
<td>T2.1</td>
<td></td>
</tr>
<tr>
<td>T2.2</td>
<td></td>
</tr>
<tr>
<td>T2.3</td>
<td></td>
</tr>
<tr>
<td>T2.4</td>
<td></td>
</tr>
<tr>
<td>T2.5</td>
<td></td>
</tr>
<tr>
<td>T2.6</td>
<td></td>
</tr>
<tr>
<td>T2.7</td>
<td></td>
</tr>
</tbody>
</table>
class CoinOutControl {
    state = No_Money_Inserted;
    COUNTER = 0;

    public void msgReceived(msg M) {
        switch state { // make transitions
            case No_Money_Inserted:
                if (M == Coin_in.TRUE) state = One_Quarter_Inserted; //*** Transition S2.a1
                break;
            case Coin_Inserted:
                if (M == Coin_in.TRUE) sendMsg(Coin_out.TRUE); //*** Transition S2.a4
                else if (M == Vend[s].TRUE) state = No_Money_Inserted; //*** Transition S2.a2
                else if (M == Coin_return.TRUE) { //*** Transition S2.a3
                    sendMsg(Coin_out.TRUE);
                    state = No_Money_Inserted;
                }
                break;
            default: Error condition
        }

        switch state { // behavior in state
            case No_Money_Inserted: //*** State S2.s1
                COUNTER = 0; break;
            case One_Quarter_Inserted: //*** State S2.s2
                COUNTER = 1; break;
            default: Unknown state
        }
    }

    Note traceability of code to statechart
    This is code from an older example
Discrete Event Simulator

- Everything is an “event”
  - Framework events wait until their time to execute, then generate other events
  - Message events only differ in that they go through a network delay model
  - Note that the event queue is sorted by time – earliest event runs next
    - In case of a time value tie, order is arbitrary (and may be randomized)
Traceability of Statecharts

◆ Sequence Diagram Arcs trace directly to statechart arcs
  • An arrow coming into an object can cause a state transition
  • That traces to changing the state variable value in the code

◆ Behaviors trace to statechart arcs too
  • This is why text behaviors are skipped by some designers
  • But we’ve informally found they reduce errors

◆ Statecharts are more “complete” than most sequence diagrams
  • Statecharts have to account for all transitions to actually work
  • Extra transitions might be necessary in design

  • Advice for non-traced arcs & states is either:
    – Invent new sequence diagrams to cover all arcs in statecharts
    OR
    – Be very careful to test non-traceable arcs to avoid undesired side effects
Statechart Construction Rules

- **Statechart transition conditions/arcs shall contain**
  - Guard conditions *only*!
  - *No actions on transitions*
  - In hardware, this would make them Moore FSMs

- **Even though actions on arcs are allowed by UML…**
  - This makes it easier to obtain clean time triggered design
  - It makes the code itself have a much cleaner structure
  - In the long run it reduces number of bugs

- **If you feel you must execute an action on a transition…**
  - Use an intermediate state instead
  - Usually a state with an action and one always-true exit arc
Concurrent Statecharts

- OK to have two or more statecharts executing in parallel
- Parallel statecharts shall not write to the same outputs or state variables
Nested Statecharts

- **Avoid using them!**
  - Difficult to implement in code
  - Requires multiple, nested switch statements

- **If you *must* use them**
  - You may *not* transition in to or out of the superstate from an inner state
States vs. State Variables

- State variables are appropriate for:
  - Integers (counters, floors numbers, etc)

- NOT suitable for:
  - Boolean flags (doorIsClosed)
  - Boolean flags should show up as states, not variables

- Statechart for door should represent the state of the door, not the state of the door motor
Test Design

- Testing statecharts
- Acceptance tests
- (A full description of testing would be an entire tutorial)
Test Design

◆ Suggestion: design tests before actual implementation
  • May uncover errors in your design before coding

◆ Test at least two levels before you run a full simulation
  • Unit/module tests
  • System integration tests

◆ Unit Tests
  • Design tests to cover every state transition in every state chart
  • Make sure erroneous state transitions aren’t taken
  • Cover every possible message/event received by each object

◆ Traceability
  • Document traceability between tests and state transitions for unit tests

◆ System Integration Tests
  • Test specified operation sequences / UML scenarios
Idea Behind Unit Test

- Isolate single module and feed it direct inputs
  - Feed in inputs that exercise the internal state machine
  - Base tests on single sequence diagrams

  ![Test Input Diagram]

  - Monitor state machine values and outputs for correctness

- Can also design tests based on looking at statechart
  - Make sure you cover all arcs and enter all states
Idea Behind Integration Test

- Run all modules in a Sequence Diagram except selected inputs
  - Artificially set up state information to meet preconditions
  - Feed primary inputs from test harness; let rest of arcs run on their own
  - Make sure other arcs perform as expected

Sequence Diagram 1B:

Test Input
Acceptance Test

◆ **Ensure system as a whole actually meets requirements**
  - In simple systems, testing all scenarios suffices
  - In real systems, need to test sequences of Use Cases

◆ **First define meaningful sequences of use cases**
  - Example: insert coin, push soda button
  - Example: insert coin, push coin return, push soda button

◆ **Next, execute tests and compare results to system requirements**
  - Generate many simulated customers and see what happens
  - Were each of R1 - R6 met during the course of each test?

◆ **Additional test strategies:**
  - Design tests to attempt requirement failure
  - Reset system partway through a scenario or between use cases
  - …
Traceability of Tests

- **Trace Unit Tests to statecharts**
  - All states & arcs in statecharts covered by a test
  - Probably want additional tests … simple coverage is just a starting point
  - Be careful about variable values since variables store “state” beyond FSM

- **Trace Integration Tests to sequence diagrams**
  - Every sequence diagram should be covered by a test
  - Probably want additional tests, especially for undocumented off-nominal situations

- **Trace acceptance tests to:**
  - Marketing requirements – that is the whole point of acceptance tests, especially testing all use cases
  - *and if possible:*
    - Engineering requirements – should have high coverage
    - Sequence diagrams – all nominal and some off-nominal should be covered
Review: General Approach  (Hybrid UML + Text)

◆ “Requirements”
  • Use cases (which are exemplary, but not necessarily coherent/definitive)
  • System-level text requirements
◆ “Architecture” (really just some parts of architecture)
  • Class Diagrams – model “nouns” in system as classes & “architecture diagram”
  • Define network variables that define architectural interfaces (message dictionary)
  • Sensors, actuators, software objects
◆ Software Requirements
  • Scenarios – details inside use cases
  • Sequence Diagrams
◆ Design
  • Detailed text behavioral requirements
  • State Charts (state transitions)
  • Test Design
◆ Implementation
  • Write the code
  • Module testing (unit tests)
◆ Integration
  • Integration tests; acceptance tests