Integration and System Testing

Chapter 12

Levels of Testing
Traditional View of Testing Levels

- Unit vs. integration vs. system testing
- The *Waterfall Model* of software development
  - Information produced in one of the development phases constitutes the basis for test case identification at that level
  - System test cases are related to the requirements specification, while unit testing are derived from the detailed design
  - Functional testing is applied in this model, especially at the system testing level.
  - ‘Bottom-up’ testing order is followed
    - Test the individual components, then integrate these into subsystems until the entire system is tested

The Waterfall Lifecycle
Traditional View of Testing Levels (cont’d)

- The structural testing techniques we studied so far & all types of BVA, equivalence classes and decision table functional testing techniques are best applied to unit testing

- Most of the discussion on Integration testing focuses on the order in which units are integrated: top-down, bottom-up or the ‘big bang’ approach (everything at once)

- The Waterfall Model is associated with top-down development and design by functional decomposition
  - The end result of preliminary design is a functional decomposition of the entire system into a tree-like structure of functional components.

A partial functional decomposition of the ATM system
Types of Integration Testing

Types of Integration Testing:

<table>
<thead>
<tr>
<th>Top-Down</th>
<th>Bottom-Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begins with the main program (tree root)</td>
<td>Start with the leaves of the decomposition tree</td>
</tr>
<tr>
<td>Lower level units called by the main subsystem as stubs: emulate a called unit</td>
<td>Driver modules emulate units at next level up in a tree</td>
</tr>
<tr>
<td>There is one stub for each child node</td>
<td>Not as many drivers, for many called units there could be one parent</td>
</tr>
<tr>
<td>Test main as a stand-alone</td>
<td>Test leaves individually</td>
</tr>
<tr>
<td>When convinced main is correct, replace stubs with actual code and start testing the next level down</td>
<td>When convinced lowest levels are correct, replace drivers with actual parent code, start testing upper level</td>
</tr>
</tbody>
</table>

- With the partial functional decomposition of the ATM system on the previous page, top-down integration would begin with the main program (ATM System) checking the calls to the three next level procedures (Terminal I/O, Manage Sessions, Conduct Transactions)

- Following the tree, the Manage Sessions procedure would be tested and then its children

- In each case, the actual code for the lower level units is replaced by a stub, which is a throw away piece of code that takes the place of the actual code

- The Big Bang approach simply puts all the units together at once, with no stubs or drivers

- Whichever approach is taken, the goal of traditional integration testing is to integrate previously tested units wrt the functional decomposition tree
The Waterfall Model & Alternative Life Cycle Models

• One of the weaknesses of the Waterfall Model is that it relies on functional *decomposition*, which in turn requires that we completely understand the whole system before actual implementation, which is not the case in most real-life applications
  – Requirements specification is gathered, put into a design document, reviewed, corrected, then after a series of iterations between specifications and design, coding starts
  – Sometimes when the end users see the product running, they indicate that this is not what they had expected and wanted!
  – Sometimes new requirements come up when the system runs

• *Composition* on the other hand is closer to the way people work
  – Start with something known and understood, then add to it gradually and maybe remove undesired portions

• The alternative models rely on composition as opposed to decomposition in the Waterfall Model

12.5
Alternative Life Cycle Models

• There are three derivatives of the waterfall model, each involves a series of increments or builds:
  – Incremental development
  – Evolutionary development
  – Spiral Model

• Within a build
  – the normal waterfall phases from detailed design through integration testing occur
  – System testing is split into
    • Regression testing
    • Progression testing

The Waterfall Lifecycle

• The goal of regression testing is to assure that things that worked correctly in the previous build still work with the newly added code and that there are no side effects

• Progression testing assumes that regression testing was successful and that the new functionality can be tested
The Three Spin-off Models
of the Waterfall Model

• The three alternative models differ in how the builds are identified

  – In Incremental Development:
    • The whole scope of the software is known. Break it into manageable increments. New functionality is constantly added onto the existing code base. Each new build is created by developing new code, unit testing the new code, testing the integration of the new code with the existing code, and testing the entire new system with both regression and progression testing.

  – In Evolutionary Development:
    • The whole scope of the software is not known. We allow for changes in the requirements and specifications as the software is developed. Thus, user feedback can be incorporated by directing future builds. The system evolves to meet the changing needs of the user

  – In the Spiral Model
    • A build is defined first in terms of rapid prototyping and then is subjected to a go/no-go decision based on technology-related risk factors

  – In all the three spin-off models
    • a build is a set of deliverable end-user functionality, thus all three yield earlier synthesis
    • Also earlier customer feedback
Specification Based Models

- Used when systems are not fully understood (by the user or developer) in the earlier phases

- The rapid prototyping life cycle deals with this by drastically reducing the specification-to-customer feedback loop to produce very early synthesis
  - Build a ‘quick and dirty’ prototype
  - Get customer feedback
  - Prototype again accordingly, till the user and developer agree on what the system should look like and do
  - Build the system according to the final correct specification

- How can we do system testing with rapid prototyping?
  - Use the prototyping cycle(s) as information gathering activities & produce a requirements specification
  - Or, capture what the user does with the prototype(s), define these as important scenarios, and use them as test cases

Rapid Prototyping Life Cycle
Formulations of the SATM System

- The Simple Automatic Teller Machine (SATM) system is built around 15 screens:

  - **Screen 1**: Welcome. Please Insert your ATM card for service.
  - **Screen 2**: Enter your Personal Identification Number — — — — Press Cancel if Error.
  - **Screen 3**: Your Personal Identification Number is incorrect. Please try again.
  - **Screen 4**: Invalid identification. Your card will be retained. Please call the bank.
  - **Screen 5**: Select transaction type: balance deposit withdrawal Press Cancel if Error.
  - **Screen 6**: Select account type: checking savings Press Cancel if Error.
  - **Screen 7**: Enter amount. Withdrawals must be in increments of $10 Press Cancel if Error.
  - **Screen 8**: Insufficient funds. Please enter a new amount. Press Cancel if Error.
  - **Screen 9**: Machine cannot dispense that amount. Please try again.
  - **Screen 10**: Temporarily unable to process withdrawals. Another transaction? yes no
  - **Screen 11**: Your balance is being updated. Please take cash from dispenser.
  - **Screen 12**: Temporarily unable to process deposits. Another transaction? yes no
  - **Screen 13**: Please put envelope into deposit slot. Your balance will be updated Press Cancel if Error.
  - **Screen 14**: Your new balance is printed on your receipt. Another transaction? yes no
  - **Screen 15**: Please take your receipt and ATM card. Thank you.
The SATM System

The SATM Terminal

Context Diagram of the SATM System
The SATM System (cont’d)

Entity/Relationship (E/R) Model of the SATM System

- Dataflow and E/R diagrams contain structural information
- Test cases are concerned with behavior
- Functional and data information are linked by a control model: A Finite State Machine (FSM) is an example
- Control models represent the point at which structure and behavior intersect
The SATM System (cont’d)

PIN Entry FSM

- If we only use structure charts to guide integration testing, we miss the fact that some (typically lower level) functions are used in more than one place.

- To support this we need either a detailed decomposition chart or a numbered decomposition
Part of a Detailed Decomposition Chart of the SATM System and a Numbered Decomposition

SATM System
1.1 Device Sense & Control
   1.1.1 Door Sense & Control
      1.1.1.1 Get Door Status
      1.1.1.2 Control Door
      1.1.1.3 Dispense Cash
   1.1.2 Slot Sense & Control
      1.1.2.1 WatchCardSlot
      1.1.2.2 Get Deposit Slot Status
      1.1.2.3 Control Card Roller
      1.1.2.3 Control Envelope Roller
      1.1.2.5 Read Card Strip
1.2 Central Bank Comm.
   1.2.1 Get PIN for PAN
   1.2.2 Get Account Status
   1.2.3 Post Daily Transactions
1.3 Terminal Sense & Control
   1.3.1 Screen Driver
   1.3.2 Key Sensor

1.4 Manage Session
   1.4.1 Validate Card
   1.4.2 Validate PIN
      1.4.2.1 Get PIN
      1.4.2.1 Get Digit
   1.4.3 Close Session
      1.4.3.1 New Transaction Request
      1.4.3.2 Print Receipt
      1.4.3.3 Post Transaction Local
   1.4.4 Manage Transaction
      1.4.4.1 Get Transaction Type
      1.4.4.2 Get Account Type
      1.4.4.3 Report Balance
      1.4.4.4 Process Deposit
      1.4.4.5 Process Withdrawal
Program Design Language

- As part of the specification and design process, each functional component is normally expanded to show its inputs, outputs and mechanism.

- We can do this using program design language (PDL) or pseudo-code

- If we follow the pseudo-code developed, we can identify the “uses” relationship among modules in the functional decomposition. This helps in integration testing
Threads

• Definition

  – A Thread:
    • Is a construct that refers to execution time behavior; when we test a system, we use test cases to select & execute threads
    • Is a path through the executable parts of a program

  – There are levels of threads:
    • System threads describe system level behavior
    • Integration threads correspond to integration level behavior
    • Unit threads correspond to unit level behavior

• Example:

  • a thread could be a path through a finite state machine description of a system
  • A thread could be a sequence of: source statements or machine instructions
Separating Integration & System Testing

Structural insights

- **Integration testing:**
  - is at a more detailed level than system testing
  - assumes that the units have been separately tested and that taken by themselves, the units function correctly
  - is concerned with the interfaces among the units
  - is concerned with preliminary design information (the ‘how’)

- **System testing:**
  - is at the level of the requirements specification (the ‘what’)

- **Examples from the SATM system:**
  - System testing:
    - Should make sure all 15 screens have been generated
    - From the E/R model, the 1:1 & 1:m relationships help us understand how much testing must be done
    - From the control model (FSMs), we can generate system test cases from paths through the FSMs
    - Examples:
      » Insertion of an invalid card (shortest system thread)
      » Insertion of a valid card, followed by 3 bad PIN entries
      » Insertion of a valid card, valid PIN, balance inquiry
      » Insertion of a valid card, valid PIN, deposit
      » Insertion of a valid card, valid PIN, withdrawal
      » Insertion of a valid card, valid PIN, withdraw more than the total account balance
  - Integration testing examples:
    - From the PDL:
    - interaction between ValidatePIN & GetPIN
      - How the result of GetPin affects ValidatePin:
        » Digit entered vs. Cancel key
    - Interaction between GetPIN & the KeySensor
Separating Integration & System Testing
Behavioral insights

• System testing:
  – Involves testing the entire system at a high level once all the modules have been integrated.
  – Can be performed in terms of port inputs (keys, buttons pressed, etc.) and port outputs (screens, printouts, etc.) or in terms of high level events occurring (certain inputs or all outputs occur, etc.)

• Integration level threads:
  – can be thought of as sequences of unit level threads. We are concerned with the interaction among those threads, and not their internals

• System level threads:
  – Can be interpreted as sequences of integration level threads