The Challenge of Teaching Software Testing Earlier into Design Courses

Ellen Francine Barbosa ¹
Richard LeBlanc ²
Mark Guzdial ²
José Carlos Maldonado ¹

¹ University of São Paulo (Brazil)
² College of Computing, Georgia Institute of Technology (USA)
Agenda

- Introduction
- CS2340 Course
  - Goals and audience
  - Prerequisites
  - Development process
- Testing Approach for CS2340 Course
  - Equivalence Partitioning Criterion
- Students’ Evaluation
  - Testing early
  - Using testing criteria
  - Writing test plans
- Conclusions and Further Work
Introduction

Teaching Software Testing…

Software Testing has traditionally been taught according to the classical approaches to software development.

- Waterfall Model
  - Analysis, Design, Coding, **Testing**, Code Release

- Students’ perception about Software Testing
  - Not a popular discipline
    - They have to do to show that their programming assignments work as required

**Something that happens at the end of the development process.**
Introduction

◆ Teaching Software Testing…
  ➢ Different approach

  Introduce testing practices during all phases of the development process, starting from analysis and going through design and implementation phases.

  ➢ eXtreme Programming (XP)
    » Write tests even before writing code
    » Development setting
      ✓ Enhance the system’s quality
      ✓ Reduce development costs

  “Test first, code later”.
Introduction

Objective

- Investigate different learning opportunities
- Provide extra motivation
  - Ideas for modern methodologies (like XP)
- CS2340 Course
  - Encourage the students on thinking about Software Testing earlier in their projects
    ✓ Provide a more pragmatic view of testing
  - Evaluate the students’ attitude

Explore the impact of introducing testing practices as early as possible, throughout all phases of the development process.
CS2340 – Objects and Design

- **Main Goal**
  
  > Explore higher-level issues of analysis and design, with some emphasis on user interface development.

- **Object-oriented paradigm**

- **Audience**
  
  > Second-year undergraduate students
  
  » Fall 2002: 160 students

- **Structure**
  
  > One semester
  
  » 2 lecture classes per week
  
  » 2 exams, 1 development project
CS2340 – *Objects and Design*

- **Pre-requisites**
  - *Introduction to Computing*
    - Design, construction and analysis of algorithms
  - *Object-Oriented Programming*
    - Java
  - *Languages and Translation*
    - Issues of language implementation, tokenizing and parsing
  - *Software Practicum* (sometimes concurrently)
    - Basics of Software Engineering
      - General view on Software Testing
Projects

- Team-oriented
- Development of a small-to-medium sized OO system
- Fall 2002: Genealogy Information
  » Create a system to...
    ✓ Collect genealogical information
    ✓ Note inconsistencies and flaws in the database
    ✓ Provide some graphical representations
    ✓ Support a standard genealogical information format
CS2340 – *Objects and Design*

- **Development Process**
  - Traditional phases of the object-oriented development
    - **OO Analysis**
      - Development of scenarios and CRC (Class-Responsibility-Collaborator) cards
    - **OO Design**
      - Development of UML class diagrams
    - **OO Programming**
      - Use of Squeak language (variant of Smalltalk)
        - Open source and highly portable language
        - Multimedia support
        - Good infrastructure for complex projects
      - Squeak: supports the XP’s testing-driven practices
        - S-Unit testing framework
Development Process

Regarding Software Testing...

*No systematic method to test the systems had been adopted in the previous terms of the CS2340 course.*

» Fall 2002: first time that testing-related activities were explored in the context of the CS2340 classes

- Decide what testing criteria to use
  - Develop lecture materials
- Figure out how to integrate testing into student assignments
  - How they should be evaluated and graded
- Put together an assessment to see how well it worked
Testing Approach for CS2340 Course

- **Equivalence Partitioning Criterion**
  - Starting point to fit testing practices into the scope of the course
    - Key criterion for functional testing
    - Seems to be easily understood even by students with little experience on testing
  - Educational setting
    - Small programs
      - Numeric input values
    - Genealogy system
      - Input domain involves more complex types of elements
        - How to apply the criterion?
Testing Approach for CS2340 Course

» Equivalence Partitioning Criterion
  - Set of “directions”
    » Help the students on using the criterion in their projects
  - OO Analysis
    CRC cards and scenarios + Definition of input conditions
    Identification of equivalence classes
  - OO Design
    UML class diagrams + Definition of test cases
    Development of test plans
  - OO Implementation
    Squeak + Execution of test cases (S-Unit framework)
ATM Simulation System

- Specification (simplified)
  - The customer is required to enter the account number and the PIN.
    - There is no need of an ATM card.
  - The ATM must provide the following transactions to the customer:
    - Cash Withdrawals, Deposits, Transfers and Balance Inquiries.
    - Only one transaction is allowed in each session.
  - The ATM communicates the transaction to the bank and obtains verification that it was allowed by the bank.
    - If the bank determines that account number or PIN is invalid, the transaction is canceled.
  - The ATM has an operator panel that allows an operator to start and stop the servicing of customers.
    - When the machine is shut down, the operator may remove deposit envelopes and reload the machine with cash.
    - The operator is required to enter the total cash on hand before starting the system from this panel.
ATM Simulation System

- Classes...
  - ATM
  - CashDispenser
  - EnvelopeAccepter
  - OperatorPanel
  - Session
  - Transaction
  - WithdrawalTransaction
  - DepositTransaction
  - TransferTransaction
  - InquiryTransaction
  - Bank

- Some Scenarios...
  - System Startup
  - Session
  - Cash Withdrawal Transaction
  - Deposit Transaction
  - Transfer Transaction
  - Balance Inquiry
  - CRC cards...
Defining Input Conditions

Although the CRC cards do not explicitly deal with the input data items, analyzing their responsibilities under a macroscopic perspective can provide some insight into that.

Input conditions are closely related with input data provided by the user.

1. Analyze each CRC card, looking for “suggestions” on the input data items provided by the user.
   (a) Consider the responsibilities in a macroscopic way. Focus on what each class really has to do.
   (b) Analyze the responsibilities of the collaborators too.
   (c) Write down all the input data items related to a specific class.

Since each class can interact with others, the collaborators should also be investigated in order to find the right class where a specific input item is being treated.
Defining Input Conditions

Class ATM

<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Collaborator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start up system operation <em>(initial cash)</em></td>
<td>OperatorPanel</td>
</tr>
<tr>
<td>Start a session for each customer</td>
<td>Session</td>
</tr>
<tr>
<td>Shut down on operator request</td>
<td>OperatorPanel</td>
</tr>
<tr>
<td>Get <strong>PIN</strong> from the customer</td>
<td></td>
</tr>
<tr>
<td>Get <strong>transaction</strong> choice from the customer</td>
<td></td>
</tr>
<tr>
<td>Get <strong>account</strong> from the customer</td>
<td></td>
</tr>
<tr>
<td>Get <strong>amount entry</strong> from the customer</td>
<td></td>
</tr>
<tr>
<td>Verify the availability of cash for withdrawal</td>
<td>CashDispenser</td>
</tr>
<tr>
<td>Dispense cash</td>
<td>CashDispenser</td>
</tr>
<tr>
<td>Accept deposit envelope</td>
<td>EnvelopeAcceptor</td>
</tr>
</tbody>
</table>
Class CashDispenser

Responsibility
- Set initial cash on hand at startup
- Report cash available
- Dispense cash

Collaborator
Operator Panel

Class EnvelopeAcceptor

Responsibility
- Accept deposit envelope

Collaborator

Class OperatorPanel

Responsibility
- Get initial cash on hand from operator
Class Session

Responsibility
- Perform session use case
- Perform invalid data
- Furnish account to Transaction
- Furnish PIN to Transaction

Collaborator
- ATM, Transaction
- ATM, Bank

Class Transaction

Responsibility
- Perform a particular transaction use case

Collaborator
- WithdrawalTransaction, DepositTransaction, TransferTransaction, InquiryTransaction, Session, Bank
# Defining Input Conditions

## Class WithdrawalTransaction

<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Collaborator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get <em>specifics</em> from customer</td>
<td>ATM</td>
</tr>
<tr>
<td>Send to bank</td>
<td>Session, Bank</td>
</tr>
<tr>
<td>Dispense cash and notify bank when complete</td>
<td>ATM, Bank</td>
</tr>
</tbody>
</table>

## Class DepositTransaction

<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Collaborator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get <em>specifics</em> from customer</td>
<td>ATM</td>
</tr>
<tr>
<td>Send to bank</td>
<td>Session, Bank</td>
</tr>
<tr>
<td>Accept envelope and notify bank when complete</td>
<td>ATM, Bank</td>
</tr>
</tbody>
</table>

## Class TransferTransaction

<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Collaborator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get <em>specifics</em> from customer</td>
<td>ATM</td>
</tr>
<tr>
<td>Send to bank</td>
<td>Session, Bank</td>
</tr>
</tbody>
</table>
Defining Input Conditions

**Class InquiryTransaction**

- Responsibility
  - Get *specifics* from customer
  - Send to bank

- Collaborator
  - ATM
  - Session, Bank

**Class Bank**

- Responsibility
  - Initiate withdrawal
  - Finish withdrawal
  - Initiate deposit
  - Finish deposit
  - Do transfer
  - Do inquiry

- Collaborator
Defining Input Conditions

- **Input Data Items**

  - **Class ATM**
    - account
    - PIN
    - transaction

  - **Class OperatorPanel**
    - initial cash

  - **Class WithdrawalTransaction**
    - amount

  - **Class DepositTransaction**
    - amount

  - **Class TransferTransaction**
    - receiver account
    - amount
2. Identify the main characteristics of each input data item.

3. Identify the interactions among the input data items. Think about how a specific input entry can be related to the other input data of the system. Focus on the system’s operation (scenarios can help on this).

4. Write down the characteristics and interactions of all input data items related to all classes of the system.

The set of characteristics and interactions related to all input data items can be seen as the **input conditions** for the system.
### Defining Input Conditions

#### Class ATM

<table>
<thead>
<tr>
<th>Input Data Item</th>
<th>Characteristic / Interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>account</td>
<td>matching with bank</td>
</tr>
<tr>
<td>PIN</td>
<td>matching with account</td>
</tr>
<tr>
<td>transaction</td>
<td>type</td>
</tr>
</tbody>
</table>

#### Class OperatorPanel

<table>
<thead>
<tr>
<th>Input Data Item</th>
<th>Characteristic / Interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial cash</td>
<td>valid characters</td>
</tr>
<tr>
<td></td>
<td>availability of money in cash dispenser</td>
</tr>
</tbody>
</table>
Defining Input Conditions

- **Class WithdrawalTransaction**
  - **Input Data Item**
    - amount
  - **Characteristic / Interaction**
    - valid characters
    - availability of money in the account for withdrawal

- **Class DepositTransaction**
  - **Input Data Item**
    - amount
  - **Characteristic / Interaction**
    - valid characters

- **Class TransferTransaction**
  - **Input Data Item**
    - receiver account
    - amount
  - **Characteristic / Interaction**
    - matching with bank
    - valid values
    - availability of money in the sender account for transfer
Identifying Equivalence Classes

1. Analyze the set of input conditions.
   (a) Look for related conditions. Try to join them.
   (b) Look for broad conditions. Try to refine them.

2. Analyze each input condition separately.
   (a) Think about the valid values to satisfy the input condition. They will correspond to the elements of a valid equivalence class.
   (b) Think about other possible values associated to the input condition, i.e., the invalid values. They will correspond to the elements of an invalid equivalence class.

3. Enumerate the equivalence classes, assigning a unique number to each class.

If two or more conditions are closely related to each other, it can make more sense to join them into a single condition.

The valid and invalid equivalence classes are established by analyzing the input conditions, in terms of the correct and incorrect values needed to cover them.
### Identifying Equivalence Classes

#### Class ATM

<table>
<thead>
<tr>
<th>Input Condition</th>
<th>Valid</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>account matches with bank size (a) of account</td>
<td>a = 8 (1)</td>
<td>a &gt; 8 (2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 ≤ a &lt; 8 (3)</td>
</tr>
<tr>
<td>valid characters for account numerical</td>
<td>numerical (4)</td>
<td>alpha, special (5)</td>
</tr>
<tr>
<td>PIN matches with account size (p) of PIN</td>
<td>p = 4 (6)</td>
<td>p &gt; 4 (7)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 ≤ p &lt; 4 (8)</td>
</tr>
<tr>
<td>valid characters for PIN numerical</td>
<td>numerical (9)</td>
<td>alpha, special (10)</td>
</tr>
<tr>
<td>type of transaction</td>
<td>withdrawal (11)</td>
<td>none (&quot;blank choice&quot;) (15)</td>
</tr>
<tr>
<td></td>
<td>deposit (12)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>transfer (13)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>inquiry (14)</td>
<td></td>
</tr>
</tbody>
</table>
## Identifying Equivalence Classes

### Class WithdrawalTransaction

<table>
<thead>
<tr>
<th>Input Condition</th>
<th>Valid</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>valid characters for amount</td>
<td>numerical (1)</td>
<td>alpha, special (2)</td>
</tr>
<tr>
<td>availability of money</td>
<td>balance - amount $\geq 0$ (3)</td>
<td>balance - amount $&lt; 0$ (4)</td>
</tr>
<tr>
<td>in the account for withdrawal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Class WithdrawalTransaction
1. Derive test cases associated with valid classes to cover all of them. A test case can cover a number of valid equivalence classes, as large a set as possible.

2. Derive test cases associated with invalid classes to cover all of them. A test case should be written for each invalid equivalence class.

3. Write a test plan providing information on the test cases.

The definition of test cases to cover the equivalence classes requires a more detailed knowledge on the classes (and objects) in terms of structure, attributes, and services that need to be provided.

Students have to provide information on:
- purpose of the test case
- system’s class it tests
- equivalence class it covers
- expected result, obtained result.

At the end of the OOD phase, a number of test cases have already been written, even before the students have started coding.

The definition of test cases to cover the equivalence classes requires a more detailed knowledge on the classes (and objects) in terms of structure, attributes, and services that need to be provided.
## Defining Test Cases

### Test plan

<table>
<thead>
<tr>
<th>Class</th>
<th>Equivalence Class #</th>
<th>Test Case</th>
<th>Expected Result</th>
<th>Actual Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Withdrawal Transaction</td>
<td>(3)</td>
<td>Balance: US$ 100,00</td>
<td>Transaction OK</td>
<td>Transaction Canceled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Amount: US$ 90,00</td>
<td>Balance: US$ 10,00</td>
<td></td>
</tr>
</tbody>
</table>

*Actual Result: fill in after the test case execution*
Executing Test Cases

- **S-Unit**
  - Unit testing framework
    - Squeak’s environment
  - Structure, describe the context of test cases and run them automatically
  - Development of a well-documented test suite
  - Mechanism to motivate regression tests
None specific structural testing criterion was adopted

» Instead… some “general” guidelines…
  ✓ Test the base class
  ✓ Test each subclass in conjunction with the base class

Students were encouraged to keep defining test cases while coding the system, based on its implementation details.

» OO Implementation
  » Design structural test cases
    ✓ Test plans
  » Execute the structural test cases against the system
    ✓ S-Unit framework
Testing Approach for CS2340 Course

- **Relevant Points**
  - CRC cards can be helpful to define input conditions and...
  - Input conditions can be useful in checking the consistency of CRC cards
    » Responsibilities and interactions among classes
  - Defining input conditions and equivalence classes during the OOA phase can provide a better understanding of the system’s requirements
  - Writing test cases even before writing code (OOD phase)
    » Encourage the students to think about the functionality they are designing
Students’ Evaluation

Survey
- Applied at the end of the course
  » After students had finished their projects
- Voluntary
  » 105 students
- Questions consisting in some statements
  » Students were required to choose all the statements they agreed with
- Evaluate the student’s attitude toward…
  » XP idea of starting testing early
  » Using testing criteria
  » Writing test plans
Students’ Evaluation

- XP testing ideas

<table>
<thead>
<tr>
<th>Statement</th>
<th>Students</th>
</tr>
</thead>
<tbody>
<tr>
<td>I prefer to deal with testing after coding the system, at the end of the development process.</td>
<td>12</td>
</tr>
<tr>
<td>I prefer to apply testing practices through all the phases of the development process.</td>
<td>53</td>
</tr>
<tr>
<td>Start thinking about testing in the analysis phase is helpful to better understand the system’s functionality.</td>
<td>41</td>
</tr>
<tr>
<td>I feel it helpful to design a set of test cases at least before implementing the system.</td>
<td>23</td>
</tr>
</tbody>
</table>

Further attempts should be made to integrate the idea of “testing early” into CS courses as a way improve students’ attitude toward testing and their ability to do it effectively.
# Students’ Evaluation

## Using testing criteria

<table>
<thead>
<tr>
<th>Statement</th>
<th>Students</th>
</tr>
</thead>
<tbody>
<tr>
<td>I prefer to test the system by my “own way”, without applying a specific testing criterion.</td>
<td>46</td>
</tr>
<tr>
<td>I feel more confident about the <strong>test cases</strong> when working with some testing criterion.</td>
<td>21</td>
</tr>
<tr>
<td>I feel more confident about the <strong>system’s quality</strong> when working with some testing criterion.</td>
<td>36</td>
</tr>
<tr>
<td>I feel motivated to think about <strong>common errors</strong> that could be committed during the development when working with some testing criterion.</td>
<td>29</td>
</tr>
<tr>
<td>I feel that working with some testing criterion I can reduce the total development effort.</td>
<td>10</td>
</tr>
</tbody>
</table>

35 students were strictly “against” the idea of working with testing criteria.

70 students agreed on the relevance of applying testing criteria for some aspect of the development process.
Students’ Evaluation

- Using testing criteria
  - Moreover…
    - Negative attitude on using Equivalence Partitioning

*Teaching the criterion for testing systems whose input values are not numerical requires more investigation.*

- Keep refining/improving our directions for applying the criterion in the next offerings of the course
- Investigate the application of other testing criteria
**Students’ Evaluation**

- Writing testing plans
  - Students’ opinion on having to do a test plan

<table>
<thead>
<tr>
<th>Attitude</th>
<th>Students</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completely worthless</td>
<td>62</td>
</tr>
<tr>
<td>Neutral position</td>
<td>31</td>
</tr>
<tr>
<td>Valuable / really valuable</td>
<td>12</td>
</tr>
</tbody>
</table>

*Negative Attitude!!!*
Students’ Evaluation

Some of these students were the same who had a negative attitude toward test plans.

Although this specific group of students had been “against” the test plans idea, they were able to recognize its usefulness for some aspects of the development process.

### Students’ Evaluation

<table>
<thead>
<tr>
<th>Statement</th>
<th>Students</th>
</tr>
</thead>
<tbody>
<tr>
<td>I believe it is more useful to elaborate test plans for complex systems than for small/medium systems.</td>
<td></td>
</tr>
<tr>
<td>I believe it is valuable to elaborate test plans as part of any development process.</td>
<td>23</td>
</tr>
<tr>
<td>Test plans are valuable as a way to keep track of test cases.</td>
<td>13</td>
</tr>
<tr>
<td>Test plans are valuable when changes in the system are required.</td>
<td>16</td>
</tr>
</tbody>
</table>

37 students (from the 62 who had a negative attitude toward test plans)
15 students (from the 31 who presented a neutral position)
02 students (from the 12 who had a positive attitude)
Conclusions

- Positive attitudes towards…
  - Starting testing activities as early as possible
  - S-Unit

  * XP’s testing-driven practices can be seen as an interesting and challenging mechanism to create new Software Testing learning opportunities.

- Negative attitudes towards…
  - Equivalence Partitioning
  - Test plans
Further Work

- **CS2340 Course**
  - Keep investigating ways to introduce testing practices into the context of the course
    - Introduce the use of S-Unit earlier in the development process
  - Keep applying the Equivalence Partitioning
    - Investigate better ways for teaching the criterion
    - Refine/improve our directions
  - Explore the use of other testing criteria
    - Functional and structural

- **Explore the idea of “testing early” into the context of other courses**
  - Software Testing
  - Software Engineering