Case Study 1– Bank example

A lot of new material illustrated via case study!

- Requirements
- Use Cases
- Design – Identify the Classes
- Design – via Tests
- Writing a Test for an aspect of a Use Case
- TDD – Test Driven Development
- EiffelStudio
  - Clusters and Classes
  - Tests and ETester for TDD
  - Project Configuration and Ace files
- Object quality: is_equal and equal
Case Study 1 – Bank example

- Requirements
  - Open an account for a customer (savings or chequing)
  - Deposit
  - Withdraw
  - Display details of an account
  - Change LOC
  - Produce monthly statements
  - Print a list of customers
  - ...

- Ambiguities
  - What is the difference between savings and chequing?
  - Many others ...
Case Study 1 – Bank example

- How should we go from Requirements to Code?
- Two basic approaches
  - Plan-driven (waterfall type models)
  - Agile (incremental approaches)
Requirements to Code?

How to bridge the gap between requirements and code?
The waterfall model of the lifecycle

- Feasibility Study
- Requirements Analysis
- Specification
- Global Design
- Detailed Design
- Implementation
- Validation & Verification
- Distribution

PROJECT PROGRESS
Arguments for the waterfall

(After B.W. Boehm: *Software engineering economics*)

- The activities are necessary.
  - (But: merging of middle activities.)

- The order is the right one.
The waterfall model of the lifecycle

- Feasibility Study
- Requirements Analysis
- Specification
- Global Design
- Detailed Design
- Implementation
- Validation & Verification
- Distribution

Project Time
Problems with the waterfall

- Late appearance of actual code.
- Lack of support for requirements change — and more generally for extendibility and reusability.
- Lack of support for the maintenance activity (70% of software costs?).
- Division of labor hampering Total Quality Management.
- Impedance mismatches.
- Highly synchronous model.
Quality control?

- Analysts
- Designers
- Implementers
- Testers
- Customers
Impedance mismatches

1. As Management requested it.
2. As the Project Leader defined it.
3. As Systems designed it.
4. As Programming developed it.
5. As Operations installed it.
6. What the user wanted.

(Pre-1970 cartoon; origin unknown)
How to think about requirements?

- Use Cases
Bank Requirements – Use Cases

Actors:
- Customer
- Teller

Use Cases:
- Open Account
- Deposit
- Withdraw
- Print Customers

System boundary
Bank Use Cases

Challenge question:

- What would the Use Case diagram look like if the bank supplies ATMs in addition to tellers?
Use Case – Open Account

- **Actors:** Customer (initiator), Teller
- **Overview:** the customer applies for and is granted a new bank account with some initial approved line of credit. The teller logs on to the system, selects open account, and supplies the new account details. The system prints a copy of the contents of the new account with the date, which the teller hands to the customer.
Design of Bank – Identify Classes

- TELLER
- CUSTOMER
- SAVINGS_ACCOUNT
- CHEQUING_ACCOUNT
- MAIN_MENU
- BALANCE_INQUIRY
- INTEREST_RATE ........

- Where do we start?
Test Driven Development

1. Write a little Test
   - Test will not work initially (there is no code)
   - Might not even compile

2. Make the Test work quickly
   - Commit whatever sins are necessary in the process (e.g. duplication in the code)

3. Refactor
   - Eliminate the duplication created in merely getting the test to work
TDD

- Goal: clean code that works
- Clean code
  - Simple ??
EiffelStudio – create a new project
Call the new project bank
Compile and Run

```
print("hello world!")
```
Compile often!

- Always start with a compiled system
  - New project wizard
  - Copy an existing system
- A compiled system is needed for browsing
- Compile often!
ETester – Create a tests cluster

www.cs.yorku.ca/eiffel/etester
Create class `account_test`
Clusters and Classes

```
indexing
    description: "Tests for CUSTOMER and ACCOUNT in bank"
    author: "JSO"

class ACCOUNT_TEST inherit
    UNIT_TEST
create
    make
feature {NONE} -- Initialization
    make is
        -- Run tests
        do
            make_test
            add_boolean_case(agent test_create_account)
            to_html ("tests.html")
        end
feature -- test cases
    test_create_account: BOOLEAN is
        do
        | 
    end
    -- class ACCOUNT_TEST
```
Ace file – Directory Structure

- Directory (cluster)
- Class file
Ace File – Ace.ace

http://www.cs.yorku.ca/eiffel/precompiled

system
  "bank2"

root
  root_class: make

default
  precompiled ("C:\Eiffel54\precomp\spec\windows\base-time-etester")
  assertion (check)
  assertion (require) ...

cluster
  root_cluster:    "C:\eiffel54_projects\bank2"

all tests:      ".\tests"

library base:    "$ISE_EIFFEL\library\base"
  exclude
    "table_eiffel3"; "desc";

end ...

end
Project Configuration (Ace file)

Root Class
ACCOUNT_TEST

All contracts turned on
Write Test

test_create_account: BOOLEAN is
local
c: CUSTOMER

do
    comment("test_create_account")
    create c.make("Archie Fox")
    Result := equal(c.name, "Archie Fox")
end

Error code: VTCT
Error: type is based on unknown class.
What to do: use an identifier that is the name of a class in the universe.
Class: ACCOUNT_TEST
Unknown class name: CUSTOMER
The Test specifies a Design

```
CUSTOMER

name: STRING

invariant

name /= Void

ANY

is_equal (other: like Current): BOOLEAN

frozen equal (some: ANY; other: like some): BOOLEAN
```
EiffelStudio – create feature tool

![EiffelStudio Create Feature Tool](image-url)
indexing
description: "A bank customer"

class CUSTOMER create
  set_name

feature -- Element change
  set_name (a_name: STRING) is
    -- Create a customer. Set 'name' to 'a_name'.
    require
      a_name_not_void: a_name /= Void
    do
      name := a_name
    ensure
      name_assigned: name = a_name
  end

feature -- Access
  name: STRING
    -- of customer
  invariant
    name_not_void: name /= Void
end
ETester GUI – Green bar
Test: A customer with a savings account

test_create_customer_with_account: BOOLEAN is
local
    a1,a2: SAVINGS_ACCOUNT
    c: CUSTOMER
    initial_deposit: REAL

do
    initial_deposit := 500
    create a1.make(initial_deposit)
    check a1.balance = initial_deposit end
create c.make(a1, "Archie Fox")
a2 := c.account
Result := equal(c.name, "Archie Fox") and
          a2 = a1 and a2.balance = initial_deposit
end
ETester 2.1

Executable test suite location: C:\eiffel54_projects\bank2\EiFGE\W_code\bank2.exe

Location of output: C:\eiffel54_projects\bank2\EiFGE\W_code\tests.htm

Run

Sun Dec 28 23:05:40 EST 2003

SIMPLE_CONCRETE_TEST_SUITE
Note: * indicates a violation test case

FAILED (1 failed & 0 passed out of 1)

<table>
<thead>
<tr>
<th>Case Type</th>
<th>Passed</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Violation</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Boolean</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>All Cases</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Contract Violation</th>
<th>Test Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test1</td>
<td>ACCOUNT_VIOLATION</td>
<td>ACCOUNT_TEST</td>
</tr>
<tr>
<td>FAILED</td>
<td>NONE</td>
<td>test_create_customer_with_account</td>
</tr>
</tbody>
</table>
Use debugger to find problem

test_create_customer_with_account: BOOLEAN is
local
  a1, a2: SAVINGS_ACCOUNT
  c: CUSTOMER
  initial_deposit: REAL
do
  comment ("test_create_customer_with_account")
  initial_deposit := 500
  create a1.make (initial_deposit)
  check
    a1.balance = initial_deposit
  end
  create c.make (a1, "Archie Fox")
  a2 := c.account
  Result := equal (c.name, "Archie Fox") and a2 = a1
end
EiffelStudio – BON tool

```c
class SAVINGS_ACCOUNT create
  make

feature {NONE} -- creation
  make(initial_deposit:REAL) is
    -- create a customer savings account
    -- with initial balance of 'initial_deposit'
    require
      initial_deposit >= 0
    do
      balance := balance + initial_deposit
    ensure
      balance = old balance + initial_deposit
  end
```
Client-Supplier relationship

CUSTOMER
- name: STRING
- account: ACCOUNT
  invariant
  name /= Void
  account /= Void

SAVINGS ACCOUNT
- balance: REAL
  invariant
  balance >= 0

account
Open Account Use case (revisited)

- **Actors**: Customer (initiator), Teller
- **Overview**: the customer applies for and is granted a new bank account with some initial approved line of credit. The teller logs on to the system, selects open account, and supplies the new account details. The system prints a copy of the contents of the new account with the date, which the teller hands to the customer.

- We have not yet dealt with the LOC, system menu and the receipt. We leave this as an exercise.
Withdraw Use Case

- **Actors**: Customer (initiator) and Teller
- **Overview**: The customer provides the teller with an account number and a withdrawal amount. The teller selects `withdraw-request` from the system menu, and enters the data. The System debits the account by the requested withdrawal amount, and prints a receipt with the date and the amount, which the teller gives to the customer.

- **Question**: What happens if there is not enough money in the account to withdraw?
test_withdraw: BOOLEAN is

local

sa1, sa2: SAVINGS_ACCOUNT
initial_deposit: REAL

do

comment("test_withdraw")
initial_deposit := 500
create sa1.make(initial_deposit)
create sa2.make(200)

sa1.withdraw(300)
Result := equal(sa1, sa2)

end
Bank design (BON static diagram)

ANY

is_equal(other: like Current): BOOLEAN

frozen equal(some: ANY; other: like some): BOOLEAN

CUSTOMER

name: STRING
account: ACCOUNT

invariant
name /= Void
account /= Void

SAVINGS_ACCOUNT

balance: REAL
withdraw (amount: REAL)

require
amount > 0
amount <= balance

ensure
balance = old balance - amount
balance >= 0
EiffelStudio feature tool – withdraw

```plaintext
feature {ANY} -- Element change
withdraw (amount : REAL) is

-- "amount" from account

require
amount > 0

local

do ¬ do ∧ once ∧ deferred ∧ external
balance := balance - amount

ensure
balance = old balance - amount

end
```

Department of Computer Science, York University

Slides based on Object Oriented Software Construction 29/12/2003 3:22 PM 49
Run tests
Object equality

sa1, sa2: SAVINGS_ACCOUNT

Result := equal(sa1, sa2)

- `equal` is inherited from `ANY`
- In this case, no redefinition is needed
- You redefine `equal` by redefining `is_equal`
is_equal (other: like Current): BOOLEAN is
  -- Is `other' attached to an object considered
  -- equal to current object?
require
  other_not_void: other /= Void
do
  -- call C version of equality
ensure
  symmetric:
    Result implies other.is_equal (Current)
  consistent:
    standard_is_equal (other) implies Result
end
equal

frozen equal (some: ANY; other: like some): BOOLEAN is
   -- Are `some' and `other' either both void or attached
   -- to objects considered equal?
   do
      if some = Void then
         Result := other = Void
      else
         Result := other /= Void and then
            some.is_equal (other)
      end
   ensure
      definition:
         Result = (some = Void and other = Void)
            or else
            ((some /= Void and other /= Void)
               and then
               some.is_equal (other))
   end
withdraw_negative

test_withdraw_negative_amount: BOOLEAN is

local

a1, a2: SAVINGS_ACCOUNT
initial_deposit: REAL

do

comment("test_withdraw_negative_amount")
initial_deposit := 500
create a1.make(initial_deposit)
create a2.make(200)
a1.withdraw(-100) – what happens here?
Result := equal(a1, a2)

end
Contract Violation

ETester 2.1

Executable test suite location: C:\eiffel54_projects\bank2\EIGEN\W_code\bank2.exe
Location of output: C:\eiffel54_projects\bank2\EIGEN\W_code\tests.htm

Sun Dec 28 23:47:26 EST 2003

SIMPLE_CONCRETE_TEST_SUITE
Note: * indicates a violation test case

FAILED (1 failed & 2 passed out of 3)

<table>
<thead>
<tr>
<th>Case Type</th>
<th>Passed</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Violation</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Boolean</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>All Cases</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

State | Contract Violation | Test Name
-----|-------------------|-------------
Test1 | ACCOUNT_TEST      |             
PASS | NONE              | test_create_customer_with_account  
PASS | NONE              | test_withdraw                        
FAIL | OCCURRED          | test_withdraw_negative_amount
Violation Test Case

In *make*:
add_violation_case (**agent** test_withdraw_negative_amount)

test_withdraw_negative_amount is
local
  a: SAVINGS_ACCOUNT
  initial_deposit: REAL
  do
  comment("test_withdraw_negative_amount")
  initial_deposit := 500
  create a.make(initial_deposit)
  a.withdraw(-100) -- a violation here is expected
end
(expected) Violation case

![ETester 2.1](image)

**Executable test suite location:** `C:\eiffel54_projects\bank2\EFG\code\bank2.exe`

**Location of output:** `C:\eiffel54_projects\bank2\EFG\code\tests.htm`

**Run**

Sun Dec 28 23:50:16 EST 2003

**SIMPLE_CONCRETE_TEST_SUITE**

Note: * indicates a violation test case

**PASSED (3 out of 3)**

<table>
<thead>
<tr>
<th>Case Type</th>
<th>Passed</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Violation</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Boolean</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>All Cases</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

**State** | **Contract Violation** | **Test Name**
---|------------------------|------------------------
Test1 | ACCOUNT_TEST | test_create_customer_with_account
test_withdraw
*test_withdraw_negative_amount

---
BON static diagram
Bank Challenge question

- Add a deposit feature
  - How would you write the test

- New Requirement: monthly statement
  - The System must generate a monthly statement for the customer
  - Each deposit or withdrawal must be date stamped
  - Interest must be added or charged as applicable

- New Requirement:
  - Print all bank customers in alphabetical order
  - Print customers with largest balances in order