OBJECT-ORIENTED TESTING

Dr. Durga Prasad Mohapatra
Professor
Department of Computer Science and Engineering
National Institute of Technology, Rourkela.

Introduction

- More than 50% of development effort is being spent on testing.
- Quality and effective reuse of software depend to a large extent on thorough testing.
- The nature of OO programs changes both testing strategy and testing tactics.
- Most research on OO paradigms focus on analysis and design
- Binder argues that more testing is needed to obtain high reliability in OO systems, since each reuse is a new context of usage.

When should testing begin?

- Analysis and Design:
  - Testing begins by evaluating the OOA and OOD models
  - *How do we test OOA models (requirements and use cases)?*
  - *How do we test OOD models (class and sequence diagrams)?*
  - Structured walk- through, prototypes
  - Formal reviews of correctness, completeness and consistency

Contd…

- Programming:
  - *How does OO makes testing different from procedural programming?*
  - Concept of a ‘unit’ broadens due to class encapsulation
  - Integration focuses on classes and their execution across a ‘thread’
    or in the context of a use case scenario
  - Validation may still use conventional black box methods

Strategic Issues

- Issues to address for a successful software testing strategy:
  - Specify product requirements long before testing commences
    For example: portability, maintainability, usability
    Do so in a manner that is unambiguous and quantifiable
  - Understand the users of the software, with use cases
  - Develop a testing plan that emphasizes “rapid cycle testing”
    Get quick feedback from a series of small incremental tests
  - Build robust software that is designed to test itself
    Use assertions, exception handling and automated testing tools (Junit).
  - Conduct formal technical reviews to assess test strategy & test cases.

Testing OOA and OOD Models

- The review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code level.
- Therefore, a problem in the definition of class attributes that is uncovered during analysis will circumvent side effects that might occur if the problem was not discovered until design or code (or even the next iteration of analysis).
- By fixing the number of attributes of a class during the first iteration of OOA, the following problems may be avoided:
  - Creation of unnecessary subclasses.
  - Incorrect class relationships.
  - Improper behavior of the system or its classes.

Contd…

- If the error is not uncovered during analysis and propagated further more efforts needed during design or coding stages.
- Analysis and design models cannot be tested in the conventional sense, because they cannot be executed.
- Formal technical review can be used to examine the correctness and consistency of both analysis and design models.

Contd…

- Correctness:
  - Syntax: Each model is reviewed to ensure that proper modeling conventions have been maintained.
  - Semantic: Must be judged based on the model's conformance to the real world problem domain by domain experts.
- Consistency:
  - May be judged by considering the relationship among entities in the model.
  - Each class and its connections to other classes should be examined.
  - The Class-responsibility-collaboration model and object-relationship diagram can be used.

Major Challenges in OO Testing

- What is an appropriate unit for testing ?
- Implications of OO Features:
  - Encapsulation
  - Inheritance
  - Polymorphism & Dynamic Binding
- State-based testing
- Test coverage analysis
- Integration strategies
- Test process strategy

Object-Oriented Testing Strategies

Black Box Testing:

- Black box testing treats the system as a "black-box", so it doesn't explicitly use Knowledge of the internal structure.
- It focuses on the functionality part of the module. It is the testing based on an analysis of the specification of a piece of software without reference to its internal workings.
- The goal is to test how well the component conforms to the published requirements for the component.
- Specifically, this technique determines whether combinations of inputs and operations produce expected results.

Characteristics

It attempts to find:

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Performance errors
- Initialization and termination errors

White Box Testing

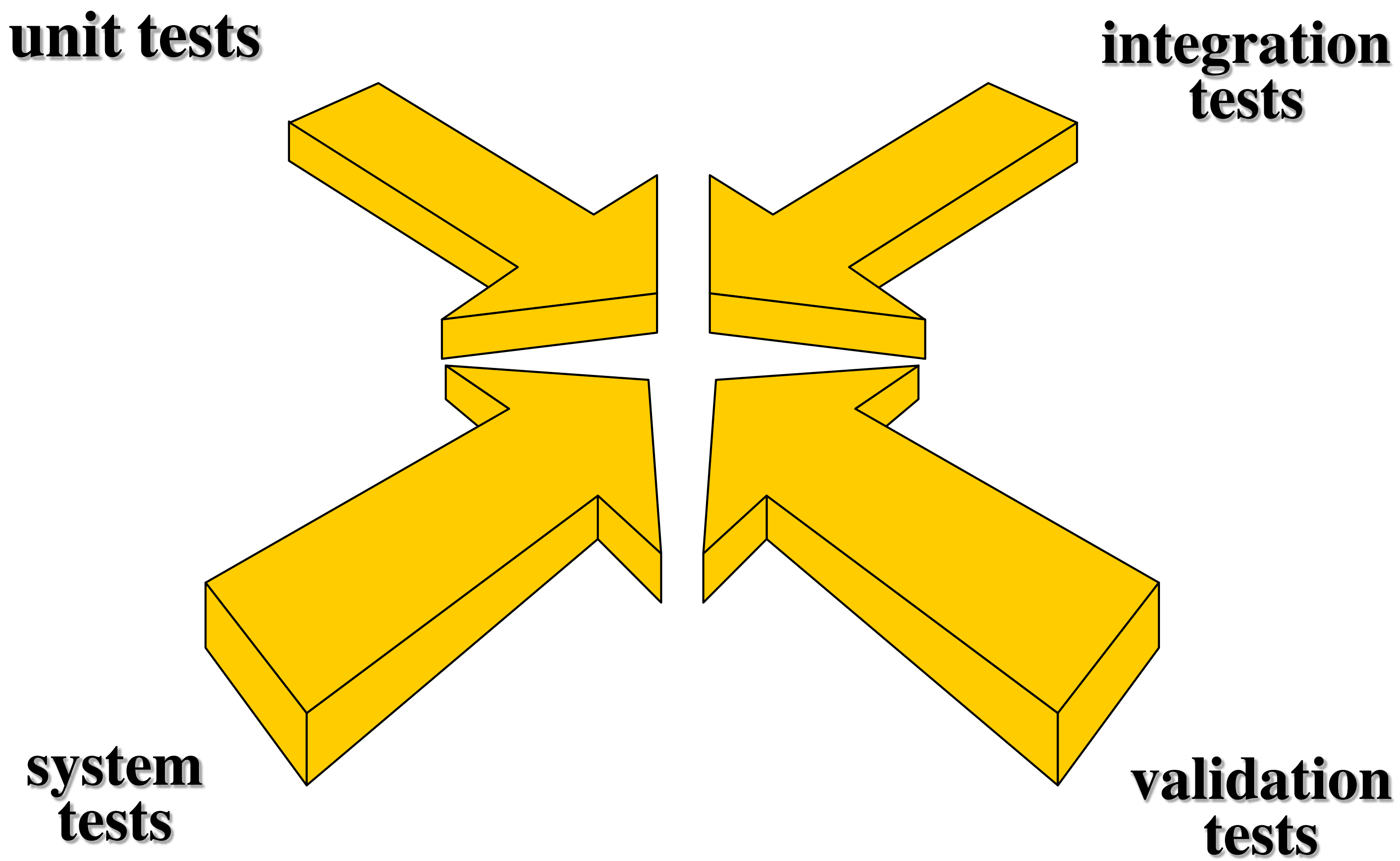
- White box testing involves looking at the structure of the code.
- All internal components should be adequately exercised.
- It is the testing based on an analysis of internal workings and structure of a piece of software.
- It is also known as Structural Testing / Glass Box Testing / Clear Box Testing.

Characteristics

white box testing tends to involve the coverage of the specification in the code.

- Aims to establish that the code works as designed.
- Examines the internal structure and implementation of the program.
- Target specific paths through the program.
- Needs accurate knowledge of the design, implementation and code.

Object-Oriented Testing Levels



Unit Testing

- Smallest testable unit is the encapsulated class or object
- A single operation needs to be tested as part of a class hierarchy because its context of use may differ subtly
- Class testing is the equivalent of unit testing in conventional software approach:
  - Methods within the class are tested
  - The state behavior of the class is examined
- Class testing focuses on designing sequences of methods to exercise the states of a class

Class testing

- In class testing, unit testing is applied for each class
- Class testing uses a variety of methods :
  - Fault-based testing
  - Random testing
  - Partition testing
- Each method exercises the operations encapsulated by class
- Test sequences are designed to ensure that relevant operations are used
- The state of the class, represented by the values of its attributes, is examined to determine if errors exist

Fault-Based Testing

- The main objective is to design tests that have a high likelihood of uncovering plausible faults (i.e. aspects of the implementation of the system that may result in defects).
- Test cases are designed to exercise the design or code to determine whether these faults exists or not.
- Fault-based testing can find significant number of errors with low expenditure of effort.
- The three types of faults encountered are : unexpected result, wrong operation/message used , incorrect invocation.

Examples

- Example 1: Boundary value error:
  - When a SQRT operation that returns errors for negative value and zero itself.
  - Zero itself checks whether the programmer made a mistake like

“if (x>0) calculate_the_square_root();”

Instead of the correct:

“if (x>=0) calculate_the_square_root(); ”

Contd…

- Example2: consider a Boolean expression:

  " if (a && !b || c) "

  - Multiconditon testing and related techniques probe for certain plausible faults such as :
  - "&& " should be an " || "
  - "! " was left out where it was needed
  - There should be a parentheses around " !b || "
- Fault-based testing misses two types of errors :
  - incorrect specifications
  - interactions among subsystems

Random Class Testing

1. Identify methods applicable to a class
2. Define constraints on their use – e.g. the class must always be initialized first
3. Identify a minimum test sequence – an operation sequence that defines the minimum life history of the class
4. Generate a variety of random (but valid) test sequences – this exercises more complex class instance life histories

- Example:
- An account class in a banking application has *open*, *setup*, *deposit*, *withdraw*, *balance*, *summarize*, *creditLimit*, and *close* methods.

Contd…

2. The constraint of all operations is that the account must be opened first and closed on completion

- Minimum behavior is

  Open – setup – deposit – withdraw – close

- Generate random test sequences using this template

  Open – setup – deposit – [deposit | withdraw | balance | summarize| creditLimit] – withdraw – close.*

- **Test case 1:**

  Open – setup – deposit –deposit – balance-summarize– withdraw – close.

- **Test case 2:**

  Open – setup – deposit –withdraw -` deposit- balance-creditLimit- withdraw – close.

Partition Class Testing

- Reduces the number of test cases required (similar to equivalence partitioning)
- Input and output are categorized , and test cases are designed to exercise each category.

Contd…

- **State-based partitioning:**
  - Categorize and test methods separately based on their ability to change the state of a class

  Example: *deposit* and *withdraw* change state but *balance* does not
- **Test case 1:**

Open – setup – deposit –deposit –withdraw– withdraw – close.

- **Test case 2:**

Open – setup – deposit –summarize-creditLimit– withdraw – close.

Contd…

- **Attribute-based partitioning:**
  - Categorize and test operations based on the attributes that they use

Example: attributes *balance* and *creditLimit* can define partitions

  - Operations are divided into three partitions:
    1. Operations that use creditLimit
    2. Operations that modify creditLimit
    3. Operations that do not use or modify creditLimit

Contd…

- **Category-based partitioning:**
  - Categorize and test operations based on the generic function each performs

Example: initialization (*open*, *setup*), computation (*deposit*, *withdraw*), queries (*balance*, *summarize*), termination (*close*)

Integration Testing

- Object-Oriented software does not have a hierarchical control structure so conventional top-down and bottom-up integration tests have little meaning
- Integration applied three different incremental strategies
  - Thread-based testing: integrates the set of classes that collaborate to respond to one input or event
  - Use-based testing: integrates classes required by one use case
  - Cluster testing: integrates classes required to demonstrate one collaboration

Thread-based integration Strategies

- A thread consists of all the classes needed to respond to a single external input.
- Each thread is integrated and tested individually.
- Regression testing is applied to ensure that no side effects occur.

Use-based integration Strategies

- Use-based testing begins by testing classes that use few or no server classes
- Next, dependent classes that use the independent group of classes are tested, followed by classes that use the dependent group, and so on.
- Sequence of testing layers of dependent classes continues until the entire system is constructed.

Cluster Testing

- A cluster is a collection of classes (possibly from different systems) cooperating with each other via messaging.
- A cluster specification should include methods from each class that will be accessed
- Cluster testing focuses on the interaction among the instances of the classes in the cluster
- It assumes that each class has been tested individually
- Cluster testing is considered a second level of integration testing

Scenario-Based Test Design

- Scenario-based testing captures the tasks (via use cases) that the user has to perform ,then applying them and their variants as tests.
- Scenarios uncover interaction errors
- Test cases must be more complex and more realistic than fault-based tests
- Scenario-based testing tends to exercise multiple subsystems in a single test

Contd…

- **Example:**
- **Use Case 1:** Fix the Final Draft
- **Background:** It's not unusual to print the "final" draft, read it, and discover some annoying errors that weren't obvious from the on-screen image. This use-case describes the sequence of events that occurs when this happens
  1. Print the entire document
  2. Move around in the document, changing certain pages.
  3. As each page is changed, it's printed.
  4. Sometimes a series of pages is printed.
- This scenario describes two things : a test and specific user needs.

Contd…

- **Use Case 2:** Print a New Copy

Background: Someone asks for a fresh copy of the document. It must be printed.

1. Open the document.
2. Print it.
3. Close the document.

- After the “Fix the Final Draft ” scenario, just selecting “Print ” in the menu will print the last corrected page again.

Contd…

So the correct scenario according to the editor is :

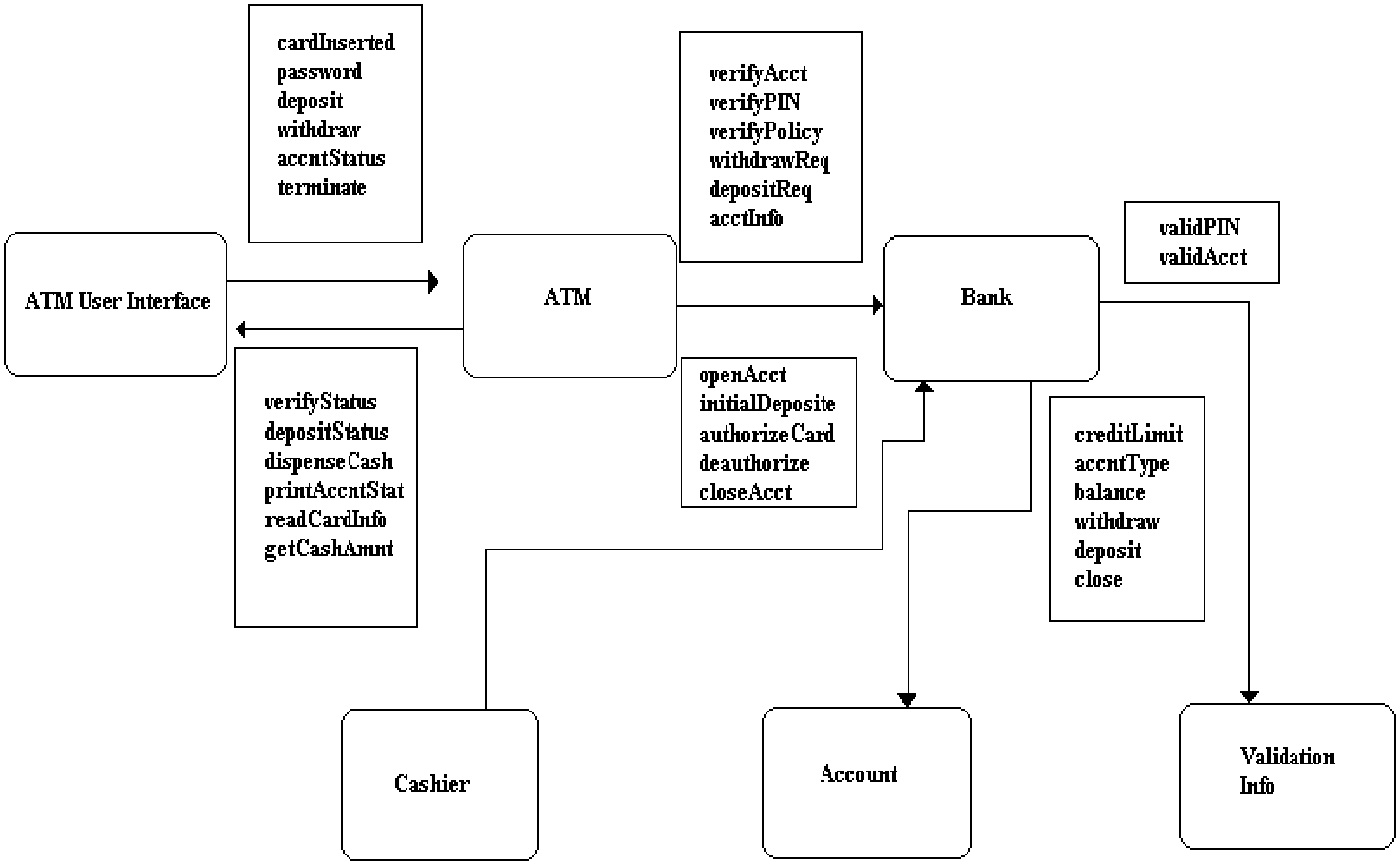
- **Use Case 2:** Print a New Copy
  1. Open the document.
  2. Select "Print " in the menu.
  3. Check if you're printing a page range; if so, click to print the entire document.
  4. Click on the "Print " button
  5. Close the document.

Random Integration Testing

- Multiple Class Random Testing

1. For each client class, use the list of class methods to generate a series of random test sequences. The methods will send messages to other server classes
2. For each message that is generated, determine the collaborating class and the corresponding method in the server object
3. For each method in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits
4. For each of the messages, determine the next level of methods that are invoked and incorporate these into the test sequence

Class Collaboration Diagram for Banking Application



Example

- Consider a sequence of operations for the BANK class relative to an ATM class

  verifyAcct-verifyPIN- [[verifyPolicy - withdrawReq] | depositReq | acctInfoREQ]^n
- A random test case for the BANK class Test case 1:

  verifyAcct-verifyPIN-depositReq

Contd …

- In order to consider the collaborators involved in this test, the msgs. Associated with each of the operns noted in the above test case 1 are considered.
- BANK must collaborate with validationInfo to execute the verifyAcct& verifyPin, with Acct. to execute depositRequest.
- So, a new test case that exercises the above collaborations is

  Test case 2: $\text{verifyAcct}_{\text{BANK}}$ – [$\text{validAcct}_{\text{validationInfo}}$]- $\text{verifyPIN}_{\text{BANK}}$- [$\text{validPin}_{\text{validationInfo}}$]- $\text{depositReq}_{\text{BANK}}$ – [$\text{deposit}_{\text{Account}}$]

Contd…

- Approach for multiple class partition testing is similar to the approach used for partition testing of individual classes.
- A single class is partitioned as discussed earlier.
- Here, the test sequence is expanded to include those operations that are invoked via messages to collaborating classes.

Contd…

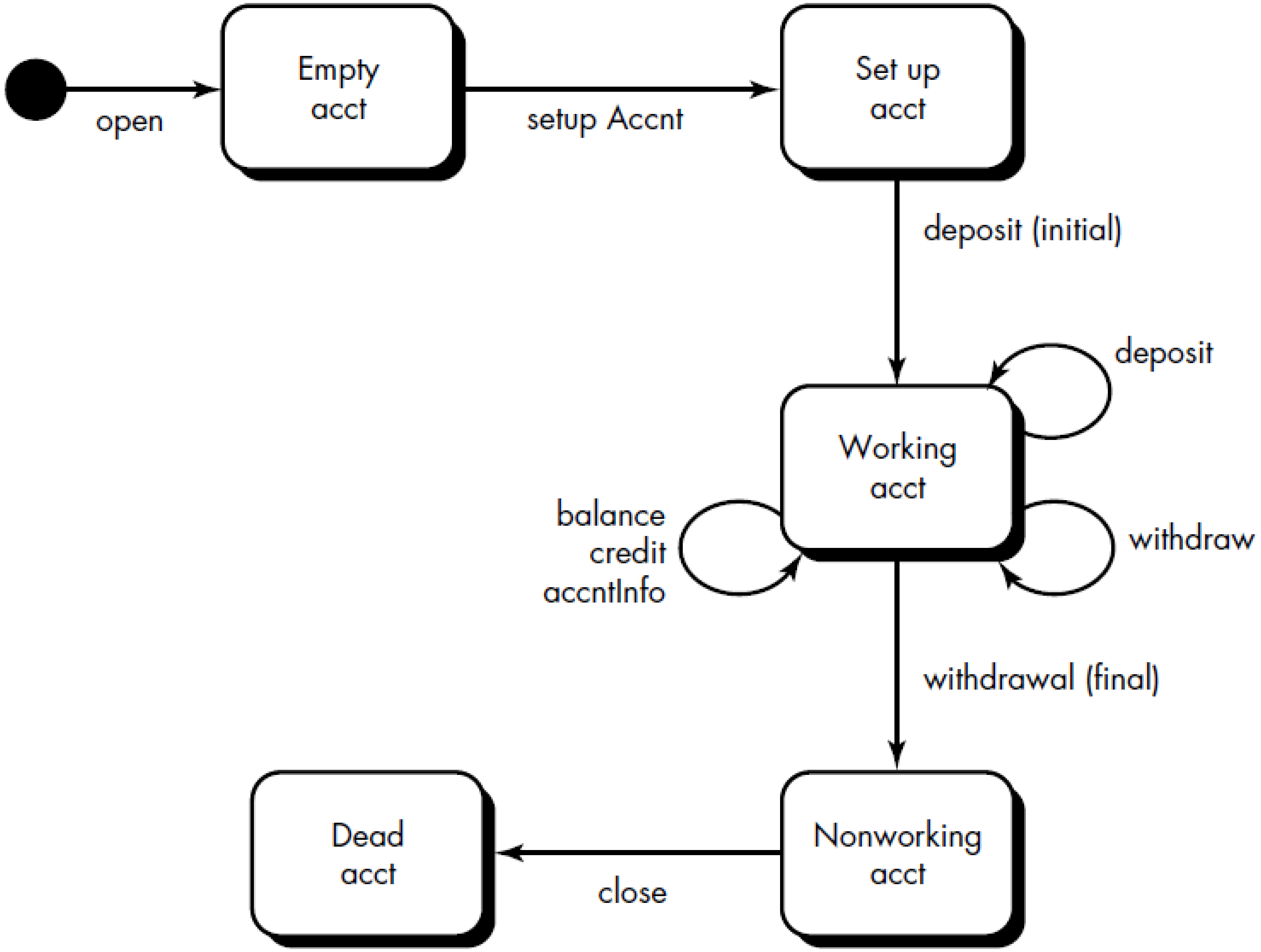
- An alternative approach

  partitions the tests based on the interfaces to a particular class.
- e. g. the bank class receives messages from **ATM** & **Cashier** classes.
- The methods within the BANK can therefore be tested by partitioning them into those that serve **ATM** & those that serve **Cashier**.
- The state based partitioning can be used to refine the partitions further.

Behavioral Integration Testing

- Tests Derived from Behavioral Model:
- Derive tests from the object-behavioral analysis model
- Each state in a State diagram should be visited in a "breadth-first" fashion.
  - Each test case should exercise a single transition
  - When a new transition is being tested only previously tested transitions are used
  - Each test case is designed around causing a specific transition

State transition diagram for Account class



Contd…

- Example: Account Class
  - Initial transitions in *Account* class move through the *empty acct* and *setup acct* states.
  - The majority of all behavior for instances of the class occurs while in the *working acct* state.
  - A final withdrawal and close cause the account class to make transitions to the *nonworking acct* and *dead acct* states, respectively

Contd…

- **Test case 1:**

Open – setupAcct – deposit(initial)–withdraw(final)– close. (identical to the minimum test sequence)

- Adding additional test sequences to it, we may get
- **Test case 2:**

Open – setupAcct – deposit(initial)-deposit-balance-credit–withdraw(final)– close.

- **Test case 3:**

Open – setupAcct – deposit(initial)-deposit-withdraw-accntInfo –withdraw(final)– close.

Assignment

- Construct the state chart diagram for Credit Class. Then, generate test cases from it.
- Hints: A credit card can move between *undefined*, *defined*, *submitted* and *approved* states
- The first test case must test the transition out of the start state *undefined* and not any of the other later transitions

Our proposed approach for Generating Test cases from State Chart Diagram

Step 1: Draw State chart diagram of the problems.

Step 2: Select predicates and draw state graph according to state chart diagram.

Step 3: Develop Java code for the state graph.

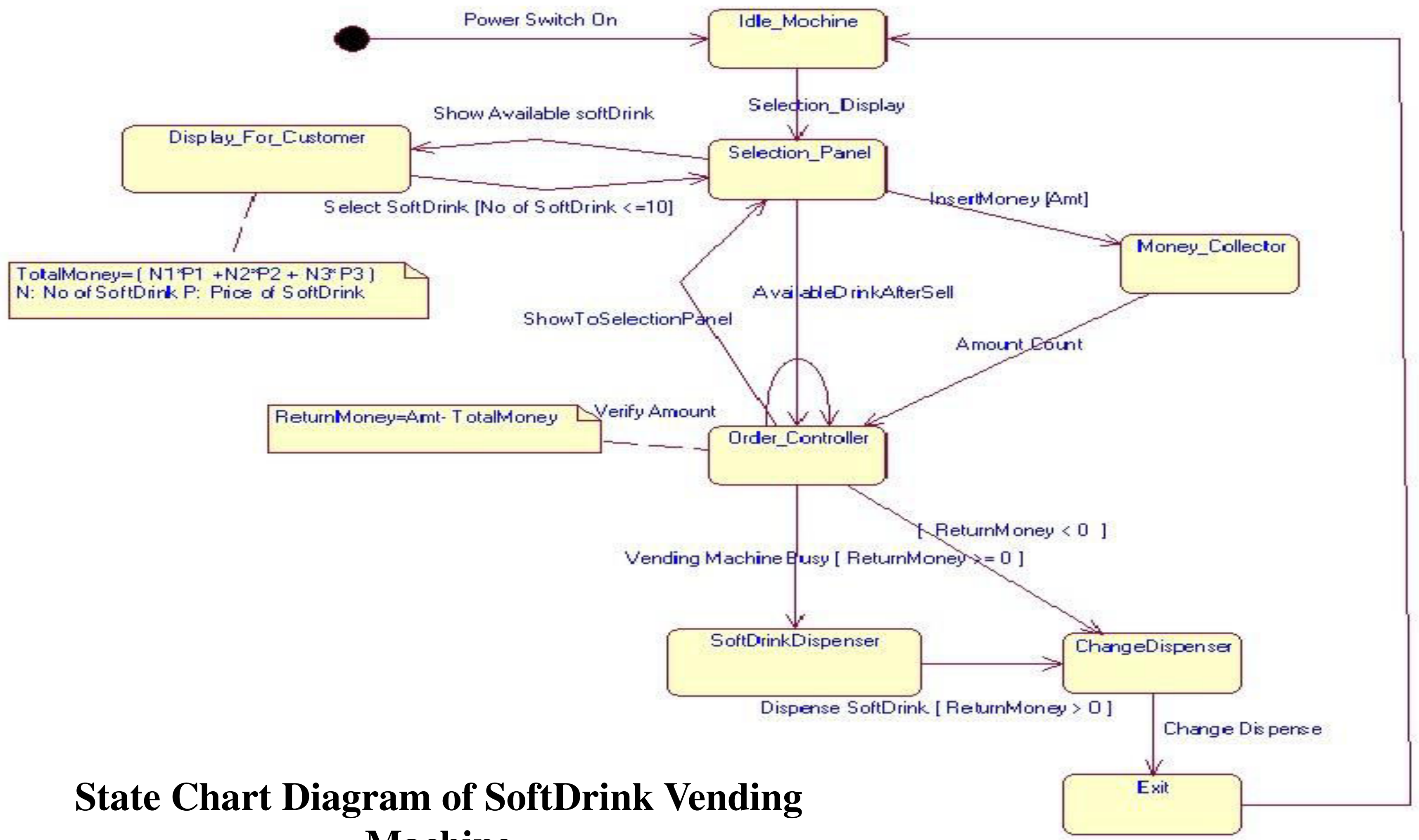
Step 4: Generate class file with the use of modelJunit.jar and JUnit.jar file.

Step 5: Generate the State coverage, Transition coverage, Transition pair coverage.

Step 6: Construction of EFSM from the source code

Step 7: Generate Test Sequences and Test Cases.

Case Study: SoftDrink Vending Machine



State Chart Diagram of SoftDrink Vending Machine

Cyclomatic Complexity of Our Case Study

McCabe cyclomatic complexity

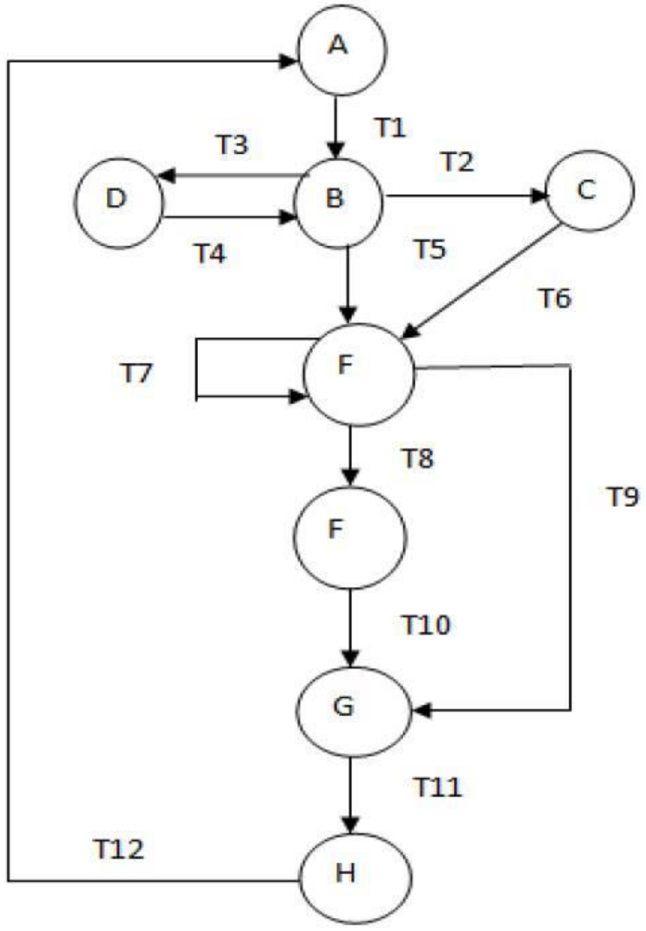
The number of independent paths is given by

V (G) = e - n + 2

where n is the number of nodes and e is the number of arcs/edges

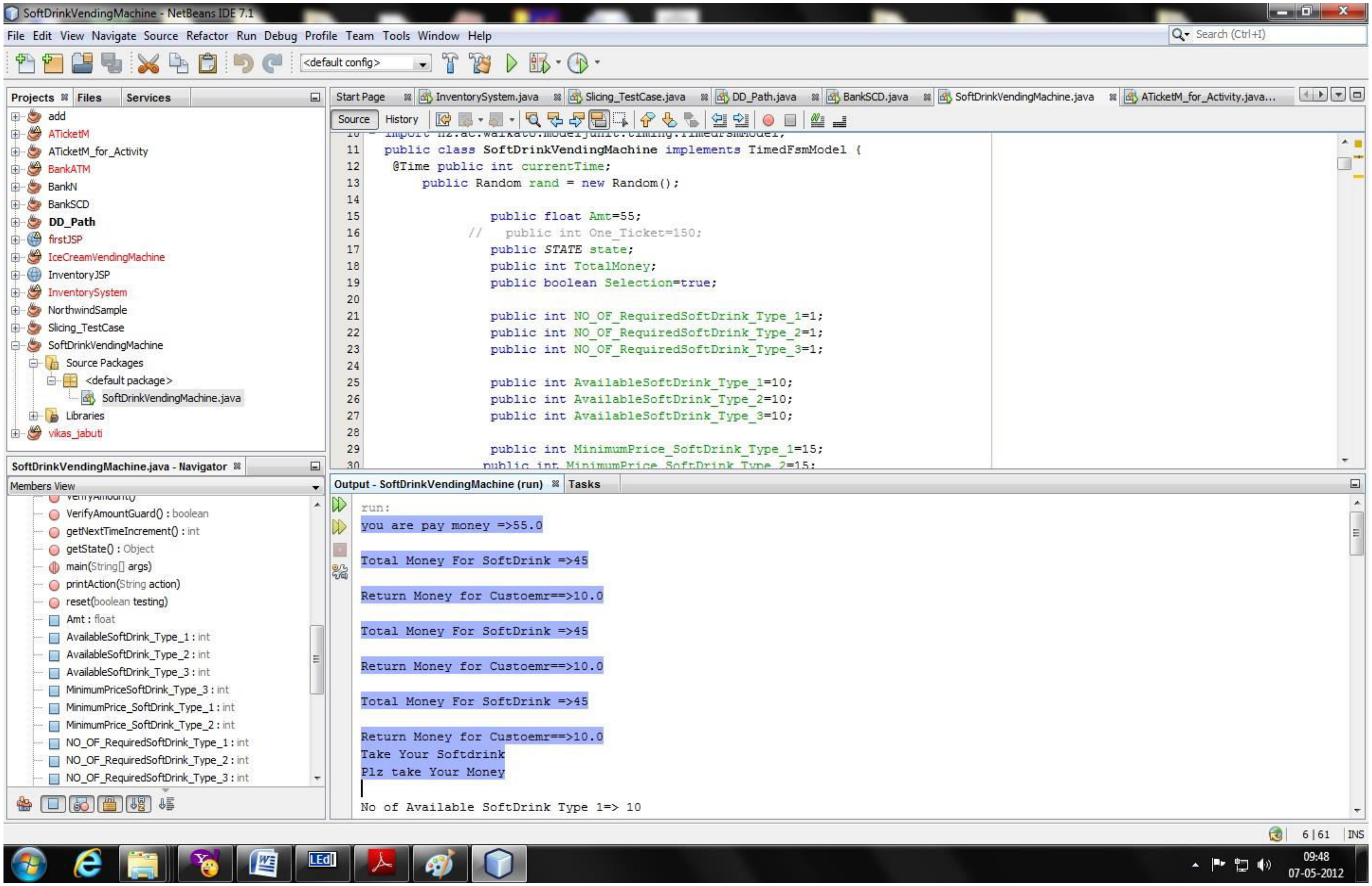
No of Edges = 12
No. Of Nodes = 8

V(G) = 12 - 8 + 2
V(G) = 6



State graph of SoftDrink Vending Machine

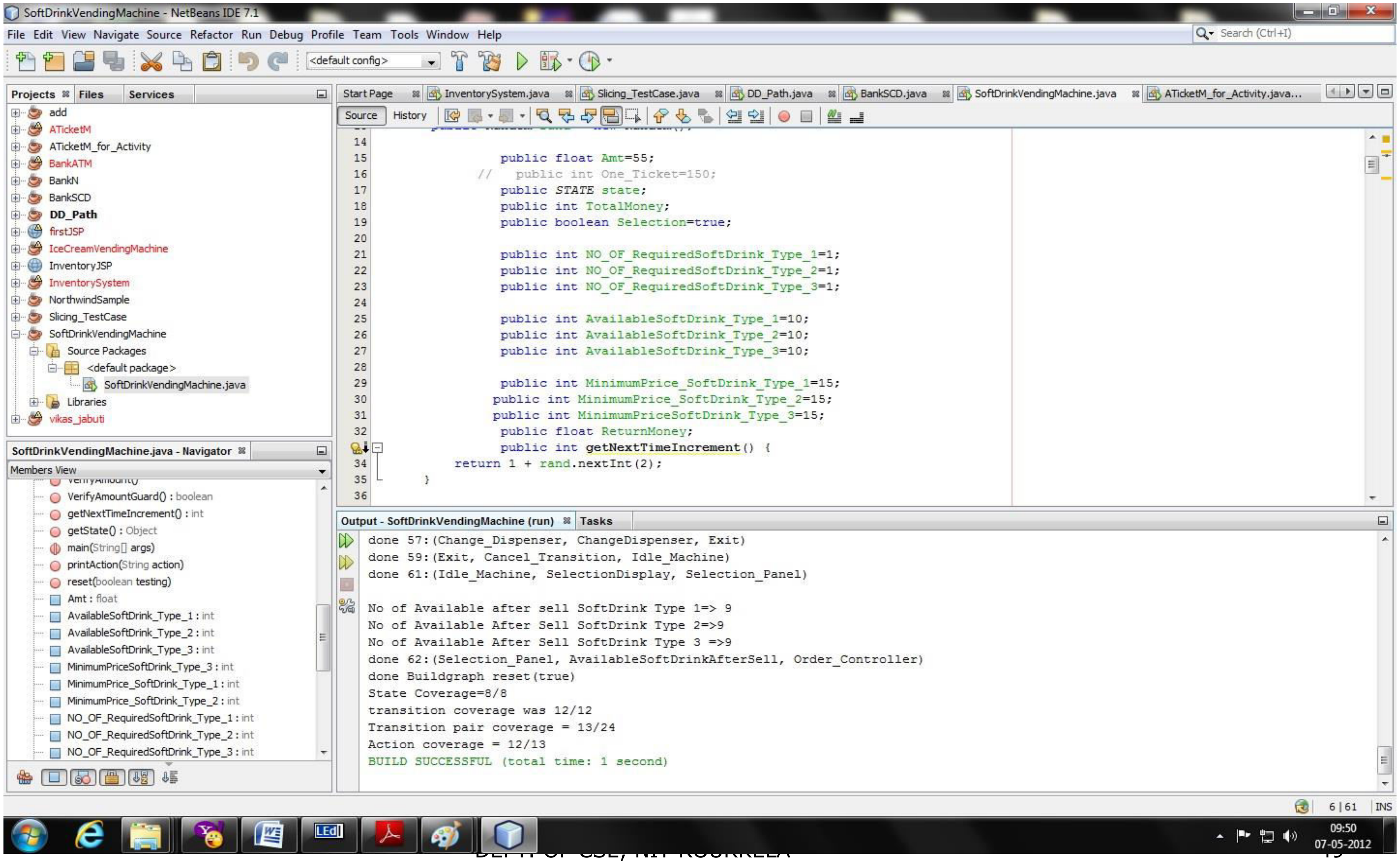
Source code for state graph


SoftDrinkVendingMachine - NetBeans IDE 7.1
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default config>
Projects Files Services
add
ATicketM
ATicketM_for_Activity
BankATM
BankN
BankSCD
DD_Path
firstJSP
IceCreamVendingMachine
InventoryJSP
InventorySystem
NorthwindSample
Slicing_TestCase
SoftDrinkVendingMachine
Source Packages
<default package>
SoftDrinkVendingMachine.java
Libraries
vikas_jabuti
Start Page
InventorySystem.java
Slicing_TestCase.java
DD_Path.java
BankSCD.java
SoftDrinkVendingMachine.java
ATicketM_for_Activity.java...
Source History
SoftDrinkVendingMachine.java - Navigator
Members View
VerifyAmountGuard() : boolean
getNextTimeIncrement() : int
getState() : Object
main(String[] args)
printAction(String action)
reset(boolean testing)
Amt : float
AvailableSoftDrink_Type_1 : int
AvailableSoftDrink_Type_2 : int
AvailableSoftDrink_Type_3 : int
MinimumPriceSoftDrink_Type_3 : int
MinimumPrice_SoftDrink_Type_1 : int
MinimumPrice_SoftDrink_Type_2 : int
NO_OF_RequiredSoftDrink_Type_1 : int
NO_OF_RequiredSoftDrink_Type_2 : int
NO_OF_RequiredSoftDrink_Type_3 : int
Output - SoftDrinkVendingMachine (run)
Tasks
6 | 61
INS
09:48
07-05-2012

```java
import nz.ac.waikato.modeljunit.timing.TimedFsmModel;
public class SoftDrinkVendingMachine implements TimedFsmModel {
 @Time public int currentTime;
     public Random rand = new Random();

              public float Amt=55;
         //   public int One_Ticket=150;
              public STATE state;
              public int TotalMoney;
              public boolean Selection=true;

              public int NO_OF_RequiredSoftDrink_Type_1=1;
              public int NO_OF_RequiredSoftDrink_Type_2=1;
              public int NO_OF_RequiredSoftDrink_Type_3=1;

              public int AvailableSoftDrink_Type_1=10;
              public int AvailableSoftDrink_Type_2=10;
              public int AvailableSoftDrink_Type_3=10;

              public int MinimumPrice_SoftDrink_Type_1=15;
             public int MinimumPrice_SoftDrink_Type_2=15;
```

```
run:
you are pay money =>55.0

Total Money For SoftDrink =>45

Return Money for Custoemr==>10.0

Total Money For SoftDrink =>45

Return Money for Custoemr==>10.0

Total Money For SoftDrink =>45

Return Money for Custoemr==>10.0
Take Your Softdrink
Plz take Your Money

No of Available SoftDrink Type 1=> 10
```

Coverage for state graph

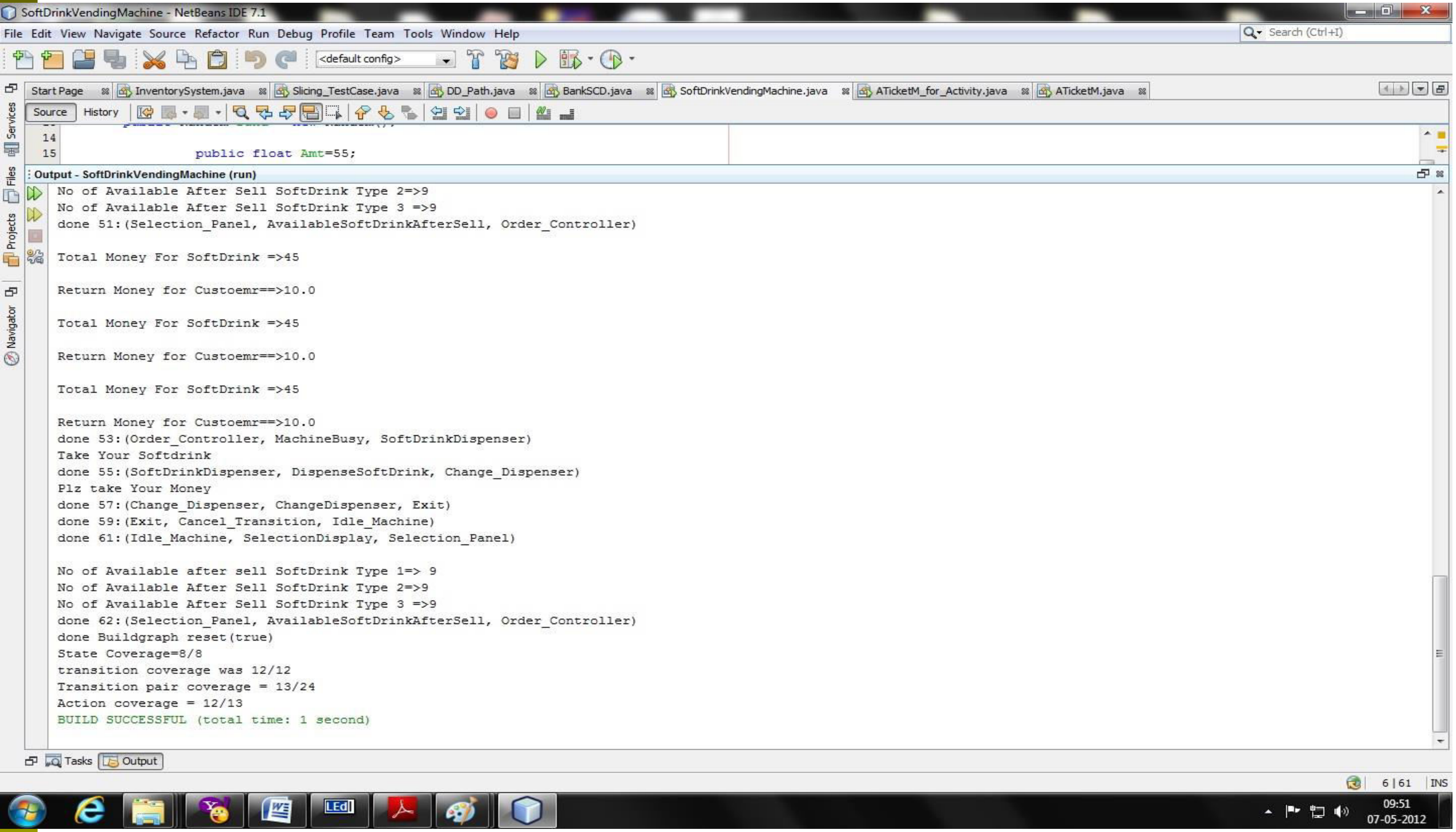


SoftDrinkVendingMachine - NetBeans IDE 7.1
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Projects Files Services
SoftDrinkVendingMachine.java - Navigator
Output - SoftDrinkVendingMachine (run) Tasks

```
        public float Amt=55;
      //   public int One_Ticket=150;
        public STATE state;
        public int TotalMoney;
        public boolean Selection=true;

        public int NO_OF_RequiredSoftDrink_Type_1=1;
        public int NO_OF_RequiredSoftDrink_Type_2=1;
        public int NO_OF_RequiredSoftDrink_Type_3=1;

        public int AvailableSoftDrink_Type_1=10;
        public int AvailableSoftDrink_Type_2=10;
        public int AvailableSoftDrink_Type_3=10;

        public int MinimumPrice_SoftDrink_Type_1=15;
       public int MinimumPrice_SoftDrink_Type_2=15;
       public int MinimumPriceSoftDrink_Type_3=15;
        public float ReturnMoney;
        public int getNextTimeIncrement() {
  return 1 + rand.nextInt(2);
    }
```

```
done 57:(Change_Dispenser, ChangeDispenser, Exit)
done 59:(Exit, Cancel_Transition, Idle_Machine)
done 61:(Idle_Machine, SelectionDisplay, Selection_Panel)

No of Available after sell SoftDrink Type 1=> 9
No of Available After Sell SoftDrink Type 2=>9
No of Available After Sell SoftDrink Type 3 =>9
done 62:(Selection_Panel, AvailableSoftDrinkAfterSell, Order_Controller)
done Buildgraph reset(true)
State Coverage=8/8
transition coverage was 12/12
Transition pair coverage = 13/24
Action coverage = 12/13
BUILD SUCCESSFUL (total time: 1 second)
```

Output With Test Coverage & Test Sequence



SoftDrinkVendingMachine - NetBeans IDE 7.1
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Start Page
InventorySystem.java
Slicing_TestCase.java
DD_Path.java
BankSCD.java
SoftDrinkVendingMachine.java
ATicketM_for_Activity.java
ATicketM.java
Source
History
Output - SoftDrinkVendingMachine (run)
Tasks
Output

```
public float Amt=55;
```

```
No of Available After Sell SoftDrink Type 2=>9
No of Available After Sell SoftDrink Type 3 =>9
done 51:(Selection_Panel, AvailableSoftDrinkAfterSell, Order_Controller)

Total Money For SoftDrink =>45

Return Money for Custoemr==>10.0

Total Money For SoftDrink =>45

Return Money for Custoemr==>10.0

Total Money For SoftDrink =>45

Return Money for Custoemr==>10.0
done 53:(Order_Controller, MachineBusy, SoftDrinkDispenser)
Take Your Softdrink
done 55:(SoftDrinkDispenser, DispenseSoftDrink, Change_Dispenser)
Plz take Your Money
done 57:(Change_Dispenser, ChangeDispenser, Exit)
done 59:(Exit, Cancel_Transition, Idle_Machine)
done 61:(Idle_Machine, SelectionDisplay, Selection_Panel)

No of Available after sell SoftDrink Type 1=> 9
No of Available After Sell SoftDrink Type 2=>9
No of Available After Sell SoftDrink Type 3 =>9
done 62:(Selection_Panel, AvailableSoftDrinkAfterSell, Order_Controller)
done Buildgraph reset(true)
State Coverage=8/8
transition coverage was 12/12
Transition pair coverage = 13/24
Action coverage = 12/13
BUILD SUCCESSFUL (total time: 1 second)
```

EFSM graph for State Graph

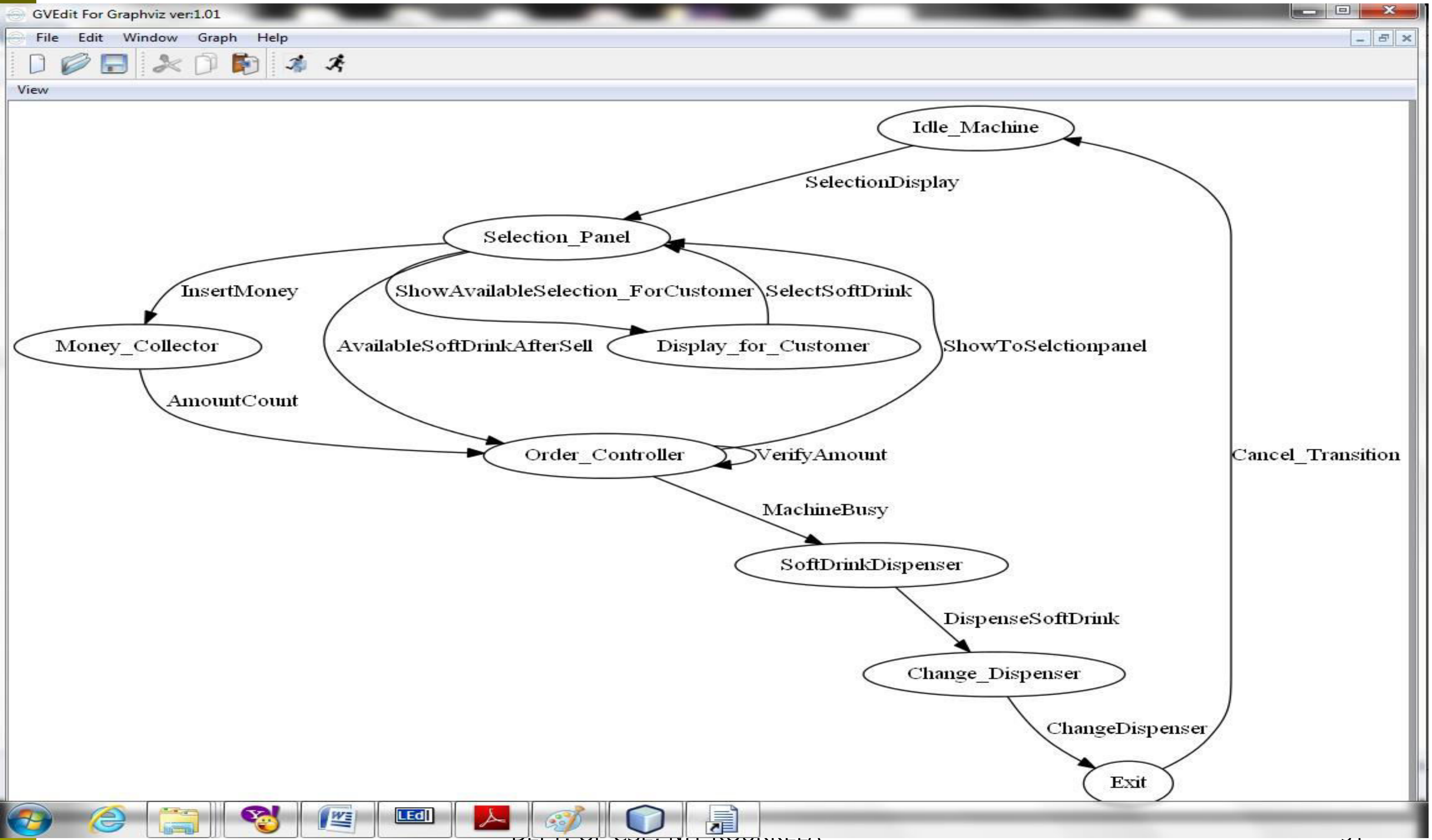
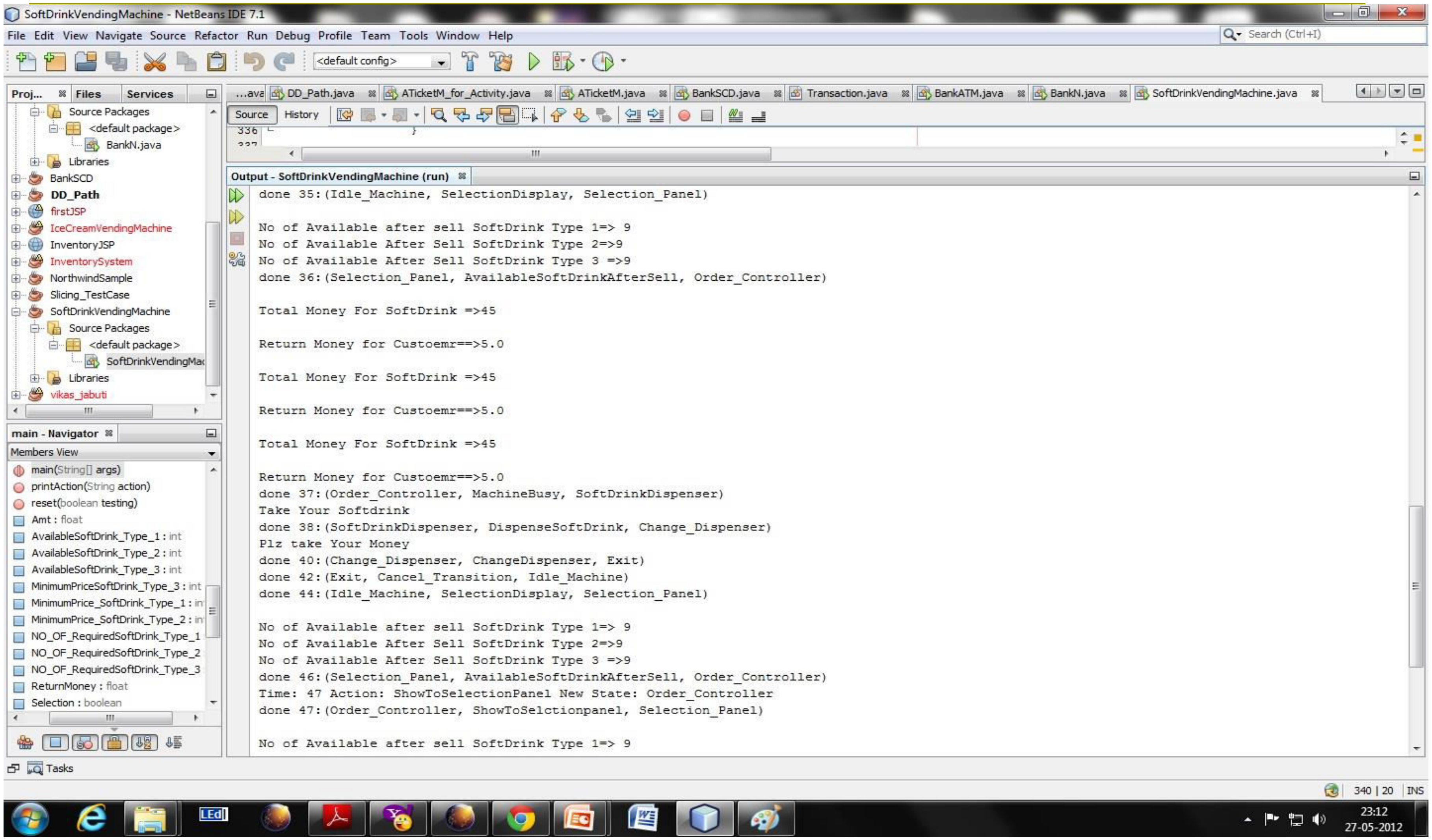


Table 1: Table Showing Test Input with Expected Output & observed Output

TC_ID	**INPUT**					**Expected**	**Observed**
ID	Amt	Selection	N1	N2	N3	Output	Output
1.	50	T	1	1	1	Issue Ticket, Return Money	Issue Ticket, Return Money
2.	35	T	4	1	5	Return Money	Return Money
3.	45	T	1	1	1	Issue Ticket	Issue Ticket
4.	100	F	1	1	3	Not Select Item	Not Select Item
5.	500	T	10	10	10	Return Money	Return Money
6.	200	T	0	0	0	Null Item Select	Null Item Select

Test Sequence 1 for TC_ID 1 Output Snapshot:

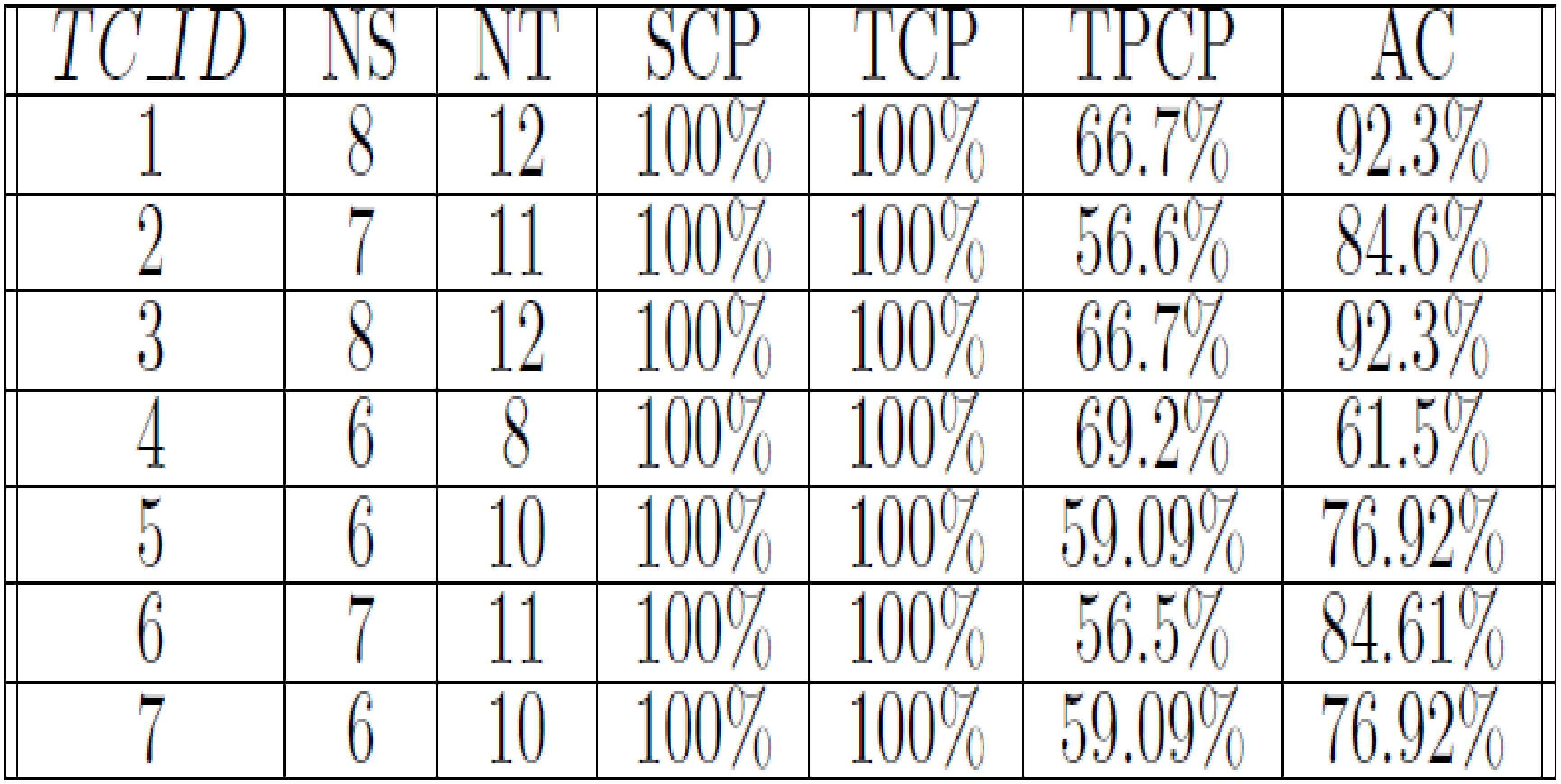

SoftDrinkVendingMachine - NetBeans IDE 7.1
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Output - SoftDrinkVendingMachine (run)

```
done 35:(Idle_Machine, SelectionDisplay, Selection_Panel)

No of Available after sell SoftDrink Type 1=> 9
No of Available After Sell SoftDrink Type 2=>9
No of Available After Sell SoftDrink Type 3 =>9
done 36:(Selection_Panel, AvailableSoftDrinkAfterSell, Order_Controller)

Total Money For SoftDrink =>45

Return Money for Custoemr==>5.0

Total Money For SoftDrink =>45

Return Money for Custoemr==>5.0

Total Money For SoftDrink =>45

Return Money for Custoemr==>5.0
done 37:(Order_Controller, MachineBusy, SoftDrinkDispenser)
Take Your Softdrink
done 38:(SoftDrinkDispenser, DispenseSoftDrink, Change_Dispenser)
Plz take Your Money
done 40:(Change_Dispenser, ChangeDispenser, Exit)
done 42:(Exit, Cancel_Transition, Idle_Machine)
done 44:(Idle_Machine, SelectionDisplay, Selection_Panel)

No of Available after sell SoftDrink Type 1=> 9
No of Available After Sell SoftDrink Type 2=>9
No of Available After Sell SoftDrink Type 3 =>9
done 46:(Selection_Panel, AvailableSoftDrinkAfterSell, Order_Controller)
Time: 47 Action: ShowToSelectionPanel New State: Order_Controller
done 47:(Order_Controller, ShowToSelctionpanel, Selection_Panel)

No of Available after sell SoftDrink Type 1=> 9
```

Test Sequence 1 for TC_ID 1:

- done :(Idle_Machine, SelectionDisplay, Selection_Panel)
- done :(Selection_Panel, ShowAvailableSelection_ForCustomer, Display_for_Customer)
- done :(Display_for_Customer, SelectSoftDrink, Selection_Panel)
- done :(Selection_Panel, InsertMoney, Money_Collector)
- done :(Money_Collector, AmountCount, Order_Controller)
- done :(Order_Controller, VerifyAmount, Order_Controller)
- done :(Order_Controller, ShowToSelctionpanel, Selection_Panel)
- done :(Selection_Panel, AvailableSoftDrinkAfterSell, Order_Controller)
- done :(Order_Controller, MachineBusy, SoftDrinkDispenser)
- done :(SoftDrinkDispenser, DispenseSoftDrink, Change_Dispenser)
- done :(Change_Dispenser, ChangeDispenser, Exit)

Table 2: TABLE Showing Test Coverage Achieved



TC_ID	NS	NT	SCP	TCP	TPCP	AC
1	8	12	100%	100%	66.7%	92.3%
2	7	11	100%	100%	56.6%	84.6%
3	8	12	100%	100%	66.7%	92.3%
4	6	8	100%	100%	69.2%	61.5%
5	6	10	100%	100%	59.09%	76.92%
6	7	11	100%	100%	56.5%	84.61%
7	6	10	100%	100%	59.09%	76.92%

Validation Testing

- Validation succeeds when software functions in a manner that can be reasonably expected by the customer.
- Focus on user-visible actions and user-recognizable outputs
- Details of class connections disappear at this level
- Apply:
  - Use-case scenarios from the software requirements specifications
  - Black-box testing to create a deficiency list
  - Acceptance tests through alpha (at developer's site) and beta (at customer's site) testing with actual customers

System Testing

- Finally ,the system as a whole is tested to ensure that errors in requirements are uncovered .
- **Types of System Testing**:
  - **Recovery testing**: how well and quickly does the system recover from faults
  - **Security testing**: verify that protection mechanisms built into the system will protect from unauthorized access (hackers, disgruntled employees, fraudsters)
  - **Stress testing**: place abnormal load on the system
  - **Performance testing**: investigate the run-time performance within the context of an integrated system

Testing Surface Structure

- Surface structure refers to the externally observable structure of an OO program.
- Here tests are based on user tasks, no matter whatever is the interface.
- Capturing these tasks involves understanding watching, and talking with representative user (also non-representative users)

Ex-In a conventional system with a command-oriented interface, the user might use the list of all commands as a testing checklist. If no test scenarios existed to exercise a command, testing has likely overlooked some user tasks (or the interface has useless commands).

- Whatever the interface style, test case design that exercises the surface structure should use both objects and operations as clues leading to overlooked tasks.

Testing Deep Structure

- Deep structure refers to the internal technical details of an OO program.
- Deep structure testing is designed to exercise dependencies, behaviors, and communication mechanisms that have been established as part of the subsystem and object design of OO software.
- Analysis and design models are used as the basis for deep structures testing.

Examples- the object-relationship diagram or the subsystem collaboration graph depicts collaborations between objects and subsystems that may not be externally visible. Test case designer checks whether some task exercises the collaboration noted on the object-relationship diagram or the sub system collaboration graph, if not then why not ?

Test Case Design For OO Software

1. Each test case should be uniquely identified and should be explicitly associated with the class to be tested
2. The purpose of the test should be stated
3. A list of testing steps should be developed for each test and should contain :
    a. A list of specified states for the object that is to be tested
    b. A list of messages and operations that will be exercised as a consequence of the test

Contd…

a. A list of exceptions that may occur as the object is tested
b. A list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test)
c. Supplementary information that will aid in understanding or implementing the test

Challenges in Test Case Design

- **Encapsulation:**
  - Difficult to obtain a snapshot of a class without building extra methods which display the classes’ state
- **Inheritance:**
  - Each new context of use (subclass) requires re-testing because a method may be implemented differently (polymorphism).
  - Other unaltered methods within the subclass may use the redefined method and need to be tested

Automated Testing Tools

- Mercury Interactive
  - Quick Test Professional: Regression testing
  - WinRunner: UI testing
- IBM Rational
  - Rational Robot
  - Functional Tester
- Borland
  - Silk Test
- Compuware
  - QA Run
- AutomatedQA
  - TestComplete

Automated Testing Tools

- **Mercury Interactive**
  - Quick Test Professional (QTP): Regression testing
  - WinRunner: Functional / Regression / UI testing
  - LoadRunner: Performance and Load testing of Client-Server applications
  - TestDirector: Web based Test mgt tool;

    Advantage: can be used when 2 testing teams are located at different locations.
- **IBM Rational**
  - Rational Robot: Functional / Regression testing
  - Rational Functional Tester (RFT): Functional / Regression testing
  - Rational Quality Software: Test Suit Management
- **Segue Software**
  - Silk Test: Functional / Regression testing,

    Advantage: It has a provision for customized in-built recovery system

Automated Testing Tools

- **Compuware**
  - QA Run
- **AutomatedQA**
  - TestComplete: UI testing / Regression testing
- **Apache’s**
  - JMeter: an open source software used for performance and load testing .

Summary

- Testing is integrated with and affects all stages of the Software Engineering lifecycle
- Strategies: a bottom-up approach – class, integration, validation and system level testing
- Techniques:
  - white box (look into technical internal details)
  - black box (view the external behavior)

References

1. R. S. Pressman, Software Engineering, McGraw-Hill, 2018.

Thank You