COMP 354: INTRODUCTION TO SOFTWARE ENGINEERING

PROJECT MANAGEMENT APPLICATION
TEST DESIGN DOCUMENTATION

Team B

Alex Huot

Dmitri Novikov

David Gurnsey

Jun Hui Chen

Philip Eloy

Robert Wolfstein

Nicholas Francoeur

Mehdi Jamal Mokhtar

Nivine Shebarou

Lee Kelly

Steven Di Lullo

Tuesday, August 26th , 2014

Revision History:

Date Created: Aug 17th, 2014

Most Recent Update: Aug 26th, 2014

Previous Updates:

Revision Date	**Contributor**	**Summary**
Aug 26th, 2014	Alex Huot	Created doc based on passed iteration template.

Table of Contents

List of Figures

Document Description:

Introduction:

This document provides design documentation for Team B's third iteration deliverable. The chosen functions to test, justification for said choices, and all test-design is provided here. The test implementations are included in a separate package in the final source code submission.

A note on selected functions:

The requirements for this iteration's testing were to select 5 functions suitable for both white-box and black-box testing. Unfortunately, without important changes to the logic of our code, we had no such functions to test. In fact, several of our black-box functions are segments of code originally part of larger algorithms, which we extracted in the form of helper functions ideal for this type of test. Despite this, tests suitable for black-box had very simple decision logic, and those suitable for white-box seldom took more than 1 parameter with a specifiable range. For this reason, we opted to perform white-box and black-box tests on a different set of 5 functions for each (save for one function). We felt this would better demonstrate our understanding of these testing principles, and offer more interesting results for discussion.

1.0 Choice of Functions, Justification and Input Ranges

This section outlines the choice of our 5 black-box and 5 white-box functions. Here we also provide a short description of each function and the justification for our choices. The input ranges are specified for each.

2.0 White Box Testing

The Control Flow Graphs (CFGs), cyclomatic-complexity and **basis-path coverage** calculations for our white-box functions are provided here.

3.0 Black Box Testing

This section provides the wort-case boundary testing design for our black-box tests. As we are not performing any white-box tests for these functions, we opted to omit the CFGs and basis-path coverage calculations for these functions. The black-box tests are performed using the **worst-case boundary value-analysis** method.

4.0 Test Implementation & Headers

A brief description on how each test case was implemented, and the information contained in each header.

5.0 Discussion

The concluding discussion of insights gained from both test-design as well as implementation.

1.0 Choice of Functions, Justification and Input Ranges

1.1 Choice of White-Box Functions

The 5 selected functions for our white-box tests are:

1. Class: Activity

```
public static boolean areValidTimes(double mostLikely, double optimistic,
        double pessimistic)
```

Description:

Used by `Activity's` constructor, this function checks that the `Activities`' projected times (`mostLikely`, `optimistic`, `pessimistic`) are within suitable ranges.

Justification:

This function is the only function we deemed acceptable for both white- and black-box testing. The justification for the black-box version is listed in section 1.2. This function is ideal for white-box as it entails a series of conditional checks to determine the validity of inputs. The importance of this function paired with its decision logic make it an ideal candidate for white-box tests.

2. `Class: MilestoneNode`

```
public boolean equals(Object obj)
```

Description:

This is an override of the `Object` classes `equals` function, and tests the equality between 2 `MilestoneNodes`.

Justification:

Although this function is simple in theory, it is an important one to test, as it performs an important task with somewhat confusing decision logic.

Input Ranges:

This function takes an `Object` as input, and thus we can pass any object we like to ensure every basis-path is covered.

3. Class: NodeGraph

```
public void addEdge(MilestoneNode v, MilestoneNode w)
```

Description:

This function adds an edge to the `NodeGraph`, connecting `MilestoneNodes` v and w.

Justification:

When building node graphs, there is a possibility that certain nodes have already been added. This coupled with the existence (or lack) of dependencies, we encounter a series of interesting conditions on which to base a white-box test.

Input Ranges:

This function depends on the state of the `NodeGraph` being built, as well as the relationship between both `MilestoneNodes` passed as inputs. These relationships can be manufactured manually in order to test all basis paths.

4. `Class: EarnedValue`

   `public void calculateActualCost()`

   Description:

   Calculates the actual cost for a `Project` based on all it's activities and their percentage of completion.

   Justification:

   The input of an `Activity` list are what make this function structurally ideal for white-box testing. Several potential paths, perfect for white-box.

   Input Ranges:

   Takes no parameters; all calculations are performed on the `EarnedValue` object's `Project` object and its `Activities`

5. Class: `MilestoneNode`

```
public String toStringArrows(String type)
```

Description:

This function generates a string describing the `MilestoneNode`'s related activities, namely those to complete by the milestone, or those available after the milestone.

Justification:

As with previous white-box tests, the decision logic entailed by this function make it an interesting candidate. It relies on both the type of arrows the caller wishes to consider, as well as the relationship between the `MilestoneNode` and other acitivities.

Input Ranges:

The function takes a `type` as input, and depending on this type and the `MilestoneNode`'s local `Activity` list, a descriptive string will be generated.

1.2 Choice of Black-Box Functions

The 5 selected functions for our black-box tests are:

1. Class: EarnedValue

```
public static double getActivityScheduleValue(double activityEarlyFinish,
      double activityLateFinish, double activityPlannedVal,
      double activityDuration, double daysSinceStart)
```

Description:

This function is important for calculating the **scheduled value** of an activity during an Earned Value Analysis procedure.

Justification:

By taking several numerical inputs with specifiable range, this function is an ideal candidate for worst-case boundary value analysis.

Input Ranges:

```
double activityEarlyFinish = [0, 731] // Days. Max set to 2 years
double activityLateFinish  = [0, 731] // Days. Max set to 2 years
double activityPlannedVal  = [0, 1000000] // $. Max set to 1 million
double activityDuration    = [0, 731] // Days. Max set to 2 years
double daysSinceStart      = [0, 73] // Days. Max set to 2 years less a day
```

2. `Class: Project`

```
public static boolean areValidPercentages(double percentage1,
        double percentage2)
```

Description:

This is a validity checking function for use in the constructor of `Project`, and verifies that the input percentages are valid. It is necessary to ensure data integrity and proper program functionality.

Justification:

This checks that the `Project's` percentages are within suitable ranges, making it ideal for boundary-value analysis.

Input Ranges:

```
double percentage1 = [0, 1] // Percentage in decimal format
double percentage2 = [0, 1] // Percentage in decimal format
```

3. Class: Project

```
public static boolean areValidValues(double budgetAtCompletion,
        double actualCost, double earnedValue)
```

Description:

This function checks that the Project's values (`budgetAtCompletion`, `actualCost` and `earnedValue`) are within suitable ranges.

Justification:

This function performs an important check on the bounds of the specified values. Ideal for black-box testing.

Input Ranges:

```
double budgetAtCompletion = [0, 10 000 000] // $. Max fixed to 10 million
double actualCost         = [0, 10 000 000] // $. Max fixed to 10 million
double earnedValue        = [0, 10 000 000] // $. Max fixed to 10 million
```

4. Class: `Activity`

```
public static boolean areValidTimes(double mostLikely, double optimistic,
        double pessimistic)
```

Description:

Used by `Activity's` constructor, this function checks that the `Activities`' projected times (`mostLikely`, `optimistic`, `pessimistic`) are within suitable ranges

Justification:

These are bounded values, making them ideal for black-box testing

Input Ranges:

```
double mostLikely  = [0, 731] // Days. Max set to 2 years

double optimistic  = [0, 731] // Days. Max set to 2 years

double pessimistic = [0, 731] // Days. Max set to 2 years
```

5. Class: Activity

```
public static boolean areValidPercentAndCost(double percentComplete,
        double actualCost)
```

Description:

Verifies the specified values passed to `Activity` constructor are valid.

Justification:

These values are bounded, making them ideal for black-box testing.

Input Ranges:

```
double percentComplete = [0, 1] // % in decimal format
double actualCost      = [0, 10 000 000] // $. Max set to 10 million
```

2.0 White-Box Testing

For each of the above specified white-box functions, the following information is provided:

- Code Snippet
- Control Flow Graph
- Cyclomatic-complexity
- Basis-path coverage test suite
    - For each test case, the input required to follow this path is provided in bullet form; each bullet represents the necessary input at the specified decision node needed to go along the correct path. If there exists a contradictory input, the test case will be labeled **INFEASIBLE** with a brief explanation.

2.1 Activity – areValidtimes()

```java
// Class:   Activity
    public static boolean areValidTimes(double mostLikely, double optimistic,
            double pessimistic) {

        if (mostLikely < 0 || optimistic < 0 || pessimistic < 0) {
            return false;
        }

        if (mostLikely > MAX_DURATION || optimistic > MAX_DURATION
                || pessimistic > MAX_DURATION) {
            return false;
        }

        if (mostLikely > pessimistic || mostLikely < optimistic) {
            return false;
        }

        return true;
    }
```

Figure 1: areValidTimes() - Code

```
Class Activity
private void areValidTimes(double mostLieky, double optmistic, double pessimistic)
```

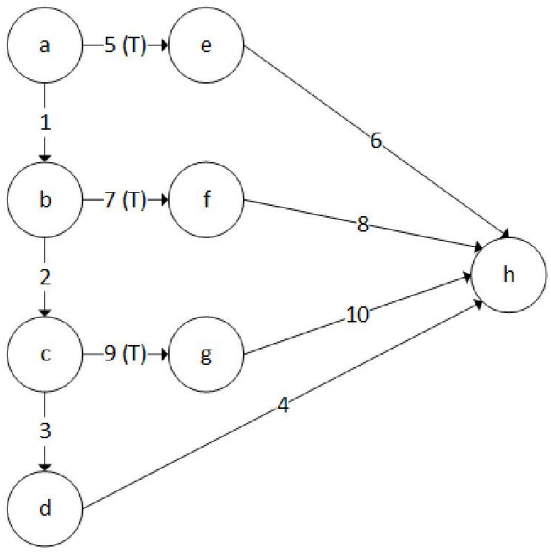

Figure 2: areValidTimes() - CFG

Cyclomatic-Complexity:

Complexity = num. Decision nodes + 1

= 3 + 1

= **4**

Basis-Path Coverage:

Test Cases: {

1: <1,2,3,4>

2: <5,6>

3: <1,7,8>

5: <1,2,9>

}

Test 1 Input:

- node a: All inputs greater than 0
- node b: All inputs less than Activity.MAX_DURATION
- node c: most likely less than pessimistic and more than optimistic

Test 2 Input:

- node a: At least 1 input less than 0

Test 3 Input:

- node a: All inputs greater than 0
- node b: At east 1 input greater than Activity.MAX_DURATION

Test 4 Input:

- node a: All inputs greater than 0
- node b: All inputs less than Activity.MAX_DURATION
- node c: most likely greater than pessimistic or less than optimistic

2.2 MilestoneNode – equals()

```
// Class:   MilestoneNode
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    MilestoneNode other = (MilestoneNode) obj;
    if (name == null) {
        if (other.name != null)
            return false;
    } else if (!name.equals(other.name)) {
        return false;
    }
    return true;
}
```

Figure 3: equals() - Code

```
Class MilestoneNode
private void equals(Object obj)
```

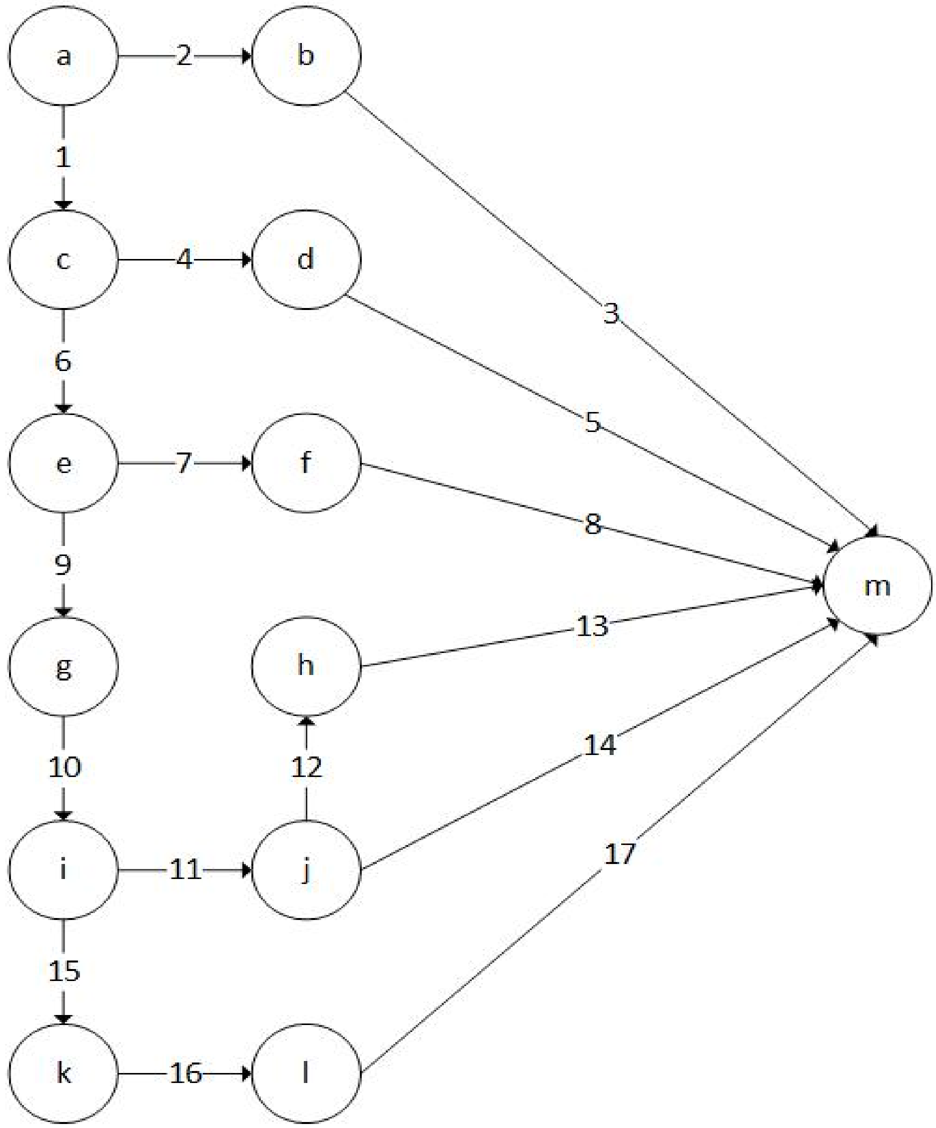

Figure 4: equals() - CFG

Cyclomatic-Complexity:

Complexity = num. Decision nodes + 1

= 6 + 1

= **7**

Basis-Path Coverage:

Test Cases: {

1: <1,6,9,10,15,16,18,19>

2: <2,3>

3: <1,4,5>

4: <1,6,7,8>

5: <1,6,9,10,11,12,13>

6: <1,6,9,10,11,12,14>

7: <1,6,9,10,15,16,17>

}

Test 1 Input:

- node a: obj is NOT equal to calling MilestoneNode
- node c: obj is NOT null
- node e: obj class is of type MilestoneNode
- node i: Caller's name NOT null
- node l: obj name is same as calling MilestoneNode

Test 2 Input:

- node a: obj is equal to calling MilestoneNode

Test 3 Input:

- node a: obj is not equal to calling MilestoneNode
- node c: obj is null

Test 4 Input:

- node a: obj is not equal to calling MilestoneNode
- node c: obj is not null
- node e: obj class is NOT of type MilestoneNode

Test 5 Input:

- node a: obj is NOT equal to calling MilestoneNode
- node c: obj is NOT null
- node e: obj class is of type MilestoneNode
- node i: Caller's name is null

INFEASIBLE - Caller's name always initialized

(**Note**: our overridden version is meant to fully match the default version of the equals function. This is why verifying the Caller's name not equal null remains)

Test 6 Input:

- node a: obj is NOT equal to calling MilestoneNode
- node c: obj is NOT null
- node e: obj class is of type MilestoneNode
- node i: Caller's name is null

INFEASIBLE - Caller's name always initialized

Test 7 Input:

- node a: obj is NOT equal to calling MilestoneNode
- node c: obj is NOT null
- node e: obj class is of type MilestoneNode
- node i: Caller's name NOT null
- node l: obj name is NOT same as calling MilestoneNode

2.3 NodeGraph – addEdge()

```java
// Class:   NodeGraph
public void addEdge(MilestoneNode v, MilestoneNode w) {

        if (!hasNode(v)) {
            nodes.put(v, new ArrayList<MilestoneNode>());
        }
        if (!nodes.get(v).contains(w)) {
            nodes.get(v).add(w);
        }
        if (!v.getDependents().contains(w)) {
            v.addDependent(w);
        }
        if (!w.getPrecedents().contains(v)) {
            w.addPrecedent(v);
        }
    }
```

Figure 5: addEdge() - Code

```
Class NodeGraph
private void addEdge(MilestoneNode v, MilestoneNode w)
```

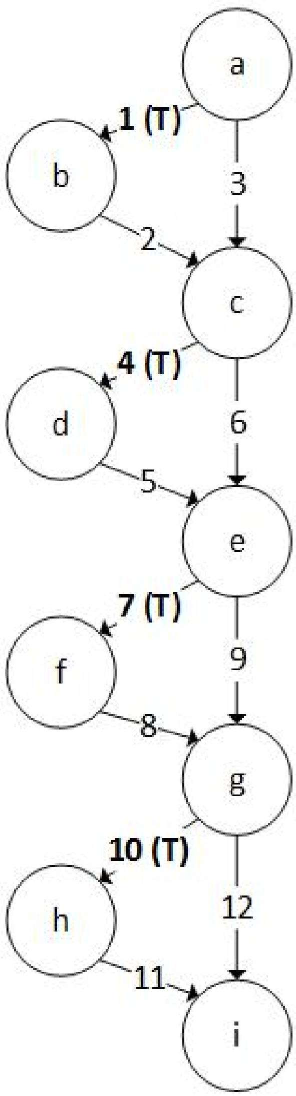

Figure 6: addEdge() - CFG

Cyclomatic-Complexity:

Complexity = num. Decision nodes + 1

= 4 + 1

= **5**

Basis-Path Coverage:

Test Cases: {

1: <1,2,4,5,7,8,10,11>

2: <3,4,5,7,8,10,11>

3: <1,2,6,7,8,10,11>

4: <1,2,4,5,9,10,11>

5: <1,2,4,5,7,8,12>

}

<u>Test 1 Input:</u>

- node a: NodeGraph DOES NOT contain v
- node c: v DOES NOT contain w
- node e: w is NOT dependent on v
- node g: v is NOT a precedent of w

Test 2 Input:

- node a: Nodegraph contains v
- node c: v DOES NOT contain w
- node e: w is NOT dependent on v
- node g: v is NOT a precedent of w

Test 3 Input:

- node a: NodeGraph DOES NOT contain v
- node c: v contains w

INFEASIBLE – IF V NOT ALREADY IN NODE GRAPH, CANNOT CONTAIN W

(**Note**: We know this to be true as these are the first and only inputs to the NodeGraph)

Test 4 Input:

- node a: NodeGraph DOES NOT contain v
- node c: v DOES NOT contain w
- node e: w is dependent on v
- node g: v is NOT a precedent of w

INFEASIBLE – IF W DEPENDS ON V, V MUST BE PRECEDENT OF W

(**Note**: We know this to be true as these are the first and only inputs to the NodeGraph)

Test 5 Input:

- node a: NodeGraph DOES NOT contain v
- node c: v DOES NOT contain w
- node e: w is NOT dependent on v
- node g: v is a precedent of w

INFEASIBLE – IF W DOEST NOT DEPEND ON V, V CAN'T BE PRECEDENT OF W

2.4 EarnedValue – calculateActualCost()

```java
// Class:   EarnedValue
private void calculateActualCost() {

    double ac = 0;

    for (Activity a : project.getActivityList()) {
        if (a.getStatus()) {
            ac += a.getActualCost();
        } else {
            if (a.getPercentComplete() > 0) {
                ac += a.getActualCost() * a.getPercentComplete();
            }
        }
    }

    project.setActualCost(ac);
}
```

Figure 7: calculateActualCost() - Code

```
Class EarnedValue
private void calculateActualCost()
```

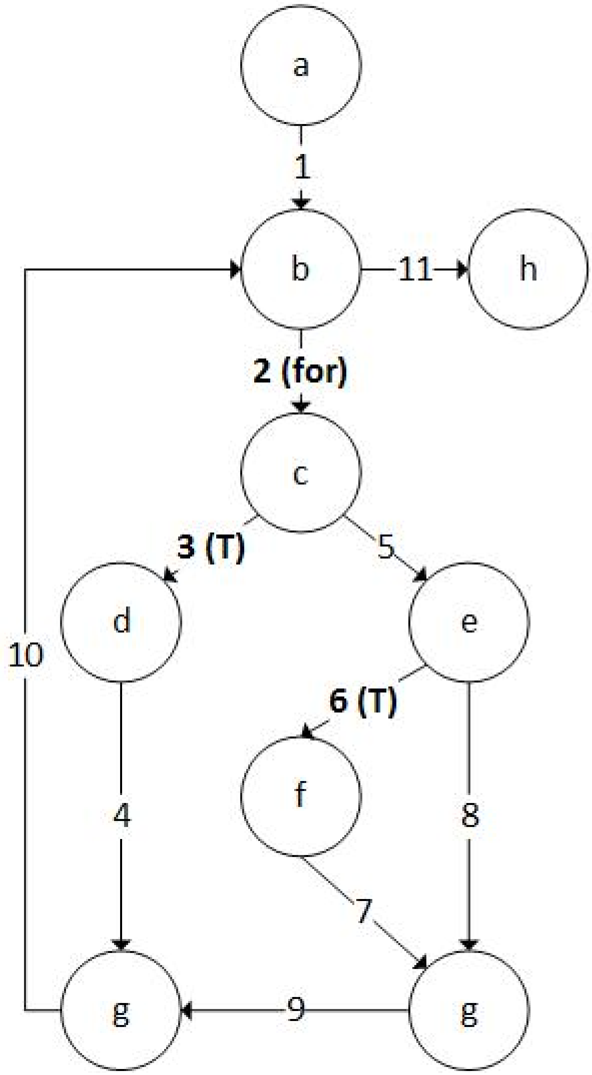

Figure 8: calculateActualCost() - CFG

Cyclomatic-Complexity:

Complexity = num. Decision nodes + 1

= 3 + 1

= **4**

Basis-Path Coverage:

Test Cases: {

1: <1,2,3,4,10,11>

2: <1,11>

3: <1,2,5,6,7,9,10,11>

4: <1,2,5,8,9,10,11>

}

<u>Test 1 Input:</u>

- node b: Project(p) has ONE Activity(a)
- node c: a.status = true

<u>Test 2 Input:</u>

- node b: Project(p) has NO Activities

Test 3 Input:

- node b: Project(p) has ONE Activity(a)
- node c: a.status = false
- node e: a.percentComplete > 0

Test 4 Input:

- node b: Project(p) has ONE Activity(a)
- node c: a.status = false
- node e: a.percentComplete = 0

2.5 MilestoneNode – toStringArrows()

```java
// Class:   MilestoneNode
public String toStringArrows(String type) {

        String  arrowString = "";
        ArrayList<Activity> arrows;

        if(type.equals("in")){
            arrowString = "Activities to Complete by this Milestone: ";
            if(this.hasInArrows()){
                arrows = this.getInArrows();
            }
            else{
                arrows = new ArrayList<Activity>();
                return "No in arrows";
            }

        } else if(type.equals("out")){
            arrowString  = "Activities to Available to start after this Milestone: ";
            if(this.hasOutArrows()){
                arrows = this.getOutArrows();
            }
            else{
                arrows = new ArrayList<Activity>();
                return "No out arrows";
            }

        } else {
            ec.showError("You didn't enter 'in' or 'out' when you should've.");
            return null;
        }

        for(Activity a : arrows){
            arrowString += a.getName() + ", ";
        }
        if(arrowString.endsWith(", ")){
            arrowString = arrowString.substring(0, arrowString.length()-2) + "\n\n";
        }
```

Figure 9: toStringArrows() - Code

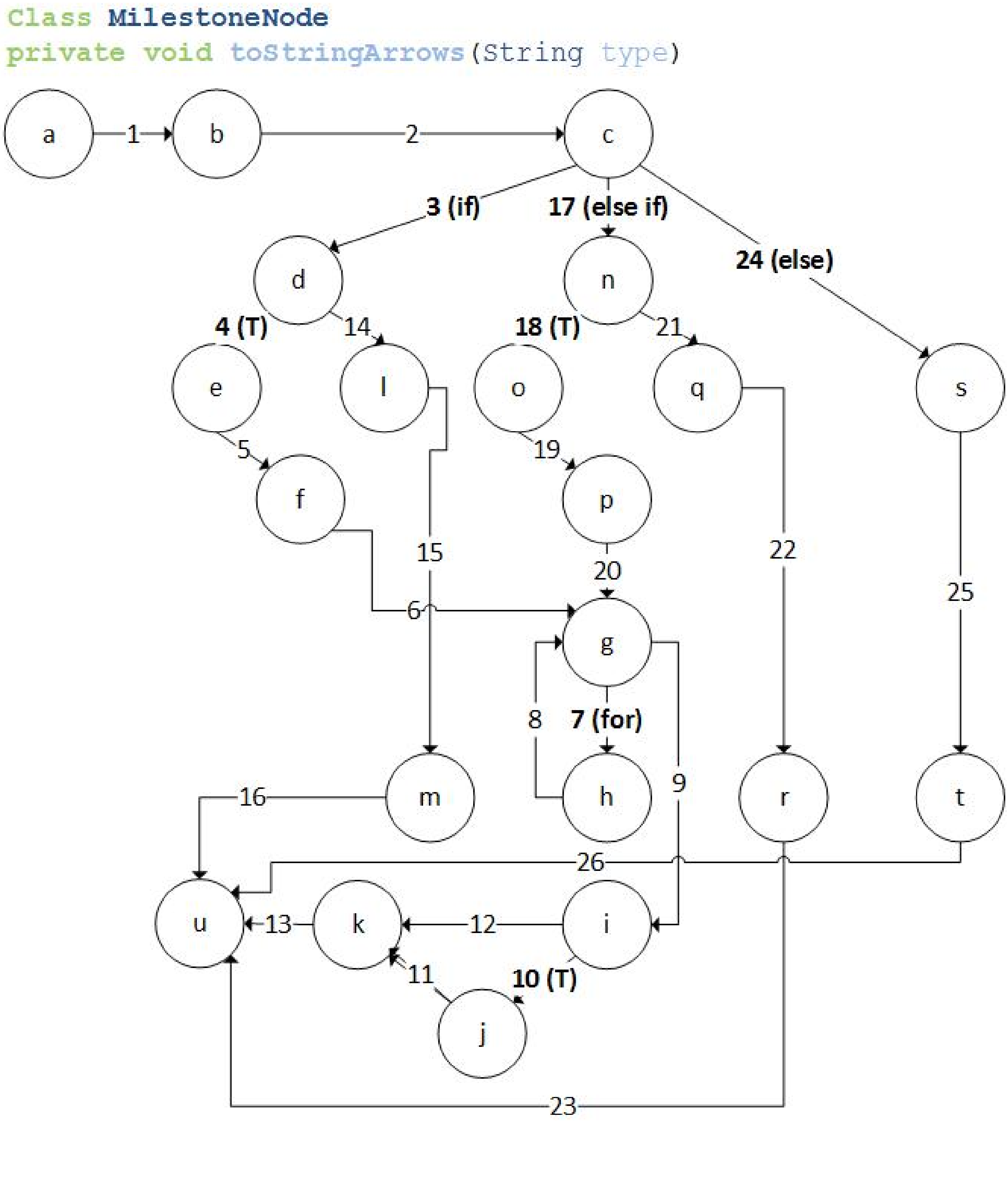

Figure 10: toStringArrows() - CFG

Cyclomatic-Complexity:

Complexity = num. Edges – num. nodes + 2

= 26 – 21 + 2

= **7**

Basis-Path Coverage:

Test Cases: {

1: <1,2,3,4,5,6,7,8,9,10,11,13>

2: <1,2,3,4,14,16>

3: <1,2,3,4,5,6,9,10,11,13>

4: <1,2,3,4,5,6,7,8,9,12,13>

5: <1,2,17,18,19,20,7,8,9,10,11,13>

6: <1,2,17,21,22,23>

7: <1,2,24,25,26>

}

<u>Test 1 Input:</u>

- node c: type = "in"
- node d: MilestoneNode has ONE inArrow
- node g: MilestoneNode has ONE inArrow
- node i: arrowString ends with ", "

Test 2 Input:

- node c: type = "in"
- node d: MilestoneNode has NO inArrow

Test 3 Input:

- node c: type = "in"
- node d: MilestoneNode has ONE inArrow
- node g: MilestoneNode has NO inArrows

INFEASIBLE – CONTRADICTION IN NUMBER OF INARROWS

Test 4 Input:

- node c: type = "in"
- node d: MilestoneNode has ONE inArrow
- node g: MilestoneNode has ONE inArrow
- node i: arrowString DOES NOT end with ", "

INFEASIBLE – ARROWSTRING WILL ALWAYS END IN "," WHEN ARROW EXISTS

Test 5 Input:

- node c: type = "out"
- node n: MilestoneNode has ONE outArrow
- node g: MilestoneNode has ONE outArrow
- node c: arrowString ends with ", "

Test 6 Input:

- node c: type = "out"
- node n: MilestoneNode has NO outArrows

Test 7 Input:

- node c: type not equal "in" or "out"

3.0 Black-Box Testing

Worst-case boundary-analysis testing requires that for each input variable we determine the 5 following values:

- **min** – the minimum value
- **min**+ – a value just above minimum
- **nominal** – that average value
- **max**- – slight below the maximum
- **max** – the maximum value

As worst-case analysis rejects the single-fault assumption, we must then test the function with every possible combination of the above . This results in $\mathbf{5^n}$ tests to run (where n is the number of parameters to the function).

For each function, a table containing these 5 values for each parameter is provided. These inputs are based on the ranges specified in section **1.2**. The tests must be run for each combination.

3.1 EarnedValue - getActivityScheduleValue()

Inputs:

Parameter	Min	Min+	Nom	Max-	Max
ActivityEarlyFinish	0.0	1.0	30.0	730.0	731.0
activityLateFinish	0.0	1.0	30.0	730.0	731.0
activityPlannedVal	0.0	1.0	30.0	730.0	731.0
activityDuration	0.0	1.0	30.0	730.0	731.0
daysSinceStart	0.0	1.0	30.0	730.0	731.0

Num. Tests : 5^5 = 3125

3.2 Project - areValidPercentages()

Inputs:

Parameter	Min	Min+	Nom	Max-	Max
Percentage1	0.0	0.01	0.5	0.99	1.0
Percentage2	0.0	0.01	0.5	0.99	1.0

Tests : 5^2 = 25

3.3 Project - areValideValues()

Inputs:

Parameter	Min	Min+	Nom	Max-	Max
BudgetAtCompletion	0.0	1.0	1000.0	9999999.0	10000000.0
ActualCost	0.0	1.0	1000.0	9999999.0	10000000.0
EarnedValue	0.0	1.0	1000.0	9999999.0	10000000.0

Tests : 5^3 = 125

3.4 Activity - areValideValues()

Inputs:

Parameter	Min	Min+	Nom	Max-	Max
MostLikely	0.0	1.0	30.0	730.0	731.0
Optimistic	0.0	1.0	30.0	730.0	731.0
Pessimistic	0.0	1.0	30.0	730.0	731.0

Tests : $5^3 = 125$

3.5 Activity - areValidPercentAndCost()

Inputs:

Parameter	Min	Min+	Nom	Max-	Max
PercentComplete	0.0	0.01	0.5	0.99	1.0
ActualCost	0.0	1.0	500000.0	999999.0	1000000.0

Tests : $5^2 = 25$

4.0 Test Implementation & Headers

The implementations for each test and all it's test cases are provided in a separate package in out source code. **Test Headers** for each implemented **test case** is provided alongside its code. The information they include are:

- Identifier for test case
- Brief description
- Preconditions
- Inputs and expected outputs
- Expected post-conditions

These test-case headers are included in **Appendix A** for white-box tests, and **Appendix B** for black-box tests.

5.0 Discussion

Blabs blabsblabs

References:

Daniel Sinnig PhD: *Lecture Slides*, *COMP 354*

Appendix

Appendix A – White-Box Function Headers

NodeGraph - addEdge()

```
/**
 * Function to test:    NodeGraph::addEdge()
 * Identifier:                addEdge_whiteBoxPath_1
 * Description:               Test for basis-coverage path 1
 * Preconditions:
 * -  NodeGraph DOES NOT contain v
 * -  v DOES NOT contain w
 * -  w is NOT dependent on v
 * -  v is NOT a precedent of w
 * Inputs:
 * -  MilestoneNode v1, w1;
 * Outputs:
 * -  void
 * PostConditions:
 * -  NodeGraph contains v1
 * -  v1 is precedent of w1
 * -  w1 is dependent of v1
 */
```

```
/**
 * Function to test:    NodeGraph::addEdge()
 * Identifier:                addEdge_whiteBoxPath_2
 * Description:               Test for basis-coverage path 2
 * Preconditions:
 * -  NodeGraph contains v
 * -  v DOES NOT contain w
 * -  w is NOT dependent on v
 * -  v is NOT a precedent of w
 * Inputs:
 * -  MilestoneNode v2, w2, dummy;
 * Outputs:
 * -  void
 * PostConditions:
 * -  NodeGraph contains v2
 * -  v2 is precedent of w2
 * -  w2 is dependent of v2
 */
```

EarnedValue - calculateActualCost()

```
/**
 * Function to test:    EarnedValue::calculateActualCost()
 * Identifier:          calcActCost_whiteBoxPath_1
 * Description:         Test for basis-coverage path 1
 * Preconditions:
 * -  Project(p) has ONE Activity(a)
 * -  a.status = true
 * Inputs:
 * -  void
 * Outputs:
 * -  void
 * PostConditions:
 * -  Correct cost calculated
 */
```

```
/**
 * Function to test:    EarnedValue::calculateActualCost()
 * Identifier:          calcActCost_whiteBoxPath_2
 * Description:         Test for basis-coverage path 2
 * Preconditions:
 * -  Project(p) has ONE Activity(a)
 * -  a.status = true
 * Inputs:
 * -  void
 * Outputs:
 * -  void
 * PostConditions:
 * -  Correct cost calculated
 */
```

```
/**
 * Function to test:    EarnedValue::calculateActualCost()
 * Identifier:          calcActCost_whiteBoxPath_3
 * Description:         Test for basis-coverage path 3
 * Preconditions:
 * -  Project(p) has ONE Activity(a)
 * -  a.status = false
 * -  a.percentComplete > 0
 * Inputs:
 * -  void
 * Outputs:
 * -  void
 * PostConditions:
 * -  Correct cost calculated
 */
```

```
/**
 * Function to test:    EarnedValue::calculateActualCost()
 * Identifier:          calcActCost_whiteBoxPath_4
 * Description:         Test for basis-coverage path 4
 * Preconditions:
 * -  Project(p) has ONE Activity(a)
 * -  a.status = false
 * -  a.percentComplete = 0
 * Inputs:
 * -  void
 * Outputs:
 * -  void
 * PostConditions:
 * -  Correct cost calculated
 */
```

MilestoneNode - equals()

```
/**
 * Function to test:    MilestoneNode::equals()
 * Identifier:                equals_whiteBoxPath_1
 * Description:               Test for basis-coverage path 1
 * Preconditions:
 * -  other is NOT equal to caller
 * -  other is NOT null
 * -  other class is of type MilestoneNode
 * -  caller name NOT null
 * -  other name is same as caller
 * Inputs:
 * -  Object other;
 * Outputs:
 * -  boolean
 * PostConditions:
 * -  returns true
 */
```

```
/**
 * Function to test:    MilestoneNode::equals()
 * Identifier:                equals_whiteBoxPath_2
 * Description:               Test for basis-coverage path 2
 * Preconditions:
 * -  other is equal to caller
 * Inputs:
 * -  Object other;
 * Outputs:
 * -  boolean
 * PostConditions:
 * -  returns true
 */
```

```
/**
 * Function to test:    MilestoneNode::equals()
 * Identifier:                equals_whiteBoxPath_3
 * Description:               Test for basis-coverage path 3
 * Preconditions:
 * -  other is NOT equal to caller
 * -  other is null
 * Inputs:
 * -  Object other;
 * Outputs:
 * -  boolean
 * PostConditions:
 * -  returns false
 */


/**
 * Function to test:    MilestoneNode::equals()
 * Identifier:                equals_whiteBoxPath_4
 * Description:               Test for basis-coverage path 4
 * Preconditions:
 * -  other is NOT equal to caller
 * -  other is NOT null
 * -  other class is NOT of type MilestoneNode
 * Inputs:
 * -  Object other;
 * Outputs:
 * -  boolean
 * PostConditions:
 * -  returns false


/**
 * Function to test:    MilestoneNode::equals()
 * Identifier:                equals_whiteBoxPath_7
 * Description:               Test for basis-coverage path 7
 * Preconditions:
 * -  other is NOT equal to caller
 * -  other is NOT null
 * -  other class is of type MilestoneNode
 * -  caller name NOT null
 * -  other name is NOT same as caller
 * Inputs:
 * -  Object other;
 * Outputs:
 * -  boolean
 * PostConditions:
 * -  returns false
 */


}
```

MilestoneNode - toStringArrows()

```
/**
 * Function to test:    MilestoneNode::toStringArrows()
 * Identifier:                toStringArrows_whiteBoxPath_1
 * Description:               Test for basis-coverage path 1
 * Preconditions:
 * -  type = "in"
 * -  MilestoneNode has ONE inArrow
 * -  MilestoneNode has ONE inArrow
 * Inputs:
 * -  String type;
 * Outputs:
 * -  void
 * PostConditions:
 * -  outputs correct string
 */
```

```
/**
 * Function to test:    MilestoneNode::toStringArrows()
 * Identifier:                toStringArrows_whiteBoxPath_2
 * Description:               Test for basis-coverage path 2
 * Preconditions:
 * -  type = "in"
 * -  MilestoneNode has NO inArrow
 * Inputs:
 * -  String type;
 * Outputs:
 * -  void
 * PostConditions:
 * -  outputs correct string
 */
```

```
/**
 * Function to test:    MilestoneNode::toStringArrows()
 * Identifier:                toStringArrows_whiteBoxPath_5
 * Description:               Test for basis-coverage path 5
 * Preconditions:
 * -  type = "in"
 * -  MilestoneNode has ONE inArrow
 * -  MilestoneNode has ONE inArrow
 * Inputs:
 * -  String type;
 * Outputs:
 * -  void
 * PostConditions:
 * -  outputs correct string
 */
```

```
/**
 * Function to test:    MilestoneNode::toStringArrows()
 * Identifier:                toStringArrows_whiteBoxPath_6
 * Description:               Test for basis-coverage path 6
 * Preconditions:
 * -  type = "out"
 * -  MilestoneNode has NO outArrows
 * Inputs:
 * -  String type;
 * Outputs:
 * -  void
 * PostConditions:
 * -  outputs correct string
 */
```

```
/**
 * Function to test:    MilestoneNode::toStringArrows()
 * Identifier:                toStringArrows_whiteBoxPath_7
 * Description:               Test for basis-coverage path 7
 * Preconditions:
 * -  type != "in" or "out"
 * Inputs:
 * -  String type;
 * Outputs:
 * -  void
 * PostConditions:
 * -  outputs null
 */
```

MilestoneNode – equals()

```
/**
 * Function to test:    Activity::areValidTimes()
 * Identifier:                areValidTimes_whiteBoxPath_1
 * Description:               Test for basis-coverage path 1
 * Preconditions:
 * -  All inputs greater than 0
 * -  All inputs less than Activity.MAX_DURATION
 * -  mostlikely less than pessimistic and more than optimistic
 * Inputs:
 * -  double mostLikely, optimistic, pessimistic
 * Outputs:
 * -  boolean
 * PostConditions:
 * -  return true
 */


/**
 * Function to test:    Activity::areValidTimes()
 * Identifier:                areValidTimes_whiteBoxPath_2
 * Description:               Test for basis-coverage path 2
 * Preconditions:
 * -  At least 1 input less than 0
 * Inputs:
 * -  double mostLikely, optimistic, pessimistic
 * Outputs:
 * -  boolean
 * PostConditions:
 * -  return false
 */


/**
 * Function to test:    Activity::areValidTimes()
 * Identifier:                areValidTimes_whiteBoxPath_3
 * Description:               Test for basis-coverage path 3
 * Preconditions:
 * -  All inputs greater than 0
 * -  At least 1 input greater than Activity.MAX_DURATION
 * Inputs:
 * -  double mostLikely, optimistic, pessimistic
 * Outputs:
 * -  boolean
 * PostConditions:
 * -  return false
 */
```

```
/**
 * Function to test:    Activity::areValidTimes()
 * Identifier:                areValidTimes_whiteBoxPath_4
 * Description:               Test for basis-coverage path 4
 * Preconditions:
 * -  All inputs greater than 0
 * -  All inputs less than Activity.MAX_DURATION
 * -  mostLikely greater than pessimistic or less than optimistic
 * Inputs:
 * -  double mostLikely, optimistic, pessimistic
 * Outputs:
 * -  boolean
 * PostConditions:
 * -  return false
 */
```

Appendix B – Black-Box Function Headers

EarnedValue - getActivitySchedule()

```
/**
 * Function to test:    EarnedValue::getActivitySchedule()
 * Identifier:                getActivitySchedule_blackBox
 * Description:               Blackbox Test for getActivityScheduleValue()
 * Inputs:
 * -  double earlyFinish, lateFinish, plannedValue, duration, daysSinceStart
 * Outputs:
 * -  double scheduleValue
 * PostConditions:
 * -  correct scheduleValue is returned
 */
```

Activity - areValidPercentAndCost()

```
/**
 * Function to test:    Activity::areValidPercentAndCost()
 * Identifier:                areValidPercentAndCost_blackBox
 * Description:               Blackbox Test for areValidPercentAndCost()
 * Inputs:
 * -  double percent, cost
 * Outputs:
 * -  boolean
 * PostConditions:
 * -  return true
 */
```

Activity - areValidTimes()

```
/**
 * Function to test:    Activity::areValidTimes()
 * Identifier:                areValidTimes_blackBox
 * Description:               Blackbox Test for areValidTimes()
 * Inputs:
 * -  double mostLikey, optimistic, pessimistic
 * Outputs:
 * -  boolean
 * PostConditions:
 * -  return true
 */
```

Project - areValidPercentages()

```
/**
 * Function to test:    Project::areValidPercentages()
 * Identifier:                areValidPercentages_blackBox
 * Description:               Blackbox Test for areValidPercentages()
 * Inputs:
 * -  double percent1, percent2
 * Outputs:
 * -  boolean
 * PostConditions:
 * -  return true
 */
```

Project - areValidValues()

```
/**
 * Function to test:    Project::areValidValues()
 * Identifier:                areValidValues_blackBox
 * Description:               Blackbox Test for areValidValues()
 * Inputs:
 * -  double value1, value2, value3
 * Outputs:
 * -  boolean
 * PostConditions:
 * -  return true
 */
```