Visual Architecture Acceptance Test Plan

Version 1.0

Doc. No.:	

Revision History

Date	Version	Description	Author
2002-00-00	0.01	Initial Draft	All
2010-12-10	1.00	Version for hand in with GMF and xText tests	All

Table of Contents

1. Introduction

1. Purpose of this document

The purpose of this document is to verify the product Visual Architecture. This document will prove as a primary source to assure that all the requirements are currently meet. Furthermore this document will provide a detailed way to show various ways to be used for the verify the product.

2. Intended Audience

The document is mostly targeted the following user groups

- Project Members
- Customers
- Stakeholders etc

3. Scope

The document will primarily focus on the verifying the assumed requirements as per the requirements specification document with the help of the test cases. Some non-functional aspects can be still covered but they will remain inside the requirement domain

4. Definitions and acronyms

1. *Definitions*

Keyword	Definitions
Students enviroment	Part of application which the student uses in order to enter and debug entered code.
Teachers enviroment	Part of application which teacher uses in order to enter the architecture on which the students code is debugged
Eclipse plugin	Application which can be integrated into Eclipse framework making it richer in content and adding aditional functionalities to it
Command	Part of the grammar used for the student assembly code
Text highlight	Help during the code input, a method of distinguishing special parts of the code (like keywords) from other code
Auto-complete	Help during the code input, a method which makes it possible for user to type in only the starting part of the word and complete it from the list of available continuations provided by the interface
Vamodel Diagram	The customized GMF plug-in file of file extension *.vadiagram* that stores the model of the custom architecture generation by the teacher's environment.
Compartment	A component that holds other components

2. *Acronyms and abbreviations*

Acronym or	Definitions

abbreviation	
GMF	Graphical modeling framework
EMF	Eclipse modeling framework
AST	Abstract syntax tree
JRE	Java Runtime Enviroment

5. References

Other projects which are important to have in mind when it comes to technology being used:

Java Eclipse project: http://www.eclipse.org/
XText project: http://www.eclipse.org/Xtext/
EMF project: http://www.eclipse.org/modeling/emf/
GMF project: http://www.eclipse.org/modeling/gmp/

Documentation of this project with relevant information for testing:

Requirements definition document(Requirements_Definition_v1.0[1].doc)
and design description document(Design_description[3].odt) are both available at:
http://www.fer.hr/rasip/dsd/projects/visual_architecture/documents

2. Test-plan introduction

Due to the fact that a lot of code is being automatically generated there will not be many (if any) unit tests. User interface will be tested for each requirement they should satisfy. There will be one test group for each user interface divided into number of tests. Each test should be as small as possible in scope and test atomic requirements if possible.

For this tests tester does some actions for which he knows what are the required outputs. After the outputs are generated he compares them to the required ones. Test passes if they match. There should also be tests which test inner functionalities like data processing or communication. For data processing test inputs are generated and processed and the given result is compared to the required result. Test passes if they match. For communication data on the source is compared with data on destination, test passes if those two match.

3. Test items

1. Student's environment
    a. Code editing environment
        - Graphical user interface
        - Syntax tree generated from the code
    b. Microcode compiler
    c. Debugger

2. Teacher's environment
    a. Diagram editor
    b. Architecture mapping environment

4. Features to be tested

- grammar parser in the code editing environment
- GMF diagram editor
- Debugger GUI
- auto-complete feature of student code editor
- highlight feature of student code editor
- error notifier feature of student code editor
- features of AST which is build from different parts of code(including code with or without errors and empty code), which includes presence of exceptions, number of errors, number of correct commands and presence and completeness of commands from inside the code in the final AST

5. Features not to be tested

The Eclipse frameworks themselves are not going to be tested, since they are all mature projects and (apart for maybe some minor bugs) their functionality can be relied on.

Code which is generated by Eclipse plugins XText and GMF will not be unit tested. We rely on the fact that that code is correct and has as few bugs as possible.

The diagrams generated by the GMF editor will not be tested to determine if they are valid assembly architecture constructs, they authenticity of the design relies entirely on the designers' expertise.

6. Approach

Since majority of the code is generated using Eclipse plugins focus will be on testing greater functionalities rather then small parts of code. Because of that we shall use the approach in which we will design scenarios and wanted outcome of them, simulate them and compare the results with wanted results. There should be no aggressive testing as this is not domain sensitive project. Great part of testing is user interface testing as there are many requirements which aim to make tools more user friendly.

Graphical user interface for code input and AST will be tested by Ame Ilirjana, Peter van Heck and Palajić Vedran.

Graphical user interface for teacher diagram design and mapping methods will be tested by Lučanin Dražen, Sarah Njeri Kuria and Prashanta Paudel.

Debugger and communication of debugger with code input and visual design will be tested by all parts of project team.

All tests will rely heavily on the requirements document made from the collaboration with the customer, who is also considered the number one user.

1. Approach to configuration and installation

Tests will be done on the default Eclipse configuration. Since some of the features being tested depend on the current configuration and setup of Eclipse(for example the way and place Eclipse notifies users about errors) testers have to be aware of the current configuration during the testing and take them into account and adapt to them.

7. Item pass/fail criteria

Pass/fail criteria:

- The tests are considered to pass if Eclipse acts as is described in each individual test when the responsible plug-ins are deployed
- Exceptionally, unit tests are supposed to be run actively and give their own output on whether they passed or failed

1. Installation and Configuration

As the product is an Eclipse plug-in , "Visual Architecture " requires a Java Run-time environment (JRE) and Eclipse framework. Currently the system is expected to work properly in the JRE version 1.6 and 1.5. Also, Eclipse 3.6 is recommended. Compatibility should be highly maintained. Our product is a set of Eclipse plug-ins and therefore it should be installed and configured like any other Eclipse plug-ins are.

2. Documentation problems

Some tests could be hard to evaluate as pass or fail due to incomplete user documentation. Lets say some new requirements are defined a test which was before marked as passed could now be a failed test if it does not satisfy the newly added requirement.

Some test which test atomic functionalities could become absolete should some requirements be dropped. If some requirements are added it would probably be needed to add some new tests which would test satisfaction of those requirements.

8. Suspension criteria and resumption requirements

Tests are mostly independent from each other. This means should the bug arise at some of them other testing can continue as planed. This includes graphical user testing of both code input and architecture design parts as well as AST tests. They can be tested independently and should the bug which needs to be fixed arise at some of them others can continue without problems.

Testing of communication of debugger with code input and visual representation can only continue after all the tests for AST and editing of visual representation is done. If the bug arises in one of those testing of communication should start from beginning.

9. Environmental needs

1. Hardware

Visual Architecture does not require any special hardware . A general purpose hardware is enough for the software to run.

2. Software

Visual Architecture primarily depends on Eclipse Framework .The only supported operating systems by Eclipse are Windows , Linux and MAC OS. It is assumed that JRE 1.6 or JRE 1.5 has been installed and also Eclipse 3.6 is recommended .

3. Other

Visual Architecture does not require any extra other specific needs.

10. Test procedure

1. Test case specifications

1. Code input and XText parse tree testing - CIXTPTT

Number of tests will be done taking both correct and incorrect examples of input code. Code will be parsed and created XText parse tree will be compared to expected results.

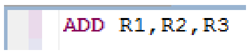
1. Correct “ADD” command - CIXTPTT-001

Description:
Syntatically correct “ADD” command is typed in the code editor. After that the code is parsed. Xtext AST is evaluated for matching with expected results.

Test type:
Positive, test code is expected to execute without errors.

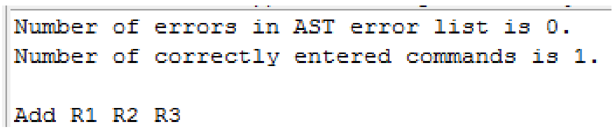
Preconditions:
Plugin based on generated XText grammar must be running. Code which is able to go trough AST, check it for errors and type out its contents must be written, correct and ready for running.

Input definition:
1. Create the file (with the proper extension) inside the plugin based on generated XText grammar
2. Enter correct “ADD” command inside the file
3. Save the file
4. Run the code which evaluates created AST



```
ADD R1,R2,R3
```

Output definition:
Code for evaluating AST should state the fallowing:
1. No exceptions during the reading process did occur.
2. Number of errors in AST error list is 0 and equal to the number of incorrectly written commands.
3. Number of correctly entered commands is 1.
4. AST contains all parts of “ADD” command.



```
Number of errors in AST error list is 0.
Number of correctly entered commands is 1.

Add R1 R2 R3
```

Remarks:
It is important to make the AST available to the code which goes trough it. For the purpose of this test, name of the file which is transformed into AST was decided upfront and hard-coded into the code which evaluates the AST. If some more advanced method of accessing AST is used it should be tested separately. Since we access the file trough its path it is important to note that we rely on the fact that the we have access to the path and no errors will occur because of the unexpected events in the file system of the operating system.

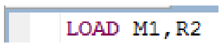
2. Correct “LOAD” command - CIXTPTT-002

Description:
Syntaxly correct “LOAD” command is typed in the code editor. After that the code is parsed. Xtext AST is evaluated for matching with expected results.

Test type:
Positive, test code is expected to execute without errors.

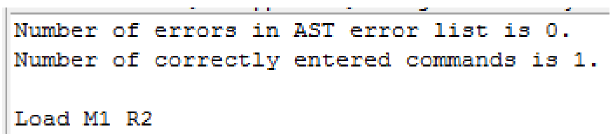
Preconditions:
Plugin based on generated XText grammar must be running. Code which is able to go trough AST, check it for errors and type out its contents must be written, correct and ready for running.

Input definition:
1. Create the file (with the proper extension) inside the plugin based on generated XText grammar
2. Enter correct “LOAD” command inside the file
3. Save the file
4. Run the code which evaluates created AST



```
LOAD M1,R2
```

Output definition:
Code for evaluating AST should state the fallowing:
1. No exceptions during the reading process did occur.
2. Number of errors in AST error list is 0 and equal to the number of incorrectly written commands.
3. Number of correctly entered commands is 1.
4. AST contains all parts of “LOAD” command.



```
Number of errors in AST error list is 0.
Number of correctly entered commands is 1.

Load M1 R2
```

Remarks:
It is important to make the AST available to the code which goes trough it. For the purpose of this test, name of the file which is transformed into AST was decided upfront and hard-coded into the code which evaluates the AST. If some more advanced method of accessing AST is used it should be tested separately. Since we access the file trough its path it is important to note that we rely on the fact that the we have access to the path and no errors will occur because of the unexpected events in the file system of the operating system.

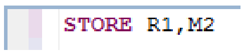
3. Correct “STORE” command - CIXTPTT-003

Description:
Syntaxly correct “STORE” command is typed in the code editor. After that the code is parsed. Xtext AST is evaluated for matching with expected results.

Test type:
Positive, test code is expected to execute without errors.

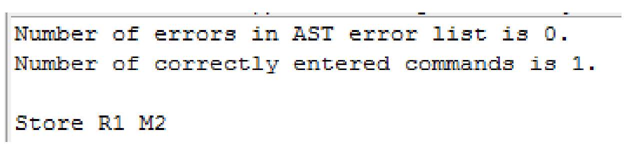
Preconditions:
Plugin based on generated XText grammar must be running. Code which is able to go trough AST, check it for errors and type out its contents must be written, correct and ready for running.

Input definition:
1. Create the file (with the proper extension) inside the plugin based on generated XText grammar
2. Enter correct “STORE” command inside the file
3. Save the file
4. Run the code which evaluates created AST



```
STORE R1,M2
```

Output definition:
Code for evaluating AST should state the fallowing:
1. No exceptions during the reading process did occur.
2. Number of errors in AST error list is 0 and equal to the number of incorrectly written commands.
3. Number of correctly entered commands is 1.
4. AST contains all parts of “STORE” command.



```
Number of errors in AST error list is 0.
Number of correctly entered commands is 1.

Store R1 M2
```

Remarks:
It is important to make the AST available to the code which goes trough it. For the purpose of this test, name of the file which is transformed into AST was decided upfront and hardc-oded into the code which evaluates the AST. If some more advanced method of accessing AST is used it should be tested separately. Since we access the file trough its path it is important to note that we rely on the fact that the we have access to the path and no errors will occur because of the unexpected events in the file system of the operating system.

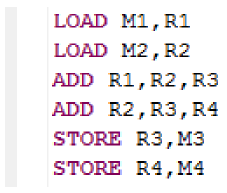
4. Multiple distinct correct commands - CIXTPTT-004

Description:
Syntaxly correct commands are typed in the code editor. Two of each, “STORE”, “ADD” and “LOAD”. After that the code is parsed. XText AST is evaluated for matching with expected results.

Test type:
Positive, test code is expected to execute without errors.

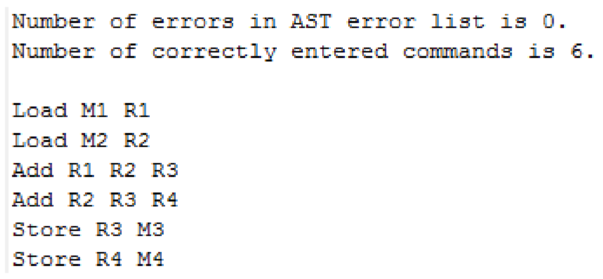
Preconditions:
Plugin based on generated XText grammar must be running. Code which is able to go trough AST, check it for errors and type out its contents must be written, correct and ready for running.

Input definition:
1. Create the file (with the proper extension) inside the plugin based on generated XText grammar
2. Enter two correct “STORE”, “LOAD” and “ADD” commands in any order inside the file
3. Save the file
4. Run the code which evaluates created AST



```
LOAD M1,R1
LOAD M2,R2
ADD R1,R2,R3
ADD R2,R3,R4
STORE R3,M3
STORE R4,M4
```

Output definition:
Code for evaluating AST should state the fallowing:
1. No exceptions during the reading process did occur.
2. Number of errors in AST error list is 0 and equal to the number of incorrectly written commands.
3. Number of correctly entered commands is 6.
4. AST contains all parts of 6 entered commands.



```
Number of errors in AST error list is 0.
Number of correctly entered commands is 6.

Load M1 R1
Load M2 R2
Add R1 R2 R3
Add R2 R3 R4
Store R3 M3
Store R4 M4
```

Remarks:
It is important to make the AST available to the code which goes trough it. For the purpose of this test, name of the file which is transformed into AST was decided upfront and hard-coded into the code which evaluates the AST. If some more advanced method of accessing AST is used it should be tested separately. Since we access the file trough its path it is important to note that we rely on the fact that the we have access to the path and no errors will occur because of the unexpected events in the file system of the operating system.

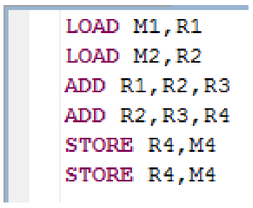
5. Multiple duplicate correct commands - CIXTPTT-005

Description:
Syntaxly correct commands are typed in the code editor. Two of each, “STORE”, “ADD” and “LOAD”. At least two os those are identicly the same. After that the code is parsed. Xtext AST is evaluated for matching with expected results.

Test type:
Positive, test code is expected to execute without errors.

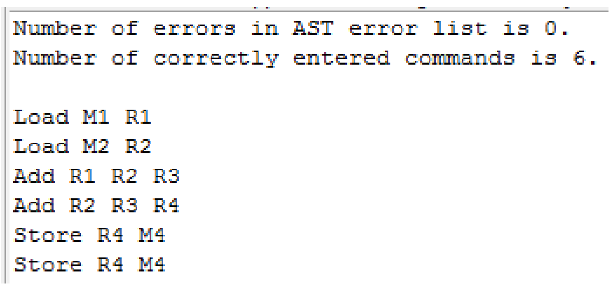
Preconditions:
Plugin based on generated XText grammar must be running. Code which is able to go trough AST, check it for errors and type out its contents must be written, correct and ready for running.

Input definition:
1. Create the file (with the proper extension) inside the plugin based on generated XText grammar
2. Enter two correct “STORE”, “LOAD” and “ADD” commands in any order inside the file making sure at least two of those are identicly the same
3. Save the file
4. Run the code which evaluates created AST



```
LOAD M1,R1
LOAD M2,R2
ADD R1,R2,R3
ADD R2,R3,R4
STORE R4,M4
STORE R4,M4
```

Output definition:
Code for evaluating AST should state the fallowing:
1. No exceptions during the reading process did occur.
2. Number of errors in AST error list is 0 and equal to the number of incorrectly written commands.
3. Number of correctly entered commands is 6.
4. AST contains all parts of 6 entered commands.



```
Number of errors in AST error list is 0.
Number of correctly entered commands is 6.

Load M1 R1
Load M2 R2
Add R1 R2 R3
Add R2 R3 R4
Store R4 M4
Store R4 M4
```

Remarks:
It is important to make the AST available to the code which goes trough it. For the purpose of this test, name of the file which is transformed into AST was decided upfront and hard-coded into the code which evaluates the AST. If some more advanced method of accessing AST is used it should be tested separately. Since we access the file trough its path it is important to note that we rely on the fact that the we have access to the path and no errors will occur because of the unexpected events in the file system of the operating system.

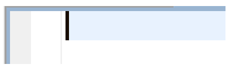
6. Empty file - CIXTPTT-006

Description:
Nothing is entered in code editor. After that the code is parsed. XText AST is evaluated for matching with expected results.

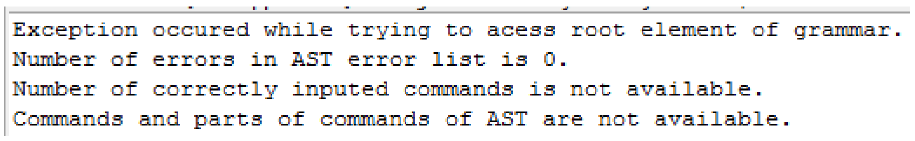
Test type:
Negative, test code is expected to execute with caught exception.

Preconditions:
Plugin based on generated XText grammar must be running. Code which is able to go trough AST, check it for errors and type out its contents must be written, correct and ready for running.

Input definition:
1. Create the file (with the proper extension) inside the plugin based on generated XText grammar
2. Leave the file empty
3. Save the file
4. Run the code which evaluates created AST



Output definition:
Code for evaluating AST should state the fallowing:
1. Exception occurred while trying to access root element of grammar.
2. Number of errors in AST error list is 0.
3.Number of correctly entered commands is not available.
4. Commands and parts of commands of AST are not available.



```
Exception occured while trying to acess root element of grammar.
Number of errors in AST error list is 0.
Number of correctly inputed commands is not available.
Commands and parts of commands of AST are not available.
```

Remarks:
It is important to make the AST available to the code which goes trough it. For the purpose of this test, name of the file which is transformed into AST was decided upfront and hard-coded into the code which evaluates the AST. If some more advanced method of accessing AST is used it should be tested separately. Since we access the file trough its path it is important to note that the we have access to the path and no errors will occur because of the unexpected events in the file system of the operating system.

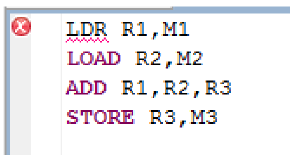
7. File containing unrecognizable command at the start- CIXTPTT-007

Description:
Code editor contains any number of correct commands and a piece of text which is not a part of a command or recognizable command at the start of file. After that the code is parsed. Xtext AST is evaluated for matching with expected results.

Test type:
Negative, test code is expected to generate an exception.

Preconditions:
Plugin based on generated XText grammar must be running. Code which is able to go trough AST, check it for errors and type out its contents must be written, correct and ready for running.

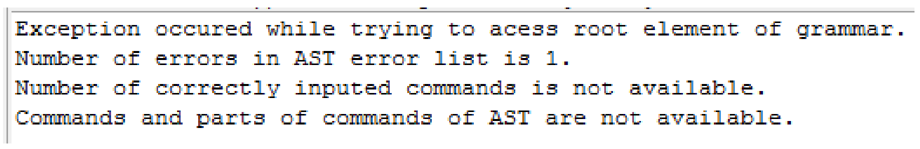
Input definition:
1. Create the file (with the proper extension) inside the plugin based on generated XText grammar
2. Enter any number of correct commands with unrecognizable part of code at the start
3. Save the file
4. Run the code which evaluates created AST



```
LDR R1,M1
LOAD R2,M2
ADD R1,R2,R3
STORE R3,M3
```

Output definition:
Code for evaluating AST should state the fallowing:
1. Exception occurred while trying to acess root element of grammar.
2. Number of errors in AST error list is 1.
3.Number of correctly entered commands is not available.

4. Commands and parts of commands of AST are not available.


```
Exception occured while trying to acess root element of grammar.
Number of errors in AST error list is 1.
Number of correctly inputed commands is not available.
Commands and parts of commands of AST are not available.
```

Remarks:
It is important to make the AST available to the code which goes trough it. For the purpose of this test, name of the file which is transformed into AST was decided upfront and hard-coded into the code which evaluates the AST. If some more advanced method of accessing AST is used it should be tested separately. Since we access the file trough its path it is important to note that the we have access to the path and no errors will occur because of the unexpected events in the file system of the operating system.

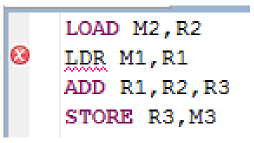
8. File containing unrecognizable command which is not at the start- CIXTPTT-008

Description:
Code editor contains any number of correct commands and a piece of text which is not a part of a command or recognizable command which is not on the start of a file. After that the code is parsed. XText AST is evaluated for matching with expected results.

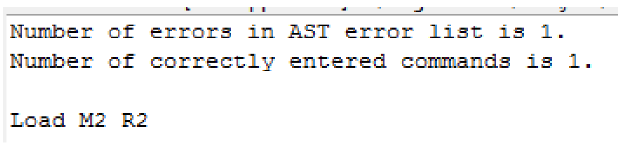
Test type:
Negative, test code is expected to have errors in error list.

Preconditions:
Plugin based on generated XText grammar must be running. Code which is able to go trough AST, check it for errors and type out its contents must be written, correct and ready for running.

Input definition:
1. Create the file (with the proper extension) inside the plugin based on generated XText grammar
2. Enter any number of correct commands with unrecognizable part of code which is not at the start of file.
3. Save the file
4. Run the code which evaluates created AST


```
LOAD M2,R2
LDR M1,R1
ADD R1,R2,R3
STORE R3,M3
```

Output definition:
Code for evaluating AST should state the fallowing:
1. No exceptions during the reading process did occur.
2. Number of errors in AST error list is 1.
3.Number of correctly entered commands equal to number of correct commands before the unrecognizable part of the code in the file.
4. AST contains parts of all commands entered before the unrecognizable part of the code.


```
Number of errors in AST error list is 1.
Number of correctly entered commands is 1.

Load M2 R2
```

Remarks:
It is important to make the AST available to the code which goes trough it. For the purpose of this test, name of the file which is transformed into AST was decided upfront and hard-coded into the code which evaluates the AST. If some more advanced method of accessing AST is used it should be tested separately. Since we access the file trough its path it is important to note that we rely on the fact that the we have access to the path and no errors will occur because of the unexpected events in the file system of the operating system. It is important that unrecognizable part of the code is not positioned after a recognizable command which has errors in it.

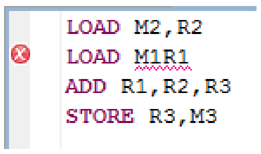
9. File containing incorrect command- CIXTPTT-009

Description:
Code editor contains any number of correct commands and at least one incorrect but recognizable command. After that the code is parsed. XText AST is evaluated for matching with expected results.

Test type:
Negative, test code is expected to have errors in error list.

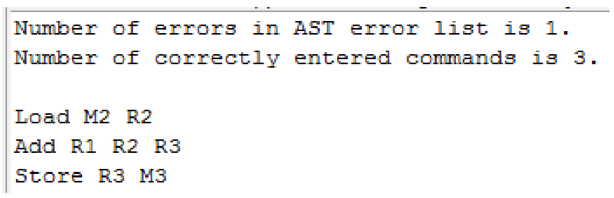
Preconditions:
Plugin based on generated XText grammar must be running. Code which is able to go trough AST, check it for errors and type out its contents must be written, correct and ready for running.

Input definition:
1. Create the file (with the proper extension) inside the plugin based on generated XText grammar
2. Enter any number of correct commands and at least one incorrect command.
3. Save the file
4. Run the code which evaluates created AST



```
LOAD M2,R2
LOAD M1R1
ADD R1,R2,R3
STORE R3,M3
```

Output definition:
Code for evaluating AST should state the fallowing:
1. No exceptions during the reading process did occur.
2. Number of errors in AST error list is equal to the number of incorrect commands in the file.
3.Number of correctly entered commands is equal to number of commands which are not incorrect
4. AST contains parts of all commands which are correct.



```
Number of errors in AST error list is 1.
Number of correctly entered commands is 3.

Load M2 R2
Add R1 R2 R3
Store R3 M3
```

Remarks:
It is important to make the AST available to the code which goes trough it. For the purpose of this test, name of the file which is transformed into AST was decided upfront and hard-coded into the code which evaluates the AST. If some more advanced method of accessing AST is used it should be tested separately. Since we access the file trough its path it is important to note that we rely on the fact that the we have access to the path and

no errors will occur because of the unexpected events in the file system of the operating system. Note that in case there is unrecognizable part of the code after incorrect command that will not generate 2 errors but only 1, unrecognizable code will become a part of the incorrect command.

2. Code input GUI testing - CIGUIT

Number of tests will be done in order to test if the user interface for entering assembly code supports wanted functionality.

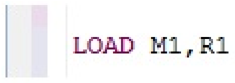
1. Code highlighting - CIGUIT001

Description:
This test is used to determine if the interface supports code highlight feature.

Test type:
Positive, recognizable terminals should be highlighted. This means they should be distinct from other parts of code by for example use of different (unusual) color.

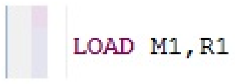
Preconditions:
Plugin based on generated XText grammar must be running.

Input definition:
1. Create the file (with the proper extension) inside the plugin based on generated XText grammar
2. Enter correct code containing some recognizable terminals
3. Check the code looking for highlighted text



```
LOAD M1,R1
```

Output definition:
After the check of the code
1. Any errors or code which does not affect the test can and should be ignored
2. Each part of code which is equal to existing terminal inside the grammar should be highlighted



```
LOAD M1,R1
```

Remarks:
Standard Eclipse highlights were used during the tests. Normal code was black and highlighted text was purple. It is important that the tester is aware of what is the current color for normal text and what is the current color for highlighted text before executing the tests. Apart from the part of the code which is important for testing there can be more code with possible errors in it. That code and errors are not part or important for this test and therefore can and should be ignored.

2. Code auto-complete - CIGUIT002

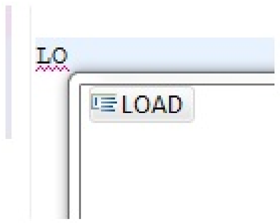
Description:
This test is used in order to check if user interface supports the auto-complete code feature.

Test type:
Positive, after using the auto-complete feature on a prefix of recognizable terminal of used XText grammar user

should get a list of terminals which have the prefix in question and is able to choose one of those to complete the prefix with a whole word.

Preconditions:
Plugin based on generated XText grammar must be running.

Input definition:
1. Create the file (with the proper extension) inside the plugin based on generated XText grammar
2. Enter a word which is a prefix of recognizable terminal
3. Use the auto-complete feature
4. Choose suited terminal inside the offered list
5. Check the code which is a result of made commands



Output definition:
After the check of the code
1. Any errors or code which does not affect the test can and should be ignored
2. Code should be updated by completing the starting prefix with the last part of the terminal chosen from the list.

Remarks:
In this test standard Eclipse shortcut is used for auto-complete feature (Ctrl+SPACE). It is important for tester to know what is the current shortcut or know any other way of accessing Eclipse auto-complete feature in order to correctly perform this test. Apart from the part of the code which is important for testing there can be more code with possible errors in it. That code and errors are not part or important for this test and therefore can and should be ignored. Code highlights are not a part of this test and they should be ignored.

3. Error notifiers- CIGUIT003

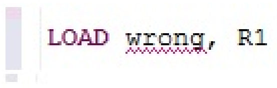
Description:
This test is used to determine if the interface notifies the user about errors made in the code.

Test type:
Positive, interface should notify the user about made errors.

Preconditions:
Plugin based on generated XText grammar must be running.

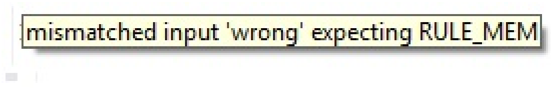
Input definition:

1. Create the file (with the proper extension) inside the plugin based on generated XText grammar
2. Enter some code which has errors in it.
3. Check the interface looking for any notice about found errors.



```
LOAD wrong, R1
```

Output definition:
After the check of the code
1. There should be a notice with details about error made in the code.



```
mismatched input 'wrong' expecting RULE_MEM
```

Remarks:
In this test errors where exclamation marks inside red circles on the left side of the code. Each notice is lined up with the row in which the error is made. This is the Eclipse default. It is important that tester knows what is the current Eclipse setup for showing errors as it may be different then this explained example.

3. GMF Diagram Editor Test (GMFDET)

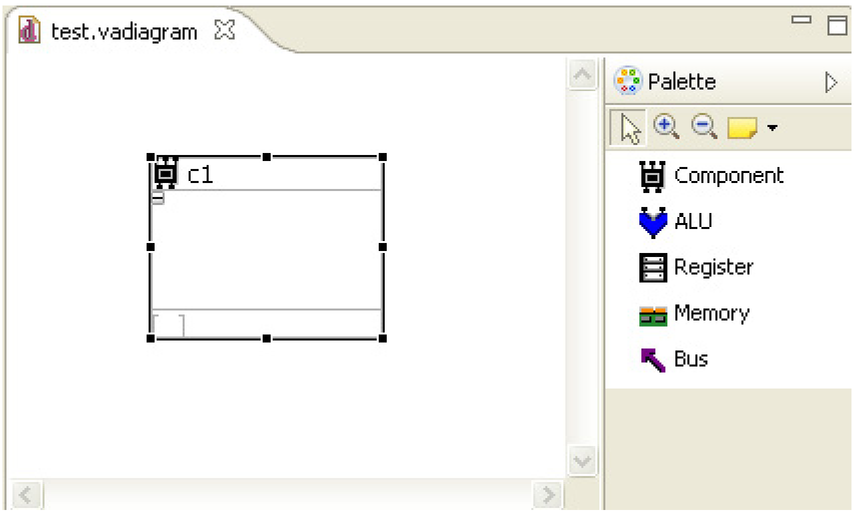
1. Including a component - GMFDET-001

Description:
Successfully adding a component, the new component should be then visible in the diagram as well as the model tree.

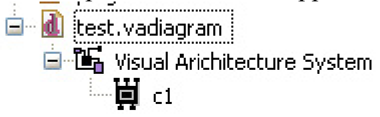
Test type:
Positive, diagram should be included without errors

Preconditions:
A Vamodel Diagram should have been included in the project (file extension .vadiagram)

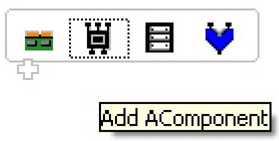
Input definition:
1.Open the .vadiagram file for edit
2.Select a component from the tools Palette
3. Draw the component in the diagram editor
4. Name the component c1
5. Save the diagram
6. Check the project Explorer view for the new component.



Output:
The compoenent c1 should appear in the diagram model tree



Remarks:
Components can also be created by clicking in the drawing area and selecting the respective tool



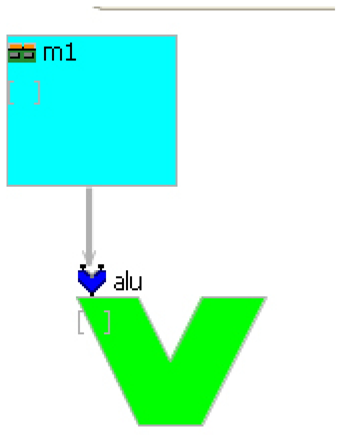
2. Including a component connection (bus) - GMFDET-002

Description:
Successfully adding a connection between two components with target and source objects being correctly reflected in the diagram model

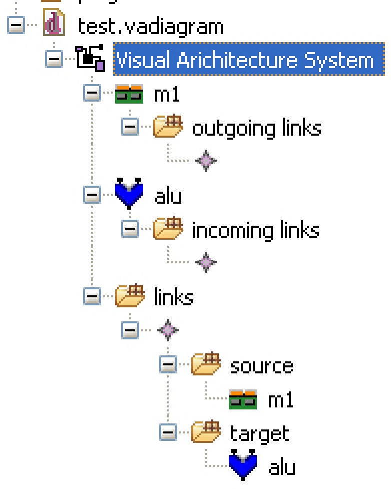
Test type:
Positive, components should be connected using buses

Preconditions:
A Vamodel Diagram should have been included in the project (file extension .vadiagram), the file should be open for edit.

Input definition:
1.Add a memory component named m1
2. Add an ALU component named alu
3. Use the Bus tool to connect m1 to alu, with m1 as the source and alu as the destination
4. Save the model diagram



Output:
The diagram editor should allow the connection between the two components and reflect these links once the diagram is saved



```
test.vadiagram
  Visual Arichitecture System
    m1
      outgoing links
        
    alu
      incoming links
        
    links
      
        source
          m1
        target
          alu
```

Remarks:
Because the bus has no name, it appears as blank tree item.

3. Including a component within a component (component compartment) - GMFDET-003

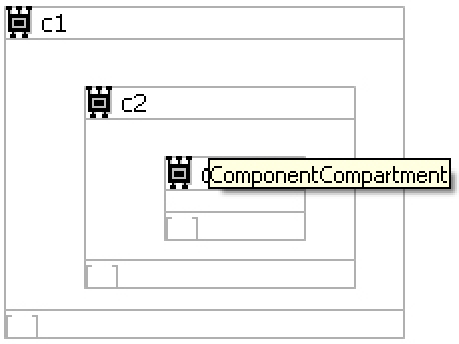
Description:
Successfully adding 3 component levels using compartments. Three nested components c1, c2, and c3 will be created.

Test type:
Positive, the general component tool should hold other general components

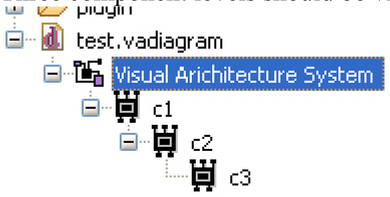
Preconditions:
A Vamodel Diagram should have been included in the project (file extension .vadiagram), the file should be

open for edit.

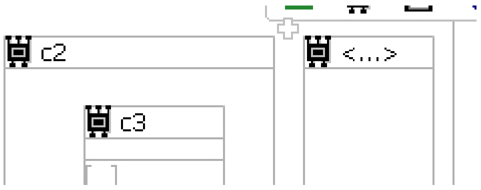
Input definition:
1. Draw component c1
2. Draw component c2 within c1
3 Draw component c3 withing c2
4. Save the diagram



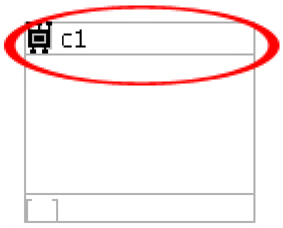
Output:
Three component levels should be visible in the model tree



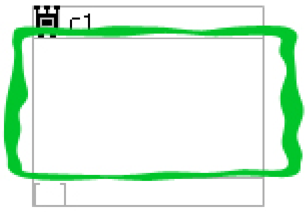
Remarks:
Anonymous components are represented with the text <...> in the model diagram but appear with blank names in the model tree



nested components cannot be placed in the name area



instead they must be placed in the center of the component



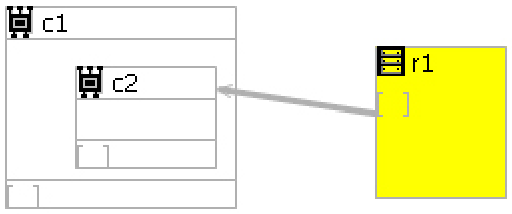
4. Including a component connection between a nested component and an external component - GMFDET-004

Description:
Successfully adding a component connection between a nested component and an external component, the new connection should be visible in the diagram as well as the model tree.

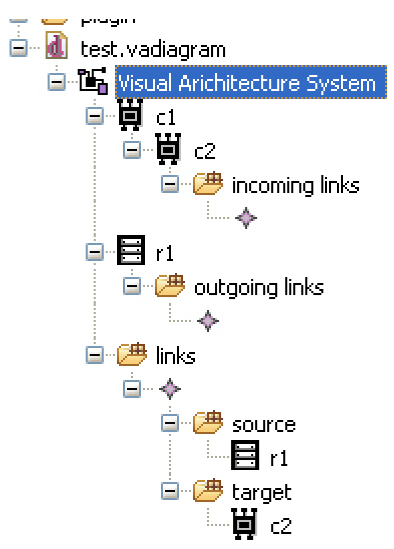
Test type:
positive, connections between nested components and external components should be permitted

Preconditions:
A Vamodel Diagram should have been included in the project (file extension .vadiagram), the file should be open for edit.

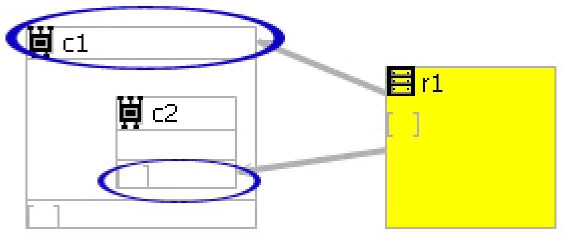
Input definition:
1.Create a component c1
2. Place a component c2 in c1
3. Draw a register component named r1
4. connect r1 to c2
5.Save the changes to the diagram



Output:
The connection between c2 and r1 should be indicated in the model



Remarks:
Connections in the compartment are restricted to the top and bottom of the component. This should probably be revised for convenience.



5. Including two connections between a two components in the same direction - GMFDET-005

Description:
Adding two connections between two components in the same direction. One component will be the source of both connections while the other will be the destination of both connections.

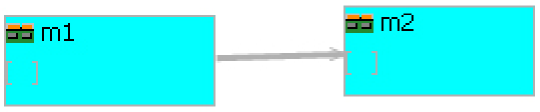
Test type:
Negative, controlling the design of the architecture was not part of the system requirements

Preconditions:
An open Vamodel digram file

Input definition:
1.Draw two memory components m1 and m2
2. Draw two connections with m1 as the source for both and m2 as the destination
3. Save the changes

Output:

The program prevents the creation of two similar connections between the same components



4. Debugger testing - DT

1. Launching from debug shortcut - DT-001

Description: Starting the debugger from a debug shortcut

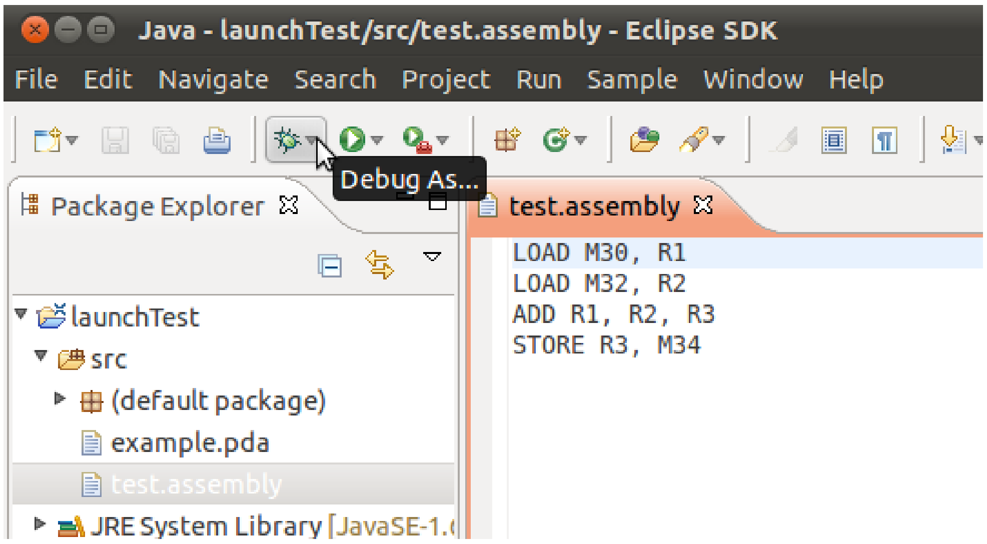
Test type: Positive

Preconditions:

- An *.assembly file exists and is opened inside the active editor

Input definition:

1. select the debug drop-down menu (next to the debug icon)
2. choose Debug As...


Java - launchTest/src/test.assembly - Eclipse SDK
File Edit Navigate Search Project Run Sample Window Help
Debug As...
Package Explorer
test.assembly
launchTest
src
(default package)
example.pda
test.assembly
JRE System Library [JavaSE-1.

```
LOAD M30, R1
LOAD M32, R2
ADD R1, R2, R3
STORE R3, M34
```

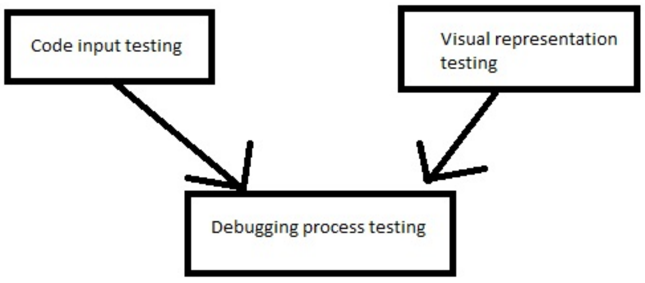
Output definition:

Flowchart entry should be present there.

Remarks:

Context launching makes sure only *.assembly files get the ability to launch our debugger.

2. Test plan



As shown on the picture testing the debugger has some dependencies all other tests can be done in any order.

11. Responsibilities

1. Developers

Write tests which depend heavily on the requirements document making sure they cover as many required functionalities as possible. Collaborate with users and customers in order to confirm that test results confirm the wanted functionalities. Provide the means for users to independently test the product.

2. User representative

Collaborate with developers about the results of tests and confirm that results show that wanted functionalities are satisfied. Independently test(try out) the product and inform the developers about problems if any.

12. Risks and contingencies

Missing out to test some parts or features ->
Effort and work should invested be to identify and test all the required functionalities. Developers should collaborate with customers and potential users and provide them with means to try things out independently so the chance of missing something out is as small as possible.

Tests which cannot be repeated ->
Make sure all tests are wrote in the way which allows them to be repeated again. This includes excluding random generations, tests depending on the time of running etc.

Trying to fix a failed test on correct component as a result of tests of its depending components being bad or incomplete ->
Always make sure when testing a component that all the tests of the components on which this component

depends are complete and passed. If the bug arises on components that some component depends on it should be fixed and tests should be run again until completed and passed. Also all the testing on the component which was depending on the component in which bug was found should be start all over from beginning.

13. Approvals

Name	Title	Date yyyy-mm-dd	**Signature**