 <b>University of Zurich</b> <small>UZH</small>	<b>Test Procedure and Revision of Markup Generator</b>	Authors: S. Plüss, E. Esati, B. Solenthaler and G.R. Prinz Doc.No.: Mark-REV-001 Date: 2017-11-18 Number of Pages: 12
--	--	---

## Contents

1	Introduction	2
2	Testing and Revision	2
2.1	General Implementation Procedure	2
2.2	Implementation Example	3
2.3	Test Results and Revision	4
3	Design Decisions	5
4	Literature	6

## Revision History

Date	Ver- sion	Description	Author(s)
2017-11-18	0.1	Document created	Gian Raphael Prinz
2017-11-18	1.0	Testing Results and Revision	Elfat Esati

# 1 Introduction

This document mainly describes the test procedure applied to the «Markup Generator» and the results coming from it. In a second step, it gives a short overview about the design decisions made in the project.

## 2 Testing and Revision

All tests implemented and applied to the «Markup Generator». are described within the document MARK-TEST-001. Thus, they are not listed again. Instead, the test procedure and a single implementation example will be described in the following.

### 2.1 General Implementation Procedure

The test classes were implemented by using «AutoTest», a tool within EiffelStudio that allows creating, managing and executing software tests. Multiple test classes consisting of a test set were introduced for structuring the test process. The tests were written manually and not extracted or autogenerated. The anatomy of a test class looks always as follows:

```
note
  description: "[
    Eiffel tests that can be executed by testing tool.

  ]"
  author: "EiffelStudio test wizard"
  date: "$Date$"
  revision: "$Revision$"
  testing: "type/manual"

class
  MY_TEST_CLASS

inherit
  EQA_TEST_SET

feature -- Test routines

  my_test
    -- New test routine
  do
    assert ("not_implemented", False)
  end

end
```

The example class MY\_TEST\_CLASS from above inherits from EQA\_TEST\_SET where all test classes have to inherit from. The feature my\_test is a procedure that is part of MY\_TEST\_CLASS accepting no arguments and is thus qualified as single test.

Executing a test can result in three possibilities:

- 1) The test is successful
- 2) The test is failing

3) The test result is unresolved

The outcome of a test is displayed in Eiffel Studio. If necessary, measures can be applied.

## 2.2 Implementation Example

The following test was described in the test suite. It is about creating documents, the frame of Markup report.

Table 1: Test case about the creation of documents

Test Case ID	T.3.1.1.003
SRS ID of Functionality	R3.2.1.003
Name of Functionality	Creating Documents
Classes and Routines	Class: Document, routine: make
Set Up	An instance of Document class should be declared.
Tear Down	None
Test Data	Internal name
Oracle	A document object should be created. It should have an internal name that was determined within the creation procedure. No messages should be displayed. Furthermore, no errors should be raised. There is no item or value returned.

It states that an instance of DOCUMENT should be created consisting of an internal name specified during the creation procedure. Furthermore, no errors or messages should be displayed. It is also not allowed for items or values to be returned. Based on these test specifications and the anatomy of a test class, the test class DOCUMENT\_TEST was implemented.

```
note
    description: "[
        Eiffel tests that can be executed by testing tool.
    ]"
    author: "EiffelStudio test wizard"
    date: "$Date$"
    revision: "$Revision$"
    testing: "type/manual"

class
    DOCUMENT_TEST

inherit
    EQA_TEST_SET

create
    default_create

feature {NONE} -- Well known field values

    Name: STRING_8 = "DOCUMENT-A"

    Name2: STRING_8 = ""
```

```

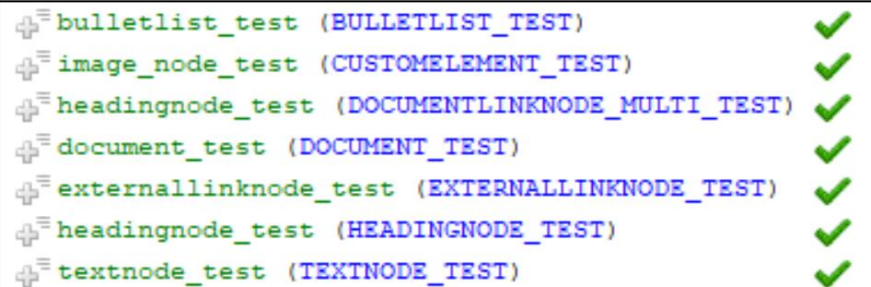
feature -- Test routines

    document_test
        -- New test routine
        local
            document: DOCUMENT
            document2: DOCUMENT
        do
            create document.make (Name)
            create document2.make (Name2)
            assert ("Document name", document.getname.is_equal (Name))
            assert ("Document name", document2.getname.is_equal (Name2))
        end
    end

end -- class DOCUMENT_TEST

```

The test class DOCUMENT\_TEST consists of a single test, called document\_test. It creates two objects of type DOCUMENT, one with a meaningful name and another with an empty string as name. At the end, the names assigned to the objects are validated by comparing them with the string values used during the construction procedure.



```

+≡ bulletlist_test (BULLETLIST_TEST) ✓
+≡ image_node_test (CUSTOMELEMENT_TEST) ✓
+≡ headingnode_test (DOCUMENTLINKNODE_MULTI_TEST) ✓
+≡ document_test (DOCUMENT_TEST) ✓
+≡ externallinknode_test (EXTERNALLINKNODE_TEST) ✓
+≡ headingnode_test (HEADINGNODE_TEST) ✓
+≡ textnode_test (TEXTNODE_TEST) ✓

```

Figure 1: Outcome of the test execution in EiffelStudio

As shown in figure 1, the test document\_test was run successfully. This is indicated by the green hook besides the test. Thus, no revision seems to be necessary.

All other test cases described in document MARK-TEST-001 were implemented analogously.

## 2.3 Test Results and Revision

With unit testing we have instantiated partial code of the library by testing them independently from other parts. We have simultaneously tested the invocation of features from different classes and verifying that the test brings the correct results for which it is tested. The results of the implemented test cases have shown no bugs. Thus, no revision of the existing code was necessary. We understand now, that the requirements specification also determines the test plan, from which we can say that implementation, test plan and test cases are derived directly from requirements specification document.

There are other implemented features for testing but are part of the Gobo Eiffel Project, under which there is a library for testing purposes and provides a number of implemented features useful for testing. However, for simplicity reasons we didn't want to install the library such that

it may be overhead for all group members as then everyone has to install the Gobo Eiffel library.

### 3 Design Decisions

There was no need to adapt the design of the «Markup Generator» as described in the document MARK-API-001. Thus, the design decisions explained there are described in the following one-to-one.

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Figure 2: Design Patterns

Out of the design patterns listed in figure 2, the following were considered in detail: visitor, composite, builder, decorator, and abstract factory. But only two of them were applied to the «Markup Generator», the composite and visitor pattern. The composite pattern allows to represent components if both, primitives, and container objects must be represented in part-whole hierarchies. It allows the client to ignore the difference between compositions of objects and individual objects and to treat all objects being part of the composite pattern uniformly [1]. When designing the «Markup Generator», this is exactly what is needed. Some elements in markup languages can be considered as single elements whereas others are rather composites. But the clients should not care about this difference. Thus, applying the composite pattern allows the client to easily manipulate with tags and handle it as unit pair, instead of dealing with opening and closing tags separately and it's much easier to add new kinds of components. The visitor pattern on the other hand represents an operation to be performed on the elements of an object structure [1]. Within the scope of this project, the operations are responsible for creating the correct markup language elements in form of strings. They are performed on the leafs and composites described by the composite pattern. It's easily possible to introduce new markup languages by adding a new visitor. There is no need for large changes. Another advantage is that related behavior is gathered in a concrete visitor and not spread over the classes defining the object structure. Unfortunately, the visitor makes it difficult to add new concrete element classes [1]. Together, the visitor and composite pattern ensure maintainability and extendibility of the «Markup Generator».

The Builder, Decorator and Abstract Factory patterns have been dismissed as to not over-complicate the project scope or have not brought enough advantage over considered patterns.

Abstract Factory, for example, does not really add simplicity or functionality as not much sub-classes have to be created so far.

## **4 Literature**

[1] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1993). Design patterns: Abstraction and reuse of object-oriented design. In European Conference on Object-Oriented Programming (pp. 406-431). Springer, Berlin, Heidelberg.