University of Zurich UZH	API Design for Markup Generator	Authors: B. Solenthaler, G. R. Prinz, E. Esati & S. Plüss Doc.No.: MARK-API-001 Date: 2017-11-24 Number of Pages: 7

Contents

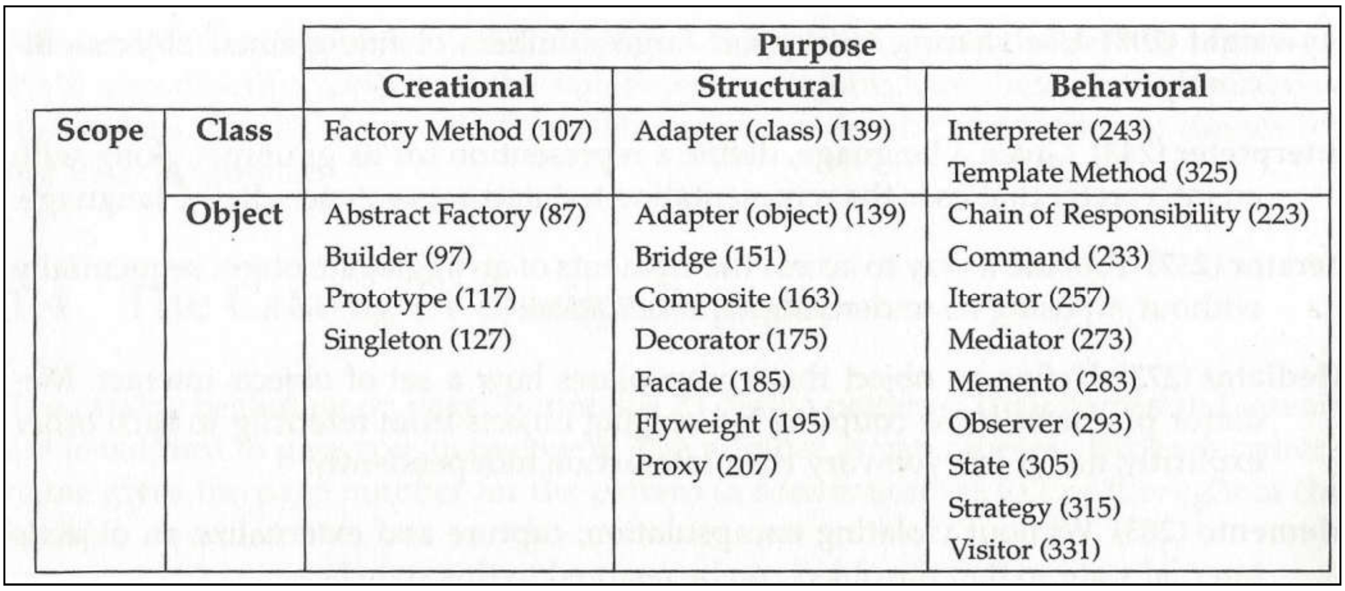
Revision History

Date	Version	Description	Author(s)
2017-10-29	0.1	Draft	Elfat Esati
2017-10-29	0.2	Draft	Benjamin Solenthaler
2017-11-25	1.0	Final version	Gian Raphael Prinz
2017-11-26	1.1	Revision	Gian Raphael Prinz
2017-11-27	1.2	Revision	Benjamin Solenthaler
2017-11-29	1.3	Revision	Severin Plüss
2017-11-30	1.4	Revision	Gian Raphael Prinz
2017-12-04	1.5	Revision	Severin Plüss

1 Introduction

The «Markup Generator» is an Eiffel-based library that allows to create documents styled in different markup languages. A good software architecture must be applied to ensure its extendibility and maintainability. Using design patterns is an appropriate tool to guarantee these characteristics. Thus, this document gives a short introduction about design patterns and explains in a second step the structure of the public API and the design patterns used. The document closes with the appendix showing the structure of the design patterns used and the class diagram.

2 Design Patterns



		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 1: Design patterns classified by scope and purpose [1]

Design patterns can be classified by purpose and scope (figure 1). The first criterion describes what a pattern does and is structured of three categories: creational, structural as well as behavioral patterns. Creational patterns focus on the process of object creation whereas structural patterns handle the composition of classes or objects. Behavioral patterns determine how classes or objects interact and distribute responsibility. The second criterion, scope, describes whether a pattern applies primarily to classes or objects. Class patterns focus on the relationship between classes and their descendants. Object patterns deal with object relationships [1].

3 Design Reasoning

3.1 Considered Patterns

Out of the design patterns listed in figure 1, 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 overcomplicate 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 subclasses have to be created so far.

3.2 Implementation of the Visitor and Composite Pattern

Composite: The composite pattern consists of three elements, the component, leaf and composite (figure 3). The component declares an interface for accessing and managing the child components as well as for objects in the composition. It also implements behavior for the interface common to all classes. In the «Markup Generator», the composite is represented by the class A_COMPOSITE. Leafs represent leaf objects and are not allowed to have children. The composite defines behavior for components that can have children. The classes A_LEAF and A_CONTAINER are used for this reason. Both inherit from the component. The effective classes used for the required elements inherit from either A_CONTAINER or A_LEAF. The detailed structure can be viewed in the class diagram (figure 2)

Visitor: The visitor pattern consists of the visitor and multiple concrete visitors represented as subclasses of the visitor (figure 4). The visitor declares visit operations for each concrete element in the object structure. A concrete visitor implements each operation declared by the visitor. The «Markup Generator» has three classes constituting the visitor pattern: A_VISITOR (visitor), HTML_VISITOR (concrete visitor) and MARKDOWN_VISTOR (concrete visitor). The object structure on which the visitor builds on is represented by the composite pattern. A_VISITOR has visit operations for each leaf and component element (subclasses of A_CONTAINER and A_LEAF). The component and leaf elements of the composite pattern on the other hand have defined an accept operation that takes either HTML_VISITOR or MARKDOWN_VISITOR as argument. This is how the classes, belonging to both the composite and visitor pattern, interact together.

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.

5 Appendix

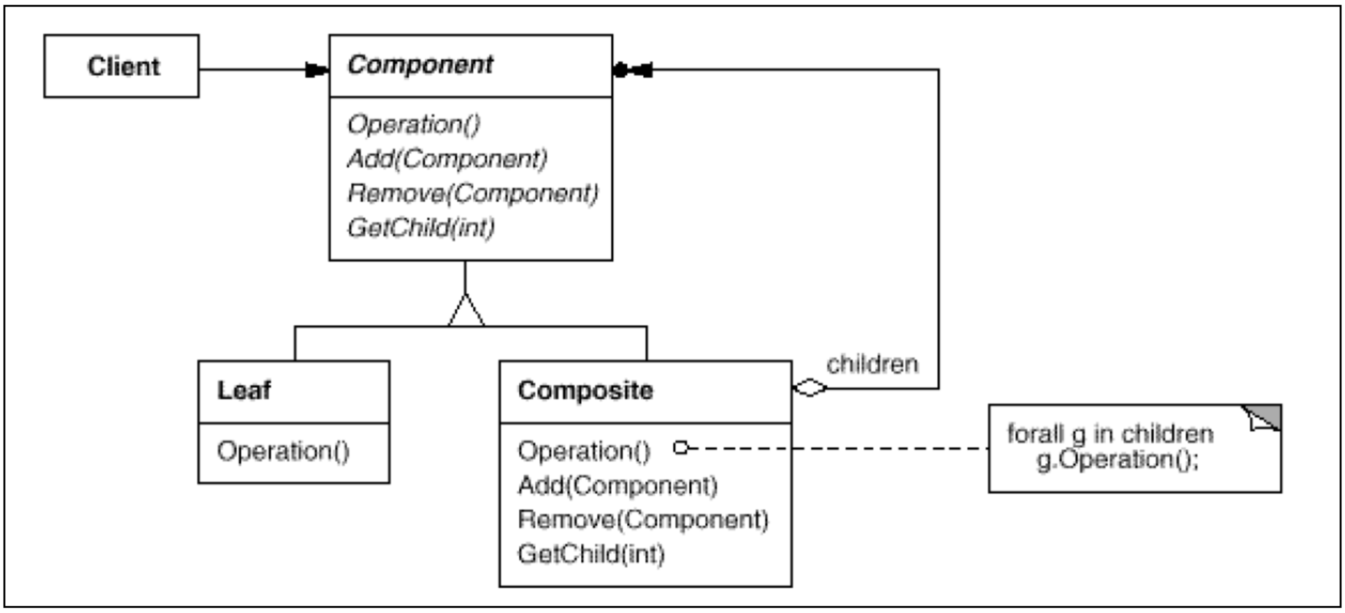


Figure 3: Structure of the composite pattern [1]

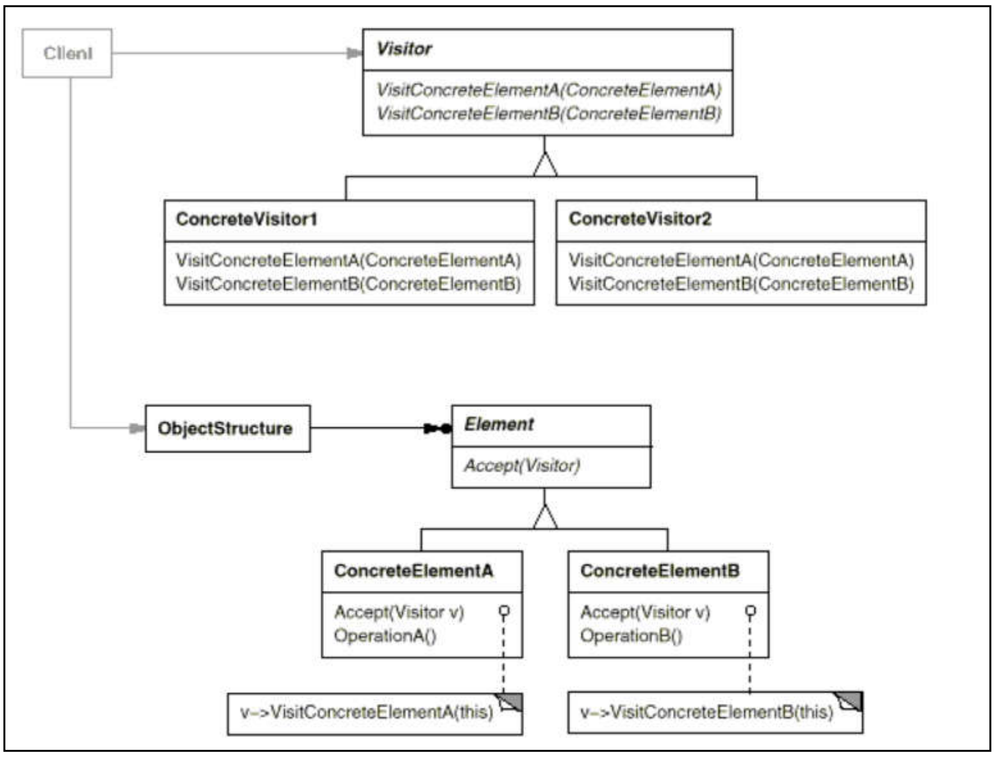

Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

Figure 4: Structure of the visitor pattern

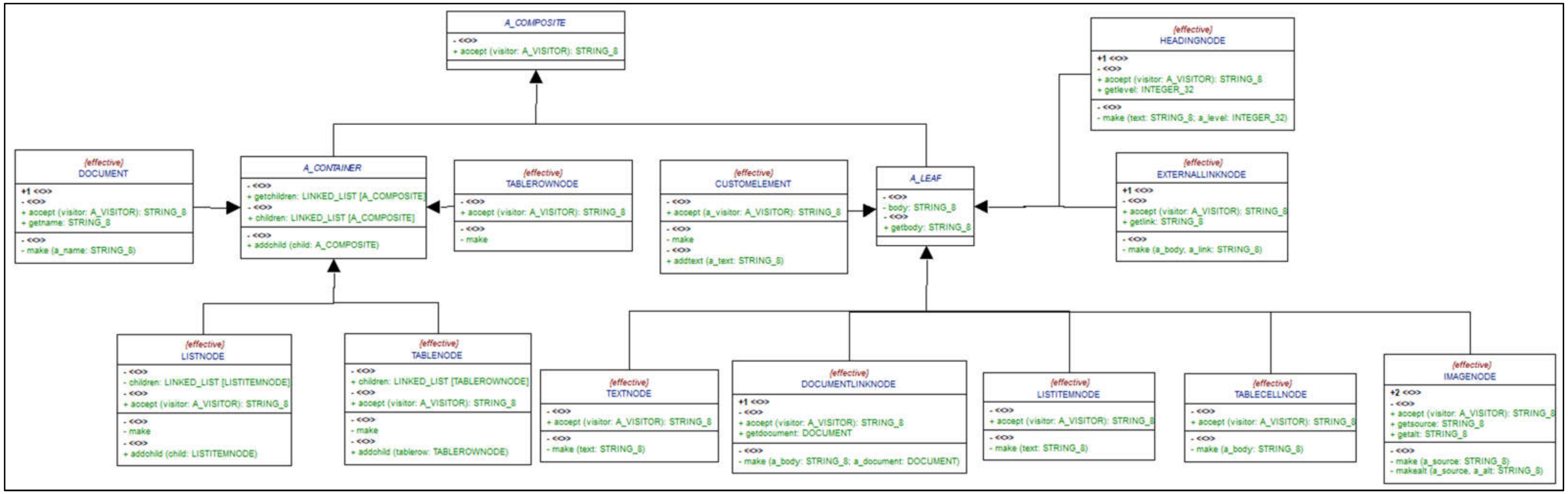
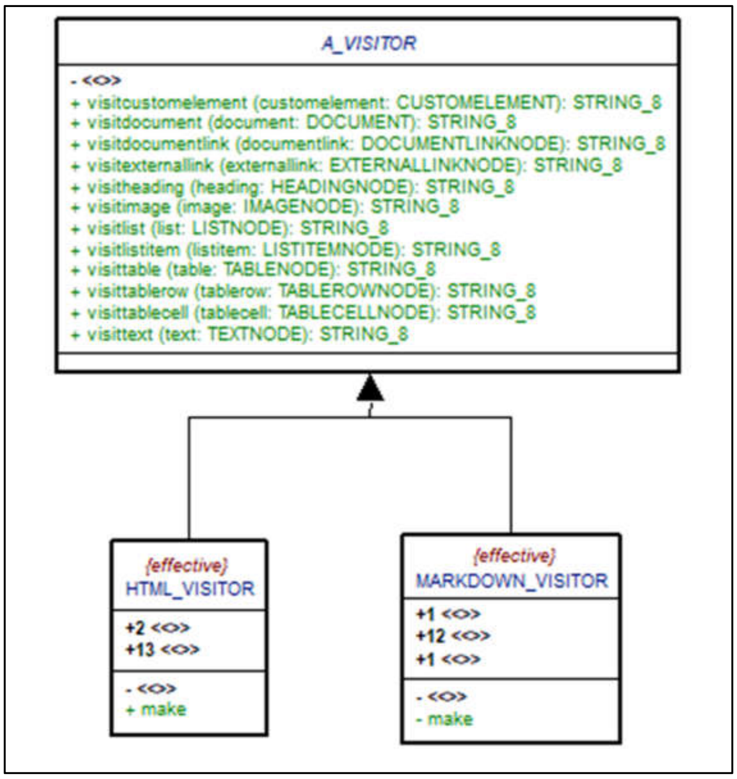

A_COMPOSITE
- <<O>>
+ accept (visitor: A_VISITOR): STRING_8
{effective}
DOCUMENT
+1 <<O>>
- <<O>>
+ accept (visitor: A_VISITOR): STRING_8
+ getname: STRING_8
- <<O>>
- make (a_name: STRING_8)
A_CONTAINER
- <<O>>
+ getchildren: LINKED_LIST [A_COMPOSITE]
- <<O>>
+ children: LINKED_LIST [A_COMPOSITE]
- <<O>>
+ addchild (child: A_COMPOSITE)
{effective}
TABLEROWNODE
- <<O>>
+ accept (visitor: A_VISITOR): STRING_8
- <<O>>
- make
{effective}
CUSTOMELEMENT
- <<O>>
+ accept (a_visitor: A_VISITOR): STRING_8
- <<O>>
- make
- <<O>>
+ addtext (a_text: STRING_8)
A_LEAF
- <<O>>
- body: STRING_8
- <<O>>
+ getbody: STRING_8
{effective}
HEADINGNODE
+1 <<O>>
- <<O>>
+ accept (visitor: A_VISITOR): STRING_8
+ getlevel: INTEGER_32
- <<O>>
- make (text: STRING_8; a_level: INTEGER_32)
{effective}
EXTERNALLINKNODE
+1 <<O>>
- <<O>>
+ accept (visitor: A_VISITOR): STRING_8
+ getlink: STRING_8
- <<O>>
- make (a_body, a_link: STRING_8)
{effective}
LISTNODE
- <<O>>
- children: LINKED_LIST [LISTITEMNODE]
- <<O>>
+ accept (visitor: A_VISITOR): STRING_8
- <<O>>
- make
- <<O>>
+ addchild (child: LISTITEMNODE)
{effective}
TABLENODE
- <<O>>
+ children: LINKED_LIST [TABLEROWNODE]
- <<O>>
+ accept (visitor: A_VISITOR): STRING_8
- <<O>>
- make
- <<O>>
+ addchild (tablerow: TABLEROWNODE)
{effective}
TEXTNODE
- <<O>>
+ accept (visitor: A_VISITOR): STRING_8
- <<O>>
- make (text: STRING_8)
{effective}
DOCUMENTLINKNODE
+1 <<O>>
- <<O>>
+ accept (visitor: A_VISITOR): STRING_8
+ getdocument: DOCUMENT
- <<O>>
- make (a_body: STRING_8; a_document: DOCUMENT)
{effective}
LISTITEMNODE
- <<O>>
+ accept (visitor: A_VISITOR): STRING_8
- <<O>>
- make (text: STRING_8)
{effective}
TABLECELLNODE
- <<O>>
+ accept (visitor: A_VISITOR): STRING_8
- <<O>>
- make (a_body: STRING_8)
{effective}
IMAGENODE
+2 <<O>>
- <<O>>
+ accept (visitor: A_VISITOR): STRING_8
+ getsource: STRING_8
+ getalt: STRING_8
- <<O>>
- make (a_source: STRING_8)
- makealt (a_source, a_alt: STRING_8)

Figure 5: Implemented composite pattern


A_VISITOR
- <<>>
+ visitcustomelement (customelement: CUSTOMELEMENT): STRING_8
+ visitdocument (document: DOCUMENT): STRING_8
+ visitdocumentlink (documentlink: DOCUMENTLINKNODE): STRING_8
+ visitexternallink (externallink: EXTERNALLINKNODE): STRING_8
+ visitheading (heading: HEADINGNODE): STRING_8
+ visitimage (image: IMAGENODE): STRING_8
+ visitlist (list: LISTNODE): STRING_8
+ visitlistitem (listitem: LISTITEMNODE): STRING_8
+ visittable (table: TABLENODE): STRING_8
+ visittablerow (tablerow: TABLEROWNODE): STRING_8
+ visittablecell (tablecell: TABLECELLNODE): STRING_8
+ visittext (text: TEXTNODE): STRING_8
{effective}
HTML_VISITOR
+2 <<>>
+13 <<>>
- <<>>
+ make
{effective}
MARKDOWN_VISITOR
+1 <<>>
+12 <<>>
+1 <<>>
- <<>>
- make
Figure 6: Implemented visitor pattern