TAMPERE UNIVERSITY OF TECHNOLOGY

José Héctor Zárate Rodríguez

MAINTAINING CONSISTENCY IN XML-BASED CONFIGURATIONS: A PATTERN-DRIVEN APPROACH

Master of Science Thesis

Examiner: Adjunct Prof. Imed Hammouda (TUT)
Supervisor: M.Sc. Mikko Matilainen (Sasken)
Examiner and topic approved in the Faculty of Computing and Electrical Engineering council meeting on 7th of December 2011

Preface

This thesis work was done for Sasken Finland and I would like to express my most sincere thanks to all my colleagues and especially to Mikko Matilainen for their help, guidance and patience that made this thesis work possible.

I would like to thank my professors at Tampere University of Technology for their vocation to teach and help me to be a more qualified professional. In particular I would like to thank my examiner Imed Hammuda for his guidance during this thesis writing period.

Finally I give thanks to God for the opportunity to live and experience these years in Finland. I want to express my deepest gratitude to my mother, father, brother and sister for their love and unconditional support without which the goal that this thesis concludes would have not been possible. Also, I am thankful to my girlfriend for encourage me to work hard and for her understanding and moral support during this thesis writing.

Abstract

TAMPERE UNIVERSITY OF TECHNOLOGY

Department of Information Technology

Institute of Software Systems

Master’s Degree Programme in Information Technology

ZÁRATE RODRÍGUEZ, JOSÉ HÉCTOR: Maintaining consistency in XML-based configurations: A Pattern-driven approach

Master of Science Thesis: 46 pages, 7 Appendix pages

December 2011

Major: Software systems.

Examiner: Adjunct Prof. Imed Hammouda

Supervisor: MSc. Mikko Matilainen

Keywords: consistency, specialization patterns, tool support, XML configurations

Several software artifacts are used along the software development process. These are used as input to perform tasks that produce new artifacts. Inconsistencies between related artifacts at different levels of abstraction can arise during the software development process.

Specialization patterns abstract the structural description of a framework extension point, and can be used to qualify an element of the framework with a status and get a list of actions needed to change the element to a different status. A tool built using these patterns can then provide an overview of the framework status and identify the actions needed to reach a status of consistency.

This thesis first identifies a source of inconsistencies between three artifacts that are generated and used in a framework (diagrams, XML schema and XML configuration files). Second this thesis shows how to adapt specialization patterns to validate XML configuration files. Finally this thesis provides a tool support that utilizes specialization patterns to guide the developer in the process of editing the configuration of the framework while maintaining the consistency of the three artifacts.

Table of figures

1. Introduction

Lehman and Belay concluded based on their studies of OS/350 and other large systems that, “the need to minimize software cost suggests that large-program structure must not only be created but must also be maintained if decay is to be avoided or postponed. Planning and control of the maintenance and change process should seek to ensure the most cost-effective balance between functional and structural maintenance over the lifetime of the program. Models, methods and tools are required to facilitate achieving such balance”. [1]

1.1. Motivation

Software development companies are moving forward to create software that can be reused, adapted for several different requirements and that can evolve instead of just developing bespoken software (software developed according to the needs of an individual or organization [2]). A common approach is to build applications on top of frameworks. A framework is a reusable, semi-complete application that can be specialized and adapted to produce custom applications [3]. These applications have access to the features provided in the framework and by using a proper configuration input this application can be parameterized to solve specific problem cases.

The first law of software evolution formulated in 1974 by Lehman and Belady states that an E-type system, a program that is embedded in the real world and has become part of it, must be continually adapted or it becomes progressively less satisfactory [1]. In other words software must evolve in order to continue pleasing its users. The evolution of applications based on frameworks implies modifications in the application and the framework. These modifications require changes that affect several items that were produced during the development phase of the system. The new changes need to be documented to provide traceability of the changes. Addendums and updates to artifacts such as software specification requirements and software design description need to be done to reflect the new needs and finally the source code must be updated with the new implementation. Keeping all related parts that are impacted for a single change in consistent state is a laborious and error prone task in which having a support tool to trace and maintain the relation up-to-date is important.

Parameterized applications can be adapted to new circumstances by altering their configuration. A commonly used approach to parameterize an application is Extensible Markup Language (XML). XML is a platform-independent programming language designed to carry data and to be self-descriptive [4]. The structure of an XML file can be well defined in an XML schema file. These characteristics make XML useful to handle structured application configuration settings. For instance Microsoft® .NET Framework uses XML configuration files since its first version to parameterize applications [5].

Maintaining consistent specifications, technical documentation, configuration files and source code is a time consuming task that may involve several people from different levels of the organization. Such a task turns out to be expensive and consequently easily dropped out of the scope of a project. Inconsistencies can be introduced during the development phase and remain hidden until a late phase of the development or even until the product is released and maintenance is needed. In the latter cases the impact and cost of inconsistencies can severely affect the project. Maintenance phase is also a common source of new inconsistencies if changes required by the maintenance task are not documented and all documents related to the change are not updated accordingly. A mechanism to link two or more different artifacts so that changes in either of them are reflected in the others would help to maintain the consistency of such artifacts. This need is not specific to one software development process and is the motivation of the thesis work.

1.2. Objectives

This thesis has four objectives. The first objective is to introduce to the reader the concepts of software artifacts and inconsistency of artifacts. The second is to show the inconsistency problem in the concrete case of XML configuration artifacts. The third is to adapt specialization patterns to provide a mechanism to detect inconsistencies between XML configuration files and their schemas. The last objective of this thesis is to develop a functional prototype support tool for editing XML configuration files based on specialization patterns as an approach to maintain the consistency of the artifacts.

The prototype tool developed during this thesis can be used to visualize XML configuration artifacts as a hierarchical diagram tree. The tool provides a user interface that allows the user to alter the tree and evaluates the consistency of the artifact after every change. In the case when the artifact becomes inconsistent the user interfaces guides the user to solve the inconsistencies. This tool is used specifically to edit meteorological message configurations and enforces the use of standard method to transform the message specifications into design diagrams and these into XML configuration files. The tool can be used to produce these artifacts, which satisfy the needs of at least two stakeholders: software designers and developers.

1.3. Structure

This thesis is divided into 8 chapters with this introduction being the first of them. The Chapter 2 defines important concepts such as software artifacts, consistency of artifacts and presents XML configurations as a concrete example of software artifacts. Chapter 3 introduces specialization patterns, its core elements: roles, contracts, tasks and program elements and how specialization patterns can be adapted to maintain the consistency of XML files with their schema. Chapter 4 presents a case study about a framework to generate meteorological messages in which different software artifacts are involved and are currently subject of inconsistencies. Chapter 5 concentrates on MENE (MEssage coNfiguration Editor) a support tool that uses specialization patterns to tackle inconsistencies between artifacts in the process of generating meteorological messages. Chapter 6 presents the use MENE. Chapter 7 lists the results of this thesis work. Chapter 8 is final words and conclusions.

2. Software artifacts and consistency

An artifact is a piece of information that is produced, modified, or used by a process. Artifacts are the tangible products of the project, which are created or utilized, while working towards the final product. Artifacts are used as input by workers to perform an activity; and they are the result or output of such activities [6]. Artifacts may take various shapes or forms. Table 1 lists some of the most common artifacts and examples of them.

Table 1 Common software artifacts

Artifact type	**Artifact example**
Document	Business case, software architecture document, software specification requirements (SRS), software design description (SDD), test plan, etc.
Diagrams	Sequence diagram, state diagram, activity diagram, collaboration diagram, entity-relationship model (ER), class diagram, use cases, etc.
Code	Source code, database scripts, batch files, web pages, XML configuration files, etc.
Executable	Programs, scripts, plug-ins.
Results	Test results, unit test results, bug report, etc.
Others	Data, prototypes, etc.

Software systems are developed following phased processes in which software engineering complexities are tackled by means of subsequent refinement activities [7]. These phases guide the development from its first conception to realization and maintenance [8]. Each software development phase produces several artifacts. For example Software Specification Requirements (SRS) is created during the requirement analysis phase; Software Design Description (SDD), use cases, sequence diagrams, among other are created during the design phase; and source code, configuration files, executables and API documentation during the implementation phase. These artifacts are tightly related as shown in Figure 1. The outcome artifact of one phase is the input for the next phase. However a phased process does not guarantee traceability of how requirements evolve into design and design into code [7] and neither the consistency of the artifacts created in each phase.

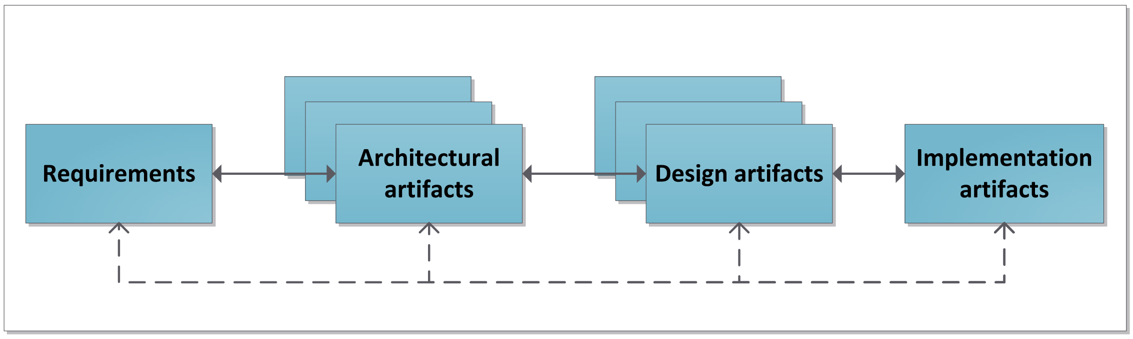


Figure 1 Relation between software artifacts.

Figure 1 shows the relation between artifacts. The solid lines represent a direct relation between artifacts. The dashed line shows the indirect relation between artifacts. When due to changing circumstances the software product needs to be adapted, the relation between requirements and design or design and code can easily break and the artifacts become inconsistent. For example, a specific requirement is converted into an architectural design artifact. Then a program element (piece of source code) satisfies that design decision and therefore the specific requirement. If a requirement requires a small change, it probably needs to be reflected in the design artifact and in the program element. Similar situation occurs when a problem is found and fixed in the code. This action can affect the architectural design causing that the requirements satisfied by the design decision may not be fulfilled.

2.1. Consistency of artifacts

Systems evolve by extending or adapting to new use cases and scenarios. This evolution includes modifications performed at all stages of the software development lifecycle, from inception to retirement and is handled differently depending of the development methodology used. Such modifications are performed on a wide range of software artifacts ranging from those at a high level of abstraction to architecture, design, and source code level artifacts [9]. Keeping the artifacts affected by a single change up-to-date and in a state of consistency is important in all phases of software development, regardless of the methodology used. It is especially in the later phases of the process for instance in the maintenance phase that this state of consistency is critical [10]. In practice maintaining these artifacts consistent with each other is a labor-intensive process of manual reviewing and as such a highly error prone activity. In order to avoid artifacts to become inconsistent in different phases of the development process, automatic support for maintaining the consistency among them or a mechanism to trace changes should be available. For example Aldrich, J. et all in their paper ArchJava:

Connecting Software Architecture to Implementation [11] propose an extension for Java language that unifies software architecture with implementation, ensuring the implementation is consistent with the architectural constraints.

In this work we define an artifact to be consistent with other artifact when a relation between both exists and the artifact in question does not present any inconsistencies with the related artifact. We use the definition of inconsistency given by Bashar Nuseibeh: “An inconsistency occurs if and only if a (consistency) rule has been broken” [12].

2.2. Example artifact: XML configuration files

XML artifacts play an important role in current software practices, especially in web application [13]. They have become the prime standard for data exchange on the Web [14]. For instance web feeds use XML to deliver information to their subscribers. Listing 1 shows an XML snippet containing the weather information of a city. Applications can use this XML as input to perform specific tasks.

Listing 1 Weather information exchange example using XML

```
<weather>
  <location city="Tampere" country="Finland"/>
  <units temperature="C" distance="km" pressure="mb" speed="km/h"/>
  <condition text="Fair" code="34" temp="8" date="Sat, 15 Oct 2011 5:20 pm EEST"/>
  <wind chill="7" direction="190" speed="8.05"/>
  <atmosphere humidity="76" visibility="9.99" pressure="1015.92" rising="0"/>
  <astronomy sunrise="8:03 am" sunset="6:09 pm"/>
</weather>
```

Besides of being a standard for data exchange XML artifacts are also widely used to store application settings. Humans can read XML and computer can easily parse and process it. This characteristic is one of the reasons why XML configuration files are widely used. For instance the compiler for Flex SDK, a framework to build mobile and desktop application using the same base code [15], uses XML configuration files to define the compiler switches to use. Listing 2 shows part of the compiler node of the configuration file of Flex compiler.

Listing 2 Flex compiler configuration file snippet

```
<compiler>
  <!-- Turn on generation of accessible SWFs. -->
  <accessible>false</accessible>
  <!-- Specifies the locales for internationalization. -->
  <locale>
    <locale-element>en_US</locale-element>
  </locale>
  <!-- Enable post-link SWF optimization. -->
  <optimize>true</optimize>
</compiler>
```

An important characteristic of XML artifacts is that they can be validated against an XML schema (see section 2.2.2) that specifies type constraints and integrity constraints

[16]. If during the validation process the XML tree satisfies all the constraints defined in the XML schema, then the artifact is considered to be consistent [17]. This validation is an important component of quality assurance.

2.2.1. XML language

Standard Generalized Markup Language (SGML) and Extensible Markup Language (XML) are "meta" languages because they are used for defining markup languages. A markup language defined using SGML or XML has a specific vocabulary (labels for elements and attributes) and a declared syntax (grammar defining the hierarchy and other features) [18].

XML is a simple and very flexible text format derived from SGML (ISO 8879), originally designed to meet the challenges of large-scale electronic publishing. XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere [4].

2.2.2. XML schema

The structure of an XML document can be well defined using an XML schema. An XML schema is a language for expressing constraints about XML documents. There are several different schema languages in widespread use, but the main ones are XML Document Type Definitions (DTDs), Schematron and W3C XSD (XML Schema Definitions) [19]. The table presented in appendix A shows a detailed comparison of these three XML schema languages.

XML DTD has limited capabilities compared to other schema languages. DTD uses a terse formal syntax that defines which elements and references may appear in the document.

Listing 3 DTD example.

```
<!DOCTYPE Student [
<!ELEMENT Name (#PCDATA+)>
<!ELEMENT StudentNumber (#PCDATA+)>
<!ELEMENT Notes (#PCDATA?)>
]>
```

XML schema is a W3C (The World Wide Web Consortium) recommendation [20]. It is written in XML itself. It expresses a set of rules that an XML document must fulfill in order to be considered valid.

Listing 4 XSD example.

```
<complexType name="Student">
    <sequence>
      <element  name="Name" type="string"/>
      <element  name="StudentNumber" type="string"/>
      <element  name="Notes" type="string"minOccurs="0"/>
    </sequence>
</complexType>
```

Schematron uses a list of XPath-based rules to define the structure of an XML file.

Listing 5 Schematron example.

```
<pattern name="Student data checks">

  <rule abstract="true" id="studentDataChecks">
    <assert test="Name">A student must have a name</assert>
    <assert test="StudentNumber">A student must have a student numner</assert>
  </rule>

  <rule context="Student">
    <extends rules="nameChecks"/>
  </rule>
</pattern>
```

The three example of XML schemas presented above can be used to validate the XML element *Student* presented below.

Listing 6 XML element example.

```
<Student>
  <Name>John</Name>
  <StudentNumber>123</StudentNumber>
</Student>
```

Despites the benefits given by the use of XML schemas, they also introduce some disadvantages. The use of XML-based syntax leads to verbosity in schema description, which makes schemas harder to read, write and maintain. In the case of XSD the W3C recommendation is complex, which makes XSDs difficult to understand and implement.

2.2.3. Consistency in XML-based configurations

An XML-base configuration is consistent with its XML schema if and only if the XML tree produced by the XML-based configuration satisfies all the constraints defined in the XML schema. Several tools exist to validate the compliance of an XML tree with a schema. However solving the inconsistencies is not always a trivial process.

2.3. Problems with inconsistent artifacts

Lack of consistency among artifacts over time, which can be seen as a mismatch between design and implementation, can cause a negative deviation of the software system architecture from its original design and results in software erosion [21] [22]. Software erosion is a common problem in software engineering processes, it comprises design erosion or architectural drift [23]. Design erosion is due to violations of the architecture while architectural drift is due to the insensitivity about the architecture (the architecturally implied rules are not clear). This insensitivity leads to inadaptability and results in an obscure architecture that is easily violated [24]. If the erosion level is too high the effort needed to fix it is such that redeveloping the system from scratch may become a viable option. Van Gurp and Bosch in their work Design Erosion: Problems

& Causes [23] present several successful cases where developing a new system instead of trying to fix an eroded system was a better solution.

Inconsistencies between requirements, design and code artifacts are one of the most common and most elusive errors in software design. Several works exist in this topic. Antoniol et al. suggest in their paper *design-code traceability for object-oriented systems* [7] the use of traceability to ensure consistency among software artifacts of subsequent phases of the development cycle. Looise in his work *inter-level consistency checking between requirements and design artifacts* [8] states that Model Driven Engineering (MDE) techniques can be used to support maintaining the consistency os software artifacts and propose the use of meta-models for requirements specification and architectural design to explicitly define the structure of these artifacts and to perform consistency checking on them. Igor Ivkovic et al. in their paper *tracing Evolution Changes of Software Artifacts through Model Synchronization* [9] present a framework whereby software artifacts at different levels of abstraction are represented by graph-based models that can be synchronized using model transformations.

3. Use of specialization patterns to maintain artifacts consistency

A general solution to keep different artifact consistent and, which can be adapted to different situations on the same context, requires a well-defined and structured instrument. This instrument must be able to abstract the problem and provide an acceptable solution for it. Design patterns own these characteristics.

3.1. Design patterns

Dirk Riehle and Heinz Züllighoven, in their work, understanding and using patterns in software development, give a good definition of the term *pattern* which is broadly applicable: “a pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts” [25].

A design solution in a specific area of expertise can be documented by using patterns. This approach is commonly use in different disciplines ranging from architecture to computer science. A pattern explains the reason why a particular situation generates a problem and why a proposed solution is well-accepted. In the field of software engineering a well-known and commonly used type of pattern is *design pattern* [26]. According to Gamma et al. [26] a design pattern has four essential elements:

- Unambiguous name that identifies the pattern.
- Problem definition that describes when to apply the pattern.
- Solution explaining elements and the relations which make up the design.
- Consequences describing the results and trade-offs of applying the pattern.

3.2. Specialization patterns

A specialization pattern is an abstract structural description of an extension point of a framework [27]. It can be seen as one approach to describe parts of software architecture and how such parts should be integrated to implement software products [28].

A specialization pattern is given in terms of *roles* to be played by *program elements*. The association between roles and program elements is called *contract.* If specialization patterns are applied to specific frameworks certain roles are played by fixed, unique program elements of the framework. When this occurs the roles are considered to be *bound*. The set of specialization patterns, contracts and the framework itself constitute a developer kit delivered for application programmers [27].

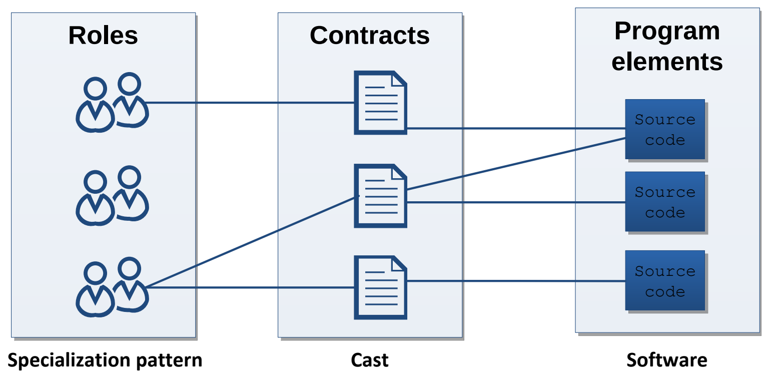


Figure 2 Relation between roles, contracts and program elements.

3.3. Roles, contracts and program elements

An application consists of *program elements*, which can in general be pieces of source code. A specialization pattern consists of *roles*. Each role is an abstraction of a required program element [28]. Roles contain dependencies, multiplicity and properties. The role properties can be of two types:

- Constraint properties which specify a requirement for a concrete program element, examples of this type of property are: inheritance, return type or overriding.

- Template properties which can be used for code generation. For instance default name, description and task title are properties of this kind.

Based on the role abstraction it is possible to generate a list of tasks that needs to be done in order to link program elements with its role. When a program element commits to play a particular role it is said that the program element has a *contract*. If a program element violates its contract, based on the specialization pattern, it is possible to generate a list of actions to instruct the developer to fix the violation. A role is *bound* to a program element if a contract that links both of them exists.

Applying or instantiating a specialization pattern is called *casting*. Casting means an incremental and interactive binding of program elements to unbound roles of the pattern [29].

3.4. Pattern graph

A specialization pattern can be represented using a directed acyclic graph. This graph is called pattern graph or pattern definition graph. The Figure 3 shows a definition graph of a simple reusable structure for a class that contains a member variable and a method called *get* that returns the variable.

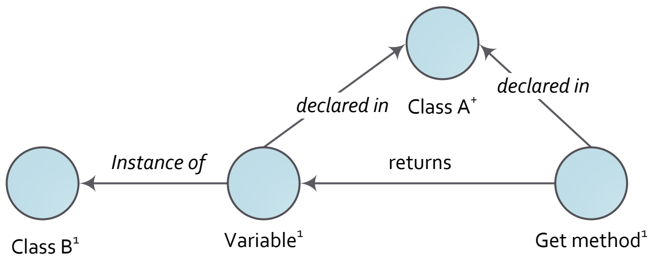
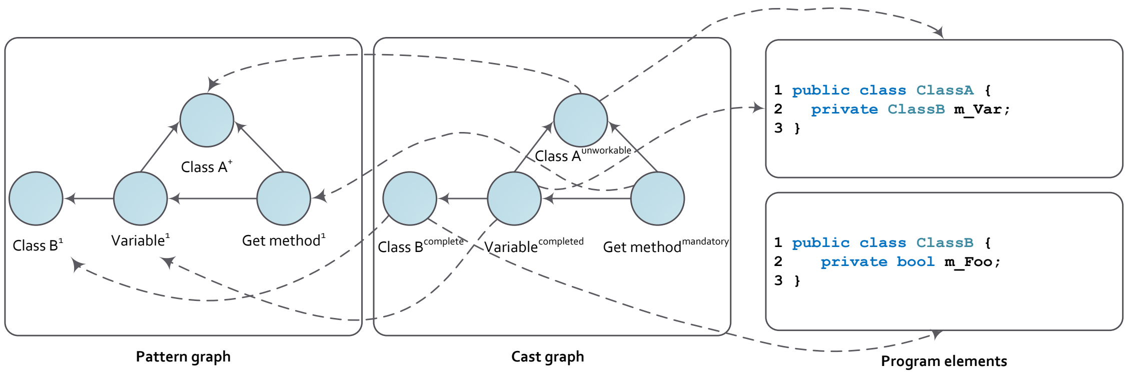


Figure 3 Example of a pattern graph.

The nodes of the graph represent roles and the directed edges between nodes are dependencies. The roles have multiplicity constraints: one to one (1), zero to one (?), one to infinity (+), and zero to infinity (*). The role multiplicity indicates if the role stands for a single program element or a set of elements. In this example the *Class A* has a variable of type *Class B* that is returned by the *get method*. Therefore *Class A* depends of the *Class B* role since in order to declare the variable of *Class A* the role *Class B* needs to be casted first.

3.4.1. Cast graph

Given that a cast is an instance of a specialization pattern it can be represented as a directed acyclic graph as well. A cast graph is used to show the state of the casting process at a specific time. The nodes of the graph represent contracts. A contract is always in a state which can be either *complete* or *unworkable*. Based on the multiplicity and dependencies a contract may be mandatory or optional. Figure 4 illustrates the cast graph and their relation to the pattern graph and program elements. The Class A contract is in unworkable state since it still depends of the contract *get method* to be completed. *Get method* is marked as mandatory because there is no program element fulfilling the contract. The rest of the tasks are in the complete state.



```
public class ClassA {
  private ClassB m_Var;
}
```

```
public class ClassB {
   private bool m_Foo;
}
```

Figure 4 Example of a simple cast graph.

3.4.2. Pattern Engine

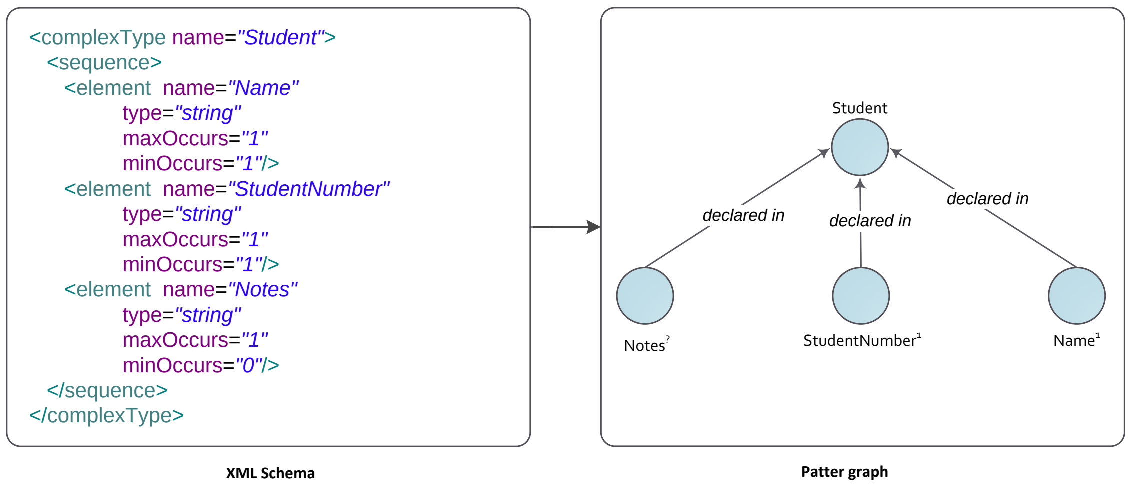
Pattern engine is an interpreter for the formal pattern specification language. It takes pattern specification as an input and produces tasks as output [28]. Pattern engine uses a casting algorithm to generate a dynamic list of tasks to be done in order to fulfill the contract requirements of a role. It monitors constantly the state of the cast graph. When the developer does actions that change the state of the graph cast, the pattern reacts to them updating the list of tasks to be done by adding new task or removing the tasks that are satisfied.

Pattern engine can make use of rules and heuristic to solve task that can be automatically completed. For instance, if a task to create a get method for a variable exists in the list the pattern engine can fix the violation by generating the method automatically.

In order to take advantage of the pattern engine it must be embedded in an environment that allows software development. Such environment should give to the developer the opportunity to interact with the source code, notify the pattern engine of the developer changes and provide the up-to-date list of tasks generated by the pattern engine.

3.5. Using specialization patterns to validate XML files

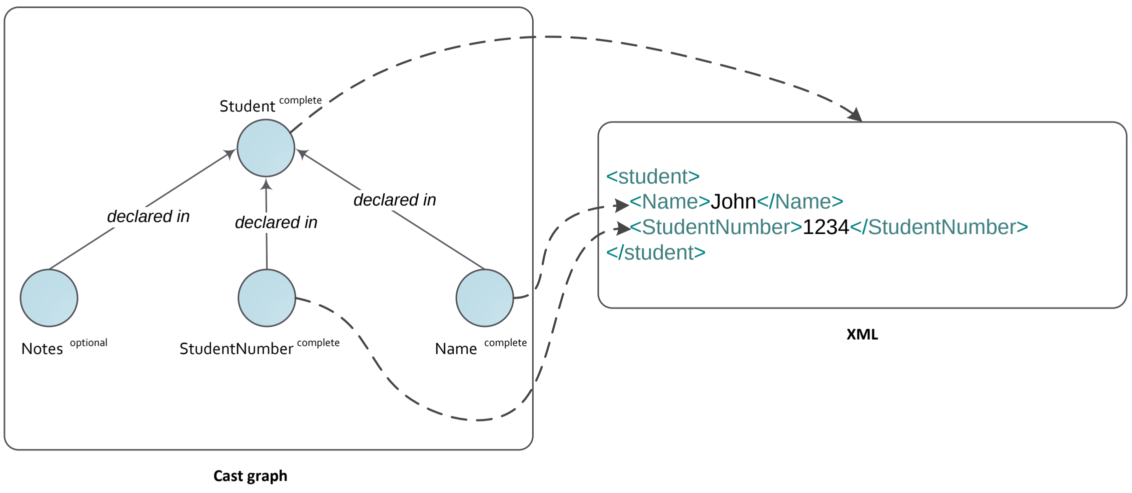
Specialization patterns can be used to validate an XML file against its XML schema. In order to do this, the XML schema needs to be converted into specialization pattern roles as shown in Figure 5 and the XML file needs to be converted into specialization pattern contracts as shown in Figure 6.



```
<complexType name="Student">
  <sequence>
    <element  name="Name"
          type="string"
          maxOccurs="1"
          minOccurs="1"/>
    <element  name="StudentNumber"
          type="string"
          maxOccurs="1"
          minOccurs="1"/>
    <element  name="Notes"
          type="string"
          maxOccurs="1"
          minOccurs="0"/>
  </sequence>
</complexType>
```

Figure 5 XML schema complex type converted to a specialization pattern graph.

An XML schema defines the structure and rules that an XML file must follow, similarly in specialization patters a role defines characteristics that instances of it need to implement. The Figure 5 shows an XML schema that defines a complex type Student with 3 properties: Name, StudentNumber and Notes. This complex type can be represented as a pattern graph that consists of 4 nodes: Student a node without outgoing edges and Notes, StudentNumber and Name nodes with edges to the Student node.



```
<student>
  <Name>John</Name>
  <StudentNumber>1234</StudentNumber>
</student>
```

Figure 6 Specialization pattern cast graph representation of an XML element.

Similarly an XML file can be represented as a cast graph. Each node of the graph maps to an XML element in the file. In the picture above the relation between nodes and XML elements is clearly shown.

Using specialization patterns to validate XML files against XML schemas opens the opportunity to create a tool support to detect inconsistencies between these two different artifacts and provide to the user a clear list of task to be done in order to reach the state of consistency.

4. Case study: Message generator framework

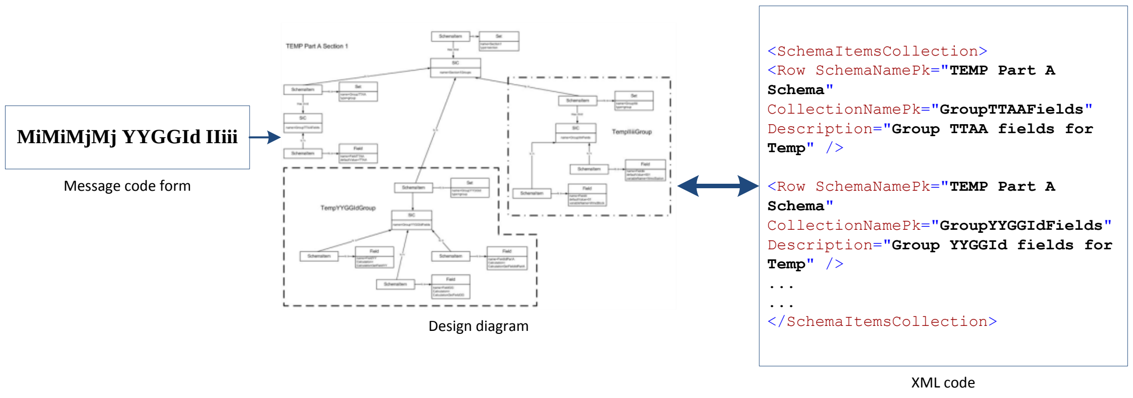
The efficient and timely movement of meteorological information is a fundamental requirement of modern meteorology. Observers record information about the environment and provide it to data processing centers so that forecast guidance products may be produced. Countries exchange information to enhance their forecasts and to produce global forecast models. The facility to move information quickly between centers, without regard to language, and in a format that may be processed by automated means is embodied in meteorological codes [30].

Meteorological messages encode meteorological information; the encoding process is systematic and can be done by a computer. Several upper air sounding system provide reports in the format of meteorological messages, for example: The Vaisala DigiCORA Sounding System MW31, Vaisala MARWIN Sounding System MW32 among other Vaisala sounding systems produce a wide variety of WMO messages [31] [32].

For purposes of discussion, an upper air sounding system consists of a radiosonde for making pressure, temperature, humidity (PTU) and wind measurements, a ground based antenna for receiving data from the radiosonde, and a system computer for controlling the antenna and providing reports and data outputs in various formats [33].

The process to produce different meteorology messages shares diverse steps no matter which message is being coded; therefore, the use of a general message generator framework is needed in order to abstract the common steps and facilitate the addition of new meteorological messages and modification of the existing ones.

In order to configure the system to generate a particular message a well-defined message code form needs to be expressed as a design diagram that later is converted into XML code. Therefore three different levels of artifacts are involved in the process and their relations are extremely tight. Maintaining a diagram and its code consistent is a tedious, difficult and error prone task that developers easily forget to carry out.



```
<SchemaItemsCollection>
<Row SchemaNamePk="TEMP Part A
Schema"
CollectionNamePk="GroupTTAAFields"
Description="Group TTAA fields for
Temp" />

<Row SchemaNamePk="TEMP Part A
Schema"
CollectionNamePk="GroupYYGGIdFields"
Description="Group YYGGId fields for
Temp" />
...
...
</SchemaItemsCollection>
```

Figure 7 Artifacts involve in the process of generating a message configuration.

The Figure 7 depicts the relation among the three different artifacts used for the system to generate a message configuration. This picture is not meant to provide details about the message code form, design diagram or the XML code, but just to illustrate the three different levels in which the same specification can be visualized and the importance of the consistency between each of them.

4.1. Meteorological message

Meteorological messages are used for the international exchange of observed and processed data required in specific applications of meteorology to various human activities and for exchanges of information related to meteorology [30].

The World Meteorological Organization (WMO) defines international meteorological codes in the WMO Manual No. 306 [34]. The codes are composed of a set of *code forms* made up of a group of letters representing meteorological or other geophysical elements. In messages, these groups of letters are transcribed into figures indicating the value or the state of the elements described [30]. As an example the section 1 of the TEMP code form, which is used for reporting pressure, temperature, humidity and wind of the upper regions of the atmosphere made by weather balloons released from the surface level is shown in Figure 8.

$$
\begin{array}{llll}
\text{SECTION 1} & M_iM_iM_jM_j & D\ldots.D^{**} & YYGGI_d \\
 & & \left\{\begin{array}{l} IIiii^{*} \\ \text{or} \\ 99L_aL_aL_a \end{array}\right. & \begin{array}{l} \\ \\ QcL_oL_oL_oL_o \quad MMMU_{La}U_{Lo}^{***} \quad h_0h_0h_0h_0i_m^{****} \end{array}
\end{array}
$$

Figure 8 Section 1 of FM 35–XI Ext. TEMP code form.

The TEMP code is the primary upper-air reporting code. TEMP codes are broken down into four parts, A, B, C, and D. Data at and below the 100 hPa level is reported in Parts A and B, and data above 100 hPa is reported in parts C and D. Additionally, each code part is divided into data sections. TEMP Section 1 contains the identification data. Each code field is mapped to figures representing real observations. **M_iM_i** defines the identification letters of the report; **M_jM_j** defines the part of the message. **YY** defines the day of the month and the units in which winds are measured. **GG** defines the actual time of observation. **I_d** is an indicator used to specify the pressure relative to the last standard isobaric surface for which the wind is reported. **II** defines the block number. **iii** defines the station number. [34]. The field codes **D….D**, **$L_aL_aL_a$**, **Q_c**, **$L_oL_oL_oL_o$**, **$MMMU_{La}U_{Lo}$** and **$h_0h_0h_0h_0i_m$** are only used on the variants TEMP SHIP and TEMP MOBIL and will not be explained in this example.

Table 2 TEMP A Section 1 example

Code format	Coded message
$M_iM_iM_jM_j$ YYGGI_d IIiii	`TTAA 65071 02313`

The Table 2 shows a typical coded identification data for a fixed land station. The observation was done at 07:00Z on the 15th day of the month from block 02, station 313, with winds measured in knots.

Coded messages are designed to allow transmission of a large amount of data using only a small number of characters. The numerically coded data allows the report to be decoded by a weather person in any country, regardless of the language spoken. More importantly, this numerically coded format can be readily transmitted by computers. These codes may be easily loaded into computer programs that analyze the upper-air data, plot graphical displays, and then calculate probable changes in the reported conditions. The resulting information serves as an invaluable forecast aid [35].

4.2. Message generation process

The message generator framework is capable of producing coded messages; but it requires the code forms to be converted into an XML *message configuration file*. The translation from code forms into XML is not a trivial process. It requires a good understanding of the message generator framework and the developer must be familiar with the WMO code forms and specific terminology used in the meteorology domain. An intermediate step is the creation of *message configuration diagrams*. These diagrams use the message generator framework tree structure to represent a message code form visually. The diagrams then can be converted by a developer to XML message configuration files. Figure 9 shows the message diagrams for TEMP A Section 1.

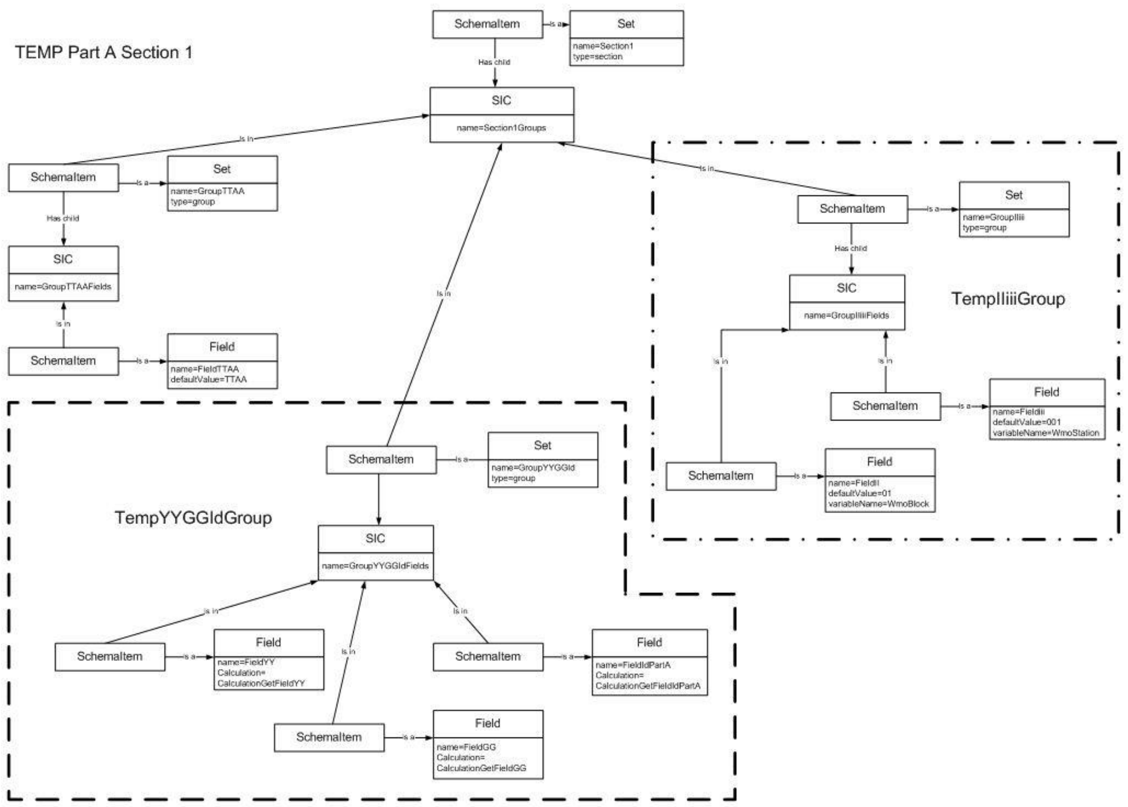


Figure 9 TEMP A section 1 message diagram.

The figure above shows the message diagram of the Section 1 of TEMP A code form. This diagram relates code form items to *SchemaItem*, *SchemaItemCollection*, and finally to *Field* elements in the message generator framework. A *Field* element contains a property *Calculation* which at the end of the chain will map to method that will return the real value of the code form. A further explanation of the elements of the message generator framework will be given later in this thesis.

From a message diagram it is possible to create the XML message configuration file, which is in essence the XML representation of the diagram.

Listing 7 XML configuration for TEMP A section 1YYGG Group

```
<SchemaItemsCollection>
 <Row SchemaNamePk="TEMP Part A Schema" CollectionNamePk="GroupYYGGIdFields"/>
</SchemaItemsCollection>

<SchemaItems>
 <Row StrategyNameFk="TEMP" SchemaNamePk="TEMP Part A Schema"
  CollectionNamePk="Section1Groups" ItemNbrPk="2"
  ChildCollectionNameFk="GroupYYGGIdFields" SetNameFk="GroupYYGGId" />
 <Row StrategyNameFk="TEMP" SchemaNamePk="TEMP Part A Schema"
  CollectionNamePk="GroupYYGGIdFields" ItemNbrPk="1" SetNameFk="GroupYYGGId"
  FieldNameFk="FieldYY" />
 <Row StrategyNameFk="TEMP" SchemaNamePk="TEMP Part A Schema"
  CollectionNamePk="GroupYYGGIdFields" ItemNbrPk="2" SetNameFk="GroupYYGGId"
  FieldNameFk="FieldGG" />
 <Row StrategyNameFk="TEMP" SchemaNamePk="TEMP Part A Schema"
  CollectionNamePk="GroupYYGGIdFields" ItemNbrPk="3" SetNameFk="GroupYYGGId"
  FieldNameFk="FieldIdPartA" />
</SchemaItems>
```

```
<Sets>
 <Row StrategyNamePk="TEMP" SetNamePk="GroupYYGGId" Type="group"
  XmlNodeFk="GroupNode" />
</Sets>

<Fields>
 <Row StrategyNamePk="TEMP" SetNamePk="GroupYYGGId" FieldNamePk="FieldYY"
  Description="Day of month" CalculationNameFk="CalculationGetFieldYY"
  DefaultValue="" XmlNodeFk="FieldNode" />
 <Row StrategyNamePk="TEMP" SetNamePk="GroupYYGGId" FieldNamePk="FieldGG"
  CalculationNameFk="CalculationGetFieldGG" DefaultValue="" XmlNodeFk="FieldNode" />
</Fields>
```

The resulting XML configuration files for the entire TEMP A message (including the SHIP and MOBIL variants) is close to 1500 lines of code. Generating this file by hand is a tedious task which is prone to errors. The constant repetition of similar elements in the file and the verbosity of XML language incite the developer to use copy and paste, this is the source of bugs that are not easily identified. The need for a tool to automate the process of generating the configuration file out of the message diagrams is evident. The use of such tool will:

- Maintain message diagrams and their configuration file in consistent state.
- Simplify the process of generating and editing XML configuration files.
- Eliminate redundant work when modifying a message configuration.
- Reduce the amount of errors.

The message generator framework can read the message configuration directly from the XML files or from a database. When the database is used an extra step to export the XML files to database tables is needed. Once the message generator has loaded a message configuration and a source of data has been properly set up, it can be commanded to code such message.

4.2.1. Message generator framework elements

The message generator framework elements refer to tables in the database or nodes in the XML configuration file. The Figure 10 shows the relation of these elements.

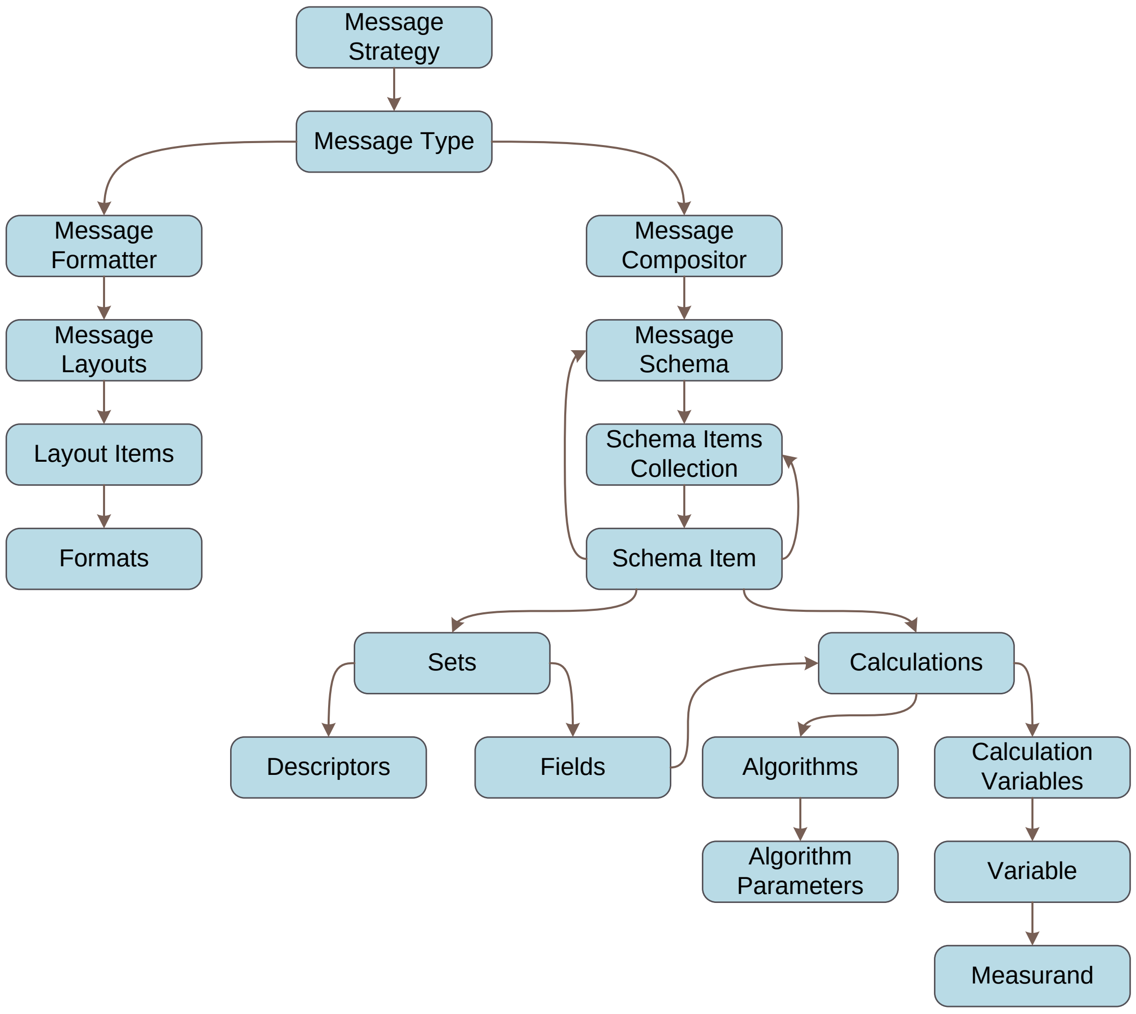


Figure 10 Relation between message generator framework elements.

The root element of a message is the *message strategy* which can contain one or more *message types*. For example TEMP is a message strategy and the different parts (A, B, C and D) and variants (SHIP, MOBIL) are message types. Each *message type* requires a *message formatter* and a *message compositor*. The *message formatter* requires a *message layout,* which contains a collection of *layout items* each of them defining the *format* of individual items. A *message compositor* defines a *message schema* with a set of *schema item collections*, *schema items*, *sets*, *descriptors***,** *fields* and *xml nodes* in order to describe the message structure in a tree format [36].

MessageSchema defines the tree structure of a WMO message code form. It comprises one or more SchemaItemCollections which are containers for other elements. A SchemaItemCollection is always associated with a SchemaItem and a Set. The latter one indicates what kind of container the collection is, this information can be used to format specific types differently when the final message output is produced.

A SchemaItem is a node in the MessageSchema tree. SchemaItems are associated with Sets, Descriptors or Fields depending of the function they have. Table 3 shows the characteristics of the possible uses of a SchemaItem:

Table 3 SchemaItem uses

SchemaItem use	Description
Set	A set can be of type heading, section, group, data subset, sequence, replication or line. This type is used by the *formatter* to support custom formatting of different types of sets
Descriptor	A descriptor is used to associate meta-data to provide extra information for sets or fields. This meta-data is used in binary format messages such as BUFR (Binary Universal Form for the Representation of meteorological data).
Field	A field that represents a leaf node, which is a real entry in the final message. The output value of the field is obtained by three different forms in this order of precedence: calculation, the value is calculated; variable, the value maps to a defined variable; default, a given default value.

The appendix B describes the use of all the elements of the message generator framework.

4.3. Inconsistencies in message configuration process

The process used in practice at the moment of this thesis writing to convert a code form into a configuration file requires three phases. First to convert the code form into a message diagram; second to convert the message diagram into an XML configuration file and third export the configuration file to the data base. Two different artifacts are produced along the process: message diagrams created in Microsoft Viso and XML configuration files. In the current practice these artifacts are maintained independently, nothing enforces the relation and dependencies of these artifacts. If a change is made in the message diagram the XML configuration file needs to be updated manually and vice versa, if not the two artifacts become inconsistent.

The experience shows that XML configuration files are updated when changes are required due to bug fixing or when new requirements appear e.g. custom section added to the message or customized layout required; but the message diagrams tend to become obsolete. This problem does not exist between database and XML configuration files since a tool that export the XML configuration files to the database that can be used every time the configuration changes.

Obsolete and inconsistent diagrams are useless for the developer and cannot be used as a documentation artifact because they do not reflect the current implementation state. The effort needed to detect inconsistencies and fixed is too high compared to the benefit the developer gets while implementing small modification to the message configuration.

However every time the XML configuration file is modified and the message diagrams are not updated the software erosion of the system increases. At the moment that this thesis was written the message diagrams for the 4 message strategies configured in the message generator framework are in a state of inconsistency with their XML configuration.

XML configuration files as mention in section 2.2.3 can be validated to check their consistency against their XML schemas. This is other source of inconsistencies in the process. Without a tool support to validate the consistency after each change made in the XML configuration file, maintaining its consistency is difficult. For example if the name of a SchemaItem is changed the elements that reference it need to be updated properly.

5. Message configuration editor (MENE)

It is common knowledge that support tools play a critical role in the software engineering process by improving quality and productivity. A huge number a support tools have been produced to assist tasks in software development processes. Most software developer teams use tools that are assembled over time and adopt a new tool when the use of it brings them a benefit [37]. The inconsistency problems presented in the section 4.3 can be resolved by the use a support tool.

MENE (MEssage coNfiguration Editor) is a functional support tool that assists the software developers in the process of create and edit XML message configuration files used by a specific meteorological message generation framework. MENE uses an adaptation of specialization patterns to validate an XML configuration file and provide the tool with a list of tasks needed to make the XML configuration file consistent with its XML schema.

By using MENE a software developer can maintain XML configuration files valid and consistent with their message diagrams.

5.1. Architecture and design

MENE is implemented using Adobe Flex 4.5.1 SDK. This technology was chosen following an internal policy of the company for whom the tool is developed. MENE follows a modular architecture consisting of three basic components: specialization patterns, message element models and user interface. Specialization pattern component is an adaptation of the specialization pattern presented section 3.2. Message element model component contains data models to store message element configurations. User interface component uses both the specialization patterns component and the data models to present the user a message configuration and guide it in the editing process. Figure 11 depicts how these three components relate.

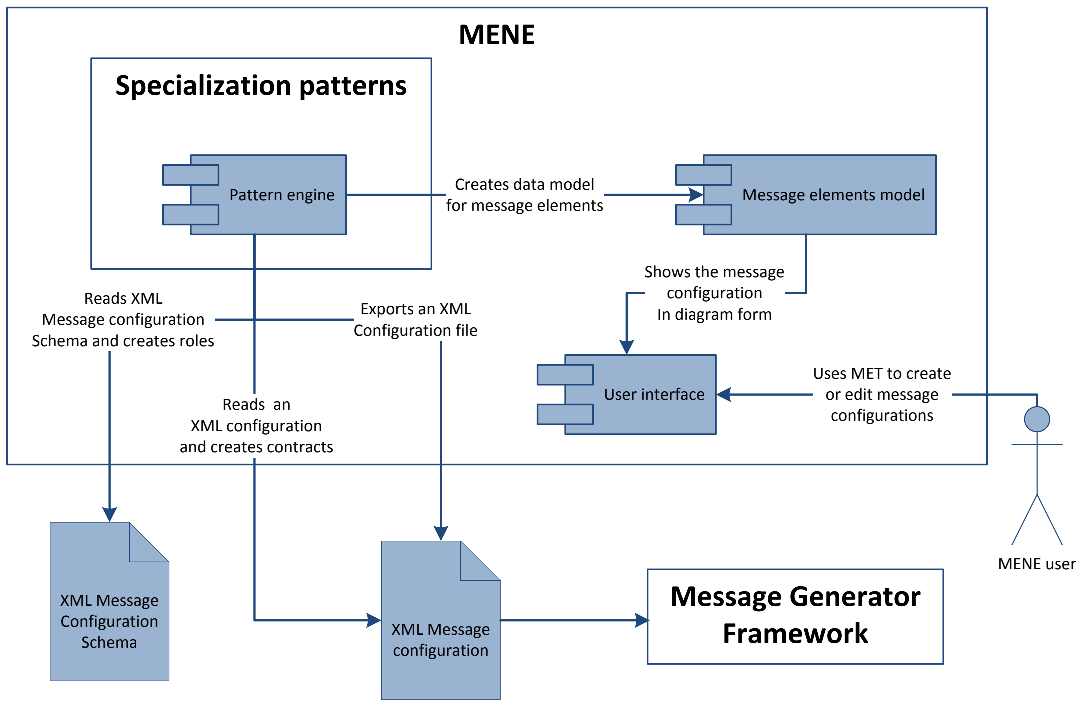

MENE
Specialization patterns
Pattern engine
Creates data model for message elements
Message elements model
Shows the message configuration In diagram form
Reads XML Message configuration Schema and creates roles
Exports an XML Configuration file
User interface
Uses MET to create or edit message configurations
Reads an XML configuration and creates contracts
MENE user
XML Message Configuration Schema
XML Message configuration
Message Generator Framework

Figure 11 Overview of message configuration editor.

5.1.1. Example sequence of using MENE

Figure 12 shows an example of editing a message element using MENE. The user starts the user interface (UI) and the XML schema that defines the message configuration is loaded automatically. The UI passes the loaded XML schema to the pattern engine, which will analyze it and extract from it roles and tasks. After this the UI is fully initialized and ready to be used; but no message configuration has been loaded yet. The user then can load an XML configuration file containing one or multiple message configuration. This XML configuration file is passed then to the specialization pattern, which converts it into contracts and task. Now the UI shows the configuration as a message diagram and the user can modify message elements and receive feedback for its actions as the pattern engine validates the consistency of the configuration on the fly. Finally the edited message configuration can be exported to an XML configuration file or to an image file.

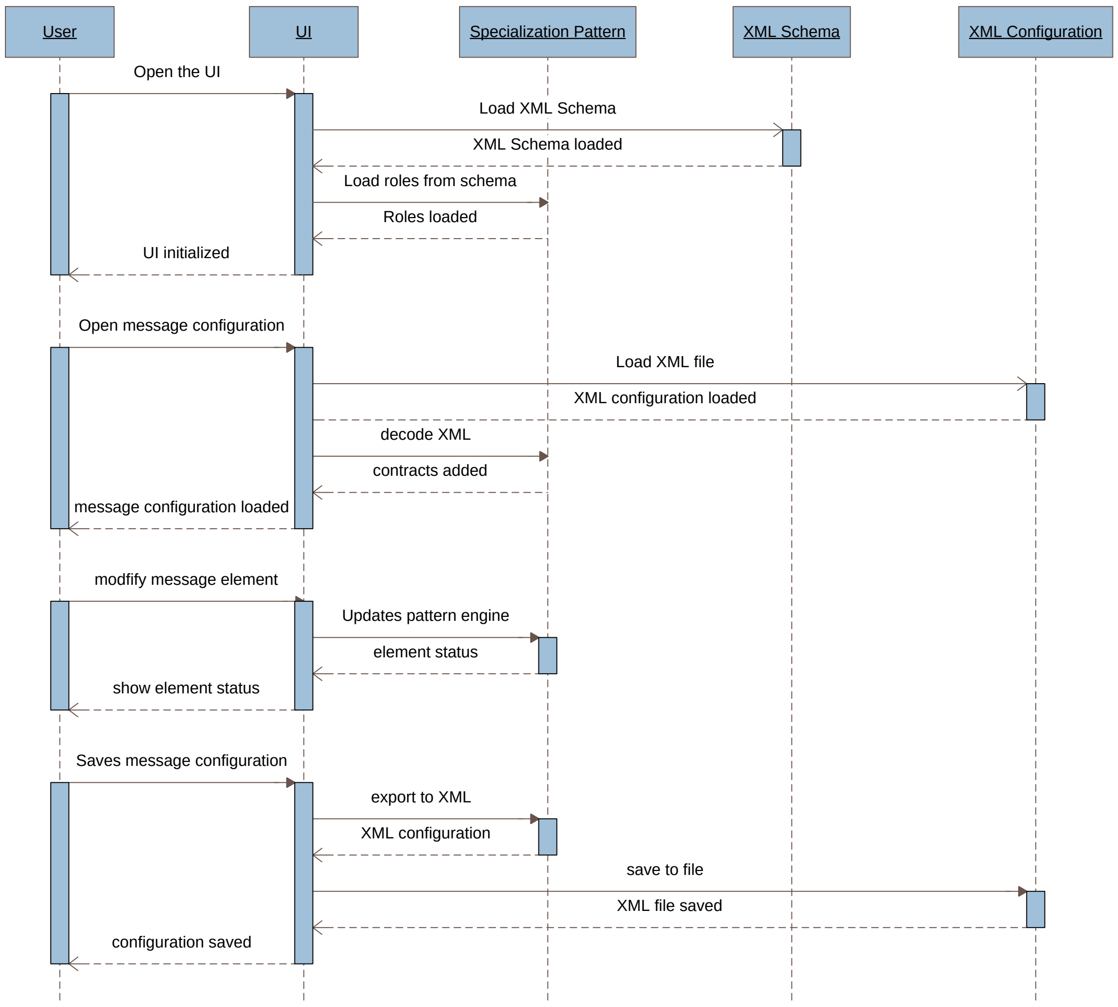


Figure 12 MENE sequence diagram.

5.1.2. Pattern engine

Pattern engine implements an adapted version of specialization patterns presented section 3.2 which is used to validate elements of the configuration files and detect when these become inconsistent. Pattern engine provides support to convert XML schema files and XML configuration files into specialization pattern items: roles, contracts and tasks and these elements back to XML configuration files.

The pattern engine component is the core of the MENE. It is implemented as a singleton that exposes a collection of all the *roles* supported by the pattern engine and other collection for all the *contracts* instances existing in the engine. The status of each contract is available at any time and it indicates the consistency state of the element that it represents. The Figure 13 shows the relation of pattern engine and the specialization pattern elements.

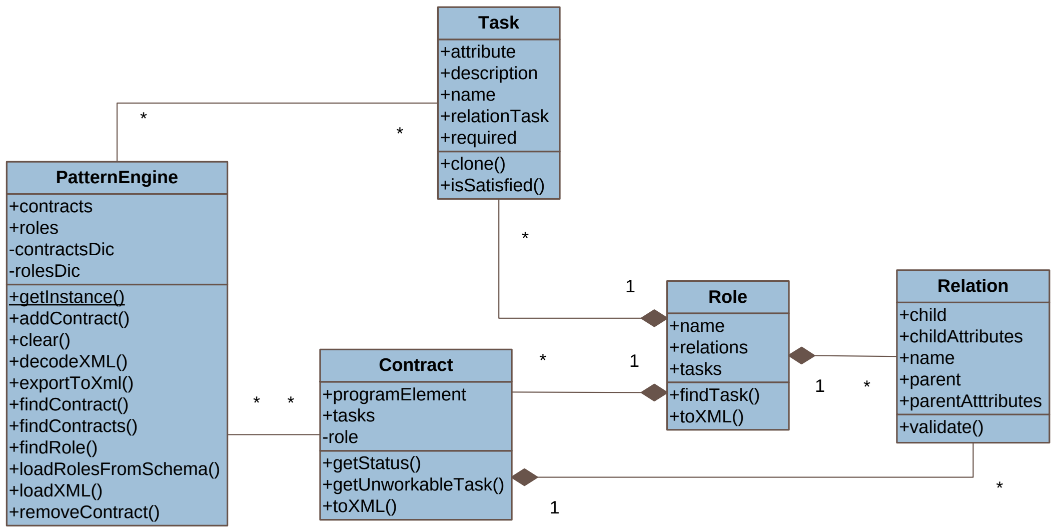


Figure 13 Specialization pattern simplified class diagram.

5.1.2.1. Roles

A specialization pattern *role* maps to an element of the XML configuration schema. A *role* includes a list of tasks and relations that a *contract* inherits when it is instantiated. Each attribute of the XML element creates an entry in the task list. If the XML element has constraints defined, the *refKey* attribute of the constraint also creates an entry in the task list and an entry in the relation list. The use of tasks and relation is explained in section 5.1.2.3.

Listing 8 MessageStrategies XML schema definition.

```
<xs:element name="MessageStrategies">
 <xs:complexType>
  <xs:sequence>
   <xs:element minOccurs="0" maxOccurs="unbounded" name="Row">
    <xs:complexType>
     <xs:attribute name="StrategyNamePk" type="EntityName" use="required" />
     <xs:attribute name="Description" type="Description" use="required" />
     <xs:attribute name="GridStorageTypeNameFk" type="EntityName" use="required" />
     <xs:attribute name="Priority" type="xs:integer" use="required" />
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:key name="IdxMessageStrategiesPk">
 <xs:selector xpath=".//MessageStrategies/Row" />
 <xs:field xpath="@StrategyNamePk" />
</xs:key>
 <xs:keyref name="IdxMessageStrategies1" refer="IdxDynamicTypesPk">
 <xs:selector xpath=".//MessageStrategies/Row" />
 <xs:field xpath="@StrategyTypeNameFk" />
</xs:keyref>
 <xs:keyref name="IdxMessageStrategies2" refer="IdxDynamicTypesPk">
 <xs:selector xpath=".//MessageStrategies/Row" />
 <xs:field xpath="@GridStorageTypeNameFk" />
</xs:keyref>
```

For instance the XML definition given above will produce a *role* named MessageStrategies that contains two relations, one for each *keyref* constraint, and six tasks, one for each required attribute plus the two relation tasks.

5.1.2.2. Contracts

A *contract* can be seen as an instance of a *role*. It maps to a row of the role in the XML configuration file. A contract stores the piece of code that it represents in a property called *programElement*. Contracts can be loaded from an XML configuration file or added manually to the engine pattern. Since a *contract* represents a piece of code it can be at any time converted to its XML representation.

Contracts inherit from their parent role the list of task to be done in order to consider a contract *done*. Contracts contain a dynamic list of *unworkable task*. This list reports all the required tasks of the contract that are not satisfied at a specific time.

Listing 9 PILOT message strategy definition.

```
<MessageStrategies>
 <Row StrategyNamePk="PILOT"
      Description="Strategy for PILOT"
      GridStorageTypeNameFk="WmoGridStorage"
      Priority="2"/>
</MessageStrategies>
```

The snippet of XML above creates a *contract* of role type MessageStrategies. The list of unworkable task is populated based on the existence of other contracts in the pattern engine. If a contract of type DynamicTypes that satisfies the relation task of this contract exists, then all the tasks are *done* and the unworkable task list is empty. Otherwise an entry for this task is reported in the list and the status of the contract is *unworkable.*

5.1.2.3. Tasks and relations

A *task* represents a constraint that a contract needs to satisfy. Every attribute defined in the XML schema definition represents a task. Tasks can be required or optional. The attributes marked as required are marked as required tasks.

Keys and reference keys in the XML schema are also required tasks. The tasks for reference keys include a *relation* object. A *relation* is an object that defines how two different roles are related.

Tasks can be either satisfied or incomplete depending of the contracts in the pattern engine. For example a task is satisfied if a contract *programElement* defines a value for the attribute and this attribute has a valid relation to other contract *programElement* attribute.

5.1.3. Message element models

Message element model component contains classes that encapsulate different elements of a message code form. These models can be used in the user interface views as sources of data for bindings.

Message element models provide a status property that combines the statuses of all the contracts that form part of the model. A list of unworkable task is populated with the tasks that are incomplete when the general status of the message elements is *unworkable*.

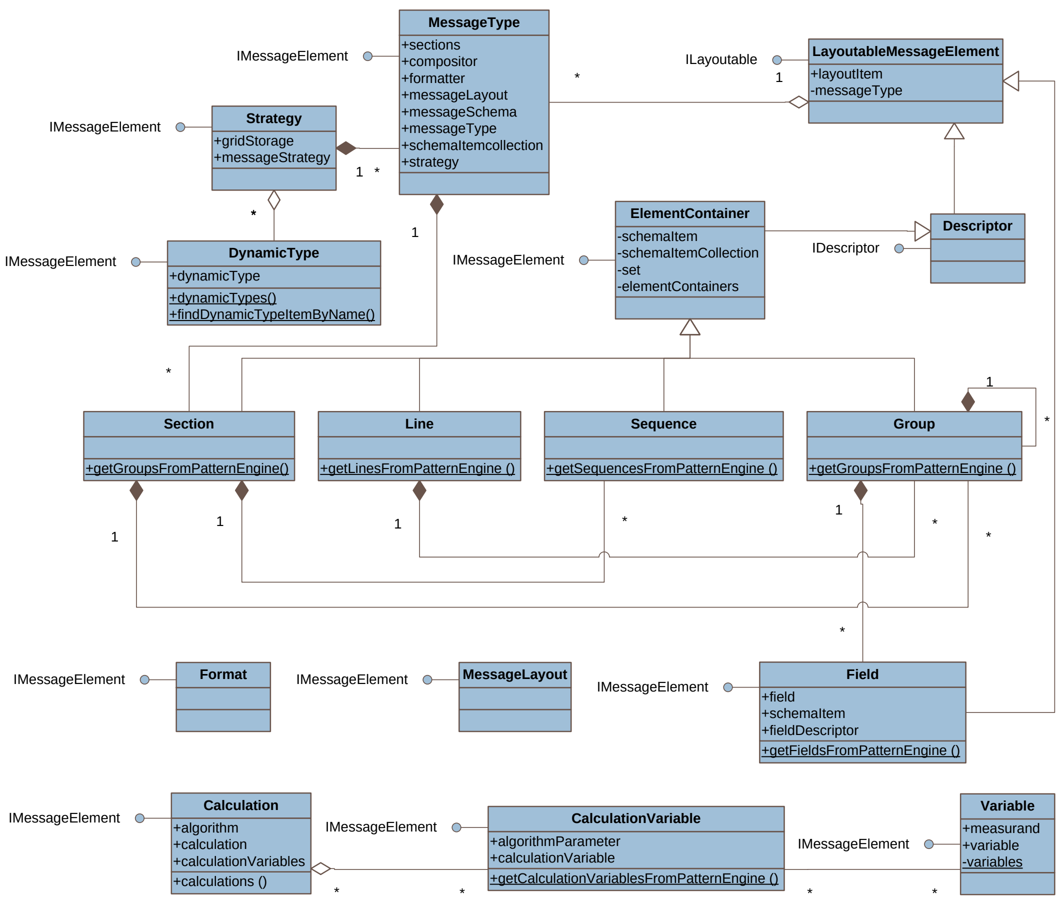


Figure 14 Simplified class diagram of message element models.

The figure above shows a simplified class diagram of the message element models. IMessageElement is an interface that all elements that can be edited with MENE need to implement since it defines the property status that can be used in the user interface view to mark an element as valid or invalid. The relations between these model classes allow to move along related elements e.g. from a Field model it is possible to reach the message type to which the Field belong. This enables the propagation of changes done in shared elements for example if the name of a MessageTypes changes all the elements that have a relation with the MessageType will detect the change and update their properties accordingly.

5.1.4. Graphical user interface

The graphical user interface allows the manipulation and edition of message elements in a visual way. The user interface is developed on top of a third party library called Kalileo and editing forms created using Flex SDK. Kalileo® diagrammer component displays data as a graph to better visualize connections and allow the manipulation of the graph providing a good editing user experience [38]. The use of our own diagrammer library could be possible in the future, and MENE can be adapted to use it, however the development of such library is consider out of the scope of this work.

5.1.4.1. Diagrammer

Diagrammer is a class that inherits from Kalileo® diagrammer component. Its main function is to represent a message type in a tree-shaped-graph format. Diagrammer has a method *drawMessageDiagram* that gets as a parameter a MessageType instance, which is the root node of the tree. Diagrammer then adds a node for the MessageType elements and iterates recursively though the children of it and adds each of them as nodes. Each node will use a different renderer to have a custom appearance on the diagram.

Diagrammer listens and handles two types of ActionScript events: *add message element* and *elements deleted*. These events are used to communicate the diagrammer that a node needs to be added or removed from the tree.

5.1.4.2. Forms

Forms package contains a collection of controls that provide and interface to edit message elements. There is a form for each message element. Each form has a property that references a message element model. The values of the form controls: text fields, drop-down lists, etc., use two-way binding to the message element model item of the form. This means that a change in the item is reflected in the form and vice versa without need of manually saving such a change.

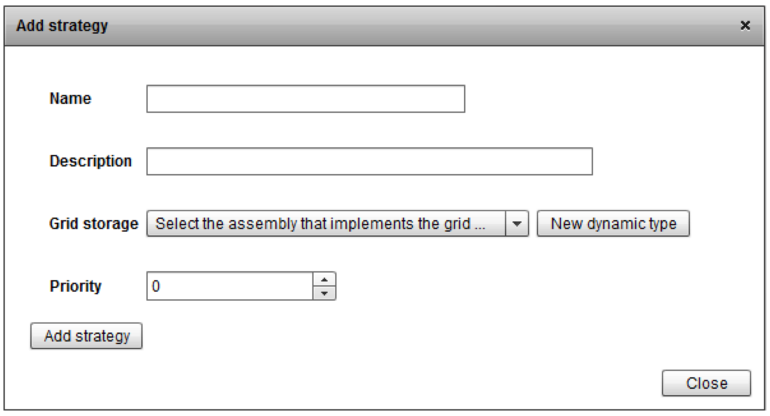


Figure 15 Message strategy edit form.

The form shown above allows the user to edit all the relevant values of a message strategy element.

5.1.4.3. Renderers

A renderer is a visual element that holds a data property and can be used to visually represent or show the data associated with it. In case of MENE all the renderers have a rectangular shape, which contains different labels where the values of node attributes are displayed. The color of the renderer border is used to show the status of the message elements it represents. If the message element status is *unworkable* the border turns red. If the diagram contains at least one node in unworkable state, this means that the XML configuration file is not consistent with the XML schema. Figure 16 shows a Field node in both states.

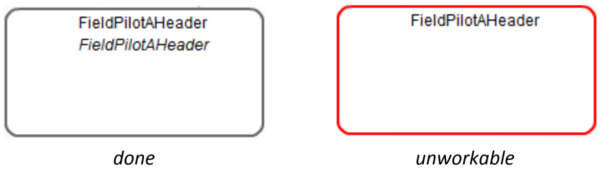


Figure 16 Field renderer displaying the message element status.

The figure above shows a contract in both states: done and unworkable. A description is required for this contract and therefore the contract that does not define a description is marked as unworkable and its renderer has a red color border.

Kalileo® provides several methodologies and tools to fully control what is rendered and how it behaves in the diagram. One of these is the use of *itemRendererFunction* [39]. When a new node is added to the diagrammer the function *itemRendererFunction* is called and inside this a specific renderer for the node is defined. Any object that implements *fr.kapit.visualizer.IRenderer* or *mx.core.IDataRenderer* can be used as a node renderer. MENE uses *mx.core.IDataRenderer* because they are part of Flex SDK and can be reused in case that our own diagrammer is used instead of Kalileo®.

The use of renderers allows customizing how data is shown, this is an important feature because gives the possibility to adapt the user interface easily. MENE defines the following renderers: *FieldRenderer*, *GroupRenderer*, *LineRenderer*, *SectionRenderer*, *SequenceRenderer*, *MessageTypeRenderer*.

5.2. WMO meteorological message structure

Message generator framework (see Chapter 4) supports complex message tree structures. This gives to the developer the freedom to convert a WMO message code specification into XML message configuration in several different but equivalent and valid forms. This freedom does not give any advantage in the process of converting the specification into XML code; on the contrary it causes that configuration for similar messages written by different developers do not follow the same structure. In order to mitigate this MENE enforces the use of the common structure shown in the Figure 17.

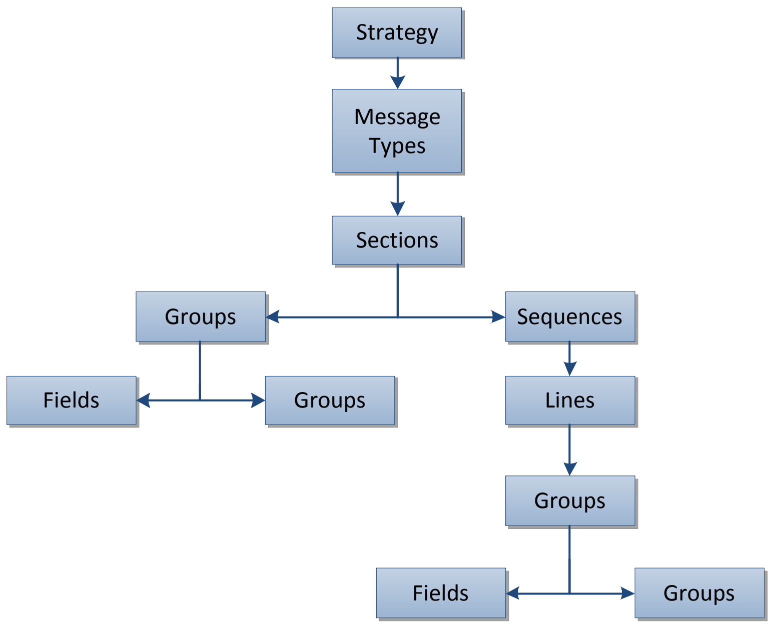


Figure 17 General WMO coded message structure.

The figure above presents a typical WMO message structure. Each node of the structure tree represents a message element. A strategy comprises a set of related WMO message codes e.g. TEMP is a strategies that groups all WMO message codes for TEMP. A message type is a specific WMO code message e.g. FM 32-XI Ext. PILOT. Every message is composed of sections which are element containers. Group is a basic element container that can have only fields or other groups. A sequence is a special type of container used to group data that repeats in the message body. A line contains the data that can be repeated in the message.

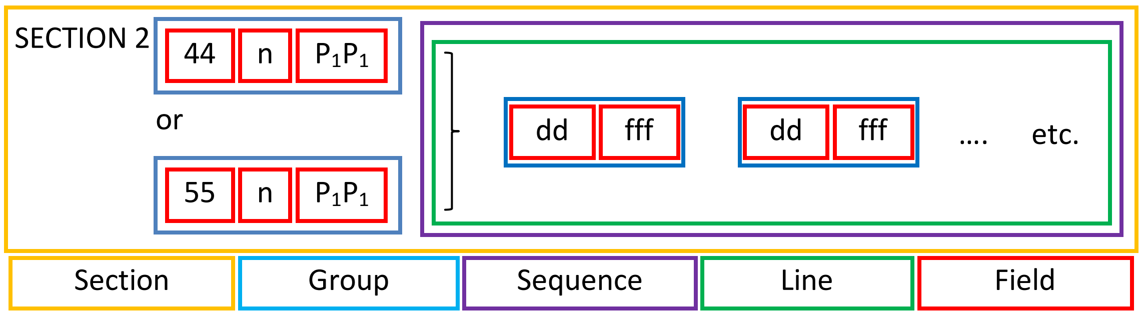
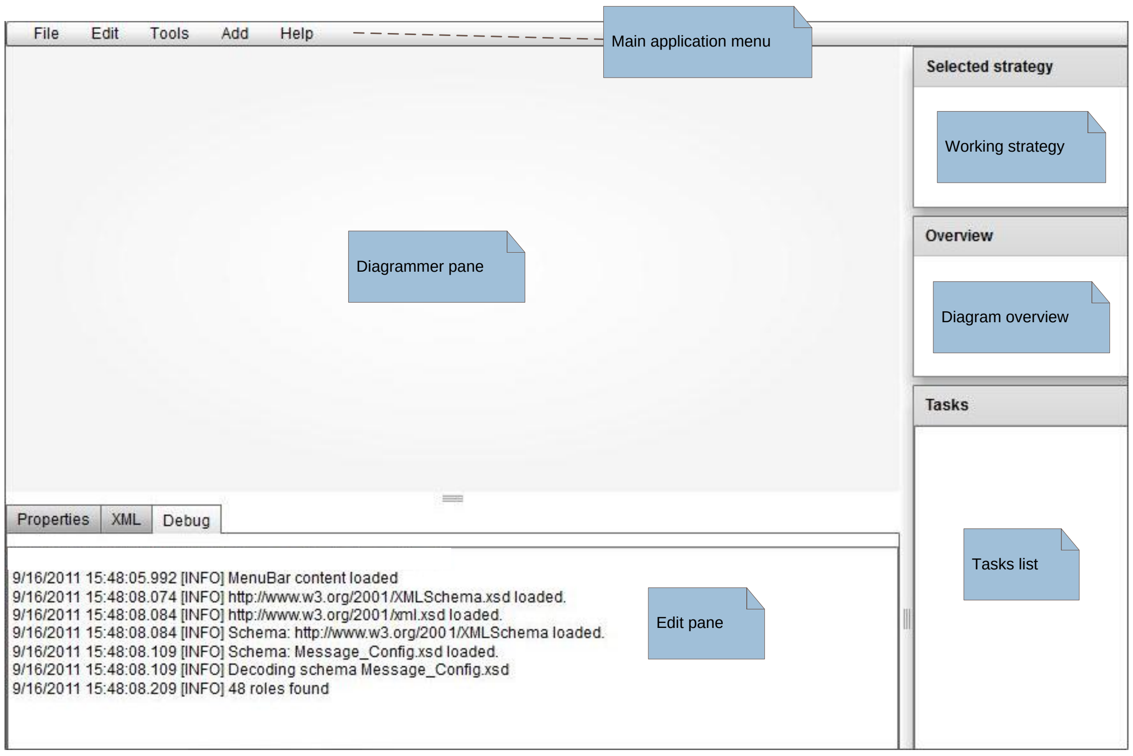


Figure 18 PILOT Section 2 code form.

The previous figure shows how the elements of a code form match with the defined message structure.

6. Using MENE

This chapter introduces the use of MENE. In this chapter it is explained how MENE can be used to create new message configurations or edit configuration created with it. The chapter focuses on the user interface. Figure 19 shows the user interface of MENE. It consists of 6 main areas: main application menu, diagrammer pane, edit pane, working strategy pane, overview pane and tasks list pane. Edit pane has 3 tabs: Properties, XML and Debug. The first one shows a form to edit a selected node in the diagrammer pane; the second one displays a preview of the XML output for the selected node; the last tab shows the runtime logs.



```
9/16/2011 15:48:05.992 [INFO] MenuBar content loaded
9/16/2011 15:48:08.074 [INFO] http://www.w3.org/2001/XMLSchema.xsd loaded.
9/16/2011 15:48:08.084 [INFO] http://www.w3.org/2001/xml.xsd loaded.
9/16/2011 15:48:08.084 [INFO] Schema: http://www.w3.org/2001/XMLSchema loaded.
9/16/2011 15:48:08.109 [INFO] Schema: Message_Config.xsd loaded.
9/16/2011 15:48:08.109 [INFO] Decoding schema Message_Config.xsd
9/16/2011 15:48:08.209 [INFO] 48 roles found
```

Figure 19 MENE user interface starting point.

MENE is a Flash® application and it can be executed locally in Flash® Player or deployed to a web server. In both cases MENE loads the XML message configuration schema from the same location from where the application is executed. Once the schema is loaded, the roles defined on it are added to the pattern engine and MENE is ready to be used.

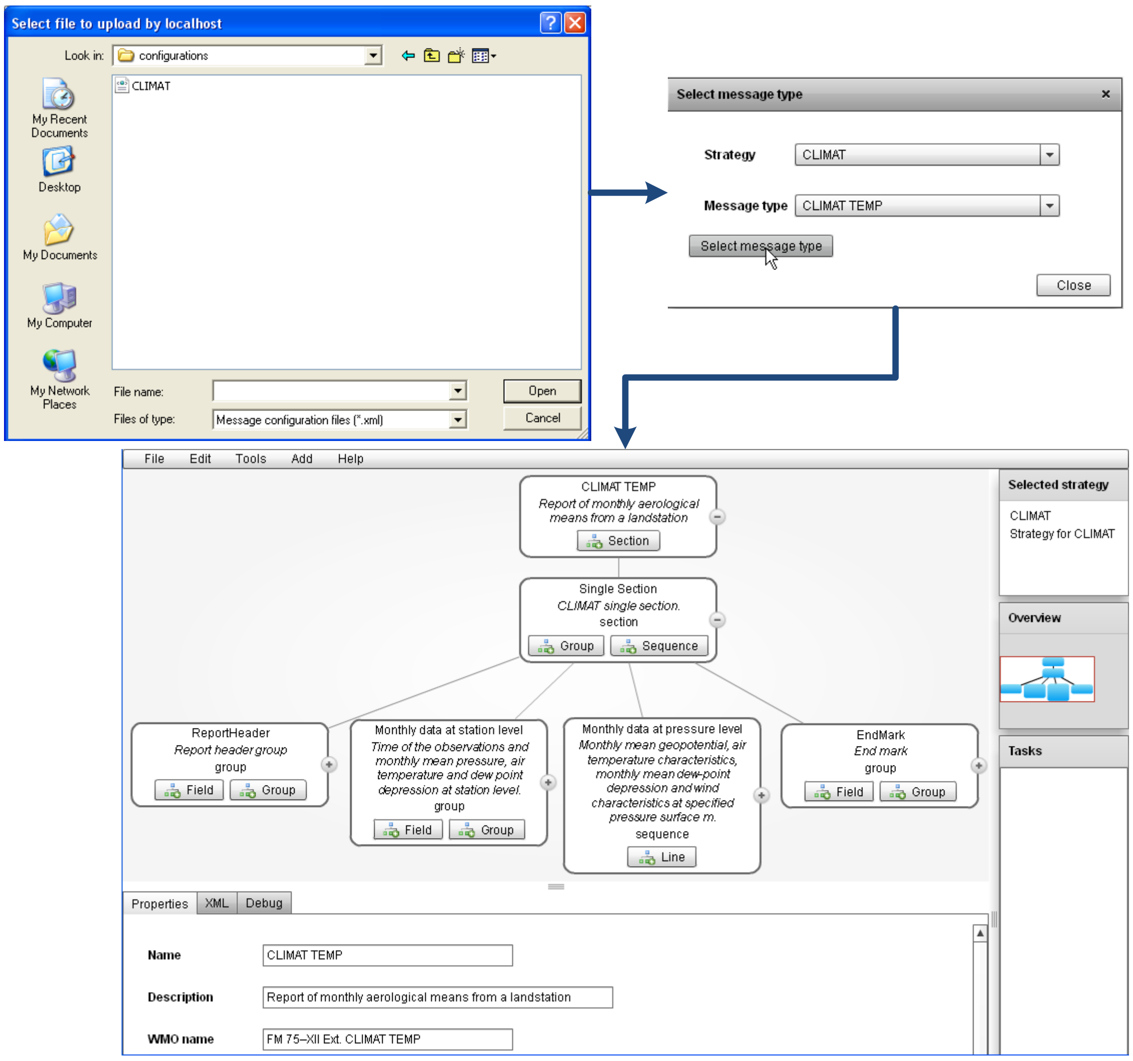
The first step to use MENE is to load a full XML configuration file or a set of common elements—appendix C provides a basic set of common elements. In both cases

MENE adds the contracts defined in the loaded XML configuration file to the pattern engine. The next step is to interact with the user interface to add, remove or edit nodes. Finally the newly edited configuration file can be saved as an XML configuration file or a PNG image.

6.1. Editing an existing message configuration

In order to edit a message configuration MENE the configuration needs to be loaded into the pattern engine. The menu File → Load file… prompts for the location and file name to load. Once the file is loaded its contents are converted into contracts and added to the pattern engine.

MENE shows always a MessageType element as root of a message configuration tree diagram. A configuration file can include several message types but MENE shows only one at the time in order to keep the diagrammer pane simple. The message type to be shown can be selected by using the menu Edit → Select message type.


Select file to upload by localhost
My Recent Documents
Desktop
My Documents
My Computer
My Network Places
CLIMAT
Open
Cancel
Select message type
Select message type
Close
File
Edit
Tools
Add
Help
CLIMAT TEMP
Report of monthly aerological means from a landstation
Section
Single Section
CLIMAT single section.
section
Group
Sequence
ReportHeader
Report header group
group
Field
Group
Monthly data at station level
Time of the observations and monthly mean pressure, air temperature and dew point depression at station level.
group
Field
Group
Monthly data at pressure level
Monthly mean geopotential, air temperature characteristics, monthly mean dew-point depression and wind characteristics at specified pressure surface m.
sequence
Line
EndMark
End mark
group
Field
Group
Selected strategy
CLIMAT
Strategy for CLIMAT
Overview
Tasks
Properties
XML
Debug

Look in: configurations

File name:

Files of type: Message configuration files (*.xml)

Strategy: CLIMAT

Message type: CLIMAT TEMP

Name: CLIMAT TEMP

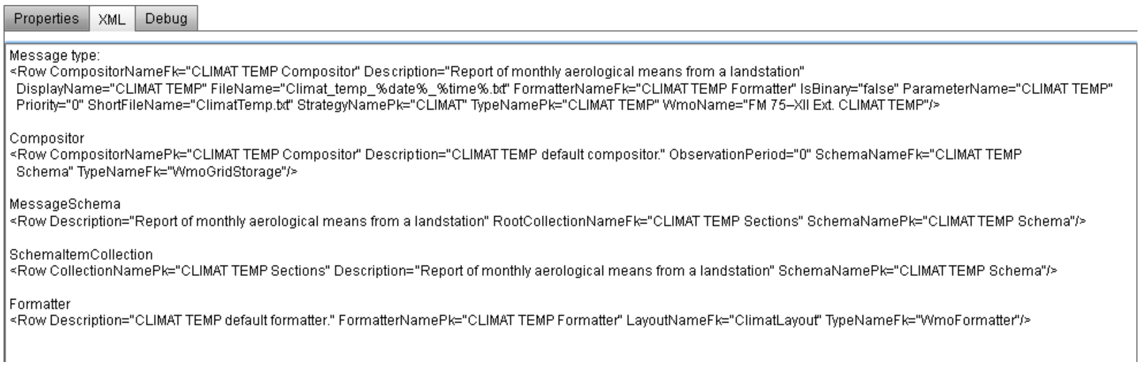
Description: Report of monthly aerological means from a landstation

WMO name: FM 75–XII Ext. CLIMAT TEMP

Figure 20 Steps to edit a message configuration.

Figure 20 shows the steps to load a configuration file and select a message type to edit. Once these steps have been done a node can be edited by clicking on it to select it and its properties will be shown in the properties tab of the edit panel.

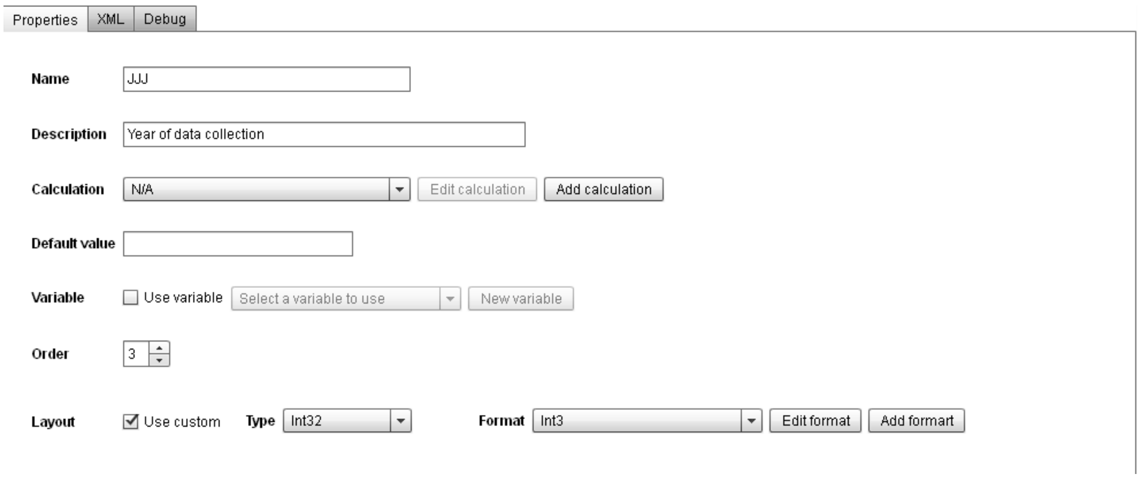
The tab XML of the edit panel shows the XML representation of a selected node. This allows the developer to check in real time the code that will be generated in the exported configuration file for that specific element.



```
Message type:
<Row CompositorNameFk="CLIMAT TEMP Compositor" Description="Report of monthly aerological means from a landstation"
 DisplayName="CLIMAT TEMP" FileName="Climat_temp_%date%_%time%.txt" FormatterNameFk="CLIMAT TEMP Formatter" IsBinary="false" ParameterName="CLIMAT TEMP"
 Priority="0" ShortFileName="ClimatTemp.txt" StrategyNamePk="CLIMAT" TypeNamePk="CLIMAT TEMP" WmoName="FM 75–XII Ext. CLIMAT TEMP"/>

Compositor
<Row CompositorNamePk="CLIMAT TEMP Compositor" Description="CLIMAT TEMP default compositor." ObservationPeriod="0" SchemaNameFk="CLIMAT TEMP
 Schema" TypeNameFk="WmoGridStorage"/>

MessageSchema
<Row Description="Report of monthly aerological means from a landstation" RootCollectionNameFk="CLIMAT TEMP Sections" SchemaNamePk="CLIMAT TEMP Schema"/>

SchemaItemCollection
<Row CollectionNamePk="CLIMAT TEMP Sections" Description="Report of monthly aerological means from a landstation" SchemaNamePk="CLIMAT TEMP Schema"/>

Formatter
<Row Description="CLIMAT TEMP default formatter." FormatterNamePk="CLIMAT TEMP Formatter" LayoutNameFk="ClimatLayout" TypeNameFk="WmoFormatter"/>
```

Figure 21 MENE XML inspection panel.

The properties tab shows the message element properties that can be edited. Figure 22 shows the properties panel of the field node JJJ. By inspecting this panel it is possible to see all the properties of the field and modified them.



Name	JJJ
Description	Year of data collection
Calculation	N/A
Default value	
Variable	☐ Use variable
Order	3
Layout	☑ Use custom · Type Int32 · Format Int3

Figure 22 MENE properties panel.

In order to edit a message element it needs to be selected, by clicking on it. If the change causes that the message elements becomes inconsistent with the message configuration schema, the message element node border turns red. When a node is red the list of task on the task panel shows the fields of the message element that need to be modified to make the node consistent with the schema. Figure 23 shows the activity diagram to edit a message element.

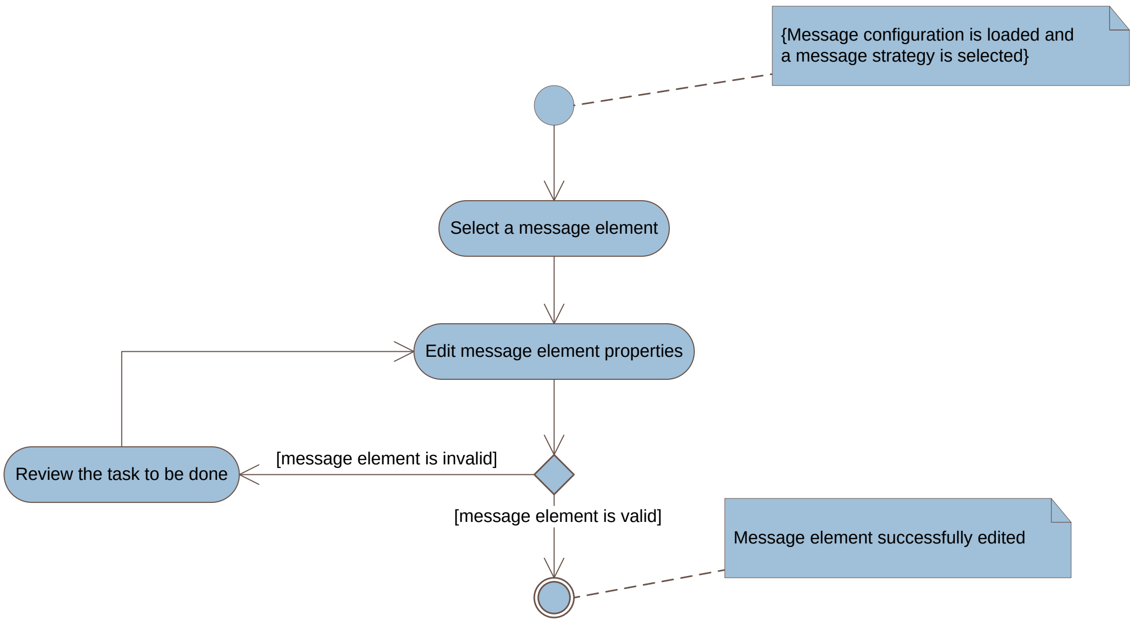
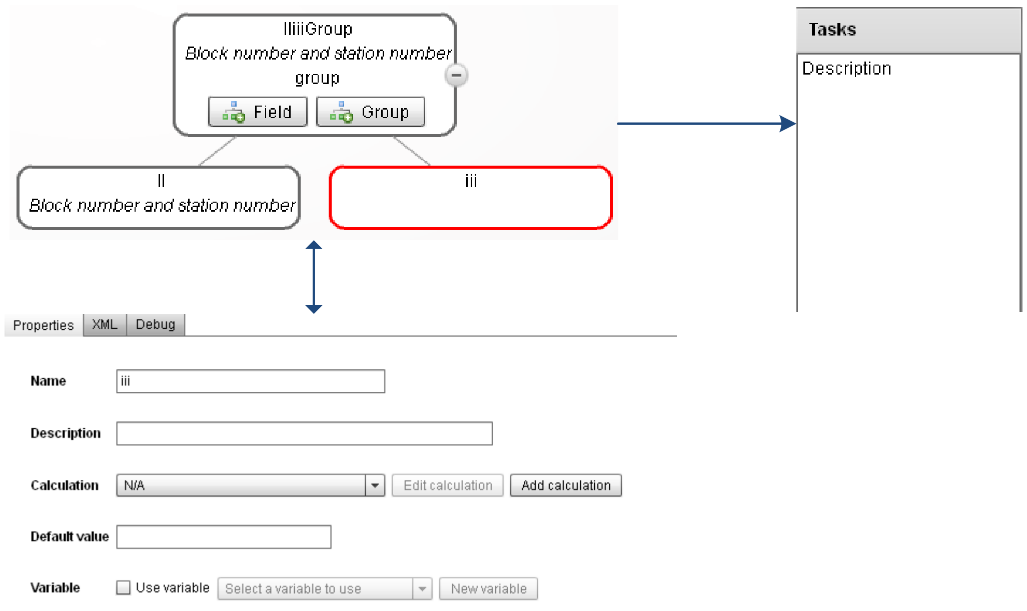

{Message configuration is loaded and a message strategy is selected}
Select a message element
Edit message element properties
Review the task to be done
[message element is invalid]
[message element is valid]
Message element successfully edited

Figure 23 Edit message element activity diagram.

For example a message element field, requires a valid description. In the Figure 24 we can see that the field *Description* in the properties panel is empty, this causes that the field is not consistent with the XML schema and the border of the field node turns red. The task list shows the action needed to fix to the inconsistency.



Name	iii
Description	
Calculation	N/A
Default value	
Variable	☐ Use variable — Select a variable to use

Figure 24 Field node in unworkable state.

6.2. Creating a message configuration

In order to create a new message configuration file first it is needed to define and select the upper level element MessageStrategy. It is a good practice to save one message strategy and all its message types in the same XML configuration file. In this way all the related messages are store in the same file and all the common elements shared among message are accessible. Figure 25 show the steps to follow in order to define a new message type.

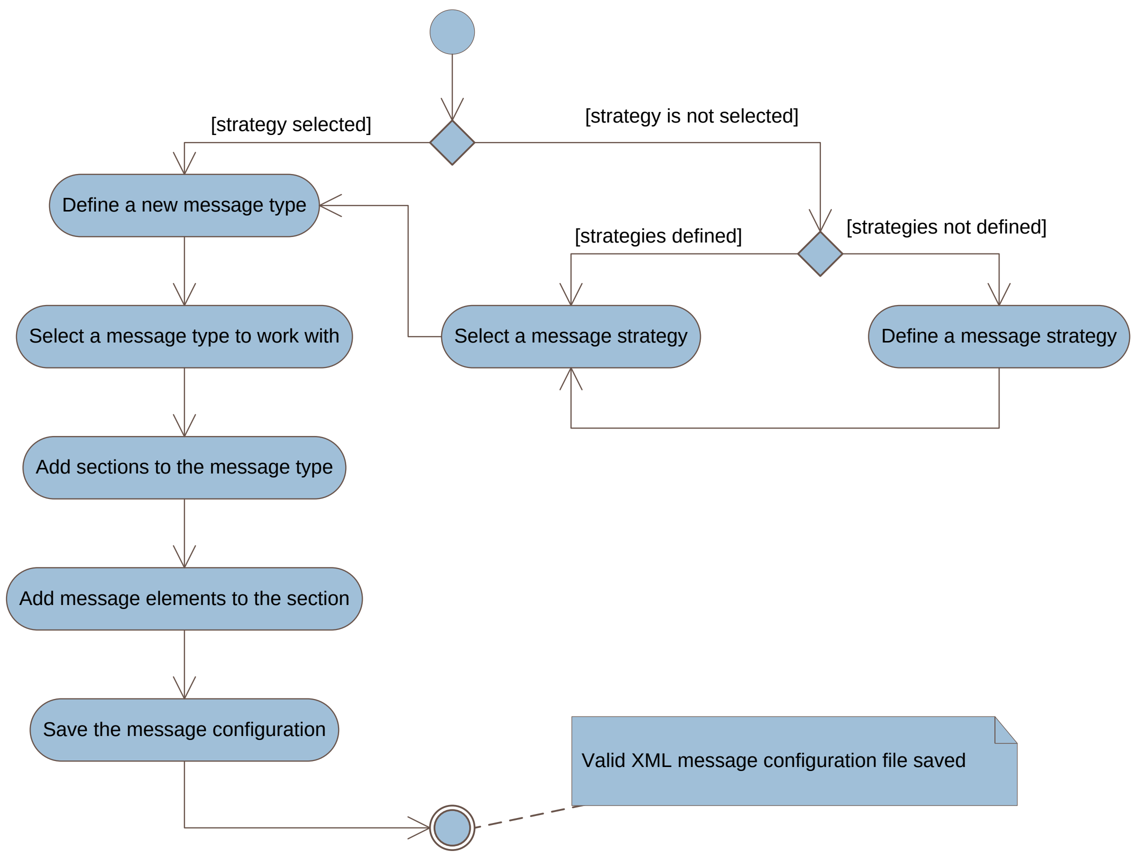

[strategy selected]
[strategy is not selected]
Define a new message type
[strategies defined]
[strategies not defined]
Select a message type to work with
Select a message strategy
Define a message strategy
Add sections to the message type
Add message elements to the section
Save the message configuration
Valid XML message configuration file saved

Figure 25 Activity diagram depicting the process of creating message configuration.

If a strategy has not been loaded in MENE the first step is to define and select one, because a message type always belongs to a strategy. Once the strategy is selected a new message type can be added. After the message exists in the pattern engine it needs to be selected as the working message as shown in the Figure 20. When an element is selected its properties can be edited. See section 6.1.

6.3. Saving the message configuration

It is possible to save at any time the message configuration file that the pattern engine contains and that is being edited in the tool by clicking the menu File → Save file.... MENE prompts for the file name and location to save it. The current version of the tool does not support auto-save since it is not possible to write to the disk without a user interaction due to a security restriction of Flash® Player.

Alternatively it is possible to save the diagram as PNG format image. In this way MENE represents a source for two different artifacts: message diagrams and XML configuration files. The artifacts, diagrams and code, produced using MENE are by nature consistent among them.

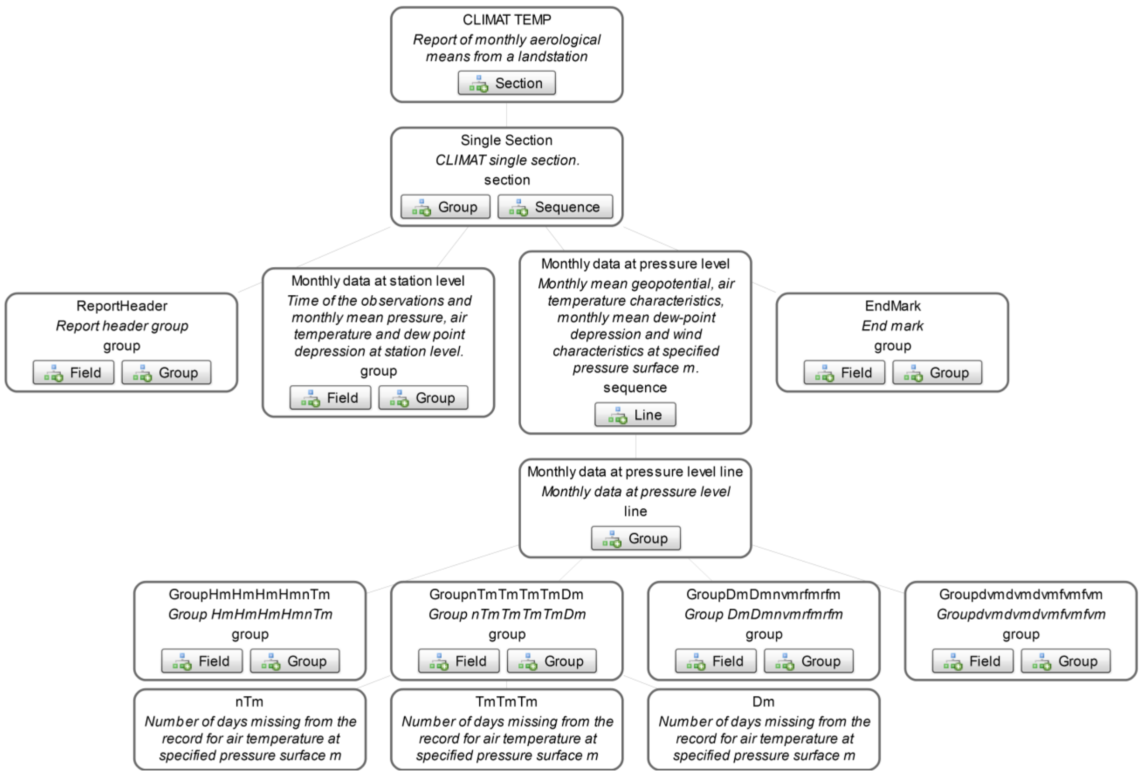

CLIMAT TEMP
Report of monthly aerological means from a landstation
Section
Single Section
CLIMAT single section.
section
Group
Sequence
ReportHeader
Report header group
group
Field
Group
Monthly data at station level
Time of the observations and monthly mean pressure, air temperature and dew point depression at station level.
group
Field
Group
Monthly data at pressure level
Monthly mean geopotential, air temperature characteristics, monthly mean dew-point depression and wind characteristics at specified pressure surface m.
sequence
Line
EndMark
End mark
group
Field
Group
Monthly data at pressure level line
Monthly data at pressure level line
line
Group
GroupHmHmHmHmnTm
Group HmHmHmHmnTm
group
Field
Group
GroupnTmTmTmTmDm
Group nTmTmTmTmDm
group
Field
Group
GroupDmDmnvmrfmrfm
Group DmDmnvmrfmrfm
group
Field
Group
Groupdvmdvmdvmfvmfvm
Groupdvmdvmdvmfvmfvm
group
Field
Group
nTm
Number of days missing from the record for air temperature at specified pressure surface m
TmTmTm
Number of days missing from the record for air temperature at specified pressure surface m
Dm
Number of days missing from the record for air temperature at specified pressure surface m

Figure 26 CLIMAT TEMP diagram showing the leaves of the node $nT_mT_mT_mT_mD_m$.

Figure 26 shows part of CLIMAT TEMP message configuration diagram created using MENE.

7. Results

This thesis presented to the reader the concepts of software artifacts and consistency. In specific the case of XML configuration file artifacts and their consistency. This thesis also described specialization patterns and how to adapt them to validate XML artifacts so that they can be qualified as consistent or inconsistent with an XML schema. The concepts presented were put together in MENE, a support tool that utilizes specialization patterns to warn the user of inconsistent XML configuration files and provide guidance to solve the inconsistencies.

The adaptation of specialization patterns proposed in this thesis was successfully implemented. It requires as input an XML schema and an XML configuration file. The latter is automatically validated and in case that it is inconsistent with the schema the specialization pattern gives a list of tasks to be done in order to restore the consistency.

The outcome of this thesis is MENE, a working prototype tool that assists the developer in the process of editing XML configuration files. MENE uses the pattern engine of specialization patterns to validate the consistency of the configuration during runtime. In this way MENE provides a mechanism to maintain the consistency of the XML configurations that is being edited. The use of MENE also removes a source of inconsistency between two artifacts: diagrams and XML configuration files by providing support to export configurations to XML files and diagrams images in PNG format.

A framework to generate meteorological messages and the process needed to convert message specification into XML message configuration was presented in the case study. The process involves three artifacts that are subject to inconsistencies. The use of MENE to edit the configuration files gives the software developer a standardized method to edit configuration and produce consistent message diagram artifacts for the messages XML configuration files.

In order to prove that MENE can be used to create and edit message diagrams and XML message configuration files the message configuration for WMO message type FM 75–XII Ext. CLIMAT TEMP was created using solely MENE. The result was 2 artifacts, message diagram and XML configuration file consistent among them state that can be used by the generator framework. It is important to mention, that the calculations needed to produce the CLIMAT TEMP are not implemented in the scope of this work and therefore generating a real message is not possible yet.

7.1. Benefits

The use of MENE improves the current process used to generate XML message configuration files since MENE is the only tool needed to pass from a message code form to a valid configuration. The use of MENE creates many benefits in the process of creating and editing message configuration files. The most significant benefit is that the XML configuration files produced with MENE are validated against the XML configuration schema; and the consistency between design artifact and XML configuration files is enforced and maintained. The validity of the XML configuration files is shown at any time to the user giving him the information needed to correct the violations in case of inconsistencies.

There is no need for writing XML configuration files by hand. The XML production is done automatically by MENE this removes the amount of human errors and inconsistencies with the schema. The user of MENE does not need to know the internals of message generator framework and XML structure in order to create or edit an XML message configuration. The elements of message generator framework (SchemaItem, SchemaItemCollection and Set and their relations) are encapsulated in message elements that appear in the final coded message and map to nodes in the diagram showed in MENE.

The configuration files produced by MENE follow a common structure. This helps to reduce discrepancies among different messages of the same strategy. The configuration files produces with MENE can be saved as diagrams. These diagrams are simpler and cleaner than the old diagrams.

7.2. Problems

Besides the benefits introduce by the use of specialization patterns and MENE, the use of the tool has also few limitations. First of all MENE enforces the creation of configurations following the structure presented in section 5.2 therefore only configuration that are compliant with that structure can be displayed and edited using MENE. This is a big limitation of MENE but old message configuration can be updated to follow the common structure.

MENE user interface only supports the current XML message configuration schema. Changes in the roles, e.g. a new attribute, are reflected and handle properly by the pattern engine, but the user interface forms do not provide means to edit the new fields. This is not a big issue since the XML message configuration schema is not expected to suffer modifications frequently. Another problem with the user interface is the use of Kalileo community license. It only allows its use for internal use of the organization, therefore as far as MENE is used as a prototype to demonstrate that the concepts of this thesis Kalileo can be used; but in case the use of MENE is extended a standard license needs to be acquired or MENE needs to be modified to use a different diagrammer library.

Finally MENE presents several usability issues than could be improved in order to increase the user experience and facilitate the execution of common task. For example the use of copy-paste and key shortcuts would increase the user experience by making the edit process faster.

8. Conclusions

Software developers use different software artifacts as input to perform tasks that produce new artifacts. Inconsistencies between artifacts at different levels of abstraction can arise during or between phases of software development. Over time, such inconsistencies must be fixed in order to produce a working software system, or at least partially resolved to produce part of a system for testing and quality assurance purposes [40]. If inconsistencies are not resolved they could lead to software erosion and poor quality products. Therefore the impact of consistency should not be underestimated. Inconsistencies among artifacts should be prevented and resolve as earliest as possible.

A design pattern that abstracts the structural description of a framework extension point can be used to qualify an element of a framework with a status. A tool built using this pattern can then provide an overview of the framework status and identify the action points needed to reach a valid status.

Inconsistency of software artifacts can be tackled and avoided by the use of tools that support traceability of changes and management of consistency. The use of specialization patterns as the back end of a user interface support tool is a working solution to maintain at least two levels of artifacts in consistent state. Specialization pattern can provide at any time the state of consistency for a given artifact. This thesis applied these concepts to create MENE, a tool that solves the consistency problem between message diagrams and XML message configuration files existing in a specific framework that can be used generate meteorological messages. MENE allows the user to read an XML message configuration schema, create a new message configuration diagram and generate an XML configuration file out of it. And the use of MENE keeps in consistency the diagrams and XML configuration files.

8.1. Further development ideas

The current adaptation of specialization patterns satisfies the need of the current version of MENE. However it is possible to extend the implementation of specialization pattern to produce class skeletons and method stubs for certain message elements. For example if a new calculation is used in a message element Field, the user needs to provide the name of the method, the parameters and the return value. This means that all the information needed to create a stub for that calculation is available in the specialization pattern already. For calculations that are defined in a DynamicType type-introspection can be used to automatically populate the information related to the calculation. This will remove a source of error and reduce the amount of work because the developer will not need to use a different tool to get the information about the existing calculations. With this features MENE would have a tighter integration with the process not only of generating the configuration files but the actual process of implementing the message itself.

Currently MENE only helps the developer in the process to generate the configuration file; but it is possible to integrate it with other existing tools. For example MENE can be linked to a tool that generates messages using message generator framework so that MENE can command the generation of messages for the configuration being edited. This will allow the developer to see how the configuration is used by the framework by analyzing the generated message. A feature that can be useful to many developers is a preview generation of a message based on the edited configuration.

The use of MENE currently has one big limitation; it cannot be used to edit configuration that do not follow the structure defined in 5.2. This limitation prevents the developers to edit the existing configurations using MENE. A possible solution to this problem is to use EXtensible Stylesheet Language (XSLT) to manipulate the old configuration and make them follow the structure supported by MENE. However this needs to be considered since the effort needed to perform this task can be bigger than the effort required to recreate the configuration files using MENE.

MENE can be extended and generalized to support other configuration schemas. For instance it can be easily adapted to allow the editing of the full message generator configuration database. This generalization can include a mechanism to configure MENE so that the forms used to edit elements are dynamically generated.

Bibliography

[1] Lehman M, Belady L. Program Evolution: Processes of Software Change. London: Academic Press; 1985.

[2] TC Software. Custom Software vs Packaged Software. [Internet]. 2010 [Last accessed November 25, 2010]. Available from: http://tcsoftware.net/articles/custom-bespoke-software.html.

[3] Fayad M, Schmidt D. Object-oriented application frameworks. Communications of the ACM. 1997;40(10):32-38.

[4] W3C. Extensible Markup Language (XML). [Internet]. 2011 [Last accessed September 16, 2011]. Available from: http://www.w3.org/XML/.

[5] Bryan P. Parameterize Your Apps Using XML Configuration In The.NET Framework 2.0. [Internet]. 2006 [Last accessed November 30, 2011]. Available from: http://msdn.microsoft.com/en-us/magazine/cc163591.aspx.

[6] Rational Software. Best practices for software development teams. A Rational Software Corporation White Paper. 1998.

[7] Antoniol G, Caprile B, Potrich A, Tonella P. Design-code traceability for object-oriented systems. Annals of Software Engineering. 2000;9(10):35-58.

[8] Looise M. Inter-level consistency checking between requirements and design artefacts. University of Twente; 2008.

[9] Ivkovic I, Kontogiannis K. Tracing evolution changes of software artifacts through model synchronization. In: Proceedings of the 20th IEEE International; 2004; Washingto, DC. p. 252-261.

[10] Baxter I, Pidgeon C. Software Change Through Design Maintenance. In: Proceedings of International Conference on Software Maintenance; 1997; Bari, Italy. p. 250-259.

[11] Aldrich J, Chambers C, Notkin D. ArchJava: connecting software architecture to implementation. In: IEEE, editor. Proceedings of the 24rd International Conference on Software Engineering, 2002. ICSE 2002.; 2002; Orlando, Florida. p. 187-197.

[12] Nuseibeh B. To be and not to be: On managing inconsistency in software development. In: Proceedings of the 8th International Workshop on Software

Specification and Design; 1996. p. 164-169.

[13] Anfurrutia F, Diaz O, Trujillo S. On Refining XML Artifacts. In: Proceedings of the 7th international conference on Web engineering; 2007. p. 473-478.

[14] Arenas M, Fan W, Libkin L. Consistency of XML specifications. Inconsistency Tolerance. 2005;3300:15-41.

[15] Adobe Platform Evangelism Team. What is Flex. [Internet]. [Last accessed November 4, 2011]. Available from: http://flex.org/what-is-flex/.

[16] Fan W. XML constraints: Specification, analysis, and applications. In: Proceedings of the 1st International Workshop on Logical Aspects and Applications of Integrity Constraints. Included in Proceedings of DEXA 2005, International Workshop on Database and Expert Systems Applications; 2005; Copenhagen, Denmark. p. 805-809.

[17] Arenas M, Fan W, Libkin L. On Verifying Consistency of XML Specifications. In: Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems; 2002; Madison, Wisconsin. p. 259-270.

[18] SGML. Cover Pages. [Internet]. 2002 [Last accessed June 1, 2010]. Available from: http://xml.coverpages.org/sgml.html.

[19] W3C. W3C. [Internet]. 2011 [Last accessed July 16, 2011]. Available from: http://www.w3.org/standards/xml/schema.

[20] W3Schools. Introduction to XML Schema. [Internet]. [Last accessed January 16, 2011]. Available from: http://www.w3schools.com/schema/schema_intro.asp.

[21] O'reilly C, Morrow P, Bustard D. Lightweight Prevention of Architectural Erosion. In: Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of; 2003. p. 59--64.

[22] Ma Y, Chen J, Wu J. Research on the phenomenon of software drift in software processes. In: eighth International Workshop on Principles of Software Evolution; 2005; Lisbon, Portugal. p. 195-198.

[23] Van Gurp J, Bosch J. Design erosion: problems and causes. Journal of Systems and Software. 2002;61(2):105-119.

[24] Perry D, Wolf A. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes. 1992;17(4):40-52.

[25] Riehle D, Züllighoven H. Understanding and using patterns in software development. TAPOS. 1996;2(1):3-13.

[26] Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns - Elements of

Reusable Object-Oriented Software. Addison-Wesley; 1995.

[27] Hakala M, Hautamäki J, Koskimies K, Paakki J, Viljamaa A, Viljamaa J. Generating Application Development Environments for Java Frameworks. 2001.

[28] Hautamäki J. Task-Driven Framework Specialization Goal-Oriented Approach. 2002.

[29] Hakala M, Hautamäki J, Koskimies K, Paakki J, Viljamaa A, Viljamaa J. Annotating Reusable Software Architectures with Specialization Patterns. In: Proceedings. Working IEEE/IFIP Conference on Software Architecture; 2001. p. 171-180.

[30] National Weather Service Internet Web Team. Meteorological Character Codes. [Internet]. 2002 [Last accessed November 6, 2011]. Available from: http://www.nws.noaa.gov/tg/code.html.

[31] Vaisala Oy. Vaisala DigiCORA® Sounding System MW31. [Internet]. 2010 [Last accessed February 6, 2011]. Available from: http://www.vaisala.com/en/meteorology/products/soundingsystemsandradiosondes/systems/Pages/DigiCORA-MW31.aspx.

[32] Vaisala Oy. Vaisala MARWIN® Sounding System MW32. [Internet]. 2010 [Last accessed February 6, 2011]. Available from: http://www.vaisala.com/en/products/soundingsystemsandradiosondes/soundingsystems/Pages/MW32.aspx.

[33] Wierenga R, Clowney F. Universal Upper Air Sounding System. International Met Systems. 2005;3:2.

[35] Naval Education And Training Professional Development And Technology Center. Aerographer’s Mate Module 2—Miscellaneous Observations and Codes. Training Material. Washington, D.C. 1998.

[36] Vaisala Oy. Message Generator Service Configuration Guide. 2007.

[37] Baik J. The Effects of CASE Tools on Software Development Effort. Dissertation. University of Southern California; 2000.

[38] Kap IT. Kalileo. [Internet]. 2009 [Last accessed July 13, 2011]. Available from: http://lab.kapit.fr/display/kalileo/Kalileo.

[39] Kap IT. Kap Lab Developer Guide. [Internet]. [Last accessed July 13, 2011]. Available from: http://lab.kapit.fr/help/index.jsp.

[40] GrundyJ, HoskingJ, MugridgeR. Inconsistency management for multiple-view software development environments. IEEE Transactions on Software

Engineering. 1998;24(11):960-981.

[41] Lee D, Wesley C. Comparative analysis of six XML schema languages. ACM Sigmod Record. 2000;29(3):76-87.

[42] W3schools. DTD Tutorial. [Internet]. [Last accessed January 16, 2011]. Available from: http://www.w3schools.com/dtd/default.asp.

[43] Vanhatupa J. Varma: Pattern-driven tool support for xml-based variation management. MSc Thesis. Tampere: Tampere University of Technology; 2006 May 8.

[44] Hoss A. Ontology-based methodology for error detection in software design. PhD Thesis. Louisiana State University Graduate School; 2007.

[45] W3schools. Introduction to XML. [Internet]. [Last accessed November 1, 2011]. Available from: http://www.w3schools.com/xml/xml_whatis.asp.

Appendix A XML schemas comparison

The following table presents a feature comparison of the three most common used XML schemas [41].

Features	DTD 1.0	XSD 1.0	Schematron 1.4
Schema			
syntax in XML	No	Yes	Yes
namespace	No	Yes	Yes
include	No	Yes	No
import	No	Yes	No
Data typing			
built-in type	10	37	0
user-defined type	No	Yes	No
domain constraint	No	Yes	Yes
null	No	Yes	No
Attribute			
default value	Yes	Yes	No
choice	No	No	Yes
optional vs. required	Yes	Yes	Yes
domain constraint	Partial	Yes	Yes
conditional definition	No	No	Yes
Element			
default value	No	Partial	No
content model	Yes	Yes	Yes
ordered sequence	Yes	Yes	Yes
unordered sequence	No	Yes	Yes
choice	Yes	Yes	Yes
min & max occurrence	Partial	Yes	Yes
open model	No	No	Yes
conditional definition	No	No	Yes

Features	**DTD 1.0**	**XSD 1.0**	**Schematron 1.4**
Inheritance			
simple type by extension	No	No	No
simple type by restriction	No	Yes	No
complex type by extension	No	Yes	No
complex type by restriction	No	Yes	No
Constraints			
uniqueness for attribute	Yes	Yes	Yes
uniqueness for non-attribute	No	Yes	Yes
key for attribute	No	Yes	Yes
key for non-attribute	No	Yes	Yes
foreign key for attribute	Partial	Yes	Yes
foreign key for non-attribute	No	Yes	No
Miscellaneous			
dynamic constraint	No	No	Yes
version	No	No	No
documentation	No	Yes	Yes
embedded HTML	No	Yes	Partial
self-describability	No	Partial	Partial

Appendix B Message generator framework elements and descriptions

Message element	**Description**	**Parent element**
Message Strategy	*Message strategy* is a container for all message types that are related to each other. For example message strategy TEMP includes all messages of type TEMP.	
Message Type	A *message type* defines a *message compositor* and *message formatter* for a WMO message code form. This is the element that is passed in the generation process time to command the coding of a specific message.	Message Strategy
Message Compositor	The *message compositor* defines which *message schema* is used to code the message. It is responsible for reading the observation data and composing an intermediate message in XML format.	Message Type
Message Schema	A *Message schema* defines the tree structure that describes the WMO message code form structure. *Message schema* is the root element of a *message type*.	Message Compositor
Schema Item Collection	*Schema item collection* is a container node in the schema tree. This is always associated with a *set* and a *schema item*.	Schema Item
Schema Item	A *schema item* is a node in the message schema tree. A *schema item* can optionally have a dynamic calculation associated to it which determines if the item is included or not in the message. If a dynamic calculation is defined it returns a boolean value indicating whether the schema item is included or not in the schema tree.	Message Schema
Set	A *schema item* that contains a group of *schema items*. A set can be of type heading, section, group, data subset, sequence, replication or line.	Schema Item

	This type is used by the *formatter* to support custom formatting of different types of sets	
Descriptor	A *schema item* that contains metadata to provide extra information for *sets*. This metadata is used in binary format messages like BUFR.	Schema Item
Field	A *schema item* that represents a leaf node, which is a real entry in the final message. The *field* output value is obtained by three different forms in this order of precedence: calculation, the value is calculated; variable, the value maps to a defined variable; default, a given default value.	Schema Item
Message Formatter	The *message formatter* defines what *message layout* is used to format a message type. The *message formatter* reads the intermediate XML message generated by the *message compositor* and transforms it into its final layout.	Message Type
Message Layout	*Message layout* is a container node of all the *layout items* used to format a message.	Message Formatter
Layout Items	A *layout item* defines the format to use for each *set* type. The *layout item* can be applied to a specific *set* or to all *set*s of the same type	Message Layout
Formats	*Format* defines a default mask to represent invalid node values and the format string to be used to format the node value when it is valid.	Layout item

Feature comparison of XML schemas [41]

Appendix C Example set of common message elements

```xml
<?xml version="1.0" encoding="utf-8"?>
<root xsi:noNamespaceSchemaLocation="Message_config.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <XmlNodes>
    <Row XmlNodePk="FieldNode" Name="Field" />
    <Row XmlNodePk="SectionNode" Name="Section" />
    <Row XmlNodePk="GroupNode" Name="Group" />
    <Row XmlNodePk="SequenceNode" Name="Sequence" />
    <Row XmlNodePk="LineNode" Name="Line" />
    <Row XmlNodePk="DataSubsetNode" Name="DataSubset" />
    <Row XmlNodePk="ElementDescriptorNode" Name="ElementDescriptor" />
    <Row XmlNodePk="SequenceDescriptorNode" Name="SequenceDescriptor" />
    <Row XmlNodePk="OperatorDescriptorNode" Name="OperatorDescriptor"/>
    <Row XmlNodePk="ReplicationNode" Name="Replication" />
  </XmlNodes>

  <Variables>
    <Row VariableNamePk="MsgGenerationTime" Description="Date and time of message"
     Type="DateTime" IsMandatory="false"/>
    <Row VariableNamePk="WMOBlock" Description="WMO block number." Type="Int32"
     DefaultValue="0" IsMandatory="false"/>
    <Row VariableNamePk="WMOStation" Description="WMO station number." Type="Int32"
     DefaultValue="0" IsMandatory="false" MeasurandNameFk="WMOStation"/>
    <Row VariableNamePk="LaunchTime" Description="Date and time of the observation"
     Type="DateTime" IsMandatory="false" MeasurandNameFk="LaunchTime"/>
    <Row VariableNamePk="WindFindingType" Description="Type of measuring equipment used"
     Type="String" DefaultValue="" IsMandatory="false"
     MeasurandNameFk="WindFindingType"/>
    <Row VariableNamePk="StationName" Description="Station name" Type="String"
     DefaultValue="0" IsMandatory="false" MeasurandNameFk="StationName"/>
    <Row VariableNamePk="HeightConfidence" Description="Height Confidence" Type="Int32"
     DefaultValue="1" IsMandatory="false" MeasurandNameFk="HeightConfidence"/>
    <Row VariableNamePk="Latitude" Description="Latitude" Type="Double" DefaultValue="0"
     IsMandatory="false" MeasurandNameFk="StationLatitude" />
    <Row VariableNamePk="Longitude" Description="Longitude" Type="Double"
     DefaultValue="0" IsMandatory="false" MeasurandNameFk="StationLongitude" />
    <Row VariableNamePk="Altitude" Description="Altitude" Type="Double" DefaultValue="0"
     IsMandatory="false" MeasurandNameFk="StationAltitude" />
    <Row VariableNamePk="WindDirection" Description="Wind direction" Type="Double"
     IsMandatory="false" MeasurandNameFk="WindDir"/>
    <Row VariableNamePk="WindSpeed" Description="Wind speed" Type="Double"
     IsMandatory="false" MeasurandNameFk="WindSpeed"/>
    <Row VariableNamePk="RadioRxTime" Description="RadioRxTime" Type="Double"
     IsMandatory="false" MeasurandNameFk="Time"/>
    <Row VariableNamePk="Height" Description="Height" Type="Double" IsMandatory="false"
     MeasurandNameFk="Height"/>
    <Row VariableNamePk="WindEast" Description="WindEast" Type="Double"
     IsMandatory="false" MeasurandNameFk="WindEast"/>
    <Row VariableNamePk="WindNorth" Description="WindNorth" Type="Double"
     IsMandatory="false" MeasurandNameFk="WindNorth"/>
    <Row VariableNamePk="UnitsOfAltitude" Description="Units of used height"
     Type="String" IsMandatory="false" MeasurandNameFk="UnitsOfAltitude"/>
    <Row VariableNamePk="IsMaxWindAtTopOfSounding"
     Description="Indicates id the max wind occured at the top of the sounding"
     Type="Boolean" MeasurandNameFk="MaxWindAtTopOfSounding" IsMandatory="false" />
    <Row VariableNamePk="MaxWindRadioRxTime"
     Description="RadioRxTime at a max wind speed level" Type="Double"
     MeasurandNameFk="MaxWindRadioRxTime" IsMandatory="false" />
    <Row VariableNamePk="MaxWindHeight" Description="Height at a max wind speed level"
     Type="Double" MeasurandNameFk="MaxWindHeight" IsMandatory="false" />
    <Row VariableNamePk="MaxWindSpeedEast"
     Description="East speed component at a max wind speed level" Type="Double"
     MeasurandNameFk="MaxWindSpeedEast" IsMandatory="false" />
    <Row VariableNamePk="MaxWindSpeedNorth"
     Description="North speed component at a max wind speed level" Type="Double"
     MeasurandNameFk="MaxWindSpeedNorth" IsMandatory="false" />
    <Row VariableNamePk="MaxWindPressure"
     Description="Pressure at a max wind speed level" Type="Double"
     MeasurandNameFk="MaxWindPressure" IsMandatory="false" />
    <Row VariableNamePk="MaxWindDirection"
     Description="Pressure at a max wind speed level" Type="Double"
     MeasurandNameFk="MaxWindDirection" IsMandatory="false" />
    <Row VariableNamePk="MaxWindSpeed"
     Description="Wind speed at a max wind speed level" Type="Double"
     MeasurandNameFk="MaxWindSpeed" IsMandatory="false" />
    <Row VariableNamePk="StdLevelPressure" Description="Pressure at the standard level."
     Type="Double" IsMandatory="false" MeasurandNameFk="FilteredStdLevelPressure"/>
    <Row VariableNamePk="StdLevelWindSpeed"
     Description="Wind speed at the standard level." Type="Double" IsMandatory="false"
     MeasurandNameFk="FilteredStdLevelWindSpeed"/>
    <Row VariableNamePk="StdLevelWindDirection"
     Description="Wind direction at the standard level." Type="Double"
     IsMandatory="false" MeasurandNameFk="FilteredStdLevelWindDirection"/>
    <Row VariableNamePk="WindGroupStartIndex"
     Description="Start index of the current wind group." Type="Int32"
     IsMandatory="false"/>
    <Row VariableNamePk="WindGroupCount" Description="Number of wind groups to follow."
     Type="Int32" IsMandatory="false"/>
    <Row VariableNamePk="WindSpeedUnits" Description="Wind speed units" Type="String"
     IsMandatory="false" MeasurandNameFk="WindSpeedUnits"/>
    <Row VariableNamePk="ShearWanted"
     Description="Flag indicating whether shear is wanted" Type="Boolean"
     IsMandatory="false" MeasurandNameFk="ShearWanted"/>
  </Variables>

  <Measurands>
    <Row MeasurandNamePk="StationName" Description="Station name."
     ExternalName="StationName" ScaleNameFk="none"/>
    <Row MeasurandNamePk="WMOStation" Description="WMO station number."
     ExternalName="WMOStation" ScaleNameFk="none"/>
    <Row MeasurandNamePk="HeightConfidence" Description="Height confidence."
     ExternalName="HeightConfidence" ScaleNameFk="none"/>
    <Row MeasurandNamePk="LaunchTime" Description="LaunchTime" ExternalName="LaunchTime"
     ScaleNameFk="none"/>
    <Row MeasurandNamePk="WindFindingType" Description="" ExternalName="WindFindingType"
     ScaleNameFk="none" />
    <Row MeasurandNamePk="WindDir" Description="Wind direction" ExternalName="WindDir"
     UnitNameFk="Degree" ScaleNameFk="none" />
    <Row MeasurandNamePk="WindSpeed" Description="Wind speed" ExternalName="WindSpeed"
     UnitNameFk="MetersPerSecond" ScaleNameFk="none"/>
    <Row MeasurandNamePk="WindEast" Description="East wind speed"
     ExternalName="WindEast" UnitNameFk="MetersPerSecond" ScaleNameFk="none"/>
    <Row MeasurandNamePk="WindNorth" Description="North Wind speed"
     ExternalName="WindNorth" UnitNameFk="MetersPerSecond" ScaleNameFk="none"/>
    <Row MeasurandNamePk="StationLatitude" Description="Station latitude"
     ExternalName="StationLatitude" UnitNameFk="Degree" ScaleNameFk="none"/>
    <Row MeasurandNamePk="StationLongitude" Description="Station longitude"
     ExternalName="StationLongitude" UnitNameFk="Degree" ScaleNameFk="none"/>
    <Row MeasurandNamePk="StationAltitude" Description="Station altitude"
     ExternalName="StationAltitude" UnitNameFk="Meter" ScaleNameFk="none"/>
    <Row MeasurandNamePk="Time" Description="" ExternalName="Time" ScaleNameFk="none" />
    <Row MeasurandNamePk="Height" Description="Height" ExternalName="Height"
     ScaleNameFk="none" />
    <Row MeasurandNamePk="UnitsOfAltitude" Description="Units of altitude"
     ExternalName="UnitsOfAltitude" ScaleNameFk="none"/>
    <Row MeasurandNamePk="MaxWindAtTopOfSounding" Description=""
     ExternalName="MaxWindAtTopOfSounding" ScaleNameFk="none"/>
    <Row MeasurandNamePk="MaxWindPressure" Description="Pressure at maximum level"
     ExternalName="MaxWindPressure" ScaleNameFk="none"/>
    <Row MeasurandNamePk="MaxWindDirection"
     Description="Wind direction at maximum level" ExternalName="MaxWindDirection"
     ScaleNameFk="none"/>
    <Row MeasurandNamePk="MaxWindSpeed" Description="Wind direction at maximum level"
     ExternalName="MaxWindSpeed" UnitNameFk="MetersPerSecond" ScaleNameFk="none"/>
    <Row MeasurandNamePk="MaxWindSpeedEast"
     Description="East wind speed at maximum level" ExternalName="MaxWindSpeedEast"
     UnitNameFk="MetersPerSecond" ScaleNameFk="none"/>
```

```
    <Row MeasurandNamePk="MaxWindSpeedNorth"
     Description="North Wind speed at maximum level" ExternalName="MaxWindSpeedNorth"
     UnitNameFk="MetersPerSecond" ScaleNameFk="none"/>
    <Row MeasurandNamePk="MaxWindRadioRxTime"
     Description="RadioRxTime at maximum wind level" ExternalName="MaxWindRadioRxTime"
     ScaleNameFk="none"/>
    <Row MeasurandNamePk="MaxWindHeight" Description="Height at maximum wind level"
     ExternalName="MaxWindHeight" ScaleNameFk="none"/>
    <Row MeasurandNamePk="HeightPilot" Description="" ExternalName="HeightPilot"
     ScaleNameFk="none"/>
    <Row MeasurandNamePk="FilteredStdLevelPressure"
     Description="Pressure at the standard level."
     ExternalName="FilteredStdLevelPressure" UnitNameFk="Pascal" ScaleNameFk="hecto"/>
    <Row MeasurandNamePk="FilteredStdLevelWindSpeed"
     Description="Wind speed at the standard level."
     ExternalName="FilteredStdLevelWindSpeed" UnitNameFk="MetersPerSecond"
     ScaleNameFk="none"/>
    <Row MeasurandNamePk="FilteredStdLevelWindDirection"
     Description="Wind direction at the standard level."
     ExternalName="FilteredStdLevelWindDirection" UnitNameFk="Degree"
     ScaleNameFk="none"/>
    <Row MeasurandNamePk="WindSpeedUnits" Description="Wind speed units"
     ExternalName="WindSpeedUnits" ScaleNameFk="none"/>
    <Row MeasurandNamePk="ShearWanted"
     Description="Flag indicating whether shear is wanted" ExternalName="ShearWanted"
     ScaleNameFk="none"/>
  </Measurands>

  <Formats>
    <Row FormatNamePk="NoFormat" Description="No format." InvalidMask=""
     FormatString="{0}"/>
    <Row FormatNamePk="LineBefore" Description="new line + value"
     InvalidMask="" FormatString="\r\n{0}"/>
    <Row FormatNamePk="9GroupsPerLineStartingWithNewLine"
     Description="new line + Blocks of nine groups" InvalidMask=""
     FormatString="\r\n{0}"/>
    <Row FormatNamePk="9GroupsPerLineNoNewLine" Description="Blocks of nine groups"
     InvalidMask="" FormatString="{0}"/>
    <Row FormatNamePk="9GroupsPerLine" Description="Blocks of nine groups + newline"
     InvalidMask="" FormatString="{0}\r\n"/>
    <Row FormatNamePk="RemoveTrailingSpaces"
     Description="Remove trailing spaces from lines." InvalidMask=""
     FormatString="{0}" />
    <Row FormatNamePk="RemoveTrailingSpacesNewLine"
     Description="Remove trailing spaces from lines and new line."
     InvalidMask="" FormatString="{0}\r\n" />
    <Row FormatNamePk="RemoveTrailingSpacesStartingWithNewLine"
     Description="New line and remove trailing spaces from lines."
     InvalidMask="" FormatString="\r\n{0}" />
    <Row FormatNamePk="SpaceDelimitered" Description="value + space."
     InvalidMask="" FormatString="{0} " />
    <Row FormatNamePk="Line" Description="line + newline."
     InvalidMask="" FormatString="{0}\r\n" />
    <Row FormatNamePk="Time1" Description="Time in yyyyMMdd24mm + space."
     InvalidMask="" FormatString="yyyyMMddHHmm " />
    <Row FormatNamePk="Int1" Description="Integer with length 1."
     InvalidMask="/" FormatString="0" />
    <Row FormatNamePk="Int2" Description="Zero padded integer with length 2."
     InvalidMask="/" FormatString="00" />
    <Row FormatNamePk="Int3" Description="Zero padded integer with length 3."
     InvalidMask="/" FormatString="000" />
    <Row FormatNamePk="Int4" Description="Zero padded integer with length 4."
     InvalidMask="/" FormatString="0000" />
  </Formats>
</root>
```