University of Oslo
Department of Informatics

Evaluating the suitability of EML 4.0 for the Norwegian Electoral System

Patricia Aas

A prototype approach

June 22, 2005

Abstract

The Election Markup Language (EML) is a communication language used between and within different subsystems of a computerized electoral information system. EML is defined by means of a set of 33 XML Schemas.

This thesis tests the hypothesis that the EML communication language is suitable for a computerized Norwegian Electoral System. The testing is performed using a prototype implementation in Java. Though the implementation does not take into consideration security and anonymity concerns, it is a full implementation of the Electoral System. The prototype system consists of five subsystems that communicate using a network connection. The implementation spans 10115 lines of code and 58 classes.

EML is found to be very close to a communication language suitable for the Norwegian Electoral System, though a few changes would have to be made to the standard to express the information exchange required by law. The shortcomings in EML are countered with proposed changes in the standard, and in addition a some parts of the Norwegian Election Law and Election regulations are proposed changed.

Contents

Chapter 1

Introduction

1.1 Introduction

An electoral system encapsulates different subsystems, either manual or computerized, with different purposes and periods of execution. Some are used in preparation for an election, some during and some after the election. These can be viewed as autonomous entities that perform very clearly defined tasks. Today these tasks are described in Norwegian laws as if they were to be performed manually, although many are already computerized. The Election Markup Language (EML) is a standardized communication language to be used between, and within, different computerized subsystems in an electoral information system. Such a standardized language is meant to make it easier to integrate different subsystems, so that these can be delivered by different software suppliers and still be able to communicate. For this reason, the Council of Europe's Committee of Ministers recommended EML in Rec(2004)11[1]. The idea of not leaving the entire electoral system, with all its subsystems, to one supplier, has its democratic advantages. One might not wish to leave the entire power of information to one company. This idea is reflected in the modern democratic principle of division of power and the arguments there can also be used here. Quite simply, a check and balance system is there to keep everyone honest. By dividing a system into subsystems with clear interfaces, it is possible to exchange one subsystem from one producer with another one from a different company. This reflects the basic idea in Object Orientation, where a part of a system (or in this case, a subsystem as a whole) adheres to a clear definition of what services it provides. Since the EML standard has been developed with very little Norwegian involvement, it is the purpose of this thesis to explore if it can

[1]"Open standards shall be used to ensure that the various technical components or services of an e-voting system, possibly derived from a variety of sources, interoperate. At present, the Election Markup Language (EML) standard is such an open standard and in order to guarantee interoperability, EML shall be used whenever possible for e-election and e-referendum applications" [22, B.66-67]

carry the data needed for the elections called for by the Norwegian Election Law as of today.

1.1.1 Concept and Word Clarification

Since this thesis concerns itself with the Norwegian Electoral System, many words and concepts are in Norwegian. This thesis, however, is written in English and to avoid misunderstandings it is important to clarify these words and concepts early on. The translation of the relevant words and concepts follow the table supplied in appendix D. Concepts that require a more detailed definition are defined when used.

1.2 Modeling the Norwegian Electoral System through the Prototype Subsystems

The communication defined in EML 4.0 [1] is based on a model that divides the Electoral System into 6 subsystems. This model can be viewed in figure 1.1, which is a simplification of the figure “High-Level Model - Technical View” from *EML Process & Data Requirements* [5]. Following this model was a prerequisite for the thesis[2].

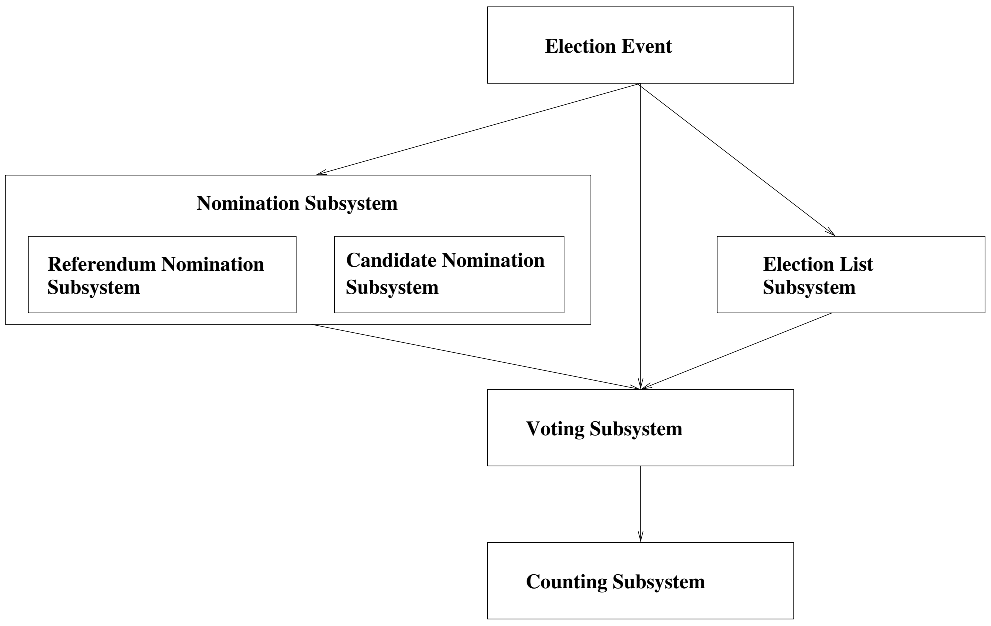

Figure 1.1: Information flow between the subsystems

[2]The thesis description is supplied in appendix C, note that it is in Norwegian

The subsystems in the model define different conceptual systems: A candidate nomination system, a referendum nomination system, an election list system, a voting system, a counting system and in addition an election event system. These systems are really only parts of the electoral system as a whole, and will be referred to as subsystems for the remainder of this thesis.

Only five of these subsystems will be explored in this thesis. The referendum nomination subsystem, which was introduced in EML 4.0, is not examined. This subsystem would, however, replace the candidate nomination subsystem in a referendum.

The Election Event subsystem defines the elections that are to be held in this *election event*[3] and outputs this information to three other subsystems through the EML document called ElectionEvent. The subsystem is, in this manner, the election definition subsystem, providing the other subsystems with the information necessary to begin the pre-election process.

The EML standard has, so far, evolved through four versions. The latest version is EML 4.0, which was released in January 2005. The versions are not backwards compatible, and though the thesis description called for the use of EML 3.0, this was later abandoned. The reasons for this are outlined in section B.1 in appendix B and further explained by the problems encountered in handling EML 3.0 that became apparent in the testing described in appendix A.

A part of this thesis will be the description of a five part prototype representing the five out of the six subsystems of an electoral system conforming to the EML model. The subsystems communicate using XML messages, often called EML documents, indicating that they are messages defined by EML.

1.3 EML Seen as a Language

The communication between subsystems and possibly between a subsystem and an external system or resource, is in the EML model achieved using XML documents. EML defines a language, and each schema defines the criteria for deciding whether an XML document is a member of this language or not, this deciding process is called validation. A XML document is valid with regard to a schema.

As a part of the prototype, the package called EML_Interface was implemented to handle the EML documents. The package EML_Interface encapsulates the concept of a document in EML. It is designed to provide a simple guarantee: All documents produced or received are valid. Each schema is represented by a class in EML_Interface, and the public methods in each class are meant to keep the above guarantee. Consequently, the

[3]The concept of an "election event" is defined in detail later in the thesis. For now, it is sufficient to say that an election event encapsulates elections that are held at the same time.

information placed into the document is clearly defined by the parameters allowed to these methods.

An EML schema consists of mostly optional fields. This being so, these methods also define a restriction to the schemas themselves in that they require some fields to be used and do not supply access to others. In this manner EML_Interface defines a subset of EML.

EML_Interface is imported by all the subsystems and none of these process a document through any other means than the public methods provided in this interface.

1.4 The Prototype Approach to EML 4.0

The communication within the EML model is defined in 33 XMLSchemas in EML 4.0. Some of these concern communication within a subsystem, others address the communication between subsystems.

Originally it was thought that the EML standard that would be used was EML 3.0. And though this was later abandoned, the description below was of the work done under that assumption.

In the initial phase of the development of the prototype subsystems, an issue that had to be dealt with was how to handle the XMLSchemas. Time was a factor, and hand coded parsers would take time away from the testing of the thesis question. Several approaches were explored, among others the approaches described in [8], which were adopted in the customizations of EML for the UK described in [6] and [7]. However, these do not explicitly deal with the parsing of XML documents, but rather with the validation of XML documents and restricting their format through requiring fields that are optional. Yet the focus of this thesis is not if the schemas are possible to restrict, but rather if they are too strict.

A parser generating tool was preferred over other more time consuming approaches. A binding compiler could be used to compile the elements in the schemas to classes that could further be used to validate and parse documents adhering to the schemas. A number of binding compilers were tested, this testing is described in appendix A.

During the testing the binding compilers it became clear that there were flaws in the EML 3.0 standard. Although an attempt was made to correct the schemas, this was really not an optimal solution since the corrected schemas could not still be thought of as representing the EML 3.0 standard. In the search for solutions to the problems found, the new EML 4.0 was investigated and it was found that in EML 4.0 the problems in EML 3.0 had been resolved. The focus of the thesis was then shifted to EML 4.0.

Although the previous problems in EML 3.0 had been resolved in 4.0, it was still not suitable for compilation. Even though most schemas compiled on their own, name conflicts appeared when they were compiled together.

It seemed preferable that they constitute one language also in code, to accomplish this, the name conflicts had to be resolved. There were two other minor changes, all changes are described in appendix B and do not change the format of the EML documents in any significant way.

The use of the binding compiler chosen is described in chapter 3. Note that the code generated by the binding compiler only facilitated the parsing and generating of documents adhering to EML schemas and did not reveal anything that was not already known through the diagrams depicting each schema in the EML Schema Descriptions [4].

Each of the subsystems will send and/or receive EML documents. The generated code described above is used by these subsystems to parse the documents on arrival and to create the documents that are to be sent. This parsing and creating of documents has been encapsulated in a separate package called EML_Interface, which contains a wrapper class for each document to provide the above mentioned guarantee: That the documents are valid at all times.

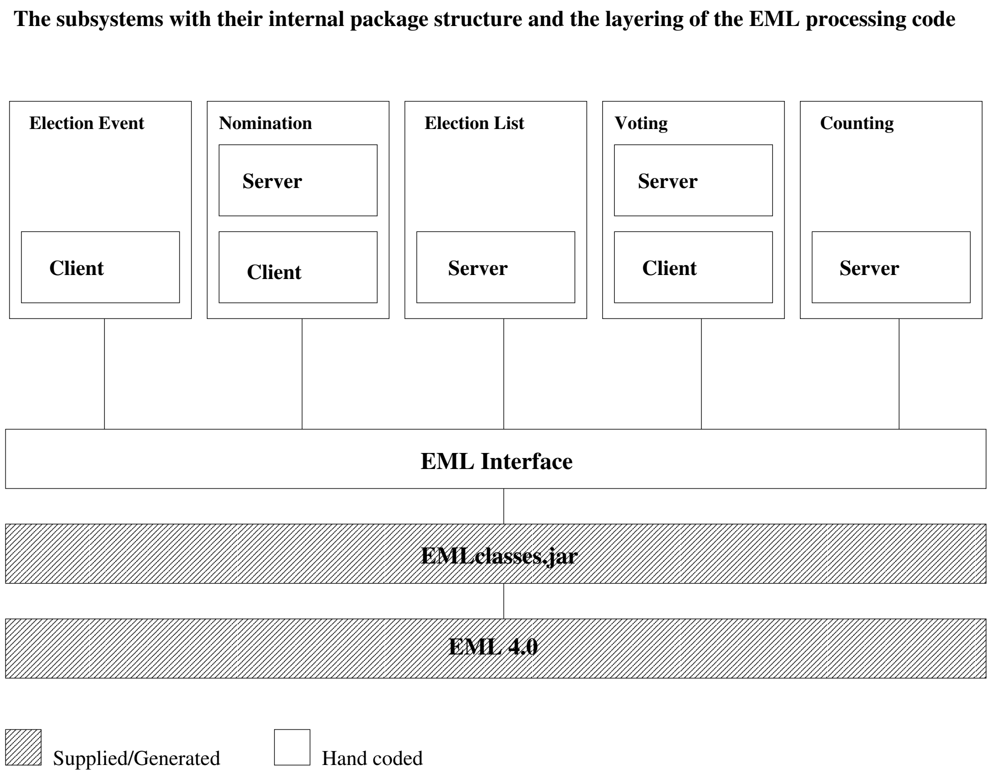

Figure 1.2: View of the layering of packages

The subsystems and EML_Interface were hand coded. Each of the subsystems can be seen as a pair consisting of a client application and server application, though some of the subsystems only contain one of these. The client and server applications are implemented as Java packages. Within each of the packages there are classes constituting the applications internal data structure and a central class representing the logical function of the application. The central classes take their name from the subsystem they are a part of, for example the central class in the Voting subsystems server application is called VotingServer.

The EML_Interface library is imported only by the central class in each application. In the package VotingClient, for example, only the VotingClient class imports EML_Interface. Figure 1.2 shows the layering of the packages. Note that everything has been hand coded by the author, with the exception of the code generated by the binding compiler (EMLclasses.jar) and, of course, EML. The subsystems and their internal applications are described in chapter 2 and the EML_Interface library is described in chapter 3.

1.5 The Scope of the Thesis

The design of the prototype subsystems is based strongly on the idea of testing the concerns of this thesis. However, it is necessary that they reflect a realistic model, so that any results are not achieved because of an unrealistic design.

The problem domain of this thesis can, in this context, be seen as a graph, where the subsystems are a collection of nodes and the EML documents passed are a collection of edges. This is also reflected in figure 1.1.

The purpose of this thesis is to evaluate the edges, that is EML, as it pertains to the subsystems. The evaluating criteria is whether EML can carry the data needed for the subsystems to perform their functions. To test this, the subsystems have to conform to the Norwegian Election Law and Election regulations.

Consequently, to be able to state anything about EML it must first be shown that the subsystems are in accordance with the laws and regulations that govern them. Only then can the conclusions about EML have any validity.

In the following chapter the subsystems will be described individually, together with the applicable laws and regulations. They will then be measured up to the requirements of the law. In chapter 3 the EML documents will be discussed individually and measured up to the requirements of the subsystems that utilize them.

Chapter 2

The Norwegian Electoral System and the Subsystems that Model it

2.1 Introduction

The purpose of this thesis is to explore how well EML is suited for the Norwegian Electoral System. This system is modeled by the prototype subsystems, and the goal of this chapter is to assure that these prototype subsystems do model the Norwegian Electoral System.

The laws that govern each subsystem will be interpreted and the implementation analyzed. In that way, each of the subsystems will be evaluated against the laws that govern it. In chapter 3 a closer examination of the documents passed between the subsystems will reveal if the EML schemas allow for the information exchange necessary for the subsystems to perform their functions.

2.2 Norwegian Laws and Regulations

The Norwegian Electoral System is governed by the Election Law [16] and, in addition, by the Electoral regulations [13] [14] issued by the Ministry of Local Government and Regional Development. These regulations expand upon the law itself and, to some extent, provides a more detailed view of the electoral process. In addition, the Electoral Handbook [18] provides information not sufficiently detailed in the law and regulations, in particular the counting algorithms.

The Election Law is divided into 16 chapters. Not all of these are relevant for this thesis, since many concern the organizational structure required when doing an election and others the rights and obligations of voters as well as the registration process of political party names (this last process is briefly discussed in conjunction with the Nomination subsystem).

As already mentioned, the Election Law describes an electoral process as if it were to be performed manually. In most cases this is not a problem, but occasionally it results in an ambiguity that has to be resolved in a computational environment. These ambiguities are probably solved today by acquiring the additionally needed information elsewhere. Examples of this will be mentioned where appropriate.

2.3 The Design of the Prototype

The prototype has different design ideas on different levels. These ideas are reflected throughout the subsystems, and it is the purpose of this section to explicitly state these ideas and justify their use. In the subsequent sections each subsystem is discussed and evaluated.

2.3.1 The Subsystems

The prototype models the Norwegian Electoral System. It does this through the implementation of five subsystems. These communicate EML messages over a standard network connection. The prototype subsystems span 40 classes with a total of 5788 lines of code. The size of the subsystems is presented in table 2.1.

Packages	Lines of code	Number of classes
CountingServer	963	6
ElectionEventClient	369	4
ElectionListServer	619	6
NominationClient	614	4
NominationServer	755	6
VotingClient	1386	7
VotingServer	1082	7
Total	5788	40

Table 2.1: Lines of code in the subsystems

The prototype simulates an election event with three elections: a Municipal Council Election, a County Council Election and a Storting Election. The simulation runs from the dispatch of the ElectionEvent document to the counting of votes in the Counting Server.

2.3.2 The Splitting of the Subsystems into a Server and a Client

The EML model is reflected in the structure of the prototype subsystems. However, some design issues are not detailed in the description of the model found in [4] and [5]. Specifically, the internal structure of each subsystem is not detailed. In a design were the responsibility of a subsystem is split into several cooperating subsystems, there are no guidelines on how the responsibility should be split. The design of the prototype subsystems has been based on the client/server architecture and the implementation of this design is briefly discussed below. This client/server application distribution is today an established model, and is, consequently, one of many likely options for such a subsystems design.

The Client/Server Architecture

Each prototype subsystem can be viewed as a client/server pair, though only two subsystems actually have both. This distinction is made here and throughout the thesis and code, to clarify whether a part of a subsystem is an active sender of EML documents or if it a passive receiver of such documents. Note that a passive receiver can also send documents, but will only do so if prompted by an active sender. Similarly an active sender can also receive documents, but will only expect these as a response to a prompt. Such an active sender will be called a client and a passive receiver will be called a server.

A server will go through three stages in its lifetime. In the first stage it will receive the input it needs to produce the required internal data structures. It will then move on to a server stage where it can be contacted by a client application and perform the services required. At some point in time the server period ends, and the server passes into its third and final stage, where it produces its output and communicates this output to the subsystems depending on it.

One can view these states simply as: an initialization/input state, a server/client state and an output state. Figure 2.1 depicts the prototype as a whole, with all its subsystems, this is a more detailed look at the same structure seen in figure 1.1 but with this state idea reflected. Each arrow indicates a transfer of information, primarily through XML documents conforming to EML schemas. Some information, however, is presumed to be transferred by other means, such as the information communicated to the voter prior to the election.

The overall view seen in figure 2.1 is crucial since the most important communications in this electoral system are between these subsystems or between a server and a client within a subsystem.

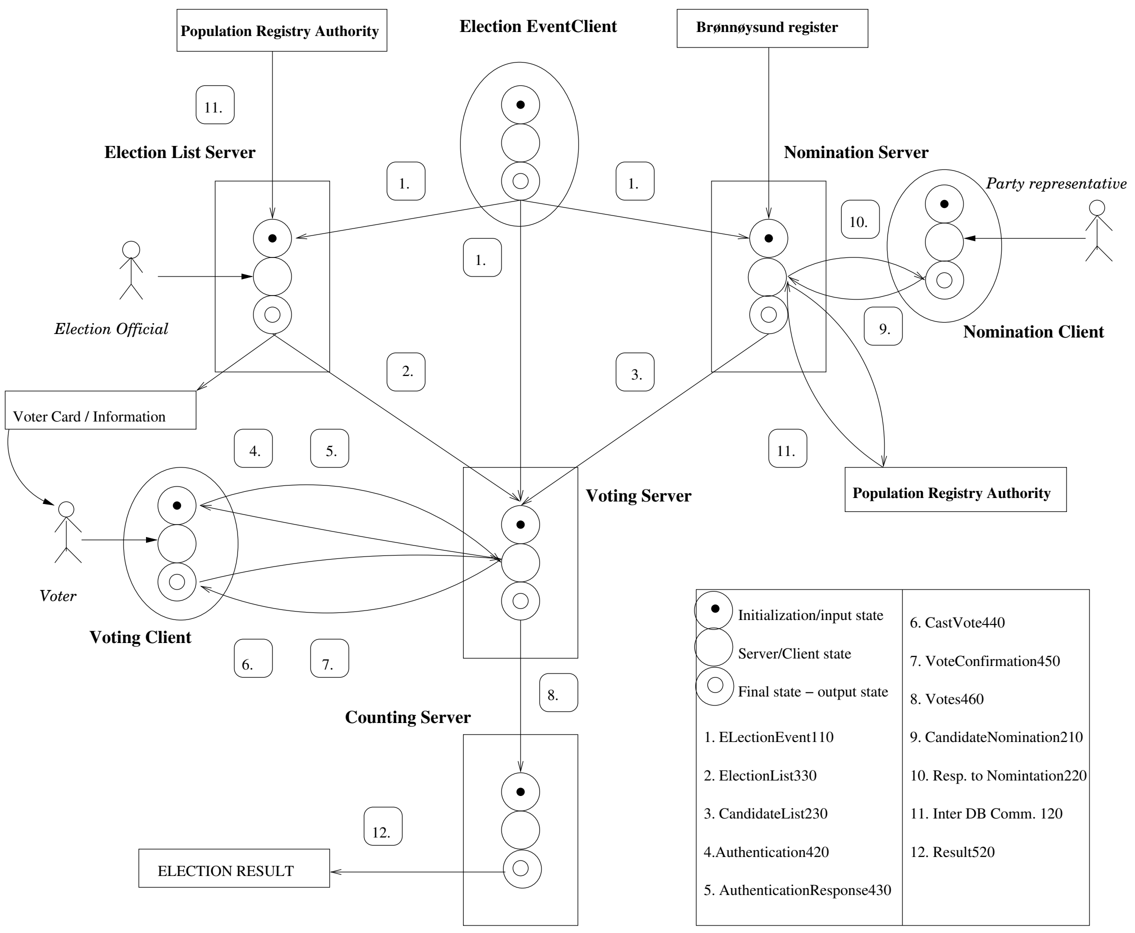

Figure 2.1: State model of the prototype subsystems

2.3.3 The Initialization Stage

The first state entered by a server or a client is the initialization/input state. What characterizes this state is that the server/client will build its internal data structure from the input received in this state.

For some of the servers and clients this input comes in the form of internal documents received from other parts of the electoral system. For some the input is read from files.

This design is to assure that a subsystem will not base its data structure on external input, for example votes received. The data structures will always be based on documents internal to the electoral system or the applications own files.

Achieving this is generally not a problem in the prototype subsystems, though it will be discussed in conjunction with the Counting subsystem where the EML model does not provide any document to build data structures from or to double check the votes received.

2.3.4 User Interfaces

In a real life implementation, some of the subsystems might have many interfaces based on the type of user and/or type of use. The prototype subsystems, in general, only reflect the interfaces dictated by law or otherwise necessary to perform the functions of the individual subsystem.

2.4 The Election Event Client

The Election Event subsystem is in charge of defining the *election event* to be held. The subsystem has one application and that is the Election Event Client.

The Election Event Client is responsible for producing the document called ElectionEvent which defines the elections that are to be held and the contests within each of the elections. This subsystem corresponds to the box at the top of figure 1.1 on page 7. Before the functionality of this subsystem can be explained, two concepts have to be clarified: *contest* and *election event*.

2.4.1 Defining a Contest

The term *contest* has a clearly defined meaning in the EML model. However, a few things will have to be explained before this meaning can be succinctly stated.

An election is for a specific organ or body, in the Storting Election the body is the Storting, in the County Council Election the bodies are the County Councils and in the Municipal Council Election the bodies are the Municipal Councils.

In the case of a County Council Election the election results from all the municipalities in the county have to be combined into the final result for the county, and this result determines the composition of the County Council representing that county.

In the Storting Election the results from all the municipalities in a county are combined into the results for the county as a whole and this determines the distribution of the seats allocated to that county in the Storting.

In the Municipal Council Election the situation is unique in that the composition of the body the election is for, is solely decided by the results of the election in that municipality. There is no combination of results.

In this context a *contest* in a Municipal Council Election will be the municipality. In the County Council Election and in the Storting Election the *contest* will be the county, since the seats are to represent the county as a whole either in the Storting or in the County Council. In short the word *contest* defines the seats that are contested in the election.

2.4.2 Defining an Election Event

The term *election event* is in the EML model meant to signify that there is not necessarily only one election in an *election event.* In Norway there are two types of *election events*, each at a four year interval. These *election events* are interleaved such that an *election event* occurs every two years. The County Council Election and the Municipal Council Election are held at the same time in Norway, the Storting Election is held on its own. In the case of a Norwegian Municipal Council and County Council Election, there will consequently be two elections in the one *election event.*

2.4.3 The Function of the Election Event Client

The Election Event Client produces the ElectionEvent document which is the fundamental document in the electoral system. The information supplied the ElectionEvent document is listed in the section 3.4.1, which is the section covering the ElectionEvent document in chapter 3.

In an *election event* one may have several elections. Within each of the elections, there is a contest for each municipality in the Municipal Council Election and for each county in the County Council and Storting Election. What is expressed in the ElectionEvent document is the elections to be held and the contests that they contain.

The Election Event Client subsystem essentially reads a set of input files, sets up its internal data structure, produces the Election Event document and sends it to the subsystems waiting for it. The internal data structure of the Election Event Client is as shown in figure 2.2.

The ElectionEvent document is input to the Nomination subsystem, the Election List subsystem and the Voting subsystem, and is used by these subsystems to set up their internal data structures.

These subsystems will, for that reason, remain in their first stage until all their input documents are received. In the case of the Nomination subsystem and the Election List system, all they require is the Election Event document. When that is received, they move on to their second server state. The Voting system, however, also depends on the Election List (output from the Election List subsystem) and the Candidate List (output from the Nomination subsystem), and will only proceed to its server state when these have been received. This is illustrated in figure 1.1 on page 7 and figure 2.1 on page 15.

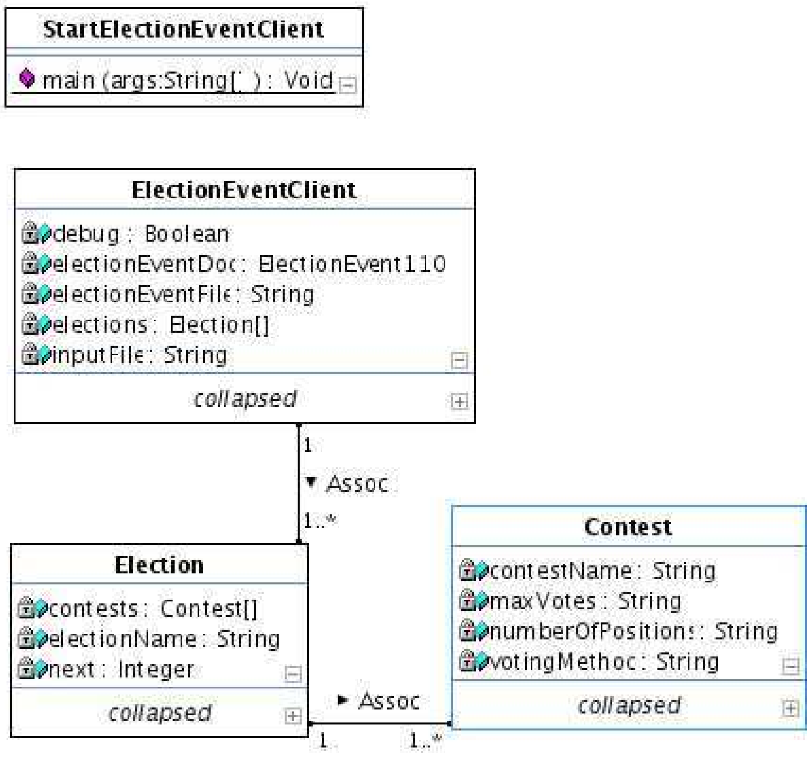

Figure 2.2: Internal data structure for the Election Event Client

2.5 The Electoral Roll

2.5.1 The Electoral Register

The Electoral Register is the Norwegian Electoral Roll. In a Norwegian context each municipality will have the Electoral Register for their municipality, the Election Law and the Election regulation [13] dictate how this register is managed by the municipality. This process is in the prototype modeled by the Election List subsystem, so called because the EML schema that defines its output is called the Election List. The Election List document is the internal representation of the Electoral Register for the Electoral System as a whole.

2.5.2 Laws and Regulations Regarding the Electoral Register

The Electoral Register and the right to vote are treated in chapter 2 of the Election Law. There are different rules that govern who can vote in the elections, but in the County Council Election and the Municipal Council Election the same rules apply. This means that a person that is eligible to vote in one is also eligible to vote in the other [16, §2-3(2)]. Consequently, there is only one Electoral Register for each *election event*.

The Electoral Committee in a municipality is in charge of the Electoral Register for that municipality [16, §2-3(1)], and the Population Registry Authority is obliged to supply information on who should be entered into the Electoral Register [16, §2-5].

The Electoral Committee is responsible for making the Electoral Register available to the public so that any errors can be discovered and corrected [16, §2-6/§2-7]. The specific instances where the Electoral Register should be corrected in the Municipal Council Election and the County Council Election are listed in the corresponding regulation [13, §1]. Paragraph 2 in this regulation concerns the period of time during which the corrections should be made and the care that should be taken that a voter is not present in two Electoral Registers at the same time.

2.5.3 Electoral Register and What it Contains

In the context of this thesis it is necessary to explore the applicable laws and regulations applying to a subsystem to find indications as to what kind of information should be stored.

For most of the subsystems this is clearly defined in the law or regulations. However, it is not clear in the law nor in the regulations what kind of information should be in the internal representation of the Electoral Register about a specific voter.

The information contained in the Population Registry is listed in §30 of the Population Registry regulation [20], and the information contained in the Electoral Register would have to be a subset of this information.

Both the Election regulation [13, §23(2)] and the Electoral Handbook [18, 4.8] deal with the exclusion of the full personal identification number from public electoral documents concerning the voters. The Electoral Handbook explicitly deals with the published versions of the Electoral Register. The above indicates that the internal representation of the Electoral Register is intended to contain personal identification numbers.

Having personal identification numbers in the internal representation of the Electoral Register will ease the assurance process of making sure that a voter is only present in one Electoral Register at any given time.

In addition to personal identification numbers it would seem desirable to store the address of the voter including the postal code. This would make it possible to send the voter information regarding the election. The voting circuit would also be useful, since this information would probably be included when sending election information to the voter.

The internal representation of the Electoral Register in the Election List prototype subsystem is discussed in the following section, and has been influenced by the concerns above.

2.5.4 The Election List Server

The Election List subsystem is responsible for the Electoral Register. This is in Norway based on the information in the Population Registry. However, each municipality is responsible for updating and maintaining the Electoral Register in the time before the election. This updating will probably be done by a few employees in the municipal administration, and since Norway does not require it citizens to register to vote, there is really no need for a client application for this server. One can, for this reason, question the "server" name, but it will be kept if only for maintaining the overall general design of the subsystems.

The Election List subsystem could, as a result, have a user interface directly, and not separate this functionality out into a client application. The user interface has not been created since it is clear that it brings no new complications to the model. Updating, adding and deleting entries are internal tasks and bring nothing new to the analysis.

This prototype subsystem is, for this reason, designed so that it reads the required initialization information from a file and from that builds the internal data structures. The Election List document is produced from these internal data structures, and sent to the Voting Server. See figure 2.3 for an overview of this subsystems internal data structure.

The information that is read from file about a specific voter is of the form:

```
<number of elections voter is entitled to vote in>
<election name (i)>
<contest name (i)>
<voter voting circuit (i)>
<voter first name(s)>
<voter last name>
<voter street address>
<voter postal code>
<voter postal location>
<voter personal identification number>
```

Note:

Middle names are not supplied, these are in the prototype subsystem grouped in first name(s). This is for convenience only.

Date of birth is not supplied, date of birth is the first 6 digits of the personal identification number and, for that reason, implied.

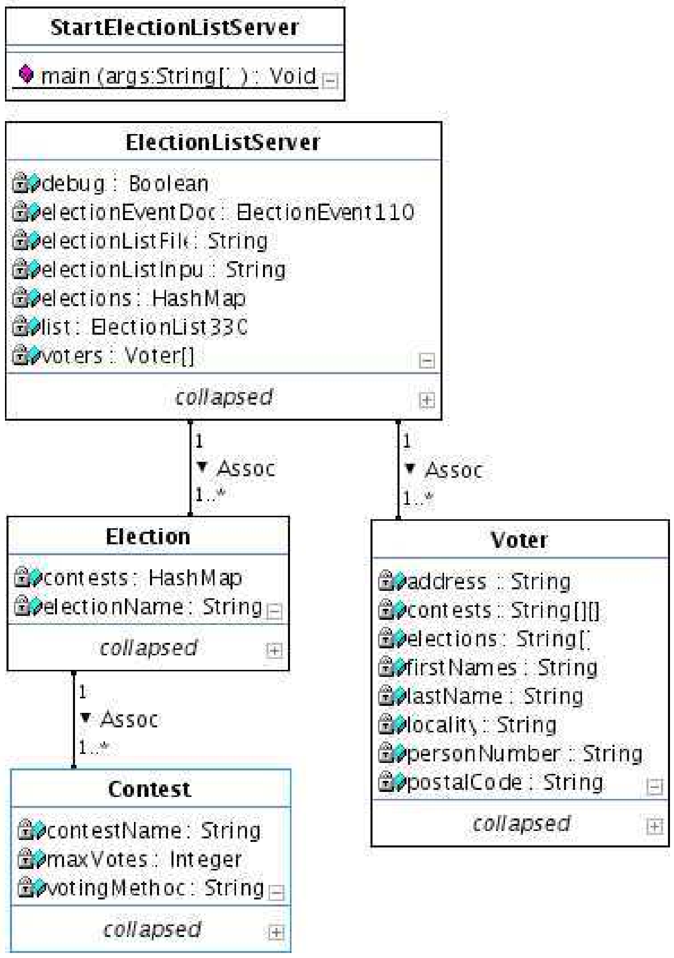

Figure 2.3: Internal data structure for the Election List Server

Sex is not supplied. The middle digit in the remaining 5 digits in the personal identification number indicates sex. Odd number being male, even being female. Sex is therefore implied as well.

The elections do not really need to be listed, since in Norway all voters entitled to vote in one election in an *election event* are entitled to vote in all. The fact that a voter is present, therefore, implies that the voter is allowed to vote in all elections in this *election event*. Nonetheless, what is modeled is a situation where a voter may not be allowed to vote in all elections in an *election event*. This will allow one to exclude a voter from an election in an *election event* without having to manage separate Electoral Registers for the elections in the *election event*.

2.5.5 The Legality of the Election List Prototype Subsystem

This subsystem introduces few, if any, complications. The subsystem would need a user interface for updating the register. Apart from that, the Norwegian process of registration in the Population Registry solves the voter registration issues.

Personal identification numbers are in the Voting and Counting subsystem used as VTokens, a token that uniquely identifies a voter. In a real life implementation this can introduce privacy and anonymity concerns.

Since most of the legislation regarding the Electoral Register deals with the updating process, there really remains only one issue: Making sure a voter is only present in one Electoral Register at any given time. This can be resolved by having the servers synchronize this process, or by keeping a central Electoral Register instead of having it distributed. Regardless of the solution chosen, this is beyond the scope of this thesis. It should be clear, based on the above, that this subsystem fulfills the requirements of the law.

2.6 The Nomination Process

The Norwegian Electoral System is based on the voter voting for a partys list of candidates rather than for singular candidates. In the pre-election process the parties submit list proposals, which are submitted to the appropriate committee for the specified election.

The registration of political party names is administrated by the Brønnøysund register and is, for that reason, not an issue for an electoral system in a Norwegian setting. It will, however, be necessary to check whether a party that submits a list is in fact registered in the Party Register. This is not implemented, though in a real world system this will have to be done, either manually or through an exchange with the appropriate database.

2.6.1 Defining a Proposer or a Signer

The Election Law demands that a list proposal has a certain number of signatures. In that respect two concepts should be made explicit, since these concepts are, in fact, equivalent: *signer* and *proposer*.

In the Election Law a person that signs a list proposal has to also supply some information that identifies him/her. Such a person will in the following be called a *signer*.

In the EML model a nomination document has associated with it a number of *proposers*. These are listed in the document with information to identify them.

If one accepts that in signing a list proposal one is endorsing it, the concept of a *signer* and a *proposer* is equivalent.

In the following the above definition will be followed, and the word *proposer* will be used to indicate the *signer* and the word *signature* to indicate the physical signature of the *proposer*.

2.6.2 The List Proposals

The form of list proposals and the processing of these are treated in chapter 6 of the Election Law. This chapter describes what a valid list proposal has to contain and the number of candidates that can be present in on the party list. In addition to this, a list proposal might need a certain number of signatures to be valid. The list proposal has to contain the following:

1. The identity of the election the list is for [16, §6-1(2a)].
2. The name of the party or group that has submitted the list [16, §6-1(2b)].
3. The candidates that are to be on the list. The information supplied about a candidate are in some cases required, in others optional:
    (a) *Names*: First and last name is required.
    (b) *Profession*: The candidates' profession can be supplied.
    (c) *Place of residence*: The candidates' place of residence can be supplied.
    (d) *Year of birth*: Is required
    (e) *Restriction of information present*: If candidates' profession and/or place of residence is supplied, it has to be supplied for all candidates on the list [13, §17(1)].
    This information is also required if it is necessary to avoid ambiguity within the list itself [16, §6-1(2c)].
    (f) *Representing party/group*: If the list is a collaborative list where several groups or parties have one common list, the affiliation of the candidates may also be supplied [13, §17(2)].
    (g) *Increased share of the poll*: In a Municipal Council Election a certain number of candidates may be given an increased share of the poll, which means that these candidates will be given a additional vote count that varies with the votes for the list. The number of candidates that may be given this increased share varies with the number of representatives in the corresponding Municipal Council [16, §6-2(3)].
4. *Number of candidates*: The number of candidates on the list is subject to paragraph 6-2, and varies with the type of election and the number of representatives in the corresponding body for which the election is for.

5. *Proposers*: The required number of signatures [16, §6-1(2d)]. The number of signatures necessary is specified in paragraph 6-3 and varies with the partys results in the last election and the type of election that will be held. Paragraph 12 in the Election regulation [13] specifies that the signatures must be written on paper.

6. The name of a representative and an alternate, which should both be among the proposers [16, §6-1(2e)].

7. Enclosed there has to be a document adhering to the following [16, §6-1(3)]: A list of the candidates date of birth, a list of the date of birth and address of the proposers and possibly a declaration that the eligibility of a candidate will be fulfilled [16, §6-4].

It is also important to note that the list proposal may not contain any other type of information for the voter but what is explicitly mentioned in paragraph 6-1 [16, §6-1(4)].

2.6.3 Personal Identification Number as a Unique Identifier

It should be clear that the form of these lists is tightly governed by the aforementioned rules. However, the enclosed document only contains the date of birth of a candidate. Now, since a candidate may only appear on one list in a contest [16, §6-1(2c)], it is necessary to check that two lists do not list the same candidate. This will be easier to check if they are listed with their personal identification numbers in the enclosed document.

There is no reason to believe that name and date of birth is enough to distinguish two people. This ambiguity can easily be eliminated by using the full personal identification number.

Since the eligibility of a candidate has to be checked to be in accordance with Chapter 3 paragraph 3-1 for the Storting Election and paragraph 3-3 for the Municipal Council Election and the County Council Election, and assuming the internal representation of the Electoral Register has full personal identification number, this check will also be made easier if one also here uses the full personal identification number.

2.6.4 The Nomination Server

The main function of the Nomination server is to receive list proposals. There are two EML documents that can be used to facilitate communication between a central Nomination Server and a Nomination Client application. These documents are Nomination and NominationResponse, where the document Nomination represents a list proposal including the *enclosed document*. This exchange, receiving a list proposal and responding to it, is the Nomination Servers second stage. This stage can be omitted and replaced with reading the lists from files if the mentioned exchange is not meant to be measured or modeled. Both of these approaches have been implemented in this prototype subsystem. The client/server approach is depicted in figure 2.4 and shows the exchange with the Nomination Client.

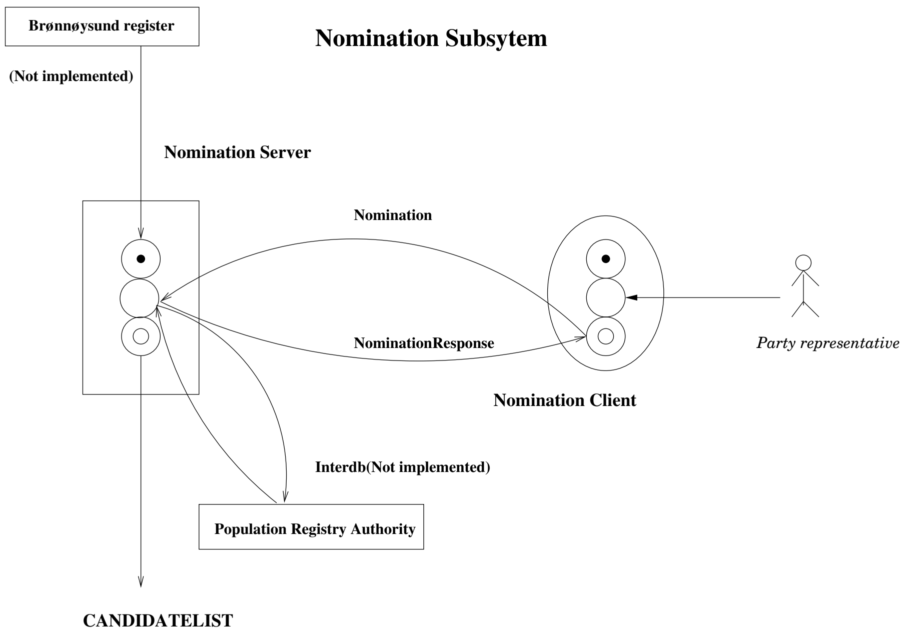

Figure 2.4: Nomination Subsystem state diagram

Receiving List Proposals and their Enclosed Documents

The nomination document sent from the Nomination Client to the Nomination Server, represents a list proposal. In the Election Law the list proposal is associated with a separate *enclosed document* that contains additional information. This information can be seen as internal information as opposed to the public information in the list proposal. The list proposal, if approved, will become the partys party list and the information it includes public.

In the prototype subsystem there is no separation between the list proposal and the *enclosed document.* Since the information that can be expressed in the *enclosed document* does not include any information that is optional in the list proposal, there is no real loss of separation here.

Producing the Candidate List

The Nomination Server builds its primary data structures when it receives the ElectionEvent document. The list proposal data structures are built either from an exchange with clients or by reading the list proposals from files. The server application then produces a Candidate List document, which actually in a Norwegian context is a list of approved party lists, and outputs this document to the Voting subsystem. The internal data structure of the Nomination Server is shown in figure 2.5.

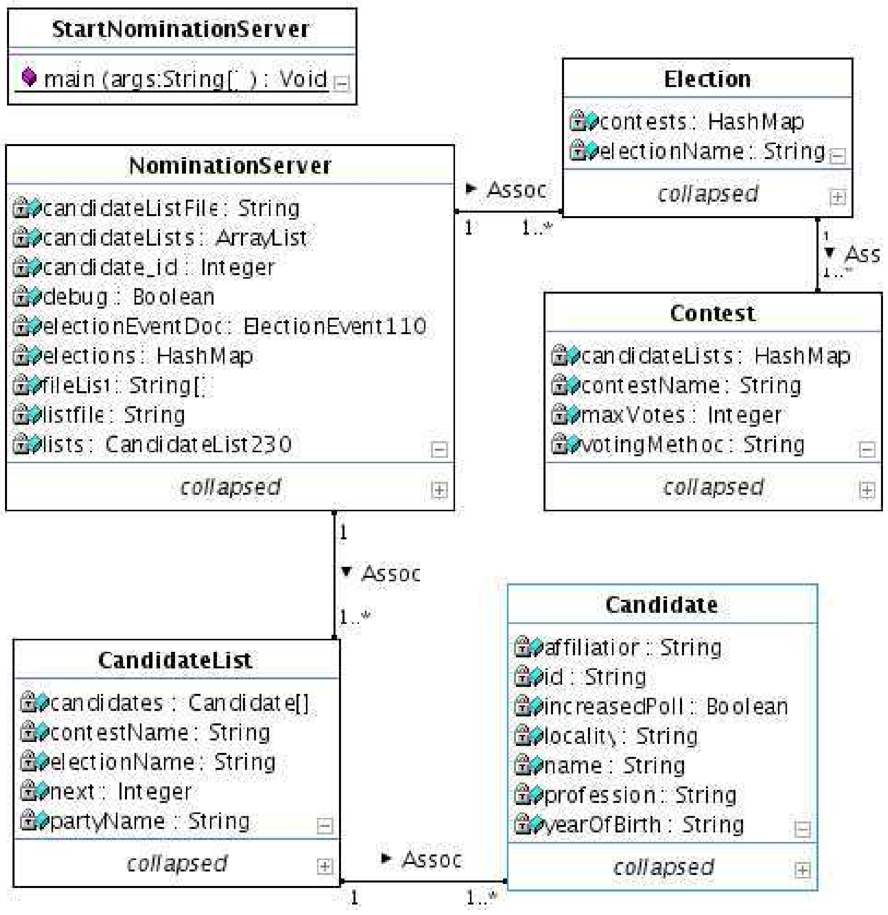

Figure 2.5: Internal data structure for the Nomination Server

2.6.5 The Nomination Client

The Nomination Client application tests the exchange of Nomination documents and NominationResponse documents. It does not bring anything really new to the process, but eases the testing of the subsystems in a simulation by having a user interface to manipulate the list proposals at will. Its internal structure is shown in figure 2.6.

The list proposals are read from file and can be edited in the user interface by adding list proposals, adding a candidate to a list proposal, removing

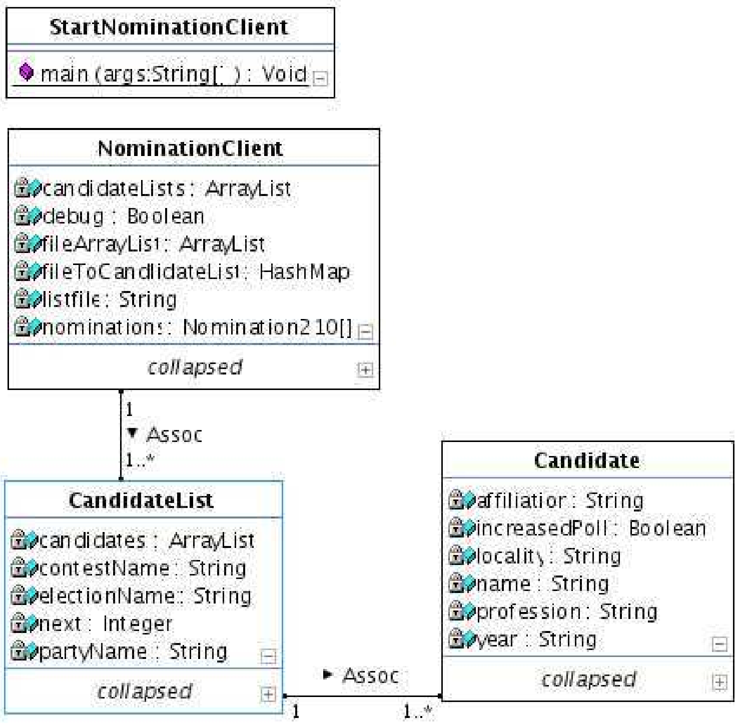

Figure 2.6: Internal data structure for the Nomination Client

a candidate from a list proposal and giving a candidate an increased share of the poll. The list proposal in the implementation of the Nomination Client contains the following:

<election name>
<contest name>
<party name>
<candidate (i) name>
<candidate (i) year of birth>
<candidate (i) profession>
<candidate (i) place of residence>
<candidate (i) increased share of the poll (true/false)>
<candidate (i) representing party/group (common lists)>

Note:

Profession: This field is optional.

Place of residence: This field is optional.

Increased share of poll: Indicates whether this candidate should be given an increased share of the poll. This field is either “true” or “false”.

Representing party/group: The party or group this candidate represents. To be used if the candidate is on a collaborating list. This field is optional.

2.6.6 The Legality of the Nomination Prototype Subsystem

Evaluating the legality of this subsystem can be reduced to whether the Nomination document can express the information that a list proposal and the *enclosed document* are required to contain. Considering the list on page 23, these requirements will have to be met by the Nomination document for this subsystem to adequately model the Norwegian nomination process.

The list below mirror the list on page 23 and will discuss the different pieces of information separately.

1. *Election identity*: Met by the field election name and contest name.
2. *Party name*: Met by the field party name.
3. *Candidates*:
   (a) *Names*: First and last name are not separated in this prototype subsystem, but both are present in the field candidate name.
   (b) *Profession*: Present
   (c) *Place of residence*: Present
   (d) *Year of birth*: Present
   (e) *Restriction of information present*: The check of whether a piece of information is present for all candidates is internal to the Nomination Server and, for that reason, not relevant for this thesis.
   (f) *Representing party/group*: Present
   (g) *Increased share of the poll*: EML does not cater to the concept of increased share of the poll. However, a workaround is possible (and has been implemented) by using a field that is meant to indicate that a candidate is independent. This field is then interpreted to indicate that the candidate should receive an increased share of the poll. This is further explained in section 3.5.1 in chapter 3 that discusses the Nomination document. This workaround would have to be known in the other relevant subsystems: Voting Client in the Voting subsystem and the Counting Server in the Counting subsystem. This is, of course, not an optimal solution and will also be treated again in the final concluding chapter.

The number of candidates that can be given an increased share of the poll is a function of the number of representatives to the Municipal Council being elected in the contest. This number is available to the subsystem, since it is supplied by the ElectionEvent document.

4. *Number of candidates*: The limit on the number of candidates that can be on the list, must be checked internally by the Nomination Server and is, therefore, not relevant for the purpose of this thesis.

5. *Proposers*: The nomination document has proposer fields that can be used to list proposers. This structure is not used, since the election regulation [13, §12] explicitly states that the signatures must be written on paper. The proposer element does open for digital signatures using the form[1]:

```
<complexType name="SignatureType">
  <sequence>
     <element ref="ds:SignedInfo"/>
     <element ref="ds:SignatureValue"/>
     <element ref="ds:KeyInfo" minOccurs="0"/>
     <element ref="ds:Object" minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="Id" type="ID" use="optional"/>
</complexType>
```

   If so desired, the Object field could here be used to store a scanned representation of a signature on paper. However, since it is clear that the Election regulation [13] requires the signatures to be on paper, an electronic transfer of these is beyond the scope of this thesis. For this reason the proposer mechanism of the Nomination document is not used. This shortcoming will also be addressed in the concluding chapter.

6. Name of a representative and an alternate, can be placed in the document. The check to see if they are in the signature list can be performed manually, if the signature list has been sent in paper form. If the electronic signature implementation above is used, this check could be done electronically. Since this touches on the above discussion on signatures, it has not been implemented.

7. The nomination document provides the fields for giving candidate date of birth, and the ProposerStructure has fields for giving addresses for proposers. However, date of birth of proposers is not provided for. The declaration of eligibility is also not provided for.

[1]this can be found at http://www.w3.org/TR/xmldsig-core/

To sum up, it is clear from the above that EML does not inherently suit this process. Specifically, the proposer requirement of the law is not fulfilled. This has, in any case, no bearing on the rest of the electoral system as long as the party lists can be sufficiently expressed. Since the information that could not be expressed is information that would be expressed in the *enclosed document* this information is internal to the Nomination subsystem and does not affect the rest of the Electoral System.

It should be clear that it is possible to isolate this deficiency to the Nomination subsystem. This will ensure that the other subsystems will not be affected, with the exception of the less than perfect solution to the "increased share of the poll" problem (see 3(g) on the list on page 23).

2.7 The Casting of Votes

The Voting subsystem deals with the casting of votes. Since votes are the central issue for this subsystem the concept of a *ballot* has to be clarified before the laws and regulations governing this subsystem are explored.

2.7.1 Defining a Ballot

A ballot in the context of the EML model, and for that reason the prototype subsystems, presents the choices available to a voter in a contest. The ballot structure, which is used in the AuthenticationResponse and Ballots document, is defined in the XML Schema 340-410-430-include-v4-0.xsd. Although this schema allows more than one election to be present in a ballot, the prototype makes one ballot for each contest in the election(s). The ballot contains all the party lists for a given contest, meaning that there is not a separate ballot for each party list in the contest.

It is important to be explicitly clear on this, since the law and regulations view a ballot as only representing one party list, and a contest will, in that context, have one ballot for each party putting forth a list in the contest.

In this section the interpretation of the law will be followed, in chapter 3 the EML interpretation will be used.

2.7.2 Voter Changes to the Ballot

The Norwegian voting process is today a manual system, where a voter physically casts physical ballots. A ballot in this context, is the party selection made by the voter in a contest. In such a manual *election event* where there is more than one election, there may be several physical votes cast, each containing one ballot.

Chapter 7 of the Election Law lists the voters rights in changing a ballot. This is described according to the elections in which the changes may be done. The same distinction will be made here:

- *Storting Election*: The voter is here entitled to change the order of the candidates on the ballot by putting a number next to the name. The voter can also strike one or more candidates from the ballot [16, §7-2(1)].
- *Municipal Council Election*: The voter can give a candidate a person vote by placing a mark by their name on the ballot [16, §7-2(2)]. The voter can also give candidates on other party lists a person vote by putting their names on the ballot. The number of such additions can not exceed a fourth of the number of members of the Municipal Council, still one can always list at least five [16, §7-2(3)].
- *County Council Election*: The voter can give a candidate a person vote by placing a mark by their name on the list [16, §7-2(2)].

Other changes to the ballot will be disregarded [16, §7-2(4)].

2.7.3 The Ballot Presented to the Voter

The form and content of the ballot presented to the voter has to be in compliance with the Election regulation [13, §19(6)c, §19(9)]. This paragraph lists the information that a ballot can express with regard to a candidate on a party list. The information allowed is:

The candidates first and last name.

The candidates year of birth.

The candidates profession.

The candidates place of residence.

Whether the candidate will receive an increased share of the poll.

The party/group the candidate represents [15, §19(6)c].

All of the above information has to be available to the voter when voting.

2.7.4 Candidate Ids as a Unique Identifier

Using the a candidates name when adding the candidate to the ballot, again brings up the discussion of using a name as though it were a unique identifier. A name is, of course, not a unique identifier; many people have the same name. In addition, text comparison in itself is prone to failure, simply because people have a tendency to misspell.

Providing each candidate with a unique identifier will easily resolve these ambiguities, and in the prototype this subsystem does, in fact, implement this.

2.7.5 The Voting Server

The Voting Server is a part of the central subsystem of the electoral system, the Voting subsystem. The information from the pre-election subsystems are input to it and the votes cast are its output. This interaction and the exchange with a voting client is depicted in the figure 2.7.

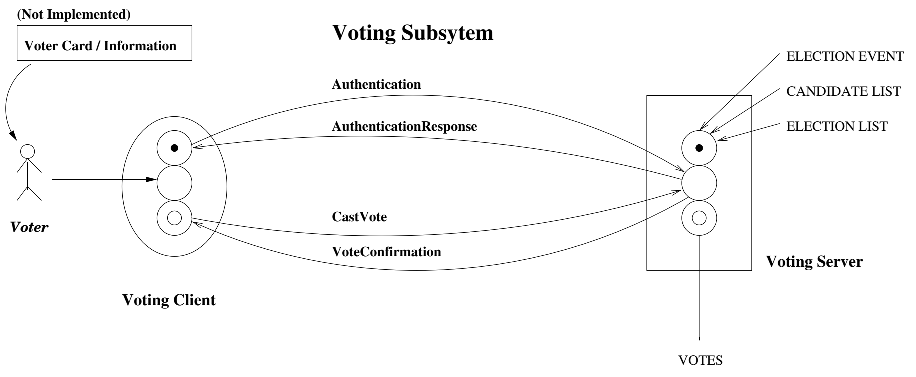

Figure 2.7: Voting Subsystem state diagram

EML does not dictate the form of the voting action, whether it be manual (as is done in Norway today) or electronic (over the Internet or by mobile phone etc). The prototype of this subsystem will, however, use electronic voting.

The Voting Server follows, as seen in figure 2.7, the three state model described earlier. It starts off waiting to receive the Election List, Candidate List and Election Event documents. Once these are received it moves on to a server state where it responds to authentications and cast votes. When the election is over, the Voting Servers server state ends, and it moves on to its final output state where it sends the votes received on to the Counting Server. The internal data structure used by the Voting Server is depicted in figure 2.8.

2.7.6 The Voting Client

The Voting Clients most important focus in terms of this thesis is whether it can present to the voter the manipulation options required by the Election Law. The Voting Client works closely with the Voting Server in an exchange of four messages constituting the authentication of the voter and the casting of the vote. This interaction is depicted in the figure 2.7 on page 32. This process seen from the Voting Client is briefly described below.

The messages sent between the Voting Client and Server are really at the center of this process and are described individually in the next chapter. The implementation of the changing of the ballot mechanism is discussed in

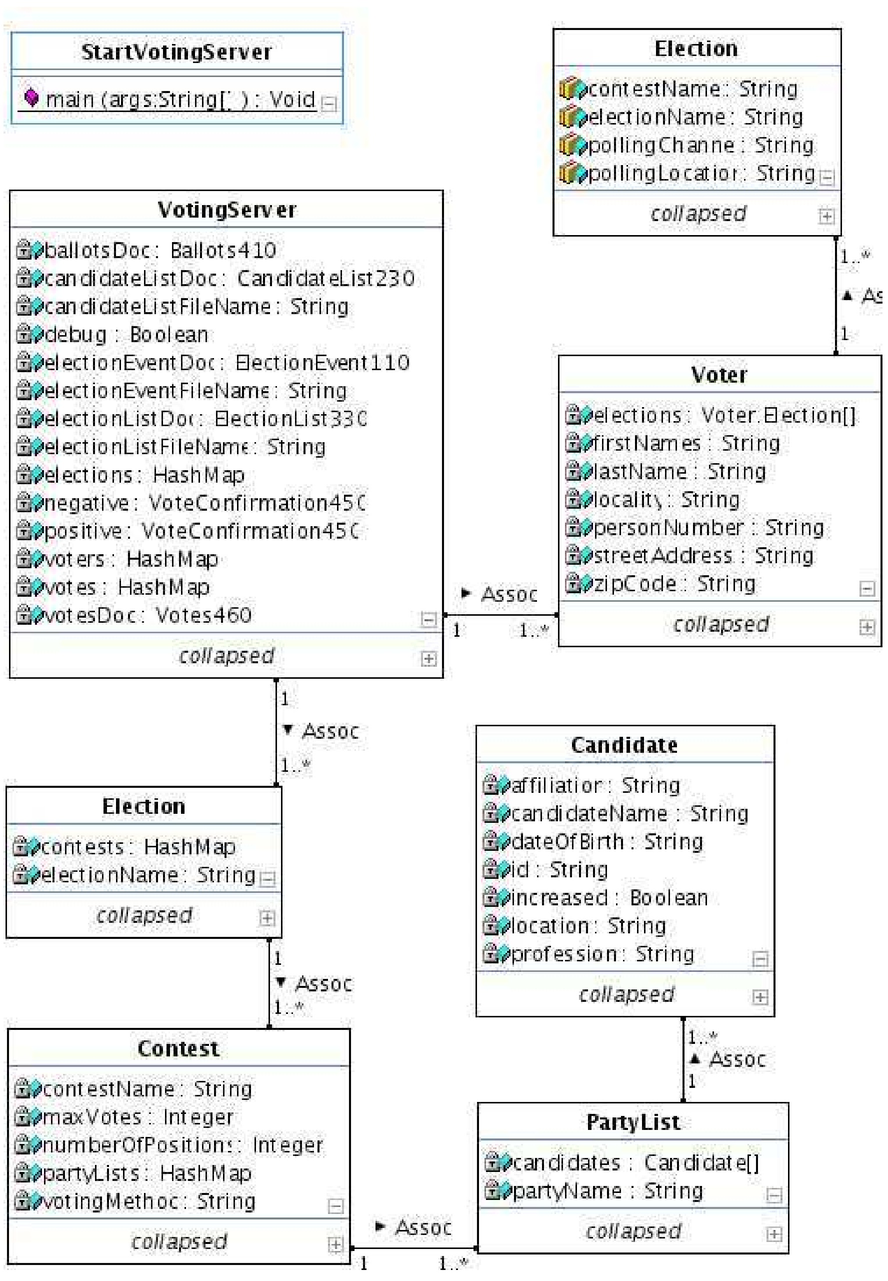

Figure 2.8: The Voting Server Java package - class diagram

section 2.7.7.

The Voting Process in the Voting Client

The client is implemented as basically one loop, the login loop. When a voter wants to vote, he/she has to login using his/her unique personal identification number. Using this number an authentication message is sent to the voting server and an authentication response is returned. This response will be negative if the voter is not in the Election List. If, however, the voter is found in the Election List, the response includes the ballots for the elections in which the voter is allowed to vote. Associated with each contest in a ballot are the party lists for the contest.

This information is then used to create the internal data structure from which the user interface is dynamically created. Subsequently, the voter makes his/her selections and finally chooses to submit the vote. A Cast Vote message is constructed from the choices made, and is sent to the server. The server responds with a Vote Confirmation message which is either positive or negative and the result is displayed to the user. The client application clears most of its internal data structures and is ready for a new voter. The internal data structure of the Voting Client is seen in figure 2.9.

2.7.7 The Legality of the Voting Prototype Subsystem

Evaluating the legality of this subsystem can be reduced to whether the Voting Client can present the required ballot and express the voter changes to the ballot. The voter changes are treated first followed by the presenting of the ballot.

Changing the Ballot

These changes allowed to the ballot were above divided into the elections that allow them, the same distinction will be made here. Note that the classes mentioned are all in the VotingClient package, a class diagram of this package is presented in figure 2.9 on page 35:

- *Storting Election - Implementation*: A party list is represented as an object of the class PartyList. When a list is chosen, the array containing the candidates is copied to an ArrayList in the Ballot class called candidateArray. These are the operations done to achieve the lawful changes:

  Strike one or more candidates from the list:

  ```
  candidateArray.remove(<index of Cand.>);
  ```

  Change the order of the candidates on the list:

  ```
  Candidate c =
  (Candidate) candidateArray.remove(<index of Cand.>);
  candidateArray.add(<New placement>, c);
  ```

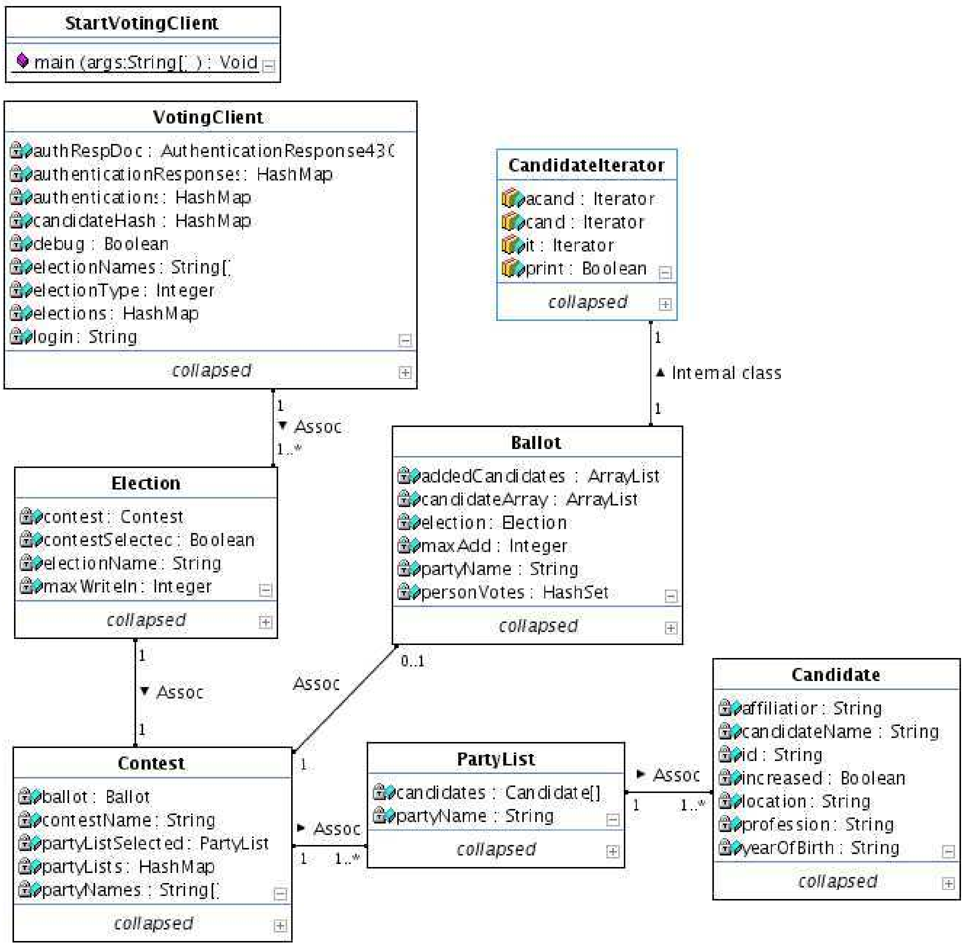

Figure 2.9: The Voting Client java package - class diagram

Expressing the changes in the CastVote document: Candidates are entered into the document in the order they are listed in the ArrayList candidateArray, and only the candidates present in the ArrayList are entered. This will express both removed candidates and a change of order.

- *Municipal Council Election - Implementation*: The candidates from other party lists, that are given person votes, are put into a separate ArrayList called addedCandidates. Person votes for candidates both on the list and on other lists, are represented by a HashSet personVotes. This will assure that a voter cannot give several person votes to the same candidate on the list. It is for this reason also necessary to check that a candidate has not already been added to the addedCandidates before adding the candidate.

Give a candidate a person vote - same list

```
personVotes.add(<Candidate Id>);
```

Give a candidate a person vote - other list

```
Candidate cand = getCandidate(<Candidate id>);
addedCandidates.add(cand);
personVotes.add(cand.getId());
```

Expressing the changes in the CastVote document: Candidates in candidateArray are first inserted into the CastVote document, followed by the candidates in addedCandidates. For each candidate the following is tested:

```
personVotes.contains(candidateNum);
```

If this is test is true, the candidate is marked as having a person vote. It should be clear that this expresses person votes for candidates both on the list and on other lists.

- *County Council Election - Implementation*: This is solved in the same manner as giving a person vote to a candidate on the same list - see "Municipal Council Election - Implementation" above.

For this subsystem to be valid with regard to the law the changes described above will have to be expressed in the CastVote document. The form of the CastVote document and the fields used are elaborated in the section concerning this document in chapter 3 section 3.7.4 on page 64.

Presenting the Ballot

The requirements to the ballot when presented to the voter, are listed in section 2.7.3 on page 31. For this application to fulfill its obligation in presenting a lawful ballot, this information has to be presented. In section 3.7.3 the AuthenticationResponse document is discussed. For the above information to be available to the Voting Client, this document must present it. Displaying information is a simple matter if the information is available.

The internal implementation of the Voting Client should, by the above, assure that the Voting Client application internally fulfills the law. This fulfillment depends on the information that can be expressed in the CastVote and AuthenticationResponse documents.

2.8 The Counting of the Votes

Since any documents used in the Counting subsystem are internal, the counting of the votes is really outside the scope of this thesis. For this reason, this subsystem is not complete with regard to the law. The Election Law describes two countings, a provisional and final count, and these are discussed below. What was implemented in this subsystem was implemented to test the inclusion of the voter corrections in the final count.

The counting algorithms in the different elections is briefly described. The Counting Server is then presented and its conformance to the law discussed.

2.8.1 Provisional/Final Count

Chapter 10 of the Election Law paragraph 10-5 and 10-6 dictate that there has to be two separate countings, the provisional count and the final count. The provisional count will be done by only counting the list votes, that is the vote for the list, not taking into account the voter changes [16, §10-5(5)]. The final count will also take into consideration the voter changes [16, §10-6(3&4)].

The Election regulation [14] explicitly states that a modification of the ballot will only be taken into account if it is done by more than half of the voters that voted for the list in both County Council Elections and Storting Elections [13, §7(2), §8(2)]. An electronic counting mechanism will make it possible to take into consideration all changes made, since such a mechanism can process a greater amount of information than any manual system within a reasonable time frame. This would provide for more voter input into the electoral process.

2.8.2 Counting Algorithms

The counting algorithms described below are from the Electoral Handbook, where the counting mechanism for each type of election is treated separately. Section 18 of the Electoral Handbook deals with the counting in a Stortings Election, section 19 with a County Council Election and section 20 with a Municipal Council Election.

The counting algorithms can be divided into two groups, seat distribution and seat allocation:

- *Seat distribution*: The result of the seat distribution count will decide how many seats each party gets in the corresponding body the election is for.
- *Seat allocation*: The result of the seat allocation count will decide which candidates from the partys list will fill the seats won by the party.

Only in the Municipal Council Election does the voter changes affect the seat distribution count. The counts described below are only counts that are affected by the voter changes.

Counting Voter Changes in a Storting Election

In a Storting election the voter can make two types of changes. The voter can change the candidate order and/or strike candidates from the list.

The counting that results in the awarding of seats to a party is based on the number of votes for the list. After the number of seats awarded to the party is known, the candidates that are to fill the seats will have to be determined. The seat allocation algorithm is based on the changes made by the voters:

- *Seat allocation*: The candidate that the most voters placed as number 1 will receive the first seat. The candidate that has the greatest sum when adding the number of voter placements of either number 1 or 2 (excluding the candidate that received the first seat) receives the second seat. This continues until all seats awarded the party are filled. If two candidates have the same sum the order on the party list takes precedence.

Counting Voter Changes in a County Council Election

In the County Council Election the voter can give person votes to the candidates on the list. This is taken into consideration in filling the seats awarded to the party. All candidates that have a total of person votes that exceeds 8% of the total votes for the list are to be allocated seats first. The algorithm for this allocation is as follows:

- *Seat allocation*: The candidate that received the most person votes (above the 8% limit) is awarded the first seat. The second seat goes to the candidate that had the second most person votes. This goes on until all seats are filled or no more candidates have person votes over the 8% limit. If there are more seats, then the remaining seats are allocated to the candidates on the list in the order they are listed on the official list, skipping candidates already allocated a seat.

Counting Voter Changes in a Municipal Council Election

In a Municipal Council Election the voter can give person votes to candidates on the same list, much like the County Council Election. However, they can also give person votes to candidates on other lists in the same contest. This last voter change affects the number of seats awarded the party, which is not the case for the voter changes in the Storting Election and County Council Election.

- *Seat distribution*: The *list number* is used when distributing the seats. This number is computed as follows:
  V = Number of votes for the list
  N = Number of representatives in the Municipal Council
  R = Number of person votes received from voters voting for other lists
  L = Number of person votes lost by voters voting for this list person voting for candidates on other lists

  List number = (V * N) + R - L

  This number is used to distribute the seats among the parties.

- *Seat allocation*: The allocation of seats among the candidates on the list is done in the following manner:

  1. Candidates that were listed to receive an increased share of the poll receives this now, which is equivalent to 25% of the total number of votes for the list.

  2. The person votes for each candidate is calculated, both from the partys own voters or from voters voting for other lists.

  The candidate that receives the most person votes receives the first seat, the candidate that received the second most person votes receives the second. This is continued until all seats are allocated. If two candidates have the same number of person votes the order on the official list takes precedence.

2.8.3 The Counting Server

The Counting Server is in charge of counting the votes. The votes are received in the form of the Votes document sent by the Voting Server. Although the counting process is really not an issue for this thesis, it is necessary to explore whether it can be done based on the information supplied to the Counting Server. The internal data structure of this subsystem is depicted in figure 2.10.

Input Documents to the Counting Server

In figure 1.1 the counting subsystem is shown as only receiving the Votes document. The Voting Server prototype subsystem, however, also provides the CandidateList document. This follows cleanly from the principals adhered to in the other subsystems, where the internal data structure is created based on internal documents and the external input is used to update this data structure.

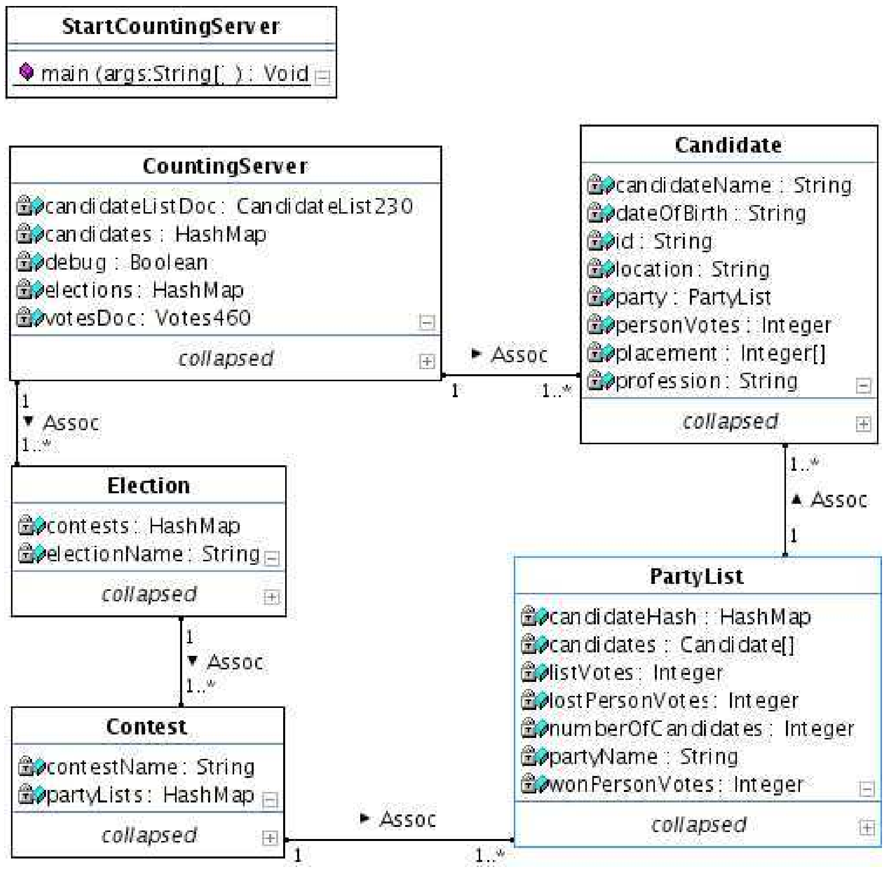

Figure 2.10: Internal data structure of the Counting Server

If security was an issue, one would also make the VTokenLog (described on page 68) available to the Counting Server. These two documents, CandidateList and VTokenLog, could then be used to check the votes to make sure they are from legitimate voters and list legitimate candidates. VTokenLog is, however, not used in the prototype, since this is beyond the scope of the thesis.

The *NumberOfPositions* field present in the Count document, described in section 3.8.1 in chapter 3, seems to indicate that the ElectionEvent document is also necessary for this subsystem. This information would otherwise be unavailable to the Counting subsystem.

2.8.4 The Legality of the Counting Prototype Subsystem

Since this subsystem is the end of the line in terms of the election process, evaluating its legality is not as critical in terms of this thesis. The point that is relevant is whether the changes that the voter could make in the Voting Client can be recognized here. The counting process for the different elections is described in the Election Manual [18], and the discussion below is based on that description. The counting processes described below will only be those that deal with the voter changes. These will be divided into the elections in which the changes can be made:

- *Storting Election - Implementation*:

  Each candidate has a int array called *placement*. For each ballot examined the placement of a candidate is noted and the corresponding array location in *placement* is incremented:

  ```
  void registerPlacement(int index){
    placement[index]++;
  }
  ```

  After all ballots have been examined the final order is found by the method below (found in the class PartyList.java in the CountingServer package):

  ```
  Candidate [] candidatesInOrder(){

    Candidate [] order  = new Candidate[candidates.length];
    boolean   [] mask   = new boolean[candidates.length];
    int       [] result = new int[candidates.length];

    for(int i = 0; i < candidates.length; i++){

      //add the previous results to the new
      for(int c = 0; c < candidates.length; c++){
        if(!mask[c]){
          result[c] += candidates[c].getResult(i);
        }
      }

      int index = 0;
      while(mask[index]){ //the first allowable value
        index++;
      }
      int max = result[index];

      for(int j = index+1; j < result.length; j++){
        if(!mask[j]){
          if(result[j] > max){
            max = result[j];
            index = j;
          }else if(result[j] == max){
            /* if alike the orig. order takes precedence
            this holds by default since:
  ```

```
                */
                assert (index < j);
            }
        }
    }

    assert !mask[index];
    order[i] = candidates[index];
    mask[index] = true;
  }

  return order;
}
```

If the result of two candidates is equal, the original placement takes precedence. This takes care of both a change of order and of strikings since striked candidates will not have a placement on the ballot.

- *Municipal Council Election - Implementation*:

  The list of candidates is split into two lists, one with the candidates that were on the original list and one with the added candidates (candidates from other lists receiving a person vote).

  - *Candidates on the party list*: For each candidate on the original list, if the candidate received a person vote, the candidates *personVotes* variable is incremented.
  - *Candidates on other party lists*: For each of the added candidates the candidates *personVotes* variable is incremented and the party lists *lostPersonVotes* variable is incremented. The candidates party lists *wonPersonVotes* is also incremented.

  The person votes received by each candidate will determine the allocation of the seats won by the party.

  The *lostPersonVotes* and *wonPersonVotes* are used in the distribution of seats among the parties.

- *County Council Election - Implementation*:

  In the County Council Election the voter is only entitled to person vote for candidates on the list chosen. The processing of personVotes for candidates on the party list is done in the same manner as the Municipal Council Election implementation.

Chapter 3

EML Interface

3.1 Introduction to EML Interface

The subsystems described in the previous chapter, model the Norwegian Electoral System. EML provides the language used when these subsystems and the applications they consist of, communicate. The subsystems need a practical way to utilize the EML language, this is what EML Interface provides.

The EML_Interface package provides both a practical layer of abstraction from the XMLSchemas themselves and the analysis criteria when evaluating the communication between the subsystems. The layer of abstraction provided serves two important purposes; it leaves the subsystems unaware of the complexities of maintaining validity, and it provides an interface to a standard that is constantly changing. It provides an analysis criteria by unambiguously defining the data placed into an EML document, since modification of the document can only be done by the public methods in the interface.

This section concerns itself with the creation of the EML Interface. First it will detail the purposes of the interface, subsequently describing the implementation of it. This will involve describing the compilation of EML by a binding compiler, and the actual implementation of the EML_Interface package and the use of it as a library package.

The rest of this chapter is dedicated to the XML schemas in EML 4.0 and the wrapper classes in EML Interface that represent them. Each of the EML documents will be discussed under the header of the subsystem that produces it. They will be described in terms of the information that is put into them, thus defining the information exchanged using them. Any shortcomings in the schemas, with regard to the Norwegian Electoral System, will be discussed in conjunction with the relevant schema and wrapper class.

In the final chapter of this thesis the aforementioned shortcomings will be further discussed, with suggestions on changes to the EML language necessary to accommodate the Norwegian Electoral System.

3.1.1 EML Interface Maintaining Validity

The EML schemas were compiled using a binding compiler. This process is further discussed later in section 3.2.2. What is important to note is that the classes produced through this compilation do not maintain validity. Meaning, one can create documents that are not valid.

EML_Interface was written primarily to provide a second layer of abstraction, in addition to the abstraction provided by the compiled schemas. The purpose of this layer was primarily to provide the guarantee that the documents would always be valid. In this way it was possible to leave the applications unaware of the complications of maintaining validity.

This layered implementation is an established model in system design. Each layer is supposed to only rely on the layer below and provide services to the layer above. Each layer usually has a guarantee associated with it. In this manner the quality of service should increase with each layer.

3.1.2 A Stable Interface to a Changing Standard

The EML standard provides no backward compatibility guarantees, meaning that a new EML version may differ substantially from the previous version and a document valid under a previous EML version may no longer be valid.

EML Interface can minimize the changes necessary in the subsystems when changing to a new EML version, by having most, if not all, changes only apply to the wrapper classes in it.

This change of standard has already been felt by the EML Interface. It was originally written to encapsulate EML 3.0. When it was found that EML 3.0 was unsuitable, the focus shifted to EML 4.0. It was then apparent that these two versions were not compatible and EML Interface had to be rewritten.

There is no reason to believe that EML 5.0 will be backwards compatible with EML 4.0. In light of this, EML Interface could limit the reprogramming effort mostly, or entirely, to EML Interface, keeping the subsystems unaffected.

3.1.3 Inserting Methods Defining Content

The information exchanged between two applications in the prototype is done using an EML document created by a wrapper class. The description of each such document is based on a simple idea. The information that can be exchanged using an EML document created by a wrapper class, is solely defined by the parameters to the *inserting methods* in the class.

The expression *inserting method* is meant to signify a method that will cause something to be added to the underlying document. Since the methods have to maintain validity, the constructors are always *inserting methods*. They will insert the minimum of information necessary for the document to be valid and still serve the purposes for which it is used.

Each EML document will therefore be discussed in terms of these methods and their parameters.

3.1.4 Table over the EML Schemas

The 33 XML Schemas that define EML 4.0 are listed in table 3.1 on page 46. Together with these are the schemas in the *external* directory, these schemas are not treated separately in this chapter since they are not strictly a part of EML. The table shows where in this chapter the schemas are described, it also indicates whether a schema is used. Some schemas are marked as *Import*, this is to indicate that they do not constitute a message in EML, but are rather imported by other schemas. Imported schemas are not discussed since they do not constitute documents in EML, but rather function as libraries for schemas sharing common definitions.

3.2 Creating EML Interface

EML as a language is defined in 33 XMLSchemas. In addition to these there are 5 XMLSchemas in the directory *external*, which are separated from the language itself, and can be substituted with country specific schemas. All together these schemas span 5606 lines of XML Schema code. Supporting the entire standard is a formidable task. There are today several different ways of dealing with such a language. The manner chosen for this thesis is using a binding compiler. This process is briefly described in section 3.2.2.

The EML Interface is a collection of wrapper classes for EML documents and a set of input/output utility classes. The wrapper classes are discussed individually throughout this chapter. The utility classes are treated separately in appendix E. These classes deal primarily with network communication, which is outside the scope of this thesis.

Where table 3.1 focused on the schemas, this section will view EML as a collection of wrapper classes. After the classes have been briefly introduced, the building of the layered architecture will be presented chronologically. This layering can be viewed in figure 1.2 on page 10 and the description below will begin will the lower layers.

Finally, before each schema is discussed, some issues regarding the implementation will be presented.

3.2.1 The Classes Constituting the EML Interface

The wrapper classes present in the interface correspond to the schemas marked as “Used” in table 3.1. In addition to these, there exists wrapper

Schema Name	Used	Page
110-electionevent-v4-0.xsd	Used	50
120-310-330-include-v4-0.xsd	Import	-
120-interdb-v4-0.xsd	Not Used	50
130-480-include-v4-0.xsd	Import	-
130-response-v4-0.xsd	Not Used	50
210-nomination-v4-0.xsd	Used	52
220-nominationresponse-v4-0.xsd	Used	54
230-candidatelist-v4-0.xsd	Used	55
310-voterregistration-v4-0.xsd	Not Used	57
330-electionlist-v4-0.xsd	Used	58
340-410-430-include-v4-0.xsd	Import	-
340-pollinginformation-v4-0.xsd	Not Used	57
350a-outgoinggeneric-v4-0.xsd	Not Used	57
350b-incominggeneric-v4-0.xsd	Not Used	57
350c-internalgeneric-v4-0.xsd	Not Used	57
360a-outgoingchanneloptions-v4-0.xsd	Not Used	58
360b-incomingchanneloptions-v4-0.xsd	Not Used	58
410-ballots-v4-0.xsd	Used	67
420-authentication-v4-0.xsd	Used	60
430-authenticationresponse-v4-0.xsd	Used	61
440-460-include-v4-0.xsd	Import	-
440-castvote-v4-0.xsd	Used	64
445-retrievevote-v4-0.xsd	Not Used	68
450-voteconfirmation-v4-0.xsd	Used	66
460-votes-v4-0.xsd	Used	66
470-vtokenlog-v4-0.xsd	Not Used	68
480-auditlog-v4-0.xsd	Not Used	68
510-count-v4-0.xsd	Not Used	69
520-result-v4-0.xsd	Not Used	71
610-optionsnomination-v4-0.xsd	Not Used	50
620-optionsnominationresponse-v4-0.xsd	Not Used	50
630-optionslist-v4-0.xsd	Not Used	50
emlcore-v4-0.xsd	Import	-
emlexternals-v4-0.xsd	Import	-
emltimestamp.xsd	Import	-
xal.xsd	Import	-
xmldsig-core-schema.xsd	Import	-
xnl.xsd	Import	-

Table 3.1: Table over the schemas in EML 4.0, indicating if they are used in the prototype subsystems and on which page they are discussed in this thesis.

classes for the Count and Result documents, though these are not utilized in the prototype subsystems. The classes present in the interface are listed in table 3.2 and the size of each class is measured in terms of the lines of code it contains. In total the EML Interface package spans 4327 lines of code.

Wrapper class	Lines of code
Authentication420	121
AuthenticationResponse430	479
Ballot	153
Ballots410	183
CandidateList230	404
CastVote440	262
Count510	217
ElectionEvent110	288
ElectionList330	377
Nomination210	315
NominationResponse220	140
Result520	191
VoteConfirmation450	111
Votes460	337
EMLServer	396
EMLClient	240
EML_io	140
Global	73
Total	4327

Table 3.2: Lines of code in EML_Interface

The classes EMLServer, EMLClient, EML_io and Global are the utility classes mentioned, and are further discussed in appendix E. The wrapper classes contain a set of get methods that are not discussed in this thesis. This is not done since these methods do not supply any information on the data transmitted.

3.2.2 Compiling EML

XMLSchemas define a language, or a set of XML documents. This definition is sufficiently formal to make it possible to compile the schemas to code. This compilation is done by a binding compiler.

In exploring manners of handling these schemas for this thesis a number of such compilers were tested, the results of this test is described in appendix A. Based on this test, the binding compiler chosen was XMLBeans. The compilation is done from the command line, although a Makefile was written to do this for this thesis. The command for doing the compilation is shown here (The options used are explained on page 86 in appendix A):

```
SCOMP = scomp
SFLAGS = -verbose -noupa -nopvr -out EMLclasses.jar -src EMLsrc
$(SCOMP) $(SFLAGS) $(XSDFILES)
```

3.2.3 Generated Code as Libraries for EML Interface

The code generated by XMLBeans is put into the directory EMLsrc and the compiled class files are jar'ed and put in a jar called EMLclasses.jar which then has to be placed in the CLASSPATH. The classes produced contain methods for creating documents, inserting elements into documents and parsing XML files adhering to the schemas.

3.2.4 EML Interface as a Library for the Subsystems

The EML_Interface is, as mentioned, a resource package for the subsystems and is imported by them as a library. It includes, besides the wrapper classes for the EML documents, also four other classes to abstract some input/output functions away from the subsystems. These classes will be discussed in appendix E.

In the same manner that the packages produced by XMLBeans are imported by the classes in EML_Interface, EML_Interface is also compiled into a jar and placed in the CLASSPATH and subsequently imported by the application subsystems. The compilation and jar'ing is also done by a Makefile in the prototype:

```
JAVAC = javac
JAVACOPTS = -d .
JAR = jar
JAROPTS = cvf
JARDEST = EML_Interface.jar
JAVACDEST = EML_Interface

$(JAVAC) $(JAVACOPTS) $(JAVAFILES)
$(JAR) $(JAROPTS) $(JARDEST) $(JAVACDEST)
```

3.2.5 Design Issues for EML Interface

Use of Ids and Names in EML Interface

Uniquely identifying objects brings with it a disproportionate amount of difficulty in most real life systems. The representation used to identify objects is often called identifiers, and one usually does not desire that user input be used unexamined as identifiers. Usually an even better solution than checking user input, is to only allow the user to choose from a list of real identifiers rather than to type one in.

This returns to the issue of using strings to uniquely identify objects and the human problem with spelling, punctuation and capitalization, to name a few.

This problem is usually solved in practice by to complementing procedures:

1. *Pick-one menus*: The user has to pick one of a set of predefined options. The user does not actually type in identifiers, unless in a specific input environment.
2. *Use of internal ids*: The string that constitutes a name might be user friendly, but might not be unique. An internal id can be made unique and is therefore used internally.

EML 4.0 provides for both name and id. However, in the prototype a name is often used as an id. As explained above, this is not an optimal solution, but sufficient in terms of testing of the thesis question. Below is a list of the structures in EML 4.0 (all defined in *emlcore*) that identify certain objects in an Electoral System. All provide for both name and id.

- *Event Id and Event Name*: **EventIdentifierStructure** - none of the fields are used. The prototype simulates one election event and is not meant to simulate several. An election event does not need to be identified, since there is only one.
- *Election Id and Election Name* **ElectionIdentifierStructure** - only the id field is used and the name of the election is the id.
- *Contest Id and Contest Name* **ContestIdentifierStructure** - only the id field is used and the name of the contest is the id.
- *Party Id and Party Name* **AffiliationIdentifierStructure** - only the RegisteredName field is used.
- *Candidate Id and Candidate Name* **CandidateIdentifierStructure** - both name and id is used.

3.3 EML Documents Outside the Prototype Subsystems

3.3.1 Multipurpose EML documents

There are two EML messages that could have been used by several subsystems, these are in addition to the messages that clearly belong to one of the subsystems. These two are briefly described here. Though neither is currently in use, they would be very interesting in a real implementation:

Inter Database (120): This schema describes a message that can be either a request or a response to a request. The schema allows for such requests/responses about a Voter Registration or Candidate. One could, in the context of this thesis, find this schema useful in communicating with the Population Registry Authority to query the eligibility of a candidate or whether a specific voter is also registered in another Electoral Register. This message can, then, be used by both the Nomination and the Election List subsystem.

Response (130): This schema defines a generic response that can be used where no explicit response exists. It has an "Accepted" field and an optional "Errors" field and can in that way communicate the error. This would be a crucial message in a real time system. In the prototype, however, error recovery is at a minimum and this message is, therefore, not used.

3.3.2 Referendum Nomination Documents

As already mentioned, referendums are not explored in this thesis, there are, however, three schemas in EML 4.0 dedicated to this, they are:

Options Nomination (610): Schema for submitting a proposal for a referendum.

Options Nomination Response (620): A response to the above Nomination with an accepted field and possibility of giving a reason for rejection in the Remark field.

Options List (630): A list of proposals for a referendum. There can be many elections, since each proposal is constitutes an election.

3.4 The Election Defined

3.4.1 ElectionEvent110.java

Only one schema is relevant to the Election Event Client: the ElectionEvent document. It contains only five different pieces of information, and the idea is that this should define an election sufficiently for the subsystems that receive it.

Table 3.3 shows the public inserting methods in ElectionEvent110.java, which is the wrapper class for this document.

Election Name is the name of an election. Each election held in an election *event* will have its name in the document.

Public set methods	Parameters
Constructor	String electionName String contestName String votingMethod String maxVotes String numberOfPositions
appendContest	String electionName String contestName String votingMethod String maxVotes String numberOfPositions

Table 3.3: Public inserting methods in ElectionEvent110.java

Contest Name is the name of a contest, in a Norwegian setting this will be the name of a municipality or a county. Note that there are municipalities that have the same name in Norway, that is, the municipality names are not unique per se. For this reason, one should have used id field for a unique identifier such as municipality number. However, as discussed earlier, this is not implemented.

Voting Method is not specified in the prototype, but is required by the EML schema, it is, therefore, only set to “other”.

Max Votes indicates the maximum number of votes for this contest.

NumberOfPositions is needed by the Voting subsystem because the number of changes a voter can make on a ballot is often a function of the number of representatives in the corresponding body being elected.

An example of such a document is shown below:

```
<ElectionEvent xmlns="urn:oasis:names:tc:evs:schema:eml">
  <EventIdentifier/>
  <Election>
    <ElectionIdentifier Id="Stortingsvalg"/>
    <Contest>
      <ContestIdentifier Id="Oslo"/>
      <VotingMethod>other</VotingMethod>
      <MaxVotes>500000</MaxVotes>
      <NumberOfPositions>40</NumberOfPositions>
    </Contest>
  </Election>
</ElectionEvent>
```

Note: The field *NumberOfPositions* is undocumented in the EML Schema Description. In the DataDictionary for EML 4.0, however, this field is said to have the following definition: “Represents the number of identical positions being elected in a contest.” The interpretation of this thesis is that this field indicates the number of representatives in the representative body that is up for election in this contest. In the Norwegian setting this is the number of representatives in the County Council in this county, Municipal Council in this municipality or the number of representatives in the Storting representing this county.

3.5 The Nomination Process

For nomination there are only three schemas, the CandidateList, the Nomination and NominationResponse. The Nomination and NominationResponse document constitute the communication between the Nomination Server and the Nomination Client, whereas the Candidate List is the output of the entire Nomination subsystem, sent to the Voting Server.

3.5.1 Nomination210.java

Nomination210.java produces and parses Nomination documents. Table 3.4 shows the public inserting methods for this document. The information inserted is discussed below:

Public set methods	Parameters
Constructor	String electionName String contestName String partyName
appendCandidate	String candidateName String yearOfBirth String profession String location boolean increased String partyName

Table 3.4: Public inserting methods in Nomination210.java

The constructor takes three parameters: election name, contest name and party name. It takes all of these to ensure a valid document is produced. These are not discussed here since they are covered by the discussion in section 3.2.5.

The additional method is the appendCandidate method, which takes six arguments. The profession, place of residence and partyName are optional in the general case. The specific requirements with regard to this document are listed on page 23 in section 2.6.2 where the laws and regulations regarding list proposals are treated. If the above mentioned fields are empty this is done by placing a dummy string in its place. Here, as is the case in all the schemas communication this information, the string used is "_".

candidateName: The full name of the candidate.

yearOfBirth: Year of birth of the candidate.

profession: Profession of the candidate.

location: Location or place of residence of the candidate.

increased: This field is either true or false indicating whether the candidate should receive an increased share of the poll.

partyName: Name of the candidates party/group, this field can be used if the list is a collaborative list.

Below is an example of a document produced in this manner:

```
<Nomination xmlns="urn:oasis:names:tc:evs:schema:eml"
 xmlns:urn="urn:oasis:names:tc:ciq:xsdschema:xAL:2.0">
  <ElectionIdentifier Id="ElectionName"/>
  <ContestIdentifier Id="ContestName"/>
  <Affiliation>
    <AffiliationIdentifier>
      <RegisteredName>Party Name</RegisteredName>
    </AffiliationIdentifier>
    <Candidate Independent="no">
      <CandidateIdentifier>
        <CandidateName>Candidate Name</CandidateName>
      </CandidateIdentifier>
      <DateOfBirth>1970-01-01</DateOfBirth>
      <QualifyingAddress>
        <urn:Locality>
          <urn:LocalityName Type="Location"/>
        </urn:Locality>
      </QualifyingAddress>
      <Affiliation>
        <AffiliationIdentifier>
          <RegisteredName>_</RegisteredName>
        </AffiliationIdentifier>
      </Affiliation>
      <Profession>Profession</Profession>
    </Candidate>
  </Affiliation>
</Nomination>
```

Note:

- *yearOfBirth*: The method appendCandidate takes year of birth and not date of birth. The schema demands date of birth, but in §6 of the Election Law only the year of birth is allowed. This is, therefore, resolved in Nomination210.java by adding a dummy day and month (day 01 and month January) and having the corresponding get method (getCandidateInfo) strip these off.

- *location*: The addresses used are meant to be *local specific* in EML. That is, each country is allowed, one might even say encouraged, to supply a country specific address schema. In this document a simple option in the eXtensible Address Language[1] (xAL) is used. Another option in xAL might be preferred or a Norwegian specific schema can be decided upon. This schema would then replace xal.xsd in the directory external.

  Location is used since it is not specified in the law to which degree of precision an address is to be specified. In this EML document it is simply treated as a string.

- *Increased share of the poll*: This schema has no facility to indicate that a candidate should receive an increased share of the poll. A workaround was implemented to provide this information. A field meant to indicate whether a candidate is independent or not has been used to indicate whether a candidate should receive an increased share of the poll. In the XML document displayed above, the candidate was inserted with *increased* set to false. This is reflected in the document in this manner:

  ```
  <Candidate Independent="no">
  ```

  This is not an optimal solution, and this deficiency in the schema and a proposed solution will be further discussed in the concluding chapter.

3.5.2 NominationResponse220.java

NominationResponse220.java produces or parses, as the name implies, a nomination response document.

Table 3.5 shows the one inserting method for this document, the constructor.

The constructor takes four arguments: election Name, contest Name, party Name and accepted. The accepted field can contain one of two strings: “yes” or “no”.

A NominationResponse document looks like this:

[1]The schema defining the eXtensible Address Language is found in the directory external in EML 4.0, the standard was developed by the OASIS Customer Information Quality Committee (CIQ)

Public set methods	Parameters
Constructor	String electionName String contestName String partyName String accepted

Table 3.5: Public inserting methods in NominationResponse220.java

```
<NominationResponse xmlns="urn:oasis:names:tc:evs:schema:eml">
  <ElectionIdentifier Id="ElectionName"/>
  <ContestIdentifier Id="ContestName"/>
  <Affiliation>
    <AffiliationIdentifier>
      <RegisteredName>PartyName</RegisteredName>
    </AffiliationIdentifier>
  </Affiliation>
  <Accepted>yes</Accepted>
</NominationResponse>
```

3.5.3 CandidateList230.java

The CandidateList schema describes a document that has to contain a bit more information than the previous two. Table 3.6 shows the public inserting methods for this document. As is the case with many of the documents, other set methods are private to maintain validity.

Public set methods	Parameters
Constructor	String electionName String contestName String partyName
appendCandidate	String electionName String contestName String partyName String [] candidateInfo

Table 3.6: Public inserting methods in CandidateList230.java

The constructor takes the following arguments: election name, contest name and party name.

AppendCandidate takes the above three as well, but in addition it takes a String array containing candidate information. This array has seven pieces of information:

Candidate name: Full name of candidate.

Candidate year of birth: Year of birth of candidate.

Candidate profession: Profession of candidate.

Candidate location: Location or place of residence of candidate.

Candidate id: Id of candidate.

Candidate party: Party of candidate.

Candidate increased share of poll: True/false indicating whether the candidate should receive an increased share of the poll.

A simple CandidateList document with only one candidate looks like this:

```
<CandidateList xmlns="urn:oasis:names:tc:evs:schema:eml"
 xmlns:urn="urn:oasis:names:tc:ciq:xsdschema:xAL:2.0">
  <Election>
    <ElectionIdentifier Id="ElectionName"/>
    <Contest>
      <ContestIdentifier Id="ContestName"/>
      <Affiliation>
        <AffiliationIdentifier>
          <RegisteredName>Party Name</RegisteredName>
        </AffiliationIdentifier>
        <Candidate Independent="no">
          <CandidateIdentifier Id="20000">
            <CandidateName>Candidate Name</CandidateName>
          </CandidateIdentifier>
          <DateOfBirth>1970-01-01</DateOfBirth>
          <QualifyingAddress>
            <urn:Locality>
              <urn:LocalityName Type="location"/>
            </urn:Locality>
          </QualifyingAddress>
          <Affiliation>
            <AffiliationIdentifier>
              <RegisteredName>_</RegisteredName>
            </AffiliationIdentifier>
          </Affiliation>
          <Profession>profession</Profession>
        </Candidate>
        <Proposer>
          <Name/>
        </Proposer>
      </Affiliation>
    </Contest>
  </Election>
</CandidateList>
```

Note:

- *Candidate id*: A unique id was added for all candidates. This id is visible to the voter at voting time and is used by the voter to indicate a specific candidate. The id is produced in the Nomination Server and remains constant for all candidates throughout the other subsystems, specifically the Voting Server, Voting Client and Counting Server.
- *Remaining fields* The same solutions to the year of birth, location and increased share of the poll problems, discussed in the section on Nomination210.java (page 54) are used here.

3.6 The Election List

The only schema used by the Election List Server is the ElectionList. A document conforming to this schema is sent to the Voting server and is the Electoral Roll for the election(s). The other schemas in EML relevant for this subsystem are:

Voter Registration (310) This schema is not used, although it could be used by the Population Registry Authority to, one by one, register the voters with the Election List Server. This is, however, not done, and it would seem that this schema is more applicable in a country that require the voters to register.

Polling Information (340) This schema is meant to give information to a voter. The information is on where, how and when to vote. This information can be tailored to the voter such that voter specific information can be provided. This can be taken advantage of when designing the security in an election where electronic voting is allowed. This schema is not used because there are no voters to communicate with and security is not an issue for this thesis. In a real setting this message has many uses, especially with regard to security.

Outgoing/Incoming Generic Communication (350a&b) These are generic message schemas to communicate to and receive communications from a voter. This schema is not used since there are no voters to communicate with. Although individual voter communication seem at first glance to be quite a massive process, this message can, for example, be used as a template for generated letters.

Internal Generic (350c) These are generic message schemas for communication between election organizers. It is not used since there are no such people to communicate with. This can be useful, perhaps, as a template for a message passing system running on the different subsystems.

Outgoing/Incoming Channel Options (360a&b) These schemas define a possible exchange between the electoral system and a voter, offering a set of channels through which the voter can vote or (possibly in response to the previous) a voter requests channels to vote through. This is not used since there are no voters to communicate with.

3.6.1 ElectionList330.java

Table 3.7 shows the constructor and appendVoter methods constituting the public insert method for this document.

Public set methods	Parameters
Constructor	String electionName String contestName String pollingPlace String firstNames String lastName String address String postalCode String postalLocation String personNumber
appendVoter	String electionName String contestName String pollingPlace String firstNames String lastName String address String postalCode String postalLocation String personNumber

Table 3.7: Public inserting methods in ElectionList330.java

The constructor takes nine arguments. The appendVoter method takes the same arguments and is, in fact, called by the constructor. Election name and contest name have already been discussed on page 48. The other arguments are:

Polling place: Location of the polling place the voter is to vote in.

First names: Voter first names.

Last name: Voter last name.

Address: Voter street address.

Postal code: Voter postal code.

Postal location: Voter postal location.

Personal id number: Voter personal identification number.

A simple ElectionList document with only one voter will look like this:

```
<ElectionList xmlns="urn:oasis:names:tc:evs:schema:eml"
 xmlns:urn="urn:oasis:names:tc:ciq:xsdschema:xNL:2.0"
 xmlns:urn1="urn:oasis:names:tc:ciq:xsdschema:xAL:2.0">
  <VoterDetails>
    <VoterRegistration>
      <Voter>
        <VoterIdentification>
          <VoterName>
            <urn:NameLine Type="First␣Names"/>
            <urn:NameLine Type="LastName"/>
          </VoterName>
          <ElectoralAddress>
            <urn1:Country>
              <urn1:AddressLine Code="PostalCode"
                                Type="StreetAddress"/>
              <urn1:Locality>
                <urn1:LocalityName Type="PostalLocation"/>
              </urn1:Locality>
            </urn1:Country>
          </ElectoralAddress>
          <Id Type="PersonalIdNumber"/>
        </VoterIdentification>
      </Voter>
    </VoterRegistration>
    <Election>
      <ElectionIdentifier Id="ElectionName"/>
      <ContestIdentifier Id="ContestName"/>
      <PollingPlace Channel="other">
        <OtherLocation Id="PollingPlace"/>
      </PollingPlace>
    </Election>
  </VoterDetails>
</ElectionList>
```

Note:

- *Address*: Here a more complex address in xAL is used, as opposed to the one used in the Nomination subsystem. This address form allows for sending information by mail to the voter.
- *Name*: The name also uses a more complex form, this one from the eXtensible Name Language (xNL). This schema (xNL) is also in the directory *external* and can be replaced if deemed appropriate.

- *Polling place*: A more complex address can be used, but in terms of this thesis this is not necessary.

- *Personal identification number*: This number is reused as a VToken in the Voting subsystem. The number uniquely identifies a voter, but, for anonymity concerns, could not be used in that manner in a real implementation.

3.7 Voting

3.7.1 Voting Server

The schemas applicable to the Voting Server are described below. Note that four of these also are applicable to the Voting Client, since these four constitute the communication between the Voting Server and Client. The final one, Votes, is the Voting Servers output, sent to the Counting Server.

3.7.2 Authentication420.java

Table 3.8 shows the only public insert method this document has, its constructor.

Public set methods	Parameters
Constructor	String vTokenString

Table 3.8: Public inserting methods in Authentication420.java

This schema describes a login message. Only one field is used: the VTokens Component field. The constructor takes the VToken string as input. Such a message is shown below:

```
<Authentication xmlns="urn:oasis:names:tc:evs:schema:eml">
  <VToken>
    <Component Type="PersonalIdNumber"/>
  </VToken>
</Authentication>
```

Note:

- *VToken*: VToken is meant as a way to uniquely distinguish a voter. Since privacy is not an issue for this thesis, the personal identification number is here used as a VToken. This is unique because all citizens of Norway have such unique numbers.

3.7.3 AuthenticationResponse430.java

AuthenticationResponse is meant as a response to an Authentication message. It also serves another significant purpose: It provides the voter with the ballots for the contests the voter is allowed to vote in. This is used in the prototype in a simple manner. The Voting Client has no information in its internal data structures before it receives an AuthenticationResponse. This message supplies the elections, contests and party lists used to fill up the internal data structures, and from these the menus are created.

Table 3.9 shows the two constructors that constitute the inserting methods for this document.

Public set methods	Parameters
Constructor	String response Ballot []ballots
Constructor	String response String ballotId

Table 3.9: Public inserting methods in AuthenticationResponse430.java

Response: Either “no” or “yes” depending on whether the voter is allowed to vote.

Ballots: The class Ballot is described below. There will be one ballot for each contest in the AuthenticationResponse document.

BallotId: Not used, but required. Can be set to anything, there is no get method in AuthenticationResponse430.java that returns it.

Table 3.10 shows the public insert methods in the class Ballot. Ballot is not a document in EML. This class was constructed so that ballots could be created by the Voting Server and inserted into the Ballots410 document. Each time an Authentication message is received, the ballots for the contest(s) the voter can vote in are retrieved from the Ballots410 document and inserted into the AuthenticationResponse document sent in reply.

Three of the parameters, electionName, contestName and partyName, are not described here, these follow the same structure as discussed on page 48.

maxWriteIn: Indicates the maximum number of candidates, on other party lists, that the voter can person vote for in a Municipal Council Election. For other elections this field will hold the value zero. This number is computed in the Voting Server based on the number NumberOfPositions (discussed in conjunction with the ElectionEvent document).

Public set methods	Parameters
Constructor	String electionName String contestName String maxWriteIn String partyName String [][] candidateArray
appendParty	String electionName String contestName String maxWriteIn String partyName String [][] candidateArray

Table 3.10: Public inserting methods in Ballot.java

The double array candidateArray, parameter to the Ballot constructor and method appendParty, contains the following for each candidate:

> *Candidate id*: This is the same id that was created in the Nomination subsystem, unique for each candidate.
>
> *Candidate information*: This is all the information about the candidate, in one long string, that the party wants to display to the voter. This is further discussed below.

An AuthenticationResponse message is shown below, where there is one election and its corresponding contest. Only one party has a submitted a party list, listing two candidates:

```
<AuthenticationResponse xmlns="urn:oasis:names:tc:evs:schema:eml">
  <Authenticated>yes</Authenticated>
  <EventIdentifier/>
  <Ballot>
    <Election>
      <ElectionIdentifier Id="ElectionName"/>
      <Contest>
        <ContestIdentifier Id="ContestName"/>
        <MaxWriteIn>200000</MaxWriteIn>
        <BallotChoices>
          <Affiliation>
            <AffiliationIdentifier>
              <RegisteredName>Party Name</RegisteredName>
            </AffiliationIdentifier>
            <CandidateIdentifier Id="ID1">
              <CandidateName>CandidateInfo1</CandidateName>
            </CandidateIdentifier>
            <CandidateIdentifier Id="ID2">
              <CandidateName>CandidateInfo2</CandidateName>
          </CandidateIdentifier>
        </Affiliation>
        <Affiliation>
          <AffiliationIdentifier>
            <RegisteredName>PartyName</RegisteredName>
          </AffiliationIdentifier>
          <CandidateIdentifier Id="ID1">
            <CandidateName>CandidateInfo1</CandidateName>
          </CandidateIdentifier>
          <CandidateIdentifier Id="ID2">
            <CandidateName>CandidateInfo2</CandidateName>
          </CandidateIdentifier>
        </Affiliation>
      </BallotChoices>
    </Contest>
  </Election>
 </Ballot>
</AuthenticationResponse>
```

Note:

- *MaxWriteIn*: In the EML Schema Descriptions, the MaxWriteIn field is not documented. In the DataDictionary, however, the following definition is found: "Where an election allows write-in candidates, this is the maximum number of such candidates that can be included." This field is used to indicate the maximum number of candidates a voter can give a person vote to, where that candidate is not on the list chosen. In the Municipal Council Election, which is the only kind of election in Norway that allows this, this limit is set to a fourth of the number of representatives to the Municipal Council (or at least 5), this was described in section 2.7.2.

- *Candidate Information*: The AuthenticationResponse document only allows a *CandidateIdentifierStructure* which means that it really only allows for *Candidate name* and *Candidate id*. This is insufficient for Norwegian purposes, since the law allows/requires that more information on a candidate be displayed to the voter (see section 2.7.3 on page 31).

  This is problem is solved in the prototype by stringing the required information together in a string separated by "," and splitting up the string on arrival at the Voting Client. The information contained in the string is:

  > *candidateName*: Candidates first and last names
  >
  > *YearOfBirth*: Here sent as year of birth, since there is no field requiring date of birth.

Profession: Candidates profession

Location: Candidates place of residence

Id: Candidate id, this is the same id produced in the Nomination subsystem.

Increased share of the poll: A string, either "true" or "false" depending on whether the candidate was awarded an *increased share of the poll* when nominated.

Representing party/group: Used if this is a collaborative list to present the individual candidates affiliation.

The fields above that are optional are listed as "_" in the string if not present.

3.7.4 CastVote440.java

The CastVote is the central schema in the EML model and the central concept in any Electoral System. In a Norwegian setting, this is document is crucial since it has to express the voters changes to the ballot, if there are any.

CastVote is perhaps one of the most complex documents used in the prototype. It has a constructor and two other insert methods. The appendSelection is a blank vote. For a vote that is not blank the appendCandidate method is used. These methods and the parameters they take are seen in table 3.11.

Public set methods	Parameters
Constructor	String electionName String contestName String vTokenString
appendCandidate	String electionName String contestName String partyName String candidateName String candidateId boolean personVote
appendSelection	String electionName String contestName

Table 3.11: Public inserting methods in CastVote440.java

Besides the parameters electionName, contestName and partyName, the parameters to the inserting methods in the CastVote document are:

VTokenString: VToken of the voter. In the prototype this is the personal identification number of the voter.

Candidate Name: Full candidate name.

Candidate Id: The same id created in the Nomination subsystem.

PersonVote: True or false. Indicates whether the voter wants to give this candidate a person vote.

An example of a document conforming to this schema is shown below. The thing to notice is the Candidate element. There can be a list of such elements in a Selection element. For a modification of order to be registered it is necessary that the order is fixed, which it is. Candidates can be added and the order can be changed. The Value attribute in the Candidate element indicates whether the candidate has received a person vote. If the field is "1" that indicates no person vote, if the field is "2" that indicates a person vote.

```
<CastVote xmlns="urn:oasis:names:tc:evs:schema:eml">
  <VToken>
    <Component Type="vToken"/>
  </VToken>
  <EventIdentifier/>
  <Election>
    <ElectionIdentifier Id="ElectionName"/>
    <Contest>
      <ContestIdentifier Id="ContestName"/>
      <Selection>
        <AffiliationIdentifier>
          <RegisteredName>PartyName</RegisteredName>
        </AffiliationIdentifier>
        <Candidate Value="2">
          <CandidateIdentifier Id="CandidateId">
            <CandidateName>Candididate Name</CandidateName>
          </CandidateIdentifier>
        </Candidate>
        <Candidate Value="1">
          <CandidateIdentifier Id="CandidateId2">
            <CandidateName>Candididate Name2</CandidateName>
          </CandidateIdentifier>
        </Candidate>
      </Selection>
    </Contest>
  </Election>
</CastVote>
```

Note:

- *Person Votes using WriteinCandidateName*: The CastVoteStructure has a field in the element Selection called *WriteinCandidateName*. The definition of this field in the DataDictionary is: "Some elections allow a voter to write-in the name of the person they would like to see elected, although they are not included on the ballot paper. This is known as a write-in candidate." The field, however, only allows one write-in candidate and is, therefore, not appropriate for Norwegian purposes.

- *Person Votes using VotingValueType*: This field is documented in the schema emlcore (line 887) as having the following meaning: "The weight or preference applied to a selection". In terms of this thesis, this is interpreted to be a person vote. A person vote indicates a "weight or preference" and the value held by this field is interpreted as: "1" - no person vote and "2"- a person vote.

3.7.5 VoteConfirmation450.java

This is a simple schema outlining a message to communicate to the voter whether a vote was accepted or not. The constructor to this wrapper class is the only public insert method. This is displayed in table 3.12.

Public set methods	Parameters
Constructor	String accepted

Table 3.12: Public inserting methods in VoteConfirmation450.java

If the vote was not accepted, the schema allows a reason to be supplied. In the prototype only the *accepted* field is used, which can be set to either "yes" or "no".

This constructor takes a string, either "yes" or "no". A message created in this way is shown here:

```
<VoteConfirmation xmlns="urn:oasis:names:tc:evs:schema:eml">
  <Accepted>yes</Accepted>
</VoteConfirmation>
```

3.7.6 Votes460.java

The Votes schema outlines a document that, essentially, is a list of votes. In the prototype the votes are transferred from the Voting Server to the Counting Server unprocessed, meaning that no field, except *VToken*, is ever read by the Voting Server.

Table 3.13 shows the public inserting methods in this wrapper class.

Public set methods	Parameters
Constructor	CastVote440 castVote440Doc
appendCastVote	CastVote440 castVote440Doc

Table 3.13: Public inserting methods in Votes460.java

The constructor and appendCastVote both only take a CastVote440 document as input. The method appendCastVote inserts a vote into the Votes document. AppendCastVote is also called from the constructor.

An example of such a document is shown below. There is only one cast vote inserted and in it there is only one candidate.

```
<Votes xmlns="urn:oasis:names:tc:evs:schema:eml">
  <CastVote>
    <VToken>
      <Component Type="VToken"/>
    </VToken>
    <EventIdentifier/>
    <Election>
      <ElectionIdentifier Id="electionName"/>
      <Contest>
        <ContestIdentifier Id="contestName"/>
        <Selection>
          <AffiliationIdentifier>
            <RegisteredName>partyName</RegisteredName>
          </AffiliationIdentifier>
          <Candidate Value="1">
            <CandidateIdentifier Id="candidateId">
              <CandidateName>candidateName</CandidateName>
            </CandidateIdentifier>
          </Candidate>
        </Selection>
      </Contest>
    </Election>
  </CastVote>
</Votes>
```

3.7.7 Ballots410.java

This document represents a list of Ballots. It can be used in many ways, in the context of this thesis it is used by the Voting Server to keep the ballots for the contests in the elections. Though not strictly necessary for this thesis, it speeds up the Voting Server because it will have very little construction to do when making an AuthenticationResponse. The relevant ballots are inserted directly, and very little actual building of the AuthenticationResponse message is needed. The public inserting methods for this wrapper class is depicted in table 3.14.

The document is used much like a HashMap. A ballot is inserted with the concatenation of election and contest name as a key. As mentioned on page 48 with regard to using a name as a unique identifier, this is not sufficient for Norwegian purposes, since there are municipalities in Norway that have the same name. However, the prototype simulates a small election event with only a few municipalities so this problem is not encountered. In a real

Public set methods	Parameters
Constructor	Ballot ballot String key
insertBallot	Ballot ballot String key

Table 3.14: Public inserting methods in Ballots410.java

implementation ids would have to be used to keep the keys unique.

3.7.8 Remaining Schemas for the Voting Subsystem

In addition to the schemas above, EML 4.0 provides the following three documents for this subsystem. These are not currently in use. VToken Log and Audit Log, both having to do with security, are outside the scope of this thesis.

> *Retrieve Vote (445)* This schema is meant to be used when a pre-ballot box is used. One might view the Voting Server as a pre-ballot box, or one might see the votes received by the Counting Server to be such a pre-ballot box. In any case, used in this prototype this would be an internal document and would, therefore, not bring anything new to the analysis.
>
> *VToken Log (470)* This is a list of VTokens and/or VTokenQualifieds (these are not pursued in this thesis). It can have a variety of purposes, in the context of this thesis' prototype, it could have communicated the VTokens (either only the used ones, or possibly all) to the Voting Server so that the votes could be checked before counting. This would ensure that invalid VTokens are not used. This is, as mentioned, a security issue and not relevant for the thesis.
>
> *Audit Log (480)* This message is used to keep an audit of seals used. It would be used as a part of the security measures of an election and is, therefore, not relevant for this thesis.

3.7.9 Voting Client

The four schemas for the Voting Client have already been discussed with regard to the Voting Server. Of the five documents discussed there the Votes document is the only one that is not either written or read by the Voting Client.

3.8 Counting

The final two documents for used in the termination of the election event are the Count510 and Result520 documents. Neither is used in the prototype since the counting implemented does not produce complete results. Wrapper classes have been implemented anyway, mostly to illustrate their use.

3.8.1 Count510.java

The Count document represents an intermediate count using a specific algorithm. As described in the section on the Counting subsystem, Norwegian law do call for a number of different counting algorithms to be performed.

The count document can represent the results of a count. In a Norwegian context this could then be used to store/present an intermediate count using a specific algorithm that can be identified using the *CountingAlgorithm* field. Each candidate is represented in the *Selection* element and equipped with a *Value* attribute and a *ValidVotes* field. These, together with the ordering, allow for some additional information to be given as to the results of the count.

The Count510 wrapper class has a constructor and an appendSelection as its public inserting methods. These methods and the parameters they demand is seen in table 3.15.

Public set methods	Parameters
Constructor	String electionName String contestName String candidateId String countingAlgorithm String numberOfPositions String partyName String validVotes String votingValueType
appendSelection	String electionName String contestName String candidateId String countingAlgorithm String numberOfPositions String partyName String validVotes String votingValueType

Table 3.15: Public inserting methods in Count510.java

The constructor and the appendSelection method both take the following parameters in addition to electionName, contestName and partyName:

candidateId: Indicates the same id generated in the Nomination subsystem.

countingAlgorithm: Indicates the algorithm used for the count.

numberOfPositions: The number of positions in the body to be elected. This field is further discussed below.

validVotes: Number of valid votes in the count.

votingValueType: This same type of field was used to express a person vote in the CastVote document. Here it may serve a variety of purposes in expressing the result of the counting algorithm.

An example of a Count document is shown below:

```
<Count xmlns="urn:oasis:names:tc:evs:schema:eml">
  <EventIdentifier/>
  <Election>
    <ElectionIdentifier Id="ElectionName"/>
    <Contests>
      <Contest>
        <ContestIdentifier Id="ContestName"/>
        <CountingAlgorithm>CountingAlgorithm</CountingAlgorithm>
        <NumberOfPositions>40</NumberOfPositions>
        <TotalVotes>
          <Selection Value="6">
            <AffiliationIdentifier>
              <RegisteredName>PartyName</RegisteredName>
            </AffiliationIdentifier>
            <CandidateIdentifier Id="CandidateId"/>
            <ValidVotes>5000000</ValidVotes>
          </Selection>
          <Selection Value="3">
            <AffiliationIdentifier>
              <RegisteredName>PartyName2</RegisteredName>
            </AffiliationIdentifier>
            <CandidateIdentifier Id="CandidateId2"/>
            <ValidVotes>700000</ValidVotes>
          </Selection>
        </TotalVotes>
      </Contest>
    </Contests>
  </Election>
</Count>
```

Note:

- *NumberOfPositions*: This field is referred to here in Count, though the Counting subsystem does not receive the ElectionEvent document that gives this number, this seems to imply that the figure "High-Level Model - Technical View" on which figure 1.1 is based, is incomplete.

3.8.2 Result520.java

The result document does not give the final results of the election, hence the name is a bit misleading. It does, however, give the candidates individual results. All candidates in a contest in the election(s) will have their results summed up in this document.

Since the Counting Server has not implemented the full counting scheme of Norwegian elections this schema is not used in the Counting subsystem. Nonetheless, a wrapper class was created, where only the set methods are implemented. These are one constructor and one public method appendSelection both requiring the same parameters. Table 3.16 shows the parameters for these methods.

Public set methods	Parameters
Constructor	String electionName String contestName String candidateId String partyName String numVotes String ranking boolean elected
appendSelection	String electionName String contestName String candidateId String partyName String numVotes String ranking boolean elected

Table 3.16: Public inserting methods in Result520.java

The three pieces of information given for a candidate are: *Votes*, *Ranking* and *Elected*.

Votes: Will imply the number of votes received by this candidate.

Ranking: Can communicate the ranking that the candidate received.

Elected: Will be a "yes" or "no" answer.

Since candidates that were not elected are still in the document, one might see this as a last step before a complete result giving the overall picture of the candidates elected and the distribution of seats between the parties.

Below is an example of such a document.

```
<Result xmlns="urn:oasis:names:tc:evs:schema:eml">
  <Election>
    <ElectionIdentifier Id="ElectionName"/>
    <Contest>
      <ContestIdentifier Id="ContestName"/>
      <Selection>
        <CandidateIdentifier Id="CandidateId"/>
        <AffiliationIdentifier>
          <RegisteredName>PartyName</RegisteredName>
        </AffiliationIdentifier>
        <Votes>200000</Votes>
        <Ranking>2</Ranking>
        <Elected>yes</Elected>
      </Selection>
      <Selection>
        <CandidateIdentifier Id="CandidateId2"/>
        <AffiliationIdentifier>
          <RegisteredName>PartyName</RegisteredName>
        </AffiliationIdentifier>
        <Votes>400000</Votes>
        <Ranking>1</Ranking>
        <Elected>yes</Elected>
      </Selection>
    </Contest>
  </Election>
</Result>
```

Chapter 4

Conclusion

4.1 Introduction

There are some shortcomings in the EML standard with regard to the Norwegian Electoral System. Though, these shortcomings are minor, they do require changes to the schemas in EML.

This chapter concerns itself with these shortcomings and suggests changes to the standard to make it a more suitable interface language for the Norwegian Electoral System. The solutions suggested will be based on the idea of keeping backwards compatibility to EML 4.0, in this manner not affecting other electoral systems supporting EML 4.0.

Finally, some issues regarding the current Election Law and Election regulations are dealt with, and possible improvements suggested.

4.2 Changing a Standard

EML 4.0 supports Norwegian elections reasonably well. This is illustrated by the fact that the prototype does manage to simulate the elections. The problems that were encountered have been described in the previous two chapters, and it is the purpose of this section to suggest changes in the schemas in EML 4.0 to further accommodate the Norwegian Electoral System. First a few concepts are clarified so that the extent of the change to the standard is clear. It is important to note that the changes that will be described are not implemented in the prototype subsystems use of EML 4.0.

Problems Encountered when Changing a Standard

Modifying a communication standard that is in use, creates difficulties for subsystems conforming to the standard. This problem occurs when two subsystems in a system implement different versions of the standard and when a system is ported to a new version of the standard.

The aforementioned issues are really two different problems, and can be better distinguished if one more clearly defines what is meant by *backwards compatibility*. Further it is necessary to discuss if it is desirable to extend EML 4.0 without changing the standard itself. These two topics are dealt with before the proposed changes to EML are presented.

4.2.1 Keeping Backwards Compatibility

When changing a language, one usually modifies it so that it expresses more than it did. This will be called a *language extension*. If one wishes to modify a language such that it expresses less than it did, it will be called a *language restriction*. Finally, a language can be changed so that, whether it expresses more or less, it does express common elements in the versions differently. This will be called a *language modification*.

Language extension, *language restriction* and *language modification* present different problems, both between subsystems implementing different versions and when porting a system to a new version.

Using Different Versions of a Standard

The problem that occurs when two subsystems with different versions of a standard communicate, is that a document that is valid with regard to one version, may not be valid with regard to the other. The subsystem that suffers depends on whether the new version is a *language extension* or a *language restriction*. If the new version is a *language modification* both subsystems may suffer.

- Consider a subsystem A implementing a version V1 and a subsystem B implementing a version V2. Assume that V2 is a *language extension* of V1. In such a case subsystem B can produce documents not in the language of V1, therefore not valid to subsystem A.

- Assume now that V2 is a *language restriction* of V1. Subsystem A can now produce documents that are not in the language of V2 and, subsequently, not valid for subsystem B.

  The above problems cannot be solved. If a subsystem implementation takes advantage of a mechanism not present in both versions, the above is unavoidable.

- Finally assume that V2 is a *language modification* of V1. If the subsystems use elements that have been modified in V2, the communication between the two will break down.

  This is also unavoidable if there are no other common mechanism in the versions facilitating the same data exchange. If there is, both subsystems will have to be rewritten to use the common mechanism in order to communicate.

Porting a Subsystem to a new Version

When porting to a new version of the standard, the problems really only occur during a *language restriction* or a *language modification.*

- A *language extension* will make it possible to keep the implementation as it is. If there is no need to take advantage of the additions to the language, the electoral system only has to be able to parse documents containing these new structures.
- A *language restriction* may not be possible to implement. Some facilities in the language critical to the function of the electoral system could have disappeared from the language, making it impossible to upgrade. Another possibility is that the electoral system will have to be rewritten to take advantage of other facilities still in the language to achieve the same effect.
- Finally, if the modifications done in the *language modification* affects the parts of the language used by the electoral system, it needs to be rewritten.

One would hope that a *language restriction* does not occur. In any case, a *language modification* is the worst of both worlds. Remember, a *language modification* can include a *language restriction* or a *language extension* or even both.

Defining Backwards Compatibility

In this context, based on the above, the concept of *backwards compatibility* will be defined as meaning a *language extension* with no *language modification.* Meaning that the additions to the language have to be elements that are optional in the schemas.

The rest of this section will be dedicated to proposed changes in the schemas in EML. It is the intent of the author that these changes maintain *backwards compatibility*, in the sense that any EML document that was valid in EML 4.0 remains valid in the altered version.

4.2.2 Extending EML using the Wildcard Mechanism

XML Schemas allow wildcard elements called “any”, these have associated with them a namespace. Most, if not all, schemas in EML 4.0 allow “any” elements. These elements in EML use the "##other" namespace. An example of this is shown below[1]:

```
<xs:any namespace="##other"
        minOccurs="0"
        maxOccurs="unbounded"/>
```

The "##other" namespace is defined to be: "Any well-formed XML that is from a namespace other than the target namespace of the type being defined (unqualified elements are not allowed)"[2]

This means that the schemas can be extended to include other elements (not in EML).

Since there are concepts in the Norwegian Electoral System that are not provided for in the EML 4.0 standard, this mechanism could be utilized in an interim phase. However, these extensions would be nation or producer specific and would to some extent erode the standard. Different subsystems would no longer be compatible unless all the implementations supported the extensions. It seems, then, to be more desirable to push for a change in the standard, rather than having nation specific elements.

4.3 Proposed Modifications to EML 4.0

The modifications proposed below all aim at maintaining *backwards compatibility* with EML 4.0. Where elements from schemas are listed, portions that are not relevant have been cut out and replaced with "/*...*/".

4.3.1 Accommodating Norwegian Party List Nomination

The Norwegian party list nomination process is hard to express in EML. The process is complex, and input received here has to be propagated throughout the electoral system. The candidate information has to be displayed on the ballot presented to the voter, and the counting process is in some elections affected by information provided in the Nomination document, i.e. the candidates receiving an *increased share of the poll.* Some issues regarding this process has to be changed by law, others can be resolved by changing EML. The latter is presented here, the former is presented in section 4.4.

Proposer Date of Birth

The *ProposerStructure*[3] does not supply a field for a proposers date of birth. This information is, however, required to be present in a list proposals *enclosed document* [16, §6-4(b)].

[1] line 72 *210-nomination-v4-0.xsd*
[2] http://www.w3.org/TR/xmlschema-0/
[3] line 546 *emlcore-v4-0.xsd*

The *ProposerStructure* contains several other optional fields, and adding another would not change the structure in any significant way. The suggested element would have the form:

```
<xs:element name="DateOfBirth" type="xs:date" minOccurs="0"/>
```

Below is the *ProposerStructure* after the element has been inserted (note lines 8 through 10):

```
1  <xs:complexType name="ProposerStructure">
2      <xs:sequence>
3          <xs:element name="Name"
4                      type="PersonNameStructure"/>
5          <xs:element name="Contact"
6                      type="ContactDetailsStructure"
7                      minOccurs="0"/>
8          <xs:element name="DateOfBirth"
9                      type="xs:date"
10                     minOccurs="0"/>
11          /*...*/
12 </xs:complexType>
```

Since the field has *minOccurs* set to 0, this is a *language extension* and not a *language modification*.

Increased Share of the Poll

In a Municipal Council Election a party can give certain candidates an *increased share of the poll* [16, §6-2(3)]. This information has to be present in the following documents to fulfill the requirements of the Election Law and regulations:

> **Nomination** This information is required to be present in the list proposals *enclosed document* [16, §6-2(3)].
>
> **CandidateList** This information is required by the Counting subsystem to perform the counting in a Municipal Council Election [16, §6-2(3)]. The CandidateList also has to be input to the Counting subsystem for this transfer of information to be possible, this is further treated in section 4.3.3.
>
> **Ballots and AuthenticationResponse** The candidates that receive an *increased share of the poll* have to be indicated on the ballot [13, §19(9)].

Adding an attribute to the *CandidateStructure*[4] would include this information in the Nomination and CandidateList documents. The *CandidateStructure* will have to be included in the Ballots and AuthenticationResponse

[4] line 155 *emlcore-v4-0.xsd*

documents for this information to be available there, this change described in section 4.3.2. The proposed attribute would have the form:

```
<xs:attribute name="IncreasedShareOfPoll"
              type="YesNoType"
              use="optional"/>
```

Below is the *CandidateStructure* after this attribute has been inserted (note lines 29 through 31):

```
1  <xs:complexType name="CandidateStructure">
2          <xs:sequence>
3                  <xs:element ref="CandidateIdentifier"/>
4                  <xs:element name="CandidateFullName"
5                              type="PersonNameStructure"
6                              minOccurs="0"/>
7                  <xs:element name="DateOfBirth"
8                              type="xs:date"
9                              minOccurs="0"/>
10                 /*...*/
11                 <xs:element name="QualifyingAddress"
12                             type="QualifyingAddressStructure"
13                             minOccurs="0"/>
14                 /*...*/
15                 <xs:choice minOccurs="0">
16                         <xs:element ref="Affiliation"/>
17                         <xs:element ref="Logo"
18                                     maxOccurs="unbounded"/>
19                 </xs:choice>
20                 <xs:element name="Profession"
21                             type="xs:token"
22                             minOccurs="0"/>
23                 /*...*/
24         </xs:sequence>
25         /*...*/
26         <xs:attribute name="Independent"
27                       type="YesNoType"
28                       use="optional"/>
29         <xs:attribute name="IncreasedShareOfPoll"
30                       type="YesNoType"
31                       use="optional"/>
32 </xs:complexType>
```

Inserting this attribute constitutes a *language extension*, and since the attribute is optional this is not a *language modification*.

4.3.2 Allowing more Candidate Information on Ballots

A CandidateStructure in AuthenticationResponse/Ballots

The Election regulation [13, §19(6), §19(9)] requires/allows the following information regarding each candidate to be present on the ballot:

The candidates first name and last name.

The candidates year of birth.

The candidates profession.

The candidates place of residence.

Whether the candidate is to receive an *increased share of the poll.*

The party/group the candidates is representing [15, §19(6)c].

The *BallotChoices* element in the *BallotStructure*[5] used in the AuthenticationResponse and Ballots documents only provides for a *CandidateIdentifierStructure*[6] when listing candidates. Of the information listed above, this structure only allows for a candidates name. To list the additional information the law allows, a full *CandidateStructure* could be used. The element within the *BallotChoices* that has to be changed is *Affiliation*[7]. The current definition of this element is as shown here:

```
<xs:element name="Affiliation">
    <xs:complexType>
        <xs:complexContent>
            <xs:extension base="AffiliationStructure">
                <xs:sequence>
                    <xs:element ref="CandidateIdentifier"
                                minOccurs="0"
                                maxOccurs="unbounded"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
</xs:element>
```

By adding a *choice* to the sequence and, by this, allowing for either a full *CandidateStructure* or the current *CandidateIdentifierStructure*, this element could express the necessary information. This assumes that the *increased share of the poll* attribute has been added to the *CandidateStructure* type.

[5]line 92 *340-410-430-include-v4-0.xsd*
[6]line 147 *emlcore-v4-0.xsd*
[7]line 97 *340-410-430-include-v4-0.xsd*

```
<xs:element name="Affiliation">
    <xs:complexType>
        <xs:complexContent>
            <xs:extension base="AffiliationStructure">
                <xs:sequence>
                    <xs:choice>
                        <xs:element ref="CandidateIdentifier"
                                    minOccurs="0"
                                    maxOccurs="unbounded"/>
                        <xs:element ref="Candidate"
                                    minOccurs="0"
                                    maxOccurs="unbounded"/>
                    </xs:choice>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
</xs:element>
```

Since the above still allows the current *CandidateIdentifierStructure* and *minOccurs* for the *CandidateStructure* is 0, this constitutes a *language extension* with no *language modification.*

4.3.3 Input Documents to the Counting Server

In a Norwegian setting, the principle of an *increased share of the poll* makes it necessary for the Counting subsystem to receive the CandidateList. When counting in a Municipal Council Election the candidates that have received such an *increased share of the poll* have to have this included in their count.

The document Count refers to the *NumberOfPositions* in the body up for election. Unless the Counting subsystem receives the ElectionEvent document, this information is unavailable. Since several of the counting algorithms use this number when computing the lists *list votes*, this document is necessary for the proper function of this subsystem.

4.4 Proposed Adaption of the Election Law and Election Regulations

There are some considerations that concern the Election Law and regulations themselves. These will be described below and possible solutions will be suggested.

4.4.1 Enclosed Document

The law requires the list proposal to be separate from the *enclosed document* [16, §6-1(3)]. This separation is not necessary as long as the information in the *enclosed document* has no overlap with the information allowed to be present in the list proposal. This will ensure that there is no miscommunication possible when submitting a list proposal.

4.4.2 Candidate Eligibility

A declaration that a candidate will be eligible has to be present if a candidate has a position that currently makes him/her ineligible [16, §6-4(d)]. The processing of these declarations is a manual process and should be separated from the automated information exchange.

4.4.3 Electronic Signatures for Proposers

The signatures required for list proposals are today written on paper, in accordance with regulation [13, §12]. This is probably practical when trying to obtain them. In the future an electronic signature might suffice.

The form of the type *SignatureType*, shown on page 28, and its Object element, allows for other than the most obvious uses. A scanned representation of the signature could be placed in the Object field, and the signatures could then be transmitted by the Nomination document.

However, this is contrary to the "writing on paper" formulation in the regulation, and is, for that reason, not pursued further.

4.4.4 Personal Identification Numbers for Candidates in List Proposals

The nomination subsystem has to cross check the list proposals for a contest to see if two list proposals list the same candidate. Since the Election Law only allows the list proposals *enclosed document* to contain the candidates date of birth, this cross check becomes computationally difficult.

There is no reason to believe that a situation cannot occur where two candidates having the same name, also have the same date of birth. The case where two candidates having the same name also have the same year of birth is treated in the law by requiring profession and/or place of residence to be supplied [16, §6-1(2c)]. This part of the law concerns itself with the candidates on one list only. Though sharing a date of birth and name is not likely, it is possible, and across several lists the probability of this occurring increases.

This cross check would be simple if the candidates were listed with their full personal identification numbers in the *enclosed document*.

4.4.5 Candidate Numbers for Municipal Council Elections

It is the opinion of this author that a unique candidate identifier should be used when person voting for candidates on other lists in a Municipal Council

Election. This will eliminate the possible ambiguities in using candidate names.

4.5 Future Work

4.5.1 Stress Testing of the Voting Server

Before EML 4.0 is used, it should be tested whether the format itself is scalable to handle the amount of pressure that such a system will have to face. Though EML has been tested in other countries, the size of the CastVote document may be an issue that is unique to the Norwegian Electoral system. Because of the actual listing of the candidates, this document can become quite large. This increased size will be in addition to the bloating that occurs with XML tags.

Other documents in the Electoral System can become a great deal larger than a CastVote. Even though, the Voting Server will receive CastVotes on a large scale and the processing of them will have to be efficient.

The prototype developed could be used to test the processing of CastVote documents. This specific part of the Voting Server could be optimized to perform this exchange more rapidly.

To ensure an auditable election where the votes cast can be recounted and/or checked to see if they were counted, the votes received will have to be stored on a persistent medium. This, together with network load, might be the bottleneck in terms of the processing time of a CastVote. In both of these contexts size does matter, and a cast vote in a Norwegian setting is quite large.

4.5.2 Using EML as an Interface Language for a N-Version Model

In addition to facilitating standardized communication, the EML documents produced and received can be seen as standardized input and output. Security models such as n-version [21] require a standardized interface like EML to make it possible to compare the outputs of different implementations of the same subsystem, receiving the same input. These models can further assure a correct implementation of the subsystems in an electoral system.

An implementation of a n-version framework using EML as an interface language, would reveal if EML is suitable for this model.

4.6 Conclusion

The Election Markup Language version 4.0 is very close to a suitable standard for the Norwegian Electoral System. There were, however, some problems when implementing it. These problems were addressed in this chapter and changes to the schemas were proposed. The changes will only affect two schemas in EML:

- Proposer date of birth: Change proposed in *emlcore-v4-0.xsd*
- Increased share of the poll: Change proposed in *emlcore-v4-0.xsd*
- More candidate information on the ballot: Change proposed in *340-410-430-include-v4-0.xsd*
- More documents to the Counting subsystem: Change proposed in the documentation of the standard.

All the changes proposed were minor and can be achieved maintaining backwards compatibility.

The following adaptations of the Election law and regulations are proposed:

- No separation between the list proposal and the *enclosed document.*
- Separate the declaration of eligibility from the *enclosed document.*
- Allow a scanned representation of the signature of a proposer in a list proposal.
- Using a candidate number instead of candidate name when person voting.
- Including personal identification numbers for candidates listed in a list proposal.

The use of an inter-system interface, like EML, together with a wide set of system suppliers, will help to ensure a more transparent electoral information system.

Appendix A

Evaluating Binding Compilers by Compiling EML 3.0

A.1 Compiling Emlcore

A.1.1 Introduction

In 2001 the W3C introduced the XML Schema standard. The idea of strictly formalizing XML documents through a schema was quickly applied in different programming tools through what is called a binding compiler. The binding compiler takes schemas and DTDs as input and produces classes and interfaces representing the elements of the schemas and DTDs. These can then be used by instantiating them to objects. These objects can be marshaled into XML documents and XML documents can be unmarshaled into object instances of the same classes. EML (Election Markup Language) chose the XML Schema format to define the communication and representation of various acts within an electoral system. The purpose of this testing is to explore some of the various tools implemented to handle XML Schemas and to create code for them. The tools that have been tested are: XML Spy, <oXygen/>, XML Beans and JAXB. The EML version tested is version 3.0 that can be obtained from the following URL: http://www.oasis-open.org/committees/download.php/10021/EML%20version%203.0.zip

Validating Emlcore

Only schemas that are valid can be compiled with a binding compiler. The XML schema that initially needed to be validated was emlcore.xsd, the reason this schema was chosen is that it is imported by a great many other schemas in EML. It would, for this reason, not be possible to validate these before emlcore itself can be validated. The tools were initially tested on the schema as is, with no changes and in the location it was when unzipped. They were then tested after making 3 changes. These changes are:

1. Storing the file **BS7666-v1.xsd** in the same directory

2. **Line 90** is changed from:
   <xsd:attribute ref="xml:lang" *type="xsd:language"* use="optional"/>
   To:
   <xsd:attribute ref="xml:lang" use="optional"/>

3. **Line 110** is changed from:
   <xsd:attribute name="Id" use="*optional*"/>
   To:
   <xsd:attribute name="Id" use="*required*"/>

The effect of these changes had different results for the different compilers. This will be discussed individually for each compiler. The results are summarized in table A.1 and the errors emitted are summarized in table A.2.

A.1.2 Testing the Binding Compilers

XMLSpy

XMLSpy[1] is the tool used to create EML and, therefore, initially was assumed to be the best tool for the job. It is a GUI based development environment and runs only on Windows. It is proprietary, but a trial "home edition" can be downloaded from Altovas website. When one opens emlcore in XMLSpy a pop up window appears telling the user that the schema BS7666-v1.xsd can not be found, any attempt to validate the schema results in the same error message. When the BS7666 schema is stored in the directory another error occurs when trying to validate emlcore. It is on line 110 and is the error referred to as error number 5 in table A.2. After this is resolved, as mentioned above, the error referred to as error number 3 was emitted. After its resolution error 9 was encountered and not resolved. As the above indicates the downside to XMLSpy is that it only emits one error message at a time.

<oXygen/>

Oxygen[2] was recommended in the article "Article on EML (v3)" by David Mertz, Ph.D and is also a GUI editing tool. It runs on multiple platforms and is also proprietary with a thirty day trial period. It seems that oxygen uses the same binding compiler that is in the command line based compiler xjc discussed below. This means that the error messages emitted from oxygen are the same word for word as the ones emitted by xjc. The upside to oxygen

[1]Can be downloaded from: http://www.altova.com/download_spy_home.html
[2]Can be downloaded from: http://www.oxygenxml.com/download.html

over XMLSpy is that all the errors are emitted at once. Even though, it still could not validate the schema after the corrections were made.

xjc - JAXB

xjc[3] is a command line binding compiler and is freely down-loadable with the Java Web Services Pack from Sun. Since it is clearly the compiler behind oxygen it is by far the best compiler in terms of finding errors and listing the line numbers for the errors, which the other command line complier scomp does not. The error messages are well formulated and, for debugging purposes, invaluable. It is, however, command line based and does not give you that ease of use that comes with oxygen and XMLSpy.

scomp - XMLBeans

scomp[4] is also a command line compiler, open source and is also freely down loadable. It has several options available as to the compilation process, but the errors are emitted without line numbers. The options are, however, what make this tool stand out as the only one in the test that actually managed to compile emlcore after the corrections were made. This was done with the following two options:

noupa: do not enforce the unique particle attribution rule

nopvr: do not enforce the particle valid (restriction) rule

The results of only for scomp are, therefore, divided into four:

Scomp(1): no corrections - no options

Scomp(2): no corrections - options: noupa

Scomp(3): no corrections - options: noupa and nopvr

Scomp(4): corrections - options: noupa and popvr

As shown in table A.1 scomp(2) removed the unique particle errors and scomp(3) removed all but the three that were corrected. Scomp(4) compiled emlcore and created java code for it.

[3]Can be downloaded from: http://java.sun.com/webservices/downloads/webservicespack.html
[4]Can be downloaded from: http://xmlbeans.apache.org/sourceAndBinaries/index.html

Validating the Rest of EML Using Scomp

The remaining schemas, with one exception, were then validated with only one correction in each schema. This correction was in the id field for each schema. Exemplified by the schema 110-electionevent.xsd. This field was altered because it did not conform to the following rule:

NCName ::= (Letter | '_') (NCNameChar)*
NCNameChar ::= Letter | Digit | '.' | '-' | '_' | CombiningChar | Extender

Abbreviated this line was (line 3):

```
<xsd:schema targetNamespace="[...]" xmlns:xsd="[...]" xmlns="[...]"
xmlns:xlink="[...]" elementFormDefault="[...]"
attributeFormDefault="[...]" version="[...]" id="110">
```

It was slightly changed by adding an underscore:

```
<xsd:schema targetNamespace="[...]" xmlns:xsd=".." xmlns="[...]"
xmlns:xlink="[...]" elementFormDefault="[...]"
attributeFormDefault="[...]" version="[...]" id="\_110">
```

With this modification the field now conformed to NCName. This change was made in all the numbered schemas and was sufficient to make them validate with the exception of 510-count.xsd. The error in 510-count.xsd is not included and still not resolved. This schema was, therefore, excluded and the rest of EML 3.0 compiled. Other errors were discovered at a later date, some are listed below:

> *Nomination (210)*: an any element was required in the element ProposerStructure.
>
> *Authentication (420)*: an any element was required in the element Component, used in the element VToken.
>
> *VTokenLog (470)*: same error as above in Authentication.

A.1.3 Results

Comparing the Error Emitting of the Binding Compilers

This table gives an overview of the results of the tests. Each row represents an error. Refer to table A.2 for a brief look at the errors, "Num" in table A.1 corresponds to "Num" in table A.2. scomp(4)[5] is not in the table since it did not produce any errors.

Note: scomp(1) does not specify line number. The errors are, therefore, assumed to apply to the lines specified. Question mark is to indicate that errors are produced, but it is not clear for which lines.

Num	Line	XMLSpy	<oXygen/>	xjc	scomp(1)	scomp(2)	scomp(3)
1	BS7666	X	X	X	X	X	X
2	90(1)	X	X	X	X	X	X
3	90(2)		X	X	?		
4	90(3)		X	X	?		
5	110	X	X	X	X	X	X
6	170		X	X	?		
7	204		X	X	?		
8	275(1)		X	X	?		
9	275(2)	X	X	X	X	X	
10	322		X	X	?		
11	344(1)		X	X	?		
12	344(2)		X	X	X	X	
13	392		X	X	?		
14	477		X	X	?		
15	525		X	X	?		
16	18		X	X	?		

Table A.1: Lists errors emitted by the binding compilers

Short Summary of the Errors Emitted

This table shows the errors and corresponding error numbers that map into table A.1. An attempt has been made to keep them as succinct as possible.

Note: Line number for error 16 is the line number for a line in emlexternals.xsd.

A.1.4 Conclusions

The different tools serve clearly different purposes in tying to validate emlcore. The GUI based tools are more user-friendly and are more forgiving when it comes to the errors usually committed by a new user. They also provide line numbers for the errors emitted, which is obviously necessary when attempting to remove them. XMLSpy, however, loses points by its one-at-a-time error emitting policy. The command line compilers score well in two different categories. Xjc is by far the most powerful validator of the

[5]Errors 1, 2 and 5 were corrected.

Num	ERROR	Line
1	Unable to load following schema(s) [...]BS7666-v1.xsd	
2	'ref' attribute precludes local simple type, 'form' or 'type' attribute	90(1)
3	Cannot resolve the name 'xml:lang' to a(n) 'attribute declaration' component.	90(2)
4	Element 'attribute' is invalid, misplaced, or occurs too often.	90(3)
5	Basetype requests attribute to be required	110
6	Unique Particle Attribution	170
7	Unique Particle Attribution	204
8	There is not a complete functional mapping between the particles.	275(1)
9	The particle of the type is not a valid restriction of the particle of the base.	275(2)
10	Element 'attribute' is invalid, misplaced, or occurs too often.	322
11	There is not a complete functional mapping between the particles.	344(1)
12	The particle of the type is not a valid restriction of the particle of the base.	344(2)
13	Unique Particle Attribution	392
14	Unique Particle Attribution	477
15	Unique Particle Attribution	525
16	Cannot resolve the name 'apd:CitizenNameStructure' to a(n) 'type definition' component.	18

Table A.2: Lists the errors emitted by the binding compilers validating emlcore in EML 3.0

set, this also gives points to oxygen since xjc is its compiler. Scomp has no line numbers for errors, scant information on the error and is, therefore, hopeless for debugging. On the upside, scomp was the only one that managed to compile at all. It seems that the final success of the validation attempt was a result of all of the tools in concert. The errors were discovered using the GUI based tools, especially oxygen (and thereby xjc), but the only tool that could validate it was the worst in debugging. The conclusion must be that these tools are all insufficient by themselves.

Appendix B

Switching to EML 4.0 and the Changes Made to it

B.1 Why EML 4.0 was Chosen.

EML 4.0 permits elections where an affiliation (read “party”) can submit a list of candidates that the voter can vote for, and since the ballot itself also includes a list of candidates, it seems that EML 4.0 can possibly be used to accommodate the list manipulations called for by the Norwegian Election Law.

This was, however, not the case with the previous EML releases. The CandidateList schema (230) raises the fundamental differences in semantics, concerning what an election is, that exists between EML 3.0 and the Norwegian Election Law. EML 1.0 though 3.0 were based on elections where the voters voted for people, that is, individuals. This is not the case in Norway, at least not directly. In Norway, a party hands in a list proposal, which is then evaluated by the appropriate committee, and, if approved, is used to produce the party list which is what is actually voted for in an election. This is a list of candidates representing the given party in this contest.

The Norwegian elections are, however, not as party focused as it may seem. In the different elections the voter is given options to change party lists by for example adding candidates from other parties, removing candidates and/or changing the order of the candidates.

The fundamental difference is, nonetheless, apparent, EML 1-3 was meant for elections where the voter voted for one candidate and not a list of candidates. This semantic interpretation of an election is reflected throughout the design of EML 1.0 to 3.0, making them not naturally tailored for handling Norwegian elections. For this reason, and because of errors in the schemas themselves in EML 3.0 (see appendix A), the focus of this thesis is EML 4.0.

B.2 Changes Made to EML4.0 for Compilation

As opposed to EML 3.0, all the schemas in EML 4.0 do compile individually, with the exception of *emlexternals* discussed below. Together, however, this is not the case. There were three problems:

1. *Name conflicts: EML and result*

   Many of the schemas had a top level element EML, all of which were named EML, although this is not a problem in practice since they do not import each other, it was a problem under compilation. This element would probably be useful if one had an automatic display facility, since this construct provides an XPath field in an element Display-Order. This construct is, however, not used in this thesis. The point is that compiled together this common name became a name conflict and was simply solved by giving them a new name. The element EML in for example 210-nomination-v4-0.xsd was renamed to EML210. In addition, the Result in 120-interdb-v4-0.xsd and 520-result-v4-0.xsd were in conflict. Here the 520 schema seem more entitled to the name Result and the name was changed to Resultdb in the 120 schema, with no real loss of semantics.

```
    error: Duplicate global element:
    EML@urn:oasis:names:tc:evs:schema:eml
    error: Duplicate global element:
    Result@urn:oasis:names:tc:evs:schema:eml
```

2. *410-ballots-v4-0.xsd*

   This schema was not possible to use without using the EML construct above. This schema was altered by moving the "Ballots" element out of the EML element and placing a reference there instead:

```
    <xs:element ref="Ballots"/>
```

3. *Differing names for the external files xAL.xsd and xNL.xsd*

```
    [..]/emlexternals-v4-0.xsd:0: error:
    java.io.FileNotFoundException:
    [..]/external/xal.xsd (No such file or directory)

    [..]/emlexternals-v4-0.xsd:0: error:
    java.io.FileNotFoundException:
    [..]/external/xnl.xsd (No such file or directory)
```

   The files in the unzipped EML 4.0 are called xAL.xsd and xNL.xsd, but in emlexternals-v4-0.xsd they are called xal.xsd and xnl.xsd. In a non case sensitive environment this would not be a problem, here it was. The names of the files were changed to minimize the changes to the schemas themselves.

Appendix C

Prototype for elektroniske valg basert på EML3.0-standarden

Oppgaven går ut på å bygge en prototype for et datamaskinbasert system for gjennomføring av elektroniske valg til Storting, fylkesting og kommunestyrer i Norge. Prototypen skal følge Election Markup Language (EML) standarden, versjon 3.0[1].

Prototypen skal bygges opp i samsvar med den arkitekturen som er vist på Figure 2B - High Level Modell - Technical View på side 14 i dokumentet EMLv3.0.doc som er å finne i ovennevnte zip-fil. Dataene skal overføres mellom modulene i EML-format i overensstemmelse med de XML-skjemaene som inngår i standarden.

Formålet med prototypen er å verifisere at standarden dekker informasjonsbehovet for gjennomføring av valg i Norge. I denne forbindelse står Lov om valg til Storting, fylkesting og kommunestyrer[2] sentralt. I prototypen er det derfor spesielt viktig å utforske eventuelle særegenheter ved den norske valgordningen.

Prototypen skal (foreløpig) ikke brukes til utprøving i brukermiljøer. Det er derfor ikke vesentlig hvordan brukergrensesnittene tar seg ut.

Det skal leveres:

1. En kjørbar prototype
2. En dokumentasjon av prototypen
3. En liste over hvilke skjemaer i EML-standarden som er benyttet, hvilke som ikke trengs, og hvilke endringer i skjemaene det eventuelt er behov for.

17. januar 2005 GS

[1]Standarden er å finne på http://www.oasis-open.org/committees/download.php/10021/EML%20version%203.0.zip
[2]se http://www.lovdata.no/all/nl-20020628-057.html

Appendix D

Concept and Word Clarification

The translation will follow the translations in the Local Government Act (LGA) [12] where possible. In addition, the translation of the Election Law [16] called the Representation of the People Act (RPA) [11] has been used, however, this is an unofficial translation. The translation used is depicted in table D.1.

English	Norwegian
Advance Voting	Forhåndsstemmegivning
County	Fylke
County Council	Fylkesting
County Council Election	Fylkestingsvalg
Election Law	Valgloven
Election Regulation	Valgforskriften
Electoral Committee	Valgstyre
Electoral Handbook	Valghåndbok
Electoral List/party list	Valgliste
Electoral Register	Manntall
Enclosed Document	Vedlegg til listeforslag
Final count	Endelig opptelling
Increased share of the poll	Stemmetillegg
List proposal	Listeforslag
Ministry of Finance	Finansdepartementet
Ministry of Local Government and Regional Development	Kommunal- og regionaldepartementet
Municipal Council	Kommunestyre
Municipal Council Election	Kommunestyrevalg
Municipality	Kommune
Polling card	Valgkort
Population Registry Authority	Folkeregisteret
Postal Code	Postnummer
Postal Location	Poststed
Provisional count	Foreløping opptelling
Referendum	Folkeavstemning
Regulation	Forskrift
Storting Election	Stortingsvalg
Voting circuit	Valgkrets

Table D.1: Translation table

Appendix E

Design of Input/Output in the Prototype

The following four classes were created and put in EML_Interface for three reasons: to give them first hand access to the layer below, to keep such implementation issues from the application, and to facilitate modifying the input/output process if needed.

Each class will be briefly described since they are not a central part of this thesis.

E.0.1 EMLServer.java

EMLServer is here to keep the transmission details from the application and to separate the applications from the compiled schemas. When a part of a subsystem needs to behave as a server it creates an object of this class and delegates the actual sending and receiving of documents to that object.

EMLServer is in this manner a wrapper class for the network transmission code, together with the EMLClient described below. Network transmission code is, by its nature, repetitious and making modifications in only one place is an established programming technique.

EMLServer uses java.nio and the select construct when listening for connections, this was done to allow the server state to be ended by writing "quit". To accomplish this, the select construct was set up to also listen for keyboard input. It does this using another class called SystemInPipe written by Ron Hitchens, author of the book *Java NIO*. This class is not described in the thesis since it was only slightly modified by this author.

The Splitting of Large Messages

A few of the messages sent in the prototype can become quite large, this has to be accommodated in the prototype.

Message	Repeated Element
330-electionlist	VoterDetails
340-pollinginformation	Polling
410-ballots	Ballot
460-votes	CastVote
470-vtokenlog	VTokens
480-auditlog	LoggedSeal

Table E.1: EML messages that can be split

The splitting of messages is commented upon in the EML Schema Descriptions, on page 12:
"When a message is split, each part must be a complete, valid message. This will contain all the background information with a number of the repeated element types. Information in the EML element indicates the sequence number of the message and the number of messages in the sequence. Each message in the sequence must contain the same TransactionId, and must indicate the repeated element according to the table below. Only the messages shown in the table may be split this way."

In the prototype, the above is not implemented. A message is split with no regard to validity, and is rather reassembled, when received, back into a valid message. The above can possibly be achieved by only modifying EMLServer and EMLClient, so that the splitting is not visible from an application. A complete document would then be passed to the server and, if necessary, it is here split and sequence numbers attached. When received, it can be reassembled or kept in it respective parts. The EML element mentioned in the quote above is shown in figure E.1 [4, p. 28].

E.0.2 EMLClient.java

A client application does not present the same complications as a server application, since only one connection is handled at a time. For this reason EMLClient uses java.io rather than java.nio.

E.0.3 EML_io.java

This class is a utility class that mostly deals with files and keyboard input.

E.0.4 Global.java

This class is a container for global constants. Among others the addresses of the servers are here, these are read from a file called *Machinelist* by a method called *setup* when an application is run.

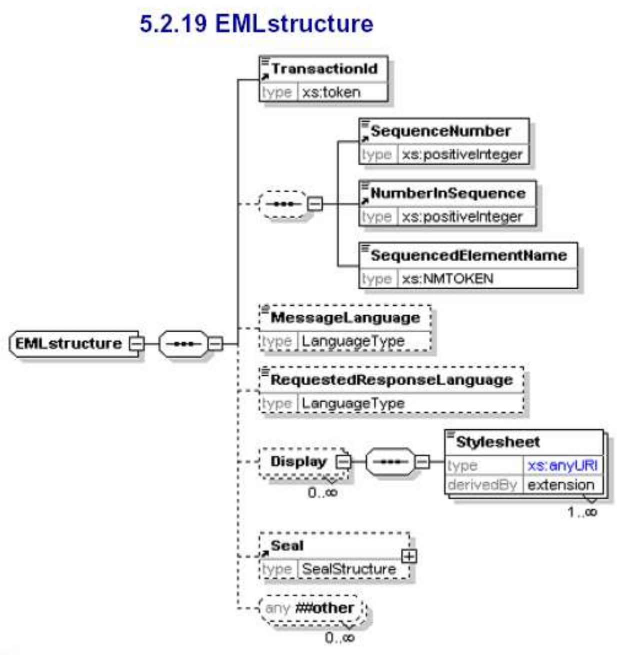

Figure E.1: EML Structure

Bibliography

[1] Oasis "EML 4.0" http://www.oasis-open.org/committees/download.php/11150/EML%20v4.0.zip [Date read: May 20 2005]

[2] Oasis "EML 3.0" http://www.oasis-open.org/committees/download.php/10021/EML%20version%203.0.zip [Date read: May 20 2005] http://xml.coverpages.org/EMLv30.pdf [Date read: May 20 2005]

[3] Mertz, David "Article on EML (v3)" http://grouper.ieee.org/groups/scc38/1622/email/msg00000.html [Date read: May 20 2005]

[4] Ross, John, Paul Spencer, John Borras and Farah Ahmed "EML Schema Descriptions Version 4.0 24. January 2005" http://www.oasis-open.org/committees/download.php/11150/EML%20v4.0.zip [Date read: May 20 2005]

[5] Ross, John, Paul Spencer, John Borras and Farah Ahmed "EML Process & Data Requirements" http://www.oasis-open.org/committees/download.php/11150/EML%20v4.0.zip [Date read: May 20 2005]

[6] Office of the e-Envoy "EML: Customisation for UK Local Elections Version 1.0" http://xml.coverpages.org/EML-UKv1.pdf [Date read: May 20 2005]

[7] Office of the e-Envoy "EML: Customisation for the UK Version 2.0" http://www.govtalk.gov.uk/documents/EML%20UK%20Localisation%202003-12-19.zip [Date read: May 20 2005]

[8] Spencer, Paul "Report on Alternative methods of EML Localisation Version 1.0" http://lists.oasis-open.org/archives/egov/200401/pdf00000.pdf [Date read: May 20 2005]

[9] Spencer, Paul "The Election Markup Language" http://www.idealliance.org/papers/dx_xmle03/papers/03-05-07/03-05-07.pdf [Date read: May 20 2005]

[10] Spencer, Paul "The Election Markup Language" First draft for comment 23/3/2004 http://lists.oasis-open.org/archives/election-services/200403/pdf00000.pdf [Date read: May 20 2005]

[11] "Representation of the People Act" http://www.ub.uio.no/ujur/ulovdata/lov-20020628-057-eng.pdf [Date read: May 20 2005]

[12] Ministry of Local Government and Regional Development "Local Government Act" http://odin.dep.no/filarkiv/237921/Local_government_act2005.pdf [Date read: May 20 2005]

[13] Ministry of Local Government and Regional Development "FOR 2003-01-02 nr 05: Forskrift om valg til fylkesting og kommunestyrer (valgforskriften)" Replaced by [20] in April 2005 [Date read: Jan 27 2005]

[14] Ministry of Local Government and Regional Development "FOR-2001-04-02-441: Forskrift om utførelse, trykking og utsendelse av stemmesedler til stortingsvalg, sametingsvalg og kommunestyre- og fylkestingsvalg." http://www.lovdata.no/for/sf/kr/kr-20010402-0441.html [Date read: May 20 2005]

[15] Ministry of Local Government and Regional Development "FOR 2003-01-02 nr 05: Forskrift om valg til Stortinget, fylkesting og kommunestyrer (valgforskriften)." Sist endret FOR-2005-04-19-323 http://www.lovdata.no/cgi-wift/ldles?doc=/sf/sf/sf-20030102-0005.html [Date read: May 20 2005]

[16] Ministry of Local Government and Regional Development "LOV 2002-06-28 nr 57: Lov om valg til Stortinget, fylkesting og kommunestyrer (valgloven)" http://www.lovdata.no/all/nl-20020628-057.html [Date read: May 20 2005]

[17] Ministry of Local Government and Regional Development "Lov om endringar i lov 28. juni 2002 nr 57 om valg til Stortinget, fylkesting og kommunestyrer (valgloven)" Norsk Lovtidend avd I nr 4 2005 http://www.lovdata.no/ltavd1/lt2005/t2005-1-04-62.html [Date read: May 20 2005]

[18] Ministry of Local Government and Regional Development "Valghåndbok" Updated 06.04.2005 http://www.dep.no/filarkiv/242998/Valghandbok_05_samlet_versjon.pdf [Date read: May 20 2005]

[19] Ministry of Finance "LOV 1970-01-16 nr 01: Lov om folkeregistrering" http://www.lovdata.no/all/hl-19700116-001.html [Date read: May 20 2005]

[20] Ministry of Finance "FOR 1994-03-04 nr 161: Forskrift om folkeregistrering" http://www.lovdata.no/cgi-wift/ldles?doc=/sf/sf/sf-19940304-0161.html [Date read: May 20 2005]

[21] Liburd, Soyini D. "AN N-VERSION ELECTRONIC VOTING SYSTEM" VTP WORKING PAPER, July 2004 http://www.vote.caltech.edu/media/documents/vtp_wp16.pdf [Date read: May 20 2005]

[22] Recommendation Rec(2004)11 and explanatory memorandum Adopted by the Committee of Ministers of the Council of Europe on 30 September 2004 "LEGAL, OPERATIONAL AND TECHNICAL STANDARDS FOR E-VOTING" http://www.coe.int/T/e/integrated_projects/democracy/02_Activities/02_e-voting/01_Recommendation/Rec(2004)11_Eng_Evoting_and_Expl_Memo-3.pdf [Date read: May 20 2005]

List of Figures

List of Tables