Creating Markup
Exploring the concept of users defining syntax

Matthias Van den Weghe

Department of Informatics and Media
Uppsala University
June 2016
Abstract

A variety of markup languages exist for formatting text and exporting to HTML. These languages are tailored to the needs in a specific context by specialising tags, selecting tags and limiting the number of possible distinctions to a subset of what is available in HTML. However, limiting the number of possible distinctions creates problems when changes occur in the context. The real world is ever-changing, thus that which models it must be able to reflect the changes in the operational environment to remain relevant and satisfactory. Incorporating new requirements and adjusting to the changes in requirements means adapting and evolving.

This thesis explores giving document authors the possibility to extend and modify the repertoire of available markup tags when new user requirements demand it. What is presented is a prototype which allows the user to tailor the markup and also adapt it to changes in the environment. The system allows users to create their own set of markup tags, annotate their documents with them, and export a generated XML document. Users create the tag and assign a meaning to it, when changes occur in the requirements they can be implemented by modifying the tags, extending the repertoire by adding tags, or changing the meaning of a defined tag.

**Keywords:** software maintenance, software evolution, information system change, formal grammar, markup languages, user-defined syntax, XML

I would like to thank Christopher Okhravi for supervising my work.
List of Figures

3.1 Software maintenance and evolution types [9] . . . . . . . . . . . . . . 10
3.2 Staged Model[3] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
3.3 Versioned staged model[3] . . . . . . . . . . . . . . . . . . . . . . . . 13
4.1 Nesting Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
5.1 System Flowchart . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
5.2 Annotation of Text . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
5.3 ANTLR 4 Parse Tree . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
5.4 Element Tree . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
6.1 Scenario A - Parse Tree . . . . . . . . . . . . . . . . . . . . . . . . . 30
1 Introduction

Software maintenance is part of the life cycle of any information system which has been successfully implemented [14, 9]. The process of maintaining a system should not be viewed as a process of simply keeping the system running and users able to gain value by using it. Any system which operates in a real-world setting must be able to evolve as a response to changes in its environment. As the world is ever-changing, the system must also be ever-changing or it will become less and less satisfactory and finally obsolete. Once a system has been developed and implemented users will start requesting modification and addition to the functionality to make the system more satisfactory for the work it supports. External changes such as new and improved technology can be developed and may be considered for the system. Finally, any change to business models must also be reflected in the system [17, 19].

Information system change is a process of creation and modification of elements and their connections, where the technological aspect is not the only dimension to consider [8, 20]. These changes can relate the organisations technological core, processes, structures, tasks and the symbols it employs [20]. Software needs to be flexible in order to rapidly and efficiently be adapted to changes in business needs [30, 3]. The continued development of a system to respond to changes is a software maintenance activity. Software maintenance is the process of modifying a software after implementation to fix errors, improve it, or adapting it to a modified environment [16].

Following the classification by Lientz & Swanson [19] Software maintenance work can be divided into four activities: Modification to the system as a response to a changed environment (adaptive). Enhancing functionality because of new or changed user requirements (perfective). Finding and correcting errors and bugs (corrective). And finally changing the software to avoid or limit future problems with maintainability (preventive) [19].

The process of changing the system after initial development can be viewed as simply software maintenance. However, a distinction must be made when changes are made to the system to optimize it as a response to changed requirements from the user or changes in the operational environment [4]. Changes which are made as a response to forces external to the system lead to evolution of the system. Software evolution is a term used for this process of making changes to optimize the system to these changes [4, 17, 9]. While the system is in a stage where it can continue to evolve it can also adapt to change, but there are challenges with this process as Lehman’s laws of software evolution have described [17, 18, 14, 3].

Lehman’s laws of software evolution have shown that systems must be continually adapted or it will become less satisfactory. During the system-lifetime it will continually grow, functionality is added to satisfy new requirements. As the system evolves and grows it will also become more and more complex unless work is done to maintain or reduce complexity. While the system is evolving everyone associated with the system must keep track, or “maintain mastery” [18], of the system but increasing size and complexity of the system makes this harder. As the system evolves the quality will appear to decline unless rigorously maintained [17, 18].

A system will not generally be able to continue evolving indefinitely, but factors such as system architecture and a skilled team can make it possible to evolve for a longer time. A suitable architecture is important when designing and developing a system, meaning the components which make up the system, their properties and how they interact [4, 3, 5]. Flexibility is important as future business needs and user requirements are not predictable [20, 3]. An architecture which is suitable and developed by a skilled team, where certain future needs have been taken into consideration during design, will still not be sufficient. The changes a system will undergo during its evolution can not be planned and the initial design can not
accommodate it\cite{5}. This stresses the importance of flexibility.

Maintaining architectural integrity is key for an evolving system, however as the system evolves the changes it undergoes lead to a gradual erosion of the original architecture, and over time turnover of personnel will lead to loss of system and problem domain knowledge\cite{4,5}. Loss of architecture and skilled personal, growing system complexity and size, and loss of quality and familiarity with the system are all contributing to the system arriving at a point where it can no-longer evolve because the skills are not in the team and the cost would be too high\cite{17,5}. After this point the system does not continue evolving and instead enters a stage or service where changes to the system are kept to a minimal. Adaptive and perfective changes are no-longer done, and preventive changes are not done as it cannot be motivated\cite{4,5}.

1.1 Markup

In a world where there are so many different markup languages to choose from, it is up to the developers to weigh the pros and cons and evaluate the language based on their requirements and the nature of the project in which it will be used. Decisions taken in the design-phase will have far-reaching consequences for the the system in the future. Change is inevitable, business changes lead to new system requirements and implementations can be improved as new technologies are developed\cite{28,20}. If a decision is made as to which language suitable for a project, it may not be the case that it is suitable tomorrow. Even a relatively simple system such as one for the creation of a website must be designed in a way which makes it possible for evolution and extension. Having the possibility to implement future user-requirements, which cannot be foreseen at the time of initial design and implementation, is necessary to maintain satisfaction\cite{17}.

HyperText Markup Language (HTML) is the language of the world-wide-web\cite{15}. Developers are however not necessarily writing HTML themselves, but can use a variety of different markup language which produce it. Developers can choose to use eXtensible Stylesheet Language Transformations (XSLT) to transform documents written in Extensible Markup Language (XML) into HTML\cite{10}. Developers can also choose to use for example web template engines such as Jade\cite{1} or text-to-HTML conversion tools such as Markdown\cite{2}.

Consider this simple example of HTML code.

```html
<!DOCTYPE html>
<html lang="en">
  <body>
    <h1>Example</h1>
    <p>I am now <b>bold</b></p>
  </body>
</html>
```

Listing 1.1: HTML Example

---

\footnote{1}Jade Template Engine http://jade-lang.com
\footnote{2}John Gruber’s specification of Markdown http://daringfireball.net/projects/markdown
The following code can be written in Markdown to get the HTML code seen in seen in Listing 1.1:

```markdown
# Example
I am now **bold**
```

Listing 1.2: Markdown Example

The same example written in Jade:

```jade
doctype html
html ( lang="en ")
  body
    h1 Example
    p I am now
      b bold
```

Listing 1.3: Jade Example

Jade is whitespace sensitive, meaning whitespace is syntactically significant. New-line is used to denote end tag and indentation for nesting elements.

GitHub Flavored Markdown[1] is used on GitHub, an online platform for development collaboration and system version control using git repositories. Users are able to upload their code projects to repositories where collaboration can take place. Users are able to give feedback on the code by posting issues found with the code and make pull requests to the repository if they have advanced the code. Communication between users is done by posting comments on created issues and pull requests which makes is possible for users to discuss problems and solutions. It is possible to format the text in comments and in readme-files in code repositories, and as these will be published on the website they will be in HTML. The users themselves are however not required to write in HTML but format their text with GitHub Flavored Markdown (GFM). GFM is a variation of the original Markdown language, specialised for the use on the website. Users can format their plain text with headings, bold text, lists, tables, and other basic formats. GFM also includes more specialized features such as highlighting code in different languages, mentioning other users in a comments which notifies a user, and also referencing issues and pull requests which creates a hyperlink between. GFM is a markup language developed for a specific use, simple formatting of text on GitHub and communications between users when collaborating. The language includes basic features for formatting text but also more advanced features which are useful in this specific context.

An example of text formatted with GFM: Typing # followed by the id of an issue creates a link to the issue posted in a repository and at sign creates a link to the page of a given user (mv00 in this case).

Read this issue #2 posted by @mv00

Listing 1.4: GitHub Flavored Markdown Example

---


3
Wiki Markup[^1] is a language developed for the writing of wiki-based websites (such as Wikipedia). Like Markdown and GFM it is a language which aims to make the creation of HTML easier and faster by employing defined tags which are then translated to HTML. It includes tags which a commonly used in the context to format text when creating wiki-entries. Wiki Markup has tags for layout (sections, align text etc), format (bold, special characters and mathematical symbols), links, images, tables, and referencing. The aim of the language is to make content creation easier by eliminating the need to write everything HTML. Wiki Markup also supports a subset[^2] of HTML tags, but users are advised to avoid HTML as much as possible.[^3]

* ''Unordered ''Bullet list
*# Nested ''ordered list ''
*# With another element

Listing 1.5: Wiki Markup Example

Output:

- *Unordered* Bullet list
  1. Nested *ordered list*
  2. With another element

Markup can be used instead of HTML to make it easier to create HTML documents by eliminating the scope of the language and focusing on a smaller set of tags which make it possible to express what is commonly used. Why have HTML with its full scope when that means including tags which are have no use in the context. A smaller set with an easier syntax make it easier to produce content and still retains the possibility to express differences in the text where it is needed.

The use of markup languages allows for more efficiency when writing content by specialising the tags for the context. The syntax can make tags easier to declare than in HTML and it also allows for creation of custom elements such as the examples seen in GitHub Flavored Markdown.

Any of these solutions can be a good choice given a certain design problem, what must also be taken into consideration is that it will have implications and their own set of constraints on future development. Given a scenario where a website is to be built where the content is not prone to change it can be suitable to not invest time in adding technologies for HTML-generation and write everything in HTML instead. If the content is prone to change it can be suitable to invest time to have a different markup language for document notation and HTML-generation as it has syntax which are faster to write and allows for easy modification of the content.

As an example: GitHub Flavored Markdown contains a smaller set of formatting options which is suitable given a certain context. However its is more restricted in the variety of tags and therefore what can be expressed and which distinctions can be made between different types of content. Attention must also be paid to work required to modify text once it has been written. It may be suitable to have markup-language such as Jade which will require more time to modify but does not have the same constraints as Markdown. This is covered more extensively in section 4.1 and 4.2.

Whatever language is chosen for implementation it will also have its own set of consequences. But some consequences can arise in the future, as a result of changes in the user-requirements or the operational environment which could not be foreseen, and can hinder the evolution of the system if the requests are not implemented (further discussed in section 3).

1.2 WebbRaket

While there are many examples of the impact of change, in this section one case will be presented showing problems encountered which are due to change and how they could be overcome. Webbraket, which the author is part of, is an online open source resource for learning web development. The aim of the project is to give readers an introduction to web development and understanding of fundamental concepts such as for example HTML, JavaScript CSS and jQuery. The learning material consists of a combination of text, video, images and code examples, and readers are encouraged to start writing their own code to get a better understanding. Contributors to the project can choose to write new or improve existing material, or to continue development of the underlying infrastructure. Developers are free to come and go, decide how much work they want to do and which aspect of the resource they want to work on. Anyone is allowed to improve existing sections, extend them, or start working on something completely different related to web development. All the written content is kept in different folders (for different chapters) and files (for different sub-chapters). This makes it easier to keep track of and make changes to the text.

Markup languages are used to manually annotate the text to generate HTML and deploy to the website. Jade was used in earlier stages but was abandoned due to its syntax being too tedious when content has to be modified. The syntax, using newline and indentation, resulted in it requiring too much work when modifying the code to change the text, code examples etc. Markdown, which has a syntax which makes it easier to modify content, was implemented instead. As the project moved forward and the content expanded into different domains of web development, the developers experienced that Markdown did not offer enough flexibility and variety of tags to represent the text, and are currently discussing abandoning Markdown in favour of a different markup language. Once again abandoning the current language will of course mean a lot of work, since every file with text has to be modified to remove the old tags and replace them with new. This has led to a situation arising where developers feel they are locked in and forced to use the current language. On the one hand they are not completely satisfied with the current language, but on the other hand a change would require investment of time from the developers in the hope that the new language will improve the situation.

The scale of WebbRaket made content harder to handle, as the content continues to grow it becomes more time-consuming to modify existing text. As in other open source projects, anyone is able to make contributions. In WebbRaket they are also allowed to expand into other areas within the domain of web development. In practice this means being able to modify and extend existing text, for example rewrite existing text, extend it to include more advanced learning examples, adding new sections for technologies which are not yet included, and so on.

What can be gathered from this example is the experienced problems lie in having a markup language which cannot represent text in a variety of ways depending on the subject. While this was not a problem in earlier stages, the expansion of the project meant that the available number of tags was not the same as the number of distinctions to be made in the text. The used markup languages are seen as too rigid and create inefficiency. Migrating to a new language is a time-consuming task as all content has to be updated with the new tags.

The markup language, which is relied upon, can not handle the changes in the project. There are no actors or events external to WebbRaket which influence this process. The experienced inefficiency arises from internal changes to the system. This problem is a software maintenance issue as the problem arises due to environment changes in and around the software, creating new user requirements, and modification of the system is needed to resolve it.

---

7 www.webbraket.se, GitHub repository https://github.com/webbraket
2 Problem Outline

What is proposed as a way to solve the experienced problems in WebbRaket is a system which will make maintenance activities easier to perform. The maintenance work in WebbRaket is concentrated around the modification of documents annotated with markup (currently the language Markdown). The goal of the solution will be to make these modifications easier to implemented and to not limit it by what can be expressed when modification is needed. The system will make developers unbound to the existing syntax of the language and able to define, modify, and extend it themselves. This would make the markup language itself very flexible, and extendable to be able to represent any element which is needed at any point in time.

Lehman’s law of continuing change states that systems must be continually adapted if they are to stay satisfactory (further discussed 3.3). Continuous change cannot be controlled or limited through maintenance work like declining quality and increased complexity can be. But flexibility makes change easier to handle. As discussed in the introduction, a system which is flexible makes it easier to respond to changes in business needs and adapt the system to it. Flexibility is valuable as future business needs cannot always be predicted and user requirements can arise which were not possible to foresee during initial design and development (discussed in 3.2).

2.1 Problem Definition

The markup languages discussed in section 1.1 are examples of languages designed for use in a specific context. They allow for a simplified syntax and the number of tags available reflect what is needed in the context. What is shown in section 1.2 is while there are benefits to this approach there are also drawbacks. The software and the operational environment will undergo change during the maintenance-phase in the life-cycle which are not always possible to predict (discussed in section 1). While the language used can be optimal at one point, changes in the context (as the WebbRaket-case shows) can result in the language no longer being satisfactory. Scenarios can arise where the number of tags available are less than the number of distinctions to be made or a desired representation having none suitable. The identified problem to be solved is how to adapt the language to a changing context.

2.2 Proposed Solution

The proposed solution to this problem is to give writers of documents the possibility to define their own syntax. The solution utilises XML, but XML is strictly generated and not written by the author of content. The argument could be made that a simple migration to full XML would solve the need for flexibility. It would however not solve the issue of decreasing the actual work required to perform maintenance activities (further discussed in section 5 and 6). In the same sense that developers avoid the inefficiency of writing in HTML, and instead use other markup languages to generate HTML, this solution will generate XML which can be used to produce HTML. As the product is XML, it is not only limited to producing HTML but can be used to for example export and import data between systems or other tasks which are not currently needed in this specific context. A flexible system which can be used for multiple purposes is still preferred. Even though WebbRaket’s intended use of the system is generation of HTML, future changes in business rules can change the requirements of the system.

The goal of this thesis is formulated as:
Give users the possibility to modify and extend syntax to support the need for flexibility and adaptation.
Markup has been used to avoid having to write HTML and in a similar sense is now used to avoid writing XML. The project will no longer be tied down the use of a specific markup language such as Markdown or Jade. The future is uncertain and the future needs are difficult to predict (see section 1). Uncertainty, in terms of future requirements, is countered with flexibility through the use of a user-defined language where users can modify and extend the language when needed.

2.2.1 User-defined Syntax

The language must define what can be and cannot be expressed and should therefore rely on formal grammar. Extended Backus-Naur Form (EBNF), notations used to make a formal description, can be used to describe the language. A description written in EBNF can be used as input to a compiler-compiler, in this case a parser generator. The parser is used to perform lexical and syntax analysis of input and the result is a parse tree or abstract syntax tree (AST). This abstract syntax tree which is the representation of the source code (input string to the parser) can then be used to create an XML-based representation.

EBNF is a language of notations which are used to define the grammar of a language. In it a language can be constructed by defining a set of terminal symbols (lexer rules), non-terminals and production rules. Production rules and non-terminals are used to recursively break down any input until the terminals are found and thus control what can and cannot be expressed in the language.

2.3 Method

A design science (DSR) approach is suitable since the aim of this thesis was to develop and implement new technological artefact. Hevner discusses DSR as containing a relevance (“why”) and rigor (“how”) cycle. Relevance stems from environment (people, organisations and technology) which creates a business need and rigor is the applicable knowledge base (theories, frameworks, instruments etc.) and methodologies (data analysis techniques, validation criteria etc). Theories and artefacts are built, justified, and evaluated from this. The relevance of this research has been motivated by earlier chapters, the rigor will be illustrated in the following chapters. This study has been conducted following the iterative five steps process proposed by Vaishnavi & Kuechler: Awareness is the articulation of the problem to be solved. Suggestion, is the formulation an idea of how the problem can be solved. Development, constructing and implementing the artefact. Evaluation, the artefact’s worth is assessed and deviations from what is expected. Conclusion, identified results which can be explained, and results which cannot yet be explained, are written up. The creation of the artefact should also be documented by a combination of text, models, diagrams, and code to create traceability between the different steps of the system development life cycle.

This work is built on theory of information system change, flexible software, software maintenance, and the nature of markup languages. Building on the mentioned theory the aim is that the developed artefact will enable developers to more easily modify the system by defining their own syntax to allow for more flexibility as a way to tackle changes in their development project. Development was be done with an established system development methodology, Test-Driven Development (TDD). The chosen methodology was suitable as it was important to do develop a simple design in small steps and not increase complexity too fast. Developing by letting tests drive the project forward is a way to do this, and testing is required in order to have confidence in the software. Developing with a focus on testing is also suitable since evaluation of the actual software will rely upon it.
2.4 Evaluation

As the discussed earlier the problem revolves around flexibility. The application was evaluated based upon if flexibility, the capability to modify and adapt syntax, was achieved. This evaluation also relates to the software engineering principles of verification and validation [7, 6, 28]. Verification, confirming that the system meets functional and non-functional requirements, was done through the use of automated testing during development. Validation, checking that the system meets expectations, was done through confirming that the expected flexibility and ability to adapt was attained (possibility to modify syntax). This is covered in section 5 and 6.

2.5 Execution

The first step was a review of relevant literature to survey the areas mentioned. This was followed by development of a first version of the solution, with a grammar for the lexer and parser to operate on. Following the initial review of literature and a first version of the prototype (which could parse a grammar) iterations of development and literature gathering were conducted. The gathering of literature was stopped once saturation was reached and literature encountered conveyed viewpoints already encountered. Focus was then shifted to development, the next step being implementing the ability to modify and extend the grammar to allow for modification of syntax. Once modification was possible the next step was introducing the ability to export documents in XML format. Finally evaluation was conducted and conclusions drawn and written up, constituting this thesis.
3 Software Maintenance

This chapter explains the processes of software maintenance and software evolution and how they are vital and unavoidable processes for any successful embedded system if it is to stay satisfactory. Lehman’s laws of software evolution offer an explanation as to why the system must be continually adapted to its environment and why this work tends to become harder and more time-consuming over time. The staged model which describes the life-cycle is explained and how it can be used as a way of explaining the work of evolving and maintaining a system once it has been developed until it is closed down.

3.1 Maintenance & Evolution

Software maintenance, which is a part of the system development life-cycle, is the modification of code and documentation after the software has been developed and delivered. IEEE defines software maintenance as: “The modification of a software product after delivery to correct fault, to improve performance or other attributes, or do adapt the product to a modified environment”[16].

The software maintenance activities, defined by Lientz & Swanson [19], are traditionally categorized into four different classes: corrective, adaptive, perfective and preventive [14, 4, 9].

Adaptive maintenance is change done to the system in order to respond to changes which have occurred in the system environment[4]. This type of software change includes activities and processes such as moving the system to new hardware, different platforms, changing the database. An example of this type of change is when governments in the European Union changed their currency to Euro, this lead to requirements for system change in for example banking [14].

Perfective maintenance describes changes done because of change or expansion in the user-requirements. Software which is successful will have changes in the user-requirements as a result of the use of the system and users exploring it. This creates an increase in the requirements for change and improvement. An example of perfective change would be users calling for new functionality or improvements on what exists [4, 14].

Corrective maintenance is related to fixing software errors and other defects. This can be errors which existed previously in the code or bugs which were introduced as a result of incorrect or incomplete changes during maintenance. A defect can result from a design error (incorrect or incomplete changes to the software), logic error (incorrect implementation of design specifications, incomplete testing of data) or coding error (incorrect implementation of design logic or use of code logic) [4, 14].

Preventive maintenance is done in order to improve the maintainability of the system, prevent problems in the future, and the problems of deteriorating structure as the software continues to evolve. Preventive changes can be the restructuring and optimisation of code, which can be compared with refactoring[27]. Making small changes to the source code can increase the complexity of the code, making it necessary to perform preventive maintenance [4, 9, 13].

Previous research has shown that adaptive and perfective maintenance make up 75% of the maintenance efforts and correcting errors took up 21%, they have also shown that adapting to new user-requirements is one of the most important problems of software maintenance [4].
In 2001 Chapin et al. proposed a new classification of the software maintenance activities. What is proposed is a more detailed classification based on what type of maintenance is performed, where it is performed and how it affects the functionality which the user experiences. As an example a maintenance activity is only classified as adaptive if the software was changed by changing the source code while not changing the experienced functionality. Activities classified as adaptive are for example changing database technology, changing the system’s interoperability. To generalise, activities which change the software properties or characteristics without changing what the user experience in terms of functionality. Chapin et al. identified four different type clusters in the maintenance activities: support interfaces, documentation, software properties and business rules (see Figure 3.1). Support interfaces involves maintenance done without changing the software. Activities in this cluster involve evaluating the software, stakeholder training and consultation. Activities in the documentation cluster are those which alter the software but without changing the source code, such as reforming the documentation to comply with user needs or updating it because of new functionality. For the sake of this thesis only the two unmentioned clusters will be explored in more detail.

If the software was changed and it involved changing the source code, according to Chapin et al., we arrive at the question of whether or not the change had an impact on the user-experienced functionality. If user-experienced functionality was changed then the change is classified as within the business rules type, otherwise within the software properties type.

Object properties cluster contains activities where the software was changed by changing the source code without affecting the user’s experienced functionality. This cluster contains four types: groomative, preventive, performance and adaptive.

Groomative activities aim to improve maintainability or security, for example changing code comments and naming conventions, making changes to improve code readability and improving components and algorithms by making them more elegant. These activities are however not aimed at improving system performance. Preventive activities, similar to the definition by Lientz & Swanson, concerns efforts taken to improve the system to reduce or completely avoid future maintenance work. But following Chapin et al.’s definition requires that the activities not change the current user-experienced functionality or existing technology.
This type of activity can be done following a schedule. However since this type of maintenance requires forecasting what the system might need in the future and forecasting can be imprecise, work that is done as preventive can be classified differently (groomative, adaptive or enhancive). Performance activities are those which are undertaken to improve some property of system performance, such as execution speed or memory usage, implementing faster algorithms, and increasing system robustness and reliability. Like preventive activities this is sometimes scheduled. Finally, adaptive type activities are those which alter the technology or resources used. Activities such as adding more supported platforms, changing supported protocols, changing design and implementation practice are adaptive activities. Like other types within this cluster, activities are done in a way which changes software properties and characteristics but preserves the user-experienced functionality [9].

**Business rules cluster** contains three types of activities: reductive, corrective, and enhancive which is also the default type [9].

An activity is deemed reductive if it restricts or reduces the user-experienced functionality. It involves limiting or removing functionality from the system such as removing components or subsystems. Corrective activities deal with fixes to the experienced functionality and making it more correct. It involves making improvements on existing business rules by for example adding more support for handling exceptions through making internal logic more precise. The final type, enhancive, is activities which alter the experienced functionality by replacing, adding or extending the current functionality. Examples of this type of activities are adding new components and subsystems, or extending existing ones [9].

Activities within the business rules cluster are frequent and among the most significant in the process of software maintenance and software evolution work, these types of changes also rely upon the use of supportive activities within the other clusters [9]. Consider an example of a decision being made to remove module X from the system because it is no-longer used or needed by the customer. This activity would be considered of the reductive type as it result in the removal of a module which the customer can use, thus altering the customer-experienced functionality. However, even though the main activity is reductive it will also rely on other activities such as altering the source code to remove the no-longer needed module and updating the documentation [9].

### 3.2 Staged Model

Software evolution and software maintenance can seem similar but the terms should be kept apart. Software maintenance is done to correct faults, improve attributes such as performance, and adapt the software to a modified environment [16]. Software evolution also takes place during post-delivery. The goal is to adapt the software to make it optimal during changing user-requirements and operational environment [4]. Software evolution takes place when maintenance work is done which can be classified as any of the types in the business rules cluster (enhancive, corrective, reductive), or when changes affect software properties which are seen as sensible to the customer (adaptive, performance) [9]. Maintenance work done within the documentation and support interface cluster is not considered software evolution as it does not advance the software per se. Preventive and groomative work is not seen as software evolution as it does not done to deal with changes in user-requirements or operational environment.

Software maintenance and evolution can be explained with the Staged model [3, 4] or the versioned staged model [3, 4], see Figure 3.2 and Figure 3.3. In the first stage the software is developed, meeting initial demands, and the first functioning version is released. Important aspects to consider in this stage are software team expertise and software architecture.
During initial development the development team increase its knowledge of the problem domain which is critical expertise in future iterations. The architecture, the systems components their properties and how they interact, can make future iterations easier or hinder them. This is followed by a transition to the evolution stage. During the evolution stage developers start adding, removing and altering functionality [3].

![Staged Model](image)

The goal is to keep the system in a state of evolution as long as possible by maintaining an appropriate architecture and skilled team. However, making substantial changes to the system without compromising the architecture requires a system architecture which is coherent and software team knowledge, as the software continues to evolve it give rise to increasing complexity and loss in architecture coherence. This means it becomes harder to take advantage of the design of the system, to see constrains and how to improve the architecture. It requires significant expertise to identify when a change to the system will have big impact on the system architecture, negative or good. This knowledge which the team has is hard to document, and over time as more members leave the team this knowledge is also lost. System servicing is entered when evolution of the system has reach a point of being to complex and expensive, at this point the team only does minor changes to the system, however each change will still further degrade the architecture as the necessary coherency and system flexibility has been lost [3, 4].

The Phaseout stage occurs when the company decides to not continue servicing the software. There may still be users of the software but no more changes are made to it. At this point the company tries to generate revenue from the unchanged software for as long as possible. At this point it is also hard to return to a service stage because of the growing backlog of requests for changes. Finally the closedown stage is entered where the company decides to shut down the system, source code and data is retained, and its users are directed to a replacement [3].

An alternative way of viewing the evolution of software is by looking at the versioned staged model. In this model the team works towards new versions of the system rather than conduct maintenance for as long as possible in the evolution stage before moving to the servicing stage and onwards. Following this model, the team make changes with high impact on architecture and once these type of changes are not possible they move on to the servicing stage, phaseout and closedown. When closedown is reached the current version is abandoned and the team continue work on a new version of the software [3].

Once initial development is done and the software is in the evolution, developers will have to respond to user requests. As user requests are difficult to predict once users start exploring the software it is important to design the system to allow flexibility to a high degree. If the system is flexible it can allow for more implementation of new requirements without degrading the architecture at a high rate, allowing the system to keep evolving and not force a move to the servicing stage [3].
3.3 Lehman’s laws of software evolution

Lehman’s laws can be viewed to better understand of the nature of software evolution, why maintenance is conducted, and how the Staged model can be used to understand the system life cycle in terms of software evolution.

Lehman’s laws show how the evolution of software tends to occur, explain why it becomes increasingly hard to keep evolving a system, and what can be done to continue the evolution for as long as possible before the software becomes unmanageable [17, 18]. Lehman divides software into three categories: S-type, P-type and E-type. Lehman’s laws can be applied to the E-type system [17].

S-type (static) programs or software are those with a function which can be defined and derived from a formal specification. The specification does not change it and the problem is understood by the user. Consider a system which is used to perform mathematical calculations. The problem which the system solves can be defined in a formal specification and is not prone to change unless the problem changes. In those cases it is not a matter of adapting the system to change but rather of designing a new system which solves a new problem [17].

P-type (practical) systems are those which have a theoretical solution but the implementation is impractical or even impossible [5]. Consider a program which is used to play chess. Operations and procedures of the program can be completely specified as the program is bound to the rules of chess. The program must analyse the current game being played, which moved are possible and select the next move. To select the next move it must have a decision tree to determine which move is best. The solution would be to analyse all possible positions which can occur from the current state and then evaluate which move is best to make. This solution is however not preferable, the tree structure containing every possible position which can arise would be too large to construct and analyse. If a complete solution can not be implemented, an approximate solution must be developed which is practical [17]. An example would be a program which looks ten moves ahead and from this decides which move to make.
E-type (embedded) are programs which are used to mechanise a human or societal activity. These programs model a process which occurs in the real world and are also part in the process [17]. Consider a program which is used by traffic control at a railway company to organise timetables for trains, and correct for late trains. The program is a component of the world it models, hence the name embedded.

Creating a system of this sort involves analysts determining requirements, design, consequences of system introduction and so on. Views based on opinion and judgement are combined to create a model and a program. Once these programs begin being used questions of correctness, appropriateness, and satisfaction will lead to request for change. As users become more experienced with the system they will also change their behaviour to minimise effort and maximise effectiveness, which will lead to demand for system change. And finally, changes in the application environment where the system operates will also lead to request for change. As the real world is ever-changing, the model must continue to evolve to stay relevant [17].

Since S-type programs have an understood problem and solution they are not prone to many or radical changes. Changes in these type of systems tend to be related to improving performance and replacing algorithms with improved ones, or making the source code more elegant [17][18]. P-type systems can not offer a complete solution which is practical even if the problem is theoretically known and agreed upon. An approximation has be made and will on some way reflect the viewpoint of analysts, which also opens up for an interpretation of value and validity of the solution. Differences can occur in the problem perception, formulation and its model, as well as in the specification and implementation of the program. P-type systems are very likely to undergo never-ending change, but they are not prone to change to the same degree as E-type systems [17]. E-type systems have a problem which has to be interpreted, a model for the solution which is also interpreted and its setting is in the real world which is constantly changing. This makes this type of systems prone to change [17]. From empirical observations of systems, Lehman derived laws which software evolution conform to [17][18][5]. In total eight laws have been formulated, five of them will be discussed in this paper as they are most relevant for the context.

Lehman’s Law of Continuing Change  E-type systems must be continually adapted else they become progressively less satisfactory.

As mentioned earlier all E-type systems operate in a real-world environment. The operational environment, for example new regulating laws, new technology, user behaviour and opinions, and so on, will keep changing so therefor the system must also change.

Lehman’s Law of Increasing Complexity  As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.

When the system is evolving the developers will change the system, adding new components and features, changing existing ones and removing others. This law states that the complexity of the system will continue to grow while it evolves unless activities such as for example groomative maintenance is done. A growing complexity has implications for the maintainability of the system, which can be seen by looking at the following law. If the complexity keeps growing the system will arrive at a point where it can no-longer evolve because familiarity can not be conserved.

Lehman’s Law of Conservation of Familiarity  As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behaviour to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
As mentioned in section 3.2 in order to keep the system in a stage where it can continue to evolve the team must have skill and expertise to be able to maintain the system and the architectural integrity must be kept. This means that people who are involved in the system in some way must be aware of how the system is changing. If the system grows too fast and becomes too complex it will be harder for people to keep track of the system, if they are not “masters of the content and behaviour” as Lehman puts it they will not be able to continue evolving the system in a satisfactory way. A team which does not have full control over they system can also make changes which lead to more loss of architectural integrity, this will further damage the possibility to keep evolving the system.

**Lehman’s Law of Continuing Growth**  The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.

During the lifetime of a system developers will keep adding functionality to keep the system relevant, this is because of changing environment and user requests. Adding functionality will increase complexity of the system and increased complexity makes it harder to conserve familiarity. More functionality means increased complexity which has implications for the ability to continue to evolve, this means the activities are required which mitigate the complexity (Lehman’s Law of Increasing Complexity).

**Lehman’s Law of Declining Quality**  The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

Increasing complexity, loss of understanding of the system, and the system size have implications for the ability to make good modifications which fulfill requirements and do not introduce bugs in the system. Quality will decrease without rigorous efforts to preserve the system’s architectural structure, mitigate increased complexity, and the loss of system understanding associated with system growth and change.
4 Markup Languages

Markup is a way formatting and adding information to the text of a document, i.e. tags `<bold>`This text is bold`</bold>`. A markup language is used for text-formatting by adding markup into the text. Markup languages are divided into these different categories:

- Punctuational markup is a part of regular text. It consists of a closed set of marks or annotations such as commas and semicolons. The author placing a period at the end of a sentence is use of punctuational markup [11].
- Presentational markup is what is done to make the presentation of text clearer, such as spacing between paragraphs, page breaks and numbering [11].
- Procedural markup is when annotations, and code instructions are embedded in the text [11]. A well-known example of procedural markup is TeX which relies on tags to define the structure of a document.
- Descriptive markup use annotations in a similar way as in procedural markup, but the difference between them is that procedural markup is what should be done, while descriptive is what something is. Descriptive annotations declare what something is, and that this piece of text is member of a particular class [11].
- Referential markup enables the author to use entities that are external to the document and used to replace entities in the document during processing. Example: replacing some string (S1) in the document with the string (S2) from an external source [23, 11].
- Metamarkup provides support for controlling the interpretation of markup and enables the author to for example extend the vocabulary [11].

These six different languages are categorised in two groups: Punctuational, referential, and metamarkup are concepts one wants to express. Presentational, procedural and descriptive are paradigms of ways to express these concepts [11, 23].

4.1 Properties of Markup Languages

Characteristics of a markup language allowing high levels of abstractions are descriptive syntax, nesting, mixed content, generic identifiers and derived text [23].

Nesting is the ability to have elements nested so that the combination of elements expresses more than if the elements were isolated [23]. see Figure 4.1.

Mixed content is when text can be interpolated with semantic descriptions. Meaning that text can at any point be extended with more semantic information. This can be exemplified by considering an arbitrary element containing content text, more description can be added to this text by adding elements to the content of the first element, i.e. `<heading>The <italic>HMS Visby</italic> is a ship</heading>` [23].

Generic identifiers allow the author of a document to name elements almost arbitrarily. An example of a markup language which has this characteristic is XML which allows the author to name elements as seen fit, with some exception such as special characters (\-, ., : are allowed) and names must always begin with a letter.

Derived text refers to the idea that part of a text can be seen as derived from other parts of the same document. An example of this is marking up sections in a document with numbers to later use this to generate a table of contents.
4.2 Modification

Markup languages tailored to the context allow for simplified syntax and faster writing of content when compared to HTML (previously discussed in section 1.1). The work required to make modifications to the document must also be taken into consideration. The work required can be measured by comparing the initial document to the desired document and calculating how much work is needed to accomplish the change. The syntax of the language is the factor which influences the amount work as raw text is the same regardless of language. This section will discuss the role of syntax when modifying documents and how the work required can be measured using Levenshtein Distance[21].

Consider this scenario where we want to use an ordered list containing Item1 and Item2, and Item should be italic and 1 should be italic as well as bold. Nesting is required to achieve this. Listing[4.1] shows how it is written in Jade and Listing[4.2] in Markdown[8]. Jade relies on indentation for nesting while Markdown uses newline for a new item in the list and declaration of a new element between the start and end-tag of an element.

```
... 
ol 
  li 
    i | Item 
      b 1 
    li Item2 
... 
```

Listing 4.1: Nesting in Jade

```
... 
1. _Item**1**_ 
2. Item2 
... 
```

Listing 4.2: Nesting in Markdown

---

*https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet*
Extending the scenario, consider the developer deciding that Item 1 should no longer be in italic but 1 should remain bold. In Markdown this can be achieved by removing the two underscores. In Jade this is achieved by removing i, and since indentation is used when nesting elements everything nested must also move one indentation to the left to have correct syntax. Similarly all elements nested within need to be moved one indentation to the right if a nested element is added. This is an example of hidden dependency \cite{13} between the elements.

Levenshtein Distance, or edit distance, is a function measuring difference between strings \cite{21}. Operations allow for edit of one character at a time at the cost of 1 by insertion, deletion, or replacement. As an example, the distance between the strings “kitchen” and “mitchell” is 3.

1. \textit{kitchen} $\rightarrow$ mitchen (replacement)
2. mitchen $\rightarrow$ mitchel (replacement)
3. mitchel $\rightarrow$ mitchell (insertion)

Considering the the distance of two earlier mentioned documents, before removal of the nested italic element and after the removal. Edit distance in Markdown can be as short as $2 \times c$, where $c$ is the number of characters in the tag, assuming the start and end tag have the same number of characters. The edit distance will be $S + E$ in cases where start $(S)$ and end $(E)$ tag are not equal amount of characters, \cite{18}. The edit distance in Listing 4.2 would 2 in Markdown. The distance in Jade (Listing 4.1) would be 3 or 5 for removing b and the indentations, depending on if indentation is considered one character or two blank spaces each. This means that the required manual work to make this change is less in a language like Markdown. Modifications to Jade reveals hidden dependencies, and the edit distance will be further since indentation is used for nesting elements and will have to be corrected. The distance will also grow or shrink depending on the number of elements nested within.
5 Empirics

5.1 Problem Relevance

Following Lehman’s classification of programs [17], WebbRaket is regarded as of the E-type. Lehman describes this type as those programs which “mechanise a human or social activity” [17]. WebbRaket should be regarded as a teaching platform where readers can go to learn about different concepts, libraries etc. within the domain of web development. It is created to solve a real world problem and, as is suggested as a characteristic of E-type systems, WebbRaket has undergone change.

**Lehman’s Law of Continuing Change** can be observed as the content and underlying components of the website are continuously modified. Content is modified for a variety of reasons. As mentioned, developers can make contributions as seen fit. Existing content can be extended or reduced and developers can also add new chapters about technologies which are not yet covered. This process of modifying the content can also be seen as a process of pruning the content to remove less than satisfactory sections of text.

**Lehman’s Law of Increasing Complexity** may seem to be self-evident because of a growing system. It can be observed more clearly by looking at what was experienced when Jade was used as the language to annotate documents. Because of the syntax used in Jade, document modification was experienced as taking too much time as the content grew and finally had to be abandoned. The move from Jade to Markdown is a case of work done to reduce complexity and the choice of Markdown a way to manage it. The problem can not be seen in the current state of WebbRaket but it can be a cause of the system having to grow more before complexity becomes a problem again.

**Lehman’s Law of Conservation of Familiarity** is not as apparent and the need for staying familiar with the system is not as critical as this system is on a small scale. Team members still need to be familiar with the system and one way it is done is by discussing any change to the code or website content before it is accepted into production. This serves as a way to review any change and propose corrections if needed or alternative ways of solving a problem, but also as a way for developers to keep track of changes to the system.

**Lehman’s Law of Continuing Growth** can be observed as the content of WebbRaket is constantly extended to include more technologies and concepts, and it is rarely a case of removing already covered technologies from the repertoire. This can be exemplified by a current task in the backlog which is to extend what is written about HTML to include explanations and examples about semantic elements introduced in HTML5. Attention is not only paid to explain concepts which are introduced in new versions of a technology but also to extend the existing repertoire to include what is not currently covered such as testing and formal verification.

**Lehman’s Law of Declining Quality** states that the quality will seem to decline unless work is done to maintain and adapt the system to operational changes. The work to maintain and adapt the system can be exemplified by the effort in extending the website and its repertoire and to update the content to include new functionality in technologies. Maintaining quality of a less technical nature is the work to correct,

---

modify and extend explanations and code examples. The migration from Jade to Markdown is also an example of work done as a response to the impression of declining quality. As the content of the website grew the problems associated with the syntax of Jade became more apparent, finally resulting in a migration.

5.1.1 Maintenance & Evolution

Maintenance activities can be identified in all of the categories defined by Lientz & Swanson [19]: adaptive, perfective, corrective and preventive.

**Adaptive maintenance** is done to respond to changes which have occurred in the system environment [19]. An example of changes which can be classified as adaptive are changes done to the content to include new technologies. The migration from Jade to Markdown can also be classified as an adaptive change, however is better classified as a preventive activity because of the reason and objective of the migration.

**Perfective maintenance** is done to respond to changes or expansion in user requirements [19]. Changes classified as perfective would be the modification and extension of content related to concepts which are already described. This can be clarifications of models and terms etc., as well as changing and adding new code examples to make it more detailed and easier to follow.

**Corrective maintenance** relates to fixing errors and other defects [19]. Changes within this category are those which related to corrections to the content in cases of mistakes, misinformation and any other corrections done to the underlying system.

**Preventive maintenance** is done to improve the maintainability of the system and prevent or minimize future problems [19]. An example of an activity classified as preventive is the migration from Jade to Markdown. As mentioned earlier this move was done to improve maintainability and is an attempt to prevent the experienced problems now and in the future. Other activities with are also preventive is the alignment of presentation, examples of this is having a uniformed way of writing and using the same tags throughout the content to represent for example code examples, flowcharts etc.

This way of classifying software maintenance activities can be seen as complete as it covers all types of work which can be done when maintaining a system. It is however a classification which is general as it does not account for where the change is done and where it will be manifested. Consider a scenario where WebbRaket conducts maintenance classified as corrective. This hypothetical maintenance can be done to correct faults in the underlying system which will not alter what is presented to the user. It can also be correction of text to remove spelling mistakes. This will alter what the user experiences and is very different from the first example. The activity is classified as corrective in both examples but is not described further. Examining the activities with the classification suggested by Chapin et al. [9] yields a more detailed description.

Three questions are used to determine which cluster the activity belongs to after which the activity can be classified [9] (as discussed in section 3.1, see Figure 3.1):

**Q1: Was software changed?** If the answer is no then the activity belongs in the support interface cluster. Activities in this cluster are those related to the studying of the software, consultation and training. An example of activities in this cluster would be discussions about the possibility of future maintenance work, the feasibility in terms of what would have the change in the system, and time and cost for the work. A concrete example would be the evaluation of Jade and Markdown and the decision that WebbRaket should migrate to Markdown as it was determined to be more suitable.
Q2: Was source code changed? The activity is of the documentation type if the software was changed but the source code was not changed. The default answer is ‘Yes’ as maintenance in WebbRaket focuses on changing of code and markup.

Q3: Was functionality changed? If the software was changed and it involved the changing of source code it is a question of whether or not the change affects what the user experiences. If the user-experience is unchanged it is a change which is of the software properties type. If the user-experience if changed it is a business rules type of change. Within these two clusters is also where changes which lead to software evolution can be found [9].

Activities in the software properties cluster are classified as adaptive, performance, preventive, groomative. Changes within this cluster will not influence what the user experiences using the system. The migration from Jade to Markdown is a change within this cluster, as tags in Jade could be replaced with a tag in Markdown which would generate the same HTML, thus not altering what the user experiences. This activity can be classified as adaptive since this change resulted in a change of technology used. However since this change also affected the maintainability of the code it is also groomative. Chapin et al. [9] suggest that in cases where changes are made in the software properties or business rules cluster they will have impact which result in other changes in the system. One can view it as the change being made has other activities which support it. In this case the adaptive change was done with supportive activities. The implementation of Markdown can however not be classified as a preventive activity. It can only be classified as preventive if the activity to reduce or avoid future maintenance problems is done without altering the user-experienced functionality, or the utilized technology and resources. Thus this activity can not be classified as preventive as it changed the technology used.

Activities in the business rules cluster are classified as enhancive, corrective, reductive. All can be identified in WebbRaket as they all relate to changing the system in a way which alters what the user experiences. Enhancive activities are those which which alter the user-experiences functionality by replacing, adding or, extending it in some way. Adding new chapters to the existing content to describe and explain technologies and extending a chapter by adding more detail (for example code examples) are examples of enhancive activities. Correcting or altering the system to better the implementation of the business rule is an corrective activity, such as corrections of content, adding better handling of exceptions and so on. Finally, limiting or removing parts of the system is seen an reductive activity, as an example removing content about a technology which is no longer relevant for the project.

5.1.2 The Evolving System

Following Chapin et al. [9] it can be concluded that work is being undertaken which leads to software evolution. As discussed in section 5.2, activities which are in the business rules cluster are all activities which evolve the system, and adaptive and performance activities in the software properties cluster also lead to an evolved system. There are also activities which are planned and some started to be implemented which will lead to further evolution of the system (see section 5.1). Thus the system is still in a state of evolution and has not yet entered the stage of servicing, following the Staged Model suggested by Bennet & Rajlich [4]. The migration from Jade to Markdown can be regarded as phase-out and closedown of a version of the system and the release of a new version (see section 3.2). The migration resulted in every content document being re-worked to have the new syntax and a new way of annotating documents. This considerable effort was undertaken because continuing with the previous version was seen as ineffective and the gain of this input of work was a system which would be easier to maintain and evolve. While this new version is an improvement in terms of maintainability it has still become less satisfactory over time.
5.2 Markup Creator

The problems experienced by WebbRaket relate to lack of distinctions which can be made when formatting text, the actual work required to do maintenance activities, and the need for more flexibility in the language itself. The second is handled by what has already been said (migration). What is proposed is a hierarchy when nesting which does not rely on whitespace for the nesting of elements. Tags should be defined with (none-whitespace) characters which represent the closing of an element, and any element within the start and end-tag is considered nested. The syntax should be defined in a way which avoids the problems countered with the Jade markup syntax (section 4.2). This reduces the work needed to perform modifications to documents when the structure is large.

This thesis is concerned with modification done manually as the modification of documents is done manually in WebbRaket. Users of the system should not be completely free when defining syntax. It is a conscious choice to eliminate the potential problems, shown to exist in Jade, which could arise again if users could use whitespace as start and end-tags. The aim is that growth in work required for modification should only be a result of a greater number of elements, and not if and how elements are nested. The work required for modification can be measured in Levenshtein Distance (see section 4.2) as it measures the number of edits made in the form of insertion, deletion or replacement to transform document $A$ into document $B$.

The program is one which allows for easy exportation of documents annotated in a syntax defined by the user to XML (see Figure 5.1). What is required of the user is a file containing the desired tags and their meaning (Input Grammar), and a document with text annotated with the chosen tags (Input Document). Input Grammar is used to create a complete grammar file which is used by the tool ANTLR to generate a lexer and parser. Lexical analysis is conducted on the input document and the resulting tokens are feed to the parser. The visitor is applied to traverse the parse tree and the result is XML output. The following sections will further explain the steps from user input to XML output.

![System Flowchart](Figure 5.1: System Flowchart)
5.3 Creating grammar

Consider the following the scenario: A user wishes to represent bold and underlined text through the use of defined tags. The chosen tags are * to represent bold and % to represent underlined.

The text “I repeat: tread carefully” would be represented by wrapping it in the chosen tags: *tread %carefully%*. This particular text would have the tree representation seen in Figure 5.2.

![Figure 5.2: Annotation of Text]

The combination of text and these two tags can be used to generate a variety of different trees depending on how the documents text has been annotated. The possible tree presentation can however be defined by how text and tags and be combined in a document. A document can contain text which has not been tagged or text which has been. A tag, or element, spans from start to end tag and can contain text or other elements. These constraints on how text can be represented will always be true regardless of how the text is annotated and how many tags have been defined and used. As these constraints are always true they can be used to create production rules for a grammar-file describing the markup syntax.

```
document  : ( text | element )*;

element   : TAG content TAG;

content   : ( text | element )*;

text      : TEXT;
```

Listing 5.1: Production Rules

Listing 5.1 describes the constrains. Words written in small letters indicate a production rule, capital letters a lexer rule. A document can contain text and n number of elements(tags). An element is made up of a tag which marks the start of its content and tag which marks the end of the content and element. The content of an element can contain text and n elements. These are the production rules which are used to parse input and determine if the input is valid. Lexer rules are defined to determine terminals (the end nodes).

\[^{12}_a\] - match all repetitions
TAG: '*' ;

WS : [ \r\n]+ -> skip ;

TEXT : ~[^]*+ ;

Listing 5.2: Lexer Rules

TAG matches to * and TEXT to any character sequence except * (it may not occur in any other way, as it is used to start and end an element). Finally WS is any occurrence of whitespace (space, tab, newline) and is not included. This rule is only considered when whitespace occurs between tags as TEXT accepts whitespace. The combination of production and lexer rules allow for any combination of tagged and untagged text in the document. Currently the only accepted tag is *. The modifications which must be made to include a new tag in the grammar (% as an example) is the extension of element to include a new option, a new lexer rule and the modification to TEXT to exclude %.

element
  : TAG1 content TAG1
    | TAG2 content TAG2
  ;

TAG1: '*' ;
TAG2: '%' ;

Listing 5.3: Extended Production Rules

5.4 ANTLR 4

This system was built using ANTLR and the programming language C#. ANTLR is a tool used for generating parsers and can be used to build languages. Given a grammar ANTLR can construct a lexer, parser, parse tree and visitors to iterate over the tree and perform desired tasks. ANTLR uses LL(*), or LL-regular, for parsing. This means it parses from left to right and performs derivation on the left node. LL(k) is a parser with k lookahead, LL(*) is not limited to a finite number of lookahead [24]. Left recursion has to be avoided when constructing grammar for a LL parser. Left recursion is when a symbol refers to itself, i.e a symbol can be derived as a form of itself as the left-most symbol. This can occur directly i.e $A \rightarrow A + B$ or indirectly $A \rightarrow B, B \rightarrow A + C$. [26] ANTLR 4 has built in support to re-write direct left recursion rules, but will not be able to parse correctly if the grammar has indirect left recursion.

The production rules in Listing 5.1 with accompanying lexer rules in Listing 5.2 make up an example of grammar as it is written in ANTLR. The idea is that with the proposed solution users should not have to see or edit the production rules as such. Users can define terminals (start and end tags) which are then combined with non-terminals to form the production rules and complete the grammar of the language. The removal of the user from production rules makes it possible for the user focus on writing documents and converting to XML by only defining tags to annotate the content. Production rules should be defined in a way which avoids problems discussed in (growing edit distance with nested elements). The problem is always avoided since the element has a defined start and end tag and the user does not have to modify the rules. The introduction of left recursion in the grammar is also avoided by removing the user from the production rules.
5.5 Implementation of Grammar

As discussed in the previous section and section 2.2.1, an aim was that a user should not have to make any modifications to the grammar file except by defining of tags. The user is therefore able to define tags and their meaning in a separate file which is then imported and used to create a grammar file. This grammar is modified by the system to contain the lexer rules and options for the element rule.

Given this input file, the system creates a grammar file which is then used to create the lexer and parser in ANTLR.

\[
\begin{align*}
*\text{text}* & \rightarrow \text{bold} \\
%\text{text}\% & \rightarrow \text{underline} \\
$\text{text}$ & \rightarrow \text{heading}
\end{align*}
\]

Listing 5.4: Input File

The following figure displays the modifications made to the grammar file when the user’s file is imported.

\[
\text{element} \\
:\text{BOLD} \ \text{content} \ \text{BOLD} \\
| \ \text{UNDERLINE} \ \text{content} \ \text{UNDERLINE} \\
| \ \text{HEADING} \ \text{content} \ \text{HEADING};
\]

Listing 5.5: Partial Grammar

The grammar now allows for formatting text with three different tags and any legal combination of them. The completed grammar now allows the user to import and parse annotated documents. The following tree is constructed given the input file in Listing 5.4 and a document containing “I am %an $example$% *input*.”

![ANTLR 4 Parse Tree](image-url)
5.6 Construction of XML

ANTLR 4 is able to generate parser and lexer given a completed grammar. ANTLR also offers the option to generate a generic visitor which can traverse a tree produced by the parser. This makes it possible to construct a class which inherits this generic class (BasicMarkupBaseVisitor) and overrides its methods to perform whichever operations are sought. The only output desired at this moment is an XML representation of documents annotated which the tags previously defined by the user. But the program can be extended to have multiple different visitors to support different outputs.

```java
public class MarkupVisitor : BasicMarkupBaseVisitor<XElement>
{
    XElement document = new XElement("document");

    public override XElement VisitDocument(BasicMarkupParser.DocumentContext context)
    {
        for (int i = 0; i < context.ChildCount; i++)
        {
            document.Add(Visit(context.GetChild(i)));
        }
        return document;
    }

    public override XElement VisitText(BasicMarkupParser.TextContext context)
    {
        // Implementation...
    }

    public override XElement VisitElement(BasicMarkupParser.ElementContext context)
    {
        // Implementation...
    }

    public override XElement VisitContent(BasicMarkupParser.ContentContext context)
    {
        // Implementation...
    }
}
```

Listing 5.6: MarkupVisitor

A document can have \( n \) number of children, elements and/or text without any specified tag. Therefore each child must be visited and since the child is either text or element its children must also be retrieved. `GetChild(int i)` returns the remaining tree and it is then visited. This means the tree is “broken down” until end nodes are found and can be returned back.

```java
public override XElement VisitDocument(BasicMarkupParser.DocumentContext context)
{
    for (int i = 0; i < context.ChildCount; i++)
    {
        document.Add(Visit(context.GetChild(i)));
    }
    return document;
}
```

Listing 5.7: VisitDocument()

![Element Tree](5.4) Figure 5.4: Element Tree

An element will always have the same structure (Figure 5.4), the value of the left-most child can therefore be retrieved without having to check what kind of child it is. `MeaningFromToken(string input)` is used to substitute a tag defined by the user for the meaning it was assigned in the input file (*text* → bold), see Listing 5.3.
public override XElement VisitElement(BasicMarkupParser.ElementContext context)
{
    string token = context.GetChild(0).GetText();
    XElement content = Visit(context.GetChild(1));

    return new XElement(grammar.MeaningFromToken(token), content.Elements());
}

Listing 5.8: VisitElement()

Content can contain both any number of text and element children. All of them need to be retrieved and visited. A new XElement is instantiated and returned once all nodes in its children have been visited, since elements found within are nested and this must be represented in the returned structure.

public override XElement VisitContent(BasicMarkupParser.ContentContext context)
{
    XElement content = new XElement("content");
    for (int i = 0; i < context.ChildCount; i++)
    {
        content.Add(Visit(context.GetChild(i)));
    }
    return content;
}

Listing 5.9: VisitContent()

Text is always a leaf nodes, as defined in the grammar, and can therefore simply be returned.

public override XElement VisitText(BasicMarkupParser.TextContext context)
{
    return new XElement("Text", context.GetText());
}

Listing 5.10: VisitText()

A generated parser and lexer from the grammar, and a visitor from the parser with an implementation to return an XML representation have been constructed. It is now possible to declare a method which instantiates them and returns an XML representation of the text in a given file.

public static XElement GenerateXML(string path)
{
    BasicMarkupLexer lexer = new BasicMarkupLexer(new AntlrFileStream(path));
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    BasicMarkupParser parser = new BasicMarkupParser(tokens);
    var tree = parser.document();
    MarkupVisitor visitor = new MarkupVisitor();
    return visitor.Visit(tree);
}

Listing 5.11: GenerateXML()
The contents of Listing 5.4 will generate the grammar necessary for the program to now output XML.

\$Hello$ A %text is *bold*%

```xml
<?xml version="1.0" encoding="utf-8"?>
<document>
  <HEADING>
    <Text>Hello</Text>
  </HEADING>
  <Text> A </Text>
  <UNDERLINE>
    <Text>text is </Text>
    <BOLD>
      <Text>bold</Text>
    </BOLD>
    <Text>bold</Text>
  </UNDERLINE>
</document>
```

Listing 5.12: XML Output
6 Usage

This section will present how the system can be used through a series of use cases which illustrate the ability to modify and extend text, tags, and their meaning. The implications for maintenance activities are also discussed.

6.1 Scenario A - Extending Grammar

Consider a scenario where the author needs to define three tags in order to make it possible to annotate text with tags representing bold, list, and list item. This could be done by passing a file to the system containing what is shown in Listing 6.1:

```
*text* -> bold
%text% -> list
$text$ -> item
```

Listing 6.1: Scenario A - Input Grammar

This file can be extended at any point if users request more tags to make distinctions between text-segments. Such as if there is a need for a fourth tag which represents italic. The file depicted in Listing 6.1 has been extended by adding a new line with the tag wanted and the meaning to be assigned to it, for example “!text! -> italic”. When run this updates the grammar file used by ANTLR to include rules for the new tag and the user can now annotate the document in more ways.

```
document : ( text | element )* EOF;

element : BOLD content BOLD
          | LIST content LIST
          | ITEM content ITEM
          | ITALIC content ITALIC;

text : ( text | element )* ;

BOLD: '⁄' ;
LIST: '% ' ;
ITEM: '$ ' ;
ITALIC: '! ' ;

WS : [ \r\n]+ -> skip ;
TEXT : "[*%$!]"+
```

Listing 6.2: Scenario A - ANTLR Grammar

Given the document in Listing 6.3 the parse tree would be constructed (depicted in Figure 6.1) and the XML document seen in 6.4 would be the output.

Example: "Behold a list"

```
*Behold a list*
%$This is an !item!$%
```

Listing 6.3: Scenario A - Input Document
New requirements can be incorporated with ability to extend the grammar and thus the number of tags and distinctions which can be made. As stated in Lehman’s law of continuing change (see 3.3), systems must be able to adapt to maintain the same level of satisfaction. This ability of an extendable and modifiable grammar facilitates the need for adaption and makes it possible to continue to evolve the system by accepting and implementing requests for more variety of tags.
6.2 Scenario B - Modifying Text

Consider a scenario with a document containing text similar to Listing 4.1 and Listing 4.2. An ordered list containing Item1 where Item is italic and 1 is italic and bold which has to be modified in the same way as discussed in section 4.2. The same format can be achieved by providing the system with a file containing the tags needed to make the distinctions and another with the text.

```plaintext
#text# -> numlist
!text!  -> item
._text_  -> italic
*text*  -> bold
```

Listing 6.5: Scenario B - Input Grammar

Annotating the text with the tags ( # ! Item *1* ! # ) produces the same distinctions and a similar hierarchy of nesting

```xml
<?xml version="1.0" encoding="utf-8"?>
<document>
  <NUMLIST>
    <ITEM>
      <ITALIC>
        <Text>Item </Text>
      </ITALIC>
      <BOLD>
        <Text>1</Text>
      </BOLD>
    </ITEM>
  </NUMLIST>
</document>
```

Listing 6.6: Scenario B - Output XML

The edit distance, i.e. actual work required, would remain relatively low in a case where the text has to be manually modified such that the italic element is removed but 1 remains bold (as in 4.2). The edit distance when modifying, removing or adding nested tags always remains at 2\*c where c is the number of characters used to declare a tag. Whatever tag used when nesting and the depth of the nested elements will produce the same edit distance. The production rule which governs how a tag can be declared (element) requires that the content of a tag is surrounded by a start and end tag. The syntax will never allow nesting of elements by opening new elements with the use of whitespace (tab, newline) as is the case in other markup languages, i.e. Jade. Edit distance can not grow if the required modification being conducted is in a nested element, but only as a result of the characters used to declare a tag.

Section 2 discusses how generation of XML would be preferred to users authoring XML documents manually. The ability to add tags is satisfied by using XML as tags can be declared without a upper limit how many can be made use of. The edit distance will however be greater in XML compared to this solution when adding or removing tags and equal in cases of substitution unless users define tags which are longer than three characters. While XML does not allow whitespace to be used when declaring a new element, XML relies upon “< >” and “< / >” when opening and closing tags. Consider a scenario where an element in XML is as short as possible, i.e. name is one character < A > text < /A >. The adding or removing of it will have a distance of 7 while the shortest distance in this system will be 2 (!text!). The cost of adding or removing text inside the tag is disregarded as there is no difference. Edit distance in cases of substitution is equal, i.e. substituting A for B and ! for some other characters.
6.3 Scenario C - Modifying Meaning

A scenario may arise where the meaning of a tag has to be changed. Consider an input grammar file containing # to denote an ordered list and ! to denote list item.

\[
\begin{align*}
\#text# & \rightarrow \text{orderedList} \\
!\text{text}! & \rightarrow \text{item}
\end{align*}
\]

Listing 6.7: Scenario C - Input Grammar

It is decided ordered lists should no longer be used and instead be replaced by unordered lists. Updating the documents to reflect this change can be achieved in two ways:

1. A new tag can be created and used to denote unordered lists, the tag can be defined with characters not used by other tags.
   In this example “%text% → unorderedList” can be used for this purpose. Documents can then be updated to use % instead of #.

2. A different more efficient way to update all documents is to redefine the semantic meaning of #text# to #text# → unorderedList as this would change all documents to reflect the new definition.

Extending on what was discussed in Scenario B. Implement this change would require considerably more work if the system relied on XML created by the user instead of generation by the system. As it would require updating all documents by changing all tags declaring an ordered list to make them declare an unordered list. The argument could be made that no updating of the XML is required, the change could be reflected by changing how HTML is generated once deployed (updating XSLT). However this is not a good choice as it implies that the names of XML-tags which are written by the user no-longer reflect what they actually represent, i.e. \(<\text{orderedList}>\) being used to define an unordered list.

The ability to change semantic meaning, and thus the generated XML, also simplifies the process of updating parts of a document when new tags are introduced. Consider a tag representing an element used to denote code-samples of JavaScript and PHP, and the same tag is used for both, let this tag be “%text% → code “.

A new requirement is introduced requesting code-samples to be separated according to which programming language the text contains by using different tags. This can be achieved by updating the meaning of the original tag to “%text% → JCode “, introducing a new tag for PHP-code, and only changing the document where there are PHP code-samples.
6.4 Maintenance Activities

As discussed previously in 5.1, WebbRaket is to be regarded as an E-type system. To specify it relates to the creation and maintenance of documents annotated with markup, which are used to teach web development. Section 1.2 outlines the encountered problems of the project (work required to make changes and the limitation of available tags). Section 5.1 shows different maintenance conducted and confirmation of Lehman’s laws.

The previously described scenarios show how different maintenance activities can be conducted through modification of existing markup tags, their meaning, and introduction of new tags. The flexibility which exists in the defining of the markup syntax allows for acceptance and implementation of new user requirements.

As outlined by Chapin et al.\[9\], the maintenance activities which constitute software evolution and show the ability of the system to implement new user-requirements are found within the business rules cluster (enhancive, corrective, reductive) and the software properties cluster (adaptive, performance).

**Adaptive** activities are undertaken by the defining and redefining the syntax of the markup. The possibility to change allowed syntax and the meaning of tags means that the system can be continuously adapted. As the system itself is available to the user it can also be subject to changes to better fit a context different from its original aim.

**Enhancive, Corrective & Reductive** activities are, according to Chapin et al.\[9\], frequent and the most significant. The focus in the scenarios has been to showcase enhancive and adaptive activities as they are central to the goals of the thesis. While little attention has been paid to reductive activities, they can be exemplified by inverting the scenarios. When the situation calls for removal of tags and consolidation of parts of text it can be achieved by removing definitions from the input grammar and updating the documents to use other tags. The modification of a tags meaning in Scenario C can be seen as both a corrective and enhancive activity. While corrective maintenance can be viewed as simply fixing and correcting the experienced functionality, the separation of text into different tags as in Scenario C can be viewed as making the representation and formatting of text more correct with what the author seeks to represent, i.e. breaking up a code-sample tag into multiple tags which can represent different languages. Enhancive activities (replace, add, extend) are undertaken in all of the scenarios and together with the ability to perform adaptive activities they are seen as most important and central.
7 Conclusion

The purpose of this thesis has been to: Give users the possibility to modify and extend syntax to support the need for flexibility and adaptation.

The proposed solution to this problem is at this stage a prototype which allows users to create what is referred to as input grammar-files containing any tag the user chooses and assigning a meaning to it. This file is used to complete a formal grammar which is then used to parse documents annotated with the syntax defined by the user. The result of this process is the output of an XML representation which can be used for exportation to other systems. Though outside the scope of this thesis, given the context of use of this prototype the next step would be to pass the XML output to the markup language XSLT and transform the XML to HTML for publication. Attention has been paid to keeping what the user defines abstract, as the context and goal of this workflow has been known. Descriptive syntax is used and constrained to the use of a start and end tag. Allowing generic identifiers, nesting of elements, mixed content and deriving the XML from the documents widens the scope of what can be defined and allows for extendability of semantic meaning.

The need for flexibility and adaptation is justified by the fact that any system which models a real-world process must be able to adapt. Flexibility and adaptation are closely related since the more flexible a system is the easier it is to adapt. As the real world is continuously changing these changes must be reflected in the system in order for it to remain satisfactory. The problem is in the domain of software maintenance as the alignment of the system to a changing environment is an effort undertaken after initial development and deployment. Flexibility is achieved by giving the user almost free reign over the syntax and only limiting it to the concept of a start and end tag.

The context of the conducted maintenance activities is manual creation and editing of documents. As seen in the different scenarios in section 6, the proposed solution allows extension and modification in three ways:

1. Content modification
2. Syntactic modification
3. Semantic modification

The most suitable way of modification is determined by the user with editing distance as a criteria to be considered. Content modification is possible in all markup languages discussed. Syntactic and semantic modification, while possible in languages such as XML, can be done more efficiently in this solution and modification of semantics is not always a viable option in an XML solution.

The possibility to modify syntax and semantics allows continuous adaptation of the language (adaptive activity). Together with enhancive, corrective, and reductive, these activities are key for the evolution of software. The activities can be conducted with less, and in some cases equal, work required compared to other markup languages. This offers the possibility to to keep the software in a stage of evolution for a long time and not be forced to enter servicing as the maintenance work can be done and with limited required work.

To conclude: the goal of allowing the user to modify and extend the syntax to support the need more flexibility and adaptation is achieved with this prototype.
8 Discussion, Limitations & Future Research

In the current stage the prototype allows for basic formatting of text but not more advanced formatting by using specialised tags (as seen in for example GitHub Flavored Markdown). The prototype does not currently allow having one tag which creates multiple XML elements, such as one tag representing a list item with text which is bold and underlined. The construction of more specialised tags should be explored as a way to increase efficiency.

As in any system, the direction of development and use of the artifact can be very different from what was imagined during the design. The implementation and use in a real scenario would therefore be interesting to investigate to determine if the underlying assumptions remain true in practice.

The final conversion of XML to HTML has been left out as it is a seemingly simple task and outside of the scope of this thesis. Extending the prototype as output HTML would however be a valuable ability.

While this thesis explores modification of syntax, a presented result is the possibility to easily modify the semantic meaning of a tag. Semantic modification is part of the results and discussed, but the implications are not explored further as syntactic modification is the focus.

Finally, the gain and opportunity which a user-defined syntax presents are investigated but potential drawbacks are not. Changing syntax should be investigated to determine its affect on the ability and learn the syntax and write documents, and how multiple authors agreement on syntax.
References


[23] Christopher Okhravi. “Markup has resolution. : In search of a more abstract language.” Uppsala University, Department of Informatics and Media, 2014. 45 pp.


