User-defined automatic journal production in MedView

Fredrik Lindahl, 2001/2002

Abstract

In order to better organize and take advantage of the large amounts of clinical knowledge gathered every day in the field of oral medicine, the MedView project has provided a way to formalize this knowledge and to store it in a manner that makes it computationally accessible for future visualization and analysis. The formalized knowledge is stored into a knowledge base, which currently contains information about more than 3000 conducted oral examinations. In order to see the contents of the knowledge base, a tool for displaying and summarizing the contained examinations by means of generated text was developed. After being used for some time, various issues arose surrounding the use of this tool, resulting in a need for a new version of the tool to be developed. The goal of this thesis has thus been to produce such a new version of this tool, taking into account both the proficiencies and deficiencies of the previous version.

Sammanfattning

För att bättre organisera och använda sig av den stora mängden klinisk kunskap som samlas varje dag inom oral medicin, så har MedView-projektet tillhandahållit en metod att formalisera denna kunskap och lagra den på ett sätt som gör den tillgänglig för framtida datoriserad analys och visualisering. Den formaliserade kunskapen lagras i en kunskapsdatabas, som idag innehåller information om över 3000 utförda undersökningar. För att kunna se vad som finns i databasen så utvecklades ett verktyg för att visa och sammanfatta utförda undersökningar i form av genererad text. Efter att ha varit i bruk ett tag, så fanns det behov av att utveckla en nyare version av detta verktyg. Målet med detta examensarbete har således varit att utveckla denna nya version med den tidigare versionens för- och nackdelar i åtanke.

Preface

This report summarizes my thesis at Chalmers University of Technology, which I have been working on periodically between summer ‘01 to autumn ‘02. The initial work at the clinic for Oral Medicine at Odontologen in Gothenburg was not intended to develop into my thesis – rather, this idea was conceived at the end of the summer (’01) when it turned out that a lot of interesting development could be done to improve the current journal generation procedure, especially in regards to the user interaction. I am very interested in designing and developing user-friendly applications that people can find appealing to work with, and that can help them in their daily work. Furthermore, during the last couple of years I have been focusing on learning and developing with the Java programming language – an object oriented language that is gaining more and more popularity amongst developers. The work at the clinic in the summer of ‘01 combined both of my major interests, so when presented with the opportunity to continue this work as my thesis, my decision was not a difficult one to make.

I would like to thank all people involved in making it possible for me to do my thesis and to write this report. Especially, I would like to thank my supervisor Olof Torgersson at the Department of Computer Science and Engineering at Chalmers University of Technology for the constructive criticism and interesting discussions during my work. I would also like to send special thanks to Professor Mats Jontell at the faculty of Odontology at Göteborg University for giving me latitude to explore and develop my ideas, as well as providing me with the opportunity to work within this highly interesting and exciting area.

- *Fredrik Lindahl, Gothenburg 2002-10-04*.

TABLE OF CONTENTS

Abstract 2
Sammanfattning 3
Preface 4
1 Introduction 6
1.1 MedView 7
1.2 Natural Language Generation 8
1.3 Iterative Development and the Unified Process 9
1.4 Design Patterns and GRASP 10
1.5 Development Goals 11
2 Methodology 12
2.1 Development History 12
2.2 A Unified Process Retrofit 14
3 Analysis 16
3.1 The Knowledge Base (KB) 16
3.2 Previous Journal Generation Procedure 16
3.3 MedSummary 18
3.4 Analysis - Conclusions 20
4 Design 21
4.1 Text Generation Strategy and Algorithm 21
4.2 System Architecture and Overview 25
4.3 SummaryCreator 27
4.3.1 Functionality and User interface 27
4.3.2 Software Design 30
4.3.3 Model (Domain) 33
4.3.4 View (Presentation) 34
4.4 MedSummary 36
4.4.1 Functionality and User Interface 36
4.4.2 Software Design 38
4.4.3 Model (Domain) 41
4.4.4 View (Presentation) 42
4.5 Utility Packages 44
4.5.1 The datahandling Package 44
4.5.2 The common Package 46
4.5.3 The dialogs Subpackage 47
4.5.4 The generator Subpackage 50
5 Conclusions and Future Development 52
6 User Documentation 55
6.1 Global Application Matters 55
6.2 About the SummaryCreator Application 58
6.3 SummaryCreator Application Overview and General Documentation 58
6.4 About the MedSummary Application 59
6.5 MedSummary Application Overview and General Documentation 60
7 Glossary 63
8 References 66
Appendix A – Examples of Previous MedView System Resources 67

1 Introduction

In the area of Oral Medicine, large amounts of clinical data are being gathered each day from conducted oral examinations. Recording this data in the traditional paper-based manner results in problems when data is to be retrieved, summarized, or in other ways processed or analyzed quickly. As time goes by, the vast amount of clinical data being accumulated becomes increasingly difficult to manage, and extracting information from it can be very cumbersome. The MedView project, initiated in 1995, aims at providing ways to formalize and store clinical data, so that subsequent analysis can be performed computationally.

One way of utilizing such formalized data is to let the system automatically generate patient journals and summaries. For instance, the system could be asked to produce a summary of previous patient visits prior to an upcoming visit, which would give the dentist a quick review of patient status and background. The focus of this thesis has been on applications that are used to create the environment necessary for the automatic generation of such text.

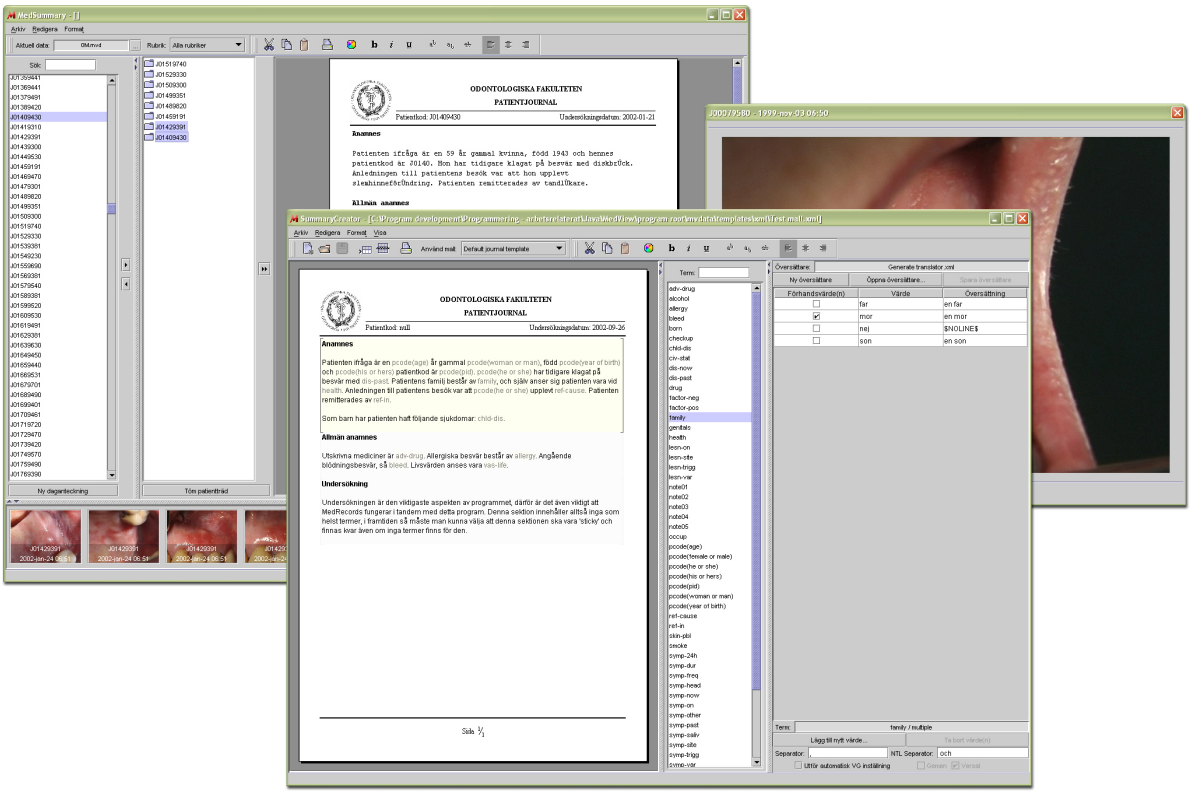


Figure 1.1 – The developed applications

In order for a user to produce an environment that can generate summaries, the user defines *templates* containing *slots* where the actual content from an examination record will be inserted at generation time. The content, in turn, is translated from its universal format (as specified in the examination record) to the language used in the template context, which is done with the *translator*. The translator maps values in their universal format to a certain translation, and may also contain various macros that are processed at generation time. To generate a journal, the user selects the examination(s) of interest, a template to use, and a translator to use, and the application produces a journal consisting of the template text with the slots filled with the values for the chosen examination. The system fills the slots of the template text with the translated values from the examination, along with performing certain

simple processing of the text to make it look more like natural language. An interesting future development aspect would be to improve the NLG aspect of the journal generation, without making it too complicated for the end users to create their own templates and translators.

The intended audience of this thesis consists of four groups: 1) people interested in object-oriented software development and in seeing a practical example as well as various common design patterns in use, 2) people who intend to use the applications described in this thesis, wishing to learn more about how to use them and/or some background information concerning their development, 3) people actively engaged in the MedView project, interested in learning more about this part of the project and/or the ideas surrounding it, and 4) people who are interested in the MedView project but knows nothing about object-oriented software development, design patterns, UML etc. and that do not intend to use the described applications.

People from the first group described above should focus on reading the ‘design’ part of the thesis, which deals with the design details of the various parts that compose the applications, including the major design patterns in use as well as how they fit into the system structure. They could also read the ‘methodology’ section in order to see how using UML and design patterns can be of great help in the development of a medium-to-large-scale system. People from the second group should focus on the ‘user documentation’ section, describing how the applications are used. Also, if interested in some background information on the applications, they should read the ‘analysis’ section describing the situation surrounding the applications and the situation as it was prior to development. People from the third group - especially developers that are to continue development of the portrayed applications – should read all sections of this thesis - with emphasis on the ‘analysis’, ‘design’, and ‘conclusions’ parts. If already familiar with the MedView project, the parts describing it can be skipped. Finally, people from the fourth group should read this introduction carefully – especially the parts about MedView - as well as the ‘analysis’, ‘conclusions’, and parts of the ‘design’ section dealing with functionality and user interface design. If interested in seeing screenshots from the developed applications and various scenarios of their use – which illustrate typical actual work with applications from the MedView project - they should also read the ‘user documentation’ section.

1.1 MedView

MedView is a joint project with participants from both oral medicine and computer science. The project has been developed in close cooperation with SOMNET (Swedish Oral Medicine NETwork) – a network consisting of dentists and practitioners in the area of oral medicine. The aim of the project is to develop tools that can assist the clinician in his or her daily work and to provide the opportunity to analyze the information that can be obtained from oral examinations. In order to do this, a formalized knowledge base containing patient examination data has been built, as well as tools to extend, view, and analyze the knowledge base.

Figure 1.2 – A clinician entering data during an examination

One of the project's main achievements so far has been to formalize the basic health care activities within oral medicine in such a way that examination data can be stored in this formalized manner in the knowledge base [5]. Tools for entering examination data directly in the examination room during an actual examination have been developed, as well as various tools for viewing and analyzing the contents of the knowledge base (including the tools used to produce textual summaries of examinations which has been the focus of this thesis). Today, the system has been in use for several years at various clinics throughout the country to gather information about more than 3000 examinations, including a large amount of digital images associated with the various examinations. More information about MedView can be found in [5], [7], and [8].

1.2 Natural Language Generation

Natural Language Generation (NLG), is about generating text from a variety of sources that as closely as possible should resemble natural language. There are two basic approaches for natural language generation - the *deep* and *shallow* approach [7]. The deep approach usually results in sophisticated and general NLG systems that can be used in a variety of contexts, while the shallow approach generates text in a simpler way - usually resulting in simpler but more context-dependent NLG systems.

Most NLG systems today divide the generation task into three functional steps [9]: 1) text planning (also known as *content determination*), 2) sentence planning, and 3) linguistic realization (also known as *surface realization*). The text planning step consists of determining purpose, content, and context of the generated text, while the sentence planning step aims at converting text plans to sentence plans by planning the structure at sentence level. The final step - linguistic realization - converts sentence plans into surface text (the generated text) through various processing steps. More about NLG and how it is used in the MedView project can be found in [7], [9], and [10].

1.3 Iterative Development and the Unified Process

When developing large-scale applications, it becomes necessary to follow some kind of process that organizes the development in a feasible manner. Several projects have been developed using the so-called *waterfall* development process. The waterfall development process basically breaks down development into a set of finite phases or steps, where each step depends on work done in earlier steps. If followed in a strict fashion, the waterfall development process prohibits you from returning to a previous step once completed – for instance, say the first step has been completed (evaluate the problem and gather information), you may not return to it if new information is discovered further on, say when working with the second step (propose solution and define requirements and time plan). The waterfall process has been shown to have considerable weaknesses - when developing modern object-oriented systems, there is a high probability of discovering new aspects that had not been considered in earlier steps. Thus, it is necessary to adopt a more iterative approach allowing you to return to previous steps and integrate newly discovered concepts into the system. The idea is not to make hap-hazard jumps back and forth between the different steps of development, but instead to follow the general ideas of the 'waterfall' model *in several iterations,* and to deal with newly discovered aspects in the following iteration cycle.

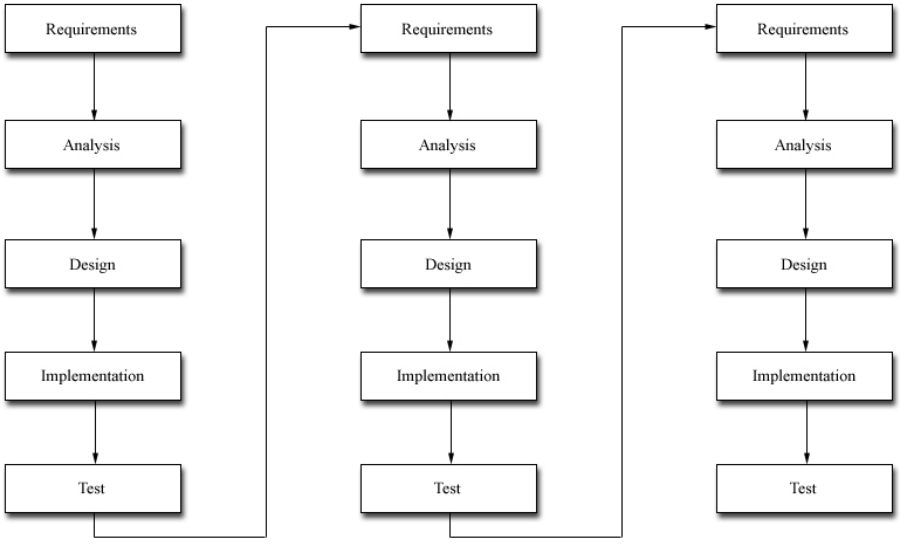


Figure 1.3 - Iterative development

Another important aspect of iterative development (adopted in the Unified Process, which I will introduce below) is that you should choose to develop the most vital and critical parts of your system early in the project, i.e. in the early iterations, in order to reduce project risk. The idea is to resolve the major risks early and detect potential project-stoppers early on, which will reduce both time spent and costs if the project turns out to be infeasible.

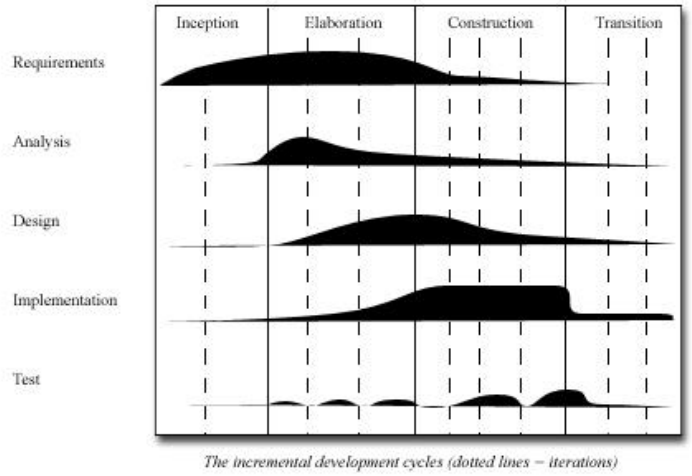


Figure 1.4 – Unified Process (UP) disciplines

The Unified Process (UP) is an iterative software development process making heavy use of the Unified Modeling Language (UML) for describing the various outputs (*artifacts* in UP terminology) of each stage (*discipline*). For instance, during analysis you might produce use-case diagrams and various use-case scenarios in order to describe existing and future 'scenarios of use' of the system or the surrounding environment. During design, class diagrams and interaction diagrams are created, describing a more detailed view of the system being developed. The full UML diagram set consists of a great deal of various diagram types, it is important to realize that it is not necessary to use all these diagram types in your project – you should use the ones necessary to clarify important aspects of the system under development. In fact, very few projects and system descriptions use all possible UML diagram types - the subset used depends on the system being described.

In the Unified Process, each step between iterations is preceeded by a tested, integrated, and executable system release. The release is not an experimental or throw-away prototype, but rather a production-grade subset of the final system [2]. One benefit with this approach is that early, visible progress of the system is made, resulting in increased developer confidence and an opportunity to systematically test releases on intended users in order to make sure you are on the right track.

1.4 Design Patterns and GRASP

Design patterns are tried-and-true solutions to reccuring problems and problem contexts that occur while developing software. Patterns help structure software by presenting general problem contexts and solutions to them. The solutions suggested by design patterns are well-established solutions crafted and used by other expert object designers. In the book 'Applying UML and Patterns' [1], attempts are made to analyze the basic building blocks of object design and well-established design patterns, and to describe nine basic design principles called the *GRASP* patterns (General Responsibility Assignment Software Patterns). In the words of the author, the GRASP patterns are described as a 'learning aid to help one understand essential object design, and apply design reasoning in a methodical, rational, explainable way'. By having a thorough understanding of the nine GRASP patterns as well as other fundamental object design principles, you understand the reasons and concepts underlying many of the established design patterns.

1.5 Development Goals

The most frequently used application to view examination data in the knowledge base in the previous MedView system was an application called MedSummary, developed using the Objective-C programming language. The purpose of the application was to generate summaries of examinations in a way that as closely as possible resembled a manually entered summary - this was done by filling ‘gaps’ in a text file with values from an examination. The format of the text file containing the gaps (also known as the *template file*) was rather tedious to work with, involving manually remembering and entering special characters in the text. It was also hard to see the various ‘parts’ of the journal (such as what part of the template constituted a section) when working with a standard text editor. In practice, this led to the situation where one expert user developed the template text files underlying the generation and the others used this set of files. As a consequence, each user became dependant upon another individuals concept of how a journal should look - which may vary from individual to individual. One of the goals of my thesis was to remove these necessities, and make it possible for each user to produce his or her own template without any expert knowledge of how templates are made up. An important issue that had to be kept in mind was that a template should be easy to modify and extend - thus it was also important to visualize the template structure to the user in an intuitive way. Since the idea is that a user should be able to define his or her own templates, it is important that it is not too complicated and that the user has an intuitive feeling of what happens with the template when the journal is generated.

In order for the knowledge base and the applications to have international use, various languages had to be supported, both in the applications and in the generated journals. A goal of my work was to remove as much language-specific information as possible from the applications, and to make it easy for users to change the language in use. Also, since the values in the knowledge base are stored in a ‘universal’ language, there had to be efficient ways to transform the values to whatever language or journal declination the user preferred (usually the ones used in the template context) - for instance, a user in Italy should be able to view information contained in a knowledge base in Sweden, and a clinician should be able to hand a layman’s summary of an examination to a patient right after the examination. This *could* be done in the earlier system, but in a tedious manner requiring different text files for different languages and huge translation files. A goal of my work has thus been to facilitate and make more flexible the context-dependent use of the examination knowledge base as well as the applications used to visualize and work with the information in it.

In order for an application to be useful and productive, it is important that it is appealing to the user as well as not being too complex. Thus, a goal has been to make the developed applications as flexible and user-customizable as possible without introducing too much complexity for the user to consider. Java is an excellent language to work with when wanting the capability to easily switch and work with various look and feels of an application, since this concept is an integrated part of the Java Swing GUI framework.

2 Methodology

When starting my work at the clinic (approximately 2001-06-01), I did not expect or plan for it to develop into my thesis. The time I had for development (approximately 2 months) was rather short and there already existed a previous system with a lot of the major ideas and issues surrounding it already resolved. The initial goal was simply to rewrite the existing system, written in Objective-C, into a system written in the Java programming language. The primary reason for rewriting to Java were that no more licenses for running Objective-C applications on Windows were available, making it impossible to distribute the applications to new users. As a consequence, I did not initially deem it necessary to follow some large-scale development process, even though I tried to keep in mind some general and essential steps from the waterfall development process (at the time, I had no idea what 'iterative development' was all about). Later on, when initiating the second iteration in summer '02, I knew a lot more about iterative processes and object-oriented design. This led to a great deal of refactoring and restructuring of the previous code taking place, as well as introducing new rather advanced functionality in the applications, such as being able to print styled journals wrapped in page templates in a WYSIWYG (What You See Is What You Get) fashion.

2.1 Development History

Initially, a very short analysis of the situation was conducted (consisting of meeting some users of the existing system and letting them demonstrate how they used it), resulting in an amount of issues surrounding the current applications as well as ideas and suggestions on upcoming versions. The issues and ideas from the users were considered, followed by some (rather ambitious, considering only 2 months of development) sketches of the proposed applications that were presented to the users at the clinic. The users had no objections to the proposed solutions and thought they were nice, so the sketches became the de-facto user requirements (the proposed applications should work as presented to the users). At the time, I had not done too much Java application development, and was rather unsure about the time it would take to implement the prototypes being presented as well as the time needed to learn everything required to be able to implement them – nevertheless, the time to learn as well as the time necessary to actually code was estimated to be sufficient, so the work began.

New suggestions and ideas were continually being added during development, and at the end of the summer, it was decided that I would continue my work at the clinic as my thesis during autumn '01. The methodology used was concept-to-code, where a concept was sketched (using no particular method for visualizing) followed by being coded. Some basic design principles were used, such as thinking observer-observable and using a basic model-view-controller design such that upper layers of objects should not be visible to lower layers. As time progressed, it became notoriously difficult to see the overall structure of the applications and references had to be passed around rather chaotically in order for the different objects to be able to communicate, resulting in an entangled - but working - system being presented at christmas '01. Screenshots from the applications as they were christmas '01 are displayed in figure 2.1.

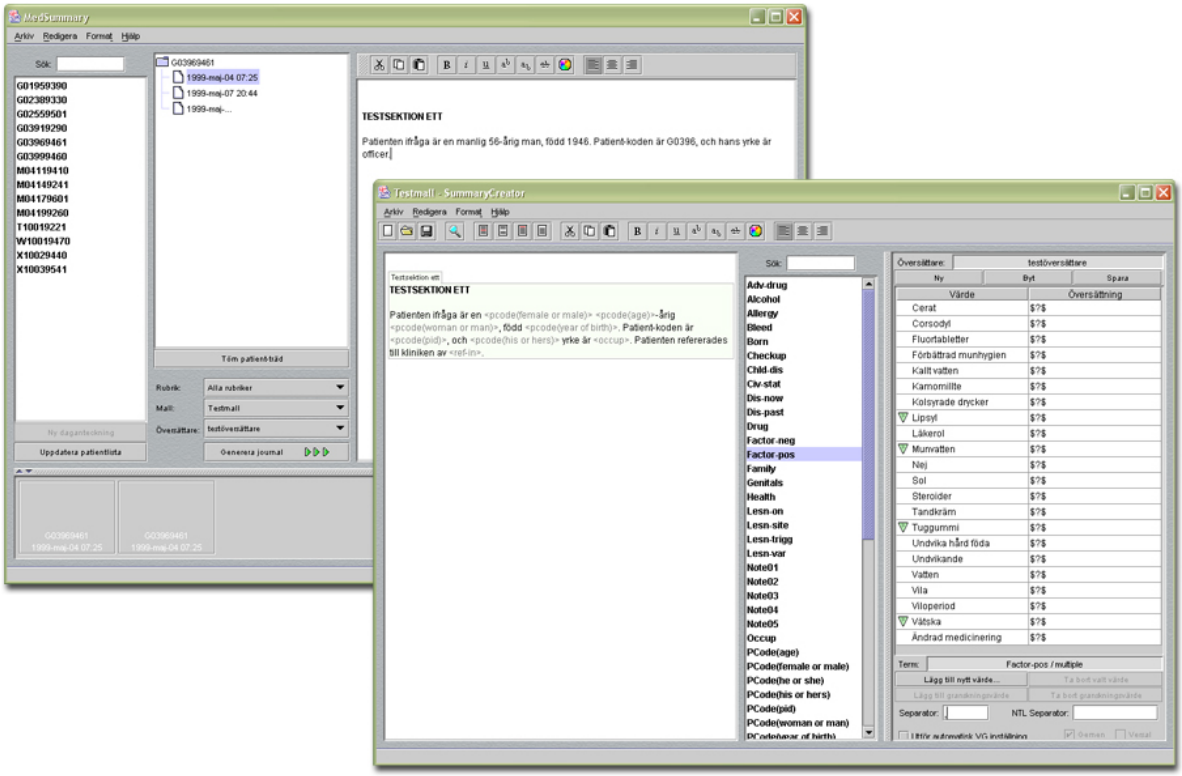


Figure 2.1 – The applications as they were christmas ’01

Since one major goal of the MedView project is to produce applications that can be deployed and distributed to users outside the clinic, it is of vital importance that the applications are well-designed in order for future further development and maintenance to be feasible. The applications presented at christmas ’01 were not ready for such deployment and distribution because of the lack of proper user testing as well as their rather muddled design. Furthermore, the major application for creating the examination records (an application being developed concurrently by other developers) was still not complete in its Java version, so any use of the system would still require running some of the previous Objective-C applications.

When attending the course ‘Objektorienterad Systemutveckling’ in spring ’02 at Chalmers University of Technology in Gothenburg, I discovered new approaches to object-oriented design such as design patterns and the Unified Modelling Language (UML), as well as new general work-flow processes structuring the development process such as the iterative Unified Process (UP). At the same time as I was taking the course, I began restructuring and refactoring my design – both as a way to facilitate my learning and as a way of closing in on the goal of being able to release an industrial-strength application system. Concurrent with the restructuring process, issues that arose from user testing of the applications I had released at christmas were also attended to and integrated into the restructured solution. During summer and early autumn 2002, I continued working with the restructuring and refactoring of the design, as well as improving and adding functionality to the system (such as, for instance, adding the option to view journals in a WYSIWYG (What You See Is What You Get) fashion by displaying pages and being able to print them exactly as they are displayed). Screenshots from the applications as they were in autumn ’02 are displayed in figure 2.2.

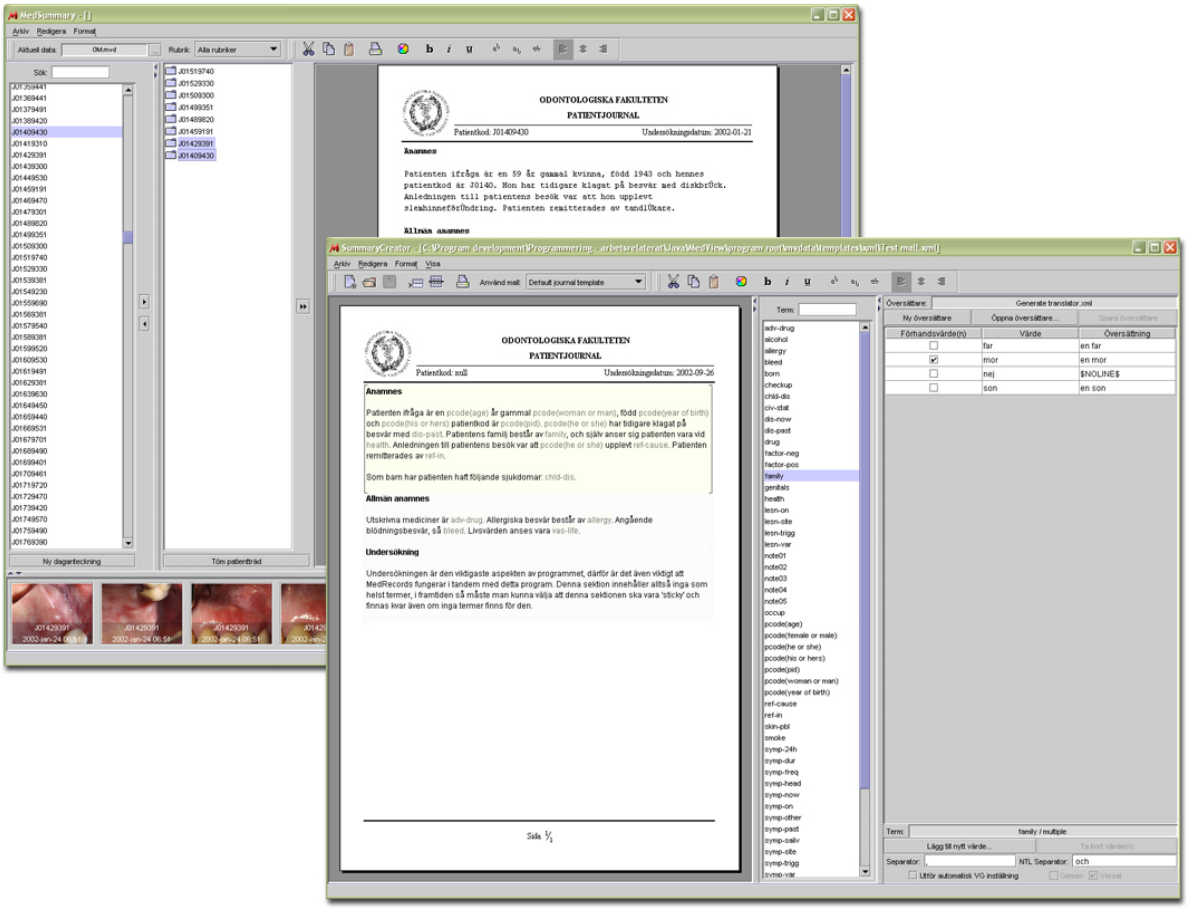


Figure 2.2 – The applications as they were in autumn ’02

At the time of this writing, there are still some minor issues to deal with regarding the applications, even though all major issues have been resolved. More user testing is necessary in order to unravel and fix bugs in the system, but the major cases of use have been tested and shown to work fine.

2.2 A Unified Process Retrofit

In retrospect, the development process used in my thesis can be seen as an iterative process consisting of two iterations – the first during summer and autumn ’01, and the second during summer and early autumn ’02. Initially, since applications already existed and simply were to be rewritten, not much analysis and requirements work was deemed necessary – the requirements were that the new applications written in Java should be like the older ones written in Objective-C. Seen in retrospect, it now seems like it might have been a good idea to have performed a more detailed analysis of the situation before the actual coding began. Coding began almost straight away after some initial light probing of the existing applications and user suggestions. A lot of time in the first iteration was spent on learning necessary Java techniques followed by using these techniques to implement the applications. From a Unified Process perspective, the design step in this first iteration (between the analysis and implementation phases) basically consisted of sketching the concepts (at least the ones complex enough to justify sketching) and then implementing based on the sketches. A large design overview of the system was simply not feasible to sketch and continually update by hand (later on, this was accomplished using *reverse-engineering* with the help of a UML

CASE tool). After some rather sparse testing, the applications were released to the users at the clinic at christmas ‘01.

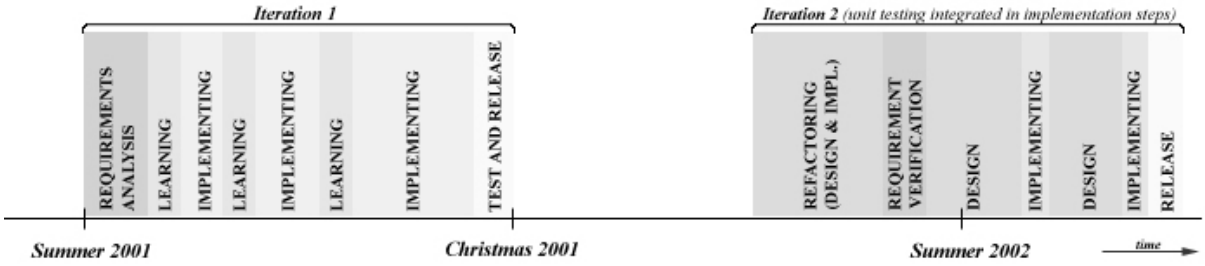


Figure 2.3 – Overview of my thesis development

With the release came the opportunity for users to test the applications and provide user feedback. When starting the second iteration, user feedback from the earlier release was available as well as a multitude of new knowledge in object-oriented design. The major part of the second iteration consisted of refactoring and introducing design patterns into the earlier applications, which improved application maintainability and extensibility dramatically. Seen from a Unified Process perspective, you could say that the requirements and analysis parts from the first iteration were deemed sufficient enough for the second iteration as well, and the main effort in the second iteration was put into the design and implementation steps. Several minor design-implementation iterations were conducted within the second major iteration, and unit testing occurred continually during all development.

Unit tests were performed on all refactored code and design, and parts of the system were integrated with other parts only after being subjected to thorough unit testing. Seen from a Unified Process perspective, each iteration should end with a user evaluation and testing phase of the iteration release - at the time of this writing, the major user tests finalizing the second iteration have not yet been performed, but the plan is to deploy the current system at the clinic so the users can thoroughly test and evaluate before considering the second iteration to be concluded.

3 Analysis

3.1 The Knowledge Base (KB)

The knowledge base has a structure consisting of a collection of *definitions*, where each definition represents one unique examination. An examination can be viewed as a collection of equations defining that particular examination, such as the one shown in figure 3.1.

```
Occup = dentist
Ref-in = doctor
Mucos-site = tungrand höger sida
Mucos-colr = röd
Vis-cause = kontrollundersökning
Next-app = 2 veckor
```

Figure 3.1 – A collection of definitions defining a small part of a fictive medical examination

The data making up the knowledge base is input by the clinician using an application called *MedRecords*. The input is done during the actual examination, making it imperative that this is done in an efficient and easy-to-use manner. The file format used to store an examination record is called a *tree-file* format – signifying that it has a tree structure. Figure 3.2 shows how a part of this conceptual tree structure might look. For an example of how an actual tree file can look, see appendix A.

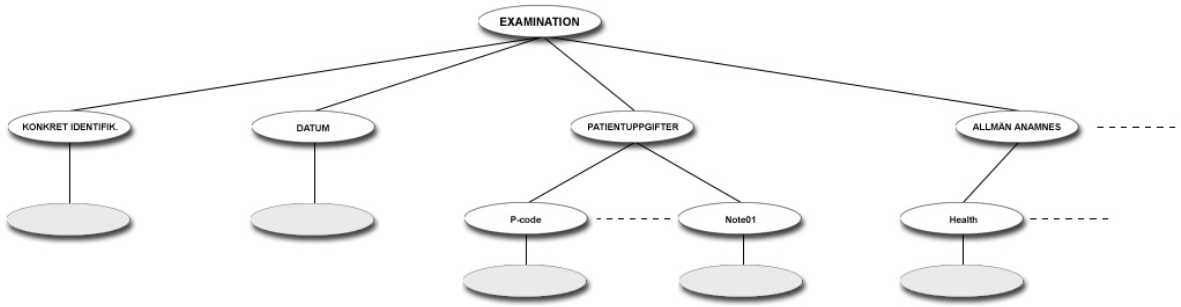


Figure 3.2 – The conceptual tree file structure

The information contained in the knowledge base is primarily used in two different ways – 1) to display information about a patient to the clinician in the examination room prior to (and during) an examination of the patient, and 2) to analyze and learn from the knowledge base content as well as to search for patterns, which is usually performed by the clinician from his or her desktop computer [5].

3.2 Previous Journal Generation Procedure

Prior to my development, there was no specific application for developing the template files used to generate the examination summaries – rather, a standard text-editor was used to create the necessary templates (usually as RTF (Rich Text Format) text files) together with a text file defining how examination values should be translated. The idea was that the system should be conceptually simple enough for a user to be able to produce his or her own template files and experiment with them in order to find the combination that produced the best output at generation. Since user-customizability was such a central issue, the choice of what kind of NLG system to use fell on one using a simple approach to NLG (a *shallow* system), i.e. one not requiring deep linguistic expertise. For more information about shallow and deep NLG

systems, see the introduction section as well as [7]. Figure 3.3 displays how parts of the various text-files surrounding text generation could look, for a larger example of such a collection - see appendix A.

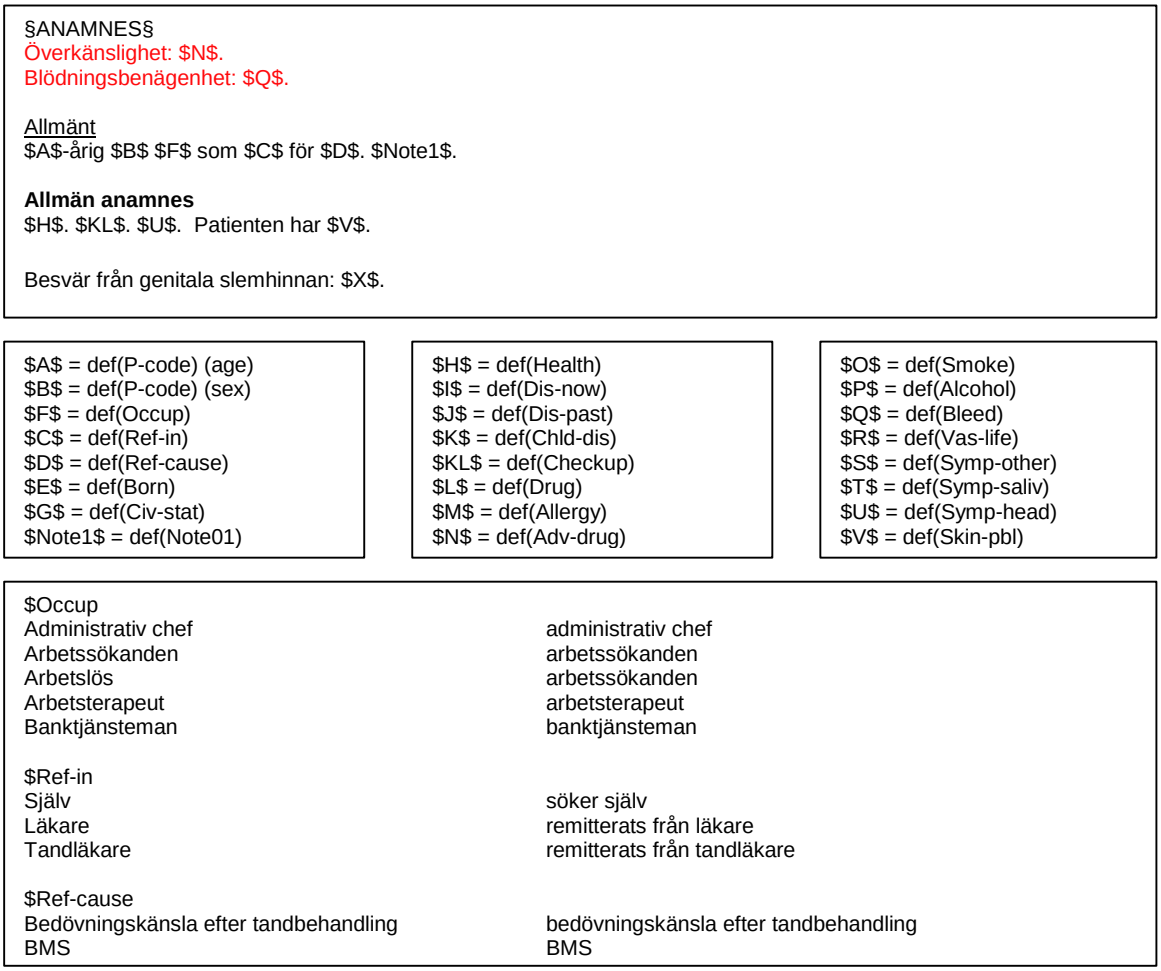


Figure 3.3 – Small excerpts from the various text files necessary for journal generation

Even though the previous approach works, creating the template and the translations with a text editor can be strenuous – especially when dealing with large files. Furthermore, viewing the translations that exist for a certain value requires searching in a huge text file (over a hundred pages long), which can also be a very tedious process. There is also a need to be able to couple more information to a translation of a value than just the actual translation, like information about when it was last modified in order for a moderator to check newly added translations for correctness and that they fit into the general scheme.

Another important issue is one about letter case. It might be the case that some values always should be in initial upper case (for instance, values for the ‘born’ term, defining in which country a patient was born) while some should always appear in initial lower case. The matter of case is also dependent on the sentence context – for instance, if a translation initiates a sentence it should appear in initial upper case, in all other cases it should appear in initial lower case (if the values are not of any special type that always should have initial upper case such as a country). As can be seen from figure 3.3, a lot of the specified translations are simply the value with initial lower case – a desirable feature would be to be able to tell the

system that a certain term should always have its translations adjusted to initial lower or upper case except when in special sentence contexts - such as when the translation initiates a new sentence. If done this way, the need to specify translations that simply change the initial case of the value would be removed - the actual value would be used at generation time and the system would decide what kind of initial case the value should have based on the users choice for the value's term (also known as the value's *attribute*). When discussing the matter of letter case in the earlier system with the users, it was clear that the case issue was a problem and that generated journals often contained mixed case that had to be adjusted manually. Furthermore, having to switch between the various text files back and forth with a text editor when creating the set of template files is tedious – it would be better if all information somehow could be displayed at once. For more information about how text generation previously was done in MedView as well as more general information about NLG, see [7], [9], and [10].

3.3 MedSummary

One of the developed visualization tools for viewing data from the knowledge base is the MedSummary application, written in Objective-C, with the purpose to view the information collected in the MedRecords application [5]. A screenshot from the MedSummary application is shown in figure 3.4.

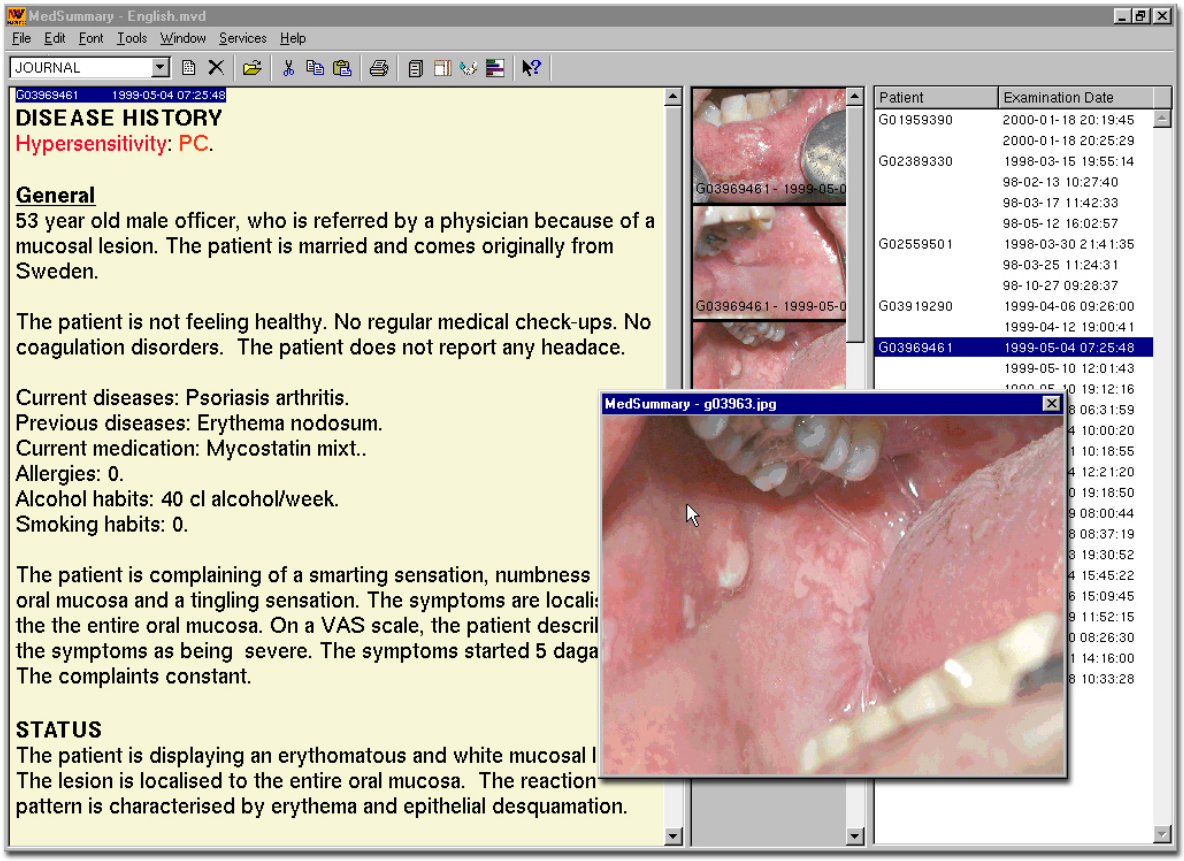


Figure 3.4 – A screenshot of the previous MedSummary application

The application is used primarily for selecting an interesting set of patients and for viewing summaries of their associated examinations. If digital photos have been taken during the examinations, these are also displayed. The summaries can be printed out directly from the

application, although without any surrounding page template (i.e. a template containing headers with patient- and examination information, footers, page numbers, graphics, etc.).

If the generated journal text is to be printed out in a page template format, a user will have to manually copy the generated text from the MedSummary application and paste it into a word processor containing the template. Furthermore, information about the patient such as the patient code and the date of the examination will have to be manually entered in the appropriate places of the template for every printout. Obviously, a way to print out the generated text directly from the application in such a template, automatically formatted with information about the current patient, would greatly simplify the printout procedure.

When it comes to the manner in which interesting patients and their associated examinations are displayed, a more intuitive (in my opinion) way would be to display them in a tree format, where patient nodes are branch nodes grouping the examinations for each specific patient as child leaf nodes. Figure 3.5 illustrates this concept by showing how the patients and their examinations are displayed in the newly developed application.

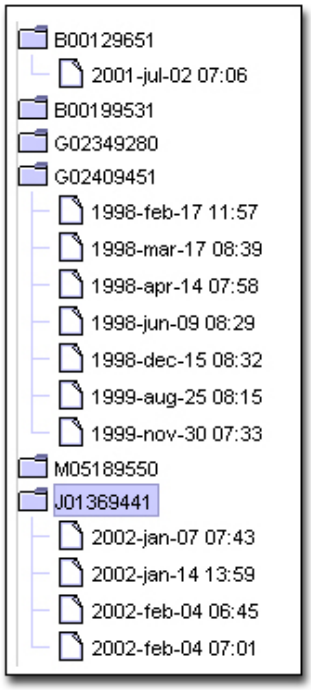


Figure 3.5 – Grouping interesting patients and their examinations in a tree format

Some further minor issues regarding the application that could be improved involve how one chooses which template files to use as well as how information about the images are displayed. Deciding which template and translation files to use is done by opening the settings dialog and choosing the files by browsing - it might be a better idea to display (and be able to choose) the current template in the toolbar or in a menu. In practice, since producing different template files is a rather tedious process for an average user (say, a dentist not used to computers), there are not many different template files in use, so this feature is seldom used. When it comes to displaying the images associated with an examination, it can sometimes be hard to see the date and patient code for an image as it is displayed in its minimized version in the vertical panel seen in figure 3.4. Furthermore, once the image is displayed in its full size,

only the name of the image file is shown in the title bar of the dialog – some kind of information about the patient and examination date would be desirable to have there.

3.4 Analysis - Conclusions

Table 3.1 summarizes the major issues with the current system, as well as some ideas on how they could be resolved or improved in an upcoming system.

Problem	Solution
The need to switch between template, translator, and term files during their construction is tedious.	Provide an application that provides the user with the opportunity to edit all these three components at once, without the need to switch between them.
More information about a translation than just the actual translation is needed, such as when the translation was last modified.	Model a translation as an object containing all desired additional information related to the translation, such as the actual translation and last modification date.
Letter case issues leads to less reusability of a translator since it is highly coupled to a certain template.	Improve the text generator so that it knows how to adjust translation's case automatically according to the textual context.
It is too lengthy a procedure to preview generated output from a certain template and translator combination during their creation.	Provide ways to preview templates and translators easily and quickly during their creation.
Printing generated journals and summaries contained in a page template requires copying and pasting into a a third-party application, such as Microsoft Word. This is undesirable, since it is tedious and imposes a need for additional software.	Provide functionality to switch page templates surrounding the generated journals and summaries, and the means to print the generated journals contained in such templates directly from the application.

Table 3.1 – Problems in earlier system and solution suggestions

4 Design

In the spirit of agile and iterative development, the design has been revised and systematically improved during the course of the entire project. When elaborating the design, the system was divided into major parts, followed by dividing each major part into more specific subparts being developed one by one, including thorough unit testing and making sure that each part worked correctly in isolation before integrating it with other parts. The major architectural design decisions were decided before more specific component development was attempted, even though these architectural design decisions were not rigidly fixed and were often subject to scrutiny followed by revision during development. Since I have learned large parts of the methodology and practices of object-oriented software development in parallell with the development of my system, restructuring and adaptation has played a central role, and refactoring the code and design has occured on several occasions.

4.1 Text Generation Strategy and Algorithm

As described earlier, the text generation approach used in the system is based on filling the template text with 'slots' where the actual values from an examination record – after being processed by the translator – are inserted into the slots at generation time. The template is composed of styled text divided into *sections*, where each section may contain a number of *terms*. A section represents a conceptual part of a journal, such as the 'anamnesis' part describing the patient's medical history or the 'diagnosis' part describing the clinician's diagnosis of the patient. A term is analogous to a 'slot' as described above, i.e. in a medical examination record each term contains zero, one, or several values that are to be processed and inserted into the slot for the term in the template.

At the time of this writing, the terms can have five different types: *regular*, *multiple*, *interval*, *free*, or *pcode*. The regular type represents terms that may have only one value associated with them, like the 'born' term (a person can only be born in one country). Such terms are easy to deal with – the translator is simply queried for the translation of the value found in the examination record, this translation is then inserted into the slot for the term. Terms that may contain several values in the examination record have a type of 'multiple', for instance – the 'family' term may have the values 'father', 'mother', 'son', 'daughter' etc. For terms of type 'multiple', the user can specify how to separate the translated values when they are placed in the generated output. There are two separators possible for the user to specify: 1) the 'regular separator' used when there are more than two values, and 2) the 'next to last' separator (abbreviated NTL) used to separate the two last values. Figure 4.1 displays an example of the use of the separator and NTL separator when applied to the term 'family'.

Template text :	'The patient has a family consisting of <family>.'
Values:	'a father', 'a son', 'a mother'
Separator:	','
NTL separator:	', and'
Output:	The patient has a family consisting of a father, a son, and a mother.

Figure 4.1 – Example of the use of separators for a multiple type term

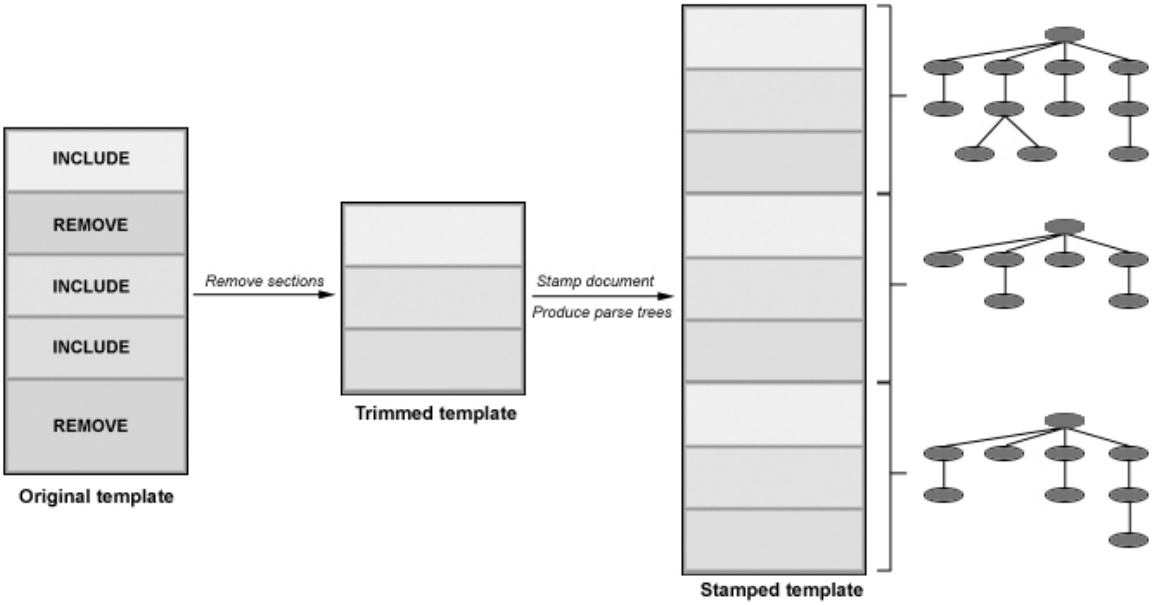
The third type of term is the 'interval' type, defining terms that have a floating point value specified in the examination record. The translator for these terms contains intervals, which specify how floating point values contained in these intervals should be translated. The fourth type of term is of the 'derived' type – terms with this type are special since they are not contained in the examination records, instead the values for such terms are derived from the patient identifier for the patient whose journal we are generating. The only place the user will

see terms of this type is in the SummaryCreator application when creating the template. The fifth and final type is the ‘free’ type, which is even simpler than the ‘regular’ type – a term with the ‘free’ type can contain anything as a value in the examination record, and whatever it contains is inserted into the term’s slot at generation time without being processed.

For generating journals, four components (the template, the translator, the sections of interest, and the examination records) are passed as input to a so-called *generator engine*, which processes the components and produces an output document (the generated journal). As of date, the generator engine implementation utilizes a tree structure (which I refer to as the *parse tree*) for processing the template document. The idea is based on the Composite GoF design pattern, and consists of creating a parse tree for each examination record to be summarized. Each parse tree is represented by its root node, and each node in the tree contains references to the template document (i.e. the styled text content making up the template) as well as start and end positions in the document text flow where it is allowed to operate. Thus, the root node of a certain examination parse tree operates within its assigned part of the document, and the children of the root operate within their assigned parts, which in turn are contained within the parent’s boundaries. Once the parse trees have been constructed, each of the root nodes is asked to process the document content they are ‘attached to’, which will result in the root nodes asking their children to process their content, resulting in each child asking its children to process their content and so on.

When creating the generator engine, you *build* the components onto the engine by using a *builder* (an object dedicated to the task of building its product, which in this case is the generator engine). When building the template model onto the engine, the builder class will clone the template model and attach this cloned copy to the engine. Thus, it is safe for the generator engine to modify the attached template model, since it is modifying only a copy of the original. The same reasoning applies to the translator model – the translator model is cloned before it is attached to the engine.

When requested to generate a document, the engine uses the document contained in the cloned template model as a starting point. The sections not to be included in the output are first stripped from the list of sections contained in the cloned template model, followed by ‘stamping’ the remaining template text so that multiple examinations can be summarized based on only one template - this process is visualized in figure 4.2. After the section removal and stamping, the parse trees are created as discussed above (one tree per examination) and ’attached’ to their respective parts of the document where they are allowed to operate. When the parse trees have been built and their respective nodes thus attached to the document, the nodes are asked to process their parts of the document. After all nodes have processed their part of the document, it is returned by the generator engine to the caller.



Anamnesis

The patient is a Pcode(age) year old Pcode(female or male) Occup. Ref-in.

Status

Palpatation of Palp-site.

Tentative diagnosis

Tent-diag.

5. *Remove unwanted sections…*

Anamnesis

The patient is a Pcode(age) year old Pcode(female or male) Occup. Ref-in.

6. *Stamp the text for multiple examinations…*

Anamnesis

The patient is a Pcode(age) year old Pcode(female or male) Occup. Ref-in.

Anamnesis

The patient is a Pcode(age) year old Pcode(female or male) Occup. Ref-in.

Anamnesis

The patient is a Pcode(age) year old Pcode(female or male) Occup. Ref-in.

Figure 4.2 – The initial processing of the template document

Each node, when asked to process it's assigned content, returns a boolean value indicating whether or not the content it represents should be included in the generated journal. In this

way, the parent nodes can decide whether or not they, in turn, should report to *their* parents that they are to be included in the generated journal based on the return values of *their* children. Delegating the responsibility of processing the template to each node in the tree structure simplifies working with the different aspects of the text generation.

1. For each section not to be included in the generated output:
    1.1. Remove the textual content that the section represents from the template document (text).
    1.2. Remove the corresponding section model representing the section from the list of section models contained in the template model.
2. For each record in the array of examination records to be summarized:
    2.1. If the examination record is not the first in the array, spawn the template text and append it to the end of existing template text (if the record is the first, the original text will represent it).
    2.2. Associate the start offset of the spawned text with the record (will be 0 for the first).
3. For each record in the array of examination records to be summarized:
    3.1. Create an examination parse node for dealing with the processing of that record's part of the template document (text).
    3.2. For each section contained in the template model (the ones remaining after step 1):
        3.2.1. Create a section parse node for dealing with the processing of that section's corresponding text. Attach the node as a child to the examination node.
        3.2.2. Extract line (sentence) information by parsing the section's corresponding text.
        3.2.3. For each line extracted in step 3.2.2:
            3.2.3.1. Create a line parse node for dealing with the processing of that line. Attach the node as a child to the containing section node.
            3.2.3.2. For each term 'slot' contained within the line:
                3.2.3.2.1. Create a term parse node (the specific node type depends on the type of the term) for dealing with the processing of that slot. Attach the node as a child to the containing line node.
                3.2.3.2.2. For those terms that are to be translated (have such a type): attach the term's corresponding translation model to the node for subsequent translation lookup.
4. For each examination parse node created in step 3.1:
    4.1. For each contained section child node:
        4.1.1. For each contained line child node:
            4.1.1.1. For each contained term child node:
                4.1.1.1.1. Retrieve the value(s) for the term as contained in the corresponding examination record.
                4.1.1.1.2. Depending on the term type, process the value (like translating it if it is of such type, perform VG adjustments, etc.).
                4.1.1.1.3. If the term has a translation of 'no line', indicate this to the containing line parse node (see 4.1.1.2 below to see how the line node deals with such an indication).
            4.1.1.2. If any of the term parse nodes indicates that a 'no line' has been encountered, remove the line node's corresponding text and indicate to the containing section parse node that the line was removed.
        4.1.2. If the section parse node, based on the processing of the contained line nodes, finds out that it contains nothing of interest, remove the node's corresponding text from the generated output.

Figure 4.3 – The text generation algorithm

Say a line node is asked to process the template content it is responsible for - the line node, in turn, will let all its children perform their processing and will register if any child node indicates that it should not be included in the generated journal. If a line node's children (might not be any if there are no terms on the line) all are to be included, it will process the line and try to produce as natural a line as possible. Note that 'lines' in this context refer to

textual sentences. The processing of the line consists of various textual processing, such as checking that the line is initiated with an upper case character, and that gaps exist between all commas and colon characters, etc. Thus, you can concentrate on the line processing when dealing with the line node, and leave the term processing to the term nodes (if there are any). This makes it very clear where you need to work for improving textual processing of lines in the future. When considering the processing of a line node's children - if some child node indicates it should not be included (for example, by containing a term with a value that translates to a 'no line'), the line node removes the textual content it represents (i.e. the line) from the output, and then returns a boolean value of *false* to it's parent indicating that it is not to be included. If the parent section node receives from all of it's child nodes that they are not to be included, it removes what it represents (special section markup characters etc.) from the output and returns a value of *false* to its parent (the examination root node). The algorithm for the entire process of transforming a template model and it's contained template document to the returned document is summarized in figure 4.3. For more information about the actual object-oriented design of the generator, see the section below describing the generator package.

4.2 System Architecture and Overview

When deciding the system architecture, the Layers [2] design pattern has been used. Functionality is grouped in such a manner that upper layers depend on lower layers, but not vice versa, which leads to the lower levels being more reusable and general, and the upper levels being more application specific. This way of structuring the design has the advantage that other applications developed in parallel or in the future have access to significant functionality without having to 'reinvent the wheel', thus reducing development time.

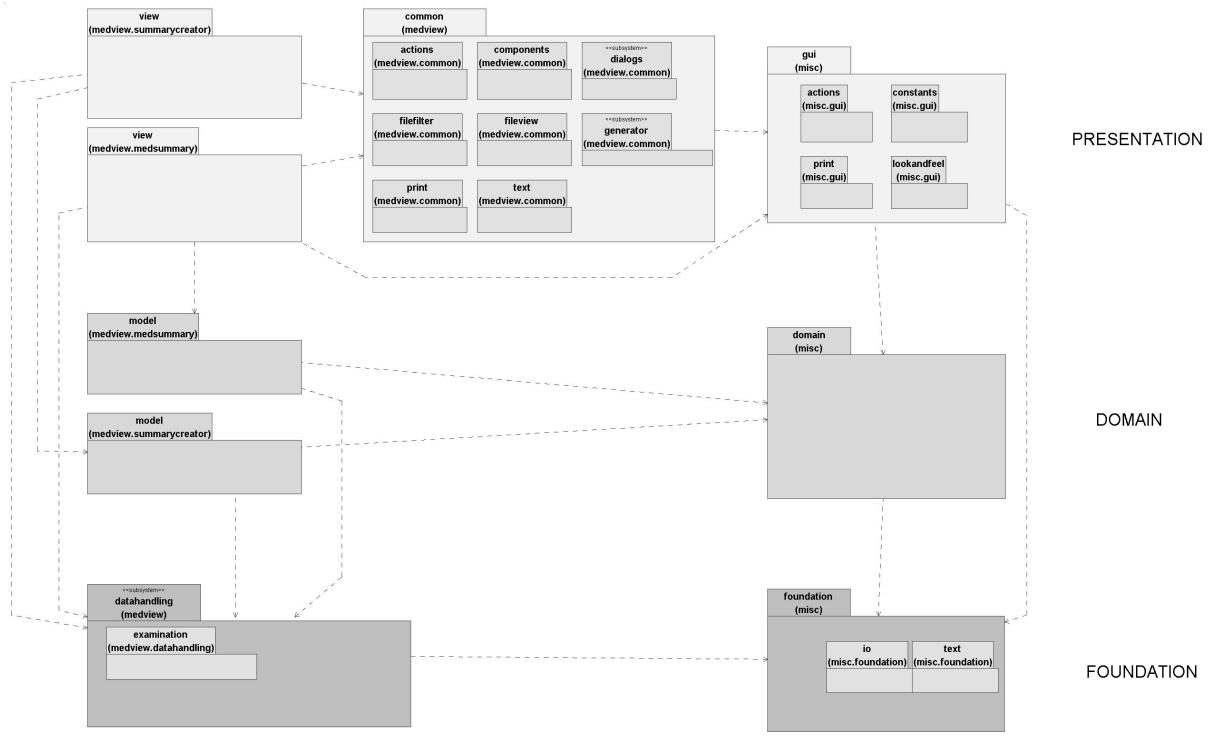


Figure 4.4 – Logical view of the system architecture

As can be seen from figure 4.4, there is no coupling from lower layers to upper layers, and general components have been extracted into separate packages (the packages to the right),

also divided in layers according to their logical place in system structure. The concept of a *package* is very similar to the concept of a directory in a filesystem – they both group related entities, where the entities in the file system are the files, and the entities in Java are interfaces and classes. For instance, a package ‘math.pi’ contains entities dealing with mathematical pi calculations. The various ‘misc’ packages contain components and logic that need not pertain only to my system context (the MedView context), but can be re-used in other applications and software. In the figure, the upper-left packages are most application specific and the least reusable, while the lower-right packages are most reusable and not coupled to any specific application.

The lower layer (the foundation / technical services layer) supply base foundation functionality, such as special data structures, being able to store settings, properties, and preferences from the upper layers, a framework for retrieving language-dependent text, retrieving media resources, etc. The more specific MedView base functionality is found in the medview.datahandling package, while application-independent foundation functionality is included in the misc.foundation package. The medview.datahandling package may use the misc.foundation package, but not vice versa. At the time of this writing, there are plans to move the medview.datahandling package into the medview.common structure since the data handling package is a common package shared by all applications – this will probably be done in the near future.

The middle layer (the domain layer) contains the core application domain logic, i.e. domain concepts and their relationships without regard to how they are to be presented to the user. The idea is that a certain visual representation of the objects in the domain layer can be replaced by another without having to recode core application logic. Thus, logic that does not pertain to visual representation should be dealt with in the domain layer – this closely resembles the idea in the MVC (Model-View-Controller) [3] design principle, which states that model objects should not have direct knowledge of view objects. When comparing the MVC and Layers design patterns, the domain layer can be seen as the ‘model’ part, and the presentation layer can be seen as the ‘view – controller’ part, where the presentation layer fulfills ‘controller’ responsibilities by having listeners listening for events from the domain layer and taking appropriate action when events are fired. Objects specific to the MedView context are located in the various medview subpackages, while more general domain components are found in the misc.domain package. The medview subpackages may depend on the misc.domain package but not vice versa.

The upper layer (the presentation layer) is responsible for visualizing the domain layer. It is a relatively thin layer that is primarily responsible for receiving and sending various user input to the appropriate domain object. Objects in the presentation layer listen to the various domain objects for events that affect their representation, and consequently update their appearance. The presentation layer is, generally speaking, the most application specific and least reusable layer, and should be made as thin as possible to avoid unnecessary duplication of work later on when a new visualization might be needed. It should be pointed out that the misc.gui packages *are* reusable even though they are located in the view layer, since they contain components and utilities that can be applied to all sorts of applications – not just applications within the MedView context. It is when considering the presentation layer in the MedView context that it is important to make the view objects as thin as possible, and place as much application logic as possible in the domain layer.

4.3 SummaryCreator

The SummaryCreator application is used to create the templates and translators used for generating examination summaries and patient journals. First, I will first describe the design and ideas behind the application's user interface and functionality, which will be followed by a description of the object-oriented design of the application.

4.3.1 Functionality and User interface

When designing the user interface for the SummaryCreator application, I had no previous application to use as reference. The general idea, though, is that the application should be conceptually simple enough for a user to be able to create his or her own templates and translators. Some shortcomings associated with the earlier method of creating the journal generation components were:

1. The text editor used to enter the journal templates did not provide any support when it came to visualizing the template structure. Rather, the user had to work with, and remember, special characters in order to denote section boundaries and term slots.
2. Available terms, the values they could contain, and the translations for these values were all kept in separate textfiles. This made it necessary to constantly switch between the files when creating templates, terms, and value translations. Also, the translation file could become very large (over a hundred pages long), making it tedious to work with.
3. The user had to keep a close eye on the letter case used in the translations, otherwise mixed-case journals were produced. Since the letter case used in the translator depended on the corresponding term's textual context in the template, the coupling between a certain template and translator was rather high, making it difficult to reuse a translator with another template.
4. There was no way to obtain a quick preview of generation-time output during the creation of template and translator files. The user had to run the MedSummary application, load the corresponding template and translator files, and then select from actual examination data in order to do this.

Figure 4.6 displays how the SummaryCreator application looks when running on a Windows platform (using the Windows look-and-feel, i.e. the look and feel of the application is like the look and feel of an application run on the Microsoft Windows platform).

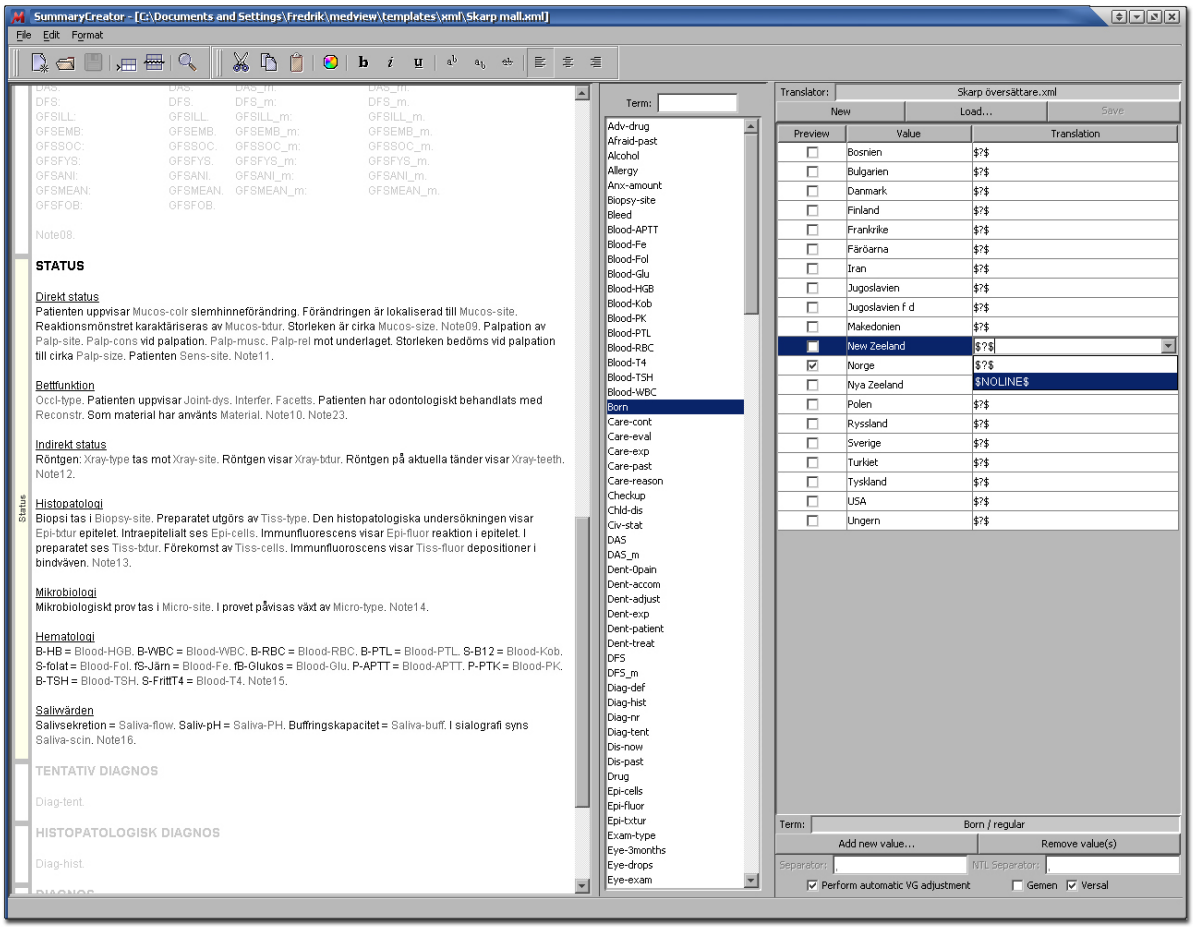


Figure 4.6 – The SummaryCreator application

As can be seen from the figure, the template structure is visualized graphically by a sidebar showing the section names and their corresponding extent in the template. The application highlights the current section (the one containing the caret) by coloring the section part of the sidebar differently than the other sections' parts, as well as dimming the corresponding text of the other sections in the template. The user does not have to manually work with special characters for specifying section boundaries, but simply chooses to create a new section and enters the section name and placement in the template, by using the 'add section' dialog displayed in figure 4.7.

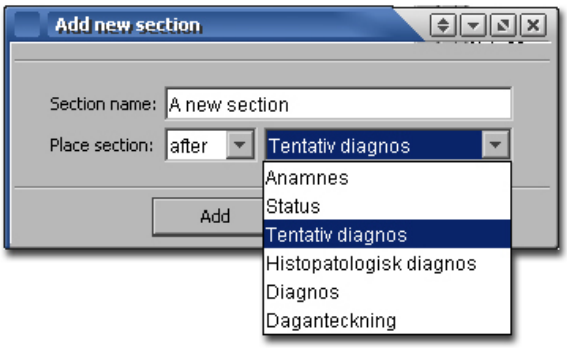


Figure 4.7 – Adding a new section

All available terms are contained in a list placed conveniently between the template and translator sections of the application. By selecting a term in the list, the possible values and their translations for the current translator are displayed to the right of the list. If you double-click (when running the applications on a Microsoft Windows platform) on a term in the list, the term is inserted into the template at the current caret location. Thus, there is no need for switching between three different files anymore - one for the available terms, one for the available term's values and translations, and one for the template - all components are displayed at once in one application.

The matter of letter case is also adressed - the application can be instructed to *automatically adjust the initial case* of the translations by selecting the 'perform automatic VG adjustment' checkbox. Whether or not the translations should automatically be adjusted to initial lower (gemen) or upper (versal) case can be selected. In this way you could specify, for instance, that all translations for the term 'born' (which are countries, that should always appear in initial upper case) should have an initial upper case, even if the user (by mistake perhaps) entered initial lower-case translations. Figure 4.8 displays settings corresponding to the choice of automatically forcing initial versal case adjustment to all translations for the term 'born'.

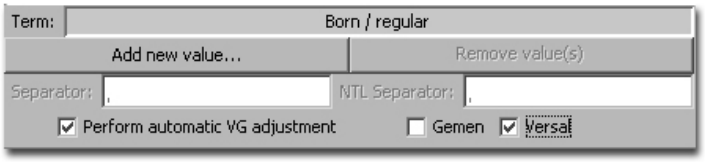


Figure 4.8 – Specifying automatic VG adjustment

A major functional improvement is the possibility for the user to quickly generate a preview of the currently worked-on template and translator pair. Each term contained in the translator contains a set of values that are tagged as being 'preview' values - when generating a preview, a simulated examination record is created based on these preview values. By viewing actual generated output during the creation of the template and/or translator, the user can quickly adjust the components based on the output. Figure 4.9 displays how the application can look after a preview of the current template and translator has been generated.

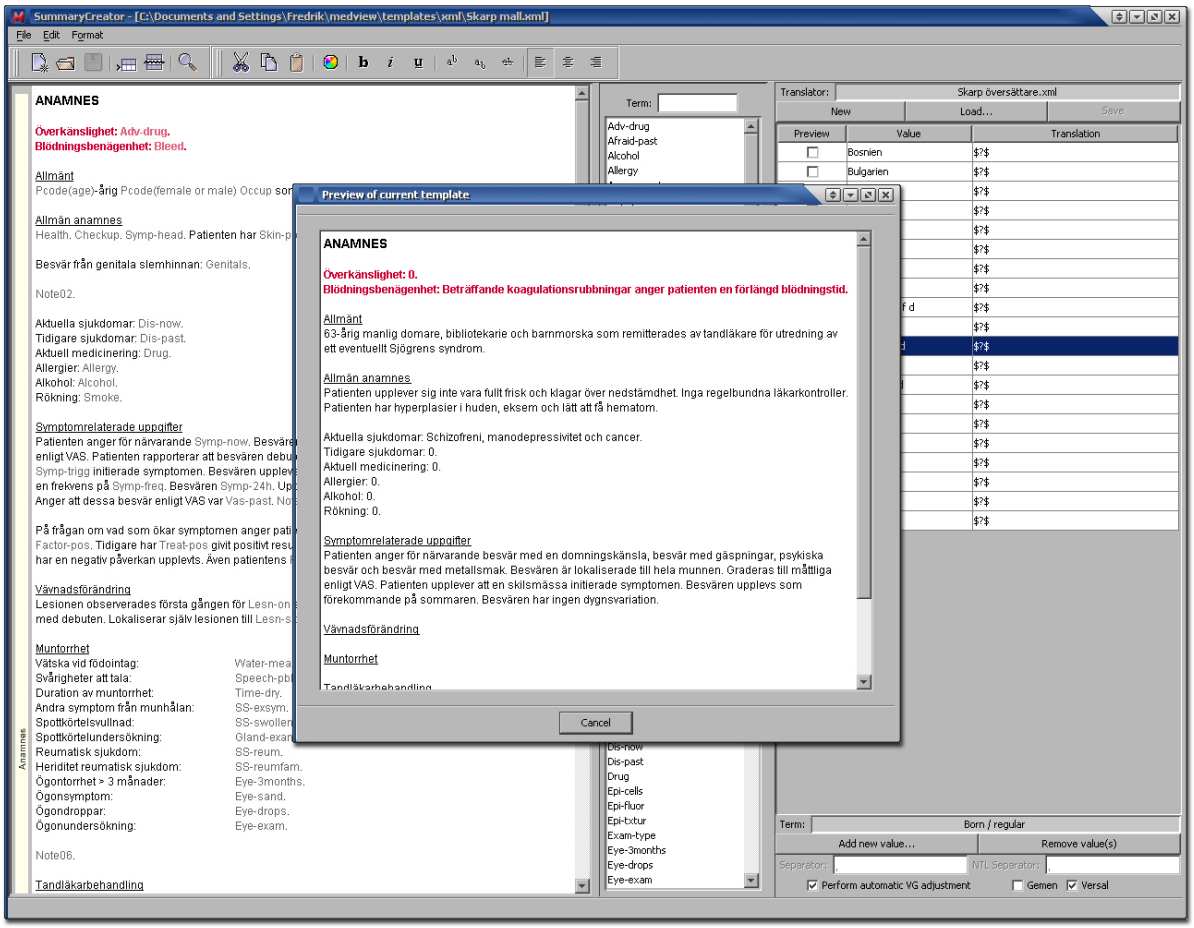


Figure 4.9 – SummaryCreator's preview functionality

4.3.2 Software Design

For the general structural design of the SummaryCreator application, I have used three major design patterns and principles, namely the Mediator design pattern, the Facade Controller GRASP pattern, and the MVC (Model-View-Controller) principle. The general (abstract, simplified) structure of the SummaryCreator application is shown in figure 4.10.

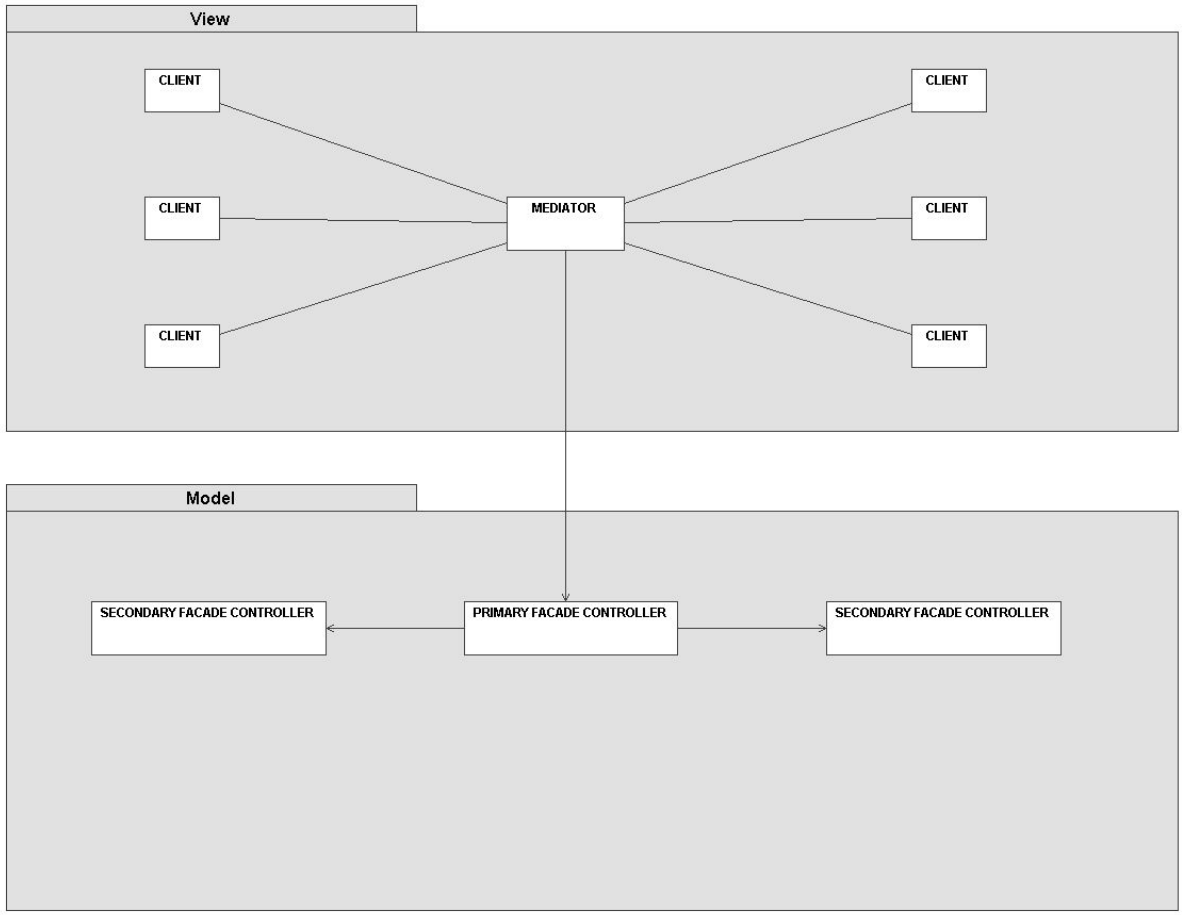


Figure 4.10 – The SummaryCreator application's abstract structure

Instead of having each client object contain references to whatever other client object it affects, it tells the mediator that it wants to have something performed. The mediator, containing references to all clients, will in turn tell the affected object(s) to perform whatever the initiator wants to have performed. In this way, a complex relationship between clients is eliminated, and a relationship between the client and the mediator is all that is needed. In order to further enhance reusability of the clients, the mediator is not referenced by the clients as a concrete application specific mediator class – instead the mediator implements a client mediator interface, specific to the client in question. The Mediator GoF [4] design pattern can be seen as a special case of the more general PV (Protected Variations) GRASP pattern [1], since it provides protection against possible variations in the structure of the view layer.

As mentioned earlier, the domain layer contains the core domain specific objects, but not the objects that are displayed to the user - these objects are located in the presentation layer. So when a user does something that should result in a domain object update (i.e. the user generates a *system event*), the view objects need to notify the domain objects of this, so they can deal with it accordingly. The Facade Controller GRASP pattern [1] states that a system event message should be passed to either a facade- or a use-case controller. Here, three facade controllers are used – one primary and two secondary. The primary facade controller is responsible for containing the secondary controllers and providing access to them, as well as providing simple responsibilities that do not pertain to the responsibilities fulfilled by the secondary controllers. The secondary facade controllers have high cohesion within a certain

domain concept, in this case the template and the translator. Furthermore, they are highly reusable since they do not couple to anything else within the structure (see navigability of the dependency lines in figure 4.10 between the primary and secondary facade controllers). In the future, these secondary facades might be decoupled from the context of the SummaryCreator application to a more general context (perhaps the medview.common structure). Because of this loose coupling between the secondary controllers and the rest of the SummaryCreator context, such a decoupling will be relatively easy to perform. Figure 4.11 displays the real structure of the SummaryCreator application, which can be seen as the 'implementation' of figure 4.10.

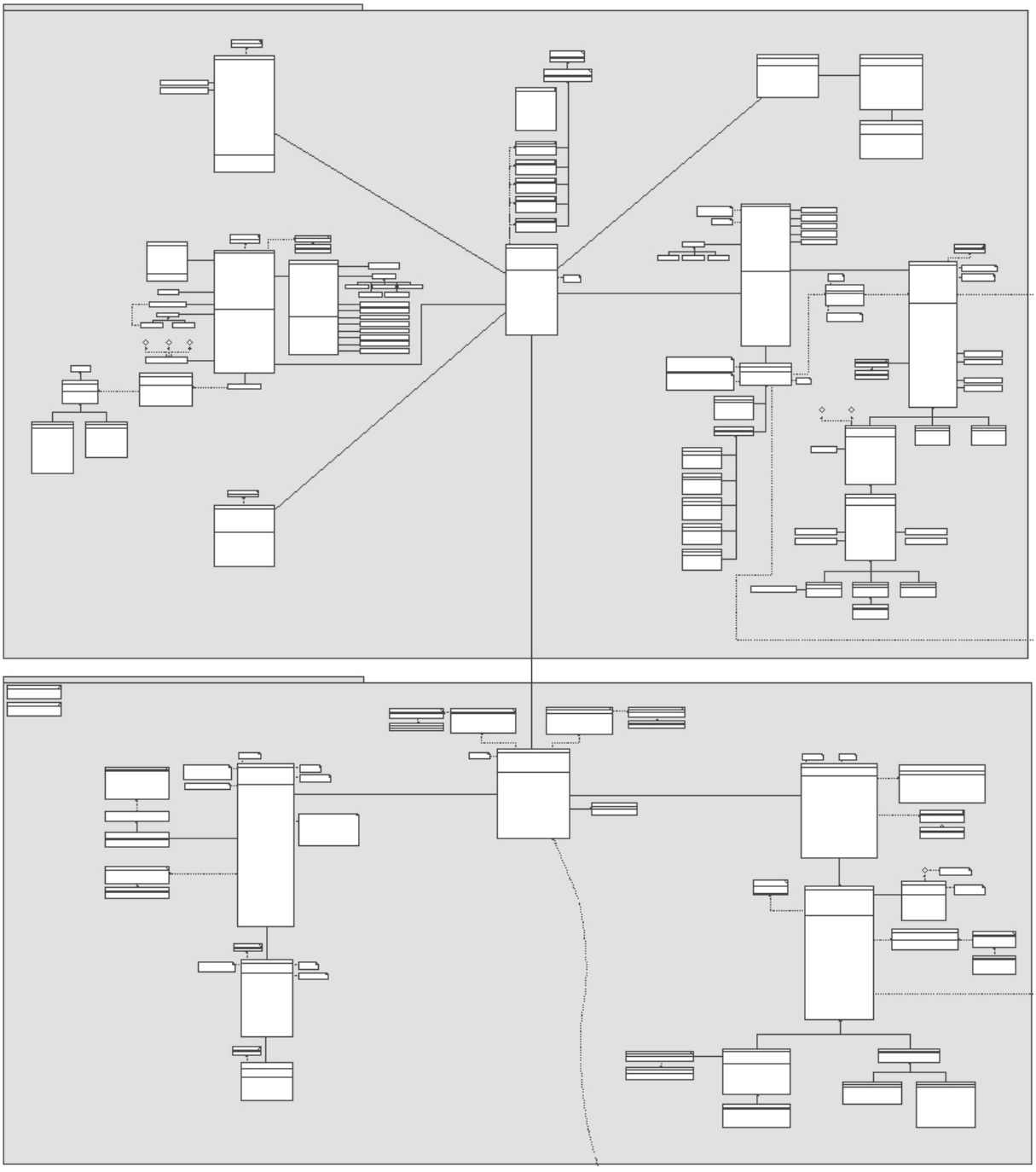


Figure 4.11 – The SummaryCreator application structure

As can be seen when comparing figures 4.10 and 4.11, the general structure is the same. The mediator class is the ‘SummaryCreator’ class in the view subpackage, while the clients are the various panels composing the application. The mediator class implements one interface per client, thus decoupling the panels from one specific view context. When, say, the term list panel wants to obtain the terms to list, it first asks its mediator for a reference to the model, and from there queries the model for the current array of terms - this interaction is shown in the sequence diagram displayed in figure 4.12.

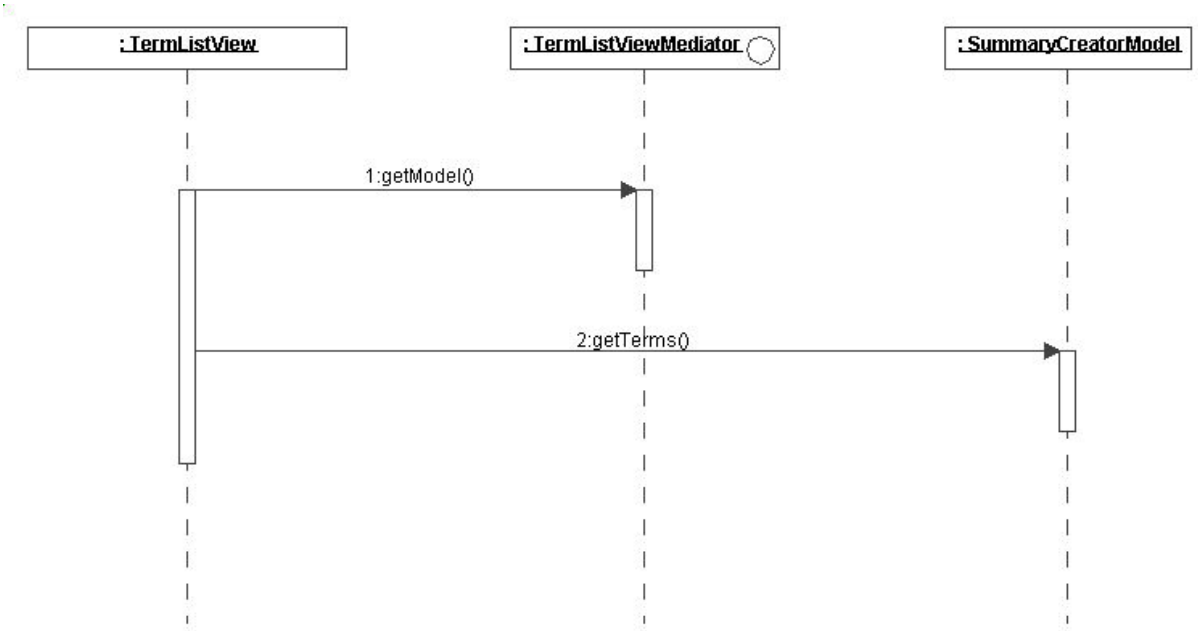


Figure 4.12 – A client obtaining the terms via the mediator

4.3.3 Model (Domain)

There are three major concepts in the domain of the SummaryCreator application - the template model, the translator model, and the collection of other ‘smaller’ concepts such as the list of terms and the location of the template and translator models. Thus, the domain layer is composed of these three major parts – the TemplateModel class dealing with template-related functionality, the TranslatorModel class dealing with translator-related functionality, and the SummaryCreatorModel class acting as a central point of access down to the domain layer as well as providing the ‘smaller’ functionality as described above. Furthermore, the SummaryCreatorModel class is responsible for providing access to the template and translator models. Thus, the TemplateModel class can be seen as an Information Expert [1] in matters pertaining to the template, the TranslatorModel class as an Information Expert in matters pertaining to the translator, and the SummaryCreatorModel class as an Information Expert in matters pertaining to finding out where the current models are located as well as matters dealing with various simple domain concepts, such as the keeping the list of available terms. The model layer of the SummaryCreator application is shown in figure 4.13.

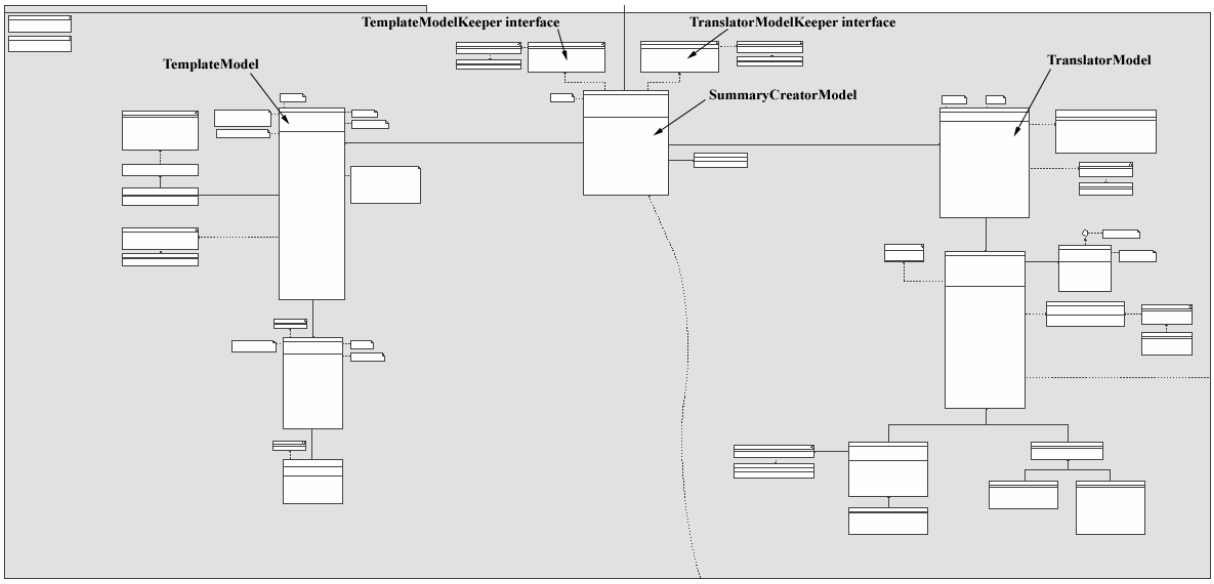


Figure 4.13 – The model layer of the SummaryCreator application

4.3.4 View (Presentation)

The view layer of the SummaryCreator application is divided into six major parts: 1) the central SummaryCreator class, which acts as the mediator and the 'glue' that references all other major view classes, 2) the SummaryCreatorMenuHandler class, which takes care of all things related to the menus in the application, 3) the SummaryCreatorToolbarHandler class, which deals with the toolbars, 4) the TemplateViewWrapper, which wraps the various ways of viewing the template (such as viewing it in a page or normal text layout), 5) the TermListView, which displays the terms contained in the SummaryCreatorModel as well as listens for changes in terms to display, 6) the TranslatorView, which visualizes the translator domain structure and presents the user with options of modifying the translator. The central SummaryCreator class implements all of the client mediator interfaces, so each client sees it in its own way. This fact that the SummaryCreator class implement the various mediator interfaces enhances reusability of the clients since they can be reused in another view by simply creating a new central mediator object implementing the client interfaces. The view layer of the SummaryCreator application is shown in figure 4.14.

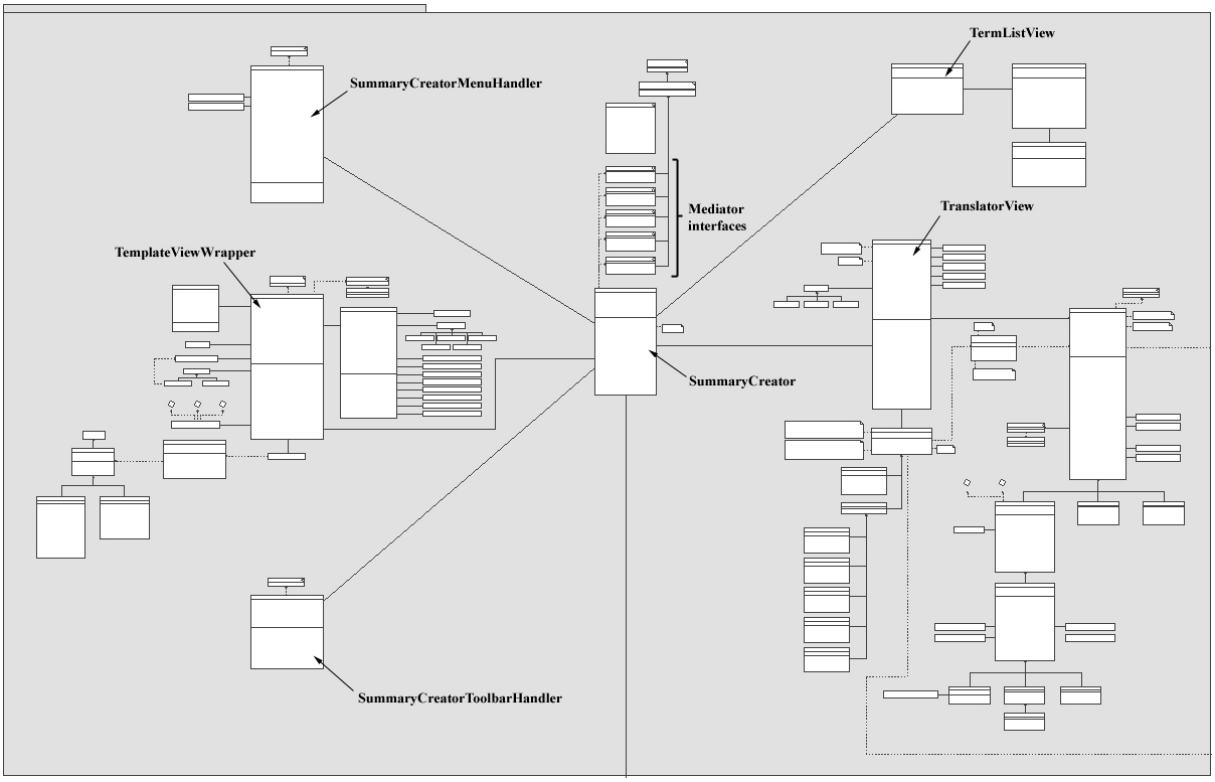


Figure 4.14 – The view layer of the SummaryCreator application

As can be seen from the UML diagram in figure 4.14, the only point of access to the domain layer is via the mediator SummaryCreator object. This does not mean that the various clients do not contain references to objects in the domain layer, but that they *obtain* their references via their mediator. The translator view, for instance, obtains a reference to the current translator model, stores it as a local variable, and also registers itself as a listener to the class keeping the translator model, i.e. the SummaryCreatorModel class, so that it can be told when the translator model has changed and thus when it needs to change its reference. In order to further enhance reusability of the translator view, the SummaryCreatorModel class is seen as being a 'keeper of a translator model' instead of 'the summary creator model' – this is done by letting the SummaryCreatorModel class implement a TranslatorModelKeeper interface, which can be seen in figure 4.13. Whenever the translator model keeper fires an event indicating that the kept translator model has changed, the translator view updates its appearance based on the changed translator model.

Since the translator view visualizes the translator model, and the template view visualizes the template model, the views contain direct references to their respective models (or a null reference if they are non-existant). I made the decision to view the respective models as facade controllers and thus being the central point of access onto the domain layer in regard to template and translator matters, respectively. In order to increase application performance and lower the representational gap between the view and the model, the choice was made to structure the various term views in such a way that there is a different type of view for each different type of translation model possible. Note, however, that there is heavy use of inheritance and template- and factory methods [3] in the view class structure, so the views share a lot of common functionality in their superclasses. Since modifying the translation model via the facade controller (the translator model) would require repetitive and unnecessary lookups of which translation model to use, which in turn would lower application

performance, The concept of using the facade controller for all downward communication was modified in those aspects where doing so would result in performance degradation. In practice this means that each term view has a direct reference to the translation model it represents, and it queries and modifies this translation model directly instead of going via the TranslatorModel class (the facade controller). When doing it this way, there is a slight loss of control, since there might be cases where the translator model needs to be notified if any of the kept translation models are modified. To make up for the loss of control, the translator model listens to the translation models for changes, and reacts when necessary (an example of this is when the translation model notifies its listeners of content change).

4.4 MedSummary

The MedSummary application is used for selecting interesting patients from the knowledge base and for viewing examination summaries and journals for the selected patients. The user can also view the digital images taken during the examinations. In the sections that follow, the design of the application's user interface and functionality will be described, followed by a description of the object-oriented design of the application.

4.4.1 Functionality and User Interface

As described in sections 3.3 and 3.4, there were some problems with the user interface and functionality in the previous MedSummary application, more precisely:

1. It was not possibly to print the generated journal contained in a page template directly from the application, the user had to copy the generated text into a page template contained in another third-party application such as Microsoft Word.
2. In order to switch the components used for generation (the template and translator files), the user had to open the settings dialog - this takes too long time if swift changes are required. An example of when such a swift change would be necessary could be when the standard template and translator pair needs to be switched to a specialized patient template and translator pair, in order for the clinician to generate and provide the patient with a summary of the examination at the end of a visit.
3. Attachments of dates and patient identifiers onto the digital photo thumbnails were hard to see when the colors of the image matched the colors of the text. Furthermore, the large-scale version of the images contained only the filename of the image, information about the associated patient and examination date would be more informative,
4. The manner in which the selected patients and their associated examinations were grouped and displayed could be improved, some means for collapsing and expanding interesting patient examination sets would be desired when the selected set of patients grows.

Figure 4.15 displays how the newly developed MedSummary application looks when running on a Windows platform (i.e. with a Windows look-and-feel). As can be seen from the figure, all of the above mentioned problems have been addressed in the new version. The user can now select – and easily switch between - page templates that surround the generated text. Also, the user can easily switch the template, translator, sections of interest, and data location in use by choosing from combo boxes in the toolbar. The combo boxes to display on the toolbar is configurable by the user - this could depend, for instance, on the user's available screen dimension. Furthermore, image thumbnails contain a transparent shadowed region overlaid on the lower part of the thumbnail where the descriptive text is placed in white, thus ensuring that the descriptive text can be read no matter what colors are used in the thumbnail.

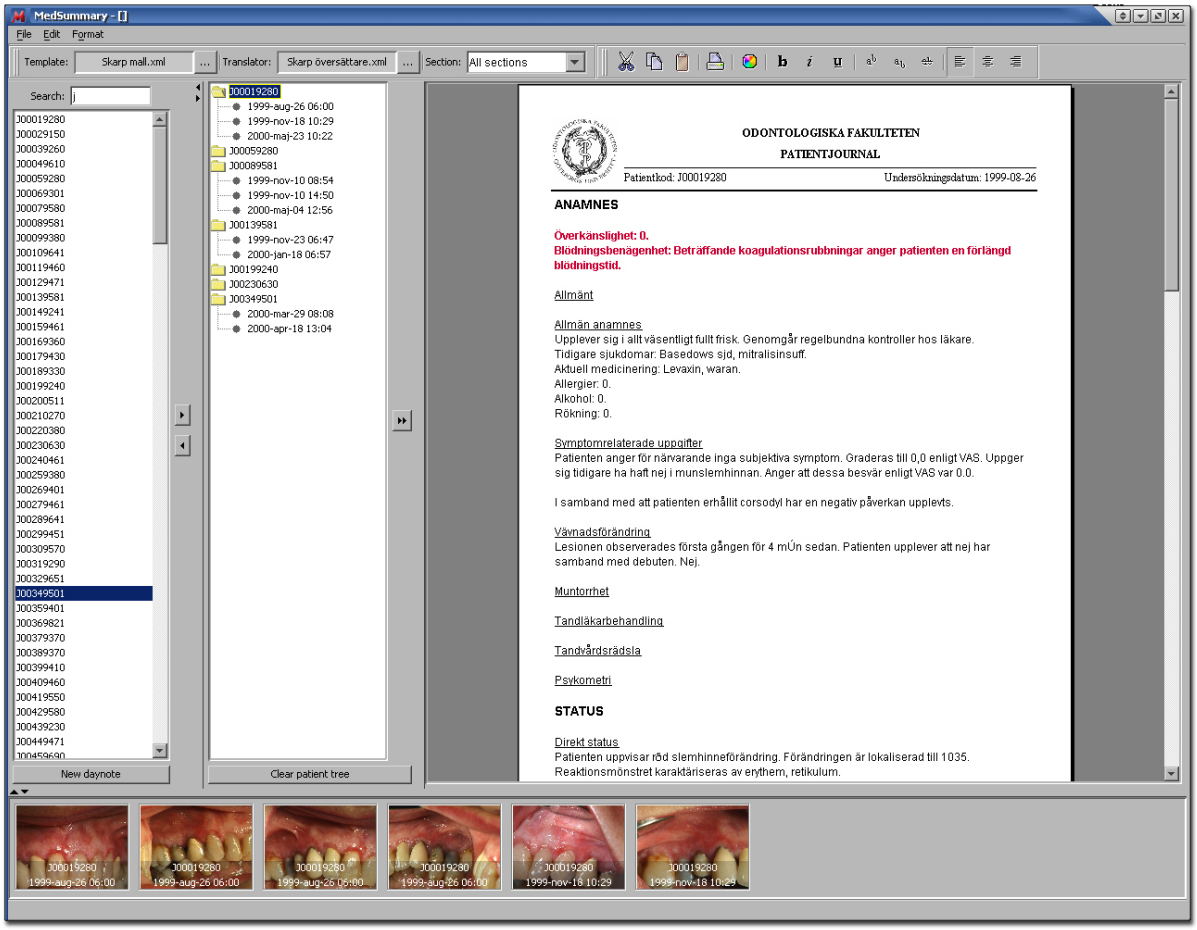


Figure 4.15 – The MedSummary application running on a Windows machine

As can also be seen from figure 4.15, the grouping of the selected patients with their associated examinations is done by using a tree structure, providing means to easily collapse and expand interesting patients in order to hide unnecessary information (this feature becomes especially attractive when a large set of patients has been chosen). Selecting a patient node in the tree automatically selects all the patient's examination nodes, while selecting an examination node only selects that node. Each selected examination node in the tree displays the associated images in the bottom part of the application, thus a simple way of viewing all images taken for a patient (perhaps during different examinations) is to simply select the patient's corresponding node in the tree.

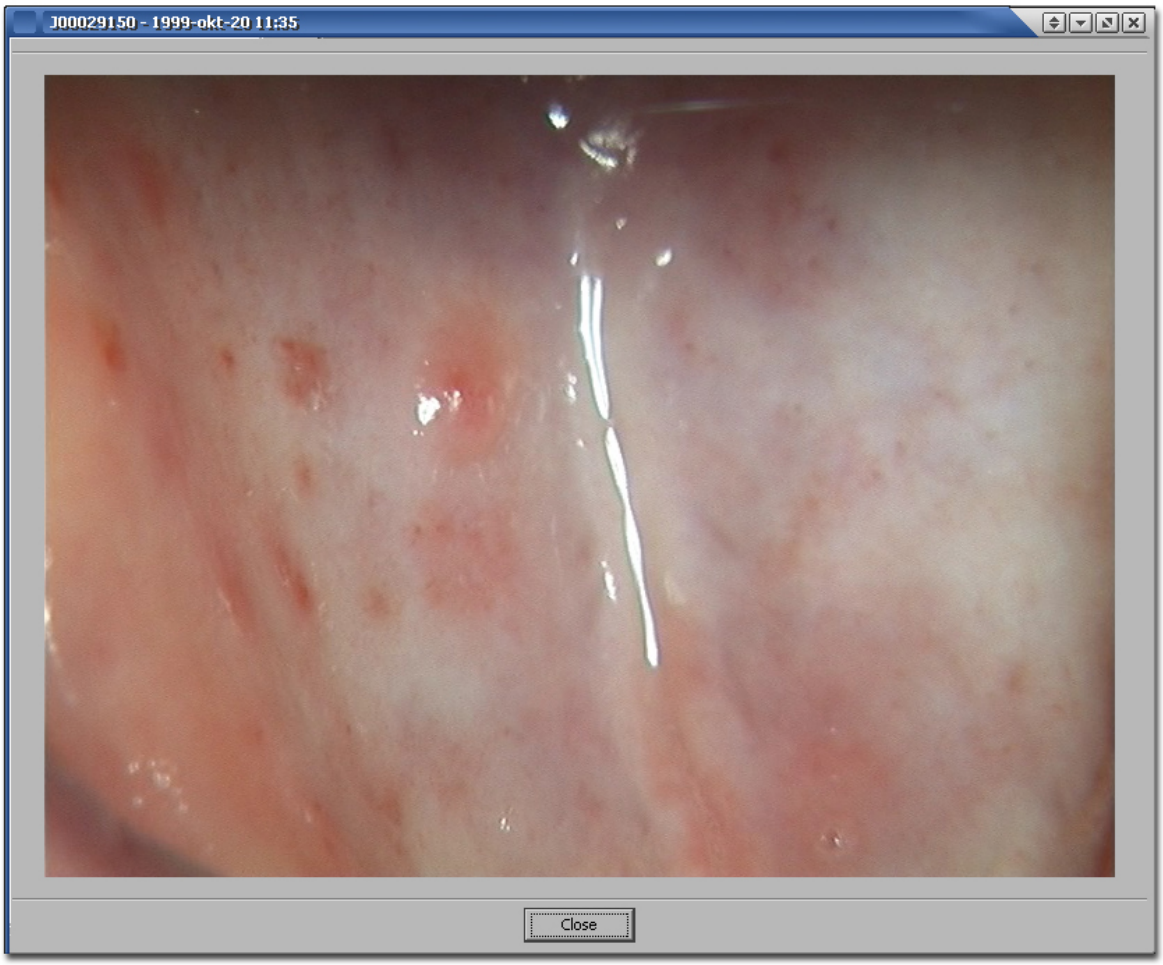


Figure 4.16 – Detailed view of an image taken during an examination

Figure 4.16 shows a frame containing a detailed view of an image, note that information about the associated patient and examination date is provided in the title bar of the dialog.

4.4.2 Software Design

The general structure of the MedSummary application is very much like the structure of the SummaryCreator application described above – namely, it is based on the Mediator, the Facade Controller, and the MVC design patterns and principles. The difference between the two applications' general structures is that the MedSummary application only uses two facade controllers as compared to the three facade controllers used in the SummaryCreator application. The difference in the amount of facade controllers reflect the difference in complexity between the two applications' respective domain models - a more complex domain model usually results in more facade controllers being used since using a smaller amount would result in the facade controllers becoming bloated. The conceptual structure of the MedSummary application is shown in figure 4.17.

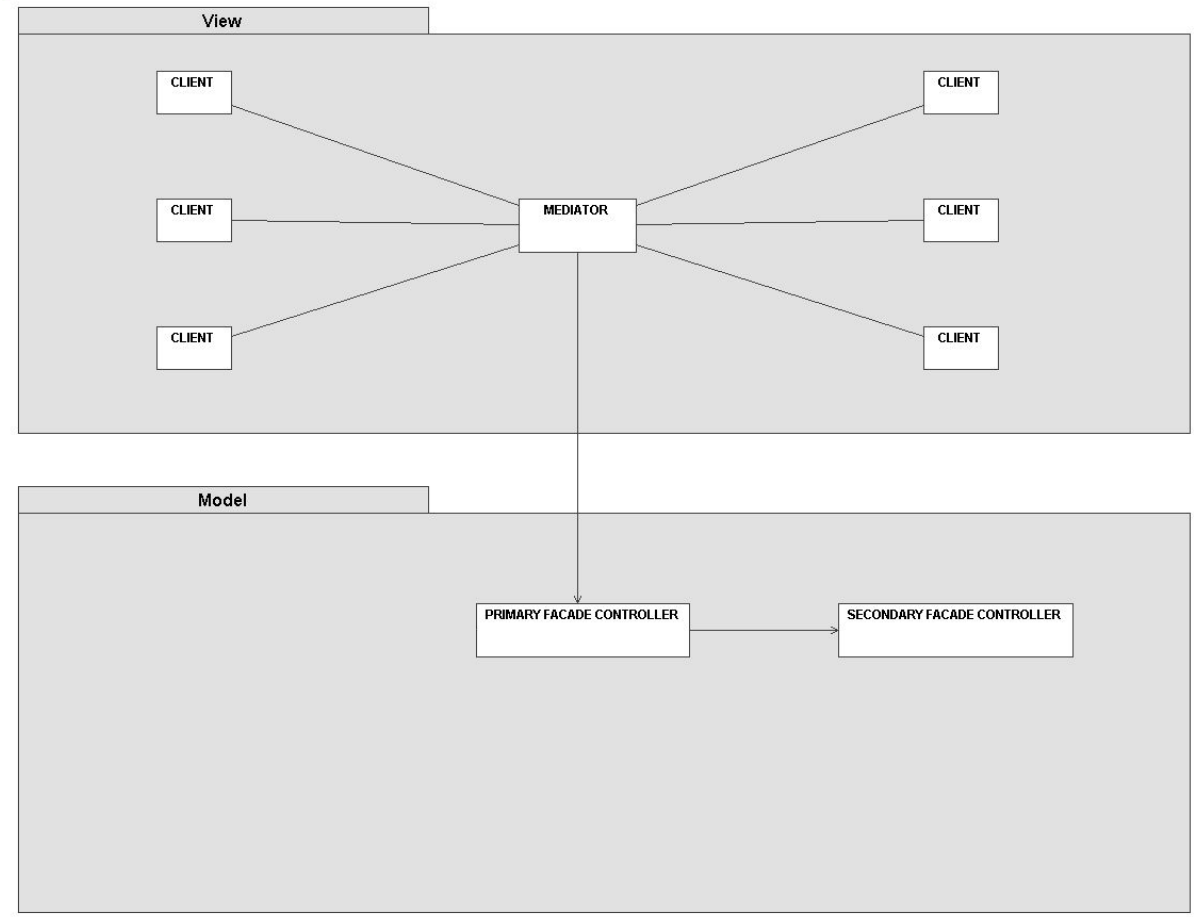


Figure 4.17 – The MedSummary application's abstract structure

As in the SummaryCreator application, the clients ask the mediator whenever they want to send information out, and the mediator sends information to the clients whenever they need to be updated based on changes in the domain model or the other clients. In other words, the mediator acts as the 'spider in the web', controlling access and containing references to all the clients. Like the SummaryCreator application, the domain objects have no direct knowledge of the objects in the presentation layer (view), except as objects implementing the various listener interfaces contained in the domain layer. The actual structure of the MedSummary application, displayed as an UML diagram, is shown in figure 4.18.

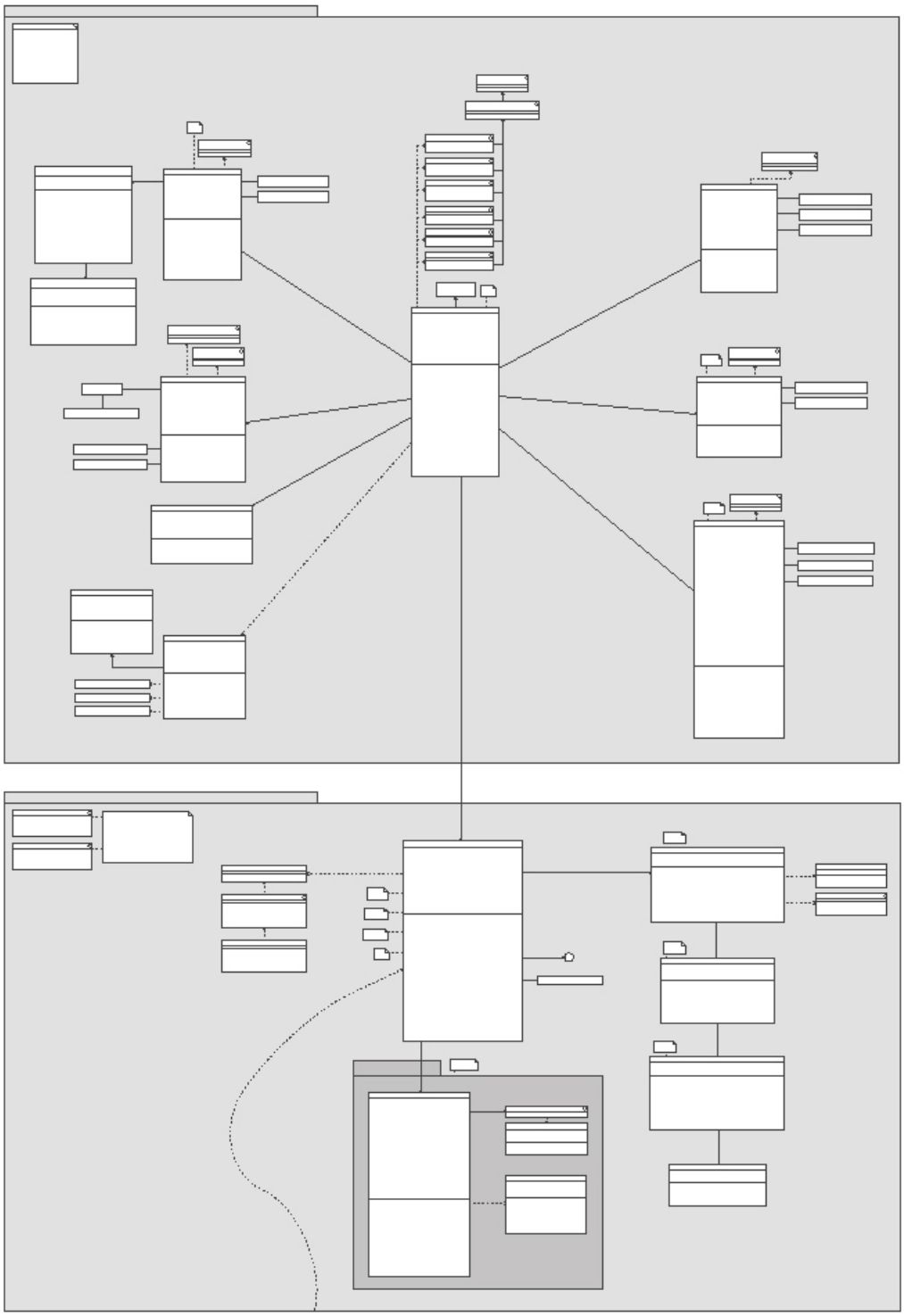


Figure 4.18 – The MedSummary application structure

When comparing figures 4.17 and 4.18, you see that the general structure is the same, and that the central object in the view layer acts as a ‘spider’ class containing references to all clients as well as the primary facade controller, which is the only entry point into the domain layer and thus controls domain layer access. The primary facade controller fulfills the non-complex responsibilities that do not delegate well to the secondary facade controller – which in this case is the tree model containing the patients with their associated examinations and images. If more functionality is added to the MedSummary application in the future, more secondary

facade controllers might be added, fulfilling responsibilities related to the added functionality concepts.

4.4.3 Model (Domain)

There are two major parts making up the object structure of the MedSummary domain layer. On one hand we have the tree model, dealing with everything related to the tree of selected patients and their associated examinations and images as well as fulfilling the role of being a secondary facade controller as described above. On the other hand we have all other 'smaller' responsibilities which are fulfilled by the primary facade controller class (the MedSummaryModel class). The model layer of the MedSummary application is shown in figure 4.19.

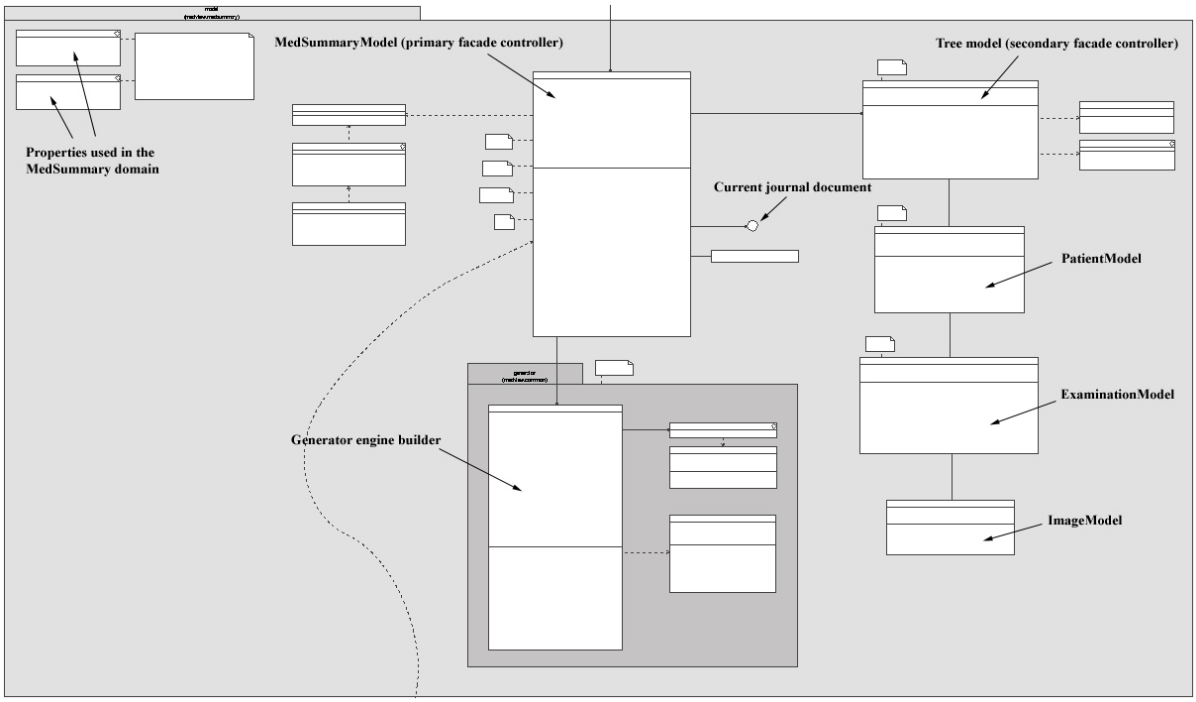


Figure 4.19 – The model layer of the MedSummary application

The tree model class can be seen as an Information Expert [1] in matters dealing with the tree of patients and associated examinations and images taken during the examinations, while the MedSummaryModel class can be seen as an information expert in matters dealing with the *containment* of the tree model as well as in everything regarding all other domain matters. When a patient is added to the tree model, the tree model creates a patient model and adds it to the tree model's set of patients. The tree model, containing the patient models, is chosen as the Creator (another GRASP pattern) of patient models. The patient model, in turn, is chosen as the Creator of examination models – which in turn are Creators of image models. By placing creation responsibility in this fashion, we follow the Creator GRASP pattern, which states that a class should be responsible of creating instances of another class if it aggregates, contains, records instances of, closely uses, or has the initializing data for the other class [1]. High cohesion (where cohesion is a measure of relatedness between the various responsibilities taken on by a certain class, see glossary) and low coupling to other classes is maintained by the tree model class, since it only deals with matters pertaining to the patient tree structure and does not know anything about the surrounding objects. The main model class has acceptable medium cohesion since it deals with the various 'smaller' domain issues

– the cohesion is acceptable for two reasons: 1) there are not too many such issues, and 2) they are all rather simple.

In the upper left corner of the model UML diagram you can see the interfaces defining the properties used in the MedSummary domain. Since the domain layer resides in a layer above the technical services layer, as shown in figure 4.4, it has access to the utilities provided by the lower-level foundation layer, such as storing and retrieving properties to and from permanent storage, respectively. Note that properties used both in the view and model package are placed in these interfaces - the properties can be seen as pertaining to the 'MedSummary application domain', which includes properties used by the various view classes.

As mentioned above, the MedSummaryModel class acts as the primary facade controller and thus as the single point of entry into the domain of the MedSummary application. It is also the initial domain object instantiated when the application starts. If only one facade controller were to be used, the class would become bloated with too many methods, which is why tree matters were placed in a separate controller class (the tree model). The sequence diagram in figure 4.20 displays the collaboration between the domain objects when adding a patient.

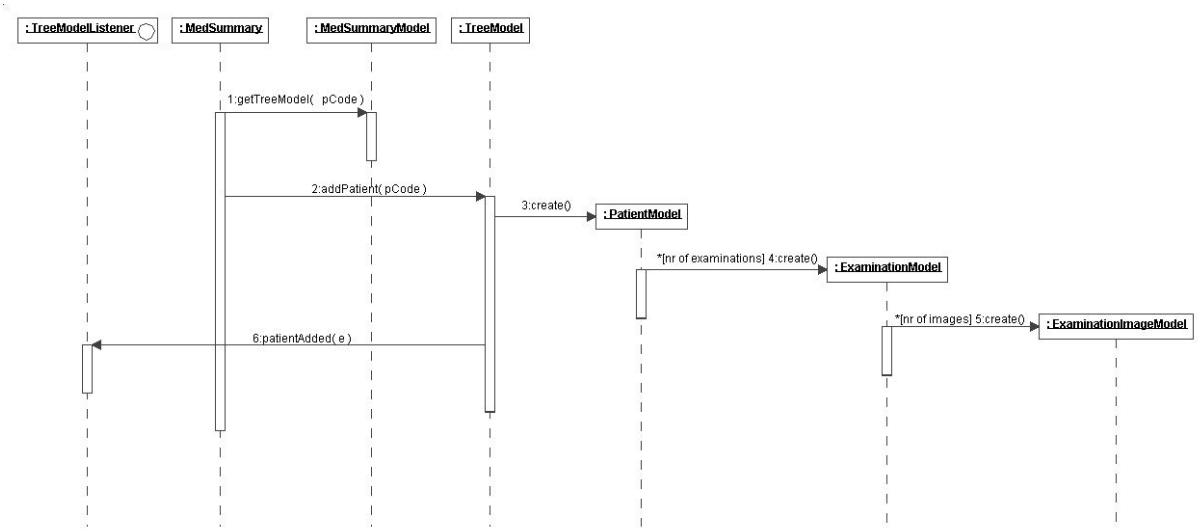


Figure 4.20 – Sequence diagram showing how a patient is added in the MedSummary domain layer

4.4.4 View (Presentation)

The view layer of the MedSummary application is divided into seven major parts: 1) the central MedSummary class, which acts as the mediator and the 'glue' that references all other major view classes, 2) the MedSummaryMenuHandler class, which takes care of all things related to the menus in the application, 3) the MedSummaryToolbarHandler class, which deals with the toolbars, 4) the MedSummaryPatientPanel class, which is responsible for listing all patients as contained by the central model class, 5) the MedSummaryTreePanel class, dealing with displaying the tree of chosen patients along with their associated examinations and photos, 6) the MedSummarySummaryPanel class, being responsible for visualizing the document kept by the main model class representing the currently generated journal content, and finally 7) the MedSummarySettingsContentPanel class, which plugs into the dialog framework as one of the settings content panels to be displayed in the settings dialog. The central MedSummary class implements all of the client mediator interfaces, so

each client sees it in its own way. This further enhances reusability of the clients since they can be reused in another view by simply creating a new central mediator object implementing the client interfaces. The view layer of the MedSummary application is shown in figure 4.21.

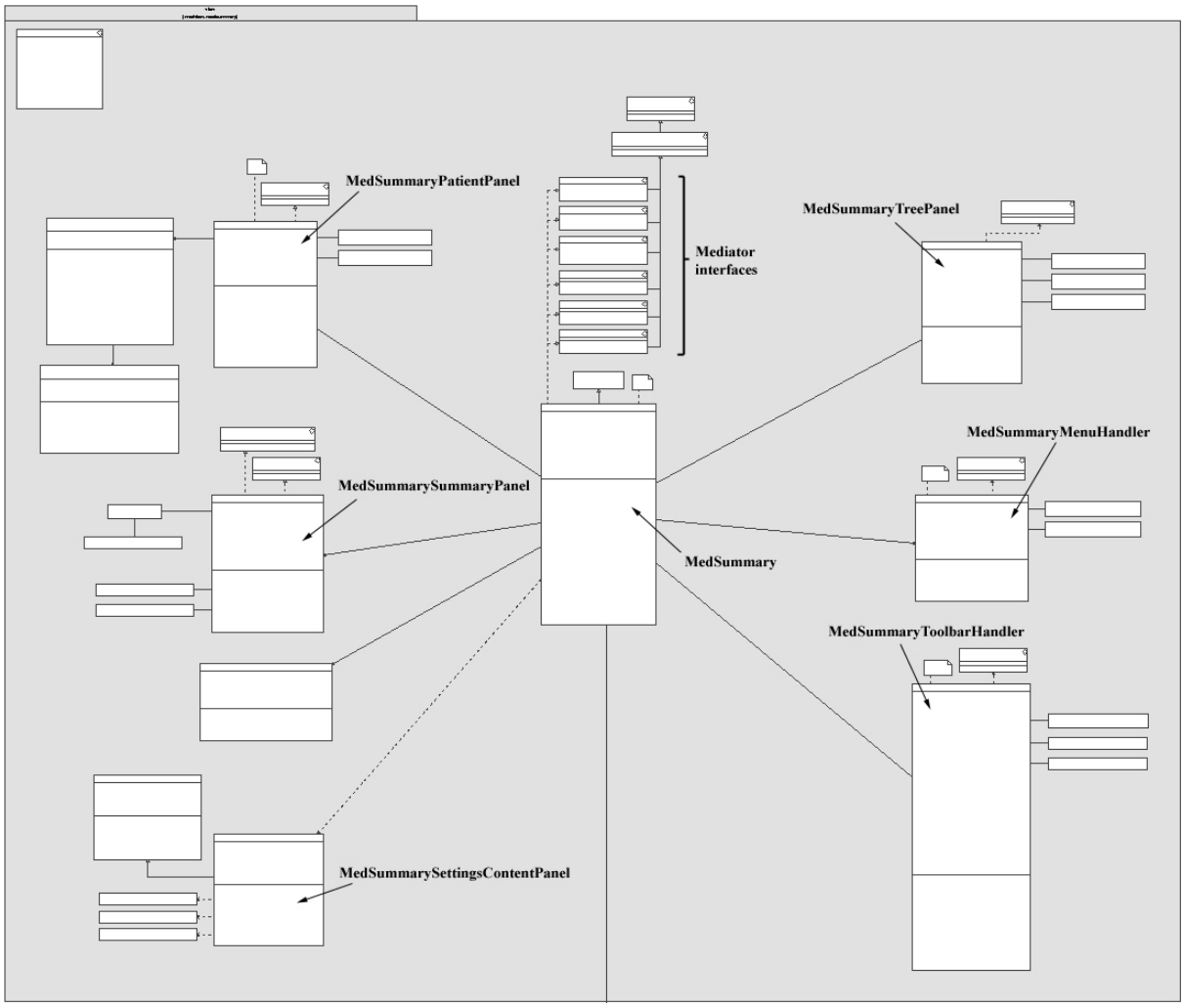


Figure 4.21 – The view layer of the MedSummary application

As in the SummaryCreator application's view layer, there is only one object containing a reference down to the domain layer, namely the Mediator class (MedSummary). All system events not dealing with the tree structure of patients and examinations are sent to the primary facade controller (MedSummaryModel). The events dealing with the tree structure are sent directly to the tree model via a reference obtained from the primary facade controller when constructing the tree panel. The fact that the tree panel contains a direct reference to the tree model infers a slight loss of control in the domain layer – the main model class (the primary facade controller) does not know if system events occur that change the tree model and can not coordinate such activities with possible other domain objects. In order to make up for this loss of control, the main model class listens to the tree model for changes, and the tree model notifies whenever some change occurs that updates the state of the tree model.

In order for the menu and toolbar handlers to retrieve the actions in use by the application, it is necessary to initialize and construct the various panels and handlers in a certain order at application startup. First, the panels are constructed, which will result in each panel registring

it's contained actions with the mediator class. The registration of actions is done by providing a constant identifier (defined in an interface visible to all classes in the view layer) along with the action to the mediator, which places the action in a hashmap keyed by it's unique identifier. After all panels have been constructed (along with all actions contained in them), the handler classes are constructed. The handler classes, in turn, obtain the actions previously registered by the panels by asking the mediator to return the actions as identified by the constants in the common action interface.

4.5 Utility Packages

There are two main utility package structures - the misc and the medview.common packages and their subpackages. The misc (short for miscellaneous) package and it's subpackages contain various components that can be used in any program, i.e. not just for use in the MedView context. The medview.common package and it's subpackages contain components that can be used in several MedView applications - these components are tightly coupled to each other and rather specific to the MedView context, thus it may be hard to find use for them in other situations. There is also a package called 'datahandling' along with several subpackages – this packages should actually be placed in the medview.common structure (i.e. the datahandling package should be called medview.common.datahandling), but for historical reasons this is not the case.

4.5.1 The datahandling Package

The datahandling package and subsystem places itself in the 'technical services' layer of the architectural Layers [2] design pattern (see figure 4.1). The reason for the naming of the package is historical – in the early development of the system the intent of this layer was to handle the possible ways to store examination data as well as to provide ways to obtain media resources located on permanent storage. In time, it expanded to include language handling, term and value processing, template handling, translator handling, and parsing of patient identifiers (also known as *pcodes*, see glossary). The major design patterns in use in the datahandling package structure are the Facade GoF pattern [4], the Factory GoF pattern, the Strategy GoF pattern, and the Singleton GoF pattern.

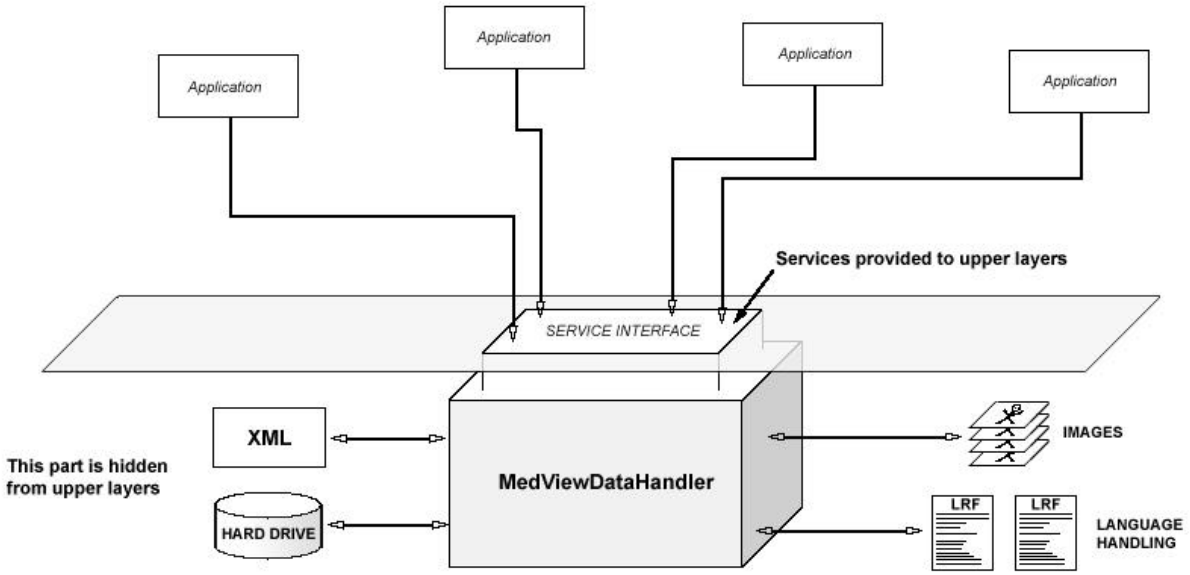


Figure 4.22 – Concept of providing application-independent lower-level services

The general structure of the datahandling package in UML can be seen in figure 4.23, while the concept of the data handler providing general services to upper layers is visualized in figure 4.22.

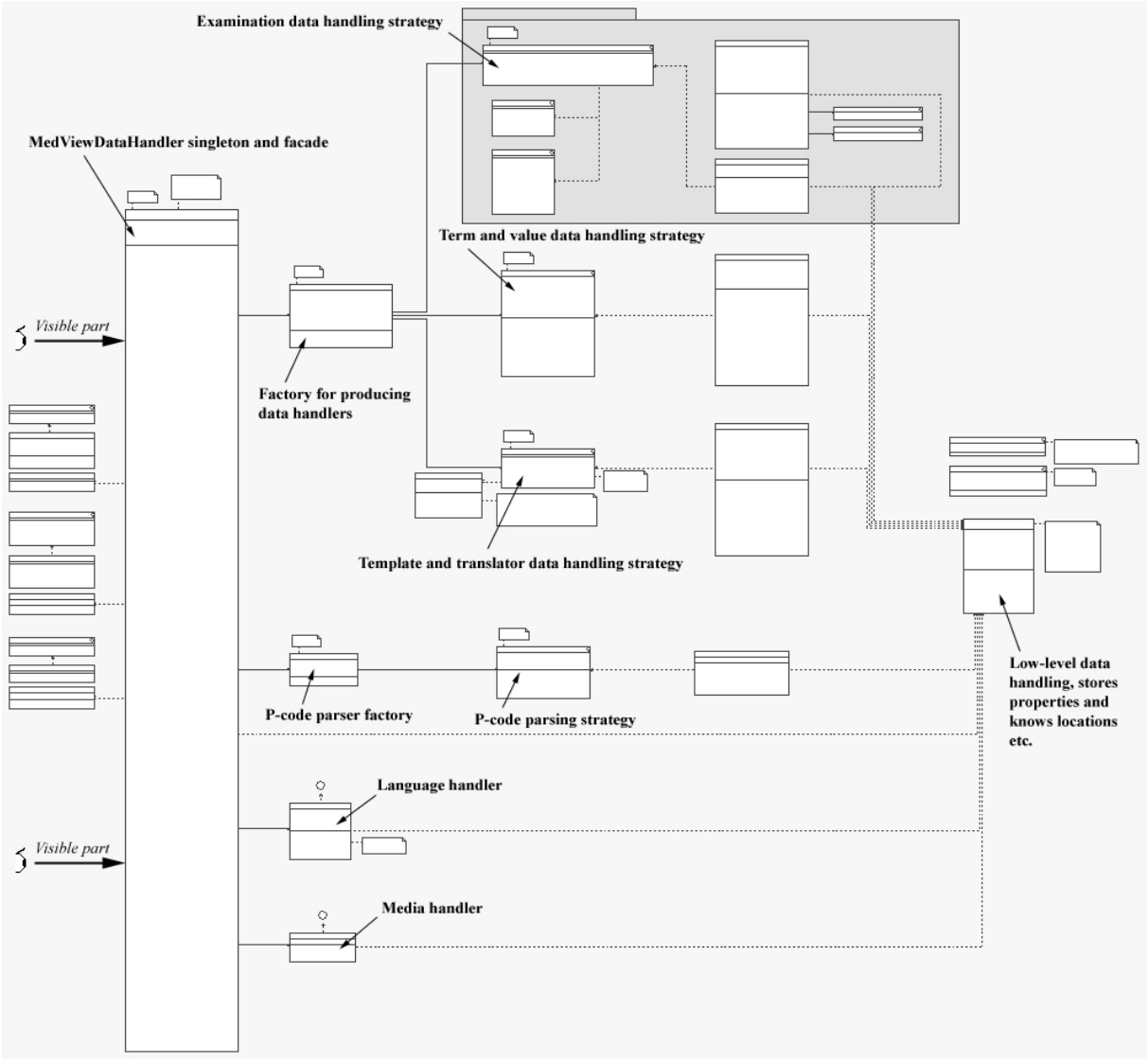


Figure 4.23 – The general structure of the datahandling package

The MedViewDataHandler class is probably the most widely used class in the entire system, and it is therefore imperative that it is designed in a sound way. The MedViewDataHandler class is a singleton-accessed facade, and is the single point of access for outside packages. It hides all other classes in the datahandling package (except the various public interfaces and listener-related utility classes), which makes it possible to change the structure and implementation without affecting the rest of the system as long as the service interface remains the same. As mentioned before, since the datahandling package is placed in the technical services / foundation layer, it's purpose is to provide rather general services to the upper layers.

The services provided by the data handler include: 1) being able to store and retrieve property values, and listen for changes in them by registering property listeners to the data handler, 2) being able to retrieve strings that should vary by language (for instance, a cancel button might

display the text “Cancel” if the current language is english, but “Avbryt” if the current language is swedish), and listening for language changes, 3) being able to retrieve media resources (image icons, images, sounds etc.) by simply specifing a constant defining the resource, 4) being able to store terms and values for terms as well as retrieving term type information, 5) being able to parse patient identifiers for gender, year of birth, patient age, examination date etc. and to verify that a certain identifier is valid, 6) being able to store and retrieve templates and translations.

At the time of this writing, the examinations at kliniken för Oral Medicin at Odontologen in Gothenburg are stored in a so-called *tree-file* format, therefore the examination data handler currently in use deals with tree files. In the future, the format may change – for instance, it has been discussed many times if it wouldn’t be more effective with an SQL database instead. Since this is a point in the system with high probability of future change and/or evolution, it is important not to design surrounding objects so that they are highly coupled to the current particular way of dealing with examination data. In the words of Larman [1], when describing the Protected Variations GRASP design pattern, we need to “identify points of predicted variation or instability and assign responsibilities to create a stable interface around them”. In order to accomplish this, I decided to use a combination of the Strategy GoF design pattern [4] (which can also be seen as a special case of the Polymorphism GRASP pattern [1]) and the Factory GoF design pattern. A strategy for what it means to be an examination data handler is defined by specifying this in an interface, and letting the concrete implementations of this interface provide the specified functionality in their own way (such as a tree file handler when dealing with examinations in the tree file format). Deciding which implementation class to use is based on an external property value (which can be set from within the applications), which is queried by a Factory object whenever various examination data needs to be obtained. The following excerpt from the DataHandlerFactory class illustrates this:

```
public ExaminationDataHandler getExaminationDataHandler( )
{
      ...

      String setEDHClass = mVDSH.getProperty(CURRENT_EDH_CLASS_PROPERTY);

      examinationDataHandler = Class.forName(setEDHClass).newInstance();

      return examinationDataHandler;

      ...
}
```

The same reasoning that applied to the examination data handler also applies to term-, template-, and translator datahandlers, so they are dealt with in the same way (see figure 4.23). The parsing of patient identifiers does not deal with storing and retrieving data, but the reasoning for obtaining a specific pcode-parser (an object dealing with the parsing of patient identifiers) is the same as for obtaining a specific datahandler, therefore a separate factory for producing pcode-parsers is used (see figure 4.23).

4.5.2 The common Package

As discussed above, the idea behind the common package is to have a place to store medview-related functionality and components that can be common to more than one medview application. Thus, the components are tightly coupled to the medview context, but they are not coupled to any specific medview application. The structure of the common package at the time of this writing is shown in figure 4.24.

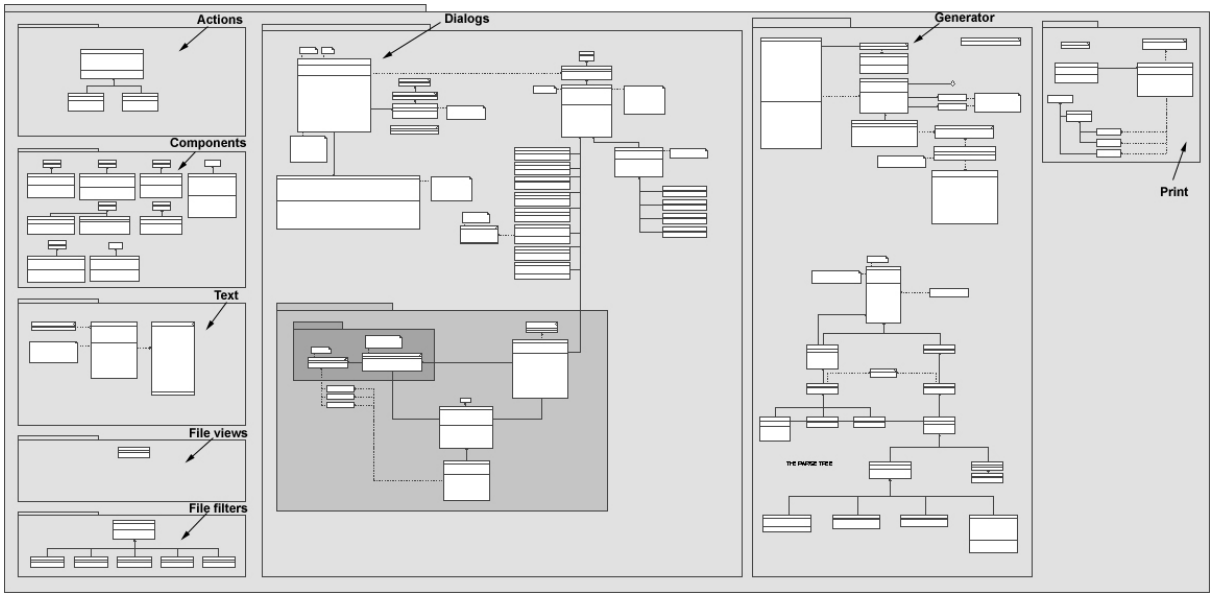


Figure 4.24 – The structure of the common package

As can be seen from figure 4.24, the various parts of the common package (as the time of writing) are: 1) actions, 2) components, 3) text functionality, 4) file views, 5) file filters, 6) dialogs (a major part), 7) journal generator (another major part), and finally 8) print functionality. As can also be seen from the figure, the major parts of the common package as of date are the dialog and the generator handling.

4.5.3 The dialogs Subpackage

The dialog subsystem is accessed through a singleton-accessed facade, namely through an instance of the MedViewDialogs class. It provides the various applications with methods for displaying various dialogs in use in the medview context, such as displaying a ‘load template’ dialog and for displaying an ‘add value to term’ dialog. The dialog subpackage is displayed in more detail in figure 4.25.

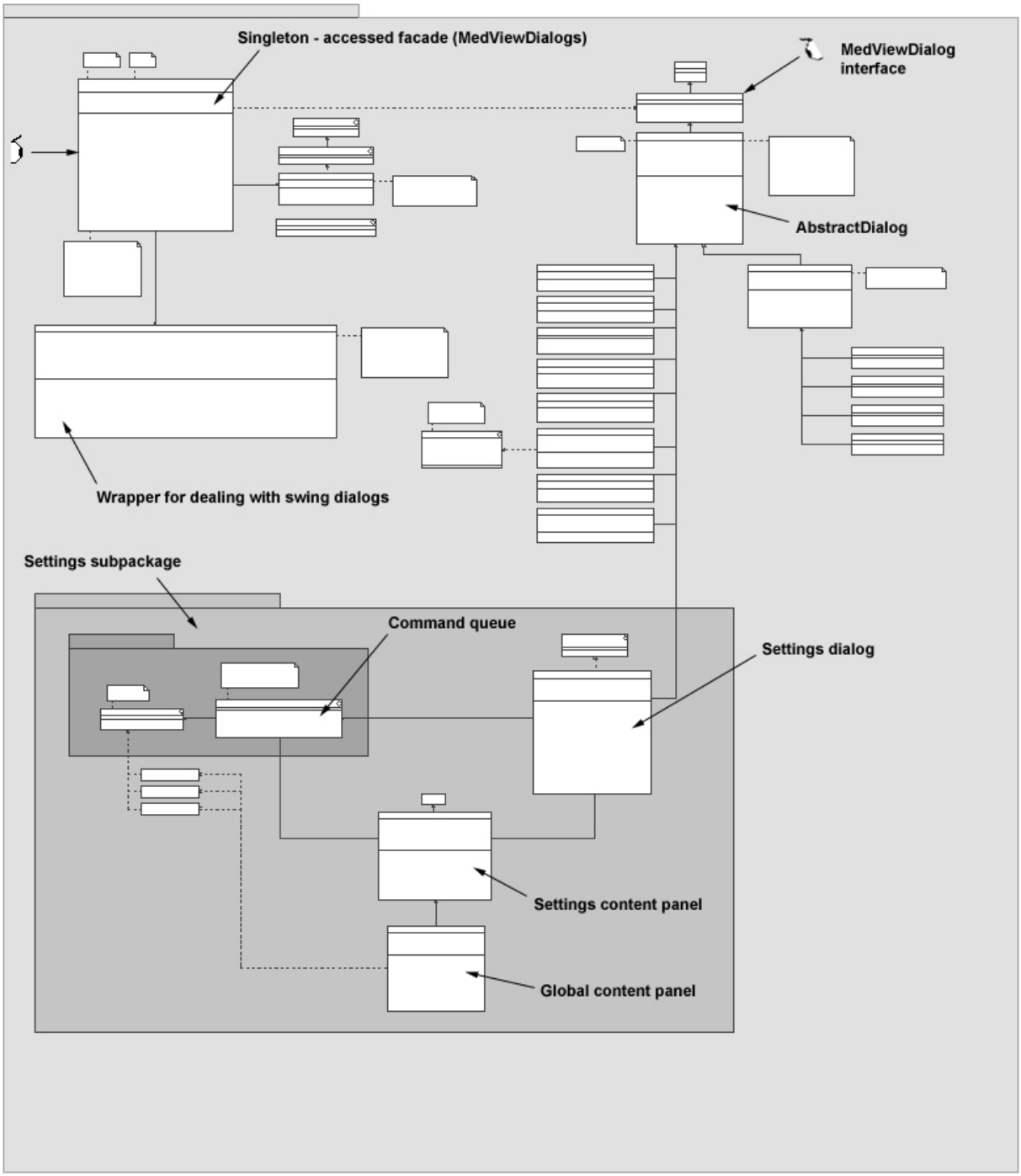


Figure 4.25 – The common dialog subsystem

The various file chooser dialogs, as well as the color chooser dialog, are obtained from the Java Swing framework - a part of the Java 2 SDK (Software Development Kit) that deals with user interface matters and graphics. The dialogs that use the Swing framework are handled by the SwingDialogWrapper class, which is delegated to from the MedViewDialogs facade whenever such a dialog is needed. Some of the other methods in the MedViewDialogs facade return objects implementing the MedViewDialog interface, usually the methods that require the user to enter some non-trivial information (like the ‘add section’ dialog, which requires the user to enter the name of the new section as well as being before or after a specified section). This interface provides methods to extract the information from the dialog, and the

methods in the facade provide documentation for how to deal with the return data for each specific method. As seen in the structure, all actual dialog implementations subclass the AbstractDialog class, which provides a framework for constructing dialogs in the medview context. Among other things, it defines the dialogs visual structure, and provides factory methods [3] for the subclasses to implement. In order to provide this framework, there is heavy use of template methods [3], such that the subclasses only need to 'fill in the gaps' (also called *hot spots*) to create a fully functional, rather complex dialog with their specific declination. Basically, what is required of the subclasses is to specify the button faces they want, the title of the dialog, the button index reflecting the default button, the content panel, and (optionally) the button listeners. The framework provides a lot of default functionality for all mentioned above that can be used or overridden in the subclasses, it also takes care of positioning the content panel and the buttons as well as margins and dividers etc. so that all dialogs get a consistent look. In practice, this results in a developer being able to produce a new rather complex dialog in very little time. Some example dialogs are displayed in figure 4.26.

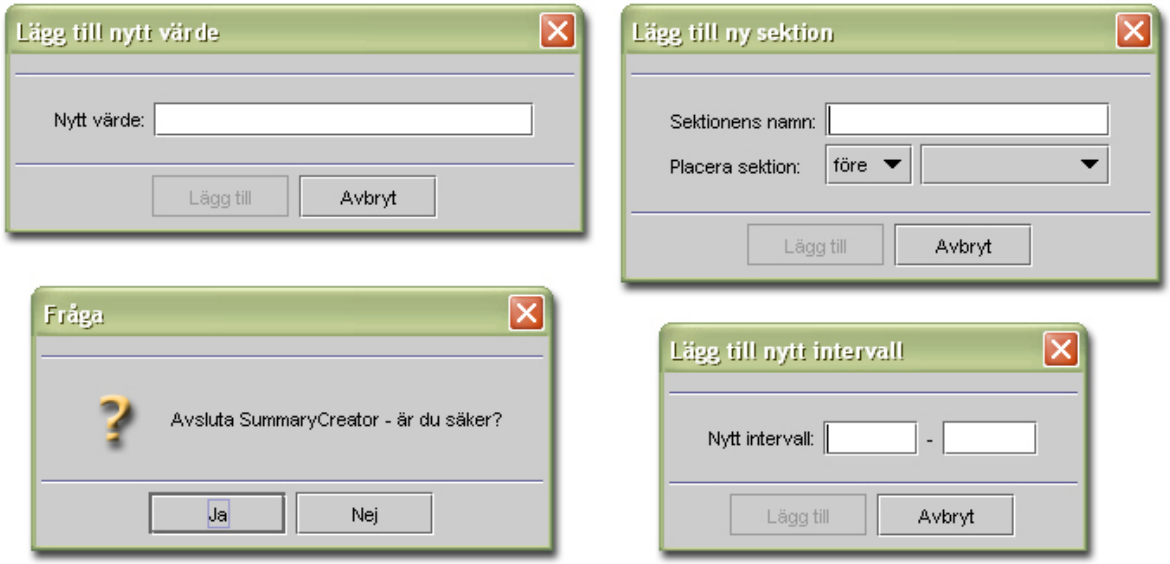


Figure 4.26 – Some dialogs produced by the dialog framework

For the design of the settings dialog, it is necessary for each application to be able to 'plug in' its own specific settings content panel since it is highly coupled to the application in question. In order to maintain the application independency of the common package, it becomes necessary to create a structure where each application can send its settings content panel to the dialog framework, followed by the framework attaching it to the settings dialog displayed when the user calls for the settings dialog via the facade. Thus, the facade contains methods for attaching content panels to the settings dialog, these should be subclasses of the SettingsContentPanel abstract class in order for the framework to recognize them as settings content panels. When changes are made in the settings content panels, these are not effectuated at once, instead they are placed on a *command queue* which places each setting change on a queue as a *command*. All commands on the command queue are effectuated when the user presses the 'apply' button. If the user cancels the settings dialog, the queue is cleared and the changes are discarded. This approach is based on the Command GoF [4] design pattern. The various settings content panels that are to be attached to the settings dialog

need access to the command queue in order to place their specific commands on the queue when the user makes changes, the queue can therefore be obtained via the facade object.

4.5.4 The generator Subpackage

All things related to the journal generation are located in the medview.common.generator package. Some of the design patterns used in the journal generator package are the Builder GoF pattern [4], the Strategy GoF pattern, and the Composite GoF pattern. As described above, you use a builder object for constructing the generator engine – this builder object is an instance of the GeneratorEngineBuilder class, which thus provides ways of attaching the necessary components for journal generation onto the generator engine. The actual generator returned from the builder is a subclass of the GeneratorEngine abstract class, which contains general functionality needed in all kinds of generator engines as well as methods for setting and obtaining the various necessary parts. Figure 4.27 provides an overview of the medview.common.generator package.

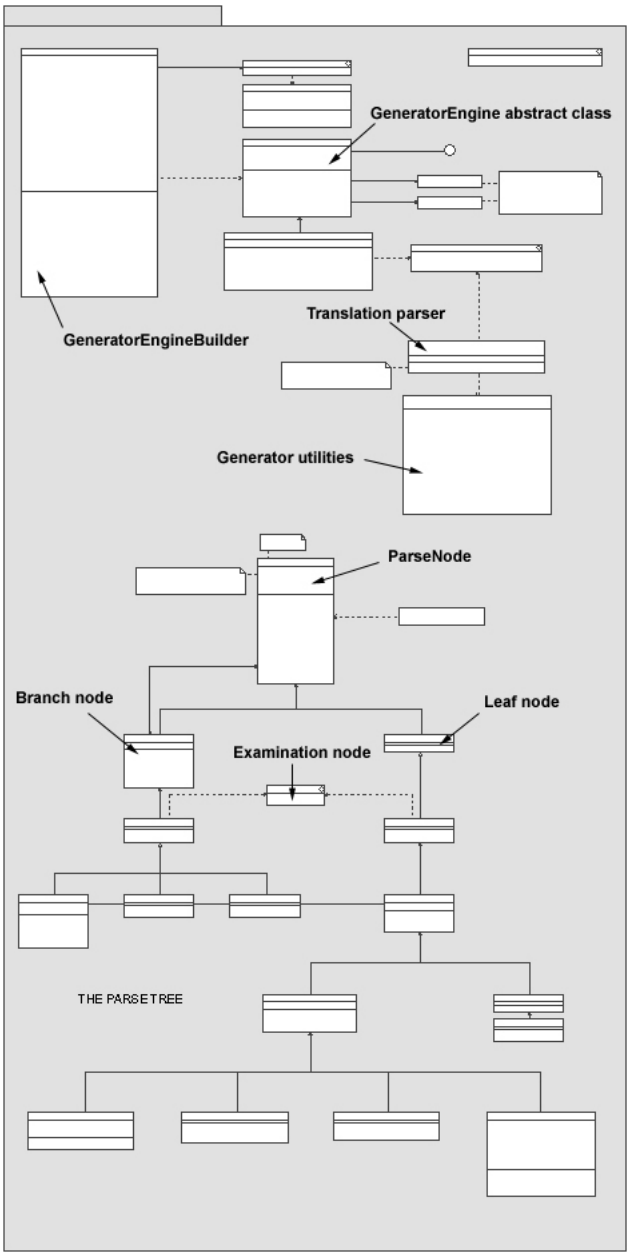


Figure 4.27 – The common generator subpackage

All nodes under the examination root node are instances of the ExaminationNode interface, which contains methods for obtaining the examination date and the patient identifier. The only node type that can be attached as a child to an examination root node is a section node, representing one of the sections that the user has chosen to include in the generated journal. The child nodes of the section node, in turn, are line nodes representing lines in the text flow. The line nodes may or may not contain term nodes as child nodes - if they do not, the line node represents a line of text that should be left ‘as is’, at least if it is not later deduced that the section containing the line should be removed from the generated journal because of lack of term content for the specific examination.

5 Conclusions and Future Development

During the first iteration of my thesis, I did not have much knowledge in object-oriented methods such as UML, GRASP patterns, reverse engineering, object-oriented system architecture (Layers), and design patterns (even though I knew a few basic ones, such as Observer and the MVC design principle) etc. As a consequence, the system became very difficult to grasp and overview at the end of the iteration, which led to a general belief that the system 'was not ready for deployment' - future maintenance and bugfixes would become very difficult to manage. By introducing UML design diagramming methods to visualize the system structure, as well as using a CASE tool to reverse engineer developed code into the diagrams, confidence increased that the system could actually be deployed and that possible (actually, most certain) bugs would be relatively easy to fix. It has been shown that, in general, the initial cost of developing a system is small when considering the total cost of the system during its entire lifetime – the major part of the total cost of the system lies in maintenance and dealing with issues occuring after the initial release. Thus, it is very important that a system is well-documented (in my case with the UML diagrams and this report) so that future developers may pick up where the previous developers left off.

It is very important to use established design patterns and diagrams when constructing a medium-to-large-scale system. If established design methods are not used, an overview of the system becomes increasingly difficult as the system grows, and future maintenance and extendability will suffer. Furthermore, by using design patterns and UML diagrams, you become much more adept in thinking in 'objects' and how the objects should communicate and hide information from each other in order to improve the overall system structure.

The practice of developing in an iterative fashion and to perform a thorough initial analysis and requirements gathering before starting to design and implement is very sound. More specifically, you should not try to 'implement all at once', but instead develop the most critical aspects initially and leave other aspects for future iterations. In order to know what parts are the 'critical' ones, a thorough analysis is necessary before initiating development. It has been difficult at times to know what matters were considered to be the most important to the users - this would have been easier if more analysis and user interaction had taken place initially. Especially, user requirements and wishes should have been more documented and established before the development began – some documentation and analysis *was* performed, but *more* such activities should have been performed in order for me to have been more sure I was putting effort into the right matters.

During the entire course of my thesis, fellow developers within the MedView project have been working concurrently on other projects. Several aspects of my developed applications have shown to be reusable in the other projects as well (especially the lower-layer data handling and 'common' packages), which has led to conflicts when I have introduced changes in my packages and classes used by the others. Thus, I have learned the importance of designing objects using information-hiding and open-closed principles, as well as the importance of agreeing which parts of the system that are to be used concurrently by others. Since developing reusable and generic components - that are to be used by others - take careful planning as well as time, it is important to specify which parts are worth the effort.

There are many possibilities for future development surrounding the developed applications. The parts dealing with NLG can always be improved in order to produce more natural generated text, one thing to keep in mind though is to make sure that the system is not too

complicated to use, since the users should be able to produce their own text-generating environment without any expert knowledge in NLG. Another possible future project could be to create some type of 'page template editor', which users can utilize to develop their own page templates (with 'page templates' I mean the ones usually seen in word processors providing frames and logotypes surrounding the actual text). Currently, these page templates have to be developed by programmers and inserted as choices into the applications, so they are not especially user-customizable.

Another interesting future project could be to distribute the knowledge base, the templates, the translators, and the possible terms and term type definitions to the users via a central server using a client-server approach, where the server is accessed by various clients over the internet. In this scenario, clinicians, students, patients etc. are given accounts on the server, thus making it possible for them to login and access information (moderated by some administrator at the clinic) from anywhere as long as they have access to the Internet. The Java programming language is well-suited for such a system, which could be implemented by using the various Java network api's and technologies (like Java RMI). Of course, such a system is a major undertaking, requiring that the applications involved are well-designed and documented – one of the aims of my thesis has been to develop applications and a system structure that will work in such a distributed system. The modular and layered structure of the developed applications, as well as the fact that the Java programming language is used, make them well-suited for use in a distributed environment. The concept is illustrated in figure 5.1.

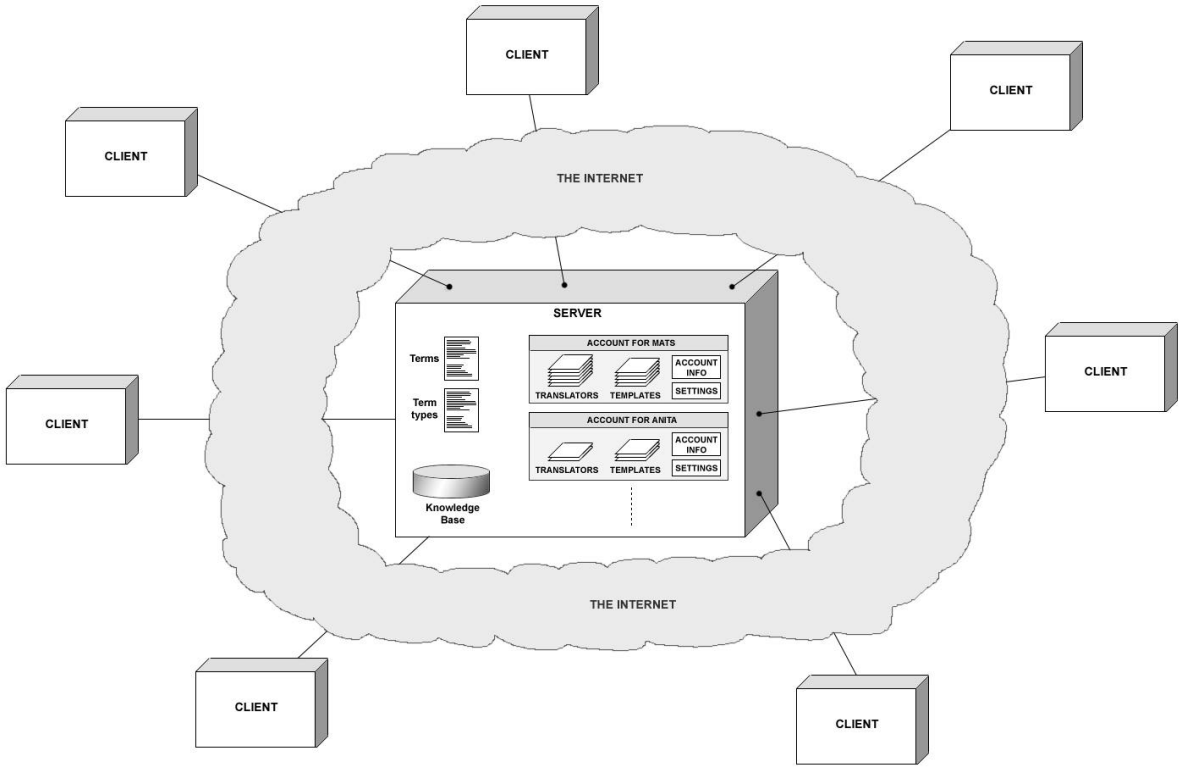


Figure 5.1 – A possible future scenario - a distributed client-server system architecture

The clients can be clinicians sitting in an examination room somewhere in the same clinic, or they could be dentists sitting in an examination room in New York – it doesn't matter as far as the client has access to the Internet. The system would deal with the client in the same manner no matter where the client is located physically. Furthermore, different accounts could have

access to different resources – for instance, students could have special student accounts, where teachers could place course-related cases, templates, and translators that the students could study. A clinician in Gothenburg could assign a guest account to a clinician in Italy and permit this account to have access to certain examinations and other information he wishes to share. Clinics all over the world could enter information into the knowledge base using applications like the ones described in this thesis, as long as their clinicians have registered accounts on the central server. Furthermore, the access to the knowledge base, the terms, and the term types (concepts used in the current system) would be wrapped and controlled by the server, making it easier to introduce various security measures and enforce integrity and confidentiality of the data.

6 User Documentation

The user documentation provides some general information about using the applications – for more detailed information you will have to read the application-specific documentation bundled with the applications. This section is divided into three parts – the first part describes global application matters, i.e. matters common to both the MedSummary and SummaryCreator applications, the second part describes the newly developed SummaryCreator application, and the third describes the improved and rewritten MedSummary application. The parts dealing with the MedSummary and SummaryCreator applications begin with a short introduction followed by an overview of the application's general GUI structure, along with some brief explanations on how the application is used.

6.1 Global Application Matters

If you are running the applications on a system using a lower screen resolution than 1280x1024, or there is some other reason for not wanting to view the template or generated journal in the default 'page layout' view style, you might want to change the view style of the template or journal to the simpler text-only style. This is done by switching the choice of view style in the 'View' menu as displayed in figure 6.1 (here seen in the SummaryCreator application context, the same menu is present in the MedSummary application as well). An example of how the SummaryCreator application can look when using the simpler text-only view style is shown in figure 6.3.

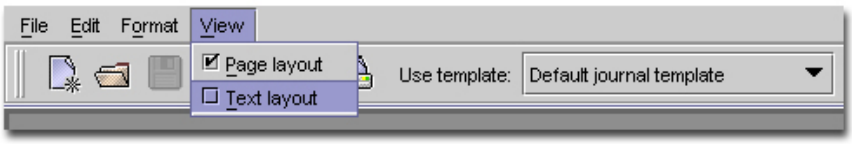


Figure 6.1 – Switching template view style

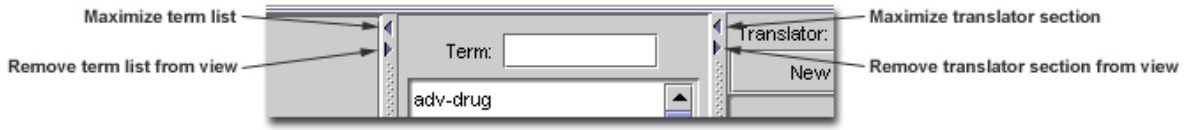


Figure 6.2 – The divider and the minimize and maximize arrows

Another way to manage the available screen area is by using the dividers to hide unnecessary information in order for other, more interesting, information to receive a larger portion of the display. The minimize and maximize divider arrows in the Metal look-and-feel (i.e. Java's standard look-and-feel - the platform-independent graphical user interface design that is the default for Java applications) are shown in figure 6.2, as they can be used in the SummaryCreator application context. For instance, say you only intend to work with the template and don't have a need for displaying the translator - you then minimize the translator section by clicking on the 'minimize' arrow on the divider between the term list and the translator section. In order to expand the translator section again, simply click on the 'maximize' arrow or click somewhere along the slider (except on the arrows). Figure 6.3 displays how the SummaryCreator application can look after the translator section has been minimized.

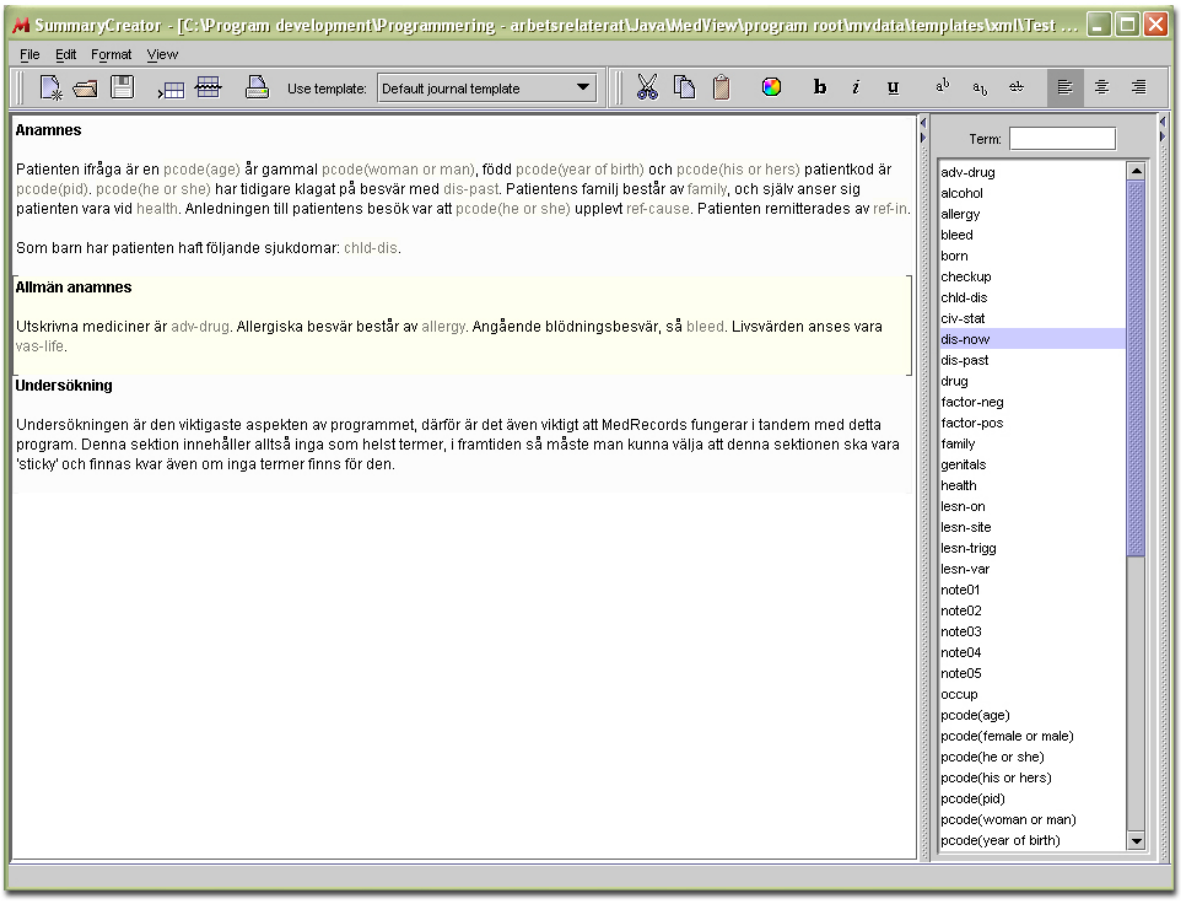


Figure 6.3 – The SummaryCreator application when viewing the template in a text view layout

To change the language or look-and-feel of the applications, open the preferences dialog, choose the ‘global’ tab, and make your choice among the available languages and look-and-feels contained in the combo boxes, as displayed in figure 6.4. The default look-and-feel used by the applications is the platform-independent Metal look and feel, which is the one used in most screenshots contained in this thesis. Note that the available list of look-and-feels may be different depending on the current platform you are using due to copyright issues. For instance, choosing the Macintosh look-and-feel can only be done if you are running the applications on a Macintosh system, while choosing the Windows look-and-feel can only be done if running on a Windows system. An example of how the applications might look after having switched to the swedish language and the Windows look-and-feel on a Windows XP system is shown in figure 6.5

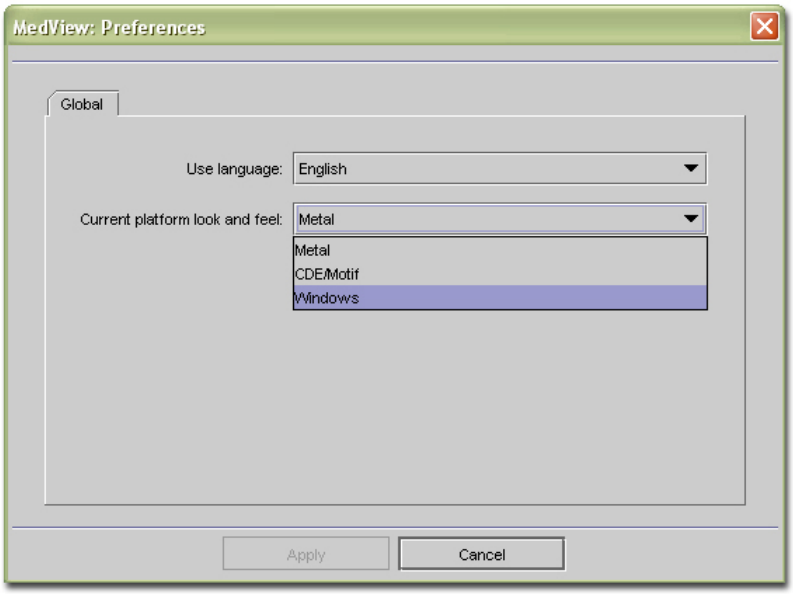


Figure 6.4 – Choosing the language and/or look and feel of the application

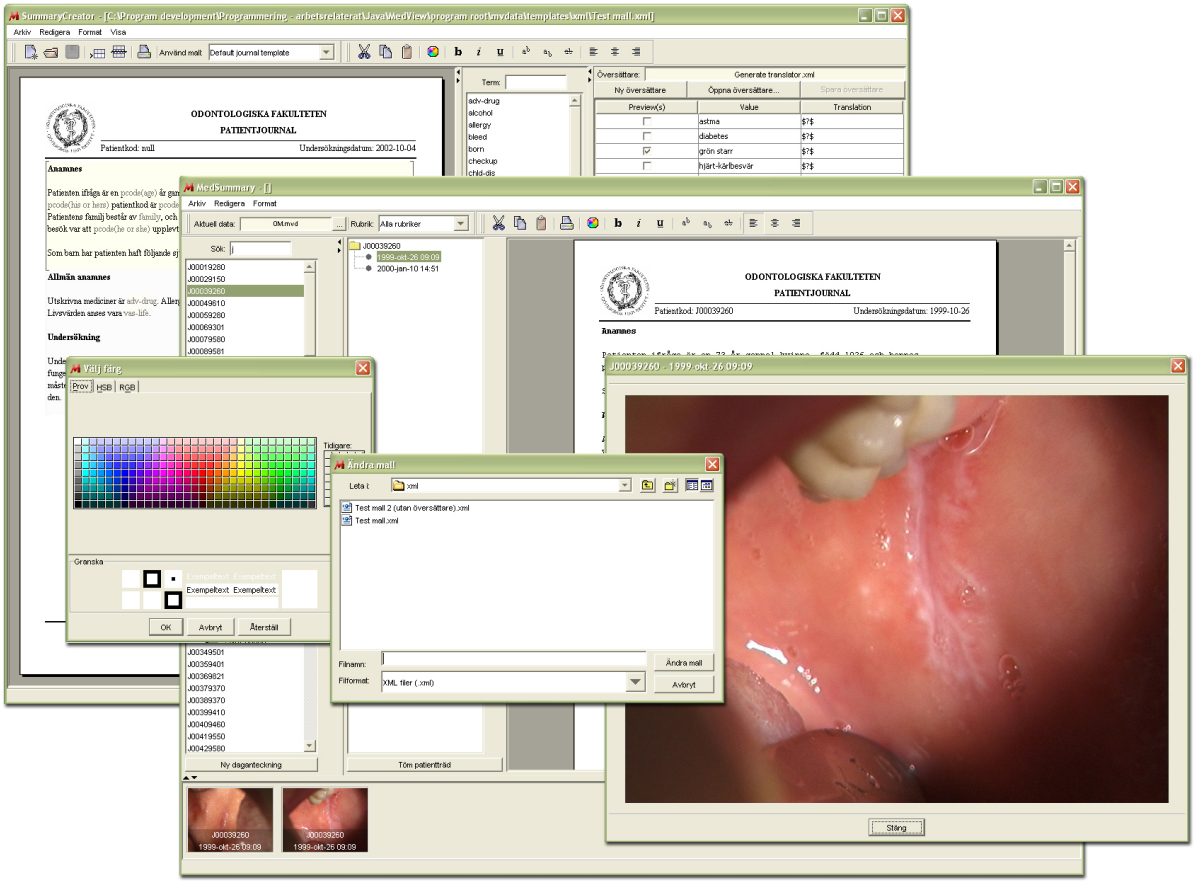


Figure 6.5 – The applications and various dialogs in swedish with a Windows look-and-feel

6.2 About the SummaryCreator Application

The SummaryCreator application's main purpose is to create the necessary components needed for the generation of patient journals. In order for a journal to be generated there are three necessary components:

1. A translator, containing translations for the possible values that may occur in an examination record.
2. A template, defining the context of the generated journal, with slots where the translated content from the examination record is placed.
3. The actual examination record, which can be seen as a number of equations defining the examination's values for the general terms as described in the 'analysis' part of this thesis.

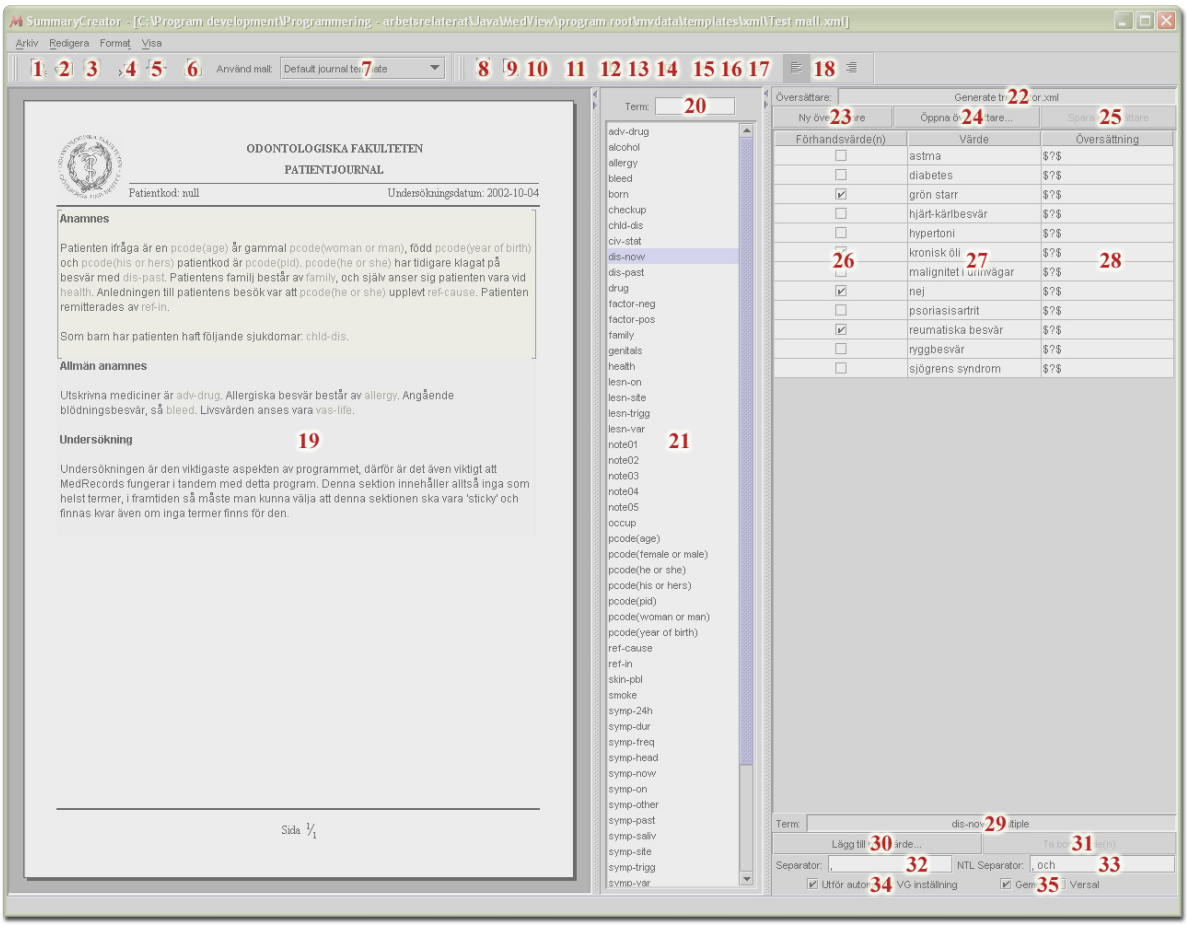
The terms contained in a translator may currently be of one of five different types:

1. The 'regular' type, corresponding to a term that may have only one value – for instance, a person can only be born in one country, so the 'born' term is of type 'regular'.
2. The 'multiple' type, corresponding to a term that may have several values – for instance, a person can be allergic to several things, thus the 'allergy' term has a type of 'multiple'.
3. The 'free' type, indicating that values for the term should not be translated or processed in any way prior to placing them into their slots in the template.
4. The 'pcode' type, indicating that the term value is derived from the patient identifier of the patient whose examination is being summarized.
5. The 'interval' type, indicating that values for the term are numerical and within certain intervals – for instance, say a translation for values in the interval '0-2' is given for a certain term, then a value of '1.5' in the examination record for this term should be translated according to this translation.

6.3 SummaryCreator Application Overview and General Documentation

Figure 6.6 presents an overview of the SummaryCreator application, as well as giving a short explanation of what the various parts do. Note that all various functionality found in the application's toolbars and buttons may also be found in the application's top-level menus. The text-edit actions (8-18) apply to the currently chosen text in the template section (19) – if no template is currently being displayed, these actions will be disabled. The translator actions (29-35) apply to the current translator being visualized in the translator section (26-28) and identified by the descriptor (22). Creating, loading, and saving templates is done by using the corresponding toolbar buttons (1-3), while creating, loading, and saving translators is done by using the corresponding buttons in the translator section (23-25).

Certain buttons and fields in the translator section might be disabled at times if the currently chosen term does not have a type matching the functionality represented by these components. For instance, if a term of type 'regular' is chosen, the components dealing with separators are disabled since they do not apply to such a term.



1. Creates a new template
2. Opens existing template
3. Saves current template
4. Creates a new section
5. Removes chosen section
6. Prints template
7. Chooses page template
8. Cuts text
9. Copies text
10. Pastes text
11. Changes color of text
12. Sets text style to bold
13. Sets text style to italic
14. Sets text style to underline
15. Sets text style to superscript
16. Sets text style to subscript
17. Sets text style to strikethrough
18. Changes paragraph alignment
19. Visualizes the template
20. Searches for a term
21. Displays the term list
22. Displays the current translator
23. Creates a new translator
24. Opens existing translator
25. Saves current translator
26. Sets value preview status
27. Displays term values
28. Displays term value translations
29. Displays current term name/type
30. Adds a new value to term
31. Removes a value from term
32. Displays Separator (’multiple’ terms)
33. Displays NTL sep. (’multiple’ terms)
34. Performs auto-VG adjustment
35. Specifies type of VG adjustment

Figure 6.6 – Overview and explanation of the various parts of the SummaryCreator application

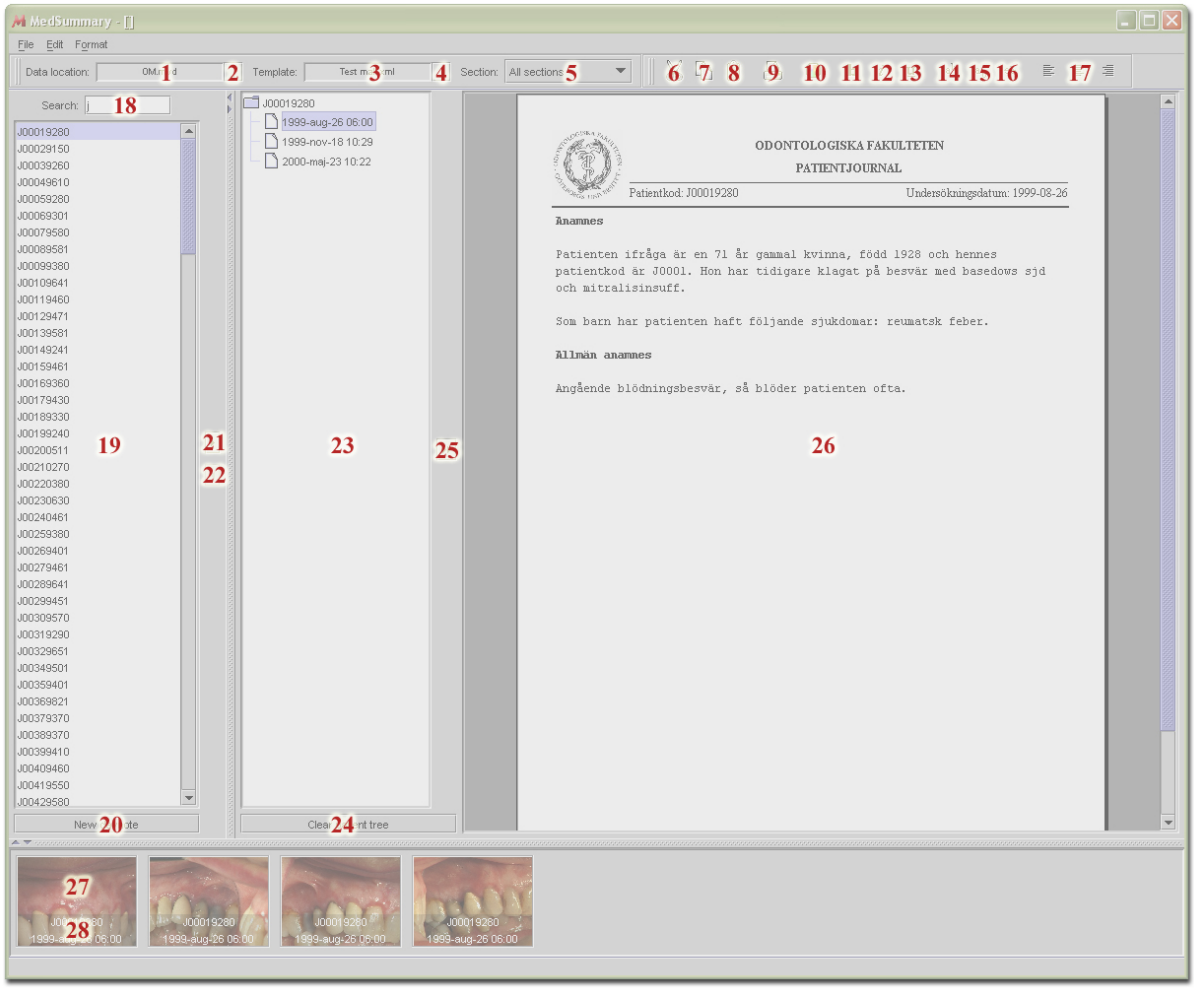
6.4 About the MedSummary Application

The main purpose of the MedSummary application is to provide information about examinations by summarizing them in various ways as well as displaying the photos taken during the examinations. A listing of the patients kept in the currently selected knowledge base is provided, and each patient’s associated examinations and photos taken during the examinations can be displayed at request. The user can choose the kind of summary to be generated by deciding the template and translator to use for generation. Furthermore, the user can choose the sections (available in the chosen template) to include in the generated journal. The MedSummary application acts as a ‘bridge’ between displaying knowledge base content

and adding new information to it - a new daynote can be added for a patient in the list by selecting the patient followed by requesting a new daynote (see (20) in figure 6.7 below).

6.5 MedSummary Application Overview and General Documentation

Figure 6.7 displays an overview of the MedSummary application, as well as some short descriptions of the various parts. Note that the translator combo box has been removed from the application toolbar in the figure – the components that are to appear in the toolbar can be adjusted in the 'MedSummary' tab in the preferences dialog. Note that all various functionality found in the application's toolbars and buttons may also be found in the application's menus.



1. Displays current data location
2. Changes current data location
3. Displays current temlpate
4. Changes current template
5. Changes included sections
6. Cuts text
7. Copies text
8. Pastes text
9. Prints current journal
10. Changes color of text
11. Sets text style to bold
12. Sets text style to italic
13. Sets text style to underline
14. Sets text style to superscript
15. Sets text style to subscript
16. Sets text style to strikthrough
17. Changes paragraph alignment
18. Searches the patient list
19. Displays the patients in the current data location
20. Requests new daynote for the chosen patient
21. Adds chosen patient to tree
22. Removes examinations from tree
23. Displays patient examination information
24. Clears the tree from patients
25. Generates a summary of the chosen patients in the tree
26. Displays the generated summary
27. Displays thumbnails of chosen patient images
28. Displays examination information of a certain patient image

Figure 6.7 – Overview and explanation of the various parts of the MedSummary application

The lower part of the photo thumbnail (28) displays the patient code for the patient being viewed as well as the date of the examination when the photo was taken. In order to view the photo in full size, simply double-click on it (27). Figure 6.8 shows the photo dialog in action. As can be seen from the figure, the photo dialog contains the patient code as well as the examination date in the title bar. When done viewing the photo, just press the ‘close’ button to close the dialog.

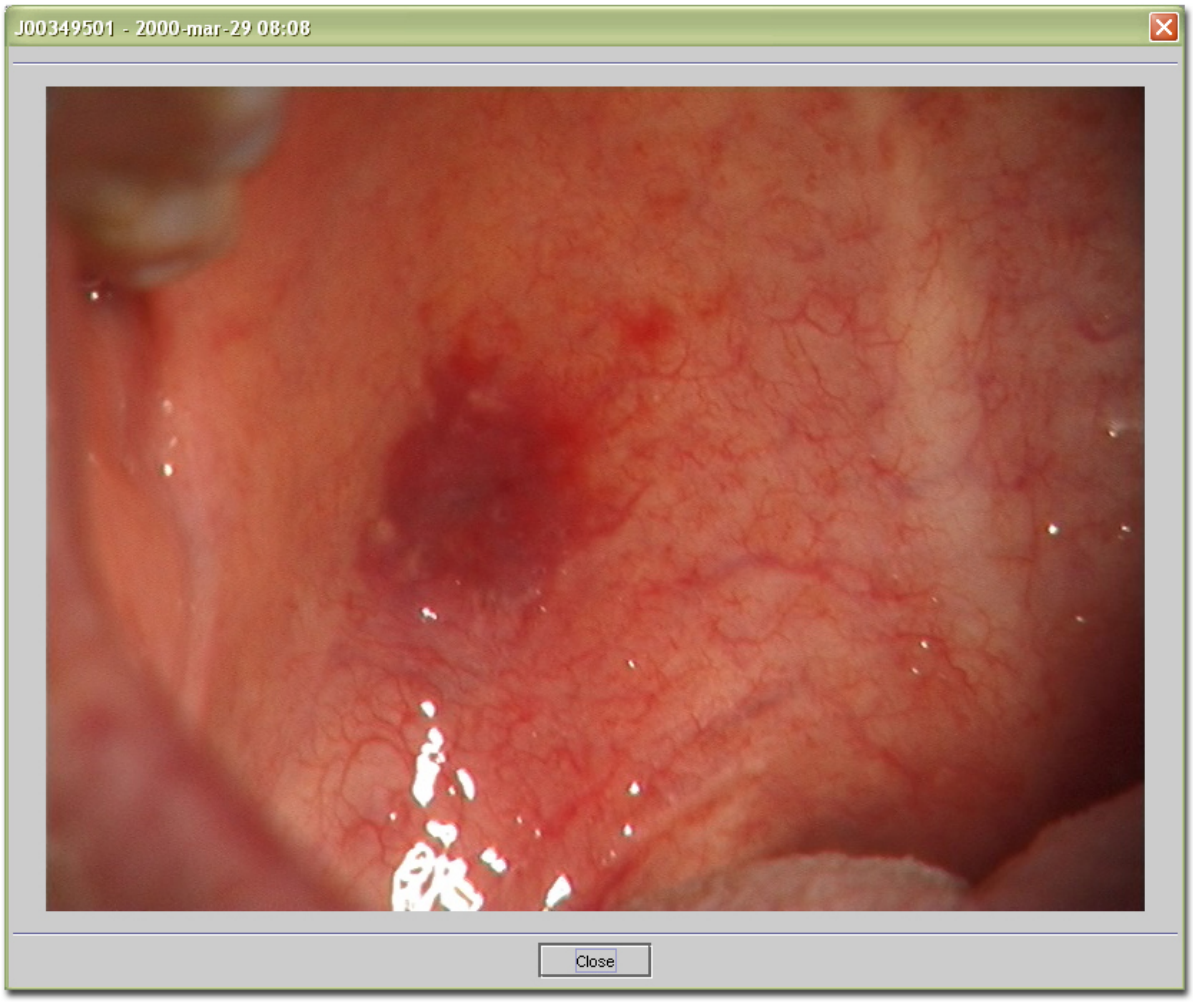


Figure 6.8 – The photo dialog in action

The text-edit actions (6-17) apply to text contained in the journal (26) – if no journal has been generated, the text edit actions (as well as the print action (9)) are disabled. If you change the data location (2), the patient listing is updated to reflect the patients in the new location. The tree of chosen patients (23) will be cleared whenever the data location is changed.

If you have a long list of patients, you may narrow down the list by using the text field above the patient listing (18) – when entering text into this field, the patient listing is narrowed down to one containing the entries that match the text field content. In order to add a patient to the tree of patients (23), you can either double-click on the patient directly in the list or use the ‘add patient’ button (21). If you wish to remove a patient from the tree, you select the patient in the tree and press the ‘remove patient’ button (22). If you want to add or remove several patients at once, hold down the control key while selecting. Furthermore, all patients

contained in the tree can be cleared in one sweep by pressing the ‘clear patient tree’ button (24).

The ‘generate journal’ button (25) is only enabled if all components required for generating a journal are available. Thus, you must have chosen a template, a translator, and the examination(s) you wish to summarize in order for the generator engine to be functional and for the generate button to be enabled. After pressing the generate button, the journal area (25) will display the generated journal. Furthermore, after a journal has been generated, the text-edit actions and the print action will be enabled.

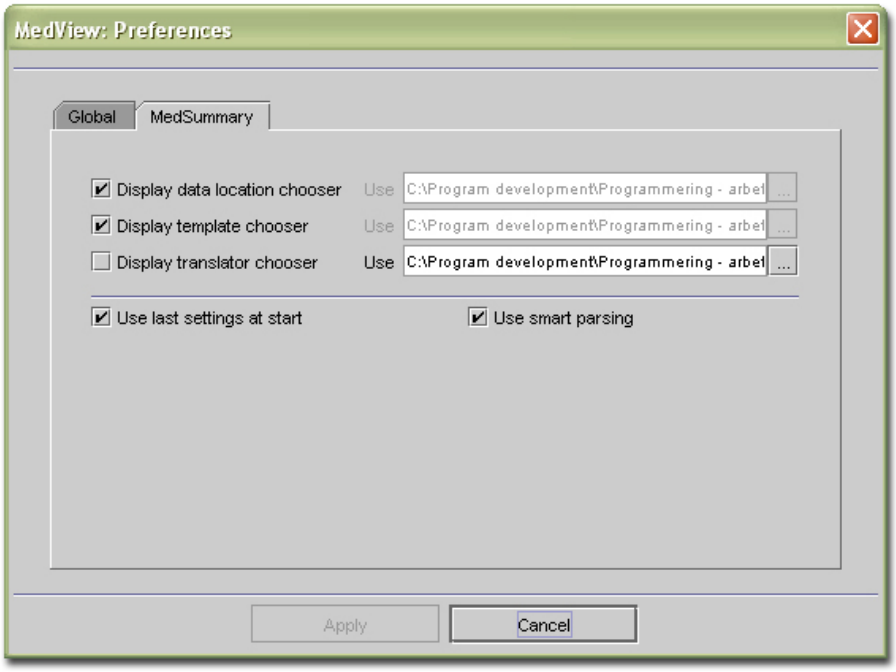


Figure 6.9 – The MedSummary tab of the settings dialog

If you are running the application on a screen resolution lower than 1280x1024, you might want to remove some of the toolbar components in order to save space. To do this, you open the preferences dialog from the file menu, select the ‘MedSummary’ tab, and deselect the components you do not want to have displayed, as seen in figure 6.9.

7 Glossary

API	Application Programming Interface. A ‘contract’ defining programming resources available to the programmer / developer. The Java API contains a bunch of packages and classes with corresponding methods and members that form the foundation when developing Java applications.
Artifact	A general term for any work product in the UP. A certain discipline usually produces certain artifacts.
CASE	Computer Assisted Software Engineering – general term for any technique using computers to assist during development of software.
CASE-tool	Some tool (usually an application, such as Rational Rose or MagicDraw UML) used to assist in software development.
Cohesion	A measure of how ‘tight’ an object is in regard to what responsibilities it fulfills. If an object only has responsibilities within one small area, it has high cohesion. If an object takes on responsibilities within several areas, it has low cohesion. Low cohesion usually infers high coupling.
Controller	One of the nine GRASP patterns [1]. Provides guidelines for how to place responsibilities related to receiving system events in the domain layer of applications.
Coupling	A measure of how ‘connected’ a class is to other elements. High coupling infers a high degree of connectivity between classes, leading to a fragile system where changes in one class has high impact on other elements, which is undesirable.
Creator	One of the nine GRASP patterns [1]. States that one should assign the responsibility of creating an instance of another class to a class that either aggregates, contains, records instances of, has the initializing data for, or closely uses the created class. If initialization is complex, the Factory GoF design pattern might be used instead.
Design pattern	Named problem-solution pair that gives good advice and principles often related to the assignment of responsibilities [1].
Discipline	UP terminology for one step in the iterative development process. Contains a set of activities related to the discipline.
GoF	A.k.a. ‘Gang of Four’, the four authors (Gamma, Helm, Johnson, Vlissides) of the book ‘Design Patterns’, which is central to the area of design patterns and object design.
GRASP	General Responsibility Assignment Software Patterns. A collection of nine basic patterns forming the ‘building blocks’ for more advanced design patterns [1].
GUI	Graphical User Interface. The interface between the application and the user, as shown graphically on screen.
High Cohesion	One of the nine GRASP patterns [1]. States that one should assign responsibilities between objects in such a way that cohesion remains high, in order to make manage complexity.
Indirection	One of the nine GRASP patterns [1]. States that you can assign responsibilities to an intermediary object if you wish to improve the coupling aspects of your system. The objects surrounding the intermediary are no longer coupled.
JAXP	Java Api for Xml Processing. An implementation-independent api for java dealing with various xml processing.

JRE	Java Runtime Environment. An environment providing the ability to run Java programs (basically the Java Virtual Machine along with the various Java support classes).
Information Expert	One of the nine GRASP patterns [1]. States that one should assign responsibilities to the classes containing the information necessary to fulfill the responsibility.
Knowledge base	An alternative name for the database containing examination data used in MedView.
Low Coupling	One of the nine GRASP patterns [1]. States that coupling between classes should be low in order to avoid the negative effects that come with high coupling (a fragile system with high change impact).
MedRecords	An application used to input information into the knowledge base. Used by the clinician during an actual examination.
MedSummary	An application used to visualize information in the knowledge base by providing the ability to generate a summary of chosen examination(s) as well as displaying images associated with the examination(s) in question.
NLG	Natural Language Generation. The process of generating text that as closely as possible resembles natural language.
Objective-C	An object-oriented programming language containing similarities to Smalltalk.
Package	A grouping entity containing related entities. Similar to the concept of a 'directory' in a filesystem. Example: a package 'math.pi' contains entities dealing with mathematical pi calculations.
Pcode	A.k.a. 'patient code', identifying uniquely a patient at Kliniken för Oral Medicin at Odontologen, Gothenburg.
Polymorphism	One of the nine GRASP patterns [1]. States that, when behavior varies by type, you should assign responsibilities for the behavior using polymorphic operations to the types for which the behavior varies.
Protected Variations (PV)	One of the nine GRASP patterns [1]. States that you should identify points of predicted instability or variation and assign responsibilities to create a stable interface around them.
Pure Fabrication	One of the nine GRASP patterns [1]. States that you can assign responsibilities to a 'made up' class, i.e. one that is not present as a domain concept in the domain of your application, if it enhances desirable qualities in your system such as low coupling or high cohesion.
Refactoring	Improving an existing design and implementation without affecting the functionality of the existing system.
Reverse-engineering	To take existing code and construct or update UML diagrams from it.
RTF	Rich Text Format. A poorly documented format used previously by Microsoft for styled text files.
Slot	A place-holder in the template text flow that is replaced by a value from a chosen examination during generation time.
SOMNET	Swedish Oral Medicine NETwork. A network of dentists and practitioners in the area of oral medicine.
Subsystem	A discrete entity with behavior and interfaces, usually modelled as a special kind of package or object [1].
SummaryCreator	An application used to develop templates and translators that are used when generating patient journals.

Template	A styled, textual document containing slots which will be filled with examination content during journal-generation time.
Term	A clinical term used to represent a clinical concept (for instance, the term ‘born’ represents the concept of where the patient is born).
Term type	The type of a term when considered from a text generation context. For instance, if a term is of type ‘interval’ its translations look like: ‘0–2: mild, 2–4: moderate, 4–6: serious, 6-8: dangerous’.
Translator	Container of mappings from universal values (found in examination records) to translations, which in turn may contain macros for further expansion at journal-generation time.
Treefile	The format used to store an examination at the time of this writing.
Unified Modeling Language (UML)	A set of diagrams for describing the various aspects of a system.
Unified Process (UP)	An iterative system development process making heavy use of the UML (Unified Modeling Language) for describing various aspects of the system under development.
Unit test	A test of a unit (could be a simple component or a complex subsystem) in isolation from surrounding units.
Value	A specific examination’s value for a specific term (for instance, the value ‘sweden’ is a specific value for the term ‘born’).
XML	eXtensible Markup Language. Provides a language-independent and platform-neutral means of describing and validating data [11].

8 References

[1] Larman, C. *Applying UML and patterns – An introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice-Hall PTR, Upper Saddle River NJ 2002.

[2] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, West Sussex England 1996.

[3] Xiaoping, J. *Object-Oriented Software Development Using Java: principles, patterns, and frameworks*. Addison-Wesley, 2000.

[4] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns*. Reading, MA 1995.

[5] Falkman, G., Hallnäs, L., Jontell, M., and Torgersson, O. *MedView – Design and Adaption of an Interactive System for Oral Medicine*. In A. Hasman et al, editors, *Medical Infobahn for Europe: Proceedings of MIE2000 and GMD2000*, IOS Press, 2000.

[6] Stelting, S., Maasen, O. *Applied Java Patterns*. Prentice Hall PTR, Upper Saddle River NJ 2002.

[7] Torgersson, O., Falkman, G. *Using Text Generation to Access Clinical Data in a Variety of Contexts*. In: Surján, G., Engelbrecht, R. & McNair, P. (eds.): *Health Data in the Information Society*, vol. 90 of Studies in Health Technology and Informatics, pp. 460-465. IOS Press.

[8] Youssef, A., Falkman, G., Hallnäs, L., Jontell, M., Mattsson, U., Nazari, N, Torgersson, O. *An Overview of MedView*.

[9] Nivre, J., Lager, T. *An Integrated Approach to Multilingual Hypertext Generation*.

[10] Reiter, E., Dale, R. *Building Applied Natural Language Generation Systems*. Cambridge University Press, 2000.

[11] Ahmed, K., Ancha, S., Cioroianu, A., Cousins, J., Crosbie, J., Davies, J., Gabhart, K., Gould, S., Laddad, R., Li, S., Macmillan, B., Rivers-Moore, D., Skubal, J., Watson, K., Williams, S., Hart, J., *Professional Java XML*. Wrox press, Birmingham 2001.

Appendix A – Examples of Previous MedView System Resources

$ANAMNES$
Överkänslighet: N.
Blödningsbenägenhet: O.

Allmänt
A-årig B F som C för D. $Note1$.

Allmän anamnes
H. $K1$. U. Patienten har Y.

Besvär från genitalia slemhinnan: X.

$Note2$.

I.
J.
L.
M.
P.
O.

Symptomanknutna uppgifter: Patienten anger för närvarande Y. Besvären är lokaliserade till AC. Graderas till Z enligt VAS. Patienten rapporterar att besvären debuterade för cirka AI sedan. Patienten upplever att AE initierade symptomen. Besvären är AF. Durationen anges till AG med en frekvens på AH. Besvären AI. Uppger sig tidigare ha haft AA i munslemhinnan. Anger att dessa besvär enligt VAS var AB. $Note3$.

På frågan om vad som ökar symptomen anger patienten AJ. Därernot så minskar symptomen AK. Tidigare har AL givit positivt resultat. I samband med att patienten erhållit AM har en negativ påverkan upplevts även patienten AN har haft liknande besvär. $Note4$.

Värmdsförändring: Lesionen observerades första gången för AP sedan. Patienten upplever att AQ har samband med debuten. Lokaliserar själv lesionen till AR. AS. $Note5$.

Mutontorlet
Vätska vid frobintag: $A0$.
Svårigheter att tala: AV.
Duration av mutontorlet: $A1$.
Andra symtom från munhålan: $A2$.
Spottkörtelsvullnad: $A3$.
Spottkörtelundersökning: $A4$.
Reumatisk sjukdom: $A5$.
Hereditet reumatisk sjukdom: $A6$.
ogonstorrhet > 3 månader: $A7$.
ogonsymtom: AU.
ogonoroppar: AZ.
ogonundersökning BA.

$Note6$.

Tandläkarebehandling Har genomgått BB hos tandläkare. Bettfysiologiskt uppvisar patienten BC. BD. $Note7$.

Tandvårdsrädsla
Patienten uppger EA rädsla vid tandläkarebesök. Regelbunden tandvård EB. Fullständig tandvård för EC. Senaste tandläkarebesöket upplevdes som ED. $Note2$.
Patienten tror att rädslan i första hand hänger samman med. EE. Ansersig alltid varit EF för tandvård och tandläkare. Som barn upplevdes tandvårdens som EG. Patienten anger att behandlingar ofta har varit EH.
Tror att rädsla hos andra i omgivningen kan ha bidragit till den upplevda EI. Samtidt anges rädsla hos EJ.
Patienten upplever dessutom EK för att förlora kontrollen i samband med tandläkarbesök. Söker just nu behandling pga att EL. Det känns EM för patienten att få hjälp med sin rädsla. Patienten uppger att det känns EN att få behandling av tänderna. Tror att möjligheten att bota rädslan är EO.
Motivationsengagemang för förbehandlig bedöms patienten som EP. Patienten tror att behandlingsformer som EQ kan vara lämpliga. Anger särskilt att tandläkaren skall: ER. ES. ET. EU. Dessutom vill patienten gärna att EV. Patienten upplever negativa konsekvenser av tandvårdsrädslan i form av: EW.
Tandvårdsrädslan ger upphov till komplikation när det gäller kontakter med ZZ. EX. EY. Andra negativa konsekvenser är: $E. $. Känslor förknippade med tandvårdsrädsla är: EG. PA. PB. FC.

Psykometri
HAD-A: BE.
HAD-D: BG.
DAS: BH.
DFS: BE.
GFSILL: FG.
GFSBMB: FI.
GFSSOC: FK.
GFSFYS: FM.
GFSANI: FO.
GFSMBAN: FQ.
GFSBOB: FR.
$Note8$.

HAD-A_m: BF.
HAD-D_m: BH.
DAS_m: FD.
DFS_m: FF.
GFSILL_m: FH.
GFSBMB_m:FJ.
GFSSOC_m: FL.
GFSFYS_m:FN.
GFSANI_m:FP.

$STATUS$
Direkt status
Patienten uppvisar BK slemhinneförändring. Förändringen är lokaliserad till BI. Reaktionsmönstret karaktäriseras av BM. Storleken är cirka BN. $Note9$. Palpation av BU. BV vid palpation. BZ. BX mot underlaget. Storleken bedöms vid palpation till cirka BY. Patienten CA $Note11$.

Bettfunktion BO. Patienten uppvisar BP. BQ. BR. Patienten har odontologiskt behandlats med BS. Som material har använts BT. $Note10$ $Note 2$

Indirekt status
Röntgen CB tas mot CC. Röntgen visar CD. Röntgen på aktuella tänder visar CE $Note12$

Histopatologi: Biopsi tas i CF. Preparatet utgörs av CG. Den histopatologiska undersökningen visar CG epitelet. Intraepitelialt ses CH. Immunfluorescens visar CI reaktion i epitelet. I preparatet ses CK. Förekomst av CL. Immunfluorescens visar CM depositioner i bindväven. $Note13$

Mikrobiologi: Mikrobiologiskt prov tas i CN. I provet påvisas växt av CO. $Note14$

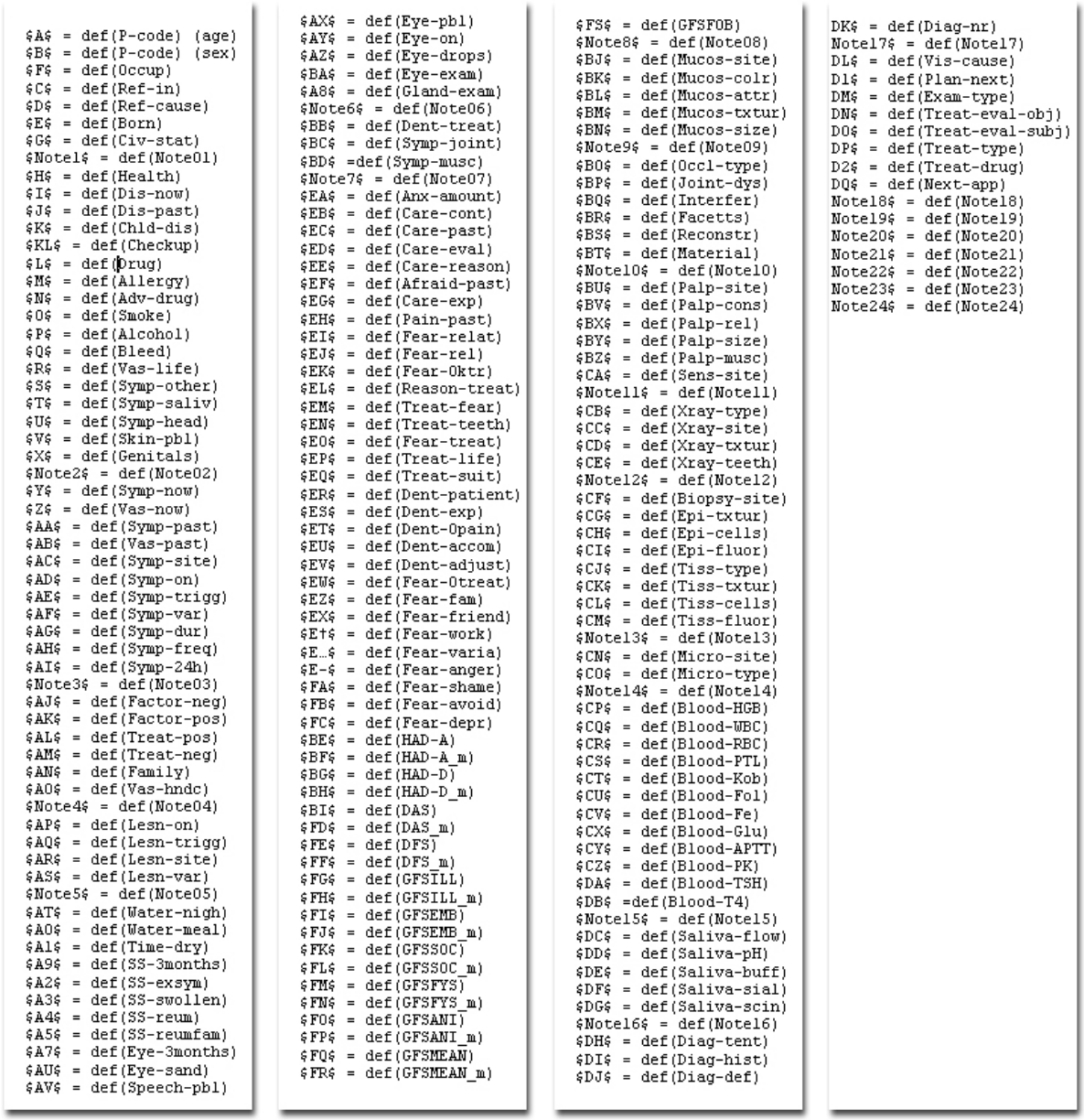
Hematologi: B-HB = CP. B-WBC = CQ. B-RBC = CR. B-PTL = CS. S-B12 = CT. S-folat = CU. ES-JÄm = CV. fB-Glukos = CX. P-APTT = CY. P-PTK = CZ. B-TSH = DA. S-FntrT4= DB. $Note15$.

Salivvärden: Salivsekretion = DC. Saliv-pH = DD. Buffringskapacitet = DE. I stimulerad saliv syres DF. I sviktgrafin syres DG. $Note16$.

STENTATIV DIAGNOS: DH.
SHISTOPATOLOGISK DIAGNOS: DI.
$DIAGNOS: DJ.
Diagnos-nummer: DK. $Note17$

$DAGANTECKNINGS
Besöket avser DL. Planerade åtgärder: $D1$. Idag genomfördes DM. Klinisk noterar att status objektivt är DN. Patienten upplever sig subjektivt DO. Patienten erhåller DP. $D2$. DQ. $Note18$.

Figure A.1 – Template (rtf) file for use in the earlier MedView system



```
$A$ = def(P-code) (age)
$B$ = def(P-code) (sex)
$F$ = def(Occup)
$C$ = def(Ref-in)
$D$ = def(Ref-cause)
$E$ = def(Born)
$G$ = def(Civ-stat)
$Note1$ = def(Note01)
$H$ = def(Health)
$I$ = def(Dis-now)
$J$ = def(Dis-past)
$K$ = def(Chld-dis)
$KL$ = def(Checkup)
$L$ = def(Drug)
$M$ = def(Allergy)
$N$ = def(Adv-drug)
$O$ = def(Smoke)
$P$ = def(Alcohol)
$Q$ = def(Bleed)
$R$ = def(Vas-life)
$S$ = def(Symp-other)
$T$ = def(Symp-saliv)
$U$ = def(Symp-head)
$V$ = def(Skin-pbl)
$X$ = def(Genitals)
$Note2$ = def(Note02)
$Y$ = def(Symp-now)
$Z$ = def(Vas-now)
$AA$ = def(Symp-past)
$AB$ = def(Vas-past)
$AC$ = def(Symp-site)
$AD$ = def(Symp-on)
$AE$ = def(Symp-trigg)
$AF$ = def(Symp-var)
$AG$ = def(Symp-dur)
$AH$ = def(Symp-freq)
$AI$ = def(Symp-24h)
$Note3$ = def(Note03)
$AJ$ = def(Factor-neg)
$AK$ = def(Factor-pos)
$AL$ = def(Treat-pos)
$AM$ = def(Treat-neg)
$AN$ = def(Family)
$AO$ = def(Vas-hndc)
$Note4$ = def(Note04)
$AP$ = def(Lesn-on)
$AQ$ = def(Lesn-trigg)
$AR$ = def(Lesn-site)
$AS$ = def(Lesn-var)
$Note5$ = def(Note05)
$AT$ = def(Water-nigh)
$AO$ = def(Water-meal)
$A1$ = def(Time-dry)
$A9$ = def(SS-3months)
$A2$ = def(SS-exsym)
$A3$ = def(SS-swollen)
$A4$ = def(SS-reum)
$A5$ = def(SS-reumfam)
$A7$ = def(Eye-3months)
$AU$ = def(Eye-sand)
$AV$ = def(Speech-pbl)

$AX$ = def(Eye-pbl)
$AY$ = def(Eye-on)
$AZ$ = def(Eye-drops)
$BA$ = def(Eye-exam)
$A8$ = def(Gland-exam)
$Note6$ = def(Note06)
$BB$ = def(Dent-treat)
$BC$ = def(Symp-joint)
$BD$ =def(Symp-musc)
$Note7$ = def(Note07)
$EA$ = def(Anx-amount)
$EB$ = def(Care-cont)
$EC$ = def(Care-past)
$ED$ = def(Care-eval)
$EE$ = def(Care-reason)
$EF$ = def(Afraid-past)
$EG$ = def(Care-exp)
$EH$ = def(Pain-past)
$EI$ = def(Fear-relat)
$EJ$ = def(Fear-rel)
$EK$ = def(Fear-Oktr)
$EL$ = def(Reason-treat)
$EM$ = def(Treat-fear)
$EN$ = def(Treat-teeth)
$EO$ = def(Fear-treat)
$EP$ = def(Treat-life)
$EQ$ = def(Treat-suit)
$ER$ = def(Dent-patient)
$ES$ = def(Dent-exp)
$ET$ = def(Dent-Opain)
$EU$ = def(Dent-accom)
$EV$ = def(Dent-adjust)
$EW$ = def(Fear-Otreat)
$EZ$ = def(Fear-fam)
$EX$ = def(Fear-friend)
$Et$ = def(Fear-work)
$E…$ = def(Fear-varia)
$E-$ = def(Fear-anger)
$FA$ = def(Fear-shame)
$FB$ = def(Fear-avoid)
$FC$ = def(Fear-depr)
$BE$ = def(HAD-A)
$BF$ = def(HAD-A_m)
$BG$ = def(HAD-D)
$BH$ = def(HAD-D_m)
$BI$ = def(DAS)
$FD$ = def(DAS_m)
$FE$ = def(DFS)
$FF$ = def(DFS_m)
$FG$ = def(GFSILL)
$FH$ = def(GFSILL_m)
$FI$ = def(GFSEMB)
$FJ$ = def(GFSEMB_m)
$FK$ = def(GFSSOC)
$FL$ = def(GFSSOC_m)
$FM$ = def(GFSFYS)
$FN$ = def(GFSFYS_m)
$FO$ = def(GFSANI)
$FP$ = def(GFSANI_m)
$FQ$ = def(GFSMEAN)
$FR$ = def(GFSMEAN_m)

$FS$ = def(GFSFOB)
$Note8$ = def(Note08)
$BJ$ = def(Mucos-site)
$BK$ = def(Mucos-colr)
$BL$ = def(Mucos-attr)
$BM$ = def(Mucos-txtur)
$BN$ = def(Mucos-size)
$Note9$ = def(Note09)
$BO$ = def(Occl-type)
$BP$ = def(Joint-dys)
$BQ$ = def(Interfer)
$BR$ = def(Facetts)
$BS$ = def(Reconstr)
$BT$ = def(Material)
$Note10$ = def(Note10)
$BU$ = def(Palp-site)
$BV$ = def(Palp-cons)
$BX$ = def(Palp-rel)
$BY$ = def(Palp-size)
$BZ$ = def(Palp-musc)
$CA$ = def(Sens-site)
$Note11$ = def(Note11)
$CB$ = def(Xray-type)
$CC$ = def(Xray-site)
$CD$ = def(Xray-txtur)
$CE$ = def(Xray-teeth)
$Note12$ = def(Note12)
$CF$ = def(Biopsy-site)
$CG$ = def(Epi-txtur)
$CH$ = def(Epi-cells)
$CI$ = def(Epi-fluor)
$CJ$ = def(Tiss-type)
$CK$ = def(Tiss-txtur)
$CL$ = def(Tiss-cells)
$CM$ = def(Tiss-fluor)
$Note13$ = def(Note13)
$CN$ = def(Micro-site)
$CO$ = def(Micro-type)
$Note14$ = def(Note14)
$CP$ = def(Blood-HGB)
$CQ$ = def(Blood-WBC)
$CR$ = def(Blood-RBC)
$CS$ = def(Blood-PTL)
$CT$ = def(Blood-Kob)
$CU$ = def(Blood-Fol)
$CV$ = def(Blood-Fe)
$CX$ = def(Blood-Glu)
$CY$ = def(Blood-APTT)
$CZ$ = def(Blood-PK)
$DA$ = def(Blood-TSH)
$DB$ =def(Blood-T4)
$Note15$ = def(Note15)
$DC$ = def(Saliva-flow)
$DD$ = def(Saliva-pH)
$DE$ = def(Saliva-buff)
$DF$ = def(Saliva-sial)
$DG$ = def(Saliva-scin)
$Note16$ = def(Note16)
$DH$ = def(Diag-tent)
$DI$ = def(Diag-hist)
$DJ$ = def(Diag-def)

DK$ = def(Diag-nr)
Note17$ = def(Note17)
DL$ = def(Vis-cause)
D1$ = def(Plan-next)
DM$ = def(Exam-type)
DN$ = def(Treat-eval-obj)
DO$ = def(Treat-eval-subj)
DP$ = def(Treat-type)
D2$ = def(Treat-drug)
DQ$ = def(Next-app)
Note18$ = def(Note18)
Note19$ = def(Note19)
Note20$ = def(Note20)
Note21$ = def(Note21)
Note22$ = def(Note22)
Note23$ = def(Note23)
Note24$ = def(Note24)
```

Figure A.2 – Template definition translation for use in the earlier MedView system

$Occup
Administrativ chef
Arbetssökanden
Arbetslös
Arbetsterapeut
Bankbiträdeman
Barnmorska
Barnsköterska
Besiktningsman
Byggnadsarbetare
Biomedicinsk analytiker
Bibliotekarie
Bilhandlare
Bitr fotsekare
Bokföring
Bovärd
Busschaufför
Byggnadssnickare
Dagbarnvårdare
Datakonsult
Datoroperatör
Domare
Ekonom
Ekonomibiträde
Florist
Lagerarbetare
Lokalvårdare
Marknadschef
Marknadsundersökare
MUIare
Maskinoperatör
MedUiggare
Mentalskötare
Murare
PensionUr
Postkassör
Postsorterare
Produktionsledare
Projektledare
Projektstatistiker
Professor
Psykolog
Officer
Rektor
Receptionssekreterare
RörlUiggare
SUIjare
SUIjprogramskoordinator
Sekreterare
Serviceteknoker
Servitör
SjukpensionUr
Sjuksköterska
Skatterevisor
Socionom
Studerande
Tandsköterska
Taxichafför

administrativ chef
arbetssökanden
arbetslös
arbetsterapeut
bankbiträdeman
barnmorska
barnsköterska
besiktningsman
byggnadsarbetare
biomedicinsk analytiker
bibliotekarie
bilhandlare
bitr fotsekare
kamrer
bovärd
busschaufför
byggnadssnickare
dagbarnvårdare
datakonsult
datoroperatör
domare
ekonom
ekonomibiträde
florist
lagerarbetare
lokalvårdare
marknadschef
marknadsundersökare
mUIare
maskinoperatör
medUiggare
mentalskötare
murare
pensionUr
postkassör
postsorterare
produktionsledare
projektledare
projektstatistiker
professor
psykolog
officer
rektor
receptionssekreterare
rörlUiggare
sUIjare
sUIjprogramskoordinator
sekreterare
serviceteknoker
servitör
sjukpensionUr
sjuksköterska
skatterevisor
socionom
studerande
tandsköterska
taxichafför

Teknikinformatör
TextillUrare
tjUrsteman
Trädgårdsarbetare
TvåUtbildUde
Undertalsmonitör
Undersköterska
VUversmed
VUndbiträde
Verkstadsarbetare
Yrkesvägledare
VUktare
–versUtbare

teknikinformatör
textillUrare
tjUrsteman
trädgårdsarbetare
tvåUtbildUde
undertalsmonitör
undersköterska
vUversmed
vUndbiträde
verkstadsarbetare
yrkesvägledare
vUktare
översUtbare

$Ref-in
SjUkv
LUkare
TandlUkare
Tandbyggenst
Oral medicin

sjuker sjUkv
remitterats fUr n UIkare
remitterats fUr n tandlUkare
remitterats fUr n tandbyggenst
nu Utetkommer

$Ref-cause
BesUvningsk Utvala efter tandbehandling
BMS
DUlagt tandstatus
EmaljfUrUUrningar
Hematologisk sjukdom
Hepatit B
Infektionsutredning
Metallsmak
MissfUrhUllande allergi mot dentala material
Osseös fUrUndring
SjUgrensutredning
SmakfUrUndring
SmUrta
SmUrutredning
SportskadesjukdomTandvUrdsUtvala
TorrhetskUnsla
–kad bUdningsbenUgenhet

$Born
Bosnien
Bulgarien
FUrbUrna
Danmark
Finland
Frankrike
Iran
Jugoslavien
Jugoslavien f d
Makedonien
Norge
Sverige
Turkiet
Ungern
USA

bosniskt
bulgariskt
fUrbUrskt
danskt
finskt
franskt
iranskt
jugoslaviskt
f d jugoslaviskt
makedonskt
norskt
svenskt
turkiskt
ungerskt
amerikanskt

Figure A.3 – A small part of the translation file used in the earlier MedView system

Figure A.4 –Part of an actual tree file defining one examination