UPTEC X 04 007
FEB 2004

ISSN 1401-2138

JONAS EKSTEDT

Implementing RegNum - a database of cladenames

Master's degree project

UPPSALA UNIVERSITET

Bioinformatics Programme

Uppsala University School of Engineering

UPTEC X 04 007	Date of issue 2004-02

Author

Jonas Ekstedt

Title (English)

Implementing RegNum - a database of cladenames

Title (Swedish)

Abstract

The RegNum database is a tool to be used by scientists wishing to register named clades in accordance with the principles put forth by the PhyloCode project. Information is supplied through a web interface open to the entire scientific community. The software product is based on Apache Cocoon, the Jetty servlet engine and the Hibernate object/relational persistence service.

Keywords

PhyloCode, biological nomenclature, phylogenetic systems

Supervisors

Mikael Thollesson
Evolutionary Biology Centre, Uppsala University

Scientific reviewer

Torsten Eriksson
The Royal Swedish Academy of Sciences, Stockholm

Project name	Sponsors
Language **English**	Security
ISSN 1401-2138	Classification
Supplementary bibliographical information	Pages **47**

Biology Education Centre Biomedical Center Husargatan 3 Uppsala
Box 592 S-75124 Uppsala Tel +46 (0)18 4710000 Fax +46 (0)18 555217

Implementing RegNum - a database of cladenames

Jonas Ekstedt

Sammanfattning

På 1700-talet skapade Linné ett system för att namnge växter och djur. Det fick stor genomslagskraft och än idag baseras de flesta taxonomiska system på de principer Linné lade fram. På senare tid har dock viss kritik förts fram mot de Linnéanska taxonomiska systemen. Det har lett fram till att ett nytt taxonomiskt system, kallat PhyloCode, har utvecklats.

PhyloCode projektet som tagit fram detta nya taxonomiska system har också bestämt att en databas (med arbetsnamnet RegNum) skall utvecklas. Via ett webbgränssnitt skall forskare kunna registrera namn på taxa i databasen enligt de regler som stipuleras i PhyloCoden. Tanken är att man loggar in på en webbsida och navigerar mellan ett antal formulär. I formulären fyller man i den information som PhyloCoden anger måste finnas med i en namnregistrering.

Detta examensarbete syftar till att utveckla RegNum databasen samt de webbgränssnitt användarna skall utnyttja för registrering av namn.

Examensarbete 20p i Bioinformatikprogrammet
Uppsala universitet februari 2004

Table of Contents

1 - Introduction

The PhyloCode document [1] is a formal set of rules governing phylogenetic nomenclature. It came into existance as an attempt to mitigate the problems exhibited in current taxonomic systems of Linnaean origin. It is also an attempt to make the concept of evolution a central tenet in taxonomy.

Currently the PhyloCode is only a document worked on by members from the taxonomic community but in July 2004 the International Society for Phylogenetic Nomenclature (ISPN) will be inaugurated. The PhyloCode document will subsequently be governed by organisations within the ISPN.

ISPN will also be responsible for providing the scientific community with the means to register taxon names according to the rules set forth in the PhyloCode. To this end it has been decided that a registration database be constructed (the *RegNum database*) which is the subject of this paper.

Before describing my work on the *RegNum database* this paper starts off describing the PhyloCode project. Specifically the basic principles of the code, its relation to present day codes and why there is a need to overhaul current taxonomic systems. The paper delve into general methodologies used in software creation and also describe how they were applied to this project. The result section shows what data model and user interface was agreed upon and how they were subsequently implemented. Finally the paper discusses the major obstacles encountered during the development process and outline which enhancements can be made before the software product is finally deployed.

1.1 - Taxa and taxonomic systems

All scientific disciplines are dependent on having a precise vocabulary with which people can communicate ideas. As stated by de Quieroz and Gauthier [2] “In some sense, progress in any scientific discipline can be measured in terms of further refinement, rather than escalating imprecision, in vocabulary”. In biology an important part of this vocabulary are the names designating species and groups of species (taxa).

A *taxon* is a named entity. The concept is not specific to biology. In biology a taxon traditionally refers to a group of organisms. A *taxonomic system* is essentially a set of naming conventions or rules. That is, what type of entities are eligible for naming and how they should be named. In biology there are several taxonomic systems. Most of the systems in widespread use today are derived from the Linnaean taxonomic system conceived in the late 18th century.

At present, communication between biologists is hampered by the fact that current naming conventions do not facilitate precise and stable definitions of taxa. Furthermore

the naming conventions are based on ideas that predates the concept of evolution. In every other field of biology, the concept of evolution have had a profound effect when introduced, not so in the current taxonomic systems.

The Linnaean taxonomic system starts with the premise that taxa should designate groups of organisms that share some organismal traits or characters [2]. For example the zoological code states that a definition "purports to give characters differentiating a taxon" [3]. Furthermore taxa are arranged in hierarchical groups (figure 1) with lower order taxa designating a subset of the organisms of the parent taxon.

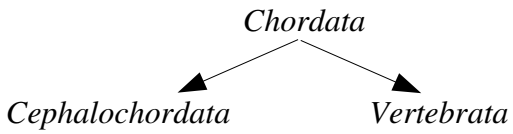

Figure 1: An example of three taxa and how they relate to each other. The taxon *Chordata* comprise, among others not shown here, *Cephalochordata* and *Vertebrata.* That is, all members of *Vertebrata* are also members of *Chordata.* Likewise for *Cephalochordata*.

The Linnaean taxonomic system also introduced the concept of rank. The ranks are levels in the taxon hierarchy to which each taxon must belong (figure 2). The rank of a taxon has no biological relevance. Linnaeus original taxonomic system had only a few discrete levels to which taxa could belong (*i.e. Kingdom*, *Phylum*, *Class*, *Order*, *Family*, *Genus* and *Species*). Later taxonomic systems have introduced intermediate ranks.

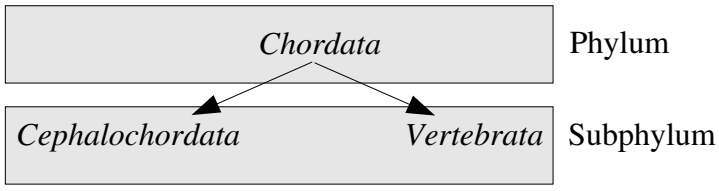

Figure 2: *Chordata* belongs to rank *Phylum* while *Cephalochordata* and *Vertebrata* are taxa of rank *Subphylum*.

Currently there are three major taxonomic systems in use.

- International Code of Zoological Nomenclature (ICZN)
- International Code of Botanical Nomenclature (ICBN)
- International Code of Nomenclature of Bacteria(ICNB)

1.2 - The criticism against current taxonomic systems

A number of objections have surfaced against the current taxonomic systems. Most notably the work of de Queiroz and Gauthier [2] shows that there are several drawbacks in using current taxonomic system. Their main argument is that rather than naming groups of species the names should designate clades.

Whereas early attempts at classification of species revolved around *shared organismal traits*, systematists today often focus on *shared evolutionary ancestry* as the primary means of grouping together species. During the last few decades this trend has also been affecting taxonomies. Taxonomists today usually investigate the evolutionary history of a set of species and then use the genealogical information to apply names so that related species are indeed closely related in the taxonomies. In effect they are trying to use the current taxonomic systems to name clades.

The problem has been that the current taxonomic systems are not well equipped to handle the concept of clades. For example the rank concept, introducing discrete levels in the taxonomic hierarchies to which each taxon must belong, is at complete odds with how the the continuous process of evolution works. It is simply not possible to name each and every clade in the tree of life if names can only be put at some limited set of arbitrary levels.

If it is clades that scientists are currently naming, albeit within a taxonomic system which does not recognize the concept of clades, de Queiroz and Gauthier reasons that it is better to create a new taxonomic system that does just that, designate clades, rather than try to tweak the current systems to use it for something it was never intended to do.

1.3 - The PhyloCode

The criticism against current taxonomic systems led de Queiroz and Gauthier to publish a series of papers [2, 4, 5] in which they outlined the theoretical foundation for a new taxonomic system that derived its principles from the central tenet of evolution. In August 1998 a workshop was held at Harvard University and the PhyloCode project was formed. In April 2000 the first public draft of the PhyloCode was announced and it has since been succeeded by several revisions [1].

The following sections are a brief introduction to the PhyloCode. The PhyloCode is a rather sizable document so only certain aspects that has any immediate implications for the *RegNum* project are covered.

1.3.1 - The definition of a name

As mentioned previously biologists today are in agreement that taxonomies should preferably reflect evolutionary history. The PhyloCode solves this by stating that a taxon name is a phylogenetic definition and should represent a specific clade. A clade is a group of organisms that are all descendants from the same ancestral species.

In conjunction with a particular phylogeny (a hypothesis of the evolution of a set of organisms), a phylogenetic definition will identify a group of species or organisms that are part of the taxon being named.

In the PhyloCode there are three types of a phylogenetic definitions. Node based, stem based and apomorphy based definitions (figure 3).

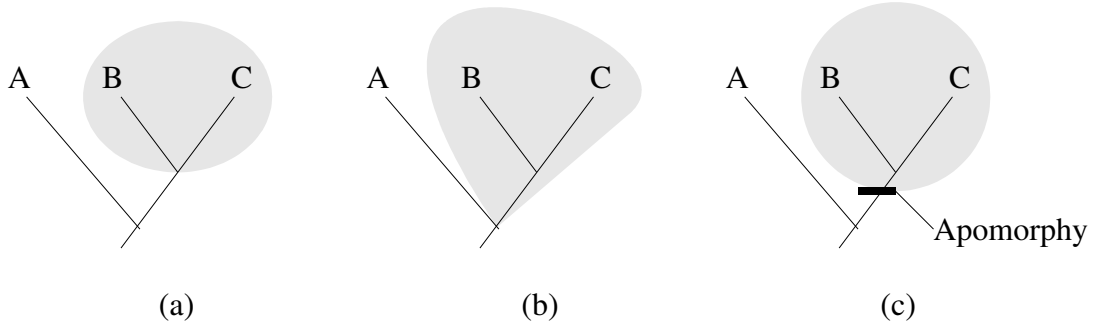

Figure 3: The three types of phylogenetic definitions used in the PhyloCode. Node based, stem based and apomorphy based definitions.

Figure 3 above shows a clade containing species A, B and C. The three illustrations depict three different ways of defining the clade containing species B and C.

A *node based definition* (figure 3a) associates a name with the clade "stemming from the immediate common ancestor of two designated descendants" [2]. The shaded area shows the minimum possible clade containing both B and C.

A *stem based definition* (figure 3b) associates a name with the clade "of all organisms sharing a more recent common ancestor with one designated descendant than with another [designated descendant]" [2]. The shaded area could be defined as the maximum clade containing B but not A or the maximum clade containing C but not A.

An *apomorphy based definition* (figure 3c) associates a name with a clade "stemming from the ancestor in which a designated character [apomorphy] arose" [2].

The entities used in the definitions above (A, B, C and the apomorphy) are called specifiers. There are five types of specifiers, species, type specimens, specimens (other than a type), apomorphies (*i.e.* organismal traits) and previously established PhyloCode names (PhyloCode names are currently not allowed in the PhyloCode as specifiers but are anticipated to appear at some later revision).

A *species specifier* is a reference to a species that is defined in one of the current taxonomic systems. A *specimen specifier* refers to a an actual specimen of an organism stored in an institution or similar. In case the specimen also typifies a species it is called a *type specimen specifier*. An *apomorphy specifier* is a general description of the character that the apomorphy designates. Apomorphies can be defined in conjunction with a specimen or species in which the apomorphy is expressed.

1.3.2 - Name types

PhyloCode names are divided into three major categories. New names, converted names and replacement names. *New names* are those names that has no relation to any present names (either governed by the PhyloCode or any other taxonomic system). *Converted names* are those names that originate from a preexisting name from another taxonomic system and has been redefined in accordance with the articles of the PhyloCode. A *replacement name* is a name that is a substitution of a previously established PhyloCode name.

1.3.3 - Registering PhyloCode names

There are three major requirements that needs to be fulfilled for a name of a taxon to be established as a PhyloCode name. It must abide by the articles of the PhyloCode, it must also have been published in a scientific journal (or similar) and it must have been registered in the PhyloCode registration database (*RegNum*).

Once a name has been published in a scientific journal (or is about to be published) the author of a name uses the registration database interface to register the name. A database administrator subsequently inspects the registration application and either accept or rejects the application or return it to the author for clarification.

1.3.4 - Data requirements for registering a name in the database

The PhyloCode has a set of requirements on what data needs to be supplied to the registration database for a successful registration.

PhyloCode name		
Name to be registered	(mandatory)	
Type of name	(mandatory)	New name, converted name, replacement name
Date of registration	(mandatory)	
Definition type	(mandatory)	Node based, stem based, apomorphy based, other ...
Phylogenetic definition	(mandatory)	Must contain at least two specifiers.
List of specifiers		The specifiers mentioned in the phylogenetic definition (see below).
Preexisting name	(mandatory if converted name)	Preexisting name of another code that is converted to the name being registered (see below).
Replaced name	(mandatory if replacement name)	The PhyloCode name that is replaced by the name being registered.
Qualifying clause		
Reference phylogeny		Bibliographic reference, URL, or Accession number in public repository
Bibliographic reference	(mandatory)	Reference to publication where name is first defined
Date of publication		
Contact information		Contact information should be supplied for each author
Name	(mandatory)	
Mailing address	(mandatory)	
Phone number	(mandatory)	
Fax number		
Email address		
Homepage URL		
Authors comment		
Administrators annotations		

If the PhyloCode name being registered is a converted name a reference to the preexisting name it replaces must be made and the following data supplied.

Preexisting name

Preexisting name	(mandatory)	
Author	(mandatory)	
Bibliographic reference	(mandatory)	Reference to the original publication of the preexisting name.
Code	(mandatory)	The code that governs the name
URL		URL to taxonomic database holding information about the name

The specifiers of the phylogenetic definition need also be defined. As mentioned previously, there are five types of specifiers, species, specimens, type specimens, apomorphies and previously established PhyloCode names. In the case of PhyloCode name specifiers a reference to the name must be made. If the specifier is an apomorphy a description must be supplied. The data requirements for species, specimens and type specimens are more complex (see below).

Species specifier

Name	(mandatory)	
Author	(mandatory)	
Year of publication	(mandatory)	
Code	(mandatory)	The code that governs the name
URL		URL of taxonomic database holding information about the name

Specimen specifier

Repository institution	(mandatory)	
Collection data	(mandatory)	Location of specimen
Description	(mandatory)	

Type specimen specifier

Species name typified	(mandatory)
Author of species name typified	(mandatory)
Year of publication of species name typified	(mandatory)

2 - Method

Although the *RegNum* application is not a large-scale development project, attempts were made to use some of the current methodologies dealing with software creation. This chapter starts with a general overview of different design processes. It particularly explains the differences between the *iterative process,* used during the development of *RegNum,* and the *waterfall process*. Section 2.2 gives further detail on how the different phases should be implemented, and later in this paper, chapter 3 describes what was the result of the execution of each phase.

2.1 - Introduction to design processes

There are several ways of formalizing the activities performed by developers during the building of a software application. A software project usually undergo the same four phases during development. These phases have been named *requirements capture phase*, *analysis phase*, *design phase* and *build phase*. From the ArgoUML manual [6]:

1. **Requirements Capture**. This is where we identify the requirements for the system, using the language of the *problem domain*. In other words we describe the problem in the customer's terms.
2. **Analysis**. We take the requirements and start to recast them in the language of a putative solution - the *solution domain*. At this stage, although thinking in terms of a solution, we ensure we keep things at a high level, away from concrete details of a specific solution - what is known as *abstraction*.
3. **Design**. We take the specification from the Analysis phase and construct the solution in full detail. We are moving from *abstraction* of the problem to its *realization* in concrete terms.
4. **Build Phase**. We take the actual design and write it in a real programming language. This includes not just the programming, but the testing that the program meets the requirements (*verification*), testing that the program actually solves the customer' s problem (*validation*) and writing all user documentation.

No matter which development methodology is adopted, in the end, the project will have been taken through these four stages. What differs between methodologies is in what order the stages are performed and what emphasis is put on them.

There are two main methodologies in widespread use today. The *waterfall process* and

the *iterative process*. The *waterfall process* mandates that all phases are performed one after the other. In figure 4, the *Requirements capture phase* has to be finalized before we move on to the *Analysis phase.* Similarly for subsequent phases until we arrive at the final product ready for deployment.

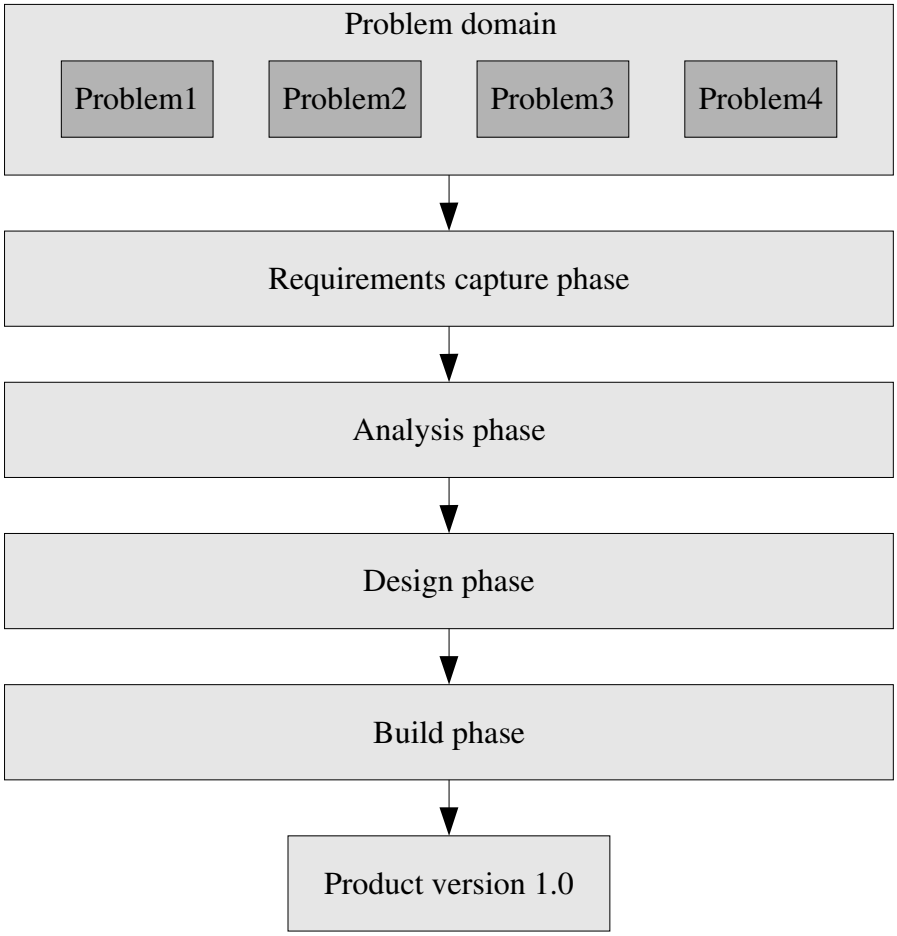

Figure 4: Graphical overview of the *waterfall process*. The entire problem domain is mapped out during *Requirements capture phase*. Subsequent phases are not initiated until the preceding phase has been completed.

This can be problematic for several reasons. It might be that we do not have a full grasp of the problem domain at the outset of the project. Additional requirements for the project often crop up once development has proceeded for a while. The customer might change his mind regarding certain functionality etc. If we demand that each of the steps above are executed sequentially it is very hard to adapt to changing requirements.

In contrast, the *iterative process* (figure 5) tries to limit the scope of each phase and let

them deal with only a part of the *problem domain*. This is beneficial for several reasons. First, it is much easier to solve small problems one at a time. Secondly, if some part of the process is stalling, that does not necessarily mean that the rest of the problems cannot be solved. Thirdly, if another set of problems in need of a solution are discovered, that does not affect the rest of the problem solving.

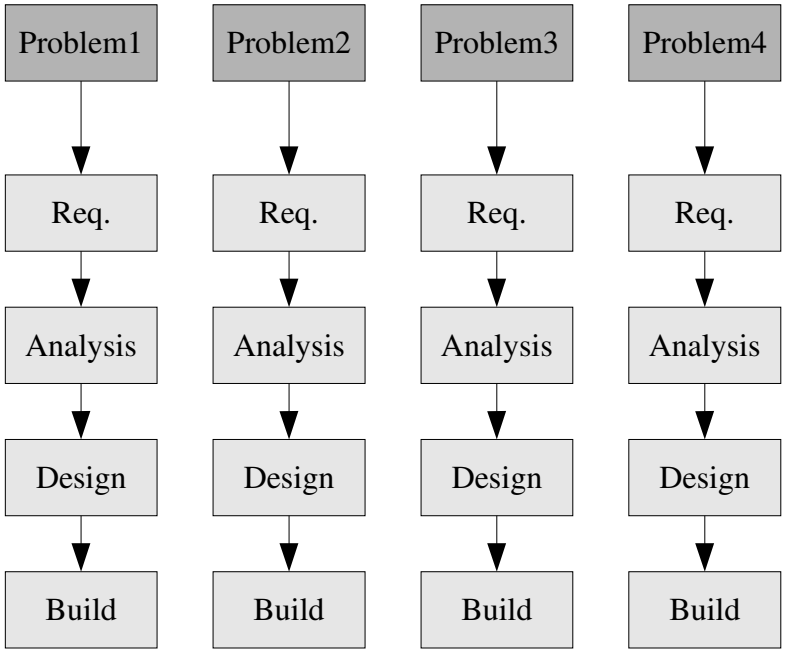

Figure 5: Graphical overview of the *iterative process*. Iterative development means solving one problem at a time.

Another benefit of *iterative programming* is that the software project can easily be divided into release cycles (figure 6). In the *waterfall* process there is only one product, the final release (called version 1.0 in the diagrams). The final release will solve all problems mapped out in the *Requirements capture phase*. In *iterative programming*, the customer and developers agree on what problems should be addressed at each release and a time table when each release is due. The end-result is that the customer has greater insight into the progress of the development. The idea of iterative development is one of the cornerstones of *Extreme Programming* [7]. In summary, the aim of *iterative programming* is to get at least part of the code up and running as quickly as possible.

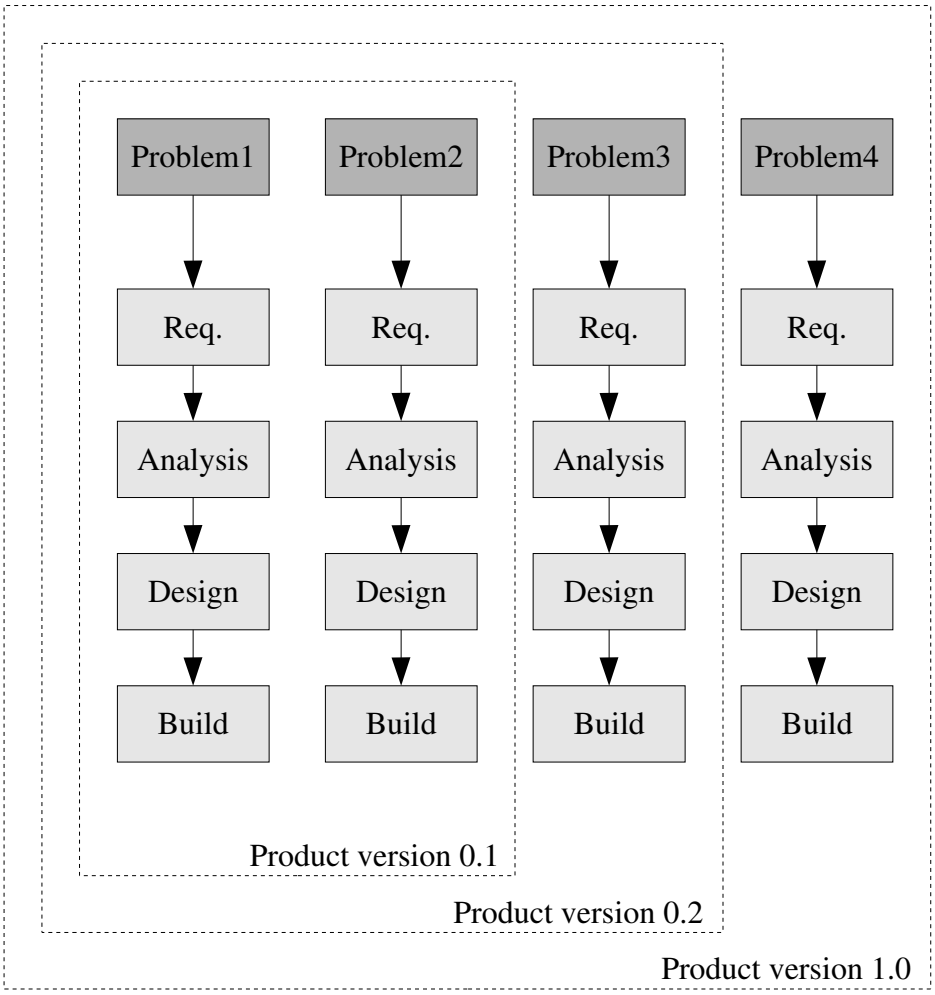

Figure 6: In the iterative design process several releases are made, each incorporating more functionality than the previous before the final product is released. Version 0.1 of the software in the graph above solves problem 1 and 2, Version 0.2 adds the solution to problem 3 and the final version 1.0 solves all four problems.

The design process for this project applies some of the principles of the *iterative process*. Not all aspects of *iterative development* were applicable. For example, as the development team consisted of only one person (me) many concerns relating to team communication and team dynamics did not apply. The scope of the problem domain was also such that it did not warrant all elements of each phase to be conducted.

2.2 - Design phases

2.2.1 - Requirements capture phase

Development projects usually has a customer, perhaps an employer or a company buying services from a developer. The customer in this project is the PhyloCode organization. The *requirements capture phase* is the process by which the customer and developer agrees upon, and documents, the problem domain in what is called a *Vision document*. This document need not, and indeed should not, describe potential solutions to the problem domain. It should only describe the problems the system should solve. The *vision document* is formalized into *UML use case diagrams* (see [8] for an extensive description of UML). These diagrams acts as an overview of the principal activities of the system.

2.2.2 - Analysis phase

During the *analysis phase* the focus is shifted towards potential solutions to the problem domain. The *use cases* are translated into *UML sequence diagrams* that explain, in further detail, the sequence of events during user interaction. At this point it is also possible to map out an overview of the data model. Such maps are transformed into *UML class diagrams*.

2.2.3 - Design phase

The design phase is a continuance of the *analysis phase*. The *sequence diagrams* and *class diagrams* are refined to the the point where it is possible to write code solely on the basis of what is in the diagram. Considerations such as what hardware and software tools are required to solve the problem are resolved.

2.2.4 - Build phase

During the *build phase* a solution is implemented based on the diagrams from the *design phase*. If the proposed solution is mapped out in enough detail, this is only a matter of converting the diagrams to code. Implementing the solution is not however only about writing code. The hardware, as well as the software, needs to be set up and configured. After the code has been written there should also be extensive testing of the code so that the customer can be assured of its quality.

3 - Result

This chapter contains the results obtained from each of the development phases described in chapter two. As mentioned previously this project used an iterative development process. The entire problem domain was not dealt with at once but rather specific problems were solved one at a time. However, for reasons of clarity the following sections describe the entire development process as if the waterfall process had been used.

3.1 - Requirements capture phase

The *RegNum database* should be a tool to be used by anyone wishing to register named clades in accordance with the principles put forth by the PhyloCode project. *Names* are submitted into the database in batches (*Submissions*) by a *Submitter*. An *Administrator* is responsible for accepting the *Submission* or return it to the *Submitter* for further clarifications.

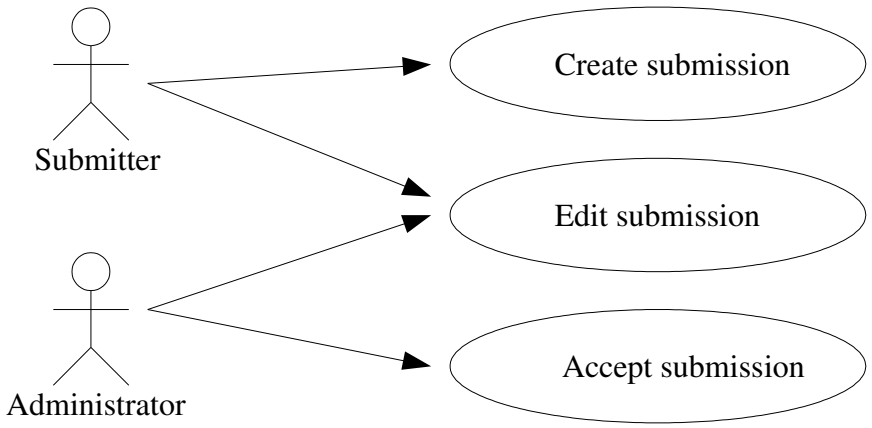

Figure 8: UML use case diagram for the *RegNum* application. A *Submitter* can create and edit a submission. An *Administrator* can edit and accept a submission.

Figure 8 shows a *use case diagram* depicting the actors of the system and the actions they can perform. A *Submitter* is able to create a submission and edit previously submitted but not established names. An *Administrator* is able to edit *Submissions* (correct errors etc.) as well as accepting *Submissions*. Note that an *Administrator* can also take the role of a *Submitter* (*e.g.* to make own submissions).

3.2 - Analysis phase

3.2.1 - General notes

Several obvious, but nonetheless important, observations of general nature can be made from the *use cases* above. First, they all have in common that they describe humans interacting with a computer system. From that follows that the application will need a user interface (a *view*). Secondly, the interaction between the users and the system is non-trivial, *i.e.* there is some application logic that controls the sequence of events (a *controller*). Thirdly, the applications primary function will be that of information storage and retrieval. The application will operate on a data *model*.

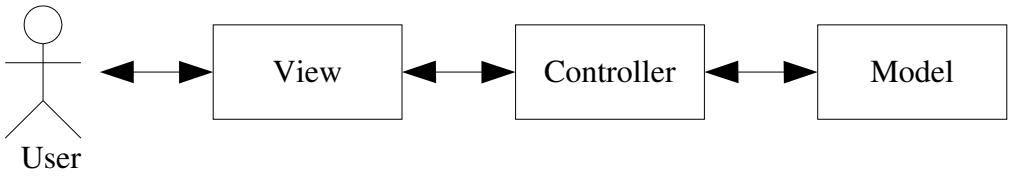

Figure 9: The Model-View-Controller pattern is central to the *RegNum* application. The graph above shows how a *User* acts upon a data model. The *View* is a description of what the web pages of the application will look like. The *Controller* contains the logic to enact the *Views* as well as the logic responsible for communicating with the data model.

Figure 9 summarizes the observations made above. The *Controller* receives input from the user through the user interface (*View*). It acts upon this input by operating on the data *model* (the information stored in the repository). Any feedback from the *Controllers* actions is communicated through the user interface.

In the following sections of the *analysis phase*, the implications of these general observations will be disregarded, as they should be considered implicit.

3.2.2 - Sequence diagrams

The use cases in figure 8 all pertain to the process of registering PhyloCode names. It therefore makes sense to translate them into one sequence diagram (figure 10).

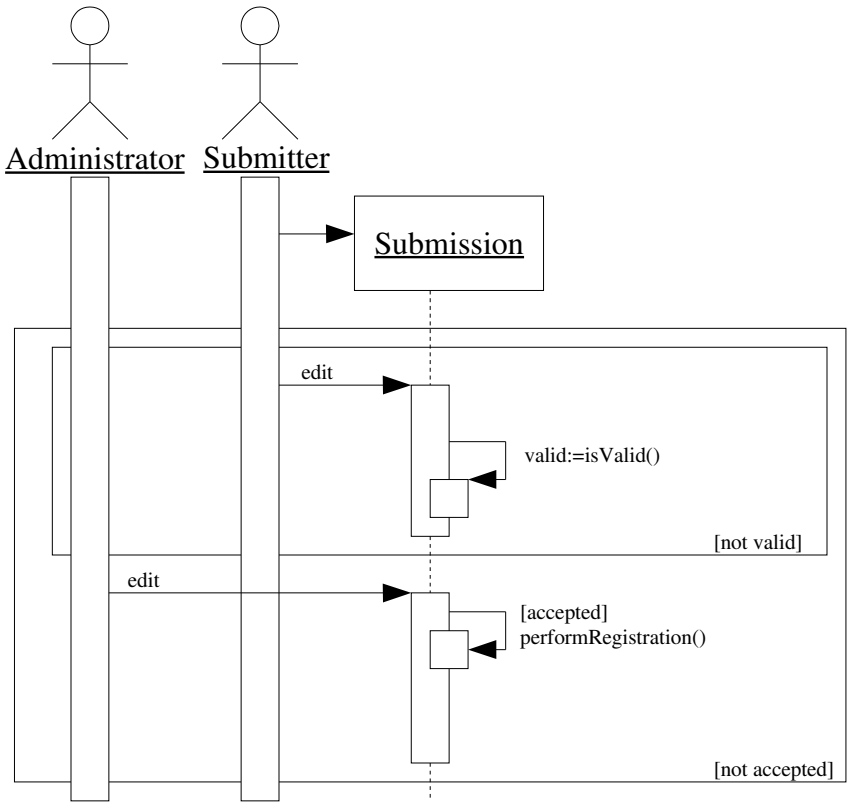

Figure 10: UML sequence diagram showing the interplay between a *Submitter* and an *Administrator* during the name registration procedure.

The diagram can be translated as follows:

1. A *Submitter* creates a *Submission* (a group of PhyloCode names to be registered).
2. The *Submitter* edits the *Submission*.
3. When the *Submitter* is finished editing, a validity check is performed on the *Submission* (see below for explanation of validity). If it is not valid, the editing process will continue until the validity test is passed.
4. When the *Submission* is valid the *Administrator* has the opportunity to edit the *Submission* and finally decide whether to accept the *Submission* or not.
5. If the *Submission* is not accepted the process will be taken back to step 2 where the *Submitter* can make alterations to the *Submission*.

The term *valid* deserves some explanation. A *Submission* is valid if all the PhyloCode names contained in it are valid. The application will regard a PhyloCode name as valid if all requirements of section 1.3.4 are fulfilled. A PhyloCode name being found valid by the application does not, however, imply that the name is valid for establishment as a proper PhyloCode name. That is up to the *Administrator* to decide.

3.2.3 - Class diagrams

The first consideration when modeling a large data structure is to determine to what extent a fixed structure needs to be imposed on the data.

A conscious decision was made early on during the *analysis phase* to limit the number of fixed data structures in the data model. A fixed data structure reduces the flexibility given to a user while entering data. For example, if a user wants to attribute a name to several people to varying degrees *(e.g.* "this name was primarily authored by NN1 with some help from NN2"). This type of information can not be stored easily in the repository unless it is modeled as a free-text string.

The primary function of the *RegNum database* is to store names and their definitions. It was therefore decided to model those entities as structured information. Other types of data (*e.g.* who submitted a name, citations, contact information and data not pertaining to the definition of a name) are modeled as free-text strings.

Table 1 lists the entities featured in the *RegNum* application and figure 11, on the next page, summarizes in a UML class diagram the relationships between these entities.

Object type	***Description***
User	A person interacting with the application.
Submission	Entity containing the names to be submitted.
PhyloCodeName	A PhyloCode name.
LegacyName	Taxon name registered outside of the PhyloCode (*e.g.* ICBN, ICZN and other codes).
NewName	A new PhyloCode name.
ReplacementName	PhyloCode name that replaces a previously established PhyloCode name.
ConvertedName	PhyloCode name that replaces a LegacyName.
Specifier	Entity used to define a PhyloCode name.
Specimen	Specimen of an organism.
Apomorphy	Organismal trait.

Table 1: A list of the entities that comprises the data model of the RegNum application.

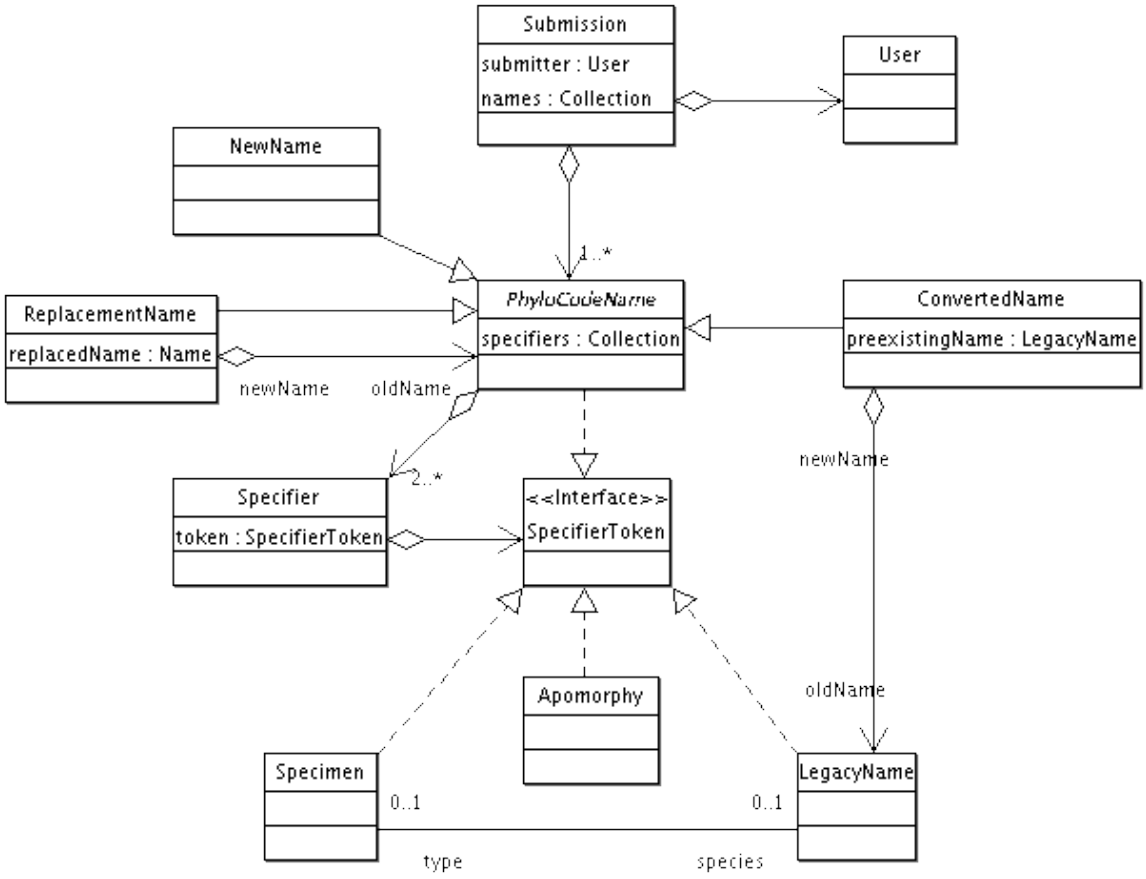

Figure 11: UML class diagram showing the relationships between entities of the *RegNum* application. Note that not all attributes and methods of the classes are shown.

- A *Submission* will refer to a *User* (the submitter).
- A *Submission* will refer to one or more instances of *PhyloCodeName*.
- *NewName*, *ConvertedName* and *ReplacementName* are subtypes of a *PhyloCodeName* .
- A *PhyloCodeName* will refer to two or more *Specifiers.*
- A *Specifier* points to a *PhyloCodeName*, *Specimen*, *Apomorphy* or *LegacyName*.
- A *ReplacementName* refer to a *PhyloCodeName* that it replaces.
- A *ConvertedName* refer to a *LegacyName* that it replaces.
- A *Specimen* may or may not refer to a *Species* that it typifies.
- A *LegacyName* may or may not refer to a *Specimen* that typifies it.

In the tables below is an account of all the properties of the object types in the previous class diagram (figure 11).

User	
Property name	***Comment***
id	
username	
password	
role	User or administrator.
firstname	
lastname	
email	

Table 2: Properties of a User object.

Submission	
Property name	***Comment***
id	
submissionDate	Date when the submission was submitted to the administrator.
registrationDate	Is set when the administrator accepts the submission.
names	Collection of all names contained in the submission.
user	The user that created the submission.

Table 3: Properties of a Submission object.

Specifier	
Property name	***Comment***
id	
placement	Internal or external.
description	Description of the specifier
token	PhyloCodeName, Specimen, LegacyName or Apomorphy that this specifier represents.

Table 4: Properties of a Specifier object.

PhyloCodeName	
Property name	***Comment***
id	
name	The actual name.
registrationNumber	Is set by the administrator when the name is established.
definitionType	Node based, stem based or apomorphy based.
definition	The definition as it appears in the publication of the name.
reference	Reference to publication where name is defined.
registrationDate	Date of registration. Only set if the administrator has accepted the name. Should be the same as the registration date of the submission.
contact	Contact information.
submitterComment	Comments made by the submitter.
adminComment	Comments made by the administrator.
referencePhylogeny	Information regarding where to find the phylogeny that was used when defining the name.
specifiers	Collection of Specifiers that defines the name.

Table 5: Properties of a PhyloCodeName object.

ReplacementName	
Property name	***Comment***
replacedName	Points to a PhyloCodeName that this name replaces.

Table 6: Properties of a ReplacementName object.

ConvertedName	
Property name	***Comment***
preexistingName	Points to a LegacyName that this name replaces.

Table 7: Properties of a ConvertedName object.

Specimen	
Property name	***Comment***
id	
description	Description of the specimen.
collection	Information needed to locate the specimen.
institution	The institution where the specimen is stored.
species	LegacyName that this Specimen typifies.

Table 8: Properties of a Specimen object.

LegacyName	
Property name	***Comment***
id	
name	The actual name.
author	The author of the name.
code	Code governing the name.
publicationDate	Publication date of the name.
reference	Bibliographic reference to original publication of the name.
url	URL to taxonomic database holding information about the name.
type	Specimen that typifies this LegacyName.

Table 9: Properties of a LegacyName object.

Apomorphy	
Property name	***Comment***
id	
description	Description of the apomorphy.

Table 10: Properties of an Apomorphy object.

3.3 - Design phase

3.3.1 - Selection of server software

Ideally, the design of an application should be agnostic to the tools used to develop, deploy and execute it. Nevertheless, current reality mandates that choice of software platform is considered relatively early in the development process.

It was decided that the application would be built as a web application residing in a Java servlet environment [9]. This was a choice made primarily based on personal preferences. Figure 12 shows an initial overview of the components required for the application. *RegNum* will be built upon functionality provided by a *Web application framework*. The framework is basically a collection of utilities that helps a developer in building web applications. The web application is contained within a *Servlet container* that handles communication between the web application and the client web browsers. Furthermore *RegNum* uses an *object/relational persistence manager* to communicate with a *relational database* where model data is stored. An *object/relational [O/R] persistence manager* is an application that converts information stored in database tables into Java objects.

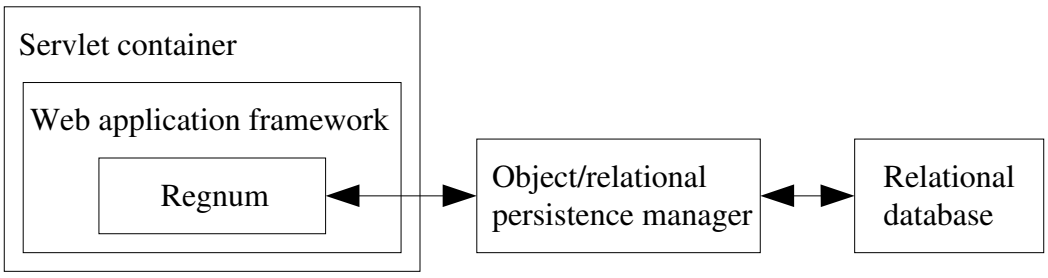

Figure 12: The software components that *RegNum* is built upon. *RegNum* will depend on functionality provided by a *Web application framework*. This framework lives inside a *Servlet container* that handles communication between the server and the client. Data will be stored in a *Relational database* and *RegNum* communicates with this database through an *Object/relational persistence manager*.

At the outset of the project it was decided that only open source tools be used. This decision was taken in order to facilitate a development environment where many people, on diverse hardware platforms, can contribute to the *RegNum* project. The tools and frameworks chosen for the *RegNum database* project are listed in table 11 together with some alternatives that were considered.

Software component	***Selected tool/framework***	***Alternatives considered***	***Motivation for selection***
Servlet container	Jetty [10]	Tomcat	Jetty is bundled with Apache Cocoon
Relational database	MySQL [11]	PostgreSQL	Developer had previous experience with MySQL
Object/relational persistence manager	Hibernate [12]	OJB	Hibernate and OJB has a similar set of features but currently the community behind Hibernate is far more active compared to OJB.
Web application framework	Apache Cocoon [13]	Struts	Apache Cocoon has a very rich set of utilities for most types of content generation (*e.g.* HTML, XML, PDF, SVG and other formats). It also features continuation based web programming.

Table 11: The table lists the software components that RegNum interacts with and what actual tool/framework was chosen.

3.3.2 - Sequence diagrams revisited

Having decided on the tool chain that will support *RegNum,* the sequence diagrams described in the previous sections could be refined.

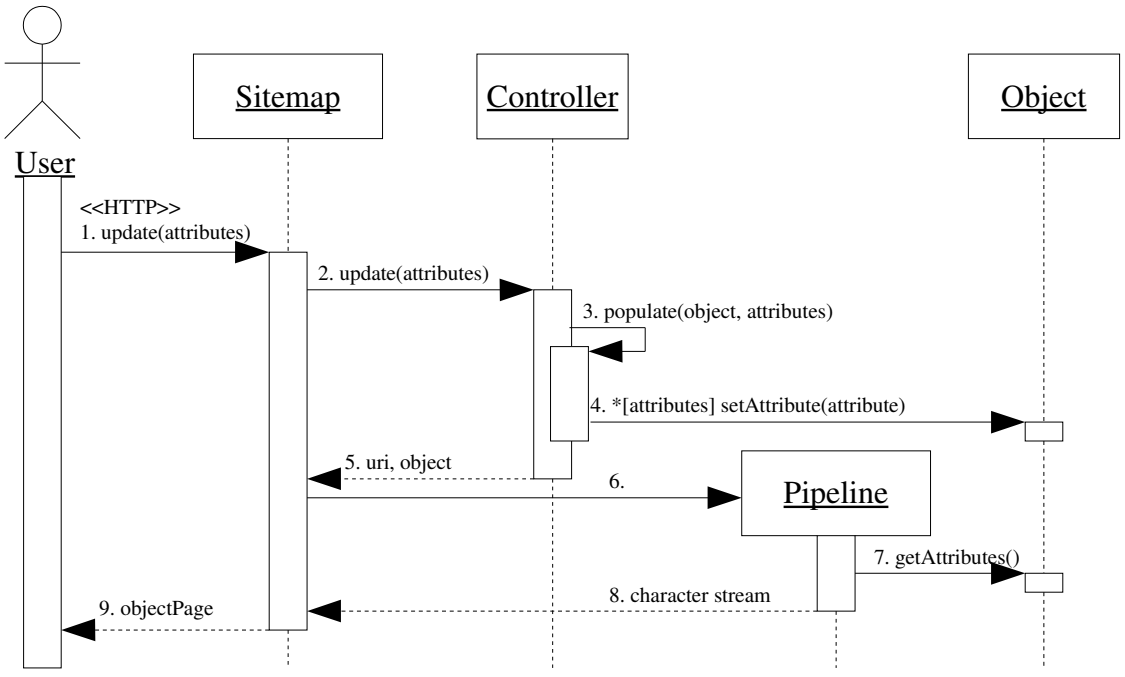

Figure 13: UML sequence diagram showing a *User* editing an arbitrary *Object*.

In general, the primary task of the *RegNum* application, is that of letting a user edit an object (*e.g.* a *Name* or *Specifier*). Figure 13 on the previous page shows how an *Object* is manipulated by a *User*. The *Object* in the diagram could be of any type shown in figure 11 (*i.e. User, Submission, NewName, ReplacementName, ConvertedName, Specimen, Apomorphy* or *LegacyName)*.

The sequence diagram describes what happens when a *User*, having just edited a web form containing attributes of an object, submits the form. The *Sitemap* [14] is an object, part of the Cocoon framework, that has two functions. First it matches the URI of the HTTP request and invokes the corresponding function of the *Controller.* The *Controller* is a function written in JavaScript and executed by the Apache Cocoon Flow [15] component.

The second function of the *Sitemap* is to assemble a SAX [16] component pipeline when the execution of the *Controller* function is finished. The composition of the pipeline is dependent on the URI that is provided in the sendPage() invocation. The pipeline creates a character stream that will eventually be sent to the *User* as a web page. Below is a walk through of the diagram:

1. The *User* sends a HTTP request, instructing the *Sitemap* to invoke the update() function in the *Controller.* The attribute values of the edited object are provided as HTTP POST request parameters.
2. The *Sitemap* invokes the update function in the *Controller* passing along the attribute values.
3. The *Controller* calls the populate() function with the attributes as parameters.
4. For each attribute provided, the corresponding setAttribute() function of the *Object* is called.
5. The *Controller* sends an URI (identifying what page to send to the *User*) as well as the object that is currently edited to the *Sitemap.*
6. The *Sitemap* assembles a *Pipeline* of SAX handling components according to the instructions in the *Sitemap* configuration file.
7. The *Pipeline* fetches the attributes of the *Object.*
8. A character stream containing a web page with an HTML form of the object is returned to the *Sitemap*.
9. The *User* is shown the web page constructed from the pipeline.

It is not enough to only enable the attributes of an object to be edited. There must also be a mechanism to add objects to one another. A typical *Submission* will consist of several *Names*, which in turn contains several *Specifiers*. Additionally checks must be made so that the data provided by the *User* is valid before the *Submission* is submitted to the *Administrator.*

Figure 14 on the next page details the interaction between a *Submitter* and the *Controller* when editing a *Submission*. Note that it omits the details of how the objects are edited (figure 13) and only deals with object creation and subsequent coupling.

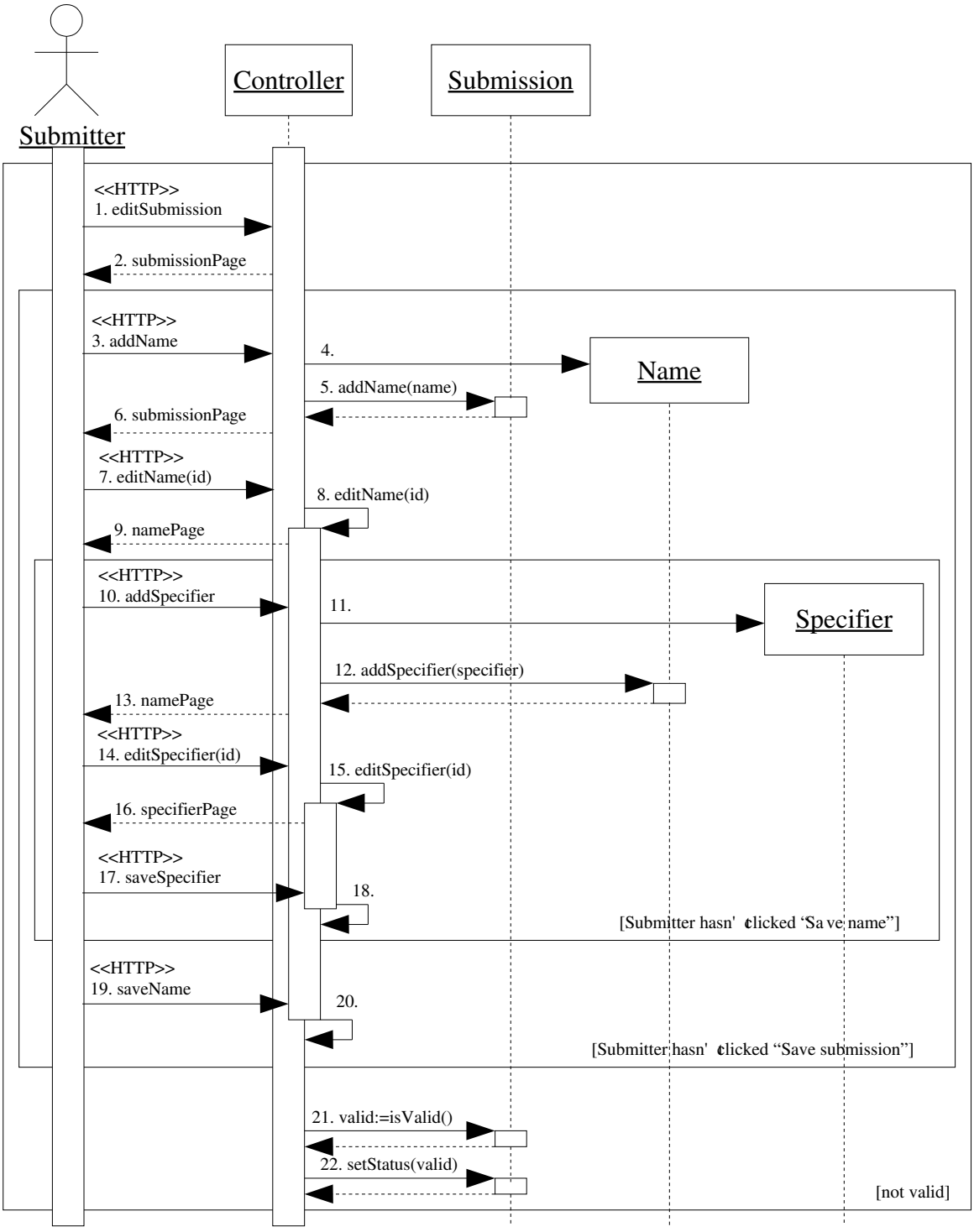

Figure 14: UML sequence diagram showing the process of editing a *Submission*.

Below is a walk through of the diagram in figure 14.

1. The *Submitter* requests the submission page.
2. The *Controller* returns a web page with a HTML form showing the attributes of the submission.
3. The *Submitter* instructs the *Controller* to add a *Name* to the *Submission*.
4. A *Name* object is created
5. The *Name* object is added to the *Submission*.
6. The *Controller* returns a web page with a HTML form showing the attributes of the *Submission* object. The web page will also show the added *Name* object.
7. The *Submitter* instructs the *Controller* to edit a *Name* object with a specific id.
8. The *Controller* calls function editName() with the id as parameter
9. The *Controller* returns a web page with a HTML form showing the attributes of the *Name* object.
10. The *Submitter* instructs the *Controller* to add a *Specifier* to the *Name*.
11. A *Specifier* object is created
12. The *Specifier* object is added to the *Name*.
13. The *Controller* returns a web page with a HTML form showing the attributes of the *Name* object. The web page will also show the added *Specifier* object.
14. The *Submitter* instructs the *Controller* to edit a *Specifier* object with a specific id.
15. The *Controller* calls function editSpecifier() with the id as parameter
16. The *Controller* returns a web page with a HTML form showing the attributes of the *Specifier* object.
17. The *Submitter* instructs the *Controller* to save the *Specifier* object.
18. The editSpecifier function is exited and control is returned to the editName() function.
19. The *Submitter* instructs the *Controller* to save the *Name* object.
20. The editName function is exited and control is returned to the editSubmission() function.

The process of editing a *Name* will continue as long as the *Submitter* does not choose to save the *Name*. Similarly for *Specifiers*.

The previous diagram in figure 14 presupposes the existence of a *Submission* object. It does not explain what actions are taken before and after the editing process. Before a *Submission* is edited it is loaded from the database. Afterwards it should be sent to the *Administrator* that will accept or reject the *Submission*. Figure 15 below shows the events prior to editing commences and how the process is finished.

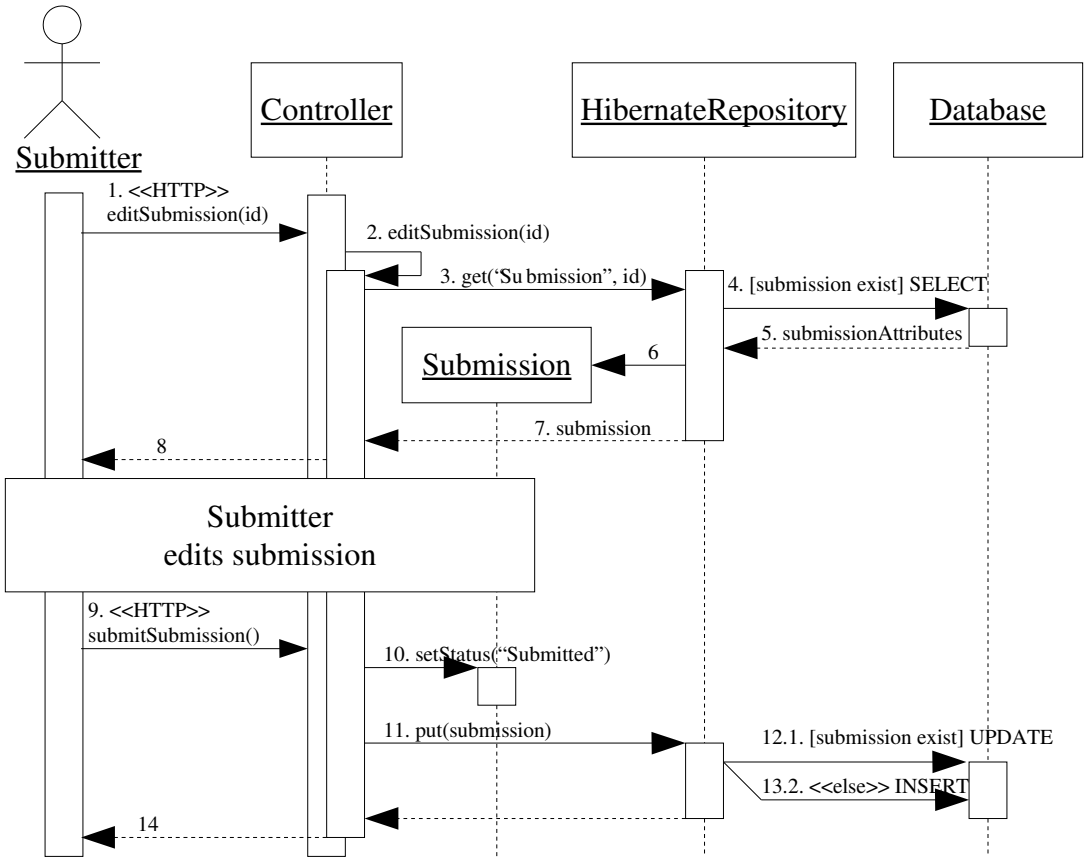

Figure 15: UML sequence diagram of the events taking place before and after a user edits a submission.

1. A *Submitter* instructs the *Controller* to execute the editSubmission() function. If the *Submitter* wants to edit an existing *Submission* an id of the submission is supplied.
2. The *Controller* calls the editSubmission() function and pass id as parameter.
3. The *Controller* requests a *Submission* object from the *HibernateRepository*. The id is passed along the request.

4. If an id was supplied from the *Submitter* (*i.e.* the submission exists) the *HibernateRepository* sends an SQL SELECT statement to the database, requesting all attributes of that *Submission*.
5. The attribute values of the *Submission* are returned.
6. A *Submission* Java object is created. If an existing submission was requested the object is populated with the values returned from the database request.
7. The *Submission* object is returned to the *Controller*.
8. The *Controller* sends a web page to the *Submitter* showing the *Submission.*
9. After the *Submitter* is finished editing the *Submission* it instructs the *Controller* to submit it.
10. The *Controller* sets the status flag of the *Submission* to "Submitted".
11. The *Controller* sends the *Submission* object to the *HibernateRepository* for storage.
12. If the *Submission* does not exist the *HibernateRepository* adds a new entry in the database. If it does exist, it merely updates the existing entry.
13. The *Controller* responds to the *Submitter* with a web page explaining that the *Submission* has been submitted.

3.3.3 - Hardware

Selection of hardware to house the application is normally considered during the design phase. It is only then, that enough is known to make a first estimate as to what requirements need to be met by the hardware. However, for this project the hardware was already provided, so no effort was made to investigate if the hardware is scaled to meet application demands. Additionally, little is presently known about what to expect in terms of server load which further complicates estimating hardware requirements.

Figure 16 on the next page shows the configuration of the two computers *Kvasir* and *Saga* dedicated to running the *RegNum* application. *Kvasir* runs the RegNum web application contained by Jetty while *Saga* houses the database where information is stored.

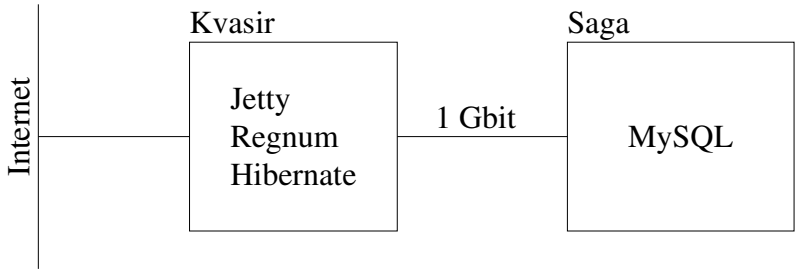

Figure 16: Hardware setup for the *RegNum* application during development. The computer *Saga* is dedicated to running the MySQL server while all other software components are housed on *Kvasir*.

3.4 - Build phase

3.4.1 - Directory overview

A directory structure to contain the *RegNum* application was set up (table 12). The names of the directories and files conform to standard practices used when developing Java applications using Ant [17, 18].

regnum/build.xml	Ant build file.
regnum/bin/	Miscellaneous Bash scripts.
regnum/build/	Directory where *RegNum* java files are built.
regnum/etc/	Miscellaneous configuration files.
regnum/lib/	External java libraries.
regnum/src/	*RegNum* source files.
regnum/tools/	Miscellaneous tools used to run *RegNum.*
regnum/webapp/	The *RegNum* webapp.

Table 12: Directory structure of the entire RegNum application.

The contents of regnum/webapp/ (table 13) is the application that will be deployed on the production server. Jetty is configured to consider regnum/webapp as the base directory for the *RegNum* web application. The directory consists of:

regnum/webapp/WEB-INF/	Apache Cocoon files.
regnum/webapp/flow/	JavaScript files that constitutes the *Controller.*
regnum/webapp/jxt/	JXTemplate files containing HTML form documents that constitutes the *View.*
regnum/webapp/resources/	Miscellaneous files sent verbatim to the client browser.
regnum/webapp/stylesheets/	XSLT-files used by Apache Cocoon to transform XML.

Table 13: Directory structure of the webapp specific files of the RegNum application

3.4.2 - The sitemap

Apache Cocoon listens to HTTP requests made by the client browser to the Jetty servlet engine. The actions taken by Cocoon upon receiving a request is configured in a file called sitemap.xmap. The sitemap [19] of *RegNum* simply declares that all incoming requests should invoke a function in the *Controller* (box 1).

```
<map:sitemap>
  ...
  <map:flow language="javascript">
      <map:script src="flow/util.js"/>
      <map:script src="flow/phylocode.js"/>
  </map:flow>

  <map:pipelines>
    <map:pipeline>
      <map:match pattern="*.func">
        <map:call function="public_{1}">
          <map:parameter name="id" value="{request-param:regnumID}"/>
        </map:call>
      </map:match>

      <map:match pattern="*.kont">
        <map:call continuation="{1}"/>
      </map:match>
      ...
    </map:pipeline>
  </map:pipelines>
</map:sitemap>
```

Box 1: Partial listing of regnum/webapp/sitemap.xmap showing the sitemap snippets responsible for invoking the Controller.

Requests matching the namespace "*.func" invokes a JavaScript function in regnum/webapp/flow/phylocode.js prefixed with "public_" while requests matching "*.kont" invokes a continuation of a previously called JavaScript function.

3.4.3 - The controller

As explained above the *Sitemap* invokes a function of the *Controller* in response to a request from the user. The *Controller* is a collection of JavaScript functions. In the listing on the next page (box 2) the function editSubmission() is shown. Note that some functionality has been removed.

```
function public_editSubmission() {

    authenticate();
    var submission = hibernateRepository.get("Submission", cocoon.request.id);
    var form = new RegnumForm(submission);

    while (true) {
        form.setModel(submission);
        cocoon.sendPageAndWait("editSubmission.form.page", {"form": form});

        switch(getCocoonAction()) {
            case "addName":

                form.populate(createPopulationMap());
                submission.addName(createBean("Name"));
                break;

            case "editName":

                 form.populate(createPopulationMap());
                 editName(submission.getName(getCocoonActionParameter()));
                 break;

            case "submitSubmission":

                 form.populate(createPopulationMap());
                 submission.status = "Submitted";
                 submission.submissionDate = (new Date()).toString();
                 RegnumBeanHelper.performSubmission(submission);
                 hibernateRepository.put(submission);
                 cocoon.sendPage("home.document.page", {});
                 return;
             ...
        }
    }
}
```

Box 2: Partial listing of the editSubmission() function in regnum/webapp/flow/phylocode.js

It is out of scope for this paper to delve too deeply into the inner workings of this function. Suffice to say that the function starts by authenticating the user. Then fetches a *Submission* from the Hibernate repository and finally goes into a loop that continuously presents the user with a HTML form where the user can edit the *Submission*. The loop will end when the *Submission* is submitted.

Similar functions exists for all the object types listed in figure 11. Table 14 on the next page contains a cursory explanation of each function of the *Controller*.

Function name	*Comment*
public_editUser()	Controls a HTML form of a *User* object. Is used to edit both existing and new users.
public_loginUser()	Controls a login page.
public_logout()	Logs out a user.
public_editSubmission()	Controls HTML form of a *Submission* object.
editName()	Controls HTML form of a *Name* object. Called from public_editSubmission().
editSpecifier()	Controls HTML form of a *Specifier* object. Called from editName().
public_select()	Controls HTML form that enables a user to select an entity. The type of entity that can be selected depends on in which context the function is called. If called from editSpecifier() the user can select from a list of PhyloCodeNames, LegacyNames, Specimens or Apomorphies. If the function is called from editName() the selection is between a number of PhyloCodeNames instead.
public_listSubmissions()	Controls a HTML form listing all submissions made by the user.
public_listNewSubmissions()	Controls a HTML form to be viewed by an *Aministrator* showing all submissions that users have submitted for registration.
authenticate()	Helper function that determines whether a user has logged in or not. If not logged in, presents the user with a login form.

Table 14: The functions contained in regnum/webapp/flow/phylocode.js

3.4.4 - The model

As explained previously all information in *RegNum* is stored in a MySQL database and the *Controller* communicates with the database through Hibernate [12]. Hibernate is a Object/relational persistence manager. Put simply, Hibernate converts rows in a database table into Java objects.

Hibernate is configured by the file regnum/src/regnum.hbm.xml that maps out the entire data model of *RegNum*. In box 3 on the next page is a small snippet of regnum.hbm.xml that shows the declaration of the *Submission* object. Similar configuration is made for all objects in figure 11 (*i.e. Submission, User, PhyloCodeName, NewName, ConvertedName, ReplacementName, Specifier, Specimen, LegacyName* and *Apomorphy).*

```
<hibernate-mapping>
  ...
  <class name="org.phylocode.regnum.beans.Submission" table="submission">
    <id name="id" type="string">
      <generator class="assigned"/>
    </id>

    <property name="submissionDate" type="string"/>
    <property name="registrationDate" type="string"/>
    <property name="status" type="string"/>

    <bag name="names" cascade="all">
      <key column="submissionId"/>
      <one-to-many class="org.phylocode.regnum.beans.Name"/>
    </bag>
  </class>
  ...
</hibernate-mapping>
```

Box 3: Partial listing of regnum/src/regnum.hbm.xml showing the declaration of the Submission object.

The snippet in box 3 tells Hibernate that submissions has certain properties (id, submissionDate, registrationDate, status and names). The *names* property is configured as a bag (a Java Collection) of several *Name* objects. It also tells Hibernate that the object should be mapped to the table "submission" in the database. When building the *RegNum* application this configuration file is used to automatically generate the database schema as well as the Java objects configured. For example the snippet above will result in the schema and Java source shown in box 4 and 5.

```
package org.phylocode.regnum.beans;

// imports ommited

public class Submission implements Serializable {

    private String id;
    private String submissionDate;
    private String registrationDate;
    private String status;
    private Collection names;

    // getters and setters ommited
}
```

Box 4: Partial source code listing of org.phylocode.regnum.beans.Submission generated from regnum.hbm.xml

```
create table submission (
  id VARCHAR(255) not null,
  submissionDate VARCHAR(255),
  registrationDate VARCHAR(255),
  status VARCHAR(255),
  primary key (id)
)
```

Box 5: Database schema for submission table generated from regnum.hbm.xml

The *Controller* connects to Hibernate using the java helper class *HibernateRepository*. It has functions for loading and saving specific objects as well as searching for a collection of objects that conform to some specified criteria.

3.4.5 - The view

The view is implemented using JXTemplate [20]. JXTemplate is a templating language that consists of tags that can be intermingled in normal web pages. The primary purpose of JXTemplate tags is to render values from a supplied data model to the resulting web page. The data model is usually a JavaBean (in the case of *RegNum* a *Submission* object for example). When the *Controller* instructs the *Sitemap* to send a page to the user, it supplies an object that the JXTemplate tags will extract values from.

Box 6 on the next page contains a snippet from regnum/webapp/jxt/editName.jxt. The page consists of HTML markup as well as some JXTemplate sections. For example the text "${model.name}" instructs the JXTemplate component to fetch the name property from the model object. The model in this case being the *Name* object supplied by the *Controller*.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<rd:document
  xmlns:rd="http://www.phylocode.org/regnum/document/1.0"
  xmlns:rf="http://www.phylocode.org/regnum/form/1.0"
  xmlns:jx="http://apache.org/cocoon/templates/jx/1.0"
  xmlns="http://www.w3.org/1999/xhtml"
  >
  ...
  <form action="${continuation.id}.kont">
    <h1>Edit PhyloCode name</h1>

    <div class="container">
      <p>
        <b>Name:</b>
        <input type="input" name="name" value="${model.name}"/>
      </p>

      <p>
        <b>Registration number:</b>
        <jx:out value="${model.registrationNumber}"/>
      </p>
      ...
    </div>
    ...
  </form>
</rd:document>
```

Box 6: Partial listing of regnum/webapp/jxt/editName.jxt

Table 15 lists all JXTemplate files of the *RegNum* application.

File	***Comment***
regnum/webapp/jxt/editSubmission.jxt	HTML form of a *Submission* object.
regnum/webapp/jxt/editName.jxt	HTML form of a *PhyloCodeName* object.
regnum/webapp/jxt/editSpecifier.jxt	HTML form of a *Specifier* object.
regnum/webapp/jxt/editUser.jxt	HTML form of a *User* object..
regnum/webapp/jxt/loginUser.jxt	HTML form of a *User* object. Used when logging in.
regnum/webapp/jxt/select.jxt	Presents the user with a list of objects to be selected from. The list of objects can either be specifier tokens, legacy names that a *ConvertedName* replaces or PhyloCode names that a *ReplacementName* replaces.
regnum/webapp/jxt/listSubmissions.jxt	Presents a list of submission. If presented to an administrator the list of all new submissions or for a normal user the list of his own submissions.

Table 15: List of all JXTemplate files that constitues the View in RegNum.

4 - Discussion

In July of 2004 the first international meeting on phylogenetic nomenclature will be hosted in Paris and one of the events of the summit is the inaugural meeting of the International Society for Phylogenetic Nomenclature (ISPN). The ISPN will oversee the formation of a registration committee that is responsible for the deployment and administration of the registration database.

Leading up to the Paris summit the *RegNum* registration database will undergo extensive user testing. The arrangements for this testing has yet to be decided.

At present there are several issues that need to be dealt with prior to the final testing phase can be initiated. I have tried to outline them below in no particular order.

1. The data model designed for this project did not take into account how to register species names. The reason being that at present it is still undecided in the PhyloCode how they should be handled. It is foreseen that major reconstruction work on the data model will need to take place when and if the PhyloCode working group comes to agreement on the issue of species.

2. As mentioned previously (section 3.2.3) a conscious decision was made to limit the number of fixed data structures, at least during this preliminary stage of developing the data model. It might be beneficiary to review the data model as a whole at some later stage to see whether some data can be locked down into a more rigid structure. As an example, the author of a name could be modeled as a separate object to which each name refers, rather than as currently only supplied as free text.

3. Apomorphies are currently not modeled exactly as specified by the *PhyloCode*. The *PhyloCode* allows a registrar to define an apomorphy as a character with an additional reference to a *Species, PhyloCodeName* or a *Specimen* in which this character is expressed. The data model does not currently support this kind of reference to be made other than as part of the free text description of the apomorphy.

4. An absolute requirement for a production ready *RegNum* application is that all actions performed by users and administrators be logged in some fashion. Currently there are no such mechanisms in place. Ideally the ability to roll back transactions should also be implemented.

5. The choice of form handling framework in *RegNum* has been the largest stumble block, by far, during the development process. The initial plan was to use Apache Cocoon Woody [21], a new component of Cocoon that is aiming to be the official form framework for Cocoon. However, at the time of development for the *RegNum* project, there were several issues with Woody that severely limited its usability in *RegNum*, so other form frameworks were considered. In the end a combination of JXTemplate and several in-house developed components were used as the form

framework. Since then, several improvements has been made to Woody. It would be beneficial for the *RegNum* application if the current form framework is scrapped entirely and instead be based on the upcoming Apache Cocoon Form component that is the successor of Woody. Specifically, this involves converting the forms currently written in the JXTemplate markup language to the Cocoon Form markup language. Also the functions written in JavaScript would have to be implemented as event handlers which are then called by Cocoon Forms when certain buttons are pressed. *E.g.* the functionality of adding a name to a submission can be written as an event handler registered on the "Add n ame" button within the submission form.

6. Several of the entities that make up a name registration, such as preexisting names, reference phylogenies, bibliographical references etc. is information that is readily available from external data sources. It would be interesting to investigate the possibilities of incorporating mechanisms to use data directly from these sources rather than requiring name submitters to enter the data by hand. This would involve creating user interfaces that could interact with these data sources directly. The data model could then be adjusted to contain only references to these sources and the information would then be acquired dynamically when requested.

7. Several ideas has emerged on what to do when a fully functioning registration database has been deployed. There will be a need to implement a search interface where users are presented with all data pertaining to the name searched for. Another important functionality would be to allow a user to supply a phylogeny and then map out the PhyloCode names of the database at the appropriate places in the phylogeny.

8. The exact arrangements for how the *RegNum* application should be deployed, in particular where and on what machines has not been decided. It has therefore been difficult to make decisions on backup procedures and general administration issues. Several such issues need to be resolved before the initial deployment sometime next year.

5 - Acknowledgments

First of all I would like to thank my supervisor Mikael Thollesson at the Department of Molecular Evolution in Uppsala for the help and support he has given me throughout the project. The help from Torsten Eriksson at Kungliga Vetenskapsakademien during the initial phases of the project is also much appreciated. I would also like to thank Kevin de Queiroz for taking the time to look at the final *RegNum* application.

6 - References

[1] PhyloCode.
http://www.phylocode.org (16 Dec. 2003)

[2] DE QUEIROZ, K., AND J. GAUTHIER. 1992. Phylogenetic taxonomy. Annu. Rev. Ecol. Syst. 23:449-480.

[3] International Commision on Zoological Nomenclature. 1985. International Code of Zoological Nomenclature. London: International Trust for Zoological Nomenclature.

[4] DE QUEIROZ, K., AND J. GAUTHIER. 1990. Phylogeny as a central principle in taxonomy: Phylogenetic definitions of taxon names. Syst. Zool. 39:307-322.

[5] DE QUEIROZ, K., AND J. GAUTHIER. 1994. Toward a phylogenetic system of biological nomenclature. Trends Ecol. Evol. 9:27-31.

[6] ArgoUML manual.
http://argouml.tigris.org/documentation/defaulthtml/manual/ (16 Dec. 2003)

[7] Extreme programming.
http://www.extremeprogramming.org (16 Dec. 2003)

[8] Unified Modeling Language version 1.5
http://www.omg.org/technology/documents/formal/uml.htm (16 Dec. 2003)

[9] Servlet essentials.
http://www.novocode.com/doc/servlet-essentials/ (16 Dec. 2003)

[10] Jetty Java HTTP Server.
http://jetty.mortbay.org/jetty/ (16 Dec. 2003)

[11] MySQL.
http://www.mysql.com (16 Dec. 2003)

[12] Hibernate.
http://www.hibernate.org (16 Dec. 2003)

[13] Apache Cocoon.
http://cocoon.apache.org/2.1/index.html (16 Dec. 2003)

[14] Understanding Apache Cocoon.
http://cocoon.apache.org/2.1/userdocs/concepts/index.html (16 Dec 2003)

[15] Apache Cocoon Flow.
http://cocoon.apache.org/2.1/userdocs/flow/index.html (16 Dec. 2003)

[16] SAX.
http://www.ibiblio.org/xml/books/xmljava/chapters/ch06.html (16 Dec. 2003)

[17] Jakarta Ant.
http://ant.apache.org/index.html (16 Dec. 2003)

[18] Apache Wiki: The ElementsOfAntStyle.
http://wiki.apache.org/ant/TheElementsOfAntStyle (17 Mar. 2004)

[19] The sitemap.
http://cocoon.apache.org/2.1/userdocs/concepts/sitemap.html (16 Dec. 2003)

[20] JXTemplateGenerator.
http://cocoon.apache.org/2.1/apidocs/org/apache/cocoon/generation/JXTemplateGenerator.html (16 Dec. 2003)

[21] CocoonWiki: Woody.
http://wiki.cocoondev.org/Wiki.jsp?page=Woody (26 Dec. 2003)