

**Migrating DELTA descriptions of  
*Elaeocarpus* of Malesia to Padme and  
publishing them on the web**

**Yiwei Dong**

August 2020

School of Biological Sciences,

University of Edinburgh

&

Royal Botanic Garden Edinburgh

Thesis submitted in partial fulfilment for the MSc in the Biodiversity and Taxonomy of Plants



THE UNIVERSITY  
of EDINBURGH



Royal  
Botanic Garden  
Edinburgh

## Abstract

*Elaeocarpus* is a genus of family Elaeocarpaceae, comprising 350-400 species. Mr Coode spent decades on this genus and stored all information about Malesian *Elaeocarpus* items in a DELTA file. The DELTA file is easily read by computer programs but difficult to understand by people. There is detailed information about structures, characters, states, and items in it, but it is difficult to compare different datasets and identify an uncertain specimen. The Padme system could solve those problem brought by DELTA file. It stores information about plants and builds the interactive key for them, which would make information simple for users to understand and identify. This project has created a Padme database to store all *Elaeocarpus* data, translated DELTA file into Padme with PHP and MySQL script, and published them in a website. The Padme system has also created an interactive key for *Elaeocarpus* automatically, which would help the identification easier and more flexible than the traditional key.

## Acknowledgements

I would like to sincerely thank Martin Pullan, Mark Newman and Mark Coode for inviting me to work on this project, supervising me throughout my work and transmitting their knowledge. Although the special Covid-19 situation makes us work at home, all staff of the Royal Botanic Garden Edinburgh still provided me with plenty of materials, instructions, and supports. Thanks so much for all help and support.

I am extremely grateful to Martin Pullan, for having introduced me to computer programming, getting me started with various software, helping me make sense of scripting and providing help through the whole project.

To Mark Newman, a very warm thanks for giving me a huge head start on the dissertation writing and the kind help on language, barriers to access to literature and using the interactive key.

I thank Mark Coode for his kind sharing of *Elaeocarpus* DELTA file, literature and giving me the expert's opinion on *Elaeocarpus* taxonomy and identification.

I appreciate the kind and helpful sharing about Xper 3 from Rebecca Hilgenhof.

I thank Louis Ronse De Craene, Mark Hughes and David Harris for all helps during my quarantine time and returning travel.

## Table of contents

Abstract.....	2
Acknowledgements.....	3
Table of contents.....	4
List of Figures.....	6
1. Introduction.....	7
1.1 Overview of <i>Elaeocarpus</i> taxonomy.....	9
1.2 Electronic encoding of descriptive information.....	10
1.3 Differences between DELTA and Padme.....	12
1.4 Migrating scripting PHP and relational database MySQL.....	14
2. Software preparation.....	15
2.1 Software installation.....	15
2.2 software initialization and configuration.....	16
3. Padme preparation: creating <i>Elaeocarpus</i> records in Padme.....	19
4. PHP Script file.....	24
4.1 Deltaparser.php.....	24
4.2 <i>Elaeocarpus</i> .php.....	25
5. Error trapping and handling.....	27
5.1 Throwing exceptions.....	27
5.2 Try catch for error handling.....	28
6. DELTA file input.....	29
6.1 Read Data.....	29
6.2 Test readData().....	29
7. Connect to and Initialise the Padme MySQL database.....	31
7.1 Connect to the Padme database.....	31
7.2 Initialise the setting of MySQL padme database.....	32

8. DELTA file navigation .....	35
9. Reading in the character types .....	37
10. Read a character .....	40
10.1 Read a character description .....	40
10.2 Test method readNextCharacter() .....	43
11. Parse all characters and store them in Padme database .....	44
12. Character class .....	49
13. Read a description.....	50
14. Parse all descriptions and store them in Padme database .....	53
15. Test and modify the interactive key .....	57
15.1 Manual checking .....	57
15.2 Code checking.....	63
15.3 Problems and solutions .....	64
16. Discussion and conclusion.....	67
16.1 Project summary .....	67
16.2 Advantages of the interactive key over the traditional key.....	67
16.3 Future work .....	68
16.3.1 Completing the definition of each character and state .....	69
16.3.2 Adding multiple records into the Padme system.....	69
16.3.3 Creating a testing system for the interactive key.....	70
References.....	71
Appendixes .....	73
Appendix 1. deltaparser.php .....	73
Appendix 2. elaeocarpus.php .....	86
Appendix 3. tester.php .....	87
Appendix 4. elaeocarpustest.php .....	96

## List of Figures

Figure 1. The distribution of <i>Elaeocarpus</i> ( <a href="https://www.gbif.org/species/3152094">https://www.gbif.org/species/3152094</a> ).	9
Figure 2. A simple triple.	11
Figure 3. Chaining a pair of triples.	11
Figure 4. The example of one whole specimen tree.	11
Figure 5. The SUBJECT-PREDICATE-OBJECT model of Padme system.	13
Figure 6. The process of setting the environment variable.	15
Figure 7. The repository of <i>Elaeocarpus</i> in the GitHub.	17
Figure 8. The contents of Trunk folder.	18
Figure 9. Padme main menu (choose Latin name option).	19
Figure 10. Padme data entry section of the Latin names form (Fields to be filled are in cycles).	20
Figure 11. All sections and species are under Genus <i>Elaeocarpus</i> .	21
Figure 12. Species of <i>Acronodia</i> are under Section <i>Acronodia</i> and subspecies are under their corresponding species.	21
Figure 13. The record of <i>Elaeocarpus inopinatus</i> .	22
Figure 14. The logical diagram of error trapping.	28
Figure 15. The tables of the Padme <i>Elaeocarpus</i> database.	34
Figure 16. The logical diagram of the method <code>getDirectiveData()</code> .	39
Figure 17. The logical diagram of the method <code>readNextCharacter()</code> .	43
Figure 18. The logical diagram of the method <code>parseCharacter()</code> .	48
Figure 19. The logical diagram of the method <code>readNextDescription()</code> .	52
Figure 20. The logical diagram of the method <code>parseDescription()</code> .	56
Figure 21. <i>Elaeocarpus myrtoides</i> specimen	58
Figure 22. The process of identifying <i>Elaeocarpus myrtoides</i> with the interactive key	58
Figure 23. <i>Elaeocarpus petiolatus</i> specimen	59
Figure 24. The process of identifying <i>Elaeocarpus petiolatus</i> with the interactive key	60
Figure 25. <i>Elaeocarpus stipularis</i> var. <i>nutans</i> specimen	61
Figure 26. The process of identifying <i>Elaeocarpus stipularis</i> var. <i>nutans</i> with the interactive key	61
Figure 27. <i>Elaeocarpus cumingii</i> specimen	62
Figure 28. The process of identifying <i>Elaeocarpus cumingii</i> with the interactive key	62

## 1. Introduction

Plant identification is the process of comparing different characteristics and assigning the specific plant to a known taxon to get a species or subspecies name. Plant identification works by having a system and principles, and many such systems have existed before the one devised by George Bentham and Joseph Dalton Hooker in their *Genera Plantarum* (Bentham and Hooker, 1862). The main methods of plant identification are single-access keys (dichotomous key), where the sequence and structure of identification steps is established by the authors (Winston, 1999) and multi-access (interactive) keys which help users freely choose the characteristics that are suitable and convenient to identify the plant (Pankhurst, 1991; Winston, 1999). The single-access key is efficient and reliable. Authors usually created single-access keys to work with the most important, dominant, and easy to observe traits, resorting only when necessary to those that are difficult to observe or easily lost from specimens. Therefore, the single-access key is user-friendly and balanced. In the real field work, however, it may be hard to determine all taxa in different kinds of situations, because the identification would lead the wrong way if the data of one given identification step is not available or misleading. The most efficient solution to this problem is the use of multi-access keys. Unlike single-access keys printed on paper, most interactive keys are based on computer and internet. These are easy to update with new species and characters, and also easy to repost with new information on the internet (Brasher, 2006). Since the early 1970s, the study of plant identification with computer programs started (Morse, 1974), and it is an important trend in modern plant taxonomy.

Taxonomists create classifications of plants based on complex analysis of observations including morphology, molecular structure, and surroundings (Allkin, White and Winfield, 1992). With the work of taxonomists and the continuous improvement of methods of plant taxonomy, plant classification has become clearer and more accurate in recent years.

Expectations of plant taxonomy have changed, from a single desire for a better understanding of specific plant classification to a desire to make better and more effective use of previous research (Pullan et al., 2005). Therefore, taxonomists are also expected to provide complete information about plants to other biologists and people who are not professional but interested in plants (Allkin, White and Winfield, 1992). Although the classification of most of

the global plants and their relationship has been largely understood, there are still too few systematic databases which store taxonomic information in digital form and create a flexible key. There is much information which should be included in databases such as geographical information, references, numeric characters, descriptive characters and state data, and too little supporting software to manage these data. Herbaria and those places with large collections of specimens have the greatest need to create electronic databases (Diederich, 1997), to store specific items with complete information including characters, states, geography, references, researchers and accurate relation of dependence. The importance of storing, managing, and sharing taxonomic data is therefore apparent. Platforms to store and share the taxonomic information and help to identify with computers have been popular with taxonomists such as Xper2 and Xper3 which are versatile web platform for descriptions management and interactive identification (Ung *et al.*, 2010; Vignes-Lebbe *et al.*, 2017). The Royal Botanic Garden Edinburgh (RBGE) herbarium is among the world's larger botanical collections housing more than 3 million specimens (<https://www.rbge.org.uk/science-and-conservation/herbarium/>) and has a database containing the specimens, but a taxonomic system like Xper3 is still lacking. An electronic taxonomic management system constructing interactive key with complete data would be useful to the RBGE herbarium and be helpful to non-specialists who want to know more about plants.

This project is going to transfer electronic keys of members of *Elaeocarpus* to the Padme system and publish them. Padme system belongs to RBGE and is similar to Xper3, but users can edit and share their information online with Xper 3 while Padme aims to publish the correct and complete description according to the authoritative classification and users can use a more accurate interactive key. The Padme system is mainly used to help users identify plants with the interactive key created by Padme instead of editing the taxonomical descriptions, which is more suitable for this project. The keys of *Elaeocarpus* are in DELTA format constructed by Mr. Mark Coode. The descriptive information of Padme is different from DELTA, so scripts (PHP and MySQL) will be needed for both the data preparation process and to import the data into Padme. The following tasks will be achieved in this project:

- (1) Create a new Padme instance of *Elaeocarpus* inserting the Latin names of *Elaeocarpus* members into the Padme database.
- (2) Edit the DELTA data and bring it into the Padme system.

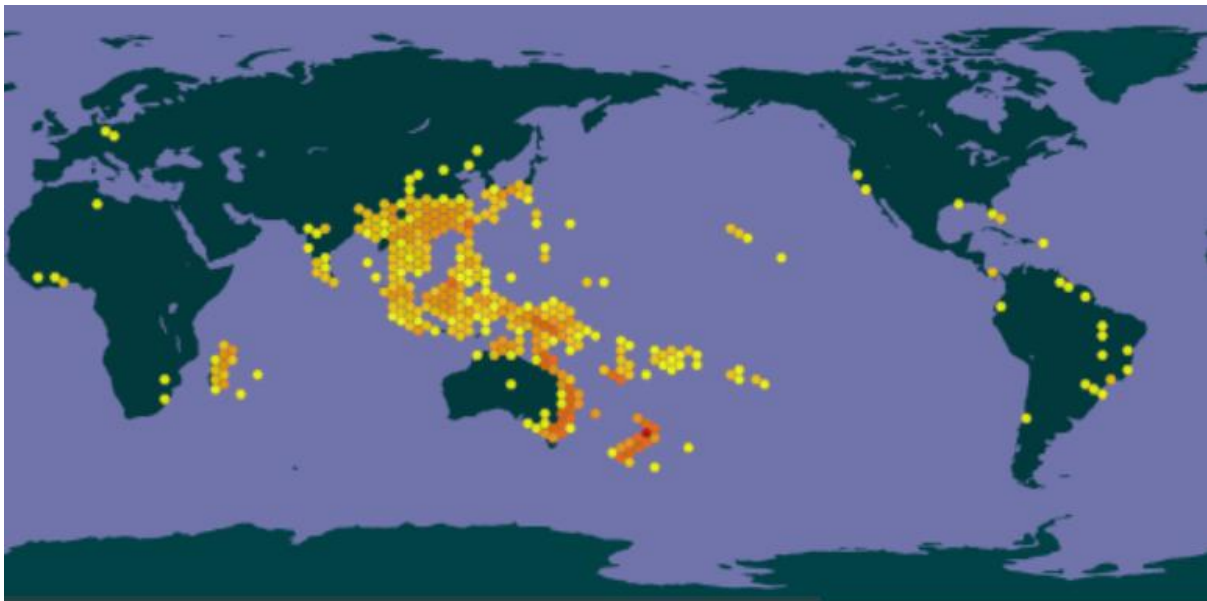


(3) Set up a new Padme web site instance publishing the interactive keys.

(4) Test the interactive key and modify the scripting.

### 1.1 Overview of *Elaeocarpus* taxonomy

Elaeocarpaceae is a flowering plant family of 12 genera and about 550 species, and is distributed predominantly in tropical forests except for mainland Africa (Coode, 2004). There have been many pieces of research estimating lineage divergence times for Elaeocarpaceae (Wikstro, Savolainen and Chase, 2001; Crayn, Winter and Smith, 2004; Heibl and Renner, 2012), but most studies concentrate on the genus *Elaeocarpus* because it is the most species-rich genus in this family comprising 350-400 species (Coode, 2004). Most species of *Elaeocarpus* are rainforest trees and shrubs distributed in palaeo-tropical, palaeo-subtropical and warm temperate regions covering Madagascar, Malesia, Asia, Australia, New Zealand, and the Pacific islands (Fig 1).



**Figure 1.** [The distribution of Elaeocarpus \(https://www.gbif.org/species/3152094\).](https://www.gbif.org/species/3152094)

The morphology of *Elaeocarpus* has been well noted, but molecular work compared with morphology is only now developing, due to the late emergence of molecular techniques.

There are many infrageneric classifications of *Elaeocarpus* species in China, Malesia, Australasia, the Pacific islands and Papuasias. The first classification was created for species in New Caledonia by Brongniart and Gris mainly based on reproductive characters (Brongniart and Gris, 1861). Then, a number of taxonomists built classifications of *Elaeocarpus* species in their local regions including Southeast Asia and India (Bentham and

JD Hooker, 1862; Masters, 1874). All these classifications were different until Schlechter proposed a classification of the Papuasian species (Schlechter, 1916). This classification used the number of ovules per loculus as the primary diagnostic character (Schlechter, 1916). Schlechter did not use seed shape (whether the seed is curved or straight) which is now known to be the primary division of *Elaeocarpus* (Weibel, 1968). Coode (Coode, 1978) later stated that *Sericolea* seeds are gently bent, and discussed variation in curved seeds in Australian species (Coode, 1984); the rest of the family has straight seeds. Coode and Weibel had started to combine vegetative and reproductive characters together to define groups of *Elaeocarpus* species, which made the groups of *Elaeocarpus* more reliable (Coode, 1996a, 1996b, 2001c, 2001a, 2001b, 2003, 2010). After several years of work and combining previous classifications, 15 infrageneric groups and 27 subgroups of *Elaeocarpus* have been recognized, but most of these have been left without formal names until molecular work is further advanced; some – *Monocera*, *Coilopetalum* – are almost certainly polyphyletic.

There are 166 characters and 382 members used in this project. These characters are used to identify plants including numeric characters which are about numbers and measurements, and state-based characters which are all the remaining characters except the numeric ones. The 382 items contain species, subspecies and varieties and the groups of some items are unpublished or uncertain. In order to ensure the accuracy of the key created by this project, the uncertain items will be omitted before any further operations.

## 1.2 Electronic encoding of descriptive information

Many different types of languages and grammars are used in the identification descriptions. Triple Store Technologies, which means dividing a descriptive statement into 3 parts, subject, predicate, and object, are not broadly applied in plant descriptions but are convenient to break down a statement and store them in a computer. And for this project, triple stores are the main way to store *Elaeocarpus* taxonomical descriptions which will be parsed by computers. The subject is those things need to be described, predicates are those described characteristics, and the object is the value of a state of the predicate according to the subject (Figure 1). For example, the stamens of a specimen are 5 mm long. ‘Stamens’ is the subject, ‘has length’ is the predicate and ‘5’ is the object. The most convenient point of this method is that the object of one triple could be the subject of another triple (Figure 2). Following the example above, the object ‘5’ could be the subject of another triple whose predicate is ‘has units’ and the object is ‘mm’. In this way, the logic of the whole specimen would be clearer

and the description of one specimen could be constructed as a tree which contains the collection of all triples as shown below (Figure 3).



Figure 2. A simple triple.

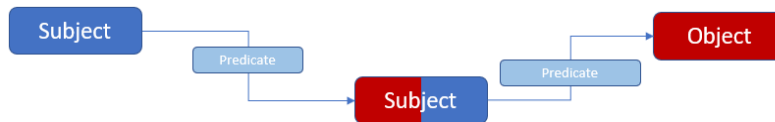


Figure 3. Chaining a pair of triples.

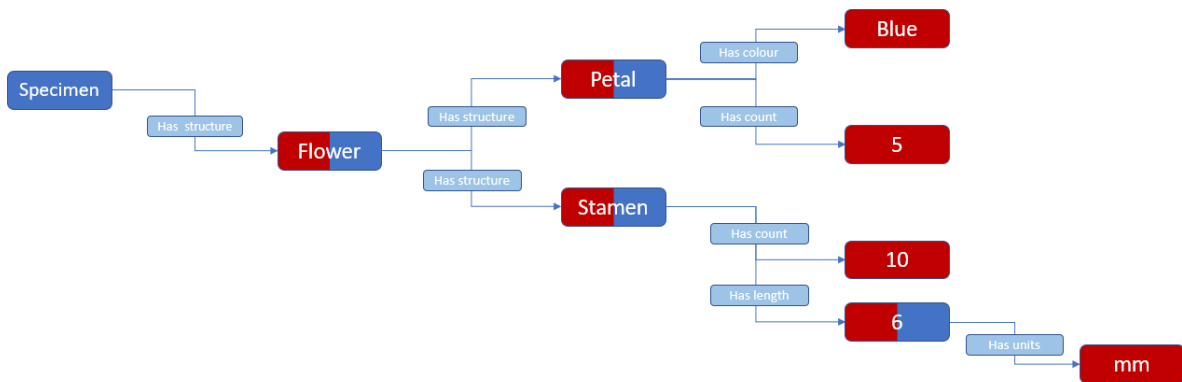


Figure 4. The example of one whole specimen tree.

From the tree above, it is easy to see that the vocabularies should be controlled. The predicate is usually restricted to structures, colour, length, and numbers. For example, a flower should be limited to having substructures including sepal, petal, stigma, stamen, style etc. The predicate of petal colour must be restricted to a range of colour choices. Controlled vocabularies limit the associations between nodes in the tree. When building an identification system, a controlled vocabulary should be created from the amalgamated characteristics of all concepts defined in the system.

The identification system that we want to develop will use the mechanism above to create model descriptions of the object categories. In this project, the object categories recognized are based on *Elaeocarpus* taxa and the descriptions built in the model are characteristics that each *Elaeocarpus* taxon displays. This system will construct an abstract conceptual framework with controlled vocabularies, against which real world objects can be compared. In taxonomy, this process (building an abstract framework) is called classification. When the real-world object matches the characteristics associated with one of the taxa, it means that

this object has been identified to this taxon. In this project, when a specific *Elaeocarpus* item matches one taxon of the framework, this item belongs to this taxon. What differs from taxonomic terminology is that and this matching process is called classification in computer science.

To summarise the above, a mechanism should be created helping users to describe a specimen with a controlled vocabulary according to a framework. This project aims to build an identification mechanism using *Elaeocarpus* information stored in a DELTA file and transfer all this information to the Padme system. The Padme system would then produce an interactive key according to information of *Elaeocarpus* to help people compare a real-world object with the abstract framework with the controlled vocabularies in a flexible way.

### 1.3 Differences between DELTA and Padme

This project is to migrate the DELTA descriptions of *Elaeocarpus* into Padme.

Understanding the differences between DELTA and Padme is important to translating statements from the DELTA to the Padme system.

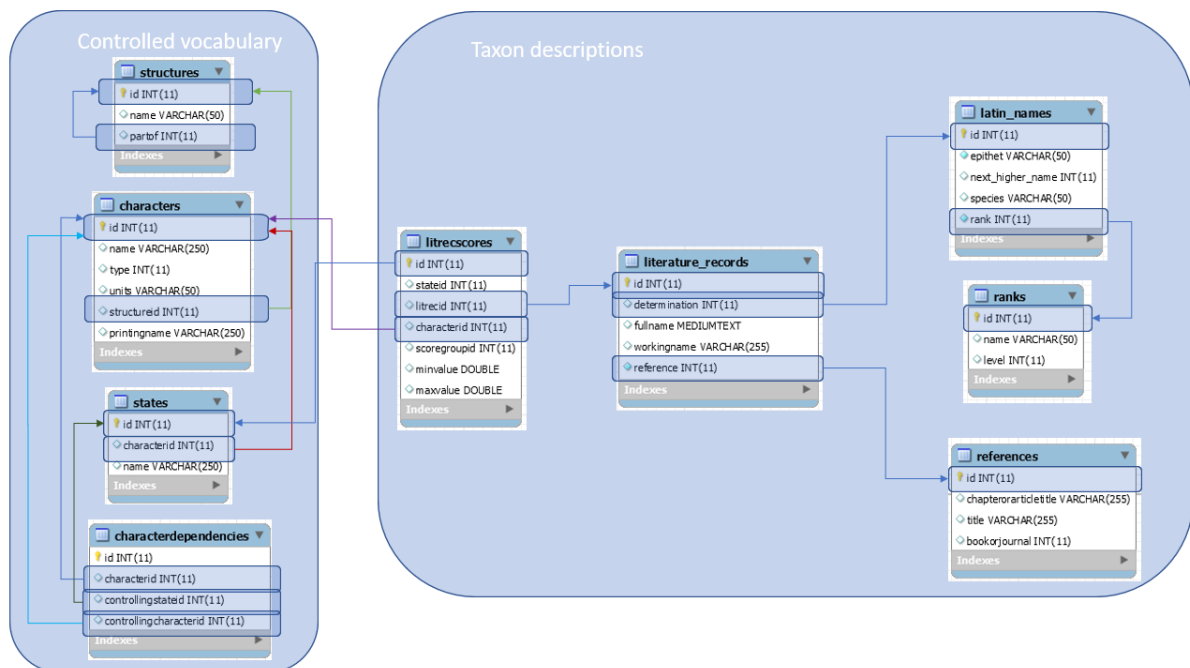
The DELTA (DEscription Language for TAXonomy) system was developed to solve some troubling questions which arose when preparing taxonomic descriptions for input to computer programs (Dallwitz, 1980). The form of coding is decided by the requirements of programs, which restricts the represented data type and the number of other programs using the data. DELTA was designed to help people to write descriptions conveniently. It could be used as a shorthand method of recording data (Dallwitz, Paine and Zurcher, 1993). DELTA could record data in free format, there is no limitation to place data in a particular way. Therefore, taxonomists could omit or add later any unknown or unimportant attributes. Those incorrect attributes could be modified or deleted and new one inserted at the end of the file. In a word, users could assign character numbers in any order and did not need to divide characters according to their types (Dallwitz, Paine and Zurcher, 1993). This system could encode all types of characters used in identification and classification including numeric characters and state-based characters with different rules. Besides, taxon descriptions could also be composed with the DELTA format using character records. The principle of character and taxon descriptions is shown in <https://www.delta-intkey.com/>

DELTA is a data format which can be manually edited. But it is hard to edit by hand and the manual editing will be error prone. The DELTA system can be converted into natural languages and in formats required by some programs (R.J.Pankhurst, 1986). But those

programs which can work directly with DELTA file are old and out of date. Most modern description recording or interactive key programs such as Padme system use their own storage system. Padme system has its own internal storage mechanism closely based on the subject, predicate, and object model, which cannot be conformed by the DELTA file.

Therefore, it is necessary to translate the DELTA file in order to be used by Padme system.

Padme system uses a MySQL database including many tables. Each table has columns and rows and the meeting of a column and a row is called a field or an element. Elements might link to other tables so that there would be a complete SUBJECT-PREDICATE-OBJECT triple and a tree containing all triples (details in 1.4). Therefore, Padme follows the SUBJECT-PREDICATE-OBJECT model to store descriptive information (Fig 4). Structures always represent the subject and they are also the object when the predicate is ‘is part of’, which means this structure is the part of the other structure. In all other cases, the predicate is the character and the object would be a state or states when the character is state-based or a min or max value in the case of numeric characters. All details of Padme tables will be described in the following parts.



**Figure 5.** The SUBJECT-PREDICATE-OBJECT model of Padme system.

## 1.4 Migrating scripting PHP and relational database MySQL

PHP is a general-purpose scripting language which is suited to web development (<https://www.php.net/>). It was built by Rasmus Lerdorf in 1994. PHP originally stood for Personal Home Page (<https://www.php.net/manual/en/history.php.php>), but it now stands for the recursive initialism PHP: Hypertext Preprocessor (<https://www.php.net/manual/en/preface.php>). PHP code is usually processed on a web server by a PHP interpreter implemented as a module, a daemon or as a Common Gateway Interface (CGI) executable. In this project, PHP is the scripting language used to write the routines parsing the DELTA file and storing the parsed data in Padme.

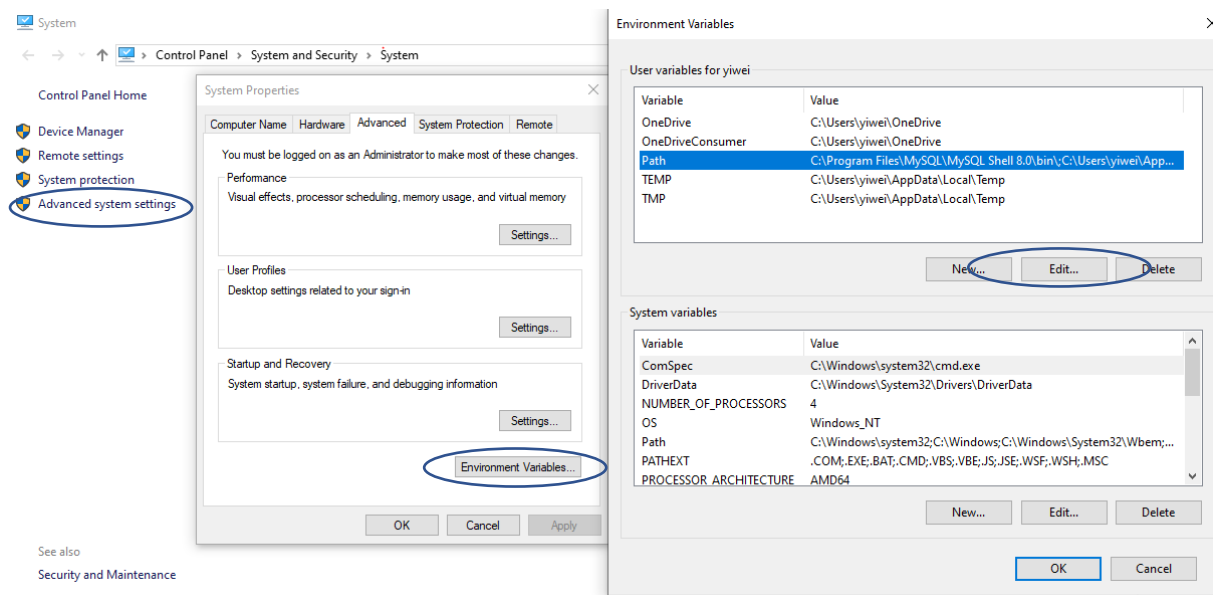
MySQL is an open-source relational database management system (<https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html>). A relational database organizes data into one or more data tables in which data types may be related to each other; these relations help structure the data (<https://en.wikipedia.org/wiki/MySQL>). Programmers can use MySQL to create, truncate, extract and change data from the database. In this project, Padme stores its data in MySQL relational database tables. Each table has rows and columns, each row represents a single record of information, and each column has a name and a datatype specifying what kind of data can be stored in this position. The point where a row meets a column called a field or an element. Data in one table can refer to the data in another table by storing the suitable record key from the referenced table in the field of the referencing table. The aim of this project is translating the DELTA data to store in Padme. The data should be inserted in appropriate tables and the references between different tables should be accurate. Details of the software preparation will be described in the chapters below.

## 2. Software preparation

Before the start of this project, some software had to be installed and set up. This part contains two aspects, software installation and software initialization and configuration.

### 2.1 Software installation

What needed to be installed was PHP, MySQL, TortoiseSVN and code editor. PHP is a scripting language used to write the routines for parsing the DELTA file and storing the parsed information in Padme. The version of PHP I installed is 7.3.17 64-Bit (<https://windows.php.net/download#php-7.3-ts-VC15-x64>). After installation of PHP, the PATH environment variable was modified to make sure PHP could be found when trying to execute it from the command line. The process of setting the environment variable is shown below (Figure 6).



**Figure 6.** The process of setting the environment variable.

MySQL is the Relational Database Management System (RDBMS) which will be used to store Padme data. The version of MySQL installed is the latest, MySQL8 (<https://dev.mysql.com/downloads/installer/>). There are many options when installing MySQL. For this project, 'standalone installation' was selected as the installation type and 'development configuration' as the configuration type. The option 'TCP/IP' was selected for

connectivity. PHP/PDO represents a connection between PHP and a database server, which was used in this project to connect PHP and MySQL database. PHP/PDO can only connect using the legacy authentication method, so this option should also be selected during MySQL installation. All other selections follow the default options and no advanced options need to set.

TortoiseSVN helps programmers manage different versions of the source code for their programs, allowing programmers to interact with a code repository to download others' changes and commit their changes. For this project, we use GitHub repository to store code, and we could access each other's code and monitor the progress of the whole project.

TortoiseSVN may be obtained from

<https://osdn.net/projects/tortoisesvn/storage/1.13.1/Application/TortoiseSVN-1.13.1.28686-x64-svn-1.13.0.msi/>. PHP script files are just text files with a .php file extension, a suitable editor

specifically designed to edit code is helpful for editing PHP files. I chose Microsoft's Visual Studio code for this project, which supports a wide range of languages including PHP and offers powerful code completion and syntax highlighting

(<https://code.visualstudio.com/download>).

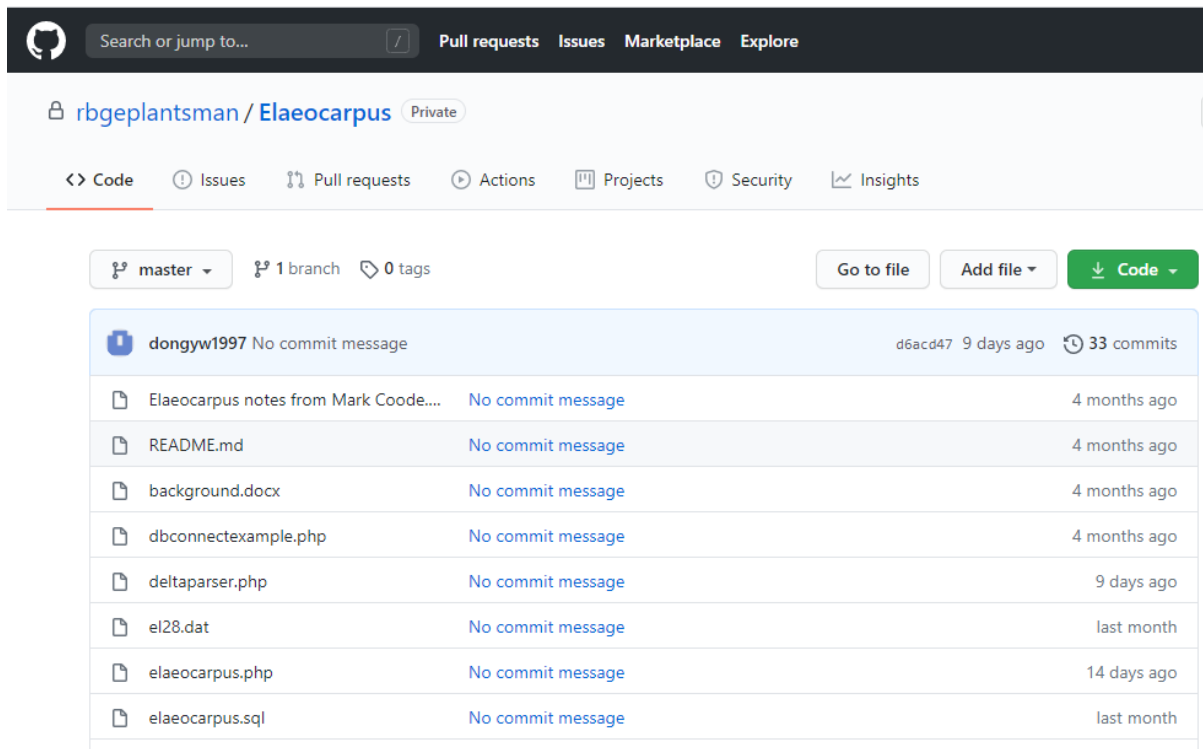
## 2.2 software initialization and configuration

GitHub is a large community, which helps programmers to discover and share codes with others. For this project, GitHub helped me update code with my supervisor, Martin Pullan.

The first thing to do was open a GitHub account for me and join the *Elaeocarpus* repository created by Martin. Then obtain a local copy of the repository contents from GitHub















(<https://github.com/rbgeplantsman/Elaeocarpus>) (Figure 7) using TortoiseSVN. After that, the local folder and GitHub would be connected, 'SVN update' and 'SVN commit' could be used to update the latest version of files from GitHub and submit the newest version from the local machine. Committing changes regularly is a good way to establish a fine grained back up which will allow easier reversion to or re-examine of earlier versions of scripts.





**Figure 7.** The repository of *Elaeocarpus* in the GitHub.

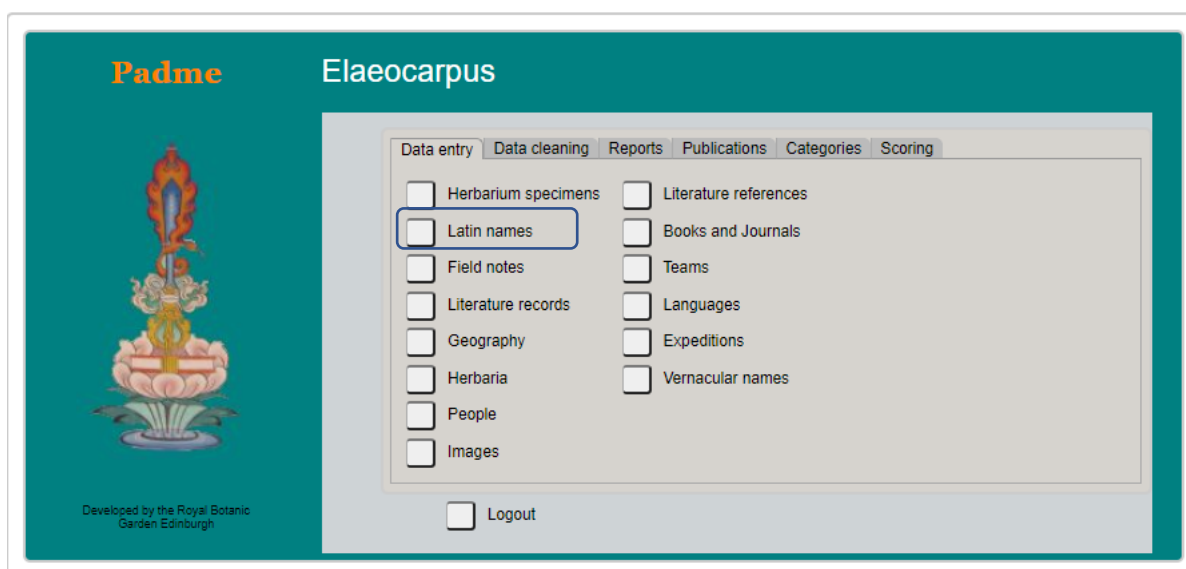
There are some files in the Trunk folder (Figure 8), of which the file `hello.php` is a testing file to test whether PHP is installed. Run it in the command prompt and if the file runs successfully, PHP has been installed in the local machine. The file `php.ini` is a PHP configuration file containing necessary settings. This file contains all necessary configuration information to use PHP to connect to a MySQL database. Copy the file from the Trunk folder into the PHP folder to connect to the database. A new connection called 'padme' should be created in MySQL. After the connection has been made, the database management should be set. The `padme.sql` file was imported; this is a copy of the Padme schema with useful data from the Trunk folder and is used to initialize the copy of Padme for the *Elaeocarpus* data set. After setting up MySQL, the `dbconnectexample.php` could be used to test the connection. If the setup is correct, a prompt for a password will appear after running this file.

 .history	15/08/2020 11:04
 background.docx	22/04/2020 13:41
 dbconnectexample.php	21/04/2020 23:29
 deltaparser.php	15/08/2020 11:04
 el28.dat	13/07/2020 12:41
 Elaeocarpus notes from Mark Coode.docx	22/04/2020 13:41
 elaeocarpus.php	26/07/2020 12:06
 elaeocarpus.sql	14/07/2020 13:28
 hello.php	21/04/2020 23:29
 padme.sql	22/04/2020 13:41
 php.ini	22/04/2020 14:31
 README.md	21/04/2020 23:29
 software.docx	21/04/2020 23:29
 workplan.docx	05/07/2020 23:04

**Figure 8.** The contents of Trunk folder.

### 3. Padme preparation: creating *Elaeocarpus* records in Padme

The aim of this project is to translate information about *Elaeocarpus* from a DELTA file to Padme. The DELTA file of *Elaeocarpus* el28.dat was constructed by Mark Coode and contains information about characters and items relating to *Elaeocarpus* including character types, numbers of states, key states, character descriptions and item descriptions. The first step of this project is linking the taxon descriptions from the DELTA file el28.dat to Padme through the Latin name (Figure 9). It will be simplest to create an entry into Padme for each section, species, variety, and subspecies manually.



**Figure 9.** [Padme main menu \(choose Latin name option\).](https://padme.rbge.org.uk/padme/Elaeocarpus/)

Log in to Padme (<https://padme.rbge.org.uk/padme/Elaeocarpus/>) and select the Latin names option (Figure 9) to enter Latin names to be used. The main fields in the form are Rank used to signify the taxonomic rank of the name being recorded such as family, genus, or species. Section, species, variety, and subspecies are the dominant ranks in this step (Figure 10). Epithet is used to store the epithet of the current name. For monomial like family and genus, the epithet is the Latin name while, for multinomial, it is the last part of the name. For example, the epithet of *Bellis perennis* is *perennis*. Next higher name is used to place the name in the taxonomic hierarchy. In the el28.dat DELTA file, the Latin names at the start of each item description contain the whole Latin name and next higher rank name, which are

more than needed when importing the description. Therefore, only specific and subspecific names are added in Padme and reserved in DELTA file while others are deleted.

The screenshot shows the 'Latin names' data entry form in Padme. The main title is 'Elaeocarpus'. The form is divided into several sections: 'Place of original publication', 'Nomenclature', 'Types', 'Citations', 'Vernacular names', 'IUCN Assessments', and 'GUIDS'. The 'Nomenclature' section contains the following fields: 'Verified', 'Verified by', 'Taxonomic status is provisional', 'Authority unknown', 'Ined.', 'Nom. nov.', and 'Imported authority'. Below these are 'Rank' (set to 'Genus'), 'Epithet' (set to 'Elaeocarpus'), 'Authority', and 'Original authority'. The 'Next higher name' field is also present. Below this is the 'Basionym' section with 'Basionym', 'Basionym Authority', and 'Original basionym authority' fields. A note states: 'This taxon associated with this name cannot be sunk because it has 360 accepted members' with a 'Browse the accepted members' button. There are also fields for 'Illegitimate', 'Reason illegitimate', 'Invalid', and 'Reason invalid'. At the bottom, there are fields for 'Homotypic names', 'Heterotypic synonyms', and 'Misapplied names', followed by a 'Notes' field. The 'Rank' and 'Next higher name' fields are circled in red in the original image.

**Figure 10.** Padme data entry section of the Latin names form (Fields to be filled are in cycles).

The first record to be added is the genus *Elaeocarpus*, followed by records for the 12 sections in *Elaeocarpus*, namely *Elaeocarpus*, *Polystachyus*, *Coilopetalum*, *Monocera*, *Acronodia*, *Ganitrus*, *Fissipetalum*, *Obovatus*, *Lobopetalum*, *Dactylosphaera*, *Oreocarpus* and *Dicera*. The rank field of these records should be set to section and the next higher name field should be set to the genus *Elaeocarpus*, which means they all belong to *Elaeocarpus*, as shown in Figure 11 below. Species, subspecies or variety names like section names, would link to their next higher rank and would obtain their generic placement through inheritance. For example, all species belonging to section *Acronodia* such as *acronodia*, *chrysophyllus*, *euneurus*, and so on are stored under section *Acronodia* while subspecies follow the same pattern (Figure 12).

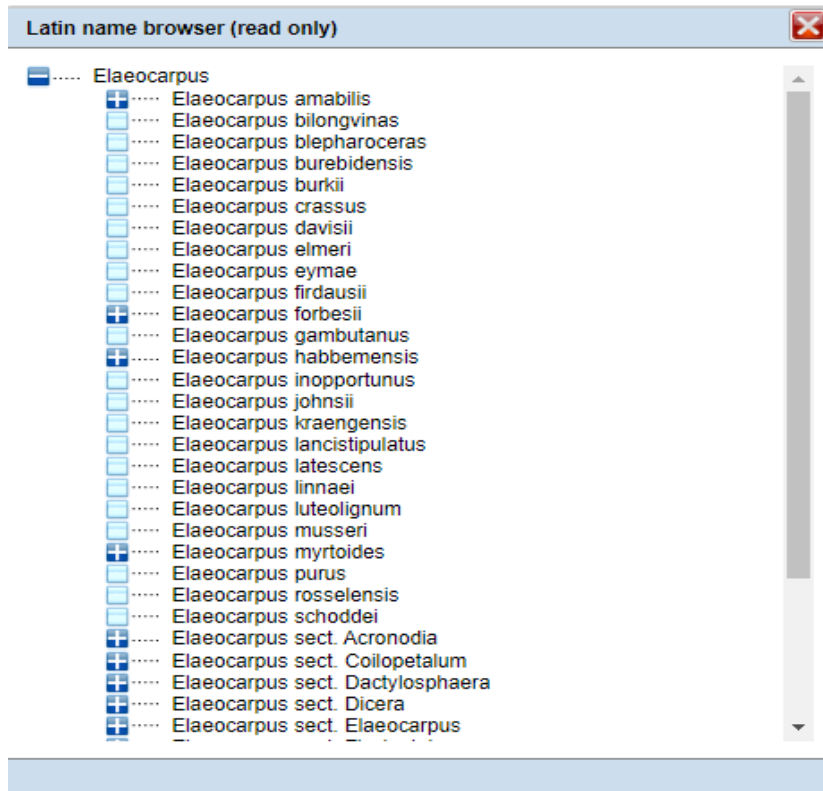


Figure 11. All sections and species are under Genus *Elaeocarpus*.

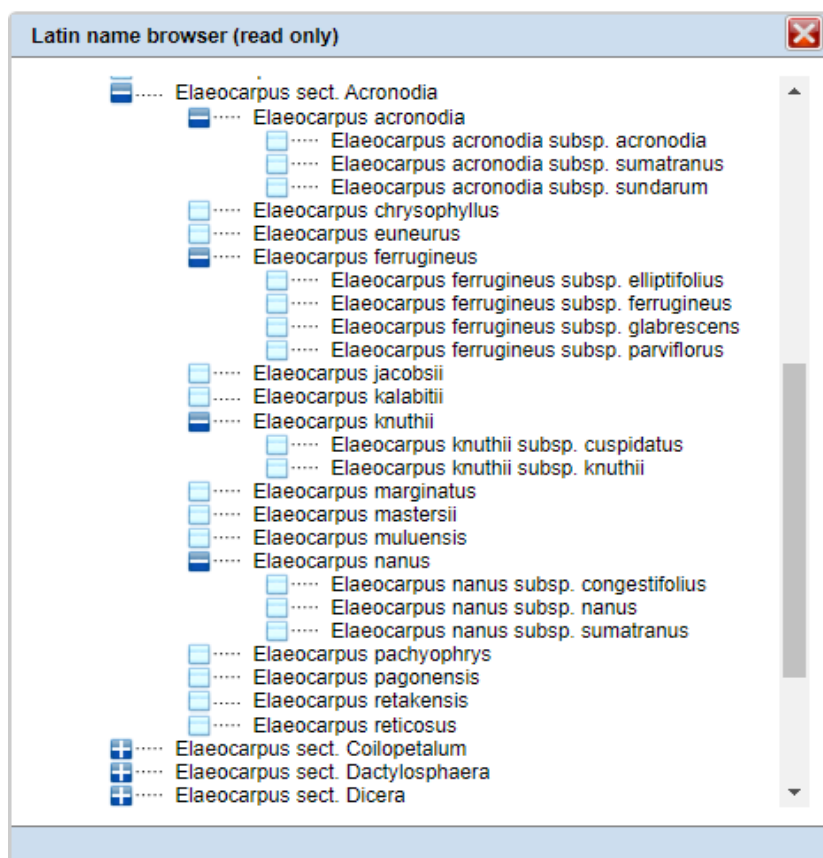


Figure 12. Species of *Acronodia* are under Section *Acronodia* and subspecies are under their corresponding species.

For each species, the DELTA file el28.dat must be opened and the ITEM DESCRIPTION directive found, which marks the beginning of the coded taxon description. Each description is separated by a line and the first line of each item contains the number assigned to the taxon in the DELTA file and the taxon name. Each item needs a new Padme record. Following the method described above, the rank field should be species, epithet is what showed in the first line of each character in el28.dat and the next higher name field should be the corresponding section from the drop-down list. For example, in el28 the first record is “#1. inopinatus /*Elaeocarpus*” which means the first item inopinatus belongs to section *Elaeocarpus*. So, the epithet field should be added in “inopinatus” and “*Elaeocarpus* sect. *Elaeocarpus*” should be selected from the drop-down list of the next higher name field (Figure 13).

The screenshot shows a web-based form for entering taxonomic data. The title bar reads "Latin names" and the main heading is "Elaeocarpus inopinatus". On the left, there is a search sidebar with a "Latin name" input field and an "Expand search to include member taxa" checkbox. The main form area contains several sections:
 

- Place of original publication:** Includes tabs for "Nomenclature", "Types", "Citations", "Vernacular names", "IUCN Assessments", and "GUIDS".
- Verification:** Fields for "Verified" (checkbox), "Verified by", "Taxonomic status is provisional" (checkbox), "Authority unknown" (checkbox), "Ined." (checkbox), "Norm. nov." (checkbox), and "Imported authority" (checkbox).
- Rank and Epithet:** "Rank" is set to "species" and "Epithet" is "inopinatus".
- Authority:** A text field for the authority name.
- Original authority:** A text field for the original authority.
- Next higher name:** A dropdown menu showing "Elaeocarpus sect. Elaeocarpus".
- Basionym and Authority:** Fields for "Basionym", "Basionym Authority", and "Original basionym authority".
- Synonymy:** Fields for "Synonym of" and "Synonym chain".
- Illegitimate/Invalid:** Fields for "Illegitimate" (checkbox), "Reason illegitimate" (dropdown), "Invalid" (checkbox), and "Reason invalid" (dropdown).
- Names:** Fields for "Homotypic names", "Heterotypic synonyms", and "Misapplied names".
- Notes:** A large text area for additional information.

 At the bottom, a status bar shows "3 of 361" records.

**Figure 13.** [The record of \*Elaeocarpus inopinatus\*.](#)

For subspecific items including variety and subspecies, most steps are similar to those for specific items except the next higher name field. If a match could be found when entering the species name in the next higher name field, there is already an entry for the species in the database and it is only necessary to create a new record for subspecific items following the process of species records. However, if there is not a match in the drop-down list, choosing “add a new item” will allow a new record for the species to be created. Then information about the species which is lacking in the new form can be added and the form saved. Information about this species would be added into Padme system. When encountering the species again, it will be displayed in the drop-down list.

There are some confusing points when creating new records. Firstly, there are some items with question marks, which means the sectional placement of this item is uncertain. In these cases, select the genus *Elaeocarpus* as the next higher name of this item instead of linking to a section or species. Secondly, when the next higher name is section *Elaeocarpus*, more patience is needed to check the species should be placed in “*Elaeocarpus* Sect. *Elaeocarpus*” rather than “*Elaeocarpus*” which means the genus *Elaeocarpus*. Lastly, the numbers of description items are not continuous, and some names are special. That is because there are some taxa in the el28.dat file that are taxonomically ambiguous, and to make work easier and more regular we eliminated those taxa from the DELTA file.

## 4. PHP Script file

There are two files used in this project: *Elaeocarpus.php* and *Deltaparser.php*.

### 4.1 Deltaparser.php

```
class DeltaParser
{
    private $inputfile;

    /**
     * Class constructor
     * Parameters
     * $inputfile - path to the input file to parse
     **/
    public function __construct($inputfile)
    {
        //Store the inputfile variable passed in the parameters (the path) in the inputfile property of this class

        $this->inputfile = $inputfile;
    }

    /**
     * Function controlling the parsing of a DELTA file
     * parameter
     * none
     **/
    public function parse()
    {
        //Call private functions(method)
    }
}
```

*Deltaparser.php* is the script to define a **DeltaParser** class. In object orientated programming, a class is an extensible program-code-template for creating objects, providing initial class properties and class methods or functions (Gamma, 1995). For example, if a class named **Plants** were created, it could have properties like *\$name*, *\$height*, *\$colour\_of\_flowers*, *\$shape\_of\_leaves* and other characters. All characters of plants could be defined as properties. When individual objects are instantiated, the individual plant will inherit all the properties and methods from the class and the values of those properties are assigned individually and differently to each object instance. This **DeltaParser** class is used to create a



class which could parse all DELTA files with the same structure as el28.dat and bring all information into the Padme system.

Each class must have a constructor to initialize the properties of an object when creating it. If creating a `__construct()` function, PHP will automatically call this function when creating an object from the class. It provides a convenient way to set the initial values of a new object. In the class above, the `__construct()` function stored the parameter `$inputfile` in the `inputfile` property of this class. The object operator (`->`) means that what is on the right of the operator is a member of the object instantiated into the variable on the left side. When properties or methods of an object need to be called in this method, the object operator is essential. The `parse()` method is used to parse the DELTA file.

What needs more attention is that private properties and methods may only be used within the class while public properties and methods may be accessed from everywhere. There is another access modifier called protected in PHP. Protected properties and methods may be accessed by the class and classes derived from this class.

## 4.2 *Elaeocarpus.php*

```
//Find and load the DeltaParser class definition
require_once "deltaparser.php";
$parser = new DeltaParser("c:/0/Elaeocarpus/trunk/el28.dat");
$parser->parse();
```

*Elaeocarpus.php* is used to load the class definition file (*Deltaparser.php*). It is the entry point for the application and the script will be run when translating the DELTA file. To import a different DELTA file a copy of *Elaeocarpus.php* would be made and edited such that the parameters passed to *deltaparser.php* were set appropriately. Once created and operating correctly the *deltaparser.php* script would not require any further editing to be re-used. The first line aims to load the class definition file. There are two kinds of functions which could load another file, `include()` and `require()`. The only difference between these two is that `include()` only generates a PHP warning but the script will continue to execute if the file to be included cannot be found, while `require()` will generate a fatal error and stop the script execution. The function `require_once()` is identical to `require()` except that PHP will check whether the file has already been included, and, if so, not require it again.

The second line is to create a new **DeltaParser** object passing in the path of DELTA file to be parsed to the `constructor` function of the **DeltaParser** class. Class is nothing without

objects. Each object has all the properties and methods defined in a class, but the property values are different. Objects of a class could be created using the `new` keyword. The third line is to call the `parse()` method of **DeltaParser** class using the object operator.

## 5. Error trapping and handling

Error trapping and handling is one of the key elements of writing good code. The user may make all sorts of mistakes which need to be trapped. Therefore, it is important to anticipate the errors which might be made by users before writing code to handle them. If these errors are not trapped, the whole coding might be prone to nasty crashes.

### 5.1 Throwing exceptions

```
private $inputfile;

/**
 * Class constructor
 * Parameters
 * $inputfile - path to the input file to parse
 * throw an exception when the file cannot be found
 */
public function __construct($inputfile)
{
    if(file_exists($inputfile))
    {
        //Store the inputfile variable passed in the parameters(the path) in the inputfile property of
        this class
        $this->inputfile = $inputfile;
    }
    else
    {
        //throw an exception when the file does not exist
        throw new Exception("file does not exist");
    }
}
```

In this project, the *\$inputfile* may not exist so, it is better to add error trapping to the **DeltaParser** constructor throwing an exception when the file specified in the *\$inputfile* parameter does not exist. Throwing an exception is almost as simple as it sounds. All it takes is to instantiate an exception object and ‘throw’ it. **Exception** is the base class for all exceptions in PHP 5. Each built-in exception has some configurable properties such as an exception code, an exception message, the line number of the exception in the source file and the source filename of the exception. This class is used to create an exception object which could be diagnosable and be read by a human.

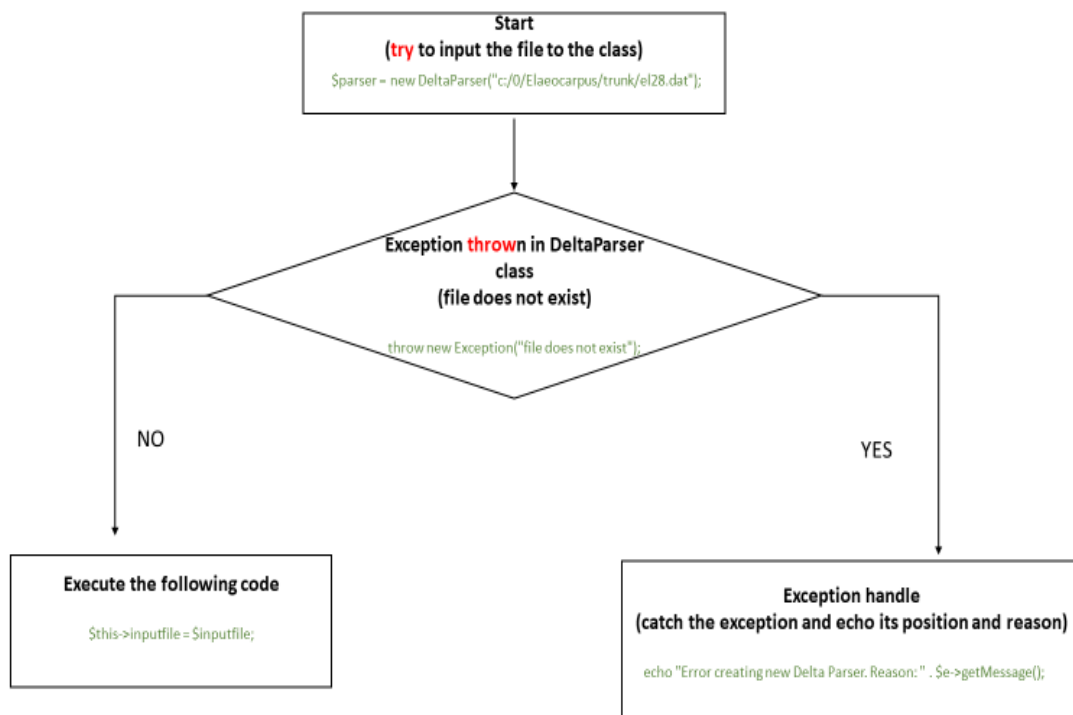
In the constructor above, store the parameter *\$inputfile* in the *inputfile* property of **DeltaParser** class when the file exists, or else throw a new exception which would show the message to users “file does not exist”. Throwing the exception in constructor could check the file automatically when creating a new object of the **DeltaParser** class.

## 5.2 Try catch for error handling

```
try
{
    $parser = new DeltaParser("c:/0/Elaeocarpus/trunk/el28.dat");
}
catch(Exception $e)
{
    //use error trapping to catch the exception
    echo "Error creating new Delta Parser. Reason: " . $e->getMessage();
}
```

Exceptions could be thrown and caught using PHP try and catch blocks. In this project, when an exception is thrown from constructor of this class the code following will not be executed, and PHP will try to find the matching ‘catch’ block. The catch block above shows the position and the reason of the error. Use the *getMessage()* method of class Exception to get the message of exception which was defined in throw.

I will make a diagram to clarify the structure of try-throw-catch using the above as an example:



**Figure 14.** The logical diagram of error trapping.

## 6. DELTA file input

Before starting to parse the data in the DELTA file, the contents of the file need to be read into memory.

### 6.1 Read Data

```
private $inputdata;

/**
 * Function reading the class property inputfile
 * parameter
 * none
 * set a newline at the end of each array element
 */
private function readData()
{
    $this->inputdata = file($this->inputfile, FILE_IGNORE_NEW_LINES
}
```

This method is reading the input file. The function *file()* is to read the entire file into an array. The required parameter of this function is the path to the file, and the optional parameter flags could be one or more constants. In the function above, `FILE_IGNORE_NEW_LINES` means omitting a newline at the end of each array element. Then store the result in the *inputdata* property of the class defined earlier. When the input file has successfully read the file, the *file()* function would return an array. If it is desirable to return the contents of the file as a string, the *file\_get\_contents()* function may be used.

### 6.2 Test readData()

```
/**
 * Function testing the class property inputdata
 * parameter
 * none
 * iterate through $this->inputdata
 */
public function echoData()
{
    foreach($this->inputdata as $value)
        echo $value;
        echo "\n";
}
```

The purpose of the *echoData()* method is only to test that the *readData()* function is working correctly. This will iterate through the class property *inputdata*. The *foreach* construct is a specific and easy way to iterate over arrays. *Foreach* works only on arrays and objects and will issue an error when using it on a variable with a different data type or an uninitialized variable. On each iteration, the value of the current element is assigned to *\$value* and the internal array pointer is advanced by one. Then all contents of each entry in the *inputdata* array will be printed in separated lines on the console using 'echo'.

```
/**
 * Function controlling the parsing of a DELTA file
 * parameter
 * none
 **/
public function parse()
{
 //Call private functions
 $this->readData();
 $this->echoData();
}
```

Because the *Elaeocarpus.php* file only calls the *parse()* method, all methods which want to be used outside the class(used in *Elaeocarpus.php*) need to be called in the *parse* method. And the call to *echoData()* from the *parse()* method could be removed or commented out when the input file has been read correctly.

## 7. Connect to and Initialise the Padme MySQL database

### 7.1 Connect to the Padme database

After inputting the DELTA file, we need to connect the Padme database in MySQL. If not, all operation would only parse the DELTA file and could not migrate the information in the DELTA file to the Padme system.

```
private $conn;

/**
 * Funtion connecting to the MySQL databasse
 * parameter
 * none
 * set the MySQL information
 **/
private function dbConnect()
{
    //Connect to the Database using PDO

    $db_host="localhost"; //The name of the machine on which the MySQL server is running - l
ocalhost means it is the same machine that the script is running on
    $db_username="padmeuser"; //User name to use when logging in (padmeuser)
    $db_db="padmeweb1_Elaeocarpus"; //Connect to the Elaeocarpus data set
    $db_password="padmepassword"; //The password of the user account
    $db_pdo_prefix = "mysql"; //Tell PDO that we will be connecting to a MySQL server
    $db_dsn="host=$db_host;dbname=$db_db;charset=utf8"; //Build the connection string speci
fyng that data will be transmitted and recieved using UNICODE utf-8 encoding
    $this->conn = new PDO("$db_pdo_prefix:$db_dsn",$db_username,$db_password); //Ask P
DO to make the connection and store the connection in a variable called $conn
    $this->conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION ); //
Tell PDO to report errors by throwing exceptions
}
```

The **PDO (PHP Data Objects)** defines a lightweight interface for accessing databases in PHP. It provides a data-access abstraction layer for working with databases in PHP. **PDO** is a class of PHP, which represents a connection between PHP and a database server (MySQL in this project). The method above uses **PDO** to connect the file to the Padme MySQL database. *\$db\_host* is the name of the machine on which MySQL server runs, and localhost means it is the same machine which the script runs on. The *\$db\_username* variable stores the username to log in the database. The *\$db\_db* variable contains the name of the data set (padmeweb1\_Elaeocarpus) to be connected to. The *\$db\_password* variable stores the

password of the user account. The `$db_pdo_prefix` variable contains a string specifying the type of database that will be connected to which is MySQL server in this project. Use `$db_dsn` to build the connection string specifying that data will be transmitted and received using UNICODE utf-8 encoding. Then ask **PDO** to make the connection and store the connection in a property of class called `conn`. `SetAttributes` is a method of **PDO** class, which aims to set an attribute on the database handle. `ATTR_ERRMODE` means error reporting and `ERRMODE_EXCEPTION` means exceptions throwing. So, the final statement aims to tell **PDO** to report errors by throwing exceptions.

The `DeltaParser` class can only connect to the same database and schema in this way. If we want to make this connection more generally useful, those variables should be as parameters of `__construct()` and defined outside the `DeltaParser` class (in `Elaeocarpus.php` file). The information used to connect to the database can be specified when instantiating the `DeltaParser` object.

The `__construct()` of **DeltaParser** class:

```
public function __construct($inputfile, $db_host, $db_username, $db_db, $db_password, $db_pdo_prefix )
{
    if(file_exists($inputfile))
    {
        $this->inputfile = $inputfile;
        $db_dsn="host=$db_host;dbname=$db_db;charset=utf8";
        $this->conn = new PDO("$db_pdo_prefix:$db_dsn",$db_username,$db_password);
        $this->conn->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
    }
    else
    {
        throw new Exception("file does not exist");
    }
}
```

Instantiate a new **DeltaParser** object in `Elaeocarpus.php`:

```
$parser = new DeltaParser("c:/0/Elaeocarpus/trunk/el28.dat", "localhost", "padmeuser", "padmeweb1_Elaeocarpus", "padmepassword", "mysql");
```

## 7.2 Initialise the setting of MySQL padme database

```
/**
 * Function to initialise the setting of MySQL
 * parameter
 * none
 * delete TABLE structures, characters, states, litrescores, literaturerecords
```

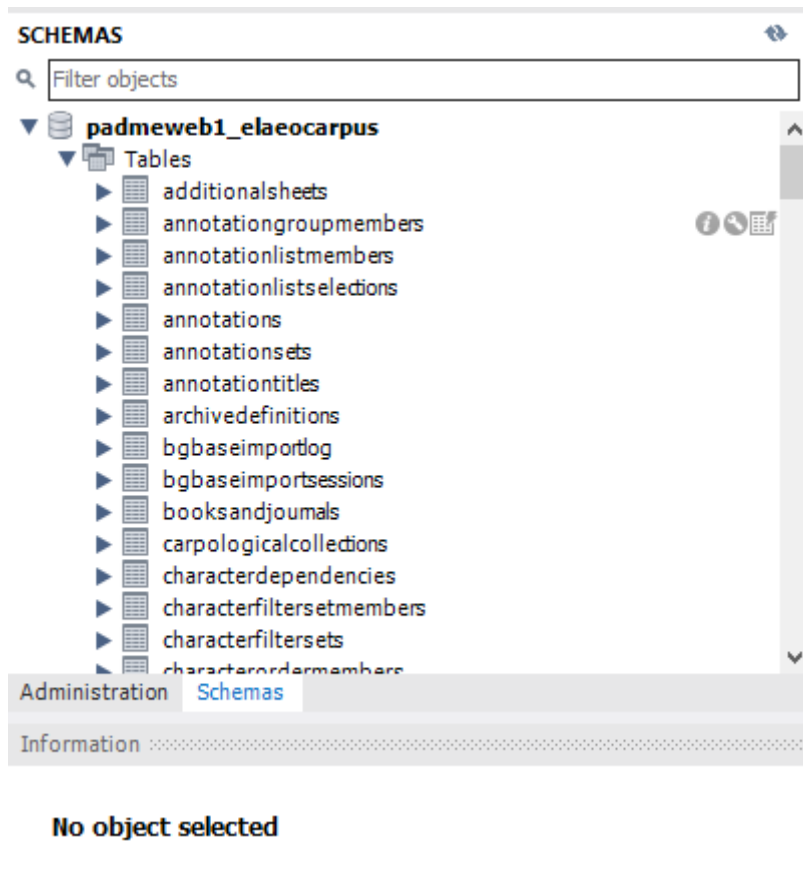


```

* catch the PDOException if there is something wrong with the query
**/
private function dbInitialise()
{
    $sql = "TRUNCATE TABLE structures; TRUNCATE TABLE characters; TRUNCATE T
ABLE states; TRUNCATE TABLE litrescores; TRUNCATE TABLE literaturerecords; ";
    //Truncate TABLE structures, characters, states, litrescores and literaturerecords.
    $p = $this->conn->prepare($sql);
    //Execute the query catching any exceptions that are thrown
    try
    {
        $p->execute();
        //Iterate through the results of the query and show them in the console.
    }
    catch(PDOException $e)//Trap any error thrown when the query is run
    {
        //Print the error to the screen.
        echo "Error running the query. Reason {$e->getMessage()}\n";
        exit; //Stop the script
    }
}

```

This method(*dbInitialise*) will empty the appropriate tables into the *Elaeocarpus* data set in preparation for reading in new records from the DELTA file. To empty and reset the record counters in tables including structures, characters, states, litrescores and literaturerecords, the truncate SQL command should be used. The *PDO::prepare* method is used to prepare a SQL query for execution and returns a *PDO::statement* object. Then execute the *PDO::statement* returned by prepare method to obtain a result set from the query which is then iterated through and each row of the results of the query are shown in the console. The try-catch block is used to catch any exceptions thrown when the query is executed (for more details, see Chapter 5, Error trapping and handling). What needs more attention is the use of ‘exit’ to stop the whole scripting when catching some exceptions. The *exit()* function prints a message and terminates the current script. If ‘exit’ is not used after exception, the scripting would continue to run, which might cause nasty crashes. After executing the method, tables in the Padme system could be used to check whether the truncate statement had worked (Figure 15).



**Figure 15.** The tables of the Padme Elaeocarpus database.

## 8. DELTA file navigation

What we need to do before reading, parsing, and storing characters is to find the line of the directive name so that we can define the position of characters.

```
private $readpos;

/**
 * Function that locates the position of a directive in the DELTA file
 * Sets the readpos class property to the index of the line in the inputdata class property to 1 greater than the index of the directive
 * Parameters
 * $directive: string containing the name of the directive find
 * Throws an exception if the directive cannot be found
 **/
private function findDirective($directive)
{
    //Search the array and find the line containing directive name
    for($line = 0; $line < sizeof($this->inputdata); $line++)
    {
        if(strpos($this->inputdata[$line], $directive))
        {
            $this->readpos = $line + 1;
            return;
        }
    }
    echo "Directive $directive not found";
}
```

In el28.dat, directives are on a single line starting with an asterisk (\*). The *findDirective* function should search the array of lines read in the last function stored in the *inputdata* class property and find the line containing the name of the directive sought, such as CHARACTER TYPES, NUMBERS OF STATES, KEY STATES and so on.

The for loop is to search the whole array and find the line containing the directive name. The first expression ( $\$line = 0$ ) is executed once unconditionally at the beginning of the loop. In this case, it means that the loop is reading from the index 0 of the array. The second expression ( $\$line < \text{sizeof}(\$this->\text{inputdata})$ ) is evaluated at the beginning of each iteration. If it evaluates to true, the loop continues, and the statements in the loop will be executed. If not, the loop will end. In the function above,  $\$line < \text{sizeof}(\$this->\text{inputdata})$  means that, if the

variable *\$line* is not less than the length of the array *inputdata*, the loop needs to stop. It is important to note that indexes in PHP are 0 based, so the largest index number would be one less than the length of the array. So, less than operator would be used more often than less than or equal sign when using the *sizeof()* function. And the last expression (*\$line++*) would execute at the end of each iteration. The *\$line* would be increased by one at the end of every iteration in this case.

The *strpos()* function finds the position of the first occurrence of a substring in a string. The first parameter of this function (*\$this->inputdata[\$line]*) is the string to search in. The second parameter (*\$directive*) is what needs to be found. There is a third optional parameter which does not appear in the function above. If it is specified, the search will start at a specified number of characters from the beginning of the string, or, the search will start at a specified number of characters from the end of the string. The default value is 0 as in this function. So, the function above is to find the first position of *\$directive* which is the parameter of the *findDirective* method. When the position has been found, its line number need to be stored in the class property *readpos*. One thing to note, the real line number shown in the editor should be *\$line + 1* due to 0 based indexes in PHP.

It may be that nothing corresponds to *\$directive*. Therefore, error trapping is still necessary in this case. If the directive name cannot be found, output a error message.

## 9. Reading in the character types

The first directive to be found is CHARACTER TYPES. To find this directive, the *parse* method of the **DeltaParser** class should be extended calling the *findDirective* method. The line number of CHARACTER TYPES could be found after calling in the *parse* method. Next step is to get information after CHARACTER TYPES and before the next directive name (Figure 16). Then store character types into an array which should correspond to the character description (Figure 16).

```
/**
 * Function that gets the string between $directive and next directive name
 * parameter
 * $directive: string containing the name of the directive find
 * add each line behind the $inputstring and use "*" to stop adding
 * return
 * $inputstring: string between $directive and next directive name
 **/
function getDirectiveData($directive)
{
    for($i = $this->readpos; $i < sizeof($this->inputdata); $i++)
    {
        //Get the string after $directive and before next directive name

        if(strpos($this->inputdata[$i], "*") === false)
        {
            $inputstring .= $this->inputdata[$i];
        }
        else
        {
            return $inputstring;
        }
    }
}
```

The function above is to get character types. *\$this->readpos* is the line number of the directive which was stored by the *findDirective* function (Chapter 6). This for loop (details of for loop details are to be found in Chapter 6) is to get the string after *\$directive* and before the next directive name. The original *\$i* is the line number of *\$directive* and the loop will iterate through all lines after this line. If the line does not include ‘\*’, execute “*\$inputstring .= \$this->inputdata[\$i]*” which means add *\$this->inputdata[i]* behind the *\$inputstring* (details of the *strpos* function are to be found in Chapter 6). If there is a ‘\*’ in the iterated line, return *\$inputstring* and stop this loop. In this case, this function is to store all lines after

CHARACTER TYPES and before the next line including '\*' to a string variable (*\$inputstring*).

```
private $characterTypes = array();

/**
 * Function that reads in the character types data and assign each character number/ character
 * type pair to the characterTypes property.
 * parameter
 * $directive: string containing the name of the directive find
 * $old: the $inputstring which got from function getDirectiveData
 * trim $old and explode it with space
 * explode $old with " " putting character number as class property $characterTypes index whi
 * le character type pair as value
 */
private function getCharacterTypes($directive)
{
    $old = $this->getDirectiveData($directive);
    $old = trim($old);
    $old = explode(" ", $old);
    foreach($old as $value)
    {
        $parts = explode(",", $value);
        $this->characterTypes[$parts[0]] = $parts[1];
    }
}
```

Create a property called *\$characterTypes* of the **DeltaParser** class. Call the *getDirectiveData* function and *\$old* will store the string returned by *\$inputstring* in the last function. There are some spaces before the character types, so the *trim()* function can be used to trim white space from the beginning and end of the string. Then use the *explode()* function to split the string into an array using the space character to indicate where to break the string such that the elements of the array include a character number and a character type. Then iterate through this array and explode each element with ',' to an array called *\$parts*. Use the 0 element of *\$parts* as an index to *\$this->characterTypes* and the 1 element as the value of *\$this->characterTypes*. What results from this function is that the index of each entry in the new array *\$this->characterTypes* corresponds to the character number and stores the corresponding value.

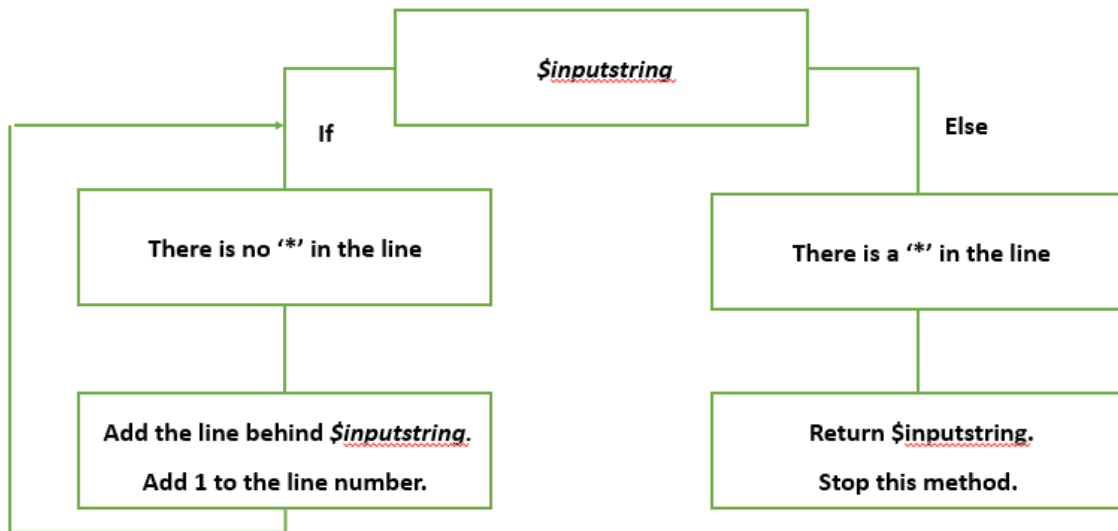


Figure 16. The logical diagram of the method `getDirectiveData()`

## 10. Read a character

The method `readNextCharacter()` aims to read a character description for parsing in the next method (Figure 17). And there is another method used to test this one.

### 10.1 Read a character description

```
/**
 * Function that extract a new array from the input data array
 * parameter
 * none
 * throw a new exception if the first element does not include "#"
 * characters are separated by empty lines, end the loop when meet empty line
 * if the previous line has "/", create a new element; else, the line should keep reading until there is a "/"
 */
private function readNextCharacter()
{
    if (strpos($this->inputdata[$this->readpos], "*") !== false)
    {
        //meeting "*" means all characters have been read and the function should stop
        echo "All characters have been read.";
        return $this->nextcharacter = array();
    }
    elseif(strpos($this->inputdata[$this->readpos], "#") !== false)
    {
        //create a new array for each character
        $i = 0;
        $this->nextcharacter = array();
        while (($this->inputdata[$this->readpos] !== " ") && ($this->inputdata[$this->readpos] !== ""))
        {
            //each character is separated from the next with an empty line
            if((strpos($this->inputdata[$this->readpos], "#") ||
(preg_match("/.+\\$/", $this->inputdata[$this->readpos-1]))) !== false)
            {
                //if the previous line has "/" or the line has "#", a new element should be created; else, the line should keep reading until there is a "/"
                $this->nextcharacter[$i] = $this->inputdata[$this->readpos];
                $this->readpos++;
                $i++;
            }
            else
            {
                $this->nextcharacter[$i-1] .= $this->inputdata[$this->readpos];
            }
        }
    }
}
```



```

        $this->readpos++;
    }
}
$this->readpos++;
return $this->nextcharacter;
}
else
{
    throw new exception ("Invalid syntax has been detected in line".$this->readpos);
}
}

```

The class property *\$readpos* was taken from the last method, which means the line number of the directive 'CHARACTER DESCRIPTIONS' is now known. This method aims to extract each character description in the array separately from the input data. The first element of each character description has the same format, '# character number. structure name <attribute or character name>/ units of measure/'. The units of measure only exist in numeric characters, i.e. those described with numbers. There are also states below the state-based character description. What this method needs to achieve is to extract all lines of a character and store the character description statement and each state in an array as different elements. Each character is separated from the next character by an empty line or space. Therefore, using an empty line or space as the symbol of the start of a new array is a good way to extract a character from the whole input data.

There are three situations in this method, correct and normal character descriptions, the ending of character descriptions and the statement with the wrong format. At the end of character descriptions, there is the next directive '\* ITEM DESCRIPTIONS'. Therefore, I use '\*' as the symbol of the ending. When meeting '\*', all characters have been read and the function should stop. The first *if()* group aims to stop this method. The function *strops()* would be used to find the position of '\*', if the '\*' could be found in the reading statement, print out 'All characters have been read' and return an empty array in the class property *\$nextcharacter* to stop this method.

The *elseif()* group is for all normal character descriptions. Firstly, initializing the class property *\$nextcharacter* as an empty array cleans out the other characters to ensure that each character is in an array. The while loop helps to stop the loop with an empty line or a spaceline. However, an entry in the DELTA character definition could span multiple lines in the input file and would be terminated with a slash. If the line from the input does not end with a slash, the input should be read onwards until there is a slash. And all lines before the

ending slash should be joined together before parsing them. If the reading line contains a '#' or the previous line ends with a slash, a new element should be created and use \$i as the key to repeat the loop. Else, which means the reading line is not finished, so the next line should be appended to the line until there is a slash.

The function *strpos()* may be used to judge whether this line contains a character. But *strpos()* function cannot determine whether the previous line end with a '/'. The regular expression could use characters to express the ending symbol. The regular expression is a pattern matched against a subject string from left to right. Most characters stand for themselves in a pattern and match the corresponding characters in the subject. When using a regular expression, it is required that the pattern is enclosed by delimiters. The most usual delimiter is the slash (/) which is used in this method, but actually, a delimiter could be any non-alphanumeric, non-backslash and non-whitespace character. '.' could match any character except newline, '+' means 1 or more quantifier and '\$' could assert end of the subject or before a terminating newline. More information about meta-characters may be found in <https://www.php.net/manual/en/regexp.reference.meta.php>. Most characters have their special meanings in the regular expression, and it might be confusing to distinguish the real meaning and the special meaning of a character. The backslash character (\) called escape sequence would take away any special meaning that character might have (<https://www.php.net/manual/en/regexp.reference.escape.php>). Therefore, the meaning of the regular expression '/.+\/\$/' in the method mentioned above is a line ending with a slash. The function *preg\_match()* is to search subject for a match to the regular expression given in pattern (<https://www.php.net/manual/en/function.preg-match.php>).

The last group (*else()* group) is used when there is something wrong in the syntax. A new exception should be thrown out and show the error message 'Invalid syntax has been detected in line (line number)' to users.

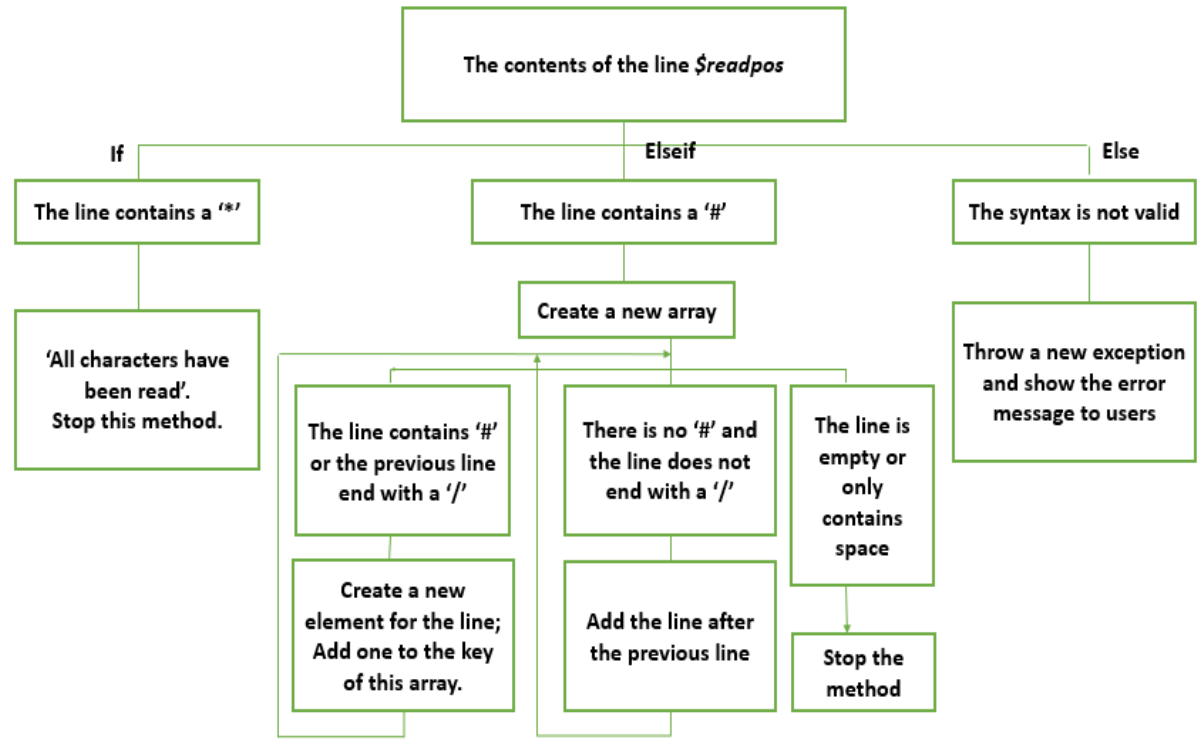


Figure 17. The logical diagram of the method *readNextCharacter()*

## 10.2 Test method *readNextCharacter()*

```

/**
 * Function to test readNextCharacter function
 **/
private function testReadCharacters()
{
  //test readNextCharacter function
  while(sizeof($this->readNextCharacter()) != 0)
  {
    print_r($this->nextcharacter);
  }
}
  
```

This method aims to test the *readNextCharacter()* method. When the size of the method *readNextcharacter()* is not 0, it means that there is still character description to be read, print out the class property *\$nextcharacter* to check whether each character has been extracted as an array.

## 11. Parse all characters and store them in Padme database

The method *parseCharacter()* aims to parse all characters and insert the information of character descriptions into the correct table of the Padme database (Figure 18)

```
/**
 * Function that insert the information of character description into data base
 * parameter
 * none
 * iterate through all character descriptions with while loop to parse all structures, characters
 and states and store them in Padme database
 * insert structure name into table structures and get the structure id when the structure has no
 t been dealt with; or link the existing structure to the old record
 * get the class property $this->structure with structure name as index and structure id as valu
 e
 * insert character name, structureid, charactertype and units (only for characters with RN and
 IN types) into table characters and get the character id
 * get the class property $this->character with character number as index and character name
 as value, and instantiate a new object of class Character
 * insert state name and characterid into table states and get the state id
 * call the function addState of Character class
 **/
private function parseCharacter()
{
    while(sizeof($this->readNextCharacter()) !== 0)
    {
        //loop through all character descriptions
        if(!((preg_match("/#\d+\.(.+)\<(.+)\>\v(.+)\v/", $this->nextcharacter[0], $matches)) || (preg
_match("/#\d+\.(.+)\<(.+)\>\v/", $this->nextcharacter[0], $matches) || (preg_match("/#\d+\.(
.+)\v/", $this->nextcharacter[0], $matches))))
        {
            //throw exceptions which cannot corresponding to formats
            throw new exception("Syntax error: " . $this->nextcharacter[0] . " is not a valid start to a
character definition");
        }
        $currentcharacternumber = $matches[1];
        $currentstructure = $matches[2];
        $currentcharactername = $matches[3];
        $currentunits = $matches[4];
        //if the structure id has already existed, get the Padme id of the structure from structures cla
ss property.
        $structureid = $this->structures[$currentstructure];
        if(!$structureid)
        {
            //if the structure id has not been dealt with, insert a new record
            $structurenamesql = "INSERT INTO structures(name) VALUES(?)";
            $p = $this->conn->prepare($structurenamesql);
```

```

    $p->execute(array($currentstructure));
    //get the maximum id number from SQL database as the structure id
    $structureidsql = "SELECT max(id) FROM structures";
    $p = $this->conn->prepare($structureidsql);
    $p->execute();
    $structureid = $p->fetchColumn();
    //using the structure name as $this->structure key and structure id as the value
    $this->structures[$currentstructure] = $structureid;
}
//insert character name, structureid, character type and units into padme
$characterinsertsql = "INSERT INTO characters (name,structureid,type,units) VALUES(?,
?,?,?)";
    $p = $this->conn->prepare($characterinsertsql);
    //use $this->character type to look up the character type with $currentcharacter number
    $currentcharacter type = $this->character types[$currentcharacter number];
    //if there is not a match in $this->character types default to UM
    if(!$currentcharacter type)
    {
        $currentcharacter type = 1;
    }
    else
    {
        //all character types are stored in padme in integer, so switch "UM" "OM" "IN" "RN" to 1,2,
        3,4 separately;
        switch($currentcharacter type)
        {
            case "RN": $currentcharacter type = 4; break;
            case "IN": $currentcharacter type = 3; break;
            case "OM": $currentcharacter type = 2; break;
            case "UM": $currentcharacter type = 1; break;
        }
    }
    $p->execute(array($currentcharacter name,$structureid,$currentcharacter type,$currentunits
));
    //find out the character id from table characters using the maximum id
    $characteridsql = "SELECT max(id) FROM characters";
    $p = $this->conn->prepare($characteridsql);
    $p->execute();
    $this->characterid = $p->fetchColumn();
    //instantiate a new object of Character class, so could link states to the corresponding chara
    cter
    $currentcharacter = new Character();
    //Use the character number as the index into the array and set the character object as the v
    alue
    $this->characters[$currentcharacter number] = $currentcharacter;
    for($i=1; $i < sizeof($this->nextcharacter); $i++)
    {
        //deal with all states after a character with "OM" or "UM" type using for loop
        if (preg_match("/(d*)\.(+)\./", $this->nextcharacter[$i], $matches))
        {

```

```

//if the statement could match the pattern, insert name and characterid into database
$states[$matches[1]] = $matches[2];
$statesql = "INSERT INTO states(name,characterid) VALUES(?,?)";
$p = $this->conn->prepare($statesql);
$p->execute(array($matches[2], $this->characterid));
//find out the stateid using max()
$stateidsql = "SELECT max(id) FROM states";
$p = $this->conn->prepare($stateidsql);
$p->execute();
$stateid = $p->fetchColumn();
//call the function addState of Character class
$currentcharacter->addState($i,$stateid);
}
else
{
//throw exception if the syntax does not match the pattern
throw new exception("Syntax error: " . $this->nextcharacter[$i] . " is not a valid definiti
on of a state");
}
}
}
}
}

```

The first aim of the *parseCharacter()* method is to read all characters. As in the previous testing method, while loop will be used for this purpose. The character description contains character number, structure name, attribute or character name and units, but not all character descriptions have all of them. The most complete format is #character number. structure name <attribute or character name>/ units of measure, but attribute or character name is not necessary and units only necessary in numeric characters. Therefore, there are three format types of character descriptions. If the statement cannot match any one of the three, there are errors in this statement, and a new exception will be thrown, showing an error message to users and stopping the method. Then capture subpatterns of the regular expression to get character number, structure name, character name and units of measure.

Subpatterns are delimited by parentheses, which could be nested

(<https://www.php.net/manual/en/regexp.reference.subpatterns.php>). In this method, the parentheses are used to set up the subpattern as a capturing subpattern. When the whole pattern matches, the part of the string that matched the subpattern will be passed back to the caller as elements of the array. This method could get character number, structure name, character name and units storing them in *\$currentcharacternumber*, *\$currentstructure*, *\$currentcharactername*, and *\$currentunitites* separately through subpatterns of regular expressions.

Create a class property called *\$structure* and initialize it as an empty array. This class variable stores a structure name which is stored in *\$currentstructure* as the key and the Padme id of this structure which will be stored in *\$structureid* as the value. Because each structure has a great number of characters, it is easier to use the information of the structure many times when meeting different characters if the Padme id of the structure and the structure name were linked. When the structure id already exist, the id of the structure may be got from the class property *\$structures* directly. When it is the first time meeting the structure id, SQL queries should be used to get the Padme id of this structure.

The SQL INSERT INTO statement is used to insert new records in a table. 'INSERT INTO states(name, characterid) VALUES(?, ?)', 'states' is the table name, 'name' and 'characterid' are column names, and '?' in parentheses after VALUE are values which will be added to the column. Certain values are left unspecified and called parameters showing with '?'. The SELECT INTO statement is like the INSERT INTO statement. The column which is needed to select would come after SELECT and the new table is after INTO while the old table is after FROM, and there could also be a WHERE to add a condition in the statement. In this method, the maximum id needs to be selected from the table 'structures'. All SQL these statements cannot run directly in PHP, *PDO::prepare* would be used to prepare these statements for execution and return statement objects

(<https://www.php.net/manual/en/pdo.prepare.php>) and *PDOStatement::execute* would execute a prepared statement (<https://www.php.net/manual/en/pdostatement.execute.php>). The *PDOStatement::fetchColumn* would help to return a single column from the next row of a result set (<https://www.php.net/manual/en/pdostatement.fetchcolumn.php>). Therefore, in this method, those statements in the *if(!\$structureid)* group aim to insert a structure name into the table 'structures' and get the maximum id as the structure id storing it in the class property *\$structures*.

Then prepare an insert statement to insert character name, structure id, character type and units into the table 'characters'. The types of some characters are empty, and the character type cannot be read directly by Padme, because the empty character type defaults to 'UM' and all character types are stored in Padme in integers. Therefore, if there is no character type, default to 'UM' and switch 'UM', 'OM', 'IN', and 'RN' into 1, 2, 3, and 4 respectively. The prepared SQL statement can then execute, and the information of the numeric characters will have been added into the Padme database. Character id may be found from the table 'characters' in the same way as structure id.

For state-based characters, it is important to link states to the corresponding character. A new class called `Character` is created to store states which will be described in detail in the next paragraph. Instantiating a new object (`$currentcharacter`) of `Character` class would help to link the states to the corresponding character by using the method of `Character` class. This object `$currentcharacter` would be stored in the class property `$characters` as a value while the character number would be the index of the array. All state descriptions have the same format, the regular expression `'/(d*)\.(.+)\//'` means the normal format: state number. state name/. The SQL statements would help to insert state name and character id into the table 'states' and get the state id from 'states'. Then call the function `addState()` of class `Character` to add state number and Padme id into the class. If there is something wrong with the syntax of the statement, error trapping should throw an exception to send an error message to users and stop the method.

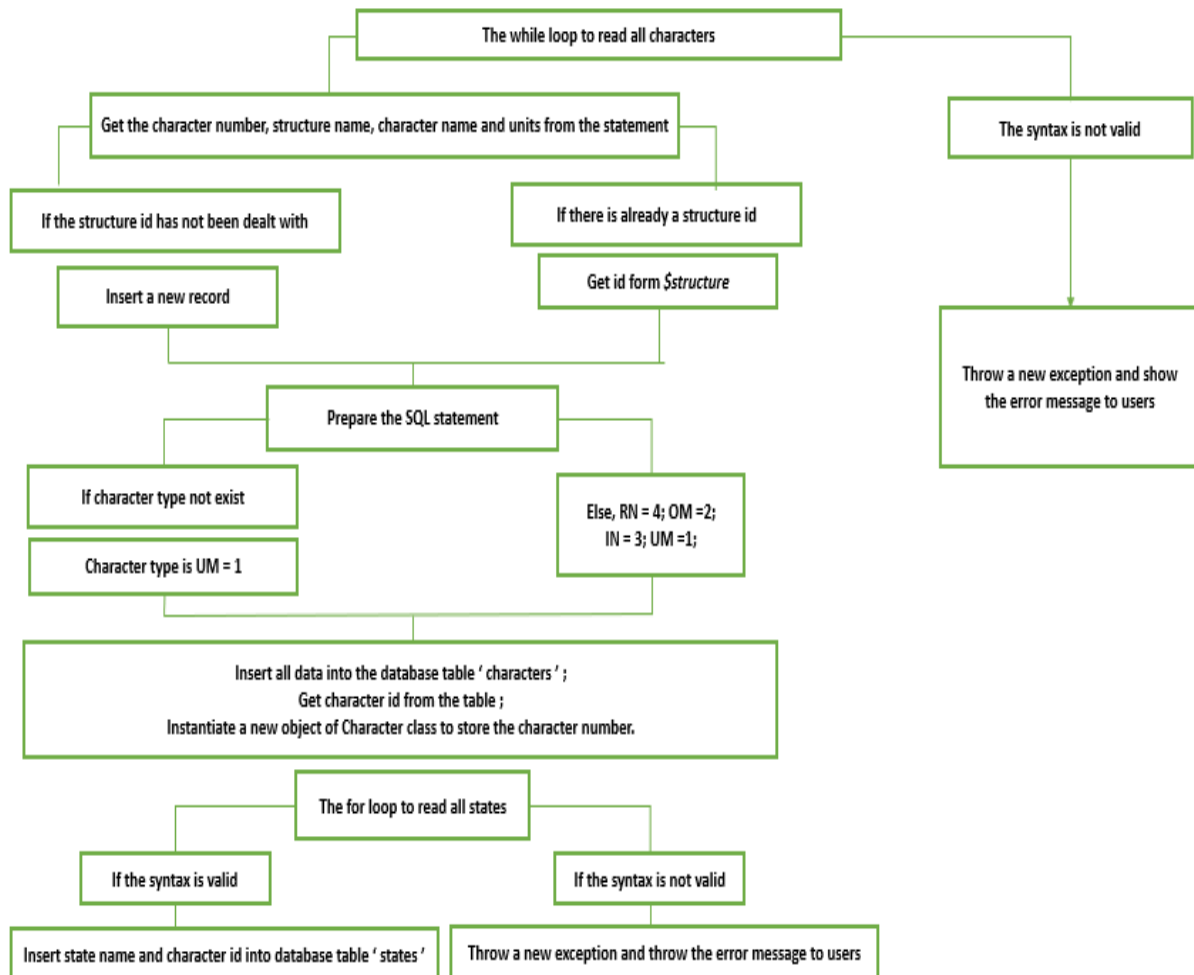


Figure 18. The logical diagram of the method `parseCharacter()`



## 12. Character class

```
class Character
{
    private $states = array();
    /**
     * Function to add new states
     * parameter
     * $id and $padmeid
     **/
    public function addState($id, $padmeid)
    {
        $this->states[$id] = $padmeid;
    }
    /**
     * Fuction to get padmeid of state
     * parameter
     * $id
     * return value
     * $padmeid
     **/
    public function getState($id)
    {
        return $padmeid = $this->states[$id];
    }
}
```

This class is created to link characters and their corresponding states. It contains two public methods, which may be called in the **DeltaParser** class. The first method aims to add new states with id and corresponding Padme id. The parameters of the *addState()* method are *\$id* and *\$padmeid*, *\$id* is the key of the *\$states* array while *\$padmeid* is the value. The second method gets the state Padme id corresponding to the character id in the DELTA file. The parameter of this method is *\$id* while the return value is *\$padmeid*.

## 13. Read a description

This function is similar to the method *readNextCharacter()* explained in chapter 10 and they are all to read information. This part is to read the description of an item (Figure 19) while chapter 10 is to read character data.

```
/**
 * Function to read item descriptions
 * parameter
 * none
 * if there is "*" in the line, all items have been read
 * the first line of each item is the Latin name, store the latin name in $itemnames
 * all lines after the first line are character description, ignore contents in angle brackets and store each character into an element of array $itemcharacters
 */
private function readNextDescription()
{
    //initialize $this->readpos with function findDirective
    $this->readpos += 1;
    if(strpos($this->inputdata[$this->readpos], "*") !== false)
    {
        //all characters have been read when meeting *END
        echo "All items have been read";
        return false;
        exit;
    }
    else
    {
        $i = 0;
        //each item is seperated from the next with an empty line
        while(!empty($this->inputdata[$this->readpos]) && ($this->inputdata[$this->readpos] != " "))
        {
            //iterate through all description of each item
            if(preg_match("/^\ \ +$/", $this->inputdata[$this->readpos], $matches))
            {
                throw new exception("Syntax error: more than one spaces have been detected in line". $this->readpos);
            }
            elseif(strpos($this->inputdata[$this->readpos], "#") !== false)
            {
                //the first line of each item is the Latin name, "#item number. latin name", storing it in class property $description
                $description[$i] = $this->inputdata[$this->readpos];
                $this->readpos++;
            }
        }
    }
}
```

```

        $i++;
    }
    else
    {
        //put all description into the class property $description
        $description[$i] .= $this->inputdata[$this->readpos];
        $this->readpos++;
    }
}
//get the item name from the first line of $description
preg_match("#\d*\.(+)\|/", $description[0], $match);
$this->itemnames = "Elaeocarpus ". trim($match[1]);
//delete all contents in angle brackets
$description[1] = preg_replace("<.+?>", "", $description[1]);
//explode the description with space and store them in the class property itemcharacters
$this->itemcharacters = explode(" ", $description[1]);
}
}

```

Before using this method, the method *findDirective()* should be called again in the method *parse()* and the parameter should be 'ITEM DESCRIPTIONS'. From the method *findDirective()*, the line number of 'ITEM DESCRIPTIONS' can be found and stored in the class variable *\$readpos*. The property *\$readpos* obtained should be incremented by one so as to ignore the line of the directive name. As in the method *readNextCharacter()*, all items would have been read when meeting the next directive name line, so a message 'All items have been read' will be displayed to users and this method would exit.

A while loop would be helpful to read item descriptions. Whenever encountering an empty line or a line only with space the loop should be stopped, because most items are separated from the next one with an empty line while others are separated by the line with space. So, the condition of the while loop should not be an empty line or line with space. If there is a '#' in the statement, which means it is the first line of each item description including the item number and Latin name, this statement should be stored in the variable *\$description* as its first element and both the *\$readpos* and the index of this array should be added one. If there is no '#', which means it is one line of the whole description, all description should be in one element of *\$description*. Therefore, put all description lines together in one element. When trying to run this method, there are some syntax errors found. Lines with more than one spaces are detected. Adding error trapping to the while loop should be used to remind users to modify the original file.

After reading the item description, some operations should be done for easier parsing. Firstly, the Latin name is *Elaeocarpus* and the short item name got from the item descriptions in the Padme database. Therefore, get the item name from the first line of item descriptions with regular expression, complete the Latin name by adding ‘*Elaeocarpus*’ before item names and ignore useless spaces using the function *trim()*. Secondly, one item description contains a large amount of information of many characters and there are also some unimportant notes in angle brackets. It would be hard to parse item descriptions with these data. To solve these problems, the *preg\_replace()* function would be used to delete all contents in angle brackets and the *explode()* function would explode the whole description with spaces storing them in different elements of the class property *\$itemcharacters*.

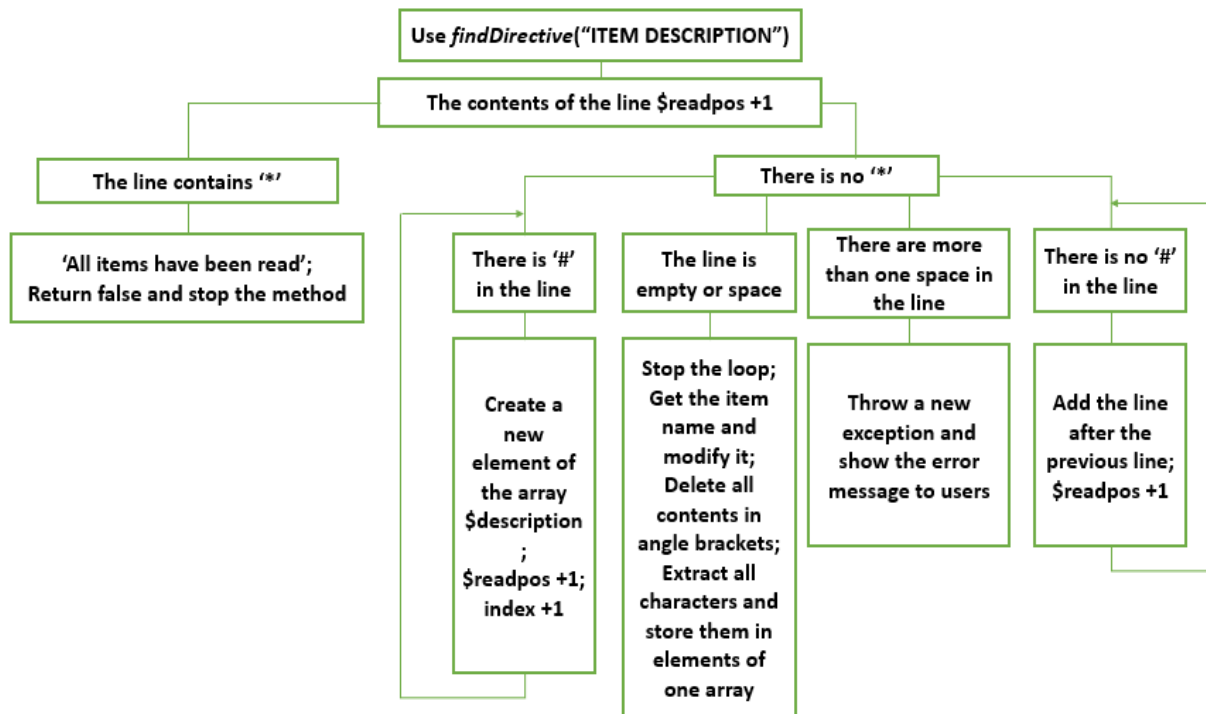


Figure 19. The logical diagram of the method *readNextDescription()*

## 14. Parse all descriptions and store them in Padme database

This chapter is similar to chapter 11, they all parse data read by the previous method. This function links different tables of the Padme database such as reference, literature\_records, latin\_name and litrescores through specific numbers (Figure 20). And Character class would be called to link states to the corresponding character (Figure 20). Following chapters 10-14 will result in a completed database with clear affiliation.

```
/**
 * Function to parse descriptions of each item
 * parameter
 * none
 * iterate through all item descriptions with while loop
 * get id from latin_name table of the padme database
 * get the reference from references table of the padme database
 * insert the id and reference into literature_records table and get id from it
 * insert literature id, character id, minvalue and maxvalue or state id into table litrescores
 */
private function parseDescriptions()
{
    //read all items using while loop
    while($this->readNextDescription() !== false)
    {
        //get id corresponding the epithet from table latin_names
        $latinnamesql = "SELECT id FROM latin_names WHERE full_name = '$this->itemnames
";
        $p = $this->conn->prepare($latinnamesql);
        $p->execute();
        $latinnameid = $p->fetch();
        //get references from the table references, all descriptions are from the same reference "Co
ode, M. Coded descriptions of Elaeocarpus." which id is 1
        $reference = 1;
        //insert the id got from table Latinname and the reference into table literature_records, and
get id
        $determinationsql = "INSERT INTO literature_records(determination,reference) VALUES
(?,?)";
        $p = $this->conn->prepare($determinationsql);
        $p->execute(array($latinnameid[0], $reference));
        $literatureidsql = "SELECT id FROM literature_records WHERE determination = '$latinn
ameid[0]";
        $p = $this->conn->prepare($literatureidsql);
        $p->execute();
        $literatureid = $p->fetch();
        //insert literature id, character id, minvalue and maxvalue or state id into table litrescores.
```

```

$litrescoresql = "INSERT INTO `litrescores`(`stateid`, `litrecid`, `characterid`, `scoregroupid`, `minvalue`, `maxvalue`) VALUES(?,?,?,?,?,?)";
$p = $this->conn->prepare($litrescoresql);
//scoregroupid of all character description is the same 1
$scoregroupid = 1;
foreach($this->itemcharacters as $value)
{
    //initialize minvalue, maxvalue and stateid
    $minvalue = NULL;
    $maxvalue = NULL;
    $stateid = NULL;
    //iterate through all characters description of each item
    preg_match("/(\d*),(.+)/", $value, $matches);
    $characterid = $matches[1];
    $currentcharacter = $this->characters[$characterid];
    if($this->charactertypes[$matches[1]] == "RN" or $this->charactertypes[$matches[1]] == "IN")
    {
        //if the character type is RN or IN, get the first number as minvalue while the second one as maxvalue
        preg_match("/(\d+\.?\d?+)\-?(\d+\.?\d?)?/", $matches[2], $matches1);
        $minvalue = $matches1[1];
        $maxvalue = $matches1[2];
        if($maxvalue == "" or $maxvalue == null)
        {
            $maxvalue = $minvalue;
        }
    }
    try
    {
        $p->execute(array($stateid, $literatureid[0], $characterid, $scoregroupid, $minvalue, $maxvalue));
    }
    catch(PDOException $e)
    {
        throw $e;
    }
}
else
{
    //if the character type is UM or OM, get the statenumber and use the getState method of class Character to get stateid
    $i = 1;
    preg_match("/(\d+)\(\/(\d+)*\)?/", $matches[2], $matches2);
    while($i < sizeof ($matches2))
    {
        $statenumber = $matches2[$i];
        $stateid = $currentcharacter->getState($statenumber);
        $i = $i+2;
    }
}

```

```
    $p->execute(array($stateid, $literatureid[0], $characterid, $scoregroupid, $minvalue, $
maxvalue));
    }
  }
}
}
```

The purpose of this method is to parse descriptions of all items. The first thing is to read all item descriptions with a while loop. The Padme database has a great number of tables to store all information. In order to link different tables, some columns exist in different tables and users could link them together through the repeated columns. There are two columns called determination and reference in the table *literature\_records*, which store the same information as id in the table *latin\_name* and *bookorjournal* in the table *references* respectively. The two columns would be extracted from the table *latin\_name* and *references* with SELECT INTO statement and insert into the table *literature\_records* with INSERT INTO statement to link these three tables. Similarly, the literature id could link the table *literature\_records* and the table *litrecscores* in the same way.

The INSERT statement to insert literature id, character id, minimum value and maximum value or state id has been prepared, but there are different types of characters to be read. There is no state id in the information of numeric characters while there is no minimum value and maximum value in the data of state-based characters. Therefore, iterate all descriptions of an item, initialize minimum value, maximum value, and state id to empty, and parse the information of different types of characters in different ways. Before parsing them separately, extract the character id through regular expression. Using the character id and the class property *\$characterTypes*, the type of character will be known.

For numeric characters, the normal format is ‘the minimum value - the maximum value’, which could be matched with regular expression. However, some descriptions have only one number and this number would be added only in the minimum value grid. It is a requirement in Padme that the minimum and maximum values should be set equal if only the minimum value is expressed in the DELTA file, so make the *\$maxvalue* equal to the *\$minvalue* if the variable *\$maxvalue* is empty or space. Then execute the prepared statement to insert all information relating to numeric characters into the table *litrescores*. Using error trapping will help users to catch errors in the process of inserting.

For state-based characters, ‘state number/ state number’ is the regular way to express the possible state choices. It is possible for an item to have many states of one character, so a while loop is needed to read them all. Then execute the prepared statement to add all information relating to state-based characters into the table *litrescores*.

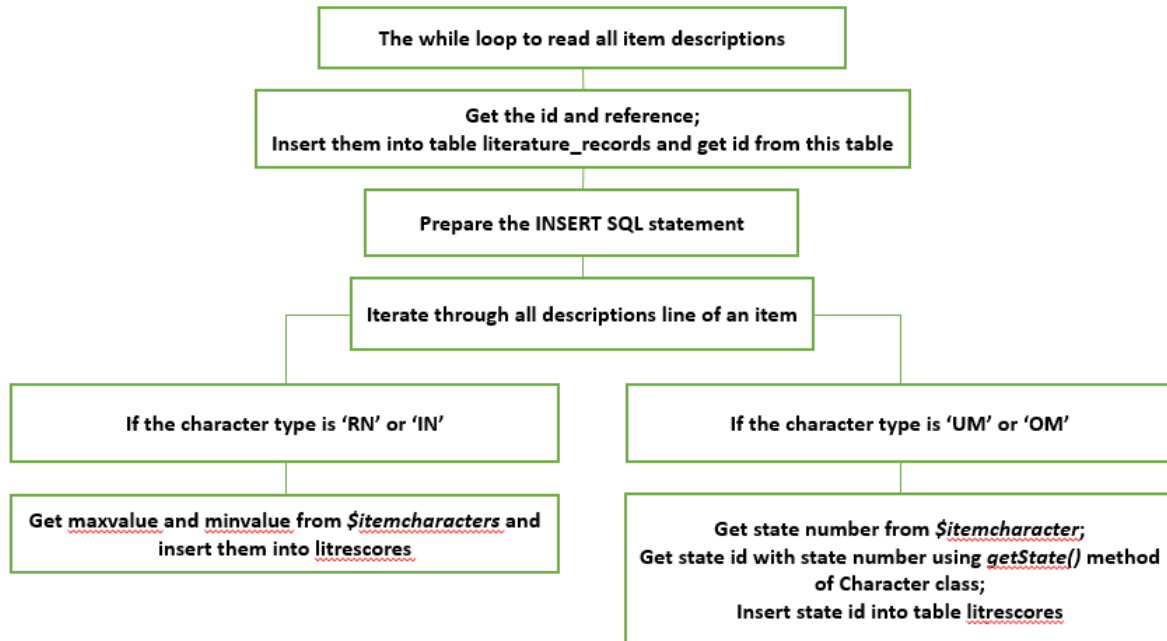


Figure 20. The logical diagram of the method `parseDescription()`



## 15. Test and modify the interactive key

After parsing the DELTA file and storing in the Padme database, a new website needs to be set up to publish the interactive key of *Elaeocarpus*. This part was achieved by Dr Pullan's previous work, and it will not be described too much in this study. The interactive key was shown in <https://padme.rbge.org.uk/Elaeocarpus/key>. When the interactive key was created automatically, it is necessary to test the key to find places of errors and modify codes to complete and correct the key.

### 15.1 Manual checking

The testing is mainly based on manual checking. Observe characters of specimens and living materials and use the key to identify the plant. Compare the result got from the interactive key and the real taxon of this plant, the key worked if the result is the same as the real taxon, alternatively, compare details of two different taxa character descriptions and find the error place. Due to the special Covid-19 quarantine situation, only pictures could be used for testing. Mr Coode has provided recommended specimens and easy identifying characters and cooperate with me about the testing.

The specimen pictures tested came from JSTOR Global Plants, which has been shown in Appendix 3. These pictures are not clear enough, so it is hard to identify taxa with those small characters. In this part, I mainly concentrate on those characters which are easy to observe, such as leaf characters including shape, width, length, margins and venations; inflorescence including length, disposition, position and branched; and flower characters including the numbers per inflorescence.

The first specimen I used is *Elaeocarpus myrtooides* (Figure 21) with special leaf characters. This specimen has small leaves which are special in *Elaeocarpus* items. Therefore, although only a part of leaves characters can be seen from the specimen pictures, the interactive key still gets the right result. From the specimen, most characters of leaves can be got such as the arrangement of leaves, the length and width of the blade and the shape of the blade. The order used in the interactive key is shown in Figure 22. Due to the special leaves of this specimen, only 4 steps are needed to get the result. The result got from the interactive key is

*Elaeocarpus myrtooides* sp. *myrtooides* which belongs to *Elaeocarpus myrtooides*. So, the interactive worked with this specimen.



Figure 21. *Elaeocarpus myrtooides* specimen

## Interactive key

Taxa remaining: 1    Retain taxa unscored for a characters: Yes    Elimination threshold: 0    Reset

Click on the grey bar below to select a plant part to describe:    *Elaeocarpus myrtooides* subsp. *myrtooides* Misses:0    Unscored characters:0

**Anthers**

Select an attribute to score:

Current description

- Most Leaves (Including Acumen if Any): blade-length 1 cm long
- Most Leaves: blades-width 0.6 cm wide
- Leaves: leafblade shape obovate
- Leaves: arrangement spirally arranged

Figure 22. The process of identifying *Elaeocarpus myrtooides* with the interactive key

The second specimen (Figure 23) is a nice one with clear leaf and some inflorescences. Although many clear characters can be observed from this picture, this specimen did not get to a single result through the interactive key (Figure 24) because some characters especially those used in the later stage of identification were not able to remove any item from chosen items. In this picture, a lot of information about the leaves, inflorescence and flowers can be observed such as the width, length, margins, shape, midrib and main venation of leaves; position, length and disposition of inflorescence; and numbers and disposition of flowers. However, some characters would not change anything and cannot be used. For example, if the character 'flowers number per inflorescence' has been used, the character 'flowers disposition' will be useless. If there are living materials, this problem will be solved easily. Luckily, the right result is among the remaining items, so if the character descriptions were more complete, the interactive key might work successfully.



Figure 23. *Elaeocarpus petiolatus* specimen

## Interactive key

Taxa remaining: 3    Retain taxa unscored for a characters: Yes    Elimination threshold: 0    Reset

Current description

- Midrib Of Leaves: prominence beneath prominent beneath ✕
- Inflorescences: length 9 cm long ✕
- Flowers: Number per inflorescence 10 ✕
- Inflorescence: branched or not simple ✕
- Leaves: base tapering towards a narrowly rounded base ✕
- Leaves: margins entire to more or less entire ✕
- Most Leaves (Including Acumen If Any): blade-length 16 cm long ✕
- Most Leaves: blades-width 6 cm wide ✕
- Leaves: leafblade shape obovate ✕

Elaeocarpus badius Misses:0 Unscored characters:0  
Elaeocarpus coloides Misses:0 Unscored characters:0  
Elaeocarpus petiolatus Misses:0 Unscored characters:0

**Figure 24. The process of identifying *Elaeocarpus petiolatus* with the interactive key**

As for the second specimen, the third (Figure 25 & 26) and the fourth (Figure 27 & 28) did not get the result, but the right taxon is covered by the chosen items. From these three unsuccessful specimens, it is possible to reach certain conclusions about the use only of characters visible on specimen pictures in *Elaeocarpus* items identification. Firstly, leaf characters are the easiest to be observed in specimen pictures because most *Elaeocarpus* items have big leaves, and they are also helpful in the process of identification. The width and length of the blade; the shape of the leaf blade, apex, and base; and margins are helpful and can make great progress of the identification. But the main venation of leaves and the midrib did not help so much, especially in the later process of identification. Then, if there is some information on flowers and inflorescences, it will be helpful for identification. The length of inflorescence, flowers arrangement in raceme and the number of flowers per inflorescence are useful characters in identification. However, the disposition of the inflorescence is useless in *Elaeocarpus* identification because almost all *Elaeocarpus* items in this project can have solitary inflorescences and only a little part have inflorescences in pairs. As mentioned above, if the character 'flowers number per inflorescence' has been used, the disposition of flowers is also a useless character. Also, other characters are important but cannot be observed in pictures clearly.



Figure 25. *Elaeocarpus stipularis* var. *nutans* specimen

## Interactive key

Taxa remaining: 8    Retain taxa unscored for a characters: Yes    Elimination threshold: 0    Reset

tapering toward a narrowly cordate base.

**Current description**

Leaves: base broadly rounded at base

Leaves: apex shape acute at apex (50-80 degrees)

Midrib Of Leaves: prominence beneath prominent beneath

Main Venation Of Leaves: prominence beneath; always less prominent than the midrib prominent beneath

Most Leaves (Including Acumen If Any): blade-length 14 cm long

Most Leaves: blades-width 4 cm wide

Flowers: Number per inflorescence 20

Inflorescences: length 10 cm long

- Elaeocarpus calomala* subsp. *calomala* Misses:0 Unscored characters:0
- Elaeocarpus clementis* var. *clementis* Misses:0 Unscored characters:0
- Elaeocarpus cupreus* var. *cupreus* Misses:0 Unscored characters:0
- Elaeocarpus ledermannii* var. *ledermannii* Misses:0 Unscored characters:0
- Elaeocarpus murudensis* Misses:0 Unscored characters:0
- Elaeocarpus sepikanus* Misses:0 Unscored characters:0
- Elaeocarpus stipularis* var. *nutans* Misses:0 Unscored characters:0
- Elaeocarpus valetonii* var. *valetonii* Misses:0 Unscored characters:0

Figure 26. The process of identifying *Elaeocarpus stipularis* var. *nutans* with the interactive key



Figure 27. *Elaeocarpus cumingii* specimen

## Interactive key

Taxa remaining: 6    Retain taxa unscored for a characters: Yes    Elimination threshold: 0    Reset

Click on the grey bar below to select a plant part to describe:  
**Flowers**

Select an attribute to score:  
**arrangement in raceme**

alternate, spiral, sometimes almost whorled:

Current description

Leaves: base cuneate at base

Most Leaves (including Acumen if Any): blade-length 7.5 cm long

Most Leaves: blades-width 4 cm wide

Flowers: Number per inflorescence 21

Inflorescences: length 13 cm long

- Elaeocarpus cumingii Misses:0 Unscored characters:0
- Elaeocarpus floribundus Misses:0 Unscored characters:0
- Elaeocarpus murudensis Misses:0 Unscored characters:0
- Elaeocarpus pedunculatus Misses:0 Unscored characters:0
- Elaeocarpus stipularis var. castaneus Misses:0 Unscored characters:0
- Elaeocarpus vaeletonii var. vaeletonii Misses:0 Unscored characters:0

Figure 28. The process of identifying *Elaeocarpus cumingii* with the interactive key



## 15.2 Code checking

Some problems took a long time to find where the problem occurred because descriptions shown in the Padme system and the DELTA file are different and hard to compare. And due to the special Covid-19 situation, it is not convenient to observe real living materials and only observing specimens with pictures is not accurate and complete. In order to reduce the time spent on manually finding the location of the problem, increase the specimen samples size and improve the accuracy of testing, using code to translate the DELTA file into an easy reading file will be helpful and convenient for testers comparing. This is just modifying the output into a simple and readable English file. It will be easy to see whether a character or a state in the input DELTA file which had been assigned to an item or a character in a description matched the output information. Problems like the state number problem will be easy to find and fix.

The **DeltaTester** class (called tester.php) has similar logic as **DeltaParser** class (Appendix 4) except different output ways. As **DeltaParser** class, **DeltaTester** class also reads and parses both character and item descriptions. But **DeltaTester** class print out all information about characters and items which is different from inserting all data into the MySQL database, the way of **DeltaParser**. Through this class and a file instantiating a new **DeltaTester** object called elaeocarpustest.php (Appendix 5), clear and readable results will be obtained, which covers item name, character number, character name, structure name, range or state name and number in English. The result in the terminal is not easy to compare with the Padme system, so use '> filename.txt' to create a .txt file which has been attached with this dissertation (called elaeocarpustest.txt).

This file will be easy to compare with both the raw DELTA file and the result in the Padme system. If there is something different between the raw DELTA file and the .txt file, there should be some errors in the parsing code. Because the logic of parsing code and testing code are similar, the error in the testing code should exist in the parsing code too. Find the specific error according to the position of the difference between two files and modify it. If there is something different between the .txt file and the result in the Padme system, some errors should appear in the SQL statement. Output ways are different in parsing code and testing code and the output way of the testing code is print out with little errors. So, if outputs are different in two files, that means the output way of the parsing code, SQL database, has some errors. Only checking the SQL statement to find errors will save more time than checking the whole file.

### 15.3 Problems and solutions

Some problems were found with the combining of manual testing and code testing. The first thing is that no units were shown on the numeric characters, which made identification confusing. If there are no units, it might make using some numeric characters like the length, height, and width, difficult. The reason for this problem appears in the process of publishing the Padme database. It is not a challenge, just adding the units will solve this problem.

The second problem is that if the remaining taxa become 0 due to a character, all taxa meet requirements which have been shown in the last step disappeared when returning to the last step. This problem also appears in some accident operations. It will create trouble for users; all steps will be restarted if there is a slight mistake. To solve this problem, all codes have been checked, but no problems were found. Then I have tried another browser and it worked. The Padme system works better with the Firefox browser rather than Chrome.

The third problem is a specific Padme problem that numeric characters with single values will be identified as long as the given value is greater than the single value, but it should be identified only when the given value is equal to the single value. That is because of the way Padme internally stores its data and I have put the single value only in the minimum value. If only the minimum value is specified, this implies that the value for the taxon can be greater than or equal to the value stored in the score. Therefore, if the value for the taxon needs to be equal to the value stored in the score, both the maximum value and minimum value should be specified.

The fourth problem is that some states are ignored. But in the DELTA file, the information of states is complete. This problem caused by the error of parsing coding shown below (belongs to the method *parseDescriptions()*).

```
$i = 1;
preg_match("/(\d+)(\(\d+\))*?/", $matches[2], $matches2);
while($i < sizeof ($matches2))
{
    $statenumber = $matches2[$i];
    $stateid = $currentcharacter->getState($statenumber);
    $i = $i+2;
    $p->execute(array($stateid, $literatureid[0], $characterid, $scoregroupid, $minvalue, $
maxvalue));
}
```

The regular expression “`/(\d+)(\(\d+\))*?`” can only read the first and the last state if there are more than two states. For example, if the possible state numbers of a character are 3 or 4 or 5,



they are shown in the DELTA file as 3/4/5. But with this regular expression, only 3 and 5 can be read while the state 4 will be ignored. I have tried two different way to solve this problem. The first method is changing the regular expression to “/(\d+?)(?:\|\$/)”. This regular expression aims to look for substrings starting with a number and terminating with a ‘/’ or the end of the string. The output of this regular expression will be like:

```
Array
(
  [0] => Array
    (
      [0] => 6/
      [1] => 7/
      [2] => 8/
      [3] => 9/
      [4] => 10
    )

  [1] => Array
    (
      [0] => 6
      [1] => 7
      [2] => 8
      [3] => 9
      [4] => 10
    )
)
```

So, statements for previous regular expression is not suitable. All statements should be modified. The second way is using the function to *explode()* explode the statement with ‘/’ directly. This way is easier than regular expression due to the confusing capture of the regular expression. When modifying codes, I found that it would be better to add an error trapping here to solve problems brought about by mistakes in the DELTA file. And if the state number in the description does not match any of the states defined in the character, the error trapping would send message to users. The coding is shown below:

```
$statearray = explode ("/", $matches[2]);
$i = 0;
while($i < sizeof($statearray) && $matches[2])
{
  $statenumber = $statearray[$i];
  $stateid = $currentcharacter->getState($statenumber);
  $i++;
  try
  {
```

```
    $p->execute(array($stateid, $literatureid[0], $characterid, $scoregroupid,
$minvalue, $maxvalue));
}
catch(PDOException $e)
{
    throw $e;
}
}
```

While fixing the issues in the online interactive key, some issues in the parsing code required correcting. Firstly, truncate statements did not work successfully. All truncate statements were put in a single query, which can work fine in MySQL. However, the PDO query mechanism cannot support multiple queries in the query string, and it will only execute the first query before the first semi-colon. To solve this, the query strings were put into an array and iterated through this array when executing each query in turn. The similar situation also exists in other PDO queries.

## 16. Discussion and conclusion

This study has parsed the *Elaeocarpus* file and provided a way to parse the DELTA file. The DELTA file has provided the full information of characters and items of the individual plants, but it is hard to compare different datasets (Pullan *et al.*, 2005). The Padme system can create an interactive key to solve this problem and provide an easier way for identification.

### 16.1 Project summary

There are four tasks in this project as shown in the Introduction. The first task is to create a new Padme instance of *Elaeocarpus* and insert the Latin names into the database. This task is the first step of this project to link the *Elaeocarpus* file to the Padme system through Latin names of items and it is achieved by inserting the Latin name of each *Elaeocarpus* item one by one manually.

The second task is to edit the DELTA data and bring it to the Padme system. This is the dominant part of this project. It used two classes, **DeltaParser** class and **Character** class, to parse the DELTA data in the el28.dat file. In this part, PHP statements are used to parse all information from *Elaeocarpus* items in the DELTA file while SQL statements are used to insert or extract data from the Padme database.

The third task is to set up a new Padme website to publish the interactive key of *Elaeocarpus*. This part is mainly based on the previous work of Dr Pullan. The Padme database is linked to the new website and creates an interactive key to *Elaeocarpus* automatically with the data from the Padme database.

The last part is to test the result. This part is achieved mainly by manual checking. It needs plenty of time to test whether the key works. To lessen the pressure of this part, there are some codes to translate the DELTA file into English, which will help testers to read and check. After rigorous testing, some problems with the interactive key were found and they have all been solved after modifying scripts.

### 16.2 Advantages of the interactive key over the traditional key

An interactive key has the same function as a traditional (dichotomous) key, which includes plenty of reliable information for identification. They all provide a way for users to identify

the correct taxon to which a plant belongs with character descriptions. Multi-access keys, especially interactive keys which work on a computer have many advantages.

Firstly, users may choose any information about the identified specimen in any order. It is not necessary to follow a fixed order. If there is a structure with an unclear or confusing character, we can ignore it and select those clear characters with a confirmed decision when using an interactive key. Secondly, the computer can provide other helpful and useful data and guidance on selecting the next character. Once a character had been selected, all items that do not match this option would be deleted. This would make the identification process more directional, which may save time.

Besides, interactive keys may also solve some of the problems inherent in traditional keys. The first thing is the medium of the key. Traditional keys are based on paper, we need to find a suitable key with the accurate range, the most helpful order, and the newest version in the book. Different families, genera and even species are in different books. If a genus should be identified, a key to whole family might be used. Then, if a species must be identified, a key to the genus should be found. So, if we want to identify a plant in detail, many books should be carried and much time would be wasted for preparation, which is not convenient for fieldwork. However, if the interactive method used, only one laptop or a microphone would be enough. All data of relating to, genera and species can be stored in one database and the different ranks have been linked. Users could identify the plants in detail with interactive keys.

Lastly, the editing and understanding of interactive keys are easier than traditional keys. Interactive keys could assist users to understand character descriptions in different ways like images, videos, and more simple language. In the traditional key, the complete description sometimes might be lacking when using the key for the fieldwork. The interactive key could solve this problem. The editing of interactive keys is also easier and more reliable than traditional keys. When a new item is discovered, the paper-based traditional key might be incomplete and cannot work while interactive keys can be updated easily by adding data for the newly discovered item (Brasher, 2006).

### 16.3 Future work

The Padme system has provided an interactive key to users, which makes the identification easier. But there is still something which is not convenient enough and should be improved in future. The process of testing the interactive key and identification needs taxonomists with

enough knowledge to spend a lot of time. To be honest, it is not necessary to do all these things manually and some of the work could be finished by computer instead of the professionals.

### 16.3.1 Completing the definition of each character and state

To standardize the format of the DELTA file, the language of character descriptions is simplified. For example, the height of the whole tree in the DELTA file is described as 'Trees <height>'. Some confusion might be caused in this way. It is not perfectly clear whether this description means the height from the longest root to the top of the tree or the height of the aboveground part. Many characters might have this confusing problem. There might be a slight effect on those professional taxonomists who used the Padme system and DELTA frequently. However, if the user is not familiar with DELTA file and Padme system, it might be hard to identify each character accurately. Besides, some state descriptions have a similar situation such as the description of the stipules shape. It is easy for experienced taxonomists to distinguish linear-subulate stipules from, oblong, foliaceous and narrow-triangular, but this might be completely meaningless to beginners.

The best way to solve this problem is to create two new parts in the database to store the complete definition of each character and to explain the words used in professional state descriptions. These two parts would be linked to the interactive key but would not show in the key page. If users are experienced and do not use the definition assistance, they could use the interactive key directly. On the contrary, if users are those who are not familiar with description language, they could use the assistance to finish the identification through the link.

### 16.3.2 Adding multiple records into the Padme system

Although completing the definition of description words would be helpful to beginners, sometimes words alone are not powerful enough for identification. When there is a character or a state hard to describe or there are two states difficult to distinguish, words alone cannot resolve the identification. Images, or videos would be more reliable and powerful to explain. Besides, these tools could also be used as a double-check tool. If there are some uncertain characters, images of these characters could be compared with the character of the uncertain specimen being identified.

As with adding description definitions, these could be stored in a new part in the database and linked to the interactive key. If users could identify a character with confidence, there would

be no need to check pictures or videos. Alternatively, users could use images as the main identifying tool or the testing and checking tool.

### 16.3.3 Creating a testing system for the interactive key

In this project, the testing part is based on manual checking. Taxonomists need to use enough specimens or living materials to check the interactive key and find the places where the key does not work. Then, the programmers need to find where this problem lies and to check whether it is a problem of the DELTA file or parsing scripts or publishing scripts. It is not necessary to use manual testing in some operations. The DELTA file was translated into English in this project which is a good way to release the pressure of people's work. However, there is still much work needed by professionals.

To release the work pressure, codes could be used to link the translated DELTA file with English sequences to the Padme system and compare them with the computer program automatically. Through this process, all errors detected should be problems in parsing scripts or publishing scripts and the error position could be known through the error message. In this way, people would only be needed to test the interactive key and check the DELTA information manually, which is easier than before.

## References

- Allkin, R., White, R. J. and Winfield, P. J. (1992). Handling the taxonomic structure of biological data. *Mathematical and Computer Modelling*, 16(6–7): 1–9. doi: 10.1016/0895-7177(92)90148-E.
- Bentham, G. and Hooker, J. D. (1862). Tiliaceae tribus VII, Elaeocarpeae. *Genera plantarum* 1: 239–240. London, A. Black.
- Brasher, J. W. (2006). The Southern Rocky Mountain Interactive Flora (SRMIF) and factors correlated with recognition of plants and mammals. *Ph. D. Thesis*. University of Northern Colorado.
- Brongniart, A. and Gris (1861). Description de quelques Eléocarpées de la Nouvelle Calédonie. *Bulletin de la Société Botanique de France*, 8: 198–203.
- Coode, M. (1978). A Conspectus of Elaeocarpaceae in Papuasia. *Brunonia* 1(2): 131–302.
- Coode, M. (1984). Elaeocarpus in Australia and New Zealand. *Kew Bulletin*, 39(3): 509-586+1–20.
- Coode, M. (1996a). Elaeocarpus for Flora Malesiana : Notes , New Taxa and Combinations in sect . Elaeocarpus: 2. *Kew Bulletin* 51(1): 83–101.
- Coode, M. (1996b). Elaeocarpus for Flora Malesiana : Notes , New Taxa and Combinations in the Acronodia Group, *Kew Bulletin* 51(2): 267–300.
- Coode, M. (2001a). Elaeocarpus for Flora Malesiana : The Coilopetalum Group in Sulawesi & Maluku. *Kew Bulletin*, 56(4): 837–874.
- Coode, M. (2001b). Elaeocarpus for Flora Malesiana : The Verticellatae Subgroup of the Monocera Group and a New Philippine Species. *Kew Bulletin* 56(4): 885–901.
- Coode, M. (2001c). Elaeocarpus in New Guinea : New Taxa in the Debruyinii Subgroup of the Monocera Group. *Kew Bulletin* 56(2): 449–460.
- Coode, M. (2003). Elaeocarpus for Flora Malesiana : Two New Taxa in the Coloides Subgroup of the Monocera Group from New Guinea. *Kew Bulletin* 58(2): 453–458.
- Coode, M. (2004). Elaeocarpaceae. In Kubitzki, K. (ed), *The Families and Genera of Vascular Plants* 6: 135–144. Berlin & Heidelberg, Springer.
- Coode, M. (2010). Elaeocarpus for Flora Malesiana : new taxa and understanding in the Ganitrus group. *Kew Bulletin* 65: 355–399.
- Crayn, D. M., Winter, K. and Smith, J. A. C. (2004). Multiple origins of crassulacean acid metabolism and the epiphytic habit in the Neotropical family Bromeliaceae. *Proceedings of the National Academy of Sciences of the United States of America*, 101(10): 3703–3708. doi: 10.1073/pnas.0400366101.

- Dallwitz, M. (1980). A General System for Coding Taxonomic Descriptions. *Taxon* 29(1): 41–46.
- Dallwitz, M., Paine, T. A. and Zurcher, E. J. (1993). User's Guide to the DELTA System : a General System for Processing Taxonomic Descriptions. 4th edition.
- Diederich, J. (1997). Basic properties for biological databases: Character development and support. *Mathematical and Computer Modelling*, 25(10): 109–127. doi: 10.1016/S0895-7177(97)00078-2.
- Heibl, C. and Renner, S. S. (2012). Distribution Models and a Dated Phylogeny for Chilean Oxalis Species Reveal Occupation of New Habitats by Different Lineages , not Rapid Adaptive Radiation. *Systematic Biology* 61(5): 823–834. doi: 10.1093/sysbio/sys034.
- Masters, M. (1874). Tiliaceae. In Hooker, J.D. (ed), *Flora of British India* 1: 400–409.
- Morse, L. E. (1974). Computer-Assisted Storage and Retrieval of the Data of Taxonomy and Systematics. *Taxon* 23(1): 29–43.
- Pankhurst, R. J. (1986). A package of computer programs for handling taxonomic databases. *CABIOS* 2(1): 33–39.
- Pankhurst, R. J. (1991). *Practical taxonomic computing*. Cambridge, Cambridge University Press.
- Pullan, M. R. *et al.* (2005). The Prometheus Description Model: An examination of the taxonomic description-building process and its representation. *Taxon* 54(3): 751–765. doi: 10.2307/25065431.
- Schlechter, R. (1916). Die Elaeocarpaceen Papuasiens. *Bot. Jahrb. Syst.* 54: 92–155.
- Ung, V. *et al.* (2010). Xper 2 : introducing e -taxonomy. 26(5): 703–704. doi: 10.1093/bioinformatics/btp715.
- Vignes-Lebbe, R. *et al.* (2017). Desktop or remote knowledge base management systems for taxonomic data and identification keys : Xper2 and Xper3: 1–3. doi: 10.3897/tdwgproceedings.1.19911.
- Weibel, R. (1968). Morphologie de l'embryon et de la graine des Elaeocarpus. *Candollea*, 23: 101–108.
- Wikström, N., Savolainen, V. and Chase, M. W. (2001). Evolution of the angiosperms : calibrating the family tree. *Proceedings of the Royal Society B, Biological Sciences*, (December 2000). doi: 10.1098/rspb.2001.1782.
- Winston, J. E. (1999). *Describing species: practical taxonomic procedure for biologists*. New York, Columbia University Press.



## Appendixes

### Appendix 1. deltaparser.php

```
<?php
class DeltaParser
{
    private $inputfile;
    private $inputdata;
    private $conn;
    private $readpos;
    private $characterypes = array();
    private $nextcharacter = array();
    private $structures = array();
    private $characterid;
    private $currentcharacter;
    private $characters = array();
    private $itemcharacters = array();
    private $itemnames;
    //Read the PHP documentation at https://www.php.net/language.oop5 to learn
    //about class constructors and their relationship to the new key word in PHP
    /**
     * Class constructor
     * Parameters
     * $inputfile - path to the input file to parse
     * throw an exception when the file cannot be found
     */
    public function __construct($inputfile, $db_host, $db_username, $db_db, $db
    _password, $db_pdo_prefix )
    {
        //Store the inputfile variable passed in the parameters (the path) in the
        //inputfile property of this class
        if(file_exists($inputfile))
        {
            $this->inputfile = $inputfile;
            $db_dsn="host=$db_host;dbname=$db_db;charset=utf8";//Build the connecti
            on string specifying that data will be transmitted and recieved using UNICODE
            utf-8 encoding
            $this->conn = new PDO("$db_pdo_prefix:$db_dsn",$db_username,$db_passwor
            d); //Ask PDO to make the connection and store the connection in a variable ca
            lled $conn
            $this->conn->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
            //Tell PDO to report errors by throwing exceptions
        }
        else
        {
```

```

        throw new Exception("file does not exist");
    }
}

/**
 * Function reading the class property inputfile
 * parameter
 * none
 * set a newline at the end of each array element
 */
private function readData()
{
    //Open and read the inputfile when the class was instantiated and call the
    //function 'file' in $inputdata.
    $this->inputdata = file($this->inputfile, FILE_IGNORE_NEW_LINES); //Omit n
    //ewline at the end of each array element
}

/**
 * Function controlling the parsing of a DELTA file
 * parameter
 * none
 */
public function parse()
{
    //Call private functions
    $this->readData();
    //$this->dbConnect();
    $this->dbInitialise();
    //$this->echoData(); Testing
    $this->getCharacterTypes("CHARACTER TYPES");
    $this->findDirective("CHARACTER DESCRIPTIONS");
    //$this->readNextCharacter();
    //$this->testReadCharacters();
    $this->parseCharacter();
    $this->findDirective("ITEM DESCRIPTIONS");
    //$this->readNextDescription();
    $this->parseDescriptions();
}

/**
 * Function testing the class property inputdata
 * parameter
 * none
 * iterate through $this->inputdata
 */
public function echoData()

```

```

{
    foreach($this->inputdata as $value)
    {
        echo $value;
        echo "\n";
    }
}

/**
 * Function to initialise the setting of MySQL
 * parameter
 * none
 * delete TABLE structures, characters, states, litrescores, literaturerec
ords
 * catch the PDOException if there is something wrong with the query
 */

private function dbInitialise()
{
    $queries = array("TRUNCATE TABLE structures;", "TRUNCATE TABLE characters
;", "TRUNCATE TABLE states;", "TRUNCATE TABLE litrescores;", "TRUNCATE TABLE lit
erature_records;"); //As a rule you should always enclose field names and tabl
e names in backticks (`) - The backtick key on a standard UK key board is loca
ted to the left of the 1 key
    foreach($queries as $sql)
    {
        //Truncate TABLE structures, characters, states, litrescores and lite
raturerecords.
        $p = $this->conn->prepare($sql); //As a rule you should always use pre
pared statements when executing a query - Read the PHP documentation on the us
e of prepared statements.
        //Execute the query catching any exceptions that are thrown
        try
        {
            $p->execute();
            //Iterate through the results of the query and show them in the cons
ole.

            //You should read the PHP documentation on control structures to lea
rn about iterative loops - you will need to understand these concepts
            //https://www.php.net/manual/en/language.control-structures.php
        }
        catch(PDOException $e)//Trap any error thrown when the query is run
        {
            //Print the error to the screen.
            echo "Error running the query. Reason {$e->getMessage()}\n";
            exit; //Stop the script
        }
    }
}

```

```

    }

/**
 * Function that locates the position of a directive in the DELTA file
 * Sets the readpos class property to the index of the line in the inputda
ta class property to 1 greater than the index of the directive
 * Parameters
 * $directive: string containing the name of the directive find
 * Throws an exception if the directive cannot be found
 */
private function findDirective($directive)
{
    //Search the array and find the line containing directive name
    for($line = 0; $line < sizeof($this->inputdata); $line++)
    {
        if(strpos($this->inputdata[$line], $directive))
        {
            $this->readpos = $line + 1; //Indexes in PHP are 0 based
            return;
        }
    }
    throw new Exception("Directive $directive not found");
}

/**
 * Function that gets the string between $directive and next directive nam
e
 * parameter
 * $directive: string containing the name of the directive find
 * add each line behind the $inputstring and use "*" to stop adding
 * return
 * $inputstring: string between $directive and next directive name
 */
private function getDirectiveData($directive)
{
    for($i = $this->readpos; $i < sizeof($this->inputdata); $i++)
    {
        //Get the string after $directive and before next directive name
        if(strpos($this->inputdata[$i], "*") === false)
        {
            $inputstring .= $this->inputdata[$i]; //if the new line does not f
ind *, put $this->inputdata[i] behind the $inputstring
        }
        else
        {
            return $inputstring; //if the new line find *, return $inputstring
and stop loop
        }
    }
}

```

```

    }
}

/**
 * Function that reads in the character types data and assign each character number/ character type pair to the charactertypes property.
 * parameter
 * $directive: string containing the name of the directive find
 * $old: the $inputstring which got from function getDirectiveData
 * trim $old and explode it with space
 * explode $old with "," putting character number as class property $charactertypes index while character type pair as value
 */
private function getCharacterTypes($directive)
{
    $this->findDirective($directive);
    $old = $this->getDirectiveData($directive); //call function getDirectiveData.
    $old = trim($old);//delete all spaces before or after $old
    $old = explode(" ", $old);//explode array with space
    foreach($old as $value)//loop for the $old
    {
        $parts = explode(",", $value);//explode the $value of $old into two elements array with ','
        $this->charactertypes[$parts[0]] = $parts[1];//make the 0 element of $parts as the index of $this->charactertypes and the 1 element as the value of $this->charactertypes
    }
}

/**
 * Function that extract a new array from the input data data array
 * parameter
 * none
 * throw a new exception if the first element does not include "#"
 * characters are separated by empty lines, end the loop when meet empty line
 * if the previous line has "/", create a new element; else, the line should keep reading until there is a "/"
 */
private function readNextCharacter()
{
    if (strpos($this->inputdata[$this->readpos], "#") !== false)
    {
        //meeting "*" means all characters have been read and the function should stop
        echo "All characters have been read.";
        return $this->nextcharacter = array();
    }
}

```

```

    }
    elseif(strpos($this->inputdata[$this->readpos], "#") !== false)
    {
        //create a new array for each character
        $i = 0;
        $this->nextcharacter = array();
        while (($this->inputdata[$this->readpos] !== " ") && ($this->inputdata[$this->readpos] !== ""))
        {
            //each character is separated from the next with an empty line
            if(((strpos($this->inputdata[$this->readpos], "#")) || (preg_match("/.+\/$/", $this->inputdata[$this->readpos-1]))) !== false)
            {
                //if the previous line has "/" or the line has "#", a new element should be created; else, the line should keep reading until there is a "/"
                $this->nextcharacter[$i] = $this->inputdata[$this->readpos];
                $this->readpos++;
                $i++;
            }
            else
            {
                $this->nextcharacter[$i-1] .= $this->inputdata[$this->readpos];
                $this->readpos++;
            }
        }
        $this->readpos++;
        return $this->nextcharacter;
    }
    else
    {
        throw new exception ("Invalid syntax has been detected in line".$this->readpos);
    }
}

/**
 * Function to test readNextCharacter function
 */
private function testReadCharacters()
{
    //test readNextCharacter function
    while(sizeof($this->readNextCharacter()) !== 0)
    {
        print_r($this->nextcharacter);
    }
}
/**

```

```

    * Function that insert the information of character description into da
ta base
    * parameter
    * none
    * iterate through all character descriptions with while loop to parse a
ll structures, characters and states and store them in Padme database
    * insert structure name into table structures and get the structure id
when the structure has not been delt with; or link the existing structure to t
he old record
    * get the class property $this->structure with structure name as index
and structure id as value
    * insert character name, structureid, charactertype and units (only for
characters with RN and IN types) into table characters and get the character
id
    * get the class property $this->character with character number as inde
x and character name as value, and instantiate a new object of class Character
    * insert state name and characterid into table states and get the state
id
    * call the function addState of Character class
*/
private function parseCharacter()
{
    while(sizeof($this->readNextCharacter()) !== 0)
    {
        //loop through all character descriptions
        if(!((preg_match("/\#(\d+)\.(.)\<(.+)\>\/(.+)\//", $this->nextcharac
ter[0], $matches)) || (preg_match("/\#(\d+)\.(.)\<(.+)\>\//", $this->nextcharac
ter[0], $matches)) || (preg_match("/\#(\d+)\.(.)\//", $this->nextcharacter[0], $m
atches))))))
        {
            //throw exceptions which cannot corresponding to formats
            throw new exception("Syntax error: " . $this->nextcharacter[0] . "
is not a valid start to a character defintion");
        }
        //The character number is prefixed by a hash (#) and ends with a ful
l-stop (.).
        //The attribute/character name is enclosed in angle brackets (<>)
        //The structure name sits between the DELTA character number and the
character name in angle brackets.
        //Units of measure are enclosed by slashes (/) at end of the DELTA c
haracter name.
        $currentcharacternumber = $matches[1];
        $currentstructure = $matches[2];
        $currentcharactername = $matches[3];
        $currentunits = $matches[4];
        //if the structure id has already existed, get the Padme id of the s
tructure from structures class property.
        $structureid = $this->structures[$currentstructure];

```

```

if(!$structureid)
{
//if the structure id has not been dealt with, insert a new record
//add a new structure to the Padme data base
$structurenamesql = "INSERT INTO structures(name) VALUES(?)";
$p = $this->conn->prepare($structurenamesql);
$p->execute(array($currentstructure));
//get the maximum id number from SQL database as the structure id
$structureidsql = "SELECT max(id) FROM structures";
$p = $this->conn->prepare($structureidsql);
$p->execute();
$structureid = $p->fetchColumn();
//using the structure name as $this->structure key and structure id
d as the value
$this->structures[$currentstructure] = $structureid;
}
//insert character name, structureid, character type and units into
characters table in padme
$characterinsertsql = "INSERT INTO characters (name,structureid,type
,units) VALUES(?,?,?,?)";
$p = $this->conn->prepare($characterinsertsql);
//use $this->character type to look up the chractertype with $current
character number
$currentcharacter type = $this->character types[$currentcharacter numbe
r];

//if there is not a match in $this->character types default to UM
if(!$currentcharacter type)
{
$currentcharacter type = 1;
}
else
{
//all character types are stored in padme in integer, so switch "U
M" "OM" "IN" "RN" to 1,2,3,4 seperately;
switch($currentcharacter type)
{
case "RN": $currentcharacter type = 4; break;
case "IN": $currentcharacter type = 3; break;
case "OM": $currentcharacter type = 2; break;
case "UM": $currentcharacter type = 1; break;

}
}
}
$p->execute(array($currentcharacter name,$structureid,$currentcharact
ertype,$currentunits));
//find out the character id from table characters using the maximum
id
$characteridsql = "SELECT max(id) FROM characters";

```



```

        $p = $this->conn->prepare($characteridsql);
        $p->execute();
        $this->characterid = $p->fetchColumn();
        //instantiate a new object of Character class, so could link states
to the corresponding character
        $currentcharacter = new Character();
        //Use the character number as the index into the array and set the
character object as the value
        $this->characters[$currentcharacternumber] = $currentcharacter;
        for($i=1; $i < sizeof($this->nextcharacter); $i++)
        {
            //deal with all states after a character with "OM" or "UM" type us
ing for loop
            if (preg_match("/(\d*)\.(.+)\//", $this->nextcharacter[$i], $matches
))
            {
                //if the statement could match the pattern, insert name and char
acterid into database
                $states[$matches[1]] = $matches[2];
                $statesql = "INSERT INTO states(name,characterid) VALUES(?,?)";
                $p = $this->conn->prepare($statesql);
                $p->execute(array($matches[2], $this->characterid));
                //find out the stateid using max()
                $stateidsql = "SELECT max(id) FROM states";
                $p = $this->conn->prepare($stateidsql);
                $p->execute();
                $stateid = $p->fetchColumn();
                //call the function addState of Character class
                $currentcharacter->addState($i,$stateid);
            }
            else
            {
                //throw exception if the syntax does not match the pattern
                throw new exception("Syntax error: " . $this->nextcharacter[$i]
. " is not a valid definition of a state");
            }
        }
    }
}

/**
 * Function to read item descriptions
 * parameter
 * none
 * if there is "*" in the line, all items have been read
 * the first line of each item is the Latin name, store the latin name
in $itemnames

```

\* all lines after the first line are character description, ignore contents in angle brackets and store each character into an element of array \$itemcharacters

```

*/
private function readNextDescription()
{
    //initialize $this->readpos with function findDirective
    $this->readpos += 1;
    if(strpos($this->inputdata[$this->readpos], "*") !== false)
    {
        //all characters have been read when meeting *END
        echo "All items have been read";
        return false;
        exit;
    }
    else
    {
        $i = 0;
        //each item is separated from the next with an empty line
        while(!empty($this->inputdata[$this->readpos]) && ($this->inputdata[
$this->readpos] !== " "))
        {
            //iterate through all description of each item
            if(preg_match("/^\ \ +$/", $this->inputdata[$this->readpos], $matc
hes))
            {
                throw new exception("Syntax error: more than one spaces have bee
n detected in line". $this->readpos);
            }
            elseif(strpos($this->inputdata[$this->readpos], "#") !== false)
            {
                //the first line of each item is the Latin name, "#item number.
latin name", storing it in class property $description
                $description[$i] = $this->inputdata[$this->readpos];
                $this->readpos++;
                $i++;
            }
            else
            {
                //put all description into one element of class property $descri
ption

                $description[$i] .= $this->inputdata[$this->readpos];
                $this->readpos++;
            }
        }
        //get the item name from the first line of $description
        preg_match("/#\d*\.(.+)\//", $description[0], $match);
        $this->itemnames = "Elaeocarpus ". trim($match[1]);
    }
}

```

```

        //delete all contents in angle brackets
        $description[1] = preg_replace("/\<.+?\>/", "", $description[1]);
        //explode the description with space and store them in the class pro
perty itemcharacters
        $this->itemcharacters = explode(" ", $description[1]);
    }
}

/**
 * Function to parse descriptions of each item
 * parameter
 * none
 * iterate through all item descriptions with while loop
 * get id from latin_name table of the padme database
 * get the reference from references table of the padme database
 * insert the id and reference into literature_records table and get id
from it
 * insert literature id, character id, minvalue and maxvalue or state id
into table litrecscores
 */
private function parseDescriptions()
{
    //read all items using while loop
    while($this->readNextDescription() !== false)
    {
        echo $this->itemnames;
        echo "\n";
        //get id corresponding the epithet from table latin_names
        $latinnamesql = "SELECT id FROM latin_names WHERE full_name = '$this
->itemnames'";
        $p = $this->conn->prepare($latinnamesql);
        $p->execute();
        $latinnameid = $p->fetch();
        //get references from the table references, all descriptions are fro
m the same reference "Coode, M. Coded descriptions of Elaeocarpus." which id i
s 1
        $reference = 1;
        //insert the id got from table Latinname and the reference into tabl
e literature_records, and get id
        $determinationsql = "INSERT INTO literature_records(determination,re
ference) VALUES(?,?)";
        $p = $this->conn->prepare($determinationsql);
        $p->execute(array($latinnameid[0], $reference));
        $literatureidsql = "SELECT id FROM literature_records WHERE determin
ation = '$latinnameid[0]'";
        $p = $this->conn->prepare($literatureidsql);
        $p->execute();
        $literatureid = $p->fetch;
    }
}

```

```

        //insert literature id, character id, minvalue and maxvalue or state
id into table litrescores.
        $litrescoressql = "INSERT INTO `litrescores`(`stateid`, `litrecid`
, `characterid`, `scoregroupid`, `minvalue`, `maxvalue`) VALUES(?,?,?,?,?,?)";
        $p = $this->conn->prepare($litrescoressql);
        //scoregroupid of all character description is the same 1
        $scoregroupid = 1;
        foreach($this->itemcharacters as $value)
        {
            //initialize minvalue, maxvalue and stateid
            $minvalue = NULL;
            $maxvalue = NULL;
            $stateid = NULL;
            //iterate through all characters description of each item
            //each character: #character id.
            preg_match("/(\d*)\,(.+)/", $value, $matches);
            $characterid = $matches[1];
            $currentcharacter = $this->characters[$characterid];
            if($this->characterTypes[$matches[1]] == "RN" or $this->character
types[$matches[1]] == "IN")
            {
                //if the character type is RN or IN, get the first number as min
value while the second one as maxvalue
                preg_match("/(\d+\.\d?)\-(\d+\.\d?)?/", $matches[2], $matche
s1);

                $minvalue = $matches1[1];
                $maxvalue = $matches1[2];
                if($maxvalue == "" or $maxvalue == null)
                {
                    $maxvalue = $minvalue;
                }
                try
                {
                    $p->execute(array($stateid, $literatureid[0], $characterid, $
scoregroupid, $minvalue, $maxvalue));
                }
                catch(PDOException $e)
                {
                    throw $e;
                }
            }
            else
            {
                //if the character type is UM or OM, get the statenumber and use
the getState method of class Character to get stateid
                $statearray = explode ("/", $matches[2]);
                $i = 0;
                while($i < sizeof($statearray) && $matches[2])

```



## Appendix 2. elaeocarpus.php

```
<?php
//load the class defintion file
require_once "deltaparser.php";
//use error trapping to catch the exception
try
{
    $parser = new DeltaParser("c:/0/Elaeocarpus/trunk/el28.dat", "localhost", "
padmeuser", "padmeweb1_elaeocarpus", "padmepassword", "mysql");
}
catch(Exception $e)
{
    echo "Error creating new Delta Parser. Reason: " . $e->getMessage();
    exit;
}
//Ask the parser to do its work
try
{
    $parser->parse();
}
catch(Exception $e)
{
    echo $e->getMessage();
    exit;
}
```

### Appendix 3. tester.php

```
<?php
```

```
class DeltaTester
{
    private $inputfile;
    private $inputdata;
    private $readpos;
    private $characterTypes = array();
    private $nextchracter = array();
    private $characters = array();
    private $structures = array();
    private $states = array();
    private $units;
    private $itemnames;
    private $itemcharacters;

    public function __construct($inputfile)
    {
        //Store the inputfile variable passed in the parameters (the path) in the
        inputfile property of this class
        if(file_exists($inputfile))
        {
            $this->inputfile = $inputfile;
        }
        else
        {
            throw new Exception("file does not exist");
        }
    }

    /**
     * Function reading the class property inputfile
     * parameter
     * none
     * set a newline at the end of each array element
     */
    private function readData()
    {
        //Open and read the inputfile when the class was instantiated and call th
        e 'file' function in $inputdata.
        $this->inputdata = file($this->inputfile, FILE_IGNORE_NEW_LINES);//Omit n
        ewline at the end of each array element
    }
}
```

```

/**
 * Function controlling the parsing of a DELTA file
 * parameter
 * none
 */
public function parse()
{
    //Call private functions
    $this->readData();
    $this->getCharacterTypes("CHARACTER TYPES");
    $this->findDirective("CHARACTER DESCRIPTIONS");
    $this->parseCharacter();
    $this->findDirective("ITEM DESCRIPTIONS");
    $this->parseDescriptions();
}

/**
 * Function that locates the position of a directive in the DELTA file
 * Sets the readpos class property to the index of the line in the inputdata
 * class property to 1 greater than the index of the directive
 * Parameters
 * $directive: string containing the name of the directive find
 * Throws an exception if the directive cannot be found
 */
private function findDirective($directive)
{
    //Search the array and find the line containing directive name
    for($line = 0; $line < sizeof($this->inputdata); $line++)
    {
        if(strpos($this->inputdata[$line], $directive))
        {
            $this->readpos = $line + 1; //Indexes in PHP are 0 based
            return;
        }
    }
    throw new Exception("Directive $directive not found");
}

/**
 * Function that gets the string between $directive and next directive name
 * parameter
 * $directive: string containing the name of the directive find
 * add each line behind the $inputstring and use "*" to stop adding
 * return
 * $inputstring: string between $directive and next directive name
 */
private function getDirectiveData($directive)

```

e



```

{
    for($i = $this->readpos; $i < sizeof($this->inputdata); $i++)
    {
        //Get the string after $directive and before next directive name
        if(strpos($this->inputdata[$i], "*") === false)
        {
            $inputstring .= $this->inputdata[$i]; //if the new line does not find
            *, put $this->inputdata[i] behind the $inputstring
        }
        else
        {
            return $inputstring; //if the new line find *, return $inputstring
            and stop loop
        }
    }
}

/**
 * Function that reads in the character types data and assign each character
 * number/ character type pair to the charactertypes property.
 * parameter
 * $directive: string containing the name of the directive find
 * $old: the $inputstring which got from function getDirectiveData
 * trim $old and explode it with space
 * explode $old with "," putting character number as class property $charactertypes
 * index while character type pair as value
 */
private function getCharacterTypes($directive)
{
    $this->findDirective($directive);
    $old = $this->getDirectiveData($directive); //call function getDirectiveData.
    $old = trim($old); //delete all spaces before or after $old
    $old = explode(" ", $old); //explode array with space
    foreach($old as $value) //loop for the $old
    {
        $parts = explode(",", $value); //explode the $value of $old into two
        elements array with ','
        $this->charactertypes[$parts[0]] = $parts[1]; //make the 0 element of
        $parts as the index of $this->charactertypes and the 1 element as the value of
        $this->charactertypes
    }
}

/**
 * Function that extract a new array from the input data data array
 * parameter
 * none

```

```

        * throw a new exception if the first element does not include "#"
        * characters are separated by empty lines, end the loop when meet empty
y line
        * if the previous line has "/", create a new element; else, the line s
ould keep reading until there is a "/"
        */
private function readNextCharacter()
{
    if (strpos($this->inputdata[$this->readpos], "*") !== false)
    {
        //meeting "*" means all characters have been read and the function s
ould stop
        return $this->nextcharacter = array();
    }
    elseif(strpos($this->inputdata[$this->readpos], "#") !== false)
    {
        //create a new array for each character
        $i = 0;
        $this->nextcharacter = array();
        while ((($this->inputdata[$this->readpos] !== " ") && ($this->inputda
ta[$this->readpos] !== ""))
        {
            //each character is seperated from the next with an empty line
            if(((strpos($this->inputdata[$this->readpos], "#")) || (preg_match
("/.+\\$/", $this->inputdata[$this->readpos-1]))) !== false)
            {
                //if the previous line has "/" or the line has "#", a new elemen
t should be created; else, the line should keep reading until there is a "/"
                $this->nextcharacter[$i] = $this->inputdata[$this->readpos];
                $this->readpos++;
                $i++;
            }
            else
            {
                $this->nextcharacter[$i-1] .= $this->inputdata[$this->readpos];
                $this->readpos++;
            }
        }
        $this->readpos++;
        return $this->nextcharacter;
    }
    else
    {
        throw new exception ("Invalid syntax has been detected in line".$thi
s->readpos);
    }
}
}

```

```

/**
 * Function that insert the information of character description into da
ta base
 * parameter
 * none
 * iterate through all character descriptions with while loop to parse a
ll structures, characters and states and store them in Padme database
 * insert structure name into table structures and get the structure id
when the structure has not been delt with; or link the existing structure to t
he old record
 * get the class property $this->structure with structure name as index
and structure id as value
 * insert character name, structureid, charactertype and units (only for
characters with RN and IN types) into table characters and get the character
id
 * get the class property $this->character with character number as inde
x and character name as value, and instantiate a new object of class Character
 * insert state name and characterid into table states and get the state
id
 * call the function addState of Character class
*/
private function parseCharacter()
{
    while(sizeof($this->readNextCharacter()) !== 0)
    {
        //loop through all character descriptions
        if(!((preg_match("/\#(\d+)\.(.+)\<(.)\>\/(.)\//", $this->nextcharac
ter[0], $matches) || (preg_match("/\#(\d+)\.(.+)\<(.)\>\//", $this->nextcharac
ter[0], $matches) || (preg_match("/\#(\d+)\.(.+)\//", $this->nextcharacter[0], $m
atches))))))
        {
            //throw exceptions which cannot corresponding to formats
            throw new exception("Syntax error: " . $this->nextcharacter[0] . "
is not a valid start to a character defintion");
        }
        //The character number is prefixed by a hash (#) and ends with a ful
l-stop (.).
        //The attribute/character name is enclosed in angle brackets (<>)
        //The structure name sits between the DELTA character number and the
character name in angle brackets.
        //Units of measure are enclosed by slashes (/) at end of the DELTA c
haracter name.
        $characternumber = $matches[1];
        $structurename = $matches[2];
        $charactername = $matches[3];
        $unit = $matches[4];
        $this->characters[$characternumber] = $charactername;
        $this->structures[$characternumber] = $structurename;
    }
}

```

```

$this->units[$characternumber] = $unit;
//if there is not a match in $this->charactertypes default to UM
if(!$this->charactertypes[$characternumber])
{
    $this->charactertypes[$characternumber] = "UM";
}
for($i=1; $i < sizeof($this->nextcharacter); $i++)
{
    //deal with all states after a character with "OM" or "UM" type us
ing for loop
    if (preg_match("/(\d*)\.(.+)\//", $this->nextcharacter[$i], $matches
))
    {
        //if the statement could match the pattern, insert name and char
acterid into database
        $statenumber = $matches[1];
        $statename = $matches[2];
        $this->states[$statenumber] = $statename;
    }
    else
    {
        //throw exception if the syntax does not match the pattern
        throw new exception("Syntax error: " . $this->nextcharacter[$i]
. " is not a valid definition of a state");
    }
}
}
}

/**
 * Function to read item descriptions
 * parameter
 * none
 * if there is "*" in the line, all items have been read
 * the first line of each item is the Latin name, store the latin name
in $itemnames
 * all lines after the first line are character description, ignore con
tents in angle brackets and store each character into an element of array $ite
mcharacters
 */
private function readNextDescription()
{
    //initialize $this->readpos with function findDirective
    $this->readpos += 1;
    if(strpos($this->inputdata[$this->readpos], "*") !== false)
    {
        //all characters have been read when meeting *END
        echo "All items have been read";
    }
}

```

```

        return false;
        exit;
    }
    else
    {
        $i = 0;
        //each item is seperated from the next with an empty line
        while(!empty($this->inputdata[$this->readpos]) && ($this->inputdata[
$this->readpos] !== " "))
        {
            //iterate through all description of each item
            if(preg_match("/^\ \ +$/", $this->inputdata[$this->readpos], $matc
hes))
            {
                throw new exception("Syntax error: more than one spaces have bee
n detected in line". $this->readpos);
            }
            elseif(strpos($this->inputdata[$this->readpos], "#") !== false)
            {
                //the first line of each item is the Latin name, "#item number.
latin name", storing it in class property $description
                $description[$i] = $this->inputdata[$this->readpos];
                $this->readpos++;
                $i++;
            }
            else
            {
                //put all description into one element of class property $descri
ption

                $description[$i] .= $this->inputdata[$this->readpos];
                $this->readpos++;
            }
        }
        //get the item name from the first line of $description
        preg_match("/#\d*\.(.+)\//", $description[0], $match);
        $this->itemnames = "Elaeocarpus ". trim($match[1]);
        //delete all contents in angle brackets
        $description[1] = preg_replace("/\<.+?\>/", "", $description[1]);
        //explode the description with space and store them in the class pro
perty itemcharacters
        $this->itemcharacters = explode(" ", $description[1]);
    }
}

/**
 * Function to parse descriptions of each item
 * parameter
 * none

```

```

* iterate through all item descriptions with while loop
* get id from latin_name table of the padme database
* get the reference from references table of the padme database
* insert the id and reference into literature_records table and get id
from it
* insert literature id, character id, minvalue and maxvalue or state id
into table litrecscores
**/
private function parseDescriptions()
{
  //read all items using while loop
  while($this->readNextDescription() !== false)
  {
    print_r("=====\n");
    print_r($this->itemnames ."\n");
    print_r("=====\n");
    foreach($this->itemcharacters as $value)
    {
      //initialize minvalue, maxvalue and stateid
      $minvalue = NULL;
      $maxvalue = NULL;
      $stateid = NULL;
      //iterate through all characters description of each item
      //each character: #character id.
      preg_match("/(\d*)\,(.+)/",$value,$matches);
      if(sizeof($matches) == 0)
      {
        continue;
      }
      $characternumber = $matches[1];
      print_r ("Character number: ". $characternumber ."\n");
      print_r ("Character name: ". $this->characters[$characternumber] .
"\n");
      print_r ("Structure name:". $this->structures[$characternumber] ."
\n");
      if($this->characterTypes[$matches[1]] == "RN" or $this->character
types[$matches[1]] == "IN")
      {
        //if the character type is RN or IN, get the first number as min
value while the second one as maxvalue
        preg_match("/(\d+\.\d+)?\d+\,-?\d+\.\d+)?/", $matches[2], $matche
s1);

        $minvalue = $matches1[1];
        $maxvalue = $matches1[2];
        if($maxvalue == "" or $maxvalue == null)
        {
          $maxvalue = $minvalue;
        }
      }
    }
  }
}

```

```
        print_r ("Range:". $minvalue ."-
". $maxvalue . $this->units[$characternumber] ."\n");
        print_r("\n");
    }
    else
    {
        //if the character type is UM or OM, get the statenumber and use
the getState method of class Character to get stateid
        $statearray = explode ("/", $matches[2]);
        $i = 0;
        while($i < sizeof($statearray) && $matches[2])
        {
            $statenumber = $statearray[$i];
            print_r("State number:". $statenumber ."\n" );
            print_r("State name:". $this->states[$statenumber] ."\n");
            print_r("\n");
            $i++;
        }
    }
}
}
```

#### Appendix 4. elaeocarpustest.php

```
<?php
//load the class defintion file
require_once "tester.php";
//use error trapping to catch the exception
try
{
    $parser = new DeltaTester("c:/0/Elaeocarpus/trunk/el28.dat");
}
catch(Exception $e)
{
    echo "Error creating new Delta Tester. Reason: " . $e->getMessage();
    exit;
}
//Ask the parser to do its work
try
{
    $parser->parse();
}
catch(Exception $e)
{
    echo $e->getMessage();
    exit;
}
```