DEPARTMENT OF
COMPUTER SCIENCE

DIOGO DE ALMEIDA ESCALEIRA

BSc in Computer Science

ASSESSMENT OF OCTAVE'S OO FEATURES BASED ON GOF PATTERNS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
March, 2023

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

ASSESSMENT OF OCTAVE'S OO FEATURES BASED ON GOF PATTERNS

DIOGO DE ALMEIDA ESCALEIRA

BSc in Computer Science

Adviser: Miguel Jorge Tavares Pessoa Monteiro
Assistant Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
March, 2023

Assessment of Octave's OO features based on GoF patterns

Abstract

This thesis aims to evaluate the object-oriented (OO) features of the Octave programming language, through the implementation of the popular Gang-of-Four (GoF) design patterns. The study explores the fundamental principles of OO, including modularity, inheritance, encapsulation, polymorphism, and abstraction, and investigates how these concepts are supported by Octave. This research is conducted through the implementation of two complete collections of the GoF patterns originally coded in Java and the subsequent analysis of the quality of the implementations thus derived. This evaluation is based on comparisons with their Java counterparts as regards modularity and flexible module composition. To our knowledge, no study of this nature has been made on Octave. This thesis is intended to contribute to a better understanding of Octave's current OO capabilities and limitations as well as its potential as a tool for developing complex software systems.

Keywords: Octave, Object-Oriented Programming, Design Patterns, Gang-of-Four Patterns, Language Assessment, Modularity, Module Composition, Separation of Concerns

Resumo

Esta tese visa avaliar as características orientadas a objetos (OO) da linguagem de programação Octave, através da implementação dos populares *design patterns* dos *Gang-of-Four* (GoF). O estudo explora alguns princípios fundamentais de OO, incluindo modularidade, herança, encapsulamento, polimorfismo e abstração, e investiga o suporte de Octave a estes conceitos. Esta investigação é conduzida através da implementação de duas coleções completas dos padrões GoF originalmente desenvolvidos em Java e da análise subsequente à qualidade das implementações assim derivadas. Esta avaliação é baseada em comparações com os seus equivalentes Java no que diz respeito à modularidade e composição de módulos flexível. Segundo a nossa pesquisa, ainda não foi feito qualquer estudo desta natureza em Octave. Esta tese destina-se a contribuir para uma melhor compreensão das atuais capacidades e limitações do paradigma OO em Octave, bem como do seu potencial como ferramenta para o desenvolvimento de sistemas de software complexos.

Palavras-chave: Octave, Programação Orientada a Objetos, Design Patterns, Padrões Gang-of-Four, Avaliação de Linguagem, Modularidade, Composição de Módulos, Separação de Interesses

Contents

List of Figures ix

List of Tables x

List of Listings xi

Acronyms xiii

1 Introduction 1
- 1.1 Context and Description . . . 1
- 1.2 Motivation . . . 1
- 1.3 Objectives and Expected Contributions . . . 2
- 1.4 Research Questions . . . 2
- 1.5 Terminology Used in This Thesis . . . 3
- 1.6 Document Structure . . . 3

2 Octave 5
- 2.1 Octave Introduction . . . 5
  - 2.1.1 Introduction to Numerical Computation . . . 5
  - 2.1.2 Octave Overview and Capabilities . . . 6
  - 2.1.3 Octave Language Properties . . . 6
- 2.2 Basic Syntax . . . 7
  - 2.2.1 Basic Input Guidelines . . . 7
  - 2.2.2 Variables . . . 8
  - 2.2.3 Data Types . . . 8
  - 2.2.4 Functions and Arguments . . . 9
  - 2.2.5 Errors and Warnings . . . 9
- 2.3 Octave's Object-Oriented Features . . . 10
  - 2.3.1 "Old Style" Classes and *Classdef* Classes . . . 10
  - 2.3.2 Creating a Class . . . 10

2.3.3 Class Member Access Rights . . . 12
2.3.4 Overloading and Object Precedence . . . 12
2.3.5 Object Identity . . . 13
2.3.6 Inheritance . . . 13
2.4 Octave's Graphics Features . . . 15
2.5 Helpful Information . . . 15
2.5.1 Packages and Extensions . . . 15
2.5.2 IDE/GUI Survey . . . 16
2.5.3 Forums and Other Sources of Information . . . 16

3 Octave Design Patterns Implementations 17
3.1 Design Patterns . . . 17
3.1.1 Gang-of-Four Design Patterns . . . 18
3.2 Introduction to the Design Pattern Implementations . . . 19
3.3 Interfaces and Abstract Classes . . . 20
3.3.1 *Abstract Factory* . . . 21
3.4 Polymorphism in Data Structures . . . 28
3.4.1 *Observer* . . . 29
3.5 *Singleton* and Static Properties . . . 33
3.6 Wrappers . . . 35
3.6.1 *Adapter* and *Façade* . . . 35
3.6.2 *Decorator* and *Proxy* . . . 36
3.7 *Visitor* and the Expression Problem . . . 39
3.7.1 The Expression Problem . . . 44
3.8 Noteworthy remarks . . . 45
3.8.1 *Memento* and Nested Classes . . . 45
3.8.2 *Iterator* and the For-loop . . . 46
3.8.3 Broader Notes on the Octave Implementations . . . 47

4 Analysis on Object-Oriented Programming in Octave 49
4.1 Abstraction . . . 49
4.2 Encapsulation . . . 50
4.3 Polymorphism and Duck-Typing . . . 51
4.4 Modularity and Module Composition . . . 51
4.4.1 Modularity in Octave . . . 52
4.4.2 Modularity Mechanisms . . . 52
4.5 OO Feature Comparison with Java . . . 54
4.6 Summing Up . . . 54

5 Related Work 55
5.1 Design Patterns, Object-Oriented Programming Languages and Modularity 55
5.2 Design Pattern Implementation in Other Languages . . . 56

Bibliography **58**

List of Figures

2.1 "Old style" class folder/file structure 11
2.2 An example window showing some of the available UI elements like a *textbox*, a slider, a *listbox* and various types of buttons [29] 15
2.3 The default GUI provided by Octave 16

3.1 Diagram exemplifying a possible structure of implementation of the Abstract Factory pattern [27] 22
3.2 Class Diagram for the Java implementation of the "GardenMaker Factory" scenario 22
3.3 *Abstract Factory* implementation in Java UI 23
3.4 Class Diagram for the Octave implementation of the "GardenMaker Factory" scenario 26
3.5 *Abstract Factory* implementation in Octave UI 27
3.6 Diagram exemplifying a possible structure of implementation of the *Observer* pattern [27] 29
3.7 *Observer* implementation in Octave UI 30
3.8 Class Diagram for the Java implementation of the "Color Observer" scenario (graphical content omitted) 30
3.9 Class Diagram for the Octave implementation of the "Color Observer" scenario 32
3.10 Class Diagram for the Octave implementation of the "MakeACuppa" *Façade* scenario 35
3.11 *Decorator* implementation in Octave UI 37
3.12 Diagram exemplifying a possible structure of implementation of the Visitor pattern [27] 39
3.13 Class Diagram for the Java implementation of the "Vacation Visitor" scenario 40
3.14 *Visitor* implementation in Octave UI 40
3.15 Class Diagram for the Octave implementation of the "Vacation Visitor" scenario 43
3.16 Class Diagram for the Java implementation of the "Dvd Memento" scenario 45
3.17 Class Diagram for the Octave implementation of the "Dvd Memento" scenario 46

List of Tables

2.1 Variable types in Octave 8
2.2 Argument and return value parameters 9
2.3 Access Rights of Class Members in Octave 12

3.1 The GoF [11] design patterns by category 19
3.2 Pattern scenario source distribution 19
3.3 *Adapter*/*Façade* functionality comparison 35
3.4 The decorated buttons and the functionality added to them 37

4.1 OO feature comparison between Java and Octave 54

List of Listings

2.1 Function definiton example . . . 9
2.2 "Old style" class constructor example . . . 11
2.3 Calling an "old style" class' constructor example . . . 11
2.4 Creating a *classdef* class example . . . 11
2.5 Setting the access rights of a *classdef* class' properties and methods example 12
2.6 Setting a class as having precedence over another class example . . . 13
2.7 Constructor of an "old style" child class example . . . 13
2.8 *Classdef* inheritance examples . . . 14
2.9 Calling the superclass' constructor from a child class example . . . 14
3.1 Octave attempt at emulating an interface . . . 21
3.2 Java *AbstractGardenFactory* interface . . . 23
3.3 Java *VegetableGardenFactory* class . . . 23
3.4 Java *GardenFactory* class . . . 24
3.5 Octave *GardenFactory* class . . . 25
3.6 Octave *VegetableGardenFactory* class . . . 25
3.7 Octave array usage . . . 28
3.8 *ObjectWrapper* class . . . 31
3.9 Array usage in *WindowSubject* class . . . 31
3.10 Java *Printer* class . . . 33
3.11 Octave *Printer* class . . . 34
3.12 Octave *FacadeCuppaMaker* class . . . 36
3.13 Octave *Decorator* class . . . 37
3.14 Octave *PaintDecorator* class . . . 38
3.15 Java *Employee* class . . . 41
3.16 Octave *Employee* class . . . 41
3.17 Java *VacationVisitor* class . . . 41
3.18 Octave *VacationVisitor* class . . . 42
3.19 Java *BossVacationVisitor* class . . . 42
3.20 Octave *BossVacationVisitor* class . . . 43

3.21 Two ways of transversing a collection in Octave 47

Acronyms

GoF Gang-of-Four x, 1, 2, 17, 18, 19, 48, 56, 57
GUI graphical user interface ix, 16

I/O Input/Output 15
IDE Integrated Development Environment 16

MSc Masters of Sciences 1, 2

OO object-oriented x, 1, 2, 4, 5, 10, 17, 18, 20, 48, 49, 52, 54
OOP object-oriented programming 1, 2, 3, 4, 7, 10, 20, 28, 49, 51, 52, 54

UI user interface ix, 15, 20, 22, 23, 26, 27, 29, 30, 36, 37, 40

1 Introduction

This chapter is an introduction to the topic to be approached in this work. Section 1.1 and Section 1.2 start by explaining the context and motivation for this dissertation. Section 1.3 addresses the objectives and expected contributions for this MSc thesis, and in Section 1.4 some research questions this thesis aims to answer are presented. Section 1.5 clarifies the intended meaning of some terms used in this thesis that could possibly lead to an ambiguous interpretations. Finally, in Section 1.6, this document's structure is described by giving a short description of each subsequent chapter.

1.1 Context and Description

This thesis explores how the Octave programming language supports the object-oriented (OO) paradigm.

GNU Octave [7] is a programming language mainly intended for numerical computations. It offers a free and open-source alternative to MATLAB, the most popular software in the field. Octave supports object-oriented programming (OOP) [16]. However, to our knowledge no studies have been made on Octave's object-oriented features and capabilities.

In the past, the well-known Gang-of-Four (GoF) design patterns [11] served as a basis to assessments of languages in terms of their support for composition and modularity mechanisms. The GoF patterns present a significant variety of composition and design problems and have been implemented in numerous programming languages.

1.2 Motivation

Octave has been a rising force in the field of numerical computation. It benefits from being mainly compatible with MATLAB and having similar syntax, making Octave easy to pick up for those already familiar with MATLAB and, being free, it provides a good budget alternative having become especially popular among students. As it would be expected from a paid and licensed software as well as an industry leader, MATLAB still offers more

functionality but as Octave develops and catches up in both the variety and quality of its features, it continues to establish itself as an alternative worth consideration. One of said developments was the introduction of object-oriented programming to Octave which brought about new possibilities. It gives its users the ability to manipulate data differently, as classes can protect internal properties, and facilitates data encapsulation while also providing OOP specific capabilities such as inheritance and function overloading.

Two important concepts to object-oriented programming are modularity [22] and module composition. A system designed with modularity in mind increases productivity in the software development process while also making software components reusable and facilitating in the debugging and maintenance of large and complex software.

Building on the importance of modularity and composition in OOP, the motivation behind this dissertation lies in the necessity of an assessment of Octave's support for these concepts as it is a matter that should be considered as Octave expands its object-oriented programming features.

1.3 Objectives and Expected Contributions

Taking into account the aforementioned context, it is clear there is a void to be filled in the available documentation relating to Octave, particularly to its OO features. This MSc thesis will aim to provide an assessment of Octave by evaluating its support for modularity and module composition through a complete implementation of the GoF design patterns and consequent analysis. To our knowledge, no such assessment of Octave exists yet and hopefully it can serve as a valuable contribution to the existing literature on the language. Accordingly, the expected contributions of this work are the Octave implementations of two complete collections of the GoF patterns and the mentioned assessment.

1.4 Research Questions

This section presents some research questions that were considered during the writing of this thesis:

- How effective is Octave's implementation of object-oriented programming concepts compared to other languages such as Java?
- What are the best practices for implementing the GoF design patterns in Octave, and how do these practices compare to those in Java?
- What classic object-oriented language features that Octave does not support would be useful when implementing the GoF design patterns?
- Does Octave provide the necessary mechanisms to support the principle of "separation of concerns" in software?

- How can concepts like inheritance and polymorphism be emulated in Octave, and what are the limitations of these approaches?

1.5 Terminology Used in This Thesis

This section aims to clarify the intended meaning of some of the terms used in this thesis.

- **Scenario/Example** - Following the example of Monteiro et al [17], this thesis uses the term "scenario" to refer to the idea or metaphor used to set up a group of classes comprising a given pattern example. For instance, Cooper's scenario for *Visitor* is based on the idea of computing the vacation days of employees [5] (Section 3.7). The study by Hannemann et al [12] uses a very different scenario about traversing a tree structure. Each implementation of a given pattern requires a suitable scenario. We use the term example to refer to a specific implementation in a given language of a scenario for a pattern. Each scenario gives rise to at least one example for each different language.

- **Interface** - The word interface can assume different meanings in the context of object-oriented programming. For the purposes of this thesis, it is important to make the distinction between two of these definitions: "Interface" can be used to refer to Java interfaces, like in Section 3.3, abstract classes that aim to increase abstraction; it can also be used to refer to the set of methods available to be called on an object by a client, like in Section 3.6.

- **Arrays/Lists** - In Java, the most notable difference between a list and an array is that an array's length cannot be revised (static) while a list is resizable (dynamic). On the other hand, Octave's arrays are not declared with a set length so, when arrays are mentioned in this thesis, it is important to note that the term does not refer to static arrays like in Java.

- **Visibility/Access Rights** - In the Octave documentation, the term "Access rights" represents the same concept as the word "visibility" usually does in an object-oriented programming context and is used throughout this thesis.

- **Property/Variable** - In Octave, a variable associated with a class is referred to as a property.

1.6 Document Structure

The rest of this document is structured as follows:

- **Chapter 2 (Octave)** – This chapter provides an overview of Octave with a focus on the features that play an important role in the current work (thesis).

- **Chapter 3 (Octave Design Patterns Implementations)** – In this chapter, a brief introduction to the concept of Design Patterns is made and some of the Octave implementations are illustrated. The chosen pattern implementations relate to certain OO features.
- **Chapter 4 (Analysis on Object-Oriented Programming in Octave)** – Building on the implementations shown in the previous chapter, chapter 4 examines object-oriented programming in Octave, relating it to some essential OO concepts associated with modularity.
- **Chapter 5 (Related Work)** – In this chapter, a survey of relevant studies relating to the topic of this thesis is presented.

2

Octave

This chapter provides an overview of the Octave programming language, an open-source numerical computing environment. Section 2.1 provides some basic information and context on Octave, followed by an overview of its basic syntax in Section 2.2. Section 2.3 highlights some of Octave's most relevant object-oriented features and properties, which are the focus of this thesis, and Section 2.4 gives a brief introduction to Octave's graphic capabilities. Finally, in Section 2.5 some helpful information on Octave is provided.

This chapter intends to provide readers with a solid foundation in Octave for a better understanding of the following analysis.

2.1 Octave Introduction

2.1.1 Introduction to Numerical Computation

The solving of complex numerical problems can be greatly aided by the use of computers. This is referred to as "Numerical Computation" and has long been widely used in many fields of engineering and science. The exponential growth in computing power achieved in the last 60 years has increased the capabilities of this technology allowing it to perform more complex numerical analysis and provide detailed and realistic mathematical models and consequently improved the relevance of this field of study.

In its early stages, numerical computation was implemented using languages such as Fortran, C and Algol but it was far from an efficient process. The rise in importance and usage of this technology made evident the need for a dedicated numerical computing framework that could clearly and efficiently define a scientific problem. The definition of library functions that can just be used as and when needed frees the user from having to write them down each time for different problems. With this, the user-base, which mostly consists of engineers and mathematicians, can use such libraries to focus on defining the problems rather than writing efficient code.

With said framework in mind, a lot of new software was released over the years and the one that garnered the most interest was MATLAB. However, being commercial software,

MATLAB comes with a price as well as with a restrictive license. There was a need for a free, open-source alternative and that is the void Octave came to fill.

2.1.2 Octave Overview and Capabilities

First released in 1994, GNU Octave [8] is a programming language primarily intended for numerical computations. It has extensive tools for solving common numerical linear algebra problems, finding the roots of nonlinear equations, integrating ordinary functions, manipulating polynomials, and integrating ordinary differential and differential-algebraic equations. It also provides built-in visualization tools that allow users to create plots, graphs and charts.

Octave is written in C++ and is easily extensible and customizable via user-defined functions written in Octave's own language, or using dynamically loaded modules written in C++, C, Fortran, or other languages. There are various specialized packages available as well. This deepens the capabilities of the main program and the large base of library functions makes Octave a great choice for defining numerical problems.

The syntax of Octave resembles that of MATLAB and an Octave program usually runs unmodified on MATLAB. However, because MATLAB has a larger function set, the reverse does not always work, especially when the program makes use of its specialized add-on toolboxes.

GNU Octave is free software under the terms of the GNU General Public License and runs on GNU/Linux, macOS, BSD, and Microsoft Windows.

2.1.3 Octave Language Properties

- **High-Level.** Octave is a high-level language and as such has strong abstraction, hiding a lot of its most complex logic to allow the users to concentrate on solving the theoretical part of numerical problems, focusing on the definition of appropriate matrixes, expressions and variables instead of concerns like memory management. This makes it suitable to be used by those with less programming experience that may come from different backgrounds such as engineering, science or mathematics.

- **Structured.** Like most modern programming languages, Octave adheres to the structured programming paradigm which facilitates the creation of programs with readable code and reusable components.

- **Interpreted.** Octave is an interpreted language, and in such languages interpreters run through a program line by line and execute each command . This allows Octave to offer advantages such as the possibility of dynamic typing and a generally smaller program size.

- **Dynamic and Weakly Typed.** As a dynamic language, Octave offers flexibility and the need for less code by allowing many common programming behaviours to be executed at runtime. Furthermore, as a weakly typed language, it lets the user work around the type system. As mentioned in Octave's documentation, "variables in Octave do not have fixed types, so it is possible to first store a numeric value in a variable and then to later use the same name to hold a string value in the same program" and "it is possible to call a function with arguments, that probably cause errors or might have undesirable side effects" [7].

- **Duck Typing.** Octave supports "Duck Typing" [6], a concept related to dynamic programming and polymorphism. The term comes from the phrase "If it walks like a duck and it quacks like a duck, then it must be a duck". It implies that an object's type is less significant than the methods it defines. Duck typing does not use any type-checking at all. The presence of a given method or attribute is verified instead.

2.2 Basic Syntax

Octave programs consist of a list of function calls or a script and are defined in m-files (files with the suffix .m). The syntax is matrix-based and provides various functions for matrix operations. It supports various data structures and allows for object-oriented programming. It supports many common C standard library functions, and also certain UNIX system calls and functions. However, it does not support passing arguments by reference to avoid unnecessary duplication, although function arguments are copy-on-write.

2.2.1 Basic Input Guidelines

With Octave, the operations to be executed can be typed in the prompt or read from scripts, which are m-files. They are imported by calling the file name without the suffix and behave as if their content was typed in line by line.

Commas (",") and semi-colons (";") are used to split commands within a line, the difference being that when the latter is used, the result of the operation is not displayed.

Ellipses ("...") indicate that an expression continues into the next line and both the percentage symbol ("%") and the sharp sign character ("#") mark a line comment. Any text following the either of these characters is ignored by the Octave interpreter and not executed.

It is also important to keep in mind that Octave is case sensitive.

2.2.2 Variables

As mentioned in Subsection 2.1.3, Octave is a dynamic language and therefore does not require any type declaration or dimension statements for its variables.

Some variable naming rules are:

- The name of a variable must be a sequence of letters, digits or underscores, but it may not begin with a digit.
- Octave does not enforce a limit on the length of variable names.
- Names that begin and end with two underscores are reserved for internal use by Octave.

There is one automatically created variable with a special meaning. The *ans* variable always contains the result of the last computation where the output was not assigned to any variable.

Variables can be declared as:

Global	Can be accessed anywhere within Octave
Persistent	Are local to a particular function and are not visible elsewhere Maintain their values through multiple calls of that same function
Local	Are only visible in their scope. The default status of variables

Table 2.1: Variable types in Octave

2.2.3 Data Types

Octave allows its users to define their own data types by writing a small amount of C++ code. It also provides built-in data types which are:

- Numeric Objects (real and complex scalars and matrices)
- Ranges
- Character Strings
- Data Structure Objects
- Cell Arrays

2.2.4 Functions and Arguments

Functions in Octave follow the same naming conventions as variables and do not need to be loaded every time they are used. The user just needs to save it in a m-file.

Functions are defined as such:

```
function name (arg-list)
    ...
endfunction
```

Listing 2.1: Function definiton example

To assist in dealing with functions where the number of arguments and return values may be unknown and problems that may arise from such situations, Octave provides four very useful statements:

varargin	Parameter that indicates that a function takes a variable number of input variables
varargout	Parameter that indicates that a function returns a variable number of output arguments
nargin	Automatic variable that is initialized to the number of arguments a function received when it was called
nargout	Automatic variable that is initialized to the number values that are expected to be returned

Table 2.2: Argument and return value parameters

Finally, it is important to note that when calling methods belonging to a class, the first argument of the function is always the object in which it was called as can be seen in the code excerpts throughout Chapter 3.

2.2.5 Errors and Warnings

In Octave, to signal when a program reaches a state where it should not continue, the error function is typically used. When the error function is called, it stops the execution of all following code, prints the provided message and returns to the Octave prompt. To detect, handle and properly fix errors the *try/catch* blocks and *unwind_protect/ unwind_protect_cleanup* blocks are the most useful. An example of use of the error function can be found in Section 3.3.

Warnings are similar to errors but have less impact on the program and the execution is not stopped. Since they are not fatal, warnings cannot be caught with the try statement. Warnings are communicated through a warning function. Besides the warning message, it is possible to assign an identifier string to the warning. Using said identifier makes it possible to deactivate specific warnings if the user does not deem them to be worthy of attention.

2.3 Octave's Object-Oriented Features

Object-oriented programming (OOP) is the most popular programming paradigm and it consists on the concept of classes, that contain data in the form of attributes and procedures in the form of methods, and objects which are instantiations of said classes. This allows for a simplification of structures that could otherwise be much more complex and favours reusability (through the objects themselves). Since the release of version 3.2, in 2009, Octave [7] has supported object-oriented programming, which is of particular relevance to the study of design patterns and this thesis. Octave allows the creation of user-defined classes. This section presents a basic summary of class handling in Octave and some of OOP's most common properties as supported by Octave.

2.3.1 "Old Style" Classes and *Classdef* Classes

There are two types of classes in Octave: "Old Style" classes (as referred to in Octave's own documentation) and *classdef* classes.

"Old style" classes are defined through a directory and its methods are represented by separate m-files in that folder. This is further explained in Subsection 2.3.2.

Since version 4.0, released in 2015, Octave has limited support for *classdef* classes, which have a much closer behaviour to classes in other object-oriented languages. Unlike the aforementioned "old style" classes, *classdef* classes can be defined within a single m-file. Other capabilities unique to *classdef* classes are the definition of static methods and access rights for properties and methods.

Classdef classes can be further divided into *value* classes and *handle* classes and this distinction is mainly based on their behaviour regarding variable assignment as discussed in Subsection 2.3.5.

Finally, its important to note the implementation of *classdef* classes in Octave is incomplete, resulting in several know bugs and missing features. A comprehensive list of these shortcomings can be found in the appropriate section in Octave's Wiki [20].

2.3.2 Creating a Class

- **Creating an "old style" class**

  Classes in Octave are defined by a directory where the name is <@ + name of the class>.

  Methods of the class are files with extension <.m> in the directory.

  The constructor is in a file with the same name of the class.

  For example, class *newClassDemo* will be directory <*@newClassDemo*>, its constructor will be in <*@newClassDemo/newClassDemo.m*> and methods of the class will be in various <*@newClassTest1/methodName.m*> files as can be seen in Figure 2.1.

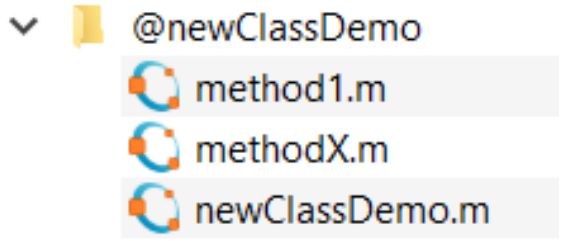

Figure 2.1: "Old style" class folder/file structure

Keep in mind that the output of the class function must be the constructor's return value. The class function takes two arguments: a structure whose fields will be used as the class' fields and the name of the class itself.

```
function p = newClassDemo (flds)
    p = class (flds, "newClassDemo");
endfunction
# this function returns an object p of type newClassDemo whose properties
# are the same as the fields of structure flds
```

Listing 2.2: "Old style" class constructor example

```
p = newClassDemo (flds);
```

Listing 2.3: Calling an "old style" class' constructor example

- **Creating a *classdef* class**

  Classdef classes are much simpler to create than "old style" classes. They can be defined in a single m-file and have a structure closer to classes used in other languages as seen in Listing 2.4.

```
classdef newClassdefDemo
  properties
    ...
  endproperties

  methods
    ...
  endmethods
endclassdef
```

Listing 2.4: Creating a *classdef* class example

If no indication is given, a *classdef* class is set as a *value* class, like the one presented in Listing 2.4. To create a *handle* class, the new class should be declared as derived from the abstract *handle* type with the inheritance notation displayed in Subsection 2.3.6.

To create an object of a *classdef* class, the constructor is still called in the same manner as shown in listing Listing 2.3.

2.3.3 Class Member Access Rights

In Octave, the access rights of properties and methods can be set to *public*, *protected* and *private*:

public	The properties/methods can be accessed from everywhere
private	The properties/methods can only be accessed from class methods but not from from subclasses of that class
protected	The properties/methods can only be accessed from class methods and from subclasses of that class

Table 2.3: Access Rights of Class Members in Octave

By default, access rights are set as *public*. To set the desired level of access security, the notation seen in Listing 2.5 is used.

```
classdef classdefVisibilityDemo
  properties (Access = private)
    ...
  endproperties

  properties (Access = protected)
    ...
  endproperties

  methods (Access = private)
    ...
  endmethods

  methods (Access = public)
    ...
  endmethods
endclassdef
```

Listing 2.5: Setting the access rights of a *classdef* class' properties and methods example

2.3.4 Overloading and Object Precedence

Octave supports function and operator overloading, the creation of separate functions with the same name but with different implementations. Any function can be overloaded, even Octave's built-in functions.

The precedence of methods and objects to be called when there are mixed objects passed to a function can be set with the use of two functions in the classes' constructors: *superiorto* and *inferiorto*.

```
function p = newClass ()
    p = class (p, "newClass");
    superiorto("someOtherClass");
endfunction
```

Listing 2.6: Setting a class as having precedence over another class example

2.3.5 Object Identity

To determine the class of an object, Octave provides functions such as the *class(obj)* function that returns the class of *obj* and the *isa(obj, classname)* function which returns true if object *obj* is of type *classname*.

Concerning object comparison in Octave, this can be done with functions such as *isequal()*. This function individually compares the fields of the provided structures but it may fail if the structs contain *NaN* values. While Octave does not support an equivalent of null references, as seen in other languages like Java, numerical attributes can have the values of *NA* ("Not Available") and *NaN* ("Not a Number"). To include the possibility of *NaN* values in object comparison, the function *isequaln()* can be used.

Noteworthy to this topic, is the distintion between *value* or *handle* classes among *classdef* classes. This separation is especially relevant for this topic of object identity as they behave differently when the object is assigned to a new variable. When working with *value* classes this assignment essentially creates a new object but if this is done with a *handle* class, the variable refers to the same object.

2.3.6 Inheritance

- **Inheritance in "old style" classes**

  "Old style" classes can be used to build new classes in Octave. The process is similar to the creation of the classes demonstrated in Subsection 2.3.2, but the new class creates the necessary fields from the given object instead of a structure.

  To make a class a child of another existing class, the parent class must be passed as the third argument of the class function as demonstrated in Listing 2.7.

```
function c = childClassDemo ()
    p = @newClassDemo ();
    c = class (c, "childClassDemo", p);
endfunction
```

Listing 2.7: Constructor of an "old style" child class example

- **Inheritance in *classdef* classes**

  Classdef classes can also inherit from other classes. Here the syntax is much simpler as seen in Listing 2.8. The properties and methods of the superclass are passed only depending of their access rights.

  Octave also supports multiple inheritance, allowing a class to inherit features from more than one parent class. This is done by listing the desired superclasses with the "&" character.

```
#Normal Inheritance
classdef childClassdefDemo < newClassdefDemo

#Multiple Inheritance
classdef childClassdefDemo < newClassdefDemo &  newClassdefDemo2
```

Listing 2.8: *Classdef* inheritance examples

It is important to note that arguments passed to the subclass constructor do not automatically carry over to the superclass constructor. Therefore, if the developer wants the superclass constructor to receive specific arguments and perform operations on them, the child class must explicitly call the superclass constructor with the appropriate notation as seen in Listing 2.9. This is demonstrated practically in Listing 3.14.

```
#Constructor of Superclass
function self = superClassdefDemo (newArg1)
    ... #operations using the argument newArg1
endfunction

#Constructor of Subclass
function self = childClassDemo (newArg1)
    super_args{1} = newArg1; #create a structure containing the arguments
     to pass to the superclass
    self = self@superClassdefDemo(super_args{:}); #explicitly call the
    superclass constructor
endfunction
```

Listing 2.9: Calling the superclass' constructor from a child class example

2.4 Octave's Graphics Features

To properly analyse and interpret numerical computations, visualization is a requirement. Producing valuable images of complex plots that aid in the understanding of numerical results is one of the most important features for this type of software and, as expected, Octave provides this functionality too.

Octave is capable of producing various types of plots in 2D and 3D formats while allowing its users to interact with them to explore the data and decorate them with additional material deemed relevant such as titles, labels, equations and other important information.

Its graphics features are one of Octave's standout points and plotting can be achieved through a large number of different functions, relating to various combinations of types of input data and desired outputs, with the plot function being the simplest one.

While Octave's graphical capabilities are mostly focused on plotting and graphs, it also offers some features for constructing graphical interfaces that interact with users. It provides I/O dialogs, a progress bar, and some UI elements for plot windows such as menus, panels, buttons and toolbars.

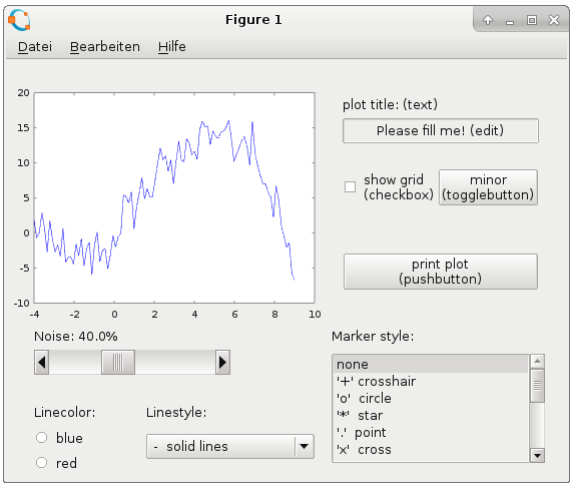

Figure 2.2: An example window showing some of the available UI elements like a *textbox*, a slider, a *listbox* and various types of buttons [29]

2.5 Helpful Information

2.5.1 Packages and Extensions

While Octave's core functionality is quite extensive on its own, there are many packages available that can extend it further. Because Octave is open-source software, it encourages its users to create and share their own extensions and programs. These packages are developed and maintained by the community and can be quite useful.

The two most popular repositories of Octave packages can be found at:

- gnu-octave.github.io/packages/
- octave.sourceforge.io/

2.5.2 IDE/GUI Survey

While Octave can be run in the command line, update 3.8 (released in 2013) contained a new graphical user interface (GUI) with an integrated development environment.

There are several available online Integrated Development Environment (IDE)s for Octave, namely the cloud IDE OctaveOnline [19], and Visual Studio Code offers Octave extensions for syntax checking, formatting and debugging.

Octave's native GUI, seen in Figure 2.3, was selected as the main tool to be used for this thesis.

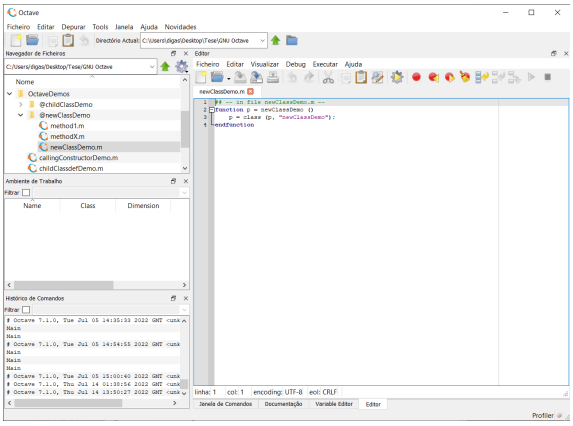

Figure 2.3: The default GUI provided by Octave

2.5.3 Forums and Other Sources of Information

As a free and open-source software, Octave relies on its users to expand the available information on the language. Interaction with other users can often bring benefits and, much like most other software, Octave has forums across the internet which may prove helpful to users, whether experienced or not.

Besides its own documentation, some of the other main sources of information on Octave are:

- wiki.octave.org/GNU_Octave_Wiki
- octave.discourse.group/
- reddit.com/r/octave/
- savannah.gnu.org

3

Octave Design Patterns Implementations

This chapter concerns the Gang-of-Four design pattern implementations produced in Octave. Section 3.1 provides an introduction on the concept of design patterns and, most specifically, on the Gang-of-Four patterns. Section 3.2 introduces the collections used as source for the implemented scenarios. Section 3.3 to Section 3.7 discuss eight pattern implementations and respective associated OO features. Finally, Section 3.8 addresses other observations relevant to the pattern implementations but not deemed worthy of its own section.

3.1 Design Patterns

Software design patterns are certified solutions to common problems in software design. Unlike functions or libraries, design patterns are not specific pieces of code but generalized concepts that can be implemented in different ways depending on the given problem and environment. A pattern details the context where it can be used as well as the relationships and interactions between the entities in which it relies.

Design patterns facilitate efficient communication between developers by providing well-known terminology for specific scenarios. They can speed up the development process by providing tested, proven development paradigms and improves code readability for those familiar with the patterns. Learning these patterns also helps inexperienced developers to learn software design in an easy and faster way.

Patterns are not really invented but instead gradually assert themselves as typical solutions to common problems in software design. When a tried and tested practice gets repeated often through various projects by the community, someone eventually puts a name to it and describes the solution in detail. That's how a pattern gets formalized. In 1987, Beck and Cunningham began experimenting with the idea of applying patterns to programming [2]. Seven years later, in 1994, software design patterns saw a great rise in popularity with the publishing of the book "Design Patterns: Elements of Reusable Object-Oriented Software" by Gamma et al [11], commonly known as the Gang-of-Four (GoF).

It should be noted that the concept of a design pattern is dependent on the terminology used as it could be argued that design patterns do not differ significantly from other forms of abstraction and the Model-View-Controller paradigm is touted as an example of a "pattern" which predates the concept of design patterns by several years.

Despite all its advantages, the concept of design patterns is not without flaws and has been criticized by many in the field of computer science. In fact, at OOPSLA 1999, the Gang-of-Four were (with their full cooperation) subjected to a mock trial, in which they were "charged" with numerous crimes against computer science. It is argued that there is an unjustified "over-use" of design patterns in situations where they are barely acceptable over well-factored implementations, leading to code that can be harder to understand and manage.

The fact remains that design patterns can be incredibly useful if used in the right situations and for the right reasons but it is important to understand that they were never meant to be shortcuts to be applied in a haphazard manner without regard for context. There is ultimately no substitute for genuine problem-solving ability in software engineering.

3.1.1 Gang-of-Four Design Patterns

The iconic book published by the Gang-of-Four [11] featured 23 unique patterns (exemplified in a C++ context) solving various problems of object-oriented design and is still considered a best-seller today. Design patterns differ by their complexity, level of detail and scale of applicability to the entire system being designed but, in the book, they are divided into three categories based on their purpose:

- **Creational Patterns** – In certain situations, the basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by controlling class instantiation of single objects or groups of related objects. Heavily relying on mechanisms such as inheritance, creational design patterns aim to increase flexibility and reuse of existing code.

- **Structural Patterns** – Structural patterns focus on class and object composition. They are intended to help define relationships between entities, as objects and classes are assembled into larger structures, while keeping these structures flexible and efficient and sometimes obtaining new functionality.

- **Behavioural Patterns** – Behavioural patterns concern communication and the assignment of responsibilities between objects. They address the way objects interact at run-time, allowing the developer to concentrate on how the objects are interconnected.

Pattern Category	Pattern
Creational	Factory Method Abstract Factory Builder Prototype Singleton
Structural	Adapter Bridge Composite Decorator Façade Flyweight Proxy
Behavioural	Adapter Interpreter Template Method Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor

Table 3.1: The GoF [11] design patterns by category

3.2 Introduction to the Design Pattern Implementations

For this thesis, two complete collections of design patterns were implemented in Octave adding up to 46 distinct pattern implementations. The scenarios were chosen from five different sets of Java implementations, predominantly from the ones written by James Cooper and Bruce Eckel and adapted and refactored by Monteiro as well as the Fluffycat collection.

Author/Name	Number of scenarios used	Source
James Cooper	15	Java design patterns: a tutorial (2000) [5]
Bruce Eckel	14	Thinking in Patterns (2003) [9]
Fluffycat	14	www.fluffycat.com/Java-Design-Patterns
Vince Huston	2	www.vincehuston.org/dp
JavaCamp	1	javacamp.org

Table 3.2: Pattern scenario source distribution

When selecting scenarios from the aforementioned sources, James Cooper's and Bruce Eckel's collections were prioritized but it was impossible to fully adapt either collection to Octave, as can be seen in Table 3.2. James Cooper's collection was preferred for providing user interface elements to help visualize the scenario however, in certain patterns, Octave does not provide the graphical features to successfully reproduce the desired UI. As for Bruce Eckel's collection, it is not complete so the other three sources had to be considered to fill in the gaps. It should also be noted that the website associated with Fluffycat's collection is unfortunately no longer accessible.

The first step in the analysis of the design pattern implementations is to compare them to their Java counterparts. As such, it is important to highlight the important role Java plays in this thesis, being one of the most widely known programming languages for OOP and serving as an appropriate benchmark for the analysis of Octave's object-oriented features.

For the purposes of conciseness and facilitating reading comprehension, and because the ultimate goal of this thesis is the evaluation of Octave as an object-oriented programming language and not the patterns themselves, the implementations were grouped according to certain topics. Section 3.3, Section 3.4 and Section 3.5 focus on valuable object-oriented features offered by Java that Octave does not support, and the solutions found to such shortcomings. Section 3.6 presents a group of four patterns that share a similar practice in its implementation while having different goals and uses. Section 3.7 considers how certain characteristics of Octave programming can be used to tackle a well-known subject in the software design community. Finally, Section 3.8 is a collection of remarks on the pattern implementations or on Octave itself that were deemed worthy of a mention in this chapter but not of its own section.

It is also noteworthy that, as explained throughout Section 2.3, *classdef* classes are the superior variant in Octave, even despite its incomplete implementation, providing much more flexibility and functionality and thus were used in all the design pattern implementations in this chapter.

3.3 Interfaces and Abstract Classes

This section concerns the design patterns: Abstract Factory, Builder, Command, Composite, Observer, State, Strategy and Visitor.

The concept of interfaces is crucial to object-oriented programming in Java, and consequently, to the sets of design patterns used as templates for the Octave implementations created in this thesis as well. However, Octave does not support this mechanism, preventing a more direct translation between languages. The "abstract class", which is present in other dynamic programming languages such as Python, is a concept similar to an interface that might be used as an alternative, but while abstract classes are meant to be

supported by Octave, its implementation is flawed and incomplete [20]. Presently, it does not seem feasible to use them.

Searching for a solution, the answer found to be closest to successfully emulate the concept of an interface was the *ExampleInterface* class displayed below, in Listing 3.1.

```
classdef ExampleInterface < handle
    methods
        function methodA (self)
            error("SUPERCLASS METHOD THAT SHOULD NOT BE CALLED -> SHOULD BE
    OVERRIDDEN");
        endfunction

        function methodB (self)
            error("SUPERCLASS METHOD THAT SHOULD NOT BE CALLED -> SHOULD BE
    OVERRIDDEN");
        endfunction
    endmethods
endclassdef
```

Listing 3.1: Octave attempt at emulating an interface

This is a simple class meant to be inherited that throws an error if its methods are called directly. However, a class like this does not offer the abstraction and security benefits a Java interface can, which brings its utility in the system into question . A possible role these attempts at interfaces could fulfil was found in some of the design pattern scenarios to be implemented. To exemplify this point, the *Abstract Factory* pattern scenario present in Cooper's collection will be used.

3.3.1 *Abstract Factory*

The *Abstract Factory* pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. Different variants of products may have different properties and the concrete class to be instantiated can be changed, even at run-time, by using a different factory.

This pattern allows the creation of different types of objects without sharing the concrete classes' implementation with the client, avoiding tight coupling between concrete products and client code. By creating objects through an interface for a class of the product family, the possibility of type mismatch is excluded which is also helpful when the concrete class to be instantiated is not known in advance.

Finally, by isolating object creation with an *Abstract Factory*, new variants of products can be easily introduced without breaking existing client code. However, it should be noted that this pattern can sometimes make code more complex as it consists of the creation of many new interfaces and classes.

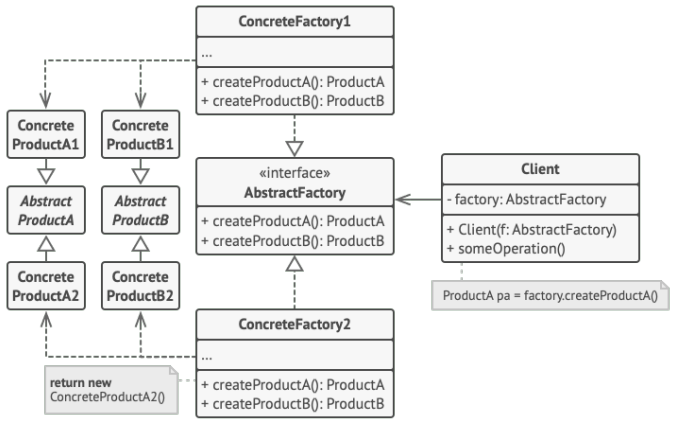

Figure 3.1: Diagram exemplifying a possible structure of implementation of the Abstract Factory pattern [27]

The scenario chosen for this section is the "GardenMaker Factory" by James Cooper which uses the *Abstract Factory* pattern in the planning of a garden layout and provides a user interface for better visualization.

In this scenario, there are three types of garden (perennial, annual and vegetable) and, for each of these types of garden, it is required to know what types of plants would do well in shade, as well as in the centre or the border of the garden. Accordingly, the abstract factory class in this example is *AbstractGardenFactory* that is implemented by three concrete factories corresponding to the types of garden and containing three methods that return the adequate centre, border or shade plant. The plants themselves are represented by a simple *Plant* object that only contains the plant's name. This setting is illustrated in a class diagram in Figure 3.2.

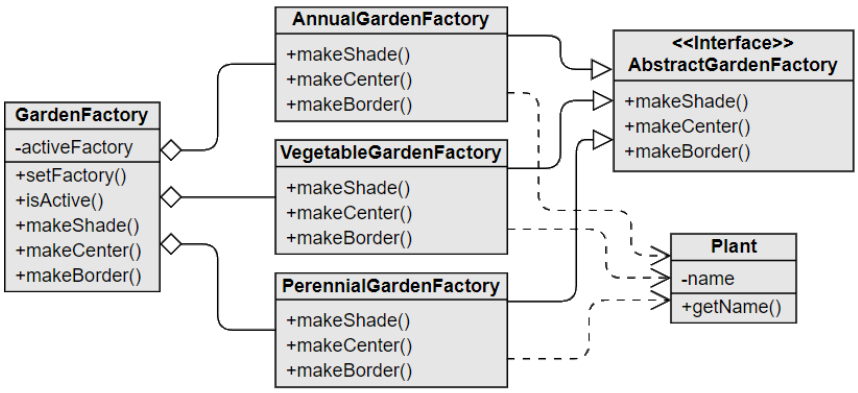

Figure 3.2: Class Diagram for the Java implementation of the "GardenMaker Factory" scenario

Regarding the user interface, it consists of three parts: at the top a canvas representing the garden layout, on the left the user can select the desired type of garden which creates the corresponding garden factory and on the right the user chooses the plant category which displays the name of the adequate plant in the corresponding space (Figure 3.3).

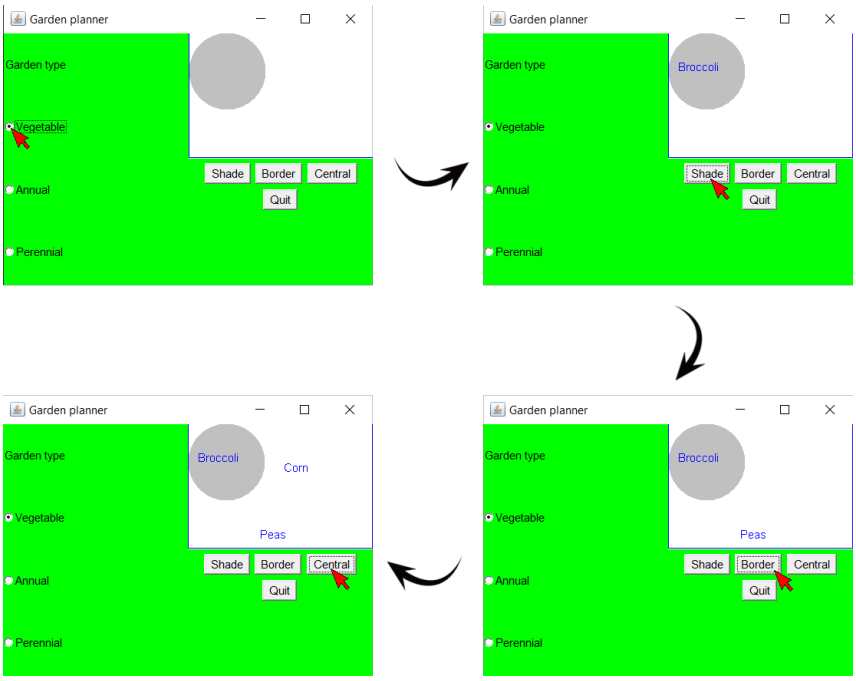

Figure 3.3: *Abstract Factory* implementation in Java UI

Because all the three concrete factories implement the *AbstractGardenFactory* interface, and therefore implement methods with the same signatures, the *GardenFactory* class can have a *activeFactory* variable declared with that interface that can be any of the factories while calling the same methods on it.

```
1 public interface AbstractGardenFactory {
2   public abstract Plant makeShade();
3   public abstract Plant makeCenter();
4   public abstract Plant makeBorder();
5 }
```

Listing 3.2: Java *AbstractGardenFactory* interface

```
1  public class VegetableGardenFactory implements AbstractGardenFactory {
2    public Plant makeShade() {
3      return new Plant("Broccoli");
4    }
5
6    public Plant makeCenter() {
7      return new Plant("Corn");
8    }
9
10   public Plant makeBorder() {
```

```
    return new Plant("Peas");
  }

}
```

Listing 3.3: Java *VegetableGardenFactory* class

```
public class GardenFactory {
  private AbstractGardenFactory activeFactory = null;
  private AbstractGardenFactory annualFactory = new AnnualGardenFactory();
  private AbstractGardenFactory perennialFactory = new PerennialGardenFactory
    ();
  private AbstractGardenFactory vegetableFactory = new VegetableGardenFactory
    ();

  public void setFactory(String gtype) {
    if (gtype.equals("Perennial"))
      activeFactory = perennialFactory;
    else if (gtype.equals("Annual"))
      activeFactory = annualFactory;
    else activeFactory = vegetableFactory; // default

    System.out.println("Garden factory set to " + gtype);
  }
  public boolean isActive() {
    return activeFactory != null;
  }
  public Plant makeShade() {
    return activeFactory.makeShade();
  }
  public Plant makeCenter() {
    return activeFactory.makeCenter();
  }
  public Plant makeBorder() {
    return activeFactory.makeBorder();
  }
}
```

Listing 3.4: Java *GardenFactory* class

On the other hand, the lack of type-checking in Octave and the fact its variables do not have a fixed type means that the *activeFactory* property in the *GardenFactory* class can still be an object of any of the concrete factories without the presence of the *AbstractFactory* interface (Figure 3.4).

```
classdef GardenFactory < handle
    properties (Access = private)
        activeFactory
        annualFactory = AnnualGardenFactory();
        perennialFactory = PerennialGardenFactory();
        vegetableFactory = VegetableGardenFactory();
    endproperties

    methods
        function setFactory (self, gtype)
            if (strcmp(gtype, "Perennial") == 1)
                self.activeFactory = self.perennialFactory;
            elseif (strcmp(gtype, "Annual") == 1)
                self.activeFactory = self.annualFactory;
            else
                self.activeFactory = self.vegetableFactory;
            endif
            printf(cstrcat("Garden factory set to ", gtype, "\n"));
        endfunction

        function retBol = isActive (self)
            if (isempty(self.activeFactory))
                retBol = false;
            else
                retBol = true;
            endif
        endfunction

        function retPlant = makeShade (self)
            retPlant = self.activeFactory.makeShade();
        endfunction

        function retPlant = makeCenter (self)
            retPlant = self.activeFactory.makeCenter();
        endfunction

        function retPlant = makeBorder (self)
            retPlant = self.activeFactory.makeBorder();
        endfunction
    endmethods
endclassdef
```

Listing 3.5: Octave *GardenFactory* class

```
classdef VegetableGardenFactory < AbstractGardenFactory
    methods
        function retPlant = makeShade (self)
            retPlant = Plant("Broccoli");
        endfunction

        function retPlant = makeCenter (self)
```

```
8             retPlant = Plant("Corn");
9         endfunction
10
11        function retPlant = makeBorder (self)
12            retPlant = Plant("Peas");
13        endfunction
14    endmethods
15 endclassdef
```

Listing 3.6: Octave *VegetableGardenFactory* class

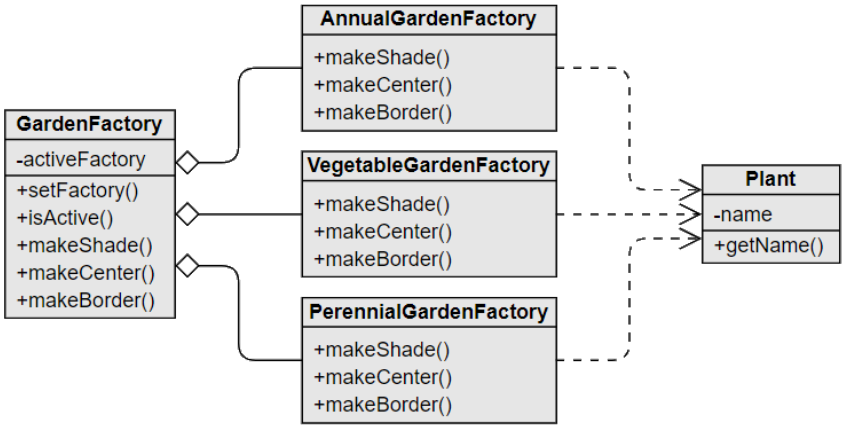

Figure 3.4: Class Diagram for the Octave implementation of the "GardenMaker Factory" scenario

In fact, the removal of the *AbstractFactory* interface has no impact in the functionality of this implementation, questioning once more its usefulness. Figure 3.5 shows the UI produced by the Octave implementation and the similar results achieved in comparison with the Java implementation.

Taking this implementation as a study case for the utility of interface-like classes in Octave, the inclusion of an *AbstractFactory* interface in the mould of the one presented in Listing 3.1 would not offer any additional functionality. However, questioning the extensibility of this program gives rise to the idea that these "interfaces" have documentation value. Because the *GardenFactory* class calls methods on the *activeFactory* variable without knowing its type of object, all the factories must implement those necessary methods. As such, any new factories to be added to the system, a *FruitGardenFactory* for example, must share the same method signatures. This is enforced in Java through compile errors issued when a class does not contain a method from an interface it implements. However, because Octave does not offer the same safe behaviour, the previously demonstrated interface-like classes can be helpful for the developer as blueprints of what new classes to be added need to look like to assure consistency within the system.

While *Abstract Factory* was chosen for demonstrative purposes, this concept concerns several other patterns that tend to implement an interface with many distinct classes that

are essential to the pattern's functioning. *Composite* is usually implemented in a tree-like structure that requires its leaf and node object to implement certain identical method signatures and, much like the factories in the *Abstract Factory* pattern, *Builder*, *Command*, *Observer*, *State*, *Strategy* and *Visitor* also use elements sharing the patterns' naming that match this notion.

Figure 3.5: *Abstract Factory* implementation in Octave UI

3.4 Polymorphism in Data Structures

This section concerns the design patterns: Command, Composite and Observer.

As mentioned in Subsection 2.3.1, the implementation of *classdef* classes is flawed and incomplete. One of the areas of object-oriented programming in Octave that causes the most issues due to acknowledged but unaddressed bugs is the storage of *classdef* objects in data structures (such as the bug solved by the *objvcat* function as explained in Subsection 3.8.3). This section focuses on the lack of support for polymorphism in Octave's arrays, the obstacles this might cause and the proposed workaround.

When data structures in Java are created, the type of objects to be contained in the structure is declared and, if this declaration is done through an interface, different object types can be stored in the data structure as long as they implement said interface. The same behaviour can also be verified in cases of inheritance.

On the other hand, an array in Octave is operated as such:

```
1  obj1 = Class1();
2  obj2 = Class2();
3  #both Class1 and Class2 are subclasses of the same class
4
5  arr = [];
6  #array declared without an associated type
7
8  arr = objvcat(obj1, arr);
9  #obj1 is inserted with success in the array
10
11 arr = objvcat(obj2, arr);
12 #an error is thrown stating that objects of type Class2 cannot be inserted in
       an array of Class1 objects
```

Listing 3.7: Octave array usage

As shown in Listing 3.7, there is no declaration of type and the array simply assumes that all its elements should be of the same class of the first object entered in the array. Unlike Java, it does not recognize subclasses of the first element's class as being of the same type either, hence it does not accept them (no inheritance polymorphism).

Instances when objects of distinct but related types should be held in the same collection are hardly rare so this is a very useful feature in object-oriented programming and, consequently, in the design pattern implementations in this thesis.

"Cell arrays" are Octave data structures worthy of a mention in this section as they provide a way to store information of objects of any type. However, this is done by transforming all the objects entered in the array into "cells", struct-like objects that maintain the class' properties but do not allow for the calling of its methods, making them unfit for use in OOP.

The solution found to this problem in Octave will be exemplified through the *Observer* pattern.

3.4.1 *Observer*

The *Observer* design pattern aims to notify a set of related interested objects of events that happen in a certain object. With this pattern, an object, defined as the "subject", maintains a list of interested objects, the "observers", and alerts them when an event occurs, usually by triggering one of their methods. The *Observer* pattern thus permits the creation or removal of relationships between objects without having to alter the code, but simply changing the list of observers.

Although Octave supports "listeners", a mechanism with conceptual similarities to the *Observer* pattern, they can only be used with graphical objects in this language and not user defined classes, which makes them unsuitable for the purposes of this thesis.

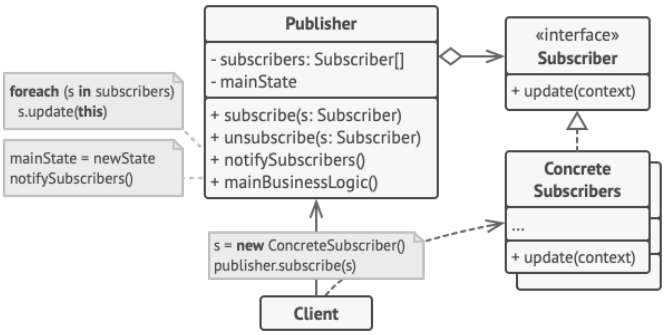

Figure 3.6: Diagram exemplifying a possible structure of implementation of the *Observer* pattern [27]

As the subject in this pattern is supposed to keep a record of its observers, which can be different types of objects, this is an opportunity to address the problem of the lack of polymorphism in Octave arrays.

Cooper's *Observer* scenario consists of an user interface with three types of windows: the first has three buttons corresponding to three colours (red, blue and green) to be pressed by the user, the second is simply a panel that assumes the chosen colour and the third is a written list that keeps track of the user's choices. As such, in the context of the *Observer* pattern, the first window plays the role of the subject and the other two are the observers that get notified whenever the user presses a button and behave accordingly. This UI is displayed in Figure 3.7.

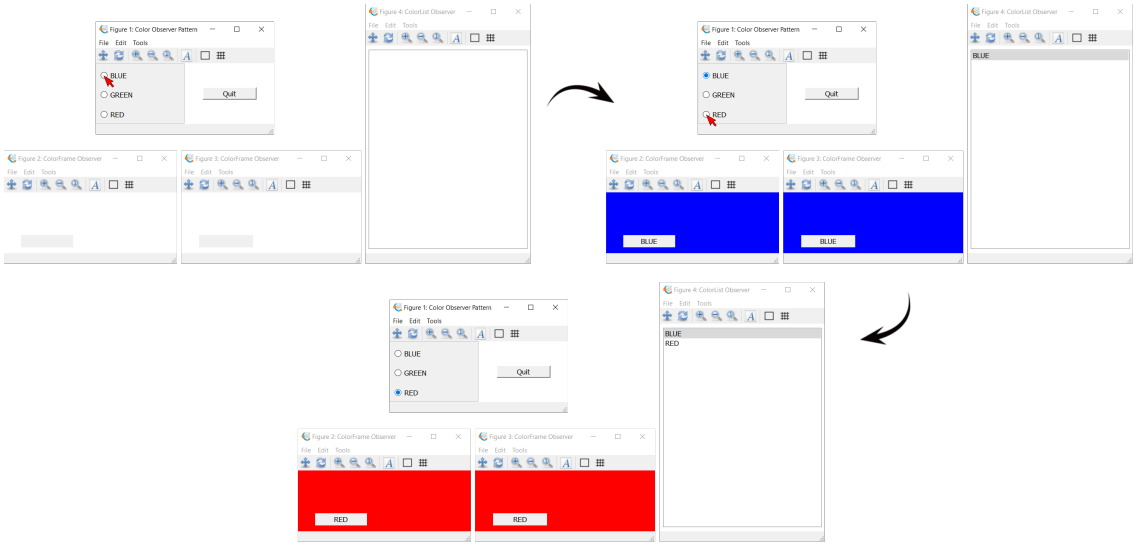

Figure 3.7: *Observer* implementation in Octave UI

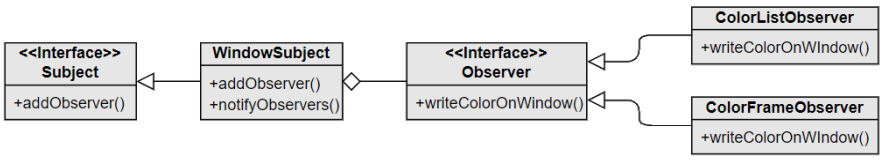

Figure 3.8: Class Diagram for the Java implementation of the "Color Observer" scenario (graphical content omitted)

The initial problem found when implementing this scenario in Octave is the storage of objects of both *ColorFrameObserver* and *ColorListObserver* objects in the *WindowSubject* object. The most obvious answer to this problem could be to just split the *observers* array into two for each type of observer. However, this is an inadequate workaround as it would require code duplication, iteration through all data structures when notifying observers and this technique does not scale to new types of observers, in case they are added to the system.

The chosen solution consists of wrapping the objects before inserting them into the array using a simple wrapper with only one property and the method to return it, as seen in Listing 3.8. This practice makes use of the fact that Octave is a weakly typed language so that the property *obj* in the *ObjectWrapper* class can be of any type. In this setting, the array will only contain objects of type *ObjectWrapper* thus introducing polymorphism to data structures in Octave through a middleman.

```
classdef ObjectWrapper < handle
    properties (Access = private)
        obj
    endproperties
    methods
        function self = ObjectWrapper (obj)
            self.obj = obj;
        endfunction

        function retObj = getObj (self)
            retObj = self.obj;
        endfunction
    endmethods
endclassdef
```

Listing 3.8: *ObjectWrapper* class

With this mechanism, it becomes possible for the *WindowSubject* class to store and manage the various types of observers in a single array while keeping the needed functionality. This is true even if new classes of observers were to be introduced in the future.

```
classdef WindowSubject < handle
    properties (Access = private)
        ...
        observers
    endproperties
    methods
        function self = WindowSubject ()
            ...
        endfunction

        function addObserver (self, obs)
           self.observers = objvcat(self.observers, ObjectWrapper(obs));
        endfunction

        function addObserver (self, obs)
           self.observers = objvcat(self.observers, ObjectWrapper(obs));
        endfunction
    endmethods
    methods (Access = private)
        function notifyObservers (self, ev, d)
            if(get(ev, 'value') != 0)
              col = get(ev, 'string');
              for k = 1:numel(self.observers)
                obi = self.observers(k).getObj();
                obi.writeColorOnWindow(col);
              endfor
            endif
        endfunction

        function quitObservers (self, ev, d)
```

```
31              for k = 1:numel(self.observers)
32                  obi = self.observers(k).getObj();
33                  obi.closeSelf();
34              endfor
35              close(self.fig);
36          endfunction
37      endmethods
38  endclassdef
```

Listing 3.9: Array usage in *WindowSubject* class

It is noteworthy that the Octave implementation of this scenario also matches the circumstances mentioned in the previous section and, accordingly, observer and subject interface-like classes were deemed unnecessary and are not included.

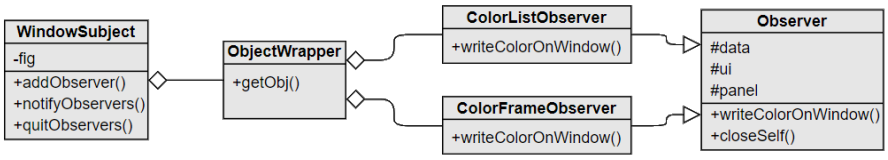

Figure 3.9: Class Diagram for the Octave implementation of the "Color Observer" scenario

Along with *Observer*, *Command* and *Composite* stand out as the prime candidates for the application of this concept. It is typical when using the *Command* pattern for there to be the need to store several command-type objects for the purposes of delayed execution or record keeping. Similarly, when implementing the *Composite* pattern's aforementioned tree-like composition, the nodes of the structure often possess a data structure containing objects that can be of different types (leaves or other nodes).

3.5 *Singleton* and Static Properties

This section concerns the design pattern: Singleton.

Singleton is a design pattern that aims to ensure a class has only one instance and provide a global point of access to it.

The scenario provided by James Cooper for this pattern is quite simple. It consists of a *Printer* class that only allows the creation of one instance of itself.

```
1  public class Printer {
2    static boolean instance_flag = false; // true if 1 instance
3
4    public Printer() throws SingletonException {
5      if (instance_flag)
6        throw new SingletonException("Only one printer allowed");
7      instance_flag = true; // set flag for 1 instance
8      System.out.println("printer opened");
9    }
10
11   public void finalize() {
12     instance_flag = false;
13   }
14 }
```

Listing 3.10: Java *Printer* class

This is done in the constructor by keeping a boolean static variable to indicate whether an instance of the *Printer* class already exists. If there are no objects of this class the constructor call is successful, otherwise an exception is thrown. While this a straightforward concept in Java, it cannot be directly replicated in Octave as it relies on static variables and their capability to preserve their value regardless of scope, which this language does not support. Octave does provide global variables which would produce the same results but, being accessible from anywhere in the system, these type of variables present a major security risk.

The workaround to this obstacle was provided by a user in the Octave Discourse forum, mentioned in Subsection 2.5.3, makes use of persistent variables in static methods [18] to emulate the behaviour of Java's static variables.

A variable that has been declared persistent within a function will retain its contents in memory between subsequent calls to the same function (Subsection 2.2.2). They differ from the aforementioned global variables by being local in scope to a particular function and not visible elsewhere. While it can be said that the behaviour of these variables is quite close emulating Java's static variables, an important distinction needs to be considered: while Java's static variables belong to a class, Octave's persistent variables can only be contained within functions and, even though they can retain its value between subsequent calls of said function, the variable does lose its value when the encapsulating

function is removed from memory even if the class itself was not. Fortunately, there is a common practice in Octave used to prevent persistent variables from being removed from memory. That is *mlock*, a function that locks the current function into memory so that it cannot be removed when cleared.

```
classdef Printer < handle
    methods (Access = private)
        function self = Printer ()
            ...
        endfunction
    endmethods

    methods (Static)
        function this = GetInstance (varargin)
            persistent my_singleton_instance;
            mlock ();

            if isempty (my_singleton_instance)
              my_singleton_instance = Printer (varargin{:});
              printf("Printer opened\n");
            else
                error("Only one printer allowed");
            endif

            this = my_singleton_instance;
          endfunction
    endmethods
endclassdef
```

Listing 3.11: Octave *Printer* class

In this implementation (Listing 3.11), the default constructor is made private to prevent other objects from calling it. Instead, a new static method is used as an intermediary to handle the persistent variable and, if there are no other instances of the object, to call the private constructor and return the created instance. It is important that the *getInstance* method is declared as static so that it can be called directly from the class and not from an object instance.

3.6 Wrappers

This section concerns the design patterns: Adapter, Decorator, Façade and Proxy.

This section relates to the patterns that are implemented through composition by wrapping objects to affect their interaction with a client, while achieving different goals.

3.6.1 *Adapter* and *Façade*

Adapter and *Façade* change the methods available to the client. *Adapter* is a simple design pattern intended to connect objects with class interfaces originally deemed incompatible. The connection of the objects with mismatching interfaces can be done through an adapter object that shares the desired interface and wraps the problematic object, making the necessary conversions to preserve the intended behaviour.

On the other hand, the *Façade* pattern consists of using a class to offer the client a simplified interface for a more complex subsystem. This façade class performs the needed processes of calling several other classes while hiding these objects from the client and providing it only with the features it desires. In doing so, it allows for the isolation of the subsystem complexity from the client code.

Pattern	Wrapped Objects	Interface
Adapter	One	Transformed
Façade	Multiple	Reduced

Table 3.3: *Adapter*/*Façade* functionality comparison

Façade Octave Implementation

FluffyCat's *Façade* scenario consists of the making of a cup of tea containing three elements: a teacup, water and a teabag. Each of these components is represented by their own class and has their own methods to be called to make the cup of tea.

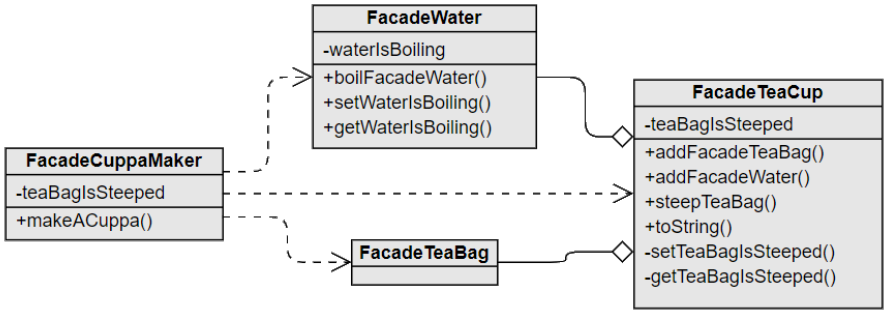

Figure 3.10: Class Diagram for the Octave implementation of the "MakeACuppa" *Façade* scenario

The façade class that ties them together is *FacadeCuppaMaker*, providing the client with only the one necessary method, *makeACuppa*, and abstracting all the underlying logic.

```
classdef FacadeCuppaMaker < handle
    properties (Access = private)
        ...
    endproperties
    methods
        ...
        function retCup = makeACuppa (self)
            retCup = FacadeTeaCup();
            teaBag = FacadeTeaBag();
            water = FacadeWater();
            retCup.addFacadeTeaBag(teaBag);
            water.boilFacadeWater();
            retCup.addFacadeWater(water);
            retCup.steepTeaBag();
        endfunction
    endmethods
endclassdef
```

Listing 3.12: Octave *FacadeCuppaMaker* class

3.6.2 *Decorator* and *Proxy*

Similarly to *Adapter*, *Decorator* and *Proxy* also consist of the wrapping of a single object but they differ from the two patterns in the previous section by providing the same methods as the target object and delegating to it the requests they receive.

The *Proxy* pattern consists of using a wrapper class as a middleman between the client and the target object. This proxy class can control access to the object, forwarding requests to it as it sees fit, making this pattern useful for safety purposes and allowing lazy initialization of the target object if necessary.

On the other hand, *Decorator* can be used to dynamically attach extra responsibilities to a certain object. As such, this pattern provides a flexible way for extending functionality, useful when subclassing is impossible or impractical, and allowing for this addition of behaviour to be performed to individual objects without affecting other objects.

Decorator Octave Implementation

The chosen scenario to demonstrate the *Decorator* pattern is from Cooper's collection. It is a simple concept that consists of three decorators that add different characteristics to a base button resulting in a user interface with four buttons (Figure 3.11).

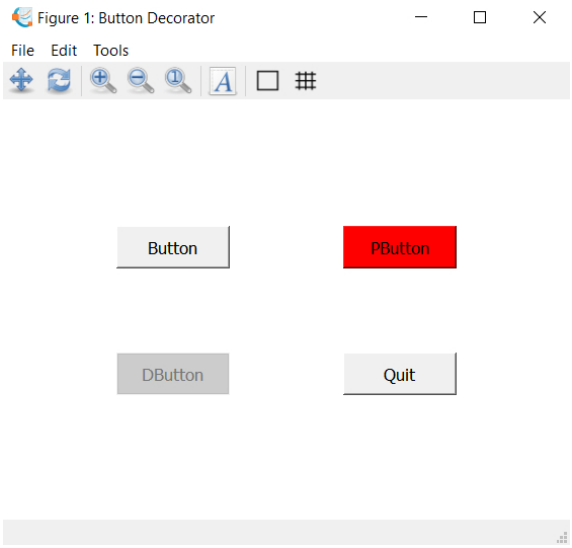

Figure 3.11: *Decorator* implementation in Octave UI

"PButton" (top right placement)	paints the button red
"DButton" (bottom left placement)	disables the button
"Quit" (bottom right placement)	when clicked, closes the UI

Table 3.4: The decorated buttons and the functionality added to them

The target object of this implementation is a simple button, represented by a *BaseButton* class. The *Decorator* class wraps this object and calls the *decorate* method.

```
classdef Decorator < handle
    properties (Access = protected)
        button
    endproperties
    methods
        function self = Decorator (button)
            self.button = button;
            self.decorate();
        endfunction

        function decorate (self)
            error("SUPERCLASS METHOD THAT SHOULD BE OVERRIDEN WAS CALLED");
        endfunction
    endmethods
endclassdef
```

Listing 3.13: Octave *Decorator* class

This class is then extended by the three actual decorators (*PaintDecorator*, *DisableDecorator* and *QuitDecorator*) that override the *decorate* method providing it with the respective desired new functionality.

```
classdef PaintDecorator < Decorator
    methods
        function self = PaintDecorator (button)
            super_args{1} = button;
            self = self@Decorator(super_args{:});
        endfunction

        function decorate (self)
            set(self.button.getButton(), "backgroundcolor", [1 0 0]);
        endfunction
    endmethods
endclassdef
```

Listing 3.14: Octave *PaintDecorator* class

3.7 *Visitor* and the Expression Problem

This section concerns the design pattern: Visitor.

The *Visitor* design pattern is a way of separating an algorithm from an object structure on which it operates. It uses a visitor class that takes the instance reference as input and implements all the needed variants of the new function, which correspond to all target classes. This allows for the addition of new operations to existing object structures without modifying their classes. Furthermore, this pattern can be of use when an object structure contains many classes of objects with differing interfaces, and some operations must be performed on these objects that depend on their concrete classes. However, a notable shortcoming of the *Visitor* pattern is that all visitors need to be updated every time a class gets added to or removed from the element hierarchy.

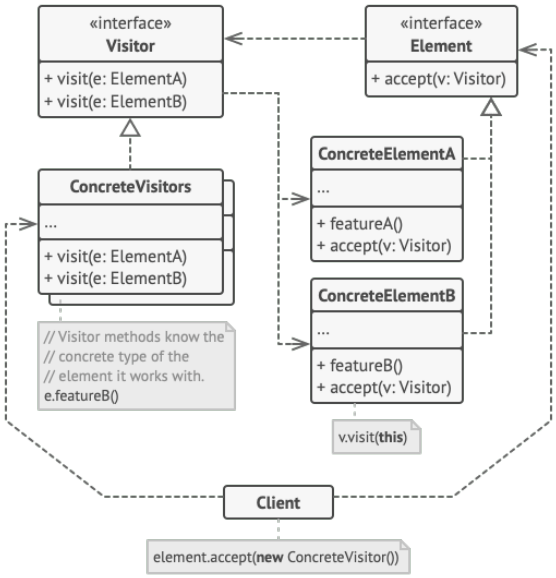

Figure 3.12: Diagram exemplifying a possible structure of implementation of the Visitor pattern [27]

Regarding this pattern, the most noteworthy difference between its Octave and Java implementations relates to Octave's lack of type-checking. Cooper's *Visitor* scenario is used to exemplify this. This scenario consists of a system that displays the number of vacation days of two types of employees, represented by classes *Employee* and *Boss*. Bosses are employees as well, so the *Boss* class extends the former. In this case, the distinction between the two roles is that bosses are allowed to have bonus days off. There are two visitors: class *VacationVisitor* just sums the vacation data for all employees, and class *BossVacationVisitor* also takes into account bosses' bonus days.

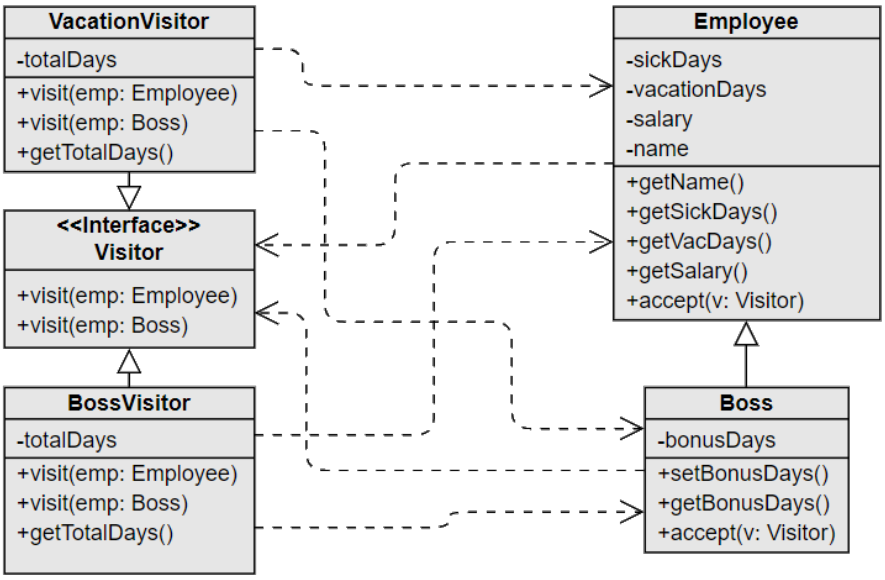

Figure 3.13: Class Diagram for the Java implementation of the "Vacation Visitor" scenario

For better understanding, a user interface is provided consisting of a list of employees and two panels, the top one displays the result obtained by the *VacationVisitor* and the bottom one the value returned by the *BossVacationVisitor*. The user can select an employee from the list and click the "Vacations" button to obtain these values. This UI can be seen in Figure 3.14.

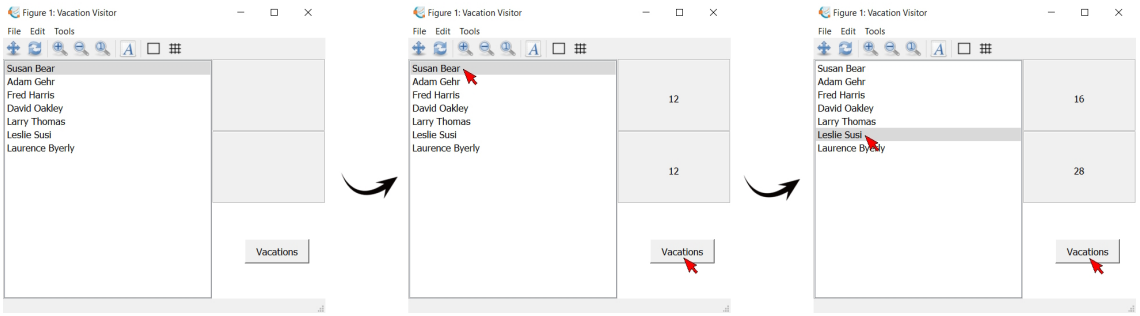

Figure 3.14: *Visitor* implementation in Octave UI

As is customary in the *Visitor* pattern, the target classes possess a method to receive the visit of the visitors and, in this example, the *accept* method plays this role. In the Java implementation, visitors implement the *Visitor* interface (in the same way discussed in Section 3.3) and the *accept* method's argument type is declared as such as seen in Listing 3.15. In Octave, there is no type declaration so a generic argument is enough (Listing 3.16).

```java
public class Employee {
  ...
  public Employee(String name, float salary, int vacdays, int sickdays) {
    ...
  }
  ...
  public void accept(Visitor v) {
    v.visit(this);
  }
}
```

Listing 3.15: Java *Employee* class

```
classdef Employee < handle
    properties (Access = private)
        ...
    endproperties
    methods
        ...
        function accept (self, v)
            v.visit(self);
        endfunction
    endmethods
endclassdef
```

Listing 3.16: Octave *Employee* class

When implementing the visitors in Octave, a significant difference is clear: the various *visit* methods in the *VacationVisitor*, which in Java are distinct depending on the type of the argument (object of class *Employee* or class *Boss*), correspond to only one method in the Octave implementation allowing for code reduction.

```java
public class VacationVisitor implements Visitor {
  protected int _total_days;

  public VacationVisitor() {
    _total_days = 0;
  }

  public void visit(Employee emp) {
    _total_days += emp.getVacDays();
  }

  public void visit(Boss boss) {
    _total_days += boss.getVacDays();
  }

  public int getTotalDays() {
    return _total_days;
  }
```

```
}
```

Listing 3.17: Java *VacationVisitor* class

```
classdef VacationVisitor < handle
    properties (Access = private)
        totalDays
    endproperties
    methods
        function self = VacationVisitor ()
            self.totalDays = 0;
        endfunction

        function visit (self, emp)
            self.totalDays = self.totalDays + emp.getVacDays();
        endfunction

        function retTd = getTotalDays (self)
            retTd = self.totalDays;
        endfunction
    endmethods
endclassdef
```

Listing 3.18: Octave *VacationVisitor* class

However, the same procedure is not possible when it comes to the *BossVacationVisitor* as it adds new functionality when the object is of class *Boss* (adding the bonus days). The Octave implementation of this class maintains only one *visit* method but checks the type of the argument to determine whether the boss-specific calculations should be performed.

```
public class BossVacationVisitor implements Visitor {
  private int _total_days;

  public BossVacationVisitor() {
    _total_days = 0;
  }

  public void visit(Boss boss) {
    _total_days += boss.getVacDays();
    _total_days += boss.getBonusDays();
    ...
  }

  public void visit(Employee emp) {
    _total_days += emp.getVacDays();
    ...
  }

  public int getTotalDays() {
    return _total_days;
  }
```

```
22    ...
23 }
```

Listing 3.19: Java *BossVacationVisitor* class

```
1  classdef BossVacationVisitor < handle
2      properties (Access = private)
3          totalDays
4      endproperties
5      methods
6          function self = BossVacationVisitor ()
7              self.totalDays = 0;
8          endfunction
9
10         function visit (self, emp)
11             self.totalDays = self.totalDays + emp.getVacDays();
12             if(isa(emp, "Boss"))
13                 self.totalDays = self.totalDays + emp.getBonusDays();
14             endif
15             ...
16         endfunction
17
18         function retTd = getTotalDays (self)
19             retTd = self.totalDays;
20         endfunction
21         ...
22     endmethods
23 endclassdef
```

Listing 3.20: Octave *BossVacationVisitor* class

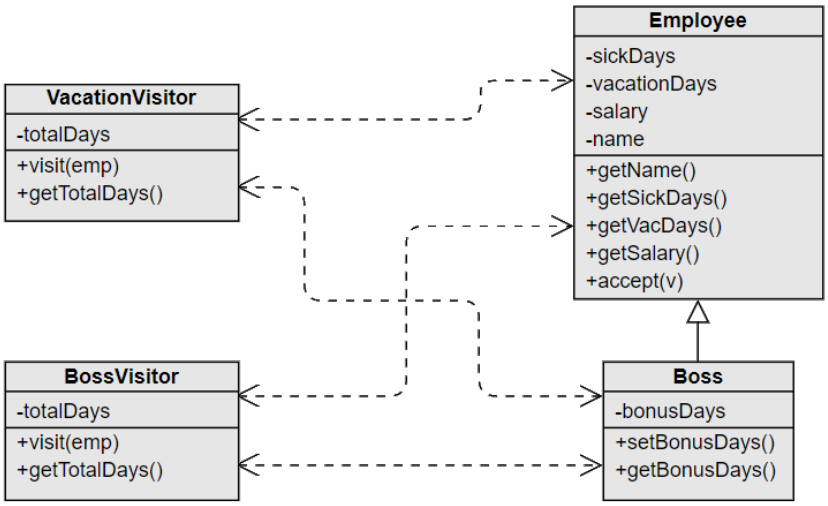

Figure 3.15: Class Diagram for the Octave implementation of the "Vacation Visitor" scenario

3.7.1 The Expression Problem

The "Expression Problem" [28] is a renowned dilemma in the world of programming, and it consists of the need to extend a program in both its set of operations and the set of data types they act on without the need to modify existing code, code repetition or runtime type errors.

Several attempts have been made at solving this problem and the *Visitor* pattern is a recurring component in many of them. This pattern allows the approach of the expression problem in a functional way as it facilitates the separation of data from the functionality operating on it which would be encapsulated in visitor objects. However, while this method makes it simple to add new operations through visitors, adding new data types would still require the update of all visitor classes.

The Octave implementation of the *Visitor* pattern presented above could be proposed as a partial solution to the "Expression Problem" in this language, although a conditional one. The aforementioned condition would be that the operations the developer wants to perform on different objects behave in the same way regardless of the specific data type. Taking the *VacationVisitor* class as an example, any number of new data types could be introduced along with *Employee* and *Boss* (*Manager* or *Intern* for example) without the need to modify the visitor class, as long as their vacations days can be computed with same code present in the *visit* method. Similarly, any new operations could be added to the *VacationVisitor* class just by creating one new function (in non-dynamic programming languages the number of functions to be created would be equal to the number of data types).

Some other variants of the *Visitor* pattern have also been proposed as possibilities to tackle this subject such as "Extensible Visitors" by Krishnamurti et al [14], further refined as "Extensible Visitors with defaults" by Zenger and Odersky [30, 31] or "Generic Visitors" by Palsberg and Jay [21], although all of them still cannot fully solve this problem.

3.8 Noteworthy remarks

This section concerns the design patterns: Memento and Iterator

3.8.1 *Memento* and Nested Classes

The *Memento* design pattern consists of the use of a memento object to store the private internal state of a target object, the "originator", without revealing the details of its implementation. By saving a certain state externally to the originator object, the saved state can be restored later without compromising the object's internal structure or breaking its encapsulation. In Java, this pattern is usually implemented making the memento a nested class, a class defined within another class, which allows the originator to set its state and related methods as private while still being accessible to the memento. Consequently, the memento class is protected and hidden inside the originator class, increasing safety and encapsulation.

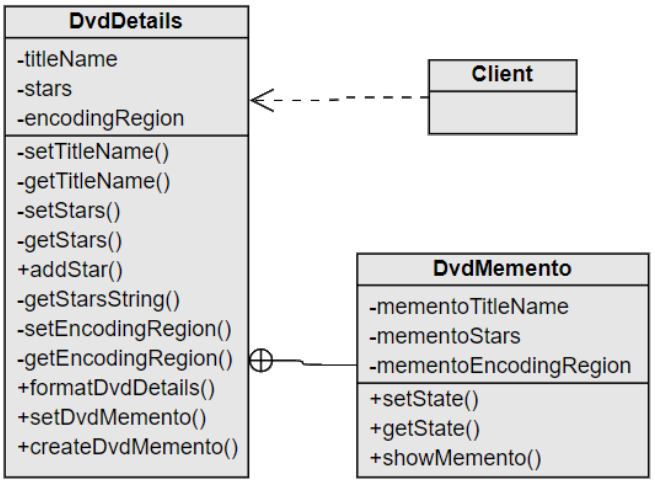

Figure 3.16: Class Diagram for the Java implementation of the "Dvd Memento" scenario

On the other hand, Octave does not support this feature so the *Memento* pattern could only be implemented through simple object composition, which exposes the methods of the memento class and makes it less safe and useful when compared to its Java counterpart.

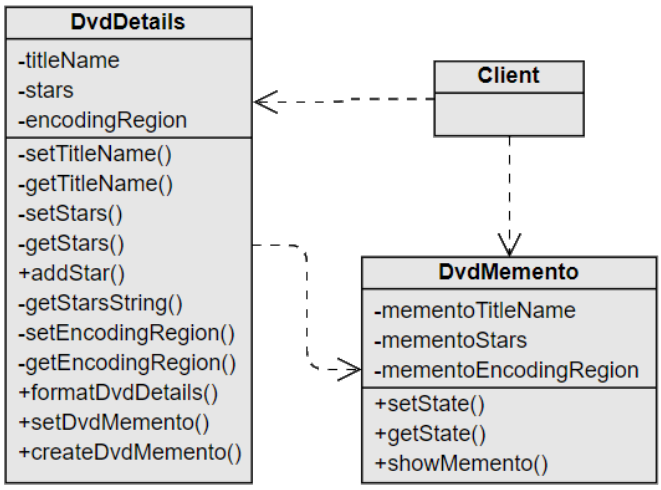

Figure 3.17: Class Diagram for the Octave implementation of the "Dvd Memento" scenario

3.8.2 *Iterator* and the For-loop

Iterator is a behavioural design pattern aimed at optimizing the exploration of the elements of a collection. This pattern extracts the traversal behaviour of a collection into a separate object called an iterator which implements the algorithm itself as well as all the necessary details, such as the current position and how many elements are left till the end. Consequently, it is possible for several iterators to iterate over the same collection in parallel, independently of each other, because each iterator object contains its own iteration state.

The *Iterator* pattern helps maintain the logic clean and understandable by extracting complex traversal algorithms into separate classes. Furthermore, because all iterators implement the same interface, the client code is compatible with any collection type as long as there is a proper iterator which allows this pattern to only provide the clients with simple methods of accessing the collection elements, allowing for the concealing of the complexity of a data structure. This is not only convenient for the client as it simplifies interactions, but it also protects the collection from careless or malicious actions which the client would be able to perform if working with the collection directly.

The Octave implementation of the *Iterator* pattern is straightforward and very similar to the Java equivalent. However, when the situation requires only one simple iterator, the utility of this pattern is put into question as Octave's *for* loop can be applied to structures with the same results and better efficiency. Both iterations shown in Listing 3.21 achieve the same behaviour, thus the *Iterator* pattern would only prove to be useful in Octave in scenario where its desired behaviour is more complex, with capabilities such as two-way iteration.

```
#Iterating through a structure using the Iterator pattern
fiveShakespeareIterator = fiveShakespeareMovies.createIterator();
while (!fiveShakespeareIterator.isDone())
    printf(strvcat(fiveShakespeareIterator.currentItem()));
    fiveShakespeareIterator.next();
endwhile

#Iterating through a structure using the For loop
fiveShakespeareMoviesTitles = fiveShakespeareMovies.getTitles();
for [movieTitle] = fiveShakespeareMoviesTitles
    printf(strvcat(movieTitle));
endfor
```

Listing 3.21: Two ways of transversing a collection in Octave

It should also be noted that modifications to the structure being transversed can be immediately seen during the iteration when using the *Iterator* pattern but not with the *for* loop.

3.8.3 Broader Notes on the Octave Implementations

- **The *objvcat* function**

  A shortcoming of the *classdef* system in Octave is the existence of a bug that prevents the concatenation of new objects to an object array using ordinary notation. This fault is acknowledged in Octave's wiki and an auxiliary function found in a related forum thread is used as a workaround: the *objvcat* function (which can be seen in use in Listing 3.9).

- **Inheritance from graphical objects**

  Inheritance from graphical objects is a capability of Java often used in the pattern implementations taken as reference for this thesis, especially in Cooper's collection. The same behaviour cannot be replicated in Octave so, in cases like this, classes did not inherit from the graphical object but kept it as a property instead.

- **The *toString* function**

  The *toString* function is a common Java function that returns a string representation of an object. Octave offers some methods that allow the developer to inspect an object such as *display* but none with the exact same behaviour. Because this was a necessary function for many of the scenarios, various specific *toString* methods had to be created.

- **Non-mentioned pattern implementations**

  From the 23 Gang-of-Four patterns, 7 of them are not mentioned in this chapter because they were implemented in Octave in a straight-forward manner without relevant distinctions to their Java implementations that could be used for a OO-themed analysis. These patterns are *Bridge*, *Factory Method*, *Flyweight*, *Interpreter*, *Mediator*, *Prototype* and *Template Method*.

4

Analysis on Object-Oriented Programming in Octave

This chapter analyses Octave's OO capabilities taking into account the observations made in Chapter 3 and how they relate to some essential concepts to object-oriented programming and modularity. These concepts are: Abstraction in Section 4.1, Encapsulation in Section 4.2, Polymorphism in Section 4.3 and Modularity in Section 4.4. Section 4.5 presents a comparison table on OOP between Octave and Java and Section 4.6 sums up the conclusions drawn throughout this thesis.

4.1 Abstraction

Abstraction is a fundamental concept in object-oriented programming that consists of hiding the implementation details of an object from the client, revealing only the necessary information, and is used to build complex software systems that are easy to use, maintain and modify. Abstraction can be achieved in several ways, including through the use of abstract classes, interfaces and encapsulation.

In other object-oriented programming languages, interfaces and abstract classes help maintain a high-level abstraction by defining a set of methods and properties that a class must implement, specifying to the client what the class can do, without exposing how it does it. Section 3.3 assesses the related features provided by Octave in the context of the *Abstract Factory* design pattern and presents an attempt to emulate these concepts to little success. Octave does not provide built-in support to interfaces and its implementation of abstract classes is flawed and unusable. The interface-like class presented in Listing 3.1 can be valuable in situations where it is efficient that several classes implement the same method signatures, but it does not provide the abstraction benefits a Java interface can.

4.2 Encapsulation

Encapsulation refers to the practice of grouping related data, functions and classes, providing several benefits such as data abstraction, information hiding, and modularity. In this thesis, this was found to be one of the most difficult principles to preserve in Octave.

In comparison with Java, it is clear this complexity comes from how fundamentally distinct the two languages' goals are. Encapsulation is most beneficial in the structuring and simplification of extensive and complex systems but, unlike Java, Octave's most common use is on smaller programs that can be used to aid the solution of specific mathematical problems. This leads to a couple of relevant core differences. Java allows for the structuring of larger systems with the use of packages, collections of related classes and interfaces. On the other hand, to our knowledge Octave requires all necessary class files to be in the same folder to successfully execute a program.

Another important mechanism to ensure proper encapsulation is the definition of class member access rights. Octave and Java provide three levels of privacy with similar application and nomenclature (private, protected and public as presented in Subsection 2.3.3). However, there is a crucial distinction to be made regarding protected access rights: while in Octave this level of protection makes a class member accessible to its own class and subclasses, in Java the class member is accessible to any class within the same package. Evidently, this makes a big difference allowing for class members to be reachable by certain classes that are not related through inheritance while still not being globally available.

Regarding the design pattern implementations, the most notable loss of encapsulation can be found in the *Memento* implementation, in Subsection 3.8.1. This relates to "nested classes", classes defined within other classes, a practice that allows the concealing of the nested class with the enclosing class while still granting it access to its private members. This mechanism is supported in Java but not in Octave.

4.3 Polymorphism and Duck-Typing

Polymorphism is a fundamental concept in object-oriented programming. It allows programmers to write code that can work with objects of different classes in a generic way, without having to know the specific details of each class, providing a mechanism for abstraction and code reuse, while increasing the flexibility and extensibility of the software. Unfortunately, Octave does not provide built-in support for polymorphism.

A form of polymorphism can be achieved in Octave, like in other dynamic languages, through duck-typing, a practice that permits objects to be handled based on their behaviour rather than their type (previously mentioned in Subsection 2.1.3). As such, while Java's polymorphism enables objects of different classes to be treated as if they were objects of a common superclass, duck-typing completely disregards the objects' type, caring only for whether it is capable of executing the operations required of it. The Octave implementation of the *Visitor* pattern presented in Section 3.7 is a good example of how helpful duck-typing can be in emulating polymorphism and enabling code reuse.

Another important use of polymorphism in Java, that duck-typing in Octave unfortunately cannot replicate, is the existence of polymorphism of *classdef* objects in data structures. It is a common need in OOP to hold collections of related objects of distinct types, but Octave does not allow this behaviour. In Section 3.4, a functioning workaround to this issue is presented but it is noteworthy that a practice so often used in object-oriented programming should have built-in support.

4.4 Modularity and Module Composition

Two essential concepts to this thesis are modularity and module composition. Modularity [22] relies on the concept of "separation of concerns", which in the context of software engineering means to divide a system according to a consistent set of responsibilities that we would like to localize in its own module (a concern). Performing this separation allows developers to work on separate modules of the same system simultaneously , helps the debugging and maintenance of code, as it is easier to test and develop smaller components centered around the same concern and can facilitate software reuse as a focused module is easier to integrate into new software.

Complementing modularity, module composition is a property which relates to the relationships and interaction between the aforementioned modules. Through these two properties and the language mechanisms that provide support for them, it possible to get a clearer understanding of a given system. There are already established works on evaluating these properties in certain systems, like the works by Sethi et al [26] or by Cai et al [4], and the support certain languages offer to them, as can be seen in some the works presented in Section 5.2.

4.4.1 Modularity in Octave

Octave does not provide much support for the implementation of modularity with its features. Encapsulation and Polymorphism are closely tied concepts to Modularity so, looking back at Section 4.2 and Section 4.3, it is possible to point out two of Octave's OO features that can aid modularity:

- **Duck-typing** allows the programmer to write generic code that can work with objects of different types, thus reducing coupling between modules that do not have to know what kind of object they are interacting with;
- The definition of **Access Rights** for class members also helps to isolate modules from each other by limiting the information made available by objects.

These two features only allow the separation of modules at a lower-level and that pales in comparison with the most popular OOP languages which mostly provide mechanisms to implement modularity on a system-wide scope.

4.4.2 Modularity Mechanisms

This section concerns modularity mechanisms that relate to design patterns, either through their purpose or implementation, and their feasibility in Octave.

Mixins

A concept first introduced by Bracha and Cook in 1990 [3], a *mixin* is a class that contains methods for use by other classes without having to rely on the inheritance mechanism. As such, *mixins* can be used to avoid the inheritance ambiguity that multiple inheritance can cause or to work around lack of support for multiple inheritance in a language.

Mixin composition differs from inheritance in that the class who receives functionality can still inherit the features of the *mixin* class but avoiding some effects that come from a child class "being a kind of" the parent class. *Mixins* are usually referred to as being "included" instead of "inherited".

It can be said that the decorator design pattern emulates the *mixin* concept as they both aim to encourage code reuse and share logic between components while avoiding the inheritance mechanism's shortcomings.

Octave does support multiple inheritance which would allow for the inheritance of the class that carries the additional functionality. In the most simple examples, where complications like the "diamond problem" [16] can be easily avoided, this would probably be the most efficient solution. However, in the development of more complex systems, the programmer may prefer to avoid the potential complications of the inheritance mechanism. The *Decorator* pattern provides an alternative by adding functionality through object composition (wrapping). On the other hand, the *mixin* class is implemented separately and then included but, while Octave's documentation presents a way to include

the content of an external file, all attempts to replicate this mechanism during this thesis were unsuccessful leading to an error where the interpreter does not recognize the given syntax. Therefore, according to the aforementioned definition of *mixins*, *Decorator* is the default choice for this purpose.

Family Polymorphism

Family polymorphism [10] is a programming language feature that allows us to express and manage multi-object relations while ensuring both the flexibility of using any of an unbounded number of families and the safety guarantee that families will not be mixed. By providing an interface for creating families of related objects, it allows for the use of several classes and several method implementations but still ensures that the chosen method implementation is always appropriate for the actual object. In doing so, it can be said that the *family polymorphism* mechanism shares the exact same purpose as the *Abstract Factory* design pattern and. Octave does not have direct support for *family polymorphism* however, similarly to what is observed in Section 3.3, its utility in this language can be put into question.

Double Dispatch

The *Visitor* pattern emulates *double dispatch* [13], a feature based on dynamic binding and the overloading of methods, which consists of the selection of different functions dynamically based on the runtime types of two objects, the object where the method is called and the object passed as argument.

Looking at Section 3.7 and the differences between the Java and Octave implementations of the *Visitor* pattern, it can be said that double dispatch is not a relevant concept in Octave due to the lack of type-checking.

4.5 OO Feature Comparison with Java

This section presents a table presenting some object-oriented features mentioned in this thesis and whether Octave and Java have direct support for it.

OO Feature	Java	Octave
Interfaces	Yes	No
Abstract Classes	Yes	No
Method Overloading	Yes	Yes
Static Methods	Yes	Yes
Static Variables	Yes	No
Class Member Access Rights	Yes	Yes
Nested Classes	Yes	No
Inheritance	Yes	Yes
Multiple Inheritance	No	Yes
Mixins	Yes	No
Polymorphism	Yes	No
Data Structure Polymorphism	Yes	No
Duck-typing	No	Yes
Class/File Packaging	Yes	No
Exception/Error Handling	Yes	Yes

Table 4.1: OO feature comparison between Java and Octave

4.6 Summing Up

Throughout Chapter 3, the design pattern implementations point out various missing features in the OO aspect of Octave. While Octave does provide some support for object-oriented programming, clearly that is not the primary focus of the language, and it is unsuited for complex OOP applications.

Java is present throughout this thesis, as the standard that Octave is being compared to regarding object-oriented programming and, as one of the most widely known programming languages for OOP, it serves as an appropriate benchmark for the analysis of Octave's object-oriented features, as can be seen in Section 4.5.

The introduction of *classdef* classes was a great improvement over the "old style" classes, having a much closer behaviour to the classes found in full-fledged OOP languages, but Octave still has too limited support for classes and objects. As a programming language, Octave is a good study case for the object-oriented paradigm and presents some interesting possibilities, especially for its behaviour as a dynamic language and its use of "duck-typing" but, as seen in this chapter from Section 4.1 to Section 4.4, its implementation of essential OOP concepts is not extensive enough and it would be challenging to use Octave to develop an elegant and efficient system that can comply with the "separation of concerns" principle.

5

Related Work

In this section, relevant studies related to the topic addressed in this thesis are presented, either by sharing the same concept of using design pattern implementation to evaluate the capabilities of a programming language, as seen in Section 5.1, or providing important and helpful information about design patterns, object-oriented programming or modularity and composition like the works mentioned in Section 5.2.

5.1 Design Patterns, Object-Oriented Programming Languages and Modularity

Baumgartner et al [1] focus on the connection between the efficiency of design patterns and the available mechanisms in the programming languages used to implement them. Starting from the observation that when a language lacks the necessary supporting constructs to implement a design pattern, said implementation may become distorted or overly complicated, and this paper attempts to find idiomatic ways of working around constraints of the implementation language. In this work, the features they indicate as particularly useful for object-oriented languages are the language constructs interface, object, and class and the mechanisms interface conformance, code reuse, lexical scoping, and multimethod dispatch. As well as the separation of the subtyping and code-reuse aspects of inheritance.

Lopez-Herrejon et al [15] evaluate five technologies considering their support for modularization and composition of features, namely AspectJ, Hyper/J, Jiazzi, Scala and AHEAD. Taking the concept of product-line design and a variant of the expression problem as base, they propose a set of technology-independent properties that feature modules should exhibit and suggest an abstract model of feature composition that is technology-independent and that relates compositional reasoning with algebraic reasoning.

Pedersen [23] aims to determine the impact of implementation language on code quality for implementations based on the GoF design patterns by comparing the implementations of the Composite, Prototype, Adapter and Decorator patterns in Python, JavaScript, C#, Go and Smalltalk. Evaluating these cases with a wide set of software quality metrics, Petersen found differences in the quality of the implementation amongst languages and similarities between Python and JavaScript as well as between C#, Go and Smalltalk, although for the latter set of languages only the size metrics showed resemblance. In this paper, it is also determined that there are mechanisms capable of increasing the quality of a design pattern implementation, namely flexible typing and inheritance schemes, low notation and/or definition overhead, protected attribute visibility and the possession of a toolkit capable of solving the same issues that the design patterns address.

5.2 Design Pattern Implementation in Other Languages

Hannemann et al [12] (HK) present AspectJ implementations of the GoF design patterns, achieving modularity improvements, in comparison with their Java counterparts, in 17 of 23 cases through the chosen criteria of code locality, reusability, composability, and (un)pluggability. The level of improvement varies and is directly correlated to the presence of crosscutting structure in the patterns. These improvements are demonstrated through the observations that the pattern implementations are more localized and therefore more comprehensible and, in several cases reusable. AspectJ implementations of the patterns are also sometimes composable. In addition, they provide an anchor for improved documentation of the code.

Using a different aspect-oriented language, Rajan [24] describes the implementations in the Eos language of all 23 GoF patterns to analyse the effect of new programming language constructs on these implementations, taking as reference the scenarios provided by Hannemann and Kiczales in the aforementioned work and presents a comparative analysis of results using the same modularity properties. The Eos implementation showed improvement in the case of 7 out of 23 design patterns compared to the AspectJ implementation while performing at the same level for the remaining 16 patterns. Rajan claims the improvements concerning his implementations are mainly manifested in being able to realize the intent of the design patterns more clearly.

Building on the two previous studies, the paper by Monteiro et al [17] presents the implementation of the GoF patterns in the Object Teams programming language (OT/J), an aspect-oriented language backwards compatible with Java. They proceed to compare it with Java and AspectJ collections of implementations, namely the ones provided by Hannemann et al for both languages and a Java collection written by James Cooper [5]. The results show that OT/J achieves notably better results in regard to flexible module extensibility, composition at the instance level and enclosing multiple pattern participants into a larger, cohesive module.

Schmager et al [25] also aim to evaluate language through the implementation of design patterns, in this case the Go language when it was still in its early stages (only one year after release). While also using the framework HotDraw, this study highlights Go-specific features such as embedding and interface inference and how they can be used to successfully implement all the 23 GoF patterns, despite the differences in object model between Go and most object-oriented languages.

Bibliography

[1] G. Baumgartner, K. Laufer, and V. J. Rego. "On the interaction of object-oriented design patterns and programming languages". In: (1996) (cit. on p. 55).

[2] K. Beck. "Using pattern languages for object-oriented programs". In: *http://c2. com/doc/oopsla87. html* (1987) (cit. on p. 17).

[3] G. Bracha and W. Cook. "Mixin-based inheritance". In: *ACM Sigplan Notices* 25.10 (1990), pp. 303–311 (cit. on p. 52).

[4] Y. Cai and S. Huynh. "An evolution model for software modularity assessment". In: *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*. IEEE. 2007, pp. 3–3 (cit. on p. 51).

[5] J. W. Cooper. "Java design patterns: a tutorial". In: (2000) (cit. on pp. 3, 19, 56).

[6] *Duck typing*. Last Accessed: 2023-03-27. Feb. 2023. URL: https://en.wikipedia.org/wiki/Duck_typing (cit. on p. 7).

[7] J. W. Eaton, D. Bateman, and S. Hauberg. "Gnu octave". In: *GNU Octave* (2013) (cit. on pp. 1, 7, 10).

[8] J. W. Eaton, D. Bateman, S. Hauberg, et al. *Gnu octave*. Network thoery London, 1997 (cit. on p. 6).

[9] B. Eckel. "Thinking in Patterns Revision 0.9". In: *http://www. pythoncriticalmass. com/downloads/TIPatterns-0.9. zip* (2003) (cit. on p. 19).

[10] E. Ernst. "Family polymorphism". In: *European Conference on Object-Oriented Programming*. Springer. 2001, pp. 303–326 (cit. on p. 53).

[11] E. Gamma et al. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995 (cit. on pp. 1, 17–19).

[12] J. Hannemann and G. Kiczales. "Design pattern implementation in Java and AspectJ". In: *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2002, pp. 161–173 (cit. on pp. 3, 56).

[13] D. H. Ingalls. "A simple technique for handling multiple polymorphism". In: *Conference proceedings on Object-oriented programming systems, languages and applications*. 1986, pp. 347–349 (cit. on p. 53).

[14] S. Krishnamurthi, M. Felleisen, and D. P. Friedman. "Synthesizing object-oriented and functional design to promote re-use". In: *European Conference on Object-Oriented Programming*. Springer. 1998, pp. 91–113 (cit. on p. 44).

[15] R. E. Lopez-Herrejon, D. Batory, and W. Cook. "Evaluating support for features in advanced modularization technologies". In: *European Conference on Object-Oriented Programming*. Springer. 2005, pp. 169–194 (cit. on p. 55).

[16] B. Meyer. *Object-oriented software construction*. Vol. 2. Prentice hall Englewood Cliffs, 1997 (cit. on pp. 1, 52).

[17] M. P. Monteiro and J. Gomes. "Implementing design patterns in Object Teams". In: *Software: Practice and Experience* 43.12 (2013), pp. 1519–1551 (cit. on pp. 3, 56).

[18] M. Mützel. *Octave Discourse Forum - Static properties in classdef objects*. Last Accessed: 2022-11-14. Oct. 2022. URL: https://octave.discourse.group/t/static-properties-in-classdef-objects/3442 (cit. on p. 33).

[19] *Octave Online*. Last Accessed: 2022-07-14. URL: https://octave-online.net/ (cit. on p. 16).

[20] *Octave Wiki - Classdef*. Last Accessed: 2023-02-14. URL: https://wiki.octave.org/Classdef (cit. on pp. 10, 21).

[21] J. Palsberg and C. B. Jay. "The essence of the visitor pattern". In: *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac'98)(Cat. No. 98CB 36241)*. IEEE. 1998, pp. 9–15 (cit. on p. 44).

[22] D. L. Parnas. "On the criteria to be used in decomposing systems into modules". In: *Pioneers and their contributions to software engineering*. Springer, 1972, pp. 479–498 (cit. on pp. 2, 51).

[23] K. B. Pedersen. "How Implementation Language Affects Design Patterns: A Comparison of Gang of Four Design Pattern Implementations in Different Languages". MA thesis. 2019 (cit. on p. 56).

[24] H. Rajan. "Design pattern implementations in Eos". In: *Proceedings of the 14th Conference on Pattern Languages of Programs*. 2007, pp. 1–11 (cit. on p. 56).

[25] F. Schmager, N. Cameron, and J. Noble. "GoHotDraw: Evaluating the Go programming language with design patterns". In: *Evaluation and usability of programming languages and tools*. 2010, pp. 1–6 (cit. on p. 57).

[26] K. Sethi et al. "From retrospect to prospect: Assessing modularity and stability from software architecture". In: *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*. IEEE. 2009, pp. 269–272 (cit. on p. 51).

[27] A. Shvets. "Dive Into Design Patterns". In: *Refactoring. Guru* (2018) (cit. on pp. 22, 29, 39).

[28] P. Wadler et al. "The expression problem". In: *Posted on the Java Genericity mailing list* (1998) (cit. on p. 44).

[29] A. Weber. *Octave Wiki - UIcontrols*. Last Accessed: 2022-10-14. Aug. 2017. URL: https://wiki.octave.org/Uicontrols (cit. on p. 15).

[30] M. Zenger and M. Odersky. "Extensible algebraic datatypes with defaults". In: *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*. 2001, pp. 241–252 (cit. on p. 44).

[31] M. Zenger and M. Odersky. "Implementing extensible compilers". In: *ECOOP workshop on multiparadigm programming with object-oriented languages*. Citeseer. 2001 (cit. on p. 44).