Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

MINH TUAN NGUYEN

MASTER THESIS

A QUALITATIVE COMPARATIVE EVALUATION OF GRACE AND JAVA FOR IMPLEMENTING DESIGN PATTERNS

Submitted on 26 September 2016

Supervisor: Prof. Dr. Dirk Riehle, M.B.A.

Professur für Open-Source-Software

Department Informatik, Technische Fakultät

Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 26 September 2016

License

Erlangen, 26 September 2016

Abstract

Design patterns are software design constructs defined and implemented using a programming language. What features of a programming language make the implementation of a wide variety of object-oriented design patterns easy? This thesis analyses Grace for its effectiveness in supporting developers in implementing design patterns. The method is comparing the implementations of standalone Gang of Four design patterns in Grace and Java. After that we extend our study by implementing Hotdraw in Grace, producing GraceHotdraw. Based on the design pattern implementations we present the Grace specific language features and analyze them against the language evaluation model developed by Michael Kölling.

Contents

1 Introduction

Grace is a new object-oriented programming language which is developed by a community of researchers in computer science from universities around the world. The purpose of Grace is to let beginners learn programming in a simple way. Current languages often used for teaching such as Java or Python are widely used in the industry but these application languages are hard to learn for novices. This thesis analyzes the Grace language features in general, and points out which features make the implementations of design pattern easy by comparing the Grace design pattern implementations against other application languages, in particular Java.

2 Research

2.1 Introduction

Grace is a new educational programming language developed by a community of computer scientists from universities around the world (A. P. Black, Bruce, Homer & Noble, 2012). Grace has potential to become the main teaching programming language for novices in universities.

Design Patterns inform programmers about good program design (Wolfgang, 1994). They provide standardized and efficient solutions for reoccurring problems and have been implemented in many programming languages. Design Pattern implementations are different for every programming language.

The contributions of this thesis are:

- Evaluation of Grace as a educational language based on experience in design pattern implementations.
- The qualitative analysis of language features which make design pattern implementations easy.

This chapter is structured as follows: In section 2.2 we review related work regarding educational programming languages and Grace's main language features. In section 2.3 we show our research questions. In section 2.4 we present our research approach. We analyze in details three of our 23 design pattern implementations and describe GraceHotdraw. In section 2.5 we show our results by comparing our study against Kölling's language evaluation model. In section 2.6 and 2.7 we discuss the limitations of our results, present some conclusions and look at future works.

2.2 Related Work

2.2.1 Educational programming languages

Back in the 1980s the choice for programming languages not only for teaching but also for general purpose was limited. The software engineering and computer science community chose Pascal as the standard language not only for teaching but also for industry software or research activities (Noble, Homer, Bruce & Black, 2013). As a result transferring knowledge and skills between institutions was easy, textbook choices were more consistent and employers could rely on the shared basis of graduate's programming skills.

As the years passed by, the choice for a standard programming languages was not obvious anymore. Many educational institutions adapted C and Java for teaching programming languages. They are application languages which are designed for professional purposes but not for novices however. There are some research works which evaluate theses languages for suitability and sustainability as introduction language (Hadjerrouit, 1998) (Farooq, Khan, Ahmad, Islam & Abid, 2014).

As the need for an educational programming languages became bigger, the computer science community identified the requirements for a good introductory programming language (Gupta, 2004) and suggested a lot of methods and developed other teaching programming languages. Some researchers recommended existing programming languages for teaching purpose, such as Python, by analyzing the experiences of beginners in learning computer programming in universities or comparing the language against other widely used teaching languages, such as Java (Begosso, Begosso, Gonçalves & Gonçalves, 2012) (Lo, Lin & Wu, 2015). Python lacks on static types however, and is therefore not the best choice for teaching programming languages.

Uehara from Toyo University developed a new educational programming language which is called Cafe. The language is based on Processing, a programming language that is developed by Reas and Fry at MIT within the context of visual arts (Uehara, 2009). The language combines the main language features in the currently teaching programming languages C and Java. Papaspyrou proposed CAL, a algorithmic language which is similar to C with some extensions dedicated for educational purpose (Papaspyrou & Zachos, 2013). These languages are not object-oriented however.

Other researchers presented different methods to make learning programming languages more simple. One popular method was to implement programming environments or systems for the current widely used application languages. These environments or systems should make learning programming easier for beginners. Pawelczak implemented Virtual-C, a IDE for teaching C in undergraduate programming courses (Pawelczak & Baumann, 2014). Lei presented concepts to develop programming environment aiming on non-technicals (Lei, 2003). Serrano-Laguna designed a system which helps novices learn programming by creating video games (Serrano-Laguna, Torrente, Iglesias & Fernández-Manjón, 2015).

All of these methods and programming languages still lack of acceptance from the computer science community. As the need for a new unified educational programming language became bigger, during ECOOP 2010, a group of language researchers and educators decided to collaborate and designed a new educational language (A. Black, Bruce & Noble, 2010). They developed Grace with the purpose to teach novices programming and software engineering (Homer & Noble, 2014) (Homer & Noble, 2013) (A. P. Black, Bruce, Homer & Noble, 2013).

2.2.2 Grace and Design Patterns

Grace is an object-oriented language designed for education. The main purpose is to make learning programming easy for novices. Design Patterns are sets of solution for commonly occurring problems in object-oriented software engineering (Cooper, 2002) (Bishop, 2007). During the development of Grace the developers integrate some design patterns to the language. In this section we introduce the main Grace language features and highlight the integrated design patterns in the language (A. P. Black, Bruce & Noble, 2013) (A. P. Black, Bruce & Noble, 2016a) (A. P. Black, Bruce & Noble, 2016b).

Object and Classes Grace classes and objects contain field declarations and methods. The following code defines an employee.

```
class aEmployee (_name) {
    var name is readable := _name
    method printName {
        print "{name}"
    }
}
```

This is the way to define an object of aEmployee:

```
var peter := aEmployee ("Peter")
```

Beside we can define an object constructor without any classes:

```
//Object Constructor
object {
    var name is readable := "Peter"
    method printName {
            print "{name}"
        }
    }
```

Singleton is a language feature in Grace. Now we can declare a variable to make a global access point to the object.

```
    def singleton = object {...}
```

We will explain the Grace build-in singleton more in details in the later sections.

Grace does not have any built-in null value. Variables which are uninitialized can not be used as a null reference. The values of those variables are uninitialized.

Types Grace supports both static and dynamic types. One does not have to specify types to declare local variables, fields and methods. Types are structural and separated from classes. The concept of defining a type can be compared to interfaces in Java. Type specifies which attributes and methods types are required for an object.

```
    type Employee = {
        name → String
        printName () → Done
    }
```

The arrow symbol separates a method name or attribute from its (return) type. Done indicates that the method does not return any value. The following example demonstrates how to define an Employee object:

```
    def peter:Employee = aEmployee (″Peter″)
```

The arrow symbol also enables the use of iterator and command pattern in Grace:

```
    for (commandList) do { command →
        command.execute
    }
```

All the information is stored in the single object commandList which can be accessed. CommandList contains a list of commands. The integrated iterator pattern in Grace helps us to access all the objects in commandList with a for loop without knowing its exact structure.

Inheritance Reuse in Grace is supported by Inheritance. The main difference to Java is the possibilities of multiple inheritance. Inheritance is possible for both objects and classes.

We can extend our previous example as follows:

```
class aContractor (_name) {
    inherit aEmployee (_name)
    var contractTime
}
```

The clause inherit is equivalent to extends in Java. After calling inherit Grace will call the super class, which is aEmployee in our example.

Information hiding Grace is designed in a way so that making mistakes is more difficult. Thus the designer of Grace decided to set the default visibility for methods to public and the default visibility for attributes to confidential. Both can be changed by adding "is confidential/public". Getter and setter for attributes is possible by adding "is readable and is writable" after the attributes.

2.3 Research Question

This research will address the following questions:

General question: Is an educational programming language better suited for implementing design patterns than an application programming language ?

Specific question: Is Grace, as an example of a educational programming language, better suited than Java, as an example of an application programming language ?

The main question is: How difficult is the implementations of design pattern in Grace ?

2.4 Research Approach

2.4.1 Grace Standalone Design Pattern Applications

We successfully implemented all 23 Gang of Four patterns in Grace (Gamma, Helm, Johnson & Vlissides, 1995). We describe the implementations and attach the code in the appendices section. This section shows detailed standalone implementations of singleton from creational patterns group, composite from structural patterns group and visitor pattern from behavior patterns group and compares them to the Java implementations (Metsker & Wake, 2006) (Hannemann &

Kiczales, 2002). We decided to show them because they cover all Grace specific language features. Singleton shows encapsulation and information hiding rules. Composite illustrates type concepts and object construction. Visitor shows inheritance structure.

2.4.1.1 Singleton Pattern

The singleton pattern belongs to the group of creational patterns and makes sure that a class only has one instance. The pattern provides a global access point to that object. In our example we create 2 packages SingleObject.grace and SingletonPatternDemo.grace. The first includes the single object and the interface to enable global access. The second tests the functionalites of the pattern.

```
//Singleton Package
var instance := false
class SingleObject is confidential{
    var testAttribute is readable, writable := "original attribute"
    method asString is public {
        print "I am a {testAttribute}"
    }
}

class Interface is public {
    method getInstance is public {
        if (instance == false) then {
            instance := SingleObject
        } else {
            //Do nothing
        }
        return instance
    }
}

//Demo package
import "SingleObject" as s

//How to access the singleton object
var instance := s.Interface.getInstance
instance.asString

var instance2 := s.Interface.getInstance
instance2.asString
```

The method getInstance is the global access point to the singleton from the class SingleObject. The method getInstance belongs to the class Interface. The global variable "instance" gives the information whether the singleton-object for the class SingleObject has been created or not and ensures that we can create only one object from SingleObject. In our example instance.asString and instance2.asString both print "original attribute".

Our implementation does not prevent creation of many Interface objects, but method getInstance forces all objects to return the same singleton object. Moreover nothing from outside the SingleObject package can modify the global variable instance, since it is confidential. Java implements singleton with static fields. Since Grace does not have any static class members, we make use of Grace's package mechanism to enable the pattern.

Another possibility for singleton is making use of an object constructor and immutable variable.

```
def singleton = object {
    var testAttribute is readable, writable := "original attribute"
    method asString is override {
        "I am a {testAttribute}"
    }
}

//How to access the singleton object
var instance := singleton
```

In this implementation we create a object which we access by calling the immutable variable "singleton".

2.4.1.2 Composite Pattern

The composite pattern belongs to the group of structural patterns and creates hierachical object models. The purpose is to treat a group of objects in the same way as a single instance of an object. In our example we implement the hierarchical system of a company composite. Employee acts as the Component class. PermanentEmployee is a Employee, acts as Composite class and can have many other Employees as child elements. Contractor is a Employee, acts as Leaf class and cannot have any child element.

```
class Employee (_name, _department) {
    var name is readable := _name
    var role is readable := _department
    //...
}

class PermanentEmployee (_name, _department) {
    inherit Employee (_name, _department)
    def subordinates = list.with []

    method addSubordinate (_subordinate) {
        subordinates.add(_subordinate)
    }
    //...
}

//Contractor cannot have any subordinates
class Contractor (_name, _department) {
    inherit Employee (_name, _department)
    //...
}
```

Employee acts as the parent class. The attribute subordinates in PermanentEmployee contains a list of Employees. This list can include other PermanentEmployee objects with their child elements creating a tree structure with Contractor objects as its leaf elements. The main differences to the Java implementation:

- The concept of abstract class does not exist in Grace. Our implementation cannot prevent creation of Employee objects. Reuse in our example is possible with inheritance which is similar to Java.

- Grace classes are basically factory methods. A class is a method of which body is treated as an object constructor. There is no explicit constructor. We pass the input arguments through the head of the class after the class name. Everytime when the class is invoked to create a new object, the object constructor will be called. In Java we must explicitly define the class constructor which lengthens the code.

- In PermanentEmployee,

  ```
  inherit Employee (_name, _department)
  ```

  will pass the input parameters directly to the parent class Employee. In Java the contructor will do this by calling

  ```
  super(_name,_department)
  ```

- Grace allows using of dynamic types and thus we do not have to worry about typing while coding the child elements. In our implementation we cannot prevent a PermanentEmployee object from having other objects than Employee objects as its children.

2.4.1.3 Visitor Pattern

The visitor pattern belongs to the behavioural patterns group and separates a set of structured data from the functionality that may be performed upon it. This pattern promotes loose coupling and enables additional operations to be added without modifying the data classes. In our visitor pattern implementation we extend our previous composite pattern implementation. EmployeeVisitor acts as the VisitorBase. PaySalary and PrintAllEmployees act as ConcreteVisitor. The composite pattern represents the object structure.

```
class EmployeeVisitor {
    method visitPermanentEmployee (employee) {}
    method visitContractor (employee) {}
}

//Visitor 1: pay salary to all employees
class PaySalary {
    inherit EmployeeVisitor
    method visitPermanentEmployee (employee) is override {
        print "{employee.name} has got the salary"
    }
    method visitContractor (employee) is override {
        print "{employee.name} has got the contracting rates"
    }
}

//Visitor 2: print all employees
class PrintAllEmployees {
    inherit EmployeeVisitor
    method visitPermanentEmployee (employee) is override {
        print "{employee}"
    }
    method visitContractor (employee) is override {
        print "Contractor {employee}"
    }
}
```

We extend the composite structure by an accept method, which serves as the loose coupling between the composite objects and the visitors.

```
//Extension of class PermanentEmployee
method accept (visitor) {
        visitor.visitPermanentEmployee (self)
        var counter := subordinates.size − 1
        for (0 .. counter) do { n:Number →
            subordinates.get(n).accept(visitor)
        }
 }

//Extension of class Contractor
method accept (visitor) {
        visitor.visitContractor (self)
 }
```

In general the Grace implementation of visitor pattern is very similar to Java. The main difference of our implementation from the Java implmementation is the visitor base class EmployeeVisitor which represents the interface between the object structure and the visitor modules. Grace allows using of both dynamic and static typing. As a result the class EmployeeVisitor is redundant. Removing that class does not have any influence on the functionalities of our application.

2.4.2 GraceHotdraw

Hotdraw is a framework for a graphic editor which was developed by Kent Beck and Ward Cunningham (Cunningham, 1994) (Johnson, 1992) (Brant, 1995) (Froehlich, Hoover, Liu & Sorenson, 1997).

Erich Gamma and Thomas Eggenschwiler developed JHotdraw, the Java implementation of the framework, for teaching purpose (Riehle, 2000) (Gamma & Eggenschwiler, 2007) (Kaiser, 2001). Thus the framework is mature and well documented. Moreover JHotdraw is still maintained by a large community. Since JHotdraw is very large we concentrate on the main functionalities which have high design pattern density. Users can:

- select tools. Available tools are Selection Tool and Creation Tool.
- create rectangle. To simplify the framework we implement no other figures than a rectangle.
- select figure.
- resize figure.
- move figure.

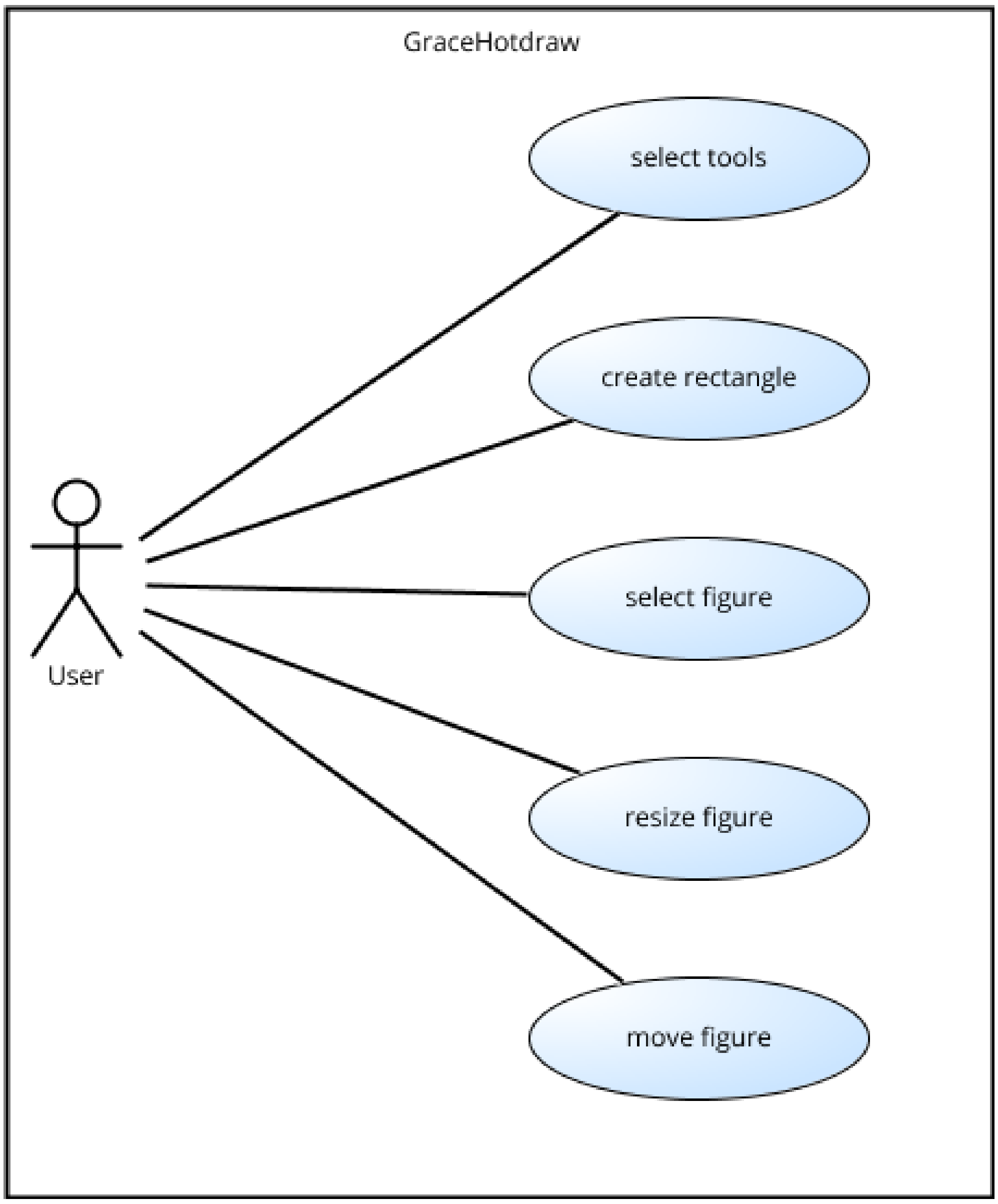

Figure 2.1: Use Case Diagram

2.4.2.1 Architecture Design

The following section aims at introducing the reader to the design of GraceHotdraw according to the Model-Controller-View pattern, which allows controlling of multiple appearances of data on the display (Hatanaka & Hughes, 1999). The controller performs actions based on user inputs. The Model receives instructions from Controller to manage data. The View is responsible for data representation. We present the three components, the interaction between them and how we apply design patterns for the framework.

Model is the central part of the framework.

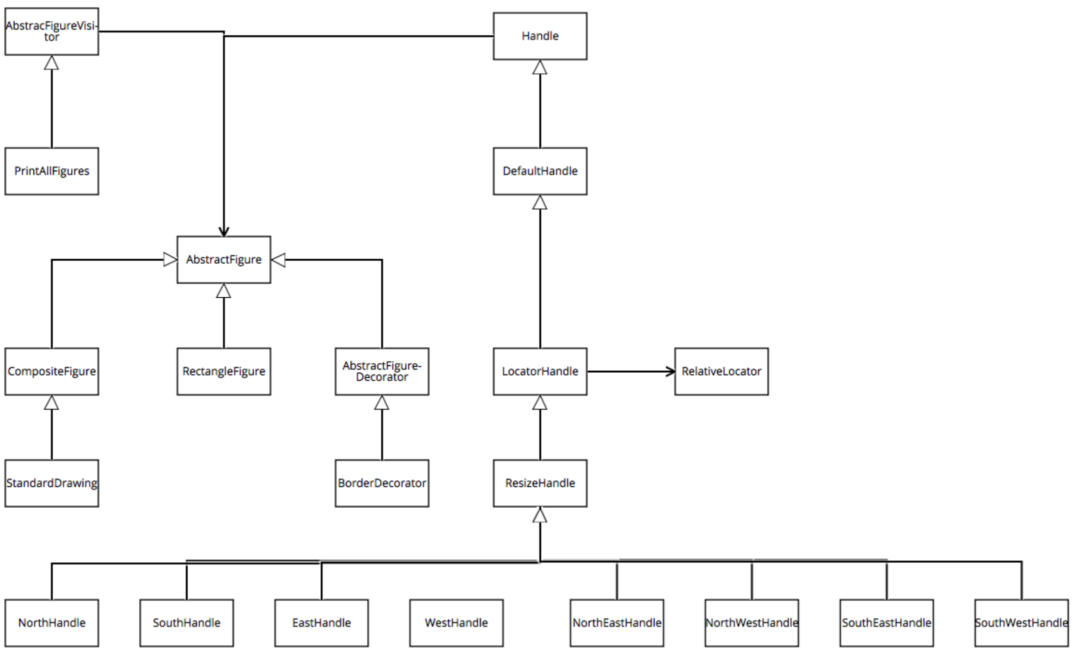

Figure 2.2: Model

AbstractFigure implements common functionalities for all figure types such as event listeners handling or manipulation of figure data. In our implementation we only implement rectangle figures, but we could easily extend the framework to all other figure types by inheritance from AbstractFigure.

Event listening acts as the observer pattern, receives figure events and updates View. CompositeFigure contains a list of other figures and we can treat an CompositeFigure object as a single figure. StandardDrawing acts as a collection of figures and is displayed by View.

We implement the composite pattern to ensure consistency of the figure objects. AbstractFigure is the Component. CompositeFigure is Composite and RectangleFigure is Leaf.

We develop AbstractFigureDecorator to add additional features to figure objects dynamically. AbstractFigureDecorator inherits from AbstractFigure. We are able to add new features to the figure objects by implementing child classes to AbstractFigureDecorator. In our implementation we implement the BorderDecorator which add an extra border to a figure object that inherits from AbstractFigure. This is part of the decorator pattern whose goal is to add additional functionalities only to objects but not to classes and hence reduces the complexity of the class itself.

The framework uses Handle objects to resize figure. There are eight handles in total. The method createHandle creates four handles on each figure corner and four on each side. We use factory pattern to create the handles while innitializing a new figure object.

```
method createHandle (_ownerFigure) is override {
    var handleList := list.with[]
    handleList.add(NorthHandle (ownerFigure, locator.createNorthLocator)
        )
    handleList.add(SouthHandle (ownerFigure, locator.createSouthLocator)
        )
    handleList.add(EastHandle (ownerFigure, locator.createEastLocator))
    handleList.add(WestHandle (ownerFigure, locator.createWestLocator))
    handleList.add(NorthEastHandle (ownerFigure, locator.
        createNorthEastLocator))
    handleList.add(NorthWestHandle (ownerFigure, locator.
        createNorthWestLocator))
    handleList.add(SouthEastHandle (ownerFigure, locator.
        createSouthEastLocator))
    handleList.add(SouthWestHandle (ownerFigure, locator.
        createSouthWestLocator))
    return handleList
}
```

We simplify the factory pattern. Our implementation statically creates the handle objects and return them. The standard factory pattern creates object types dynamically based on its receiver. The handles use RelativeLocator to locate the position on the figure in order to display the handles.

View is responsible for the presentation of the data model, receives user inputs and forward them to Controller.

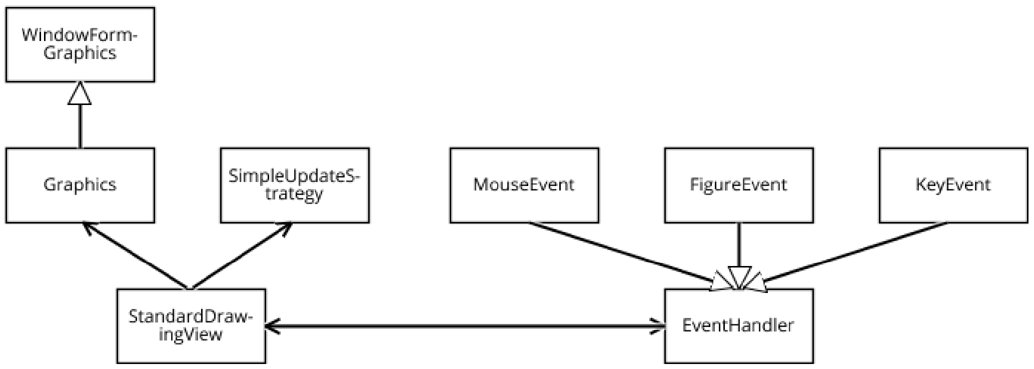

Figure 2.3: View

StandardDrawingView renders a StandardDrawing and listens to its changes. StandardDrawingView is a part of the observer pattern which receives mouse and key events and forwards them to the correspond handler. Since there ane no graphic libraries available in Grace we create Graphic which acts as an interface between Grace and graphic libraries. In that interface, we define all methods that represents functionalities which are necessary for the graphical user interface in our framework. Moreover we develop WindowFormGraphics as the concrete implementation of Graphic. Due to time constraints we are not able to finish the implementation of Grace graphic libraries however. As a result we do not have any user interface for our framework.

SimpleUpdateStrategy is a part of the strategy pattern which listens to the user interface changes and every time when changes with input events are recognized, it creates a new drawing by repainting the whole drawing.

```
//Strategy Pattern
class SimpleUpdateStrategy {
    method draw (graphics, drawingView) {
        drawingView.draw(graphics)
        var size := graphics.getWindowSize()
        graphics.repaint(0,0,0,0,size.width, size.height)
    }
}
```

Controller performs actions based on the user inputs in form of events. View listens to the events and forwards them to the corresponding controller part.

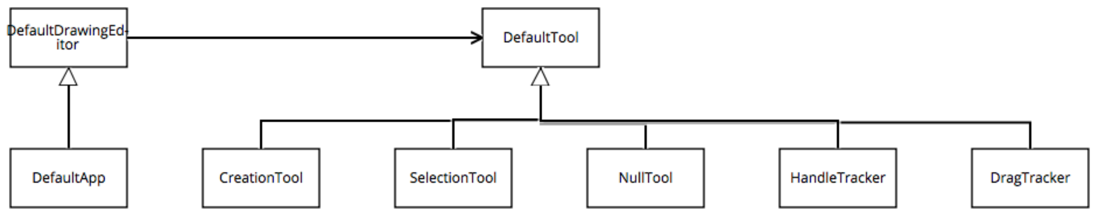

Figure 2.4: Controller

DefaultDrawingEditor defines the interface for coordinating the different objects that participate in a drawing editor. DefaultDrawingEditor delegates requests from view to the current tool. We use the mediator pattern to decouple the communication between view and tools. DefaultDrawingEditor is mediator, view and tool are colleagues. We use the factory pattern in our application to choose the current tool based on key events.

```
    // Factory Pattern to create tools
    method keyDown(keyEvent) {
        if (keyEvent.keyCode == "F1_KEY") then {
            self.setTool(Tool.SelectionTool(self))
        } elseif {keyEvent.keyCode == "F2_KEY"} then {
            self.setTool(Tool.CreationTool(self,AbstractFigure.RectangleFigure
                ))
        }
    }
}
```

When the key “F1” from the keyboard is pressed, the current tool changes to SelectionTool. When the key “F2” from the keyboard is pressed, the current tool changes to CreationTool. We use the state pattern to enable different behaviors for the different states of current tool. In our implementation there are 2 possible states in the first level: SelectionTool and CreationTool. While the current tool is SelectionTool. There are following possible states:

- User clicks outside of figures: Nothing happens.
- User clicks on one of the 8 handles: SelectionTool changes the state to HandleTracker. HandleTracker locates the selected Handle and performs actions based on the following mouse events inputs.
- User clicks inside the figure, but not on one of the eight handles: SelectionTool changes the state to DragTracker. DragTracker modifies the location of the selected figure based on following mouse events inputs.

We ignore the option to select several figures to simplify the framework. In that case nothing happens. The CreationTool creates a new figure and uses prototype pattern which has already predefined a figure. The method createFigure in CreationTool calls the clone method in AbstractFigure which returns a predefined prototype.

```
//In CreationTool
method createFigure {
    if (self.prototypeFigure == false) then {
        print "No prototype defined"
    }
    return self.prototypeFigure.clone
}

//In AbstractFigure
method clone {
    var cloneRect := RectangleFigure
    cloneRect := cloneRect.newRectangleFigureFromRect(self.
        getDisplayBox)
    return cloneRect
}
```

2.5 Research Results

Michael Kölling developed an evaluation framework for a teaching programming language by presenting eleven criteria which a first programming language should have (Kölling, 1999). In this chapter we use his evaluation model and make a qualitative comparative evaluation of the design pattern implementations in Grace and Java based on our experience in developing GraceHotdraw and Grace design pattern standalone implementations. In addition we present the constraints of Grace libraries and how they affects our GraceHotdraw implementation.

2.5.1 Evaluation of Grace

In this section we point out the language design differences between Java and Grace by applying the evaluation framework developed by Michael Kölling on Grace as first programming language. We observe the differences during the design pattern implementations in Grace.

Clean concepts The Grace developers mentioned that objects, classes and types are three separate clean concepts which can be used independently (Homer

& Noble, 2015). Our Grace design pattern implementations confirmed that statement. In the singleton Pattern we develop a single object without defining any classes. Using type is not necessary to define a class and its objects. Although types can be ignored, we define types in some of the design patterns to have some simple comparison base. In standalone design pattern applications types are often redundant and make the code unnecessarily longer. However in GraceHotdraw, we observe that having types can make the code easier to read, as they serve as bridges or interfaces between the different independent software components.

Pure object-orientation Kölling points out that a first teaching programming language should be only object-oriented. Grace supports this feature by making everything an object, including numbers and booleans, which we observe from the design pattern implementations. In the Visitor standalone application we present functionalities such as pay salary or print all employees as classes: PaySalary and PrintAllEmployees. In the Command standalone application we implement the commands remove product and store product as classes. These are only two examples which show the Grace high degree of pure object-orientation. In comparisons, Java is not even pure object-oriented because it has some primitive data types such as char, int or long which are not objects.

Safety The developers of Grace aim to make it strongly typed and catch type errors latest at run time. However in our current compiler version of Kernan, type does not have any effect on any classes at run time. As a result, using type for our study in this thesis only has documentation purpose. The developers of Grace plan more type checking at run time for future Kernan versions though. Both type static and dynamic checking would be possible.

High level According to Kölling programmers should not be responsible for machine internal tasks such as dynamic storage. Similar to Java, Grace has a built-in garbage collector which eliminates a lot of errors for students. During the the implementations of our study we have never encountered any storage problems.

Simple object/execution model Regarding memory allocation for objects many languages distinguish between stack and heap. Grace allocates all objects to the heap. Based on our study we confirm that this occurs automatically and is not visible on the language itself.

Readable syntax Based on our experience during the study there are both good and bad practices in Grace syntax:
Good

- Grace is a C similar programming language, making it easy for students to transit in one of the languages in the C-family.
- In general the syntax of Grace is simple comparing to application languages such as Java. For instance the syntax requires no semicolon required at the end of the line.

  ```
  //Grace vs Java
  command.execute(argument)
  command.execute(argument);
  ```

  The print method, for rookies in programming one of the most used method in the learning phase, is in Grace is much shorter than Java.

  ```
  //Grace vs Java
  print "Peter has got the contracting rates"
  System.out.println("Peter has got the contracting rates");
  ```

  Or when calling a method without any arguments in Grace, the brackets can be ignored.

  ```
  //Grace vs Java
  player.pressPlayButton
  player.pressPlayButton();
  ```

  The simple syntax in Grace lets students concentrate on its language specific features.

Bad

- Regarding control statements, in Java they all use parentheses to include their expression. Grace lacks the uniformity about whether to use braces or parentheses to include the expression. For and If-then statements use parentheses whereas while and else-if statements use braces

  ```
  // Statements in parentheses
  for (commandList) do { ...
  if (audioType == "mp3") then...
  // Statements in brackets
  while {iterator.hasNext} ...
  elseif {foodType == "Sandwich"} then ...
  ```

  We find the different usage of braces vs parentheses for control statements very confusing, especially for beginners in programming.

- Regarding the differences between syntax for mutable vs immutable variable declarations, for mutable the syntax is:

  ```
  var name := _name
  ```

  Whereas for immutable declaration the syntax is:

  ```
  def name := _name
  ```

  The distinction between ":=" and "=" for "mutable vs. immutable variable declarations" takes some time to getting used to for programmers with C similar languages background. The developer of Grace distinguish ':=' and '=' because they represent different language concepts. However we think that the expressions "def" and "var" should be enough to distinguish between mutable and immutable variables. During the study we often forget the ":" for declaration of mutable variables. For beginners in programming this syntax is another additional unnecessary thing they have to pay attention to.

No redundancy Pattern matching seems to be redundant for Grace. The developers of Grace argue that pattern matching makes Grace more effective in terms of teaching language and that student can compare and decide which way is easier to learn coding (Homer, Noble, Bruce, Black & Pearce, 2013). Our study cannot confirm or deny that argument. We can only show that the design pattern implementations are possible without using pattern matching.

Small Although the Grace core is quite small, we are able to implement all design patterns with no big difficulties. For GraceHotdraw implementation we observe some challenges regarding multiple constructors. In Grace multiple constructors are not explicitly possible. In the Hotdraw framework we often have to create multiple constructors to enable different object behaviors. Not explicitly having multiple constructors made the implementation of GraceHotdraw more difficult. There are two workarounds.

1. We define multiple classes that inherit from each other, with the same effect:

```
class RectangleFigure { ... }
class RectangleFigure (pointTopLeft, pointBottomRight) { inherit
    RectangleFigure ... }
class RectangleFigure(rectangle) { inherit RectangleFigure(...) ... }
```

2. We create multiple methods which depend on each other.

```
class RectangleFigure {
    method newRectangleFigure {
```

```
            return newRectangleFigureFromPoints(Util.Point(0, 0), Util.Point(0,
                0))
        }
        method newRectangleFigureFromPoints(pointTopLeft, pointBottomRight)
            {
            var figure := RectangleFigure
            figure.setBasicDisplayBox(pointTopLeft, pointBottomRight)
            return figure
        }
        method newRectangleFigureFromRect(rectangle) {
            return self.newRectangleFigureFromPoints(
                Util.Point(rectangle.x, rectangle.y),
                Util.Point(rectangle.x + rectangle.width, rectangle.y + rectangle.
                    height))
        }
        ...
    }
```

We observe another main difference between Java and Grace implementation of the prototype pattern. In Grace we must implement clone:

```
    class Hamburger (_typeOfburger,_levelofSpiciness,_size,_friesIncluded, _sauce) {
        ...
        method clone is override {
            var cloneHamburger := Hamburger (_typeOfburger,_levelofSpiciness,
                _size,_friesIncluded, _sauce)
            return cloneHamburger
        }
        ...
    }
```

This implementation is much worse than in Java, which build-in clone .

Grace is a teaching programming language. With its small core of language features Grace still enabled us to implement all design patterns and almost all of GraceHotdraw components.

Easy transition to other languages Other application languages such as Java support Grace basic language features as well. One Grace language feature which is not supported in Java is dynamic types. We show in our study that dynamic typing reduces the length of the code and is easier to implement design patterns. For example the state pattern (see the whole code in Appendix A.3.8). Java or C# implementations require an interface or abstract class RadioPlayer-State for the Radioplayer class to access all its states. This is not necessary in

Grace. Radioplayer can access all its states without any RadioPlayerState abstract class.
The implementations of design pattern and object-oriented programming in general widely use abstract classes and interfaces. Grace does not support abstract classes. Modules reusing is possible with inheritance.

Support for correctness assurance Since the Grace developers want to keep Grace core as small as possible, annotations for pre- and post-condition, variants and invariants are not implemented. They will develop these features as libraries. For our study there is no case where we think that correctness assurance is necessary for the implementations.

Suitable environment A suitable development environment is essential for every educational programming language. As Grace developers are still on the way to develop user friendly development environment, for our study we use Sublime Text as the editor, Kernan as the compiler. We navigate to file folders and run all the compiling commands on Terminal.

2.5.2 Comparison of GraceHotdraw and JHotdraw

In this section we point out the differences between GraceHotdraw and JHotdraw implementation.

User Interface Java contains many mature graphic libraries in its core such as AWT or Swing. Grace does not have any stable runnable graphic libraries. For displaying graphics we try to extend the compiler and build a graphic library on the top of the compiler. The implementation of graphic library is time consuming and not the focus of this thesis. Thus we define interfaces between GraceHotdraw and all future graphic libraries. We implement the adapter pattern to enable all third party graphic libraries for GraceHotdraw.

Collection In design pattern implementations we often have to use collections. Grace library does not include collection however. We implemented list.grace which acts as a LinkedList.

Abstract classes and interfaces Grace does not support any abstract classes and interfaces. A workaround for interfaces is using type. We do not need these language features to implement GraceHotdraw components. The concept of dynamic types in Grace makes all the interfaces in Hotdraw framework redundant.

2.6 Results Discussion

Our goal is to analyze the language features in Grace with regards to object-oriented design patterns by comparing design pattern implementations in Grace and Java.

Our results show that design pattern implementations in Grace are very similar to application languages such as Java. In the first place Grace is developed for teaching purposes and not explicitly for design pattern implementations. Nevertheless some of the language features and syntax of Grace make the design pattern implementations much easier compared to Java. Singleton, iterator and command patterns are in Grace core. Only the prototype pattern implementation is in Grace more difficult than in Java as clone method is in Java core.

As the development of a fully runnable GraceHotdraw framework is not a central research point there are limitations in our framework. Some of the functionalities are not complete and we cannot implement a functional graphic user interface. However with our implementation we set the basis for Grace developers to complete and further extend the framework.

There are many different ways to implement design patterns in Grace and many different versions and implementations of the language. We limit our study on one particular stable implementation of Kernan.

In the future we would like extend our study to other programming languages. So far our study is rather qualitative and we only compare the implementation of Grace and Java. We would like to compare Grace against other object-oriented languages to observe more design pattern language features. Moreover we would also like to extend the study to other implementations of Grace in order to further evaluate the language. We find the design pattern implementations of the current Kernan version very easy compared to Java. One possible approach is to restrict dynamic typing and then evaluate the design pattern implementations again.

2.7 Conclusion

In this thesis we present Grace as a educational language and show the differences of the design pattern implementations to Java. We highlight the Grace specific language features and syntax which make design pattern implementations easier. We show that the design pattern implementations in Grace are not significantly different, but still easier than an application language, in particular Java. Moreover our results show the strength and weaknesses of Grace as a teaching programming language based on the design pattern implementations and help the designers to further develop the language.

Appendix A Grace Design Patterns

In this appendix we present the implementation of 23 GoF design patterns in Grace. Since the collection libraries are not implemented in Grace we have to create our own:

```
def nothing is public = object {
    method get(index) {
        Exception.raise "Index out of bounds"
    }

    method do(procedure) {}

    method asString {
        ""
    }
}

class node(element′) is confidential {
// class node(element′) {
    def element = element′
    var next := nothing

    method get(index) {
        if (index == 0) then {
            element
        } else {
            next.get(index − 1)
        }
    }

    method add(element′) {
        if (next == nothing) then {
            next := node(element′)
        } else {
            next.add(element′)
        }
    }

    method do(procedure) {
        procedure.apply(element)
        next.do(procedure)
    }
```

```
        method asString {
            if (next == nothing) then {
                element.asString
            } else {
                "{element.asString}, {next.asString}"
                //element.asString
            }
        }
    }

    class with(elements) {
        var node := nothing
        var sizeOfList := 0
        method get(index) {
            node.get(index)
        }

        method add(element′) {
            if (node == nothing) then {
                node := node(element′)
            } else {
                node.add(element′)
            }
            sizeOfList := sizeOfList + 1
        }

        method do(procedure) {
            node.do(procedure)
        }

        method size {
            sizeOfList
        }

        method asString {
            "{node.asString}"
        }

        for (elements) do { element′ →
            add(element′)
        }
        method printAll {
            print "{node.asString}"
        }
    }
```

A.1 Creational Patterns

A.1.1 Abstract Factory

Definition Creation of groups of related objects without the requirement of specifying the exact concrete classes that will be used. The pattern creates the concrete object family at run time.

Standalone application We implement a delivery service system. There are 2 different kinds of delivery: Standard and fragile delivery. The package and calculate cost processes differs based on the current delivery type.

```
    class DeliveryService (factory, _nameOfProduct, _materialForPackaging) {
        var nameOfProduct := _nameOfProduct
        var packaging is readable,writeable := factory.createPackaging (
            _materialForPackaging)
        var deliveryCost is readable,writeable := factory.createDeliveryCost
    }

    class DeliveryFactory {
        method createPackaging (_materialForPackaging) {}
        method createDeliveryCost {}
    }

    class StandardDeliveryFactory {
        inherit DeliveryFactory

        method createPackaging (_materialForPackaging) is override {
            return StandardPackaging (_materialForPackaging)
        }

        method createDeliveryCost is override {
            return StandardCost
        }
    }
```

```
class FragileDeliveryFactory {
    inherit DeliveryFactory

    method createPackaging (_materialForPackaging) is override {
        return FragilePackaging (_materialForPackaging)
    }

    method createDeliveryCost is override {
        return FragileCost
    }
}

class Packaging (_materialForPackaging) {
    var materialForPackaging := _materialForPackaging
    method packageProduct {
        print "{materialForPackaging} is used to for this
            packaging process"
    }
}

class StandardPackaging (_materialForPackaging) {
    inherit Packaging (_materialForPackaging)
      alias packageStandardProduct = packageProduct

    method packageProduct is override {
        packageStandardProduct
    }
}

class FragilePackaging (_materialForPackaging) {
    inherit Packaging (_materialForPackaging)
      alias packageFragileProduct = packageProduct

    method packageProduct is override {
        packageFragileProduct
    }
}

class DeliveryCost {
    var cost := 10

    method calculateDeliveryCost {
        print "The cost for this delivery is {cost} Euro"
        }
    }

    class StandardCost {
        inherit DeliveryCost
          alias calculateCost = calculateDeliveryCost

        method calculateDeliveryCost is override {
            calculateCost
            return cost
        }
    }

    class FragileCost {
        inherit DeliveryCost
          alias calculateCost = calculateDeliveryCost

        method calculateDeliveryCost is override {
            cost := cost + 10
            calculateCost
            return cost
        }
    }
```

The following code create first a standard delivery and after that a fragile delivery:

```
var standardDeliveryFactory := StandardDeliveryFactory
var standardDeliveryService := DeliveryService (standardDeliveryFactory, "
    Iphone 10", "standard box")
standardDeliveryService.packaging.packageProduct
standardDeliveryService.deliveryCost.calculateDeliveryCost

var fragileDeliveryFactory := FragileDeliveryFactory
var fragileDeliveryService := DeliveryService (fragileDeliveryFactory, "Samsung
    Universe 7", "special box")
fragileDeliveryService.packaging.packageProduct
fragileDeliveryService.deliveryCost.calculateDeliveryCost
```

Output:

```
standard box is used to for this packaging process
The cost for this delivery is 10 Euro
special box is used to for this packaging process
The cost for this delivery is 20 Euro
```

A.1.2 Builder

Definition The builder pattern is a design pattern that enables step-by-step creation of complex objects using the correct sequence of actions. The construction is controlled by a director object that only needs to know the type of object to create.

Standalone application The program represents a burger menu order system. There are 2 menu choices: The standard burger and surprise meal menu. One can choose a type of burger independently from the type of menu. In our application we provide 2 different types of burger: Hamburger and CheeseBurger.

```
class MenuOrder {
    method makeMenu( menuBuilder) {
        menuBuilder.addMainDish
        menuBuilder.addSideOrder
        menuBuilder.addDrink
        menuBuilder.setPrice
    }
}

type MenuBuilder = {
    addMainDish → Done
    addSideOrder → Done
    addDrink → Done
    setPrice → Done
    getMenu → Done
}

class BurgerMenu (_burger){
    var menu is readable := Menu

    method addMainDish { menu.mainDish := _burger}
    method addSideOrder {menu.sideOrder := "Fries"}
    method addDrink {menu.drink := "Soft Drink"}
    method setPrice {menu.price := 8 }
}

class SurpriseMealMenu (_burger) {
    var menu is readable := Menu
```

```
        method addMainDish { menu.mainDish := _burger}
        method addSideOrder {menu.sideOrder := "Toy"}
        method addDrink {menu.drink := "Cola"}
        method setPrice {menu.price := 5 }
    }

    class Menu {
        var mainDish is readable, writable
        var sideOrder is readable, writable
        var drink is readable, writable
        var price is readable, writable

        method printOrder {
            "{mainDish} with {sideOrder}, {drink}, {price}"
        }
    }

    class Burger (_size) {
        var size := _size
    }

    class Hamburger (_size, _sauce) {
        inherit Burger (_size)
        var sauce := _sauce

        method asString is override {
            "Hamburger, {size}, {sauce}"
        }
    }

    class CheeseBurger (_size, _extraCheese) {
        inherit Burger (_size)
        var extraCheese := _extraCheese

        method asString is override {
            "CheeseBurger, {size}, {_extraCheese}"
        }
    }
```

We implement type MenuBuilder as a replacement for interface in Java. Alternatively we can make MenuBuilder a class and let BurgerMenu and SurpriseMealMenu inherit from it. The following code creates one hamburger menu and one surprise meal menu with Hamburger and CheeseBurger.

```
var hamburgerMenu : MenuBuilder := BurgerMenu (Hamburger("medium","
    ketchup"))
var menuOrder := MenuOrder
menuOrder.makeMenu (hamburgerMenu)
print "{hamburgerMenu.menu.printOrder}"

var surprisemealMenu : MenuBuilder := SurpriseMealMenu (CheeseBurger("
    large","yes"))
menuOrder.makeMenu (surprisemealMenu)
print "{surprisemealMenu.menu.printOrder}"
```

Output:

```
Hamburger, medium, ketchup with Fries, Soft Drink, 8
CheeseBurger, large, yes with Toy, Cola, 5
```

A.1.3 Factory Method

Definition Creation of objects without specifying the object type that is to be created in code. Subclasses decide which object type is to instantiate.

Standalone application We implement a food factory system. FoodFactory can generate either Western or Asian FoodFactory. WesternFoodFactory can create either Burger or Sandwich. AsianFoodFactory can create either BeijingDuck or Sushi.

```
class FoodFactory {
    method createFood(foodType) {
        print "Factory is generating {foodType}"
    }
}

class WesternFoodFactory {
    inherit FoodFactory

    method createFood (foodType) is override {
        if (foodType == "Burger") then {
            return Burger (foodType)
        } elseif {foodType == "Sandwich"} then {
            return Sandwich (foodType)
            } else {
                print "We do not have this kind of food"
            }
        }
}

class AsianFoodFactory {
    inherit FoodFactory

    method createFood (foodType) is override {
        if (foodType == "Beijing Duck") then {
            return BeijingDuck (foodType)
        } elseif {foodType == "Sushi"} then {
            return Sushi (foodType)
        } else {
            print "We do not have this kind of food"
        }
    }
}

class Food (_foodType) {
    var foodType := _foodType

    method serveFood {}
}

class Burger (_foodType) {
    inherit Food (_foodType)
    method serveFood is override {
        print "Its {foodType}, so bring fork and knife."
    }
}

class Sandwich (_foodType) {
    inherit Food (_foodType)
    method serveFood is override {
        print "Its {foodType}, so bring fork and knife."
    }
}

class BeijingDuck (_foodType) {
    inherit Food (_foodType)
    method serveFood is override {
        print "Its {foodType}, so bring the additional rice and
            chopstick to the table."
    }
}

class Sushi (_foodType) {
    inherit Food (_foodType)
    method serveFood is override {
        print "Its {foodType}, so bring Wasabi and ginger to the
            table."
    }
}
```

The following code produces first a Burger and then Sushi:

```
var westernFoodFactory := WesternFoodFactory
var burger := westernFoodFactory.createFood("Burger")
burger.serveFood

var asianFoodFactory := AsianFoodFactory
var sushi := asianFoodFactory.createFood("Sushi")
sushi.serveFood
```

The method createFood creates different types of food based on its receivers. The method serveFood performs different actions.

Output:
Its Burger, so bring fork and knife.
Its Sushi, so bring Wasabi and ginger to the table.

A.1.4 Prototype

Definition Instantiate a new object by copying all of the properties of an existing object, creating an new independent clone of that existing object.

Standalone application We implement Burger as the parent class, CheeseBurger and Hamburger as child classes of Burger. CheeseBurger and Hamburger both implements the method clone, which uses copy by value and creates new object by copying all properties of the existing object.

```
class Burger (_typeOfburger,_levelofSpiciness,_size,_friesIncluded) {
    method clone {}
```

```
        var typeOfburger is readable, writable := _typeOfburger
        var levelofSpiciness is readable, writable := _levelofSpiciness
        var size is readable, writable := _size
        var friesIncluded is readable, writable := _friesIncluded
    }

    class CheeseBurger (_typeOfburger,_levelofSpiciness,_size,_friesIncluded,
        _extraCheese) {
        inherit Burger (_typeOfburger,_levelofSpiciness,_size,_friesIncluded)
        var extraCheese is readable, writable := _extraCheese

        method clone is override {
            var cloneCheeseBurger := CheeseBurger (_typeOfburger,
                _levelofSpiciness,_size,_friesIncluded, _extraCheese)
            return cloneCheeseBurger
        }

        method asString is override {
            "{typeOfburger}, {levelofSpiciness}, {size}, {
                friesIncluded}, {extraCheese}"
        }
    }

    class Hamburger (_typeOfburger,_levelofSpiciness,_size,_friesIncluded, _sauce) {
        inherit Burger (_typeOfburger,_levelofSpiciness,_size,_friesIncluded)
        var sauce is readable, writable := _sauce

        method clone is override {
            var cloneHamburger := Hamburger (_typeOfburger,_levelofSpiciness,
                _size,_friesIncluded, _sauce)
            return cloneHamburger
        }

        method asString is override {
            "{typeOfburger}, {levelofSpiciness}, {size}, {
                friesIncluded}, {sauce}"
        }
    }
```

The following code produces first a CheeseBurger object and after that a clone from that object:

```
var CheeseBurger := CheeseBurger ("CheeseBurger", "Very spicy", "
    Medium", true, false)
var CheeseBurgerClone := CheeseBurger.clone
CheeseBurgerClone.extraCheese := true
print (CheeseBurgerClone)
print (CheeseBurger)
```

After modifying the extraCheese property we now have two different objects.

Output:

```
CheeseBurger, Very spicy, Medium, true, true
CheeseBurger, Very spicy, Medium, true, false
```

A.1.5 Singleton

Definition Makes sure that only one object can be instantiated for a class. This pattern provides a global access point to that object.

Standalone application Our implementation makes use of the object construction feature in Grace. We define a single object and define "singleton" as the global access point.

```
def singleton = object {
    var testAttribute is readable, writable := "original attribute"
    method asString is override {
        "I am a {testAttribute}"
    }
}
```

We now test for correct behavior of our implementation:

```
//How to access the singleton object
var instance := singleton
```

both variables instance and instance2 points to singleton. So in case we change the variable testAttribute in singleton object, both instance and instance2 are changed.

Output:

```
I am a testAttribute is modified
I am a testAttribute is modified
```

A.2 Structural Patterns

A.2.1 Adapter

Definition The adapter pattern provides a link between two incompatible types by wrapping the "adaptee" with a class that supports the interface required by the client.

Standalone application Our application is a media player which plays only audio files. We extend the player so that it is now able to play video (avi) files.

```
type MediaPlayer = {
    play → Done
}

class AviPlayer {
    method playAviFile (fileName) {
        print "Now we play Avi file {fileName}"
    }
}

class Player (_name) {
    var videoAdapter : MediaPlayer
    var name := _name

    method play (mediaType, fileName) {
        if (mediaType == "audio") then {
            //inbuilt support to play mp3 music files
            print "Now play audio file {fileName}"
        } elseif {mediaType == "avi"} then {
            //mediaAdapter is providing support to play other file formats
            videoAdapter := VideoAdapter(mediaType)
            videoAdapter.play(mediaType, fileName)
        } else {
            print ("Media {mediaType} is invalid. The Format is not
                supported")
        }
    }
}

class VideoAdapter (audioType) {
    var aviPlayer
```

```
        method play (mediaType, fileName) {
            if (mediaType == "avi") then {
                aviPlayer := AviPlayer
                aviPlayer.playAviFile(fileName)
            } else {
                print "File undefined"
            }
        }
    }
```

Originally Player can only plays audio as mediaType. The class VideoAdapter extends the mediaType of Player by avi. The type of both Player and VideoAdapter are MediaPlayer.

The following codes let the radio player plays different media in different media types

```
    var player : MediaPlayer := Player ("Media Player")
    player.play("audio", "My love.mp3")
    player.play("avi", "Grace is cool.avi")
    player.play("mov", "The Lord of Tthe Rings.mov")
    player.play("mkw", "Harry Potter.mkw")
```

```
Output:
Now play audio file My love.mp3
Now we play Avi file Grace is cool.avi
Media mov is invalid. The Format is not supported
Media mkw is invalid. The Format is not supported
```

A.2.2 Bridge

Definition Promotes loose coupling by separating the abstract elements of a class from its concrete implementation.

Standalone application Our application is a message sender systems. The system has three different sender types: Email, mobile and web sender. For each sender type we can either send messages or have the option to send them with a security layer.

```
    class MessageSendingBase {
        method sendMessage (messageTitle, messageBody) {}
```

```
}

class EmailSender {
    inherit MessageSendingBase
    method sendMessage (messageTitle, messageBody) is override {
        print "Message on the Email: '{messageTitle}' with the
            message: '{messageBody}' sent"
    }
}

class MobileSender {
    inherit MessageSendingBase
    method sendMessage (messageTitle, messageBody) is override {
        print "Message on the mobile phone: '{messageTitle}' with
            the message: '{messageBody}' sent"
    }
}

class WebSender {
    inherit MessageSendingBase
    method sendMessage (messageTitle, messageBody) is override {
        print "Mesaage on the web: '{messageTitle}' with the
            message: '{messageBody}' sent"
    }
}

class MessageBase (_messageSendingBase, _title, _body) {
    var messageSendingBase := _messageSendingBase
    var title := _title
    var body := _body

    method send {
        messageSendingBase.sendMessage(title, body)
    }
}

class MessageSecurity (_messageSendingBase, _title, _body, _securityCode) {
    inherit MessageBase (_messageSendingBase, _title, _body)
    var securityCode := _securityCode

    method send is override {
        if (securityCode == 1) then {
            messageSendingBase.sendMessage(self.title, self.body)
            } else {
                print "securityCode is wrong. Send failed"
            }
        }
    }
```

The base class for the implementation is MessageSendingBase. This class defines the method sendMessage that will be used to send a message. EmailSender, MobileSender and WebSender are concrete implementations which perform different actions for the abstract method sendMessage. The class MessageBase is the base class for abstractions. This class define the default properties of a message regardless of its sender type and hold a reference to an implementation object. We refine MessageBase with MessageSecurity for additional security layer. MessageSecurity simply requests a integer code from client. If the code is correct, the message will be sent.

The following code uses a WebSender to send a message with security layer:

```
var webSender := WebSender
var message := MessageSecurity (webSender,"Hello", "How are you", 1)
message.send
```

Output:
Mesaage on the web: 'Hello' with the message: 'How are you' sent

A.2.3 Composite

Definition Creates hierarchical object models and relationships between classes or entities. This pattern creates a tree structure of group of objects and treat them in similar way as a single object.

Standalone application Our application defines a company's composite structure. Employee is the base class. Contractor cannot have subordinates. A permanent employee can have other employees as subordinates which can also be permanent employee. The imported library list is our own developed library for collection.

```
import "../list" as list

class Employee (_name, _department) {
    var name is readable := _name
    var role is readable := _department
```

```
        method asString is override {
            "{name}, {role}"
        }

        method executeTask {
            print "{name}, {role} is executing a task"
        }

        method printAll {
        }
    }

    class PermanentEmployee (_name, _department) {
        inherit Employee (_name, _department)
        def subordinates = list.with []
        method addSubordinate (_subordinate) {
            subordinates.add(_subordinate)
        }

        method printAllEmployees {
            print "{self}"
            printAll
        }

        method printAll is override {
            var counter := subordinates.size − 1
            for (0 .. counter) do { n:Number →
                print "{subordinates.get(n)}"
                subordinates.get(n).printAll
            }
        }
    }

    //Contractor cannot have any subordinates
    class Contractor (_name, _department) {
        inherit Employee (_name, _department)
    }
```

Now we want to create a hierarchical object models and print all employees:

```
    var headManager is public:= PermanentEmployee ("Christ", "HeadManager"
        )
    var headRD is public := PermanentEmployee ("Florian", "Head Research
    and Development")
var headSoftwareDevelopment is public := PermanentEmployee ("Harry", "
    Head Software Development")
var scientist is public := PermanentEmployee ("Fabian", "Scientist in
    Research and Development")
var contractor is public := Contractor ("Andrew", "Contractor Scientist
    in Research and Development")
var developer is public := PermanentEmployee ("Ben", "Developer in
    Software Development")
var tester is public := PermanentEmployee ("Corey", "Tester in Software
    Development")
headManager.addSubordinate(headRD)
headManager.addSubordinate(headSoftwareDevelopment)
headSoftwareDevelopment.addSubordinate(developer)
headSoftwareDevelopment.addSubordinate(tester)
headRD.addSubordinate(scientist)
headRD.addSubordinate(contractor)
headManager.printAllEmployees
```

We implement the method printAll as the only common functionality of the object model. The method executeTask is a extension for decorator pattern which we will explain later.

Output:
Christ, HeadManager
Florian, Head Research and Development
Fabian, Scientist in Research and Development
Andrew, Contractor Scientist in Research and Development
Harry, Head Software Development
Ben, Developer in Software Development
Corey, Tester in Software Development

A.2.4 Decorator

Definition Add additional features to an object dynamically. We use decorator pattern when we want to add other functionalities to the object, but do not want to increase the complexity of its class.

Standalone application We demonstrate this pattern by extending our application for composite pattern. We add the feature to determine whether an employee is busy or not. First we add the method executeTask to the base class Employee. Then we implement two new classes:

```
import "CompositePattern" as CompositeEmployeePackage

class DecoratorEmployee (_employee) {
    inherit CompositeEmployeePackage.Employee (_employee.name, _employee.
        role)

    var decoratorEmployee is readable := _employee

    method executeTask {
        decoratorEmployee.executeTask
    }
}

class DecoratedTask (_employee) {
    inherit DecoratorEmployee (_employee)
      alias executeTaskEmployee = executeTask
    var isBusy := false

    method executeTask is override {
        executeTaskEmployee
        print "{name} is busy"
        isBusy := true
    }
}
```

We set the attribute isBusy := true every time after executing the method executeTask.

The following codes let decoratedCEO execute a task and set isBusy := true:

```
var decoratedCEO := DecoratedTask (CompositeEmployeePackage.
    headManager)
decoratedCEO.executeTask
```

Output:

```
Christ, HeadManager is executing a task
Christ is busy
```

A.2.5 Facade

Definition In a complex or bad designed system this pattern simplifies the access to the modules of that system and provides a simple interface that hides the implementation details.

Standalone application We demonstrate this pattern by extending our application for the prototype pattern. We provide a simple interface to the method clone. The application will copy either CheeseBurger or Hamburger in run-time environment.

```
import "../01_CreationalPatterns/PrototypePattern" as
    prototypePattern

class BurgerFacade (_burger) {
    var burger := _burger

    method cloneBurger {
        burger.clone
    }
}
```

The following code produces a cheese burger and a clone from the cheese burger. After that the code modifies the attribute extraCheese from false to true. Finally we print both objects.

```
var cheeseBurger := prototypePattern.CheeseBurger ("CheeseBurger", "Very
     spicy", "Medium", true, false)
var burgerFacade := BurgerFacade (cheeseBurger)
var cloneCheeseBurger := burgerFacade.cloneBurger
cloneCheeseBurger.extraCheese := true
print (cheeseBurger)
print (cloneCheeseBurger)
```

Output:

```
CheeseBurger, Very spicy, Medium, true, false
CheeseBurger, Very spicy, Medium, true, true
```

A.2.6 Flyweight

Definition Minimizes resource usages while working with large number of objects by avoiding the creation of similar objects.

Standalone application We demonstrate this pattern by extending our application for the Factory Pattern. The only difference to factory pattern are the food objects which we declare in the base class FoodFactory. In the concrete implementation of FoodFactory we only create the food objects once. They reuse them after the first use.

```
class FoodFactory {
    var burgerObject := false
    var sandwichObject := false
    var sushiObject := false
    var beijingDuckObject := false
    method createFood(foodType) {
        print "Factory is generating {foodType}"
    }
}

class WesternFoodFactory {
    inherit FoodFactory
      alias createWesternFood (foodType) = createFood (foodType)

    method createFood (foodType) is override {
        createWesternFood (foodType)
        if (foodType == "burger") then {
            if (burgerObject == false) then {
                burgerObject := Burger (foodType)
            } else {
                //do nothing
            }
            return burgerObject
        } elseif {foodType == "sandwich"} then {
            if (sandwichObject == false) then {
                sandwichObject := Sandwich (foodType)
            } else {
                //do nothing
            }
            return sandwichObject
        } else {
            print "We do not have this kind of food"
        }
    }
}

class AsianFoodFactory {
    inherit FoodFactory
      alias createAsianFood (foodType) = createFood (foodType)

    method createFood (foodType) is override {
        createAsianFood (foodType)
        if (foodType == "Beijing Duck") then {
                if (beijingDuckObject == false) then {
                    beijingDuckObject := BeijingDuck (foodType)
                } else {
                    //do nothing
                }
                return beijingDuckObject
            } elseif {foodType == "Sushi"} then {
                if (sushiObject == false) then {
                    sushiObject := Sushi (foodType)
                } else {
                    //do nothing
                }
                return sushiObject
            } else {
                print "We do not have this kind of food"
            }
        }
}

class Food (_foodType) {
    var foodType := _foodType
    print "{foodType} created"

    method serveFood {}
}

class Burger (_foodType) {
    inherit Food (_foodType)
    method serveFood {
        print "Its {foodType}, so bring fork and knife"
    }
}

class Sandwich (_foodType) {
    inherit Food (_foodType)
    method serveFood {
        print "Its {foodType}, so bring fork and knife"
    }
}

class BeijingDuck (_foodType) {
    inherit Food (_foodType)
    method serveFood {
            print "Its {foodType}, so bring the additional rice and
                chopstick to the table"
        }
    }

    class Sushi (_foodType) {
        inherit Food (_foodType)
        method serveFood {
            print "Its {foodType}, so bring Wasabi and ginger to the
                table"
        }
    }
```

The following codes produce two burger objects. However WesternFoodFactory will not create any more burger objects after it instantiates the first one.

```
var westernFoodFactory := WesternFoodFactory
var burger := westernFoodFactory.createFood("burger")
var burger2 := westernFoodFactory.createFood("burger")
burger.serveFood
```

Output:
Factory is generating burger
burger created
Factory is generating burger
Its burger, so bring fork and knife

A.2.7 Proxy

Definition acts as an interface to other objects or software components.

Standalone application In this application we demonstrate how to use the pattern to cache files.

```
class File (_fileName) {
    var fileName := _fileName
    method open {}
}

class PDFDocument (_fileName) {
    inherit File (_fileName)
```

```
        method loadFile {
            print "Load file: {fileName}"
        }

        method open is override {
            print "Open file: {fileName}"
        }
    }

    class ProxyFile (_fileName) {
        inherit File (_fileName)
        var pDFDocument := false

        method open is override {
            if (pDFDocument == false) then {
                pDFDocument := PDFDocument (fileName)
                pDFDocument.loadFile
            }
            pDFDocument.open
        }
    }
```

The following codes creates a file and open this file twice.

```
    var file := ProxyFile("textBook.pdf")
    file.open
    file.open
```

The first time opening a file, the application loads the file and then opens it. The second time the application does not have to load the file anymore in order to open this file.

Output:

```
Load file: textBook.pdf
Open file: textBook.pdf
Open file: textBook.pdf
```

A.3 Behavioural Patterns

A.3.1 Chain of Responsibility Pattern

Definition Defines a linked list of handlers which are able to process requests. The first handler in the list processes a request first. If the first handler is unable to process the request, it will pass the request to the next handler in the list.

Standalone application This is a coin machine handler system which only accepts 50 Cent and 1 Euro coins. The system examine the type of coins by analyzing the weight and diameter of the coins.

```
class Coin (_weight, _diameter) {
    var weight is readable, writable := _weight
    var diameter is readable, writable := _diameter
}

class CoinHandlerBase {
    var _successor is readable, writable

    method handleCoin (coin:Coin) is abstract {}

    method setSuccessor(successor) {
        _successor := successor
        _successor._successor := false
    }
}
class FiftyCentHandler {
    inherit CoinHandlerBase

    method handleCoin (coin) is override {
        if ((coin.weight == 20) && (coin.diameter == 0.15)) then {
            print "50 cent recieved"
        } elseif {_successor != false} then {
            _successor.handleCoin(coin)
        } else {
            print "Cannot handle this coin"
        }
    }
}

class OneEuroHandler {
        inherit CoinHandlerBase
        method handleCoin(coin) is override {
            if ((coin.weight == 30) && (coin.diameter == 0.2)) then {
                print "1 Euro recieved"
            } elseif {_successor != false} then {
                _successor.handleCoin(coin)
            } else {
                print "Cannot handle this coin"
            }
        }
    }
```

Now the client creates objects from FiftyCentHandler and OneEuroHandler and determines the order of the handlers.

```
    var h50 := FiftyCentHandler
    var h100 := OneEuroHandler
    h50.setSuccessor(h100)
```

Every time when a coin is passed to the handler the system calls 50 cent handler first. If the coin is not a 50 cent coin, the handler passes this coin to the 1 Euro handler.
Now we create three different types of coins and let the handlers process them.

```
    var fiftyCent := Coin (20,0.15)
    var oneEuro := Coin (30,0.2)
    var twoEuro := Coin (40,0.3)
    h50.handleCoin(fiftyCent)
    h50.handleCoin(oneEuro)
    h50.handleCoin(twoEuro)
```

Output:

```
50 cent recieved
1 Euro recieved
Cannot handle this coin
```

A.3.2 Command

Definition Stores all information which are needed to perform an action within a single object.

Standalone application Command pattern is a part of Grace core:

```
for (commandList) do { command →
            command.execute
}
```

Now we extend the pattern to an example application. Our implementation is a product handler system. The commands are “remove - ” and “store products”.

```
import "../list" as list

class Command {
    method execute {}
    var product
}

class Invoker {
    var commandList := list.with []

    method takeComamnd (command) {
        commandList.add (command)
    }

    method placeCommands{
        for (commandList) do { command →
            command.execute
        }
    }
}

class Product (productName) {
    var name := productName
    print "product {name} created"
    method store{
        print "Store product {name}"
    }
    method remove{
        print "Remove product {name}"
    }
}

class RemoveProduct (aProduct) {
    inherit Command
    product := aProduct
```

```
        method execute() {
            product.remove
        }
    }

    class StoreProduct (aProduct) {
        inherit Command
        product := aProduct
        method execute {
            product.store
        }
    }
```

The following code creates a product, one remove command and one store command. After that our code passes the two commands to Invoker. Finally Invoker place the commands and execute them.

```
    var product := Product ("Smartphone")
    var removeCommand := RemoveProduct (product)
    var storeComamnd := StoreProduct (product)
    var invoker := Invoker
    var test := "test"
    invoker.takeComamnd (storeComamnd)
    invoker.takeComamnd (removeCommand)
    invoker.takeComamnd (test)
    invoker.placeCommands
```

Output:
product Smartphone created
Store product Smartphone
Remove product Smartphone

A.3.3 Interpreter

Definition Specifies how to evaluate a sentence or the grammar in a language. The pattern can easily extends its grammar.

Standalone application Our implementation enables a simple AND expression with 2 terminal expressions.

```
    class ExpressionBase {
        method interpret(context) {}
```

```
}

class TerminalExpression (_data) {
    inherit ExpressionBase
    var data is readable := _data
    var exp1 := ""
    var exp2 := ""

    method interpret (context) is override {
        parseExpression(context)
        if ((exp1 == data) || (exp2 == data)) then {
            return true
        } else {
            return false
        }
    }

    method parseExpression (context) {
        for (1 .. context.size) do { n:Number →
            if (context.at(n) == " ") then {
                exp1 := context.substringFrom(1) to (n − 1)
                exp2 := context.substringFrom(n + 1) to (context.size)
            }
        }
    }
}

class AndExpression (exp1,exp2){
    inherit ExpressionBase
    var expression1 is readable,writable := exp1
    var expression2 is readable,writable := exp2

    method interpret (context) is override {
        if (expression1.data == expression2.data) then {
            print "Warning: expression1 is equal expression2"
            return false
        } else {
            return expression1.interpret(context) && expression2.interpret(
               context)
        }
    }
}
```

The client creates two new TerminalExpression and provides them as input for AndExpression. After that the variable andrewisMarried from AndExpression proves whether this expression is true or not.

```
var andrew := TerminalExpression ("Andrew")
var married := TerminalExpression ("Maried")
var andrewisMarried := AndExpression(andrew,married)
print "{andrewisMarried.interpret ("Maried Andrew")}"
```

The output is true.

A.3.4 Iterator

Definition Provides an interface to access the elements of an aggregate object in sequence without knowing its underlying structure.

Standalone application Since iterator pattern is part of Grace core we do not implement any standalone application.

A.3.5 Mediator

Definition Reduces coupling between classes that communicate with each other by providing a central object. The classes send the messages to this central object which forwards them to their receivers.

Standalone application Our application is a demonstration of an online lecture. Teacher objects can present slides to all students, receives question from a particular student and answer directly to it. Student objects can receive slides, ask questions to Teacher and receive answers. Mediator is responsible for the whole communication. Teacher and Student do not send messages to each other directly. Mediator takes the messages from one participant and send them to another participant. Each time a new Student joins the lecture, Mediator will add this Student object to its Student list. When Teacher updates the slides on the board, Mediator notifies all its Student objects and presents them the new slides on the board.

```
import "../list" as list

class ClassMember (_mediator, _name) {
    var mediator := _mediator
    var name is readable:= _name
}

class Teacher (_mediator, _name) {
    inherit ClassMember (_mediator, _name)

    method receiveQuestion (question, attendee) {
        print "Teacher received question: {question} from {
            attendee} "
    }

    method answerQuestion(answer, attendee) {
        print "Teacher answered: {answer} to {attendee.name} "
        mediator.sendAnswer(answer, attendee)
    }

    method presentSlides (url){
        print "Teacher changed slide to {url}"
        mediator.updateBoard(url)
    }
}

class Student (_mediator, _name) {
    inherit ClassMember (_mediator, _name)
    _mediator.addNewStudentToClass(self)

    method askQuestion (question) {
        print "Student {name} asks question: {question}"
        _mediator.sendQuestion(question,self)
    }

    method receiveAnswer (answer) {
        print "{name} has received the answer {answer}"
    }

    method receiveSlides (url) {
        print "Student {name} revceives the slides from the file:
            {url}"
    }

    method asString {
        "{name}"
    }
}
```

```
class Mediator {
    var teacher is readable, writable
    var classMember := list.with []

    method addNewStudentToClass (_student) {
        classMember.add(_student)
    }
    method sendAnswer(answer, _student) {
        _student.receiveAnswer(answer)
    }

    method sendQuestion(question, _student) {
        teacher.receiveQuestion(question, _student)
    }

    method updateBoard (url) {
        for (classMember) do { member →
            member.receiveSlides (url)
        }
    }
}
```

The following code creates an online lecture with one teacher and two students. They communicate with each other via the Mediator object "mediator".

```
var mediator := Mediator
var teacherDavid := Teacher (mediator, "David")
mediator.teacher := teacherDavid
var andrew := Student(mediator, "Andrew")
var michael := Student(mediator, "Michael")
```

Now we let the Teacher object present a new slides. The Mediator object immediately notifies all students and presents them the new slides.

```
teacherDavid.presentSlides ("MarketingStrategyMethods.pdf")
```

Output:

```
Teacher changed slide to MarketingStrategyMethods.pdf
Student Andrew revceives the slides from the file: MarketingStrategyMethods.pdf
Student Michael revceives the slides from the file: MarketingStrategyMethods.pdf
```

After that Andrew asks the teacher a question. The Mediator object first forwards the question to teacher and then forward the teacher's answer back to Andrew.

```
andrew.askQuestion("'What does economy of scale mean ?'")
teacherDavid.answerQuestion("'A proportionate saving in costs gained
    by an increased level of production.'", andrew)
```

Output:
Student Andrew asks question: 'What does economy of scale mean ?'
Teacher received question: 'What does economy of scale mean ?' from Andrew
Teacher answered: 'A proportionate saving in costs gained by an increased level of production.' to Andrew
Andrew has received the answer 'A proportionate saving in costs gained by an increased level of production.'

A.3.6 Memento

Definition Captures the current state of an object and stores it in a way that it can be restored in the future.

Standalone application We implement a log system of firmware versions of machines. Memento stores the all the necessary information on a machine object. Machine can create a Memento object and restore to its last firmware version.

```
class Machine (_id, _version, _firmWareDescription) {
    var id is readable,writable := _id
    var firmWareDescription is readable,writable := _firmWareDescription
    var version is readable,writable := _version

    method createUndo {
        return Memento (id, version, firmWareDescription)
    }

    method restoreFromUndo (memento) {
        version := memento.version
        firmWareDescription := memento.firmWareDescription
    }

    method logMachine {
        print "MachineID: {id}. Version: {version}. Firmware : {
            firmWareDescription}"
    }
}
```

```
class Memento (_id, _version, _firmWareDescription) {
    var id is readable := _id
    var version is readable := _version
    var firmWareDescription is readable := _firmWareDescription
}

class Caretaker {
    var memento is readable,writable
}
```

The client creates a Machine object and its Memento object.

```
// Initialise Machine
var machine := Machine("12","1.0","Beta version");
machine.logMachine
// Set undo point
var history := Caretaker
history.memento := machine.createUndo
```

Now our client modifies this Machine object and later restores its firmware to its previous version.

```
// Modify version and firmware of that Machine object
machine.version := "2.0"
machine.firmWareDescription := "Stable Version"
machine.logMachine

// Undo
machine.restoreFromUndo(history.memento)
machine.logMachine
```

Output:
```
MachineID: 12. Version: 1.0. Firmware : Beta version
MachineID: 12. Version: 2.0. Firmware : Stable Version
MachineID: 12. Version: 1.0. Firmware : Beta version
```

A.3.7 Observer

Definition The pattern is used when there is one-to-many relationship between objects. When one object is changed, it notifies all other objects about its changes.

Standalone application The application monitors all the actions from FamousPerson instances and provides notifications for all the changes. Based on the changed state of the FamousPerson instance Newspaper and Tabloid objects perform different actions.

```
import "../list" as list

class FamousPerson {

    var monitors := list.with []
    var action

    method attach (_monitor) {
        monitors.add (_monitor)
    }

    method setAction (_action) {
        action := _action
        notify
    }

    method getAction {
        return action
    }

    method notify {
        for (monitors) do { _monitor →
            _monitor.update (self)
        }
    }
}

class FamousActor (_name) {
    inherit FamousPerson
    var name is readable := _name
}

class FamousPersonMonitor {
    method update (_famousPerson) {}
}

class Newspaper {
    inherit FamousPersonMonitor
```

```
        method update (_famousPerson) is override {
            print "Headline: {_famousPerson.name} {_famousPerson.
               getAction}"
        }
    }

    class Tabloid {
        inherit FamousPersonMonitor

        method update (_famousPerson) is override {
            print "Headline: Oh my God. {_famousPerson.name} {
               _famousPerson.getAction}"
        }
    }
```

The client creates a FamousActor, Newspaper and Tabloid object.

```
    var emma := FamousActor ("Emma")
    var cnn := Newspaper
    var theSun := Tabloid
```

After that the client attaches the Newspaper and Tabloid objects to the FamousActor object.

```
    emma.attach (cnn)
    emma.attach (theSun)
```

Finally when the state of the FamousActor object changes, it notifies all the attached observers.

```
    emma.setAction ("takes part in a new movie")
```

Output:

```
Headline: Emma takes part in a new movie
Headline: Oh my God. Emma takes part in a new movie
```

A.3.8 State

Definition Performs different object behaviors based on the its internal state changes. The pattern enables the client to determine the class for an object at run-time.

Standalone application Our application implements the different states of a radio player. The four different states are StandbyState, MusicPlayingState, MusicPausedState and RadioState. The player has two buttons which invokes different actions on each state. The below state diagram describes how the internal states of the radio player change.

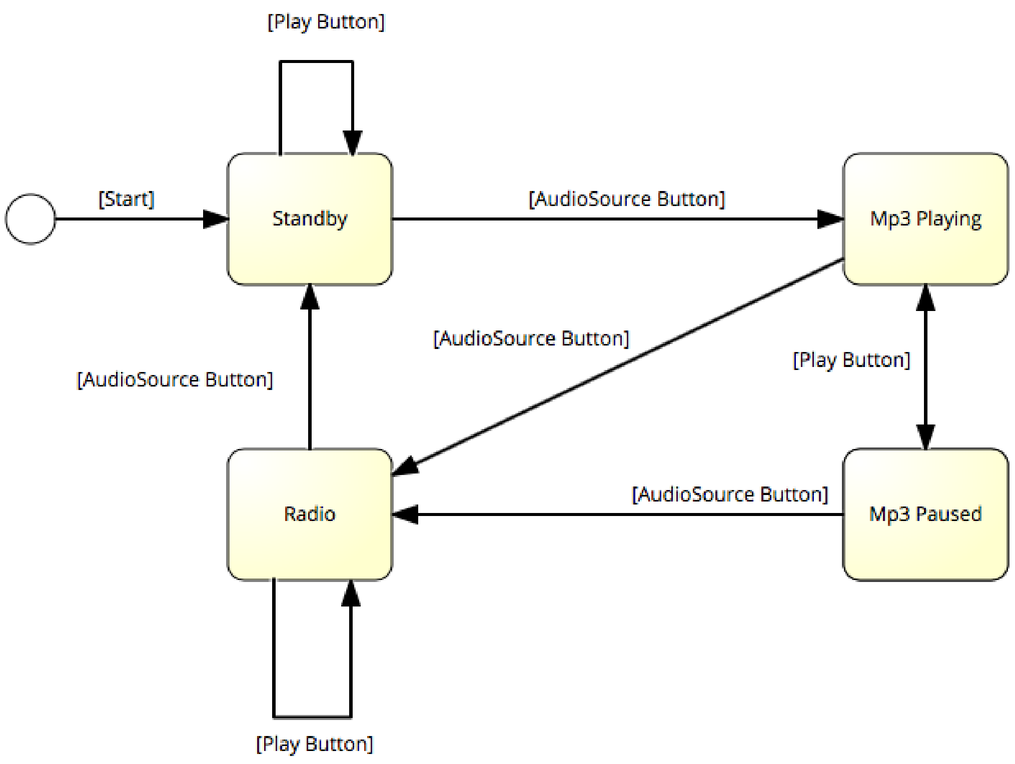

Figure 2.5: States of RadioPlayer

```
class RadioPlayer (stateOfPlayer) {
    var currentState is readable, writable := stateOfPlayer

    method pressPlayButton {
        currentState.pressPlayButton(self)
    }

    method pressAudioSourceButton {
        currentState.pressAudioSourceButton(self)
    }
    method asString {
        print "{currentState}"
    }
```

```
}

//Class RadioPlayerState not required for dynamic typing.
class RadioPlayerState {
    method pressPlayButton(player) {}
    method pressAudioSourceButton(player) {}
}

class StandbyState {
    inherit RadioPlayerState
    print "In Standby State"

    method pressPlayButton (player) is override {
        print "Play Button pressed. No Action"
    }

    method pressAudioSourceButton (player) is override {
        print "Audio Soure Button pressed"
        player.currentState := MusicPlayingState
    }
}

class MusicPlayingState {
    inherit RadioPlayerState
    print "In Music Playing State"

    method pressPlayButton (player) is override {
        print "Play Button pressed. Pause the Music"
        player.currentState := MusicPausedState
    }

    method pressAudioSourceButton (player) is override {
        print "Audio Soure Button pressed"
        //print "{player}"
        player.currentState := RadioState
    }
}

class MusicPausedState {
    inherit RadioPlayerState
    print "In Music paused State"

    method pressPlayButton (player) is override {
            print "Play Button pressed. Play the Music"
            player.currentState := MusicPlayingState
        }

        method pressAudioSourceButton (player) is override {
            print "Audio Soure Button pressed"
            player.currentState := RadioState
        }
    }

    class RadioState {
        inherit RadioPlayerState
        print "In Radio State"

        method pressPlayButton (player) is override {
            print "Play Button pressed. Change the Radio Chanel"
            player.currentState := MusicPausedState
        }

        method pressAudioSourceButton (player) is override {
            print "Audio Soure Button pressed. Back to Standby"
            player.currentState := StandbyState
        }
    }
```

The following codes demonstrate how the player works. We create a RadioPlayer instance and set its default internal state to StandbyState.

```
    var player := RadioPlayer(StandbyState)
    player.pressPlayButton
    player.pressAudioSourceButton
    player.pressPlayButton
    player.pressPlayButton
    player.pressAudioSourceButton
    player.pressAudioSourceButton
```

Output:

```
In Standby State
Play Button pressed. No Action
Audio Soure Button pressed
In Music Playing State
Play Button pressed. Pause the Music
In Music paused State
Play Button pressed. Play the Music
```

```
In Music Playing State
Audio Soure Button pressed
In Radio State
Audio Soure Button pressed. Back to Standby
In Standby State
```

A.3.9 Strategy

Definition This pattern creates a group of algorithms which are chosen at runtime.

Standalone application Our application represents a calculator which can add and subtract 2 numbers.

```
type Calculator = {
    executeOperation (num1:Number, num2:Number) → done
}

class OperationAdd {
    method executeOperation(num1, num2) {
        return num1 + num2
    }
}

class OperationSubstract {
    method executeOperation(num1, num2) {
        return num1 − num2
    }
}

class Client (_strategy) {
    var strategy := _strategy
    method executeCalculator (num1, num2) {
        return strategy.executeOperation(num1, num2)
    }
}
```

The following codes let client choose different kind of operation at runtime.

```
var operationAdd:Calculator := OperationAdd
var operationSubtract:Calculator := OperationSubstract
var context := Client(operationAdd)
var result := context.executeCalculator (20,10)
print "20 + 10 = {result}"

context := Client (operationSubtract)
result := context.executeCalculator (20,10)
print "20 - 10 = {result}"
```

Output:
20 + 10 = 30
20 - 10 = 10

A.3.10 Template Method

Definition Template method pattern defines the basic flow of an algorithm, enables multi step algorithm and changes on the individual steps.

Standalone application The application defines the basic sequence of all different kind of applications.

```
class Application {

    method initialize {}

    method startApplication {}

    method endApplication {}

    method runApplication {
        initialize
        startApplication
        endApplication
    }
}

class WebApplication {
    inherit Application

    method initialize is Override {
        print "Initialize the Web Application"
    }
```

```
        method startApplication is Override {
            print "Start the Web Application"
        }

        method endApplication is Override {
             print "End the Web Application"
        }
    }

    class DesktopApplication {
        inherit Application

        method initialize is Override {
            print "Initialize the Desktop Application"
        }

        method startApplication is Override {
            print "Start the Desktop Application"
        }

        method endApplication is Override {
             print "End the Desktop Application"
        }
    }
```

The following codes create both WebApplication and DesktopApplication and demonstrate how the behavior of runApplication changes on each case.

```
    var application := WebApplication
    application.runApplication
    application := DesktopApplication
    application.runApplication
```

Output:

```
Initialize the Web Application
Start the Web Application
End the Web Application
Initialize the Desktop Application
Start the Desktop Application
End the Desktop Application
```

A.3.11 Visitor

Definition This pattern separates an complex algorithm from an object structure it operates on.

Standalone application In this application we extend our previous composite standalone application by adding two functionalities: Print all Employees and pay salary to all Employees.

```
import "../list" as list

class Employee (_name, _role) {
    var name is readable := _name
    var role is readable := _role

    method asString is override {
        "{name}, {role}"
    }

    method executeTask {
        print "{name}, {role} is executing a task"
    }
}

class PermanentEmployee (_name, _department) {
    inherit Employee (_name, _department)
    def subordinates = list.with []
    method addSubordinate (_subordinate) {
        subordinates.add(_subordinate)
    }

    method accept (visitor) {
        visitor.visitPermanentEmployee (self)
        var counter := subordinates.size − 1
        for (0 .. counter) do { n:Number →
            subordinates.get(n).accept(visitor)
        }
    }
}

//Contractor cannot have any subordinates
class Contractor (_name, _department) {
    inherit Employee (_name, _department)
```

```
        method accept (visitor) {
            visitor.visitContractor (self)
        }
    }

    //Base class for visitors of Employee class
    class EmployeeVisitor {
        method visitPermanentEmployee (employee) {}
        method visitContractor (employee) {}
    }

    //Visitor 1: pay salary to all employees
    class PaySalary {
        inherit EmployeeVisitor
        method visitPermanentEmployee (employee) is override {
            print "{employee.name} has got the salary"
        }
        method visitContractor (employee) is override {
            print "{employee.name} has got the contracting rates"
        }
    }

    //Visitor 2: print all employees
    class PrintAllEmployees {
        inherit EmployeeVisitor
        method visitPermanentEmployee (employee) is override {
            print "{employee}"
        }
        method visitContractor (employee) is override {
            print "Contractor {employee}"
        }
    }
```

The following codes create a hierarchical object models, pay salary to all Employee objects and finally print them all:

```
    var headManager is public:= PermanentEmployee ("Christ", "HeadManager"
        )
    var headRD is public := PermanentEmployee ("Florian", "Head Research
        and Development")
    var headSoftwareDevelopment is public := PermanentEmployee ("Harry", "
        Head Software Development")
```

```
var scientist is public := PermanentEmployee ("Fabian", "Scientist in
    Research and Development")
var contractor is public := Contractor ("Andrew", "Scientist in Research
     and Development")
var developer is public := PermanentEmployee ("Ben", "Developer in
    Software Development")
var tester is public := PermanentEmployee ("Corey", "Tester in Software
    Development")

headManager.addSubordinate(headRD)
headManager.addSubordinate(headSoftwareDevelopment)
headSoftwareDevelopment.addSubordinate(developer)
headSoftwareDevelopment.addSubordinate(tester)
headRD.addSubordinate(scientist)
headRD.addSubordinate(contractor)

headManager.accept(PaySalary)
headManager.accept(PrintAllEmployees)
```

Output:

```
Christ has got the salary
Florian has got the salary
Fabian has got the salary
Andrew has got the contracting rates
Harry has got the salary
Ben has got the salary
Corey has got the salary
Christ, HeadManager
Florian, Head Research and Development
Fabian, Scientist in Research and Development
Contractor Andrew, Scientist in Research and Development
Harry, Head Software Development
Ben, Developer in Software Development
Corey, Tester in Software Development
```

References

Begosso, L. C., Begosso, L. R., Gonçalves, E. M. & Gonçalves, J. R. (2012). An Approach for Teaching Algorithms and Computer Programming using Greenfoot and Python. In *2012 Frontiers in Education Conference Proceedings* (pp. 1–6). IEEE.

Bishop, J. (2007). *C# 3.0 Design Patterns*. O'Reilly Media, Inc.

Black, A. P., Bruce, K. B., Homer, M. & Noble, J. (2012). Grace: The Absence of (Inessential) Difficulty. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software* (pp. 85–98). ACM.

Black, A. P., Bruce, K. B., Homer, M. & Noble, J. (2013). The Grace Programming Language. Retrieved May 20, 2016, from http://gracelang.org/applications/home/

Black, A. P., Bruce, K. B. & Noble, J. (2013). The Grace Programming Language Draft Specification Version 0.3. 1261.

Black, A. P., Bruce, K. B. & Noble, J. (2016a). The Grace Programming Language Draft Specification Version 0.7.0.

Black, A. P., Bruce, K. B. & Noble, J. (2016b). The Grace Standard Prelude Draft Specification Version 0.7.0.

Black, A., Bruce, K. B. & Noble, J. (2010). Designing the Next Educational Programming Language.

Brant, J. M. (1995). Hotdraw.

Cooper, J. W. (2002). *C# Design Patterns: A Tutorial*. Addison-Wesley Professional.

Cunningham, W. (1994). A CRC Description of HotDraw. Retrieved May 20, 2016, from http://www.c2.com/doc/crc/draw.html

Farooq, M. S., Khan, S. A., Ahmad, F., Islam, S. & Abid, A. (2014). An Evaluation Framework and Comparative Analysis of the Widely Used First Programming Languages. *PLOS ONE*, *9*(2).

Froehlich, G., Hoover, H. J., Liu, L. & Sorenson, P. (1997). Hooking into Object-Oriented Application Frameworks. In *Proceedings of the 19th international conference on Software engineering* (pp. 491–501). ACM.

Gamma, E. & Eggenschwiler, T. (2007). JHotDraw as Open-Source Project. Retrieved May 20, 2016, from http://www.jhotdraw.org

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Gupta, D. (2004). What is a Good First Programming Language? *Crossroads*, *10*(4), 7–7.

Hadjerrouit, S. (1998). Java As First Programming Language: A Critical Evaluation. *ACM SIGCSE Bulletin*, *30*(2), 43–47.

Hannemann, J. & Kiczales, G. (2002). Design Pattern Implementation in Java and AspectJ. In *ACM Sigplan Notices* (Vol. 37, *11*, pp. 161–173). ACM.

Hatanaka, I. & Hughes, S. C. (1999, July). Providing Multiple Views in a Model-View-Controller Architecture. US Patent 5,926,177. Google Patents.

Homer, M. & Noble, J. (2013). A Tile-Based Editor for a Textual Programming Language. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)* (pp. 1–4). IEEE.

Homer, M. & Noble, J. (2014). Combining Tiled and Textual Views of Code. In *2014 Second IEEE Working Conference on Software Visualization (VISSOFT)* (pp. 1–10). IEEE.

Homer, M. & Noble, J. (2015). Object Creation in Grace. In *Proceedings of the 18th European Conference on Pattern Languages of Program* (p. 21). ACM.

Homer, M., Noble, J., Bruce, K. B., Black, A. P. & Pearce, D. J. (2013). Patterns as Objects in Grace. *ACM SIGPLAN Notices*, *48*(2), 17–28.

Johnson, R. E. (1992). Documenting Frameworks using Patterns. In *ACM Sigplan Notices* (Vol. 27, *10*, pp. 63–76). ACM.

Kaiser, W. et al. (2001). Become a Programming Picasso with JHotDraw. *JavaWorld, February.*

Kölling, M. (1999). The Problem of Teaching Object-Oriented Programming, Part II: Environments. In *Journal of Object-Oriented Programming.* Citeseer.

Lei, S. (2003). Towards Programming for the Non-Technical. In *2003 IEEE Symposium on Human Centric Computing Languages and Environments, 2003. Proceedings* (pp. 291–292). IEEE.

Lo, C.-A., Lin, Y.-T. & Wu, C.-C. (2015). Which Programming Language Should Students Learn First? A Comparison of Java and Python. In *2015 International Conference on Learning and Teaching in Computing and Engineering (LaTiCE)* (pp. 225–226). IEEE.

Metsker, S. J. & Wake, W. C. (2006). *Design Patterns in Java.* Addison-Wesley Professional.

Noble, J., Homer, M., Bruce, K. B. & Black, A. P. (2013). Designing Grace: Can an Introductory Programming Language Support the Teaching of Software Engineering? In *2013 26th International Conference on Software Engineering Education and Training (CSEE&T)* (pp. 219–228). IEEE.

Papaspyrou, N. S. & Zachos, S. (2013). Teaching Programming through Problem Solving: The Role of the Programming Language. In *2013 Federated Conference on Computer Science and Information Systems (FedCSIS)* (pp. 1545–1548). IEEE.

Pawelczak, D. & Baumann, A. (2014). Virtual-C - a Programming Environment for Teaching C in Undergraduate Programming Courses. In *2014 IEEE Global Engineering Education Conference (EDUCON)* (pp. 1–7). IEEE.

Riehle, D. (2000). Case Study: The JHotDraw Framework. *Framework Design: A Role Modeling Approach*, 138–158.

Serrano-Laguna, Á., Torrente, J., Iglesias, B. M. & Fernández-Manjón, B. (2015). Building a Scalable Game Engine to Teach Computer Science Languages. *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje*, *10*(4), 253–261.

Uehara, M. (2009). Design and Implementation of Cafe: A Programming Language for Beginners. In *2009 29th IEEE International Conference on Distributed Computing Systems Workshops, 2009. ICDCS Workshops' 09* (pp. 368–373). IEEE.

Wolfgang, P. (1994). *Design Patterns For Object-Oriented Software Development.* Reading, Mass.: Addison-Wesley.