NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

JAIME SILVA VICENTE BRITO

Bachelor in Computer Science

ASSESSMENT OF JAVASCRIPT'S ADVANCED COMPOSITION MECHANISMS BASED ON DESIGN PATTERNS

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon

June, 2025

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

ASSESSMENT OF JAVASCRIPT'S ADVANCED COMPOSITION MECHANISMS BASED ON DESIGN PATTERNS

JAIME SILVA VICENTE BRITO

Bachelor in Computer Science

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

Examination Committee

Chairs: Sérgio Marco Duarte
Assistant Professor, NOVA University Lisbon
José Vicente Pereira dos Reis
Assistant Professor, Iscte

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
June, 2025

Assessment of JavaScript's advanced composition mechanisms based on design patterns

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.18) [26].

Abstract

In the past, several studies have been conducted that analyzed the modularity support of object-oriented languages. However, few studies have focused on JavaScript's support for modularity and module composition based on design patterns.

JavaScript has some advanced features that are not available in many object-oriented languages and are not used in the traditional implementation of these patterns, as we have found documented in several books on JavaScript implementations of Gang-of-Four patterns.

To the best of our knowledge, no study has used these advanced JavaScript features to improve the traditional approach to implementing design patterns and analyze JavaScript's modularity support.

This thesis aims to fill this gap by evaluating JavaScript's support for modularity based on modularity properties. The results show that some patterns benefit significantly from JavaScript's advanced features, while others remain structurally unchanged or gain little.

Keywords: JavaScript, Object-Oriented programming, Module Composition, Modularity, Gang-of-Four Design Patterns, Separation of concerns

Resumo

No passado, foram realizados vários estudos que analisaram o suporte à modularidade em linguagens orientadas a objetos. No entanto, poucos se focaram no suporte da linguagem JavaScript à modularidade e à composição de módulos com base em padrões de design.

O JavaScript possui funcionalidades avançadas que não estão disponíveis em muitas linguagens orientadas a objetos e que não são utilizadas nas implementações tradicionais destes padrões, como foi identificado em diversos livros sobre implementações dos padrões Gang-of-Four em JavaScript.

Tanto quanto sabemos, nenhum estudo utilizou essas funcionalidades avançadas do JavaScript para melhorar a abordagem tradicional de implementação dos padrões de design e analisar o seu impacto no suporte à modularidade.

Esta dissertação pretende preencher essa lacuna, avaliando o suporte do JavaScript à modularidade com base em propriedades de modularidade. Os resultados demonstram que alguns padrões beneficiam significativamente das funcionalidades avançadas do JavaScript, enquanto outros permanecem estruturalmente inalterados ou apresentam melhorias reduzidas.

Palavras-chave: JavaScript, Programação Orientada a Objetos, Composição de Módulos, Modularidade, Padrões de Design Gang-of-Four, Separação de preocupações

Contents

List of Figures

Listings

Acronyms

AOP Aspect-Oriented Programming *(p. 73)*

GoF Gang of Four *(pp. 1–3, 39, 66, 72–74)*

OO Object-Oriented *(pp. 1, 6, 9, 12, 13, 16, 18, 20, 24, 28, 39, 61, 66, 69, 71, 73, 74)*

OOP Object-Oriented Programming *(pp. 7–10, 12, 19, 28, 59, 70, 72)*

1 Introduction

This chapter serves as the thesis preparation's introduction, presenting the context and description of the problem, the motivation and objectives of the thesis, the proposed approach and the document structure.

1.1 Context and Description

JavaScript is a programming language that is widely used in web development, being the sixth most popular language by the TIOBE index [47]. Since their release, functions and Objects has been a fundamental component [20]. JavaScript got several updates and new features, but the most important was the ES6 in 2015 that introduced classes to bring it closer to other programming languages like Java and C++.

Design patterns, more specifically Gang-of-Four (GoF) design patterns [16] were commonly used as a basis for evaluating languages based on their support for composition and modularity mechanisms [18][39][31][42][37]. The GoF patterns enclose a wide range of composition and design challenges and have been implemented in numerous programming languages.

1.2 Motivation and Objectives

Understanding JavaScript's object-oriented (or OO) features is essential for comprehending how the language supports modularity and separation of concerns [43]. In the past, various studies have been made to evaluate the modularity support for diverse programming languages [18][39][31][42], but there is a lack of studies that evaluate the support of JavaScript for modularity and composition mechanisms [44][19], and some books have already implemented the GoF design patterns in JavaScript [46][36][11][35], but never deeply explore the support for modularity or thoroughly explore advanced features from JavaScript (such as object delegation and reflection) to implement the design patterns. This thesis aims to do this, thus contributing to filling this gap in the state of the art.

The main goal of this thesis is to evaluate the support of JavaScript for modularity and composition mechanisms, by analyzing the implementation of the GoF design patterns in JavaScript using modularity properties [18][39][31]. JavaScript has some advanced composition mechanisms, such as object delegation, reflection, mixins classes and prototype based delegation that can be used to implement the GoF design patterns. However, the traditional implementation of the GoF design patterns in JavaScript do not take advantage of these features [46][36][11][35], so the implementation can be improved using them. In the end, modularity properties will be used to evaluate the support from JavaScript and determine if some pattern benefits from these advanced mechanisms or if the patterns retain their classical structure.

1.3 Proposed Approach

To achieve the objectives, we will use two sets of complete collections of the GoF design patterns with one of them from the author Simon Timms [46], that have all 23 GoF implementation and the other from different sources and authors [11][36][35], to create another complete set of implementations (to ensure that the study is not affected by any special cases and to test reusability, a part of the code is reusable if it can be used in both collections). We will try to improve all implementations using the advanced features present in JavaScript (such as object delegation and reflection).

For the JavaScript environment we will use Visual Studio Code [50] as the IDE using JavaScript plugin to run the code. After all, we will analyze the results based on modularity properties to evaluate the support of JavaScript for modularity and composition mechanisms.

1.4 Document Structure

The remainder of this document is organized as follows:

- Chapter 2 (State of the Art) - reviews the main concepts of object-oriented programming and several advanced language features including reflection, mixins, duck typing, polymorphism and method Dispatch.
- Chapter 3 (JavaScript) - presents key features of JavaScript related to module representation and composition and advanced composition mechanisms that appear promising in bringing advantages over existing implementations [46][36][11][35].
- Chapter 4 (Study Setup) - presents the study setup, including the terminology used in this thesis and the study design.
- Chapter 5 (Results and Discussion) - presents the results of the study, including the modularity properties of the traditional and improved implementations of the GoF design patterns in JavaScript.

- Chapter 6 (Design Pattern) - presents some implementations of the GoF design patterns in JavaScript with improvements from the traditional implementations.
- Chapter 7 (Related work) - presents some studies about the implementation of the GoF design patterns in different programming languages.
- Chapter 8 (Conclusions and Future Work) - presents the conclusions of the thesis and some future work that can be done to improve the results obtained.

2

State of the Art

This chapter provides an overview of the state of the art in the field of programming languages, focusing on the concepts of *type checking*, *polymorphism*, *duck typing*, *mixin classes*, *method dispatch*, *family polymorphism*, *virtual classes and functions* and *aspect oriented programming*. These concepts are essential to understand how in JavaScript the different modules are better represented (designed) and how to better compose them in a flexible and scalable way, using modularity properties like *Locality, reusability, composition transparency, (un)pluggability, extensibility* and *closure under composition*. The chapter also discusses how these concepts are applied in modern programming languages, mainly in JavaScript.

2.1 Type Checking

Within the realm of programming languages, there is a concept known as 'Type checking'. This involves validating and enforcing data type restrictions to ensure type safety and reduce the likelihood of type errors. This validation can occur either during the compilation phase (static check) or during program execution (dynamic check). Furthermore, it has the potential to influence the flexibility of automatic type conversions without compromising information, with this flexibility categorized as either Strong or Weak.

2.1.1 Dynamic and Static Type Checking

Dynamic typing is a feature found in specific programming languages that don't need explicit declarations of data types. These languages can decide on the type for each variable dynamically, allowing for adjustments during program execution or while it's running.

On the other hand, *static typing* is a system where variables get associated with a data type during compilation and this assignment stays constant throughout the program's execution. This kind of binding enhances type safety and identifies errors early on. However, it can also be more restrictive and less flexible than dynamic typing.

JavaScript employs dynamic type checking. This means that the type of variable is determined at runtime and it can change during the program's execution.

2.1.2 Weakly and Strongly typed languages

Weakly or *loosely typed* languages don't bother much about the data type stored in a variable. They let the programmer skip type conversions (casting), meaning variables can change types more easily and doesn't require type declaration.

In contrast, a *strongly typed* language has more rigid typing rules during compilation. This implies that errors and exceptions are more prone to occur at this stage. These rules impact variable assignment, function return values, procedure arguments and function calling. In other words, variables can't change types during program execution and require a type declaration.

JavaScript is an example of a loosely typed language. The type of variable is determined at runtime and it can change during the program's execution. This is a double-edged sword, as it can lead to unexpected behavior and errors.

2.2 Duck Typing

In computer programming, `duck typing` follows the principle "If it walks like a duck and it quacks like a duck, then it must be a duck" [14].

Duck typing is an approach to figure out if an object follows an interface by checking if it has methods with the same names as those specified in the interface.

This is a concept that is often used in dynamic languages, such as JavaScript and Python. It is a way to implement polymorphism without the need for *inheritance*.

In JavaScript, duck typing is often used due to its dynamic nature. Here is an example from the listing 2.1:

```
1  function Duck(name) {
2      this.name = name;
3      this.quack = function() {
4          return "Quack!";
5      };
6      this.walk = function() {
7          return "I'm walking like a duck.";
8      };
9  }
10 function Dog(name) {
11     this.name = name;
12     this.bark = function() {
13         return "Woof!";
14     };
15     this.walk = function() {
16         return "I'm walking like a dog.";
17     };
18 }
19 function makeItQuack(animal) {
20     if ('quack' in animal) {
21         return animal.quack();
```

```
    } else {
        return "This animal can't quack.";
    }
}
function makeItWalk(animal) {
    return animal.walk();
}
let donald = new Duck("Donald");
let bobby = new Dog("Bobby");

console.log(makeItQuack(donald)); // Outputs: "Quack!"
console.log(makeItQuack(bobby)); // Outputs: "This animal can't quack."
console.log(makeItWalk(donald)); // Outputs: "I'm walking like a duck."
console.log(makeItWalk(bobby)); // Outputs: "I'm walking like a dog."
```

Listing 2.1: Example of Duck Typing

In the example 2.1, the function `makeItWalk` prints the way the animal walks and the function `makeItQuack` checks if the object passed to it has a `quack` method. If it does, it calls the method; if not, it returns a string stating that the animal can't quack. This is a simple demonstration of duck typing in JavaScript.

2.2.1 Duck Typing vs Polymorphism

While both polymorphism and duck typing concern concepts that are closely related, they are not exactly the same.

Polymorphism is an OO concept in which, objects of multiple types, can be handled in a generic way by using the same interface. It does this by allowing a method or function to adopt several selves based on the type of object, in other words acts differently based on the object type.

Duck typing is similar to dynamic typing in that duck typing is more interested in the methods and the properties an object of a class possesses than the actual type of the object. The polymorphism feature, on the other hand, enables classes with disparate class structures to interact seamlessly by using a shared interface, while duck typing facilitates the substitution of objects from different types while still responding to the same methods. Both are ways of attaining the same goal [53].

In the Polymorphism example 2.2, `Dog` is a subclass of the `Animal` that overrides the `speak` method. And when Dog is Simply put, Animal's speak method is overridden and Animal's version is no longer used. In the duck typing scenario, the `makeItQuack` function which takes any object that owns a quack method independent of class or type.

```
// Polymorphism in JavaScript
class Animal {
    speak() {
        return 'I am an animal';
    }
}
```

```
class Dog extends Animal {
    speak() {
        return 'Woof!';
    }
}
let myPet = new Dog();
console.log(myPet.speak()); // Output: 'Woof!'
// Duck Typing in JavaScript
let duck = {
    quack: function() {
        return 'Quack!';
    }
};
let notADuck = {
    quack: function() {
        return 'I am not a duck, but I can quack!';
    }
};
function makeItQuack(animal) {
    console.log(animal.quack());
}
makeItQuack(duck); // Outputs: 'Quack!'
makeItQuack(notADuck); // Outputs: 'I am not a duck, but I can quack!'
```

Listing 2.2: Example of the difference between Polymorphism and Duck Typing

2.3 Family Polymorphism

Family polymorphism in OOP is a generalized form of polymorphism that allows to statically declare and manage relationships between several classes enclosed within a family class [15]. This concept is based on the idea of a family of classes can be used interchangeably through a common interface or base class. This concept ensures that objects of different classes can be treated as objects of a common superclass, thus allowing for dynamic method binding and increased flexibility in code design, this concept is every similar with the purpose of the *abstract factory* pattern[31].

Family polymorphism is used to improve reusability and safety of the code, however JavaScript don't have a built-in support for family polymorphism, but it can be implemented using the prototype system, but we only can achieve the code reuse and not the safety of the code.

In the example in 2.3 we have a `createFamily` function that creates a new family of objects. Each family has a private `familyName` variable that is shared among all members of the family. The `Parent` and `Child` constructors have access to this `familyName` variable through a closure, which allows them to share this common state. This is not exactly the same as Family Polymorphism in static typed languages, but it does allow for a similar kind of behavior where a group of related objects can share common state and behavior.

```
function createFamily() {
    let familyName = 'Smith';
    const Parent = function() {
        this.name = `Parent ${familyName}`;
    };
    Parent.prototype.describe = function() {
        console.log(`I am a parent of the ${familyName} family.`);
    };
    const Child = function() {
        this.name = `Child ${familyName}`;
    };
    Child.prototype.describe = function() {
        console.log(`I am a child of the ${familyName} family.`);
    };
    return {
        newParent: function() {
            return new Parent();
        },
        newChild: function() {
            return new Child();
        }
    };
}
const family = createFamily();
const parent = family.newParent();
const child = family.newChild();

parent.describe(); // Outputs: "I am a parent of the Smith family."
child.describe();  // Outputs: "I am a child of the Smith family."
```

Listing 2.3: Example of emulating Family polymorphism in JavaScript

2.4 Virtual Classes and functions

In OOP, a *virtual class* is a nested class within a family class that can be replaced and refined by subclasses of a family class that inherits from the previous one [27]. This concept is related with *virtual function*, which is a function that can be overridden by a subclass. From the mainsteam programing languages, Scala is the sole language that allows the use of virtual classes and family polymorphism. Other languages like Java and C++ allows the use for nested classes but these can't be overridable and polyformic like in Scala.

In JavaScript the concept of virtual classes doesn't directly translate, because JavaScript uses a prototype-based inheritance and object delegation rather than class-based inheritance. However, using the concept of virtual, we can use the concept of *virtual function* or *virtual method*, which is an inheritable and overridable function or method that is dynamically dispatched and being an important part of polymorphism [49]. JavaScript by default treat all functions or methods as virtual and don't require a modifier to change

this behavior like in Java. It can be implemented using the prototype system as shown in the example in 2.4.

```
function Person(name) {
    this.name = name;
}
// Add a new method to the prototype
Person.prototype.greet = function() {
    console.log(`Hello, ${this.name}!`);
};
// override the greet method using the prototype
Person.prototype.greet = function() {
    console.log(`Hi, ${this.name}!`);
};

const person = new Person('John');
person.greet(); // Output: Hi, John!
```

Listing 2.4: Example of Virtual Function in JavaScript

2.5 Method Dispatch

In *Object-Oriented Programming* (OOP), dispatch or *dispatching*, also known as method dispatch, is the mechanism through which a call to an overridden method or an interface method is resolved at runtime [6][52]. This dynamic resolution enables the execution of different implementations of a method based on the actual type of the object involved. Dispatch serves as the algorithm determining which method should be invoked in response to a message, although its implementation can vary significantly across different programming languages.

To learn more about the specifics of dispatch in JavaScript see section 3.8

2.5.1 Single Dispatch

In most OO languages, messages are dispatched to receiver objects, with the method invoked determined by the `runtime type` of the receiver, constituting *single dispatch*. This approach, while effective for messages where one argument dictates the method, proves inadequate for messages where multiple arguments hold equal significance, such as arithmetic operations like addition, subtraction, multiplication and division. These limitations extend to other operations like equality testing and ordering relations, alongside parallel iteration through collections. This asymmetry in single dispatch not only complicates everyday programming tasks beyond system implementation but also exemplifies a type of polymorphism where method execution hinges on the type of single parameter, most the time the receiver of the message.

2.5.2 Double Dispatch

Double dispatch is a technique in which the method executed depends on the runtime types of both the receiver and the argument. Unlike method overloading, which resolves calls based solely on compile-time types, double dispatch dynamically selects the method by applying single dispatch successively. It was originally proposed to address the *asymmetry problem* in single-dispatch languages, where only the receiver determines the method selected. In practice, double dispatch typically requires each participant to implement a method that resends the call to the other object, encoding the type of the sender in the method name. While this approach resolves type asymmetry, it comes with increased maintenance complexity.

2.5.3 Multiple Dispatch

Multiple dispatch is a form of polymorphism where method resolution depends on multiple parameters, normally depending on the type of all parameters involved in the method call, but in other cases the method resolution might consider only a subset of parameters, extending the concept of double dispatch. Some object-oriented languages overcome the limitations of single dispatch by employing multiple dispatch, allowing multiple arguments to participate in method lookup, known as multi-methods. In multiple dispatch languages, programmers can handle several significant arguments by writing multi-methods that dispatch on each. This approach simplifies programming tasks such as object equality, pairwise iteration and rendering shapes on output devices.

Multiple dispatch offers several advantages over double dispatch: it eliminates the need for double dispatch, reducing complexity and potential errors; it enhances coding speed and readability by streamlining method bodies and offering a clear overview of available implementations and it allows for declarative specification of method lookup in multi-methods, which is preferred over procedural specification in double dispatch, improving code clarity and reducing the likelihood of errors.

2.6 Modularity and Modularity Properties

In OOP, modularity is a design principle that breaks downs and organize a software system into distinct, independent modules or components. Each module encapsulates a piece of functionalities or a group of related functions, allowing a separation of concerns, improving code reuse, maintainability and scalability [30]. There are a lot of modularity properties, but of this thesis, we will be focusing on *locality*, *reusability*, *composition transparency* and *(un)pluggability* used by Jan Hannemann and Gregor Kiczale [18], the *extensibility* added by Monteiro and Gomes [31] and *closure under composition* used by Herrejon [25] (in the section 6.1 is possible to see the modularity properties definitions).

2.7 Mixin Classes

Mixin classes provide a means of code reuse without relying on strict inheritance [5]. These classes include general-purpose methods designed for sharing among various classes. The process typically involves creating a class with the desired methods, which is then utilized to enhance other classes. These classes, featuring the general-purpose methods, are termed mixin classes. Importantly, mixin classes are not usually instantiated or directly called; their sole purpose is to pass on their methods to other classes. In JavaScript, mixin classes can be implemented using the *prototype* system. It is possible to see the use of mixin class in the example from the listing 2.5:

```
// Define a mixin class
let mixin = {
  sayHello: function() {
    console.log(`Hello, my name is ${this.name}`);
  },
  sayGoodbye: function() {
    console.log(`Goodbye, my name was ${this.name}`);
  }
};
// Define a base class
function Person(name) {
  this.name = name;
}
// Apply the mixin to the base class
Object.assign(Person.prototype, mixin);
// Create a new instance
let person = new Person('Jaime Brito');
// Use the methods from the mixin
person.sayHello();  // Outputs: "Hello, my name is Jaime Brito"
person.sayGoodbye(); // Outputs: "Goodbye, my name was Jaime Brito"
```

Listing 2.5: Example of mixin class implemented through the prototype mechanism

In this example, we define a mixin with two methods: `sayHello` and `sayGoodbye`. We then define a `Person` class and use `Object.assign()` to copy the methods from the mixin into the `Person` class's prototype [34] (but also can be used the `extends` or `Object.setPrototypeOf(obj, mixin)`). This allows all instances of `Person` to use the methods defined in the mixin. `Object` is a built-in JavaScript constructor that turns the input into an object. `Object.assign()` copies all properties from one or more 'source' objects to a 'target' object, returning the modified 'target' object

This approach provides a flexible way to share behavior between classes in JavaScript. It's important to note that this is not true inheritance, as the `Person` class does not inherit from the mixin. Instead, the mixin's methods are simply copied into the `Person` class's prototype.

3

JavaScript

This chapter aims to present an overview of JavaScript's features, highlighting the mechanisms that support modularity. Also provide a basic understanding of the language and its main features, so that the reader can understand the code examples provided in the following chapters.

3.1 Introduction to JavaScript

JavaScript, widely integrated into all modern browsers, occupies a pivotal role in empowering websites and enriching web interfaces. It is a high-level, interpreted language that is used to create interactive effects within web browsers. JavaScript was developed in May 1995 by Brendan Eich for use in the Netcape browser and was later adopted by Microsoft for Internet Explorer. JavaScript is expressive and has unique features not found in the C family of languages. The present chapter looks into the language's expressiveness, with a particular focus on OO and the support for module composition. It explores alternative routes to OOP and module composition, drawing inspiration from functional programming concepts. The most important version of the JavaScript language is ECMAScript 6, that is sixth edition of JavaScript introduced in June 2015 and ECMAScript is the standard specification of JavaScript to ensure compatibility in all browsers and environments. Since then, new versions have been released annually. However, the main goal of this chapter is to give the basic information to understand the various ways design patterns can be implemented in JavaScript as a means to enhance modularity as represented by the modularity properties.

3.2 The Flexibility of JavaScript

JavaScript's remarkable flexibility empowers programmers to craft programs of varying complexity. Supporting diverse programming styles, including functional and OO paradigms, the language accommodates both straightforward procedural code and the more complex world of OOP. Its simplicity in the fundamental subset makes it accessible

to beginners while seamlessly scaling up for advanced programmers tackling complex projects. JavaScript not only has the capacity to emulate patterns from other languages but also introduces its own distinctive features [11].

In the listing 3.1, the procedural approach involves defining *functions* for actions, while the OO yields a class with start and stop methods, showcasing *prototype-based inheritance*. An alternative OO approach introduces a different syntax for declaring methods within the class. Further exploration involves adding methods to the Function object prototype to create methods in classes and employing method chaining for a more concise syntax using `Function.prototype.method`.

```
1  // Procedural aproach for a start and stop functions for an animation
2  function startAnimation() {
3      ...
4  }
5  function stopAnimation() {
6      ...
7  }
8  // Animation class using object-oriented approach
9  var Anim = function() {
10     ...
11 };
12 Anim.prototype.start = function() {
13     ...
14 };
15 Anim.prototype.stop = function() {
16     ...
17 };
18 // Usage
19 var myAnim = new Anim();
20 myAnim.start();
21 ...
22 myAnim.stop();
23 // Alternative object-oriented approach
24 var Anim = function() {
25     ...
26 };
27 Anim.prototype = {
28     start: function() {
29         ...
30     },
31     stop: function() {
32         ...
33     }
34 };
```

Listing 3.1: Example of flexibility in JavaScript

In essence, JavaScript's support for diverse coding styles allows developers to select

the most suitable approach for a specific project. These styles come with varying characteristics, impacting factors such as *code size*, *efficiency*, *performance* and *modularity*, providing programmers with a versatile toolkit to address the unique demands of their coding endeavors.

3.3 General concepts

3.3.1 Variables, Data and Reference Types

As a dynamically typed language, JavaScript does not require explicit declarations of *variable types*. As a weakly typed language, these are dynamically assigned based on their content, however JavaScript is case sensitive which means it distinguishes between uppercase and lowercase letters. In JavaScript, everything is an object, except for the primitive *datatypes* (which are automatically wrapped with objects when needed) that includes three types: *booleans*, *numbers* and *strings*. Notably, JavaScript treats *integers* and *floats* as the same type, differentiating itself from other languages.

Functions, objects, *arrays* and *lists* constitute essential data structures. Functions enclose executable code, objects serve as composite datatypes and arrays represent ordered collections within the realm of specialized objects. Additionally, *null* and *undefined* serve as distinct datatypes in JavaScript.

JavaScript employs a mechanism where primitive datatypes are passed by value, while other datatypes are passed by reference. Other feature is the ability of variables to change their type dynamically, adapting to the type of values assigned to them.

Type casting in JavaScript offers flexibility. For instance, `toString` can be utilized for numbers or booleans, `parseFloat` and `parseInt` convert strings to numbers and double negation (`!!num`) help in casting to a boolean. This adaptability reduces concerns about type errors, reflecting the weakly typed nature of JavaScript.

JavaScript provides various ways to define variables, such as `var`, `let` and `const` [17]. Being function-scoped, `Var` can be raised and has either function or global *scope*. On the other hand, `let` is block-scoped, limited to the block in which it's declared. `Const` variables, also block-scoped and immutable, meaning their values cannot be reassigned after the first assignment. If we try to reassign a value to a `const` variable after has been initialized, it would throw a `TypedError`. By adopting `let` and `const`, developers can write more predictable, readable and maintainable code, avoiding common pitfalls associated with `var`.

3.3.2 Lists, Arrays and Matrices

The terms "list" and "array" are often used interchangeably, but definitions vary. In JavaScript, both lists and arrays store collections of values. Lists, created with square brackets, are versatile, holding values of any type and being modifiable. Arrays, a specialized type, are created with the Array constructor or square brackets, limited to

a specific type and fixed length. Arrays shine with large datasets, while lists prioritize flexibility. There's no specific ArrayList in JavaScript as in Java. *Matrices*, similar to Java, are essentially arrays of arrays, offering a two-dimensional structure for data organization.

3.3.3 Loops and Conditional Statements

In JavaScript, like other contemporary languages, loops are implemented using the keywords `for` and `while`. Can also be used `forEach` that calls a function for each element of an array, but can't be executed for empty elements. We also have the `map()` and `reduce()` that are array methods that both iterate over each element but with different purposes. In one hand `map()` applies a callback to each item and returns a new array of transformed values, functioning like a loop specifically for data transformation, in the other hand `reduce()` also iterates through each element but accumulates the results into a single value (e.g., summing numbers or building an object), replacing more manual approaches such as a `for` loop with an accumulator variable.

Conditional statements employ `if`, `else` and `else if` keywords to replicate the `elif` functionality found in Python. This allows for the creation of structured and versatile control flow in JavaScript programs.

3.3.4 Functions

In JavaScript, functions are treated as first-class objects, providing them with the ability to be stored in variables, passed as arguments, returned from functions and even constructed dynamically at runtime, being able to have any number of arguments, regardless of what the function declaration specifies. Functions and *methods* are very similar, but a function is a standalone entities, an independent block of reusable code, while a method is a function associated with an object or class.

Anonymous functions, constructed using the `function(){ ... }` syntax, play a significant role in JavaScript. They are defined and executed without ever being assigned to a variable and are capable of taking arguments and returning values, enhancing the language's versatility. Creating *closures* (more in section 3.9) is a notable application of anonymous functions, allowing the establishment of a protected variable space through nested functions. This approach leverages JavaScript's function-level scope and lexical scoping, contributing to the creation of secure and encapsulated variable environments.

```
1 // Named function
2 function add(a, b) {
3     return a + b;
4 }
5
6 // Anonymous function stored in a variable
7 var multiply = function(a, b) {
8     return a * b;
9 };
```

```
console.log(multiply(2, 3)); // output: 6

// Immediately Invoked Function Expression (IIFE)
(function() {
    console.log('This is an IIFE');
})();

// Function with dynamic number of arguments
function sum() {
    var total = 0;
    for (var i = 0; i < arguments.length; i++) {
        total += arguments[i];
    }
    return total;
}
```

Listing 3.2: Example of different ways to implement functions

These examples from Listing 3.2 illustrate the flexibility and power of functions in JavaScript. They can be named or anonymous, stored in variables, immediately invoked function expression (IIFE) and accept a dynamic number of arguments.

3.4 JavaScript's Object-Oriented features

3.4.1 Classes

JavaScript's OO capabilities are rooted in its prototype-based system [24][48]. The language does not have a traditional *class-based inheritance* system, but it does offer a class-like syntax for creating objects. This syntax, introduced in *ECMAScript 6*, provides a more familiar and structured approach to object creation, allowing developers to define classes and create objects using the `class` keyword and also like in other languages, JavaScript classes can have *constructors* (just one per class 3.4.2), methods, properties and self-variables using the `this` keyword [7].

```
// Class definition
class Car {
    constructor(brand, model, year) {
        this.brand = brand;
        this.model = model;
        this.year = year;
    }
    displayCar() {
        return this.brand + ' ' + this.model + ', ' + this.year;
    }
}
// Creating an object
let myCar = new Car('Toyota', 'Corolla', 2005);
// Using a method
```

```
15 console.log(myCar.displayCar());  // Outputs: "Toyota Corolla, 2005"
```

Listing 3.3: Example of a Class

3.4.2 Class Constructors

Constructors in JavaScript serve as special methods for the creation and initialization of objects within a class. JavaScript classes come with a built-in constructor method that is automatically invoked when a new object is instantiated. In the listing 3.5 is an example of a constructor in JavaScript:

```
1 class Car {
2   constructor(brand) {
3     this.carname = brand;
4   }
5 }
6 let mycar = new Car("Toyota");
```

Listing 3.4: Example of a simple car class

This method plays a crucial role in setting the initial state of the object or executing any necessary setup procedures. In the context of the Car class, an example is presented where the constructor takes a single parameter, brand. Upon creating a new object from the Car class, the constructor is supplied with the car's brand and subsequently, the carname property is assigned the provided value.

It's essential to understand that a class can only possess one method named constructor; attempting to include more than one will result in a `SyntaxError`.

In cases where a constructor method is not explicitly defined, a default constructor is automatically provided. This default constructor is empty and does not take any parameters. For base classes, the default constructor is:

```
constructor() {}
```

And for derived classes, the default constructor is:

```
constructor(...args) {
  super(...args);
}
```

The `super` keyword is used to get access properties and call functions of an object's parent or class's prototype, or if is needed to invoke a superclass's constructor [45]. It can be used to achieve two different things: as a "function call" or as "property lookup". It can be used inside a method or in a constructor, but remember that the `super` keyword is only allowed in derived classes and it must be called before the `this` keyword. Also, `super` cannot overwrite the value of a property in the parent class, only access it.

In the example shown in listing 3.5, the `super` keyword is incorrectly used. The super keyword is not a standalone variable; it is specifically used within class constructors or

methods to reference the prototype of the object's parent class. Attempting to use `super` as a direct reference, as shown in the example, results in a `SyntaxError`.

```
const child = {
    myParent() {
        console.log(super); // SyntaxError: 'super' keyword unexpected here
    },
};
```

Listing 3.5: Example of a wrong use of super()

3.4.3 Mutability of Objects

All objects in JavaScript are mutable, a characteristic that allows for unconventional techniques like assigning attributes to functions and the properties of objects can be freely modified, added, or removed [54].

The *mutability* of objects in JavaScript is closely tied to *introspection*, enabling the examination of objects at runtime and dynamic instantiation and execution of methods, a concept known as *reflection*. These features to emulate traditional OO features, leveraging object mutability and reflection (more about reflection in **??**).

Unlike statically-typed languages like C++ or Java, JavaScript permits the runtime modification of objects and classes. This flexibility, driven by object mutability and reflection, serves as a powerful tool, enabling *dynamic scripting* and unlocking features that are challenging in static languages.

However, the flip side of this mutability is the inability to definitively define a class with a fixed set of methods and ensure their preservation over time, leading to less frequent type checking in JavaScript. This mutability is a double-edged sword, which risks yielding unexpected behavior and bugs if not handled with care. Nonetheless, it also provides significant flexibility and power, empowering developers to create intricate and dynamic applications.

To tackle challenges associated with object mutability and dynamic features, JavaScript relies on techniques like duck typing (more in the section 2.1), offering solutions to maintain code integrity in the face of a mutable environment.

3.4.4 Abstract Classes

JavaScript lacks native support for *abstract classes* or methods, a feature present in languages like Java, TypeScript and Python. Despite this, developers can implement custom solutions to emulate abstract class behavior in JavaScript, such as utilizing mixin classes (see section 2.7). We'll explore mixin classes in more detail later in section 3.6.3.

3.4.5 Null References

JavaScript defines two special values: `undefined` and `null`. When a variable doesn't have a value, we call it `undefined`. It's like saying the variable doesn't have any specific

information or value assigned to it. Now, `null` in JavaScript means "nothing" or that something doesn't exist. But here's the tricky part: in JavaScript, when we check the type of `null` using `typeof`, it says it's an object. This might seem like a mistake in JavaScript, but it's just one of those odd things in the language. Despite that, we can actually use `null` to make an object empty, wiping out any information it had before. So, in a nutshell, `undefined` is for when there's no value and `null` is like saying there's nothing there, even though technically it's labeled as an object.

3.4.6 Function signature overloading

Function *overloading* is a feature in OOP where you can have different functions with the same name as long as they have different parameter lists. However, not all OOP languages support overloading in the same way.

For example, JavaScript doesn't directly support having multiple versions of a function with different jobs (like some other programming languages do, for example Java or C++). However, clever developers can achieve a similar trick by using if-statements or other conditions inside the function. This way, even though JavaScript doesn't have a ready-made way for function overloading, you can still make a function do different things.

3.4.7 Garbage Collection

The role of a *garbage collector* is to keep an eye on how memory is used and figure out when a chunk of allocated memory is no longer needed so that it can be reclaimed. It's kind of like a cleanup crew for your computer's memory. However, it's important to note that this cleanup process is not perfect because it's challenging to decide definitively if a specific piece of memory is still necessary.

In the world of JavaScript, the garbage collection system takes care of automatically freeing up memory that is no longer in use. This helps prevent memory leaks and ensures memory is used efficiently. The system identifies and removes objects that are no longer reachable, making room for new objects. The garbage collector runs on a schedule, scanning through the memory space to find and reclaim the memory occupied by unused objects. Very conveniently, developers don't have to worry about managing this memory cleanup manually, the garbage collector handles it behind the scenes, making the process transparent and hassle-free.

3.4.8 Static Properties, Methods and Object

Static class methods are a fundamental concept in OOP, exemplified by languages like Java. These methods are associated with the class itself rather than instances of the class, meaning they can only be invoked on the class and not on individual objects. You can call it directly on the class, but if you try to call it on an instance of the class JavaScript

will throw a `TypeError`. Static methods, marked by the `static` keyword, serve for a lot of purposes, such as creating utility functions or managing static properties like caches and fixed configurations.

In JavaScript, the paradigm of *static members* takes a slightly different form. Unlike traditional OO languages, static members in JavaScript operate at the class level rather than the instance level. They are declared using `closures`, that are functions that capture the local variables from its surrounding scope, allowing the function to access these variables even when it is executed outside that scope (see more in 3.9), enabling the creation of both public and private static attributes and methods.

Private static members, as a unique feature in JavaScript, are declared within the closure of the constructor function. This ensures they are stored in memory only once, improving efficiency and simplifying usage. While private static members can be accessed by private or privileged (that are a special type that bridges the gap between public and private) methods within the constructor, they can't access private instance attributes.

On the other hand, public static members are created directly off the constructor, functioning as a namespace for behaviors related to the class. These members are accessible without creating an instance of the class and are particularly useful for tasks independent of specific instances of the class.

JavaScript's feature of treating functions as objects allows for the addition of attributes and methods dynamically. This flexibility means that functions, including static methods, can have attributes and methods like any other object. This capability shows the versatility of static methods in JavaScript, making them useful for tasks related to the class as a whole, regardless of specific instance data.

3.5 Interfaces

In the realm of OO JavaScript programming, *interfaces* emerge as invaluable assets, following the principle of "Program for an interface, not an implementation" [16]. However, the absence of a native mechanism for crafting or executing interfaces in JavaScript presents a challenge. JavaScript doesn't have built-in functions to easily tell if one object has the exact same set of methods as another [12]. This makes it difficult to swap objects interchangeably. Nonetheless, due to JavaScript's inherent flexibility, solutions can be crafted to overcome these obstacles.

Benefits of Using Interfaces:

Interfaces in OO JavaScript fulfill several crucial roles:

- Firstly, they serve as documentation, clarifying which methods a class implements and thereby enhancing code readability while facilitating *Reusability*.

- Secondly, familiarity with an interface streamlines the utilization of any class that adheres to it, promoting the reuse of classes and encouraging modular design.

- Thirdly, interfaces play a pivotal role in stabilizing communication between disparate classes, mitigating integration issues and allowing for the precise specification of desired functionalities and operations. Additionally, interfaces foster collaboration in large-scale projects, as one programmer can define an interface that another programmer can implement in various ways, provided that the class maintains adherence to the interface standards.

- Finally, interfaces simplify testing and debugging, particularly in the context of a loosely typed language like JavaScript, by showing explicit error messages for type-mismatch errors and ensuring that modifications made to an interface are uniformly reflected across all implementing classes, improving code stability.

Drawbacks of Using Interfaces:
The challenges of using interfaces in JavaScript are varied:

- Firstly, JavaScript's flexibility and loose typing face limitations when interfaces are applied, as they enforce strict typing, restricting the language's flexibility.

- Secondly, the absence of native support for interfaces in JavaScript complicates the task of replicating similar functionalities from other languages, posing a significant hurdle.

- Additionally, implementing interfaces can lead to performance issues, especially with large interfaces or objects expected to implement many interfaces, due to the overhead of method invocation.

- Another challenge is the inability to force other programmers to follow created interfaces manually, unlike languages with built-in interface support where a compiler ensures adherence to the implemented interface. While strategies like coding conventions and helper classes can help mitigate this issue, achieving complete resolution remains difficult. Ultimately, effective interface usage depends on consensus among project members.

3.5.1 Emulating Interfaces with Comments

One of the simplest methods, but the least effective, to mimic an interface in JavaScript involves using comments. This requires the use of keywords such as `interface` and `implements` within the comments, but commenting them out to avoid syntax errors. The listing 3.6 is an example of emulating interfaces in JavaScript using comments.

```
1 /*
2 interface Composite {
3     function add(child);
4     function remove(child);
5     function getChild(index);
6 }
```

```
7  interface FormItem {
8  function save();
9  }
10 */
11 var CompositeForm = function(id, method, action) {
12     // implements Composite, FormItem
13 ...
14 };
15 // Implement the Composite interface.
16 CompositeForm.prototype.add = function(child) {
17 ...
18 };
19 CompositeForm.prototype.remove = function(child) {
20 ...
21 };
22 CompositeForm.prototype.getChild = function(index) {
23 ...
24 };
25 // Implement the FormItem interface.
26 CompositeForm.prototype.save = function() {
27 ...
28 };
```

Listing 3.6: Emulating interfaces in JavaScript using comments

This method doesn't truly provide interface functionality since it doesn't check if implementations are correct. There are no error messages to warn programmers about implementation problems; it's more like documentation than a working interface. However, it's easy to use and doesn't require extra classes or functions, making it reusable without affecting file size or speed. Additionally, comments used can be removed easily during deployment, preventing any increase in file size due to interfaces. But it doesn't help with testing and debugging since it doesn't check conditions and/or produce error messages.

3.5.2 Emulating Interfaces with Attribute Checking

This implementation involves in explicit interface declaration in classes, supported by checking objects interacting with these classes. The interfaces remain comments, but classes declare the interfaces they implement using an array attribute (in the listing 3.7: `implementsInterfaces`).

```
1  /*
2  interface Composite {
3      function add(child);
4      function remove(child);
5      function getChild(index);
6  }
7  interface FormItem {
8      function save();
```

```
}
*/
var CompositeForm = function(id, method, action) {
    this.implementsInterfaces = ['Composite', 'FormItem'];
    ...
};
function addForm(formInstance) {
    if(!implements(formInstance, 'Composite', 'FormItem')) {
        throw new Error("Object does not implement a required interface.");
    }
    ...
}
// The implements function, which checks to see if an object declares that it
// implements the required interfaces.
function implements(object) {
    for(var i = 1; i < arguments.length; i++) {
        // Looping through all arguments
        // after the first one.
        var interfaceName = arguments[i];
        var interfaceFound = false;
        for(var j = 0; j < object.implementsInterfaces.length; j++) {
            if(object.implementsInterfaces[j] == interfaceName) {
                interfaceFound = true;
                break;
            }
        }
        if(!interfaceFound) {
            return false; // An interface was not found.
        }
    }
    return true; // All interfaces were found.
}
```

Listing 3.7: Emulating interfaces using Attribute Checking

This method from listing 3.7 offers several benefits, including the documentation of implemented interfaces in classes and the generation of errors if a class fails to declare support for a required interface, encouraging other programmers to specify interfaces. However, it has drawbacks: it doesn't guarantee that a class truly implements the interface, only verifies if it claims to do so; this may lead to potential issues if a class forgets to add a required method despite declaring the interface. Moreover, explicitly declaring supported interfaces demands extra effort.

3.5.3 Emulating Interfaces with Duck Typing

Duck typing (see more at 2.2) can be used to emulate interfaces in JavaScript. This method involves the creation of an `Interface` class that checks if an object implements the required methods. The `ensureImplements` function verifies that the object implements the required methods, throwing an error if it doesn't.

```
1  // Interfaces.
2  var Composite = new Interface('Composite', ['add', 'remove', 'getChild']);
3  var FormItem = new Interface('FormItem', ['save']);
4  // CompositeForm class
5  var CompositeForm = function(id, method, action) {
6      ...
7  };
8  ...
9  function addForm(formInstance) {
10     ensureImplements(formInstance, Composite, FormItem);
11     // This function will throw an error if a required method is not
           implemented.
12     ...
13 }
```

Listing 3.8: Emulating interfaces using Duck Typing

In the listing 3.8 `Interface.ensureImplements` function checks if the given object implements the methods declared in the specified interfaces. This method is more effective than the previous ones, as it verifies that the object truly implements the interface, not just claims to do so. However, classes do not declare the interfaces they implement, reducing both code reusability and self-documentation. It necessitates the use of a helper class (`Interface`) and a helper function (`ensureImplements`). It falls short in verifying the names, numbers, or types of arguments in methods, only focusing on ensuring the accuracy of method names.

3.6 Inheritance

JavaScript's approach to inheritance stands out for its complexity when compared to many other OO languages. Unlike those languages where inheritance is facilitated by a simple keyword, JavaScript necessitates a series of steps to transmit public members similarly. This complexity appears from JavaScript's utilization of *prototypal inheritance* [8][9], a distinctive feature offering flexibility in selecting between standard class-based inheritance and prototypal inheritance. While prototypal inheritance is potentially more efficient, it might pose slight challenges in its application. Within this section, various techniques for crafting subclasses in JavaScript are explored, offering insights into scenarios where each method proves suitable.

Inheritance serves as a crucial tool in class design, reducing code redundancy and promote loose coupling between objects. Inheritance achieves the design principles of minimizing code duplication and enhancing flexibility by building upon existing classes and leveraging their methods. Also, inheritance allows for the extension of existing classes and the utilization of their methods. This approach simplifies the implementation of changes; for instance, if multiple classes necessitate a `toString` method, replicating it across each class would require updates in multiple locations for any modifications [4]. Instead of duplicating code, the creation of a shared class (for example `ToStringProvider`)

and inheriting from it consolidates the method declaration in one location, simplifying maintenance. However, inheritance can introduce strong coupling, where one class becomes reliant on the internal workings of another. To mitigate this issue, techniques like mixin classes are explored, offering methods to other classes without direct inheritance and addressing the challenge of strong coupling.

3.6.1 Classical Inheritance

JavaScript can emulate *classical inheritance* by using functions to declare classes and the `new` keyword to create instances of these classes. A basic class declaration in JavaScript is illustrated with the example of a `Person` class in the listing 3.9:

```
1 /* Class Person. */
2 function Person(name) {
3     this.name = name;
4 }
5 Person.prototype.getName = function() {
6     return this.name;
7 }
8 var reader = new Person('Jaime Brito');
9 reader.getName();
```

Listing 3.9: Example of a class for a Person

The constructor is created within the function, following the convention of initializing the class name with a capital letter. Instance attributes are defined using the `this` keyword within the constructor, while methods are added to the class's prototype object, such as `Person.prototype.getName`. To instantiate the class, the constructor is invoked with the 'new' keyword (as exemplified in the line 8 from the listing 3.9). Subsequently, all instance attributes and methods become accessible for the created instance.

3.6.1.1 Prototype Chain

Object prototypes are the mechanism through which JavaScript objects inherit features from one another [33]. In JavaScript, each object has a connection with a prototype object, which serves as a template for the object's properties and methods [21]. To create a class that inherits from the example above `Person` class, the `Author` class is introduced in the listing 3.10:

```
1 /* Class Author. */
2 function Author(name, books) {
3     Person.call(this, name); // Call the superclass's constructor in the scope
          of this.
4     this.books = books; // Add an attribute to Author.
5     }
6 Author.prototype = new Person(); // Set up the prototype chain.
7 Author.prototype.constructor = Author; // Set the constructor attribute to
      Author.
```

```
8  Author.prototype.getBooks = function() { // Add a method to Author.
9      return this.books;
10 };
11 // Create an instance of Author.
12 var author = [];
13 author[0] = new Author('J. K. Rowling', ['Harry Potter']);
14 author[1] = new Author('George Orwell', ['Nineteen Eighty-Four']);
15 author[1].getName();
16 author[1].getBooks();
```

Listing 3.10: Example of inheritance between the class Person and Author through constructor functions and the prototype chain.

The constructor function is created similarly to the previous example (Listing 3.9). Within it, the superclass's constructor (`Person.call(this, name)`) is invoked, manually passing the empty object generated by the `new` operator. The *prototype chain* is set up by assigning `Author.prototype` to a fresh instance of `Person` (Listing 3.10 line 6) and subsequently, the constructor attribute is reassigned to `Author` (Listing 3.10 line 7) to prevent its overwrite. Despite the additional setup lines for inheritance, the process of instantiating subclasses mirrors that of the superclass.

Demonstrating this, instances of the `Author` class are generated in the example, showcasing the usage of inherited methods, in this particular example is used an array that convenient allows for organization and to access multiple instances of the `Author` class. The complexity of classical inheritance primarily lies in the class declaration, while the creation of new instances remains straightforward.

Even through ECMAScript 6 introduced the `class` syntax, which is a *Syntactic sugar* for prototypes, JavaScript still uses prototypes under the hood. The class syntax in JavaScript is more about convenient syntax than about changing the fundamental prototype-based system

3.6.2 Prototypal Inheritance

In prototypal inheritance, the traditional method of defining a class structure through a class declaration is substituted by creating an object, which serves as the prototype object for subsequent objects. This prototype object contains default attributes and methods that are inherited by other objects [21].

```
1  /* Person Prototype Object. */
2  var Person = {
3      name: 'default name',
4      getName: function() {
5      return this.name;
6      }
7  };
8  var reader = clone(Person);
9  alert(reader.getName()); // This will output 'default name'.
10 reader.name = 'Jaime Brito';
```

```
11 alert(reader.getName()); // This will now output 'Jaime Brito'.
12
13 /* Author Prototype Object. */
14 var Author = clone(Person);
15 Author.books = []; // Default value.
16 Author.getBooks = function() {
17     return this.books;
18 }
19 var author = [];
20 author[0] = clone(Author);
21 author[0].name = 'J. K. Rowling';
22 author[0].books = ['Harry Potter'];
23 author[1] = clone(Author);
24 author[1].name = 'George Orwell';
25 author[1].books = ['Nineteen Eighty-Four'];
26 author[1].getName();
27 author[1].getBooks();
```

Listing 3.11: Example of Prototypal Inheritance

The `Person` prototype object is established as an object literal instead of utilizing a constructor function, that serves as *prototype object* for any other `Person`-like object that will be created, incorporating default values for both attributes and methods. In order to generate a new object with the prototype of `Person`, the `clone` function is employed. This function provides an empty object with the prototype attribute set to the prototype object. With the integration of supplementary attributes and methods, or the overriding of existing ones. Afterwards, the `Author` prototype object is replicated to produce new objects with `Author`-like characteristics, providing the opportunity for customization of their attributes. This method, illustrated by `var author = [];`, facilitates a dynamic and adaptable approach to object creation, enabling the inheritance of behavior from prototype objects.

3.6.3 Emulating inheritance with Mixin Classes

In this section we explore an example of mixin class in JavaScript. To understand more about mixins classes see section 2.7.

```
1  var Mixin = function() {};
2  Mixin.prototype = {
3      serialize: function() {
4      var output = [];
5      for(key in this) {
6          output.push(key + ': ' + this[key]);
7      }
8      return output.join(', ');
9      }
10 };
11 augment(Author, Mixin);
12 var author = new Author('George Orwell', ['Nineteen Eighty-Four']);
```

```
13 var serializedString = author.serialize();
```

Listing 3.12: Example of Mixin Classes to emulate inheritance

In the Listing 3.12, the `Mixin` class contains a single method called `serialize` which converts each member of the class into a string. While this method could be useful in various classes, directly inheriting from Mixin for each class or duplicating the code in every class isn't practical. Instead, a function called 'augment' is used to add the serialize method to the classes that need it.

For instance, the `Author` class is augmented with all methods from the Mixin class, allowing instances of Author to call the serialize method. This technique mimics multiple inheritance in JavaScript. In JavaScript, a class can't inherit from more than one superclass due to the limitation of the prototype attribute, but augmenting a class with multiple mixin classes provides similar functionality.

3.6.4 When should Inheritance be used?

In JavaScript, inheritance introduces complexity so, it should be employed prudently, reserved for situations where the benefits, such as code reuse, outweigh the drawbacks. Inheritance allows methods to be defined once, reducing redundancy and aiding in maintenance and debugging.

Two main paradigms exist: prototypal inheritance and classical inheritance. Prototypal inheritance is efficient in terms of memory and is suitable for scenarios where memory efficiency is crucial. Classical inheritance is preferable for developers familiar with inheritance in other OO languages. Both are effective for hierarchies with subtle class differences. However, for classes with significant differences, augmenting them with methods from mixin classes is a more sensible approach.

For simpler JavaScript programs, such abstraction levels are often unnecessary. They become essential in larger projects involving multiple developers.

3.7 Object Delegation

In OOP, *delegation* refers to the evaluation of a member (property or method) of one object, known as the receiver, within the context of another original object, named the sender, in other words delegation is Implementing the prototype approach to sharing knowledge in OO systems [24][48][8][9]. There are two types of delegations: One, that involves transferring the responsibilities of the sending object to the receiving one, achievable through method invocation or property access on another object. Another, that operates based on the member lookup rules of the language, facilitating the reuse of behaviour without explicit processing of the responsibilities.

In JavaScript, *object delegation* is closely linked to prototypes, operating through a mechanism known as the prototype chain (see more at 3.6.1.1). Each object in JavaScript, except the root object, is associated with a prototype. When accessing a property or

method on an object, JavaScript traverses the prototype chain to locate it. If the property or method is not found on the object itself, JavaScript extends the search to its prototype and continues upward along the chain. This process entails delegating behavior: when invoking a method or accessing a property on an object, JavaScript implicitly delegates the lookup to its prototype. If the prototype does not possess the property or method, the search persists along the chain until it reaches the root object(usually `Object.prototype`).

Delegation in JavaScript is a powerful tool, enabling the sharing of behavior among objects and facilitating the creation of complex systems. By leveraging the prototype chain, developers can establish relationships between objects, allowing them to inherit properties and methods from other objects. This approach promotes code reuse, simplifies maintenance and enhances the flexibility of JavaScript programs.

```
1  // Parent object
2  const parent = {
3      greet: function() {
4        return "Hello," + this.name + "!";
5      }
6  };
7  // Child object that delegates to the parent object
8  const child = Object.create(parent);
9  // Set the name property on the child object
10 child.name = "Jaime";
11 // Delegate the greet method to the parent object
12 child.sayHello = function() {
13     return this.greet()
14 };
15 console.log(child.sayHello()); // Output: "Hello, Jaime!"
```

Listing 3.13: Example of Object Delegation

In the example from Listing 3.13 we have a `parent` object with a `greet` method that returns a greeting message. The child object is created using `Object.create()`, which sets up the delegation. That means the `child` inherits properties and methods from `parent`. The child object then sets its name property and delegates the greet method to the parent object by defining a `sayHello` method that calls the greet method. When the `child.sayHello()` method is called on the child object, it first looks to the `greet` method within itself. Since it doesn't find it, delegates the method lookup to its prototype, which in this case is the `parent` object, resulting in `this.greet` in the `sayHello` method effectively calls the `parent.greet()`, allowing the child object to inherit behaviour from its parent.

Important to note that, if the `greet()` method is changed in the `parent` object, the `child` object will inherit the changes, because the `child` object delegates the `greet()` method from the `parent` object. To prevent this from happening it is necessary that the method `greet()` be overwriten by the `child` object (for example by doing `child.greet(...)`). In this way, the `greet()` method of the `parent` would stay the same but in the `child` would change.

3.7.1 Delegation vs Inheritance

Object delegation and inheritance represent distinct approaches to structuring and sharing behavior in programming languages, each with its conceptual differences and mechanisms. Object delegation focuses on behavior sharing and method lookup, with objects delegating property/method lookups to their prototypes, forming a chain where each object inherits from its prototype. This process allows for dynamic and runtime-based behavior resolution, enabling behavior extension without modifying existing objects. Mechanically, objects in delegation have an associated prototype and property/method lookup occurs by travelling across the prototype chain.

By contrast, inheritance revolves around class-based or constructor-based inheritance, where classes define blueprints for creating objects and subclasses inherit properties and methods from their parent class. This relationship is static and defined during class creation, facilitating a static and compile-time relationship between classes and subclasses. Object delegation is ideal for composition, mixins and behavior reuse, enabling multiple inheritance via prototype chaining and it is commonly employed in prototype-based languages like JavaScript. Inheritance, on the other hand, is useful for defining hierarchies and relationships, providing a clear 'is-a' relationship (which means one class inherits another class) and is commonly used in class-based languages.

3.8 Dispatch in JavaScript

In JavaScript, the concepts of Single Dispatch, Double Dispatch and Multiple Dispatch (more about dispatch in section 2.5) are treated differently due to the language's dynamic nature and the lack of native support of these feature:

1. **Single Dispatch:** JavaScript natively supports Single Dispatch. Single Dispatch is facilitated by its prototype-based inheritance model. When a method is invoked on an object, JavaScript searches for the method within the object's prototype chain until it locates it or reaches the end of the chain (more in 3.6.1.1). This process allows JavaScript to resolve method calls based on the prototype chain, enabling dynamic method lookup and invocation [32].

2. **Double Dispatch:** In JavaScript, Double Dispatch lacks native support, yet it can be implemented through alternative methods such as `instanceof` or `typeof` to ascertain the type of argument. Alternatively, design patterns like the Visitor pattern offer a structured approach to achieve Double Dispatch functionality [31]. These techniques enable the dynamic resolution of method calls based on both the receiver and the argument types, simulating the behavior of Double Dispatch in JavaScript's dynamic environment [13].

3. **Multiple Dispatch:** In JavaScript, Multiple Dispatch lacks native support, necessitating the use of alternative techniques to simulate its functionality. This can

involve checking the types of multiple arguments to determine method resolution. However, this approach can swiftly become complex and challenging to manage due to JavaScript's dynamic nature and lack of built-in support for multiple dispatch. As a result, while it is possible to emulate multiple dispatch in JavaScript through a combination of techniques, it may lead to code that is difficult to maintain and understand [32][22].

It's crucial to remember that JavaScript is a dynamically typed language, meaning that method resolution occurs at runtime. In JavaScript, the actual types of the arguments involved in a method call can influence which method is invoked. This dynamic behavior contrasts with statically typed languages, where method resolution typically occurs at compile time, based on the declared types of variables.

3.9 Closures and Scopes

In JavaScript, only functions have scopes, that means a variable declared within the function is not reachable outside the function. Clsses by itself do not create a new scope in JavaScript. Instead, their methods are added to the prototype of the constructor function. Vareables declared within class methods are scoped to the function (or method) in which they are defined. Instances of a classes have their own properties and methods, but these do not affect variable scoping rules. Private attributes are essentially variables that you would like to be inaccessible from outside the object, so it makes sense to look to this concept of scope to achieve that inaccessibility. Variables defined within a function are reachable by its nested functions.

```
1 function foo() {
2     var a = 10;
3     function bar() {
4         a *= 2;
5     }
6 bar();
7 return a;
8 }
```

Listing 3.14: Example of Scope in JavaScript

In Listing 3.14, variable `a` is declared within the function `foo`, but function `bar` can access it because it's nested within `foo`. When `bar` runs, it modifies `a` by doubling its value. While it's reasonable for `bar` to access `a` within `foo`, what happens if you execute `bar` outside of `foo`?

```
1 function foo() {
2     var a = 10;
3     function bar() {
4         a *= 2;
5         return a;
```

```
6       }
7       return bar;
8   }
9   var baz = foo(); // baz is now a reference to function bar.
10  baz(); // returns 20.
11  baz(); // returns 40.
12  baz(); // returns 80.
13  var blat = foo(); // blat is another reference to bar.
14  blat(); // returns 20, because a new copy of a is being used.
```

Listing 3.15: Example of Closure in JavaScript

The function `bar`, from the listing 3.15, is returned and assigned to the variable `baz`. Even though `bar` is executed outside of `foo`, it retains access to variable `a`. JavaScript's lexical scoping enables functions to run in the scope they're defined in, not where they're executed. As long as `bar` is within `foo`, it can access all of `foo` variables, even after `foo` finishes executing.

This scenario illustrates a closure, where the scope of `foo` is retained after it returns, accessible only to the functions it returns. `baz` and `blat` each have their own copy of this scope and `a`, which they can modify independently. Closures are commonly created by returning nested functions.

3.10 Modules and Namespaces

As JavaScript applications grow in complexity, organizing code becomes increasingly important. Without a structured approach, large applications can become difficult to maintain, especially when multiple scripts are loaded. Two common techniques for structuring JavaScript code are *namespaces* and *modules*.

A *global namespace* refers to the top-level scope where all variables and functions are accessible throughout the program. If everything is defined in the global scope, conflicts can arise when different parts of an application use the same variable or function names. To prevent such issues developers group related functionality within objects, effectively creating *namespaces* that isolate different components of a program.

A *module* is a self-contained unit of code that exposes only what is necessary while keeping internal logic private. Unlike namespaces, which rely on a single global object, modules allow explicit imports and exports, ensuring that only required components are shared.

JavaScript originally lacked built-in support for modules, leading to custom solutions like manually defined namespaces. However, modern JavaScript includes native ES6 modules, which provide a cleaner and more structured way to organize code.

3.10.1 Modules using namespaces

To start we need to create an object and attach to a global namespace. This object will contain the root namescape and so our namescape will have the name of `Westeros`.

```
1 var Westeros = Westeros || {};
2
3 Westeros.Families = Westeros.Families || {};
4 Westeros.Families.Stark = function() {
5     this.name = "Stark";};
```

Listing 3.16: Basic Namespace

In the listing 3.16 the `Westeros` object acts as a namespace, containing a `Families` module. This prevents adding multiple global variables and keeps related code grouped together. In the line 1 we can check if the object already exists and use that version instead of reassigning the variable. This allows you to spread your definitions over a number of files.

Of course with JavaScript there is more than one way to build the same code structure. An easy way to structure the preceding code is to make use of the ability to create and immediately execute a function (IIFE) ensuring that the namespace is initialized only once and prevent accidental global modifications.

```
1 var Westeros;
2 (function (Westeros) {
3     (function (Families) {
4         Families.Stark = function () {
5             this.name = "Stark";
6         };
7     })(Westeros.Families || (Westeros.Families = {}));
8 })(Westeros || (Westeros = {}));
```

Listing 3.17: Namespace with IIFE

Using the listing 3.17 ensures that `Westeros` and its submodules exist before any properties are added, avoiding unintended overwrites.

This way of create modules is the way that Simon Timmms decides to use in his book [46] and the way we used to show the examples in the section about design patterns (see more in 5).

3.10.2 Modules using syntactic sugar

In the ES6 brings support for some syntactic sugar for making classes, also also brings a well thought out module system for JavaScript. There is also syntactic sugar for creating modules which looks like listing 3.18.

```
1 module 'Westeros' {
2     export function Rule(rulerName, house) {
3         return "Long live " + rulerName + " of house " + house;
4     }
```

```
5 }
```

Listing 3.18: Defining a JavaScript Module Using ES6 Export

From the listing 3.18 we can see how to define a module using ES6's `export` statement to encapsulate a function. Instead of relying on a global object, this module defines a `Rule` function and explicitly exports it. As modules can contain functions they can, of course, contain classes. ES6 also defines a module import syntax and support for retrieving modules from remote locations.

```
1 import westeros from 'Westeros';
2 westeros.Rule("Rob Stark", "Stark");
```

Listing 3.19: Importing an ES6 Module

The listing 3.19 demonstrates how to import and use an ES6 module, ensuring clean separation of concerns. Unlike namespaces, ES6 modules are file-scoped and must be explicitly imported, eliminating the risk of global variable conflicts.

3.11 Reflection

In JavaScript, an object possesses the capability to introspect itself, enabling the listing and modification of its properties and methods, a process known as *reflection* or *self-representation* [28][51]. This ability allows a JavaScript object to examine its own structure, incorporating structures that represent itself [40]. The `Reflect` namespace object provides static methods for invoking interceptable internal methods of JavaScript objects. Reflection, in this context, means examining the structure of a program and its data. The programming language (PL) of the program and the language (MPL) in which the examination occurs, known as the meta programming language, can be the same or different, in the context of JavaScript they yield similar results. JavaScript offers various mechanisms for inspecting its own program [29].

Why is Reflection important?

The `Reflect` object strengthen JavaScript's capabilities by facilitating more efficient object manipulation and simplifying function invocation, therefore empowering the creation of dynamic and potent applications. Examples of its utility span various domains, including performance monitoring, debugging, interfacing, control reasoning, self-optimization, self-modification and self-activation. By extending JavaScript's functionality, the Reflect object grant methods for modifying language operations and internal object state, thereby enabling more expressive and powerful code. These built-in methods, such as `Reflect.get()`, `Reflect.set()`, `Reflect.apply()` and `Reflect.construct()`, provide path for accessing and modifying internal object states, each serving distinct purposes in manipulating object internals [23]. In the table 3.1 shows the most common Reflection methods presents in JavaScript.

Table 3.1: Some Reflection Methods from JavaScript

Method	**Definition**
Reflect.get()	Retrieves the value of a property from an object, with an optional receiver to control getter behavior.
Reflect.set()	Sets a property on an object, returning a Boolean indicating success; supports a custom receiver for setter context.
Reflect.apply()	Invokes a function with a specified `this` value and arguments.
Reflect.construct()	Creates a new object instance using a constructor and arguments (like the `new` operator), optionally specifying a different prototype.
Reflect.deleteProperty()	Deletes a property from an object, functioning like the `delete` operator but as a function call.
Reflect.getPrototypeOf() Object.getPrototypeOf()	Returns the prototype of a specified object.
Reflect.has()	Checks whether a property exists on an object (including in its prototype chain), like the `in` operator.
Reflect.isExtensible() Object.isExtensible()	Determines if new properties can be added to an object.
Reflect.ownKeys()	Retrieves an array of the target object's own property keys from an object.
Reflect.preventExtensions() Object.preventExtensions()	Prevents any new properties from being added to an object and returns a Boolean indicating success.
Reflect.setPrototypeOf() Object.setPrototypeOf()	Sets the prototype of an object, returning a Boolean indicating success.
hasOwnProperty()	Checks if a property exists directly on the object (not inherited), returning true if it does.
Object.getOwnPropertyNames()	Returns an array of all string-keyed properties found directly on an object, including non-enumerable ones (but excluding symbols).
Object.getOwnPropertyDescriptor()	Retrieves a property's descriptor (value, writability, enumerability, configurability, or getter/setter functions) for an own property of the object.
Object.defineProperty()	Defines or modifies a property on an object with specific descriptor settings (writable, enumerable, configurable, etc.) , returning the modified object.
Array.isArray()	Determines whether a given value is an array, returning true if it is and false otherwise.
Object.keys()	Returns an array of an object's own enumerable string-keyed property names, similar to iterating with `for...in`.

4

Study Setup

This chapter outlines the methodology adopted for evaluating the design pattern implementations in JavaScript. The study is structured to compare different approaches and to analyze the practical implications of dynamic design choices. This overview sets the stage for the in-depth discussions on terminology (Section 4.1) and detailed study design (Section 4.2).

4.1 Terminology

We use the term *scenario* to refer to the conceptual idea or metaphor that sets up the context and relationships of the classes involved in a design pattern example. For instance, Simon Timms's [46] scenario for the prototype pattern is based on the idea that we have a family in Westeros where each generation is born by copying and modifying an existing family member. The scenario helps to understand the context and relationships between the classes involved in the pattern. We used the term *example* to represent a specific implementation of the scenario for a pattern. In some way we can have different examples for the same scenario, one example of that is the updated version of the prototype pattern using object delegation (more in 5.2).

In classical object-oriented languages (like Java), design patterns typically define *roles* by prescribing certain classes or interfaces that fulfill specific responsibilities (for example *Observer* defines the observer and subject roles or the *Decorator* defines component and decorator). In JavaScript, which is more dynamic and not strictly class-based, these roles are fulfilled by any construct, such as objects, modules, or functions, that implements the patterns's expected behavior.

Because JavaScript is a dynamic and weakly typed languag, it is the behavior and structure of these objects or modules that determines whether they correctly serve as *participants* in the pattern (also referred to as *pattern roles*). This term *participant* or *pattern role* clarifies how each piece of the example interacts with others and what responsibilities it has, without tying the concept strictly to classes or classical inheritance.

4.2 Study Design

For this thesis, was used two complete sets of design patterns scenarios. The first set is from the book by Simon Timms [46], which provides scenarios for 23 design patterns. The second set is from the book by Harmes and Diaz [11], which provides scenarios for 10 design patterns. We also used scenarios from the book by Den Odell [35] and from the repository by Felipe Beline [10]. The distribution of the pattern scenarios used in this thesis is shown in Table 4.1.

Author	**Number of scenarios used**	**Source**
Simon Timms	23 (All 23)	*Mastering Design Patterns [46]*
Harmes and Diaz	10 (*Bridge, Chain of Responsibility, Command, Composite, Decorator, Facade, Factory Method, Flyweight, Observer, Proxy*)	*Pro JavaScript Design Patterns [11]*
Den Odell	8 (*Abstract Factory, Adapter, Builder, Iterator, Mediator, Memento, Prototype, Singleton*)	*Pro JavaScript Development [35]*
Felipe Beline	5 (*Interpreter, State, Strategy*) *Template Method, Visitor*	*GitHub repository [10]*

Table 4.1: Pattern scenario source distribution for JavaScript design patterns

We prioritize the usage of the implementation from the books because, in general, the books give the description of the scenario and the exaplantion for the implementation of the pattern, helping to understand the pattern itself. The repository by Felipe Beline was used to fill a gap in the scenarios from the books.

To ensure the reliability and comprehensiveness of the results, we used two distinct sets of implementations for the same design pattern. By analyzing more than one collection reduces the risk of the conclusions being skewed by special or atypical cases in a single dataset. This approach increases the reliability, as it allows me to confirm that observed behaviors and design trade-offs are consistent across different contexts. Also the use of two sets helps to verify some modularity properties, for example, allows to verify if some piece of code is reusable by taking from one scenario and try in the other scenario from the other set, allowing to analyse the *Reusability* property.

Previous studies on design patterns and modularity properties have generally focused on comparing implementations between two or more different programming languages. In contrast, this thesis compares two different styles or approaches to implementing design patterns in JavaScript.

All the examples from the chapter 5 are from the book by Simon Timms [46] so that we can keep consistency between all the examples. This ensures that the scenarios and examples are aligned in terms of context and explanation, providing a coherent basis for analysis and discussion throughout the chapter.

Mechanism	Simon Timms	Harmes & Diaz	Den Odell	Felipe Beline
Classical Inheritance	Yes	Yes	Yes	No
Prototype-based Inheritance	No	Yes*	Yes*	Yes*
Functions	Yes	Yes	Yes	Yes
Object Delegation	No	No	No	No
Mixins	No	No	No	No
Reflection	No	No	No	No
Duck Typing	No	Yes*	Yes*	Yes*

Table 4.2: Mechanisms used across different sources. The use of "yes*" in Prototype-based Inheritance and Duck Typing indicates that these mechanisms are not used universally across all design pattern implementations within each source. Instead, they are selectively applied in specific cases where they provide particular benefits.

Authors Mechanisms	Updated versions Mechanims
Interfaces (Emulated)	Duck Typing
Classical Inheritance	Prototype-Based Inheritance
Rigid Class Hierarchies	Mixins and Object delegation
Manual property checks	Reflection
Old JS syntax	Up to date syntax

Table 4.3: Side-by-side comparison of the mechanisms used on the original implementation and updated implementation

In the table 4.3 we can see the side-by-side comparison of the mechanisms used on the original implementation and what mechanisms were used instead in the updated version. The original implementation used a tradicional (or classical) implementation of the design patterns, while the updated version used a dynamic aproach (and advanced mechanisms) to implement the design patterns that fits better with the JavaScript language.

For the modularity analysis, is need to identify example by example three things: the code of the pattern implementation, the case-specific participant code (usually is the example of usage) and the *glue code* (the code that connects the participants with the pattern code). The code of the pattern implementation is the code that implements the pattern itself, the participant code is the code of the case-specific scenario that uses the pattern. The modularity analysis is done by analyzing the code of the pattern implementation. A pattern that is modular is when the code of the pattern is isolated in a separate module from the participant code and the glue code.

5

Design Patterns

In this chapter, we present the concept of design patterns and their importance in software development. We also discuss the *Prototype*, *Observer*, *Abstract Factory* and *Composite* design patterns, providing an overview of their implementation in JavaScript because these patterns are the best ones to show the use of the avanced mechanisms in the design patterns.

5.1 Design Patterns

The term "pattern" was introduced by Alexander [1][2] in 1977. It explores architecture and urban design, proposing 253 patterns or design ideas for creating functional spaces. Initially used in the field of architecture, the concept has been adapted to software engineering as design patterns. A design pattern is a general, reusable solution to a commonly occurring problem within a specific context in software design. Though design patterns are not part of the source code or libraries in many programming languages, their descriptions can solve numerous design problems encountered in coding. The main goal of design patterns is to facilitate the rapid transfer of knowledge from experienced designers to less experienced programmers. They also promote efficient communication among programmers, allowing them to discuss design problems using a common technical vocabulary.

Inspired by the work of Christopher Alexander and other authors, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides wrote the book 'Design Patterns: Elements of Reusable OO Software'[16]. This book has been so influential that it is commonly referred to as the GoF book or *Gang of Four book*, named after its four authors. The book outlines 23 patterns for use in OO design, which are divided into three major groups:

Category	Definition	Pattern
Creational	*These patterns provide various ways to create objects, abstracting the instantiation process to enhance flexibility and reuse.*	Abstract Factory Builder Factory Method Prototype Singleton
Structural	*These patterns focus on composing classes and objects into larger structures, optimizing the organization and relationships between entities.*	Adapter Bridge Composite Decorator Facade Flyweight Proxy
Behavioral	*These patterns deal with object interaction and responsibility distribution, defining how objects communicate and collaborate to perform tasks.*	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

Table 5.1: GoF Design Patterns, their Categories and Definitions

5.2 Prototype

The *Prototype* pattern probably sounds familiar because is the mechanism through which inheritance in JavaScript is supported. In terms of JavaScript we look to prototypes for inheritance, but it is not only limited to inheritance. The *Prototype* pattern suggest that creates objects based on a template of an existing object through cloning. Copying an existing object can be a useful technique when the cost of creating a new object is expensive.

5.2.1 Prototype implementation

The *Prototype* pattern have a scenario where in Westeros, we find that members of a family are frequently very similar; as the adage goes: "like father, like son". As each generation is born it is easier to create the new generation through copying and modifying an existing family member than to build one from scratch.

```
1 var Westeros;
2 (function (Westeros) {
3     (function (Families) {
4         var Lannister = (function () {
```

```
5             class Lannister {
6                 clone() {
7                     var clone = new Lannister();
8                     for (var attr in this) {
9                         clone[attr] = this[attr];
10                    }
11                    return clone;
12                }
13            }
14            return Lannister;
15        })();
16        Families.Lannister = Lannister;
17    })(Westeros.Families || (Westeros.Families = {}));
18    var Families = Westeros.Families;
19})(Westeros || (Westeros = {}));
20var jamie = new Westeros.Families.Lannister();
21jamie.swordSkills = 9;
22jamie.charm = 6;
23jamie.wealth = 10;
24var tyrion = jamie.clone();
25tyrion.charm = 11;
26tyrion.defense = 4;
27console.log(tyrion.charm); // 11
28console.log(tyrion.wealth); // 6
29console.log(tyrion.swordSkills);// 9
30console.log(tyrion.defense) // 4
```

Listing 5.1: Implementation of the prototype pattern

The *Prototype* pattern enables the creation of a complex object just once, which can then be cloned into multiple slightly different objects. If the original object is simple, cloning offers minimal benefit. When using this pattern, it's important to consider any dependent objects carefully.

However, since 2009 with the introduction of ECMAScript 5, the `Object.create()` method was introduced, which allows to create an object with a specified prototype and optionally contains specified properties (for example: `Object.create(prototype, optionalDescripton)`).

The example from the listing 5.2 illustrates the use of the `Object.create()` method to modify the example of the Listing 5.1 using object delegation (more in 3.7) instead of a `clone` method.

```
1 // Define the namespace and class structure
2 var Westeros = Westeros || {};
3 Westeros.Families = Westeros.Families || {};
4
5 class Lannister {
6     constructor(charm = 0, wealth = 0, swordSkills = 0) {
7         this.charm = charm;
8         this.wealth = wealth;
```

```
9           this.swordSkills = swordSkills;
10      }
11  }
12
13  Westeros.Families.Lannister = Lannister;
14
15  // Create an instance with default properties
16  const jamie = new Westeros.Families.Lannister(6, 10, 9);
17
18  // Use Object.create to delegate properties from Jamie and add/override
        specific ones
19  const tyrion = Object.create(jamie);
20  tyrion.charm = 11; // Overrides the charm property
21  tyrion.defense = 4; // New property unique to Tyrion
22
23  // Outputs demonstrating prototype linkage and property shadowing
24  console.log("Tyrion's Charm: ", tyrion.charm); // 11 (overridden)
25  console.log("Tyrion's Wealth: ", tyrion.wealth); // 10 (delegated)
26  console.log("Tyrion's Sword Skills: ", tyrion.swordSkills); // 9 (delegated)
27  console.log("Tyrion's Defense: ", tyrion.defense); // 4 (unique to Tyrion)
```

Listing 5.2: Prototype pattern using object delegation

In the Listing 5.2 we can see that we used the `Object.create()` method to create a new object `tyrion` based on the object `jamie` and we added a new property `defense` to the new object. Using Object Delegation we can easily implement advanced concepts such as *differential inheritance* [36], which means objects are able to directly inherit from other objects. In other words one object serves as the prototype for another object, allowing the new object to inherit properties and methods from the prototype of the other object.

The use of object delegation removes the need for the `clone` method, making the *Prototype* pattern not so useful in the contex of the JavaScript.

5.3 Observer

Observer is one of the most frequently used patterns. This pattern allows one or several objects (Observers) to be notified when another object (Subject) changes its state without explicit dependencies between them. Typically, this involves a subject maintaining a list of dependent objects (Observers) and automatically notifying them of any state changes. The most common usage for the observer pattern is to keep components updated on state changes.

When a subject needs to inform observers about an event or change, it sends a notification to all observers, which can include relevant data about the event. If an observer no longer wants to receive updates from the subject, it can be removed from the list of observers. This pattern is useful when you need to maintain consistency between related objects without making them tightly coupled.

5.3.1 Observer implementation

From the book by Simon Timms we have a scenario where in Westeros, controlling who sits on the throne and monitoring their actions is a complex game. Many players in this game of thrones use numerous spies to uncover the moves of others. Often, these spies are employed by multiple players and must report their findings to all of them.

The spy scenario is an ideal application for the *Observer*. In this example, the spy is the official doctor to the king and the players are keenly interested in the amount of painkiller being prescribed to the ailing king. This information can provide a player with advanced knowledge of when the king might die.

```
var Westeros;
(function (Westeros) {
    var Court;
    (function (Court) {
        class GetterSetter {
            GetProperty() {
                return this._property;}
            SetProperty(value) {
                var temp = this._property;
                this._property = value;
                if (this._listener) {
                    this._listener.Event(value, temp);}}
        }
        class Listener {
            Event(newValue, oldValue) {
                //do something
                }
        }
        class Spy {
            constructor() {
                this._partiesToNotify = [];}
            Subscribe(subscriber) {
                this._partiesToNotify.push(subscriber);}
            Unsubscribe(subscriber) {
                this._partiesToNotify = this._partiesToNotify.filter(sub =>
                    sub !== subscriber);}
            SetPainKillers(painKillers) {
                this._painKillers = painKillers;
                for (let i = 0; i < this._partiesToNotify.length; i++) {
                    this._partiesToNotify[i](painKillers);
                }
            }
        }
        class Player {
            OnKingPainKillerChange(newPainKillerAmount) {
                // Perform some action
                console.log(`The king has ${newPainKillerAmount} painkillers`)
                    ;}}
```

```
37          //Example
38          let spy = new Spy();
39          let player = new Player();
40          spy.Subscribe(player.OnKingPainKillerChange); // subscribe from
                notifications
41          spy.SetPainKillers(12); // spy will notify all subscribers
42          spy.Unsubscribe(player.OnKingPainKillerChange); // Unsubscribe from
                notifications
43          spy.SetPainKillers(15); // No notifications will be sent
44          spy.Subscribe(player.OnKingPainKillerChange); // subscribe from
                notifications
45          spy.SetPainKillers(20); // spy will notify all subscribers
46      })(Court = Westeros.Court || (Westeros.Court = {}));
47  })(Westeros || (Westeros = {}));
```

Listing 5.3: Implementation of the observer pattern

In the Listing 5.3 we can see that we have a `Spy` class that has a list of subscribers and a method `SetPainKillers` that notifies all subscribers when the painkillers are changed. We also have a `Player` class that has a method `OnKingPainKillerChange` that is called when the painkillers are changed. When the `Player` class is subscribed to the `Spy` class, its `OnKingPainKillerChange` method is invoked whenever the painkillers are changed.

In other languages, subscribers typically need to follow a specific interface, with the observer calling only the interface method. This constraint doesn't apply to JavaScript. Instead, we can simply provide the `Spy` class with a function, meaning there is no strict interface requirement for the `subscriber`.

The *Observer* can also be applied to both methods and properties. By doing so, you can create *hooks* for additional behavior, which is a common approach for providing a plugin infrastructure in JavaScript libraries.

```
1  // Mixin for observable behavior
2  const ObservableMixin = {
3      listeners: new Map(),
4
5      subscribe(callback, key) {
6          this.listeners.set(key, callback);
7      },
8
9      unsubscribe(key) {
10         this.listeners.delete(key);
11     },
12
13     notify(...args) {
14         this.listeners.forEach(listener => listener(...args));
15     }
16 };
17
18 //Spy(Observer) Using Object.create for delegation and mixing in observable
       behavior
```

```
19 const Spy = Object.create(ObservableMixin);
20
21 Spy.setPainKillers = function (painKillers) {
22     this.painKillers = painKillers;
23     this.notify(painKillers); // Notify all observers
24 };
25
26 //Player (subject) - Using a prototype-based approach
27 class Player {
28     constructor(name) {
29         this.name = name;
30     }
31     onPainKillerChange(newPainKillerAmount) {
32         console.log(`${this.name} has been notified: The king has ${
               newPainKillerAmount} painkillers`);
33     }
34 }
35
36 // Example of reflection using Reflect API
37 function dynamicallySubscribe(spy, player, methodName, key) {
38     const boundMethod = player[methodName].bind(player);
39     Reflect.apply(spy.subscribe, spy, [boundMethod, key]);
40 }
41
42 // Example usage
43 const spy = Object.create(Spy); // Create an instance of Spy with the mixin
       behavior
44
45 const player1 = new Player("Player 1");
46 const player2 = new Player("Player 2");
47
48 dynamicallySubscribe(spy, player1, 'onPainKillerChange', 'player1'); //
       Subscribe dynamically using reflection
49 dynamicallySubscribe(spy, player2, 'onPainKillerChange', 'player2');
50
51 spy.setPainKillers(12); // Notify all observers
52
53 spy.unsubscribe('player1'); // Unsubscribe player 1 using its key
54 spy.setPainKillers(20); // Only player 2 will be notified
```

Listing 5.4: Update version of the Observer pattern

In the Listing 5.4 we used several mechanisms were introduced to enhance flexibility and modularity. The implementation of an `ObservableMixin` allows any object to subscribe, unsubscribe and notify listeners, promoting code reuse and reducing duplication by enabling observable behavior across different objects. This is a significant improvement over a dedicated `Spy` class handling notifications. Additionally, prototype-based inheritance using `Object.create` provides a more lightweight and dynamic structure compared to traditional class-based inheritance, leveraging JavaScript's prototypal nature

for greater flexibility.

Reflection is used through the `Reflect` API, which enables the dynamic subscription of players to the `Spy`, allowing for more adaptable runtime interactions. This dynamic behavior facilitates flexible and efficient changes to object properties and methods as needed. Lastly, object delegation allows the `Spy` to delegate its observable behavior to the `ObservableMixin`, separating concerns and enhancing design flexibility by avoiding the rigidity of classical inheritance. These improvements optimize the *Observer* for reuse and flexibility while leveraging key JavaScript features.

5.4 Abstract Factory

The *Abstract Factory* is used to create kits of objects without knowing the concrete types of the objects. The *Abstract Factory* declares an interface for creating famalies of related or dependent objects without specifying their concrete classes, enabling family polymorphism (see more in 2.3) and flexibility in the object creation process. But, in the case of the JavaScript we dont need to implement the interface for the *Abstract Factory*, instead of interfaces, JavaScript trusts that the class you provide implements all the appropriate methods. At runtime the interpreter will attempt to call the method you request and, if it is found, call it. The interpreter simply assumes that if your class implements the method then it is that class. This is known as duck typing (see more at 2.2).

5.4.1 Abstract Factory implementation

The scenario for *Abstract Factory* is that in some kingdom, the ruling house changes relatively often and while there is likely some conflict during the transition, we will set that aside for now. Each house governs in different ways, some prioritize peace, while others rule with strict control. Since a kingdom is too vast to be governed by a single person, the king delegates certain decisions to a trusted second-in-command, known as the Hand of the King. Additionally, the king is advised by a council composed of influential lords and ladies from the realm.

The *Abstract Factory* pattern is a good fit for this scenario. The kingdom's ruler, the king, is the abstract factory and the houses are the concrete factories. The king can create a Hand of the King and a council, which are the products of the abstract factory. The Hand of the King and the council are the abstract products and the lords and ladies are the concrete products.

```
1 var Westeros;
2 (function (Westeros) {
3     (function (Ruling) {
4         (function (Lannister) {
5             var KingJoffery = (function () {
6                 function KingJoffery() {
7                 }
```

```
8                 KingJoffery.prototype.makeDecision = function () {
9                     console.log("Decision made by King Joffery");
10                };
11                KingJoffery.prototype.marry = function () {
12                };
13                return KingJoffery;
14            })();
15            Lannister.KingJoffery = KingJoffery;
16
17            var LordTywin = (function () {
18                function LordTywin() {
19                }
20                LordTywin.prototype.makeDecision = function () {
21                    console.log("Decision made by Lord Tywin");
22                };
23                return LordTywin;
24            })();
25            Lannister.LordTywin = LordTywin;
26
27            var LannisterFactory = (function () {
28                function LannisterFactory() {
29                }
30                LannisterFactory.prototype.getKing = function () {
31                    return new KingJoffery();
32                };
33                LannisterFactory.prototype.getHandOfTheKing = function () {
34                    return new LordTywin();
35                };
36                return LannisterFactory;
37            })();
38            Lannister.LannisterFactory = LannisterFactory;
39        })(Ruling.Lannister || (Ruling.Lannister = {}));
40        var Lannister = Ruling.Lannister;
41    })(Westeros.Ruling || (Westeros.Ruling = {}));
42    var Ruling = Westeros.Ruling;
43})(Westeros || (Westeros = {}));
44
45var Westeros;
46(function (Westeros) {
47    (function (Ruling) {
48        (function (Targaryen) {
49            var KingAerys = (function () {
50                function KingAerys() {
51                }
52                KingAerys.prototype.makeDecision = function () {
53                    console.log("Decision made by King Aerys");
54                };
55                KingAerys.prototype.marry = function () {
56                };
57                return KingAerys;
```

```
58              })();
59              Targaryen.KingAerys = KingAerys;
60
61              var LordConnington = (function () {
62                  function LordConnington() {
63                  }
64                  LordConnington.prototype.makeDecision = function () {
65                      console.log("Decision made by Lord Connington");
66                  };
67                  return LordConnington;
68              })();
69              Targaryen.LordConnington = LordConnington;
70
71              var TargaryenFactory = (function () {
72                  function TargaryenFactory() {
73                  }
74                  TargaryenFactory.prototype.getKing = function () {
75                      return new KingAerys();
76                  };
77                  TargaryenFactory.prototype.getHandOfTheKing = function () {
78                      return new LordConnington();
79                  };
80                  return TargaryenFactory;
81              })();
82              Targaryen.TargaryenFactory = TargaryenFactory;
83          })(Ruling.Targaryen || (Ruling.Targaryen = {}));
84          var Targaryen = Ruling.Targaryen;
85      })(Westeros.Ruling || (Westeros.Ruling = {}));
86      var Ruling = Westeros.Ruling;
87  })(Westeros || (Westeros = {}));
88
89  var Westeros;
90  (function (Westeros) {
91      (function (Ruling) {
92          var CourtSession = (function () {
93              function CourtSession(abstractFactory) {
94                  this.abstractFactory = abstractFactory;
95                  this.COMPLAINT_THRESHOLD = 10;
96              }
97              CourtSession.prototype.complaintPresented = function (complaint) {
98                  if (complaint.severity < this.COMPLAINT_THRESHOLD) {
99                      this.abstractFactory.getHandOfTheKing().makeDecision();
100                 } else
101                     this.abstractFactory.getKing().makeDecision();
102             };
103             return CourtSession;
104         })();
105         Ruling.CourtSession = CourtSession;
106     })(Westeros.Ruling || (Westeros.Ruling = {}));
107     var Ruling = Westeros.Ruling;
```

```
})(Westeros || (Westeros = {}));

var courtSession1 = new Westeros.Ruling.CourtSession(new Westeros.Ruling.
    Targaryen.TargaryenFactory());
courtSession1.complaintPresented({ severity: 8 });
courtSession1.complaintPresented({ severity: 12 });

var courtSession2 = new Westeros.Ruling.CourtSession(new Westeros.Ruling.
    Lannister.LannisterFactory());
courtSession2.complaintPresented({ severity: 8 });
courtSession2.complaintPresented({ severity: 12 });
```

Listing 5.5: Implementation of the abstract factory pattern

In the Listing 5.5 we can see that we have two factories, one for the Lannister family and one for the Targaryen family. We also have a `CourtSession` class that receives an abstract factory and depending on the severity of the complaint it calls the `makeDecision` method of the `HandOfTheKing` or the `King`.

Implementing the *Abstract factory* in JavaScript is easier compared to other languages, but this comes at the cost of losing compiler checks that ensure full implementation of the factory or its products. This is a recurring theme when applying design patterns in JavaScript. Despite the differences between static languages and JavaScript, the *Abstract Factory* remains valuable, especially when creating a set of objects that collaborate and may need to be replaced entirely. It is also useful for ensuring a specific set of objects are used together without substitutions.

```
// Mixin for decision-making behavior
const DecisionMakerMixin = {
    makeDecision() {
        console.log(`Decision made by ${this.name}`);
    }
};

// Lannister Kingdom
class KingJoffery {
    constructor() {
        this.name = "King Joffery";
    }
}
// object.assign is used to copy the mixin methods to the prototype of the
    class
Object.assign(KingJoffery.prototype, DecisionMakerMixin);

class LordTywin {
    constructor() {
        this.name = "Lord Tywin";
    }
}
```

```
22 // object.assign is used to copy the mixin methods to the prototype of the
       class
23 Object.assign(LordTywin.prototype, DecisionMakerMixin);
24
25 class LannisterFactory {
26     constructor() { }
27     getKing() {
28         return new KingJoffery();
29     }
30     getHandOfTheKing() {
31         return new LordTywin();
32     }
33 }
34
35 // Targaryen Kingdom
36 class KingAerys {
37     constructor() {
38         this.name = "King Aerys";
39     }
40 }
41 Object.assign(KingAerys.prototype, DecisionMakerMixin);
42
43 class LordConnington {
44     constructor() {
45         this.name = "Lord Connington";
46     }
47 }
48 Object.assign(LordConnington.prototype, DecisionMakerMixin);
49
50 class TargaryenFactory {
51     constructor() { }
52     getKing() {
53         return new KingAerys();
54     }
55     getHandOfTheKing() {
56         return new LordConnington();
57     }
58 }
59
60 // Court Session using reflection for decisions
61 class CourtSession {
62     constructor(abstractFactory) {
63         this.abstractFactory = abstractFactory;
64         this.COMPLAINT_THRESHOLD = 10;
65     }
66     complaintPresented(complaint) {
67         let decisionMaker;
68         if (complaint.severity < this.COMPLAINT_THRESHOLD) {
69             decisionMaker = this.abstractFactory.getHandOfTheKing();
70         } else {
```

```
71              decisionMaker = this.abstractFactory.getKing();
72          }
73
74          // Using reflection to call the makeDecision method
75          Reflect.apply(decisionMaker.makeDecision, decisionMaker, []);
76      }
77  }
78  // Example usage
79  const courtSession1 = new CourtSession(new TargaryenFactory());
80  courtSession1.complaintPresented({ severity: 8 });
81  courtSession1.complaintPresented({ severity: 12 });
82
83  const courtSession2 = new CourtSession(new LannisterFactory());
84  courtSession2.complaintPresented({ severity: 8 });
85  courtSession2.complaintPresented({ severity: 12 });
```

Listing 5.6: Update version of the Abstract Factory pattern

From the Listing 5.6 we can see several key mechanisms were introduced to enhance flexibility and modularity. The `DecisionMakerMixin` provides a shared `makeDecision()` method, which reduces code duplication between different characters like kings and hands. By using this mixin, objects such as King Joffrey and King Aerys share common behavior while maintaining their distinct identities, making the design cleaner and more maintainable.

The system also leverages reflection through `Reflect.apply` to dynamically invoke the `makeDecision()` method. Instead of calling this method directly, reflection adds flexibility, allowing dynamic interactions with objects and improving the runtime behavior of the system.

Finally, duck typing is preserved in the `CourtSession` class. It continues to expect that both kings and hands implement a `makeDecision()` method, without relying on their specific class types. This allows the Abstract Factory pattern to remain flexible and decoupled from strict type enforcement, enhancing its adaptability in JavaScript environments.

5.5 Structural Patterns

Structural patterns have a main goal to simplify design by providing straightfoward ways for objects to interact. They help ensure that if one part of a system changes, the entire system doesn't need to do the same. For the purpose of this thesis the update versions of the structural patterns are all made in a similar way and we will use the *Composite* pattern as an example.

The scenario for the *Composite* pattern is in Westeros, in some occasions like weddings and religious holidays are celebrated with elaborate feasts, featuring both simple and complex dishes. Each dish is made up of ingredients. Simple recipes, like baked apples, consist of individual ingredients, while complex dishes, like a dessert similar to tiramisu,

are composed of multiple ingredients, some of which are composites themselves. This pattern allows for complex recipes to contain ingredients that are also made up of other ingredients, with operations being applied uniformly across all components.

```
var Westeros;
(function (Westeros) {
    var Food;
    (function (Food) {
        class SimpleIngredient {
            constructor(name, calories, ironContent, vitaminCContent) {
                this.name = name;
                this.calories = calories;
                this.ironContent = ironContent;
                this.vitaminCContent = vitaminCContent;
            }
            GetName() {
                return this.name;
            }
            GetCalories() {
                return this.calories;
            }
            GetIronContent() {
                return this.ironContent;
            }
            GetVitaminCContent() {
                return this.vitaminCContent;
            }
        }
        Food.SimpleIngredient = SimpleIngredient;
        class CompoundIngredient {
            constructor(name) {
                this.name = name;
                this.ingredients = new Array();
            }
            AddIngredient(ingredient) {
                this.ingredients.push(ingredient);
            }
            GetName() {
                return this.name;
            }
            GetCalories() {
                let total = 0;
                for (let i = 0; i < this.ingredients.length; i++) {
                    total += this.ingredients[i].GetCalories();
                }
                return total;
            }
            GetIronContent() {
                let total = 0;
                for (let i = 0; i < this.ingredients.length; i++) {
```

```
                    total += this.ingredients[i].GetIronContent();
                }
                return total;
            }
            GetVitaminCContent() {
                let total = 0;
                for (let i = 0; i < this.ingredients.length; i++) {
                    total += this.ingredients[i].GetVitaminCContent();
                }
                return total;
            }
        }
        Food.CompoundIngredient = CompoundIngredient;
    })(Food = Westeros.Food || (Westeros.Food = {}));
})(Westeros || (Westeros = {}));
let egg = new Westeros.Food.SimpleIngredient("Egg", 155, 6, 0);
let milk = new Westeros.Food.SimpleIngredient("Milk", 42, 0, 0);
let sugar = new Westeros.Food.SimpleIngredient("Sugar", 387, 0, 0);
let rice = new Westeros.Food.SimpleIngredient("Rice", 370, 8, 0);
let ricePudding = new Westeros.Food.CompoundIngredient("Rice Pudding");
ricePudding.AddIngredient(egg);
ricePudding.AddIngredient(rice);
ricePudding.AddIngredient(milk);
ricePudding.AddIngredient(sugar);
console.log("A serving of rice pudding contains:");
console.log(ricePudding.GetCalories() + " calories"); // 954 calories
```

Listing 5.7: Implementation of the composite pattern

In the Listing 5.7 we can see that we have two classes, one for the simple ingredients and one for the compound ingredients. The compound ingredients are made up of simple ingredients and the `GetCalories`, `GetIronContent` and `GetVitaminCContent` methods are applied uniformly across all components. In this example, a composite ingredient iterates over its internal ingredients, applying the same operation to each. Since the prototype model allows objects to share behavior, there's no need to define a separate interface for the ingredients, as the structure and behavior are inherently shared through prototypes.

Of course this example is a simple one. The same rice pudding can be used as an ingredient in an even more complex recipe, and so on. The *Composite* pattern is useful when you need to treat individual objects and compositions of objects uniformly, allowing you to create complex structures with simplicity.

```
class SimpleIngredient {
    constructor(name, calories, ironContent, vitaminCContent) {
        this.name = name;
        this.calories = calories;
        this.ironContent = ironContent;
        this.vitaminCContent = vitaminCContent;
    }
```

```
8
9       getName() {
10          return this.name;
11      }
12
13      getCalories() {
14          return this.calories;
15      }
16
17      getIronContent() {
18          return this.ironContent;
19      }
20
21      getVitaminCContent() {
22          return this.vitaminCContent;
23      }}
24  class CompoundIngredient {
25      constructor(name) {
26          this.name = name;
27          this.ingredients = [];
28      }
29
30      addIngredient(ingredient) {
31          this.ingredients.push(ingredient);
32      }
33
34      getName() {
35          return this.name;
36      }
37
38      getCalories() {
39          return this.ingredients.reduce((total, ingredient) => total +
                ingredient.getCalories(), 0);
40      }
41
42      getIronContent() {
43          return this.ingredients.reduce((total, ingredient) => total +
                ingredient.getIronContent(), 0);
44      }
45
46      getVitaminCContent() {
47          return this.ingredients.reduce((total, ingredient) => total +
                ingredient.getVitaminCContent(), 0);
48      }}
49  // Example usage
50  const egg = new SimpleIngredient("Egg", 155, 6, 0);
51  const milk = new SimpleIngredient("Milk", 42, 0, 0);
52  const sugar = new SimpleIngredient("Sugar", 387, 0, 0);
53  const rice = new SimpleIngredient("Rice", 370, 8, 0);
54  const ricePudding = new CompoundIngredient("Rice Pudding");
```

```
ricePudding.addIngredient(egg);
ricePudding.addIngredient(milk);
ricePudding.addIngredient(sugar);
ricePudding.addIngredient(rice);

console.log("A serving of rice pudding contains:");
console.log(`${ricePudding.getCalories()} calories`);
```

Listing 5.8: Update version of the Composite pattern

In the Listing 5.8 we can see that we used the same classes as in the Listing 5.7 but we used the ES6 class syntax. The `getCalories`, `getIronContent` and `getVitaminCContent` methods are applied uniformly across all components. We also used the `reduce` method to calculate the total calories, iron content and vitamin C content of the compound ingredient. The JavaScript's `reduce` method is a powerful tool for working with arrays, allowing you to apply a function to each element of an array and accumulate the results. Using reduce improves performance by avoiding multiple iterations over the same data set.

However in this updated version of the *Composite* pattern we can see that we don't used none of the advanced mechanism that we talked and used before. This is because the *Composite* pattern is a simple pattern and doesn't need any of those mechanisms, the only one that can be used is the mixin class. Lets see an example using the mixin class.

```
// NutritionalInfoMixin to add nutritional calculations
const NutritionalInfoMixin = Base => class extends Base {
    calculateCalories() {
        return this.ingredients.reduce((total, ingredient) => total +
            ingredient.getCalories(), 0);
    }
    calculateIronContent() {
        return this.ingredients.reduce((total, ingredient) => total +
            ingredient.getIronContent(), 0);
    }
    calculateVitaminCContent() {
        return this.ingredients.reduce((total, ingredient) => total +
            ingredient.getVitaminCContent(), 0);
    }
};
// SimpleIngredient class
class SimpleIngredient {
    constructor(name, calories, ironContent, vitaminCContent) {
        this.name = name;
        this.calories = calories;
        this.ironContent = ironContent;
        this.vitaminCContent = vitaminCContent;
        this.ingredients = [this]; // Add self to ingredients
    }
    getCalories() {
        return this.calories;
    }
```

```
25      getIronContent() {
26          return this.ironContent;
27  }
28      getVitaminCContent() {
29          return this.vitaminCContent;
30      }
31  }
32  class CompoundIngredient extends NutritionalInfoMixin(class {}) {
33      constructor(name) {
34          super();
35          this.name = name;
36          this.ingredients = [];
37      }
38
39      addIngredient(ingredient) {
40          this.ingredients.push(ingredient);
41      }
42  }
43  // Example usage
44  const egg = new SimpleIngredient("Egg", 155, 6, 0);
45  const milk = new SimpleIngredient("Milk", 42, 0, 0);
46  const sugar = new SimpleIngredient("Sugar", 387, 0, 0);
47  const rice = new SimpleIngredient("Rice", 370, 8, 0)
48  const ricePudding = new CompoundIngredient("Pudding");
49  ricePudding.addIngredient(egg);
50  ricePudding.addIngredient(milk);
51  ricePudding.addIngredient(sugar);
52  ricePudding.addIngredient(rice);
53  console.log("Total Calories in Pudding:", ricePudding.calculateCalories()); //
       Total Calories in Pudding: 954
```

Listing 5.9: Composite pattern using mixin class

In the Listing 5.9 we can see that we used the mixin class to add the nutritional calculations to the `CompoundIngredient` class. The use of the mixin class allows to improve the code *Reusability* and maintainability and increase the flexibility, but in this case the use of the mixin class is not necessary and increase complexity of the code. The trade-off in this case is not worth it, adding so much complexity to a simple pattern in trade of less methods in some classes.

The improvement that was made in the structural patterns was to use recent sintax of the JavaScript and other specific methods of the language like the `reduce` method. This was common in most of the updated version of the structural patterns (only exception is the *Decorator*), that some of the mechanisms don't make sense in the context of these patterns and the ones that can be used just add complexity. we can see that Reflection is more useful in scenarios requiring dynamic method invocation or property manipulation, which are not typically central to structural patterns, but some Reflection function can be used to facilitate the interaction between classes (was not used in this set of pattern but in the second one was used in some examples). Mixins and Object Delegation can be used to

share functionalities across different classes and objects but are less about the arrangement of systems and more about enhancing individual object capabilities. Duck Typing and Prototype-based Inheritance may be used to a degree, but they do not fundamentally change the primary purpose of these structural patterns, which is to manage how different parts of a system work together more efficiently.

So we can conclude that for structural patterns, advanced mechanisms often do not provide additional value and they don't align well with the goals of these patterns. Structural patterns focus on simplifying interactions between different classes or objects, often through composition or delegation. They don't typically need to share behavior (as with mixins) or dynamically manipulate objects at runtime (as with reflection). In structural patterns, the primary concern is managing complexity by providing simplified interfaces or access control. Using advanced mechanisms would introduce unnecessary complexity in situations where simpler delegation or method calls are enough.

In the end is important to recognize that these advanced mechanisms don't universally apply or add value across all scenarios. It is important to consider the context before selecting the right tools and approaches based on the specific requirements and goals of the software design, rather than using complex solutions for the sake of complexity.

6

Results and Discussion

In this chapter, we present the results of the study and discuss the implications of the findings. We start by presenting the results of the analysis of the modularity properties of both original and Updated versions of the design patterns in JavaScript, comparing the different approaches. Then, we discuss the results in terms of modularity and some of its properties that was used to evaluate the design patterns implementations. Finally, we discuss the implications of the results and the limitations of the study.

6.1 Modularity Properties

To analysis the difference between the two styles of implementing the design pattern is based on six modularity properties:

- **Locality** refers to the ability to encapsulate all code related to a concern within a single module. Ensuring code locality at some level is essential for an efective modularization at that level and directly influences all subsequent modularity properties.
- **Reusability** indicates whether a module can be applied in different scenarios without requiring invasive modifications. A reusable module should support adaptation to different contexts while maintaining a consistent implementation.
- **Composition Transparency** evaluates whether multiple instances of a module can be composed within the same system without interference. A module with high composition transparency should not impose constraints on the composition of other instances of that module.
- **(Un)Pluggability** refers to the ability to add or remove a module from a system without modifying the surrounding code. This property ensures that a module can be optionally integrated without affecting other parts of the system.
- **Extensibility** is defined as the ability to extend a module's functionality without modifying its source code. This property supports incremental development and facilitates adaptation to new requirements.

- **Closure Under Composition** ensures that the combination of multiple modules results in a new module that maintains the same structural integrity and can be used in further compositions.

6.1.1 Limitations of Module Based Pattern Isolation in JavaScript

In classical OOP languages, design patterns are often structured as separate modules that encapsulate all their logic (*Locality*). However, JavaScript doesn't provide a strict module system that easily isolates pattern code into separate modules because, using object delegation, mixins, or function-based composition rather than strict class encapsulation, pattern logic (glue code) must often interact dynamically with participant classes, making strict separation not suitable.

The example of the *Observer* serves to show that all patterns in JavaScript have the same difficulty to have their pattern code isolated in a module and for that reason the results and analysis of the modularity properties were done at the participant and pattern level and not at module level.

In traditional OOP, an *Observer* pattern would consist of a dedicated Observer module that all subjects (participants) interact with through well-defined interfaces. This modular structure ensures clear separation of concerns, where the observer logic remains encapsulated in a separate module and subjects only rely on it through interfaces.

In JavaScript, the *Observer* logic is dynamically composed into participants rather than existing as a separate module. This is evident in the updated implementation of the *Observer* (see the complete implementation in listing 5.4), where the observable behavior is applied dynamically using `Object.create(ObservableMixin)`. As a result, there isn't clear boundary between where the pattern module ends and where the participants begin and the mixins (ObservableMixin) modify the behavior of the participants at runtime, meaning there isn't a strict *Observer* module in the traditional sense.

Given that JavaScript doesn't enforce strict module separation, a more appropriate approach is to evaluate modularity at the participant level by distinguishing glue code from participant-specific code.

For example in the implementation of the *Observer* in listing 6.1, `dynamicallySubscribe` acts as glue code because it links participants dynamically via reflection. This function connects the Observer (`spy`) with the Subject (`player`), but it does not belong to either. It is neither purely Observer logic nor purely Participant logic, its a glue between them.

```
1 function dynamicallySubscribe(spy, player, methodName, key) {
2     const boundMethod = player[methodName].bind(player);
3     Reflect.apply(spy.subscribe, spy, [boundMethod, key]);}
```

Listing 6.1: Glue code of the *Observer* example

From the same *Observer* example the `ObservableMixin` and `Spy` (from the listing 6.2) correspond to the pattern code because it defines the core logic of the observers

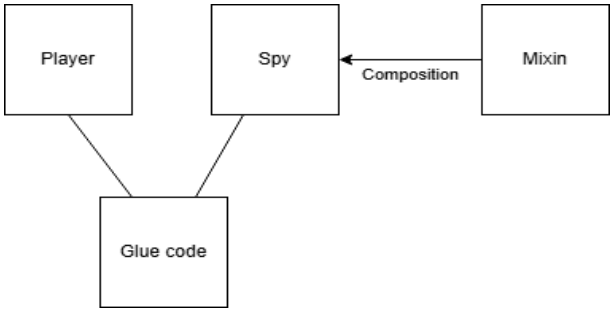

Figure 6.1: Structure of the *Observer* example, illustrating the separation between pattern logic, participant and glue code

(subscribing, unsubscribing and notifying) and It's designed to be reused across different contexts (not tied to any specific `Player`)

```
const ObservableMixin = {
    listeners: new Map(),

    subscribe(callback, key) {
        this.listeners.set(key, callback);
    },

    unsubscribe(key) {
        this.listeners.delete(key);
    },

    notify(...args) {
        this.listeners.forEach(listener => listener(...args));
    }
};
const Spy = Object.create(ObservableMixin);
Spy.setPainKillers = function (painKillers) {
    this.painKillers = painKillers;
    this.notify(painKillers); // Notify all observers
};
```

Listing 6.2: Pattern code of the *Observer* example

```
class Player {
    constructor(name) {
        this.name = name;
    }
    onPainKillerChange(newPainKillerAmount) {
        console.log(`${this.name} has been notified: The king has ${
            newPainKillerAmount} painkillers`);
    }
}
```

Listing 6.3: Participant code of the *Observer* example

The `Player` class (from the listing 6.3) is a participant-specific code that defines a concrete application object (`Player`) that interacts with the pattern code (`Spy`). This class is not inherently part of the pattern code but simply a participant that interacts with it. While it depends on the pattern, the Observer logic does not belong to it, reinforcing the need to analyze modularity at the participant level rather than assuming all pattern-related logic exists in a separate module, because applying modularity properties at the module level, as done in traditional OO studies [18][31][25], can lead to misleading conclusions.

Table 6.1 presents a comparative assessment of the modularity properties of the original and updated versions of the design patterns. The table summarizes the key results of the analysis, highlighting the differences in modularity properties between the two implementations. To analyze the table is need to understand that *Locality* apply to any concern (including a single participant in a pattern that comprises multiple participants) and it is necessary to have *Locality* to have the other properties and the analysis of the other properties is made at the same level that *Locality*.

In the next sections we will discuss the results based on the table 6.1 and the implications of the results.

6.1.2 Locality

Most of the original implementations generally demonstrate *Locality* of Pattern logic (visible in table 6.1 in the locality column and the original rows) as most encapsulate their core behavior within specific classes or functions. However, in some cases (such as *Singleton* and *Visitor*) some methods or data were split between prototypes or manual checks reducing clarity but not in a extreme that reduces the *locality* of the whole pattern

By implementing the updated versions, *Locality* at the pattern level was universally preserved. Typically, the use of ES6 classes and mixins ensures that all related logic is well packaged. Overall, the updated versions showed *Locality* with most of the paterns centralizing behavior in mixins or base classes, reducing code scattered across multiple participant classes. A case of that is the *Observer*, significantly improved using `ObservableMixin`, which removes observer logic from observer objects (which are the participants from the original example), centralizing it in the mixin. Previously, each subject had to maintain its own observer list, but now any class can inherit observer capabilities without modifications.

Full isolation of the code related to the pattern participants wasn't always attained. Some patterns still required participant classes to contain explicit references to pattern behavior. JavaScript's dynamic features like mixins and delegation helped in many cases but did not eliminate all dependencies in patterns where participant interaction was intrinsic. Its possible to see that, for example, the *Singleton* still has limited *Locality* at participant level (as presented in the table 6.1 in the Locality column and Singleton row of the updated version) because instance remains a global property, which participants directly reference, preventing full encapsulation.

	Version	Locality	Reusability	Composition Transparency	(Un)pluggability	Extensibility	Closure Under Composition
Abstract Factory	Original	Yes	No	(Yes)	No	No	No
	Updated	Yes	Yes	Yes	Yes	Yes	No
Builder	Original	Yes	No	(Yes)	No	No	No
	Updated	Yes	Yes	Yes	Yes	Yes	No
Factory Method	Original	Yes	No	No	No	No	No
	Updated	Yes	Yes	No	(Yes)	(Yes)	No
Prototype	Original	Yes	Yes	No	No	No	Yes
	Updated	Direct Language Support					
Singleton	Original	(Yes)	No	No	No	No	No
	Updated	(Yes)	No	No	(Yes)	(Yes)	No
Adapter	Original	Yes	No	(Yes)	No	No	No
	Updated	Yes	Yes	Yes	Yes	Yes	Yes
Bridge	Original	Yes	Yes	(Yes)	(Yes)	(Yes)	Yes
	Updated	Yes	Yes	Yes	Yes	Yes	Yes
Composite	Original	Yes	Yes	Yes	No	No	(Yes)
	Updated	Yes	Yes	Yes	Yes	Yes	yes
Decorator	Original	Yes	(Yes)	(Yes)	No	No	Yes
	Updated	Yes	Yes	Yes	Yes	Yes	Yes
Facade	Original	Yes	No	No	No	No	No
	Updated	Yes	(Yes)	Yes	Yes	Yes	Yes
Flyweight	Original	Yes	No	(Yes)	(Yes)	No	(Yes)
	Updated	Yes	Yes	(Yes)	Yes	Yes	Yes
Proxy	Original	Yes	No	(Yes)	(Yes)	No	No
	Updated	Yes	Yes	Yes	Yes	Yes	No
Chain of Responsibility	Original	Yes	No	(Yes)	No	No	(Yes)
	Updated	Yes	Yes	Yes	Yes	Yes	Yes
Command	Original	Yes	No	(Yes)	No	No	No
	Updated	Yes	Yes	Yes	Yes	Yes	No
Iterator	Original	Yes	No	(Yes)	(Yes)	No	No
	Updated	Yes	Yes	Yes	Yes	Yes	(Yes)
Interpreter	Original	Yes	No	(Yes)	No	No	No
	Updated	Yes	Yes	Yes	Yes	Yes	No
Mediator	Original	Yes	No	(Yes)	(Yes)	No	No
	Updated	Yes	Yes	Yes	Yes	Yes	Yes
Memento	Original	Yes	No	(Yes)	(Yes)	No	No
	Updated	Yes	Yes	Yes	Yes	Yes	No
Observer	Original	Yes	No	(Yes)	No	No	No
	Updated	Yes	Yes	Yes	Yes	Yes	No
State	Original	Yes	No	Yes	No	No	No
	Updated	Yes	Yes	Yes	Yes	Yes	Yes
Strategy	Original	Yes	No	Yes	(Yes)	No	No
	Updated	Yes	Yes	Yes	Yes	Yes	Yes
Template Method	Original	Yes	No	(Yes)	(Yes)	No	No
	Updated	Yes	Yes	(Yes)	Yes	Yes	No
Visitor	Original	(Yes)	No	(Yes)	No	No	No
	Updated	(Yes)	Yes	(Yes)	No	(Yes)	No

Table 6.1: Modularity properties of design patterns in JavaScript obtained from the 2 sets, organized by pattern. When Yes or No is used in the table means that the modularity properties is fully present or not present. However, "(Yes)" indicates that a modularity property is present in the implementation but with some limitations or areas for improvement. While the property is functional and contributes to the system's modularity, it may not fully align with the ideal implementation due to factors such as incomplete optimization, restricted flexibility, or added complexity. This designation helps highlight intermediate cases where the modularity principle is partially addressed but could benefit from refinement.

6.1.3 Reusability

The original versions often relied on manual checks or strong coupling between classes (for example, referencing global objects, manually verifying interfaces or using rigid inheritance hierarchies). These practices significantly limited the *Reusability* of the patterns,

as reflected in the Reusability column of Table 6.1 under the original versions. For instance, in the *Factory Method* pattern, subclasses had to be instantiated manually, which reduced the flexibility and reuse potential of the pattern across different contexts.

The updated versions usually introduced a base classes, mixins, or reflection to allow the main logic to be ported elsewhere with minimal change, making it easier to adopt and adapt in new settings. Therefore showing *Reusability* by modularizing shared behaviors because *Reusability* thrives when there's a shared interface or base (or Mixin) class ensuring all components can be substituted or reused in new contexts. Patterns like *Adapter*, *Strategy*, *Memento* and *Decorator* showed *Reusability* (as illustrated in the table 6.1 in the Reusability column and the updated rows). For example, the *Memento* was implemented as a mixin (`MementoMixin`) which can be applied to any object, giving it snapshot/restore functionality without modifying the object's core class.

6.1.4 Composition Transparency

By analyzing the composition transparency column from table 6.1 we can see that patterns like *Composite*, *Decorator* and *State* showed *composition Transparency* in their original versions (as demonstrated in the table 6.1 in the Composition Transparency column and original rows), as they clearly defined how components interacted and combined to form a larger system. However, patterns like *Factory Method* and *Prototype* were not transparent, as they required manual instantiation or cloning, which could be less clear or explicit.

Composite, *Decorator* and *State* inherently support *Composition Transparency* because they revolve around a single abstraction for both single and composed entities making easy to treat a nested structure, an object wrapped with extra features, or an object delegated to a state, exactly the same way it would a standalone object. This design principle is especially valuable for JavaScript based applications, which rely on dynamic object composition and can leverage advanced language features (mixins and reflection).

For the updated versions that used mixins classes and adopting dynamic techniques such as reflection, making composition more explicit and transparent. From all the enhancements the *Adapter* and *Visitor* were the most benefited. *Composition transparency* is visible with techniques that decouple components and make their interactions explicit that's why reflection and modular design play an important role in enhancing this property. A good example is the *Observer* by allowing each subject maintains its own observer list, allowing multiple observer instances to coexist without affecting each other.

Some patterns that require explicit structural references in participant classes. Patterns that rely on class-based registration rather than dynamic delegation often struggle with multiple pattern instances in the same system. A case of this is the *Template Method*, where each subclass enforces a strict method structure, meaning multiple templates cannot easily be combined, limiting *Composition Transparency*.

6.1.5 (Un)pluggability

In the table 6.1 show that some original version of the patterns didn't show *(un)pluggability* (for example *Chain of Responsibility*), typicaly required changing existing code in several places to add or remove components.

In the updated versions the use of reflection, object delegation, object mappings (instead of if statements when applicable) and base classes relying on prototypal inheritance simplified the addition or removal of components. Adding a new element, such as a new state, command, or adapter, typically requires only creating a new object, without modifying existing code. Decoupling logic and using dynamic features make possible the *(Un)pluggability* of the patterns. Most of the updated implementation of the patterns can achieve flawless integration or removal of participant code.

6.1.6 Extensibility

The *extensibility* of the original versions of the patterns were generally not visible due to rigid structures (as shown in the table 6.1 in the extensibility collum), reliance in subclassing or manual checks (if or switch), that means, original patterns required invasive changes on the source code to add new states/behaviors (for example new states in *State* or new logic in *Strategy* Pattern)

In the updated versions, object delegation, base classes, mixins, modular validations (mostly using reflection) mechanisms enable easy extension, letting you add new classes or logic in a separate file or method. In the table 6.1 its proper to see that patterns like *Strategy*, *State* and *Decorator* demonstrated significant improvements.

Extensibility benefits greatly from modular designs that allow extensions without modifying existing logic, based on improvements focused on enabling non-invasive extensions through subclassing or mixins.

6.1.7 Closure Under Composition

As showed in the table 6.1 on the closure under composition column many original versions of the patterns focused on binary behaviors (for example *singleton* and *proxy*) making this property irrelevant or very hard to achive because they have no meaningful composition aspect. However, some patterns (like *Composite* or *Chain of Responsibility*) partially supported *Closure Under Composition* but having manual checks or repetead code (as outlined in table 6.1 in the Closure under Composition column).

With the improvements was possible to see closure through dynamic composition (mostly mixins), but most patterns retained their original limitations. This property is not universally relevant across all patterns and depends on the design's intent, but patterns designed for hierarchical relationships or dynamic compositions see the biggest improvements from a well-structured approach that ensures each "child" remains a valid instance. The pattern that got the most improvements was the *Composite*, with the usage of

mixnins supports tree structures with dynamic hierarchy composition, allowing multiple trees to be combined cleanly.

6.2 Advanced Mechanisms in Design Patterns

Pattern	Mixins	Object Delegation	Reflection	Duck typing
Observer	X	X	X	X
Builder	X			
Template Method	X		X	
Memento	X			
Strategy	X	X		X
State	X	X		
Prototype		X		
Decorator	X	X		
Flyweight		X		
Command			X	X
Chain of Responsibility			X	X
Proxy			X	

Table 6.2: List of Patterns that got most benefited from the JavaScript advanced mechanins. "X" indicates that the pattern was improved by the advanced mechanism.

The table 6.2 only the patterns that demonstrated clear improvements in modularity through the use of advanced JavaScript mechanisms. These patterns were selected because the application of features like mixins, delegation, reflection, or duck typing led to visible and meaningful gains in one or more modularity properties. Patterns not listed either maintained their classical structure (for example, Singleton or Interpreter), exhibited minimal change or used these mechanisms without producing a substantial impact on modularity. However, this does not imply that the remaining patterns did not use advanced features, instead, their use did not contribute meaningfully to the analysis of these techniques effectiveness in improving modularity.

In summary, each advanced mechanism can significantly improve modularity by enhancing the properties. Their relevance is context-dependent: while duck typing allow for a flexible approach, object delegation remains essential for prototype-based systems and a vital role in sharing common logic. In scenarios where dynamic composition is central to the design, reflection and mixin-based approaches offer enormous benefits. Together, these advanced JavaScript features bridge the gap between traditional (or classical) inheritance approaches and a truly modular, dynamic codebase, underscoring their importance for achieving a robust, modern design.

6.3 Comparative Analysis of the Classical and Updated Approaches

One of the objectives of this thesis was to assess whether JavaScript's dynamic nature and advanced mechanisms enhanced the modularity of the GoF design patterns when compared to their tradicional OO implementations. The results show a spectrum of modularity improvements, with some patterns achieving significal gains, while others remained structurally constrained, offering little to no improvement over classical OO implementations.

6.3.1 Direct language support: Prototype

As expected, in JavaScript *Prototype* stands out as an exception. In classical object-oriented languages, *Prototype* is a necessary pattern used to clone objects without relying on class instantiation. However, JavaScript's native prototype-based delegation makes this pattern redundant, as the language itself provides built-in mechanisms for object cloning and prototype inheritance. By using this mechanisms eliminates the need to use this pattern in terms of JavaScript.

Unlike other patterns that require adaptation and/or improvements, *Prototype* is naturally supported by JavaScript. Since object delegation removes the need for explicit cloning logic required (new objects automatically inherit properties from their prototype) and eliminating deep copying (changes to original object affect clone unless explicitly overridden).

6.3.2 19 Patterns Improved via Advanced Mechanisms

Patterns in this group benefited from JavaScript's dynamic features and advanced mechanisms. These pattern mostly showed higher modularity in JavaScript than in traditional OO implementations at participant level.

This group of pattern can be divided in 3 sub groups:

1. **Highly benefited from JavaScript mechanism**: *Observer*, *Decorator*, *Proxy*, *Strategy*, *State*, *Iterator*.

2. **Improved but still structurally tied**: *Flyweight*, *Mediator*, *Adapter*, *Bridge*, *Memento*, *Façade*, *Abstract Factory*, *Composite*.

3. **Limited JavaScript Benefits**: *Builder*, *Command*, *Chain of Responsibility*, *Template Method*, *Factory Method*.

6.3.2.1 Highly benefited from JavaScript mechanism: Observer, Decorator, Proxy, Strategy, State, Iterator

Patterns in this group can utilized the JavaScript's advanced mechanism at almost maximum potential. They are very close to be direct language supports but the use of the mechanins doesn't eliminate the need of the pattern, like happens with the *Prototype* pattern. The *Observer* pattern is a good example of this group, where the use of mixins to centralize observer logic in a single object, allowing any class to inherit observer capabilities without modifications.

But, other good example is also from the decorator pattern that benefits a lot from using mixins classes and object delegation:

```
1   // Base Armor object
2   const BasicArmor = {
3       CalculateDamageFromHit(hit) {
4           return hit.Strength * 0.2;
5       },
6       GetArmorIntegrity() {
7           return 1;
8       }
9   };
10
11  // General enhancement as a decorator mixin
12  const ArmorEnhancementMixin = {
13      CalculateDamageFromHit(hit) {
14          hit.Strength *= 0.8; // Reduce damage before passing it to the
                base armor
15          return this.base.CalculateDamageFromHit(hit);
16      },
17      GetArmorIntegrity() {
18          return 0.9 * this.base.GetArmorIntegrity();
19      }
20  };
21
22
23  // Apply mixins directly using Object.assign and Object.create
24  const ChainMailArmor = Object.assign(
25      Object.create(BasicArmor),
26      ArmorEnhancementMixin,
27      { base: BasicArmor, resistanceFactor: 0.8, integrityFactor: 0.9 }
28  );
29  // Test the implementation
30  console.log(ChainMailArmor.CalculateDamageFromHit({ Location: "head",
        Weapon: "Sock filled with pennies", Strength: 12 }));
31  console.log("Armor Integrity:", ChainMailArmor.GetArmorIntegrity());
```

Listing 6.4: Updated version of the Decorator pattern

In the listing 6.4, it is possible to see that the *Decorator* pattern is another example of this group, where the use of mixins to enhance the functionality of an object at

runtime without affecting the object's functionality. The listing 6.4 shows the updated version of the *Decorator* pattern using mixins to enhance the functionality of an object at runtime without affecting the object's functionality. In the specific example the use of mixin classes (`ArmorEnhancementMixin` and `Object.assign()`) and object delegation (`Object.create()`) its possible too add new behavior to an object dynamically at runtime without modifying the object structure.

To add a new behavior to a new object with these advanced mechanisms becames a lot simpler like is show in the listing 6.5:

```
1     const SteelMailArmor = Object.assign(
2         Object.create(ChainMailArmor),
3         ArmorEnhancementMixin,
4         { base: ChainMailArmor, resistanceFactor: 0.7, integrityFactor: 0.85 }
5     );
```

Listing 6.5: Adding new behavior to an object using the Decorator Mixin

Other pattern that is worth talk about is the *Proxy*, for a differece reason than the provious patterns. In JavaScript exists one fruntion called `Proxy()` [38]. A `Proxy` object in JavaScript enables the creation of an intermediary object that can substitute for the original, while potentially redefining essential object-level operations such as property retrieval, assignment, or definition. Typical use cases include logging property accesses, validating or sanitizing inputs, or formatting data before it is stored or retrieved. To instantiate a `Proxy`, two parameters are required: target (the original object to be proxied) and handler (an object that specifies which operations will be intercepted and how these operations should be redefined).

```
1  // Define a Proxy API for BarrelCalculator to add caching behavior
2  const barrelCalculatorProxy = new Proxy(barrelCalculator, {
3      _cache: new Map(),
4
5      get(target, property, receiver) {
6          if (property === "calculateNumberNeeded") {
7              return (volume) => {
8                  if (this._cache.has(volume)) {
9                      console.log("Returning cached result for volume:", volume)
                           ;
10                     return this._cache.get(volume);
11                 }
12
13                 console.log("Calling actual method on BarrelCalculator...");
14                 const result = target[property].call(target, volume);
15                 this._cache.set(volume, result);
16                 return result;
17             };
18         }
19         return Reflect.get(target, property, receiver);
20     },
```

```
21 });
```

Listing 6.6: Using Proxy() in the *Proxy* pattern

Often `Proxy` objects are used alongside the `Reflect` object (like as showed in the listing 6.6), which contains a suite of methods mirroring the `Proxy` traps (for instance, `Reflect.get`). These methods provide reflective semantics for invoking the corresponding internal object methods. Crucially, invoking a `Reflect` method on a `Proxy` does not bypass the proxy, it continues to interact with the underlying object via its internal methods. Consequently, if a `Reflect` method is called inside a `Proxy` trap and is intercepted again by that same trap, an infinite recursion loop may occur.

6.3.2.2 Improved but still structurally tied: Flyweight, Mediator, Adapter, Bridge, Memento, Façade, Abstract Factory

This group of pattern have some improvements over the tradicial OO implementation but still have some limitations. Gained some modularity improvements via JavaScript's advanced mechanisms, but still have some limitations and still require explicit and static structural dependencies or participant dependencies in many cases.

The *Abstract Factory* is good example of this group (implementation in the listing 5.6), where the use of mixins to group related objects under a common "family", reflection and ducktyping to emmulate the need of interfaces and abstract classes. However, Factory logic still exists inside a dedicated factory function, meaning adding new product types requires modifying the factory object. In other words, flexibility was improved by allowing new product types can be registered dynamically but Factory logic remains tied to specific product families.

6.3.2.3 Limited JavaScript Benefits: Builder, Command, Chain of Responsibility, Template Method, Factory Method

While JavaScript advanced mechanisms improve some patterns, others show limited benefits. This is particularly true for this group, despite benefiting from JavaScript's function-based and prototype-based features, retained much of their traditional structure and limitations. They fall into the middle, not so rigid but don't fully take advantage of JavaScript's Strength. Mostly these patterns rely on predefined method calls and structured execution flow, limiting the impact of JavaScript's delegation and dynamic behavior. They still require explicit participant relationships, preventing full decoupling and they don't fully align with JavaScript's prototype-based and function-first paradigm, making their structure less adaptable to JavaScript's dynamic features.

A good example is the *Chain of Responsibility*, while benefiting from JavaScript's reflection mechanisms (`Reflect.has()`) and function-based handler registration, still retains structural constraints that limit its ability to fully leverage JavaScript's dynamic nature. The pattern's primary goa, decoupling request senders from receivers by passing

requests along a chain of handlers, remains intact in JavaScript, but the execution flow and explicit chaining structure still resemble its classical OOP counterpart.

Nonetheless, JavaScript allows for more flexible handler composition by replacing rigid class-based hierarchies with dynamic handler registration. Instead of requiring each handler to explicitly reference the next handler, JavaScript's `ComplaintResolver` class dynamically maintains a list of handlers and iterates over them, checking which one should handle the request.

```
class ComplaintResolver {
    constructor() {
        this.complaintListeners = [];
    }

    addHandler(handler) {
        if (Reflect.has(handler, "IsInterestedInComplaint") && Reflect.has(
            handler, "ListenToComplaint")) {
            this.complaintListeners.push(handler);
        } else {
            throw new Error("Handler does not implement required methods.");
        }
    }

    ResolveComplaint(complaint) {
        for (const handler of this.complaintListeners) {
            if (handler.IsInterestedInComplaint(complaint)) {
                return handler.ListenToComplaint(complaint);
            }
        }
        return "No resolution found for the complaint.";
    }
}
```

Listing 6.7: Part of the update version of the *Chain of Responsibility*

From the listing 6.7 it is possible to note that the handler registration is dynamic, handlers are added at runtime using `addHandler()`, the use of reflection (`Reflect.has()`) prevents adding invalid handlers, improving extensibility and the `ResolveComplaint` dynamically checks each handler, allowing for flexible complaint routing.

6.3.3 Patterns that did not improve: Visitor, Singleton, Interpreter

Patterns in this category did not gain significant modularity benefits from JavaScript. They still require participant classes to have explicit dependencies, cannot fully decouple behavior, or don't use JavaScript's strengths effectively. Most of the restriction comes from the pattern itself, not from the language.

Visitor still requires explicit `accept()` methods in each element, keeping visitor logic separate across participant classes, meaning new visitors require modifying every element class. JavaScript does not remove the need for explicit visitor calls, keeping *extensibility*

low *Singleton* remains tied to a global instance, which participants directly reference, preventing full encapsulation, as JavaScript cannot enforce single, instance constraints more effectively than OO. *Interpreter* pattern relies on a fixed and static grammar parsing, requiring hard coded class structures, making it difficult to adapt to dynamic language features.

7

Related Work

In this chapter, we go through some relevant studies related to the topic of this thesis, in terms of design patterns implementations, modularity and OOP. We start by presenting some studies about the implementation of the GoF design patterns in different programming languages, that we get from a lightweight systematic and organized search on Google Scholar. Then, the evaluation about their performance and an overview in the different mechanism in the different programming languages.

7.1 Lightweight systematic search on Google Scholar

We utilized Google Scholar for our searches and established specific criteria to ensure efficient and effective results. These criteria include the publication timeline-older papers often represent the beginning of the topic, while recent papers provide updated information. We also considered the authors involved and the programming languages used in the analysis of design patterns. By applying these criteria, we were able to expedite the process of finding relevant and high-quality papers.

For the search we used the following settings: all results between the year 2000 and 2023, disable the 'include citations' and used these search strings: design pattern module modularity, modularity "design pattern" and programming language with design patterns "design patterns". We also removed some words from the search like intelligence, human and ontology when we used the word modularity in the search string because modularity is a term that is also used in others scientific areas.

7.2 Design Patterns in other Programming Languages

The study by Jan Hannemann and Gregor Kiczales [18] digs into the utilization of design patterns in Java, drawing comparisons with AspectJ, an aspect-oriented extension. Hannemann demonstrates that AspectJ, when applied to various GoF patterns, show modularity improvements in 17 out of 23 patterns, offering a more modular, local, reusable, composable and (un)pluggable solution (see more in 6.1), these are the properties they used to

evaluate modularity support for the patterns. They also divided the patterns in to two different roles, the *defining* and *superimposed*, where defining means that the participants have no functionality outside the pattern and superimposed means that the participants have functionality outside the pattern.

Hridesh Rajan's exploration of the Eos language [39] extends the analysis of design pattern implementations beyond AspectJ. Rajan investigates all 23 GoF patterns in Eos, aiming to evaluate the impact of novel programming language constructs on these implementations. Drawing upon scenarios from Hannemann and Kiczales' prior research, Rajan conducts a comparative analysis, focusing on modularity properties. His findings indicate that while Eos has improved up to 7 out of 23 patterns compared to AspectJ, it performs similarly for the remaining 16 patterns. Rajan attributes these improvements to Eos' capacity to explain the intent of design patterns more effectively, suggesting promising advancements in AOP languages.

Based on the two previous studies, Monteiro and Gomes' paper [31], introduces the implementation of GoF patterns in the Object Teams programming language (OT/J), an aspect-oriented language compatible with Java. The study compares OT/J implementations with those in Java and AspectJ, including collections created by Hannemann and Kiczales for both languages, as well as a Java collection by James Cooper. Their findings reveal OT/J's significant advancements in flexible module extensibility, instance-level composition and cohesive module encapsulation, surpassing the capabilities of Java and AspectJ implementations. They also added one property to the modularity properties used by Jan Hannemann and Gregor Kiczale in their study, the *extensibility* (more about this property in 6.1).

In their exploration of the Go programming language's applicability to design patterns, Schmager in their 2010 publication 'GoHotDraw: Evaluating the Go Programming Language with Design Patterns' [42][41], utilize the HotDraw framework to implement all 23 GoF patterns. This study, conducted just one year after Go's release, showing the Go-specific features like embedding and interface inference, demonstrating their efficacy in pattern implementation despite disparities in the object model compared to traditional OO languages like Java. The paper not only discusses the successful implementation of all GoF design patterns but also delves into specific patterns such as Singleton, Adapter and Template Method, offering insights into Go programming idioms that may evolve into design patterns.

The thesis by Kristian Pedersen [37] explores the impact of different programming languages on the implementation of Gang of Four (GoF) design patterns, specifically focusing on Python, JavaScript, C#, Go and Smalltalk. Pedersen's work aims to identify how language-specific features influence the implementation quality of design patterns like Composite, Prototype, Adapter and Decorator. Regarding JavaScript, the thesis highlights its flexible typing and minimalistic syntax as advantageous for implementing design patterns, allowing for concise and adaptable code structures. Pedersen notes similarities between JavaScript and Python, both performing well due to their dynamic

nature and lightweight structures, which simplify the adaptation and implementation of complex design patterns. He also emphasizes that we should see the results of the study with a little caution, as the study was based on only four of the 23 GoF patterns.

Diogo Escaleira [3] in his thesis conducted a comprehensive evaluation of Octave's OO programming capabilities by implementing two complete collections of the GoF design patterns. His work emphasizes Octave's syntactic and semantic limitations compared to Java, particularly regarding interfaces, polymorphism and class composition. Despite these constraints, Escaleira identified viable patterns and adaptations, leveraging Octave's `classdef` constructs and dynamic typing. The thesis offers a structured analysis of OO principles, modularity, inheritance, encapsulation and abstraction, within Octave's environment, contributing a foundational assessment of its suitability for complex software development through design patterns.

8

Conclusions and Future Work

In this chapter, we present the conclusions of this thesis and discuss some future work that can be done to improve the results obtained.

8.1 Conclusions

This thesis evaluated the modularity support of JavaScript through the implementation of 2 sets with 23 design patterns each using advanced mechanisms such as mixins, object delegation, duck typing and reflection. By comparing original object-oriented implementations with updated versions that leverage JavaScript specific features, the aim was to assess whether these mechanisms enhance modularity based on established properties.

The findings show that JavaScript's flexibility allow improvements in modularity, particularly in *(Un)pluggability*, *Reusability* and *Extensibility*, but not uniformly across all patterns. 19 out of 23 patterns showed improvements over the traditional by leveraging the JavaScript composotion mechanisms. Certain patterns, such as *Observer*, *Decorator*, *Proxy*, *Strategy*, *State* and *Iterator* benefit significantly from JavaScript's features.

Not all patterns experienced gains. Patterns such as *Visitor*, *Singleton* and *Interpreter* remained largely unchanged in their modularity characteristics making the update versions of these pattern not bettern than the classical implementations, even with JavaScript's dynamic features. In fact, JavaScript's lack of strict module boundaries sometimes makes these patterns harder to isolate or reuse compared to traditional object-oriented languages.

The *Prototype* stands out as a unique case of direct language support where JavaScript's object delegation natively provides the behavior the pattern is meant to simulate, making the used of this pattern in spectrum of the JavaScript unnecessary.

During this study, it became clear that modularity properties such as *Locality* are difficult to apply at the module level in JavaScript. Unlike in static languages, JavaScript rarely separates pattern code from participant logic into strict modules. Therefore, the modularity evaluation was made at the participant level, differentiating between pattern code and participant specific behavior. This approach captures best how patterns are

structured in JavaScript, reflecting a more realistic application of the modularity criteria.

Summing up, JavaScript's dynamic and prototype-based nature does offer benefits for modularity, but its advantages are pattern dependent and often rely on using its advanced mechanisms intentionally. A general improvement across all patterns is not possible and evaluating modularity in JavaScript requires a more participant level and context specific perspective than in class-based languages.

8.2 Future Work

While this thesis provides a a modularity centreded assessment of JavaScript using design patterns, there are several directions that future work can explore to futher deepen the understanding of modularity in dynamic languages:

- **Formal Metric Evaluation**: This thesis rely on qualitative analysis and structural comparison. Future work could include metric analysis (for example, lines of code, coupling and cyclomatic complexity) using tools specifically adapted for JavaScript to validate modularity improvements with quantitative data.
- **Modularity Evaluation in TypeScript**: TypeScript adds static typing and interfaces to JavaScript, it would be interesting to explore whether TypeScript can enforce or improve modularity in ways that JavaScript alone cannot, especially for patterns like *Visitor* or *Template Method*.
- **Framework Analysis**: Applying the modularity evaluation to frameworks (React, Vue.js or Node.js) could reveal how patterns behave in production scenarios, where patterns are often used in combination or adapted with specific tools.
- **Compare with other Dynamic Languages**: comparing JavaScript to other dynamic languages like Python or Ruby could reveal how universal these modularity challenges are and whether different language features (Ruby's mixins or Python's decorators) offer better support for certain patterns.
- **Measuring Runtime Modularity Effects**: Modularity is usually treated as a design or code structure concern, but in JavaScript, dynamic behaviors (like delegation or reflection) may have runtime costs. Future work could measure how reflection or delegation affects performance, memory usage, or execute time. Identify tradeoffs between modularity and runtime efficiency in patterns that use `Reflect`, `Object.create()`, or `Proxy`.

Bibliography

[1] C. Alexander. *A pattern language: towns, buildings, construction*. Oxford university press, 1977 (cit. on p. 39).

[2] C. Alexander. *A pattern language: towns, buildings, construction*. Oxford university press, 2018 (cit. on p. 39).

[3] D. de Almeida Escaleira. "Assessment of Octave's OO Features Based on GOF Patterns". MA thesis. Universidade NOVA de Lisboa (Portugal), 2023 (cit. on p. 74).

[4] K. Beck, M. Fowler, and G. Beck. "Bad smells in code". In: *Refactoring: Improving the design of existing code* 1.1999 (1999), pp. 75–88 (cit. on p. 24).

[5] G. Bracha and W. Cook. "Mixin-based inheritance". In: *ACM Sigplan Notices* 25.10 (1990), pp. 303–311 (cit. on p. 11).

[6] C. Chambers. "Object-oriented multi-methods in Cecil". In: *European Conference on Object-Oriented Programming*. Springer. 1992, pp. 33–56 (cit. on p. 9).

[7] *Classes*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes (visited on 2024-07-07) (cit. on p. 16).

[8] *Delegation (object-oriented programming)*. URL: https://en.wikipedia.org/wiki/Delegation_%28object-oriented_programming%29 (visited on 2024-07-07) (cit. on pp. 24, 28).

[9] *Delegation (object-oriented programming)*. URL: https://labviewwiki.org/wiki/Delegation_%28object-oriented_programming%29 (visited on 2024-07-07) (cit. on pp. 24, 28).

[10] *Design pattern in JavaScript by Filipe Beline*. URL: https://github.com/fbeline/design-patterns-JS (visited on 2024-02-12) (cit. on p. 37).

[11] D. Diaz and R. Harmes. *Pro JavaScript design patterns*. Apress, 2008 (cit. on pp. 1, 2, 13, 37).

[12] *Does JavaScript have the interface type (such as Java's 'interface')?* URL: https://stackoverflow.com/questions/3710275/does-javascript-have-the-interface-type-such-as-javas-interface (visited on 2024-07-07) (cit. on p. 20).

[13] *Double dispatch*. URL: https://en.wikipedia.org/wiki/Double_dispatch (visited on 2024-07-07) (cit. on p. 30).

[14] *Duck typing*. URL: https://en.wikipedia.org/wiki/Duck_typing (visited on 2024-07-07) (cit. on p. 5).

[15] E. Ernst. "Family polymorphism". In: *European Conference on Object-Oriented Programming*. Springer. 2001, pp. 303–326 (cit. on p. 7).

[16] E. Gamma et al. "Elements of reusable object-oriented software". In: *Design Patterns* (1995) (cit. on pp. 1, 20, 39).

[17] *Grammar and types*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types (visited on 2024-07-07) (cit. on p. 14).

[18] J. Hannemann and G. Kiczales. "Design pattern implementation in Java and AspectJ". In: *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2002, pp. 161–173 (cit. on pp. 1, 2, 10, 61, 72).

[19] D. Herman and S. Tobin-Hochstadt. "Modules for JavaScript". In: *Preprint, April* (2011) (cit. on p. 1).

[20] *History of JavaScript*. URL: https://www.geeksforgeeks.org/history-of-javascript/ (visited on 2024-07-01) (cit. on p. 1).

[21] *Inheritance and the prototype chain*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain (visited on 2024-07-07) (cit. on pp. 25, 26).

[22] *Is it possible to dispatch multiple actions?* URL: https://stackoverflow.com/questions/68823462/is-it-possible-to-dispatch-multiple-actions (visited on 2024-07-07) (cit. on p. 31).

[23] *JavaScript Reflect*. URL: https://playcode.io/javascript/reflect (visited on 2024-07-07) (cit. on p. 34).

[24] H. Lieberman. "Using prototypical objects to implement shared behavior in object-oriented systems". In: *Conference proceedings on Object-oriented programming systems, languages and applications*. 1986, pp. 214–223 (cit. on pp. 16, 28).

[25] R. E. Lopez-Herrejon, D. Batory, and W. Cook. "Evaluating support for features in advanced modularization technologies". In: *ECOOP 2005-Object-Oriented Programming: 19th European Conference, Glasgow, UK, July 25-29, 2005. Proceedings 19*. Springer. 2005, pp. 169–194 (cit. on pp. 10, 61).

[26] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[27] O. L. Madsen and B. Moller-Pedersen. "Virtual classes: A powerful mechanism in object-oriented programming". In: *Conference proceedings on Object-oriented programming systems, languages and applications*. 1989, pp. 397–406 (cit. on p. 8).

[28] P. Maes. "Concepts and experiments in computational reflection". In: *ACM Sigplan Notices* 22.12 (1987), pp. 147–155 (cit. on p. 34).

[29] *Meta programming*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Meta_programming (visited on 2024-07-07) (cit. on p. 34).

[30] *Modularity and its Properties*. URL: https://www.geeksforgeeks.org/modularity-and-its-properties/ (visited on 2024-07-07) (cit. on p. 10).

[31] M. P. Monteiro and J. Gomes. "Implementing design patterns in Object Teams". In: *Software: Practice and Experience* 43.12 (2013), pp. 1519–1551 (cit. on pp. 1, 2, 7, 10, 30, 61, 73).

[32] *Multiple inheritance and multiple dispatch in JavaScript*. URL: https://lisperator.net/blog/multiple-inheritance-and-multiple-dispatch-in-javascript/ (visited on 2024-07-07) (cit. on pp. 30, 31).

[33] *Object prototypes*. URL: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes (visited on 2024-07-07) (cit. on p. 25).

[34] *Object.assign*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign (visited on 2024-07-07) (cit. on p. 11).

[35] D. Odell. *Pro JavaScript development: coding, capabilities, and tooling*. Apress, 2014 (cit. on pp. 1, 2, 37).

[36] A. Osmani. *Learning JavaScript design patterns*. " O'Reilly Media, Inc.", 2023 (cit. on pp. 1, 2, 42).

[37] K. B. Pedersen. "How Implementation Language Affects Design Patterns: A Comparison of Gang of Four Design Pattern Implementations in Different Languages". MA thesis. 2019 (cit. on pp. 1, 73).

[38] *Proxy API*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy (visited on 2005-03-20) (cit. on p. 68).

[39] H. Rajan. "Design pattern implementations in Eos". In: *Proceedings of the 14th Conference on Pattern Languages of Programs*. 2007, pp. 1–11 (cit. on pp. 1, 2, 73).

[40] *Reflect*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect (visited on 2024-07-07) (cit. on p. 34).

[41] F. Schmager. "Evaluating the go programming language with design patterns". MA thesis. 2010 (cit. on p. 73).

[42] F. Schmager, N. Cameron, and J. Noble. "Gohotdraw: Evaluating the go programming language with design patterns". In: *Evaluation and usability of programming languages and tools*. 2010, pp. 1–6 (cit. on pp. 1, 73).

[43] *Separation of concerns*. URL: https://en.wikipedia.org/wiki/Separation_of_concerns (visited on 2024-07-05) (cit. on p. 1).

[44] D. Skoko. *Applying design patterns and testing it in JavaScript*. 2018 (cit. on p. 1).

[45] *super*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super (visited on 2024-07-07) (cit. on p. 17).

[46] S. Timms. *Mastering JavaScript design patterns*. Packt Publishing Ltd, 2016 (cit. on pp. 1, 2, 33, 36, 37).

[47] *TIOBE index*. URL: https://www.tiobe.com/tiobe-index/ (visited on 2024-07-01) (cit. on p. 1).

[48] D. Ungar and R. B. Smith. "Self: The power of simplicity". In: *Conference proceedings on Object-oriented programming systems, languages and applications*. 1987, pp. 227–242 (cit. on pp. 16, 28).

[49] *Virtual function*. URL: https://en.wikipedia.org/wiki/Virtual_function (visited on 2024-07-07) (cit. on p. 8).

[50] *Visual Studio Code*. URL: https://code.visualstudio.com/ (visited on 2024-07-01) (cit. on p. 2).

[51] *What is concept of reflection in JavaScript?* URL: https://stackoverflow.com/questions/53170245/what-is-concept-of-reflection-in-javascript (visited on 2024-07-07) (cit. on p. 34).

[52] *What is Method Dispatch?* URL: https://stackoverflow.com/questions/1805510/what-is-method-dispatch (visited on 2024-07-07) (cit. on p. 9).

[53] *What is the difference between polymorphism and duck typing?* URL: https://stackoverflow.com/questions/11502433/what-is-the-difference-between-polymorphism-and-duck-typing (visited on 2024-07-07) (cit. on p. 6).

[54] *Working with objects*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_objects#inheritance (visited on 2024-07-07) (cit. on p. 18).