Evaluating the Go Programming Language with Design Patterns

by

Frank Schmager

A thesis
submitted to the Victoria University of Wellington
in partial fulfilment of the
requirements for the degree of
Master of Science
in Computer Science.

Victoria University of Wellington
2010

Abstract

GO is a new object-oriented programming language developed at Google by Rob Pike, Ken Thompson, and others. GO has the potential to become a major programming language. GO deserves an evaluation.

Design patterns document reoccurring problems and their solutions. The problems presented are programming language independent. Their solutions, however, are dependent on features programming languages provide.

In this thesis we use design patterns to evaluate GO. We discuss GO features that help or hinder implementing design patterns, and present a pattern catalogue of all 23 Gang-of-Four design patterns with GO specific solutions.

Furthermore, we present GoHotDraw, a GO port of the pattern dense drawing application framework JHotDraw. We discuss design and implementation differences between the two frameworks with regards to GO.

Acknowledgments

I would like to express my gratitude to my supervisors, James Noble and Nicholas Cameron, whose expertise, understanding, and patience, added considerably to my graduate experience. They provided timely and instructive comments and evaluation at every stage of my thesis process, allowing me to complete this project on schedule.

I would also like to thank my family for the support they provided me through my entire life and in particular, I owe my deepest gratitude to my beautiful wife Mary, without whose love and moral support I would not have finished this thesis.

Preliminary work for this thesis has been conducted in collaboration with my supervisors and published at the PLATEAU 2010 workshop [83].

Frank Schmager

Contents

Chapter 1

Introduction

GO is a new programming languages developed at Google by Robert Griesemer, Rob Pike, Ken Thompson, and others. GO was published in November 2009 and made open source; was "Language of the year" 2009 [7]; and was awarded the Bossie Award 2010 for "best open source application development software" [1]. GO deserves an evaluation.

Design patterns are records of idiomatic programming practice and inform programmers about good program design. Design patterns provide generic solutions for reoccurring problems and have been implemented in many programming languages. Every programming language has to solve the problems addressed by patterns. In this thesis we use design patterns to evaluate the innovative features of GO.

In this thesis we presents our experiences in implementing the design patterns described in *Design Patterns* [39], and the JHotDraw drawing editor framework in GO. We discuss differences between our implementations in GO with implementations in other programming languages (primarily Java) with regards to GO features.

1.1 Contributions

We present:

- an evaluation of the GO programming language.
- GoHotDraw, a GO port of the pattern dense drawing application framework JHotDraw.
- a pattern catalogue of all 23 Design Patterns described in *Design Patterns* [39].

1.2 Outline

The remainder of this thesis is organized as follows:

Chapter 2 gives background information concerning GO, design patterns, and programming language evaluation.

Chapter 3 discusses selected design patterns with regards to GO's language features.

Chapter 4 describes GoHotDraw and highlights design and implementation differences to its predecessor JHotDraw.

Chapter 5 evaluates GO.

Chapter 6 presents our conclusions, emphasizes this thesis's original contributions, and discusses possibilities for extending this work in the future.

Chapter 2

Background

In this chapter we give background information necessary for understanding the remainder of the thesis. In Section 2.1 we describe the history of GO — the languages and developers that influenced the design of GO — and give necessary information about GO language features; in Section 2.2 we introduce design patterns; and in Section 2.3 we review existing literature on language evaluation.

2.1 GO

GO is an object-oriented programming language with a C-like syntax. GO is designed as a systems language, has a focus on concurrency, and is "expressive, concurrent, garbage-collected" [4]. GO was developed by Rob Pike, Ken Thompson and others at Google in 2007 [5] and was made public and open source in November 2009. GO supports a mixture of static and dynamic typing, and is designed to be safe and efficient. A primary motivation is compilation speed [70].

2.1.1 History of GO

GO is inspired by many programming languages, environments, operating systems and the developers behind them. Ken Thompson developed the B programming language [50] in 1969, a predecessor of the C programming language [82] developed by Dennis Richie in 1972. C is the main programming language for the Unix operating system [55], the latter developed initially by Thompson, Richie and others. Many of today's languages have a C-like syntax (C++, Java, C♯).

Communicating Sequential Processes (CSP) [47] was developed by C. A. R. Hoare in 1978. Hoare introduces the idea of using channels for interprocess communication. CSP's channels are unbuffered, i.e. a process delays until another process is ready for either input or output on a channel. CSP's channels are also used as guards for process synchronization: a process waits for multiple processes to finish by listening on multiple channels for input and continues once a signal has arrived on a channel. CSP's approach to concurrency and interprocess communication was highly influential for, amongst others, Occam [27] and Erlang [91].

In 1985 Luca Cardelli and Rob Pike developed Squeak [23], a language to demonstrate the use of concurrency for handling input streams of user interfaces. Squeak was limited in that it didn't have a type system, dynamic process creation or dynamic channel creation. Pike redesigned Squeak as Newsqueak in 1989 remedying Squeak's limitations [69]. Newsqueak was syntactically close to C, had lambda expressions and the `select` statement for alternations. Newsqueak introduced `:=` for declaration-and-assignment and the left arrow `<-` as communication operator (the arrow points in the direction information flows) [74].

In 1992 Thompson, Winterbottom and Pike developed the operating system Plan 9 [75]. Plan 9 was intended to become the Unix successor. Plan 9 was widely used within Bell Labs, and provided the programming language Alef [95], developed by Winterbottom. Alef is a compiled C-like language with Newsqueak's concurrency and communications model.

Unfortunately Alef was not garbage collected and concurrency was hard to do with the C-like memory model.

Winterbottom and Pike went on to develop Inferno [31] in 1995, a successor to Plan 9. They also developed the Limbo programming language [32] used to develop applications for Inferno. For Limbo, Winterbottom and Pike reused and improved Alef's abstract data types and Newsqueak's process management [53].

Thompson and Pike's experience and involvement in the development of Unix, C, Plan 9, Inferno and Limbo have clearly influenced the design of GO. GO draws its basic syntax from C. GO's interprocess communication is largely influenced by CSP and its successors as outlined above. GO uses typed channels which can be buffered and unbuffered; GO's `select` statement implements the process synchronization mechanism described by Hoare and implemented in Newsqueak. GO's compiler suite is based on Plan 9's compiler suite. For declarations and modularity, GO drew inspiration from the Pascal, Modula and Oberon programming languages [96, 97], developed by Wirth between 1970 and 1986.

2.1.2 Overview of GO

```
package main

import "fmt"

func main() {
  fmt.Println("Hello World!")
}
```

The above listed GO program prints "Hello World!" on the console. The package declaration is part of every GO file. GO's syntax is C-like with minor variations (e.g., optional semicolons, identifiers are followed by their type). Functions are declared with the keyword `func`. The entry point to every GO program is the function `main` in package `main`.

GO is systems programming language with focus on concurrency. Every

function can be run concurrently with the keyword `go`. Concurrently running functions are called goroutines. Goroutines are lightweight and multiplexed into threads. Goroutines communicate via channels. GO is garbage collected, and supports a mixture of static and dynamic typing. GO has pointers, but no pointer arithmetic. Function arguments are passed by value, except for maps and arrays.

Objects

GO is an object-oriented language. GO has objects, and structs rather than classes. The following code example defines the type `Car` with a single field `affordable` with type `bool`:

```
type Car struct {
  affordable bool
}
```

Methods

GO distinguishes between methods and functions. Methods are functions that have a receiver, and the receiver can be any valid variable name. Every type in GO can have methods. Following C++, rather than Java, methods in GO are defined outside struct declarations. Multiple return values are allowed and may be named; the receiver must be explicitly named. GO does not support overloading or multi-methods.

The following listing defines the method `Refuel` for `Car` objects:

```
func (this Car) Refuel(liter int) (price float) {...}
```

Note that the method `Refuel` is defined for type `Car`. Objects of pointer type `*Car` do not implement this method, the base type and the pointer type are distinct. However, if a method is defined for objects of pointer type, object of the base type automatically have the method too.

Embedding

GO has no class-based inheritance, code reuse is supported by *embedding*. The embedding type gains the embedded type's methods and fields. Types can be embedded in structs by omitting the identifier of a field. For example, the following code shows a `Truck` type which embeds the `Car` type and, therefore, gains the method `Refuel`:

```
type Truck struct {
  Car
  affordable bool
}
```

If a method or field cannot be found in an object's type definition, then embedded types are searched, and method calls are forwarded to the embedded object. This is similar to subclassing, the difference in semantics is that an embedded object is a distinct object, and further dispatch operates on the embedded object, not the embedding one.

Objects can be deeply embedded and multiply embedded. Name conflicts are only a problem if the ambiguous element is accessed. The example above could be written as:

```
type Truck struct {
  car Car
  affordable bool
}
```

Here, `Truck` does not embed `Car`, but has an element `car` of type `Car`. `Truck` could provide a `Refuel` method and manually delegating the call to `car`:

```
func (this *Truck) Refuel(liter int) (price float) {
  return this.car.Refuel(liter)
}
```

Interfaces

Interfaces in GO are abstract representations of behaviour — sets of method signatures. An object satisfies an interface if it implements the methods of

an interface. This part of the type system is structural: no annotation by the programmer is required. Interfaces are allowed to embed other interfaces: the embedding interface will contain all the methods of the embedded interfaces. There is no explicit subtyping between interfaces, but since type checking for interfaces is structural and implicit, if an object implements an embedding interface, then it will always implement any embedded interfaces. Every object implements the empty interface `interface{};` other, user-defined empty interfaces may also be defined. The listing below defines `Refuelable` as an interface with a single method `Refuel()`.

```
type Refuelable interface {
  Refuel(int) float
}
```

Both `Car` and `Truck` implement this interface, even though it is not listed in their definitions (and indeed, `Refuelable` may have been defined long afterwards). Type checking of non-interface types is not structural: a `Truck` cannot be used where a `Car` is expected.

Dynamic Dispatch

Dynamic dispatch and method calls with embedding in GO work differently than with inheritance in other languages.

Consider the classes `S` and `T`, where `S` extends `T`. `T` defines the public methods `Foo` and `Baz` with `Baz` calling `Foo`. The listing below shows class `T` in pseudo-code:

```
class T {
  void Baz() {
    this.Foo()
  }

  void Foo() {
    print("T")
  }
}
```

The class `S` is a sub-class of `T`. `S` inherits `Baz` and overrides `Foo`:

```
class S extends T {
  void Foo() {
    print("S")
  }
}
```

With class based inheritance calling `Baz` on an object of type `S` will print "S". The dynamic type of `s` is `S`, so `Baz` will call `S`'s `Foo` method:

```
T s = new S();
s.Baz() //prints S
```

Embedding is GO's mechanism for code reuse. The dispatch of message calls with embedding is different from that of class-based inheritance. Consider the types `S` and `T`, with `S` embedding `T`:

```
type T struct {}
type S struct {
  T
}
```

We define the method `Baz` on type `T`:

```
func (this T) Baz() {
  this.Foo()
}
```

Type `S` implements the method `Foo` printing "S":

```
func (this S) Foo() {
  print("S")
}
```

Type `S` gains the method `Baz`, because `S` embeds `T`. The call `s.Baz()` prints "T", whereas with class based inheritance "S" would be printed:

```
s := new(S)
s.Baz() //prints T
```

Packages

Source code organization and information hiding in GO is defined with packages. The source code for a package can be split over multiple files. A package resides in its own folder. A folder should only contain a single

package. Packages cannot be nested. Folders can be nested to group packages, but the packages have to be imported and addressed individually. Wildcards cannot be used for importing packages, every package has to be listed individually.

Information hiding

GO provides two scopes of visibility: package and public. A package private member is only visible within a package. Client code residing outside the package cannot access package private members. Public members can be accessed by package-external clients. The visibility is defined by the first letter of an identifier. A lower case fist letter of the identifier renders the member package private, an upper case first letter renders the member public.

Object Creation

GO objects can be created from struct definitions with or without the `new` keyword. Fields can be initialised by the user when the object is initialised, but there are no constructors.

Consider type T with two members a and b:

```
type T struct {
  Name string
  Num int
}
```

To initialize `T`'s members, composite literals can be used. A composite literal is the type name followed by curly braces and returns an instance of the type. The parameters between the braces refer to the types members. The parameters refer to the members in order they are declared. Composite literals to initialize members can only be used within the same package or on public members.

An object of type `T` can also be created with the built-in function `new()`. `new(T)` allocates zeroed storage for a new object of type `T` and returns a

pointer to it. The created object is of pointer type `*T`.

Newly created objects in GO are always initialised with a default value (`nil`, `0`, `false`, `""`, etc.). There are multiple ways to create objects. On each line in the following listing, an `T` object is created and a pointer to the new object stored in `t`. The ampersand `&` operator returns the memory address of a value. In each case the fields have the following values:

`t.Name == ""` and `t.Num == 0`.

```
var t *T
t = new(T)
t = &T{}
t = &T{"",0}
t = &T{Name:"",Num:0}
```

Multiple assignment/Multiple return parameters

GO supports multiple assignment:

```
a, b := "hello", 42
```

and multiple return values:

```
func Foo() (string, int) {...}
...
a, b := Foo()
```

This functionality can be used in functions and methods or can be used to check and get an object from a map. In the following listing `elements` is a map. Multiple assignment is used to get the `element` belonging to `key`. The second parameter (`isPresent`) is to distinguish between returning a `nil` object belonging to a key or because the key does not exist. Note that the first return parameter could be `nil` even though the second is `true`, since values stored in a map can be `nil`.

```
element, isPresent := elements[key]
if !isPresent {
  ...
}
```

2.2 Design Patterns

In software engineering certain problems reoccur in different applications and common solutions re-emerge. Patterns are a description of reoccurring design problems and general solutions for the problem. Patterns originate from work on urban design and building architecture from Christopher Alexander [10]. In 1987, Ward Cunningham and Kent Beck started first applying patterns to software design [13]. Software patterns became popular with *Design Patterns* [39] by Gamma, Helm, Johnson and Vlissides — the "Gang-of-Four" (GoF) — published in 1994. Design patterns are the most well known kind of software patterns.

A pattern consists usually of a name; a concise, abstract description of the problem; an example; details of the implementation; advantages and disadvantages; and known uses.

The patterns described in *Design Patterns* are object-oriented design patterns, showing the relationships and interactions between objects and classes. The problems address by patterns are mostly programming language independent, the solutions, however, differ depending on the features of a programming language. The GoF patterns have been implemented in many programming languages (e.g. Java [65], JavaScript [46], Ruby [67], Smalltalk [11], C♯ [15]).

Design patterns are just one category of software patterns, other categories are: software architecture [22], concurrency and networking [84], resource management [56] and distributed computing [21]. Even whole pattern languages have emerged [26, 93, 61, 34, 60].

2.2.1 Design patterns vs Language features

Norvig discusses design patterns and how features of the Dylan programming language can make the patterns simpler or invisible [66]. He claims that 16 of the 23 design patterns can be simplified, but his slides give no more information about how he arrived at these numbers. Amongst

other features Norvig exploits Dylan's multimethods, syntax abstraction mechanisms and runtime class creation to implement design patterns.

Gil and Lorenz argue that patterns can be grouped according to how far the pattern is from becoming a language feature [42]. Their groups are clichés (trivial mechanisms not worth implementing as a language feature), idioms (patterns that have already become features of some languages) and cadets (patterns that are not mature enough to be language feature yet).

Bosch states that programming languages should have facilities to support implementations of design patterns [17]. He identified problems implementing design patterns in C++ and proposes an extended object model to remedy the shortcomings.

Agerbo and Cornils are of the opinion that the number of design patterns should be minimal [8]. They develop four guidelines for evaluating patterns: design patterns should be: domain independent, not an application of another design pattern, language independent, and not a language construct. They analyse patterns and conclude that 12 of the 23 the Gang of Four patterns fulfil their guidelines. In a later work they call these patterns fundamental design patterns [9].

Bünning et. al. address the problem that pattern implementations can be hard to trace and hard to reuse [20]. They propose the language PaL that elevates patterns to first class elements. They believe patterns can be encapsulated by class structures enabling reuse and traceability.

Chambers, Harrison and Vlissides discuss the relationship and importance of patterns, languages, and tools [24]. They agree that the role of design patterns is mainly for communication between humans. They disagree over how far languages and tools should go in aiding programmers to apply patterns.

Bishop focusses on the relationship of abstraction and patterns [16]. She implements a selection of patterns in C♯ to convey her point that the higher-level the functionality used to implement the pattern, the easier and more understandable are the implementations of the patterns. Bishop

argues for higher-level functionality in programming languages to make pattern implementations more straight forward.

2.2.2 Client-Specified Self

The Client-Specified Self pattern enables programmers to simulate dynamic dispatch in languages without inheritance by replacing method sends to method receivers with method sends to an argument [90]. The "self" refers to Smalltalk's pseudovariable `self` (`this` in Java, C++ or C♯). The pattern allows the sender to effectively change the value of the recipient's `self`.

Consider the example given in Section 2.1.2. `S` embeds `T`. `S` overrides `T`'s `Foo` method. The embedded type `T` is actually an object of type `T`, the call that has been forwarded to the embedded object stays in that object. The method `Baz` is defined on type `T` and the receiver object of `Baz` is `self`. `self` is of type `T`. The call `self.Foo()` will call `T`'s `Foo` method, printing "T". To achieve the same behaviour as with class based inheritance, the actual receiver object has to be passed as an additional parameter.

We define an interface `I` that both type `S` and `T` implement:

```
type I interface {
  Foo()
}
```

The changes necessary are bold in the next listing:

```
func (self B) Baz(i I) {
  i.Foo()
}

a.Baz(a)
```

The parameter `i` of interface type `I` is added and the receiver of the `Foo` call is the object `i`, not `self`. The client explicitly defines the receiver of following method calls.

2.3 Language Evaluations

2.3.1 Language Critiques

In his letter "Go To Statement Considered Harmful" [30] Edsger W. Dijkstra critizises a single programming language feature: `goto` statements. He argues that structural programming with conditional clauses and loops should be used instead of excessively using `goto` statements. He bases his critique on the need to be able to reason about the state of a program. According to him, `goto` makes such reasoning "terribly hard".

In "Lisp: Good News Bad News How to Win Big" [37] Richard P. Gabriel discusses why Lisp is not as successful as C. Gabriel bases his argument not only on language features, but he also includes the Lisp community as contributing factor. Gabriel's article gave rise to the idea that incremental improvement of a program and programming language might be better for their adoption and long term success than trying to produce something perfectly initially.

In "Why Pascal is Not My Favourite Programming Language" [54] Brian W. Kernighan points out deficiencies of the Pascal programming language. He bases his observations on a rewrite of a non-trivial Ratfor (Fortran) program in Pascal. Kernighan selectively chooses language features to support his conclusion that Pascal "is just plain not suitable for serious programming".

2.3.2 Evaluation of Programming Languages for Students

In the late 80's and early 90's a change in programming languages taught in undergraduate courses can be observed: the change from procedural to object-oriented. The early reports found in literature do not give much in terms of evaluation of which language to chose. Later reports develop and draw on a rich set of language evaluation qualities suitable for teaching.

Pugh et al. were among the first to apply the object-oriented approach

in their under-graduate courses [78]. Smalltalk was the language of choice, because it had a rich library, forced students to program in an object-oriented style, and came with an IDE. Others followed and described their experiences using Smalltalk to teach object-oriented programming [86, 87]. Their reasoning why they chose Smalltalk for teach matches mostly with Pugh's report (pure object-oriented, extensive library, IDE).

Decker and Hirschfelder present more pragmatic reasons why they chose Object Pascal for teaching [29]. They argue that the underlying concepts can be taught in many languages. They needed a language that supported multiple programming paradigms. They admit that C++ and Smalltalk were more popular with employers, but the faculty was used to Pascal.

None of the above reports present a comprehensive set of criteria underlying their decision for which language to choose for teaching. This was also observed by Mazaitis who surveyed academia and industry concerning the choice of programming language for teaching [63].

A relatively comprehensive evaluation of programming languages is presented by Kölling et al. [57]. They give a list of 10 requirements for a first year teaching language. They evaluate C++, Smalltalk, Eifel and Sather and come to the conclusion that none of these languages are fit for teaching. They go on and develop the programming language and IDE Blue [58, 59].

Further evaluations of programming languages follow: Hadjerrouit evaluates Java with seven criteria [44]; Gupta presents 18 criteria and a short evaluation of four languages (C, Pascal, C++, Java) [43]; Kelleher and Pausch present a taxonomy of languages and IDE catered to novices from languages used in industry like Smalltalk and COBOL to languages for children like Drape or Alice [52]; Parker et. al propose a formal language selection process with an extensive list of criteria [68] .

2.3.3 Empirical Evaluations

An early experiment comparing two programming languages to determine if statically typed or typeless languages affect error rates was conducted by Gupta [40]. Two programming languages have been designed with similar features: one statically typed and one typeless language. The subjects were divided in two groups and had to implement a solution for the same problem first with one and then with the other programming language. The number of errors made while developing with either language were compared with each other. Gupta finds that using a statically typed language can increase programming reliability.

Tichy argues that computer scientists should experiment more — experiments according to the scientific method [88]. He dismisses main arguments against experimentation by comparing computer science to other fields of science by giving examples strengthening the need for experiments, and by showing the shortcomings of alternative approaches.

Prechelt conducted a programming language comparisons by having groups of programmers develop the same program in different languages [76]. He compared Java with C++ in terms of execution time and memory utilization by comparing the implementations of the same program implemented by 40 students. Gat repeated Prechelt's experiment and had the same program implemented by different programmers in Lisp [41]. Gat found that the Lisp programs execute faster than C++ programs, and development time is lower in Lisp than in C++ and Java. He concludes that Lisp is an alternative to Java or C++ where performance is important. Prechelt extends the experiment to compare seven programming languages in [77] according to program length, programming effort, runtime efficiency, memory consumption. Prechelt observed amongst other findings, that designing and writing the program in C, C++, or Java takes twice as long as with Perl, Python, Rexx, or Tcl; with no clear differences in program reliabilty

McIver argues that languages should be evaluated in connection with

an IDE by conducting empirical studies observing the error rate [64]. Her evaluation is geared towards novice programmers.

In a more recent study Hanenberg studies the "impact" of the type system on development [45]. He has one group of developers implement a program in a statically typed language and the other group implementing the same program in a dynamically typed language. He uses development time and passed acceptance tests to measure the "impact". Hanenberg found no differences, in terms of development time, between static type systems and dynamic type systems.

2.3.4 Publications on GO

In the first public introduction of GO [70] Rob Pike lays out reasons why a new programming language was needed. Pike names goals for GO and points out shortcomings of until then existing languages. According to Pike, GO was needed because building software takes too long, existing tools are too slow, type systems are "clunky", and syntaxes are too verbose. GO is to be type- and memory safe, supporting concurrency and communication, garbage-collected and quick to compile. He places emphasis on GO's concise, clean syntax, lightweight type system and superior package model and dependency handling.

In another talk [71] Pike reasons about shortcomings of existing mainstream languages: mainstream languages have an "inherent clumsiness"; C, C++ and Java are "hard to use", "subtle, intricate, and verbose"; and "their standard model is oversold and we respond with add-on models such as 'patterns' ". Pike points out how languages like Ruby, Python, or JavaScript react to these problems. He identifies a niche for a new programming language and shows how GO tries to fill that niche with qualities that a language has to have to avoid deficiencies of existing languages. He explains GO concepts and functionality like interfaces, types, concurrency. Pike concludes by presenting achievements and testimonials of GO users.

At OSCON 2010 Pike gave two presentations on GO. The first talk lays out GO's history, the languages that influenced GO's approach to concurrency [74]. The second talk [72] is another overview of GO's type system (making comparisons with Java); GO's approach to concurrency and memory management; GO's development status.

In the most recent presentation of GO [73] Pike points out that GO is a reaction to Java's and C++'s "complexity, weight, noise" and JavaScript's and Python's "non-static checking". He backs up his claim that GO is a simple language, by comparing the number of keywords in various languages (with GO having the least). He gives the same examples of earlier talks and focusses on mainly the same functionality (interfaces and types, and concurrency).

Chapter 3

Design Patterns in GO

Design patterns are records of idiomatic programming practice and inform programmers about good program design. Design patterns have been the object of active research for many years [26, 92, 35, 85]. We use design patterns to evaluate the GO programming language. In this Chapter we present patterns from *Design Patterns* [39] which highlight the innovative features of GO.

In Section 3.1 we discuss why we are using patterns to evaluate GO; in Section 3.2 we discuss pattern where either embedding or composition should be used; in Section 3.3 we discuss patterns that rely on structural subtyping; in Section 3.4 we discuss patterns that can be implemented with GO's first class functions; in Section 3.5 we discuss how applying the Client-Specified Self pattern affects the Template Method pattern; in Section 3.6 we discuss patterns that make use of GO's approach to information hiding.

3.1 Why Patterns?

Many researchers have proposed methods and criteria for evaluating programming languages (see Section 2.3). The methods and criteria proposed in the literature are usually hard to measure and are prone to be subjective.

Design patterns encapsulate programming knowledge. The intent and

the problems patterns address, and furthermore the necessity for patterns, are programming language independent. Implementations of design patterns differ due to specifics of the language used. There is a need for programming language specific pattern descriptions as can be seen from the number of books published on the subject (see Section 2.2).

Gamma et al. say that the choice of programming language influences one's point of view on design patterns and that some patterns are supported directly by some programming languages. Thus, design patterns don't have to be programming language independent. Others argue that design patterns should be integral parts of programming languages [17, 9]. Agerbo and Cornils analysed the GoF design patterns to determine which patterns could be expressed directly, using features of a sufficiently powerful programming language [8, 9].

Patterns address common problems and provide accepted, good solutions. Programming languages have to be able to solve these problems, either by supporting solutions directly or by enabling programmers to implement the solution as a pattern.

We implemented the design patterns in GO to determine possible solutions to the problems the patterns address. Our use of patterns should give programmers insight into how GO-specific language features are used in everyday programming, and how gaps, compared with other languages, can be bridged. We think that using design patterns is a viable way to evaluate a programming language. By using design patterns to evaluate a programming language we give advanced programmers an insight into language specific features, and enable novices to learn a new programming language with familiar concepts. The implementations in different programming languages could serve as the basis for further comparison.

3.2 Embedding or Composition

In this section we highlight patterns where both embedding and composition are viable alternative design decisions. Furthermore we point out patterns that should (or should not) be implemented with embedding.

3.2.1 Observer

In the Observer pattern a one-to-many relationship between interesting and interested objects is defined. When an event occurs the interested objects receive a message about the event.

Observers register with subjects. Subjects inform their observers about changes. Observers get the changes from the subject that notified them. All subjects have operations in common to attach, detach and notify their observes. (See Appendix A.3.7)

In the *Design Patterns*' description of the Observer pattern, subject is an abstract class. GO does not have abstract classes. The definition of an interface and embedding can be used, even though it is not as convenient.

A major advantage of embedding becomes apparent in the implementation of the Observer pattern. Every type can be made observable, only a type implementing the subject's interface needs to be embedded. In Java, this is not possible. Java supports single class inheritance, which limits the ability to inherit the observable functionality. In GO however, multiple types can be embedded. Types that are doing their responsibility can easily be made observable. Embedding the observable functionality in subjects works only as long as the subjects expect the same type of observer (e.g. a mouse only accepting mouse observers). The embedding type can still override the embedded type's members to provide different behaviour.

Even though the embedding makes it easy to implement the Observer pattern, as far as we can see, the pattern is not as widely used in GO as it is in Java libraries (Listeners in Swing for example).

3.2.2 Adapter

The Adapter pattern enables programmers to translate the interface of a type into a compatible interface. An adapter allows types to work together that normally could not because of incompatible interfaces.

The solution is to provide a type (the adapter) that translates calls to its interface (the target) into calls to the original interface (the adaptee). (See Appendix A.2.1)

The adapter implements the target's interface. The adapting methods delegate calls of the target to the adaptee. Adapters can be implemented as described in *Design Patterns*: the adapter maintains a reference to the adaptee and delegates calls to that reference. GO provides an extended from of composition: embedding. The adapter could embed the adaptee. The adapter still has to implement the target's interface by delegating calls of the target to the adaptee.

Having the choice between embedding and composition is similar to class-based languages where an adapter can use inheritance or composition. Composition should be used when the public interface of the adaptee should remain hidden. Embedding exhibits the adaptee's entire public interface but reduces bookkeeping.

3.2.3 Proxy

The Proxy pattern allows programmers to represent an object that is complex or time consuming to create with a simpler one.

The proxy, or surrogate, instantiates the real subject the first time the client makes a request of the proxy. The proxy remembers the identity of the real subject, and forwards the instigating request to this real subject. Then all subsequent requests are simply forwarded directly to the encapsulated real subject. Proxy and real subject both implement the interface subject, so that clients can treat proxy and real subject interchangeably. (See Appendix A.2.7)

Proxy can be implemented in GO with object composition as described in *Design Patterns*. In GO we also have the option to use embedding. Instead of maintaining a reference to a subject, the proxy could *embed* a real subject. The proxy would gain all of the subjects functionality and could override the necessary methods. The advantage of embedding is that message sends are forwarded automatically, no bookkeeping is required. The disadvantage of embedding a real subject is, that the proxy type cannot be used dynamically for different real subjects. Each real subject type would need their own proxy type.

3.2.4 Decorator

The Decorator pattern lets programmers augment an object dynamically with additional behaviour.

A decorator wraps the component, the object to be decorated, and adds additional functionality. The decorator's interface conforms to the components's interface. The decorator will forward requests to the decorated component. Components can be decorated with multiple decorators. It is transparent to clients working with components if the component is decorated or not. (See Appendix A.2.4)

The Decorator pattern in GO needs to be implemented with composition. Decorators should not embed the type to be decorated. The decorator maintains a reference to the object that is to be decorated. If the decorator embedded a component, every time a new component type were added a new decorator type would be needed too. This would cause an "explosion" of types, which the decorator pattern is trying to avoid. In a hypothetical solution the decorator had to embed the component's common interface, but in GO only types can be embedded in structs, not interfaces.

3.3 Abstract Classes

We describe in this section how the lack of abstract classes affects the implementation of certain design patterns.

Composite The Composite pattern lets programmers treat a group of objects in the same way as a single instance of an object and creates tree structures of objects. The key is to define an interface for all components (primitives and their containers) to be able to treat them uniformly. (See Appendix A.2.3)

Bridge The Bridge pattern allows programmers to avoid permanent binding between an abstraction and an implementation to allow independent variation of each. (See Appendix A.2.2)

Design Patterns describes both pattern, Composite and Bridge, with abstract classes: *Composite* in the Composite pattern and *Abstraction* in the Bridge pattern.

GO has no abstract classes. To achieve the advantages of abstract classes (define interface and provide common functionality) two GO concepts have to be combined. An interface for the interface definition, and a separate type encapsulating common functionality. Types wishing to inherit the composite functionality have to embed this default type.

Like in Java, interfaces can be extended in GO too. This is done by one interface embedding another interface. Interfaces can embed multiple interfaces. In the Composite pattern the composite interface embeds the component interface ensuring that all composites are components too.

3.4 First Class Functions

In this section we discuss patterns that take advantage of GO's support for first class functions.

Strategy The Strategy pattern decouples an algorithm from its host by encapsulating the algorithm into its own type.

A strategy provides an interface for supported algorithms which concrete strategies implement. A context maintains a reference to a strategy. The context's behaviour depends on the dynamic type of the strategy and can be changed dynamically. (See Appendix A.3.9)

State The State pattern lets you change an object's behaviour by switching dynamically to a set of different operations.

The object whose behaviour is to be changed dynamically is the context and maintains a state object. Asking the context to execute its functionality will execute the context's current state's behaviour. The behaviour depends on the dynamic type of the state object. (See Appendix A.3.8)

Command The Command patterns allows programmers to represent and encapsulate all the information needed to call a method at a later time, and to decouple the sender and the receiver of a method call. The Command pattern turns requests (method calls) into command objects. An invoker maintains a reference to a command object. Clients interact with the invoker. The invoker's behaviour depends on the dynamic type of the command. (See Appendix A.3.2)

The Strategy, State or Command pattern can be implemented as suggested in *Design Patterns*: the algorithm for each strategy, state, command is its own type encapsulating the specific behaviour. The states' and strategies' context and the commands' invoker maintain references to the respective

objects. The context's/invoker's behaviour depends on the dynamic type of the objects. In GO however, we don't need to define a separate type for each algorithm. GO has first class functions. Strategies, states or commands can be encapsulated in functions and the contexts/invokers store references to functions, instead of objects.

Defining a type and a method for each algorithm is elaborate compared to defining a function. Strategies, states or commands could be Singletons or Flyweights. Encapsulating algorithms as functions removes the necessity for Singleton or Flyweight. However, if the algorithms need to store state, they are better implemented with types. Functions could store state in package-local static variables, but this would break encapsulation, since other functions in the package have access to those variables.

Using objects to encapsulate the algorithms, the context's/invoker's behaviour depends on the dynamic type of the objects. Figuring out which method is called at a certain point can be a daunting task. Using functions makes the code easier to follow.

There are advantages of having a type with methods over having functions only. Objects can store state. Functions in GO can store state only in static variables defined outside the function's scope. Variables defined inside the function's scope cannot be static; they are initialized every time the function is called. The problem with static variables is that they can be altered externally too. The function cannot rely on the variable's state. Having a type allows programmers to use helper methods and embedding for reuse. This possibility is not given to functions. Subtypes could override the helper functions or override the algorithm and reuse the helper functionality.

Encapsulation could be achieved by organizing each function in a separate package, but this comes at a cost: each package has to be in a separate folder, additionally the packages need to be imported before the functions can be used. This might be more work than implementing separate types.

3.5 Client-Specified Self

Embedding does not support the usual object-oriented behaviour of dynamic dispatch on `self`. In particular, GO will dispatch from outer objects to inner objects (up from subclasses to superclasses) but not in the other direction, from inner objects to outer objects (down from superclasses to subclasses). Downwards dispatch is often useful in general, and particularly so in the Template Method and Factory Method patterns. Downwards dispatch can be emulated by passing the outermost "self" object as an extra parameter to all methods that need it — implementing the client-specified self pattern (see Section 2.2.2). To use dynamic dispatch, the method's receiver must also still be supplied.

3.5.1 Template Method

The Template Method pattern enables the programmer to define a stable outline of an algorithm and letting subclasses implement certain steps of the algorithm without changing the algorithm's structure.

The usual participants of Template Method are an abstract class declaring a final method (the algorithm). The steps of the algorithm can either be concrete methods providing default behaviour or abstract methods, forcing subclasses to override them. GO however, does not have abstract classes nor abstract methods.

Consider the template method `PlayGame`:

```
func (this *BasicGame) PlayGame(game Game,
    players int) {
  game.SetPlayers(players)
  game.InitGame()
  for !game.EndOfGame() {
    game.DoTurn()
  }
  game.PrintWinner()
}
```

`PlayGame` is defined for type `BasicGame`. Steps like `SetPlayers` are to

be implemented by subclasses of `BasicGame` (e.g `Chess`).

The major difference compared with standard implementations is that we must pass an object to the template method twice: first as the receiver `this *BasicGame` and then as the first argument `game Game`. This is the Client-Specified Self pattern (Section 2.2.2). `BasicGame` will be embedded into concrete games (like `Chess`). Inside `PlayGame`, calls made to `this` will call methods on `BasicGame`; GO does not dispatch back to the embedding object. If we wish the embedding object's methods to be called, then we must pass in the embedding object as an extra parameter. This extra parameter must be passed in by the client of `PlayGame`, and must be the same object as the first receiver. Every additional parameter places a further burden on maintainers of the code. Furthermore the parameter has the potentially for confusion: `chess.PlayGame(new(Monopoly), 2))`. Here we pass an object of type `Monopoly` to the `PlayGame` method of the object chess. This call will not play Chess, but Monopoly.

The combination of `BasicGame` and the interface `Game` is effectively making `EndOfGame` an abstract method. `Game` requires this method, but `BasicGame` doesn't provide it. Embedding `BasicGame` in a type and not implementing `EndOfGame` will result in a compile time error when it is used instead of a `Game` object.

The benefits of associating the algorithm with a type allows for providing default implementations, using of GO's compiler to check for unimplemented methods. The disadvantage is that sub-types like `Chess` could override the template method, contradicting the idea of having the template method fixed and subtypes implementing only the steps. In Java, the `PlayGame` method would be declared `final` so that the structure of the game cannot be changed. This is not possible in GO because there is no way to stop a method being overridden.

In class-based languages, the `BasicGame` class would usually be declared abstract, because it does not make sense to instantiate `BasicGame`. GO, however, has no equivalent of abstract classes. To prevent `BasicGame`

objects being used, we do not implement all methods in the `Game` interface. This means that `BasicGame` objects cannot be used as the client-specified self parameter to `PlayGame` (because it does not implement `Game`).

3.6 Information Hiding

In this section we look at patterns that take particular advantage of GO's approach to information hiding.

3.6.1 Singleton

The Singleton pattern ensures that only a single instance of a type exists in a program, and provides a global access point to that object. (See Appendix A.1.5)

The Singleton pattern is usually implemented by hiding the singleton's constructor (using `private` scope in Java). In GO, constructors cannot be made private, but we have two ways to limit object instantiation by using GO's package access mechanisms. The first option is to declare the singleton type's scope package private and to provide a public function returning the only instance. Clients outside the package can't create instances of the un-exported type directly, only through the public function. The Singleton pattern is only effective for clients outside the package a singleton is declared. Within the singleton's package multiple instances can be created, since GO does not have a private scope; in particular, this might be done inadvertently by embedding the private type.

The other option is to use a package as the singleton. Packages cannot be instantiated and package initialization is only done once. The singleton package maintains its state in package private static variables and provides public functions to access the private state. The first approach requires a separate type for representing the singleton and a function controlling instantiation; the approach using the package as the singleton is less com-

plex, because instantiation does not need to be controlled and no separate type is necessary. Both approaches need to define public accessors for the singleton's state.

3.6.2 Façade

The Façade pattern defines a high-level abstraction for subsystems, and provides a simplified interface to a larger body of code to make subsystems easier to use.

Clients can be shielded from a system's low-level details by providing a higher-level interface. To hide the implementation of subsystems, a *Façade* is defined to supply a unified interface. Most Clients interact with the façade without having to know about its internals. The Façade pattern still allows access to lower-level functionality for the few clients that need it. (See Appendix A.2.5)

We have two options to implement the Façade pattern in GO. The first option uses composition and follows closely the solution given in *Design Patterns*: A façade type is declared maintaining references to objects of the subsystems that are to be hidden. The additional option that GO offers is to use a package to represent a façade. The subsystems are maintained in package static variables. Access to the subsystems is given through public functions. Using a package does not require creating an façade object. The package and its subsystems are initialized in the `init` function, which is automatically called on import. The initialization could be done in a separate method (`init` would call that method), thus allowing clients to re-initialize the package.

The Façade pattern should not be implemented with embedding: composition should be used instead, because the purpose is to hide low-level functionality and to provide a simple interface. A façade that embeds subcomponents publishes the entire public interface of the subcomponents, counteracting the pattern's intent.

3.6.3 Flyweight

The Flyweight pattern enables the programmer to minimize memory use by sharing as much data as possible between similar objects; it is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory.

The solution is to share objects which have the same intrinsic state and provide them with extrinsic state when they are used. This will reduce the number of objects greatly. A small number of objects stores a minimum amount of data, hence Flyweight. (See Appendix A.2.5)

The implementation of the Flyweight pattern as described in *Design Patterns* requires a factory to control creation of flyweight objects. In contrast to the Singleton pattern, the flyweight type can be public. Flyweights that need to be shared are created by using the flyweight factory. Flyweights can be used unshared by instantiating them directly.

The additional flyweight factory type is overhead. A public function could do the factory's job instead. The existing flyweights would be kept in a package local static variable. Only the factory function grants package external clients access to flyweights. Singleton's limitation of only being effective for package external clients is present in Flyweight as well. If the flyweight objects have to be shared, the flyweights' type has to be package private to avoid uncontrolled instantiation. The hiding of the type is only effective for package external clients, since flyweights can be instantiated uncontrollably within the flyweight's package.

3.6.4 Memento

The Memento pattern lets the programmer bring an object back to a previous state by externalizing the object's internal state.

The object whose state is to be restored is called the originator. The originator instantiates a memento, an object that saves the originator's current state. Clients asks the originator for mementos to capture the

originator's state. Client then use the mementos to restore the originator. (See Appendix A.3.6)

Mementos have two interfaces: a narrow interface for clients and a wide one for originators. In C++ originator is a friend of memento gaining access to the wide interface. The methods in the narrow interface are declared public. In GO we can use visibility scopes to shield the memento from clients and keep it accessible for originators. We define the wide interface with public methods and the narrow one with package local methods. Memento and originator should reside in same package and clients should be external, to protect mementos from being accessed.

Chapter 4

Case Study: The GoHotDraw Framework

In this chapter we present GoHotDraw: a GO port of JHotDraw, a framework for building graphical drawing editor applications. We used JHotDraw as a baseline for comparison to evaluate how GO specifics influence the design and implementation of GoHotDraw.

JHotDraw was developed by Erich Gamma for teaching purposes. It is a mature Java framework that is publicly available [38] and still maintained by an open source community. JHotDraw itself is based on a long history of drawing editor application frameworks, most notably HotDraw[48]. HotDraw was originally developed by Ward Cunningham and Kent Beck in the programming language Smalltalk [18]. HotDraw is a pattern-dense framework, making use of the majority of the design patterns proposed in [39] and other design patterns. JHotDraw is also influenced by ET++, a C++ application framework, which was also developed by Erich Gamma [94]. JHotDraw and its predecessors were developed and designed by expert programmers. JHotDraw is still maintained by its community. As a consequence JHotDraw is a very mature framework, and a good way to evaluate a programming language.

We highlight and discuss design similarities and differences to JHot-

Draw and GoHotDraw with regards to design patterns, and we point out differences in the overall design and implementation details we came across.

This chapter is organized as follows: in Section 4.1 we present our methodology; in Section 4.2 we give a brief overview of GoHotDraw's design; in Section 4.3 we discuss differences between GoHotDraw and JHotDraw with regards to design and implementation.

4.1 Methodology

Porting an existing application into GO has the advantage of having a basis for comparison. The HotDraw frameworks were chosen because of their extensive use of design patterns and framework's maturity. The decision to port JHotDraw instead of HotDraw is based on the author's preference for Java over Smalltalk.

We started by studying relevant literature describing HotDraw's [14, 18, 28, 36, 49] and JHotDraw's [25, 51, 81] design to gain an understanding of how the patterns fit into the framework and how the pattern relate to each other. Furthermore we investigated the source code of JHotDraw [38] to find the subset of functionality necessary to cover the main patterns and to implement basic functionality (drawing, selecting, moving and resizing of figures).

JHotDraw is quite large. As a guide JHotDraw v5.3 is about 15,000 lines of code and v7.3 about 70,000 lines, including several complete applications [38]. We concentrated on the essence of the framework's functionality (Drawings contains Views, Views contain Figures, Editors enable Tools to manipulate Drawings). This functionality subset suffices to cover all the key patterns in JHotDraw's design.

We started at the core of the framework: the Figure interface and implemented CompositeFigure and RectangleFigure. Next came the Drawing and View interfaces with a concrete implementation each to allow graphical

output, followed by the Editor. The Tool hierarchy was developed last. We wanted to gain a better understanding of the framework and of GO before attempting to implement the resize functionality. Creation and dragging of figures is trivial compared to resizing. Resizing involves the actual change of the underlying model to reflect the size changes as well as displaying handles, the rectangular boxes that appear when a figure is selected.

The subset of functionality that we implemented amounts to around 2,000 lines. We incorporated nearly all patterns embodied in JHotDraw. We did not implement the connection of figures, which uses the Strategy pattern. We implemented painters, however, with the Strategy pattern. The omitted pattern operates quite isolated from other patterns. After implementing the basic functionality we estimated that the effort necessary to implement the connection functionality would be too high for the remaining time. From studying the patterns individually (see Chapter 3) we knew that a GO implementations of this pattern is possible and feasible.

4.2 The Design of GoHotDraw

In this section we describe the GoHotDraw framework. Like its Smalltalk and Java counterparts, GoHotDraw is designed to create drawing editor applications. These applications can be created by combining already existing features, as the framework can be easily extended. New components, like tools or figures, need only to conform to the core interfaces and will fit in seamlessly.

The GoHotDraw framework is designed according to the Model-View-Controller (MVC) pattern. The MVC pattern is an architectural pattern. The intent of the pattern is to separate the application into three distinct units: the domain knowledge, the presentation, and the actions based on user input. In the following we explain the three parts (Model, View and Controller); describe the main types and interfaces; and highlight design patterns used in the design of GoHotDraw.

Figure 4.1 depicts the main interfaces and types of the GoHotDraw framework.

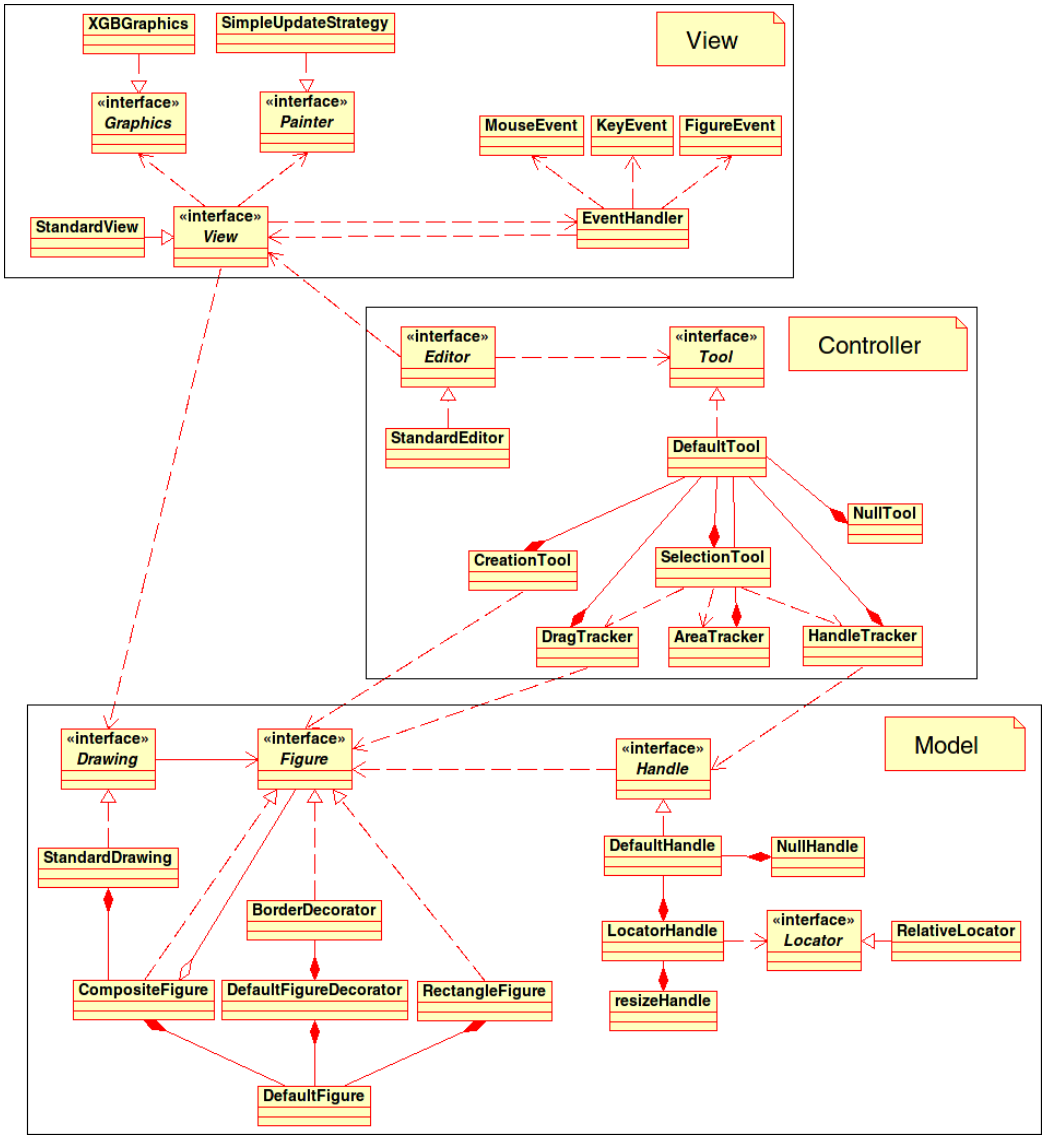


Figure 4.1: Type diagram of GoHotDraw

4.2.1 Model

"A Model is an active representation of an abstraction in the form of data in a computing system" [80]. A model is a single object or a structure of objects and "represents knowledge" [79]. The model is the domain-specific representation of the data upon which the application operates.

The Model of GoHotDraw represents the data users operate on. GoHotDraw's model comprises figures and their handles. Figures are elements users can draw and manipulate. Users manipulate figures by dragging handles (the little rectangles on the perimeter of the top left figure in Figure 4.2 are handles).

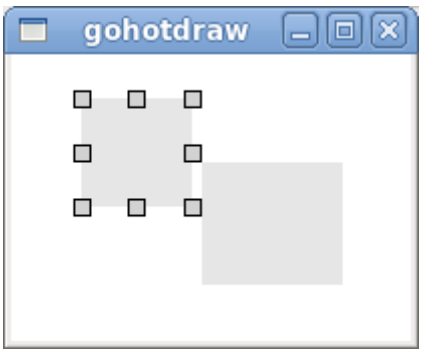


Figure 4.2: Handles on a figure

The diagram in Figure 4.3 shows the types that form the Model of the GoHotDraw framework. In this section we will explain the elements of the Model in detail, how they relate to each other and we will highlight how the elements relate to design patterns.

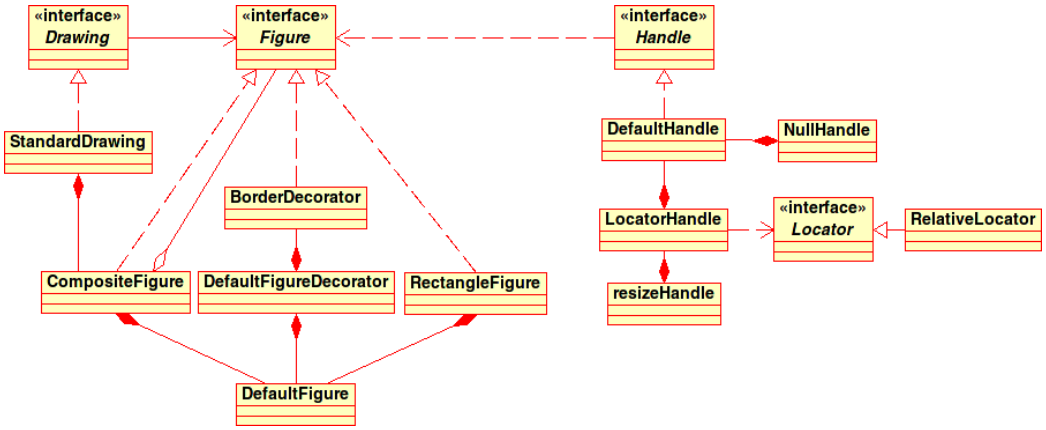


Figure 4.3: Model of GoHotDraw

Figure

Figure is a central abstraction in the GoHotDraw framework. Figure is the common interface other GoHotDraw components work with. Figures represent graphical figures that users can arrange to drawings. Figures can draw themselves and have handles for their manipulation.

DefaultFigure implements functionality common to all figures, like handling of event listeners. Event listening is part of the Observer pattern, that is used to update the View. Concrete implementations of Figure embed DefaultFigure. All types embedding DefaultFigure can be observed. That way it is easy to extend the selection of available figure types without having to change the rest of the framework. Even though Figure has more than 20 methods, the implementation of RectangleFigure consists of less than 60 lines of code.

The Figure type-hierarchy uses the Template Method pattern frequently. Template methods are defined in the DefaultFigure type. The concrete implementations of Figure embed DefaultFigure and implement the hook methods.

Composite Figure and Drawing

CompositeFigure is a figure that is composed of several figures, which in turn can be composite figures. With CompositeFigure, a composition of figures can be treated like a single figure. CompositeFigure embeds DefaultFigure and maintains a collection of figure objects.

A Drawing is a container for figures. Drawings are displayed by views. StandardDrawing implements the Drawing interface and the Figure interface by virtue of embedding CompositeFigure.

The Composite pattern is used to enable us to treat simple figures (like rectangles) and complex figures (like drawings) uniformly. The Figure interface plays the role of *Component*, CompositeFigure and StandardDrawing are the *Composites*, and RectangleFigure is a *Leaf*. The relationship between these types is depicted on the left hand side of Figure 4.3.

Border Decorator

In GoHotDraw figures are drawn without borders. To draw borders around figures the figure objects get decorated with BorderDecorators. We implemented a DefaultFigureDecorator, which implements the interface Figure. DefaultFigureDecorator embeds DefaultFigure and maintains a reference to the figure object that is to be decorated. Method calls are forwarded to the figure object.

The listing below shows the definition of DefaultFigureDecorator and one of its methods forwarding calls to the decorated figure. DefaultFigureDecorator acts as the base type for other decorators.

```
type DefaultFigureDecorator struct {
  *DefaultFigure
  figure Figure //the decorated figure
}

func (this *DefaultFigureDecorator) GetDisplayBox() *Rectangle {
  return this.figure.GetDisplayBox()
}
```

BorderDecorator embeds DefaultFigureDecorator, gaining its functionality. BorderDecorator overrides the Draw method, which draws the figure and then a border on top:

```
func (this *BorderDecorator) Draw(g Graphics) {
  this.DefaultFigureDecorator.Draw(g)
  g.SetFGColor(Black)
  g.DrawBorderFromRect(this.GetDisplayBox())
}
```

Clone has to be overridden to return a new instance of BorderDecorator, instead of an instance of the decorated figure.

```
func (this *BorderDecorator) Clone() Figure {
  return NewBorderDecorator(this.figure)
}
```

The actual figure object is oblivious that it is decorated with a border and other parts of the framework don't have to distinguish between bordered or un-bordered Figure objects, since DefaultFigureDecorator implements the Figure interface.

Handle types

A Handle (displayed by a little rectangle, see Figure 4.2) is used to change a figure by direct manipulation. A figure has eight handles (one on each corner and one on each side). A handle knows its owning figure and provides its location to track changes.

Handles use Locators to find the right position on a figure to display the handles. Locators encapsulate a Strategy to locate a handle on a figure. The handle is the Context for Locator objects.

For the creation of Handles, the Factory Method pattern is used. Concrete implementations of Figure create handle objects on demand by implementing `GetHandles`. In the listing below `GetHandle` creates a set of handles for the current figure object. The actual factory method is the function `AddAllHandles`. `AddAllHandles` first adds four handles (one for each corner) to `handles` and then adds four handles for each side. The

parameter `figure` gets passed to the handle creation, so that each handle knows its figure. Note that `handles` is an output parameter.

```
func (this *RectangleFigure) GetHandles() *Set {
  handles := NewSet()
  AddAllHandles(this, handles)
  return handles
}

func AddAllHandles(figure Figure, handles *Set) {
  AddCornerHandles(f, handles)
  handles.Push(newSouthHandle(f))
  handles.Push(newEastHandle(f))
  handles.Push(newNorthHandle(f))
  handles.Push(newWestHandle(f))
}
```

These are simplifications of the Factory Method pattern. The listed examples create objects and return them. Fully fledged factory methods return different products depending on the dynamic type of their receiver.

4.2.2 View

"A view is a (visual) representation of its model" [79] and is "...capable of showing one or more [...] representations of the Model" [80]. The view renders the model into a form suitable for interaction.

The View part of GoHotDraw is concerned with displaying drawings. The View does not manipulate the Model directly. User input is forwarded to the Controller. The diagram below show the type that are part of GoHotDraw's View.

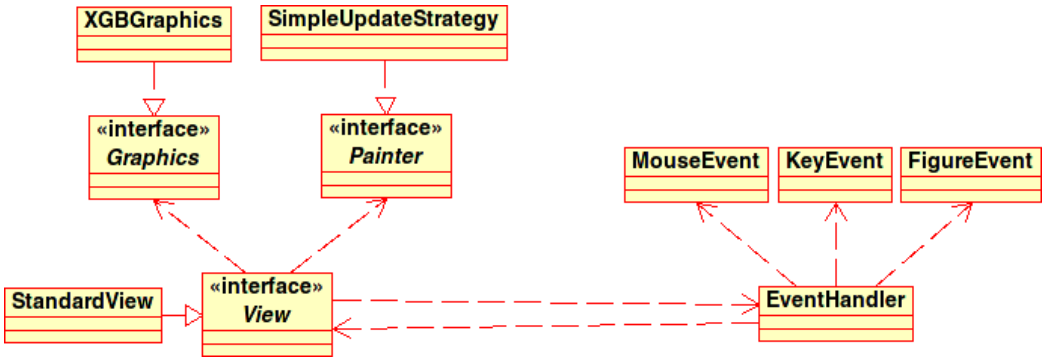


Figure 4.4: View types of GoHotDraw

View Interface

View is another important interface in GoHotDraw. A view renders a drawing and listens to its changes. It receives user input from the user interface (Graphics) and forwards the mouse or key events to an editor. Views manage figure selections, including drawing of handles.

Drawing changes are propagated to views with the Observer pattern. Drawings send FigureEvents to their listeners (the EventHandlers of views) and the views redraw themselves. The EventHandler type acts as listener for views. The EventHandler receives events (figure added, removed, changed) from the drawing and informs the view to update itself. Controllers changing figures, trigger FigureEvent, which are send to the figures listeners.

Graphics

Graphics is an abstraction of the user interface. The displaying and handing of user input could be done with a variety of GUI-libraries. We used the XGB library to communicate with the X Window Server for user input and graphical output.

The X Window System, or X11, is a software system and network protocol that provides a graphical user interface (GUI) for any computer that implements the X protocol. XBG provides low level functionality for communication with an X server. XGB is a GO port of the XCB library, written in C. XGB is used to display the user interface and to capture user input.

We use the Adapter pattern to adapt the third-party graphic library XGB, to GoHotDraw's Graphics interface. We implement a XGBGraphics to conform to the Graphics interface, thus enabling us to use the XGB library with GoHotDraw.

Painter

A painter encapsulates an algorithm to render drawings in views. In GoHotDraw, different update strategies can be defined and views select the appropriate ones. We implemented a SimpleUpdateStrategy, which redraws the entire drawing, but more sophisticated strategies are possible, like accumulating damage and only redrawing the parts affected. View is the Context, Painter the Strategy interface, and SimpleUpdateStrategy a Concrete Strategy.

4.2.3 Controller

"A controller is the link between a user and the system" [79]. The controller receives input and initiates a response by making calls on model objects. A controller accepts input from the user and instructs the model and viewport to perform actions based on that input.

The Controller part of GoHotDraw is concerned with figure creation and manipulation. Clients don't interact with the Model (figures in a drawing) directly. The View forwards user input to the Controller which manipulates the Model accordingly. The diagram below shows the types that are part of GoHotDraw's Controller.

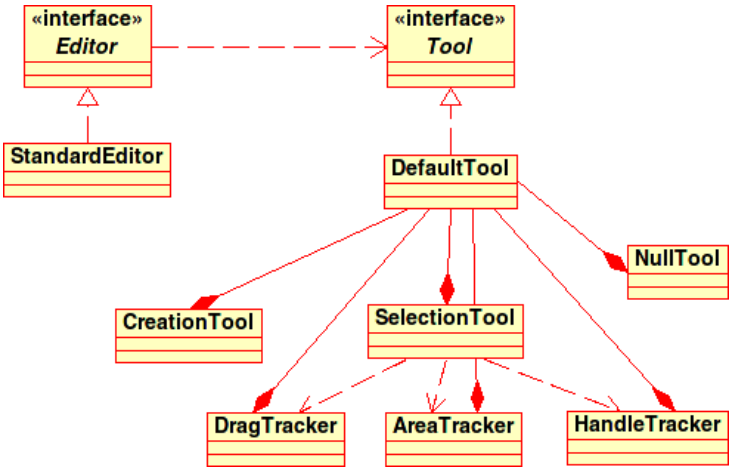


Figure 4.5: Controller types of GoHotDraw

Editor

An editor coordinates and decouples views and tools. Editors maintain the available and selected tools and delegate requests from the view to the currently selected tool.

The editor keeps references to its view and its current tool. Views and tools don't communicate directly with each other, but through the editor. This is an example of the Mediator pattern. Editors are mediators, views and tools are colleagues.

The editor's behaviour is dependent on its current tool. Clients define which tool is the active tool. Depending on the dynamic type of the tool object, the editor behaves differently. This is an example of the State pattern with the editor being the context and the tools being the states.

Tool

Tool is another major interface in GoHotDraw. Tools define a mode of a drawing view (create new figures; select, move, resize figures).

In GoHotDraw we use Null Object in the tools type hierarchy. NullTools do nothing. Editors can be initialized with NullTools, so that should clients

forget to set a specific tool, the editor will not cause a nil-pointer error, it will just do nothing.

Creation Tool

A creation tool is a tool that is used to create new figures. We use the Prototype pattern in creation tool. The figure to be created is specified by a prototype. A creation tool creates new figures by cloning the prototype.

The CreationTool maintains a reference to a prototypical instance of Figure:

```
type CreationTool struct {
  ...
  prototype     Figure
}
```

On request, the prototype figure is cloned and returned:

```
func (this *CreationTool) createFigure() Figure {
  if this.prototype == nil {
    panic("No prototype defined")
  }
  return this.prototype.Clone()
}
```

All concrete figures have to implement the `Clone` method. The GoHotDraw specific participants of the pattern are Figure as the Prototype and creation tool as the client. The next listing shows the `Clone` methods of RectangleFigure. RectangleFigure creates a new instance and copies the prototype's display box.

```
func (this *RectangleFigure) Clone() Figure {
  figure := NewRectangleFigure()
  figure.displayBox = this.displayBox
  return figure
}
```

Selection Tool and Trackers

A selection tool is a tool that is used to select and manipulate figures. The behaviour of SelectionTool is implemented with the State pattern. Depending

on external conditions, the SelectionTool's current tool changes. A selection tool can be in one of three states: figure selection with AreaTrackers, figure movement with DragTrackers, and handle manipulation with HandleTrackers. An AreaTracker is a selection tool for background selection (selecting one or more figures). On mouse drag events the area tracker informs the view about figures contained in the area covered, so that the found figure's handles can be drawn. A DragTracker is a selection tool used to select and move figures under the mouse pointer. On mouse down the drag tracker either informs the view to select the figure under the pointer or deselects all figures. On mouse drag the drag tracker moves the selected figures. The moved figures inform the view about being moved. A HandleTracker is a selection tool to manipulate handles. The handles do the actual figure manipulation.

The following listing shows the method `MouseDown`. `MouseDown` alters the state (`currentTool`) of `SelectionTool`. If the `MouseEvent` happened on a handle, `currentTool` becomes a `HandleTracker`; if the `MouseEvent` happened on a figure, the `currentTool` becomes a `DragTracker`; otherwise it becomes an `AreaTracker`. Depending on the dynamic type of `currentTool`, the call of `MouseDown` will behave differently. The HandleTrackers, DragTrackers and AreaTrackers are created on demand by factory methods.

```
func (this *SelectionTool) MouseDown(e *MouseEvent) {
  selectedHandle := this.editor.GetView().FindHandle(e.GetPoint())
  if selectedHandle != nil {
    this.currentTool = NewHandleTracker(this.editor, selectedHandle)
  } else {
    selectedFigure := this.editor.GetView().GetDrawing().FindFigure(e.GetPoint())
    if selectedFigure != nil {
      this.currentTool = NewDragTracker(this.editor, selectedFigure)
    } else {
      if !e.IsShiftDown() {
        this.editor.GetView().ClearSelection()
      }
      this.currentTool = NewAreaTracker(this.editor)
    }
  }
```

```
  this.currentTool.MouseDown(e)
}
```

4.3 Comparison of GoHotDraw and JHotDraw

GoHotDraw's design is very similar to the Smalltalk HotDraw and the Java JHotDraw. GoHotDraw applies the same patterns for similar functionality and they work as well in GO as in other languages.

As a result, GoHotDraw's structs and embedding generally parallel JHotDraw's classes and inheritance, and GoHotDraw's interfaces are generally similar to those in the Java versions.

JHotDraw is meant to be used to create drawing editor applications. Its architecture allows for easy extension of the existing functionality and for a convenient combination of existing components. We tried to achieve the same with GoHotDraw. We used the same core abstractions (Figure, View, Drawing, Editor, Tool) and provided common functionality. New functionality that conform to those abstractions will integrate well and allow for an convenient extension of the framework. We implemented a drawing application, allowing mouse controlled figure manipulation and keyboard controlled tool selection, by using the functionality provided by GoHotDraw, with only 90 lines of code.

4.3.1 Client-Specified Self in the Figure Interface

The key difference between the GO and the Java design comes from the difference between GO's embedding and Java's inheritance (see Section 2.1.2): in GO methods on "inner" embedded types cannot call back to methods on "outer" embedding types. In contrast, Java super class methods most certainly can call "down" to methods defined in their subclasses, and this is the key to the Template Method pattern. As a result, GoHotDraw's Figure interface is significantly different to the Java version. While both interfaces

provide the same methods, many (if not most) of those methods have to have an additional Figure parameter. Those methods are template methods (see Section 3.5.1) and need to use the Client-Specified Self pattern (see Section 2.2.2).

A simple example: A Figure is empty if its size is smaller than 3-by-3 pixels. The method `IsEmpty` in the following listing is defined for `DefaultFigure` (to be embedded, contains common functionality of all Figure sub-types), and call the `GetSize` method defined in the Figure interface.

```
func (this *DefaultFigure) IsEmpty(figure Figure) bool {
  dimension := figure.GetSize(figure)
  return dimension.Width < 3 || dimension.Height < 3
}
```

The problem is that this needs to call the `GetSize` method of the correct sub-type: here RectangleFigure or CompositeFigure. Both types embed DefaultFigure and provide a suitable definition for `GetSize` (as well as other methods). In Java, DefaultFigure could simply call `this.GetSize()`, the call will be dynamically dispatched and will run the correct method. In GO, this call would try to invoke the (non-existent) `GetSize` method on `DefaultFigure`: a client-specified self is needed for dynamic dispatch.

This problem is exacerbated when a design needs multiple levels of embedding or inheritance. Following JHotDraw, GoHotDraw's DefaultFigure is embedded in CompositeFigure; CompositeFigure is embedded in Drawing; Drawing is then further embedded in StandardDrawing. These multiple embeddings mean many Figure methods require a client-specified self parameter to work correctly. Of 21 methods in that interface, six require the additional client-specified self argument, as the final version of the Figure interface illustrates:

```
type Figure interface {
  MoveBy(figure Figure, dx int, dy int)
  basicMoveBy(dx int, dy int)
  changed(figure Figure)
  GetDisplayBox() *Rectangle
  GetSize(figure Figure) *Dimension
  IsEmpty(figure Figure) bool
  Includes(figure Figure) bool
  Draw(g Graphics)
  GetHandles() *Set
  GetFigures() *Set
  SetDisplayBoxRect(figure Figure, rect *Rectangle)
  SetDisplayBox(figure Figure, topLeft, bottomRight *Point)
  setBasicDisplayBox(topLeft, bottomRight *Point)
  GetListeners() *Set
  AddFigureListener(l FigureListener)
  RemoveFigureListener(l FigureListener)
  Release()
  GetZValue() int
  SetZValue(zValue int)
  Clone() Figure
  Contains(point *Point) bool
}
```

The size of interfaces in GoHotDraw is bigger than the GO norm ("In Go, interfaces are usually small: one or two or even zero methods." [71]). Some of the interfaces in JHotDraw contain dozens of methods. We do not believe that the size of an interface is a sign of good program design. We could have implemented some of Figure's methods as functions, reducing the number of methods in its interface, but we doubt that this would make GoHotDraw easier to understand, maintain or extend.

4.3.2 GoHotDraw User Interface

The Java user interface and graphics framework Swing is tightly integrated into JHotDraw. GO is designed to be a systems language. The UI capabilities are rather limited and are still marked as experimental.

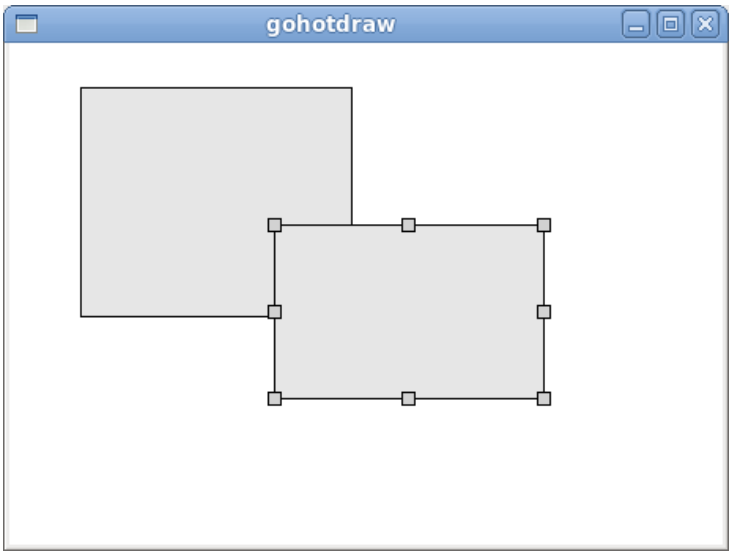


Figure 4.6: User interface GoHotDraw

Figure 4.6 shows a screenshot of a sample application created with GoHotDraw. Tools can be selected with function keys and the mouse is used for rectangle creation and manipulation.

Graphics library

GO is a systems systems language. The UI libraries are not very mature yet. We failed to get GO's standard graphics library "exp.draw.x11" (exp for experimental) to work on our machine. We tried the go-gtk framework [62]. We compiled the library and a simple application, but it crashed with a segment fault error. Eventually we settled for the "XGB framework Go-language Binding for the X Window System".

The X Window System, or X11, is a software system and network protocol that provides a graphical user interface (GUI) for any computer that implements the X protocol. XGB is a GO port of the XCB library, written in C. XBG provides low level functionality for communication with an X server. The XGB project's activity seized. Updates and changes of the GO

language were not incorporated for some months. We continued to use XGB with an older version of GO.

To limit the impact a change in the graphics library would have, we designed an additional adaptation layer. GoHotDraw uses the Graphics interface to show a window and render the shapes. XGBGraphics maintains a reference to an instance of `xgb.Conn` (a connection is a link between clients and the X-Server) as depicted in the diagram below:

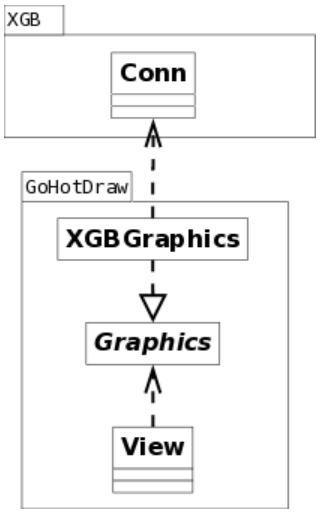


Figure 4.7: XGBGraphics adapting XGB library

The following listing shows one of the adapter methods. `DrawBorder` takes the location and dimension of a rectangle, creates a XGB-conform rectangle and sends the rectangle via the connection to the X-server to draw the border of a rectangle.

```
func (this *XGBGraphics) DrawBorder(x, y, width, height int) {
  rect := this.createRectangle(x, y, width, height)
  this.connection.PolyRectangle(this.pixmapId, this.contextId, rect)
}
```

In terms of the Adapter pattern, the interface Graphics is the Target, XGBGraphics the Adapter, xgb.Conn (the xgb connection to the X-Server) is the Adaptee, and GoHotDraw is the Client. XGBGraphics also shields

clients from having to fully understand the workings of the XGB library (connections, pixmaps, contexts...), making XGBGraphics a Façade for the XGB framework.

The current GUI library could be replaced by adapting the new library to the Graphics interface. This layer of abstraction is unique to GoHotDraw and is not present in JHotDraw. JHotDraw initially used the AWT framework and major have been necessary in to switching to Swing. Even though we have the adaptation layer in place we did not replace XGB with a different library.

Low-level Functionality

Apart from display purposes the JHotDraw uses the Swing framework for calculations regarding points, rectangles and dimensions. To be graphic library independent, we implemented our own Dimension, Point, and Rectangle types. The following listing shows the type Rectangle. A rectangle is represented by the x and y coordinates of their top left corner and its width and height. We declared the Rectangle's members public for convenience to not having to declare and use separate getters and setter.

The functionality for handling rectangles – grow, translate, union, contains – was mainly ported from Java's Swing framework. The porting was straight forward, but inconvenient due to the necessity of type casts for numbers. Our Rectangle type, as well as Swing's, work with integers. The position of a rectangle can be negative if its top left corner is outside the top or left of the canvas. We allow negative sizes to avoid type errors at runtime. Negative sizes are interpreted as zero. We used GO's math library, which works with floating point numbers. The type differences make type casts necessary. The method `ContainsRect` in the listing below checks if the passed in rectangle `rect` is fully contained in the receiver object `this`. The method uses the math library – part of the standard GO distribution. The `Fmin` and `Fmax` functions of the math package take float64 numbers as arguments and return the minimum or maximum as float64, therefore,

the result has to be cast back to int.

```
func (this *Rectangle) ContainsRect(rect *Rectangle) bool {
  return (
    rect.X >= this.X && rect.Y >= this.Y &&
    (rect.X+int(math.Fmax(0, float64(rect.Width)))) <=
      this.X+int(math.Fmax(0, float64(this.Width))) &&
    (rect.Y+int(math.Fmax(0, float64(rect.Height)))) <=
      this.Y+int(math.Fmax(0, float64(this.Height))))
}
```

GO's strict type system make type cast necessary. This is not only annoying to program, but makes it harder to read and maintain.

4.3.3 Event Handling

Transforming user input into events is a little bit awkward. The graphics library XGB can be polled for events. The events returned are of type xgb.Event, which is an interface. To extract what kind of event happened, a type switch is necessary.

Listing 4.1 shows an example for the `ButtonPressEvent` (aka MouseDown). The `StartListening` method repeatedly polls for event replies from the X-server. The returned event is of interface type `xgb.Event`. `xgb.Event` is an empty interface. A type switch determines the dynamic type of the `reply` object. If the reply is a `ButtonPressEvent`, a new `MouseEvent` gets instantiated; the event's X and Y coordinates, the pressed button, and the key modifier (pressed shift key for example) are extracted from `xgbEvent`. The created event is then forwarded to the `fireMouseDown` method. `fireMouseDown` calls `MouseDown` on all of its listeners and forwards the `MouseEvent` object. In the current implementation of GoHotDraw, the listeners are applications. Applications maintain a reference to the graphics object and to an editor. The editor in turn knows its tools and view.

The troublesome part about this kind of implementation is that with each new kind of input event, the switch-case has to be extended. This in not only tedious, but also hard to maintain, and bad object-oriented style.

```
func (this *XGBGraphics) StartListening() {
  for {
    reply := this.GetEventReply()
    switch xgbEvent := reply.(type) {
    case xgb.ButtonPressEvent:
      event := &MouseEvent{}
      event.X = int(xgbEvent.EventX)
      event.Y = int(xgbEvent.EventY)
      event.Button = int(xgbEvent.Detail)
      event.KeyModifier = int(xgbEvent.State)
      this.fireMouseDown(event)
    }
    //the other events (MouseUp, MouseDrag etc.)
  }
}

func (this *XGBGraphics) fireMouseDown(event *MouseEvent) {
  for i := 0; i < this.listeners.Len(); i++ {
    currentListener := this.listeners.At(i).(InputListener)
    currentListener.MouseDown(event)
  }
}
```

Listing 4.1: Transforming XGB events into GoHotDraw events

We simplified JHotDraw's event handling. The simplification was not due to GO specifics but a deliberate design decision, currently we support only one drawing and one view active at a time. We did not implement separate drawing events. The Drawing sends figure events to the view as well. An extension to allow multiple views of drawings might make a DrawingEvent type necessary.

4.3.4 Collections

JHotDraw maintains references to other objects in a variety of collections (ArrayList, Vector, Enumeration). GO's collection library is still very limited. We started out using Vector, but realized soon that a Set type was required. The implementation of Set was easy enough since most of Vector's functionality was reused.

Our Set embedded Vector and we overrode the Push method.

```
type Set struct {
  *vector.Vector
}

func (this *Set) Push(element interface{}) {
  this.Add(element)
}

func (this *Set) Add(element interface{}) {
  if !this.Contains(element) {
    this.Vector.Push(element)
  }
}
```

We also implemented other convenience methods, like Contains, Replace, and a Remove method based on object identity rather than indexes:

```
func (this *Set) Remove(element interface{}) {
  for i := 0; i < this.Vector.Len(); i++ {
    currentElement := this.Vector.At(i)
    if currentElement == element {
      this.Vector.Delete(i)
      return
    }
  }
}
```

Chapter 5

Evaluation

In this Chapter we reflect on what we have learned about GO from implementing design patterns and GoHotDraw. We discuss GO functionality, GO idioms we found, and tools provided with GO.

5.1 Client-Specified Self

A common design principle of frameworks is Inversion of Control [33]: frameworks provide interfaces that clients implement; the framework then calls the client's code, instead of the client calling the framework. This is also known as the Hollywood Principle: "don't call us, we call you" [12]. Inversion of Control is often achieved by applying the Template Method pattern: the framework defines the algorithm and provides hook methods that are to be implemented in client code. In GO, Template Method (and with that frameworks) will have to apply the Client-Specified Self pattern with all of its inconveniences. We are not aware of frameworks written in GO, but we expect to see Client-Specified Self being applied widely, causing suboptimal designs.

Using client-specified self is less satisfactory than inheritance: it requires collaboration of an object's client to work; the invariant that the self parameter is in fact the self is not enforced by the compiler, and

there is scope for error if an object other than the correct one is passed in (e.g. `chess.Play(monopoly)` see Appendix A.3.10). Furthermore, client-specified self is a cause for confusion on the implementer side as well: are hook methods to be called on the template method's receiver, or on the the passed in parameter? The extra parameter complicates code, making it harder to read and write, for no clear benefit over inheritance.

5.2 Polymorphic Type Hierarchies

In most object-oriented languages types can form polymorphic type hierarchies. A type can be extended by adding and overriding existing behaviour. The extended type is a subtype of the original, in that any value of the extended type can be regarded as a value of the original type by ignoring the additional fields.

In GO, types can only be used polymorphically if there is an interface defined that all subtypes implement. The supertype can either implement all or only some of the interface's methods.

It is considered good object-oriented design to define the supertype in a type hierarchy as an abstract class [19]. Abstract classes are types which are partially implemented and cannot be instantiated. Abstract classes are commonly used both in design patterns and object-oriented programming in general. Abstract classes define an interface and provide implementations of common functionality that subtypes can either use or override. Interfaces can be used to define methods without implementations, but these cannot easily be combined with partial implementations. GO does not have an equivalent language construct to abstract classes.

GO's implicit interface declarations provide a partial work around: provide an interface listing methods, and a type that provides default behaviour of all or a subset of the methods. In Java, classes implementing the interface would extend the abstract class and define the missing methods. In GO, types implementing an interface would embed the type

that is providing default implementations and implement the missing functionality. In our designs, we applied a naming convention for types providing default behaviour for an interface. We suffix the interface name with "default" (consider the interface `Foo`, the type to be embedded is then called `DefaultFoo`, and a concrete type embeds it: `type ConcreteFoo struct {DefaultFoo}`).

The methods the default type are not implementing are effectively abstract methods (C++'s pure virtual or Smalltalk's subclassResponsibility), since the compiler will raise an error if the embedding type does not provide the missing methods. We consider this less convenient than abstract classes, because it involves two language constructs: the interface and the type implementing default behaviour. Furthermore, the default type has to be embedded and potentially initialized (see Section 5.3).

Another property of abstract classes is that they cannot be instantiated. Most default types we implemented were meant to be embedded only and not to be instantiated directly. There is no feature in GO to allow embedding and prevent instantiation. Types can always be instantiated within packages. Defining a type's visibility to package private mitigates that problem, as only publicly defined functions of that package can return an instance of the private type. But there is a major catch here, making a type package private prevents clients from being able to embed the type. These are contradicting forces. Either instantiation can be controlled and the type can't be embedded, or embedding is possible but the type can be instantiated freely.

5.3 Embedding

GO favours composition over inheritance. The Gang-of-Four's experience was that designers overused inheritance in their principle "favor object composition over class inheritance" [39, p.20]. Since embedding is an automated form of composition, it is not obvious whether the same will

apply to embedding vis-à-vis composition. Embedding has many of the drawbacks of inheritance: it affects the public interface of objects, it is not fine-grained (i.e, no method-level control over embedding), methods of embedded objects cannot be hidden, and it is static. Embedding should be used with caution, since the entire public interface of the embedded type is published to clients of the embedding type (fields and methods). The alternative is to use composition with delegation. There is no middle ground in GO to selectively publish certain methods and fields.

5.3.1 Initialization

An embedded type is actually an object providing its functionality (methods and members). If any of the embedded objects is of pointer type the embedded object needs to be instantiated. Not initializing the embedded type can result in runtime errors. We see this as a disadvantage compared with class based inheritance. Having to initialize embedded types is unwieldy and easy to forget, because GO does not have constructors which could conveniently be used to initialize an object.

5.3.2 Multiple Embedding

We found multiple embedding helpful in implementing the Observer pattern. A public type had to be defined providing the observable behaviour (add, remove and notify observers). Every type could be made observable by embedding that type. The embedding type could still override the embedded type's methods and provide different behaviour if necessary.

As with multiple inheritance, multiple embedding introduces the problem of ambiguous members. We find GO solves this problem in a nice way: ambiguity is not an error, only calling an ambiguous member is a mistake and ambiguous calls are detected at compile time. The ambiguity can be resolved by fully qualifying the member in question.

5.4 Interfaces and Structural Subtyping

We think GO's approach to polymorphism, through interfaces and structural subtyping instead of class-based polymorphism, is a good one, because interface inference reduces syntactical syntactic overhead, and GO's concise and straightforward syntax for interfaces declaration encourages the use of interfaces. Using interfaces reduces dependencies, which is expressed in the Gang-of-Four's principle of "program to an interface, not an implementation" [39, p.18].

GO uses nominal typing and structural subtyping: types and interfaces have names, but whether a type implements an interface depends on the type's structure. Parameters in GO are often given empty interface type which indicate the kind of object expected by the method, rather than any expected functionality. This idiom is common in other languages, including Java: e.g., `Cloneable` in the standard library. In GO, however, all objects implicitly implement empty interfaces, so using an empty interface, even a named one, does not help the compiler check the programmer's intent. With named empty interfaces one gets the documentary value of the nominal typing, but loses the compiler checking of static typing.

An alternative solution is to use interfaces with a single, or very small number of, methods. This lessens the likelihood of objects implementing the interface by accident — but does not remove it. We found this idiom used in the standard library ("In GO, interfaces are usually small: one or two or even zero methods." [71]) and used it frequently in our own code. Unlike much GO code, however, the key interfaces in GoHotDraw have several tens of methods, closer to design practice in other object-oriented languages.

5.5 Object Creation

We found GO's multifarious object creation syntax — some with and some without the `new` keyword — hard to interpret. Having to remember when to use `new` and when to use `make` makes writing code more difficult than it needs to be. Furthermore the lack of explicit constructors and a consistent naming convention can require clients to refer to the documentation or to jump back and forth between definition and use in order to find the right function for object creation.

GO is a systems language and thus requires access to low-level functionality. The built-in function `new()` accepts a type as parameter, creates an instance of that type, and returns a pointer to the object. The function `new` can be overridden, giving GO programmers a powerful tool for defining custom object-instantiation functionality. The possibility of overriding applies to all built-in functions.

We think GO should have constructors. Many complex objects need to be initialized on instantiation, in particular if pointer types are embedded. Without function overloading it is hard to provide convenience functions creating the same kind of object with alternative parameters. Object creation would be simplified with constructors and with constructor overloading.

The lack of constructors could be mitigated with a consistent naming convention for factory function; we tried to stick to such a convention. If there is only one type in the package, the type is called the same name as the package, the function is called `New` returning an object of that type (e.g. `package ``container/list''`, `list` contains the type `List`, `list.New()` returns a List object). In cases with multiple types per package the method is to be called `New` plus the type name it creates. The convention fails if convenience functions accepting different parameters are needed.

5.6 Method and Function Overloading

Methods with the same name but different signature (number and type of input and output parameters) are called overloaded methods. The value of overloaded methods is that it allows related methods to be accessed with a common name. Overloading allows programmers to indicate the conceptual identity of different methods. The problem is that overloaded methods do not have to be related.

The system we adopted to get around overloading is to add a description of the method's parameters to the method name. It is easy to lapse on that convention and different developers can have different ideas of what a "description of the parameters" is. The necessary differentiation of the methods with differing names makes the names longer and harder to remember. Even though we designed these methods we found ourselves going back to their definition to find the right name.

On the other hand methods that are overloaded excessively can be hard to be used. It is hard to discern which method is called by reading the code, especially if the parameters of overloaded methods are of types that are subtypes of other possible parameters.

Function overloading would make convenience functions and factory functions accepting different parameters easier to declare. GO's designers claim that the benefit of overloading is not high enough to make GO's type system and with that the compiler more complex [5]. We don't think method and function overloading is a necessary feature, but we would like to have constructor overloading (see Section 5.5).

5.7 Source Code Organization

Code in Java or C++ is structured in classes with one public class per file. Source code in GO is structured in packages. A package can span multiple files. A file can contain more than one type, interface or function. Methods

don't have to be declared in the same file as the type they belong to. This requires discipline to maintain a usable structure. The GO documentation gives no indication about how source code should be organized. Should types and their methods be in one file or separated, each type in a separate file, where do interfaces go? Developers are left to their own devices which can make understanding third-party libraries hard. Defining methods outside classes makes methods hard to find, and hard to distinguish from functions in the same package. This freedom also opens up possibilities however. Related functions can be grouped in their own source file rather than being static methods of a certain type.

5.8 Syntax

The C-like syntax made the transition from Java to GO easy. Interfaces and type declarations are concise. The optional line-end semicolons are convenient and encourage one instruction per line, at the cost of having to follow rules regarding placement of braces.

5.8.1 Explicit Receiver Naming

We found having to name receivers explicitly tedious and error prone. A default receiver like `this` or `self` could ease the problem by allowing developers to name the receiver, but not forcing them to do so. Most receiver names in the standard library seem to favour brevity, using the first character of the receiver type (e.g. `func (f Foo) M()`). We mostly used `this` due to our Java background, but found reading existing code harder, because we were unsure which argument named the receiver.

5.8.2 Built-in Data Structures

GO's data structures do not provide uniform access to their members. There is no uniform way for iterating over elements of collections. `for`-loops with

a `range` clause work on maps, arrays/slices, strings and channels, and return mostly two parameters (index and retrieved element), but in case of channels only the retrieved element — without an index. Furthermore there is no convention or facility for iterating over client defined data-structures. Arrays and Slices can be copied easily, but there is no built-in functionality to copy a map.

The syntax for deleting elements from maps is unsatisfactory: `aMap[key] = aValue, false`. If the boolean on the right is `false`, the element gets deleted. The object representing the entry to be deleted (here `aValue`) has to be compatible with the values of the map. If `aValue` is of pointer type `nil` can be used, but in all other cases `aValue` has to be a value of the concrete type. If the map contains strings, the empty string `""` could be used, for numbers `0` is possible and so on. Built-in types in GO do not have methods enabling a more conventional way of deleting an element from maps.

5.8.3 Member Visibility

Having only visibility scopes "package" and "public" takes the optimistic view that packages only encapsulate a single responsibility. There is no mechanism to prevent instantiation or access to package private members within a package. Encapsulation within packages cannot be guaranteed as seen in the Singleton (Appendix A.1.5) and Memento (Appendix A.3.6) pattern. We think that using the first letter of an identifier to declare visibility is bad as it is too easily overlooked.

5.8.4 Multiple Return Values

The multiple return value facility is often used to return an error signal: the first return value is the result and the second is used to signal an error. This *"Comma OK"* idiom is encouraged by the GO authors [2] and used in the libraries; unsurprisingly, it is hard to avoid. An advantage

of this idiom is that GO's exception handling mechanism is rarely used and programs are not littered with `try...catch` blocks, which harm readability. Furthermore, values like `nil`, `EOF` or `-1` do not have to be misused as in-band error values. On the other hand, error conditions are easier ignored or forgotten because error checking is not enforced by the compiler.

5.8.5 Interface Values

Methods can be declared on value types and on pointer types. Variables of interface type can hold values and pointers. This can help to abstract away the need for a client to know if the value is of pointer type or not, but on the other hand it is inconsistent with "normal" types. From looking at the source code it is not apparent if a variable is holding a pointer or a value.

Chapter 6

Conclusions

GO's concise syntax, type inference in combination with types implicitly implementing interfaces, first class functions and closures, and fast compilation, made programming in GO very enjoyable.

GO's interfaces are easy to declare. Types implement interfaces automatically. Combined with GO's approach to polymorphism through interfaces, GO encourages good object-oriented design, as expressed in the Gang-of-Four's principle of "program to an interface, not an implementation" [39, p.18]. On the other hand, GO's unusual approach to dynamic dispatch requires the application of the Client-Specified Self pattern which is inconvenient and worsens designs.

GO lacks abstract classes. Abstract classes combine interface definitions and common functionality. GO's interface definitions are straightforward, but having to define and embed a separate type to provide default behaviour is inconvenient and also allows errors.

We found that embedding is a poor replacement for inheritance. Embedding a type publishes all of the embedded type's public members, because GO has only two visibility scopes and GO does not provide an alternative mechanism for hiding members of embedded types. Having to initialize the embedded type is inconvenient and easy to forget. The only alternative to embedding is to use composition and to forward calls. This is an all

or nothing approach, reducing the code reuse capability of embedding drastically.

Most patterns and applications can be implemented in GO to a great extent as in Java. Our implementations might not be the most idiomatic solutions, but that can tell programming language designers something too: how steep is the learning curve, how could that be improved, is better documentation needed, should certain solutions be more obvious? We hope this thesis helps to start and foster a discussion of ways to implement design patterns in GO, to discover GO specific patterns and idioms, and to help identify which features GO might want to be added or changed.

Rob Pike has described design patterns as "add-on models" for languages whose "standard model is oversold" [71]. We disagree: patterns are a valuable tool for helping to design software that is easy to maintain and extend. GO's language features have not replaced design patterns: we found that only Adapter was significantly simpler in GO than Java, and some patterns, such as Template Method, are more difficult. However, first class functions and closures simplify the Strategy, State and Command patterns. In general, we were surprised by how similar the GO pattern implementations were to implementations in other languages such as C++ and Java.

6.1 Related Work

As far as we know, we are the first to use design patterns to evaluate a programming language. Norvig analysed how features of the Dylan programming language affects implementations of design patterns and he concludes that 16 of the 23 design patterns disappear or are easier to implement with this dynamically typed language [66]. Where Norvig focusses on replacing and simplifying patterns, we focus on evaluating GO's language features and patterns are a means to that evaluation. We did not try to find the simplest pattern implementations, but those that might

be used in everyday programs.

Agerbo and Cornils analysed how and which design patterns can be supplanted with language features [8, 9]. Agerbo and Cornils took multiple languages into account to support their thesis, that design patterns should be programming language independent. We focussed on a single programming language, GO, and how GO's features affect the implementation of design patterns.

Many pattern catalogues presenting the patterns of *Design Patterns* for various programming languages have been published (e.g. Java [65], JavaScript [46], Ruby [67], Smalltalk [11], C♯ [15]). We add to this collection the appendix of this thesis: a design patterns catalogue for the GO programming language, albeit focussing primarily on the implementation of the solutions.

Empirical studies conduct experiments to evaluate and compare programming languages. Prechelt [76, 77] and Gat [41] had a number of students implement the same program in different languages and compared the languages with quantitative metrics like development time, memory usage, execution speed. In contrast, our study is qualitative and based on case studies.

6.2 Future Work

GO is a systems language with a focus on concurrency and networking. In this thesis we implemented general purpose patterns in GO. We'd like to extend this study by implementing concurrency and networking patterns [84] in GO to evaluate GO's features in that area.

GO is a new programming language combining existing and new concepts. We have a strong Java background. A catalogue of GO idioms and GO specific patterns would help to make learning GO and developing GO idiomatic programs easier. In this study we implemented existing patterns in GO. In a follow-up study we will investigate GO applications

and libraries to collect idioms and patterns that have emerged.

The interfaces in GoHotDraw are bigger than GO's preferred two, one or even none methods [71]. Refactoring GoHotDraw towards smaller interfaces could discover valuable insights into the differences between GO and Java programs.

6.3 Summary

In this thesis we have presented an evaluation of the GO programming language; GoHotDraw, a GO port of the pattern dense drawing application framework JHotDraw; and a catalogue of GO implementations of all 23 Design Patterns described in *Design Patterns* [39]. We have found that most pattern implementations in GO are similar to implementations in Java, that GO's approach to dynamic dispatch on subtypes complicates designs, and that embedding has major drawbacks compared to inheritance.

Appendix A

Design Pattern Catalogue

In this Appendix we present our GO implementations of all 23 design pattern from *Design Patterns*. We organise the patterns in the same order as they appear in *Design Patterns*. There are three sections: Creational Patterns, Structural Patterns and Behavioural Patterns. Each pattern description consists of Intent, Context, Examples and Discussion.

A.1 Creational Patterns

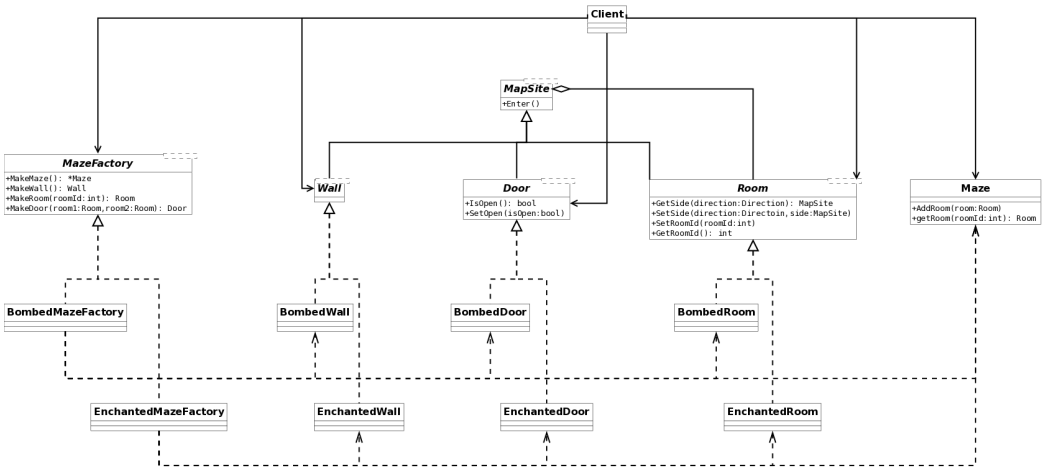
Creational patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

A.1.1 Abstract Factory

Intent Insulate the creation of families of objects from their usage without specifying their concrete types.

Consider the creation of a mazes for a fantasy game. Mazes can come in a variety of types such as enchanted or bombed mazes. A maze consists of map sites like rooms, walls and doors. Instantiating the right type of map sites throughout the application makes it hard to change the type of maze.

A solution is a factory interface that defines methods to create an instance of each interface that represents a map site.



Each map site (Door, Wall, Room) is an *Abstract product*. Concrete factories implement the methods to create instances of concrete map site types for the same maze type. An enchanted factory creates a maze with enchanted rooms, enchanted doors and enchanted wall.

Example The following example shows how a maze with bombed map sites can be created using a factory.

The following listing shows four interfaces. `MapSite` defines the `Enter` method. The other interfaces embed `MapSite` – gaining Enter – and list further methods. That means concrete doors, rooms and walls have to implement their respective methods and `Enter` as well.

```
type MapSite interface {
  Enter()
}

type Wall interface {
  MapSite
}

type Door interface {
  MapSite
  IsOpen() bool
  SetOpen(isOpen bool)
}

type Room interface {
  MapSite
  GetSide(direction Direction) MapSite
  SetSide(direction Direction, side MapSite)
  SetRoomId(roomId int)
  GetRoomId() int
}
```

`BombedDoor` is a concrete product, an implementation of the `Door` interface. `BombedDoor` keeps references to its connecting rooms and knows if it is open or not.

```
type BombedDoor struct {
  room1  Room
  room2  Room
  isOpen bool
}
```

`NewBombedDoor` instantiates a `BombedDoor` object and sets its rooms. `isOpen` is initialized to `false` per default.

```
func NewBombedDoor(room1 Room, room2 Room) *BombedDoor {
  return &BombedDoor{room1: room1, room2: room2}
}
```

`IsOpen` and `SetOpen` are the getter and setter for `isOpen`.

```
func (door *BombedDoor) IsOpen() bool {
  return door.isOpen
}

func (door *BombedDoor) SetOpen(isOpen bool) {
  door.isOpen = isOpen
}
```

`Enter` prints a message according to `isOpen`.

```
func (door *BombedDoor) Enter() {
  if door.isOpen {
    fmt.Println("Enter bombed door")
  } else {
    fmt.Println("Can't enter bombed door. Closed.")
  }
}
```

`String` returns a string describing the door. The method name `String` is a convention. GO's formatting package *fmt* uses the interface `Stringer` with `String` as its only method. The idea being that types implementing `String` can be used in the `fmt` package for formatted printing. `String` is the equivalent to Java's `toString`.

```
func (door *BombedDoor) String() string {
  return fmt.Sprintf("A bombed door between %v and %v",
    door.room1.GetRoomId(), door.room2.GetRoomId())
}
```

The next listing shows the type `Maze` which maintains a vector. Vector is not generic and its elements can be of any type, more precisely any type implementing the (empty) interface `interface`. The other two methods handle adding and retrieval of rooms from a Maze object.

```
type Maze struct {
  rooms *vector.Vector
}

func NewMaze() *Maze {
  return &Maze{rooms: new(vector.Vector)}
}
```

`AddRoom` and `GetRoom` handle adding and retrieval of rooms from a `Maze` object.

```
func (maze *Maze) AddRoom(room Room) {
  maze.rooms.Push(room)
}

func (maze *Maze) GetRoom(roomId int) Room {
  for i := 0; i < maze.rooms.Len(); i++ {
    currentRoom := maze.rooms.At(i).(Room)
    if currentRoom.GetRoomId() == roomId {
      return currentRoom
    }
  }
  return nil
}
```

`MazeFactory` is an interface listing a method for each Abstract Product.

```
type MazeFactory interface {
  MakeMaze() *Maze
  MakeWall() Wall
  MakeRoom(roomNo int) Room
  MakeDoor(room1 Room, room2 Room) Door
}
```

`BombedMazeFactory` is an implementation of the `MazeFactory` interface. `MakeMaze` returns an instance of a plain maze. Each of the other methods return an instance of the bombed product family. `MakeWall`, for example, returns an instance of `BombedWall`. The static types of the returned objects are of the interface type, but dynamic types are of the bombed variety. It is transparent to clients of the `Make` methods which dynamic type is returned.

```
type BombedMazeFactory struct{}

func (factory *BombedMazeFactory) MakeMaze() *Maze {
  return NewMaze()
}

func (factory *BombedMazeFactory) MakeWall() Wall {
  return new(BombedWall)
}
```

```
func (factory *BombedMazeFactory) MakeRoom(roomNo int) Room {
  return NewBombedRoom(roomNo)
}

func (factory *BombedMazeFactory) MakeDoor(room1 Room, room2 Room) Door {
  return NewBombedDoor(room1, room2)
}
```

The function `CreateMaze` creates an instance of `Maze` and returns a pointer to the object. The maze has two rooms with a door in between. The concrete map site types depend the dynamic type of the parameter `MazeFactory` (i.e. consider factory being of type `BombedFactory`, then the rooms, doors and walls will be of the bombed product family).

```
func CreateMaze(factory MazeFactory) *Maze {
  aMaze := factory.MakeMaze()
  room1 := factory.MakeRoom(1)
  room2 := factory.MakeRoom(2)
  aDoor := factory.MakeDoor(room1, room2)

  aMaze.AddRoom(room1)
  aMaze.AddRoom(room2)

  room1.SetSide(North, factory.MakeWall())
  room1.SetSide(East, aDoor)
  room1.SetSide(South, factory.MakeWall())
  room1.SetSide(West, factory.MakeWall())

  room2.SetSide(North, factory.MakeWall())
  room2.SetSide(East, factory.MakeWall())
  room2.SetSide(South, factory.MakeWall())
  room2.SetSide(West, aDoor)

  return aMaze
}
```

Here we show how Clients use a bombed maze factory to create a maze.

```
  var factory *BombedMazeFactory
  var maze *Maze
  maze = CreateMaze(factory)
  maze.GetRoom(1).Enter() //Prints: Can't enter bombed door. Closed.
```

Discussion The interfaces `Wall`, `Door` and `Room` embed the `MapSite` interface. `MapSite`'s `Enter` is added to the embedding interfaces. Embed-

ding an interface in another interface is similar to Java's interface extension.

A default factory type could provide common behaviour for concrete factories. Concrete factories would embed default factory. The factory interface with the embedded default type emulates abstract types.

The Abstract Factory pattern describes a Create method accepting a factory object. The method uses the factory to create and return the end product. From the structure diagram of the Abstract Factory pattern as described by the GoF it is not apparent where this method is supposed to live. Functions in GO are not tied to a type like in Java or C++. Functions are independent. The Abstract Factory pattern can take advantage of that. There is no need to introduce a separate type just to hold the Create method. Create can just be function since the receiver of this function would not be used anyway.

A.1.2 Builder

Intent The intention is to abstract steps of construction of objects so that different implementations of these steps can construct different representations of objects.

Context Consider the creation of mazes for a fantasy game. Mazes are complex object structures. Mazes consist of map sites like doors, walls and rooms. The map sites come in a variety of types like standard, enchanted or bombed. The creation process of mazes should be independent of the type of the map sites and not have to change when new types of doors, walls and rooms are added.

A solution is a builder interface defining methods for for creation of parts of the end product. Concrete builders implement the builder interface; construct and assemble the parts; maintain a reference to the end product they create; and provide means for retrieving the end product. A director construct an object by using the builder interface.

Example In the following we demonstrate how a maze with standard map sites can be created using a builder.

The `MazeBuilder` interface is the common builder interface concrete builders implement. The method `GetMaze` provides access to the end product the builder creates and there are build method for each part.

```
type MazeBuilder interface {
  GetMaze() *Maze
  BuildMaze()
  BuildRoom(roomId int)
  BuildDoor(roomId, room2 int)

}
```

This listing shows the declaration of `StandardMazeBuilder` which has a reference to a `Maze` object (the end product). `GetMaze` provides public access to the end product.

```
type StandardMazeBuilder struct {
  maze *Maze
}
func (builder *StandardMazeBuilder) GetMaze() *Maze {
  return builder.maze
}
```

`BuildMaze`, `BuildRoom` and `BuildDoor` each create a map site in the standard variety. `BuildMaze` initializes the end product with a new maze object. `BuildRoom` instantiates a new room if the room does not exist. The room gets four standard walls. `BuildDoor` retrieves the rooms that the door is to connect. A new standard door is is inserted at each room.[1] Note that each build function operates on the same `maze` object. Each build step contributes to the creation of a whole end product.

```
func (builder *StandardMazeBuilder) BuildMaze() {
  builder.maze = NewMaze()
}

func (builder *StandardMazeBuilder) BuildRoom(roomId int) {
  if builder.maze.GetRoom(roomId) == nil {
    room := NewStandardRoom(roomId)
    builder.maze.AddRoom(room)
    room.SetSide(North, new(StandardWall))
    room.SetSide(East,  new(StandardWall))
    room.SetSide(South, new(StandardWall))
    room.SetSide(West,  new(StandardWall))
  }
}

func (builder *StandardMazeBuilder) BuildDoor(roomId1 int, roomId2 int) {
  room1 := builder.maze.GetRoom(roomId1)
  room2 := builder.maze.GetRoom(roomId2)
  door := NewStandardDoor(room1, room2)
  room1.SetSide(West, door)
  room2.SetSide(East, door)
}
```

`BuildMaze` uses a `MazeBuilder` object to create a maze. The map site types of the maze depend on the dynamic type of the builder object. The method takes an object of type `MazeBuilder` and calls the builder's

[1] For simplicity the door replaces the east and the west wall without checking the validity of that operation.

`Build` methods. Each step of `BuildMaze` adds a part to the end product which the `builder` object keeps track off.

```
func BuildMaze (builder MazeBuilder) *Maze {
  builder.BuildMaze()
  builder.BuildRoom(1)
  builder.BuildRoom(2)
  builder.BuildDoor(1, 2)
  return builder.GetMaze()
}
```

The code below shows how clients use a builder to create a maze with standard map sites. A builder gets instantiated; the builder object is passed to `BuildMaze`, which creates the maze calling `builder`'s Build methods; `maze` is assigned the maze `builder` created; and finally `maze`'s structure is printed to the console.

```
  builder:= NewStandardMazeBuilder()
  BuildMaze(builder)
  maze := builder.GetMaze()
  fmt.Print(maze)
```

Discussion *Design Patterns* describes a distinct Director type with a Construct method. This implementation of Builder omits the separate Director type. The task of a Director is to "construct an object using the Builder interface"[39]. In GO, a distinct Director type is not necessary. Directors don't store state of the product nor the builder they use to create the product. Director is a mere vessel for Construct methods. The Director's Construct method could be a function making the Director type superfluous. However, a Director type could be used to group different Build methods together.

If common functionality between builders is identified, a default builder type should be implemented. The default builder could also maintain the reference to the product it creates. Concrete maze builders would embed the default builder to gain the common functionality and the reference to the product.

A.1.3 Factory Method

Intent Define an interface for creating an object, but let subclasses decide which type to instantiate.

Context Consider a frameworks for documents. Applications will have to use different types of documents like resumes or reports and each type of document will have different pages. Adding new types of documents will complicate the creation of documents, since the framework doesn't know which concrete document type it should instantiate.

The Factory method offers a solution. In the document interface a method for creating the pages is defined and concrete documents implement this method. Letting subtypes decide how to create the pages of a document encapsulates that knowledge out moves it out of the framework. The page creating method is the factory method because it "manufactures" objects.

The Factory Method pattern can be implemented by using methods or by using functions. We discuss both alternatives.

Example - Factory *methods* In this example we describe the implementation of document-creating factory methods.

`Document` defines the common interface. `CreatePages` is the factory method concrete document types implement.

```
type Document interface {
  CreatePages()
}
```

`DefaultDocument` provides common behaviour for document subtypes. Concrete documents can embed `DefaultDocument` to inherit the member `Pages`, a reference to the pages of a document. `Pages` is a public vector, to grant `DefaultDocument`-embedding types access, even when they are defined outside of `DefaultDocument`'s package. The `String` method returns the document's pages concatenated with a comma.

```
type DefaultDocument struct {
  Pages *vector.StringVector
}

func NewDefaultDocument() *DefaultDocument {
  return &DefaultDocument{new(vector.StringVector)}
}
```

The `String` method returns the document's pages concatenated with a comma.

```
func (this *DefaultDocument) String() string {
  var result string
  for i := 0; i < this.Pages.Len(); i++ {
    page := this.Pages.At(i)
    result += fmt.Sprintf("%v, ", page)
  }
  return result
}
```

`Resume` and `Report` are implementations of the interface `Document`. Both types define `CreatePages` methods, adding pages (here simple strings) to their `Pages` member they gained from embedding `DefaultDocument`.

```
type Resume struct {
  *DefaultDocument
}

func NewResume() *Resume {
  return &Resume{NewDefaultDocument()}
}

func (this *Resume) CreatePages() {
  this.Pages.Push("personal data")
  this.Pages.Push("education")
  this.Pages.Push("skills")
}

type Report struct {
  *DefaultDocument
}

func NewReport() *Report {
  return &Report{NewDefaultDocument()}
}
```

```
func (this *Report) CreatePages() {
  this.Pages.Push("introduction")
  this.Pages.Push("background")
  this.Pages.Push("conclusion")
}
```

Depending on the dynamic type of the `Document` object `doc` the method `CreatePages` behaves differently.

```
  var doc Document

  doc = NewResume()
  doc.CreatePages()
  fmt.Println(doc)

  doc = NewReport()
  doc.CreatePages()
  fmt.Println(doc)
```

Example - Factory *functions* A problem with factory methods is that a new type is necessary to create the appropriate product objects. We take advantage of GO's function type `func`.

In the following example we look at the same problem as above: Creation of documents with different pages. But instead of defining separate types for object creation we declare functions encapsulating the creation process.

`Document` maintains a reference to its pages. The method `CreatePages` create the pages. `CreatePages`' parameter is a function returning a pointer to a Vector. `CreatePages` calls the passed function `createPages` and assigns the return value to `pages`. The method `CreatePages` accepts functions that have no parameters and return a pointer to a vector object (`func() *vector.Vector`). Two examples of functions that can be passed to `CreatePages` are shown in the next listing.

```
type Document struct {
  Pages *vector.Vector
}
func (this *Document) CreatePages(createPages func() *vector.Vector) {
  this.Pages = createPages()
}
```

`CreateResume` and `CreateReport` create a vector, add the necessary pages, and return the result. Both methods have a signature that is accepted by `Document`'s `CreatePages`.

```
func CreateResume() *vector.Vector {
  pages := new(vector.Vector)
  pages.Push("personal data")
  pages.Push("education")
  pages.Push("skills")
  return pages
}

func CreateReport() *vector.Vector {
  pages := new(vector.Vector)
  pages.Push("introduction")
  pages.Push("background")
  pages.Push("conclusion")
  return pages
}
```

The following listing shows how the factory functions `CreateResume` and `CreateReport` are used to create the pages of a `Document` object `doc`.

```
  var doc Document = new(Document)

  doc.CreatePages(CreateResume)
  fmt.Println(doc)

  doc.CreatePages(CreateReport)
  fmt.Println(doc)
}
```

Discussion The first example is an implementation of the classic Factory Method. An interface for product creation that subtypes implement. The second example uses functions to avoids having to define a new type in order to create different products.

In the first example a type is used to provide default behaviour. This type is meant to be embedded, not to be instantiated. GO does not provide a mechanism to avoid instantiating types.

A.1.4 Prototype

Intent Create objects by cloning a prototypical instance.

Context Consider again mazes for a fantasy game and the creation of mazes with mixed map site types (bombed, enchanted or standard). Abstract Factory creates objects of the same family.

The Prototype pattern offers a solution. The products have a common interface for cloning itself. Concrete products implement that interface to return a copy of itself. New instances of the products are created by asking the prototypical products to clone themselves. This approach can be combined with the Abstract Factory pattern. The factory maintains references to prototypical objects and depending on the dynamic type of the prototype objects the factory clones the prototype object, and initialises and returns it. The dynamic type of the object determines what type of product the factory is to create.

Example Once again we employ the example of creating mazes for a fantasy game. This time we use a prototype factory to create mazes with mixed type map sites (bombed rooms, enchanted doors and standard walls). All prototypical objects have to implement a Clone method, returning a copy of themselves.

`MazePrototypeFactory` maintains references to a maze and to prototypes of map sites. `NewMazePrototypeFactory` constructs a maze prototype factory object with the passed prototypical objects.

```
type MazePrototypeFactory struct {
  prototypeMaze *Maze
  prototypeRoom Room
  prototypeWall Wall
  prototypeDoor Door
}

func NewMazePrototypeFactory(
    maze *Maze, room Room, wall Wall, door Door) *MazePrototypeFactory {
  this := new(MazePrototypeFactory)
```

```
  this.prototypeMaze = maze
  this.prototypeRoom = room
  this.prototypeWall = wall
  this.prototypeDoor = door
  return this
}
```

To use map sites with the prototype factory we add the `Clone` method to the `MapSite` interface, which is described in Appendix A.1.1. We can't put the Clone method in MapSite, since we want to avoid having to cast the returned object. Each Product type interface has to declare its own Clone method with the return value type being of the right product type.

```
type MapSite interface {
  //snip
  Clone() Door
}
```

The `Make` methods are factory methods used in the template "method" `CreateMaze` (see Appendix A.1.1). The factory methods copy the factories prototypes by calling `Clone` on them. `MakeDoor` and `MakeRoom` do further initialization of the cloned prototype. `Clone` for `EnchantedDoor` is shown in the next listing.

```
func (this *MazePrototypeFactory) MakeMaze() *Maze {
  return this.prototypeMaze.Clone()
}

func (this *MazePrototypeFactory) MakeWall() Wall {
  return this.prototypeWall.Clone()
}

func (this *MazePrototypeFactory) MakeDoor(room1, room2 Room) Door {
  door := this.prototypeDoor.Clone()
  door.SetRooms(room1, room2)
  return door
}

func (this *MazePrototypeFactory) MakeRoom(roomId int) Room {
  room := this.prototypeRoom.Clone()
  room.SetRoomId(roomId)
  return room
}
```

`Clone` instantiates a new EnchantedDoor object and reassigns its rooms and if it is open or not.

```
func (this *EnchantedDoor) Clone() Door {
  door := new(EnchantedDoor)
  door.room1 = this.room1
  door.room2 = this.room2
  door.isOpen = this.isOpen
  return door
}
```

Discussion Implementing the Clone method correctly is – like in many other languages – the the hardest part, especially when the object contain circular references. Copy methods could be working like C++'s copy constructors, but that still doesn't solve the problem of shallow versus deep copying. A solution would be the serialization of objects into streams of bytes. GO's gob package manages binary values exchanged between an Encoder (transmitter) and a Decoder (receiver). Objects would be Encoded into a stream of bytes, sent to a Writer object and received by an Reader and the decoded by a Decoder.

A.1.5 Singleton

Intent Ensure that only a single instance of a type exists in a program, and provides a global access point to that object.

Context Consider a registry for services. There should always be only one instance of a registry per application.

The Singleton pattern provides a solution by providing a single way to create and get hold of the object. The type which should be instantiated needs to be a package private type. A public function ensures that there is only one instance of the private type at a time. The private type can define public members to give clients access to its state and functionality.

Example - Package encapsulates Singleton The following example outlines the implementation of a registry of which we only want to have one instance. The following two listings reside inside the package `registry`.

The struct `registry` cannot be named outside the `registry` package, so it can only be instantiated inside the package. The variable `instance` is a reference to the only instantiation of the type `registry`.

```
package registry

type registry struct {
  state *State
}

var instance *registry
```

The public function `Get` provides access to the singleton `instance`, creating a new object if none exists when it is called.

```
func Get() *registry {
  if instance == nil {
    instance = &registry{}
  }
  return instance
}

func (this *registry) GetState() *State { ... }
```

The function `Get` checks if `instance` is `nil` (`nil` is GO's null pointer). `Get` returns the pointer of the object stored in `instance`. Only if `instance` is `nil`, a new instance is created, otherwise the existing one will be returned. `Get` uses lazy initialization. The object `instance` is only initialized when needed. The alternative would be to initialize `instance` on package import:

```
var instance = new(registry)
```

The only way for clients to get hold of an instance of type `registry.registry` is by calling `registry.Get`. Consider the following code examples outside of the package `registry`:

```
aRegistry := registry.Get()
```

The following does not work:

```
aRegistry := new(registry.registry)
```

Even though the type registry is not visible, its method `GetState` is. Clients can use the public functions and public members of private types:

```
aRegistry.GetState()
```

For documentation purposes the `registry` package could define a public interface `Registry` listing `registry`'s public methods.

Example - Package as Singleton As an alternative to encapsulating a singleton type in a package, packages themselves could be used as singletons. Singleton-types become packages; singleton-type members become static variables; methods become functions. Consider the following code example:

```
package registry

var state *State

func init() {
  state = new(State)
}
```

The package `registry` itself acts as singleton. The singleton's state is kept in static variables, here in `state` of type `State`. The object initialization is now done in the `init` function instead of in `Get`.

```
func GetState() *State {
  return state
}
```

Packages cannot be instantiated, therefore there is no need to control or limit instantiation. Clients can use the singleton's functionality by importing the package and interacting with it directly:

```
import (
  registry1 "registry"
  registry2 "registry"
)
```

Packages can be imported multiple times. The package `registry` is imported twice with two different names, but both names refer to the same "instance" of the package. The `init` method is executed only once, even though there are two references to the same package.

```
registry1.GetState() == registry2.GetState() // true
```

The aliases `registry1` and `registry2` refer to the same package. Calling `GetState` on either returns the same object.

Discussion The Singleton pattern is usually implemented by hiding (using scope `private`) the singleton's constructor. In GO, constructors cannot be made private. The examples above show two ways to limit object instantiation by using GO's package access mechanisms. The first option is to declare the singleton type's scope as package. Clients outside the package can't create instances of the un-exported type. The other option is to use a package as singleton. Packages can't be instantiated and package initialization is only done once. The static package variables are the singleton's state.

The Singleton pattern is only effective for clients outside the package a singleton is declared. Within the singleton's package multiple instances

can be created, since GO does not have a private scope; in particular, this might be done inadvertently by embedding `registry`.

A.2 Structural Patterns

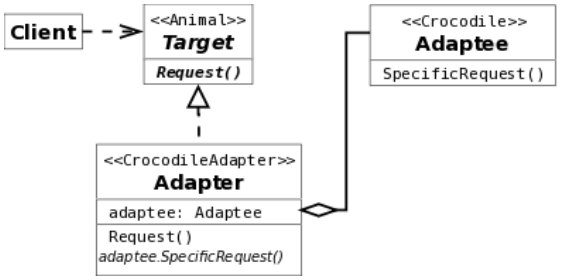
Structural design patterns identify simple ways to realize relationships between entities.

A.2.1 Adapter

Intent Translate the interface of a class into a compatible interface. An adapter allows classes to work together that normally could not because of incompatible interfaces.

Context Consider a framework for animals. The core of the framework is an interface defining the behaviour of animals. Now, we want to incorporate a third-party library that provides reptiles, but the problem is that the reptiles can't be used directly because of incompatible interfaces.

The solution is to provide a type (the *Adapter*) that translates calls to its interface (the *Target*) into calls to the original interface (the *Adaptee*). The diagram below shows this structure.



We show two alternatives of implementing the adapter pattern in GO. The first example uses embedding and the second object composition.

Example - Embedding In the following example we adapt a Reptile object to the Animal interface.

`Animal` is the *Target* interface that concrete animals have to implement.

```
package animals

type Animal interface {
  Move()
}
```

The type `Cat` is a concrete animal, since it implements the method `Move`.

```
type Cat struct { ... }

func (this *Cat) Move() { ... }
```

The package reptiles provides the type `Crocodile`. `Crocodile` has the method `Slither`. `Crocodile` is the *Adaptee*. We want be able to use `Crocodile` where an `Animal` is required. We need to adapt the `Slither` method to a `Move` method.

```
package reptiles

type Crocodile struct { ... }

func (this *Crocodile) Slither() { ... }
```

The struct `CrocodileAdapter` adapts an embedded crocodile so that it can be used as an `Animal`. `CrocodileAdapter` objects implicitly implement the `Animal` interface. Clients of `CrocodileAdapter` can call `Move` as well as `Slither` and any other public behaviour `reptiles.Crocodile` may provide.

```
package animal

type CrocodileAdapter struct {
  *reptiles.Crocodile
}

func NewCrocodile() *CrocodileAdapter {
  return &CrocodileAdapter{new (reptiles.Crocodile)}
}

func (this *CrocodileAdapter) Move() {
  this.Slither()
}
```

Clients of package `animals` can use `Cat`s and `Crocodile`s where an object of type `Animal` is required. It is transparent to clients that `Crocodile`'s implementation is provided by a third-party library.

```
import "animals"
...
var animal animals.Animal
animal = new(animals.Cat)
animal.Move()
animal = animals.NewCrocodile()
animal.Move()
```

Example - Composition Alternatively, we could use composition rather than embedding in our adapter object. The following listing highlights the necessary changes.

`CrocodileAdapter` has a reference to a `reptiles.Crocodile` object. As with embedding the object still has to be initialized, therefore we still need `NewReptile()`.

```
type CrocodileAdapter struct {
  cocodile *reptiles.Crocodile
}

func (this *CrocodileAdapter) Move() {
  this.crocodile.Slither()
}
```

Discussion Both examples we showed here are object adapters, even the embedding example, since the embedded type is actually an object.

Having the choice between embedding and composition is a similar to class-based languages where an adapter can use inheritance or composition. Composition should be used when the public interface of the adaptee should remain hidden. A minor drawback of composition is that the call of `Slither` needs to be fully quantified (`this.crocodile.Slither()`). Embedding exhibits the adaptee's entire public interface but reduces bookkeeping (`this.Slither()`).

The embedded type `reptiles.Crocodile` of `CrocodileAdapter` has to be initialized (to avoid nil reference errors), therefore we need `NewCrocodile`. The initialization of embedded types is an additional burden we would like to avoid.

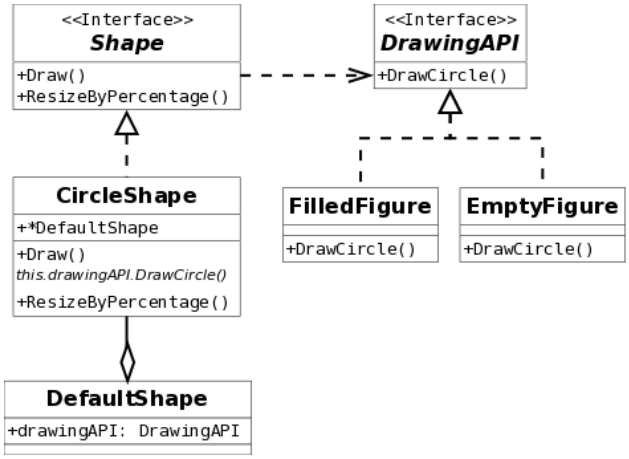
A.2.2 Bridge

Intent Avoid permanent binding between an abstraction and an implementation to allow independent variation of either.

Context Consider a framework for drawing shapes. There are different shapes and have to be displayed in multiple ways: bordered, unbordered etc. A permanent binding between the shape and how they are displayed should be avoided so that the two can vary independently.

The bridge pattern offers a solution, by separating abstraction from implementation. The shapes are the abstractions and how they are rendered is the implementation.

Example In this example we decouple shapes (the abstractions) from their drawing API (the implementation) by implementing the Bridge pattern.



The above diagram shows the structure of the Bridge pattern we are using in this example. Abstractions (here Shapes) use the Implementor interface (here DrawingAPI). Shapes are not concerned how the DrawingAPI is implemented. A CircleShape maintains a reference to a DrawingAPI object by embedding DefaultShape. The dynamic type of the DrawingAPI

object determines if the shape will be drawn as a filled or empty figure. This decouples the implementation details from the abstraction.

The following listings show how this can be implemented in GO.

```
type Shape interface {
  Draw()
  ResizeByPercentage(pct float64)
}

type DefaultShape struct {
  drawingAPI DrawingAPI
}

func NewDefaultShape(drawingAPI DrawingAPI) *DefaultShape {
  return &DefaultShape{drawingAPI}
}
```

The `Shape` interface defines two methods that all concrete shapes have to implement. The type `DefaultShape` is meant to be embedded by all concrete shapes and maintains a reference to a `DrawingAPI` object of interface type. It is transparent to concrete shapes how the API is implemented. `DefaultShape` would also hold common behaviour for shapes.

```
type CircleShape struct {
  x,y,radius float64
  *DefaultShape
}

func NewCircleShape(x,y,radius float64, drawingAPI DrawingAPI) *CircleShape{
  this := new(CircleShape)
  this.x, this.y ,this.radius = x, y, radius
  this.DefaultShape = NewDefaultShape(drawingAPI)
  return this
}
```

CircleShape is a concrete shape with coordinates of its center; a radius; and it embeds DefaultShape, inheriting a reference to a DrawingAPI object. NewCircleShape is a constructor method initializing the location, radius and the embedded type.

```
func (this *CircleShape) Draw() {
  this.drawingAPI.DrawCircle(this.x, this.y, this.radius)
}
```

```
func (this *CircleShape) ResizeByPercentage(pct float64) {
  this.radius *= pct
}
```

`Draw` is the implementation specific method, making use of the implementation object. `Draw` calls the `DrawCircle` on its `drawingAPI` object. The dynamic type of `drawingAPI` determines how the circle will be drawn. `ResizeByPercentage` changes the circle's radius and abstraction specific, only operating on the abstraction, not on the drawing API.

```
type DrawingAPI interface {
  DrawCircle(x,y,radius float64)
}

type FilledFigure struct {}

func (this *FilledFigure) DrawCircle(x,y,radius float64) {
  //draw a filled circle
}

type EmptyFigure struct {}

func (this *EmptyFigurs) DrawCircle(x,y,radius float64) {
  //draw an empty circle
}
```

`DrawingAPI` is the main interface for implementations. Concrete implementations of `DrawingAPI` implement `DrawCircle`. The example above implements two concrete drawing APIs with different implementations of `DrawCircle`.

```
func main() {
  filledShape := NewCircleShape(1, 2, 3, new(FilledFigure))
  emptyShape = NewCircleShape(5, 7, 11, new(EmptyFigure))

  filledShape.ResizeByPercentage(2.5)
  filledShape.Draw()

  emptyShape.ResizeByPercentage(2.5)
  emptyShape.Draw()
}
```

The above listing shows how clients can use this implementation of the Bridge pattern. Two shape objects get instantiated, the abstractions. The

interesting part is that they get initialized with different Implementation objects, thus changing how their Draw method draws the shapes.

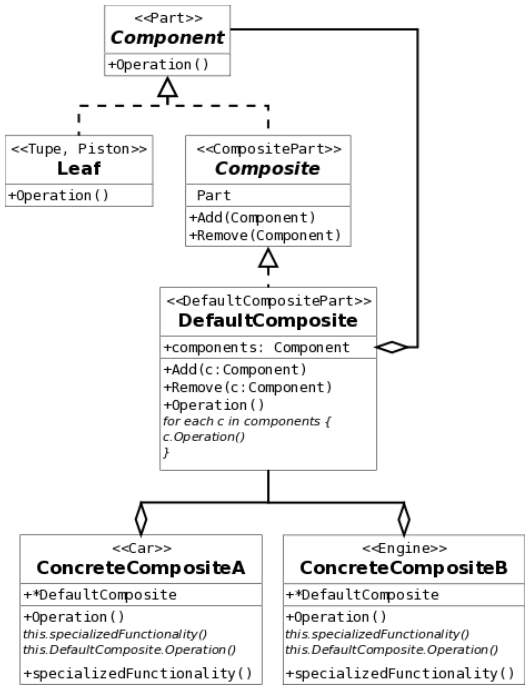
Discussion Shape in the original pattern is an abstract class to define the interface and to maintain a reference to a DrawingAPI object. GO doesn't have abstract classes, so we use the combination of an interface (Shape) and provide a default type (DefaultShape) with that reference (and common functionality if necessary). Alternatively each refined abstraction could maintain their own reference. The advantage of embedding DefaultShape become apparent when DefaultShape implemented common functionality for refined abstractions.

A.2.3 Composite

Intent Treat a group of objects in the same way as a single instance of an object and create tree structures of objects.

Context Consider the assembly of car. A car is made of parts and each part is mad of other parts in turn. A car has an engine, an engine has pistons, and so on. Parts can be composed to form larger components. The problem is that we don't want to treat simple parts and parts that are made of other parts differently.

The Composite pattern offers a solution. The key is to define an interface for all components (primitives and their containers) to be able to treat them uniformly.



All elements of the structure (*Leafs* as well as *Composites*) implement the *Component* interface. The interface Composite is for components that are made of sub components, which in turn can be Composites. *Default*

Composite provides common functionality for *Concrete Composites*. The method Operation in Default Composite is called recursively on all of its components. Concrete Composites embed Default Composite and provide their implementation of Operation, which is called by Default Composite's method Operation.

Example Here we look at the car example mentioned above. Cars have engines and tires; engines have pistons; tires have tubes. Each of the car's components is a part. Parts that are assemblies of other parts are composite parts. At the end of the assembly process we ask the car to check all of its parts if they are working correctly, and the sub parts will ask their sub parts.

`Part` is the main interface of all components of a car. All elements (nodes and leaves) of the composite structure will have to implement the `CheckFunctionality` method.

```
type Part interface {
  CheckFunctionality()
}
```

Piston implements the Part interface and is a Leaf: Piston doesn't have sub parts. There are other parts like Tubes we omitted for brevity.

```
type Piston struct {}

func NewPiston() *Piston {
  return new(Piston)
}

func (this *Piston) CheckFunctionality() {
  fmt.Println("up and down")
}
```

To bundle all functionality that is common to parts that have sub parts, we define the `CompositePart` interface. `CompositePart` lists the `Add` and `Remove` methods and embed the `Part` interface, adding the `CheckFunctionality` method to the interface. Implementations of CompositePart will have to implement all four methods.

```
type CompositePart interface {
  Add()
  Remove()
  Part
}
```

`DefaultCompositePart` defines common functionality for composite parts and is meant to be embedded. `DefaultCompositePart` maintains a vector to store the sub parts.

```
type DefaultCompositePart struct {
  parts *vector.Vector
}

func NewDefaultCompositePart() *DefaultCompositePart {
  return &DefaultCompositePart{parts:new(vector.Vector)}
}
```

The vector `parts` can be manipulated with the `Add` and `Remove` methods. This functionality is common to all composite parts.

```
func (this *DefaultCompositePart) Add(part Part) {
  this.parts.Push(part)
}

func (this *DefaultCompositePart) Remove(part Part) {
  for i := 0; i < this.parts.Len(); i++ {
    currentPart := this.parts.At(i).(Part)
    if currentPart == part {
      this.parts.Delete(i)
    }
  }
}
```

`CheckFunctionality` recursively calls `CheckFunctionality` on all of its parts (which can in turn be of type `CompositePart`).

```
func (this *DefaultCompositePart) CheckFunctionality() {
  for i := 0; i < this.parts.Len(); i++ {
    currentPart := this.parts.At(i).(Part)
    currentPart.CheckFunctionality()
  }
}
```

In our example cars are composite parts enabling cars to contain parts. `Car` implements `CompositePart` since it embeds `DefaultComposite-`

`Part`. `CheckFunctionality` overrides the embedded `DefaultCompositePart.CheckFunctionality`. `CheckFunctionality` prints a message on the console and then calls `CheckFunctionality` of the embedded type. The implementation of `Engine` and `Tire` are similar to `Car`, but omitted for brevity.

```
type Car struct {
  *DefaultCompositePart
}

func NewCar() *Car {
  return &Car{NewDefaultCompositePart()}
}

func (this *Car) CheckFunctionality() {
  fmt.Println("move")
  this.DefaultCompositePart.CheckFunctionality()
}
```

The following creates a composite object structure: an engine with two pistons, two tires with a tube, and a car with that engine and the two tires. `CheckFunctionality` called on car traverses the structure and each part prints a status message.

```
car := NewCar()
engine := NewEngine()
tire1 := NewTire()
tire2 := NewTire()

engine.Add(NewPiston())
engine.Add(NewPiston())
tire1.Add(NewTube())
tire2.Add(NewTube())
car.Add(engine)
car.Add(tire1)
car.Add(tire2)

car.CheckFunctionality()
```

Discussion Like in Java, interfaces can be extended in GO too. This is done by one interface embedding another interface. Interfaces can embed multiple interfaces. In the example above the interface `CompositePart`

embeds the interface `Part` and "inherits" all of `Part`'s methods.

Composite objects have functionality in common, like Add or Remove. Since GO has no abstract classes to encapsulate common functionality a separate type doing just that has to be implemented. Types wishing to inherit from the "abstract" type have to embed this default type. In the above example the default type `DefaultCompositePart` implemented all of `CompositePart`'s methods. Types embedding `DefaultCompositeType` automatically fulfil the `CompositePart` and therewith the `Part` interface.

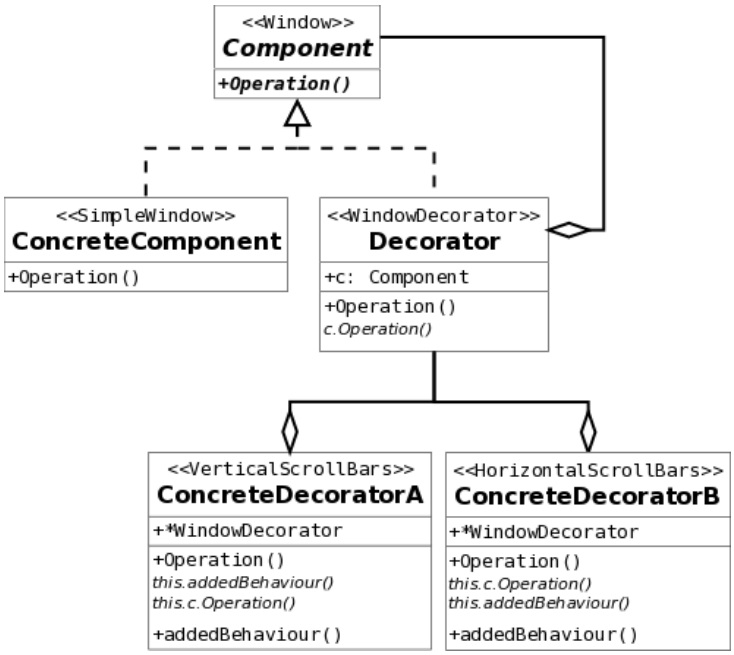
`Car`'s `CheckFunctionality` overrides the `CheckFunctionality` method of the embedded type `DefaultCompositeType`. However, it is still possible to call the overloaded function by fully qualifying the method. An embedded type is actually just an ordinary object of the embedded type. The object can be accessed by its type name. This enabled us to call `DefaultCompositeParts`'s `CheckFunctionality` inside the overriding method, akin to a `super` call in Java:
`aConcreteComposite.DefaultCompositePart.CheckFunctionality()`

A.2.4 Decorator

Intent Augment an object dynamically with additional behaviour.

Context Consider a window based user interface. Depending on their content, windows can either have horizontal, vertical, none or both scrollbars. The addition and removal of scrollbars should be dynamic and windows should not be concerned about scrollbars. The scrollbars should not be part of the window.

The Decorator pattern offers a solution. Windows are going to be *decorated* with scrollbars. The image below shows the structure of the Decorator pattern.



A *Decorator* (window decorator) wraps the *Component* (window) and adds additional functionality (scrollbars). The Decorator's interface conforms to the Components's interface. The Decorator will forward requests to the decorated component. Components can be decorated with multiple decorators. This allows independent addition of horizontal and vertical

scrollbars to windows. It is transparent to clients working with Components if the Component is decorated or not.

Example In this example we implement the decoration of a simple window with horizontal an vertical scrollbars.

`Window` is the common interface for *Components*. Windows and decorators have to implement `Draw` and `GetDescription`.

```
type Window interface {
  Draw()
  GetDescription() string
}
```

`SimpleWindow` is a *Concrete Component*, an implementation of the Window interface. `SimpleWindow` is oblivious that it is going to be decorated with scrollbars.

```
type SimpleWindow struct {}

func (window *SimpleWindow) Draw() {
  //draw a simple window
}

func (window *SimpleWindow) GetDescription() string {
  return "a simple window"
}
```

`WindowDecorator` is the *Decorator* and maintains a reference to a object of the interface type `Window`. All window decorators have to implement the Window interface. To provide default functionality `WindowDecorator` implements `Window` and forward calls to its window object. `WindowDecorator` is meant to be embedded by concrete window decorators.

```
type WindowDecorator struct {
  window Window
}

func NewWindowDecorator(window Window) *WindowDecorator {
  return &WindowDecorator{window}
}
```

```
func (this *WindowDecorator) Draw() {
  this.window.Draw()
}

func (this *WindowDecorator) GetDescription() string {
  return this.window.GetDescription()
}
```

`VerticalScrollBars` is a *Concrete Decorator* and embeds `WindowDecorator`. Thus, it automatically implements the `Window` interface. `VerticalScrollBars` overrides `WindowDecorator`'s methods. `Draw` draws vertical scrollbars (actually it prints a message on the console) and then draws the actual window. `GetDescription` appends some text to the decorated window's description.

```
type VerticalScrollBars struct {
  *WindowDecorator
}

func NewVerticalScrollBars(window Window) *VerticalScrollBars{
  this := new(VerticalScrollBars)
  this.WindowDecorator = NewWindowDecorator(window)
  return this
}

func (this *VerticalScrollBars) Draw() {
  this.drawVerticalScrollBar()
  this.window.Draw()
}

func (this *VerticalScrollBars) drawVerticalScrollBar() {
  fmt.Println("draw a vertical scrollbar")
}

func (this *VerticalScrollBars) GetDescription() string {
  return this.window.GetDescription() + ", including vertical scrollbars"
}
```

There is also a `HorizontalScollBars` type. The implementation is omitted for brevity.

This code creates a window with vertical and horizontal scrollbars and prints the description.

```
var window = NewHorizontalScrollBars(
  NewVerticalScrollBars(
    new(SimpleWindow)))
fmt.Println(window.GetDescription())
```

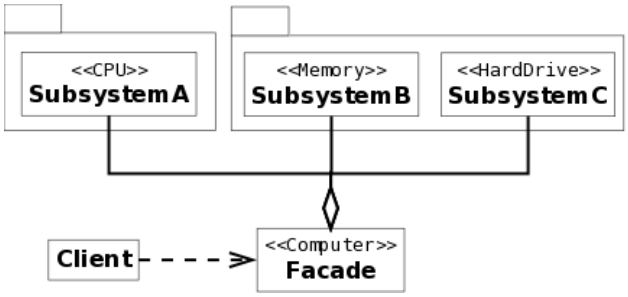
Discussion The Decorator pattern in GO is implemented with composition. The Decorator maintains a reference to the object that is to be decorated. Decorators should not embed the type to be decorated. `WindowDecorator` could embed `SimpleWindow`, but every time a new window type is added a new decorator type is needed too. This would cause an "explosion" of types, which the Decorator pattern is trying to avoid. `WindowDecorator` had to embed `Window`, but the problem is `Window` is an interface and only types can be embedded in structs, but not interfaces.

`WindowDecorator` has to implement all methods defined in `Window` and forward calls to its member window. Concrete window decorators embed `WindowDecorator` and override the necessary methods. Since there are no abstract types in GO, `WindowDecorator` can be instantiated and `Window` objects could be decorated with a `WindowDecorator`. This cannot be avoided by limiting `WindowDecorator`'s visibility to package. Decorators for `Windows` are likely to be provided outside of `WindowDecorator`'s package. Decorators embed `WindowDecorator` and need therefore to instantiate it.

A.2.5 Façade

Intent Define a high-level abstraction for subsystems and provide a simplified interface to a larger body of code to make them easier to use.

Context Consider a highly complex system of objects representing computer components like CPU, memory or hard drives. It is desirable to be able to use the computer systems having to know the sub systems.



Clients can be shielded from the low-level details by providing a higher-level interface. To hide the implementation of the *Subsystems* a *Façade* type (computer) supplies a unified interface. Most Clients interact with the Façade without having to know about its internals. The Façade pattern still allow access to lower-level functionality for the few Clients that need it.

In GO the Façade pattern can be implemented as its own type or a package could be used to represent a Façade. We show both alternatives.

Example - Façade as type In the following example we hide the CPU, memory and hard drive subsystems behind a computer façade.

In the following listing the three types CPU, Memory and HardDrive are the complex *Subsystems* we are providing a unified interface for. Clients will not have to access the subsystems, but can if they need to. The methods here are simplistic for the sake of understandability.

```
type CPU struct {}
func (this *CPU) Freeze() {
  fmt.Println("CPU frozen")
```

```
}
func (this *CPU) Jump(position int64) {
  fmt.Println("CPU jumps to", position)
}
func (this *CPU) Execute() {
  fmt.Println("CPU executes")
}

type Memory struct {}
func (this *Memory) Load(position int64, data []byte) {
  fmt.Println("Memory loads")
}

type HardDrive struct {}
func (this *HardDrive) Read(lba int64, size int64) []byte {
  fmt.Println("HardDrive reads")
  return nil
}
```

The type `Computer` is the *Façade* and maintains a references to the subsystems.

```
type Computer struct{
  cpu *CPU
  memory *Memory
  hardDrive *HardDrive
}

func NewComputer() *Computer {
  cpu := new(CPU)
  memory := new(Memory)
  hardDrive := new(HardDrive)
  return &Computer{cpu,memory,hardDrive}
}
```

The method `Start` knows how the subsystems work together. `Computer` shields the client from having to know about the inner workings of the subsystem.

```
func (this *Computer) Start() {
  this.cpu.Freeze()
  this.memory.Load(0, this.hardDrive.Read(0,1023))
  this.cpu.Jump(10)
  this.cpu.Execute()
}
```

Most clients will use `Start`, but the low-level functionality of the subsystems is still available to those how need it.

```
//the use of the facade
computer := NewComputer()
computer.Start()
//access to lower-level systems
subpart := new(CPU)
subpart.Execute()
```

Example - Façade as package Instead of encapsulating the façade in its own type, packages could be used. We use the same example as above.

The package `computer` imports the package `system` containing the subtypes `CPU`, `Memory` and `HardDrive` (declarations see example above). The package maintains references to the subsystems in package local static variables. The `init` method instantiates the subsystems as did `NewComputer` from above does.

```
package computer

import "system"

var cpu *system.CPU
var memory *system.Memory
var hardDrive *system.HardDrive

func init() {
  cpu = new(system.CPU)
  memory = new(system.Memory)
  hardDrive = new(system.HardDrive)
}
```

`Start` is now a function using the local variables to call the subsystems.

```
func Start() {
  cpu.Freeze()
  memory.Load(0, hardDrive.Read(0,1023))
  cpu.Jump(10)
  cpu.Execute()
}
```

Clients import the package `computer` (the package representing the façade) and call `Start` directly instead of instantiating a façade object.

```
import "computer"
...
computer.Start()
```

Discussion Using a package does not require to create an Façade object. The package and its subsystems are initialized in an init function, which is called on import. The initialization could be done in a separate method (init would call that method too), thus allowing clients to re-initialize the package.

The Façade pattern should not be implemented with embedding. Composition should be used instead, because the purpose is to hide low-level functionality and to provide a simpler interface. A Façade that embeds subtypes publishes the entire public interface of the subsystem, counteracting the pattern's intent.

A.2.6 Flyweight

Intent Minimize memory use by sharing as much data as possible with other similar objects; it is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory.

Context Consider drawing a large number of coloured lines. A line has an intrinsic state: its color and an extrinsic state: its start and endpoints. Creating an object for each individual line would not be feasible.

The solution is to reuse line objects with the same intrinsic state and provide them with extrinsic state to redraw draw them. This will reduce the number of line object to the number of colours. One line per colour, not more. When a line should draw itself the extrinsic state, the start and end coordinates, is passed with the message call. A small number of line stores a minimum amount of data, hence flyweight.

Example The application implemented in this example draws a large number of lines. The start and end coordinates of the line are generated randomly. The color of each line is also selected randomly. There will at most one line object per color. The drawing of lines will be simulated by printing a message on the console.

The first two listings show some helper code making the following code more concise.

We import the package `fmt` for console output to simulate drawing, and the package `rand` for generation of pseudo-random numbers. We provide the alias `random` instead of the somewhat cryptic package name `rand`.

```
package line

import (
  "fmt"
  random "rand"
)
```

`colors` is local static array containing the available colors. Having a separate `Color` type improves type safety and readability. For simplicity we define the type `Color` as an extension of `string` allowing us to treat the two nearly interchangeably (as seen in the initialization of `colors`). The function `GetRandomColor` selects a random color from the `colors` array.

```
type Color string

var colors = []Color{"red", "blue", "green", "yellow"}

func RandomColor() Color {
  return colors[random.Intn(len(colors))]
}
```

The following listings are the core of the pattern implementation.

Objects of type `line` are the flyweights. A line's `color` is its intrinsic state, the state that distinguishes it from other lines. Note that `line` is defined with package scope to avoid uncontrolled instantiation of line objects.

```
type line struct {
  color Color
}
```

The method `Draw` prints out a message describing the line to simulate real drawing. The start and end coordinates are a line's extrinsic state, since it is not stored in the line objects, but provided externally. `Draw` is a public method. Even though clients outside the current package can't refer to the type line, they can access its public methods.

```
func (this *line) Draw(x1, y1, x2, y2 int) {
  fmt.Printf("drawing %v line from %v:%v to %v:%v\n", this.color, x1, y1, x2, y2)
}
```

A line can be drawn multiple times on different locations, but there will only be one Line object per color. The the next listings shows how to ensure that. Lines are not aware of each other, therefore lines should not to be instantiated directly. The current package keeps the lines in a map. A line's color acts as the key.

```
type lineMap map[color.Color]line

var lines lineMap
```

The `init` function uses `make`, a function built into GO, for allocating memory for the map of lines. `init` is executed on package initialization.

```
func init() {
  lines = make(lineMap)
}
```

`GetLine` is the heart of the pattern. `GetLine` checks the `lines` map. If there is already a line of the particular color the line will be returned, otherwise a new line with the given color is created, added to the map of lines and returned. This ensures that there will only be one line per color as long as the method `GetLine` is used.

```
func GetLine(color Color) *line {
  currentLine, isPresent := lines[color]
  if !isPresent {
    currentLine = line{color}
    fmt.Printf("new %v line\n", currentLine.color)
    lines[color] = currentLine
  }
  return &currentLine
}
```

The next listing shows how clients can create and draw a number of flyweight lines of random color at random positions.

```
import "line"
...
numberOfLines := 5
for i := 0; i < numberOfLines; i++ {
  aLine := line.GetLine(line.GetRandomColor())
  x1, y1, x2, y2 := random.Int(), random.Int(), random.Int(), random.Int()
  aLine.Draw(x1, y1, x2, y2)
}
```

Discussion In other programming languages a flyweight factory is necessary to control the instantiation of flyweight objects. In GO we can take advantage of packages as encapsulation units. To enforce sharing of

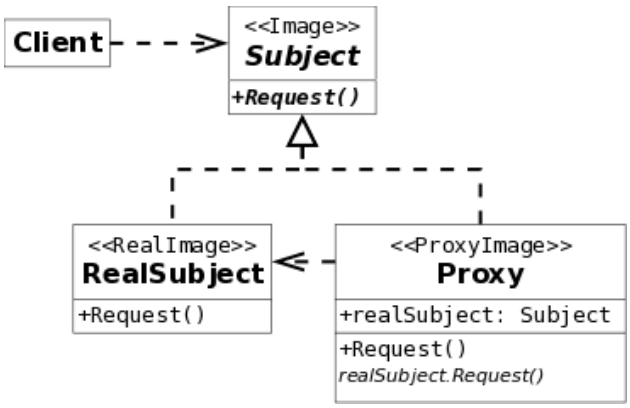
flyweights their type should be of package scope to avoid uncontrolled instantiation. The flyweight objects are kept in a package variable.

A.2.7 Proxy

Intent Represent an object that is complex or time consuming to create with a simpler one.

Context Consider a word processing application that allows images to be embedded in the documents. Those documents can be big and contain many images. Loading every image when the document is loaded takes a long time and the images might never be displayed.

The Proxy design pattern is a solution.



The *Proxy* (proxy image), or surrogate, instantiates the *Real Subject* (real image) the first time the *Client* makes a request of the proxy. The Proxy remembers the identity of the Real Subject, and forwards the instigating request to this Real Subject. Then all subsequent requests are simply forwarded directly to the encapsulated Real Subject. Proxy and Subject both implement the interface Subject, so that Clients can treat Proxy and Real Subject interchangeable.

Design Patterns lists four types of proxy. We implement a virtual proxy (lazy initialisation) in the following example.

Example Images can take a long time to load and should only be loaded when they are needed. An image proxy loads the real image only on the

first request. Subsequent calls to the proxy are forwarded to the real image. In this example we show how to provide a image proxy for real images.

`Image` is the *Subject* interface for real images and for their surrogate, the image proxy.

```
type Image interface {
  DisplayImage()
}
```

`NewRealImage` instantiates a `RealImage` object and loads the image from the disc, which can take some time for larger images.

```
type RealImage struct {
  filename string
}

func NewRealImage(filename string) *RealImage {
  image := &RealImage{filename:filename}
  image.loadImageFromDisk()
  return image
}
```

`loadImageFromDisk` would read the image data. `DisplayImage` is rendering the image on the screen. For simplicity, we simulate those actions by printing out a status message on the console.

```
func (image *RealImage) loadImageFromDisk() {
  fmt.Println("Load from disk: ", image.filename)
}

func (image *RealImage) DisplayImage() {
  fmt.Println("Displaying ", image.filename)
}
```

`ProxyImage` is the surrogare for real images. Like `RealImage`, `ProxyImage` keeps the filename of the image, and in a addition a reference to an `Image` object. On creation, the proxy only stores the filename. This is fast and lightweight. `NewProxyImage` does not load the image on instantiation, but only when the image is to be displayed.

```
type ProxyImage struct {
  filename string
  image Image
}
```

```
func NewProxyImage(filename string) *ProxyImage {
  return &ProxyImage{filename:filename}
}
```

The first time `DisplayImage` is called, a RealImage is instantiated, which loads the file from disk. On subsequent calls, the request to display the image is forwarded to the RealImage object.

```
func (this *ProxyImage) DisplayImage() {
  if this.image == nil {
    this.image = NewRealImage(this.filename)
  }
  this.image.DisplayImage()
}
```

This demonstrates how clients would be using the real image and the proxy. The instantiation of `eagerImage` will call `loadFileFormDisk` immediately, even though it might never be displayed. `lazyImage` is a `ProxyImage` and on instantiation only the image's filename will be stored.

```
  var eagerImage, lazyImage Image
  eagerImage = NewRealImage("realImage")   // loadFileFromDisk() immediately
  lazyImage  = NewProxyImage("proxyImage") // loadFileFromDisk() deferred

  // file is already loaded and display gets called directly
  eagerImage.DisplayImage()
  // load file from disk
  // and then forward display call to the real image
  lazyImage.DisplayImage()
```

The call `eagerImage.DisplayImage` renders the image immediately. Calling the same method on `lazyImage` will first load the image and then display it. Loading the image will only happen on the first call. Subsequent calls to the proxy image will be equally fast, since the call is directly forwarded to the loaded image.

Discussion Proxy can be implemented in GO with object composition as in other programming languages. In GO we have the option to use embedding. `ProxyImage` could embed `RealImage` instead. Then `ProxyImage` would gain all of `RealImage` members and `ProxyImage` could override

the necessary methods (like `DisplayImage`). The advantage of embedding is that message sends are forwarded automatically, no bookkeeping is required. The embedded type is actually an instance of the type. This instance needs to initialized to avoid null pointer errors. `DisplayImage` would initialize the embedded type if necessary. The call is then delegated to the embedded type:

```
func (this *ProxyImage) DisplayImage() {
  if this.RealImage == nil {
    this.RealImage = NewRealImage(this.filename)
  }
  this.RealImage.DisplayImage()
}
```

A.3 Behavioral Patterns

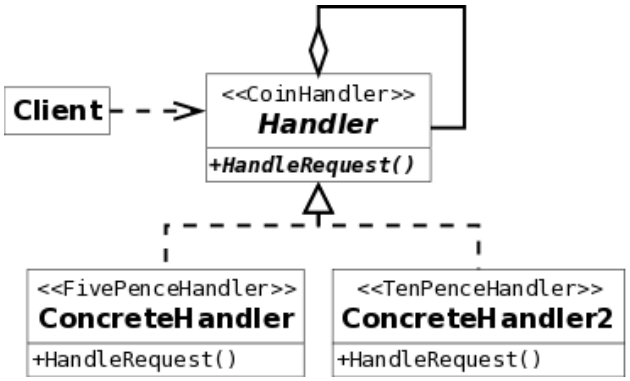
Behavioural patterns identify common communication patterns between objects and describe ways to increase flexibility in carrying out this communication.

A.3.1 Chain of Responsibility

Intent Define a linked list of handlers, each of which is able to process requests. Loose coupling is promoted by allowing a series of handlers to be created in a linked list or chain. When a request is submitted to the chain, it is passed to the first handler in the list that is able to process it.

Context Consider a software controlling a vending machine with a coin slot. The machine has to weight and measure the coin. The machine should accept different denominations of coins. Should the size or weight of coins be changed it should be easy to change the software too.

The solution is to create a chain of handlers, one for each coin denomination.



The request (a coin) is passed to the first handler in the chain (e.g. five pence handler), which will either process it or pass it on to its successor. This continues until the request is processed or the end of the chain is

reached. The handler responsible for the final processing of the request need not be known beforehand.

Example We implement the handling of different sized coins in this example.

A `Coin` has a weight and a diameter. The handler will measure the coins to decide if they should capture them.

```
type Coin struct {
  Weight float
  Diameter float
}
```

The `CoinHandler` interface defines the methods `HandleCoin` and `GetNext`. Each coin handler has to implement the `HandleCoin` method. The `GetNext` method will be common to all coin handlers. The `DefaultCoinHandler` type will implement this method as shown in the next listing.

```
type CoinHandler interface {
  HandleCoin(coin *Coin)
  GetNext() CoinHandler
}
```

`DefaultCoinHandler` encapsulates common code for coin handlers and maintains a reference to the successor of the current coin handler. The method `GetNext` returns the reference to the next coin handler. `DefaultCoinHandler` should be embedded by coin handlers to inherit the method `GetNext` and the slot for their successor. Coin handlers embedding `DefaultCoinHandler` still have to implement `Handle`, making `Handle` effectively an abstract method.

```
type DefaultCoinHandler struct {
  Successor CoinHandler
}

func (this *DefaultCoinHandler) GetNext() CoinHandler {
  return this.Successor
}
```

The `FivePenceHandler` is a concrete coin handler and embeds `DefaultCoinHanlder`, inheriting `GetNext` and the slot `Successor` for a successor coin handler. Since `*DefaultCoinHandler` is a reference to a `DefaultCoinHandler` object, the object needs to be instantiated, which is done in `NewFivePenceHandler`.

```
type FivePenceHandler struct {
  *DefaultCoinHandler
}

func NewFivePenceHandler() *FivePenceHandler {
  return &FivePenceHandler{&DefaultCoinHandler{}}
}
```

The HandleCoin method checks if the measurements of the coin match the description of a five pence coin. If the match, a message is printed. If the measurements don't match the coin is forwarded to the next coin handler (if there is one).

```
func (this *FivePenceHandler) HandleCoin(coin *Coin) {
  if coin.Weight == 3.25 && coin.Diameter == 18 {
    fmt.Println("Captured 5p")
  } else if this.Successor != nil {
    this.GetNext().HandleCoin(coin)
  }
}
```

For brevity we don't show the code of the handler handling ten pence, twenty pence and so on. They are similar to the `FivePenceHandler` and differ only in the measurements of the coins and the status message they print.

The next listing shows how clients can create chains of coin handlers. The handlers are instantiated and then the successor of each handler are set. The bottom part of the listing shows how a coins is passed to the beginning of the chain. Note that the coin `counterfeit` is not handled by any handler. It virtually fell off the end of the chain.

```
h5 := NewFivePenceHandler()
h10 := NewTenPenceHandler()
h20 := NewTwentyPenceHandler()
h5.Successor = h10
h10.Successor = h20

twentyPence := &Coin{6.5, 24.5}
counterfeit := &Coin{10, 10}

h5.HandleCoin(twentyPence) //prints: Captured 10p
h5.HandleCoin(counterfeit) //prints:
```

Discussion Each `HandleCoin` method has to check if it has a successor and will then forward the request. The `else if` branch of the `HandleCoin` methods is repeated for each coin handler. This is redundant code, which is bad style and it is easy to forget to forward a call to the next handler. One solution would be to employ the Template Method pattern (see Appendix A.3.10). Each `HandleCoin` method would return if it had handled. Forwarding of the request would be done externally to the handlers as shown in the next listing.

```
func Execute(handler CoinHandler, coin *Coin) {
  handled := handler.HandleCoin(coin)
  if handler.GetNext() != nil && !handled {
    Execute(handler.GetNext(), coin)
  }
}
```

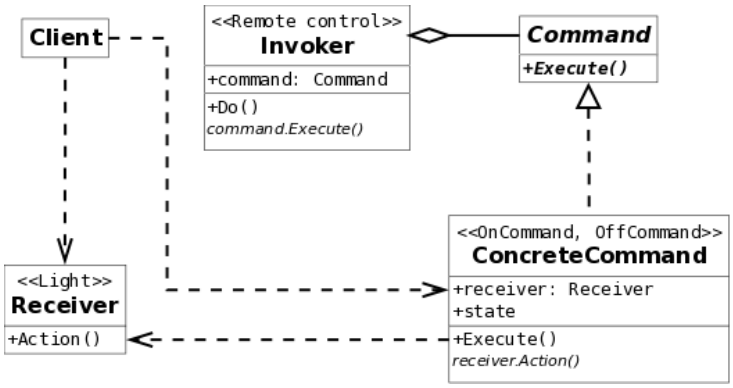
`Execute` is a function controlling the chain's execution. As long as the request has not been handled `Execute` gets called recursively with the current `CoinHandler`'s successor. Once the request (the coin) has been handled or the current handler has no successor, the recursion stops.

A.3.2 Command

Intent Represent and encapsulate all the information needed to call a method at a later time, and decouple the sender and the receiver of a method call.

Context Consider a remote control controlling lights and other equipment. The remote has to be able to work with a multitude of electronic devices. The remote has to be able to switch yet unknown devices on and off. Adding new devices should not result changes to the remote.

The solution is to turn requests (method calls) into Command objects.



The *Client* instantiates the *Command* object and provides the *Receiver* (light) of the Command. The *Invoker* (remote control) is initialized with a *Concrete Command* (on, off). The *Receiver* knows how to perform the operations associated with carrying out a request (switch light on or off).

In many programming languages the Command pattern is implemented by encapsulating Commands in their own type. GO supports first class functions and closures, which gives us a design alternative. We will be showing both implementations in the following examples.

Example - Command as types In this first example we are encapsulating the command for switching on the light in its own type.

We define `Receiver` as an interface. In our example the remote control will be able to work with objects that can be switched on and off.

```
type Receiver interface {
  SetOn(bool)
  IsOn() bool
}
```

A `Light` has a field `isOn`. `Light` objects are the *Receivers* of requests. `SetOn` and `IsOn` are accessors for the `isOn` field.

```
type Light struct {
  isOn bool
}

func (this *Light) SetOn(isOn bool) {
  this.isOn = isOn
}

func (this *Light) IsOn() bool {
  return this.isOn
}
```

The interface `Command` defines the method `Execute` that all concrete commands have to implement. `Execute` performs operations on the command's receiver.

```
type Command interface {
  Execute()
}
```

`OnCommand` is a *Concrete Command*. `OnCommand` maintains a reference to a `Receiver` object. The method `Execute` sets the `isOn` field of receiver to `true`.

```
type OnCommand struct {
  receiver Receiver
}

func NewOnCommand(receiver Receiver) *OnCommand {
  return &OnCommand{receiver}
}

func (this OnCommand) Execute() {
  this.receiver.SetOn(true)
}
```

Objects of type `RemoteControl` are the *Invoker*s of commands. `RemoteControl` maintains a reference to a `Command` object. `PressButton` call the `Execute` method of that object. The behaviour of `PressButton` depends on the dynamic type of `command`. This decouples the Invoker from the Receiver of a request. A remote control is not limited to working with lights.

```
type RemoteControl struct {
  command Command
}

func (this *RemoteControl) PressButton() {
  this.command.Execute()
}
```

The following listing ties it all together. `remote` and `light` get instantiated; `lightOnCmd` is initialized with the object `light`; The `remote`'s field `command` is set to the object `lightOnCmd`. `PressButton` calls the `Execute` method of the object `lightOnCmd`, which in turn sets the `isOn` field of the `light` object to `true`.

```
  remote := new(RemoteControl)
  light := new(Light)
  lightOnCmd := NewOnCommand(light)
  remote.command = lightOnCmd
  remote.PressButton()
  if !light.isOn {
    panic("light should be on")
  }
```

Example - Command functions Adding the off-command requires the addition of a new command type. In the following we show and alternative using GO's function pointers and closures. The `Receiver` interface and the type Light are unchanged.

We define `CommandFunc` of function type. Functions that accept a `Receiver` object can be used where a `CommandFunc` is required.

```
type CommandFunc func(Receiver)
```

We redefine the `Command` type. `Commands` still maintain a reference to the receiver of a request. The second member of `Command` `command` is of function type `CommandFunc`.

```
type Command struct {
  receiver Receiver
  command CommandFunc
}

func NewCommand(receiver Receiver, command CommandFunc) *Command {
  return &Command{receiver, command}
}
```

`Command` implements the method `Execute`. `Execute` calls the function command with `receiver` as parameter.

```
func (this *Command) Execute() {
  this.command(this.receiver)
}
```

New commands can implemented as functions:

```
func OffCommand(receiver Receiver){
  receiver.SetOn(false)
}
//somwhere else:
lightOffCmd := NewCommand(light, OffCommand)
```

or with Closures:

```
lightOnCmd := NewCommand(light, func(receiver Receiver){
  receiver.SetOn(true)
})
```

Discussion The first example follows the description as it is given in *Design Patterns*. It is apparent that adding a new command type can be quite elaborate. The second example shows, that it is easier to add new commands with function pointers and Closures than having to define a new command type.

Having a type with the command method has some advantages over having functions only. Command objects can store state. Functions in GO can store state only in static variables defined outside the function's scope.

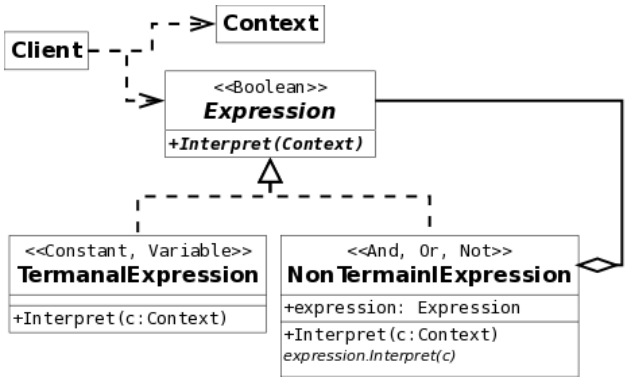
Variables defined inside the function's scope cannot be static; they are initialized every time the function is called. The problem with static variables is that they can be altered externally too. The function cannot rely on the variable's state. Having a Command type allows to use helper methods and inheritance for reuse. This possibility is not given to functions. Subtypes could override the helper functions (Template Method) or override the algorithm but reuse the helper functionality (Strategy).

A.3.3 Interpreter

Intent Interpreter specifies how to evaluate a sentence in a language. The pattern allows the grammar to be represented in an easily extended fashion.

Context Consider an application for evaluating complex boolean expressions. Such expressions can be AND, OR or NOT and others.

The Interpreter pattern defines a language for expressions and shows how to evaluate structures of expressions. A language consists of a vocabulary (the words or expressions) and a grammar (the rules how expressions can be combined). The following diagram shows the structure of the Interpreter pattern:



Expression defines the common interface for the vocabulary. There are *Terminal Expressions* (constants, variables) and *Non-terminal Expressions* (And, Or, Not). Non-terminal Expressions contain other Expressions. Expression can be combined according to the Composite pattern (see Appendix A.2.3) to create nested expressions or expression trees, with Terminal Expressions being leafs and Non-terminal Expressions being nodes. The *Interpret* method of Non-terminal Expressions calls recursively its children's Interpret methods. The recursion stops at Terminal Expressions. The result of the interpretation of an Expression depends on the *Context* passed to Interpret of the Expression tree.

Example In this example we implement a simple boolean expression interpreter with AND-, OR- NOT expressions (*Non-Terminal Expressions*), and variables and constants (*Terminal Expressions*).

A `Context` stores two `Variables`. Each having a boolean value. The result of the evaluation of the expression tree depends on the value of a context object's variables.

```
type Context struct {
  x, y *Variable
  xValue, yValue bool
}
```

The `Assign` methods are convenience methods for initialization of a context object.

```
func (this *Context) AssignX(variable *Variable, xValue bool) {
  this.x = variable
  this.xValue = xValue
}

func (this *Context) AssignY(variable *Variable, yValue bool) {
  this.y = variable
  this.yValue = yValue
}
```

`GetValue` accepts a variable and returns the value of the according variable.

```
func (this *Context) GetValue(variable *Variable) bool {
  if variable == this.x {
    return this.xValue
  }
  if variable == this.y {
    return this.yValue
  }
  return false;
}
```

The following listings show the source code of expressions.

`BooleanExpressions` are used to build the expression tree. Expressions are composite. An expression can consist of other expressions. Every expression has to implement the method `Evaluate` returning a boolean. The parameter of type `*Context` contains the names and values of the

variables the expression is to be evaluated with.

```
type BooleanExpression interface {
  Evaluate(*Context) bool
}
```

A `NotExpression` contains a field operand of type `BooleanExpression`. `NewNotExpression` is the constructor method for `NotExpressions` setting the operand. The method `Evaluate` returns the negated value of its operand's `Evaluate` method. `Evaluate` is called recursively passing along the `context` parameter. The expression tree is traversed and the evaluation starts at the leaves.

```
type NotExpression struct {
  operand BooleanExpression
}

func NewNotExpression(operand BooleanExpression) *NotExpression {
  return &NotExpression{operand}
}

func (this *NotExpression) Evaluate(context *Context) bool {
  return !this.operand.Evaluate(context)
}
```

The constructor methods of the other expressions are similar. We omit them for brevity.

An `AndExpression` has two operands of type `BooleanExpression`. `Evaluate` calls its operands `Evaluate` method and concatenates their values with the AND-operator `&&`. The type `OrExpression` is similar, differing only in its `Evaluate` operator OR `||`. We omit the code for `OrExpression`.

```
type AndExpression struct {
  operand1 BooleanExpression
  operand2 BooleanExpression
}

func (this *AndExpression) Evaluate(context *Context) bool {
  return this.operand1.Evaluate(context) && this.operand2.Evaluate(context)
}
```

The type `Variable` is a `BooleanExpression` with a name. The parameter `context` of `Evaluate` stores two variables and their according values. `Evaluate` returns the value for the current `Variable` object `this`.

```
type Variable struct {
  name string
}

func (this *Variable) Evaluate(context *Context) bool {
  return context.GetValue(this);
}
```

A `ConstantExpression` stores a boolean value. `Evaluate` returns that value.

```
type Constant struct {
  operand bool
}

func (this *Constant) Evaluate(context *Context) bool {
  return this.operand
}
```

The types `Constant` and `Variable` are leafs in the expression tree, `And`-, `Or`-, and `NotExpression` are composite expressions, containing other expressions.

The following listing show the construction of an expression tree and its evaluation.

```
1  func main() {
2    var expression BooleanExpression
3    x := NewVariable("X")
4    y := NewVariable("Y")
5    expression = NewOrExpression(
6      NewAndExpression(NewConstant(true), x),
7      NewAndExpression(y, NewNotExpression(x)))
8
9    context := new(Context)
10   context.AssignX(x, true)
11   context.AssignY(y, false)
12
13   result := expression.Evaluate(context)
14   fmt.Println("result:", result)
15 }
```

The `expression` object assembled in lines 5-7 represents: `(true AND x) OR (y AND (NOT x))`. The `context` object is initialized with the `Variable` objects `x` and `y`. `x`'s value is true and `y`'s value is `false`. Evaluating the `expression` object with the `context` object results in `true` (`(true AND true) OR (false AND (NOT true))`).

Discussion This implementation of the Interpreter pattern is like in most other object-oriented languages. Clients build an abstract syntax tree (a sentence in the grammar to be interpreted). A Context object is initialized and passed alongside the sentence to the Interpret operation. The Interpret operation stores and accesses the Interpreters state in the Context object at each node.

As a design alternative, the Interpret operation (in this example `Evaluate`) could be implemented as a function. No separate type necessary to hold that operation as a method.

A.3.4 Iterator

Intent Access the elements of an aggregate object sequentially without exposing its underlying representation.

Context Aggregate object should give a way to access its elements without exposing its internal structure.

The Iterator pattern is a solution for that problem. There are multiple ways to implement Iterator. We present GO's built-in iterators for certain types, we show how to implement an internal iterator and we demonstrate how external iterators can be implemented.

Example - Built-in iterators GO provides built-in iteration functionality for maps, arrays/slices, strings and channels. `for` loops with a `range` clause iterate through all entries of an map, array/slice or string, or values received on a channel. We show an example for each data structure.

Maps `days` is a map with keys being the days of the weeks as strings and values being the the index of the days of the week as integers. The for look iterates over the days map. On each loop `key` and `value` are assigned the values range returns. range for maps returns two values: the map's key and the according value.

```
days := map[string]int{ "mon": 0, "tue": 1, "wed": 2, "thu": 3,
                        "fri": 4, "sat": 5, "sun": 6}

for key, value := range days {
  fmt.Printf("%v : %d,", key, value)
}
//Prints: fri : 4,sun : 6,sat : 5,mon : 0,wed : 2,tue : 1,thu : 3,
```

Arrays/Slices `words` is an array of `string`s. The for loop iterates over the elements. `range` returns the `index` of the current element and the `element` itself.

```
words := []string{"hello", "world"}

for index, character := range words {
  fmt.Printf("%d : %v,", index, character)
}
//Prints: 0 : hello,1 : world,
```

Strings `helloWorld` is a string with 11 characters. The for loop iterates over each character and prints it on the console. The range clause returns the `index` of the current character and the `character` itself.

```
helloWorld := "hello world"

for index, char := range helloWorld {
  fmt.Printf("%d : %c,", index, char)
}
//Prints: 0 : h,1 : e,2 : l,3 : l,4 : o,5 : ,6 : w,7 : o,8 : r,9 : l,10 : d
```

Channels `strings` is a channel of `string`s with a buffer size of `2`. Two strings are sent on the channel and the channel is closed for input. The for loop prints the received values on the channel strings. range on a channel returns the values received on the channel in first-in first-out order.

```
strings := make(chan string, 2)
strings <- "hello"
strings <- "world"
close(strings)

for aString := range strings {
  fmt.Printf("%v, ", aString)
}
//Prints: hello, world,
```

Example - Internal Iterator The task of an internal, or passive, iterator is to execute a function on each element of a collection. Clients pass an operation to the internal iterator and the iterator performs that operation on every element of the collection. In this example we demonstrate the internal iterator of the type Vector.

`list` is of type Vector and two string are added to it. A Closure printing the current element is passed to `list`'s `Do` method.

```
list := new(vector.Vector)
list.Push("hello")
list.Push("world")

list.Do(func(e interface{}){
  fmt.Println(e)
})
```

The code of Vector's method `Do`[2] is shown below:

```
func (p *Vector) Do(f func(elem interface{})) {
  for _, e := range *p {
    f(e)
  }
}
```

`Do` accepts a function with one parameter of the empty interface type `interface`. `Do` iterates over the aggregate and calls the passed in function with the current element of the aggregate. `range` can be used here because type Vector is an extension of array type (`type Vector []interface`).

[2]Taken from `Vector`'s package documentation [6]

Example - External Iterator Here we show an implementation of an external iterator for slices. This is a demonstrative example, as seen above slices can be traversed with the for-range clause.

The `Iterator` interface defines `Next` and `HasNext` controlling the iterators position in the aggregate.

```
type Iterator interface {
  Next() interface{}
  HasNext() bool
}
```

`SliceIterator` maintains a reference to the slice `slice`. `index` stores the current location of the iterator. On instantiation of `SliceIterator` the slice `slice` has to be set and `index` will be initialized to 0.

```
type SliceIterator struct{
  slice []interface{}
  index int
}
func NewSliceIterator(slice []interface{}) Iterator {
  this := new(SliceIterator)
  this.slice = slice
  return this
}
```

The method `HasNext` compares the current position with the length of the slice. `HasNext` returns `false` if there are no more elements in the slice, `true` otherwise.

```
func (this *SliceIterator) HasNext() bool {
  return this.index < len(this.slice)
}
```

`Next` returns the current element of the slice and increments `index`. If the current index points to an invalid element in the slice it panics.

```
func (this *SliceIterator) Next() interface{} {
  if this.index >= len(this.slice) {
    panic("NoSuchElement")
  }
  element := this.slice[this.index]
  this.index++
  return element
}
```

The next listing shows how a `SliceIterator` can be used. A slice `aSlice` containing two strings is created (behind the scenes: an array of length two gets created, the values assigned and then the array gets sliced). A `SliceIterator` object `iter` gets instantiated with the just created slice. `iter` handles the internal iteration of the slice and knows its the current position in the slice. The for-loop runs as long as there is another element in the slice. Inside the loop the iterator advances and prints the next element in the slice.

```
aSlice := []interface{}{"hello", "world"}
iter := NewSliceIterator(aSlice)
for iter.HasNext() {
  current := iter.Next()
  fmt.Println(current)
}
```

Example - Iteration with channels In an earlier version[3], the type Vector provided an external Iterator by sending its elements on a channel. The channel containing the elements can then be used to iterate of the vector's elements.

The method `Iter` creates and returns a channel containing the vectors elements. The sending of elements to the channel happens in the private method `iterate`, which runs in a separate goroutine. The returned channel `c` is an output channel (the operator <− is on the left).

```
func (p *Vector) Iter() <-chan interface{} {
    c := make(chan interface{})
    go p.iterate(c)
    return c
}
```

The method `iteratate` iterates over the elements of the vector and sends them on the channel `c`. `c` is defined as an input channel (the operator <− is on the right).

```
func (p *Vector) iterate(c chan<- interface{}) {
    for _, v := range *p {
        c <- v
    }
    close(c)
}
```

Clients call `Iter` on vector objects and receive a channel containing the vector's elements. This channel can be used to iterate over the vector's elements:

```
for element := range aVector.Iter() {
  DoSomething(element)
}
```

Discussion GO provides a mechanism to iterate over some of the built-in types. External iterators are still necessary and their implementation is similar to *Design Patterns*.

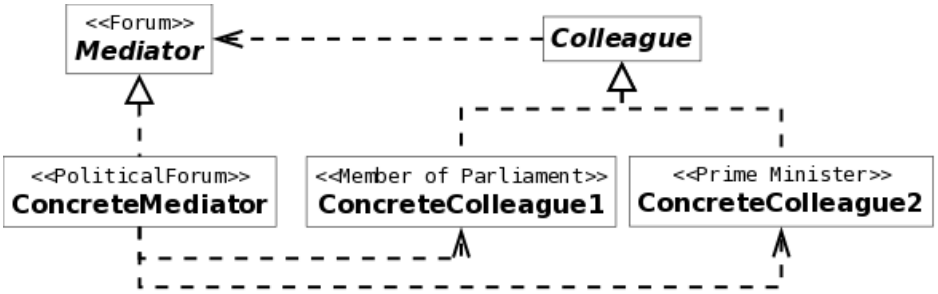
[3]The method `Iter` has been removed from Vector with the release from the 14/07/2010. [3]

A.3.5 Mediator

Intent Provide a central hub to guide the interactions between many objects. Reduce many-to-many to one-to-many relationships.

Context Consider implementing the communication between the members of parliament. This communication would create dependencies if every member maintained a reference to the other members.

The Mediator pattern is a solution for this problem, encapsulating the collective behaviour in a separate mediator object. The following diagram depicts the structure of the mediator pattern.



The *Mediator* interface (forum) defines an interface for communication with *Colleague* objects. *Concrete Mediators* (political forum) implement the behaviour defined in the Mediator interface. Concrete Mediators maintain references to their colleagues. Each *Concrete Colleague* (member of parliament, prime minister) knows its Mediator object. Concrete Colleagues communicate through their Mediator with their Colleagues.

Example In this example we show how to implement a political forum, where members of parliament can send messages to a single recipient or to every member of the forum. The members of the forum don't have to know all other members. A forum distributes messages between participants — we call them colleagues to go with the pattern.

`Forum` lists a small set of methods concrete forums have to implement. `Forum` is the *Mediator* interface. `Send` is for broadcasts, accepting the name of the sender and the message to be send to all colleagues. `SendFromTo` is for directed message sends, from one colleague to the other. The `Add` message is used for colleagues to join the forum.

```
type Forum interface {
  Send(from, message string)
    SendFromTo(from, to, message string)
    Add(c Colleague)
}
```

`Forum` does not define how concrete forums have to maintain their members. `PoliticalForum` keeps a `map`, with the name of the colleague as keys, and `Colleague` objects as values.

```
type PoliticalForum struct {
  colleagues map[string]Colleague
}
```

`NewPoliticalForum` instantiates a `PoliticalForum` and initializes the map of colleagues. The built-in function make reserves memory for the map and returns a pointer to it.

```
func NewPoliticalForum() *PoliticalForum {
  return &PoliticalForum{make(map[string]Colleague)}
}
```

The `Add` method adds the passed in `Colleague` object to the map and sets the colleague's forum. `Forum` and `Colleague` have a two-way relationship. Forums know their members and colleagues know in which forum they are.

```
func (this *PoliticalForum) Add(colleague Colleague) {
  this.colleagues[colleague.GetName()] = colleague
  colleague.SetForum(this);
}
```

`Send` does a broadcast, forwarding the message to all colleagues of the political forum, except to the sender.

```
func (this *PoliticalForum) Send(sender, message string) {
  for name, colleague := range this.colleagues {
    if name != sender {
      colleague.Receive(sender, message)
    }
  }
}
```

`SendFromTo` does directed message sends. The method gets the receiver from the map of colleagues, checks if the receiver exists and sends the message to the colleague. `Send` and `SendFromTo` are the mediating methods. They decouple the colleagues from each other. Colleagues don not have to keep track of other colleagues.

```
func (this *PoliticalForum) SendFromTo(sender, receiver, message string) {
  colleague := this.colleagues[receiver]
  if colleague != nil {
    colleague.Receive(sender, message)
  } else {
    fmt.Println(sender, "is not a member of this political forum!")
  }
}
```

`Colleague` defines the interface members of forums have to implement. `Colleague`s have a name for identification, can be member of a forum, can send messages to a single recipient or to all members of their forum, and they can receive messages from other members of a forum.

```
type Colleague interface {
  GetName() string
  SetForum(forum Forum)
  Send(to, message string)
  SendAll(message string)
  Receive(from, message string)
}
```

`MemberOfParliament` (MOP) are *Concrete Colleagues*. MOPs have a name and a forum.

```
type MemberOfParliament struct {
  name string
    forum Forum
}
```

`NewMemberOfParliament` instantiates a new MOP and sets its name. `GetName` and `SetForum` are public accessors to the MOP's name and forum.

```
func NewMemberOfParliament(name string) *MemberOfParliament {
        return &MemberOfParliament{name:name}
}
func (this *MemberOfParliament) GetName() string {
  return this.name
}
func (this *MemberOfParliament) SetForum(forum Forum) {
  this.forum = forum
}
```

`Send` and `SendAll` forward the messages to their forum, which then determines the the right recipients.

```
func (this *MemberOfParliament) Send(receiver, message string) {
  this.forum.SendFromTo(this.name, receiver, message);
}
func (this *MemberOfParliament) SendAll(message string) {
  this.forum.Send(this.name, message);
}
```

`Receive` prints the message on the console including the sender and the receiver.

```
func (this *MemberOfParliament) Receive(sender, message string) {
  fmt.Printf("The Hon. MP %v received a message from %v: %v\n",
    this.name, sender, message)
}
```

`PrimeMinister` (PM) is a specialization of `MOP`. By embedding `MOP`, `PM` inherits all of `MOP`'s functionality.

```
type PrimeMinister struct {
  *MemberOfParliament
}
func NewPrimeMinister(name string) *PrimeMinister {
  return &PrimeMinister{NewMemberOfParliament(name)}
}
```

`PrimeMinister` defines its own `Receive` method, overriding MOPs, to print a more suitable message.

```
func (pm *PrimeMinister) Receive(sender, message string) {
  fmt.Printf("Prime Minister %v received a message from %v: %v\n",
```

```
    pm.name, sender, message)
}
```

In the following listing, a forum and three colleagues get instantiated. The colleagues are added to the forum, by which the forum sets itself as the colleagues forum.

```
forum := NewPoliticalForum()
pm := NewPrimeMinister("Prime")
mp1 := NewMemberOfParliament("MP1")
mp2 := NewMemberOfParliament("MP2")
forum.Add(pm)
forum.Add(mp1)
forum.Add(mp2)
```

Colleagues sending messages are send to their forums, which distribute them to the receivers. The commented lines show the console output of each call.

```
pm.SendAll("Hello everyone")
//The Hon. MP   M P 2   received a message from   P r i m e   : Hello everyone
//The Hon. MP   M P 1   received a message from   P r i m e   : Hello everyone
mp1.Send("MP2", "Hello back")
//The Hon. MP   M P 2   received a message from   M P 1   : Hello back
mp2.Send("Prime", "Dito")
//Prime Minister   P r i m e   received a message from   M P 2   : Dito
pm.Send("MP1", "Bullsh!t")
//The Hon. MP   M P 1   received a message from   P r i m e   : Bullsh!t
mp1.SendAll("I second")
//Prime Minister   P r i m e   received a message from   M P 1   : I second
//The Hon. MP   M P 2   received a message from   M P 1   : I second
```

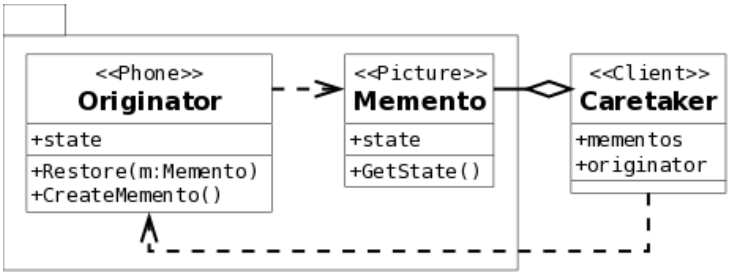
Discussion In implementing the Mediator pattern, the interface Forum is optional. The political forum would still be the mediator. Other forum types could be implemented against that interface in connection with the colleague interface. The concrete colleagues we implemented here could then be used to interact through any type of forum.

Colleagues are only able to be member of one forum at a time. This restriction could be lifted by colleagues being able to maintain a collection of forums.

A.3.6 Memento

Intent Brings an object back to a previous state by externalizing the object's internal state.

Context Consider you decide to take a phone apart in order to replace an internal part. The problem is to get it back to its original state. A solution is taking pictures while dismantling the phone. Each picture is a memento, a reminder or reference of how an object should look.



The above shows the structure of the Memento pattern. The *Originator* (the phone) has internal state (the phone's parts). CreateMemento instantiates a *Memento* (a picture) and the Memento stores the current state of the Originator. The *Caretaker* (the client taking a phone apart) asks the Originator for a Memento. The Caretaker uses the Mementos to restore the Originator. Originator and Memento should be in the same package, separate from the Caretaker, to prohibit Caretakers changing the state of Mementos.

Example In this example we remove parts from a phone and show how the Memento pattern is used to restore the phone to a certain state in the dismantling process.

`Phone` objects are the *Originators*. `Phone` stores a `map` of `Parts`[4]. The map of parts represents the internal state.

```
type Part string
type Phone struct {
  parts map[int]Part
}
```

`RemovePart` removes parts from the phone's `parts` map. To remove entries from maps multiple assignment is used in GO. If the additional boolean on the right is false, the entry is deleted. No error occurs if the key does not exist in the map.

```
func (this *Phone) RemovePart(part int) {
  this.parts[part] = "", false
}
```

`TakePicture` creates a `Picture` object with the phone's current internal state. `Picture` is the `Memento`, storing the phone's current state. Clients call this method to create a snapshot of the phone in its current state. `RestoreFromPicture` replaces the phone's parts with those of the picture when it was taken.

```
func (this *Phone) TakePicture() *Picture {
  return NewPicture(this.parts)
}
func (this *Phone) RestoreFromPicture(picture *Picture) {
  this.parts = picture.getSavedParts()
}
```

Pictures are *Mementos* for phones. `Picture` maintains, like `Phone`, a map of Part objects to store the phone's state.

```
type Picture struct {
  parts map[int]Part
}
```

`NewPicture` instantiates a picture and copies the phone's parts. The creation of a new map and copying of the elements is necessary. Consider assigning `partsToSafe` to `this.parts`. Both variables would point to the same map object. Changes to phones parts would result in a change in picture's parts, which is not intended.

[4]For simplicity `Part` is an extension of `string`

```
func NewPicture(partsToSave map[int]Part) *Picture {
  this := new(Picture)
  this.parts = make(map[int]Part)
  for index, part := range partsToSave {
    this.parts[index] = part
  }
  return this
}
```

The method `getSavedState` is package local, e.g. not visible outside the current package (identifier starts with lower case character). The method returns the parts of the phone when the Picture object `this` was created. Caretakers residing outside Picture's package cannot access the Pictures's state.

```
func (this *Picture) getSavedParts() map[int]Part {
  return this.parts
}
```

The next listings show how Pictures can be used to restore a Phone. `NewPhone` creates a Phone object and adds four parts.

```
func NewPhone() *Phone {
  this := new(Phone)
  this.parts = make(map[int]Part)
  this.parts[0] = Part("body")
  this.parts[1] = Part("display")
  this.parts[2] = Part("keyboard")
  this.parts[3] = Part("mainboard")
  return this
}
```

The following listing demonstrates the *Caretaker*. `pictures` is a map of `Picture` objects storing the Mementos. An initial memento of the phone's original state is created, and two additional pictures are taken during the disassembly of the phone object (`RemovePart`). `TakePicture` creates a `Picture` object storing the phone's current parts. The commented lines describe the state of the phone after each removal. The last three lines show how the phone can be restored to a given state by the caretaker. The caretaker selects the necessary state and passes it to the phone's `RestoreFromPicture` method.

```
phone := NewPhone()
pictures := make([]*Picture, 3)
pictures[0] = phone.TakePicture()
phone.RemovePart(0)
phone.RemovePart(2)
//mainboard and display remaining
pictures[1] = phone.TakePicture()
phone.RemovePart(1)
pictures[2] = phone.TakePicture()
//mainboard remaining
phone.RemovePart(3)
//phone dismantled
phone.RestoreFromPicture(pictures[2])
phone.RestoreFromPicture(pictures[1])
phone.RestoreFromPicture(pictures[0])
```

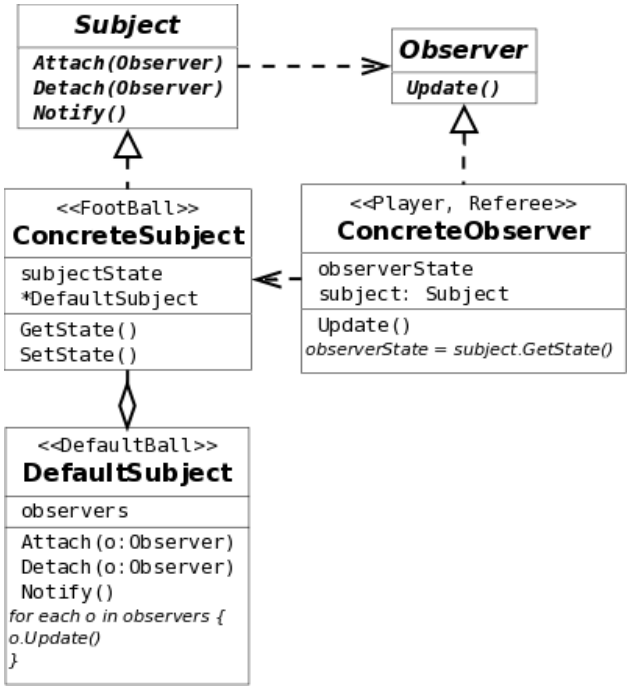
Discussion In the Memento pattern, Memento objects have two interfaces: a narrow interface for the Caretaker and a wide one for Originators. In C++ Originator is a friend of Memento gaining access to the narrow interface. The methods in the wide interface are to be declared public. Java has no functionality that could be used. In GO we define the wide interface with public methods and the narrow one with package local methods. Memento and Originator should reside in same package and Caretakers should be external, to protect Mementos from being accessed.

A.3.7 Observer

Intent A one-to-many relationship between interesting and interested objects is defined. When an event occurs the interested objects receive a message about the event.

Context Consider a football game. Players and referees know the last position of the ball. They should not constantly poll the ball for its current position. Whenever the ball moves all interested participants should be notified.

A solution is the Observer pattern. The image below shows the structure of the observer pattern.



Observers (players, referees) register with *Subjects* (balls). The *Concrete Subjects* informs their Observers about changes. *Concrete Observers* get the changes from the Subject that notified them. All Subjects have the Attach, Detach and Notify operations in common. The functionality is

encapsulated in a *Default Subject* type. Every type that embeds Default Subject is automatically a Concrete Subject.

Example We show how a ball can observed by players. The players get notified about the change of position and poll the ball for the new position.

The `Subject` interface lists the methods `Attach`, `Detach` and `Notify`.

```
type Subject interface {
  Attach (observer Observer)
  Detach (observer Observer)
  Notify()
}
```

`DefaultSubject` implements the methods defined in the Subject interface. `DefaultSubject` maintains a collection of observers. `DefaultSubject` is not meant to be instantiated directly, but to be embedded. Types embedding `DefaultSubject` will gain the necessary functionality to be observable. In this example the type `FootBall` (shown further down) embeds `DefaultSubject`.

```
type DefaultSubject struct {
  observers *vector.Vector
}

func NewDefaultSubject() *DefaultSubject {
  return &DefaultSubject{observers:new(vector.Vector)}
}
```

`Attach` and `Detach` handle adding and removing of observers.

```
func (this *DefaultSubject) Attach(observer Observer) {
  this.observers.Push(observer)
}

func (this *DefaultSubject) Detach(observer Observer) {
  for i := 0; i < this.observers.Len(); i++ {
    currentObserver := this.observers.At(i).(Observer)
    if currentObserver == observer {
      this.observers.Delete(i)
    }
  }
}
```

`Notify` informs all observers in the list. The method loops over all observers and calls their Update method, which triggers the observers to poll their subject's state.

```
func (this *DefaultSubject) Notify() {
  for i := 0; i < this.observers.Len(); i++ {
    observer := this.observers.At(i).(Observer)
    observer.Update()
  }
}
```

`Position` objects represent a point in the 3-dimensional space with `x`, `y` and `z` coordinates.

```
type Position struct {type Position struct {
  x, y, z int
}

func NewPosition(x, y, z int) *Position {
  return &Position{x, y, z}
}
```

`FootBall` is a *Concrete Subject*. It inherits the necessary functionality from embedding `DefaultSubject`. A `FootBall` knows its current position.

```
type FootBall struct {
  *DefaultSubject
  position *Position
}
func NewFootBall() *FootBall {
  return &FootBall{DefaultSubject:NewDefaultSubject()}
}
```

`GetPosition` provides public access the `FootBall`'s current position. `SetPosition` updates the the `FootBall`s position and informs all its observers about the change, which in turn query the football for its new position.

```
func (this *FootBall) GetPosition() *Position {
  return this.position
}
```

```
func (this *FootBall) SetPosition(position *Position) {
  this.position = position
  this.Notify()
}
```

The `Observer` interface defines one method: `Update`. The `Update` method is called by subject's `Notify` method.

```
type Observer interface {
  Update()
}
```

`Player` is a *Concrete Observer*. `Player`s have a `name`, maintain a reference to a `FootBall` and remember the the ball's last position.

```
type Player struct {
  name         string
  lastPosition *Position
  ball         *FootBall
}

func NewPlayer(name string, ball *FootBall) *Player {
  this := new(Player)
  this.name = name
  this.ball = ball
  return this
}
```

`Update` updates the player's last known position of the ball and prints a message on the console.

```
func (this *Player) Update() {
  this.lastPosition = this.ball.GetPosition()
  fmt.Println(this.name, "noticed that ball has moved to", this.lastPosition)
}
```

In the following listing a ball and three players are created. The players get names and a reference to the ball. The players are attached to the ball to be notified about the ball's position changes. The first time the ball changes position all three players get notified and print a message; the second time only player1 and player3, since player2 is no longer an observer of the ball.

```
var aBall = NewFootBall()
var player1 = NewPlayer("player1", aBall)
var player2 = NewPlayer("player2", aBall)
var player3 = NewPlayer("player3", aBall)
aBall.Attach(player1)
aBall.Attach(player2)
aBall.Attach(player3)
aPosition := NewPosition(1, 2, 3)
aBall.SetPosition(aPosition)
aBall.Detach(player2)
aBall.SetPosition(NewPosition(2, 3, 4))
```

Discussion Java's abstract classes combine the definition of interface and default implementation. GO does not have abstract classes. Embedding can be used, even though it is not as comfortable. One advantage of embedding becomes apparent in this implementation of the Observer pattern. To make a type observable, only DefaultSubject needs to be embedded. Java supports single class inheritance, but in GO multiple types can be embedded. Types that are doing their responsibility can easily be made observable. And the embedding type can still override the embedded type's members to provide different behaviour.

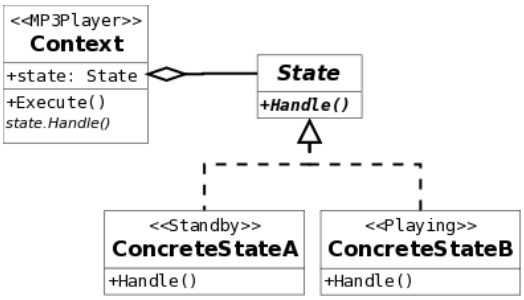
The Observer pattern is widely used in Java libraries (Listeners in Swing for example), but not in GO, as far as we can see.

A.3.8 State

Intent Change an object's behaviour by switching dynamically to a set of different operations.

Context Consider an MP3 player. Depending on what the player is doing (being in standby mode or playing music) a click on the play button should change the player's behaviour.

The State pattern is a solution for this problem of dynamic reclassification.



A *Context* (MP3 player) has a *State*. Asking the Context to execute its functionality (clicking the play button), will execute the Context's current State's behaviour. The behaviour depends on the dynamic type of the State object. *Concrete States* (playing, standby) handle the State transitions (if play is clicked while the player is in standby, the player wakes up and is set into the play state).

GO offers two alternatives for implementing the State pattern. One way is to encapsulate states in state objects of different type, the other is to use functions. We show source code for both alternatives by implementing the above outlined MP3 player example.

Example - States as types In this example, the states, that an MP3 player can be in, are implemented as separate types. The MP3 player maintains a reference to a state object and executes the states behaviour.

All `States` have to implement the `Handle` method. The state objects are `Flyweight` objects (see Appendix A.2.6), they can be shared between contexts. State objects do not store their Context. The context the states operate on is passed in on every call of `Handle`. The concrete states we implement in this example could be implemented as Singleton objects (see `appx:singleton`). We omit this in order to focus on the State pattern itself.

```
type State interface {
  Handle(player *MP3Player)
}
```

`Standby` is a *Concrete State*. Upon calling of `Handle`, a message is printed (symbolizing the actions to wake up the MP3 player) and the player's state is changed to `Playing`. Note that a new `Playing` object is created each time `Handle` is called. The concrete states should be Singleton as explained above.

```
type Standby struct{}

func (this *Standby) Handle(player *MP3Player) {
  fmt.Println("waking up player")
  player.SetState(new(Playing))
}
```

`Playing` is another *Concrete State*. `Handle` prints out messages on the console describing what the player does. After playing music, the player is set into `Standby`.

```
type Playing struct{}

func (this *Playing) Handle(player *MP3Player) {
  fmt.Println("start playing music")
  fmt.Println("music over")
  fmt.Println("waited ten minutes")
  fmt.Println("send player to sleep")
  player.SetState(new(Standby))
}
```

The `MP3Player` type is the *Context*. The player maintains a reference to a State object. `NewMP3Player` instantiates a new `MP3Player` object and sets its initial state. `SetState` provides write access to the player's state.

`SetState` is public to enable state transition for package external State types (not shown).

```
type MP3Player struct {
  state State
}

func NewMP3Player(state State) *MP3Player {
  this := new(MP3Player)
  this.state = state
   return this
}

func (this *MP3Player) SetState(state State) {
  this.state = state
}
```

`PressPlay` calls the current state's Handle method. The behaviour of `PressPlay` depends on the dynamic type player's current `state`.

```
func (this *MP3Player) PressPlay() {
  this.state.Handle(this)
}
```

A `MP3Player` is initialized with state `Playing`. The first call of `PressPlay` will perform the behaviour defined in `Playing.Handle` and will set `player` into `Standby`. The second call of `PressPlay` will wake up the player (Standby's behaviour).

```
player := NewMP3Player(new(Playing))
player.PressPlay()
player.PressPlay())
```

Example - States as functions GO provides first class functions. Instead of defining separate types for each state, states can defined in functions. The MP3 player maintains a reference to a function instead of to an object.

Instead of being an interface with a single method, `State`s are of function type.

```
type State func(player *MP3Player)
```

`Standby` is a function conforming to the function type `State`. The functionality performed by `Standby` is as before. Note that the parameter

`Playing` passed to `SetState` is a function.

```
func Standby(player *MP3Player) {
  fmt.Println("waking up player")
  player.SetState(Playing)
}
```

Here `Playing` is also a function of type `State`. As with `Standby`, `Playing` has the same behaviour as before.

```
func Playing(player *MP3Player) {
  fmt.Println("start playing music")
  fmt.Println("music over")
  fmt.Println("waited ten minutes")
  fmt.Println("send player to sleep")
  player.SetState(Standby)
}
```

`MP3Player` is syntactically like before. The difference lies in the type of its member `state`, now it is function type, it was interface type. `NewMP3Player` and `SetState` are syntactically like before, but their parameters `state` are now of function type instead of interface type.

```
type MP3Player struct {
  state State
}
```

`PressPlay` executes the `states` behaviour directly, by passing the `MP3Player` object to `this.state`, which is a function.

```
func (this *MP3Player) PressPlay() {
  this.state(this)
}
```

Below, a new `MP3Player` object is created and initialized with `Playing`. `PressPlay` calls the function stored in the player's `state`.

```
player := NewMP3Player(Playing)
player.PressPlay()
player.PressPlay()
```

Discussion Creating types for each state can become a burden, since each state-type should be implemented as a singleton or flyweight. Using functions to encapsulate state specific behaviour avoids this overhead.

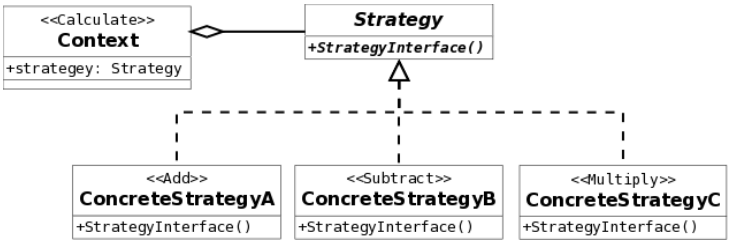
In addition, the code is easier to follow. With state objects the context's behaviour is dependent on the dynamic type of the state objects. Figuring out which Execute method is called at a certain point can be a daunting task.

A.3.9 Strategy

Intent Decouple an algorithm from its host by encapsulating the algorithm into its own type.

Context Consider a simple calculator that accepts two numbers as input and a button which defines the calculation to done with the two numbers. New types of calculation need to be added without having to change the calculator.

The solution is to encapsulate the calculations in independent constructs and provide a uniform interface to them.



Strategy provides an interface for supported algorithms. *Concrete Strategies* (add, subtract, multiply) encapsulate algorithms. The *Context*'s (calculator) behaviour depends on the dynamic type of the Strategy and can be changed dynamically. In GO, Strategies can be encapsulated in objects and in functions. We demonstrate both alternatives with the above described calculator example.

Example - Strategies as objects In this example we implement the strategies (add, multiply and subtract) as their own types.

`Strategy` defines the common interface for all algorithms. The method `Execute` accepts two integers and returns the result of the operation as an integer.

```
type Strategy interface {
  Execute(int, int) int
}
```

`Add`, `Subtract`, and `Multiply` are *Concrete Strategies*. Each having an `Execute` method that conforms to the `Strategy` interface. The `Execute` methods encapsulate the algorithms.

```
type Add struct{}
func (this *Add) Execute(a, b int) int {
  return a + b
}

type Subtract struct{}
func (this *Subtract) Execute(a, b int) int {
  return a - b
}

type Multiply struct{}
func (this *Multiply) Execute(a, b int) int {
  return a * b
}
```

`Calculator` is the *Context*. A `Calculator` object maintains a reference to a `Strategy` object and provides public write access. This enables package external Clients to configure `Calculator`'s strategy.

```
type Calculator struct {
  strategy Strategy
}
func (this *Calculator) SetStrategy(strategy Strategy) {
  this.strategy = strategy
}
```

Clients call `Calculate` on a `Calculator` object, which in turn executes the current strategy's `Execute` method. `Calculate`'s behaviour depends on the dynamic type of the object `strategy`.

```
func (this *Calculator) Calculate(a, b int) int {
  return this.strategy.Execute(a, b)
}
```

The following listing demonstrates how Clients alter the Calculators behaviour by configuring its strategy. The `calculator` object is the context that a strategy resides in. Clients set the `calculator`'s strategy. Upon calling `Calculate` on the context, the context calls its current strategy's `Execute` method and passes two numbers. The results differ since the dynamic type of calculator's strategy is different with each call of `Calculate`.

```
calculator := new(Calculator)
calculator.SetStrategy(new(Add))
calculator.Calculate(3, 4)  //result: 7
calculator.SetStrategy(new(Subtract))
calculator.Calculate(3, 4)  //result: -1
calculator.SetStrategy(new(Multiply))
calculator.Calculate(3, 4)  //result: 12
```

Note that new instances of the strategies are created. These strategies would make good Singletons (see Appendix A.1.5), but we kept the example simple to not distract from the pattern at hand.

Example - Strategies as functions GO has first class functions. Strategies can be encapsulated in functions and the Contexts store references to strategy *functions*, instead of strategy *objects*.

`Strategy` is a function type. Every function accepting two and returning one integer can be used where a `Strategy` is required.

```
type Strategy func(int, int) int
```

Here `Multiply` is not a struct, but a function encapsulating the algorithm. No additional types necessary.

```
func Multiply(a, b int) int {
  return a * b
}
```

`Add` and `Subtract` are not shown here for brevity. They follow the same pattern.

The definition of `Calculator` is the same as above. There is no syntactical difference in `SetState` either. However, there is a semantic difference, `Calculator` keeps a reference to a function instead of an object. Type `Strategy` is of function type, not of interface type.

```
type Calculator struct {
  strategy Strategy
}
```

`Calculate` differs in that it calls the strategy function directly instead of referring to the strategy's method `Execute`.

```
func (this *Calculator) Calculate(a, b int) int {
  return this.strategy(a, b)
}
```

Clients pass functions to the context instead of `Strategy` objects. On calling `Calculate` the context object calls its current strategy function and forwards the two integers and returns the result.

```
calculator := new(Calculator)
calculator.SetStrategy(Add)
calculator.Calculate(3, 4)  //result: 7
calculator.SetStrategy(Subtract)
calculator.Calculate(3, 4)  //result: -1
calculator.SetStrategy(Multiply)
calculator.Calculate(3, 4)  //result: 12
```

Discussion Defining a types and a method for each strategy is elaborate compared to defining a function. Strategies could be Singletons, if they don't need to store context specific state. Encapsulating strategies as functions removes the necessity for Singleton. If the strategies need to store state, they are better implemented with types. Strategy functions could store state in package-local static variables, but this would break encapsulation, since other functions in the package have access to those variables.

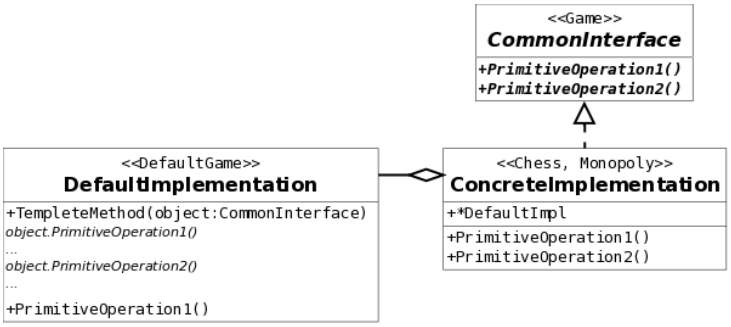
Encapsulation could be achieved by organizing each function in a separate package, but this comes at a cost: each package has to be in a separate folder, additionally the packages need to be imported before the strategy functions can be used. This seems to be more work than implementing separate types for the strategies.

A.3.10 Template Method

Intent The Template Method design pattern can be used when the outline of an algorithm should be static between classes, but individual steps may vary.

Context Consider a framework for turn-based games. All turn-based games in the framework are played in a similar order. The framework provides the playing algorithm and concrete games implement the steps of the algorithm.

The Temple Method pattern is a solution for this problem. The usual participants of Template Method are an abstract class declaring a final method (the algorithm). The steps of the algorithm can either be concrete methods providing default behaviour or abstract methods, forcing subclasses to override them. GO however, does not have abstract classes nor abstract methods. The diagram below shows the structure of the Template Method pattern as it is implemented in GO.



The variant steps (or hooks) of the *Template Method* are defined in a *Common Interface*. The invariant steps are implemented in a base type, *Default Implementation*. The variant steps (*Primitive Operation*s) can be given a default implementation in the base type. *Concrete Implementations* embed the Default Implementation. Concrete Implementations have to implement the missing methods of the Common Interface and can override the default behaviour.

Example We illustrate Template Method with a framework for turn-based games.

`Game` is the *Common Interface* and lists the *Primitive Operations*.

```
type Game interface {
  SetPlayers(int)
  InitGame()
  DoTurn(int)
  EndOfGame() bool
  PrintWinner()
}
```

`BasicGame`, the *Default Implementation*, defines `PlayGame`, the *Template Method*. `PlayGame` defines the algorithm games follow and calls for each variant step a primitive operation. The additional parameter game of type `Game` on `PlayGame` is necessary, to call "right" primitive operations. The receiver of `PlayGame` is `this` and `this` is of type `BasicGame`. The template method however has to call the primitive operations of the concrete game. Hence, calling `this.EndOfGame()` will result in an error, since `BasicGame` does not implement this method.

```
type BasicGame struct {}

func (this *BasicGame) PlayGame(game Game,
    players int) {
  game.SetPlayers(players)
  game.InitGame()
  for !game.EndOfGame() {
    game.DoTurn()
  }
  game.PrintWinner()
}
```

The `Chess` struct embeds `BasicGame`, some methods are overridden, some are not; the `EndOfGame` method is also supplied; therefore, `Chess` implicitly implements `Game`. Games of chess can be played by calling `PlayGame` on a Chess.

```
type Chess struct {
  *BasicGame
  ...
}
func (this *Chess) SetPlayers(players int) {}
func (this *Chess) DoTurn() { ... }
func (this *Chess) EndOfGame() bool { ... }
```

Here is how clients would use `Chess`:

```
chess := NewChess()
chess.PlayGame(chess, 2)
```

An instance of `Chess` is created and `PlayGame` called. The `chess` object gets passed alongside the numbers of players.

Discussion At first glance, the GO implementation looks much like the one from *Design Patterns*: a set of methods is defined in the `Game` interface, these methods are the fine grained parts of the algorithm; a `BasicGame` struct defines the static glue code which coordinates the algorithm (in the `PlayGame` method), and default implementations for most methods in `Game`.

The major difference compared with standard implementations is that we must pass a `Game` object to the `PlayGame` method twice: first as the receiver `this *BasicGame` and then as the first argument `game Game`. This is the client-specified self pattern[90]. `BasicGame` will be embedded into concrete games (like `Chess`). Inside `PlayGame`, calls made to this will call methods on `BasicGame`; GO does not dispatch back to the embedding object. If we wish the embedding object's methods to be called, then we must pass in the embedding object as an extra parameter. This extra parameter must be passed in by the client of `PlayGame`, and must be the same object as the first receiver. Every additional parameter is places a further

burden on maintainers of the code. Furthermore the parameter has the potentially for confusion: `chess.PlayGame(new(Monopoly), 2))`. Here we pass an object of type `Monopoly` to the `PlayGame` method of the object chess. This call will not play Chess, but Monopoly. We discuss this idiom further in Section 2.2.2.

The combination of `BasicGame` and the interface `Game` is effectively making `EndOfGame` an abstract method. `Game` requires this method, but `BasicGame` doesn't provide it. Embedding `BasicGame` in a type and not implementing `EndOfGame` will result in a compile time error, when used instead of a `Game` object.

The benefits of associating the algorithm with a type allows for providing default implementations, using of GO's compiler to check for unimplemented methods. The disadvantage being, that sub-types like `Chess` could override the template method, contradicting the idea of having the template method fixed and sub types implementing only the steps. In Java, the `PlayGame` method would be declared `final` so that the structure of the game cannot be changed. This is not possible in GO because there is no way to stop a method being overridden.

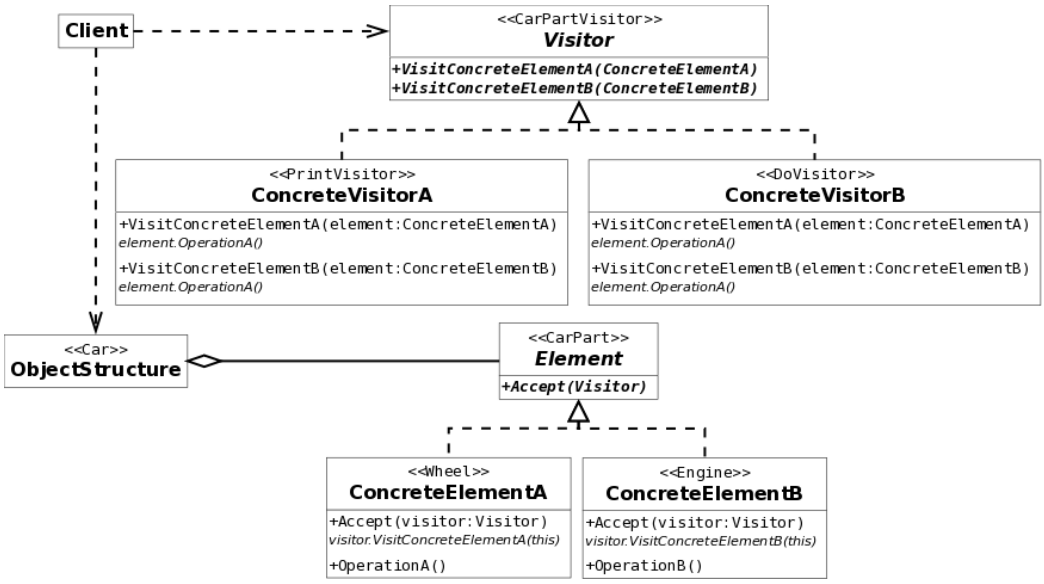
In class-based languages, the `BasicGame` class would usually be declared abstract, because it does not make sense to instantiate it. GO, however, has no equivalent of abstract classes. To prevent `BasicGame` objects being used, we do not implement all methods in the `Game` interface. This means that `BasicGame` objects cannot be used as the Client-specified self parameter to `PlayGame` (because it does not implement `Game`).

A.3.11 Visitor

Intent Separate the algorithm from an object structure it operates on.

Context Consider performing an action on every part of a complex object structure like a car, actions like printing out the parts name or checking the parts status. Each part offers different operations. The algorithms to be performed on the car parts are not known in advance. It is necessary to be able to add new algorithms without having to change the car or its parts.

The Visitor pattern offers a solution. The diagram below shows the structure of the pattern.



Visitor (car part visitor) defines a visit method for each type of *Concrete Element* (print visitor, check status visitor). The object that is being visited determines which visit method is to be called. *Concrete Visitors* implement the visit methods. The *Object Structure* (car) is made up of *Elements*. All Elements have to define a method accepting a visitor. Visitors can move over the Object Structure and visit each part of it calling their Accept

methods. The Accept methods call the visitor's corresponding visit method.

Example We demonstrate the Visitor pattern with the above described car example. A car consists of car parts. Visitors can visit the parts and perform an operation of each part.

`CarPart` defines the common interface for all elements that should be visitable. `CarPart` is the *Element* interface mentioned above.

```
type CarPart interface {
  Accept(CarPartVisitor)
}
```

`Wheel` and `Engine` are *Concrete Elements*. The `Accept` methods call the according `visit`XXX method of the visitor and pass a reference of themselves. The passed in `visitor` object will then do its functionality with the current object.

```
type Wheel struct {
  Name string
}
func (this *Wheel) Accept(visitor CarPartVisitor) {
  visitor.visitWheel(this)
}

type Engine struct {}
func (this *Engine) Accept(visitor CarPartVisitor) {
  visitor.visitEngine(this)
}
```

`Car` is the *Object Structure* maintaining a list of `CarPart` objects. `NewCar` assembles a car `this` with four wheels and an engine.

```
type Car struct {
  parts []CarPart
}

func NewCar() *Car {
  this := new(Car)
  this.parts = []CarPart{
    &Wheel{"front left"},
    &Wheel{"front right"},
    &Wheel{"rear right"},
    &Wheel{"rear left"},
    &Engine{}}
```

```
  return this
}
```

`Car` also implements the `Accept` method which iterates over the car parts and calls their `Accept` method. This is an example of the Composite Pattern (see Appendix A.2.3).

```
func (this *Car) Accept(visitor CarPartVisitor) {
  for _, part := range this.parts {
    part.Accept(visitor)
  }
}
```

`CarPartVisitor` is the common *Visitor* interface mentioned before declaring `visit`XXX operations, one for each concrete element type.

```
type CarPartVisitor interface {
  visitWheel(wheel *Wheel)
  visitEngine(engine *Engine)
}
```

The `PrintVisitor` is a *Concrete Visitor* implementing the visit methods declared in `CarPartVisitor`. The methods print a message on the command line, describing what kind of element was visited.

```
type PrintVisitor struct{}

func (this *PrintVisitor) visitWheel(wheel *Wheel) {
  fmt.Printf("Visiting the %v wheel\n", wheel.Name)
}

func (this *PrintVisitor) visitEngine(engine *Engine) {
  fmt.Printf("Visiting engine\n")
}
```

Using a `PrintVisitor` to print the elements of an object structure, e.g. a car, is straight forward:

```
car := NewCar()
car.Accept(new(PrintVisitor))
```

The `car` object's `Accept` method is called with an instance of `PrintVisitor`. `Accept` loops over the `car`'s parts and calls their `Accept` method passing on the `PrintVisitor` object. Each part calls the according `visit`XXX method of the visitor.

Discussion In programming languages supporting method overloading, like C++ and Java, every method listed in the Visitor interface can be called "visit". The distinction is made by overloading the type of the concrete elements they are visiting. GO lacks method overloading. Method names have to unique for each type. This causes some implementation overhead since each visit method carries its corresponding type twice: in the method name and in the type of its parameter.

Adding new behaviour is comparatively easy. All that is necessary is a type implementing the visit methods defined in CarPartVisitor. The existing code can remain unchanged. But it is hard to add new types. Consider adding the type Window as a new CarPart. The CarPartVisitor interface has to be extended with a visitWindow method and all existing visitors had to be changed to implement the new method. On the one hand the visitor pattern makes it easy to add new functionality to an existing type structure, but hard to add new types. On the other hand using embedding makes it easy to add new types, but adding additional functionality is hard. GO's language features do not help overcoming this problem, known as the Expression Problem [89].

Consider the implementation of the `PrintVisitor` below.

```
func PrintVisitor(part CarPart) {
  switch this := part.(type) {
  case *Wheel:
    fmt.Printf("Visiting the %v wheel\n", this.Name)
  case *Engine:
    fmt.Println("Visiting engine")
  case *Body:
    fmt.Println("Visiting body")
}
```

This function does the same as the type PrintVisitor: depending on the type of the visited car part the corresponding message is printed. Type switches are a bad alternative to dynamic dispatch.Each element type needs its own case statement and it is easy to forget a case. By using types, the compiler can be used to find missing visit methods, which is not the case when using functions with type switches. Furthermore visitors can accu-

mulate state, which is hard to do for functions (see Discussion of Strategy Appendix A.3.9). The advantage of implementing visitor with functions is that it is easier to add a function than a type with the corresponding methods.

Bibliography

[1] Bossie Awards 2010: The best open source application development software, 2010. http://www.infoworld.com/d/open-source/bossie-awards-2010-the-best-open-source-application-development-software-140; Accessed August–November 2010.

[2] Effective Go, 2010. http://golang.org/doc/effective_go.html; Accessed March–August 2010.

[3] Go Release History, 2010. http://golang.org/doc/devel/release.html; Accessed March–November 2010.

[4] Golang.org, 2010. http://golang.org; Accessed March–August 2010.

[5] Golang.org FAQ, 2010. http://golang.org/doc/go_faq.html; Accessed March – November 2010.

[6] Package container/Vector, 2010. http://golang.org/pkg/container/vector/; Accessed March–November 2010.

[7] TIOBE programming community index, 2010. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html; Accessed March–November 2010.

[8] AGERBO, E., AND CORNILS, A. *Theory of Language Support for Design Patterns*. Aarhus University, Department of Computer Science, 1997.

[9] AGERBO, E., AND CORNILS, A. How to preserve the benefits of design patterns. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications* (New York, NY, USA, 1998), vol. 33 of *OOPSLA '98*, ACM, pp. 134–143.

[10] ALEXANDER, C., ISHIKAWA, S., AND SILVERSTEIN, M. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, USA, 1977.

[11] ALPERT, S. R., BROWN, K., AND WOOLF, B. *The Design Patterns Smalltalk Companion*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[12] ALPHONCE, C., CASPERSEN, M., AND DECKER, A. Killer "Killer Examples" for Design Patterns. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education* (New York, NY, USA, 2007), SIGCSE '07, ACM, pp. 228–232.

[13] BECK, K., AND CUNNINGHAM, W. Using pattern languages for object-oriented programs. Tech. Rep. CR-87-43, OOPSLA '87: Workshop on the Specification and Design for Object-Oriented Programming, Sept. 1987.

[14] BECK, K., AND JOHNSON, R. Patterns generate architectures. In *Proceedings of the 8th European Conference on Object-Oriented Programming* (Bologna, Italy, July 1994), R. P. Mario Tokoro, Ed., no. 821 in Lecture Notes in Computer Science, Springer-Verlag, pp. 139–149.

[15] BISHOP, J. *C# 3.0 Design Patterns*. O'Reilly Media, Inc., 2007.

[16] BISHOP, J. Language features meet design datterns: Raising the abstraction bar. In *Proceedings of the 2nd International Workshop on the Role of Abstraction in Software Engineering* (New York, NY, USA, 2008), ROA '08, ACM, pp. 1–7.

[17] BOSCH, J. Design patterns as language constructs. *Journal of Object-Oriented Programming 11* (1998), 18–32.

[18] BRANT, J. M. Hotdraw. Master's thesis, University of Illinois, 1995.

[19] BREESAM, K. M. Metrics for object-oriented design focusing on class inheritance metrics. In *Proceedings of the 2nd International Conference on Dependability of Computer Systems* (Washington, DC, USA, 2007), DEPCOS-RELCOMEX '07, IEEE Computer Society, pp. 231–237.

[20] BÜNNIG, S., FORBRIG, P., LÄMMEL, R., AND SEEMANN, N. A programming language for design patterns. In *GI-Jahrestagung* (1999), Springer-Verlag, pp. 400–409.

[21] BUSCHMANN, F., HENNEY, K., AND SCHMIDT, D. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*, vol. 4 of *Wiley Software Patterns*. John Wiley & Sons, 2007.

[22] BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. *Pattern-Oriented Software Architecture: A System of Patterns*, vol. 1 of *Wiley Software Patterns*. John Wiley & Sons, Chichester, UK, 1996.

[23] CARDELLI, L., AND PIKE, R. Squeak: a language for communicating with mice. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1985), SIGGRAPH '85, ACM, pp. 199–204.

[24] CHAMBERS, C., HARRISON, B., AND VLISSIDES, J. A debate on language and tool support for design patterns. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2000), POPL '00, ACM, pp. 277–289.

[25] CHRISTENSEN, H. B. Frameworks: Putting design patterns into perspective. In *Proceedings of the 9th Annual SIGCSE Conference on Innova-*

tion and Technology in Computer Science Education (New York, NY, USA, 2004), ITiCSE '04, ACM, pp. 142–145.

[26] COPLIEN, J. O., AND SCHMIDT, D. C., Eds. *Pattern Languages of Program Design*. Addison-Wesley Professional, New York, NY, USA, 1995.

[27] CORP, I. *Occam Programming Manual*. Prentice Hall Trade, 1984.

[28] CUNNINGHAM, W. A CRC description of HotDraw, 1994. `http://c2.com/doc/crc/draw.html`; Accessed March – November 2010.

[29] DECKER, R., AND HIRSHFIELD, S. Top-down teaching: Object-oriented programming in CS 1. In *Proceedings of the Twenty-Fourth SIGCSE Technical Symposium on Computer Science Education* (New York, NY, USA, 1993), SIGCSE '93, ACM, pp. 270–273.

[30] DIJKSTRA, E. W. Letters to the editor: go to statement considered harmful. *Communications of the ACM 11* (Mar. 1968), 147–148.

[31] DORWARD, S., PIKE, R., PRESOTTO, D., RITCHIE, D., TRICKEY, H., AND WINTERBOTTOM, P. Inferno. In *Proceedings of the 42nd IEEE International Computer Conference* (Feb. 1997), COMPCON '97, pp. 241 –244.

[32] DORWARD, S., PIKE, R., AND WINTERBOTTOM, P. Programming in Limbo. In *Proceedings of the 42nd IEEE International Computer Conference* (Feb. 1997), COMPCON '97, pp. 245 –250.

[33] FAYAD, M., AND SCHMIDT, D. C. Object-oriented application frameworks. *Communications of the ACM 40* (Oct. 1997), 32–38.

[34] FOOTE, B., ROHNERT, H., AND HARRISON, N., Eds. *Pattern Languages of Program Design 4*. Addison-Wesley Professional, Boston, MA, USA, 1999.

[35] FREEMAN, E., FREEMAN, E., BATES, B., AND SIERRA, K. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004.

[36] FROEHLICH, G., HOOVER, H. J., LIU, L., AND SORENSON, P. Hooking into object-oriented application frameworks. In *Proceedings of the 19th International Conference on Software Engineering* (New York, NY, USA, 1997), ICSE '97, ACM, pp. 491–501.

[37] GABRIEL, R. Lisp: Good News Bad News How to Win Big. *AI Expert 6*, 6 (1991), 30–39.

[38] GAMMA, E. JHotDraw, 1996. `http://jhotdraw.sourceforge.net/`; Accessed March – November 2010.

[39] GAMMA, E., HELM, R., JOHNSON, R. E., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Mar. 1995.

[40] GANNON, J. D. An experimental evaluation of data type conventions. *Communications of the ACM 20*, 8 (Aug. 1977), 584–595.

[41] GAT, E. Lisp as an alternative to Java. *Intelligence 11*, 4 (2000), 21–24.

[42] GIL, J., AND LORENZ, D. Design patterns and language design. *Computer 31*, 3 (Mar. 1998), 118 –120.

[43] GUPTA, D. What is a good first programming language? *Crossroads 10*, 4 (2004), 7–7.

[44] HADJERROUIT, S. Java as first programming language: A critical evaluation. *SIGCSE Bulletin 30*, 2 (1998), 43–47.

[45] HANENBERG, S. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *Proceedings of the ACM International Conference on Object-oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2010), OOPSLA '10, ACM, pp. 22–35.

[46] HARMES, R., AND DIAZ, D. *Pro JavaScript Design Patterns*, 1 ed. Apress, Dec. 2007.

[47] HOARE, C. A. R. Communicating Sequential Processes. *Communications of the ACM 21* (Aug. 1978), 666–677.

[48] JOHNSON, R. E. Documenting frameworks using patterns. In *Proceedings on Object-Oriented Programming Systems, Languages, and Applications* (New York, NY, USA, 1992), OOPSLA '92, ACM, pp. 63–76.

[49] JOHNSON, R. E., AND BRANT, J. M. Creating tools in HotDraw by composition. In *13th International Conference on Technology of Object-Oriented Languages and Systems* (Versailles, France, Europe, 1994), B. Magnusson, B. Meyer, J.-M. Nerson, and J.-F. Perrot, Eds., no. 13 in TOOLS '94, Prentice Hall, pp. 445 – 454.

[50] JOHNSON, S., AND KERNIGHAN, B. The programming language B. Tech. Rep. 8, Bell Laboratories, Murry Hill, New Jersy, January 1973.

[51] KAISER, W. Become a programming Picasso with JHotDraw. *JavaWorld* (Feb. 2001).

[52] KELLEHER, C., AND PAUSCH, R. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys 37*, 2 (June 2005), 83–137.

[53] KERNIGHAN, B. A descent into Limbo. *Online White-Paper. Lucent Technologies 12* (1996).

[54] KERNIGHAN, B. W. Why Pascal is not my favorite programming language. Tech. Rep. 100, AT&T Bell Laboratories, Murray Hill, New Jersey 07974, Apr. 1981.

[55] KERNIGHAN, B. W. *The UNIX Programming Environment*. Prentice Hall Professional Technical Reference, 1984.

[56] KIRCHER, M., AND JAIN, P. *Pattern-Oriented Software Architecture: Patterns for Resource Management*, vol. 3 of *Wiley Software Patterns*. John Wiley & Sons, 2004.

[57] KÖLLING, M., KOCH, B., AND ROSENBERG, J. Requirements for a first year object-oriented teaching language. In *Proceedings of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education* (New York, NY, USA, 1995), SIGCSE '95, ACM, pp. 173–177.

[58] KÖLLING, M., AND ROSENBERG, J. Blue – a language for teaching object-oriented programming. In *Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education* (New York, NY, USA, 1996), SIGCSE '96, ACM, pp. 190–194.

[59] KÖLLING, M., AND ROSENBERG, J. An object-oriented program development environment for the first programming course. In *Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education* (New York, NY, USA, 1996), SIGCSE '96, ACM, pp. 83–87.

[60] MANOLESCU, D., VOELTER, M., AND NOBLE, J., Eds. *Pattern Languages of Program Design 5*. Addison-Wesley Professional, 2005.

[61] MARTIN, R. C., RIEHLE, D., AND BUSCHMANN, F., Eds. *Pattern Languages of Program Design 3*. Addison-Wesley Professional, Boston, MA, USA, 1997.

[62] MATSUMOTO, Y. Go-Gtk library, 2010. `http://mattn.github.com/go-gtk/`; Accessed March–June 2010.

[63] MAZAITIS, D. The object-oriented paradigm in the undergraduate curriculum: A survey of implementations and issues. *SIGCSE Bullettin 25* (Sept. 1993), 58–64.

[64] MCIVER, L. Evaluating languages and environments for novice programmers. In *14th Workshop of the Psychology of Programming Interest Group* (June 2002), Brunel University, pp. 100–110.

[65] METSKER, S. J., AND WAKE, W. C. *Design Patterns in Java*, 2. ed. Addison-Wesley Professional, Upper Saddle River, NJ, 2006.

[66] NORVIG, P. Design patterns in dynamic programming. *Object World 96* (1996). http://norvig.com/design-patterns/.

[67] OLSEN, R. *Design Patterns in Ruby*, 1 ed. Addison-Wesley Professional, Dec. 2007.

[68] PARKER, K. R., OTTAWAY, T. A., CHAO, J. T., AND CHANG, J. A formal language selection process for introductory programming courses. *Journal of Information Technology Education 5* (2006), 133–151.

[69] PIKE, R. Newsqueak: A language for communicating with mice. Tech. Rep. 143, Bell Laboraties, Apr. 1994.

[70] PIKE, R. Go, the programming language, Oct. 2009. http://golang.org/doc/talks/go_talk-20091030.pdf; Accessed May – November 2010.

[71] PIKE, R. Another Go at language design, Apr. 2010. http://www.stanford.edu/class/ee380/Abstracts/100428-pike-stanford.pdf; Accessed May – November 2010.

[72] PIKE, R. Another Go at language design, July 2010. http://assets.en.oreilly.com/1/event/45/Another%20Go%20at%20Language%20Design%20Presentation.pdf; Accessed July – November 2010.

[73] PIKE, R. The expressiveness of Go, October 2010. http://golang.org/doc/ExpressivenessOfGo.pdf; Accessed October – November 2010.

[74] PIKE, R. Go, July 2010. http://assets.en.oreilly.com/1/event/45/Go%20Presentation.pdf; Accessed July – November 2010.

[75] PIKE, R., PRESSOTO, D., THOMPSON, K., AND TRICKEY, H. Plan 9 from Bell Labs. *Computing Systems 8* (1995), 221 – 253.

[76] PRECHELT, L. Comparing Java vs. C/C++ efficiency differences to interpersonal differences. *Communications of the ACM 42*, 10 (1999), 109–112.

[77] PRECHELT, L. An empirical comparison of seven programming languages. *Computer 33*, 10 (Oct. 2000), 23–29.

[78] PUGH, J. R., LALONDE, W. R., AND THOMAS, D. A. Introducing object-oriented programming into the computer science curriculum. In *Proceedings of the Eighteenth SIGCSE Technical Symposium on Computer Science Education* (New York, NY, USA, 1987), SIGCSE '87, ACM, pp. 98–102.

[79] REENSKAUG, T. Models - Views - Controllers. Tech. rep., Xerox Parc, Dec. 1979.

[80] REENSKAUG, T. Thing - Model - View - Editor. Tech. Rep. 5, Xerox Parc, May 1979.

[81] RIEHLE, D. Case study: The JHotDraw framework. In *Framework Design: A Role Modeling Approach*. ETH Zürich, 2000, ch. 8, pp. 138–158.

[82] RITCHIE, D. M. The development of the C language. In *The Second ACM SIGPLAN Conference on History of Programming Languages* (New York, NY, USA, 1993), HOPL-II, ACM, pp. 201–208.

[83] SCHMAGER, F., CAMERON, N., AND NOBLE, J. GoHotDraw: Evaluating the Go programming language with design patterns. In *Evaluation and Usability of Programming Languages and Tools (PLATEAU) 2010* (2010).

[84] SCHMIDT, D. C., STAL, M., ROHNERT, H., AND BUSCHMANN, F. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, vol. 2 of *Wiley Software Patterns*. John Wiley & Sons, Chichester, UK, 2000.

[85] SHALLOWAY, A., AND TROTT, J. *Design Patterns Explained: A new Perspective on Object-Oriented Design*, 2 ed. Software Patterns. Addison-Wesley Professional, 2004.

[86] SKUBLICS, S., AND WHITE, P. Teaching smalltalk as a first programming language. In *Proceedings of the Twenty-Second SIGCSE Technical Symposium on Computer Science Education* (New York, NY, USA, 1991), SIGCSE '91, ACM, pp. 231–234.

[87] TEMTE, M. C. Let's begin introducing the object-oriented paradigm. In *Proceedings of the Twenty-Second SIGCSE Technical Symposium on Computer Science Education* (New York, NY, USA, 1991), SIGCSE '91, ACM, pp. 73–77.

[88] TICHY, W. Should computer scientists experiment more? *Computer 31*, 5 (May 1998), 32–40.

[89] TORGERSEN, M. The expression problem revisited. In *ECOOP 2004 - Object-Oriented Programming* (2004), vol. 3086 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 1–44.

[90] VILJAMAA, P. Client-Specified Self. In *Pattern Languages of Program Design*. Addison-Wesley Publishing Co., New York, NY, USA, 1995, ch. 26, pp. 495–504.

[91] VIRDING, R., WIKSTRÖM, C., AND WILLIAMS, M. *Concurrent Programming in Erlang*, 2 ed. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[92] VLISSIDES, J. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1998.

[93] VLISSIDES, J. M., COPLIEN, J. O., AND KERTH, N. L., Eds. *Pattern Languages of Program Design 2*. Addison-Wesley Professional, Boston, MA, USA, 1996.

[94] WEINAND, A., AND GAMMA, E. ET++ – a portable, homogenous class library and application framework. In *Proceedings of the UBILAB '94 Conference* (1994), Universitätsverlag Konstanz.

[95] WINTERBOTTOM, P. *Alef Language Reference Manual*, 2 ed. Bell Labs, Murray Hill, 1995.

[96] WIRTH, N. The programming language Pascal. *Acta Informatica 1*, 1 (1971), 35–63.

[97] WIRTH, N., AND GUTKNECHT, J. *Project Oberon: The Design of an Operating System and Compiler*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1992.