GoHotDraw: Evaluating the Go Programming Language with Design Patterns

Frank Schmager
Victoria University of Wellington, New Zealand
frank.schmager@ecs.vuw.ac.nz

Nicholas Cameron
Victoria University of Wellington, New Zealand
ncameron@ecs.vuw.ac.nz

James Noble
Victoria University of Wellington, New Zealand
kjx@ecs.vuw.ac.nz

Abstract

Go, a new programming language backed by Google, has the potential for widespread use: it deserves an evaluation. Design patterns are records of idiomatic programming practice and inform programmers about good program design. In this study, we evaluate Go by implementing design patterns, and porting the "pattern-dense" drawing framework HotDraw into Go, producing GoHotDraw. We show how Go's language features affect the implementation of Design Patterns, identify some potential Go programming patterns, and demonstrate how studying design patterns can contribute to the evaluation of a programming language.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General

General Terms Design, Languages

Keywords Go, Design Patterns

1. Introduction

Go is a new object-oriented programming language, developed at Google by Rob Pike and others. Go is used "internally at Google for some production stuff" [18] and has found an active community since its publication in November 2009 [1]. This paper describes our experiences implementing the patterns from *Design Patterns* [9], and the HotDraw drawing editor framework in Go. We discuss the differences between our implementations of design patterns in Go and in other programming languages (primarily Java) and suggest programming idioms in Go that may develop into design patterns.

Many researchers have proposed methods and criteria for evaluating programming languages [26, 22]. Direct comparisons of programming languages have been conducted: a comparison of Java with C♯ [5]; a comparison of FORTRAN-77, C, Pascal and Modula-2 [13]. Others investigated suitability as a first programming language: Parker et al. compiled a list of criteria for introductory programming courses at universities [20]; McIver proposed evaluating languages together with IDEs [16]; Hadjerrouit examined Java's suitability as a first programming language [11]; Clarke used questionnaires to evaluate a programming language [6]; Gupta discussed requirements for programming languages for beginners [10]. The methods and criteria proposed in the literature are usually hard to measure and are prone to be subjective.

In this paper, we use design patterns to evaluate a programming language. There are a number of books describing the GoF design patterns in different programming languages (Smalltalk [4], Java [17], JavaScript [12], Ruby [19]). Implementations of design patterns differ due to specifics of the language used. There is a need for programming language specific pattern descriptions. 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 have implemented all 23 patterns from *Design Patterns* in Go: for space reasons we discuss only the Singleton, Adapter, and Template Method patterns in this paper. The implementations of these patterns illuminate specific Go language features: Singleton demonstrates how Go's source code is structured in packages rather than classes, and Go's visibility rules; Adapter allows us to compare embedding with composition in Go; and Template Method illustrates the flow of control between embedding and embedded objects. To investigate how design patterns integrate in Go, we ported the core of the well-known drawing framework *HotDraw* into Go, thus *GoHotDraw*.

This paper is organised as follows: in Sect. 2 we give a brief overview of Go; in Sect. 3 we present case studies implementing three design patterns and JHotDraw in Go; in Sect. 4 we discuss our experience using Go, with a focus on design patterns; in Sect. 5 we discuss future work; in Sect. 6 we conclude.

2. Background

In this section we introduce the Go programming language.

2.1 The Go Programming Language

Go is an object-oriented programming language with a C-like syntax. Go is designed as a systems language and has a focus on concurrency; it is "expressive, concurrent, garbage-collected" [3]. It supports a mixture of static and dynamic typing, and is designed to be safe and efficient. A primary motivation seems to be compilation speed (and indeed, Go programs compile blazingly fast).

Go has objects, and structs rather than classes. Go has no concept of inheritance, reuse is supported by *embedding*. There are no explicit subtype declarations, but in some cases, structural subtypes are inferred. Pointers are explicit and indicated by an asterisk before the type. Unlike C, however, pointer arithmetic is not allowed.

Objects and methods The following code example defines the type `Car` with a single field `affordable` with type `bool`:

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

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) {...}
```

Embedding Reuse in Go is supported by embedding. 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. 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
}
```

Objects can be deeply embedded and multiply embedded. Name conflicts are only a problem if the ambiguous element is accessed.

Interfaces Interfaces in Go are abstract representations of behaviour — sets of methods. The Go compiler infers if an object satisfies an interface. This part of the type system is structural: if an object implements the methods of an interface, then it has that interface as a type. 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.

Functions and goroutines Go supports functions (methods with no receiver) as well as function pointers and closures. *Goroutines* are functions that execute in parallel with other goroutines. Goroutines are designed to be lightweight and to hide many of the complexities of thread creation and management. The keyword 'go' in front of a function call executes the function in its own goroutine. Goroutines can communicate (asynchronously) with other goroutines through *channels*.

Object initialisation 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.

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 `Example` object is created and a pointer to the new object stored in `anExample`. In each case the fields have the following values:
`Example.Name == ""` and `Example.aField == 0`.

```
1 var anExample *Example
2 anExample = new(Example)
3 anExample = &Example{}
4 anExample = &Example{"",0}
5 anExample = &Example{Name:"",aField:0}
```

Packages Go source code is structured in packages. Go provides two scopes of visibility: members beginning with a lower case letter (e.g., `affordable`) are only accessible inside their package; members beginning with an upper case letter (e.g., `Car`, `Refuel()`) are publicly visible.

3. Case Studies

We have implemented all 23 GoF design patterns in Go. In this section, we describe our experience in implementing three of them: Singleton, Adaptor, and Template Method.

3.1 Singleton

The Singleton design pattern [9] ensures that only a single instance of a type exists in a program, and provides a global access point to that object.

In Java, Singletons are implemented with static fields. In Go, there are no static class members, so instead we use Go's package access mechanisms and functions to provide similar functionality.

For example, we may wish to have only a single registry object in a program:

```
package reg

type Registry interface {
  DoSomething()
}

type registry struct {
    ...
}

var theRegistry Registry

func GetRegistry() Registry {
  if theRegistry == nil {
    theRegistry = &registry{}
  }
  return theRegistry
}

func (this *registry) DoSomething() { ... }
```

The variable `theRegistry` is a reference to the only instantiation of `registry`. The struct `registry` cannot be named outside the `reg` package, so it can only be instantiated inside the package. The public function `GetRegistry` provides access to the singleton `registry`, creating a new object if none exists when it is called. The object is provided via the interface `Registry` so that it can be used and stored outside the package. Variables with interface type are always passed by reference, so copies of `theRegistry` will not be made when calling functions or methods. `Registry` is a non-empty interface, so it is unlikely to be implemented by accident (see Sect. 4.1); other objects implementing `Registry` can be created (as in other implementations of Singleton). Our implementation does not prevent multiple instantiations of `registry` being created inside the `reg` package; in particular, this might be done inadvertently by embedding `registry`.

3.2 Adapter

The Adapter design pattern adapts the interface of an object into a different interface to make otherwise incompatible objects work together.

In the following example, we will adapt a `Reptile` object to the `Animal` interface; the `Slither` method of `Reptile` provides the functionality found in `Animal`'s `Move`.

```
package animals

type Animal interface {
  Move()
}

type Cat struct {}
func (this *Cat) Move() { ... }


package reptiles

type Reptile struct {}

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

The struct `ReptileAdapter` adapts an embedded reptile so that it can be used as an `Animal`. `ReptileAdapter` objects implicitly implement the `Animal` interface.

```
package client

type ReptileAdapter struct {
  *reptiles.Reptile
}

func (this *ReptileAdapter) Move() { Slither() }
```

Alternatively, we could use composition rather than embedding in our adapter object. This is a similar situation to class-based languages where an adapter can use inheritance or composition. We discuss composition vs. embedding as a design choice in Sect. 4.

3.3 Template Method

The Template Method design pattern can be used when the outline of an algorithm should be invariant between classes, but individual steps may vary.

We illustrate Template Method in Go with a framework for turn-based games. At first glance, the Go implementation looks much like the standard one: 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`.

```
type Game interface {
  SetPlayers(int)
  InitGame()
  DoTurn(int)
  EndOfGame() bool
  PrintWinner()
}

type BasicGame struct {}

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

func (this *BasicGame) SetPlayers(players int) {}
func (this *BasicGame) InitGame() {}
func (this *BasicGame) DoTurn() {}
func (this *BasicGame) PrintWinner() {
  fmt.Println("A Player won")
}
```

Below is a concrete game of chess. The `Chess` struct embeds `BasicGame`, some methods are overridden, some are not; the `EndOfGame` method is also supplied; therefore, `Chess` implicitly implements `Game` and games of chess can be played by calling `PlayGame` on a `Chess` object.

```
type Chess struct {
  *BasicGame
    ...
}

func (this *Chess) SetPlayers(players int) {}
func (this *Chess) DoTurn() { ... }
func (this *Chess) EndOfGame() bool { ... }
```

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 [25]. `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. We discuss this idiom further in Sect. 4.1.

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`). We cannot prevent `BasicGame` ever being instantiated, but we can prevent it being used within this code.

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.

3.4 GoHotDraw

HotDraw is a framework for drawing editors and graphical applications [14]. As a larger case study, we ported the core of HotDraw to Go, thus GoHotDraw. We chose to port HotDraw because its design is well known, and because it makes extensive use of design patterns [7, 23, 15]. We based our port on the the source code and documentation of JHotDraw 5.3 and 7.2 [8].

Because JHotDraw is quite large, we concentrated on the essence of the framework (Drawings contain Views, Views contain Figures, Editors enable Tools to manipulate Drawings) and implemented only a subset of HotDraw's features — we support only rectangle Figures, but those figures can be selected, and then moved, and resized via handles. This subset is sufficient to cover all the key patterns in HotDraw's design. As a guide, JHotDraw v5.3 is about 15000 lines of code and v7.2 about 70000 lines, including several complete applications. Our GoHotDraw subset is around 2000 lines.

GoHotDraw's design is very similar to the Smalltalk and Java versions of HotDraw. GoHotDraw uses many patterns, and generally they work as well in Go as in other languages. For example, the Composite pattern supports composite Figures; the Observer pattern notifies Views of changes in the Figures they display; the Chain of Responsibility pattern manipulates Figures via Handles; the Mediator pattern couples Views and Tools via DrawingEditors; the Strategy pattern supports multiple repainting algorithms; the

Adapter pattern couples GoHotDraw to one of Go's underlying graphics libraries (XGB); the Null Object pattern [27] introduces NullHandles and NullTools to avoid handling null objects.

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 — perhaps for this reason, they tend to declare more than just the one or two methods usually preferred in Go [21].

The key difference between the Go and Java HotDraw designs comes from the difference between Go's embedding and Java's inheritance (described in Sect. 3.3): methods on "inner" embedded structs in Go cannot call back to methods in "outer" embedding objects. 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 — the central interface of HotDraw — 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, used as a client-specified self to support template methods.

To take one simple example: a figure is empty if its size is smaller than 3-by-3 pixels. This method is defined for DefaultFigure, and calls the GetSize method defined in the Figure interface:

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

The problem is that this method needs to call GetSize on the correct "substructure" (aka subclass): in HotDraw, a RectangleFigure inherits from DefaultFigure and thus inherits a suitable definition for GetSize (as well as other methods). In Java, DefaultFigure could simply call `this.GetSize()` and the call will be dynamically dispatched and run the correct method. In Go, this call will try to invoke the (non-existent) GetSize method on DefaultFigure: a client-specified self is needed for dynamic dispatch. We would be interested to see if there were a more Go-flavoured way to implement this functionality.

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, as the final version of the Figure interface illustrates. Of 21 methods in that interface, six require the additional client-specified self argument (highlighted in bold):

```
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(other 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
}
```

4. Discussion

In this section, we reflect on what we have learnt about design patterns in Go: both how existing patterns are implemented in Go, and what new, Go-specific patterns we may have uncovered.

Rob Pike has described design patterns as "add-on models" for languages whose "standard model is oversold" [21]. 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, seem to be more difficult. In general, we were surprised by how similar the Go pattern implementations were to implementations in other languages such as C++ and Java.

As well as design patterns, the Gang of Four [9] suggest some general design principles:

"Program to an interface, not an implementation" Go helps programmers to follow this advice. Go's syntax for interfaces is concise and straightforward. Types implicitly implement interfaces; there is no additional syntactic overhead. On the other hand, Go does not have abstract classes. Go can simulate abstract *methods* however, as discussed below.

"Favour object composition over class inheritance" Go has no inheritance. Reuse can be achieved through either language-level embedding or program-level composition. Go thus favours composition over inheritance. The GoF's experience was that designers overused inheritance. 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.

4.1 Go idioms

We found that programming in Go required some idiomatic programming practices. It is probably premature to call these design patterns: there may be better, more Go-specific ways of addressing the problems, or they may be found to be indicative of bad practice rather than good.

Client-specified self Embedding does not support all the usual object-oriented behaviour of dynamic dispatch on the self/this object. 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 [25]. To use dynamic dispatch, the method's receiver must also still be supplied.

Using client-specified self is less satisfactory than proper 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. The extra parameter complicates the code making it harder to read and write, for no clear benefit over inheritance.

Abstract classes Go has no equivalent of abstract classes (classes which are partially implemented and cannot be instantiated). Abstract classes are commonly used both in design patterns and object-oriented programming in general. Interfaces can be used to define methods without implementations, but these cannot easily be combined with partial implementations. As illustrated by our discussion of the Template Method pattern (Sect. 3.3), however,

Go's implicit interface declarations provide a partial work around: concrete methods are provided in a base-struct and an interface is provided which is a superset of these methods. The difference between the two are the abstract methods (C++'s pure virtual or Smalltalk's subclassResponsibility). The interface is then used as the type of the client-specified self parameter. This idiom is deficient because the base object can still be instantiated, and the idiom relies on clients obeying the client-specified self protocol.

Multiple constructors Go does not support constructors. If initialisation code is required for an object, then the recommended solution is to use a simple kind of factory method [2]. The Factory Method pattern [9] (or at least a simplification of it) is thus regularly used in Go programs. Go does not support overloading, and so if multiple 'constructors' are required, then the factory methods must have different names. We found that this was common and that having to use different names was inconvenient — there is no naming convention for multiple different factory methods, so clients wishing to instantiate an object must either guess or check the documentation.

Comma OK Methods in Go can return multiple values. This facility is often used to return an error signal: one return value is the result and the other 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. On the other hand, error conditions are easier to ignore because error checking is not enforced by the compiler.

Not-quite-a-marker interface Parameters in Go are often given empty interface types 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 does not help the compiler check the programmers intent. 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." [21]) 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.

4.2 Syntax

The Go syntax is an improvement over languages such as Java or C++ (e.g., it supports built in slices and maps). However, it is more verbose and less elegant than other modern languages such as Haskell and Python. For example,

- Not requiring semicolons is nice, but requiring braces, is clumsy e.g. compared to Haskell's layout syntax (though this is a somewhat personal preference). Forcing the programmer to put braces only on particular lines seems indefensibly inconvenient.
- Interface types are implicitly references, but other types must be explicitly marked as pointers; we found this inconsistency tripped us up repeatedly. There is no convention for distinguishing interface names from type names. It is therefore hard to interpret interface declarations without knowing which other types are interfaces (and therefore always passed by reference) or structs (always passed by value, and thus should often be pointer types).
- The initial-capital-for-public encapsulation scheme is nice to write, but hard to read, and making mistakes is easy. Especially because many other languages use the incompatible lower-case-for-values, inital-capital-for-types convention.
- We found Go's multifarious object creation syntax — some with and some without the `new` keyword — hard to interpret.
- Defining methods outside classes, and having to specify the receiver's name and type explicitly is repetitive, and makes methods hard to find, and hard to distinguish from functions in the same package.

5. Future Work

We have attempted to evaluate Go using design patterns. Any such evaluation is dependent on the design patterns used. The Gang of Four patterns are general-purpose programming patterns, and so our evaluation is of Go as a general-purpose language. Go is specifically targeted at the systems and concurrent domains. Our most immediate future work is thus to evaluate Go using concurrency and networking patterns [24], and patterns for systems programming.

Our evaluation has so far been qualitative. We would like to extend our study into quantitative evaluation by applying metrics to existing Go source code to understand how Go language features are used 'in the wild'. Such a study would currently be hampered by the small volume of Go source code, but this situation is rapidly improving.

6. Conclusion

In this paper we have introduced Go, and evaluated it using design patterns. Our implementations of design patterns have highlighted Go-specific features including embedding and interface inference. Embedding allows for an easy implementation of the Adapter pattern, but can make flow of control more complicated, as seen in our implementation of Template Method.

Go is a language which aims to do things differently, adopting a new object model that is significantly different from most object-oriented languages. At least for classical object-oriented programs, such as drawing editors and frameworks, designs in Go do not seem to be significantly different to designs in Java or C++. In some circumstances, the differences between embedding and inheritance can make some patterns more difficult to implement, but Go-specific idioms (or patterns) can resolve most of these difficulties.

Acknowledgments

This work was funded in part by a Build IT Postdoctoral Fellowship.

References

[1] Golang.org Community, 2010. `http://golang.org/doc/community.html`; Accessed March – November 2010.

[2] Effective Go, 2010. `http://golang.org/doc/effective_go.html`; Accessed March–August 2010.

[3] Golang.org, 2010. `http://golang.org`; Accessed March–August 2010.

[4] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[5] Shyamal Suhana Chandra and Kailash Chandra. A Comparison of Java and C♯. *J. Comput. Small Coll.*, 20(3):238–254, 2005.

[6] Steven Clarke. Evaluating a new programming language. In *13th Workshop of the Psychology of Programming Interest Group*, pages 275–289, 2001.

[7] Ward Cunningham. A CRC Description of HotDraw. `http://c2.com/doc/crc/draw.html`, 1994. Retrieved 15/07/2010.

[8] Erich Gamma. JHotDraw. `http://jhotdraw.sourceforge.net/`, 1996. Retrieved 15/07/2010.

[9] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, March 1995.

[10] Diwaker Gupta. What is a good first programming language? *Crossroads*, 10(4):7–7, 2004.

[11] Said Hadjerrouit. Java as first programming language: a critical evaluation. *SIGCSE Bull.*, 30(2):43–47, 1998.

[12] Ross Harmes and Dustin Diaz. *Pro JavaScript Design Patterns*. Apress, 1 edition, December 2007.

[13] Neal M. Holtz and William J. Rasdorf. An evaluation of programming languages and language features for engineering software development. *Engineering with Computers*, 3(4):183–199, December 1988.

[14] Ralph E. Johnson. Documenting frameworks using patterns. In *OOPSLA '92: Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, pages 63–76, New York, NY, USA, 1992. ACM.

[15] Wolfram Kaiser. Become a programming Picasso with JHotDraw. *JavaWorld*, February 2001.

[16] Linda McIver. Evaluating languages and environments for novice programmers. In *14th Workshop of the Psychology of Programming Interest Group*, pages 100–110. Brunel University, June 2002.

[17] Steven J. Metsker and William C. Wake. *Design Patterns in Java*. Addison-Wesley, Upper Saddle River, NJ, 2. edition, 2006.

[18] Cade Metz. Google programming Frankenstein is a Go. *The Register*, May 2010.

[19] Russ Olsen. *Design Patterns in Ruby*. Addison-Wesley Professional, 1 edition, December 2007.

[20] Kevin R. Parker, Thomas A. Ottaway, Joseph T. Chao, and Jane Chang. A Formal Language Selection Process for Introductory Programming Courses. *Journal of Information Technology Education*, 5:133–151, 2006.

[21] Rob Pike. Another go at language design. `http://www.stanford.edu/class/ee380/Abstracts/100428-pike-stanford.pdf`, April 2010. Retrieved 15/07/2010.

[22] Terrence W Pratt and Marvin V Zelkowitz. *Programming Languages: Design and Implementation*. Pearson Education, Inc., 4 edition, 2001.

[23] Dirk Riehle. Case Study: The JHotDraw Framework. In *Framework Design: A Role Modeling Approach*, chapter 8, pages 138–158. ETH Zrich, 2000.

[24] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, Chichester, UK, 2000.

[25] Panu Viljamaa. Client-specified self. In *Pattern Languages of Program Design*, chapter 26, pages 495–504. Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[26] N. Wirth. Programming languages: What to demand and how to assess them. In *Symposium on Software Engineering*. Belfast, April 1976.

[27] Bobby Woolf. Null object. In *Pattern Languages of Program Design 3*, chapter 1, pages 5–18. Addison-Wesley Longman Publishing Co., Inc., 1997.