Journal of Universal Computer Science, vol. 16, no. 2 (2010), 246-270
submitted: 19/10/09, accepted: 10/12/09, appeared: 28/1/10

Revisiting the Visitor: the "Just Do It" Pattern

Didier Verna
(EPITA Research and Development Laboratory, Paris, France
didier@lrde.epita.fr)

Abstract: While software design patterns are a generally useful concept, they are often (and mistakenly) seen as ready-made universal recipes for solving common problems. In a way, the danger is that programmers stop thinking about their actual problem, and start looking for pre-cooked solutions in some design pattern book instead. What people usually forget about design patterns is that the underlying programming language plays a major role in the exact shape such or such pattern will have on the surface. The purpose of this paper is twofold: we show why design pattern expression is intimately linked to the expressiveness of the programming language in use, and we also demonstrate how a blind application of them can in fact lead to very poorly designed code.
Key Words: Design Patterns, Lisp, Object Orientation, Meta-programming
Category: D.1.5, D.3.3

1 Introduction

A software design pattern is a three-part rule which expresses a relation between a certain context, a problem, and a solution. The well-known "GOF book" [Gamma et al., 1994] describes 23 software design patterns. Its influence in the software engineering community has been dramatic. However, two years after its publication, Peter Norvig noted that "16 of [these] 23 patterns are either invisible or simpler [. . .] in Dylan or Lisp" [Norvig, 1996].

We claim that this is not a consequence of the notion of "pattern" itself, but rather of the way patterns are generally described; the GOF book being typical in this matter. Whereas patterns are supposed to be general and abstract, the GOF book is actually very much oriented towards mainstream object languages such as C++. Interestingly enough, the book itself acknowledges this fact:

> Although design patterns describe object-oriented designs, they are based on practical solutions that have been implemented in *mainstream* object-oriented programming languages [. . .]

and later:

> Similarly, some of our patterns are supported directly by the less common object-oriented languages.

Several polls conducted by the author reveal that most programmers well acquainted with design patterns have either skipped, or completely forgotten about the above excerpts from the book's introduction.

The GoF book classifies design patterns in 3 categories: creational, structural and behavioral. Suffice to say that this classification is *usage-oriented*. Another important series of books in the design patterns field, hereby referred to as the Posa books [Buschmann et al., 1996] provide another classification for design patterns: architectural patterns, design patterns, and idioms. This classification is *abstraction-oriented* instead. Their definition of *idioms* is of interest to us:

> An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them *using the features of the given language*. [...] They address aspects of both design and implementation.

With this in mind, it turns out that most of the GoF book's 23 design patterns are actually closer to "programming patterns", or "idioms", if you choose to adopt the terminology of the Posa books. As a consequence, there is a risk in seeing a specific design pattern as more universal than it actually is, and this risk is described explicitly in the first of the Posa books:

> [...] sometimes, an idiom that is useful for one programming language does not make sense into another.

The Visitor pattern is a perfect example of this. As indicated in the GoF book:

> Use the Visitor pattern when [...] many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations.

But who said these operations belong to the classes? Not all object systems have methods inside classes.

The purpose of this paper is twofold: to demonstrate the risk of inappropriate usage of a design pattern, and to show how a generally useful pattern can be blurred into an expressive enough language, sometimes to the point of complete disappearance. These demonstrations will be conducted through the Visitor pattern example, in the context of dynamic languages, and specifically from the angle of Clos, the Common Lisp Object System.

In a first step, we ground our discussion in a C++ example in which the Visitor pattern, as described in the GoF book, is actually very useful. Our second step is perhaps rather unconventional: instead of directly showing the "correct" way to do the same thing in Lisp, we build a solution by blindly applying the same pattern, and we explain why the resulting code is in fact very poorly designed. From the author's experience, this approach has proven quite educational. The next steps are devoted to progressively cleaning up our initial solution by introducing features of the Common Lisp language. The last section present extensions to the original problem, and show that Common Lisp has enough expressive power to allow for quite simple and short solutions, something not as straightforward in other object-oriented languages.

For the sake of conciseness and clarity, only simplified code excerpts are presented throughout this paper. However, fully functional code (both in C++ and in Lisp) is available for download at the author's website[1].

2 The Visitor pattern in C++

In this section, we describe a possible application for the Visitor pattern in C++ [C++, 1998], with one of the most widely used example[2]. Suppose we want to model a Porsche, as an object composed of an engine and a body, itself composed of four wheels. We also want to implement two actions on the Porsche: painting and recycling. These actions must propagate to every component. For example, painting the Porsche means painting the engine, the body and every wheel in turn.

2.1 Naive implementation

A mostly complete listing of a naive implementation in C++ is given in appendix A on page 21. In this implementation, one C++ class per component is created (`Wheel`, `Body`, `Engine` and `Porsche`), and every instance of these classes must implement a `paint` and `recycle` method to perform the requested action.

This implementation suffers from two problems that the Visitor pattern is precisely designed to solve.

2.1.1 New actions

Implementing a new action, say `cleanup`, requires that one modifies the original classes by hand, so as to provide a new `cleanup()` method in all of them. This is at best undesirable and might not even be possible.

In general, we would rather keep the original model read-only (and this also means *without* the original `paint` and `recycle` methods) because conceptually speaking, the actions that we want to perform on a Porsche are orthogonal to the Porsche model itself. It is hence quite unsatisfactory to have them integrated directly into the model.

The situation can also be worse, if for instance the Porsche model is available *via* a third-party library for which you don't have the source code, or even just a license letting you modifying it. In that case, it can be completely impossible to extend the original model in this naive way.

[1] `http://www.lrde.epita.fr/~didier/research/publis.php`
[2] See `http://en.wikipedia.org/wiki/Visitor_pattern` for instance.

2.1.2 Code duplication

The second problem with the naive approach lies in code duplication. This is particularly visible in the implementation of the `paint` and `recycle` methods for the `Body` class (see listing 1).

```
void paint ()
{
  /* paint the body */
  for (std::vector<Wheel>::iterator i = _wheels.begin ();
       i != _wheels.end ();
       i++)
    (*i).paint ();
};
void recycle ()
{
  /* recycle the body */
  for (std::vector<Wheel>::iterator i = _wheels.begin ();
       i != _wheels.end ();
       i++)
    (*i).recycle ();
};
```

Listing 1: Code Duplication

We can see that the code for these two methods is very similar. The similarity comes from the fact that both of these methods, after performing on the `Body` object itself, need to call the eponymous method on every `Wheel` object in turn. This will also be the case for every new action we might add to the Porsche model in the future. Another place where this phenomenon can be observed is in the Porsche object itself, where every action need to propagate to both the engine and the body.

It is worth mentioning that the aforementioned code duplication has nothing to do with the actions themselves, but only with our model *structure*: every action on a Porsche must propagate to the engine and the body, and in turn, every action on the body must propagate to the four wheels. The knowledge of this structure is however completely orthogonal to the actions that apply to it.

2.1.3 Summary

This first approach has exhibited two problems: first we would like to keep the original model read-only so that it can be easily extended with new actions, from the outside. In other words, we would like to be able to implement our `paint` and `recycle` actions *outside* of the target classes. Next, we would prefer to avoid code duplication, and in particular abstract away the portion related to the knowledge of our model structure. For instance, the fact that an action on the body must execute and then propagate to the four wheels should be expressed only once. Fortunately, the Visitor pattern is here to help us solving these two problems

2.2 Visitor-based implementation

A mostly complete solution based on the Visitor pattern in C++ is given in appendix B on page 22. A general explanation of its design follows. The Visitor pattern is articulated around two concepts: `visit` and `accept`. A "visit" is a process by which an action (hereafter called a *visitor*) can traverse an object and perform its task on it. On the other hand, an object must first "accept" a visitor before the visitor can perform its task. Accepting a visitor means first letting it perform on the object, and then possibly propagate it on the rest of the model structure.

2.2.1 Creating visitors

A visitor is represented by an object of some class. In our example, a Porsche visitor must know how to visit every Porsche component, which means that it must implement four `visit` methods taking respectively a `Porsche`, `Engine`, `Body` and `Wheel` object as argument. The `PaintPorscheVisitor` class is given in listing 2.

```
struct PaintPorscheVisitor : public PorscheVisitor
{
  virtual void visit (Wheel& wheel)   { /* paint the wheel  */ };
  virtual void visit (Body& body)     { /* paint the body   */ };
  virtual void visit (Engine& engine) { /* paint the engine */ };
  virtual void visit (Porsche& porsche) { /* paint the Porsche */ };
};
```

Listing 2: The `PaintPorscheVisitor` class

In C++, the fact that implementing a set of specific methods is mandatory can be represented by an abstract class providing "null" virtual methods and letting its subclasses defining them properly. Failure to implement one of the requested methods will result again in an abstract (uninstantiable) class. As a consequence we can implement every Porsche visitor class as a subclass of an abstract class named `PorscheVisitor`, as represented in listing 3 on the following page. This also explains the presence of the `virtual` keyword in the `PaintPorscheVisitor` methods definitions.

```
struct PorscheVisitor
{
  virtual void visit (Wheel&)   = 0;
  virtual void visit (Body&)    = 0;
  virtual void visit (Engine&) = 0;
  virtual void visit (Porsche&) = 0;
};
```

Listing 3: The `PorscheVisitor` abstract class

Providing a common base class for all Porsche visitors is cleaner because it is a way of ensuring that they conform to the requested interface, but it is also a necessity as we will soon discover.

2.2.2 Accepting visitors

Every Porsche component must now know how to "accept" visitors, which basically means calling their `visit` method. This is accomplished by equipping all such components with an `accept` method, as illustrated in listing 4.

```
struct Body : public VisitablePorscheComponent
{
  Body () { for (int i=0; i<4; i++) _wheels.push_back (Wheel()); };

  virtual void accept (PorscheVisitor& visitor)
  {
    visitor.visit (*this);
    for (std::vector<Wheel>::iterator i = _wheels.begin ();
         i != _wheels.end ();
         i++)
      i->accept (visitor);
  };

  std::vector<Wheel> _wheels;
};
```

Listing 4: The visitable `Body` class

We now also understand why the `PorscheVisitor` abstract class was necessary: the reason has to do with the statically typed nature of C++. In order for the `accept` methods to be definable and take any kind of Porsche visitor as argument, we need a common type for all such visitors, hence, a super-class for all of them.

In order to make sure that all Porsche components actually implement the `accept` method, we can define them as subclasses of a `VisitablePorscheComponent` abstract class, in the same way we defined the

PorscheVisitor abstract class. This is illustrated in listing 5 on the following page.

```
struct Wheel : public VisitablePorscheComponent
{
virtual void accept(PorscheVisitor& visitor){visitor.visit(*this);};
};
```

Listing 5: The VisitablePorscheComponent abstract class

2.2.3 Summary

As we can see, the Visitor approach solved both of our original problems: creating a new visitor involves subclassing the `PorscheVisitor` class and implementing the requested `visit` methods, all of this *outside* the original model, as in listing 2 on page 5. Moreover, the visitors don't require knowledge of the model structure in order to traverse it, as this knowledge is contained in the `accept` methods only (listing 4 on the preceding page) hence avoiding duplication of the traversal code in every new visitor.

3 The Visitor pattern in Lisp

We now turn to an implementation of the same example in Common Lisp [Ansi, 1994], and as before, we start by a naive implementation in Clos, the Common Lisp Object System [Bobrow et al., 1988, Keene, 1989].

3.1 Naive implementation

A mostly complete listing of a naive implementation in Lisp is given in appendix C on page 23. In this implementation, one Lisp class per component is created (`wheel`, `body`, `engine` and `porsche`) with calls to `defclass`. The `body` class contains a `wheels` *slot* (the equivalent of a structure/class *member* in C++) which is initialized to a list of four `wheel` objects.

A striking difference with the C++ version, however, is that there are no methods in the Lisp classes. Instead, the `paint` and `recycle` actions are implemented as so-called *generic functions* .

3.1.1 The generic function model

C++ features a *record-based* object model [Cardelli, 1988], meaning that methods belong to classes. In such object systems, the polymorphic dispatch depends only on one parameter: the class of the object through which the method is called (represented by the "hidden" pointer `this` in C++; sometimes referred to as `self` in other languages). CLOS, on the other hand, differs in two important ways.

1. Methods do not belong to classes (there is no privileged object receiving the messages). A polymorphic call *appears* in the code like an ordinary function call. Functions whose behavior is provided by such methods are called *generic functions* .

2. Another important difference, although we are not going to need it in our particular example, is that CLOS supports *multi-methods*, that is, polymorphic dispatch based on *any* number of arguments (not only the first one).

```
(defgeneric paint (object)
  (:method ((porsche porsche))
    #| paint the Porsche |#
    (paint (engine porsche))
    (paint (body porsche)))
  (:method ((engine engine))
    #| paint the engine |#)
  (:method ((body body))
    #| paint the body |#
    (dolist (wheel (wheels body))
      (paint wheel)))
  (:method ((wheel wheel))
    #| paint the wheel |#))
```

Listing 6: The `paint` generic function

To clarify this, let us look at the code for the `paint` action in listing 6. A generic function `paint` is defined by a call to `defgeneric`. Four *specializations* of the generic function are subsequently provided by passing a `:method` option to the `defgeneric` call. We could have achieved the same effect by using calls to `defmethod` outside the call to `defgeneric`. As you can see, a particular syntax allows us to specify the expected class of the method's argument. The effect of our `defgeneric` call is in fact to provide four different versions of the `paint` action: one that will be used when the argument is a `porsche` object, one that will be used when the argument is an `engine` object etc.

Painting the Porsche now involves calling the generic function just as one would call an ordinary function: by passing a `porsche` object to it:

```
(paint porsche)
```

According to class of the argument, the proper method is used (the first one in this example).

3.1.2 First problem solved

Recall that the naive C++ implementation exhibited two problems, the first one being the need to modify the original classes for each new action we wanted to implement. It is worth mentioning that this problem is already a non-issue in our naive Lisp version. Because CLOS is not a record-based object system, our methods are already outside the original classes, by language design. The Porsche model is hence already accessed in a read-only fashion.

The second problem (code duplication) however, remains unsolved. This is visible when you compare the implementations of the `paint` and `recycle` methods specialized for the `body` class, as illustrated in listing 7. It is clear that we are still duplicating the model structure traversal process across our different actions.

```
(:method ((body body))
  #| paint the body |#
  (dolist (wheel (wheels body))
    (paint wheel)))
```

```
(:method ((body body))
  #| recycle the body |#
  (dolist (wheel (wheels body))
    (recycle wheel)))
```

Listing 7: Code Duplication

3.2 Visitor-based implementation

Although one of our original problems is already solved, we might think of using the Visitor pattern to help us with the second one, since it was so successful in the C++ case. A brute-force application of the Visitor pattern in the Lisp case is given in appendix D on page 24. The author has actually *seen* code like this in use. In the remainder of this section, we show how bad this solution is, and we clean it up step-by-step, by using a series of features from the Lisp language.

3.2.1 Step 1: getting rid of static typing idiosyncrasies

3.2.1.1 Visitable Porsche components

Even the reader unfamiliar with Lisp should immediately notice something wrong in this picture. We have an empty `visitable-porsche-component` class that serves as a super-class for every Porsche component, but which is actually not used anywhere.

This class was originally an abstract class designed to make sure that every Porsche component implemented its own version of the `accept` method. Even in the case of C++, this "safety precaution" is not very robust, and in fact not necessary (it is however considered good style).

– Not very robust because it requires that every Porsche component be made a subclass of `VisitablePorscheComponent` explicitly, which the programmer could forget.

– Not necessary because even if subclassing is omitted, and the implementation of `accept` is forgotten, the code would not compile. For instance, if there were no `accept` method in the `Wheel` class, compilation would break on the `Body` class, where its `accept` method is attempting to call the (missing) `Wheel` one (see listing 4 on page 6).

In the Lisp case the concern expressed by the `VisitablePorscheComponent` class is much less relevant because the `accept` generic function, along with its methods, is located *outside* the classes in question, so it makes no sense to check for their implementation within the original classes. The initial concern might stand however: it is legitimate to wonder if it is possible to ensure that there is a specialized `accept` method for every Porsche component. Our answer to that question is: yes, it is possible, but it is also most of the time undesirable.

– **It is possible.** Clos itself is written on top of a *Meta Object Protocol*, simply known as the Clos Mop [Paepcke, 1993, Kiczales et al., 1991]. Although not part of the Ansi specification, the Clos Mop is a *de facto* standard well supported by many Common Lisp implementations. The Mop is designed with behavioral reflection in mind [Maes, 1987, Smith, 1984] and provides considerable introspection (examining behavior) and intercession (modifying behavior) capabilities to Clos. As a consequence, it is possible to introspect on the `accept` generic function and check if such or such specialization exists. For an in-depth discussion on implementing safety protocols around generic functions, see [Verna, 2008].

– **It is also undesirable.** One of the attractive points of Lisp in a development cycle is its "fully dynamic" nature. In Clos for instance, one can add or remove methods to generic functions while the program is running. More generally, Lisp dialects provide a "read-eval-print" loop (Repl for short) in which you have full control over your lisp environment, including the ability run, compile, interpret or even modify any part of your program interactively. This also means that it is possible to run parts of an incomplete program, and explains why the development, testing and debugging cycles are usually completely intermixed. For that reason, early checks or safety measures usually get in the way instead of being helpful, and are considered undesirable most of the time.

In order to comply with the fully dynamic tradition of Lisp, it is hence better to simply get rid of the `visitable-porsche-component` class altogether.

3.2.1.2 The `porsche-visitor` class

Along with the same lines, we can also remove the `porsche-visitor` class in the Lisp case. Recall that this class was *necessary* in C++ because of static typing, for correct prototyping of the `accept` methods. In the Lisp case, the spurious nature of the `porsche-visitor` class (which, by the way, is empty) becomes apparent when we confront corresponding `accept` and `visit` methods.

```
  (:method ((wheel wheel) (visitor porsche-visitor))
    (visit wheel visitor))
(defmethod visit ((wheel wheel) (visitor paint-porsche-visitor))
  #| paint the wheel |#)
```

Listing 8: The spurious `porsche-visitor` class

Listing 8 shows the specialization of the `accept` method for the `wheel` class. The `visitor` argument is specialized as a `porsche-visitor`, which is only a super-class of the object's actual class, and the sole purpose of this method is to call the `visit` generic function. Now, in order to select the appropriate behavior, every `visit` method needs to be specialized on the *actual* class of the visitor object, as exhibited in the same listing. We now understand that the `porsche-visitor` class is effectively unused and unnecessary: as long as the `visit` methods are properly specialized, we can leave the `visitor` argument of the `accept` methods completely unspecialized.

3.2.2 Summary

By respecting the dynamic nature of Lisp and getting rid of static typing idiosyncrasies, we have been able to clean up the code by removing spurious and unnecessary abstractions. This results in a thinned version of our Porsche example, demonstrated in appendix E on page 24. The `porsche`, `body`, `engine` and `wheel` classes are back to their original state, that is, exactly as in the naive approach of appendix C on page 23, so they are not redisplayed here. Only the thinned versions of the `accept` and `paint` generic functions are shown.

3.3 Step 2: Getting rid of visitor objects

At this point, it is usually less obvious for the reader unfamiliar with Lisp to figure out how we can further improve our implementation. The key feature of Lisp that we are going to use in this second step is the fact that Lisp is a functional language.

3.3.1 First-class citizens

The term “first-class citizen” was originally used by Christopher Strachey [Burstall, 2000] in order to informally reference programming languages entities that can be stored in variables, aggregated in structures, given as argument to functions or returned by them *etc.*. The interesting thing is that this term originally appeared in the context of first-class (or higher-order) *functions* which is exactly what we are going to use.

Lisp, as a functional language, supports first-class functions, including generic ones. The precise feature which is of interest to us is that generic, first-class functions can be passed as arguments to other functions. With this in mind, it will soon become clear that we don’t need visitor *objects* or *classes*; only visitor *functions*.

3.3.2 Introducing visitor functions

Listing 9 illustrates how dispatching to the appropriate `visit` method works in our current implementation. Each visitor is an instance of a particular class (in our example, `paint-porsche-visitor`). This visitor is passed to the `accept` method (here, the `wheel` one) which has no use for it, besides propagating it to the `visit` generic function. This is one indirection too many. Indeed, we can see that the existence of a visitor class (which happens to be empty) is justified *only* to select the appropriate `visit` method (in our case, visiting a `wheel` object with the `paint` action).

```
(defgeneric accept (object visitor)
  (:method ((wheel wheel) visitor)
    (visit wheel visitor)))
(defclass paint-porsche-visitor () ())
(defmethod visit ((wheel wheel) (visitor paint-porsche-visitor))
  #| paint the wheel |#)
```

Listing 9: The use for visitor classes

From a functional point of view, this kind of code is evidently poorly designed. A much better style is to provide `accept` not with a visitor object, but with a visitor *function*, so that it is applied directly to the concerned component. We are then led to define `paint` and `recycle` generic functions with appropriate methods for every Porsche component, as illustrated in listing 10 on the following page.

Accepting the visitor on a `wheel` object now becomes simply a matter of calling the visitor function on the object itself, as shown in listing 11 on the

```
(defgeneric paint (object)
  (:method ((wheel wheel))
    #| paint the wheel |#)
  (:method ((body body))
    #| paint the body |#)
  (:method ((engine engine))
    #| paint the engine |#)
  (:method ((porsche porsche))
    #| paint the Porsche |#))
```

Listing 10: The `paint` visitor generic function

next page. For compound objects like the Porsche itself or its body, accepting the visitor means calling the visitor function, and also accepting it on the sub-components. On the other hand, "terminal" components like the wheels or the engine need only call the visitor function. This would lead to code duplication because their `accept` methods are all the same. In order to avoid that, we can provide a completely unspecialized, default `accept` method, performing the same operation as the one in listing 11, only without the need to copy it for every terminal component.

```
(defgeneric accept (object visitor-function)
  (:method ((object wheel) visitor-function)
    (funcall visitor-function object)))
```

Listing 11: The wheel `accept` method

3.3.3 Summary

After step 1, we ended up with empty classes for typing visitors, the purpose of which was only to select the appropriate `visit` method. We saw that thanks to the functional nature of Lisp, we don't need to perform dispatching based on the visitors classes, as we can directly pass a visitor *function* to the `accept` methods. The final result of this step is given in appendix F on page 25 (only the modified parts appear in this listing). Painting a Porsche now simply becomes a matter of calling its `accept` method, passing it the `paint` generic function, as demonstrated below (the `#'` syntax is a shortcut to retrieve the function object denoted by the `paint` symbol):

`(accept porsche #'paint)`

As a side note, we now also feel the urge to rename the `accept` generic function as "visit", because writing `(visit porsche #'paint)` is in fact much more readable.

Although Common Lisp is not the only functional language around, one specific feature that we have used here is that generic functions are first-class objects, just as ordinary functions. As a consequence, even from a functional point of view, it is not completely trivial that we are able to pass `paint` and `recycle` as arguments to the `accept` function.

3.4 Step 3: explicit mapping

A this point, a reader well acquainted with the functional programming paradigm would still remain unsatisfied. The reason is that our current solution makes a silent use of *mapping*, a very important concept in functional languages. This concept should hence appear much more explicitly in the code.

3.4.1 Functional mapping

Mapping is the process of applying a particular treatment to every element of a data structure. This idiom is probably the most important one in the functional programming paradigm because it can be implemented thanks to first-class functions. In fact, it is so important that mapping was actually the very first example of higher-order function exhibited by John McCarthy in his original paper on Lisp [MacCarthy, 1960]. For example, the Common Lisp function `mapcar` applies a function to every element of a list, and returns the new list as a result:

```
(mapcar #'sqrt '(4 16 256)) => (2 4 16)
```

With this in mind, we need to realize that the `accept` generic function actually performs a kind of *structural mapping*: it applies a (visitor) function to every element of the Porsche structure. As a consequence, we should make this fact explicit in the code.

3.4.2 Structural mapping with `mapobject`

We propose to replace the `accept` generic function with `mapobject`, a generic function designed along the lines of the other mapping facilities of Common Lisp. The `mapobject` generic function is given in listing 12 on the following page. It features an unspecialized method that only applies the function to the object. In our case, this will be useful for every terminal Porsche component (engine and wheels). For non terminal components, the mapping needs to be propagated to the sub-components as well, hence the two other methods. Notice that compared to the `accept` generic function, the order of the arguments is reversed, to remain conformant with the other Common Lisp mapping functions.

```
(defgeneric mapobject (func object)
  (:method (func object)
    (funcall func object))
  (:method (func (body body))
    (funcall func body)
    (dolist (wheel (wheels body))
      (mapobject func wheel)))
  (:method (func (porsche porsche))
    (funcall func porsche)
    (mapobject func (engine porsche))
    (mapobject func (body porsche))))
```

Listing 12: The `mapobject` generic function

Note that compared to step 2, we did not provide any new functionality, nor did we use other features of Lisp. We simply made the concept of structural mapping more explicit. Our next refinement, however, will make use of specific features of CLOS.

3.5 Step 4: generic mapping

Step 3 is satisfactory from a functional point of view. A programmer well acquainted with the Common Lisp object system, however, may feel the need for an even deeper refinement.

The main characteristic of step 3 is that we need to provide a specialized `mapobject` method for every non-terminal component (the Porsche itself and the Porsche's body in our case). However, all these methods actually do the same thing in essence: they call the mapping function on the object, and then propagate on every slot in the object. The Porsche maps the function on its engine and body, and the Porsche's body maps the function on its four wheels. In some way, this feels very much like code duplication again: couldn't we define a *single* `mapobject` method that works on any CLOS object by calling the function on it, and propagating on every slot?

We already introduced the CLOS MOP in section 3.2.1 on page 9, along with its introspective capabilities. It is now time to make use of them. Please note that Common Lisp is not the only language with reflective capabilities. Younger languages like Java or C# have similar features (there is even one weakness on the side of Common Lisp, which is that the CLOS MOP is only a *de facto* standard).

Listing 13 shows a `mapobject` method that does exactly what we want. This method is specialized on `standard-object`, which is the default class for CLOS objects. The first thing it does it to call the mapping function on the object. The rest deserves a bit more explanation.

The CLOS MOP provides a function called `class-direct-slots` that allows one to dynamically retrieve a list of direct slot definitions from any class. The

```
(defgeneric mapobject (func object)
  (:method (func (object standard-object))
    (funcall func  object)
    (mapc
     (lambda (slot)
       (mapobject func (slot-value object (slot-definition-name slot))))
     (class-direct-slots (class-of object)))))
```

Listing 13: Generic mapping

class of an object is returned by the function `class-of`. Finally, given a slot name, the value of that slot in a particular object can be retrieved by calling `slot-value`. The only thing left to do is map the generic function `mapobject` itself on the list of retrieved slot values in the object.

Note that the version of `mapobject` presented here is simple and convenient for our current example. However, many improvements or alternatives may be thought of. For instance, we may want to map on *all* slots (not only direct ones) in the class, including those inherited from superclasses. We may also want to filter out "unmappable" slots, maybe by defining a special `mapable` superclass for the selected ones, *etc.*

In order to have a completely functional new version of our example, we need one last small addition to the `mapobject` generic function. In the `body` class, the `wheels` slot contains a *list* of four `wheel` objects. Lists are not "standard CLOS objects" so we need to explain to `mapobjects` how to handle them. This is done by providing a new method, specialized on `list` objects, that consists in applying the mapping function on every list element, as shown in listing listing 14. Common Lisp already provides a function `mapc` for doing that.

```
  (:method (func (object list))
    (mapc func object)))
```

Listing 14: `mapobject` method for lists

As before, "visiting" a Porsche with the `paint` generic function is written like this: `(mapobject #'paint porsche)`, and as before, the visitor generic functions don't need to be modified in any way (neither does the original Porsche components). It is also worth pointing out that with only 9 lines of code, we have implemented a "universal" mapping facility that works on all standard CLOS objects, and on Lisp lists. We call it universal because it is now completely orthogonal to the Porsche model, and can be used on any other problem. If needed elsewhere, adding new `mapobject` methods for other native Lisp data types (vectors for instance) is also straightforward.

4 Visiting with state

In the previous section, we have demonstrated how a brute-force use of the Visitor pattern can be "cleaned up", to the point where the only actual trace of it is in the `mapobject` generic function, which is 9 lines of code long. In this section, we are going to continue our exploration by introducing state to our visitors.

Suppose we want to implement a "component counter" visitor, returning the number of traversed objects. Calling `(mapobject #'count porsche)` would then return 7 (a Porsche, an engine, a body and four wheels). A solution to this problem in the C++ case is left as an exercise, although it would admittedly not be very complicated: since visitors are *objects* in a C++ context, it is easy to provide a `counter` member in the visitor's class, and have the `visit` methods increment it.

In the Lisp case, this problem does not seem to fit well with visitor *functions*: how can function calls retain state? We could think of using a global `counter` variable somewhere and have the methods in the `count` generic function increment it. This is obviously very unsatisfactory from a software engineering point of view. We could also get back to objects (step 1, appendix E on page 24) and add state to them, just as in the case of C++, but this again seems wrong because one of our first refinements was precisely to get rid of visitor objects.

Fortunately, a better solution is made available to us, thanks to another feature called "lexical closures" (available in Lisp but also in functional langages in general, and even in some others like C#).

4.1 Lexical closures

Lexical closures can be seen as a way to encapsulate state with behavior, only without the object-oriented machinery. They are particularly useful in conjunction with anonymous functions (or lambda expressions), another characteristic of first-class functions.

```
(defun make-adder (n)
  (lambda (x) (+ n x)))

(funcall (make-adder 3) 5)
;; => 8
```

```
(let ((count 0))
  (defun increment () (incf count)))

(increment) ;; => 1
(increment) ;; => 2
;;...
```

Listing 15: Lexical closures examples

A typical example of a lexical closure is a `make-adder` function, written in Lisp as depicted on the left part of listing listing 15. This function creates "$+n$" functions, n being its argument. The interesting thing here is that the variable `n` is free in the created anonymous function: it refers to the `n` given as an argument to `make-adder`.

Lisp being an *impure* functional language (that is, where side effects are authorized), state encapsulated in a lexical closure is mutable. An example of this would be an `increment` function updating a counter at every call, as shown in the right part of the same listing.

4.1.1 Visiting with a lexical closure

It should now be clear that a lexical closure with mutable state is the solution we are looking for in order to implement the `count` visitor. This solution is presented in listing 16. In addition to the Porsche object, the `let` form introduces a mutable

```
(let ((porsche (make-instance 'porsche))
      (counter 0))
  (mapobject (lambda (obj)
               (declare (ignore obj))
               (incf counter))
             porsche)
  (format t "The Porsche has ~S components.~%" counter))
```

Listing 16: A counting visitor

`counter` initialized to 0. The visitor function used in the call to `mapobject` is in fact an anonymous function taking an object as argument and ignoring it. Its sole purpose is however to increment the `counter` it lexically refers to every time it is called.

There are two specificities worth emphasizing in this approach.

1. We do not need to create a `count` visitor function explicitly. The function is only created dynamically, locally (in `mapobject`) and anonymously when it is needed. This is made possible by the first-class status of functions in Lisp.

2. Contrary to the `paint` and `recycle` visitors, this one is not a generic function; only a standard (anonymous) one. A generic function would be overkill here, because the counting process is the same for every object, so there is no need for specialization. As generic functions *are* in fact functions, `mapobject` works equally well on generic and standard functions.

5 Conclusion

In this paper, we have examined one of the most well known design patterns, the Visitor pattern, from the perspective of Common Lisp, a dynamic, functional language featuring a very powerful object-oriented layer. After grounding our analysis in a C++ example where the pattern is indeed very useful, we have demonstrated that the same idea applied to Lisp takes a very different shape, when done correctly.

This demonstration was done in a somewhat unconventional, but we believe quite educational way: we started with a brutal application of the pattern, as described in the GoF book, and gradually cleaned up the code by introducing a set of specific features from the target language.

- Thanks to the Clos generic function model disconnecting methods from classes, one of the two original problems (read-only access to the visited model) is a non-issue.
- Thanks to the dynamic nature of Lisp, we don't need to base our visitor or model objects on abstract classes, which is a static-typing idiosyncrasy.
- But in fact, thanks to the functional nature of Lisp, we don't need visitor objects. We only need visitor (first-class) *functions*.
- In addition to that, the concept of "visit" is identified as a particular form of *mapping*, something well known in the functional languages domain.
- Thanks to the introspection capabilities of the Clos Mop, we can provide a universal mapping facility that becomes completely orthogonal to the target model.
- Thanks to lexical closures, it is possible to implement stateful visitors (like the component counter) without resorting back to the object-oriented machinery.

It should also be noted that the essence of the visitor pattern in Lisp is encapsulated into a generic function (`mapobject`) which, in 9 lines of code, is already much more general than what we provided in the C++ case. Does this mean that the Visitor pattern is useless? Far from it. It simply means that this pattern is natively supported at different levels by different languages, which brings us to what we call the "iceberg metaphor", depicted in figure 1 on the next page. A software design pattern is like an iceberg. There is always a part immersed into the language: this is the part you shouldn't need to write because the language directly supports it. There is also a part emerged into the application: this is the part you need to write on top of what the language supports. Depending on the language's expressiveness, the immersed part may be more

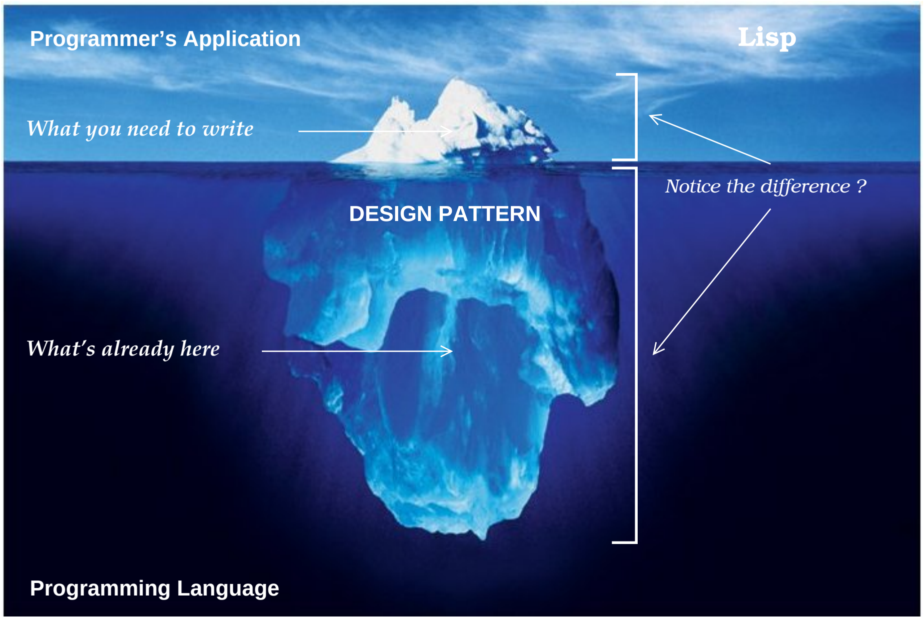

Figure 1: The Iceberg Metaphor

or less important, but the pattern itself will always be there. The lesson to be learned is that software design patterns should be used with care because the literature is often too language-specific, and usually fails in describing them in a sufficiently abstract way. In particular, design patterns will never replace an in-depth knowledge of your preferred language. By using patterns blindly in a language that already supports them, your risk missing the obvious and most of the time simpler solution: the "Just Do It" pattern.

References

[Bobrow et al., 1988] Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. (1988). Common lisp object system specification. *ACM SIGPLAN Notices*, 23(SI):1–142.

[Burstall, 2000] Burstall, R. (2000). Christopher Strachey — understanding programming languages. *Higher Order Symbolic Computation*, 13(1-2):51–55.

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., P.Sommerlad, and Stal, M. (1996). *Pattern-Oriented Software Architecture*. Wiley.

[C++, 1998] C++ (1998). International Standard: Programming Language – C++. ISO/IEC 14882:1998(E).

[Cardelli, 1988] Cardelli, L. (1988). A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164. A revised version of the paper that appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pages 51–66.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[Keene, 1989] Keene, S. E. (1989). *Object-Oriented Programming in Common Lisp: a Programmer's Guide to* CLOS. Addison-Wesley.

[Kiczales et al., 1991] Kiczales, G. J., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA.

[MacCarthy, 1960] MacCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3:184–195. Online version at `http://www-formal.stanford.edu/jmc/recursive.html`.

[Maes, 1987] Maes, P. (1987). Concepts and experiments in computational reflection. In *OOPSLA*. ACM.

[Norvig, 1996] Norvig, P. (1996). Tutorial on design patterns in dynamic programming. In *Object World Conference*.

[Paepcke, 1993] Paepcke, A. (1993). User-level language crafting – introducing the CLOS metaobject protocol. In Paepcke, A., editor, *Object-Oriented Programming: The CLOS Perspective*, chapter 3, pages 65–99. MIT Press. Downloadable version at `http://infolab.stanford.edu/~paepcke/shared-documents/mopintro.ps`.

[Smith, 1984] Smith, B. C. (1984). Reflection and semantics in lisp. In *Symposium on Principles of Programming Languages*, pages 23–35. ACM.

[ANSI, 1994] ANSI (1994). American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999).

[Verna, 2008] Verna, D. (2008). Binary methods programming: the CLOS perspective. *Journal of Universal Computer Science*, 14(20):3389–3411. `http://www.jucs.org/jucs_14_20/binary_methods_programming_the`.

A C++ raw version

```
struct Wheel
{
  void paint   () { /* paint the wheel   */ };
  void recycle () { /* recycle the wheel */ };
};

struct Body
{
  Body () { for (int i=0; i<4; i++) _wheels.push_back (Wheel()); };

  void paint ()
  {
    /* paint the body */
    for (std::vector<Wheel>::iterator i = _wheels.begin ();
         i != _wheels.end ();
         i++)
      (*i).paint ();
  };
  void recycle ()
  {
    /* recycle the body */
    for (std::vector<Wheel>::iterator i = _wheels.begin ();
         i != _wheels.end ();
         i++)
      (*i).recycle ();
  };

  std::vector<Wheel> _wheels;
};

struct Engine
{
  void paint   () { /* paint the engine   */ };
  void recycle () { /* recycle the engine */ };
```

```
};

struct Porsche
{
  void paint ()
  {
    /* paint the Porsche */
    _engine.paint ();
    _body.paint ();
  };
  void recycle ()
  {
    /* recycle the Porsche */
    _engine.recycle ();
    _body.recycle ();
  };

  Body   _body;
  Engine _engine;
};


int main (int argc, char *argv[])
{
  Porsche porsche;

  porsche.paint   ();
  porsche.recycle ();
}
```

B C++ visitor version

```
struct Wheel;
struct Body;
struct Engine;
struct Porsche;

struct PorscheVisitor
{
  virtual void visit (Wheel&)   = 0;
  virtual void visit (Body&)    = 0;
  virtual void visit (Engine&)  = 0;
  virtual void visit (Porsche&) = 0;
};

struct VisitablePorscheComponent
{
  virtual void accept (PorscheVisitor&) = 0;
};

struct Wheel : public VisitablePorscheComponent
{
virtual void accept(PorscheVisitor& visitor){visitor.visit(*this);};
};

struct Body : public VisitablePorscheComponent
{
  Body () { for (int i=0; i<4; i++) _wheels.push_back (Wheel()); };

  virtual void accept (PorscheVisitor& visitor)
  {
    visitor.visit (*this);
    for (std::vector<Wheel>::iterator i = _wheels.begin ();
         i != _wheels.end ();
         i++)
      i->accept (visitor);
  };

  std::vector<Wheel> _wheels;
```

```
};

struct Engine : public VisitablePorscheComponent
{
virtual void accept(PorscheVisitor& visitor){visitor.visit(*this);};
};

struct Porsche : public VisitablePorscheComponent
{
  virtual void accept (PorscheVisitor& visitor)
  {
    visitor.visit (*this);
    _engine.accept (visitor);
    _body.accept   (visitor);
  };

  Body   _body;
  Engine _engine;
};

struct PaintPorscheVisitor : public PorscheVisitor
{
  virtual void visit (Wheel& wheel)   { /* paint the wheel   */ };
  virtual void visit (Body& body)     { /* paint the body    */ };
  virtual void visit (Engine& engine) { /* paint the engine */ };
  virtual void visit (Porsche& porsche) { /* paint the Porsche */ };
};

// Do the same for RecyclePorscheVisitor

int main (int argc, char *argv[])
{
  Porsche porsche;

  PaintPorscheVisitor painter;
  porsche.accept (painter);

  RecyclePorscheVisitor recycler;
  porsche.accept (recycler);
}
```

C Lisp raw version

```
(defclass wheel () ())

(defclass body ()
  ((wheels :initform (loop :for i :from 1 :upto 4
                           :for wheel = (make-instance 'wheel)
                           :collect wheel)
           :accessor wheels)))

(defclass engine () ())

(defclass porsche ()
  ((engine :initform (make-instance 'engine)
           :accessor engine)
   (body :initform (make-instance 'body)
         :accessor body)))

(defgeneric paint (object)
  (:method ((porsche porsche))
    #| paint the Porsche |#
    (paint (engine porsche))
    (paint (body porsche)))
  (:method ((engine engine))
    #| paint the engine |#)
  (:method ((body body))
    #| paint the body |#
    (dolist (wheel (wheels body))
```

```
      (paint wheel)))
  (:method ((wheel wheel))
    #| paint the wheel |#))

;; Do the same for recycle

(defun main ()
  (let ((porsche (make-instance 'porsche)))
    (paint porsche)
    (recycle porsche))
  (quit))
```

D Lisp visitor version

```
(defclass visitable-porsche-component () ())

(defclass wheel (visitable-porsche-component) ())

(defclass body (visitable-porsche-component)
  ((wheels :initform (loop :for i :from 1 :upto 4
                           :for wheel = (make-instance 'wheel)
                           :collect wheel)
           :accessor wheels)))

(defclass engine (visitable-porsche-component)
  ())

(defclass porsche (visitable-porsche-component)
  ((engine :initform (make-instance 'engine)
           :accessor engine)
   (body :initform (make-instance 'body)
         :accessor body)))

(defclass porsche-visitor () ())

(defgeneric accept (object visitor)
  (:method ((wheel wheel) (visitor porsche-visitor))
    (visit wheel visitor))
  (:method ((body body) (visitor porsche-visitor))
    (visit body visitor)
    (dolist (wheel (wheels body))
      (accept wheel visitor)))
  (:method ((engine engine) (visitor porsche-visitor))
    (visit engine visitor))
  (:method ((porsche porsche) (visitor porsche-visitor))
    (visit porsche visitor)
    (accept (engine porsche) visitor)
    (accept (body porsche) visitor)))

(defgeneric visit (object visitor))

(defclass paint-porsche-visitor (porsche-visitor) ())
(defmethod visit ((wheel wheel) (visitor paint-porsche-visitor))
  #| paint the wheel |#)
(defmethod visit ((body body) (visitor paint-porsche-visitor))
  #| paint the body |#)
(defmethod visit ((engine engine) (visitor paint-porsche-visitor))
  #| paint the engine |#)
(defmethod visit ((porsche porsche) (visitor paint-porsche-visitor))
  #| paint the Porsche |#)

;; Do the same for recycle-porsche-visitor

(defun main ()
  (let ((porsche (make-instance 'porsche)))
    (accept porsche (make-instance 'paint-porsche-visitor))
    (accept porsche (make-instance 'recycle-porsche-visitor)))
  (quit))
```

E Lisp thinned visitor version

```lisp
(defgeneric accept (object visitor)
  (:method ((wheel wheel) visitor)
    (visit wheel visitor))
  (:method ((body body) visitor)
    (visit body visitor)
    (dolist (wheel (wheels body))
      (accept wheel visitor)))
  (:method ((engine engine) visitor)
    (visit engine visitor))
  (:method ((porsche porsche) visitor)
    (visit porsche visitor)
    (accept (engine porsche) visitor)
    (accept (body porsche) visitor)))

(defclass paint-porsche-visitor () ())
(defmethod visit ((wheel wheel) (visitor paint-porsche-visitor))
  #| paint the wheel |#)
(defmethod visit ((body body) (visitor paint-porsche-visitor))
  #| paint the body |#)
(defmethod visit ((engine engine) (visitor paint-porsche-visitor))
  #| paint the engine |#)
(defmethod visit ((porsche porsche) (visitor paint-porsche-visitor))
  #| paint the Porsche |#)

;; Do the same for recycle-porsche-visitor
```

F Lisp visitor functions

```lisp
(defgeneric accept (object visitor-function)
  (:method (object visitor-function)
    (funcall visitor-function object))
  (:method ((body body) visitor-function)
    (funcall visitor-function body)
    (dolist (wheel (wheels body))
      (accept wheel visitor-function)))
  (:method ((porsche porsche) visitor-function)
    (funcall visitor-function porsche)
    (accept (engine porsche) visitor-function)
    (accept (body porsche) visitor-function)))

(defgeneric paint (object)
  (:method ((wheel wheel))
    #| paint the wheel |#)
  (:method ((body body))
    #| paint the body |#)
  (:method ((engine engine))
    #| paint the engine |#)
  (:method ((porsche porsche))
    #| paint the Porsche |#))

;; Do the same for recycle

(defun main ()
  (let ((porsche (make-instance 'porsche)))
    (accept porsche #'paint)
    (accept porsche #'recycle))
  (quit))
```