Traits, Classes and Instances in the Lisp Package

A. Motzek

Abstract
The Lisp Package is a Lisp interpreter / compiler written in Java. It can be embedded into Java applications when scripting is needed [1]. The Lisp Package implements a Lisp-1 [2] but otherwise tries to conform to Common Lisp, for example regarding the names of predefined functions and macros. Other features are borrowed from programming languages like Ruby or Go or were found in the literature. This paper describes the object-oriented features of the Lisp Package.

1. Introduction
The design spectrum of object-oriented programming languages is large. Two interesting points in this spectrum are Common Lisp (CLOS) [3] and Ruby [4]. Even if these languages are not at opposite ends of the spectrum they differ in many design decisions.

Whereas CLOS class definitions also specify the instance variables (called slots) the class definitions of Ruby don't do this. Instance variables are created when they are used within methods.

Ruby uses single dispatch. This has the consequence that methods are defined within classes and are bound to these. CLOS allows multiple dispatch and methods with similar signatures are grouped together in so called generic methods.

Multiple inheritance is supported by CLOS. Ruby classes have only one superclass (with the exception of the root of the class hierarchy, which has none) but Ruby allows to mix in additional functions from multiple modules.

The diamond problem of repeated inheritance [5] is solved via linearization with class precedence lists in CLOS and some rules in Ruby: a definition in a class has precedence over a definition in an included module, the module included last wins. Another possible solution to the diamond problem is the use of traits [6, 6a], which are not implemented in either CLOS or Ruby.

The implementation of the object-oriented features in the Lisp Package is influenced by CLOS, Ruby and traits. It is described in the following chapter.

2. Approach

This chapter describes the design decisions regarding the object-oriented features of the Lisp Package [7].

2.1 Classes

Classes are used within instance creation (see 2.2) and for so called specializers in the definition of methods (see 2.4). Classes do not define any inner structure for their instances (e.g. instance variables or slots). Classes themselves are not instances of any class and there are also no classes for other built-in types (e.g. symbols, lists or numbers). Classes can have none, one or multiple superclasses.

Classes are defined with the special function **defclass** (→ Common Lisp has defclass but there a class defines the slots of its instances and more). It takes two arguments that will not be evaluated. The first argument must be a symbol that gives the name of the class. The second argument is a possibly empty list of symbols that are the names of the superclasses.

Superclasses must be defined before they can be used. This makes sure that the dependency graph of classes has no cycles. The list of superclasses must not include a class and one of its direct or indirect superclasses as this could cause surprising effects during the calculation of the most specific method when a generic function is invoked (see 2.4). An additional restriction applies when using traits (see 2.3).

As a side effect defclass binds the created class to its name. Therefore the name must be either unbound or bound to nil before.

The list of direct superclasses of a class can be queried with the function **direct-superclasses** (→ Common Lisp). It takes one argument that will be evaluated.

The function **class?** (→ classes are embedded in the type hierarchy in Common Lisp so class-of can be used) checks if it's argument is a class.

Each call to defclass creates a new class even if the supplied class name might repeat. If **equal?** (→ both Common Lisp and Ruby have multiple functions, methods or operators for testing equality) is applied to two classes it will only return t if both arguments refer to the same class.

2.2 Instances

Instances belong to a class and contain instance variables called slots. Classes don't specify which slots their instances can have. Slots are created within an instance when a value gets assigned to them.

Instances are created with the function **allocate-instance** (→ Common Lisp). It takes one argument which must be a class. The result of a call to allocate-instance is an instance of the class that was provided as argument.

This can be checked with the function **class-of** (→ Common Lisp). This function takes one argument. The argument must be an instance. The result of applying class-of to an instance is the class that was provided to allocate-instance when the instance was created.

Instances can be frozen. A frozen instance is immutable. That means no additional slots can be added to the instance and the values of the already existing slots cannot be changed anymore. Freezing an instance is done with the function **freeze** (→ Ruby method). It accepts an instance as argument, freezes it and returns it as result. The function **frozen?** (→ Ruby method) can be used to check if an instance is frozen.

It is not possible to unfreeze an instance but a copy of the instance can be created with **duplicate** (→ Ruby method dup). The copy is not frozen, even if its original was frozen. The duplicate is of the same class and has the same values for the slots as the original. Modifying the copy does not change the original.

The function **change-class** (→ Common Lisp does not create a copy but modifies the original object) takes two arguments: an instance and a class. It creates a new instance of the given class and copies the slots from the provided instance. Modifying the freshly created instance does not affect the argument instance.

The value of a slot can be read with the function **slot-value** (→ Common Lisp). The function takes two arguments: an instance and a symbol that is the name of the slot. The result is the value assigned to the given slot in the given instance.

Assigning values to slots can be done with the function **assign** (→ Common Lisp uses setf with slot-value for this, only slots defined with defclass can be used). The function takes three arguments: an instance, a symbol for the name of the slot and the new value. If the slot does not exist for the given instance, it will be created.

Checking if a value is an instance can be done with **instance?** (→ instances are embedded in the type hierarchy in Common Lisp).

Equality of instances follows the concepts described in [8]: if an instance is mutable (that means not frozen) it is only equal to itself. For two frozen (immutable) instances **equal?** returns t if the two instances belong to the same class, have the same count of slots, have the same slot names and have equal values for corresponding slots.

2.3 Traits

Traits [6] are similar to classes but no instances can be created from traits. Traits can only be used as supertraits in the definition of new traits or new classes and as specializers in the definition of methods (see 2.4).

Traits are defined with the special function **deftrait** (→ the most similar concepts are the mixin in Common Lisp and the inclusion of modules in Ruby). It takes two arguments that will not be evaluated. The first argument must be a symbol that gives the name of the trait. The second argument is a possibly empty list of symbols that are the names of the supertraits. Class names are not allowed in this list, traits can only inherit from other traits. In defclass both class names and trait names are allowed.

For supertraits in deftrait the same restrictions apply as for superclasses in defclass: Supertraits must be defined before they can be used. The list of supertraits must not include a trait and one of its direct or indirect supertraits.

When using traits in defclass it is additionally checked that independent direct or indirect supertraits of the class do not conflict. Two traits are assumed independent if not one is the direct or indirect supertrait of the other. Two independent traits are in conflict if it is possible to find arguments that invoke methods in both traits for the same generic function. This check makes sure that independent traits do not overlap in provided functionality and thus can be combined without problems and in any order.

Each call to deftrait creates a new trait even if the supplied trait name might repeat. If **equal?** is applied to two traits it will only return t if both arguments refer to the same trait.

2.4 Methods and Generic Functions

Methods are part of generic functions. Each generic function contains at least one of them. If a generic function is invoked, depending on the count (→ in Common Lisp all methods in a generic function must have the same arity, this is not required for the Lisp Package) and the classes of the arguments the most specific applicable method is selected and then executed.

To express applicability the parameters of a method can be specialized. This means that a parameter can be annotated by a class or trait. The class or trait used for annotation is called specializer.

Methods are created with the special function **defmethod** (→ defmethod in Common Lisp and def within class in Ruby). The function takes four arguments. None of the arguments gets evaluated.

The first argument must be a symbol that gives the name of the generic function. If the name is unbound or bound to nil in the current environment, a new generic function is created and the method is added to it. If the name is bound to a function that evaluates its arguments, this function is converted to a method and both methods (the converted and the created) are included in the generic function. The generic function is bound to its name in the current environment.

The second argument to defmethod is a generalized parameter list. This list contains either symbols or lists with two symbols in them (→ Common Lisp allows also eql clauses, these can be expressed by guards in the Lisp Package, Ruby only dispatches over the implicit parameter self). Just a symbol denotes the name of a parameter. A list of two symbols gives first the name of the parameter and second a name of a class or trait. This class or trait is called specializer and restricts on which arguments the method can be invoked.

The third argument to defmethod is a guard (→ this concept does not exist in Common Lisp or Ruby). It is executed with the method parameters bound to the arguments of the call and only if it evaluates to a value unequal to nil, the method is applicable. Otherwise the next less specific method (see call-next-method below) is checked.

The fourth argument to defmethod defines the body of the method. If the method is applied, it is executed with the method parameters bound to the arguments of the call. The result of this execution is the result of the method call.

Within the body of a method the function **call-next-method** (→ Common Lisp also supports providing arguments to call-next-method) can be used. The function takes no arguments and calls the next less specific method. For this call the same arguments were used as in the original method call.

This rises the questions how the most specific method for a list of arguments is selected out of the methods in a generic function and how the next less specific method is found. The algorithm works in two steps.

First, the list of applicable methods is calculated. A method is only applicable to a list of arguments if the count of arguments matches the count of parameters and each argument provided for a specialized parameter has a class that either equals the specializer or is a direct or indirect subclass of the specializer.

Second, the list of applicable methods is sorted from most to least specific. To compare two methods, the lists of parameters are scanned from left to right. If at the same parameter position one method has a specializer and another method is not specialized, then the method with the specialized parameter is more specific. If both methods have a specializer at the same parameter position, the specializers are compared.

Comparing two specializers S and T for an argument of class A works by ordering the superclasses of A into a list and then finding S and T in that list. If S is before T, then the method specialized on S is more specific.

The ordered list of superclasses (linearization, class precedence list) is created by a breath first and left-to-right traversal of superclasses starting at class A. Visited classes are only appended to the list once and if their subclasses are already in the list.

3. Examples

This chapter shows some examples that illustrate the features described above.

3.1 Special Sum

Common Lisp allows to use eql specializers to declare that a method is specialized for an instance or value. This feature does not exist in the Lisp Package but the method guard can be used to express the same constraint.

The following method calculates the ordinary sum of its two arguments, expect in the case where the first argument is 3 and the second argument is 4. In this case it returns the atom seven:

```
  (defmethod sum (x y) t (+ x y))
  (defmethod sum (x y) (and (equal? x 3) (equal? y 4)) (quote seven))
```

Note that a method with guard is more specific than a method without guard when the parameter specializers are the same (or do not exist as in the example).

3.2 Length

In Common Lisp the provided data types like number, list or string are also represented as classes and embedded in the class hierarchy. This is not the case in the Lisp Package. Therefore it is not possible to use parameter specializers for dispatching over built-in types. But method guards can be used for this:

```
  (defmethod length (s) (string? s) (string-length s))
  (defmethod length (l) (list? l) (list-length l))
```

3.3 Method Order

Given the following class and method definitions

```
  (defclass a ())
  (defclass b (a))
  (defclass c (b))
  (defclass d (b))
  (defclass e (c))
  (defclass f (c))
  (defclass g (e d))
  (defclass h (f))
```

```
(defclass ij (g h))
(defmethod m ((p a)) t (quote (1)))
(defmethod m ((p b)) t (cons 2 (call-next-method)))
(defmethod m ((p c)) t (cons 3 (call-next-method)))
(defmethod m ((p d)) t (cons 4 (call-next-method)))
(defmethod m ((p e)) t (cons 5 (call-next-method)))
(defmethod m ((p f)) t (cons 6 (call-next-method)))
(defmethod m ((p g)) t (cons 7 (call-next-method)))
(defmethod m ((p h)) t (cons 8 (call-next-method)))
(defmethod m ((p ij)) t (cons 9 (call-next-method)))
```

a call to the method m with an argument of class ij

```
(m (allocate-instance ij))
```

gives the result

```
(9 7 8 5 4 6 3 2 1).
```

This means that the order of specializers was (ij g h e d f c b a). This is the order in which classes in the superclass hierarchy of ij are visited the first time by a breadth first, left-to-right traversal.

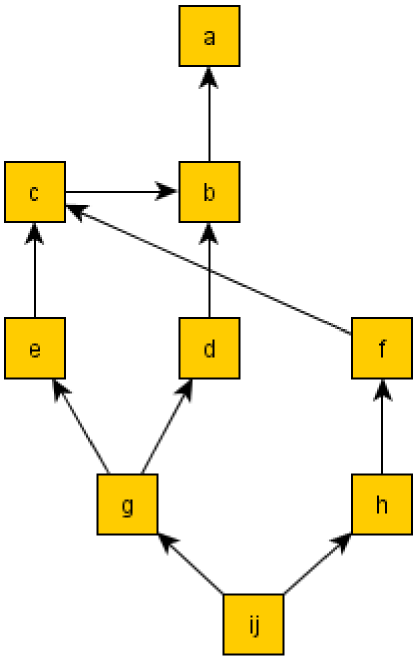

Diagram: Superclass hierarchy of class ij

Note that class c is visited twice and classes b and a are visited three times because they are reachable on two respectively three ways from ij. For the given example, all classes are appended to the class precedence list on the first visit because at this time their subclasses are already in the list.

4. Discussion

This chapter discusses the design decisions that lead to the object-oriented features of the Lisp Package.

4.1 Classes do not define any inner Structure for their Instances

This feature is inspired by the Ruby language. It was adopted because

- class definitions become short and simple and
- introducing a new instance variable means just using it in methods.

On the other hand most industrially and widely used programming languages [9] (e.g. Java, Objective-C, C++, C#) let their classes define the instance variables. In these languages access to a non-defined instance variable can be detected at compile time (if the access is not done via reflection). Predefined instance variables also allow for a fixed memory layout inside instances and therefore performance gains for variable access over a for example hash map based implementation.

In principle a compiler or IDE for the Lisp Package could detect write-only access to an instance variable and issue a warning in the same cases as errors (like “cannot resolve symbol”) are detected by the compilers of Java, Objective-C, C++ or C# - if the access is not done via reflection. In context of the Lisp Package, “not done via reflection” means that the slot name given in calls to the functions slot-value or assign (see chapter 2.2) can be computed at compile time.

4.2 Classes are no Instances

In Common Lisp classes are themselves instances of classes. It is even possible to use custom meta classes. These can be used to customize the object system itself [3].

Currently the Lisp Package does not have this feature. The main reason for this is that (custom) meta classes are not used frequently.

4.3 No Classes for built-in Types but Method Guards

In Common Lisp there are classes for many built-in types, but not for all (e.g. if the type specifier is a list, there is no corresponding class for it) [10]. Having classes for built-in types allows for using them as specializers in method definitions.

The Lisp Package has no classes for any built-in type (not even for the built-in type class – see chapter 4.2). To circumvent the missing possibility for specializers based on classes for built-in types, method guards (see chapter 2.4) can be used in the Lisp Package:

Common Lisp	(defmethod my-length ((a-list list)) ...)
Lisp Package	(defmethod my-length (a-list) (list? a-list) ...)

4.4 Classes must be defined before they can be used as Superclass

The Lisp Package uses the superclass names provided to a call to defmethod to resolve the corresponding classes in the current environment. Therefore they must be created and assigned before they can be used. This prevents cycles in the inheritance graph which so is ensured to be a proper hierarchy.

The side effect of defclass is to assign the created class to the given name in the current environment. This is a consistent behavior for a Lisp-1 – there is only one namespace for all kinds of objects (function, macro, class etc.).

4.5 A List of Superclasses must not include a class and one of its direct or indirect Superclasses

Consider the following example

```
 (defclass b ())
 (defclass c (b))
 (defclass d (b c))
 (defmethod m ((p b)) t 1)
 (defmethod m ((p c)) t 2)
```

with three classes b, c and d and methods specialized on b and c.

If the generic function m is called with an instance of class d, the method specialized on class c would be called first. This is because class c is before class b in the class precedence list of class d.

The expectation of the definer of class d might be different: class b before class c.

To prevent confusion and unexpected behavior in these situations, the Lisp Package rejects a definition of a class, if the list of superclasses includes a class and one of its direct or indirect superclasses.

4.6 Independent Traits must not overlap in provided Functionality

The key idea of traits is that multiple of them can be combined into a class with mechanisms for resolving conflicts: renaming or excluding methods [6].

It is not clear how renaming or excluding methods should work in a language with multiple dispatch, where a method does not necessarily belong to single class. Until now the best that the Lisp Package can do with regard to conflicts between traits is to make sure that they don't occur (see chapter 2.3).

4.7 No Arguments are accepted by call-next-method and change-class creates a new Instance

In Common Lisp two forms of call-next-method are supported. If it is called without arguments, the next method is implicitly called with the same arguments as the currently executing method. It is also possible to pass arguments. If arguments are passed, exceptional situations must be checked [11]:

> *When providing arguments to call-next-method, the following rule must be satisfied or an error of type error should be signaled: the ordered set of applicable methods for a changed set of arguments for call-next-method must be the same as the ordered set of applicable methods for the original arguments to the generic function.*

If change-class is destructive as in Common Lisp [12], it is possible to change the class of an argument that is passed to call-next-method implicitly. The same exceptional situations would need to be checked in this case, too.

To avoid the (runtime and language implementation) overhead of this check, the Lisp Package provides a version of change-class, that does not modify its argument but creates a changed copy and only allows the non-argument form of call-next-method.

4.8 Methods can have different Arities within one Generic Function

In Common Lisp all methods of a generic function must have congruent lambda lists [13]. This means they must have the same number of required and the same number of optional parameters.

The Lisp Package has only required parameters in lambda lists. Optional parameters, &rest and &key are unknown. When a generic function is called with some arguments, there is no ambiguity regarding applicable methods. Only methods with matching parameter count are eligible. Therefore methods with different arities are allowed to participate in a generic function.

4.9 Subclasses first, left-to-right Search for next Method

For non-trivial class hierarchies there are multiple sensible linearizations. For the example from chapter 3.3 (which is equivalent to the pedalo example from [14]) the following linearizations all mention subclasses before superclasses (ij before g, ij before h, g before e, g before d, h before f, e before c, d before b, f before c, c before b, b before a) and respect the left-to-right order of superclasses as given in the defclass expressions (g before h, e before d):

(ij g e d h f c b a)
(ij g e h d f c b a)
(ij g e h f c d b a)
(ij g e h f d c b a)
(ij g h e d f c b a)*
(ij g h e f c d b a)
(ij g h e f d c b a)
(ij g h f e c d b a)
(ij g h f e d c b a)

The one marked with a star (*) is used by the Lisp Package for this example. Other linearization algorithms choose different linearizations. The reason for this is the traversal of the class hierarchy: the C3 algorithm for example works depth first, whereas the algorithm in the Lisp Package works breath first.

5. Outlook

The use of the object-oriented features of the Lisp Package in larger development efforts will show, if the "non-standard" choices for some features (e.g. classes are themselves no instances, not using C3 linearization) are acceptable or must be reworked.

For own experiments with the Lisp Package, a spreadsheet demo that uses the Lisp Package as engine to evaluate the formulas in cells can be downloaded from http://simplysomethings.de/functional+programming/calc.html (click the link "Calc.jar" at the end of the text). The JAR file also includes the source codes.

6. References

[1] http://www.simplysomethings.de/functional+programming/calc.html

[2] R. P. Gabriel, K. M. Pitman
"Technical Issues of Separation in Function Cells and Value Cells"

[3] G. Kiczales, J. Rivières, D. G. Bobrow
"The Art of the Metaobject Protocol"

[4] D. Flanagan, Y. Matsumoto
"The Ruby Programming Language"

[5] A. Snyder
"Encapsulation and Inheritance in Object-Oriented Programming Languages"

[6] G. Curry, L. Baer, D. Lipkie, B. Lee
"Traits: An approach to multiple-inheritance subclassing"

[6a] N. Schärli, S. Ducasse, O. Nierstrasz, A. Black
"Traits - Composable Units of Behavior"

[7] http://www.simplysomethings.de/lisp+package/index.html

[8] H. G. Baker
"Equal Rights for Functional Objects or, The More Things Change, The More They Are the Same"

[9] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[10] http://www.lispworks.com/documentation/HyperSpec/Body/04_cg.htm

[11] http://www.lispworks.com/documentation/HyperSpec/Body/f_call_n.htm

[12] http://www.lispworks.com/documentation/HyperSpec/Body/f_chg_cl.htm

[13] http://www.lispworks.com/documentation/HyperSpec/Body/07_fd.htm

[14] K. Barrett, B. Cassels, P. Haahr, D. A. Moon, K. Playford, P. T. Withington
"A Monotonic Superclass Linearization for Dylan"