WIKIPEDIA

Mixin

In object-oriented programming languages, a **Mixin** is a class that contains methods for use by other classes without having to be the parent class of those other classes. How those other classes gain access to the mixin's methods depends on the language. Mixins are sometimes described as being "included" rather than "inherited".

Mixins encourage code reuse and can be used to avoid the inheritance ambiguity that multiple inheritance can cause[1] (the "diamond problem"), or to work around lack of support for multiple inheritance in a language. A mixin can also be viewed as an interface with implemented methods. This pattern is an example of enforcing the dependency inversion principle.

Contents

History

Mixins first appeared in the Symbolics's object-oriented Flavors system (developed by Howard Cannon), which was an

approach to object-orientation used in Lisp Machine Lisp. The name was inspired by Steve's Ice Cream Parlor in Somerville, Massachusetts:[2] The owner of the ice cream shop offered a basic flavor of ice cream (vanilla, chocolate, etc.) and blended in a combination of extra items (nuts, cookies, fudge, etc.) and called the item a "mix-in", his own trademarked term at the time.[3]

Definition

Mixins are a language concept that allows a programmer to inject some code into a class. Mixin programming is a style of software development, in which units of functionality are created in a class and then mixed in with other classes.[4]

A mixin class acts as the parent class, containing the desired functionality. A subclass can then inherit or simply reuse this functionality, but not as a means of specialization. Typically, the mixin will export the desired functionality to a child class, without creating a rigid, single "is a" relationship. Here lies the important difference between the concepts of mixins and inheritance, in that the child class can still inherit all the features of the parent class, but, the semantics about the child "being a kind of" the parent need not be necessarily applied.

Advantages

1. It provides a mechanism for multiple inheritance by allowing multiple classes to use the common functionality, but without the complex semantics of multiple inheritance.[5]
2. Code reusability: Mixins are useful when a programmer wants to share functionality between different classes. Instead of repeating the same code over and over again, the common functionality can simply be grouped into a mixin and then included into each class that requires it.[6]
3. Mixins allow inheritance and use of only the desired features from the parent class, not necessarily all of the features from the parent class.[7]

Implementations

In Simula, classes are defined in a block in which attributes, methods and class initialization are all defined together; thus all the methods that can be invoked on a class are defined together, and the definition of the class is complete.

In Flavors, a Mixin is a class from which another class can inherit slot definitions and methods. The Mixin usually does not have direct instances. Since a Flavor can inherit from more than one other Flavor, it can inherit from one or more Mixins. Note that the original Flavors did not use generic functions.

In New Flavors (a successor of Flavors) and CLOS, methods are organized in "generic functions". These generic functions are functions that are defined in multiple cases (methods) by class dispatch and method combinations.

CLOS and Flavors allow mixin methods to add behavior to existing methods: :before and :after daemons, whoppers and wrappers in Flavors. CLOS added :around methods and the ability to call shadowed methods via CALL-NEXT-METHOD. So, for example, a stream-lock-mixin can add locking around existing methods of a stream class. In Flavors one would write a wrapper or a whopper and in CLOS one would use an :around method. Both CLOS and Flavors allow the computed reuse via method combinations. :before, :after and :around methods are a feature of the standard

method combination. Other method combinations are provided.

An example is the + method combination, where the resulting values of each of the applicable methods of a generic function are arithmetically added to compute the return value. This is used, for example, with the border-mixin for graphical objects. A graphical object may have a generic width function. The border-mixin would add a border around an object and has a method computing its width. A new class bordered-button (that is both a graphical object and uses the border mixin) would compute its width by calling all applicable width methods—via the + method combination. All return values are added and create the combined width of the object.

In an OOPSLA 90 paper,[8] Gilad Bracha and William Cook reinterpret different inheritance mechanisms found in Smalltalk, Beta and CLOS as special forms of a mixin inheritance.

Programming languages that use mixins

Other than Flavors and CLOS (a part of Common Lisp), some languages that use mixins are:

- Ada (by extending an existing tagged record with arbitrary operations in a generic)
- Cobra
- ColdFusion (Class based using includes and Object based by assigning methods from one object to another at runtime)
- Curl (with Curl RTE)
- D (called "template mixins" (http://www.digitalmars.com/d/template-mixin.html); D also includes a "mixin" (http://dlang.org/mixin.html) statement that compiles strings as code.)
- Dart
- Factor[9]
- Groovy
- JavaScript Delegation - Functions as Roles (Traits and Mixins)
- OCaml
- Perl (through roles in the Moose extension of the Perl 5 object system)
- Perl 6
- PHP's "traits"
- Python
- Racket (mixins documentation (http://docs.racket-lang.org/guide/classes.html#(part._.Mixins)))
- Ruby
- Scala[10]
- XOTcl/TclOO (http://wiki.tcl.tk/18152) (object systems builtin to Tcl)[11]
- Sass (A stylesheet language)
- Vala
- Swift
- SystemVerilog

Some languages do not support mixins on the language level, but can easily mimic them by copying methods from one

object to another at runtime, thereby "borrowing" the mixin's methods. This is also possible with statically typed languages, but it requires constructing a new object with the extended set of methods.

Other languages that do not support mixins can support them in a round-about way via other language constructs. C# and Visual Basic .NET support the addition of extension methods on interfaces, meaning any class implementing an interface with extension methods defined will have the extension methods available as pseudo-members.

Examples

In Common Lisp

Common Lisp provides mixins in CLOS (Common Lisp Object System) similar to Flavors.

object-width is a generic function with one argument that uses the + method combination. This combination determines that all applicable methods for a generic function will be called and the results will be added.

```
(defgeneric object-width (object)
 (:method-combination +))
```

button is a class with one slot for the button text.

```
(defclass button ()
 ((text :initform "click me")))
```

There is a method for objects of class button that computes the width based on the length of the button text. + is the method qualifier for the method combination of the same name.

```
(defmethod object-width + ((object button))
  (* 10 (length (slot-value object 'text))))
```

A border-mixin class. The naming is just a convention. There are no superclasses, and no slots.

```
(defclass border-mixin () ())
```

There is a method computing the width of the border. Here it is just 4.

```
(defmethod object-width + ((object border-mixin))
 4)
```

bordered-button is a class inheriting from both border-mixin and button.

```
(defclass bordered-button (border-mixin button) ())
```

We can now compute the width of a button. Calling object-width computes 80. The result is the result of the single applicable method: the method object-width for the class button.

```
? (object-width (make-instance 'button))
80
```

We can also compute the width of a bordered-button. Calling object-width computes 84. The result is the sum of the results of the two applicable methods: the method object-width for the class button and the method object-width for the class border-mixin.

```
? (object-width (make-instance 'bordered-button))
84
```

In Python

In Python, the SocketServer module[12] has both a UDPServer class and a TCPServer class. They act as servers for UDP and TCP socket servers, respectively. Additionally, there are two mixin classes: ForkingMixIn and ThreadingMixIn. Normally, all new connections are handled within the same process. By extending TCPServer with the ThreadingMixIn as follows:

```
class ThreadingTCPServer(ThreadingMixIn, TCPServer):
  pass
```

the ThreadingMixIn class adds functionality to the TCP server such that each new connection creates a new thread. Alternatively, using the ForkingMixIn would cause the process to be forked for each new connection. Clearly, the functionality to create a new thread or fork a process is not terribly useful as a stand-alone class.

In this usage example, the mixins provide alternative underlying functionality without affecting the functionality as a socket server.

In Ruby

Most of the Ruby world is based around mixins via Modules. The concept of mixins is implemented in Ruby by the keyword include to which we pass the name of the module as parameter.

Example:

```
class Student
  include Comparable # The class Student inherits the Comparable module using the 'include' keyword
  attr_accessor :name, :score

  def initialize(name, score)
```

```
  @name = name
  @score = score
end

# Including the Comparable module requires the implementing class to define the <=> comparison operator
# Here's the comparison operator. We compare 2 student instances based on their scores.

def <=>(other)
  @score <=> other.score
end

# Here's the good bit - I get access to <, <=, >,>= and other methods of the Comparable Interface for free.
end

s1 = Student.new("Peter", 100)
s2 = Student.new("Jason", 90)

s1 > s2 #true
s1 <= s2 #false
```

In JavaScript

The *Object-Literal and* extend *Approach*

It is technically possible to add behavior to an object by binding functions to keys in the object. However, this lack of separation between state and behavior has drawbacks:

1. It intermingles properties of the model domain with that of implementation domain.
2. No sharing of common behavior. Metaobjects solve this problem by separating the domain specific properties of objects from their behaviour specific properties.[13]

An extend function (in this case from the Underscore.js library, which copies all of the functionality from a source object, to a destination object, attributes, functions, etc.) is used to mix the behavior in:[14]

```
// This example may be contrived.
// It's an attempt to clean up the previous, broken example.
var Halfling = function (fName, lName) {
  this.firstName = fName;
  this.lastName = lName;
}

var NameMixin = {
  fullName: function () {
    return this.firstName + ' ' + this.lastName;
  },
  rename: function(first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
```

```
};

var sam = new Halfling('Sam', 'Lowry');
var frodo = new Halfling('Freeda', 'Baggs');

// Mixin the other methods
_.extend(Halfling.prototype, NameMixin);

// Now the Halfling objects have access to the NameMixin methods
sam.rename('Samwise', 'Gamgee');
frodo.rename('Frodo', 'Baggins');
```

The pure function and delegation based *Flight-Mixin Approach*

Even though the firstly described approach is mostly widespread the next one is closer to what JavaScript's language core fundamentally offers - Delegation.

Two function object based patterns already do the trick without the need of a third party's implementation of extend.

```
// Implementation
var EnumerableFirstLast = (function () { // function based module pattern.
  var first = function () {
    return this[0];
  },
  last = function () {
    return this[this.length - 1];
  };
  return function () {     // function based Flight-Mixin mechanics ...
    this.first  = first;  // ... referring to ...
    this.last   = last;   // ... shared code.
  };
}());

// Application - explicit delegation:
// applying [first] and [last] enumerable behavior onto [Array]'s [prototype].
EnumerableFirstLast.call(Array.prototype);

// Now you can do:
a = [1, 2, 3];
a.first(); // 1
a.last();  // 3
```

In other languages

In the Curl web-content language, multiple inheritance is used as classes with no instances may implement methods. Common mixins include all skinnable ControlUIs inheriting from SkinnableControlUI, user interface delegate objects that require dropdown menus inheriting from StandardBaseDropdownUI and such explicitly named mixin classes as FontGraphicMixin, FontVisualMixin and NumericAxisMixin-of class. Version 7.0 added library access so that mixins do not need to be in the same package or be public abstract. Curl constructors are factories that facilitates using

multiple-inheritance without explicit declaration of either interfaces or mixins.

Interfaces and traits

Java 8 introduces a new feature in the form of default methods for interfaces.[15] Basically it allows a method to be defined in an interface with application in the scenario when a new method is to be added to an interface after the interface class programming setup is done. To add a new function to the interface means to implement the method at every class which uses the interface. Default methods help in this case where they can be introduced to an interface any time and have an implemented structure which is then used by the associated classes. Hence default methods adds a possibility of applying the concept in a mixin sort of a way.

Interfaces combined with aspect-oriented programming can also produce full-fledged mixins in languages that support such features, such as C# or Java. Additionally, through the use of the marker interface pattern, generic programming, and extension methods, C# 3.0 has the ability to mimic mixins. With C# 3.0 came the introduction of Extension Methods[2] and they can be applied, not only to classes but, also, to interfaces. Extension Methods provide additional functionality on an existing class without modifying the class. It then becomes possible to create a static helper class for specific functionality that defines the extension methods. Because the classes implement the interface (even if the actual interface doesn't contain any methods or properties to implement) it will pick up all the extension methods also.[16][17][18]

ECMAScript (in most cases implemented as JavaScript) does not need to mimic object composition by stepwise copying fields from one object to another. It natively[19] supports Trait and Mixin[20][21] based object composition via function objects that implement additional behavior and then are delegated via `call` or `apply` to objects that are in need of such new functionality.

In Scala

Scala has a rich type system and Traits are a part of Scala's type system which help implement mixin behaviour. As their name reveals, Traits are usually used to represent a distinct feature or aspect that is normally orthogonal to the responsibility of a concrete type or at least of a certain instance.[22] For example, the ability to sing is modeled as such an orthogonal feature: it could be applied to Birds, Persons, etc.

```
trait Singer{
  def sing { println(" singing ... ") }
  //more methods
}

class Birds extends Singer
```

Here, Bird has mixed in all methods of the trait into its own definition as if class Bird had defined method sing() on its own.

As `extends` is also used to inherit from a super class, in case of a trait `extends` is used if no super class is inherited and only for mixin in the first trait. All following traits are mixed in using keyword `with`.

```
class Person
class Actor extends Person with Singer
class Actor extends Singer with Performer
```

Scala allows mixing in a trait (creating an anonymous type) when creating a new instance of a class. In the case of a Person class instance, not all instances can sing. This feature comes use then:

```
class Person{
 def tell {  println (" Human ") }
 //more methods
}

val singingPerson = new Person with Singer
singingPerson.sing
```

In Swift

Mixin can be achieved in Swift by using a language feature called Default implementation in Protocol Extension.

```
protocol ErrorDisplayable {
    func error(message:String)
}

extension ErrorDisplayable {
    func error(message:String) {
        // Do what it needs to show an error
        //...
        print(message)
    }
}

struct NetworkManager : ErrorDisplayable{
    func onError() {
        error("Please check your internet Connection.")
    }
}
```

See also

- Abstract type
- Decorator pattern
- Policy-based design
- Trait, a similar structure that doesn't require linear composition

References

1. Boyland, John; Giuseppe Castagna (26 June 1996). "Type-Safe Compilation of Covariant Specialization: A Practical Case". In Pierre Cointe. *ECOOP '96, Object-oriented Programming: 10th European Conference* (https://books.google.com/books?id=sGvtaGy8SJ8C&pg=PA16&lpg=PA16&dq=pathologies+of+multiple+inheritance&source=bl&ots=sF3Ah-XZSq&sig=JG7VV5hrjm781yqT5iGHSnlLI0I&hl=en&ei=Obi-TuLMCebV0QHD9bnyBA&sa=X&oi=book_result&ct=result&resnum=2&ved=0CCMQ6AEwAQ#v=onepage&q&f=false). Springer. pp. 16–17. ISBN 9783540614395. Retrieved 17 January 2014.
2. Using Mix-ins with Python (http://www.linuxjournal.com/article/4540)
3. Mix-Ins (Steve's ice cream, Boston, 1975) (http://listserv.linguistlist.org/cgi-bin/wa?A2=ind0208a&L=ads-l&P=11751)
4. http://c2.com/cgi/wiki?MixIn
5. http://culttt.com/2015/07/08/working-with-mixins-in-ruby/
6. http://naildrivin5.com/blog/2012/12/19/re-use-in-oo-inheritance.html
7. http://justinleitgeb.com/ruby/moving-beyond-mixins/
8. OOPSLA '90, Mixin based inheritance (pdf) (http://www.bracha.org/oopsla90.pdf)
9. slava (2010-01-25). "Factor/Features/The language" (http://concatenative.org/wiki/view/Factor/Features/The%20language). concatenative.org (http://concatenative.org). Retrieved 2012-05-15. "Factor's main language features: ... Object system with Inheritance, Generic functions, Predicate dispatch and *Mixins*" External link in |publisher= (help)
10. "Mixin Class Composition" (http://docs.scala-lang.org/tutorials/tour/mixin-class-composition.html). École polytechnique fédérale de Lausanne. Retrieved 16 May 2014.
11. Mixin classes in XOTcl (http://media.wu-wien.ac.at/doc/tutorial.html#mixins)
12. Source code for SocketServer in CPython 3.5 (https://hg.python.org/cpython/file/3.5/Lib/socketserver.py)
13. http://raganwald.com/2014/04/10/mixins-forwarding-delegation.html
14. http://bob.yexley.net/dry-javascript-with-mixins/
15. https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html
16. Implementing Mix-ins with C# Extension Methods (http://www.zorched.net/2008/01/03/implementing-mixins-with-c-extension-methods/)
17. I know the answer (it's 42) : Mix-ins and C# (http://blogs.msdn.com/abhinaba/archive/2006/01/06/510034.aspx)
18. Mixins, generics and extension methods in C# (http://erwyn.bloggingabout.net/2005/10/20/mixins-generics-and-extension-methods-in-c/)
19. The many talents of JavaScript for generalizing Role Oriented Programming approaches like Traits and Mixins (http://peterseliger.blogspot.de/2014/04/the-many-talents-of-javascript.html#the-many-talents-of-javascript-for-generalizing-role-oriented-programming-approaches-like-traits-and-mixins), April 11, 2014.
20. Angus Croll, A fresh look at JavaScript Mixins (http://javascriptweblog.wordpress.com/2011/05/31/a-fresh-look-at-javascript-mixins/), published May 31, 2011.
21. JavaScript Code Reuse Patterns (https://github.com/petsel/javascript-code-reuse-patterns/tree/master/source/components/composition/), April 19, 2013.
22. https://gleichmann.wordpress.com/2009/07/19/scala-in-practice-traits-as-mixins-motivation

External links

- MixIn (http://c2.com/cgi/wiki?MixIn) at Portland Pattern Repository
- Mixins (http://www.macromedia.com/support/documentation/en/flex/1/mixin/index.html) in ActionScript
- The Common Lisp Object System: An Overview (http://www.dreamsongs.com/NewFiles/ECOOP.pdf) by Richard P. Gabriel and Linda DeMichiel provides a good introduction to the motivation for defining classes by means of generic functions.