Eiffel, a pure OO language with support for Design by Contract

From the perspective of a PHP programmer

by Berend de Boer

close

Hello World

```
1  class HELLO_WORLD

2  creation

3   make

4  feature -- Initialization

5   make is
6    do
7     print ("hello world")
8    end

9  end
```

History

1. 1985: Bertrand Meyer and Jean Marc Nerson begin with the development of a new, original programming language: Eiffel.
2. 1986: first Eiffel compiler.
3. 1991: Description of the language with the publication of “Eiffel the language” (ETL2) by Bertrand Meyer (Prentice Hall).
4. First version of **SmartEiffel**.
5. 1997: Publication of Object-Oriented Software Construction (OOSC2) by Bertrand Meyer.
6. 1997: first release of Gobo Eiffel Project.
7. 1997: NICE, the Eiffel Consortium, organises first **Eiffel Struggle** (six held since). Prises are awarded for Eiffel applications and Eiffel libraries.
8. 2005: Publication of the **Eiffel ECMA standard**: ECMA 367.

PHP and Eiffel

The words object and class have the same meaning in Eiffel and PHP literature.

In the next couple of slides I compare PHP's object-oriented facilities with Eiffel. I focus mainly on the similarities. After that I'll return to how Eiffel is different.

Examples should be familiar, they're taken from O'Reilly's Programming PHP book.

Creating an object

```
$rasmus = new Person;
```

It’s a convention in Eiffel to use uppercase class names.

```
1  local
2     erasmus: PERSON;
3  do
4     create erasmus
5  end
```

Accessing features

Eiffel calls properties attributes and methods routines. Or just features to mean both of them.

```
echo $rasmus->age;
echo $rasmus->birthday();
echo $rasmus::TYPE_CREDITCARD;
```

Eiffel does not distinguish between accessing a property or calling a method.

```
1  demonstrate is
2    do
3      print (erasmus.age)
4      print (erasmus.birthday)
5      print (erasmus.type_creditcard)
6    end
```

In PHP internal details bleed through to the caller (the client in Eiffel lingo). It violates Eiffel's Uniform Access Principle:

> "All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation."

Declaring a class

```
class Person {
  var $age;
  const TYPE_CREDITCARD = 0;

  function birthday () {
    ...
  }
}
```

```
1  class PERSON

2  feature -- Access

3    age: INTEGER
4    type_creditcard: INTEGER is 0

5    birthday: DATE is
6      do
7        ...
8      end

9  end
```

Public and private

Most languages have mechanisms to influence visibility (accessibility) of features (properties and methods).

They need those because they don't have Design by Contract, in particular class invariants. They are there so you can not screw up. And they usually cast a class on stone, so you can access nor change the class (Delphi/C#).

Eiffel uses visibility for entirely different reasons: to provide a clean interface to the client and not clutter the visible interface with internal routines.

```php
class Person {
  public $username = 'Anyone can see me';
  protected $rowId = 0;
  private $hidden = true;
}
```

```
1  class PERSON

2  feature {ANY}

3    username: STRING is "Anyone can see me"

4  feature {PERSON}

5    row_id: INTEGER is 0

6  feature {NONE}

7    hidden: BOOLEAN is true

8  end
```

Static and final

```
class Person {
  static $global = 23;
  final function get_name () {
    ...
  }
}
```

Eiffel does not have the concept of static variables. But you can declare methods as final (frozen in Eiffel lingo).

```
1  class PERSON

2  feature

3    frozen get_name: STRING is
4      do
5        ...
6      end

7  end
```

Abstract methods

```
abstract class Component {
  abstract function printOutput()
}
```

```
1  deferred class COMPONONENT

2  feature

3    print_output is
4      deferred
5      end

6  end
```

Inheritance

```
class Employee extends Person {
}
```

```
1  class EMPLOYEE
2  inherit
3      PERSON
4  end
```

Interfaces

```
interface Printable {
  function printOutput();
}

class ImageComponent implements Printable {
  function printOutput() {
    ...
  }
}
```

Eiffel does not have the interface concept. Interfaces are a poor man's concept of multiple inheritance, and Eiffel implements full multiple inheritance. Instead you can use deferred classes and deferred methods. And give default implementations if you so wish.

```
1  deferred class PRINTABLE

2    function print_output is
3      deferred
4      end

5  end
```

```
6   class IMAGE_COMPONENT

7   inherit

8      PRINTABLE

9   feature

10     print_output is
11        do
12           ...
13        end

14  end
```

Constructors

```
class Employee extends Person {
  function __construct($name, $age, $salary) {
    $this->Person($name, age)
    $this->salary = $salary;
  }
}
```

```
1  class EMPLOYEE

2  inherit

3    PERSON
4      redefine
5        make as make_person
6      end

7  creation

8    make

9  feature {NONE} -- Initialisation
```

```
10      make (a_name: STRING; an_age: INTEGER; a_salary: DOUBLE) is
11        do
12          make_person (a_name, an_age)
13          salary := a_salary
14        end

15    feature -- Access

16      salary: DOUBLE

17    end
```

Destructors

```
class Building {
  function __destructor() {
    ...
  }
}
```

Like PHP, destructors in Eiffel are only called at the end of the life of an object. And it happens automatically.

```
1  class BUILDING

2  inherit
3    MEMORY
4      redefine
5        dispose
6      end

7  feature {NONE} -- Dispose

8    dispose is
9      do
10       ...
```

```
11          precursor
12        end

13    end
```

Precursor is the keyword to call the override feature, the method of the parent (ancestor) class.

Other

Introspection and reflection: can be achieved somewhat with tuples and agents. There is also a separate library: http://se.ethz.ch/people/leitner/erl_g/

Serialisation: depending on compiler, for example Eiffel Studio can serialise to text files and object-oriented or relational databases.

Language goal

From ETL2:

> "Eiffel embodies a "certain idea" of software construction: the belief that it is possible to treat this task as a serious engineering enterprise . . . Such aims lead to a *new culture* of software development . . . Eiffel is nothing else than these principles taken to their full consequences. In particular, the engineering of quality software components requires an appropriate notation . . . "

Principles (for full set see OOSC2):

- Eiffel is a language of least surprise: when more than one interpretation is possible, it asks. It does not just declare a variable if you have made a typo (static typing).
- Software written in Eiffel is open and closed: it is open for extension, and it is closed in the sense that it is available for use by other modules.
- Command-query separation: functions should not produce abstract side effects.

Features

1. Compiled language, uses C as its portable assembly language. Also compiles to other back-ends as machine code, .NET, or Java Virtual Machine.
2. Automatic garbage collection.
3. Pure Object-Oriented: everything is an object.
4. Static typing.
5. Genericity: write data structures only once.
6. Multiple inheritance.
7. Design by Contract built-in.
8. Tuples and agents.

Pure OO

string.count instead of count(string).

```php
$s = "hello world";
echo strlen($s);
```

```
1   class EXAMPLE

2   creation make

3   feature

4     make is
5       local
6         s: STRING
7       do
8         s := "hello world"
9         print (s.count)
10      end

11  end
```


Hello World

History

PHP and Eiffel

Language goal

Design by Contract

DbC benefits

Web apps

Demo

Resources

Conclusion

close

Operations (functions) always operate on objects. This has an important influence on discoverability: what operations are applicable on my object?

Is the strlen operator applicable on numbers? Arrays?

```
$s = 10;
echo strlen($s);
```

But Eiffel does not compile this:

```
1    make is
2      local
3        s: INTEGER
4      do
5        s := 10
6        print (s.count)
7      end
```

Static typing

Eiffel is statically and strongly typed. PHP is dynamically and weakly typed. If you made a typo in your PHP variable, it just gets declared. In Eiffel you cannot make such typos.

Genericity

```
1  class STACK [G]
2  feature -- Access
3      top: G
4  end
```

Multiple inheritance

```
1  class  RADIO_ALARM
2  inherit
3      RADIO
4      ALARM
5  end
```

Design by Contract

Other languages depend on tools to do Design by Contract (DbC). In Eiffel it is built into the language.

It is important. For the Ariadne 5 rocket the decision was made to reuse software from Ariadne 4. However the conditions when this piece of software could be reused was buried in an obscure document. Eiffel programmers put the contracts right there where they belong: in the code.

Bertrand Meyer: Reuse without a contract is sheer folly!

> "From CORBA to C++ to Visual Basic to ActiveX to Java, the hype is on software components. The Ariane 5 blunder shows clearly that naïve hopes are doomed to produce results far worse than a traditional, reuse-less software process. To attempt to reuse software without Eiffel-like assertions is to invite failures of potentially disastrous consequences. The next time around, will it only be an empty payload, however expensive, or will it be human lives?"

See his article on the topic: `http://archive.eiffel.com/doc/manuals/technology/contract/ariane/page.html`

Pseudo DbC

Most languages have an assert facility. PHP has that as well.

```
class Person {

  function Person($name, $age) {
    assert($age >= 0);
    $this->name = $name;
    $this->age = $age;
  }
}
```

This guarantees that you cannot create a Person with a negative age. But that's also where the guarantee stops.

```
$rasmus = new Person;
$rasmus->age = -1;
```

Class invariants

With Eiffel you can put the guarantee right there were it belongs, in the class itself:

```
1  class PERSON
2  feature
3      age: INTEGER
4  invariant
5      age_not_negative: age >= 0
6  end
```

You cannot violate this invariant without triggering an exception.

Preconditions

Preconditions are an obligation put upon the caller: the caller has to make sure it fulfils them.

```
1   class PERSON

2   creation

3     make

4   feature

5     make is (an_age: INTEGER) is
6       require
7         age_not_negative: an_age >= 0
8       do
9         age := an_age
10      end

11  feature

12    age: INTEGER
```

```
13  invariant
14      age_not_negative: age >= 0
15  end
```

Non-Redundancy principle: under no circumstances shall the body of a routine ever test for the routine's precondition.

Postconditions

Postconditions are a promise by the called routine: if you fulfil the precondition, this is what I shall do for you.

```
1   class PERSON

2   feature

3     age: INTEGER

4     set_age (an_age: INTEGER) is
5       require
6         age_not_negative: an_age >= 0
7       do
8         age := an_age
9       ensure
10        age_set: age = an_age
11      end

12  invariant
13    age_not_negative: age >= 0

14  end
```

DbC is fully integrated with inheritance

Class contracts are inherited, as well as the pre– and postconditions for redefined features.

```
1   class EMPLOYEE

2   inherit
3     PERSON
4       redefine
5         set_age
6       end

7   feature

8     set_age (an_age: INTEGER) is
9       do
10        precursor (an_age)
11        ...
12      end

13  end
```

Correctness formula

$$\{P\}\ A\ \{Q\}$$

Meaning:

> "Any execution of A, starting in a state where P holds, will terminate in a state where Q holds."

Loop (in)variants

Does your loop terminate?

```
1   local
2    i: INTEGER
3    code: INTEGER
4   do
5    from
6     i := 1
7    variant
8     2 + (s.count - i)
9    until
10    i > s.count
11   loop
12    code := s.item_code (i)
13          ...
14   end
15  end
```

DbC checking

In Eiffel you have the option to compile an application with:

1. No contract checking at all.
2. Only certain kind of contracts enabled, for example only the preconditions.
3. Contracts only enabled for certain classes.
4. Contracts only enabled for certain features.

DbC benefits

1. Help in writing correct software.
2. Documentation aid.
3. Support for testing, debugging and quality assurance.
4. Support for software fault tolerance.

close

Web apps

Whatever way you want: ASP.NET, CGI, FastCGI, built-in web server with servlets.

```
1  class HTML_PAGE

2  inherit EPX_CGI

3  feature
4     execute is
5        do
6           content_text_html
7           doctype
8           b_html
9           b_head
10          title ("Convert Xplain to SQL")
11          e_head
12          b_body
13          -- ...
14          e_body
15          e_html
16       end
17 end
```

Demo

Eiffel IDE demonstration: Eiffel Studio.

What:

1. Inheritance overview.
2. Flat and short forms.

Resources

Well-known libraries:

- Gobo, data structures and much more: `http://www.gobosoft.com/`
- ecli, ODBC binding: `http://sourceforge.net/projects/safe`
- eposix, POSIX API binding: `http://www.pobox.com/~berend/eposix/`

Web sites:

- Eiffel Studio: `http://www.eiffel.com/downloads/`
- Lots of links: `http://www.cetus-links.org/oo_eiffel.html`
- Lots of libraries: `http://eiffelzone.com/`
- EiffelRoom: `http://www.eiffelroom.com/`
- Eiffel Blog: `http://teameiffel.blogspot.com/`

Conclusion

Eiffel or PHP? It’s more Eiffel and PHP. Each has its place.

I wouldn’t want to write a credit card processing application or an XSLT processor in PHP. But delivering a content management system without PHP support is likewise unthinkable.