Eiffel in Depth

Bertrand Meyer
Emmanuel Stapf (Eiffel Software)
Pei Yu
& members of the ETH Chair of Software Engineering

Plan of these slides

1. Overview
2. The environment(s)
3. Method overview
4. Language basics
5. Dynamic model
6. Genericity & inheritance
7. Design by Contract™
8. External interface
9. Agents
10. Advanced design
11. Advanced mechanisms
12. Conclusion
13. Supplements

Course organization

Course page:

Teaching staff:
- Lecturer: Bertrand Meyer & members of the Chair of Software Engineering
- Course assistant: Yu (Max) Pei

Grading: 70% project, 30% exam
Project will be a web-based system using the new EiffelWeb
Exam will be on last lecture slot of the semester: 15 Dec.

Purpose of this course

To give you an in-depth understanding of a software method, language and environment: Eiffel (and EiffelStudio)
To improve your understanding of software engineering and software architecture

To give you a feel for the challenges involved in both software design and language design

The software of the future

Product quality
- Correctness
- Robustness
- Security

Process quality
- Fast development
- No semantic gap ("impedance mismatch") between developers and other stakeholders
- Self-validating, self-testing
- Ease of change
- Reusability
Why Eiffel?

Productivity: faster time to market, fewer developers
Reliability: fewer bugs
Extendibility: be responsive to customer needs
Reuse: stand on the shoulder of giants
Efficiency: make the best use of hardware resources
Maintainability: spend your time on new developments

Language versions

Eiffel 1, 1986
- Classes, contracts, genericity, single and multiple inheritance, garbage collection, ...

Eiffel 2, 1988 (Object-Oriented Software Construction)
- Exceptions, constrained genericity

Eiffel 3, 1990-1992 (Eiffel: The Language)
- Basic types as classes, infix & prefix operators...

Eiffel 4, 1997
- "Precursor" and agents

- www.ecma-international.org/publications/standards/Ecma-367.htm
- Attached types, conversion, assigner commands...

Eiffel: Method, Language, Environment

Method:
- Applicable throughout the lifecycle
- Object-oriented to the core
- Seamless development
- Based on Design by Contract™ principles

Language:
- Full power of object technology
- Simple yet powerful, numerous original features

Environment:
- Integrated, provides single solution, including analysis and modeling
- Lots of platforms (Unix, Windows, VMS, .NET...)
- Open and interoperable

Some typical users

Axa Rosenberg
Investment management: from $2 billion to >$100 billion
2 million lines

Chicago Board of Trade
Price reporting system
Eiffel + CORBA + Solaris + Windows + ...

Xontech (for Boeing)
Large-scale simulations of missile defense

Northrop-Grumman

Swedish social security: accident reporting & management

Learning Eiffel

- Simple syntax, no cryptic symbols
- Eiffel programmers know all of Eiffel
- Wide variety of user backgrounds
  - "If you can write a conditional, you can write a contract"
- Fast learning curve
- Lots of good models to learn from
- Strong style rules
- May need to "unlearn" needless tricks

The Eiffel method: some principles

- Abstraction
- Information hiding
- Seamlessness
- Reversibility
- Design by Contract
- Open-Closed principle
- Single choice principle
- Single model principle
- Uniform access principle
- Command-query separation principle
- Option-operand separation principle
- Style matters

... See next...
The Eiffel language

- Classes
- Uniform type system, covering basic types
- Genericity
- Inheritance, single and multiple
- Conversion
- Covariance
- Statically typed
- Built-in Design by Contract mechanisms
- Agents: objects encapsulating behavior
- "Once" mechanisms, replacing statics and globals
- Void safety (new!)

Libraries

- Fundamental data structures and algorithms
- Portable graphics
- Internet, Web
- Lexical analysis, parsing
- Database interfaces

Dogmatism and flexibility

Dogmatic where it counts:

- Information hiding (e.g. no \( x, a := b \))
- Overloading
- "One good way to do anything"
- Style rules

Flexible when it makes no point to harass programmers:

- Give standard notations (e.g. \( a + b \)) an O-O interpretation
- Syntax, e.g. semicolon

The Eiffel language: there is a hidden agenda

That you forget it even exists

The environment

EiffelStudio
EiffelStudio: Melting Ice™ Technology

Fast recompilation: time depends on size of change, not size of program

Full type checking

"Freeze" once in a while

Optimized compilation: finalize.

Performance

"Finalization" mode of compilation applies extensive optimizations:

-Inlining
-Dead code removal
-Contract removal
-...

Optimizations are compiler-applied and automatic; no need for manual hacking

Compacting garbage collection takes care of memory issues

Intended to match the most exacting demands of industry applications

Openness

Eiffel can be used as "component combinator" to package elements from different sources:

- Mechanisms for integrating elements in C, C++, Java, CIL (.NET)
- Interfaces and libraries: SQL, XML, UML (XMI), CORBA, COM, others
- Particularly sophisticated mechanisms for C/C++ interfacing
- Outside of .NET, compiles down to ANSI C code, facilitates support for C and C++ easier:
- On .NET, seamless integration with C#, VB .NET etc.

C/C++ support

Functions, macros, include files, setters, getters, constructors, destructors etc.

Inline C

From the outside into Eiffel:

- CECIL (C-Eiffel Common Interface Library)
Portability

Source-code portability across:

- Windows NT, 2000, XP, Vista
- Windows 98, Me
- .NET
- Solaris, other commercial Unix variants
- Linux
- Mac OS X (forthcoming)
- BSD (Berkeley System Distribution)
- VMS

The waterfall model of the lifecycle

Feasibility study
Requirements
Global design
Detailed design
Implementation
V & V
Deployment

Traditional lifecycle model

Rigid model:
- Waterfall: separate tasks, impedance mismatches
- Variants; e.g. spiral, retain some of the problems
Separate tools:
- Programming environment
- Analysis & design tools, e.g. UML
Consequences:
- Hard to keep model, implementation, documentation consistent
- Constantly reconciling views
- Inflexible, hard to maintain systems
- Hard to accommodate bouts of late wisdom
- Wastes efforts
- Damages quality

The Eiffel model

Seamless development:
- Single notation, tools, concepts, principles throughout
- Eiffel is as much for analysis & design as implementation & maintenance
- Continuous, incremental development
- Keep model, implementation and documentation consistent
- Reversibility: go back & forth
- Saves money: invest in single set of tools
- Boosts quality

Seamlessness

Seamlessness Principle
Software development should rely on a single set of notations & tools
Reversibility

Reversibility Principle
The software development process, notations and tools should allow making changes at any step in the process.

The seamless, reversible model

Example classes:
- PLANE
- ACCOUNT
- TRANSACTION
- STATE
- COMMAND
- HASH_TABLE
- TEST_DRIVER
- TABLE

Analysis
Design
Implementation
V&V
Generalization

Analysis classes

defined class VAT
invariant
feature
TANK
in_valve, out_valve : VALVE
fill
require
in_valve.open
out_valve.closed
deferred
ensure
in_valve.closed
out_valve.closed
end

specification
empty, is_full, is_empty, gauge, maximum

is_full = (gauge >= 0.97 * maximum) and (gauge <= 1.03 * maximum)

Single model

Use a single base for everything: analysis, design, implementation, documentation...

Use tools to extract the appropriate views.

Single Model Principle
All the information about a software system should be in the software text.

The seamless, reversible model

Analysis
Design
Implementation
V&V
Generalization

Prepare for reuse:
- Remove built-in limits
- Remove dependencies on specifics of project
- Improve documentation, contracts...
- Abstract
- Extract commonalities, revamp inheritance hierarchy
The cluster model

Mix of sequential and concurrent engineering

Permits dynamic reconfiguration

Tool support for seamless development

- Diagram Tool
  - System diagrams can be produced automatically from software text
  - Works both ways: update diagrams or update text
    - Other view immediately updated
- No need for separate UML tool
- Metrics Tool
- Profiler Tool
- Documentation generation tool
- ...

EiffelStudio diagram tool

Text-graphics equivalence

- Diagram Tool
  - System diagrams can be produced automatically from software text
  - Works both ways: update diagrams or update text
    - Other view immediately updated
- No need for separate UML tool
- Metrics Tool
- Profiler Tool
- Documentation generation tool
- ...

Eiffel mechanisms

- Classes, objects, ...
- Single and multiple inheritance
- Inheritance facilities: redefinition, undefinition, renaming
- Genericty, constrained and unconstrained
- Safe covariance
- Disciplined exception handling, based on principles of Design by Contract
- Full GC
- Agents (power of functional programming in O-O)
- Unrestricted streaming: files, databases, networks...

Equivalence

Equivalence Principle

Textual, graphical and other views should all represent the same model
What is not in Eiffel

- Goto
- Functions as arguments
- Pointer arithmetic
- Special increment syntax, e.g. $x++$, $++x$
- In-class feature overloading

Syntax conventions

Semicolon used as a separator (not terminator)
It’s optional almost all the time. Just forget about it!

Style rules are an important part of Eiffel:
- Every feature should have a header comment
- Every class should have an indexing clause
- Layout, indentation
- Choice of names for classes and features

The class

From the module viewpoint:
- Set of available services (“features”)
- Information hiding
- Classes may be clients of each other

From the type viewpoint:
- Describes a set of run-time objects (the instances of the class)
- Used to declare variables (more generally, entities), e.g.
  \[ x : C \]
- Possible type checking
- Notion of subtype

Information hiding

Information Hiding principle

Every module should have a public specification,
listing a subset of its properties

An object has an interface

An object has an implementation
**Information Hiding**

The designer of every module must select a subset of its properties as the official information about the module, made available to authors of client modules.

**Uniform Access**

Uniform access principle

It does not matter to the client whether you look up or compute

A call such as

```
your_account.balance
```

could use an attribute or a function

**Uniform Access: an example**

```
balance = list_of_deposits, total - list_of_withdrawals, total
```

(A1)

```
list_of_deposits
200 100 500 1000

list_of_withdrawals
800 100 100
```

(A2)

```
list_of_deposits
200 300 500 1000

list_of_withdrawals
800 100 100
```

**POINT: as an abstract data type**

```
x: POINT → REAL

y: POINT → REAL

ρ: POINT → REAL

θ: POINT → REAL
```

In polar representation, ρ and θ are attributes, x and y are routines.
**POINT: as a class**

```plaintext
class POINT feature
  x, y : REAL -- Cartesian coordinates
  move (a, b : REAL)
    do x := x + a
       y := y + b
    end

  scale (factor : REAL)
    do x := factor
       y := factor
    end
end
```

**Uniform access through feature call**

To access a feature of a point, same notation regardless of representation.

Example:

```plaintext
p1.x
```

- Cartesian representation: attribute call
- Polar representation: function call

No difference for clients (except possibly performance)

**Uniform access in practice**

Class `COMPLEX`, switching silently and on demand between cartesian and polar representation

Secret attributes:

```plaintext
  cartesian_uptodate, polar_uptodate: BOOLEAN
```

Representation invariant:

```plaintext
  invariant
  at_least_one: cartesian_uptodate or polar_uptodate
```

**Updating representation: secret routine**

```plaintext
update_cartesian
  require
  polar_ok: polar_uptodate

  do
    if not cartesian_uptodate then
      internal_x := ro * cos(theta)
      internal_y := ro * sin(theta)
    end
  end

  ensure
  cart_ok: cartesian_uptodate
  polar_ok: polar_uptodate
end
```

**Public query**

```plaintext
x: REAL -- Abscissa of current point

  do
    if not cartesian_available then
      update_cartesian
    end
  end

  ensure
  cart_ok: cartesian_uptodate
  same_as_internal: Result = x_internal
end
```
Adding two complex numbers

\[ \text{plus}(\text{other} : \text{COMPLEX}) \]

\[ \begin{align*}
&\quad \text{do} \\
&\quad \quad \text{update}_{\text{cartesian}} \\
&\quad \quad x\_{\text{internal}} := x\_{\text{internal}} + \text{other}.x \\
&\quad \quad y\_{\text{internal}} := y\_{\text{internal}} + \text{other}.y \\
&\quad \text{ensure} \\
&\quad \quad \text{cartesian}\_\text{ok}: \text{cartesian}\_\text{uptodate}
\end{align*} \]

Beyond information hiding

Single choice principle

If a system supports a set of choices, only one of its elements should know the list

Single choice: examples

Graphic system: set of figures
Editor: set of commands
Compiler: set of language constructs

Single choice principle

If a system supports a set of choices, only one of its elements should know the list

Without dynamic binding!

\[ \text{display}(f : \text{FIGURE}) \]

\[ \begin{align*}
&\quad \text{do} \\
&\quad \quad \text{if} \ "f\ is\ a\ \text{CIRCLE}\" \text{ then} \\
&\quad \quad \quad \ldots \\
&\quad \quad \quad \ldots \\
&\quad \text{elseif} \ "f\ is\ a\ \text{POLYGON}\" \text{ then} \\
&\quad \quad \quad \text{end} \\
&\quad \quad \quad \ldots \\
&\quad \quad \text{end}
\end{align*} \]

and similarly for all other routines!

Tedious; must be changed whenever there's a new figure type

With inheritance & associated techniques

With:

\[ f : \text{FIGURE} \]
\[ c : \text{CIRCLE} \]
\[ p : \text{POLYGON} \]

and:

\[ \begin{align*}
&\quad \text{create} \_\text{c.make}(\ldots) \\
&\quad \text{create} \_\text{p.make}(\ldots)
\end{align*} \]

Initialize:

\[ \begin{align*}
&\quad \text{if} \ ... \ \text{then} \\
&\quad \quad f := c \\
&\quad \quad \text{else} \\
&\quad \quad f := p \\
&\quad \text{end}
\end{align*} \]

Then just use:

\[ \begin{align*}
&\quad f\_\text{move}(\ldots) \\
&\quad f\_\text{rotate}(\ldots) \\
&\quad f\_\text{display}(\ldots) \\
&\quad \quad \text{-- and so on for every} \\
&\quad \quad \text{operation on } f
\end{align*} \]

Memory management

Memory management principle

It is the implementation's responsibility to reclaim unused objects
What to do with unreachable objects

Reference assignments may make some objects useless.

Two possible approaches:
- Manual "free" (C++).
- Automatic garbage collection

The C programmer's view

Newsgroup posting by Ian Stephenson, 1993 (as cited in Object-Oriented Software Construction, 2nd edition):

I say a big NO! Leaving an unreferenced object around is BAD PROGRAMMING. Object pointers ARE like ordinary pointers — if you allocate an object you should be responsible for it, and free it when it's finished with. (Didn't your mother always tell you to put your toys away when you'd finished with them?)

Arguments for automatic collection

Manual reclamation is dangerous for reliability.
- Wrong "frees" are among the most difficult bugs to detect and correct.

Manual reclamation is tedious.
Modern garbage collectors have acceptable performance overhead.

GC is tunable: disabling, activation, parameterization....

Properties of a garbage collector (GC)

- Soundness: If the GC reclaims an object, it is unreachable
- Completeness: If an object is unreachable, the GC will reclaim it

Soundness is an absolute requirement. Better no GC than an unsound GC
But: safe automatic garbage collection is hard in C-based languages

Language style

Consistency principle
The language should offer one good way to do anything useful

Compatibility principle
Traditional notations should be supported with an O-O semantics
**Infix and prefix operators**

In

\[
\begin{align*}
\text{a} - \text{b} \\
\text{the - operator is } \text{"infix"} \\
\text{(written between operands)}
\end{align*}
\]

In

\[
\begin{align*}
- \text{b} \\
\text{the - operator is } \text{"prefix"} \\
\text{(written before the operand)}
\end{align*}
\]

---

**The object-oriented form of call**

some_target.some_feature(some_arguments)

For example:

my_figure.display

my_figure.move(3, 5)

\[
x := a + \text{plus}(b)
\]

---

**Operator features**

expanded class INTEGER feature

\[
\begin{align*}
\text{plus alias } + (\text{other : INTEGER}) & : \text{INTEGER} \\
& \text{-- Sum with other} \\
\text{do } & \text{... end} \\
\text{times alias } \ast (\text{other : INTEGER}) & : \text{INTEGER} \\
& \text{-- Product by other} \\
\text{do } & \text{... end} \\
\text{minus alias } {-} & : \text{INTEGER} \\
& \text{-- Unary minus} \\
\text{do } & \text{... end} \\
\end{align*}
\]

Calls such as \text{i plus j} can now be written \text{i + j}

---

**Assignment commands**

It is possible to define a query as

\[
\text{temperature: REAL assign set_temperature}
\]

Then the syntax

\[
x_\text{.temperature} := 21.5
\]

is accepted as an abbreviation for

\[
x_\text{.set_temperature}(21.5)
\]

Retains contracts and any other supplementary operations

---

**Command-query separation**

**Command-Query Separation Principle**

A function must not change its target object’s state

---

**Command-Query separation**

A command (procedure) does something but does not return a result.

A query (function or attribute) returns a result but does not change the state.
Command-Query Separation

Asking a question should not change the answer!

Command-query separation

Command-Query Separation Principle

A function must not change its target object’s state

This principle excludes many common schemes, such as using functions for input (e.g. C's `getint` or equivalent).

Referential transparency

If two expressions have equal value, one may be substituted for the other in any context where that other is valid.

If \( a = b \), then \( f(a) = f(b) \) for any \( f \).

Prohibits functions with side effects.

Also:
- For any integer \( i \), normally \( i + i = 2 \times i \)
- But even if \( \text{getint}() = 2 \), \( \text{getint}() + \text{getint}() \) is usually not equal to 4.

Command-query separation

Input mechanism using EiffelBase

(instead of \( n := \text{getint}() \)):

\[
\text{io.read_integer} \\
\text{n := io.last_integer}
\]

A discipline of development

Reuse Principle
Design with reuse in mind

Typical API in a traditional library (NAG)

```
nonlinear_ode
  (equation_count: in INTEGER;
   epsilon: in out DOUBLE;
   func: procedure
      (eq_count: INTEGER; a: DOUBLE;
       eps: DOUBLE; b: ARRAY[DOUBLE];
       cm: pointer Libtype);
   left_count, coupled_count: INTEGER ...)
```

And so on. Altogether 19 arguments, including:
- 4 in out values;
- 3 arrays, used both as input and output;
- 6 functions, each with 6 or 7 arguments, of which 2 or 3 arrays!
The EiffelMath routine

... Create e and set-up its values (other than defaults) ...

\[ e \text{.solve} \]

... Answer available in \( e, x \) and \( e, y \) ...

The Consistency Principle

Consistency Principle

All the components of a library should proceed from an overall coherent design, and follow a set of systematic, explicit and uniform conventions.

Two components:

- Top-down and deductive (the overall design).
- Bottom-up and inductive (the conventions).

The key to building a library

Devising a theory of the underlying domain

Some of the theory behind EiffelBase

<table>
<thead>
<tr>
<th>Representation</th>
<th>Access</th>
<th>Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FINITE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INFINITE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOUNDED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNBOUNDED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIXED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESIZABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COLLECTION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBSET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DISENABLED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDEXABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CURSOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STRUCTURE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEQUENCE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRAVERSABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HIERARCHICAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHICAGO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINEAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BILINEAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COUNTABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operands and options

Two possible kinds of argument to a feature:

- Operands: values on which feature will operate
- Options: modes that govern how feature will operate

Example (non-O-O): printing a real number

\[ \text{print}(\text{real_value}, \text{number_of_significant_digits}, \text{zone_length}, \text{number_of_exponent_digits}, ...) \]

The number is an operand; format properties (e.g. number of significant digits, width) are options

O-O example:

\[ \text{my_window.display}(x\_position, y\_position, \text{height, width, text, title\_bar\_text, color, ...}) \]

The size of feature interfaces

More relevant than class size for assessing complexity.

Statistics from EiffelBase and associated libraries:

<table>
<thead>
<tr>
<th>Feature Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of features</td>
<td>4408</td>
</tr>
<tr>
<td>Percentage of queries</td>
<td>66%</td>
</tr>
<tr>
<td>Percentage of commands</td>
<td>34%</td>
</tr>
<tr>
<td>Average number of arguments to a feature</td>
<td>0.5</td>
</tr>
<tr>
<td>Maximum number</td>
<td>5</td>
</tr>
<tr>
<td>No arguments</td>
<td>57%</td>
</tr>
<tr>
<td>One argument</td>
<td>36%</td>
</tr>
<tr>
<td>Two arguments</td>
<td>6%</td>
</tr>
<tr>
<td>Three or more arguments</td>
<td>1%</td>
</tr>
</tbody>
</table>
Recognizing options from operands

Two criteria to recognize an option:

- There is a reasonable default value.
- During the evolution of a class, operands will normally remain the same, but options may be added.

Option-Operand separation

Option values:

- Defaults (specified universally, per type, per object)
- To set specific values, use appropriate "setter" procedures

Example:

```
my_window.set_background_color("blue")
```

```
my_window.display
```

Naming (classes, features, variables...)

Traditional advice (for ordinary application programming):

- Choose meaningful variable names!

New and old names for EiffelBase classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
<td>put, enter, item, entry</td>
</tr>
<tr>
<td>STACK</td>
<td>put, push, item, top</td>
</tr>
<tr>
<td>QUEUE</td>
<td>put, add, item, oldest, remove, oldest, remove, delete</td>
</tr>
<tr>
<td>HASH_TABLE</td>
<td>put, insert, item, value, remove, delete</td>
</tr>
</tbody>
</table>

Naming rules

Achieve consistency by systematically using a set of standardized names.

Emphasize commonality over differences.

Differences will be captured by:

- Signatures (number and types of arguments & result)
- Assertions
- Comments

Some standard names

Queries (non-boolean):

- count, capacity
- item
- to_X, from_X

Commands:

- put, extend, replace, force
- wipe_out, remove, prune
- make -- For creation

Queries (boolean):

- writable, readable, extendible, prunable
- is_empty, is_full

-- Usual invariants:

- 0 <= count ; count <= capacity
- is_empty = (count = 0) ; is_full = (count = capacity)