Towards Trusted Components
Bertrand Meyer

Lesson 1: Focusing on reuse

Working hypothesis

For progress in software, focus on high-quality components

A definition

Trusted component
A reusable software element accompanied by a guarantee of quality
Working hypothesis

Where to focus effort for progress in software?

- Tools?  A priori, a posteriori
- Languages?
- Methods, education?
- High quality components

Levels involved

End-user applications

Specialized components

General-purpose components

Compilers & operating systems

What is a software component?

Program element with the following properties:

- Can be used by other program elements ("clients")
- Has an official description sufficient for client authors to use it  
  (information hiding)
- Component authors do not need to know who are the client authors
The two aspects of reuse and their benefits

The consumer view
- Less software to develop: gain productivity
- Facilitate maintenance
- Gain on quality (?): Reliability, efficiency...
- Learn from models, standardize practices

The producer view
- Improve interoperability
- Turn know-how into capital

Classifying components by...

Lifecycle role:
- Analysis
- Design
- Implementation

Abstraction level:
- Functional (subroutine)
- Casual (package)
- Data (class)
- Cluster (framework)
- System (binary comp.)

Flexibility:
- Static
- Dynamic
- Replaceable

Form of use:
- Interface only
- Source only
- Source + hiding

Economics:
- Free
- Purchased
- Rented

In hardware engineering

John Hennessy (Stanford)

"Most of the improvement in the reliability of computer systems has come from improvement in the basic components"

"You’ll see ever increasing portions of the effort devoted to design and verification"
Today's software is often good enough

Overall:
- Works most of the time
- Doesn't kill too many people
- Negative effects, esp. financial, are diffuse

Significant improvements since early years:
- Better languages
- Better tools
- Better practices

Beyond good enough?

Stable economic system:
- Sum of individual optima = Global optimum

Traditional, non-component-based development:
- Individual optimum: Good Enough
- To make software better: consumer is responsible!

Component-based development:
- Consumer & producer both want better components
- Improvements: Producer does the job

From “good enough” to good?

Beyond “good enough”, quality is economically bad
He who perfects, dies

Quality

Choose to release?

Choose to release?
Components and quality

The good news:

Reuse scales up everything

The bad news:

Reuse scales up everything

The opportunity to do things right?

In ordinary development (the construction of applications), programmer perfectionism is often considered a nuisance

In component development, perfectionism is good

*Formula-1 racing* of software engineering
Eiffel library experience

EiffelBase (collection classes), EiffelVision (portable graphics), EiffelNet, EiffelStore, EiffelMath, EiffelLex, EiffelParse

- Strong consistency principles, strict interface & design rules
- Systematic use of Eiffel techniques (genericity, multiple inheritance, inheritance machinery)
- Design by Contract throughout
- Strict design discipline: command-query separation, operand-option separation, taxonomy, uniform access...
- Extensively reused in practice

Trusted Components: how to get there

High road:
- Proofs of correctness
- Assumes source code
- In fact, assumes we write the components ourselves

Low road:
- Focused on commercial components
- Component Certification
- Component Quality Model

Towards a Component Quality Model

A: Acceptance
B: Behavior
C: Constraints
D: Design
E: Extension
Towards a Component Quality Model

A: Acceptance
A.1 Some reuse attested
A.2 Producer reputation
A.3 Published evaluations

B: Behavior
B.1 Examples
B.2 Usage documentation
B.3 Preconditioned
B.4 Some postconditions
B.5 Full postconditions
B.6 Observable invariants

C: Constraints
C.1 Platform spec
C.2 Ease of use
C.3 Response time
C.4 Memory occupation
C.5 Bandwidth
C.6 Availability
C.7 Security

D: Design

E: Extension
Towards a Component Quality Model

A: Acceptance
B: Behavior
C: Constraints
D: Design
  D.1 Precise dependency doc
  D.2 Consistent API rules
  D.3 Strict design rules
  D.4 Extensive test cases
  D.5 Some proved properties
  D.6 Proofs of preconditions, postconditions & invariants
E: Extension

Towards a Component Quality Model

A: Acceptance
B: Behavior
C: Constraints
D: Design
  E.1 Portable across platforms
  E.2 Mechanisms for addition
  E.3 Mechanisms for redefinition
  E.4 User action pluggability
E: Extension

The culture of reuse

From consumer to producer
Management support is essential, including financial
The key step: generalization
A reuse policy

The two principal elements:
- Focus on producer side
- Build policy around a library

Library team, funded by Reuse Tax
Library may include both external and internal components
Define and enforce strict admission criteria

Levels of reusability for a software element

0 - Usable in some program

1 - Usable by programs written by the same author

2 - Usable within a group or company

3 - Usable within a community

4 - Usable by anyone

Nature or nurture?

Two modes:
- Build and distribute libraries of reusable components (business model is not clear)
- Generalize out of program elements

(Basic distinction:
  Program element --- Software component)
Generalization

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

Few companies have the guts to provide the budget for this.

Two keys to component development success

Substance: Rely on a theory of the application domain

Form: Obsess over consistency
- High-level: design principles
- Low-level: style

Eiffelbase hierarchy
**Old and old names for EiffelBase classes**

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

**Design principles in the Eiffel method**

Object technology: Module ≡ Type  
Design by Contract  
Command-Query Separation  
Uniform Access  
Operand-Option Separation  
Inheritance for subtyping, reuse, many variants  
Bottom-Up Development  
Design for reuse and extension  
Style matters

**Typical API in a traditional library (NAG)**

```plaintext
nonlinear_ode
  (equation_count: in INTEGER
   epsilon: in out DOUBLE
   func: procedure
     (eq_count: INTEGER; a: DOUBLE
      e: 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 approach

```e
ORDINARY_DIFFERENTIAL_EQUATION
create e.make("...values ...")
```

```
..solve
-- Answer available in e.x and e.status ...
```

Style rules

- No routine without header comments
- Preconditions always fully expressed
- Postconditions and invariants: the more the better
- Redundancy OK in class invariants (axioms and theorems)
- Standardized layout
- Queries never use verbs!
- Systematic naming conventions
- No exceptions; rules strictly enforced

Feature categories: keeping a class in order

```e
class C
inherit

feature -- Category 1
... Feature declarations ...
feature -- Category 2
... Feature declarations ...
feature -- Category n
... Feature declarations ...

invariant
...
end
```

Class ACCOUNT: `balance, not get_balance`
Feature categories

Standard categories (the only ones in EiffelBase):
- Initialization
- Creation
- Access
- Measurement
- Comparison
- Status report
- Status setting
- Cursor movement
- Element change
- Removal
- Resizing
- Transformation
- Inapplicable
- Implementation
- Miscellaneous
- Conversion
- Duplication
- Basic operations
- Transformation
- Internal

The context

Seamless, reversible development as supported in the Eiffel method

The traditional model

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 and forth
- Saves money: invest in single set of tools
- Boosts quality

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

The cluster model

Mix of sequential and concurrent engineering

Permits dynamic reconfiguration

Summary of lesson 1

**My conjecture:** reuse-based development holds the key to substantial progress in software engineering

Reuse is a culture, and requires management commitment ("buy in")

The process model can support reuse

Generalization turns program elements into software components

A good reusable library proceeds from systematic design principles and an obsession with consistency