From Eiffel and Design by Contract to Trusted Components

Bertrand Meyer

ETH Zürich / Eiffel Software

Budapest, December 2003

My background

- Since 1985: Founder (now Chief Architect) of Eiffel Software, in Santa Barbara. Produces advanced tools and services to improve software quality, based on Eiffel ideas

- Since 2001: Professor of Software Engineering at ETH Zürich

- Also adjunct professor at Monash University in Australia (since 1998)

Software engineering

The collection of processes, methods, techniques, tools and languages for developing quality operational software.

The challenge

- What does it take to bring software engineering to the next level?

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 (configuration management)

Eiffel

- Method, language and environment
- Fully object-oriented; not a hybrid with other approaches
- Focuses on quality, especially reliability, extendibility and reusability
- Emphasizes simplicity
- Used for many mission-critical projects in industry
- International standard in progress through ECMA

Large Eiffel projects in industry

AXA Rosenberg

Boeing

Chicago Board of Trade

AMP Investments

EMC

Lockheed Martin

Environmental Protection Agency

Hewlett Packard

Cap Gemini Ernst & Young

Swedish National Health Board

ENEA

Northrop Grumman

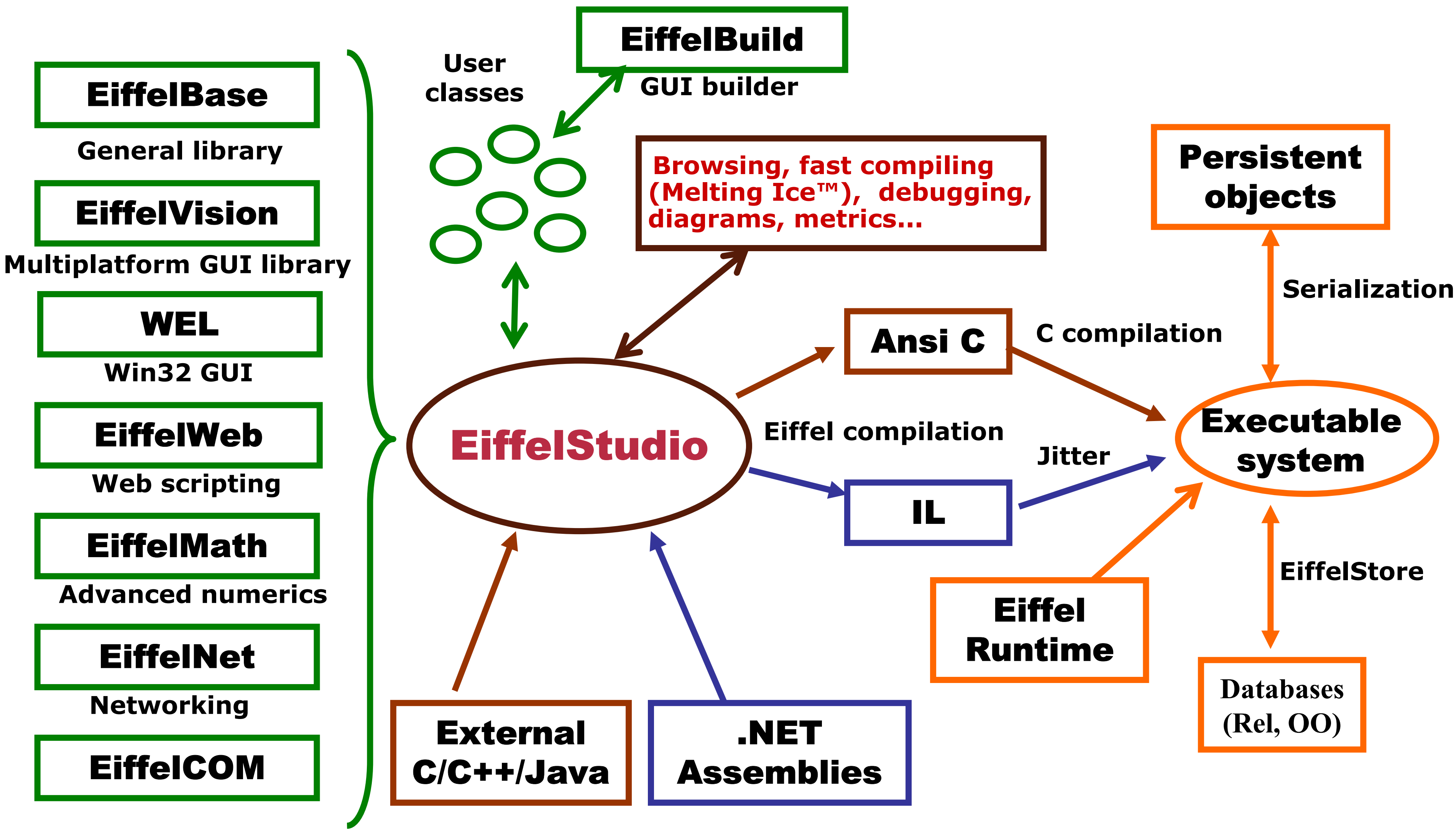
Environment: the two offerings from Eiffel Software

- EiffelStudio ("Classic Eiffel")
  Windows, Unix, Linux, VMS, .NET ...

- ENViSioN! for Visual Studio .NET

Projects are compatible

EiffelStudio

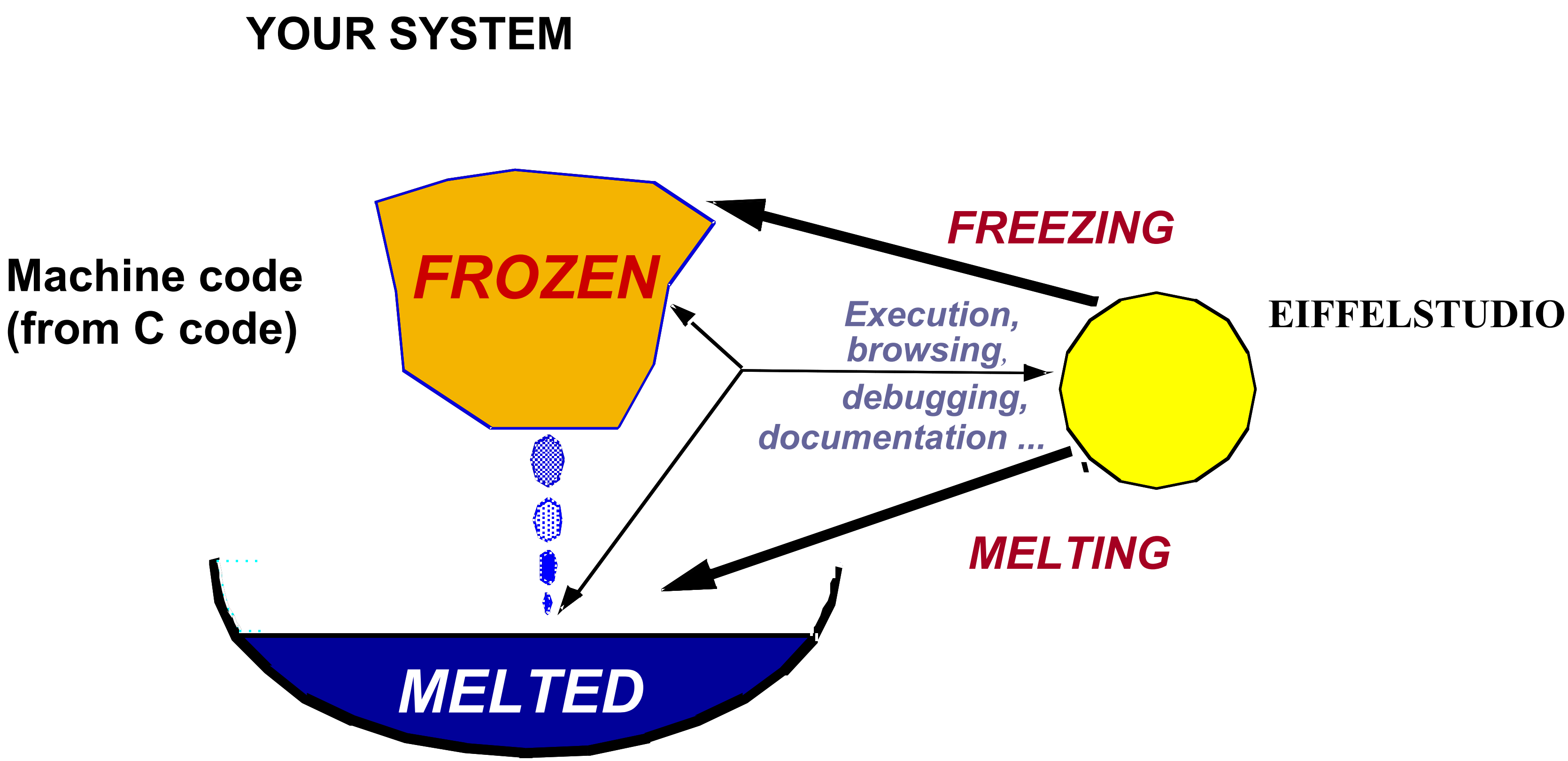


....

EiffelStudio: Melting Ice™ Technology

- Fast recompilation: time depends on size of change, not size of program
- "Freeze" once in a while
- Optimized compilation: finalize.

Melting Ice Technology



Portability

Full source-code portability across:

- Windows NT, 2000, XP
- Windows 98, Me
- Solaris, other commercial Unix variants
- Linux
- BSD (Berkeley System Distribution)
- VMS

Portable graphics

EiffelVision 2 library:

- Simple programming model
- Produce impressive GUI simply and quickly
- Easy to learn
- Completely portable across supported platforms
- Rich set of controls, matches users' most demanding needs
- Adapts automatically to native look & feel

EiffelVision layers

Openness to other approaches

- Extensive mechanisms to support C and C++ constructs
- Java interface
- On .NET, seamless integration with C#, Visual Basic etc.

Special syntax for C/C++ support

```
class
    RECT_STRUCT
feature -- Access
    x (a_struct: POINTER): INTEGER is
            external
                    "C struct RECT access x use <windows.h>"
            end
feature -- Settings
    set_x (a_struct: POINTER; a_x: INTEGER) is
            external
                    "C struct RECT access x type int use <windows.h>"
            end
end
```

Performance

- Optimizations are automatic: Inlining, dead code removal…
- Garbage collection takes care of memory issues
- Performance matches the demand of the most critical industry applications

Eiffel mechanisms

- Classes, objects, ...
- Single and multiple inheritance
- Inheritance facilities: redefinition, undefinition, renaming
- Genericity, 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...

Genericity

Since 1986
(First time genericity & inheritance combined)

Unconstrained

LIST [G]

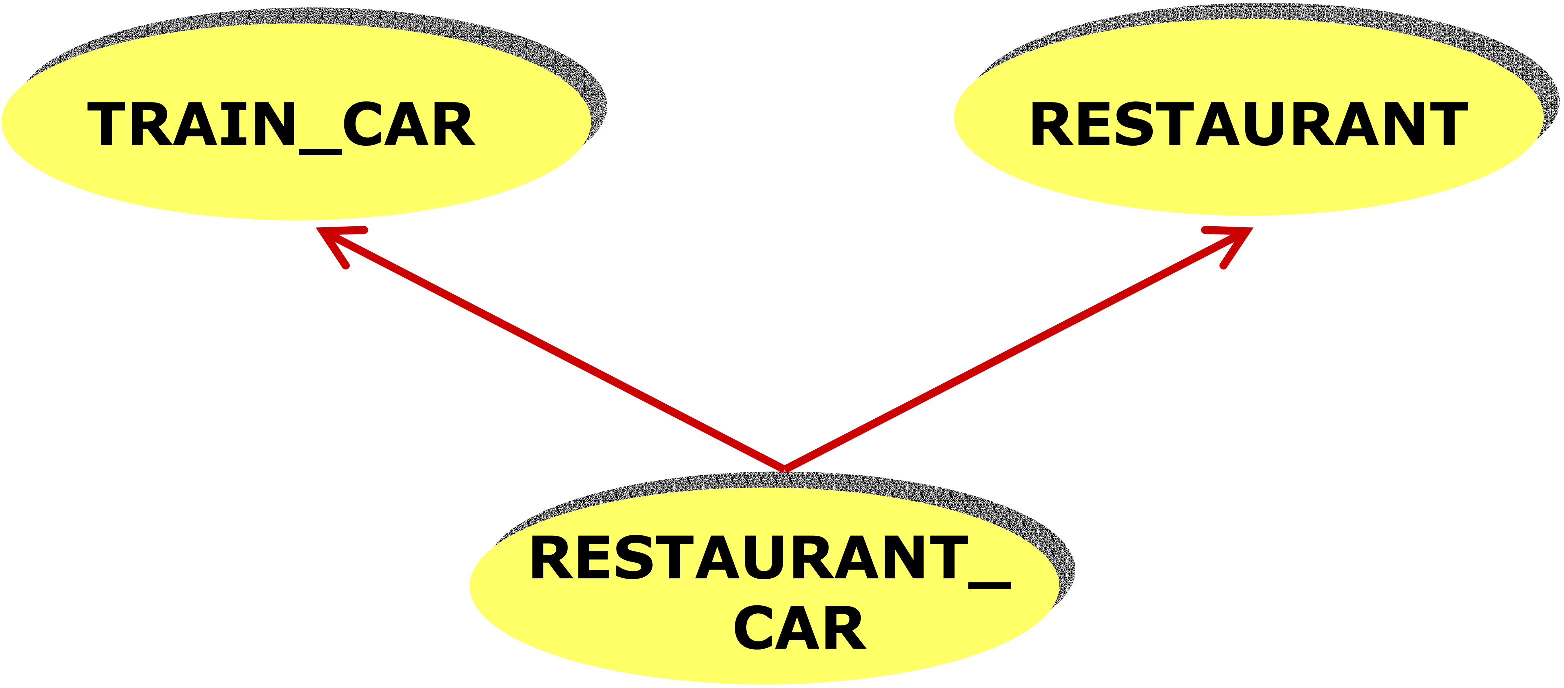
e.g. LIST [INTEGER], LIST [PROFESSOR]

Constrained

HASH_TABLE [G —> HASHABLE]

VECTOR [G —> NUMERIC]

Multiple inheritance



Development: the traditional model

Separate tools:

- Programming environment
- Analysis & design tools, e.g. UML

Consequences:

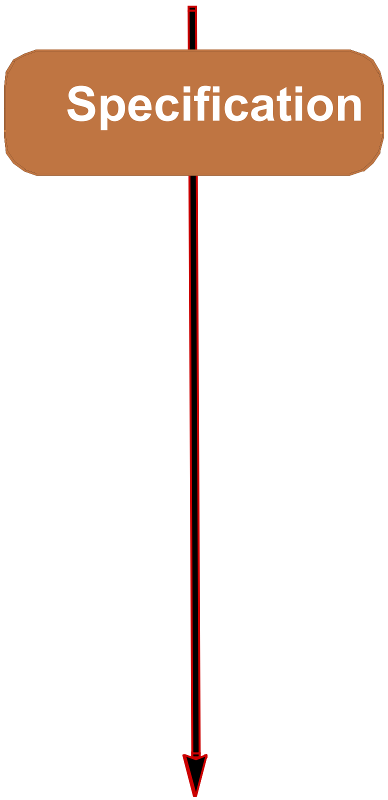
- Hard to keep model, implementation, documentation consistent
- Constantly reconciling views
- Inflexible, hard to maintain systems
- Hard to accommodate bouts of late wisdom
- Wastes efforts
- Damages quality

Development: the Eiffel model

Seamless development:

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

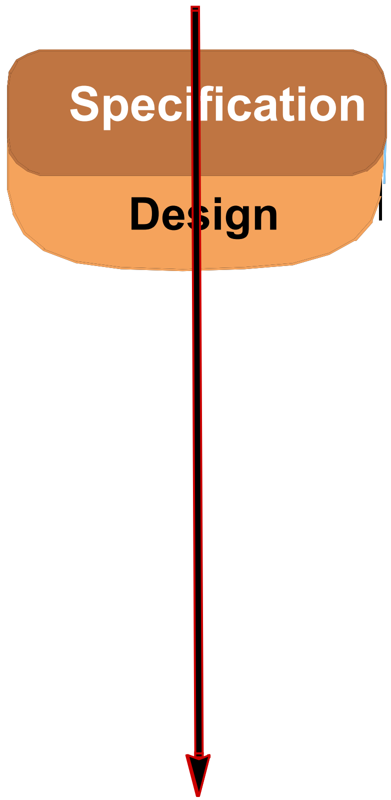
Seamless development (1)



TRANSACTION, PLANE,
CUSTOMER, ENGINE...

Example classes

Seamless development (2)

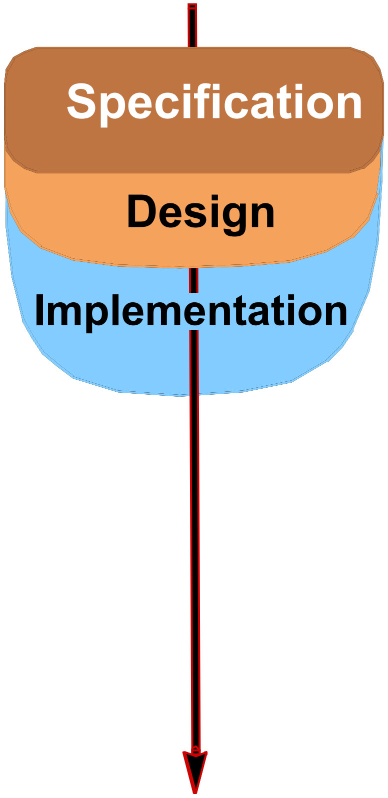


TRANSACTION, PLANE, CUSTOMER, ENGINE...

STATE, USER_COMMAND...

Example classes

Seamless development (3)



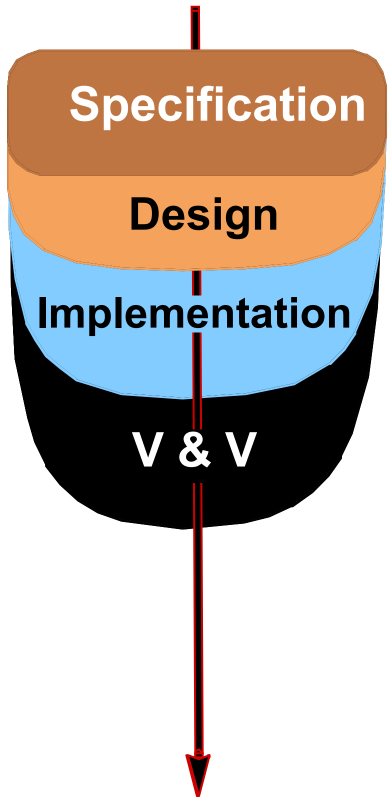
TRANSACTION, PLANE,
CUSTOMER, ENGINE...

STATE, USER_COMMAND...

HASH_TABLE,
LINKED_LIST...

Example classes

Seamless development (4)



TRANSACTION, PLANE,
CUSTOMER, ENGINE...

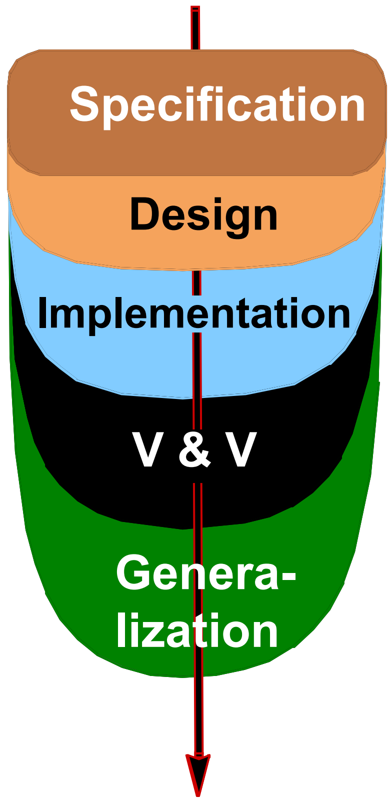
STATE, USER_COMMAND...

HASH_TABLE,
LINKED_LIST...

TEST_DRIVER, ...

Example classes

Seamless development (5)



TRANSACTION, PLANE,
CUSTOMER, ENGINE...

STATE, USER_COMMAND...

HASH_TABLE,
LINKED_LIST...

TEST_DRIVER, ...

AIRCRAFT, ...

Example classes

Eiffel for analysis

```
deferred class VAT inherit
    TANK
feature
    in_valve, out_valve: VALVE
    fill is
                -- Fill the vat.
        require
                in_valve.open
                out_valve.closed
        deferred
        ensure
                in_valve.closed
                out_valve.closed
                is_full
        end
    empty, is_full, is_empty, gauge, maximum, ... [Other features] ...
invariant
    is_full = (gauge >= 0.97 * maximum)  and  (gauge <= 1.03 * maximum)
end
```

Precondition

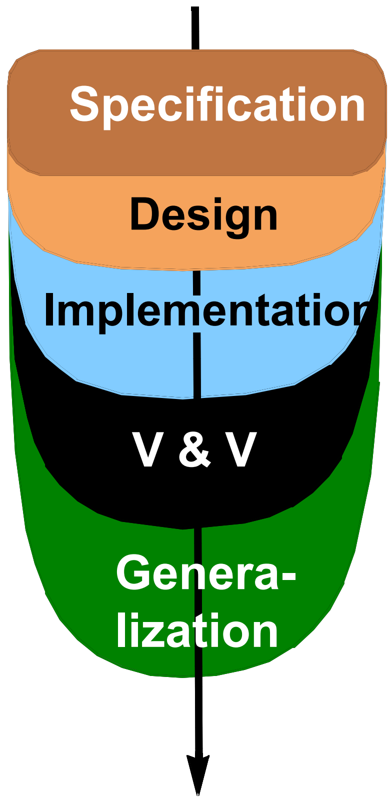
-- Specified only.

-- not implemented.

Postcondition

Class invariant

Seamless development



TRANSACTION, PLANE, CUSTOMER, ENGINE...

STATE, USER_COMMAND...

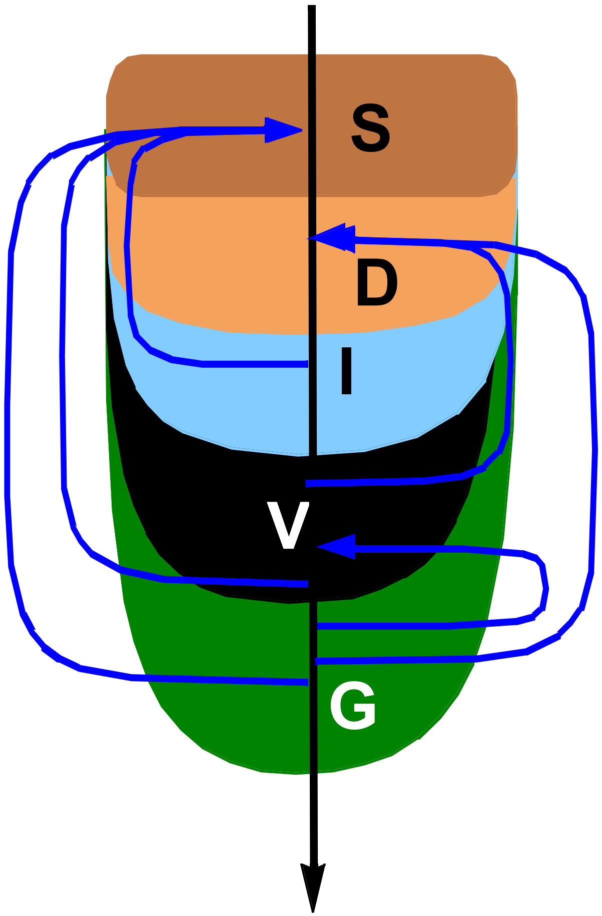
HASH_TABLE, LINKED_LIST...

TEST_DRIVER, ...

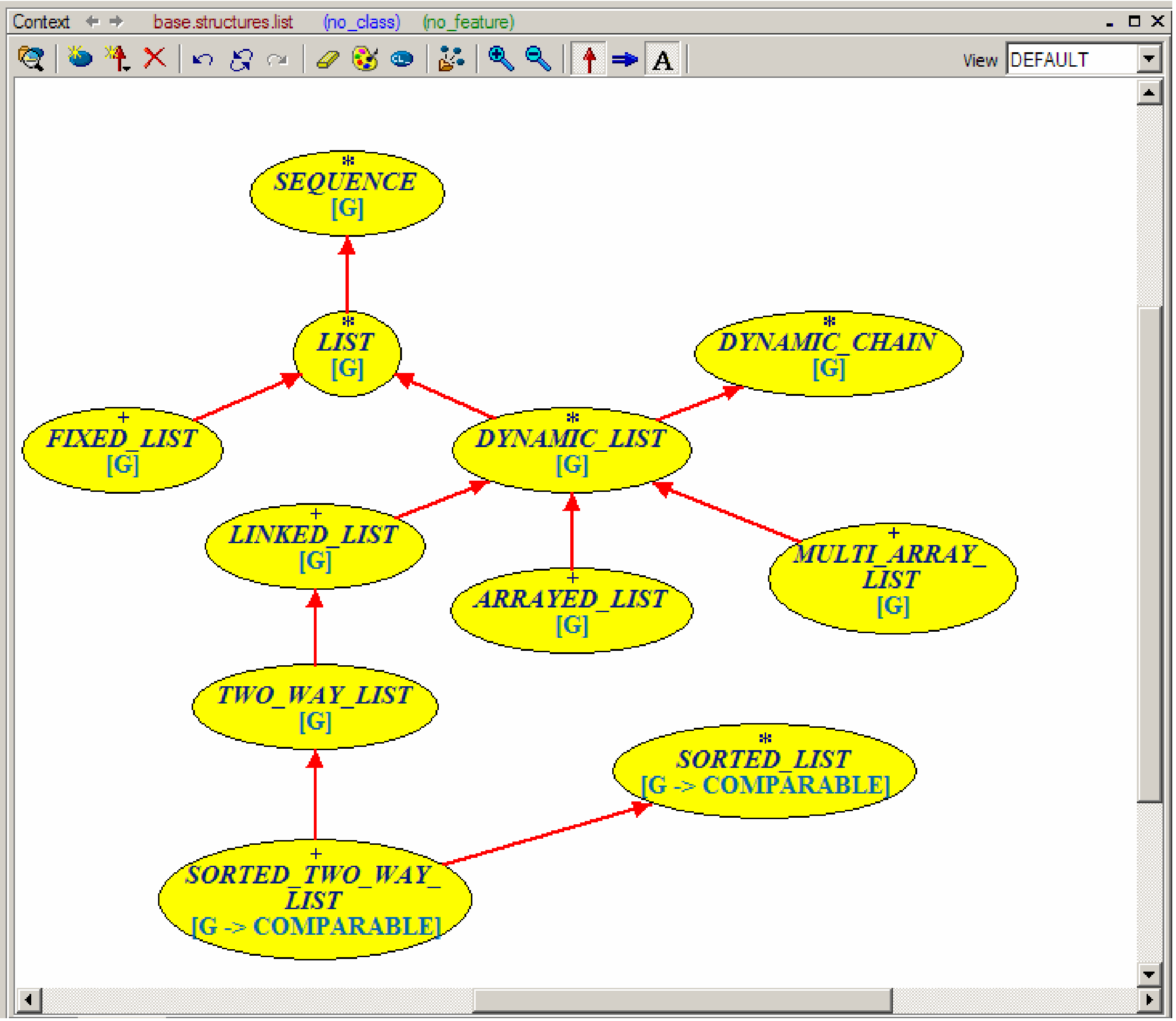
AIRCRAFT, ...

Example classes

Reversibility



Inheritance structure (in EiffelStudio)



Design by Contract™

- Get things right in the first place
- Automatic documentation
- Self-debugging, self-testing code
- Get inheritance right
- Give managers the right control tools

Applications of contracts

- Analysis, design, implementation:
  Get the software right from the start
- Testing, debugging, quality assurance
- Management, maintenance/evolution
- Inheritance
- Documentation

Design by Contract

- A discipline of analysis, design, implementation, management

A view of software construction

- Constructing systems as structured collections of cooperating software elements — suppliers and clients — cooperating on the basis of clear definitions of obligations and benefits.
- These definitions are the contracts.

Design by Contract (cont'd)

- Every software element is intended to satisfy a certain goal, for the benefit of other software elements (and ultimately of human users).
- This goal is the element's contract.
- The contract of any software element should be
  - Explicit.
  - Part of the software element itself.

A human contract

deliver	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) Bring package before 4 p.m.; pay fee.	(From postcondition:) Get package delivered by 10 a.m. next day.
Supplier	(Satisfy postcondition:) Deliver package by 10 a.m. next day.	(From precondition:) Not required to do anything if package delivered after 4 p.m., or fee not paid.

Properties of contracts

A contract:

- Binds two parties (or more): supplier, client.
- Is explicit (written).
- Specifies mutual obligations and benefits.
- Usually maps obligation for one of the parties into benefit for the other, and conversely.
- Has no hidden clauses: obligations are those specified.
- Often relies, implicitly or explicitly, on general rules applicable to all contracts (laws, regulations, standard practices).

A human contract

deliver	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) Bring package before 4 p.m.; pay fee.	(From postcondition:) Get package delivered by 10 a.m. next day.
Supplier	(Satisfy postcondition:) Deliver package by 10 a.m. next day.	(From precondition:) Not required to do anything if package delivered after 4 p.m., or fee not paid.

A class without contracts

```
class
    ACCOUNT

feature -- Access

    balance: INTEGER
                    -- Balance

    Minimum_balance: INTEGER is 1000
            -- Minimum balance

feature {NONE} -- Implementation of deposit and withdrawal

    add (sum: INTEGER) is
                    -- Add sum to the balance (secret procedure).
        do
                    balance := balance + sum
        end
```

Without contracts (cont'd)

```
feature -- Deposit and withdrawal operations

    deposit (sum: INTEGER) is
                    -- Deposit sum into the account.
        do
                    add (sum)
        end

    withdraw (sum: INTEGER) is
                    -- Withdraw sum from the account.
        do
                    add (– sum)
        end

    may_withdraw (sum: INTEGER): BOOLEAN is
                    -- Is it permitted to withdraw sum from the account?
        do
                    Result := (balance - sum >= Minimum_balance)
        end
end
```

Introducing contracts

```
class ACCOUNT

create

    make

feature {NONE} -- Initialization

    make (initial_amount: INTEGER) is
            -- Set up account with initial_amount.
        require
            large_enough: initial_amount >= Minimum_balance
        do
            balance := initial_amount
        ensure
            balance_set: balance = initial_amount
        end
```

Introducing contracts (cont'd)

```
feature -- Access

    balance: INTEGER
                -- Balance

    Minimum_balance: INTEGER is 1000
                -- Minimum balance

feature {NONE} -- Implementation of deposit and withdrawal

    add (sum: INTEGER) is
                -- Add sum to the balance (secret procedure).
        do
                balance := balance + sum
        ensure
                increased: balance = old balance + sum
        end
```

With contracts (cont'd)

```
feature -- Deposit and withdrawal operations

    deposit (sum: INTEGER) is
            -- Deposit sum into the account.
        require
            not_too_small: sum >= 0
        do
            add (sum)
        ensure
            increased: balance = old balance + sum
        end
```

With contracts (cont'd)

```
withdraw (sum: INTEGER) is
            -- Withdraw sum from the account.
    require
            not_too_small: sum >= 0
            not_too_big:
                sum <= balance – Minimum_balance
    do
            add (– sum)
                    -- i.e. balance := balance – sum
    ensure
            decreased: balance = old balance - sum
    end
```

The contract

withdraw	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) Make sure *sum* is neither too small nor too big.	(From postcondition:) Get account updated with *sum* withdrawn.
Supplier	(Satisfy postcondition:) Update account for withdrawal of *sum*.	(From precondition:) Simpler processing: may assume *sum* is within allowable bounds.

The imperative and the applicative

do *balance* := *balance* - *sum*	**ensure** *balance* = **old** *balance* - *sum*
PRESCRIPTIVE	DESCRIPTIVE
How?	What?
Operational	Denotational
Implementation	Specification
Command	Query
Instruction	Expression
Imperative	Applicative

With contracts (end)

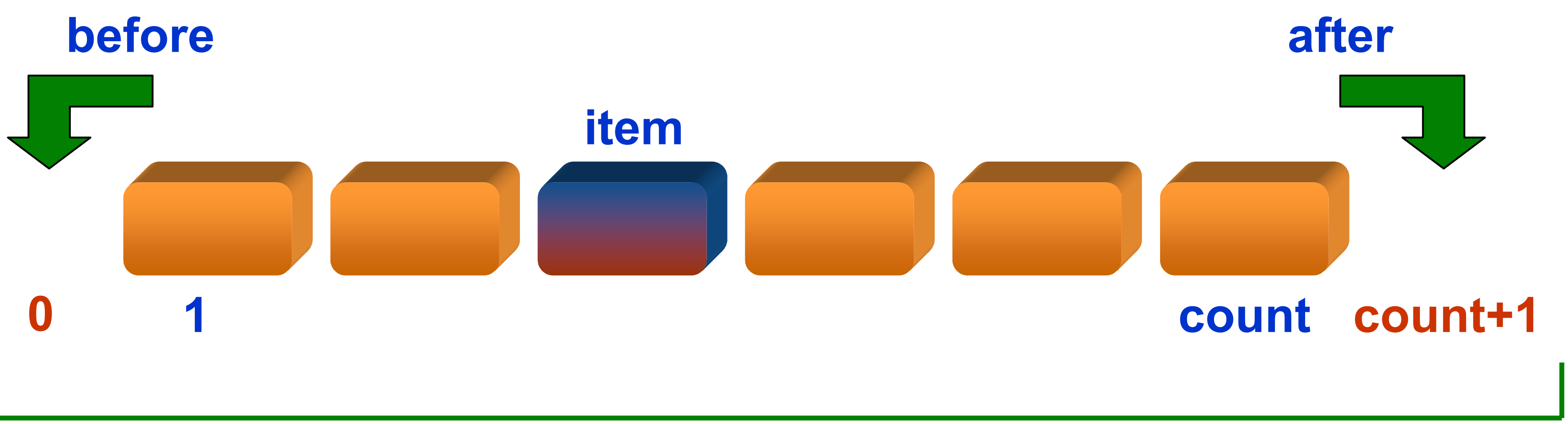
```
may_withdraw (sum: INTEGER): BOOLEAN is
            -- Is it permitted to withdraw sum from the
            -- account?
    do
            Result := (balance - sum >= Minimum_balance)
    end

invariant

    not_under_minimum: balance >= Minimum_balance

end
```

The class invariant

- Consistency constraint applicable to all instances of a class.

- Must be satisfied:
  - After creation.
  - After execution of any feature by any client.
    (Qualified calls only: *a.f* (...))

Lists with cursors



Valid cursor positions

From the invariant of class LIST


```
invariant

    prunable: prunable
    before_definition: before = (index = 0)
    after_definition: after = (index = count + 1)
        -- from CHAIN
    non_negative_index: index >= 0
    index_small_enough: index <= count + 1
```

Valid cursor positions

Applications of contracts

- Analysis, design, implementation:
  Get the software right from the start
- Testing, debugging, quality assurance
- Management, maintenance/evolution
- Inheritance
- Documentation

Contracts and documentation

- Rich documentation produced automatically from class text
- Available in text, HTML, Postscript, RTF, FrameMaker and many other formats
- Numerous views, textual and graphical

Contracts as automatic documentation

Demo

LINKED_LIST Documentation, generated by EiffelStudio

Contracts for analysis

```
deferred class VAT inherit
    TANK
feature
    in_valve, out_valve: VALVE
    fill is
                    -- Fill the vat.
            require
                    in_valve.open
                    out_valve.closed
            deferred
            ensure
                    in_valve.closed
                    out_valve.closed
                    is_full
            end
    empty, is_full, is_empty, gauge, maximum, ... [Other features] ...
invariant
    is_full = (gauge >= 0.97 * maximum)  and  (gauge <= 1.03 * maximum)
end
```

Precondition

-- Specified only.

-- not implemented.

Postcondition

Class invariant

Contracts for testing and debugging

- Contracts express implicit assumptions behind code
- A bug is a discrepancy between intent and code
- Contracts state the intent!
- In EiffelStudio: select compilation option for run-time contract monitoring. Can be set a system, cluster, class level.
- May disable monitoring when releasing software
- A revolutionary form of quality assurance

Contract monitoring

A contract violation always signals a bug:

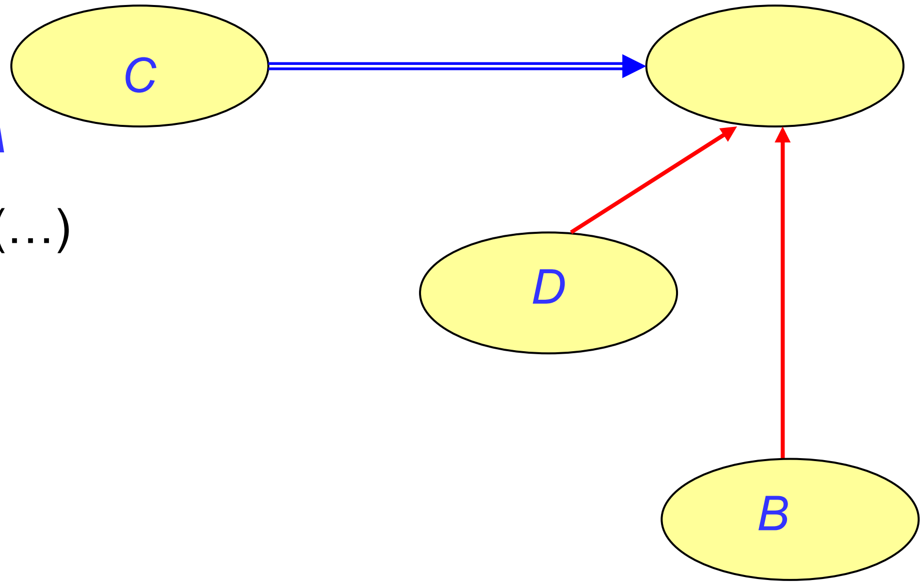
- Precondition violation: bug in client
- Postcondition violation: bug in routine

Contracts and inheritance: invariants

- Invariant Inheritance rule:
    - The invariant of a class automatically includes the invariant clauses from all its parents,

      “and”-ed.
- Accumulated result visible in flat and interface forms.

Contracts and inheritance



a1: *A*

…

a1.*r* (…)

Correct call:

if *a1*.α **then**

a1.*r* (...)

-- Here *a1*.β holds.

end

r **is**

require

α

ensure

β

r **is**

require

γ

ensure

δ

Assertion redeclaration rule

- When redeclaring a routine:
  - Precondition may only be kept or weakened.
  - Postcondition may only be kept or strengthened.

Assertion redeclaration rule in Eiffel

- A simple language rule does the trick!
- Redefined version may have nothing (assertions kept by default), or

  require else *new_pre*

  ensure then *new_post*

- Resulting assertions are:
  - *original_precondition* **or** *new_pre*
  - *original_postcondition* **and** *new_post*

Principles in the Eiffel method

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

Single-model principle

All the information about a system
should be in the system's text

Automatic tools extract various views:

- Interface
- Implementation
- Inheritance structure
- Client-supplier structure
- Operations (features)
- etc.

From Eiffel and Design by Contract...

- ... to Trusted Components

Today’s software is often good enough

Overall:

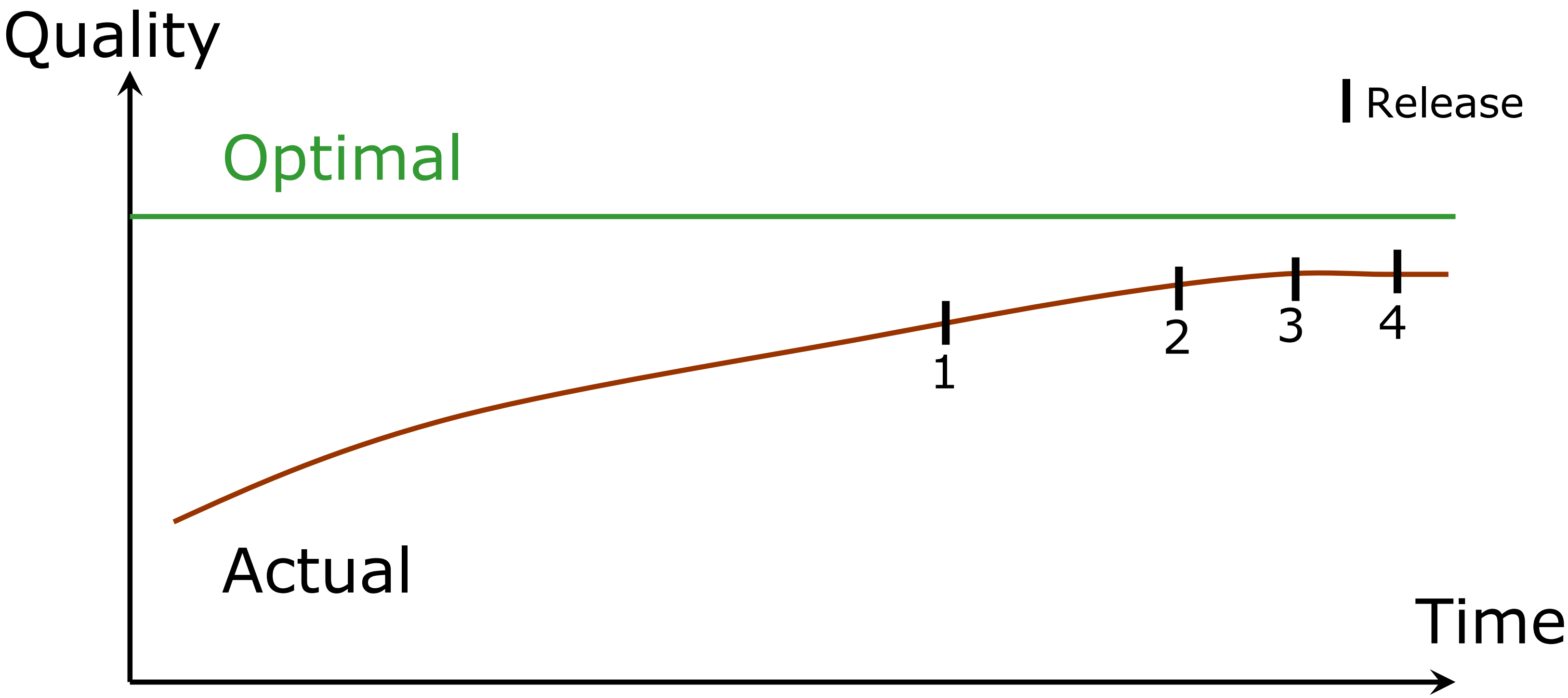
- Works most of the time
- Doesn’t kill too many people
- Negative effects, esp. financial, are diffuse

Significant improvements since early years:

- Better languages
- Better tools
- Better practices (configuration management)

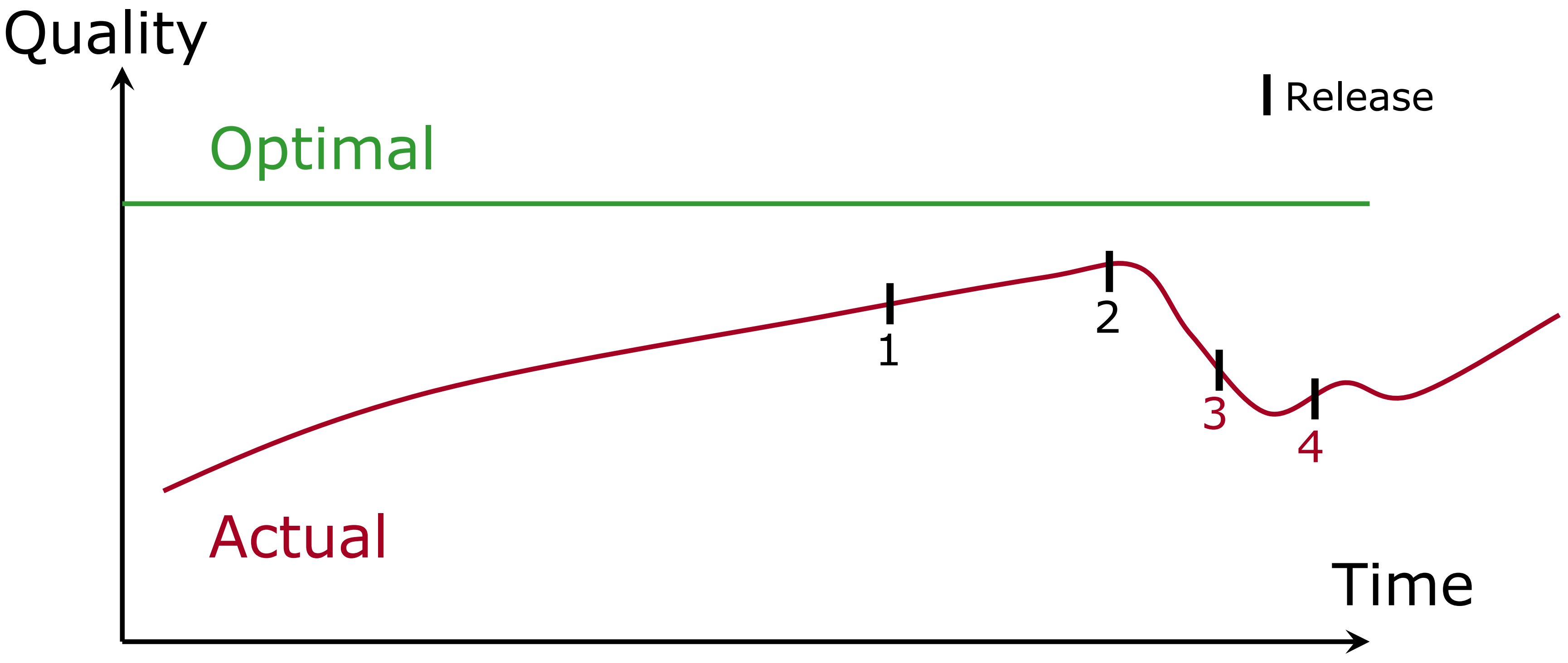
From “good enough” to good?

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



From "good enough" to good?

- Beyond "good enough", quality is economically bad
- He who perfects, dies



The economic argument

- Stable system:
  - Sum of individual optima = Global optimum
- Non-component-based development:
  - Individual optimum = “Good Enough Software”
  - Improvements: I am responsible!
- Component-based development:
  - Interest of both consumer and producer: Better components
  - Improvements: Producer does the job

Quality through reuse

- The good news:

  Reuse scales up everything

Quality through reuse

- The good news:

  Reuse scales up everything

- The bad news:

  Reuse scales up everything

Software design in the future

Component-based for

- Guaranteed quality
- Faster time to market
- Ease of maintenance
- Standardization of software practices
- Preservation of know-how

Trusted components

- Confluence of
  - Quality engineering
  - Reuse

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"

Component quality: the inevitable issue

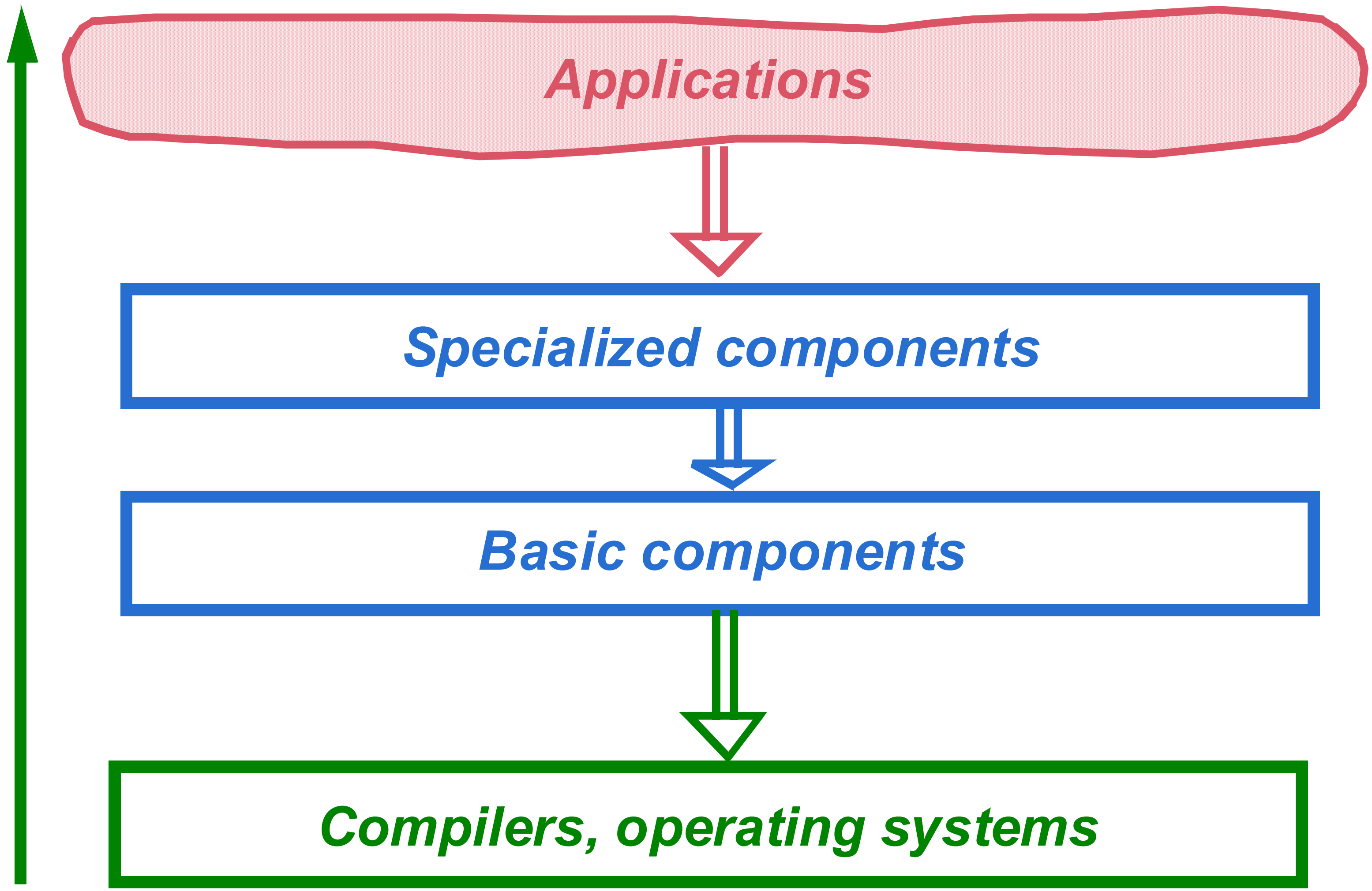
The key issue

- Bad-quality components are major risk

Deficiencies scale up, too

- High-quality components could transform the state of the software industry (if it wanted to — currently doesn't)

Where to focus effort?



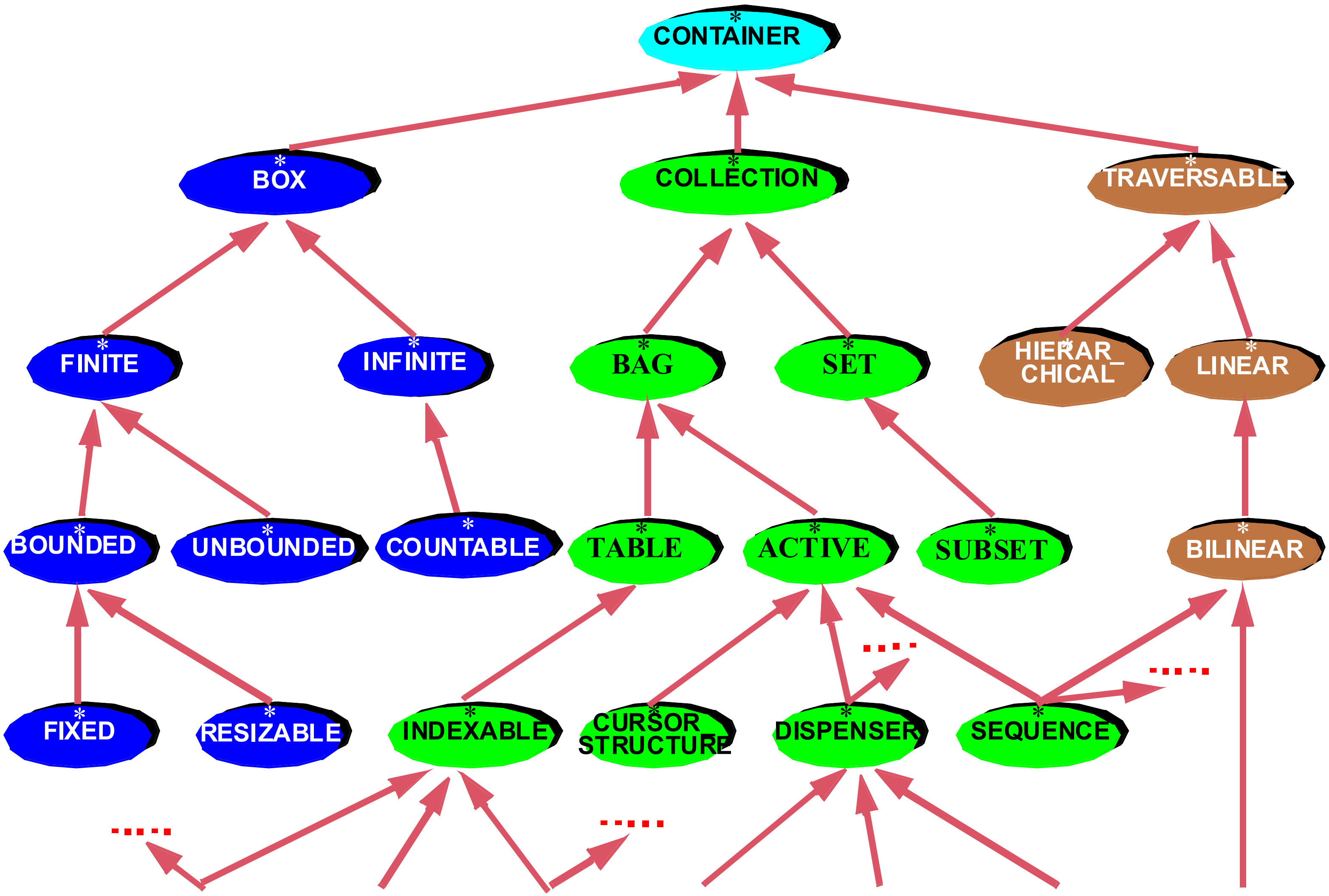
Perfectionism

- Component design should be Formula-1 racing of software “engineering”.
- In component development, perfectionism is good.

Our experience: Eiffelbase

- Collection classes (“Knuthware”)
- Consistency principle
- Strict design principles: command-query separation, operand-option separation, taxonomy, uniform access...
- Strict interface and style rules

Eiffelbase hierarchy



Trusted Components: how to get there

- Low road:
  - Component Certification
    → Component Certification Center
  - Component Quality Model

- High road:
  - Proofs of correctness

A Component Certification Center

- Principles
- Methods and processes
- Standards: Component Quality Model
- Services for component providers and component consumers

Component Quality Model

A: Acceptance

B: Behavior

C: Constraints

D: Design

E: Extension

Component Quality Model

A: Acceptance

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

B: Behavior

C: Constraints

D: Design

E: Extension

Component Quality Model

A: Acceptance

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

D: Design

E: Extension

Component Quality Model

A: Acceptance

B: Behavior

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

Contract levels

1. Type
2. Functional specification
3. Performance specification
4. Quality of Service

(Source: Jézéquel, Mingins et al.)

Component Quality Model

A: Acceptance

B: Behavior

C: Constraints

D: Design

E: Extension

- E.1 Portable across platforms
- E.2 Mechanisms for addition
- E.3 Mechanisms for redefinition
- E.4 User action pluggability

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

Proof technology and formal methods

- Constant advances in recent years

- Most applications: life-critical systems in transportation, defense etc. Example: security system of Paris Metro METEOR line, using the B method

Formal methods and reuse

- Components should be good
- Proofs should be economical!

“Proving classes”

EiffelBase libraries (fundamental data structures and algorithms):

- Classes are equipped with contracts
- “Proving a class” means proving that the implementation satisfies the contracts

Scope of our work at ETH: basics

- Help move software technology to the next level through
  - Trusted Components
  - Advanced O-O techniques
  - Teaching (including introductory)

- Approaches of special interest
  - Eiffel
  - .NET
  - B

Scope of our work at ETH: other

- Journal of Object Technology JOT

  www.jot.fm

- Numerous workshops and conferences

- LASER (Laboratory for Applied Software Engineering Research); summer school starting September 2004

Teaching introductory programming today

- Long, prestigious tradition of teaching programming at ETH
- Ups and downs of high-tech economy
- Widely diverse student motivations, skills
- Some have considerable operational knowledge
    - New forms of development: “Google-and-Paste” programming
- Short-term pressures (e.g. families), IT industry fads
- The “Bologna process”

The objectives

Educate students so that they will:

- Understand today's software engineering.
- Become competent professionals.
- Find work and have a successful career.

“Outside-in”

The key skill that we should convey: abstraction

Teach, don’t preach.

- Start from libraries
- “Progressive opening of the black boxes”, “Inverted Curriculum”
- From programmer to producer
- Not bottom-up or top-down; **outside-in.**

Students are able, right from the start, to “program” impressive and significant applications.

My first program


```
class TOUR inherit
    TRANSPORT
feature
    explore is
            -- Prepare
            -- and animate
            -- route
      do
            Paris.display
            Louvre.spotlight
            Line8.highlight
            Route1.animate
      end
end
```

Text to input

Summary

- Bring every one of your developers to the level of your best developers
- Bring every one of your development days to the level of your best days
- Open, portable, reusable, flexible, efficient

For info

"*Object-Oriented Software Construction*", 2nd edition
Prentice Hall

http://www.eiffel.com

http://se.inf.ethz.ch

http://www.inf.ethz.ch/~meyer