Aims of this course

To provide a survey of

- Reuse and component technology, with a special emphasis on object-oriented approaches
- Techniques for quality components
- Formal methods and proofs
Topics

- Quality issues in software engineering
- Components and the notion of trusted component
- Designing O-O libraries
- Axiomatic Semantics and Program Correctness
- Componentization: turning patterns into Components
- Testing Components
- Fundamentals of Program Analysis
- Model Checking
- Abstract Interpretation
- Proof-Carrying Code
Basic references

Clemens Szyperski, *Component Software*, Addison-Wesley, 1998


Organization

Course page

http://se.inf.ethz.ch/teaching/ws2005/0239/

Assistant: Bernd Schoeller, RZ-J5

Lectures:
- **Monday:** 2 hours
- **Wednesday:** 1 hour -- exercises and applications

All exercises are optional, but will be corrected. They are an important preparation for the exam and the project.

Grading:

- **Oral Exam (end of semester):** 40%
- **Project:** 60%
Project

Trusted Web-Based File Repository

- Stand-alone application (own webserver)
- User management
- Reuse of existing libraries
- Assessment of software quality
- Proof of properties

Would we trust the webserver? Why?
Aims of the course

Introduction to issues of software quality

Introduction to component-based development
“Ariane” paper:


Also read Ken Garlington’s criticism (link in the article) and the official report on the Ariane crash.
PART 1: Introduction

Issues of Software Quality
Software quality: external vs internal

**External** factors: visible to customers

(not just end users but e.g. purchasers)

- *Examples*: ease of use, extendibility, timeliness

**Internal** factors: perceptible only to developers

- *Examples*: good programming style, information hiding

Only external factors count in the end, but the internal factors make it possible to obtain them.
Software quality: product vs process

**Product**: properties of the resulting software

For example: correctness, efficiency

**Process**: properties of the procedures used to produce and “maintain” the software
External quality factors

Product quality (immediate):
- Correctness
- Robustness
- Security
- Ease of use
- Ease of learning
- Efficiency

Product quality (long-term):
- Extendibility
- Reusability
- Portability

Process quality:
- Timeliness
- Cost-effectiveness

Diagram:
- Specification
- Errors
- Hostility
- Correctness
- Correctness
- Security
Correctness:
The systems’ ability to perform according to specification, in cases covered by the specification

Robustness:
The systems’ ability to perform reasonably in cases not covered by the specification

Security (integrity):
The systems’ ability to protect itself against hostile use
Non-quality

September 1997: missile Cruiser **USS Yorktown**, “dead in the water” for two hours and 45 minutes, due to a divide by zero in Windows NT.

**Ariane 5** ESA rocket launcher

**Therac-25**

**London Ambulance System**

**Year 2000** (see Christopher Creele, BM and Philippe Stephan, IEEE Computer, November 1997)
Ariane 5, 1996

$500 million, not insured.

40 seconds into flight, exception in Ada program not processed; order given to abort the mission.

Exception was caused by an incorrect conversion: a 64-bit real value was incorrectly translated into a 16-bit integer.

- Not a design error.
- Not an implementation error.
- Not a language issue.
- Not really a testing problem.
- Only partly a quality assurance issue.

Systematic analysis had “proved” that the exception could not occur – the 64-bit value (“horizontal bias” of the flight) was proved to be always representable as a 16-bit integer!
Ariane-5 (Continued)

It was a REUSE error:

- The analysis was correct – for Ariane 4!
- The assumption was documented – in a design document!

With assertions, the error would almost certainly (if not avoided in the first place) detected by either static inspection or testing:

```plaintext
integer_bias (b: REAL): INTEGER is
    require
        representable (b)
    do
        ...
    ensure
        equivalent (b, Result)
end
```
Ariane 5 (Conclusion)

The main lesson:

Reuse without a contract is sheer folly

See:
Jean-Marc Jézéquel and Bertrand Meyer

*Design by Contract: The Lessons of Ariane*

IEEE Computer, January 1997

Also at [http://www.eiffel.com](http://www.eiffel.com)
US software industry, 1998

Source: Standish report

Project leaders and CIOs representing several thousand software projects

Project outcome:

- 28% failure (1994: 31%)
- 27% success (1994: 16%)
- Rest: completed over budget, over time, under features

Smaller projects have a higher chance of succeeding
Monetary effect on Developers and User due to “insufficient testing infrastructure”:

$59.5$ billion

(Financial sector: $3.3$ billion, auto/aerospace $1.8$ billion etc.)
Buffer overflow

(Morris worm, most viruses)

See http://www.cert.org

Some_innocuous_public_command “Some message”
(Or maybe just inputting text into a browser field)
Buffer overflow

Memory Setup

0 → Max

Program Heap Stack

Stack frames

Stack growth

Rout_n Rout_1 Main

Stack top Stack bottom
Calling a routine

Program Heap Stack

Locals of Rout Args of Rout Rout\_n \ldots Rout\_1 Main

Return address

0 Max

Trusted Components, Winter Semester 2005/2006
Calling a utility

syslogd "Some error message"

finger Some_name

some_command "some text"

(Text input into some browser field)
Allocating the buffer

Program  Heap  Stack

0  Other locals  Max

Buffer  Args of Rout  Rout\textsubscript{n}  \ldots  Rout\textsubscript{1}  Main

Return address
How was the routine coded?

(1)

```
from i := 1 until i > input_size
    loop
        buffer [i] := input [i]
        i := i + 1
    end
```

(2)

```
from i := 1 until i > input_size or i > buffer_size
    loop
        buffer [i] := input [i]
        i := i + 1
    end
```
Allocating the buffer

Program  Heap  Stack

Buffer  Args of Rout  Rout_1  Main  Stack frames

0  Max

Other locals

Return address

Return address
Getting close

Stack frames

Program Heap Stack

Other locals

Buffer

Return address

Max
Getting closer

Program Heap Stack

Other locals

Buffer

Stack frames

Return address

Available!

Rout_n ... Rout_1 Main
Inserting the code

Diagram showing the memory layout with Program, Heap, and Stack sections. The Stack frames contain the buffer, modified return address, and your code. Other locals are also shown.
Buffer overflow: lessons

Lack of specification
Lack of specification enforcement
Programming techniques
Security concepts

At the core, a programming methodology issue
Software quality (through technology)

- **A priori (build it right)**
  - Object technology, formal development
- **A posteriori (validate and fix it)**
  - Testing, abstract interpretation, model checking
Management aspects

Process standards: Capability Maturity Model, ISO 9001
Get software in source from, benefit from public scrutiny
Metrics collection and application
Code reviews?