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

Choose to release?
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
Trusted components

Confluence of

- Quality engineering
- Reuse
Approaches: components

Reuse, components, COTS (Commercial Off-The-Shelf), CBD (Component-Based Development)

Component experience:
O-O libraries: Smalltalk, Eiffel, Java, STL, ...
Binary components

Binary component standards:
CORBA
COM/DCOM
Enterprise Java Beans
.NET component model
The conjecture behind this course

A breakthrough is necessary in software quality (and more generally in software engineering)

Only components can provide this breakthrough

Components raise major quality issues on their own
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

Reuse: consumer view, producer view
What exactly is a component?

Working definition:

Program element such that:

- It may be used by other program elements (not just humans, or non-software systems). These elements will be called “clients”

- Its authors need not know about the clients.

- Clients’ authors need only know what the component’s author tells them.
Classifying components by...

Lifecycle role:
- Analysis
- Design
- Implementation

Flexibility:
- Static
- Dynamic
- Replaceable

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

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

Economics:
- Free
- Purchased
- Rented
This is a broad view of components

Encompasses patterns and frameworks

Software, especially with object technology, permits “pluggable” components (“don’t call us, we’ll call you), where client programmers can insert their own mechanisms.

Supports component families
Patterns are both one of the greatest advances in software engineering, and a step backwards from the push for reuse through object technology.

We should try to turn successful patterns into components!

See Karine Arnout’s thesis.
Component quality

The key issue

Bad-quality components are a major risk

Deficiencies scale up, too

High-quality components could transform the state of the software industry (if it wanted to – currently doesn’t)
Component Source: calendar

Home page
Product catalog
Calendar components
Calendar component documentation home
Calendar component operations
Perfectionism

Component design should be Formula-1 racing of software “engineering”.

In component development, perfectionism is good.