German National Research Center for Information Technology
Research Institute for Computer Architecture and Software Technology

Scenario Adapters: Efficiently Adapting Components

Antônio Augusto Fröhlich
Wolfgang Schöder-Preikschat

guto@first.gmd.de
wosch@ivs.cs.uni-magdeburg.de

July 2000

Outline

- Introduction
  - Component-based software engineering
- Adaptable abstractions
  - Scenario independence
  - Component granularity
- Scenario Adapters
- EPOS: Embedded Parallel Operating System
- Discussion

Component-based Software Engineering

Object-oriented
+
Component-based
= Software Assembly Line

- Components are for reuse
- Software components can also be adapted to new scenarios

Component

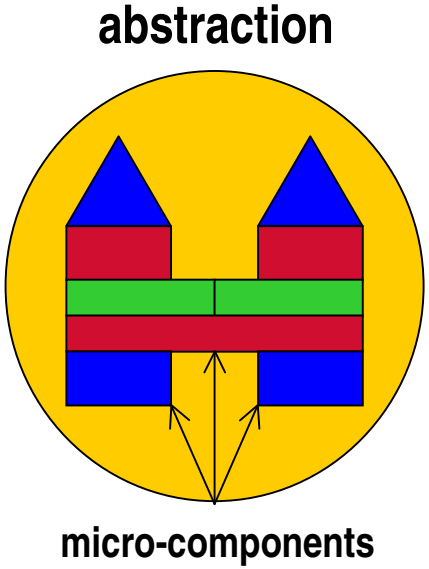
- No clear definition
  - Module, class, class category, object file, server ...
- Oxford dictionary

  “any of the parts of which something is made”
- Absence of definition does not prevent use
- Some open questions
  - What is a good size for a component?
  - How can a component be arranged together?
  - How can a component be adapted?
  - Is semantic preserved after composition?

GMD FIRST
German National Research Center for Information Technology
Research Institute for Computer Architecture and Software Technology

Adaptable Abstractions

- Our concept of component
  - "Any abstraction in the system's application domain"
    - Application-ready
    - Scenario independent
- Open component architecture
  - "Any kind of software part"
- Examples
  - **`thread, mailbox, file`**



Scenario Independence

- Reusability
  - Scenario-independent abstractions can be adapted to join several scenarios
  - New scenario => new “glue” + old abstraction
- Hard to achieve
  - Proper software analysis and design can yield scenario-independent components
  - Components are usually polluted with target scenario aspects during implementation

Scenario Independence

- Mailbox: an example
  - Analysis: *n*-to-*n*, bi-directional communication
  - Design: message sending and receiving, identification, location and synchronization
  - Implementation: network architecture, protocols, buffering, security …
- Scenario dependence increases complexity
  - Each interface gets several scenario-specific realizations
- A development guide-line

Component Granularity

e.g. file system
Do I have to take this junk as well?
The component I need is not there!
a few large components

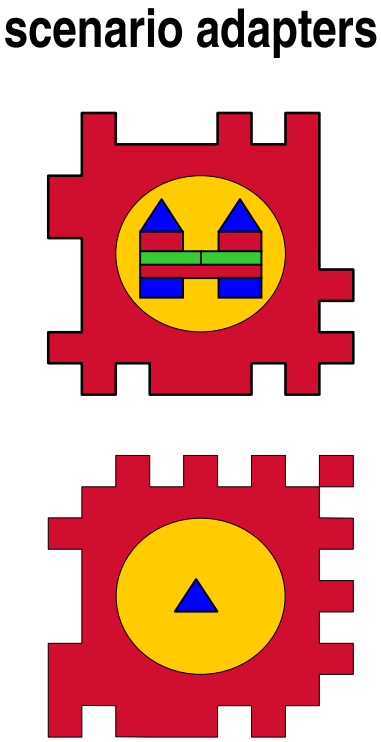
a lot of small components
What is this component for?
What is the difference to that other one?
How do I use it?
e.g. queue container

Component Adaptability

- Ability to play in different scenarios
- A “noble” attribute
  - High cost is accepted
  - Prevents use with high-performance computing
- Dynamic X Static
  - Overhead versus flexibility
  - Is the scenario known before run-time?
    - Static adaptability with low overhead and no flexibility loss

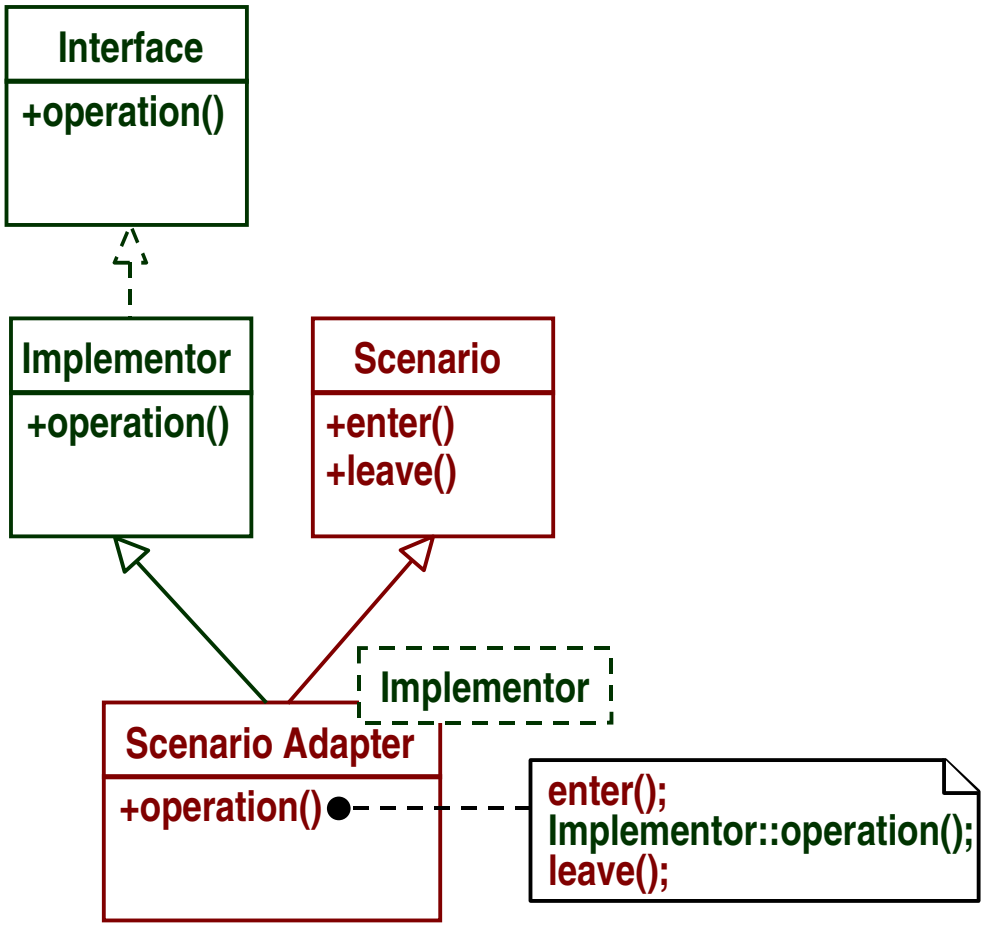
Scenario Adapters

- Adapt existing abstractions to a given scenario
- Adapt system micro-components to grant the semantics dictated by a given execution scenario
- Examples:
  - An SMP adapter
  - A secure remote invocation adapter



Scenario Adapters

- Resembles the **`Adapter`** design pattern
- Several implementors realize the interface
- Static metaprogrammed
  - At compile-time
  - No polymorphism
  - No run-time overhead
- Scenario and implementor are given as parameters to the scenario adapter



Scenario Adapters Implementation in C++

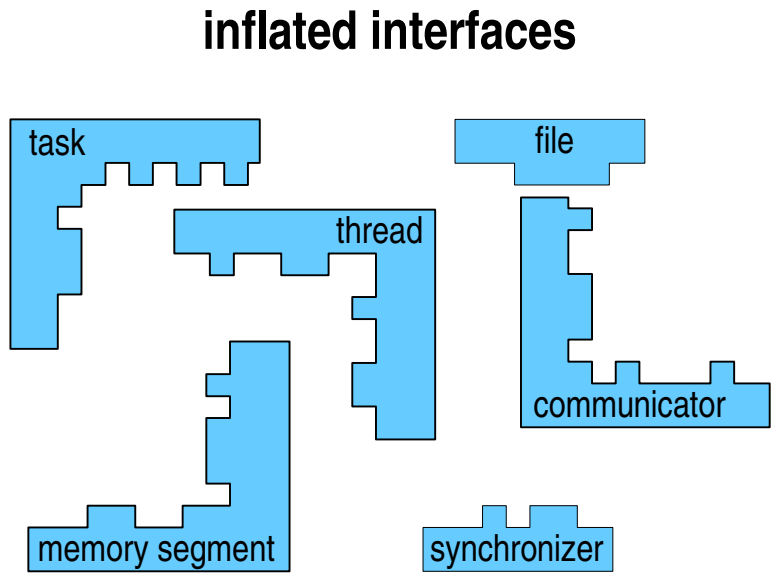
- Interfaces
  - Pure class declaration
  - Particular namespace
- Scenario adapters
  - Class templates
  - Inline methods
- Excellent performance
  - If no adaptation is needed, a direct call is generated
  - If adaptation is needed, the call is surrounded by scenario specific primitives

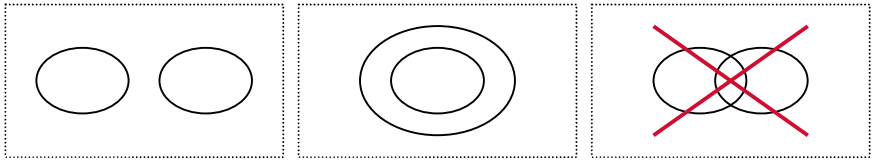
EPOS: Embedded Parallel Operating System

- EPOS operating system
  - Application-driven assemblage of system abstractions
- EPOS abstractions
  - Scenario-independent, statically configurable, application-ready
- EPOS and the real world
  - Intelligent visual tools for configuration
  - Invisibility
    - Standard libraries (Posix files, libc, libstdc++, libm)
    - Standard APIs (Posix threads, MPI)

Inflated Interfaces

- Export system abstractions to applications
- Well-known to application programmers
- Comprehensive
- Promote requirement analysis
- Examples:
  - thread
  - communicator
  - synchronizer
- Not a fat interface
  - if intersection => subset





Framework

- Composition of scenario adapters
  - Static metaprogram
  - Implemented as C++ templates
- System abstraction implementation
  - Wrapped
    - No allocation or sharing control
    - No cross-domain invocation
    - Always embedded in the application
  - Controlled
    - Allocation and sharing control is possible
    - Cross-domain invocation is possible
    - Embedded in the application or packed in a kernel

GMD FIRST
German National Research Center for Information Technology
Research Institute for Computer Architecture and Software Technology

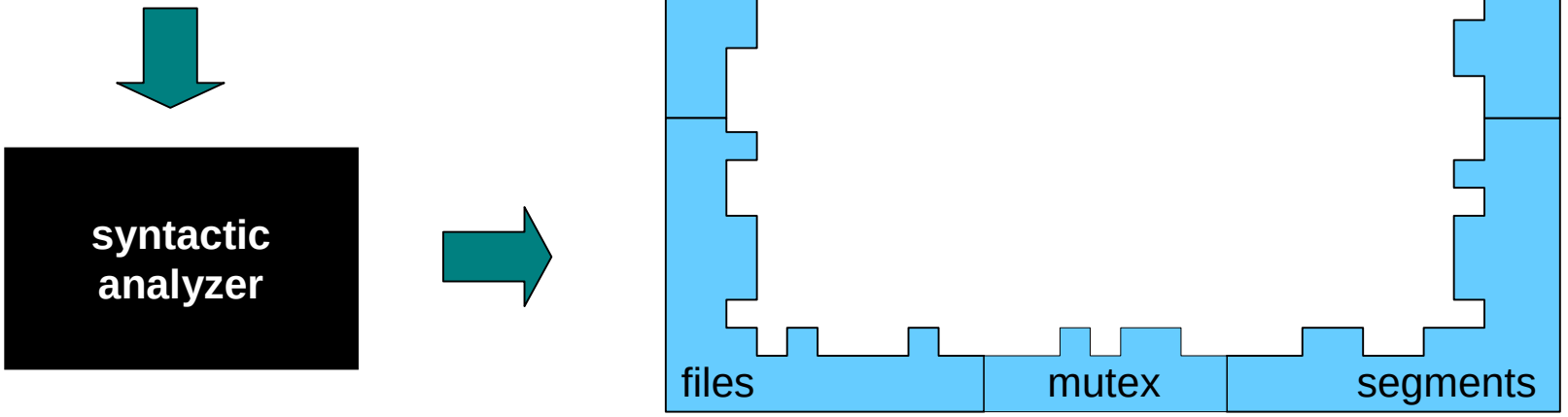
Automatic System Generation

- Application refers to inflated interfaces
- Requirement analyzer parses the application searching for inflated interface references
  - Which interfaces are referred to?
  - How they are referred to?
- Expert system select the realizations that better match the referred inflated interfaces
- An application-oriented operating system is compiled

Requirement Analysis

complex application program

```
code = new Segment(buffer, size);
task = new Task(code, data);
thread = new Thread(task, &entry_func,
      priority, SUSPENDED, arguments);
mutex->entry();
Mailbox mailbox;
mailbox >> message;
File file(name);
file << mailbox;
```

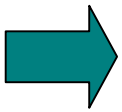


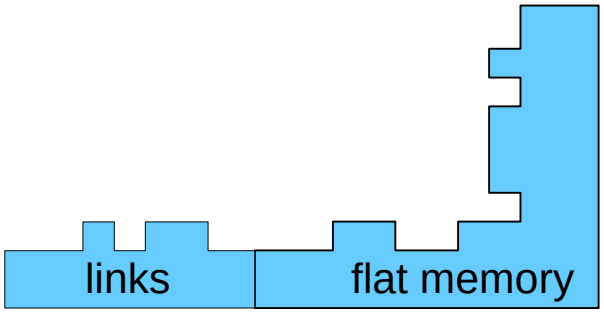
Requirement Analysis

real application program (MPI)

```
Channel link(destination);
link << mesage;
```

syntactic analyzer



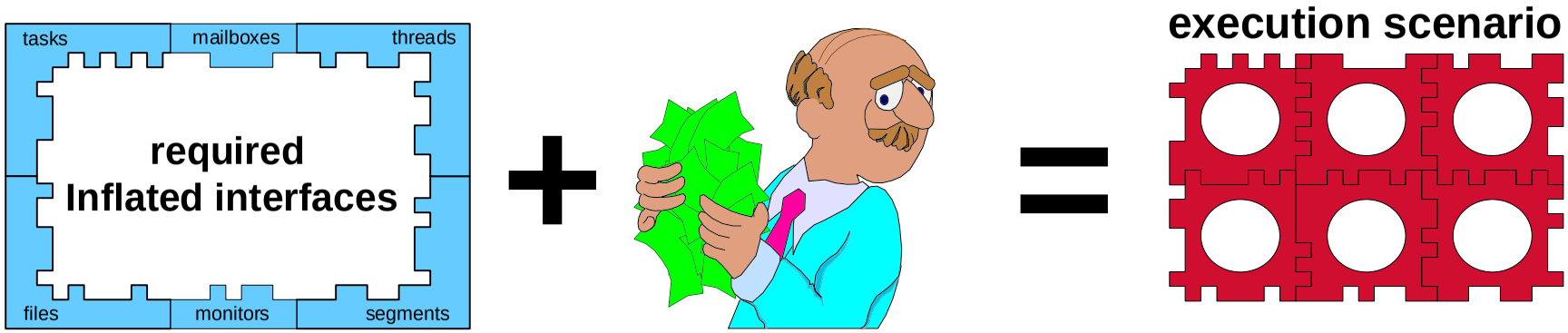


GMD FIRST
German National Research Center for Information Technology
Research Institute for Computer Architecture and Software Technology

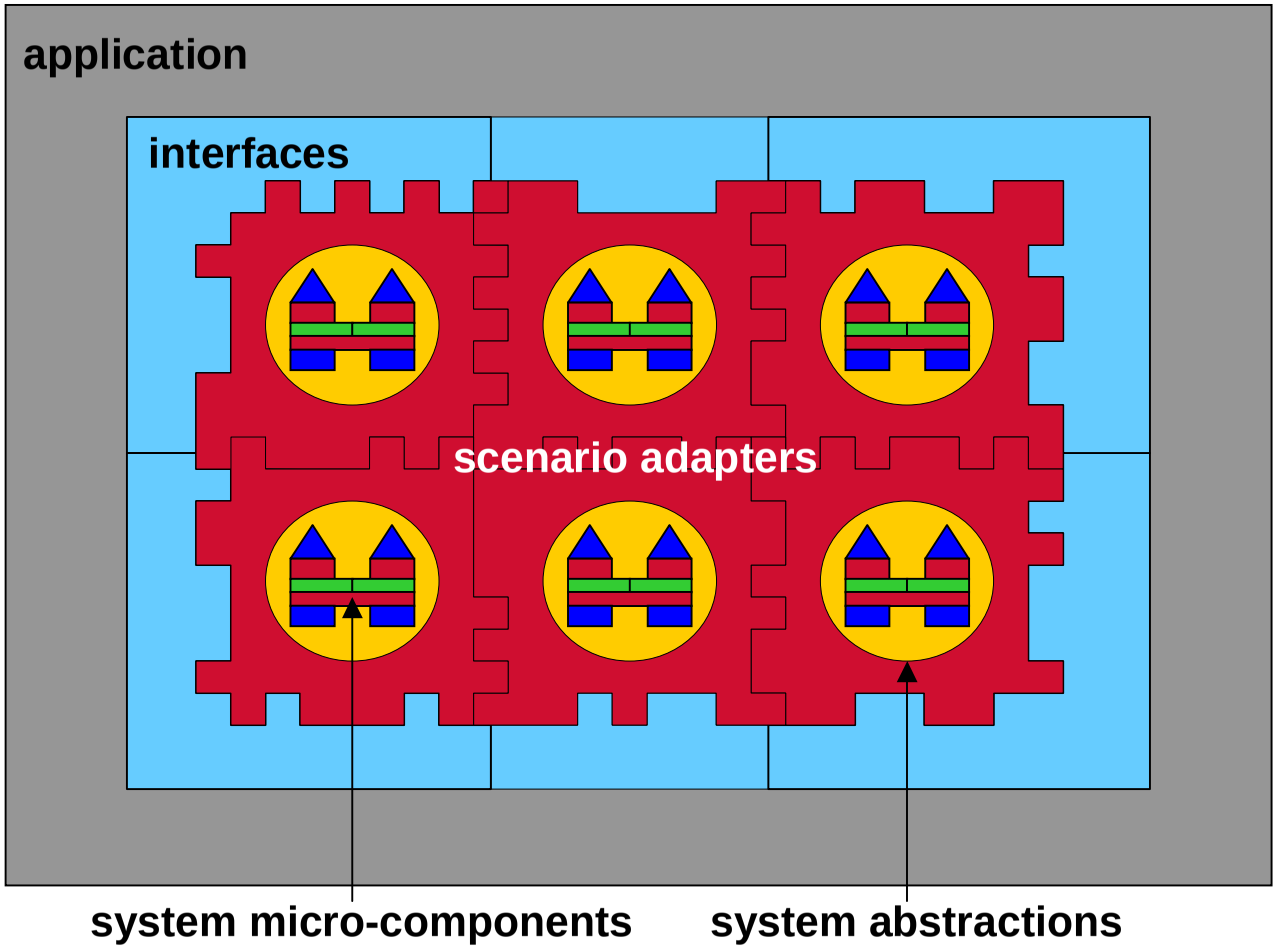
Context Information

- We need context information
  - `new Thread(task, &func)` => location?
  - `mailbox << message` => protocol?
  - Absence of `Task` => single-task?
- Buildup databases
  - Scenario dependencies
  - Target machine description

Execution Scenario



Application-oriented EPOS



Discussion

- Very good performance
  - No run-time overhead
- Flexibility comparable to dynamic techniques when the scenario is known beforehand
- Compared to other generative techniques
  - No special languages
  - No special tools
  - Semantic preservation
- Template metaprograms are hard to develop and maintain