From Patterns to Components

Karine Arnout

Ph.D. defense, ETH Zurich

Assumption and conjecture

Assumption:

- It is better to reuse than to redo
  (even to redo with the help of a model)

Conjecture:

- Many design patterns can be turned into reusable components

Main contributions

- Pattern componentizability classification
- Pattern Library
- Pattern Wizard

Main contributions

- Pattern componentizability classification
- Pattern Library
- Pattern Wizard

Componentization: a definition

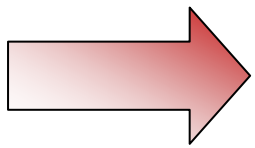
Process of devising a reusable component that provides a ready-made implementation of a design pattern directly usable by any client application.

A design pattern is given by one or more of

- A description of the pattern's intent
- Use cases
- A software architecture for typical implementations

Componentization mechanisms

- Client-supplier relationship
- Simple inheritance
- Multiple inheritance
- Unconstrained genericity
- Constrained genericity
- Design by Contract
- Automatic type conversion
- Agents
- Aspects



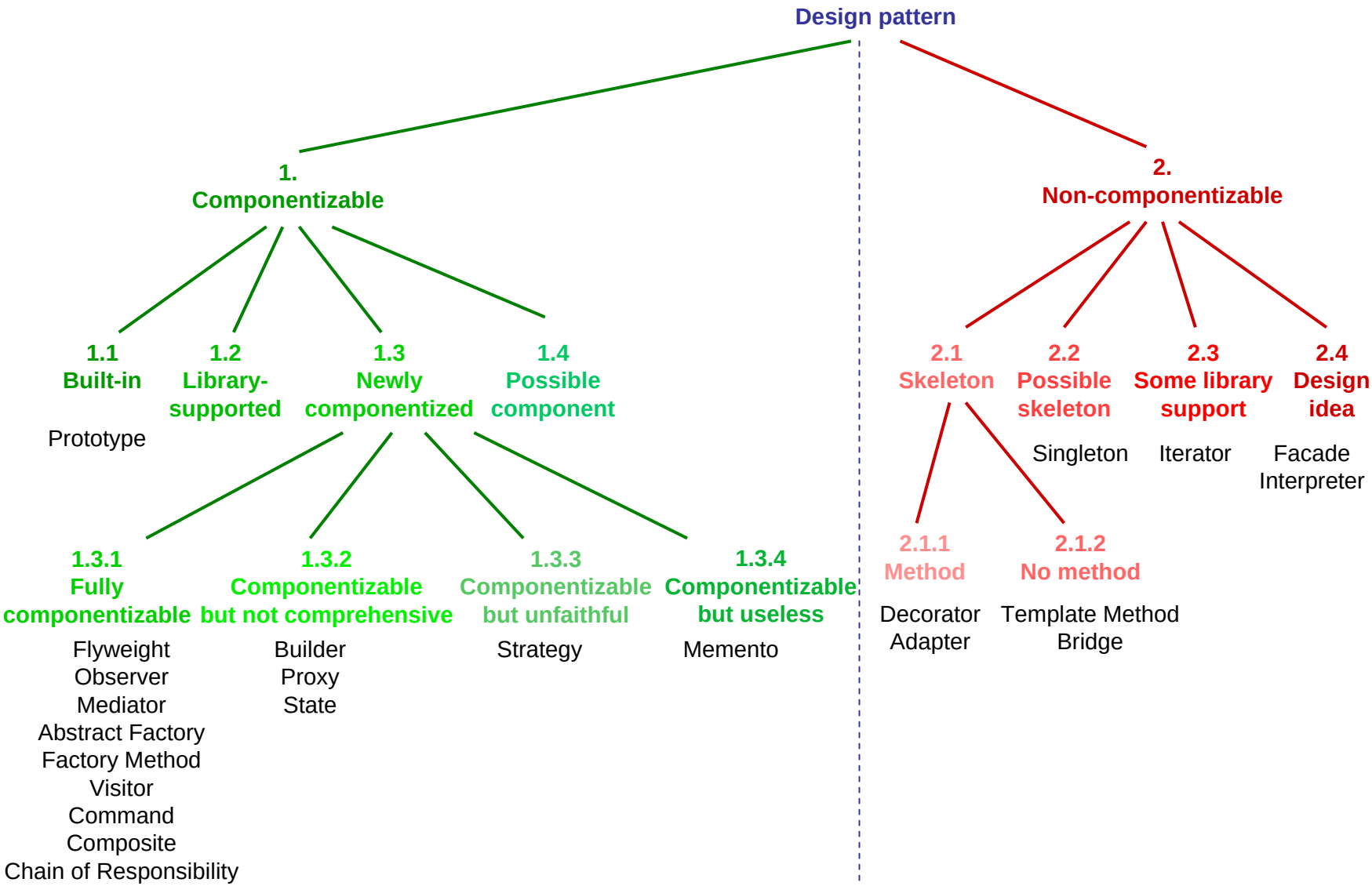
2 categories of patterns:

- Componentizable
- Non-componentizable

Criteria for success

- Completeness
- Usefulness
- Faithfulness
- Type-safety
- Performance
- Extended applicability

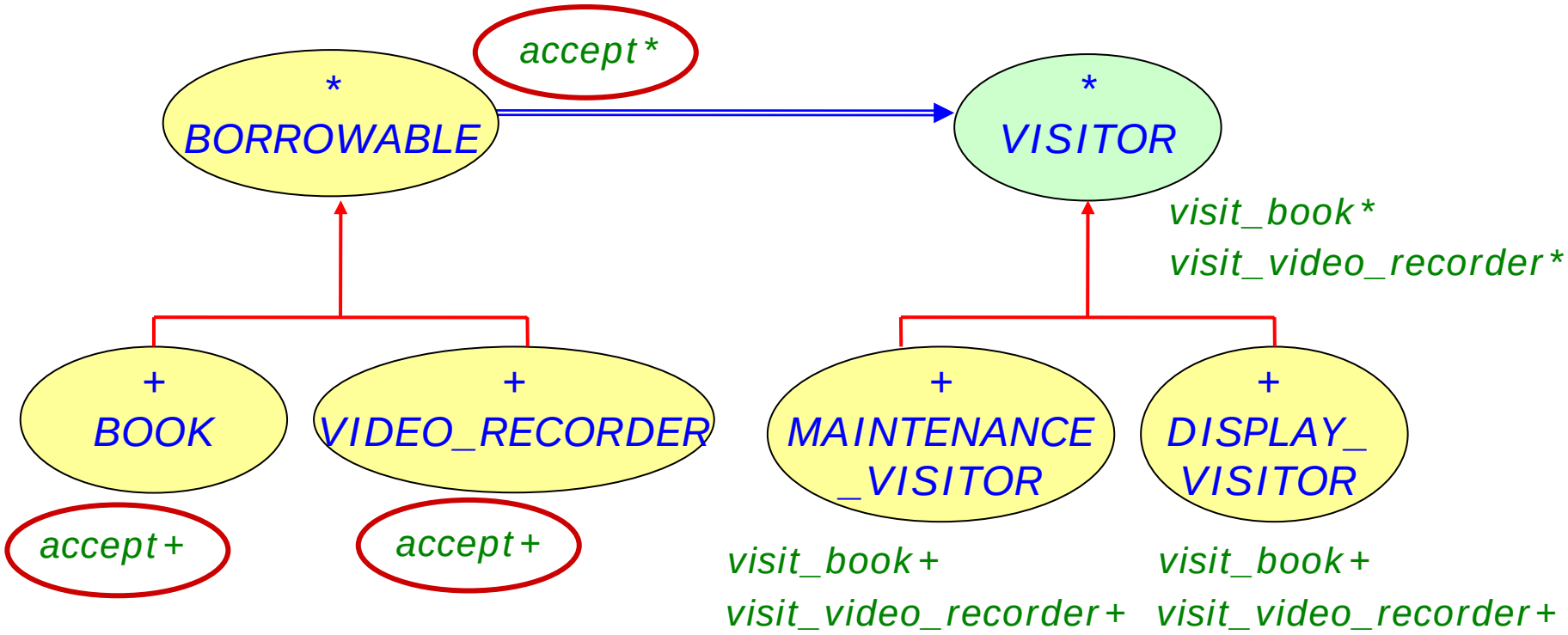
Componentizability classification



Main contributions

- Pattern componentizability classification
- Pattern Library
- Pattern Wizard

The original Visitor pattern



Can we make it easier for the application developer?

The Visitor Library

- One generic class *VISITOR* [*G*]
  e.g. *maintenance_visitor*: *VISITOR* [*BORROWABLE*]

- Actions represented as agents
  actions: *LIST* [*PROCEDURE* [*ANY*, *TUPLE* [*G*]]]

- No need for *accept* features
  - *visit* determines the action applicable to the given element

- For efficiency
  - Topological sort of actions (by conformance)
  - Cache (to avoid useless linear traversals)

Visitor Library interface (1/2)


```
class interface
    VISITOR [G]
create
    make
feature {NONE} -- Initialization
    make
                -- Initialize actions.
feature -- Visitor
    visit (an_element: G)
                -- Select action applicable to an_element.
        require
                an_element_not_void: an_element /= Void
feature -- Access
    actions: LIST [PROCEDURE [ANY, TUPLE [G]]]
                -- Actions to be performed depending on the element
```

Visitor Library interface (2/2)


```
feature -- Element change
    extend (an_action: PROCEDURE [ANY, TUPLE [G]])
                -- Extend actions with an_action.
        require
                an_action_not_void: an_action /= Void
        ensure
                one_more: actions.count = old actions.count + 1
                inserted: actions.last = an_action

    append (some_actions: ARRAY [PROCEDURE [ANY, TUPLE [G]]])
                -- Append actions in some_actions
                -- to the end of the actions list.
        require
                some_actions_not_void: some_actions /= Void
                no_void_action: not some_actions.has (Void)
invariant
    actions_not_void: actions /= Void
    no_void_action: not actions.has (Void)
end
```

How to use the Visitor Library


```
maintenance_visitor: VISITOR [BORROWABLE]
a_book: BOOK
a_video_recorder: VIDEO_RECORDER
...
create maintenance_visitor.make
maintenance_visitor.append ([
                                agent maintain_book,
                                agent maintain_video_recorder
                            ])
maintenance_visitor.visit (a_book)
maintenance_visitor.visit (a_video_recorder)
...
maintain_book (a_book: BOOK) is ...
maintain_video_recorder (a_recorder: VIDEO_RECORDER) is ...
```

Visitor Library: practical assessment

- The case study
  - The target: Gobo Eiffel Lint (*gelint*)
  - Consistency analyzer for Eiffel programs
  - Realistic, full-scale example
- The benchmarks
  - *gelint* applied to *gelint* itself (≈ 700 classes)
  - *gelint* applied to a system from AXA Rosenberg (large-scale financial application, ≈ 9800 classes)

Effect on program size

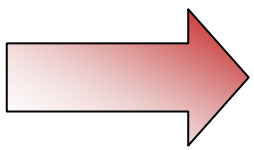
Metric	Original *gelint*	Modified *gelint*	Difference (in value)	Difference (%)
Lines of code	198 263	195 512	-2751	-1.4%
Classes	717	718	+1	+0.1%
Features	67 382	63 421	-3961	-5.9%
Clusters	109	110	+1	+0.9%
Executable size	4104 KB	3660 KB	-444 KB	-10.8%

Effect on performance

Measurements for the AXA Rosenberg system:

Degrees	Original *gelint*	Modified *gelint*	Difference (in value)	Difference (%)
Degree 6	6	6	0	0%
Degree 5	51	51	0	0%
Degree 4	23	30	+7	+30%
Degree 3	25	36	+11	+44%

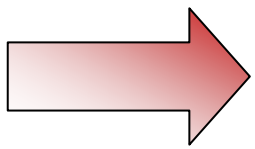
(All times in seconds)



The Visitor Library is usable on a real-world large-scale system.

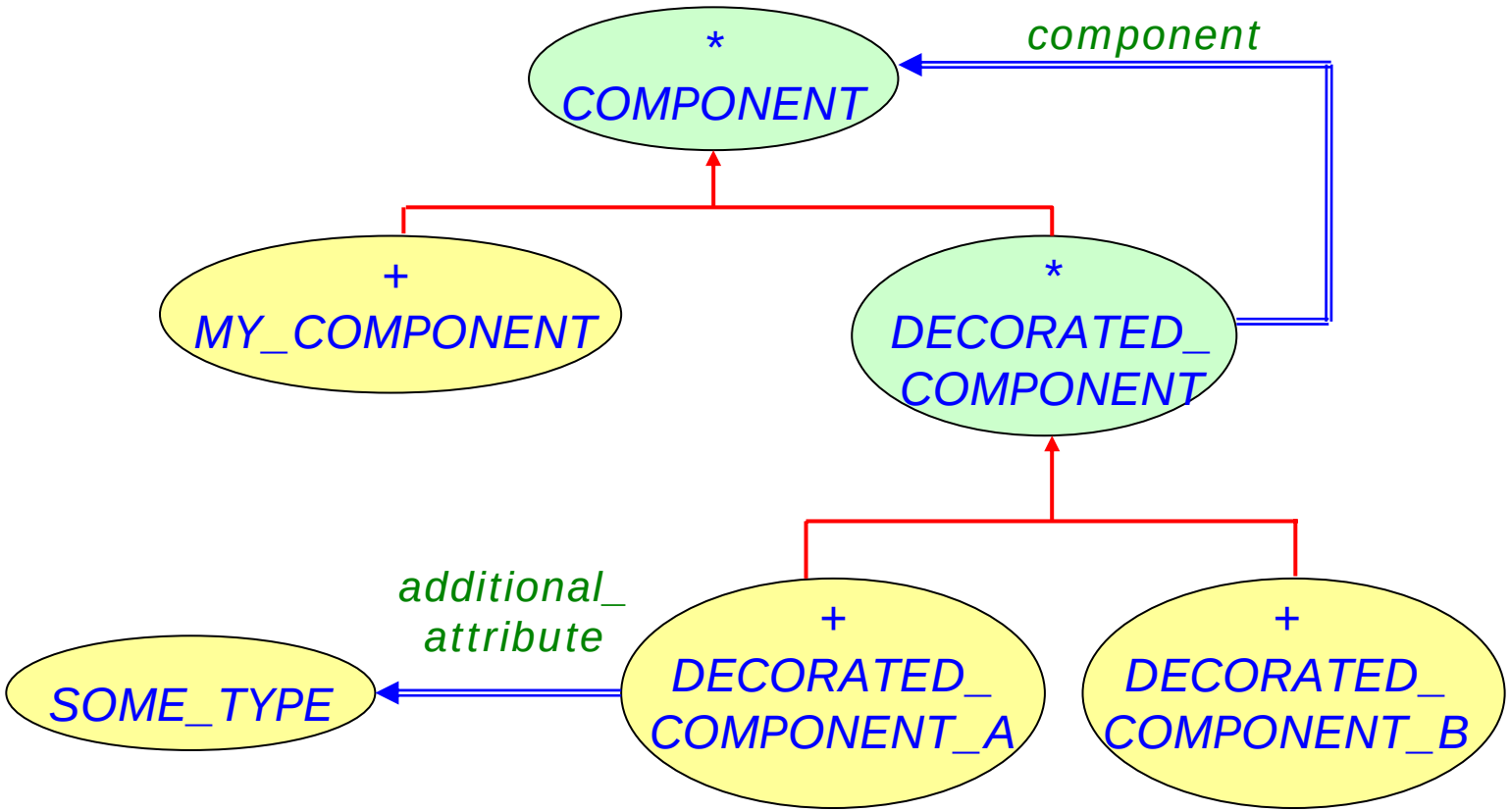
Visitor: Componentization outcome

- Completeness
  - All cases of the pattern
- Usefulness
  - Reusable
  - Easy-to-use (no *accept* feature)
- Faithfulness
  - No double-dispatch mechanism; agents instead
- Type-safety
  - Type-safe (there may be no action associated with a type)
- Performance
  - Less than twice as slow as the *Visitor* pattern
- Extended applicability
  - No more cases



Successful componentization

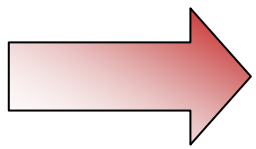
Decorator pattern



Decorator: Componentization outcome (1/2)

- Genericity
  - Idea: have a class *DECORATED_COMPONENT* [*G*]
  - Constraint: a *DECORATED_COMPONENT* must be a *COMPONENT*

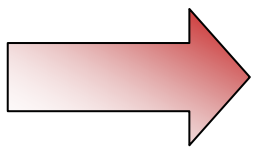
```
class
    DECORATED_COMPONENT [G -> COMPONENT]
inherit
    G
...
end
```



Invalid code

Decorator: Componentization outcome (2/2)

- Automatic type conversion
  - Decoration added to a clone of the original object, not the object itself
- Agents
  - Cannot add an attribute to a given component
- Design by Contract
  - Improves a reusable component; does not make a component reusable
- Aspects
  - Cannot decorate only certain components



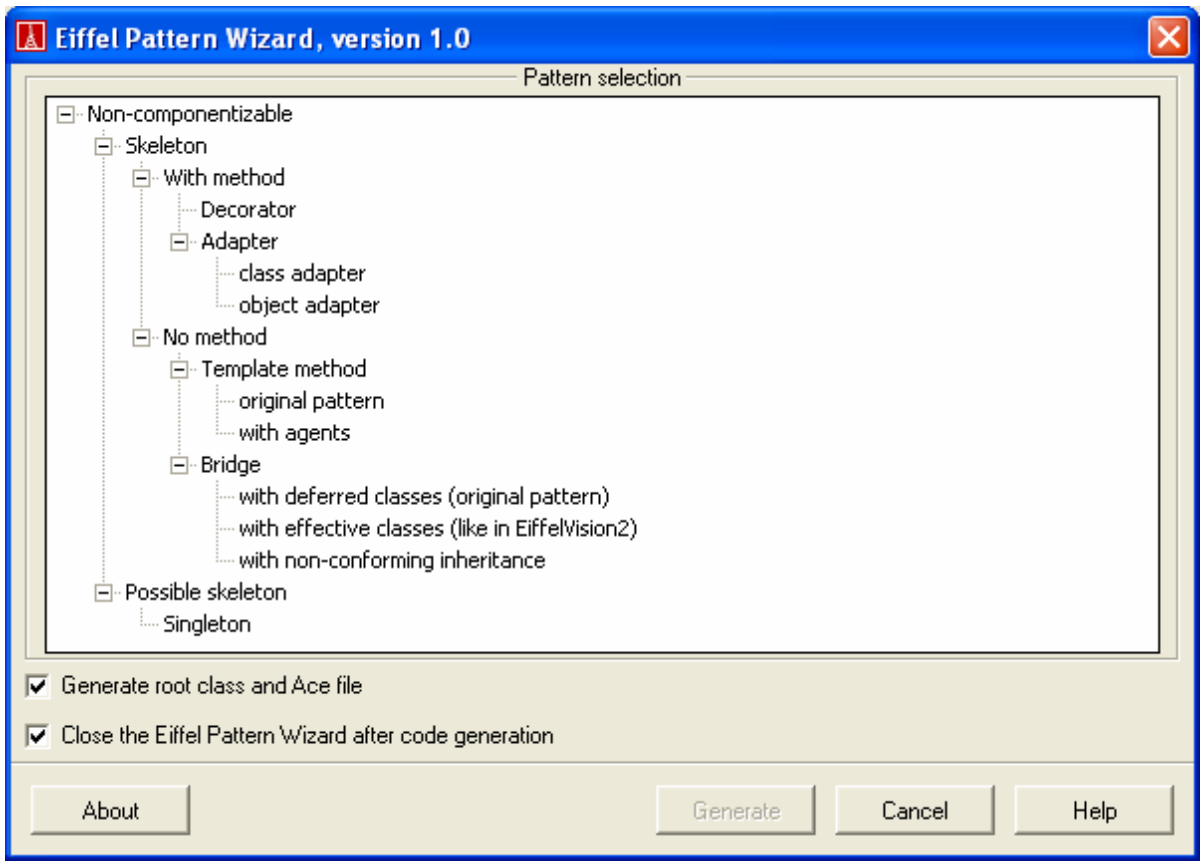
Non-componentizable pattern

Main contributions

- Pattern componentizability classification
- Pattern Library
- Pattern Wizard

Pattern Wizard

- Applicable to non-componentizable patterns
- Automatically generates skeleton classes



→ Generated code

Limitations of the approach

- One pattern, several possible implementations
- Language dependency
  - Genericity
  - Agents
- Componentizability vs. usefulness
  - Usage complexity
  - Performance overhead

Future work

- More patterns, more components
- More steps towards quality components
  - Contract-based testing
  - Contracts for non-Eiffel components

Conclusion

- Originally, an academic work with three goals:
  - New pattern classification
  - Pattern Library
  - Pattern Wizard
- Outcomes directly applicable in the industry:
  - High-quality reusable components
  - Automatic generation tool simplifies the programmer's task
  - Classification tells where to look for help

Design patterns are good, components are better

"*A successful pattern cannot just be a book description*: *it must be a software component*, *or a set of components*".

Bertrand Meyer, *Object-Oriented Software Construction*, 2nd edition, 1997, p 72.

Thank you very much

Correctness and validity (1/2)

- Design patterns are not formally specified:

  "Patterns are not, by definition, fully formalized descriptions. They can't appear as a deliverable."

  J-M. Jézéquel, *Design Patterns and Contracts*, 1999, p 22.

- Componentization:
  - I made my understanding of each pattern explicit through assertions in the
    - componentized version of componentizable patterns
    - Skeleton classes of non-componentizable patterns
  - The Pattern Wizard has been tested according to these contracts

Validation strategy (2/2)

- Validation strategy for the Pattern Library and the skeleton classes generated by the Pattern Wizard
  - 1st step: Test-cases (implementation meets the contracts) http://se.inf.ethz.ch/people/arnout/patterns/
  - 2nd step: Use a real-world application or library and replace its usage of a given pattern by calls to the component or skeleton classes
- Validation of the Pattern Library:
  - Visitor Library in Gobo Eiffel Lint
  - Event Library in ESDL and EiffelVision2
- Validation of the Pattern Wizard:
  - Good candidate for the Bridge pattern: EiffelVision2
  - Limitation of the Wizard: Build classes from scratch, cannot use existing classes

    $\Rightarrow$ Cannot apply 2nd step of the validation strategy
  - Future work:
    - Accept existing classes in the Pattern Wizard
    - Validate the wizard with the Bridge pattern in Vision2

Mechanisms used for componentization

Mechanism	Number of patterns	Percentage
Unconstrained genericity (non-exclusive)	13	72.2%
Constrained genericity (non-exclusive)	7	38.9%
Agents (non-exclusive)	11	61.1%

Performance of agents

- One million calls to a routine that does nothing:
  - Directly: 2s (2µs per call)
  - With agents: 14s (14µs per call)
- One million calls to a routine that executes *do_nothing* twenty times:
  - Directly: 33s (33µs per call)
  - With agents: 46s (46µs per call)
- In real applications, no more than 5% of the time spent in feature calls will be calls to agents

  ⇒ Application with agents ≈ 0,07 times as slow

  ⇒ Acceptable performance overhead in most cases