Trusted Components

Reuse, Contracts and Patterns

Prof. Dr. Bertrand Meyer
Dr. Karine Arnout
Lecture 12: Componentization
Agenda for today

- Componentization
- Componentizability classification
- Role of language mechanisms
- Validation strategy
- Previous work
“Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to this problem in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

Erich Gamma et al., Design Patterns, 1995
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
Challenge of this work

“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

- Challenge:
  - Provide reusable components that programmers can reuse directly without recoding the same functionalities again and again
Target audience

- Program implementers
- Program designers
- Library developers
- Programming language designers
- Both academia and industry
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
Genericity

\textbf{class} \hspace{1cm} \textit{STACK} \left[ G \right]

\textbf{feature}

\textit{put} (x: G) \hspace{0.2cm} \textbf{is} \hspace{0.2cm} ...

\textit{item:} \hspace{0.2cm} G \hspace{0.2cm} \textbf{is} \hspace{0.2cm} ...

\textbf{end}

To use the class: obtain a \textbf{generic derivation}, e.g.

\textit{account_stack:} \hspace{0.2cm} \textit{STACK} \left[ \textit{ACCOUNT} \right]
Using generic derivations

point_stack: STACK [POINT]
p, q, r: POINT
...
point_stack.put (p)
point_stack.put (q)

r := point_stack.item
r. move (3.0, -5.0)
...
Adding two vectors

\[ u + v = w \]

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
</tr>
</thead>
</table>

Chair of Software Engineering
class VECTOR [G]
feature infix "+" (other: VECTOR [G]): VECTOR [G] is
  -- Sum of current vector and other
  require
    other_not_void: other /= Void
    lower_compatible: lower = other.lower
    upper_compatible: upper = other.upper
local
  a, b, c: G
do
  ... See next ...
end
... Other features ...
end
The body of **infix** "+":

```plaintext
create Result.make (lower, upper) from
    i := lower
until
    i > upper
loop
    a := item (i)
    b := other.item (i)
    c := a + b -- Requires a "+" operation on G!
    Result.put (c, i)
    i := i + 1
end
```
Constrained genericity: The solution

- Declare class `VECTOR` as

  ```
  class VECTOR [G -> NUMERIC]
  feature
    ... The rest as before ...
  end
  ```

- Class `NUMERIC` (from the Kernel Library) provides features `infix "+", infix "*", ....`
Componentization mechanisms

- Client-supplier relationship
- Simple inheritance
- Multiple inheritance
- Unconstrained genericity
- Constrained genericity
- Design by Contract
- **Automatic type conversion**
- Agents
- Aspects
class
    MY_CLASS
create
    from_type_1
convert
    from_type_1 ({TYPE_1})
    to_type_2: {TYPE_2}
feature -- Conversion
    from_type_1 (arg: TYPE_1) is
        -- Build from arg.
        do
            -- Something
        end
    to_type_2: TYPE_2 is
        -- Instance of TYPE_2 built from Current object
        do
            -- Something
        end
end
my_attribute: MY_TYPE
attribute_1: TYPE_1
attribute_2: TYPE_2
...
my_attribute + attribute_1
   -- Equivalent to:
   -- my_attribute + create {MY_TYPE}.from_type_1 (attribute_1)

attribute_2 + my_attribute
   -- Equivalent to:
   -- attribute_2 + my_attribute.to_type_2
Componentization mechanisms

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

- Object encapsulating a routine ready to be called
  - May be viewed as:
    - Delayed call, or typed function pointer

- To create an agent:
  
  ```
  my_routine := agent my_feature
  ```

- To call an agent:
  
  ```
  my_routine.call ([args])
  ```
Componentization mechanisms

- Client-supplier relationship
- Simple inheritance
- Multiple inheritance
- Unconstrained genericity
- Constrained genericity
- Design by Contract
- Automatic type conversion
- Agents
- Aspects
aspect DecoratedComponent {
    /*Special construct (called pointcut) to specify when and where the aspect
     * should be applied.
     * A pointcut typically lists the features to which the aspect applies.*/

    before:
    pointcutName...{
        /*You may view pointcutName as a feature name as a first approximation.*/
        doSomething;
    }

    after:
    pointcutName...{
        doSomething;
    }

    around:
    pointcutName...{
        if (someCondition)
            proceed ();
        else
            System.out.println ("Error");
    }
}
Criteria for success

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

Property of a system for which any call of the form \( x.f(a) \) that is valid at compilation time, there exists exactly one version of \( f \) applicable to the dynamic type of \( x \) at run time and this version of \( f \) has the right number of arguments and the appropriate types.
Why are the components trusted?

- Extensive use of contracts

- Criteria assessing the quality of the components:
  - Completeness
  - Usefulness
  - Faithfulness
  - Type-safety
  - Performance
  - Extended applicability
## Componentization statistics

<table>
<thead>
<tr>
<th>Category</th>
<th>Number of patterns</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Componentizable patterns</td>
<td>15</td>
<td>65%</td>
</tr>
<tr>
<td>Non-componentizable patterns</td>
<td>8</td>
<td>35%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category</th>
<th>Number of patterns</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Componentizable patterns</td>
<td>15</td>
<td>65%</td>
</tr>
<tr>
<td>Non-componentizable patterns (possible skeleton classes or some library support)</td>
<td>6</td>
<td>26%</td>
</tr>
<tr>
<td>Remaining patterns (no skeleton classes and no library support)</td>
<td>2</td>
<td>9%</td>
</tr>
</tbody>
</table>
Agenda for today

- Componentization
- Componentizability classification
- Role of language mechanisms
- Validation strategy
- Previous work
Componentizability classification

1. Componentizable
   - 1.1 Built-in
   - 1.2 Library-supported
   - 1.3 Newly componentized
   - 1.4 Possible component
     - 1.3.1 Fully componentizable
     - 1.3.2 Componentizable but not comprehensive
     - 1.3.3 Componentizable but unfaithful
     - 1.3.4 Componentizable but useless

2. Non-componentizable
   - 2.1 Skeleton
   - 2.2 Possible skeleton
   - 2.3 Some library support
   - 2.4 Design idea
     - 2.1.1 Method
     - 2.1.2 No method
Componentizability classification

1. Componentizable
   1.1 Built-in
      Prototype
   1.2 Library-supported
   1.3 Newly componentized
      1.3.1 Fully componentizable
         Flyweight
         Observer
         Mediator
         Abstract Factory
         Factory Method
         Visitor
         Command
         Composite
         Chain of Responsibility
      1.3.2 Componentizable but not comprehensive
         Builder
         Proxy
         State
      1.3.3 Componentizable but unfaithful
         Strategy
      1.3.4 Componentizable but useless
         Memento
   1.4 Possible component

2. Non-componentizable
   2.1 Skeleton
      2.1.1 Method
         Decorator
         Adapter
      2.1.2 No method
         Template Method
      2.1.3 Componentizable but not comprehensive
         Singleton
         Iterator
         Interpreter
   2.2 Possible skeleton
      2.2.1 Componentizable but unfaithful
         Proxy
   2.3 Some library support
      2.3.1 Componentizable but useless
         Memento
   2.4 Design idea
      2.4.1 Componentizable
         Bridge
Agenda for today

- Componentization
- Componentizability classification
- Role of language mechanisms
- Validation strategy
- Previous work
Fully componentizable thanks to ...

- Flyweight
- Observer
- Mediator
- Abstract Factory
- Factory Method
- Visitor
- Command
- Composite
- Chain of Responsibility

Unconstrained genericity

Constrained genericity

Agents

+ Design by Contract
+ Client-supplier relationship
+ Inheritance
### Mechanisms used for componentization

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Number of patterns</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconstrained genericity (non-exclusive)</td>
<td>13</td>
<td>72.2%</td>
</tr>
<tr>
<td>Constrained genericity (non-exclusive)</td>
<td>7</td>
<td>38.9%</td>
</tr>
<tr>
<td>Agents (non-exclusive)</td>
<td>11</td>
<td>61.1%</td>
</tr>
</tbody>
</table>

Approach portable to any language with these mechanisms
Agenda for today

- Componentization
- Componentizability classification
- Role of language mechanisms
- Validation strategy
- Previous work
Correctness and validity

- Design patterns are not formally specified:

  “Patterns are not, by definition, fully formalized descriptions. They can’t appear as a deliverable.”


- Componentization:
  
  - Understanding of each pattern explicit through assertions
Semantics of design patterns

- Patterns have a description (intent)
  - e.g. Chain of Responsibility: “Avoid coupling the sender of a request to its received by giving more than one object a change to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.” [GoF, 223]
- No formal specification

- Components add semantics through contracts
  - e.g. class **HANDLER** \([G]\) has `can_handle`, `handled` used in postcondition of `handle`:

```plaintext
ensure
  can_handle (a_request) implies handled
  (not can_handle (a_request) and then next /= Void)
  implies handled = next.handled
  (not can_handle (a_request) and then next = Void)
  implies not handled
```
Validation strategy

- **1st step:** Test-cases
  - Check that implementation meets the contracts
  - [http://se.inf.ethz.ch/people/arnout/patterns/](http://se.inf.ethz.ch/people/arnout/patterns/)

- **2nd step:** Use a real-world application or library
  - Example: Visitor Library in Gobo Eiffel Lint
Agenda for today

- Componentization
- Componentizability classification
- Role of language mechanisms
- Validation strategy
- Previous work
Reusability classification

(Reusability: The abstract aspect can be reused. Programmers need to write concrete aspects.)

Similar to the componentizability classification, except for:

- Reusable / Non-componentizable, skeleton:
  - Singleton
- Reusable / Non-componentizable, some library support:
  - Iterator
- Not reusable / Componentizable but not comprehensive:
  - Proxy, Builder, State
- Not reusable / Fully componentizable:
  - Abstract Factory, Factory Method
Aspect implementation - Pros and cons

- **Strengths:**
  - Reduction of the number of pattern’s participants (one aspect instead of several classes)
  - Traceability of the patterns in the code
  - Localization of the pattern code
  - Reusability of the pattern code

- **Weaknesses:**
  - Language dependency
  - Problem shift (many small aspects vs. several classes)
  - Difficulty to understand (aspects + classes) and to maintain
Componentization vs. aspects

- [Hannemann 2002]:
  Reusability classification of the GoF patterns

- Why is componentization still useful?
  - “Reusability: The abstract aspect can be reused. Programmers need to write concrete aspects.”
    ⇒ Still requires some repetitive work from the programmers.
  - Language dependency (not all OO language has an aspect version)
  - Problem shift (many small aspects vs. several classes)
  - Difficulty to understand (aspects + classes) and to maintain
Language support of design patterns

- Established fact: Design patterns “have proved so useful that some have called for their promotion to programming language features” [Chambers 2000].

- Which pattern deserves a direct language support?
  - Chambers: “Clearly, languages lacking the appropriate mechanisms benefit from tool support, but this should be viewed as an undesirable intermediate state in language development, to be replaced in the future by true language support without tools requirements.”

  - Avoid “featurism” [Meyer 2002]; the construct should solve a general problem (e.g. Singleton vs. frozen classes).

  - Vlissides: “While several of the more fundamental design patterns may be transliterated easily into programming language constructs, many others cannot – or at least should not.”
“A successful pattern cannot just be a book description: it must be a software component, or a set of components”.

Complementary material

- From Patterns to Components:
  - Chapter 6: Pattern componentizability classification

  - Chapter 4: Previous work
    - 4.2 Aspect implementation
    - 4.3 Language support
End of lecture 12