Trusted Components
Reuse, Contracts and Patterns

Prof. Dr. Bertrand Meyer
Dr. Karine Arnout
Lecture 25: Limitations and Future work
Agenda for today

- Limitations of the approach
- Future work
- Presentation of the Chair of Software Engineering
One pattern, several implementations

- “A pattern may hide another pattern”
  - Could be taken into account in certain reusable components:
    - Composite Library (transparency vs. safety versions, with or without knowledge of the parent container)
    - Command Library (history-executable vs. auto-executable)
  - Other components are non-comprehensive:
    - State Library (state-driven transitions)
    - Builder Library (two- and three-part products)
    - Proxy Library (virtual proxy)
• Pattern Library uses some mechanisms that are only supported in Eiffel for the moment:

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

Approach portable to any language with these mechanisms
using System;
using System.Collections;

class Composite<T> extends Component<T> {
    // Initialization
    // Initialize component Parts.
    public Composite() {
        Parts = new ArrayList();
    }
    /** Set Parts to someComponents.
     * require
     * someComponents != null;
     * !someComponents.Contains(null);
     * ensure
     * Parts == someComponents;
     */
    public Composite(ArrayList<Component<T>> someComponents) {
        Parts = someComponents;
    }
}
// Element change
/** Add aPart to component Parts. */
* require
* aPart != null;  
* !Has(aPart);
* ensure
* Parts.Count == old Parts.Count + 1
* Has(aPart);
*/
public void Add(Component<T> aPart){
    int index = Parts.Add(aPart);
}

...
import java.lang.*;
import java.lang.reflect.*;

class Command{
    // Initialization
    /** Set action to anAction, undoAction to anUndoAction, isOnceCommand to aValue.
     * require
     * anAction != null;
     * anUndoAction != null;
     * ensure
     * action == anAction;
     * undoAction == anUndoAction;
     * isOnceCommand == aValue;
     */
    public Command(Method anAction, Method anUndoAction, boolean aValue){
        action = anAction;
        undoAction = anUndoAction;
        isOnceCommand = aValue;
    }

    // Access
    // Action to be executed
    public Method action;

    // Action to be executed to undo the effects of calling action
    public Method undoAction;
}
// Command pattern
// Call action on anObject with args.
public void execute(Object anObject, Object [] args) throws IllegalAccessException, IllegalArgumentException, InvocationTargetException {
    action.invoke (anObject, args);
}
...
/**
 * invariant
 * action != null;
 */
}
using System;
using System.Collections;
using System.Reflection;

class Visitor<T> {
    // Initialization
    // Initialize Actions.
    public Visitor() {
        Actions = new ArrayList();
        ...
    }

    // Access
    // Actions to be performed depending on the element
    public ArrayList<MethodBase> Actions;
}
// Visitor
/** Visit anElement.
 * (Select appropriate action depending on anElement.)
 * require
 *   anElement!= null;
 */

public void Visit (T anElement){
    MethodBase anAction = null;
    IEnumerator<MethodBase> cursor = Actions.GetEnumerator();
    while (cursor.MoveNext()){
        anAction = cursor.Current;
        if (IsValidOperands (anAction, anElement)){
            anAction.Invoke (anAction.ReflectedType, 
                            new Object [1] {anElement});
            break;
        }
    }
}
// Element change
/*** Extend Actions with anAction.
 * require
 *  anAction != null;
 * ensure
 *  Actions.Contains (anAction);
 */
public void Extend (MethodBase anAction){...}

/*** Append actions in someActions to the end of the Actions list.
 * require
 *  someActions != null;
 *  !(someActions.Contains (null));
 */
public void Append (ArrayList<MethodBase> someActions){...}

/***/
* invariant
*  Actions != null;
*  !Actions.Contains (null);
*/
}
Componentizability vs. usefulness

- Some reusable components may be less user-friendly than a customized pattern implementation

- Limitations of the Pattern Library:
  - Usage complexity (e.g. Memento Library)
  - Performance overhead (e.g. Visitor Library)
Agenda for today

- Limitations of the approach
- Future work
- Presentation of the Chair of Software Engineering
More patterns, more components

- There exists more patterns than the 23 described in *Design Patterns*
  - e.g. Model-View-Controller, Publisher-Subscriber in *Pattern-Oriented Software Architecture* [Bushmann 1996]

- Most of these other patterns are domain-specific (e.g. concurrency, database,...)

- Componentized versions of these patterns would better belong to a domain-specific library
  - SCOOP could help componentizing the patterns dealing with concurrency
Which of the following is true?

1. Eiffel libraries have contracts just because Eiffel supports contracts.
2. Contracts are inherent in any good library design.

Results of the analysis:
- Routine preconditions are expressed through exception cases
- Routine postconditions and class invariants are scattered across the documentation
Architecture of the precondition extraction tool

.NET assembly

ildasm

CIL code

Scanner → Parser

AST

Basic block analysis → directed graph → Basic block classification

Code path extraction → Subgraph of basic blocks → Symbolic execution → Precondition extraction

Preconditions
## First results (1/2)

<table>
<thead>
<tr>
<th>Precondition clauses</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>this /= Void</code></td>
<td>7</td>
</tr>
<tr>
<td><code>c /= Void</code></td>
<td>4</td>
</tr>
<tr>
<td><code>type /= Void</code></td>
<td>2</td>
</tr>
<tr>
<td><code>array /= Void</code></td>
<td>1</td>
</tr>
<tr>
<td><code>not (index &lt; 0)</code></td>
<td>22</td>
</tr>
<tr>
<td><code>count &gt;= 0</code></td>
<td>14</td>
</tr>
<tr>
<td><code>(this._size - index) &gt;= count</code></td>
<td>7</td>
</tr>
<tr>
<td><code>index &lt; this._size</code></td>
<td>3</td>
</tr>
<tr>
<td><code>index &lt;= this._size</code></td>
<td>2</td>
</tr>
<tr>
<td><code>value &gt;= 0</code></td>
<td>1</td>
</tr>
<tr>
<td><code>arrayIndex &gt;= 0</code></td>
<td>1</td>
</tr>
<tr>
<td><code>(startIndex + count) &lt;= this._size</code></td>
<td>1</td>
</tr>
</tbody>
</table>
First results (2/2)

<table>
<thead>
<tr>
<th>Precondition clauses</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>not (startIndex &lt; 0)</td>
<td>1</td>
</tr>
<tr>
<td>startIndex &lt; this._size</td>
<td>1</td>
</tr>
<tr>
<td>startIndex &lt;= this._size</td>
<td>1</td>
</tr>
<tr>
<td>this._remaining &gt;= 0</td>
<td>1</td>
</tr>
<tr>
<td>False</td>
<td>36</td>
</tr>
<tr>
<td>this.version = this.list._version</td>
<td>4</td>
</tr>
<tr>
<td>this._baseVersion = this._baseList._version</td>
<td>1</td>
</tr>
<tr>
<td>this._firstCall = 0</td>
<td>1</td>
</tr>
<tr>
<td>this._size /= 0 implies count &lt;= (startIndex + 1)</td>
<td>1</td>
</tr>
<tr>
<td>this._size /= 0 implies count &gt;= 0</td>
<td>1</td>
</tr>
<tr>
<td>this._size /= 0 implies not (startIndex &lt; 0)</td>
<td>1</td>
</tr>
<tr>
<td>this._size /= 0 implies not (startIndex &gt;= this._size)</td>
<td>1</td>
</tr>
<tr>
<td>TOTAL</td>
<td>115</td>
</tr>
</tbody>
</table>
Contract Wizard

- .NET assembly
  - without contracts
  - with contracts

- .NET reflection mechanism

- XML contract files
  - with contracts

- XML parser

- AST

- GUI

- AST with contracts
  - user contracts

- Code generator

- Proxy Eiffel classes
  - ACE file
  - XML contract files

- Eiffel compiler

- .NET assembly with contracts

Errors in the contracts

Chair of Software Engineering
Performance

- Testbed: mscorlib
  < 1 minute to add 5000 contracts (at once)

- Performance gain resulting from incrementality:

<table>
<thead>
<tr>
<th>Metrics</th>
<th>.NET reflection</th>
<th>XML parsing</th>
<th>Difference (value)</th>
<th>Difference (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time</td>
<td>270s</td>
<td>112s</td>
<td>-158s</td>
<td>-59%</td>
</tr>
<tr>
<td>Memory consumption</td>
<td>78MB</td>
<td>51MB</td>
<td>-27MB</td>
<td>-35%</td>
</tr>
</tbody>
</table>
Four-step process:

1. Parse the contract-equipped Eiffel library provided as input to get the system information (list of classes, features, ...) to be displayed to the user

2. Gather user information (list of features to be tested, with what arguments, ...) to define the test scenario

3. Generate the test executable

4. Run the test executable and store the test results into a database
Architecture of the TestStudio (1/2)

1. Contract-equipped Eiffel library
   - Eiffel analyzer
   - System information
   - Information handler
   - Test scenario
   - Test scenario handler

2. Code generator
   - Test executable
   - GUI + Context handler
   - Context generator
   - Feature call simulator
   - Test result
   - DB

3. DB

4. DB

5. DB query handler

6. Test result

7. Test result handler

8. GUI Test result displayer
   - XML files
   - Gobo Eiffel Test format

9. DB

10. Input / Output
    - Tool
    - Storage
1. Gather system information.
2. Display system information to the user and gather user information.
3. Build the test scenario from the criteria selected by the user.
4. Store the test scenario into a database (for regression testing).
5. Generate a test executable corresponding to the test scenario.
6. Run the executable: it creates a pool of objects (the “Context”) possibly helped by the user.
7. Store the order of class instantiations (for regression testing).
8. The executable performs feature calls on the pool of instantiated objects.
9. Store the test results into a database.
10. Output the results to the users.
11. Display the results graphically with diagrams.
12. Generate XML files corresponding to the test results.
13. Generate files using the Gobo Eiffel Test format.
14. Query the database and retrieve test results to be passed to the test result handler.
From Patterns to Components:
- Chapter 22: Limitations of the approach
- Chapter 23: More steps toward quality components

Further reading:
Further reading:


Agenda for today

- Limitations of the approach
- Future work
- Presentation of the Chair of Software Engineering
People

Bertrand Meyer’s group:
- Dr. Karine Arnout
- Volkan Arslan
- Dr. Arnaud Bailly
- Stephanie Balzer
- Till Bay
- Ruth Bürkli (editorial/administrative assistant, JOT)
- Ilinca Ciupa
- Piotr Nienaltowski
- Michela Pedroni
- Joseph Ruskiewicz
- Bernd Schoeller
- Sébastien Vaucouleur

Peter Müller’s group
- Adam Darvas
- Werner Dietl

Jean-Raymond Abrial’s group
- Dr. Stefan Hallerstede
- Farhad Mehta
- Laurent Voisin

Visitors
- Vijay d’Silva

Project, Master’s and Diplom students
Scope

- Help move software technology to the next level through:
  - Trusted Components
  - Advanced O-O techniques
  - Teaching (including introductory)

- Technologies of special interest:
  - Eiffel
  - .NET
  - B
Publications and events

- We publish JOT, the Journal of Object Technology
  
  *The leading international journal on objects and components*
  
  [www.jot.fm](http://www.jot.fm)

- FATS seminar (Formal Approaches To Software)

- Workshops and conferences

- LASER summer school on applied software engineering (started September 2004)
Grants

- Hasler foundation, 3 years: Object-oriented concurrency (SCOOP)

- Swiss National Science Foundation, 3 years: SCOOP

- ETH TH-Gesuch, 3 years: Class proofs
Components equipped with a clear definition of their properties, and a guarantee that they satisfy these properties.
Two complementary approaches:

- **Low road**: examine and certify today’s components (COM, .NET, EJB, O-O libraries)...
  
  Towards a Component Certification Center

- **High road**: proofs of component correctness
Towards a Component Quality Model (Till Bay)

A: Acceptance

B: Behavior

C: Constraints

D: Design

E: Extension
Proofs of Class Correctness
(Bernd Schoeller, Joseph Ruskiewicz)

- Use model for contracted Eiffel classes
- Use B to prove properties of model and implementation
- Contracts in non-Eiffel libraries
  - The Closet Contract Conjecture
  - We performed a systematic analysis of the .NET Collection library and found many implicit contracts
  - Contract elicitation can be partially automated

- Contracts and Testing
  - Contracts provide the best known basis for automatic test generation
  - Extensive tool under development
  - Supported by database for regression testing
Pattern elimination conjecture: patterns are good, components are better

Karine Arnout performed systematic analysis of GoF patterns to see which ones can be turned into components

Language mechanisms play critical role, e.g. genericity, multiple inheritance
O-O development: Concurrency
(Dr. Arnaud Bailly, Volkan Arslan, Piotr Nienaltowski, Sébastien Vaucouleur)

- **SCOOP model**
  - Very simple and high level language extension (one keyword: `separate`) supporting many different forms of concurrency and distribution

- **Processor:**
  An autonomous thread of control capable of supporting the sequential execution of instructions on one or more objects (processors can be mapped e.g. to threads, processes, coroutines, ...)

- **Research directions**
  - Access control in concurrent object-oriented programming
  - Applying concurrent object-oriented programming to real-time applications
  - Implementation of SCOOP for .NET and multithreading
General framework for persistent programming

- Seamless approach: No explicit programming of
  - store/retrieve operations or
  - database interactions

- Schema evolution:
  - Class differs from stored objects
    - Object retrieval mismatch
  - Retrieve objects from the storage when their class descriptions have changed
O-O development: Eiffel

- ECMA standard in progress
- 12 meetings so far
- Hope to have a standard by mid-2005
- Conservative and innovative
Advanced classes
- Trusted Components
- Advanced Topics in Object Technology

Introductory teaching
- Outside-in
- Object-oriented
- Reuse-based
- TRAFFIC multimedia library
Student projects

- Wide range of projects

- One of the permanent team members becomes supervisor

- Students become members of our group; attendance at Wednesday meetings expected

- See project page for list of topics, and completed projects:
  http://se.inf.ethz.ch/projects/index.html
Thank you!