14) Functional, Action-, Data-Flow, ECA-Based Design Illustrated by Example

- Prof. Dr. U. Aßmann
- Technische Universität Dresden
- Institut für Software- und Multimediatechnik
- http://st.inf.tu-dresden.de
- Version 10-03. 20.12.2010

1. The KWIC Case Study
Obligatory Readings

- Ghezzi Chapter 3, Chapter 4, esp. 4.2
- Pfleeger Chapter 5, esp. 5.7
  - [http://www.stormingmedia.us/65/6538/A653882.html](http://www.stormingmedia.us/65/6538/A653882.html)
Pfleeger 5.7, Shaw/Garlan 1996

14 THE KWIC EXAMPLE PROBLEM
"Keyword in Context" problem (KWIC) is one of the 10 model problems of architecture systems [Shaw-ModelProblems, www.cmu.edu] [Shaw/Garlan96, Pfleeger 5.7]

Originally proposed by Parnas to illustrate advantages of different designs [Parnas72]

For a text, a KWIC algorithm produces a permuted index

- Every sentence is replicated and permuted in its words, i.e., the words are shifted from left to right.
- Every first word of a permutation is entered into an alphabetical index, the permuted index.
The KWIC index system accepts an ordered set of lines
  • Each line is an ordered set of words,
  • and each word is an ordered set of characters.
Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line.
The output of the KWIC index system is a listing of all circular shifts of all lines in alphabetical order

[Parnas]
<table>
<thead>
<tr>
<th>every sentence is replicated</th>
<th>and</th>
<th>permuted</th>
</tr>
</thead>
<tbody>
<tr>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>every sentence is replicated and permuted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>every sentence is replicated and permuted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>every sentence is replicated and permuted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>every sentence is replicated and permuted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>every sentence is replicated and permuted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>every sentence is replicated and permuted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>every sentence is replicated and permuted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>every sentence is replicated and permuted</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Input: reads the sentences
- Input formats
- Are all lines stored in memory? (bad for large texts)
- Packed or unpacked character storage
- Store the index?
- Distributed or non-distributed memory?

Output: outputs the KWIC index
- Highlighting of keywords?
- Text or PS, or PDF-output

Circular Shifter: permutes the generated sentences

Sorter: sorts the shifted sentences so that they form a keyword-in-context index
- Sort all the index or look entries up?
- Complete or partial sorting

Caps: replicates the sentences as necessary
- Lazy or eager replication
KWIC are very important for technical documents

Examples

• "Beitrag zur Populationsgenetik der sauren Erythrocytenphosphatase-acP-EC3.1.3.2 unter besonderer Berücksichtigung des reinerbigen Typus C" (1980)
• "Lepton-Hadron-Korrelationen in (2+1)-Jet-Produktion in tief-inelastischer Elektron-Proton-Streuung zur O(alpha2 s)" (1992)
• "Die molekulare Wirkung von 2,4,5- und 2,4,6-Trichlorphenol auf Eukaryontenzellen" (1990)
• "Aufklärung, Vernunft, Religion – Kant und Feuerbach" (2005)
1. Variability: Changes of implementations of components
   1. When does the circular shifter work?
   2. When does the sorting work?

2. Variability: Changes of data representations
   1. Representation of sentences, words, lines
   2. Use of indices?
   3. How to avoid redundancy?

3. Extension with new functionality
   1. E.g., insertion of fill words

4. Speed

5. Reusability of components
The KWIC Problem in Action-Based Design: Call-Based Style with Shared Repository

CALL-REP

Control

Input

Circular Shifter

Sorter

Output

Repository

Characters

Index

Alphabetized Index

input

Call

output

Prof. U. Aßmann

Comparison of Architectural Styles
The KWIC Problem in Call-Based Repository Style

- **Bad:**
  - State of the repository visible to several callers
  - A change in the data affects all modules
  - High costs if algorithm have to be changed
  - The modules are not reusable
  - **Bad encapsulation of module secrets!**

- **Good:**
  - Fast, due to shared memory access
  - Easy to code
  - Shared memory is a fast concept, but provides few information hiding.
The KWIC Problem in Function-Based Design:
(Abstract Data Types with Private Decentralized Memory)

CALL-DEC

Input

Control

Output

input

CALL-DEC

Char Representation

Characters
- setchar
- char()
- word()

Sentence Representation

CircularShift
- setup()
- setchar()
- char()
- word()

Index Representation

Alphabetized Index
- alph()
- ith()
The KWIC Problem in Decentralized Memory

Good:

- Data and algorithm are easier to change (e.g., packing and storing the whole character) since
  - Data representation is hidden in functions
  - Algorithm partly hidden
  - The control flow works "on demand" from the Control through the Output backwards to the Input
- More module secrets: char, sentence, and index representation
- Layering

Bad:

- Adding new functions may be hard, since control flow intertwines the modules tightly
The KWIC Problem in Event-Condition-Action Design
(Implicit Invocation Style)

Prof. U. Aßmann

Comparison of Architectural Styles

Call

Event (implicit invocation)

ECA

Input

- insert()
- delete()
- ith()

Sentence Representation

Output

- insert()
- delete()
- ith()

Sentence Representation

Control

Circular Shift

Sorter

---

Lines

input

output

15
The KWIC Problem in Implicit Invocation Style (ECA Style)

- **Good:**
  - Data and algorithm are easy to change
  - they are hidden in functions
  - The control flow works forward by "implicit invocation", i.e., sending an event, from the Input/Lines through the Shifter and the Sorter
    - The listeners test conditions and execute an action
  - Layering
  - Event-based style simplifies the addition of new functions, since they may additionally listen to the events; event sources need not be changed (even more module secrets)

- **Bad:**
  - Flow of control is hard to predict
  - Hard to analyze statically; unusable for safety-critical systems
The KWIC Problem in Action-Based Design (Pipe-and-Filter Data-flow Style, SA)

Comparison of Architectural Styles

Pipe

DATAFLOW

Input

Data Representation

Circular Shift

Data Representation

Sorter

Data Representation

Output

Data Representation

Input

Output
The KWIC Problem in Pipe-And-Filter Data-Flow Style

- **Good:**
  - Data and algorithm are easy to change (by filter exchange)
  - Adding new functions is easy (new filters)
  - Flow of control is easy to say
  - **Data representation is completely hidden in the filters**
  - Highly reusable filter modules

- **Bad:**
  - No evolution to interactive system
Comparison of KWIC Designs

<table>
<thead>
<tr>
<th>Easy to use</th>
<th>CALL-REP</th>
<th>CALL-DEC</th>
<th>ECA</th>
<th>DATA-FLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Data representation</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Function</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Good performance</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Easy reuse</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

[Shaw/Garlan 1996] Comparison can be improved with weighted priorities.
What Have We Learned?

- When designing with functions, use function trees and subfunction decomposition
- When grouping to modules, fix module secrets
- The more module secrets, the better the exchange and the reuseability
  - Change-oriented design means to encapsulate module secrets
- Functional and modular design are still very important in areas with hard requirements (safety, speed, low memory)