26. Data-Oriented Design Methods

1) Jackson Structured Programming (JSP) and Jackson Structured Diagrams (JSD)
2) Grammar-Driven Programming
3) Extensibility of JSP and Grammar-Based Applications
Obligatory Reading

- Ghezzi Ch. 3.3, 4.1-4, 5.5
- Pfleeger Ch. 4.1-4.4, 5
  - http://www.jacksonworkbench.co.uk/stevefergspages/jackson_methods/index.html
- Non-obligatory literature:
23.1 Jackson Structured Programming as Data-Oriented Development with Regular Data

- „Grammarware“ is the technical space of all grammars describing data structures.
Data-Oriented Development (for „Grammarware“)

- **Data-oriented development** focuses **first** on the development of a data structure
  - Tree specification with string grammars or tree grammars
  - Attributed tree specification with attributed grammars
  - Link tree specifications (e.g., with XML schema)
  - Graph specifications with graph grammars and graph transformation systems (e.g., reducible graphs)
  - Path specifications with automata

- **Divide**: find subdata structures

- **Conquer**: compose subdata structures to larger data units

- **Second step**: Derive a visiting algorithm that works on all elements of the data structure in a pre-defined, specified way (similar to design pattern Visitor)
  - Surprising: Grammars cannot only be used to parse strings, but to specify the **walk order** of a visiting algorithm!

**Design Question:**
How is the data structured?
so that the algorithms can homomorphically be derived from its structure
Example for Data-Oriented Design: Jackson Structured Programming JSP

- Data-oriented developing with hierarchical tree diagrams, a variant of a function/action tree
- The tree defines a *walk order* over a sequence of data elements or an event stream from which code is generated
  - JSP was one of the earliest model-driven development methods (from specifications, code is generated)

**Design Question:**
How is the data structured?
so that the algorithms can homomorphically be derived from its structure
Jackson Structured Diagrams (Jackson Process Trees)

A Jackson Structured Diagram (JSD Jackson Process Tree) is a function free with iteration and alternatives. Its tree constructors stem from regular expressions:

- **Sequence**: transforms to sequenced statements
- **Repetition**: transforms to loops or recursion (Kleene star)
- **Alternative**: transforms to if- and case-instructions

\[
\text{produce tea = (fetchGreenTea | fetchBlackTea)* AddBoilingWater Wait}
\]

// regular expression in regular language:
produceTea = (fetchGreenTea | fetchBlackTea)* AddBoilingWater Wait
Example for Data-Oriented Design: Jackson Structured Programming JSP

- **Notation:**
  - Jackson Structured Diagrams JSD (regular actions), equivalent to regular expressions on actions and finite state machines

- **Development Process:**
  - **Elaboration:** Draw JST trees for inputs and outputs
  - **Transformation:** Merge them
  - **Elaboration:** List the operations and allocate to program parts
  - **Elaboration:** Convert program to code (generate code)
  - **Elaboration:** Add conditions

- **Heuristics:**
  - Readahead
  - Backtracking
  - Program inversion if structure of input does not match output

- **Extension points:**
  - Where can sub-data structures be added?
When Should JSP Be Applied?

- JSP is good for problems that are “governed” by a data structure that corresponds to a regular expression:
  - if data has the structure of a regular expression
  - and input is homomorphic to output
  - -> Algorithm becomes homomorphic to data structure

- JST can describe the activity in a DFD (instead of minispecs in pseudocode)
  - Then, input is read from the input channels until end-of-stream
  - Output is produced by the JST

- **Table processing in information systems** is a perfect application area for JSP
  - DFD form the data flow; JSP is the specification of the elementary activities
  - The generated implementation is in COBOL(!) or another imperative language
Deriving a Regular Grammar from a JSD Tree

- The generated grammar can be fed into a *parser generator* to produce a parser recognizing the order of events, e.g., [www.antlr.org](http://www.antlr.org)

```plaintext
Grammar TeaPot {
  RULES
  ProduceTea ::= PutTealnPot
              AddBoilingWater  Wait .
  PutTealnPot ::= PourSpoonTealnIntoPot* .
  PourSpoonTealnIntoPot ::= FetchGreenTea
                            | FetchBlackTea
}  
```

<<generate>>
Deriving a System of Procedures from the JSD Tree

```
procedure ProduceTea() {
    PutTeaInPot();
    AddBoilingWater();
    Wait();
}

procedure PutTeaInPot() {
    while (condition) {
        PourSpoonTeaIntoPot();
    }
}

procedure PourSpoonTeaIntoPot() {
    if (condition) {
        FetchGreenTea();
    } else {
        FetchBlackTea();
    }
}```
Table- and Record-Manipulation Programs in Information Systems with JSP

- Many information systems rely on relational data processing with tables containing records (tuples) with information about employees, insured persons, members of networks, unemployed people, customers, etc.
- Algorithms on these tables with records can easily be expressed by JSP process trees:
  - „which persons earn more than 1500€ in our company?“ (threshold query)
  - „who earns most of our Austrian employees“ (max query)
  - „compute the average salary of our employees“ (avg operator)
  - „how much would a 5% increase of salary cost our company?“ (map-reduce operator)

<table>
<thead>
<tr>
<th>id</th>
<th>Name</th>
<th>FirstName</th>
<th>Street</th>
<th>Town</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>John</td>
<td>Silver</td>
<td>Obergasse 2a</td>
<td>Wien</td>
<td>1200€</td>
</tr>
<tr>
<td>13</td>
<td>Bobby</td>
<td>Brown</td>
<td>Traubengasse 12</td>
<td>Bad Tölz</td>
<td>600€</td>
</tr>
<tr>
<td>14</td>
<td>Frank</td>
<td>Foster</td>
<td>Blumenweg 6</td>
<td>München</td>
<td>2000€</td>
</tr>
<tr>
<td>20</td>
<td>Sue</td>
<td>Smith</td>
<td>Tulpengasse 3</td>
<td>Füssen</td>
<td>2300€</td>
</tr>
<tr>
<td>25</td>
<td>Mary</td>
<td>Miller</td>
<td>Heurigenweg 2</td>
<td>Linz</td>
<td>1500€</td>
</tr>
</tbody>
</table>
„Big Data“: Analysing with Algebraic Operators
A Table-Processing Program (Sum and Max)

- Operators **Sum, Max, Min, Avg, Map, Reduce, Map-Reduce, Group-By** are simple to use
- JSP was used to generate COBOL applications in banks and insurances
- → JSP was also the first Big-Data approach

```
SumUpSalaries

Open table

Print header

Sum := 0
CurrentMax = NIL

Print Date
Print "Salary Summary"

Process record *

Sum += CurrentRecord.Salary

If (CurrentMax < CurrentRecord.Salary) then CurrentMax := CurrentRecord.Salary

Print footer
Print "Average salary is ", Sum
Print "Max Salary is", CurrentMax

Close Table
```
26.2 Programming with Data Structure Grammars

Grammars can indirectly specify a Visitor for a data structure
A context-free grammar extends a regular grammar with free recursion: left, right, intertwined

Like in the regular grammar case, from the grammar similar code can be derived

- Visitors, parsers, generators

Grammar TeaPot

```
RULES
ProduceTea ::= PutTeaInPot
              AddBoilingWater
              Wait.

PutTeaInPot ::= PourSpoonTeaIntoPot*
               AddBoilingWater.

PourSpoonTeaIntoPot ::= FetchGreenTea | FetchBlackTea.

AddBoilingWater ::= BoilWater AddBoilingWater ProduceTea.
```
Applications of Grammar-Driven Programming

- EBNF is the standardized grammar language for all kind of actions based on context-free languages
  - Generation of code: parsers, generators, analyzers visitors

- **Parsing** character streams in compilers and software tools
  - Many parser generators exist
  - But parsing of lists of objects is also possible

- **Generators** of data
  - Test data generators for databases, compilers, software tools, metric tools, BI tools,...

- **Visitors** and **Analyzers** for complex data structures
  - Complex Big Data applications, which are non-regular
  - Complex Event Recognition in event streams in cyber-physical and embedded systems
    - „If several cars enter a parking house simultaneously through different gates, who gets the last free parking lot?“
26.3 Extensibility of JSD- and Grammar-Based Applications

Extensibility Question:
How can the data structure be extended?
so that the extended algorithms can be derived
Tree Constructors in a JSD are Open Constructs

- A new slice (view) can be added easily to the core algorithm (aspect-based extension, see chapter „Aspect-oriented development“)

```
SumUpSalaries

Open table

Print header

Print Date

Print "Salary Summary"

Sum := 0
CurrentMax = NIL

Process record *

Sum += CurrentRecord.Salary

If (CurrentMax < CurrentRecord.Salary) then CurrentMax := CurrentRecord.Salary

Close Table

Print footer

Print "Average salary is ", Sum

Print "Max Salary is", CurrentMax
```
Tree Constructors in a JSD are Open Constructs

- A new slice (view) can be added easily to the core algorithm (aspect-based extension, see chapter „Aspect-oriented development“)

```
SumUpSalaries

Open table

Print header
  - Print Date
  - Print "Salary Summary"

Print header
  - Sum := 0
  - CurrentMax = NIL

Process record *
  - Sum += CurrentRecord.Salary

If (CurrentMax < CurrentRecord.Salary) then CurrentMax := CurrentRecord.Salary

If (CurrentMin > CurrentRecord.Salary) then CurrentMin := CurrentRecord.Salary

CurrentMin := 0

Close Table

Print footer
  - Print "Average salary is ", Sum
  - Print "Max Salary is", CurrentMax
  - Print "Min Salary is", CurrentMin
```
Further Data-Driven Design Methods

- **Grammars:**
  - **String Grammars** can be used to generate parsers
  - **Attribute grammars** define more complex languages (→ course MOST)
    - Structure function spaces according to a hierarchic data structure
  - **Graph grammars** describe the structure of graphs
    - Room generation in MOOD games
    - Test data generation for graphs

- **Map-Reduce based „Big Data“ Processing**
  - Modern Map-Reduce frameworks such as Hadoop, Sparc, Flink (Apache) offer distributed processing of data with many operators
The End

- Why is table and record processing important? Describe how the operators max, min, avg, sum are used on the records of a table.
- Give an example for a DFD in which the activities are specified by JSD.
- Why will COBOL never die? (unfortunately)
- Compare the structure of a JST with its generated implementation in an imperative language.
- Do the same for a generated grammar.