COMP6209: Automated Code Generation

...or:
Programs that write Programs

Julian Rathke
jr2@ecs.soton.ac.uk
Introduction
Historically, progress in software development has relied on reuse.

- originally based on abstraction and encapsulation:
  - function libraries (LAPACK, Booch, …)
  - components (Java Beans, COM / .NET)
  - dedicated “processors” (SQL processors)
- complemented by open abstractions:
  - frameworks (Eclipse, Struts, Rails, …)
  - design patterns (GoF patterns)
- middleware systems (CORBA, WebSphere, …)
Reuse-oriented methods have reached their limit.

Libraries and components:

- encapsulation can prevent optimization

Math: \[ C = (A+B)^\top \]

Library code: add(A, B, M); transpose(M, C);

Base code:

```c
for(i=0;i++;i<\text{A.rows}())
  for(j=0;j++;j<\text{A.cols}())
    M[i,j] = A[i,j]+B[i,j];

for(i=0;i++;i<\text{M.rows}())
  for(j=0;j++;j<\text{M.cols}())
    C[j,i] = M[i,j];
```

Optimized code:

```c
for(i=0;i++;i<\text{A.rows}())
  for(j=0;j++;j<\text{A.cols}())
    M[i,j] = A[i,j]+B[i,j];
    C[j,i] = A[i,j]+B[i,j];

for(i=0;i++;i<\text{M.rows}())
  for(j=0;j++;j<\text{M.cols}())
    C[j,i] = M[i,j];
```

⇒ split into two procedures makes loop fusion difficult!
Reuse-oriented methods have reached their limit.

Libraries and components:
• encapsulation can prevent optimisation
• (static) error checking is limited
  Math: \[ C = A + B \]
  Library code: `add(A, B, C);
  Base code: 
  ```
  for(i=0;i++<A.rows())
  for(j=0;j++<A.cols())
      C[i,j] = A[i,j]+B[i,j];
  ```
  – what happens if dimensions of \( A \) and \( B \) are different?
  ⇒ prefer compile-time errors over run-time errors
      (some experimental languages would catch this error!)
  ⇒ prefer domain-specific error messages
Reuse-oriented methods have reached their limit.

Libraries and components:

- encapsulation can prevent optimisation
- (static) error checking is limited
- combinatorial explosion (library scaling problem)
  - vertical vs horizontal scaling
  - individual components
  - but multiple independent dimensions
    - memory management
    - concurrency / transaction handling
    - iteration

Reuse-oriented methods have reached their limit. (cont’d)

Frameworks:

• APIs too large / difficult
  – see library scaling problem
  – need to write large amounts of **boilerplate code**
    ▶ constructors, getters/setters, equality, hashing,…
  ▶ complicated usage patterns

```java
FileWriter fout = new FileWriter("foo.txt");
BufferedWriter bout = new BufferedWriter(fout);
PrintWriter pout = new PrintWriter(bout);  // (Java)
```

• framework evolution
  – interaction between framework and completion code needs constant updating
Reuse-oriented methods have reached their limit. (cont’d)

Middleware:

• APIs too large / difficult
  – see frameworks
• runtime overheads
  – enforcement of middleware protocol at runtime
Reuse-oriented methods have reached their limit. (cont’d)

Causes of the limitations:

• insufficient / wrong abstractions ("geek egoism")
  – need domain-specific abstractions

• functionality == components ("embodiment")
  – many concepts not bound to (single) components
    ▶ distribution  ▶ security
    ▶ persistence

• host language restrictions ("linguistic poverty")
  – cannot extend syntax to reflect domain notations
    ▶ mathematical operators  ▶ regular expressions
    ▶ parser combinators  ▶ SQL query syntax
Further progress requires more automation of programming.

Need to free programmer from routine jobs by

• ... using better abstractions!
  – consider programs as data objects (e.g., aspects, templates, models)

• ... using more powerful mechanisms!
  – write programs that write programs (e.g., meta-programming, model transformations, ...)

• ... using better languages!
  – capture domain-specific abstractions

⇒ Make computers do the programming!
“Automation of programming” can be hidden in programming languages.

- compilers often simplify programs into a core syntax:
  \[ a[i][j] \mapsto \ast\ast(a+i)+j \]
- pre-processors (e.g., \texttt{cpp}) unfold macro definitions:
  ```cpp
  #define SWAP(T,x,y) {T tmp=x; x=y; y=tmp;}
  ...
  int lo, hi;
  ...
  SWAP(int, lo, hi);
  ```
- templates --- “macros for classes”
- aspects --- code fragments “woven” into base code
Programming can be automated by dedicated separate tools. (cont’d)

- scanner / parser generators (e.g., JavaCC)
  - input: grammar rules with embedded action code
  - output: code comprising
    ▶ functions for recursive descent parse rules
    ▶ action code spliced in at correct locations
    ▶ driver code (“boilerplate”)
  - generator: implementation of LL-parsing theory
    ▶ analyses input file
    ▶ analyses grammar
    ▶ constructs parse functions
    ▶ writes driver code and functions
Programming can be automated by dedicated separate tools. (cont’d)

- scanner / parser generators
  - input: grammar rules with embedded action code
    
    ```
    expression = ["+"|"-" ] term { ("+"|"-" ) term } .
    term = factor { ("*"|"/" ) factor } .
    ```
  - output: recursive descent parse functions
    ```
    void expression(void) {...}
    ...
    void term(void) {
      factor();
      while (sym == times || sym == slash) {
        getsym(); factor();
      }
    }
    ```
Programming can be automated by dedicated separate tools. (cont’d)

- scanner / parser generators
- CASE tools
  - common design
    - input: system configuration represented in GUI
    - output: code skeleton
    - generator: traverses system structure, emits text
  - many different flavors
    - class diagrams to SQL (object-relational mapping)
    - class diagrams to EJB
    - message sequence charts to process stubs
    - message sequence charts to web pages
    - …
Programming can be automated by dedicated separate tools. (cont’d)

- scanner / parser generators
- CASE tools
  - e.g., Simulink Coder
    (formerly Real-Time Workshop)
Course Outline

• Macros and language embeddings

• Aspect-oriented programming  
  – Separation of concerns  
  – Static and dynamic aspects  
  – AspectJ  
  – Research topics in AOP

• Generic programming  
  – Templates  
  – Generics

• Model-based software development  
  – Domain modelling: EMF  
  – Model transformations  
  – Domain Specific Languages: Xtext

Abstraction levels:  
programming  
meta-programming  
modelling
Course Goals

• understanding of basic concepts in
  – aspect-oriented programming
  – templates and template meta-programming
  – model-based software development

• basic skills in
  – aspect-oriented programming
  – meta-programming
  – modelling and model transformation