Generative Software Development

Description:
The purpose of this course is to give a general introduction to generative techniques in software development. The focus will be on generics in various programming languages.

Course team:
- Robert Glück <glueck@diku.dk>
- Jyrki Katajainen <jyrki@diku.dk> (course coordinator)
- Torben Mogensen <torbenm@diku.dk>
- Peter Sestoft (KVL and ITU Copenhagen) <sestoft@dina.kvl.dk>

Course home page:
http://www.diku.dk/forskning/performance-engineering/Generative-software-development/
Who Am I

shell> whoami
jyrki

Algorithmist,

Programmer, and

Teacher

I am the leader of the Performance Engineering Laboratory.

I claim to be an expert in programming, but I do not make any claims about expertise in generative software development.
Generative Programming

“Generative programming is about modeling and implementing system families in such a way that a given system can be automatically generated from a specification written in one or more textual or graphical domain-specific languages.”

[Czarnecki, interview 2003–2004]

“Generative programming is about automating the manufacture of intermediate and end-products (i.e., components and applications).”

[Czarnecki and Eisenecker 2000, p. 131]
Automation Levels

**Manual assembly:** Applications are assembled from components manually.

**Automated assembly support:** The assembly of components is supported with various tools including component browsing and search tools and generators automating selected aspects of applications development.

**Automatic assembly:** When a customer orders an application, the entire application is generated based on the order record (except the parts requiring customary development).

Generative programming intends to achieve the last level, but it acknowledges the possibility of different levels of automation.
“I now prefer to use the term generative software development in order to convey the fact that the generative approach is concerned with the full range of software development activities.”

[Czarnecki, interview 2003–2004]
Generic Programming

The term **generic programming** has been used in at least four different but related meanings.

1. Programming with *generic parameters*
2. Programming by abstracting from concrete types
3. Programming with parametrized components
4. Programming method based on finding the most abstract representation of efficient algorithms

[Czarnecki and Eisenecker 2000, Chapter 6]
Discussion: Expectations

- Why are you here?
- What do you expect from this course?
Course Overview

**Monday:** [Jyrki]  
Generative software development

**Tuesday:** [Peter]  
Generics in Java and C#

**Wednesday:** [Jyrki]  
Generic programming techniques in C++

**Thursday:** [Robert]  
Theoretical foundations

**Friday:** [Torben]  
Generic programming and metaprogramming in functional languages
Monday [Jyrki]

We are in this room the whole day.

**9.15–10.00**  
Introduction to the course

**10.15–11.30**  
General introduction to generative software development

**11.30–12.00**  
- Group formation  
- Distribution of the topics for the first brief  
- Administration for getting accounts to our computers

**12.00–14.00**  
Lunch and group work (2–3 people per group)

**14.00–15.00**  
Presentations of group work

**15.00–**  
First brief; deadline tomorrow by 9.15
Course Activities

**Lectures:** 3 hours each day

**Exercises:** 2 hours each day

**Briefs:** 5 in all; one each day

We provide copies of the most important material related to the topics covered by the speakers (see resources on the course home page).

Active participation and at least 25% of all exercises/briefs are required to get the course credit.
Briefs

- Summary of the given topic
- Length about one A4
- Should contain 2–3 references to the most relevant sources (if you have used any)

“Of course, students don’t learn by being lectured at, anyway; they learn by thinking hard, solving problems, dissecting proofs. Requiring them to write briefs was the most important component of our teaching. . . . After students have thought hard about a topic, a lecture can help them sort out and organize their thoughts.”

[Simon 1996, p. 97]
Workload

1 ECTS $\approx$ 25 hours

This course gives 2.5 ECTS
Related Depth Courses

This course is a breadth course.

**Quarter 3:** Scripting languages and domain-specific languages [Torben and Julia]

**Quarter 4:** Generic programming and library development [Jon, Jyrki, and Kenny]

**Projects:** For possible topics, see CPH STL report 2005-5 which is available at http://www.cphstl.dk/reports.html
Recommended Reading

Coming Soon . . .

- some terminology
- five examples of generative programming
- quick summary of the key ideas in generative software development
Domains

**Domain:** An area of knowledge

- scoped to maximize the satisfaction of the requirements of its stakeholders
- includes a set of concepts and terminology understood by practitioners in that area
- includes the knowledge of how to build software systems (or parts of software systems) in that area.

1. A bank accounting domain includes concepts such as accounts, customers, deposits, withdrawals, and so on.

2. In algorithmic domains, abstract data types and algorithms are the main abstractions.
Domain-Specific Languages

Also known as little languages.

**Domain-specific language (DSL):** A language designed for writing system specifications. It can be used to “order” concrete members of a system family.

**Fixed separate DSLs:** These languages are usually implemented by a separate translator. Examples: \textsc{LaTeX} and SQL.

**Embedded DSLs:** The languages are usually embedded in a general-purpose programming language. Example: C++ providing built-in metaprogramming facilities.

**Modularly composable DSLs:** The idea is to view each DSL as a component and different DSL components can be composed in different configurations. Example: embedded SQL.
Generators

**Generator:** A program that takes a higher-level specification of a piece of software and produces its implementation.

In general a generator performs the following tasks:

- Checks the validity of the input specification and reports warnings and errors if necessary.
- Completes the specification using default settings if necessary.
- Performs optimizations.
- Generates the implementation.
Shell> cd /usr/include/g++-3/stl_algobase.h
Shell> cat stl_algobase.h

...template <class _Tp>
inline void swap(_Tp& __a, _Tp& __b) {
  _Tp __tmp = __a;
  __a = __b;
  __b = __tmp;
}
...

Shell> cat swap.c++
#include <algorithm>
#include <cassert>

int main() {
  int i = 10;
  int j = 0;
  std::swap(i, j);
  assert(i == 0 && j == 10);
  return 0;
}

Shell> g++ swap.c++
Shell> ./a.out

Scenario: from the generic swap<>() routine a specialized int-version is generated automatically

DSL: C++ templates

Generator: C++ compiler
Macro Facilities

shell> cat macro.c++
#define XCAT(x, y) x ## y
#define CAT(x, y) XCAT(x, y)

#define LOOP_BODY(n, continuation)
unsigned int parity = n & 1;
if (parity != 0) goto CAT(label, __LINE__);
n = n / 2;
continuation;
CAT(label, __LINE__):
n = 3 * n;
n = n + 1;
continuation

void pure_c(unsigned int n) {
  if (n <= 1) return;
  loop:
    LOOP_BODY(n, if (n > 1) goto loop; return);
}

Scenario: program text is generated automatically based on textual substitution

DSL: C++ macros

Generator: C++ preprocessor
Database Queries

SELECT p.name
FROM Person p
WHERE NOT EXISTS (  
    SELECT *
    FROM Car c
    WHERE NOT EXISTS (  
        SELECT *
        FROM Driver d
        WHERE d.cpr = p.cpr AND d.regno = c.regno
    )
)

**Scenario:** A parser maps the query to an inefficient relational algebra expression, a query optimizer optimizes the expression automatically, and an efficient query evaluation program is generated from the optimized expression.

**DSL:** SQL

**Generator:** relational query optimizer (or database management system in general)
Code Generation in Benz

... else:
    program = """\n%(include_statements)s
int main() {
    double primal_in_seconds = measure_primal(%(constructor_call)s);
    double t = primal_in_seconds * %(time_factor)s;
    std::cout << t << std::endl;
    return 0;
}
""" % self.__dict__

Scenario: A user fills in a form, which is a Python program, to benchmark his or her C++ program. Benz generates a driver, executes the driver with the inputs specified by the user, and reports the execution time.

DSL: Python (a collection of prefabricated classes)

Generator: Python interpreter

© Performance Engineering Laboratory 22
Generation of Web Pages

```php
<?php
    include("index-header.html");
?>

<H3>Source code</H3>
...

<?php
    $filemod = filemtime(__FILE__);
    $filemodtime = date("F j Y h:i:s A", $filemod);
    echo (<"FONT SIZE="2">Last change $filemodtime</FONT>");
?>
...
```

**Scenario:** Client requests a web page written in PHP, web server locates instructions file, generates the corresponding HTML code, and sends it to the client.

**DSL:** PHP (embedded in HTML)

**Generator:** PHP engine inside the web server
Discussion: Example Application

a) A course-evaluation system for one course

b) A course-evaluation system for all courses offered at our university

c) What next?
Discussion: Applications

Give some additional application areas were generative techniques would be useful.
Applications Areas

Generators for the application logic, user interfaces, database integration, network integration, and so.

The generated products may also contain non-software artifacts, such as test plans, manuals, tutorials, help systems, maintenance and troubleshooting guidelines, and so on.

Yet another category of artifacts is test data, test scripts, and profiling and testing instrumentation code.
Claim: In the future, most programming will be based on generative techniques.

Consequence: General-purpose programming languages should be improved to support this type of programming.
Research Problem II

- We need technology to render and compile domain-specific languages, and methods to systematically design domain-specific languages.

- We should provide methods to systematically scope and develop domain-specific languages, underlying architectures and components.

[Czarnecki, interview 2003–2004]
Research Problem III

**Generic architecture:** A generic architecture can be thought of as a fixed frame with a number of sockets where we can plug in some alternative and extension components. A generic architecture has a fixed topology and fixed interfaces.

**Highly flexible architecture:** A highly flexible architecture supports structural variation in its topology.

How to make generic architectures highly flexible?
Discussion: Research

Can you see other important research directions?
What Is New?

- The book by Czarnecki and Eisenecker describes the state of the art in 2000.
- They try to unify the field and tie different pieces together.
- They list best practices.
- They approach the field in a systematic way.
Feature Modelling

**Feature modelling** is the activity of modelling the common and the variable properties of concepts and their interdependencies.

- commonalities
- variabilities
- dependencies

**Functional features:** Operations provided by a data structure.

**Implementation features:** Different layouts for storing the elements in a container, various optimizations, alternative implementation techniques, and so on.
What Next?

- You should read the book by yourself.
- When doing this, the pictures the text generates in our mind are important.