Instructional Design of a Programming Course
— A Learning Theoretic Approach

Michael E. Caspersen
Jens Bennedsen
Calibration of Expectations ;-)
Theses and Research Question

**Theses**
Revealing the programming process to novices eases and promotes the learning of programming.

Teaching skill as a supplement to knowledge promotes the learning of programming.

**Research Questions**
What is the foundation in learning theory that supports (or contradicts) the theses above?
Agenda

**Background**
Motivation, model-based programming, course context

**Learning-Theoretic Foundation**
Cognitive (architecture, load theory, and apprenticeship)

**Pattern-Based Instruction**
Direct injection of cognitive schemas

**Instructional Design**
Macro and micro perspective

**Conclusion**
Future work, evaluation, how? Discussion...
Agenda

Background
Motivation, model-based programming, course context

Learning-Theoretic Foundation
Cognitive (architecture, load theory, and apprenticeship)

Pattern-Based Instruction
Direct injection of cognitive schemas

Instructional Design
Macro and micro perspective

Conclusion
Future work, evaluation, how? Discussion...
Grand Challenges in Computing

Perception of computing

Innovation

Competencies

Formalism

About e-learning

Pre-university issues

Programming issues

Understand the programming process and programmer practice to deliver effective educational transfer of knowledge and skills.

Grand Challenges in Computing (GCC ’04)

Education and Research

Newcastle upon Tyne

29-31 March 2004
GCE 4: Programming Issues

“The particular concern is that, after more than forty years of teaching an essential aspect of our discipline to would-be professionals, we cannot do so reliably.”

“Indeed, there are perceptions that the situation has become worse with time.”

A Summary

1. Incorporate the results of psychological studies into our curricula and textbooks.
2. Establish student aptitude for programming.
3. Understand the programming process and programmer practice to deliver effective educational transfer of knowledge and skills.
4. Encourage students to practice programming.
5. Establish assessment methods that assess individual programming competency effectively.
## Failure Rates

<table>
<thead>
<tr>
<th></th>
<th>Universities</th>
<th>Students</th>
<th>Avg. fail rate</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total</strong></td>
<td>54</td>
<td>5,513</td>
<td>33%</td>
</tr>
<tr>
<td><strong>Non US</strong></td>
<td>17</td>
<td>2,653</td>
<td>41%</td>
</tr>
</tbody>
</table>

Average failure rate for introductory programming courses

Many European universities reported failure rates of more than 50%.

The maximum failure rate reported was 95%.

Internal report of community colleges in a coalition to improve their retention rates in computing:

One school reported an avg. failure rate over a ten year period of 90%!
Over the past 25 years, study after study, even multi-institutional and multi-national studies, have provided empirical evidence that students cannot program and that the major problems they experience are composition-based—how to put the pieces together. It is not syntax!

We have a long-standing problem of international scale, which we are aware of, and yet we persist to teach programming primarily by explaining language constructs and show-casing finished programs even though it is procedural knowledge and strategies for putting the pieces together, that is needed!

From language issues to programming skills
In Short...

“Houston, we have a problem!”
Course Description

Aims: The participants will after the course have insight into principles and techniques for systematic construction of simple programs and practical experience with implementation of specification models using a standard programming language and selected standard classes.

7 weeks (a quarter)
1/3 of the students time

4 lecture hours per week

Goals: Upon completion the participants must be able to apply fundamental constructs of a common PL identify and explain the architecture of a simple program identify and explain the semantics of simple specification models implement simple specification models in a common PL apply standard classes for implementation tasks.

Weekly mandatory assignment (individual)

~ 400 students per year (mixed group)

Evaluation: Each student is evaluated through a practical examination where the student solves a simple programming task.

~ 20 TAs
Programming as a Modeling Process

The process involves identification of concepts and phenomena in the referent system and representation of these in the model system.

The process consists of three subprocesses:

- Abstraction in the model system (implementation)
- Abstraction in the referent system (analysis)
- Modeling (design)

No particular ordering is imposed on the subprocesses.
General Approach

Model-Based
Programming tasks starts from a specification model. The specification model is expressed as a class model with functional specifications of all methods.

Progression
Models become increasingly complex during the course. Associated systematic programming techniques (at three levels). Language issues are covered “by need”.

Programming Techniques
Inter-class structure: Standard coding patterns for the implementation of relations between classes
Intra-class structure: Class invariants and techniques for evaluating these
Method structure: Algorithmic patterns and loop invariants.
Course Progression
Model-Based Progression

Stand-alone class

Simple association and composition

Recursive Association

Association (to another class)

Implementing an interface
Agenda

Background
Motivation, model-based programming, course context

Learning-Theoretic Foundation
Cognitive (architecture, load theory, and apprenticeship)

Pattern-Based Instruction
Direct injection of cognitive schemas

Instructional Design
Macro and micro perspective

Conclusion
Future work, evaluation, how? Discussion...
The Human Cognitive Architecture

Learning
New information is processed in WM to form knowledge structures called schemas stored in LTM.

Schemas
Memory structures that permit us to treat a large amount of information elements as one.
Cognitive Load Theory

**Cognitive load**: the load on working memory during problem solving, thinking, reasoning, ...

**Cognitive load theory**: a universal set of learning principles that are proven to result in efficient instructional environments as a consequence of leveraging human cognitive learning processes

**The fundamental axiom of CLT**: that learning outcome is optimized when cognitive load fully utilizes the capacity of working memory with elements that allow for optimal schema acquisition
Three Categories of Cognitive Load

- **Intrinsic cognitive load (I)**
  - cognitive load intrinsic to the problem that cannot be reduced without reducing understanding

- **Extraneous cognitive load (E)**
  - a non-intrinsic cognitive load caused by instructional procedures that interfere with rather than contribute to learning

- **Germane cognitive load (G)**
  - a non-intrinsic cognitive load that contributes to, rather than interfere with, learning by supporting schema acquisition

\[ L = I + E + G \]

Optimize learning:
- minimize E
- maximize G
Selected CLT Effects

**Worked examples effect**
Alternation of worked examples and problems increase learning outcome and transfer.

**Variability effect**
Worked examples with high variability increase cognitive load and learning provided that intrinsic cognitive load is sufficiently low.

**Guidance-fading effect**
Complete examples followed by partially completed examples followed by full problems is superior to any of the three used in isolation.
Cognitive Apprenticeship

The theory of cognitive apprenticeship holds that masters of a skill often fail to take into account the implicit processes involved in carrying out complex skills when teaching novices.

**Modeling**
Give models of expert performance.

**Scaffolding**
Support given by the master to carry out a task.

**Fading**
The master gradually pulls back leaving the responsibility for performing the task more and more to the apprentice.

**Coaching**
The entire process of apprenticeship—overseeing the process of learning.
Agenda

Background
Motivation, model-based programming, course context

Learning-Theoretic Foundation
Cognitive (architecture, load theory, and apprenticeship)

Pattern-Based Instruction
Direct injection of cognitive schemas

Instructional Design
Macro and micro perspective

Conclusion
Future work, evaluation, how? Discussion...
Pattern-Based Instruction

Motivation
Patterns capture chunks of programming knowledge and skills.

Reinforces schema creation when cognitive load is “controlled” (+germane cognitive load).
(Cognitive science and educational psychology)

Origin

East et al. (1996) and Wallingford (1996) were among the first to accept the challenge put forward by Soloway.

Patterns at Two Levels

**Inter-class structure**
Standard coding patterns for the implementation of relations between classes.

**Intra-class structure**
Class invariants and techniques for evaluating these.

**Method structure**
Algorithmic patterns, elementary patterns and loop invariants.

- Association pattern
- Aggregation pattern
- Specialization pattern

- Sweep pattern
- Search pattern
- Divide, solve, and combine
...
class B {
    ...
}

class A {
    ...
    // A-fields
    public A() {
        ...
        bs = new ArrayList();
    }
    ...
    // A-methods
    private List bs;
    public void add(B b) {
        bs.add(b);
    }
    public void remove(B b) {
        bs.remove(b);
    }
}

import java.util.*;
Sweep Pattern (findOne)

class B {}

class A {
    ...  
    private List<B> bs;

    public B findOneX() {
        B res = bs.get(0);
        for (B b : bs) {
            if ("b is a better X than res") {
                res = b;
            }
        }
        return res;
    }
}
Inductive vs. Deductive

**Inductive**
Patterns are “discovered” by the students after N examples.

**Deductive**
Patterns are presented up-front and students are asked to apply them to N examples.
Agenda

Background
Motivation, model-based programming, course context

Learning-Theoretic Foundation
Cognitive (architecture, load theory, and apprenticeship)

Pattern-Based Instruction
Direct injection of cognitive schemas

Instructional Design
Macro and micro perspective

Conclusion
Future work, evaluation, how? Discussion...
Practice consume before produce

Present worked, exemplary examples

Reinforce patterns and conceptual frameworks

... Allows more interesting things to be done.

Helps separate spec and impl.

Applies at many levels (code specification, class libraries, design patterns, frameworks, ...)

<table>
<thead>
<tr>
<th></th>
<th>Method</th>
<th>Class</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Extend</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Create</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Course Phases

(1) **Getting started**: Overview of fundamental concepts. Learning the IDE and other tools.

(2) **Learning the basics**: Class, object, state, behaviour, control structures.

(3) **Conceptual framework and coding recipes**: Control structures, data structures (collections), class relationship, recipes for implementing structure (class relations).

(4) **Programming method**: The mañana principle, schemas for implementing functionality.

(5) **Subject specific assignment**: Practice on harder problems.

(6) **Practice**: Achieve routine in solving standard tasks.
Faded Guidance and Cognitive Apprenticeship

1. In a lecture we present an example of development of a program with two classes, Playlist and Track.

2. A video presentation of a partial development example (say Account and Transaction) is made available.

3. A lab session follows where the students interact with, modify, and extend both examples.

4. A follow-up exercise is provided where the students extend the Playlist-Track example by adding an Image class.

5. In the following week we give a mandatory assignment where the students implement a system of three classes, say Notebook, Note, and Keyword.

Worked examples with high variability increase cognitive load and learning provided that intrinsic cognitive load is sufficiently low.
<table>
<thead>
<tr>
<th>Agenda</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Background</strong></td>
</tr>
<tr>
<td><strong>Learning-Theoretic Foundation</strong></td>
</tr>
<tr>
<td><strong>Pattern-Based Instruction</strong></td>
</tr>
<tr>
<td><strong>Instructional Design</strong></td>
</tr>
<tr>
<td><strong>Conclusion</strong></td>
</tr>
</tbody>
</table>
Conclusion

Instructional Design of a Programming Course
Based upon results of cognitive science and educational psychology.
   Model- and pattern-based approach.
   Progression defined by complexity of specification models.
   Focus on skill development rather than just knowledge.

Objects-First Debate
   Heated debate (SIGCSE mailing list, symposium, ...)

Formal Evaluation of our Approach
   Be my guest! ;-)

Instructional Design of a Programming Course
— A Learning Theoretic Approach

Jens Bennedsen
Michael E. Caspersen
Sample Final Exam Assignment

1. Create a simple class `Track` ... with constructor and a `toString` method...
2. Create a test method...
3. Create two instances...
4. Create a new class `Playlist` ...
5. Implement the methods `addTrack` and `removeTrack`
6. Modify the test method to create a `Playlist` object...
7. Implement the method `findShortestTrack`
8. Modify the test method to use the `findShortestTrack` method to...
9. Let the `Track` class implement the `Comparable` interface...
10. Implement the method `printTracks` and modify ...

---

<table>
<thead>
<tr>
<th>Track</th>
</tr>
</thead>
<tbody>
<tr>
<td>String artist</td>
</tr>
<tr>
<td>String songName</td>
</tr>
<tr>
<td>int min</td>
</tr>
<tr>
<td>int sec</td>
</tr>
<tr>
<td>String toString()</td>
</tr>
</tbody>
</table>

* *

<table>
<thead>
<tr>
<th>Playlist</th>
</tr>
</thead>
<tbody>
<tr>
<td>String playlistName</td>
</tr>
<tr>
<td>void addTrack(Track t)</td>
</tr>
<tr>
<td>void removeTrack(Track t)</td>
</tr>
<tr>
<td>Track findShortestTrack()</td>
</tr>
<tr>
<td>void printTracks()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>void exam()</td>
</tr>
</tbody>
</table>