Teaching the Unifying Mathematics of Software Design

Emil Sekerinski
McMaster University
Hamilton, ON, Canada
emil@mcmaster.ca

ABSTRACT

We report on our experience on teaching the mathematics of reliable software design as a unifying force for various elements of software design, rather than as an additional element of software design. This is in line with the use of mathematics in traditional engineering disciplines, but in contrast to teaching a "formal method" optionally after an "informal" exposition to software design or teaching a formal method only with specific applications in mind.

1. INTRODUCTION

Whereas in the design of a mechanical device breaking design rules would quickly lead to recognizable failure, one can very well break rules of software design and still get a "sufficiently functional" and marketable product. Qualities of software are not as evident or measurable as qualities of physical products; *design qualities* are even harder to judge than the qualities that can be observed of a product. Students follow the rules of software design because they are told so and not because they would experience the consequences of not doing so. Students grow up with unreliable software to the extent that they consider such poorly working software to be normal or unavoidable. Job advertisements suggest that programming skills are sufficient to write software and students take software design courses with the expectation of preparing them for the job market. All this makes is difficult to convince students that software can be better designed, that it is worth doing so, and that it is worth learning the mathematics for doing so.

Typical textbooks on software design, or software engineering as they are sometimes called, devote few chapters on formal specification and verification, with the rest of the book not depending on those. Curricula in computer science and software engineering may include an upper level elective on specification, verification, or semantics. Students perceive the mathematics of software design as optional to their education and largely as a burden.

We report on our experience in teaching a sequence of two courses in software design in which mathematics is used as the unifying force of all core elements of reliable software design. This is in contrast to teaching a "formal method" optionally after an "informal" exposition to software design: our students had only taken an introductory first year course in programming and a course in discrete mathematics. The approach is more in line with the use of mathematics in traditional engineering disciplines. The two courses cover all core topics in software design rather than a specific topic for which a dedicated formalism or tool exist:

Uniform Design Notation A uniform textual design notation is used, in order to emphasize the similarities among the concepts and help students interconnect these concepts, rather than making students switch to a new mindset due to the notational differences. Graphical notations like flowcharts, class diagrams, and statecharts are presented as appropriate and defined as alternative representations of specific aspects in terms of the textual notation

Uniform Mathematical Basis A mathematical basis for all design constructs is given. A typed logic using the same type system as a programming language is used. Equational reasoning is used for all proofs because of the familiarity from calculus. Weakest preconditions are used for statements.

Middle-out Sequencing of Topics Courses on software design are typically structured according to the phases in which software is developed. However, students who have only written small programs so far, do not see the need for, say, elaborate requirements. Instead, we start with writing and analyzing small programs and gradually move to topics to which students become motivated, approximately in middle-out order of a normal software development. This also gives enough opportunities for students to catch up with their understanding of programming techniques.

The material is distributed over Software Design 1, a one semester second year course, and Software Design 2, a one semester third year course. Both are required for Computer Science at McMaster. The courses were taught repeatedly since 1999/2000 and had a peak enrollment of 190 students. Many students perceive these two courses as the core courses for their career in software development. The same material was presented in a condensed form in a graduate course in 2005/06 and in 2008/09.

WCCCE '09 Burnaby, BC, Canada
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

Program Annotations

$\{x \geq 0\}$
$z, u := 0, x$;
$\{$**invariant**
$\quad(z + u \times y = x \times y) \wedge$
$\quad(u \geq 0)\}$
while $u > 0$ **do**
$\quad z, u := z + y, u - 1$
$\{z = x \times y\}$

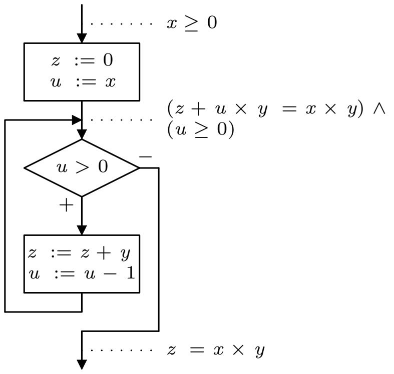


Figure 1: Excerpt from *Elements of Programming*

Statements with Partial Expressions

Assignment. Assume a: **array** N **of** T.

$$\begin{array}{lcl} wp(x := E, P) & = & \Delta E \wedge P[x\backslash E] \\ wp(a(E) := F, P) & = & \Delta E \wedge \Delta F \wedge (0 \leq E < N) \wedge P[a\backslash(a; E : F)] \end{array}$$

Conditional.

$$wp(\textbf{if}\, B\, \textbf{then}\, S, P) \quad = \quad \Delta B \wedge ((B \wedge wp(S, P)) \vee (\neg B \wedge P))$$
$$wp(\textbf{if}\, B\, \textbf{then}\, S\, \textbf{else}\, T, P) \quad = \quad \Delta B \wedge ((B \wedge wp(S, P)) \vee (\neg B \wedge wp(T, P)))$$

Repetition. If

$$\begin{array}{lcll} B \wedge P & \Rightarrow & wp(S, P) & (P \text{ is invariant of } S) \\ B \wedge P \wedge (T = v) & \Rightarrow & wp(S, T < v) & (S \text{ decreases } T) \\ B \wedge P & \Rightarrow & T > 0 & (T < 0 \text{ causes termination}) \\ P & \Rightarrow & \Delta B & (B \text{ is always defined}) \end{array}$$

then

$$P \quad \Rightarrow \quad wp(\textbf{while}\, B\, \textbf{do}\, S, P \wedge \neg B)$$

Figure 2: Excerpt from *Elements of Programming*

The next section elaborates on the topics covered in both courses, in the order of presentation. We conclude with an evaluation of the courses and a discussion.

2. MIDDLE-OUT SEQUENCING OF TOPICS

Lecture 1: Elements of Programming

The first course starts with an introduction to basic control structures. The notion of (total) correctness is defined through the triple $\{P\}\, S\, \{Q\}$. For the purpose of proving, correctness is restated in terms of weakest preconditions as $P \Rightarrow wp(S, Q)$. Using the rules of wp, intermediate annotations can be derived automatically and annotations can be restricted to preconditions, postconditions, and loop invariants. Equational logic is introduced by appealing to the analogy with calculus. A formal definition of the syntax of programs and annotations and of typing is included; trying to skip this did lead to syntactically incorrect mixtures of programs and annotations. Flowcharts, like in Fig. 1, are used for explaining annotations. Trying to use them in exercises did lead to spaghetti charts; they are still retained as they are intuitive and as they are later used for explaining exceptions. Limitations of machine arithmetic and partial expressions are discussed and formalized as in Fig. 2. Further control structures (case, repeat, and for statements) and recursion complete the picture. Basic sorting and searching techniques, as they are taught in introductory and high school courses, illustrate the use of annotations. Quicksort and Boyer-Moore search are used to demonstrate the need for correctness arguments of more complex algorithms. Stepwise refinement is illustrated with the example of printing images [1]. Proper programming style (indentation, comments, naming) is discussed and from that point on enforced. The assignments are with paper-and-pencil only and force students to argue about programs without running and testing them. While this comes as a surprise to students, it puts students with and without programming experience on the same level. Verification is continued to be practiced throughout both software design courses: our observation is that one semester is not sufficient for students to feel comfortable with writing annotations.

Lecture 2: Program Modularization

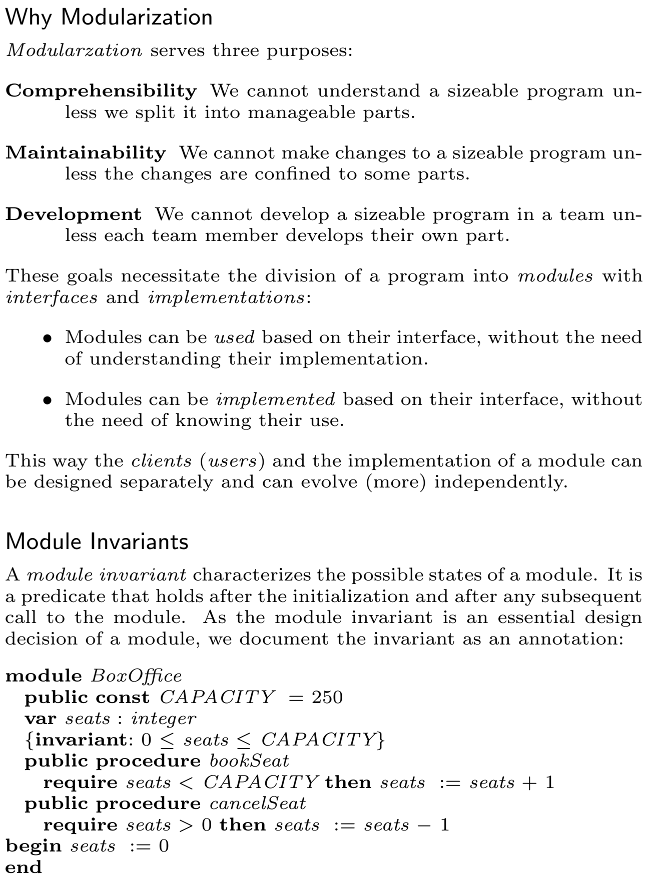
The goals and principles of modularization are discussed and a notation for modules is introduced, see Fig. 3. The principles are supported by a discussion of the consequences of local module invariants (cohesion) and global module invariants (coupling). The KWIC example is used for illustrating the difference in qualities that arise from different modularizations [5]. The key point is that students learn that there is not the ideal modularization, but only one for—explicitly stated—anticipated changes. The need for robustness of modules is discussed and defined formally. Exercises continue to practice formally reasoning about programs, but also show how to map modules to the constructs found in programming languages, in particular how encapsulation is enforced in common languages.

Lecture 3: Abstract Programs

Four means of abstraction are presented: (1) multiple assignments, as a way of abstacting from an arbitrary ordering of individual assignments, (2) guarded commands, as a way of expressing bounded nondeterminism, (3) specification statements, as a way of expressing unbounded nondeterminism, and (4) abstract data types, for data abstraction. These are first illustrated with simple, abstract algorithms and then with examples as typically found in books on algorithms and data structures. Exercises practice the use of abstract data types for modelling information systems, as a preparation for object-oriented modelling. The techniques are then used for the specification of modules: the previous notation of a syntactic interface of a module is augmented by a complete abstract specification. Abstraction continues to be repeated throughout both courses. While we tried to start the first course with abstract programs instead of concrete programs, our experience was that students didn't have an understanding of what they are abstracting from; we find that it takes students significant time to appreciate for example nondeterminism.

Lecture 4: Testing

The role and need for testing are discussed. Testing is presented as complementing verification. Testing the internal consistency of modules is illustrated with checking module invariants. Specification based testing is used for both black box and white box testing. The wp calculus is used for de-



Why Modularization

Modularzation serves three purposes:

Comprehensibility We cannot understand a sizeable program unless we split it into manageable parts.

Maintainability We cannot make changes to a sizeable program unless the changes are confined to some parts.

Development We cannot develop a sizeable program in a team unless each team member develops their own part.

These goals necessitate the division of a program into *modules* with *interfaces* and *implementations*:

- Modules can be *used* based on their interface, without the need of understanding their implementation.
- Modules can be *implemented* based on their interface, without the need of knowing their use.

This way the *clients* (*users*) and the implementation of a module can be designed separately and can evolve (more) independently.

Module Invariants

A *module invariant* characterizes the possible states of a module. It is a predicate that holds after the initialization and after any subsequent call to the module. As the module invariant is an essential design decision of a module, we document the invariant as an annotation:

```
module $BoxOffice$
  public const $CAPACITY = 250$
  var $seats : integer$
  {invariant: $0 \le seats \le CAPACITY$}
  public procedure $bookSeat$
    require $seats < CAPACITY$ then $seats := seats + 1$
  public procedure $cancelSeat$
    require $seats > 0$ then $seats := seats - 1$
begin $seats := 0$
end
```

Figure 3: Excerpts from *Modularization*

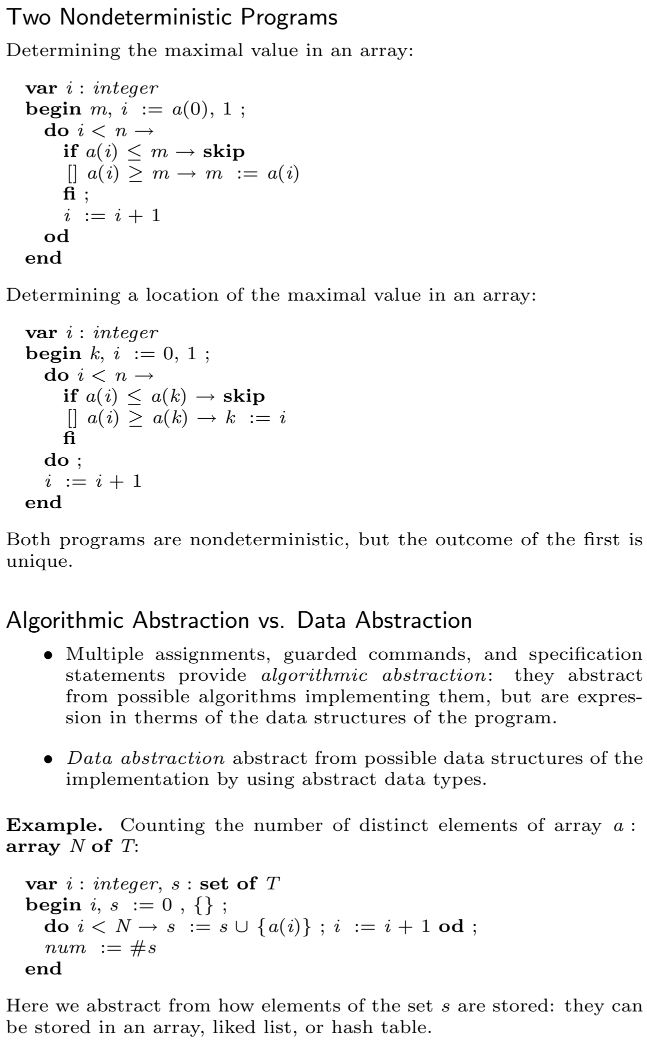
riving test cases to achieve various types of coverage, as in Fig. 5. Test strategies are discussed. The assignments practice writing both implementations of modules and test suites according to formal specifications; in one assignment both implementations and test suites are run against (faulty) ones from other students. From that point on students are convinced of the need for precise specifications.

Lecture 5: Exception Handling

Failures, the need for exception handling despite verification and testing, and ways of reacting to exceptions are discussed. The raise and the try-catch statements, as the dominant exception mechanism, are introduced textually and graphically, and then defined formally through *wp*. Verification of exception handing is practiced with small examples. The correct design of exception handlers in modular programs is discussed. Students should understand that, for example, an exception handler is supposed to re-establish the module invariant and establish an alternative postcondition. Programming exercises from now on have to include proper exception handling.

Lecture 6: Functional Specifications

A formal model of programs in terms of relations is given and connected to *wp*. The use of predicative specifications is discussed. Tabular specifications are used to discuss the concepts of completeness of consistency of specifications [6], see Fig. 7. Algorithmic refinement and data refinement are formalized with relations. Previous examples of module specifications and implementations are revisited and formally analyzed; the emphasis here is on understanding the concept of a refinement relation, as a preparation for class refinement.



Two Nondeterministic Programs

Determining the maximal value in an array:

```
var $i : integer$
begin $m, i := a(0), 1$ ;
  do $i < n \rightarrow$
    if $a(i) \le m \rightarrow$ skip
    [] $a(i) \ge m \rightarrow m := a(i)$
    fi ;
    $i := i + 1$
  od
end
```

Determining a location of the maximal value in an array:

```
var $i : integer$
begin $k, i := 0, 1$ ;
  do $i < n \rightarrow$
    if $a(i) \le a(k) \rightarrow$ skip
    [] $a(i) \ge a(k) \rightarrow k := i$
    fi
  do ;
  $i := i + 1$
end
```

Both programs are nondeterministic, but the outcome of the first is unique.

Algorithmic Abstraction vs. Data Abstraction

- Multiple assignments, guarded commands, and specification statements provide *algorithmic abstraction*: they abstract from possible algorithms implementing them, but are expression in therms of the data structures of the program.
- *Data abstraction* abstract from possible data structures of the implementation by using abstract data types.

Example. Counting the number of distinct elements of array a : **array** N **of** T:

```
var $i : integer$, $s$ : set of $T$
begin $i, s := 0 , \{\}$ ;
  do $i < N \rightarrow s := s \cup \{a(i)\}$ ; $i := i + 1$ od ;
  $num := \#s$
end
```

Here we abstract from how elements of the set s are stored: they can be stored in an array, liked list, or hash table.

Figure 4: Excerpts from *Abstract Programs*

Lecture 7: Object-Oriented Programs

Classes and inheritance are introduced textually and graphically. The object-oriented style is contrasted with the traditional style. A formal model of classes and inheritance is given that makes the additional complexity of object-oriented design explicit by translation into a model with variables and procedures only, see Fig. 8. Class invariants and class refinement are studied in this model. Class refinement is then used to justify "good" and "bad" used of inheritance. Small exercises enforce the understanding of the formalism whereas programming assignments practice class design without formal proofs.

Lecture 8: Object-Oriented Modeling

Object-oriented models are presented as an alternative, graphical way of specifying data structures. For this, class dia-

Path Coverage

Alternatively, we can derive a set of test cases such that all *paths* are covered. This includes coverage of all statements. In the example below, the paths are A-C, A-D, B-C, B-D:

```
**if** $a(0) < a(1)$ **then**
  $l := 1$            – A
**else**
  $l := 0$ ;          – B
**if** $a(l) < a(2)$ **then**
  $l := 2$            – C
**else**
  **skip**            – D
```

Test cases are determined by calculating the weakest precondition that excludes all alternative paths. For example, for testing the path B-C we calculate:

```
   $\neg(a(0) < a(1)) \;\wedge$
   $wp$ (**if** $a(0) < a(1)$ **then** $l := 1$ **else** $l := 0$, $a(l) < a(2)$)
= ”wp of **if**, logic”
   $(a(0) \geq a(1)) \;\wedge\; wp(l := 0,\ a(l) < a(2))$
= ”wp of $:=$ ”
   $(a(0) \geq a(1)) \;\wedge\; (a(0) < a(2))$
```

From there, we pick arbitrary values that satisfy the precondition, for example $a(0) = 5,\ a(1) = 4,\ a(2) = 7$.

Testing Modules

Since modules may have private variables, we can neither set nor inspect their values directly.

- In order to set their values to a desired state, we have to call a sequence of *modifiers* (modifying public procedures).
- In order to inspect their values, we have to call one or more *observers* (observing public procedures).

With testing in mind, we should include sufficiently many modifies and observers in the interface. This leads to the requirement of designing modules for *testability.*

Figure 5: Excerpts from *Testing*

grams are extended with various forms of associations that are given a textual definition. The use of object-oriented models for a partial (abstract) view and for guiding the modularization according to data is practiced (repeating that lesson from *Modularization*). The transition from an object-oriented model to an object-oriented implementation is explained as a refinement step, in accordance with the earlier formalization of refinement, and practiced informally.

Lecture 9: Requirements Analysis

The need for formulating requirements in the “user’s world” and the need for distinguishing user requirements from program specifications is discussed. The step of delineating the context of a software system is discussed with use cases and use case diagrams. The notion of the interaction of a software system with its environment is motivated with sequence diagrams. Proving the consistency of a specification with respect to sequence diagrams is discussed using *wp*, see Fig. 10. The derivation of test cases from sequence diagrams is discussed, thus connecting the topic of requirements analysis with specification, verification, and testing.

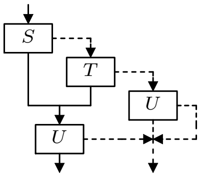
Lecture 10: Object-Oriented Design

Object-oriented techniques (like forwarding and delegation), design patterns, frameworks, subsystems, and components are discussed. Class invariants and class refinement are used in explaining class structures, but without formal proofs. Ten of the common 24 design patterns are selected. Assignments use the java.util framework for illustrating the use of design patterns when extending frameworks.

Finally Clause

A try-catch block may have a finally clause that is executed whether an exception occurred or not. Suppose S, T, U are statements:

try S **catch** T **finally** U



Either the catch or the finally clause is optional:

try S **catch** T = **try** S **catch** T **finally skip**
try S **finally** T = **try** S **catch raise finally** T

Weakest Exceptional Precondition

Recall that $wp(S, Q)$ is the weakest precondition such that statement S terminates and condition Q holds finally. We define:

$wp(S, Q, R)$ = weakest precondition such that S terminates and
– on normal termination Q holds finally
– on exceptional termination R holds finally

A statement that never raises an exception can be equivalently defined through weakest precondition or weakest exceptional precondition:

$$wp(S,\ Q) \ = wp(S,\ Q,\ \mathbf{false})$$

The weakest precondition $wp(S, Q, R)$ is monotonic in both Q and R:

if $Q \Rightarrow Q'$ and $R \Rightarrow R'$ then $wp(S, Q, R) \Rightarrow wp(S, Q', R')$

Figure 6: Excerpts from *Exception Handling*

Lecture 11: Reactive Programs

The characteristics of reactive programs are contrasted with those of transformational programs. Statecharts are introduced as a dedicated formalism for reactive programs. The event-based approach to reactive systems is contrasted with the state-based approach. The elements of statecharts are first illustrated and then defined in terms of guarded commands, following [9, 10], see Fig. 11. Assignments practice the use of statecharts with *iState*, a statechart compiler that follows that definition and allows statecharts to be animated. In the fall of 2006, we experimented with adding the invariants to statecharts, with automatic verification in iState following [4].

Lecture 12: Software Development Process

Different software development processes are mentioned, without going into detail (for time).

Interlude: Software Tools

Additionally, the topic of *Configuration Management* was included at the beginning of Software Design 2 and subversion was used from that point on for all assignment submissions. The assignments used Pascal, Java, jUnit (for testing), and iState (for statecharts); introduction to these was provided in optional tutorials.

3. EVALUATION

Except for the topics of Configuration Management and Software Development Process, all other topics used a coherent notation and coherent mathematical basis. We did not

Completeness and Consistency with Tabular Specifications

When specifying complex systems, two fundamental issues arise:

Completeness: Does the specification cover all possible cases, or did we forget some cases?

Consistency: Are there contradictions in our specification, or is is consistent?

Both issues are addressed by *tabular specifications*: besides making it easier to check specifications for completeness and consistency, they make large specifications more readable and more appealing by the two-dimensional notation.

Properties of Tabular Specifications

Let a table with header $H = (p_1, \ldots, p_n)$ be given.

	p_1	$\ldots$	p_n
$\vdots$	$\vdots$	$\vdots$	$\vdots$

- H is *disjoint* if $\neg(p_i \wedge p_j)$ for all $i \neq j$.
- H *covers* p if $p_1 \wedge \ldots \wedge p_n = p$.
- H *partitions* p if H is disjoing and H covers p.

Figure 7: Excerpts from *Functional Specifications*

Definition of Inheritance

Methods correspond to procedures that take an additional parameter for **this**; the body must not assign to **this**. Self-calls are resolved to the methods of the class itself:

class C
  var $a : A$
  method s
    S
  method t
    T
end
	$=$	**var** C : **set of** *Object*
		var a : **map** *Object* **to** A
		invariant
		  nil $\in C \ \wedge$ **dom** $a \ = C - \{$**nil**$\}$
		procedure $C.s$(**this** : C)
		  $S[s,\ t \setminus C.s,\ C.t]$
		procedure $C.t$(**this** : C)
		  $T[s,\ t \setminus C.s,\ C.t]$

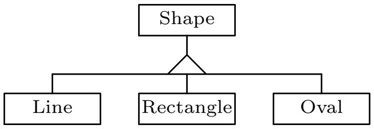
class D **inherit** C
  var $b : B$
  override method t
    T'
  method u
    U
end
	$=$	**var** D: **set of** *Object*
		var b: **map** *Object* **to** B
		invariant $D \subseteq Y \ \wedge$
		  nil $\in C \ \wedge$ **dom** $a \ = C - \{$**nil**$\}$
		procedure $C.s$(**this** : C)
		  $S[s,\ t,\ u \setminus D.s,\ D.t,\ D.u]$
		procedure $C.t$(**this** : C)
		  $T'[$**super**$.s,$ **super**$.t \setminus C.s,\ C.t]$
		    $[s,\ t,\ u \setminus D.s,\ D.t,\ D.u]$
		procedure $C.u$(**this** : C)
		  $U[s,\ t,\ u \setminus D.s,\ D.t,\ D.u]$

Forms of Inheritance

Object-oriented languages own much of their popularity due to the way in which inheritance can be used (or misused). Good uses of inheritance are:

- Inheritance for modelling
- Inheritance for specification
- Inheritance for code sharing

Inheritance for modelling is used to let the structure of the program reflect the structure of the problem: the program becomes easier to understand and to maintain. It applies when the one class is a *generalization*, or conversely a *specialization* of another class.



The essence is that each subclass must be a *refinement* of its superclass.

Figure 8: Excerpts from *Object-Oriented Programs*

observe that students are in any sense math-phobic: they are sceptical towards the use of logic in software design as much as they are sceptical toward design patterns and configuration management systems; they haven't seen the need for any of these. In a series of assignments, the use of each concept of the course is practiced. At the end of the second course, students take the use of logic for granted. The topic that caused the most difficulties was object-oriented modelling: while the concepts are mathematically easy to explain, acquiring proficiency in practice would have taken more time than was available. The topic was dropped in later editions of Software Design 2.

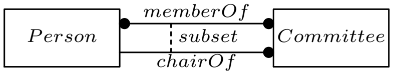
Requiring students to take a course in logic and discrete math before Software Design 1 had only a moderate effect on their ability to use logic and abstract data types for the description of problems. Our explanation is that logic and discrete math courses traditionally teach a body of knowledge, and do not practice the use for description and do not practice proving. Additionally, the difference in notation, as for implication and for quantification, prevents students from seeing the connection even if there is an obvious one (one can easily find a dozen different notations for quantification; we doubt that calculus would have the same influence if that many different notations for addition or integration would be used). Operators that are useful in software design, like relational overwrite, are not taught in discrete math courses; usually (untyped) first-order logic is taught, rarely equational proofs. A good portion of Software Design 1 is spent—or rather wasted—with introducing notation for typed logic, data types, and equational proofs. One would wish that the field would have matured by now to standardize them.

We did not observe that students with previous experience in programming did better than those without. To the contrary, students with previous experience are being more dismissive of topics like modelling, design patterns, configuration management, except if they have already been exposed to those. In questions on verification, these students generally performed poorer. One reason is that these students find it more difficult to think abstractly about programs. The other reason is purely notational: with = meaning assignment in some programming languages but equality in mathematics, they confuse these two notions and write meaningless correctness statements like $\{true\}\, x = x + 1\, \{x = x + 1\}$. They have initial difficulties distinguishing between a property of a state and a statement leading to that property. At the end of the second course the difference between students with and without previous experience disappears. Still, one would wish that programming languages would follow the tradition of mathematics in the use of =.

Students are required to complete a two-semester design project in their fourth year. Software Design 2 was consistently ranked as the most useful course in a questionnaire at the end of the project and Software Design 1 as the third most useful course. While that may sound encouraging, the projects rarely show a sufficiently systematic application of the techniques. That may be partly due to the explorative

Constraints

Constraints allow to express additional restrictions which are not captured by the diagrams on their own. Constraints can be attached to classes. Such constraints become an additional part of the invariant [example omitted]. Constraints can also be written next to associations. Dashed lines are used to connect the involved associations:

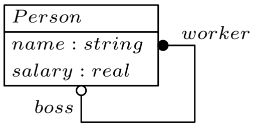


var *Person* : **set of** *Object*
  Committee : **set of** *Object*
  memberOf : **rel** *Object* **to** *Object*
  chairOf : **rel** *Object* **to** *Object*
invariant
  $\mathbf{dom}\ \mathit{memberOf} \subseteq \mathit{Person}\ \wedge$
  $\mathbf{ran}\ \mathit{memberOf} \subseteq \mathit{Committee}\ \wedge$
  $\mathbf{ran}\ \mathit{chairOf} = \mathit{Committee}\ \wedge$
  $\mathit{injective}(\mathit{chairOf})\ \wedge$
  $\mathit{chairOf} \subseteq \mathit{memberOf}$

Refining Class Diagrams

Class diagrams are a way of graphically expressing object-oriented system models. Some concepts, like classes with attributes and methods, can be readily implemented in programming languages, while others, like associations and qualification, cannot. In such cases, we have to refine our model, perhaps in several steps, until we arrive at a model that is sufficiently close to our programming language. The refined models can be expressed graphically as well.

Refining Associations by Pointers. Consider:



Suppose we decide to implement the association by adding an attribute *boss* to each person. That attribute could be **nil** or a pointer to the boss, reflecting the zero-or-one multiplicity.

Figure 9: Excerpts from *Object-Oriented Modeling*

nature of these projects. However, the author believes that this is mainly due to these concepts not being repeated and practiced elsewhere. In courses on databases, operating systems, compilers, user interfaces, networks, real-time and algorithms terms like invariants and robustness are not used, giving the impression that these notions are not universally relevant. To give evidence to this claim, we refer to the analysis of the five most popular algorithm textbooks in [11]: four books, with 550 to 770 pages, devote zero pages on correctness and one book with 790 pages devotes eight on correctness. One would wish that textbooks and instructors would acknowledge the usefulness of these notions more widely.

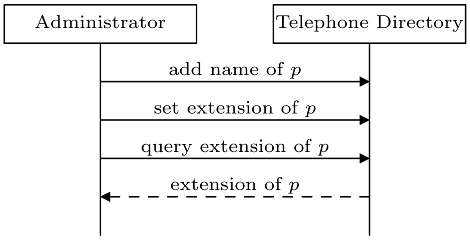
Over the years, in the course evaluations 30%–65% of the students report that 81%–100% of the course material seems valuable and 35%–50% report that 61%–80% seems valuable. The numbers were on the higher end in later years and for Software Design 2 (not all students continue with Software Design 2). The use of independent critical judgement was rated high, particularly in later years. The overall delivery of the course received mixed evaluations, because the material was not fully developed in earlier years, the material was not motivated well in earlier years, most teaching assistants were of little help to the students, and because students felt overloaded. Except in the first year, there were

Sequence Diagrams

Scenarios can be described by *sequence diagrams* (*message sequence charts*) showing vertically an interaction sequence among actors and the system over time:

- Solid horizontal arrows indicate a message (interaction).
- Dashed horizontal arrows indicate a response to a previous message.

For example, for part of the Set Extension use case followed by the Query Extension use case:



Checking Specifications Against Scenarios

Consider the scenario:

> Setting the extension of a specific person and querying the extension of that person will return the same extension again.

To check if our specification allows this scenario, we analyze the sequence of calls

$$S\ = \mathit{setExtension}(p,\ e1)\ ;\ \mathit{queryExtension}(p,\ e2,\ \mathit{found})$$

by determining its weakest precondition with respect to postcondition $\mathit{found}\ \wedge (e1\ = e2)$:

$$\mathit{wp}(S,\ \mathit{found}\ \wedge (e1\ = e2))$$

In general, the goal is to check:

$$\mathit{inv}\ \wedge \mathit{pre} \Rightarrow \mathit{wp}(\mathit{scenario},\ \mathit{post})$$

Note that if every procedure called by the scenario preserved *inv*, we also have:

$$\mathit{inv} \Rightarrow \mathit{wlp}(\mathit{scenario},\ \mathit{post})$$
$$\mathit{inv}\ \wedge \mathit{pre} \Rightarrow \mathit{wp}(\mathit{scenario},\ \mathit{post}\ \wedge \mathit{inv})$$

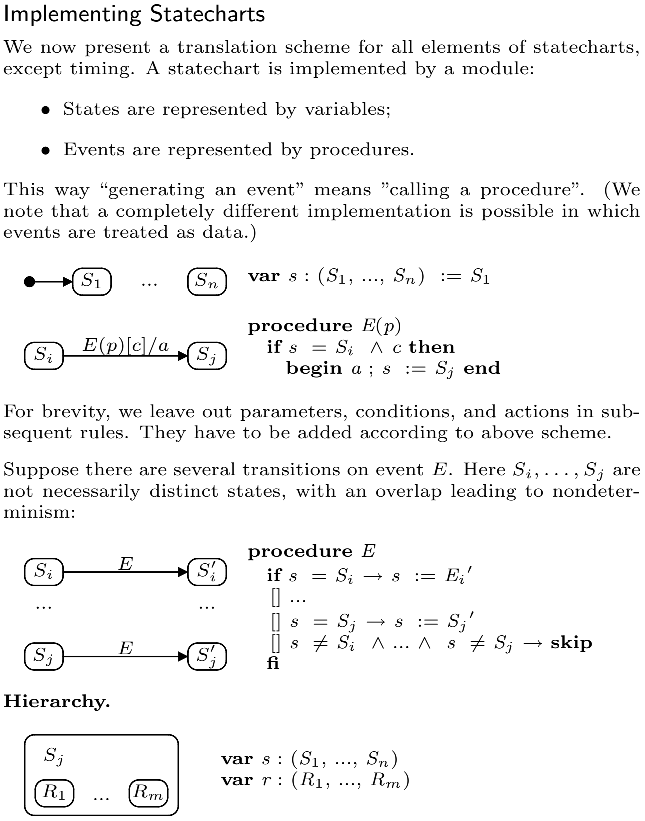
Figure 10: Excerpts from *Requirements Analysis*

no complaints that the contents is overly mathematical.

4. DISCUSSION

We believe that we have successfully used mathematics as the unifying force for elements of software design into a two-semester courses in software design. The material is covered in 710 pages of lectures notes by the author plus a couple of original articles and book chapters (a course pack with the material is printed for students on demand). The mixture of mathematical and less mathematical topics gives students confidence that the use of mathematics is justified. We could not have done this with a single one-semester course.

If teaching the mathematics of software design is to be useful, it has to be taught as early as possible, before students acquire “bad habits,” a point that has been repeatedly made, e.g. [7]; we wish we could have started even earlier in the curriculum. We have deliberately not used a specific formal tool; we find those more appropriate for upper level courses. Gordon [3] also offers a two-semester course, also



Implementing Statecharts

We now present a translation scheme for all elements of statecharts, except timing. A statechart is implemented by a module:

- States are represented by variables;
- Events are represented by procedures.

This way "generating an event" means "calling a procedure". (We note that a completely different implementation is possible in which events are treated as data.)

```
var $s$ : ($S_1$, ..., $S_n$) := $S_1$
```

```
procedure $E(p)$
  if $s = S_i \wedge c$ then
    begin $a$ ; $s := S_j$ end
```

For brevity, we leave out parameters, conditions, and actions in subsequent rules. They have to be added according to above scheme.

Suppose there are several transitions on event E. Here $S_i, \dots, S_j$ are not necessarily distinct states, with an overlap leading to nondeterminism:

```
procedure $E$
  if $s = S_i \rightarrow s := E_i'$
  [] ...
  [] $s = S_j \rightarrow s := S_j'$
  [] $s \neq S_i \wedge ... \wedge s \neq S_j \rightarrow$ skip
  fi
```

Hierarchy.

```
var $s$ : ($S_1$, ..., $S_n$)
var $r$ : ($R_1$, ..., $R_m$)
```

Figure 11: Excerpts from *Reactive Program*

using higher order logic as a unifying framework, but covers the logical aspects in more detail, and includes hardware verification. We have not tried to use any "light" method that makes formal techniques "invisible"; in our experience students appreciate being taught the theory in an isolated, minimal way, before seeing it applied with constraints.

Compared to the inverted curriculum, or outside-in order by Pedroni and Meyer [8], we do not teach class design before control structures, but we do teach programming before requirements analysis. Our way of introducing formal techniques is less gentle than theirs, but our reason is the same as for their outside-in order: to put all students on the same level and keep them motivated. We have succeeded with the first one, even if it comes by shocking the students in the first classes with mathematics, for which they were not prepared; we were less successful in keeping them motivated during Software Design 1.

While we believe that the courses influenced the way how students think about programs, the main obstacle for having a profound influence on their practice of programming is that concepts are not being repeated and practiced in other courses. We agree with Dijkstra's observation on computing science [2]:

> ... providing symbolic calculation as an alternative to human reasoning ... is sometimes met with opposition from all sorts of directions: ... 6. the educational business that feels that if it has to teach formal mathematics to CS students, it may as well close its schools.

If anything, with low enrollment numbers the pressure to eliminate mathematics has increased.

5. ACKNOWLEDGEMENTS

The author would like to thank David Parnas, Michael Soltys, and the two reviewers for their careful reading and thoughtful suggestions.

6. REFERENCES

[1] E. W. Dijkstra. Notes on structured programming. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*. Academic Press, 1972.

[2] E. W. Dijkstra. On the cruelty of really teaching computing science. *Communications of the ACM*, 32(12):1398–1404, 1989.

[3] M. Gordon. *Specification and Verification, Parts I and II*. http://www.cl.cam.ac.uk/~mjcg/, 2006.

[4] D. T. M. Le, E. Sekerinski, and S. West. Statechart verification with iState. In M. Chechik, editor, *Formal Methods 2006—Posters and Research Tools*, Hamilton, Ontario, 2006. http://fm06.mcmaster.ca/istate.pdf.

[5] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[6] D. L. Parnas. Tabular representation of relations. CRL Report 260, McMaster University, October 1992.

[7] D. L. Parnas. "Formal Methods" technology transfer will fail. *Journal of Systems and Software*, 40(3):195–198, 1998.

[8] M. Pedroni and B. Meyer. The inverted curriculum in practice. In *SIGCSE Technical Symposium on Computer Science Education*, Houston, Texas, USA, 2006. ACM Press.

[9] E. Sekerinski and R. Zurob. iState: A statechart translator. In M. Gogolla and C. Kobryn, editors, *UML 2001 – The Unified Modeling Language, 4th International Conference*, Lecture Notes in Computer Science 2185, Toronto, Canada, 2001. Springer-Verlag.

[10] E. Sekerinski and R. Zurob. Translating statecharts to B. In M. Butler, L. Petre, and K. Sere, editors, *Third International Conference on Integrated Formal Methods*, Lecture Notes in Computer Science 2335, Turku, Finland, 2002. Springer-Verlag.

[11] A. B. Tucker, C. F. Kelemen, and K. B. Bruce. Our curriculum has become math-phobic! In *SIGCSE Technical Symposium on Computer Science Education*, Charlotte, North Carolina, USA, 2001. ACM Press.