CIS300 and 200 and the core

David Schmidt

Kansas State University

`www.cis.ksu.edu/~schmidt`

Some history

My history regarding CIS200

I rebuilt CIS200 and taught it from 1997 to 2000 (with Bill Shea's help in 1998–2000).

Constraints (in 1996) on redoing CIS200:

- use a language that supports object orientation
- computer engineers wanted C experience
- some subsequent CIS courses required C++ and C
- beginners need to be protected from dangerous language features (e.g., address arithmetic, buffer overruns)

I made *Java (Version 1.02!) the compromise choice*

How it went...

- teaching "objects first" was difficult — *too much propaganda* ("all the world are objects"), *too much Java overhead*, too much advanced material: o-o, GUI frameworks, events, interfaces

- a serious problem developed: *programming by oracle (IDE)* — the student cuts and pastes some code into the IDE, runs a test or two from the assignment sheet, and watches what happens (like the monkey typing Hamlet). ***Students fail to develop a semantic model of execution*, one they can *draw with pencil and paper* and use to *generate an execution trace*.**

  I tried to sell them a model, but *the object model of execution overwhelmed them* — classes, object instances generated from classes, method activations, local variables, intra-method control flow, inter-method control flow, `this`, `super`, etc.!)-:

- I wrote a 750+-page text — complete with machine model, operational semantics, programming logic — that was/is contracted to a publisher; after about 3-5 *complete* rewrites and numerous disagreements with the editor and his moving-target reviewers, the text rests in electronic-only form at `www.cis.ksu.edu/~schmidt/PPJ`.

  The text is used by a variety of people around the planet, and I learned a lot from writing it (but I won't tell you exactly what!).

CIS300: Algorithms and Data Structures

I've taught this from 2000 to today.
(`www.cis.ksu.edu/~schmidt/300s03`)

What I expect of incoming students: not much

Gurdip's advice to me in 2000 was on target: "just hope that they can program a bit with arrays!"

Students are confused by components, packages, and (especially) interfaces; they have trouble writing loops; many employ cut-paste-and-test methodology.

Topics I cover in CIS300:

TOPIC (Weeks 1-8)	*APPLICATION*
Review of software design methodology; What is a data structure?	
The array as a data structure; Sorting, searching, and time complexity	implement/modify an array-based database to sort and binary-search its contents
Stacks and their array and linked-list implementations; processing patterns that use stacks	travelling-salesman problem
storage layout in the JVM: activation-record stack and heap; semantics of object creation, method invocation, parameter passing	
Queues and their linked-list implementation; processing patterns	moving averages, breadth-first search, simple simulation
Linked list variants, e.g., doubly-linked	implement set or bag specification via linked lists

TOPIC (Weeks 9-16)	*APPLICATION*
Flat (iterative) vs. Layered (recursive) data types; inductive definition and processing patterns; immutable and mutable data types	ConsLists, Trees, Folder systems, other dynamic structures
Binary trees and their variants: ordered (search) trees, n-ary trees, spelling trees, AVL trees	database implemented by an ordered tree; dictionary implemented by spelling tree
Hash tables	rework an earlier project with a hash table
priority queues implemented by heaps	implement a priority queue
Graphs (if time allows)	

Course text: *none* — they tend to be too Java- and Java-library specific. (And in the past, the students didn't read the text!) I write my own notes, which the students download (see `www.cis.ksu.edu/~schmidt/300s03/Lectures` for the last complete set)

Java influence in what I do: I try to make it almost none, although I use Java interfaces in several exercises, and I use some simple Java-coded GUIs in one or two assignments as a review of CIS200 (e.g., "modify the event handler in this GUI to do ...").

Why I don't use `java.util`: Mostly, its method suites are too complex and a bit nonstandard (e.g., its specification of "Set" has about 25 methods yet lacks union and intersection operations; its specification of "List" lacks head and tail operations).

Alas, by de-emphasizing "collections," I miss the opportunity to promote *iterators*. (But a Java-coded iterator is a bit ugly.)-:

Some assessment

CIS200 is too Java-dependent and “heavy”

- It’s wrong to introduce *subclassing, event handling, multi-threading, and interfaces* (all required to use Java GUIs) as well as *exception handling* (for sequential-file processing) in a first course!

- Even objects-as-instances-of-templates (classes) is heavy. (*Why can’t we limit ourselves to* `components = modules` *in CIS200!?*)

- *Computer hardware, virtual-machine organization, and operational semantics are not drilled into the students.* As a consequence, the student’s survival semantics is cut-it-paste-it-and-ask-the-IDE.

CIS300 is better defined but flawed:

- I am constantly torn between teaching “CS 1.5” (give the students what they missed/forgot in CS1) and CS2.

- Java’s clumsy interfaces, abstract classes, subclasses, *and* the requisite down-castings — Castor oil! — distract my students and make me wish I could use *Scheme* or a language with type inference.

- There is tension between teaching *processing (control) patterns* e.g., structural recursion on trees, and *design (component) patterns*, e.g., the visitor pattern for trees.

- I would like to emphasize class invariants and pre-post-conditions, but the students don’t have the background.

- And what about support tools (**debuggers, IDEs**)?

Some speculation

We live in interesting times...

Due to fierce price competition, tech companies increase lay offs [*"Job Cuts in Tech Sector Soar"*, Reuters 18 Oct.], moving towards a "worker free" workplace, where automation and contracters carry the load. *This is a paradigm shift.*

As a double penalty, our field suffers from an image problem:

`Computer_scientist = programmer = Dilbert`

and everyone knows you don't need a computing degree to be a programmer....)-:

Nonetheless, the demand for computing expertise will increase.

We must look towards training these *computing generalists*:

$$\neg \mathit{computingSpecialist} \equiv \mathit{intelligentUser} \land \mathit{lightweightProgrammer}$$

and look towards supporting multi-disciplinary programs.

We must design a *computing core* that tells one **what you need to know to be a computing generalist**.

Computer Science "core" courses at

- ***Cornell:*** intensive programming; data structures and functional programming; architecture; numerical computation
- ***CMU***: intensive programming; data structures; programming paradigms and formal methods; architecture; algorithms
- ***MIT:*** structure and interpretation of computer programs (Abelson and Sussman); circuit theory I and II; system and signal theory
- ***Berkeley***: structure and interpretation of computer programs; data structures; machine structures (architecture and OS)
- ***Stanford:*** intensive programming; programming paradigms (LISP, OO, assembly); automata and computational complexity; artificial intelligence

What is the "core" (foundation) of computing?

1. ***models for computation*** (architecture, computability)
2. ***algorithms*** (programming the models: control-, data- and component-structures)
3. ***meta-programming*** (programs that (help) do programming: translators, operating systems, analyzers, support environments)

One bold proposal would be a CS:2-1-3 sequence. Another would be to teach the first 3 CIS courses from Abelson and Sussman's *Structure and Interpretation of Computer Programs* (MIT Press, 1985), which covers most of the intellectual import in computing (programming, induction, data structures, modules, interpreters, theorem provers, hardware simulators, translators). But maybe this works only for the very best CS departments.)-:

A presentation that matches our curriculum

1. **CIS200:** hardware/software systems ***control structure***
2. **CIS300:** hardware/software systems ***data structure***
3. **CIS501:** hardware/software ***component structure***
4. **CIS301:** hardware/software systems ***logical structure***

Rationale for the course titles: Each programming paradigm (procedural, o-o, event-driven, logical, functional) and/or language (Prolog, CAML, C#, Pascal) and/or methodology (stepwise refinement, object-orientation, Jackson methodology) has well-defined notions of *control* (evaluation ordering), *data structure* (aggregates), *component structure* (modular organization), and correctness *logic*.

These same notions are embedded in architecture and support software, and indeed, “software” was introduced to eliminate need of rewiring hardware.

CIS200: Control structure

***Theme:* "structure and interpretation of computer programs" for computing generalists — how to *command* coherently**

1. *computer-hardware architecture, networks, and internet*: what happens where (execution ordering, what controls what); single-computer machine model
2. *support tools* (operating system, web browser, text-editor, high-level-language translators, frameworks (e.g., Visual Basic)): what happens where (execution ordering, what controls what in the machine model)
3. *programming in "script"* (straight-line coding/control): expression and command evaluation in the machine model; inserting script into a program generator
4. *classical control structures*: sequencing, conditional, repetition; operational semantics on machine model; standard algorithms for numbers and strings
5. *methods*-as-functions, and modules-as-method-suites; 2-3 component systems
6. *introduction to aggregates*: arrays, sequential files, and their standard algorithms
7. *introduction to components*: local state in modules; methods as state-mutators

Practical impact: The course emphasizes *control* — procedural programming — and is somewhat language neutral, although a carefully chosen subset of C# is an OK choice.

No more programming Java-GUIs and animations — they are distracting — use a GUI-generator (a la *Visual ..X..*)

Secondary issues: BlueJ is a clunky IDE; the students grumble. Better to introduce Eclipse, which is freely available, "cool," and can be applied in subsequent courses.

Java should be retired: its core and libraries have ballooned beyond control.

Why C# ? built-in iterator, `foreach`; built-in get,set templates for classes; reintroduces record structures to good advantage; points towards C and C++; simpler event-handling methodology; compatible with .NET and Mono (open-source version for Linux: `www.mono-project.com`) which is better organized than Java libraries.

CIS300: Data structure

Theme: **Presentation of meta-programming tools (operating system, compiler, database) with the data structures they utilize.**

1. the role of *data structure in hardware* architecture
2. data-structures as aggregates; data structures as object instances of class definitions; class-representation invariants
3. sorting, searching of arrays; *time-complexity hierarchy*
4. *stacks: virtual-machine architecture* (activation-record stack); stack-based syntactic recognition in compilers/text editors
5. *queues: application to OS*; intro to multithreading, mutual exclusion, and synchronization
6. *trees: application to compilation*: parsing and translation; use of XML for tree representation
7. *hash tables, heaps*, mumble, ...

CIS501: Component structure

Theme: ***connecting*** **components — control issues (multi-threading, synchronization), data issues (communication mechanisms, mutual exclusion), case studies**

1. distinctions: modules vs. classes vs. objects; types vs. interfaces vs. specifications; components vs. connectors
2. components and *connectors in hardware architecture and in meta-programs* (operating system, compiler)
3. *"design theory"*: design patterns, UML-like notations, connector- and architecture-description languages
4. heterogeneous systems, *linking languages*, interface description languages; *case studies*: client-server architecture, XML application, Corba
5. multithreaded systems; *patterns for mutual exclusion*
6. *software-architecture paradigms*: layered, database, blackboard, etc.; analyses of architectures of IDEs, debuggers, (graphics) frameworks, etc.
7. software-architecture views, UML diagrams, aspects

CIS301: Logical structure

Theme: **logic for computing is dynamic logic**

1. *Hoare logic* of while + (recursive) procedures
2. data-representation *(class) invariants*; design and program by contract
3. soundness of assertions/reasoning with respect to machine model
4. *design-by-contract project* completed with manual and tool-assisted verifications
5. proof and model theory of *first-order logic*, introduced on a demand-driven basis.
6. finite-state automata, regular languages; context-free grammars
7. hardware architecture as Turing machine; unsolvable problems

Comments

The core produces an intelligent *computing generalist* who can *use*, *connect* together, and to some degree, *program* systems; can understand the hardware/software *foundations*; and can *specialize* to project manager, systems administrator, bio-informatician, graphics designer, etc.

The self-contained core gives *flexibility* to the CS curriculum, allowing *"modular" degree programs*, and should improve the *recruiting of non- and dual-CS majors*.

Bill (Hankley) suggested we might create a *track of CIS300/501* tailored towards *software-engineering specialization* (is this an IS core?). This sounds fine, but extra courses mean extra teaching, and indeed, any revisions at all will generate lots of extra work.

Throughout all this, we don't lower academic standards.