Research-based Design of the First Weeks of CS1

Conference Paper · November 2014

DOI: 10.1145/2674683.2674690

2 authors:

Juha Sorva
Aalto University
32 PUBLICATIONS **409** CITATIONS

SEE PROFILE

Otto Seppälä
Aalto University
19 PUBLICATIONS **740** CITATIONS

SEE PROFILE

Research-Based Design of the First Weeks of CS1

Juha Sorva
Aalto University, Finland
juha.sorva@aalto.fi

Otto Seppälä
Aalto University, Finland
otto.seppala@aalto.fi

ABSTRACT

On the basis of cognitive load theory, theoretical models of instructional design, and empirical findings from computing education research, we propose three independent but compatible and complementary frameworks that can be used in introductory programming education. *Motivate–isolate–practice–integrate* is a framework that marries project-driven learning to careful management of cognitive load through the selection of learning activities and the isolation of partial tasks. *Head Straight for Objects* is an outline of an introduction to programming that emphasizes object-orientation early while mediating the cognitive load intrinsic to object-oriented concepts. Finally, the *principle of explicit program dynamics* states that the runtime dynamics of programs should be a continuous and explicit theme in introductory programming education. We illustrate the application of the three frameworks in the context of a university course.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms

Human factors

Keywords

CS1, instructional design, cognitive load, project-driven learning, objects early, notional machines, program visualization

1. INTRODUCTION

In this article, we explore the applicability of theoretical models of instructional design and research findings from computing education research (CER) to the instructional design of an introductory programming course at the university level (a *CS1* course). The goal of this effort is to enhance the effectiveness of learning. More specifically, we look to address known challenges in CS1 instruction, which include the cognitive complexity of program-authoring in general and object-oriented programming (OOP) in particular as well as students' difficulties with program dynamics. We present three research-based frameworks that underlie the design of the first few weeks of our CS1 and illustrate how they can be used.

Section 2, below, reviews the earlier research that we build on. Section 3 gives an overview of our CS1 course and its context, which motivate the present work. Each of the three sections that follow introduces a guiding framework or principle that we have used in the design of our course and illustrates it with examples. Section 4 describes the *Motivate–isolate–practice–integrate* framework, whose main purpose is to manage cognitive load while introducing multiple topics through a motivating project. Section 5 introduces the *Head Straight for Objects* framework, in which OOP is introduced early after a brief preparatory phase that is designed to ease the learning of OOP. The topic of Section 6 is the principle of *explicit program dynamics*, which seeks to promote programming skill by highlighting the execution model of programs via means such as vocabulary and visualization. Finally, in Section 7, we discuss the nature of our contribution and opportunities for future work.

Koli Calling '14 November 20 – 23 2014, Koli, Finland
 ACM 978-1-4503-3065-7/14/11...$15.00
http://dx.doi.org/10.1145/2674683.2674690.

2. BACKGROUND

In this section, we review pertinent studies of the difficulties of beginner programmers, cognitive load theory and theoretical models of instructional design.

2.1 Difficulties of beginner programmers

Writing computer programs imposes high cognitive demands on novices, as it involves the synthesis of many programming concepts, the use of unfamiliar tools and notations, and the application of new problem-solving patterns. This challenge is evidenced in the various studies within CER that document beginners' difficulties in learning to program. Students' skills for independent code-authoring have been found to fall short of their teachers' expectations at a wide range of institutions, which has led to the lowering of expectations and heavier use of scaffolding that supports beginners in code-writing tasks [17, 33]. Misconceptions about concepts and code constructs plague many learners [28]. Introductory programming features core concepts that are tightly interconnected so that failing to master just one early concept may snowball into an avalanche of difficulties [24].

Object-oriented programming as embodied in languages

such as Java has become mainstream in CS1, although non-OOP CS1s are also common. Some educators in computing view OOP as merely an extension of imperative programming while others view it additionally as a distinct paradigm [4]. Researchers have pointed out that OOP requires more concepts and a more complex model of execution than procedural programming, which makes OOP harder and may cognitively overload beginners [25]. According to one estimate, "writing even the smallest meaningful class requires approximately 15 basic concepts: object, class, instantiation, constructor, method declaration, formal parameter, return type, variable, type of variable, value, integer, assignment, method invocation, actual parameter, returning a value" [5]. Many of the misconceptions reported in the CER literature are specific to OOP. The positive flip side of this challenge is that when learning is successful, students master more concepts and a paradigm that is in demand.

Research shows that a major source of many students' struggles and misconceptions is the dynamic but often invisible nature of computer programs at execution time. In other words, students fail to form a viable mental model of a *notional machine*, that is, an abstract model of execution that explains the runtime behavior of programs and can be used to reason about program behavior [28]. Indeed, students are often disinclined or unable to systematically trace or debug programs, and they may rely on guesswork as they try to determine what programs do [16, 28]. Program dynamics may constitute an integrative and transformative *threshold* within introductory programming, in the sense defined by Meyer and Land's theory of threshold concepts [20, 28].

2.2 Cognitive load theory

Human working memory can only hold a handful of items at one time. With this as its point of departure, *cognitive load theory* is an empirically supported framework that can be used to analyze the cognitive demands that learning tasks place on working memory and the effects of those demands on the effectiveness of learning [22].

Two types of cognitive load are intrinsic and extraneous load. *Intrinsic load* is the necessary cognitive load imposed by a learning task. It reflects the degree of interconnectivity between key elements present in the task; connected elements need to be held in working memory simultaneously in order to process them and learn effectively. Prior knowledge of the subject being learned enables the learner to abstract larger wholes into single "chunks", which reduces intrinsic load, and automation through prolonged practice may effectively eliminate the cognitive load inherent in an element. *Extraneous load* is the unnecessary cognitive load brought about by ineffective pedagogy, distractions, and the like. It is harmful when intrinsic load is high, as is typical when beginners learn complex subjects.

Drawing on empirical studies, cognitive load theorists have produced many pedagogical recommendations [22]. One of the most prominent findings is that beginners cannot effectively manage the high cognitive load of problem-solving tasks, so *only* solving problems (e.g., authoring original programs) is not a very effective form of learning for beginners. Instead, beginners should engage in lower-load practice before transitioning to problem solving. Examples of such activities include *worked examples*, which provide the entire solution to a problem for the learner to study, and *completion problems*, which provide partial solutions for learners to fill in. Ideally, each learning task would match the learner's present level of knowledge.

Within programming education, worked examples are common, but there is relatively little empirical research about them [26]. Completion problems (of the form studied by cognitive load scholars) were conceived and originally evaluated in the context of programming; there is some evidence of their effectiveness [35]. *Parsons problems* [7], in which students piece together small (sub)programs from given fragments, are an example of a computing-specific assignment which has been put forward as a low-load task that can enhance beginners' code-writing skill effectively.

Given a level of prior knowledge, the only way to reduce intrinsic cognitive load is to temporarily compromise learning goals and work on parts of them in isolation. According to the *isolated elements effect* identified by cognitive load researchers, such an approach promotes efficient learning of material that has very high element interactivity [22], as introductory programming does. Once learners have knowledge of parts of the material, and possibly some automation of that knowledge, the parts can be integrated through whole-task practice whose intrinsic load is now lower. An agenda akin to this has been pursued in a programming context by Mead et al., who explored the ways in which certain concepts may carry part of the cognitive load for others [18].

2.3 Models of instructional design

Many models of instructional design have been proposed on the basis of a variety of theories of learning. In recent times, instructional approaches have often been grouped into *direct instruction* and *constructivist instruction*, both labels whose definitions and relative merits are being debated [32]. Despite the often polemic arguments, the sides have produced many similar recommendations. Some of this common ground was charted by Merrill [19], who reviewed leading theories of instructional design to synthesize five "first principles of learning" that are very widely agreed upon by educationalists of various persuasions. In paraphrase, the principles are:

Problem-centered Use real-world problems. Show learners in advance what they will be able to do on their own.

Activation Draw on prior experiences. Provide new experiences on which to found further knowledge. Provide explicit organizing themes to curricula.

Demonstration Show examples of solutions. Demonstrate procedures. Visualize processes. Use multiple representations and compare them.

Application Have learners practice problem-solving. Align practice with goals. Provide guidance which fades with growing ability. Use a variety of learning activities.

Integration Encourage learners to connect their skills to everyday life by having them demonstrate the skills publicly, reflect on them, or create new uses for them.

Models of instructional design differ in how much emphasis they lend to each of Merrill's principles. Constructivist approaches often stress integration more than demonstration, for example.

The amount of scaffolding provided to students is especially high in direct approaches, some of which are based on cognitive load theory. One model of direct instruction, *4C/ID* [34], for instance, interleaves problem-solving tasks

with activities such as worked examples, completion problems, and even occasional "drills" on subtasks for which a high degree of automaticity is needed. Nevertheless, 4C/ID has next to nothing in common with the lecture-and-quiz caricature of direct instruction, as it has at its core whole-task practice on realistic problems.

Another difference is the degree of control that learners have over learning objectives, which is emphasized in various constructivist approaches and less prominent in most direct approaches. For example, in the constructivist pedagogies of *problem-* and *inquiry-based learning*, students investigate ill-defined problems or phenomena and have control over which skills they choose to learn in order to tackle them. *Project-based learning* [31] is similar; it particularly emphasizes large, motivating projects that students create over a length of time and that define the curriculum. Constructivist approaches also relatively often involve groupwork.

Practices such as *guided discovery*, in which learner empowerment is high but heavy scaffolding is available in the form of guidance from a human tutor, are widely acknowledged as potentially useful and are used in various constructivist approaches. Proponents of direct instruction have argued, however, that while guided discovery can certainly be useful, it is best suited to situations where time-effectiveness is not a driving concern [34].

A detailed review of the principles of instructional design that have been used in CS1 courses is beyond the scope of this article, but we list here a few illustrative examples from the CER literature. Caspersen [5] presents several principles for instructional design in CS1, which draw in part on cognitive load theory. These include the use of worked examples, gradual progression to complex tasks, and the *consume-before-produce* principle that recommends students use given instances of a construct (e.g., classes) before they implement their own. Kölling and Barnes [14] describe a pedagogy in which students first see a worked example, then work on similar problems before inventing open-ended extensions to given programs. Barg et al. [2] discuss a CS1 built around problem-based learning: groups of students are given a problem statement and discover for themselves, under guidance, which programming skills they need to learn to solve the problem. Another guided discovery approach is that of Repenning and his colleagues [23]: learners (schoolchildren) design computer games and tutors help them discover computing principles "just in time" as they are needed for the project.

3. OUR TEACHING CONTEXT

In this section, we describe our introductory programming course, one of the two alternative CS1s offered by Aalto University, a Finnish research university. This course, named *Programming 1*, or *Prog1* for short, is only being offered for the second time in Fall 2014; however, we have continuously developed it and its predecessors, in small steps and occasional leaps, over a period of more than 10 years.

Prog1 is taken annually by over 250 students. The typical student is just out of high school and has had good to excellent success in their studies at the K-12 level. Roughly a third major in computing, but most students come from other majors, primarily industrial management, technical physics, and information networks; the course is obligatory in all these degrees. Lengthy prior programming experience is uncommon, but a large minority have some programming experience. According to surveys, a large majority of the students start Prog1 with good to excellent motivation to learn to program and/or to do well in the course.

The primary goals of the course are to help students come to appreciate and enjoy programming and to teach them to read, write, and modify small applications written in the style of *imperative OOP* (IOOP)[1]. The students are expected to learn to apply various specific concepts[2] typical to CS1, such as classes, selection, iteration, and inheritance. In addition, Prog1 aims to lay foundations for the subsequent learning of functional OOP and functional programming more generally: there is a running theme of mutability versus immutability, and function literals and higher-order functions feature prominently in the second half of Prog1. We use the multiparadigm programming language Scala.

A significant feature of Prog1 is that students write nearly all their code to detailed specifications of interfaces rather than designing classes and methods from loose problem statements. Prog1 is part of a broader programming curriculum — our computing majors, for instance, take four programming courses in their freshman year — and modeling ill-specified problems is covered in other courses.

The pedagogy of Prog1 is built on the axiom that beginner programmers need a lot of *deliberate practice* [9]. That is, they need to do a lot of work. Our job as teachers of this course is to motivate them to put in the hours and to design learning activities to make those hours as effective as possible, so that the students make substantial learning gains towards specific objectives within the scope of the 5 ECTS credits afforded to us.

At the heart of Prog1 are its bespoke online learning materials (an eBook or "newk", if you will). These materials fuse together programming assignments, interactive animations, questions (e.g., MCQs), occasional video, and other learning activities; we do not use a textbook in the traditional sense. Below is a list of content that is covered in the first quarter of the course, which is its most germane part for present purposes.[3]

1. Expressions, values, types, variables
2. References and collections (mutable buffers)
3. Functions
4. Objects
5. Classes
6. Booleans, selection
7. Composite classes
8. Option types
9. Logical operations
10. Additional integrative practice on the above

[1]By imperative OOP, we mean OOP with mutable state as opposed to *pure functional OOP* without. We emphatically do not mean "procedural programming with some objects".
[2]We use words such as "concepts", "topics" and "knowledge" as shorthand in this article. The goal of Prog1 is not for students to learn the conceptual content *per se* but to develop skills for writing good programs that feature this content.
[3]The list only presents the content in terms of language features; longitudinal themes such as pragmatics and tools, abstraction, program dynamics, software quality, roles of variables, and (im)mutability are not shown here. The learning materials, which are in Finnish, are openly accessible online at `https://greengoblin.cs.hut.fi/o1_current/`. There are tentative plans for an English translation, should a suitable funding opportunity surface.

Students complete the course by working through the materials either individually or in pairs. There are weekly deadlines, but otherwise the students may work at their own pace. Members of course staff are available on campus to help, and the students are free to make use of this assistance (or not) as much as they like. There are no lectures beyond two introductory ones and a course wrapup at the end. Students are required to send reflections, questions, and/or feedback as they complete each segment of the course (usually several times per week); summaries of questions, answers and comments are published at least once per week. TAs monitor the students' progress weekly and send brief encouraging feedback via e-mail. Given that students write their code to specifications, we are able to use automatic assessment, which is fortunate both from a budgetary perspective and because it provides immediate feedback. A small number of open-ended assignments are assessed by TAs rather than the automatic assessment system.

In the following three sections, we present three frameworks that guide the design of our course. In particular, we will relate these frameworks to our learning materials and the guidance that we have worked into them. This is not to belittle the role of human tutoring in Prog1 or elsewhere, which is significant but not the focus of this article.

4. MOTIVATE, ISOLATE, PRACTICE, INTEGRATE

In this section, we describe a research-based framework that we have formulated in order to structure our teaching of Prog1 and its first weeks in particular. The framework is named after its four components: *Motivate–isolate–practice–integrate*, or *M-i-p-int*. We illustrate the use of M-i-p-int in the context of our course.

Instruction based on M-i-p-int starts with the introduction of a motivating project that is clearly out of reach of the students' present skills. Students are then guided to deliberately practice on isolated topics that are identified as necessary for the project. These topics are gradually integrated to each other and eventually brought to bear on the original project as well as other projects of similar complexity. Although the components of motivation, isolation, practice and integration are analytically separate, they interact and interleave in practice. The project does not need to span the whole course.

The primary influences of M-i-p-int are cognitive load theory (Section 2.2) and its recommendations of 1) progressing to problem-solving tasks via time-effective low-load tasks with heavy scaffolding, and 2) isolating topics temporarily to manage intrinsic load. M-i-p-int is additionally inspired by project-based learning's emphasis on using a concrete project which runs for a length of time, through which many topics are introduced, and to which the learned skills are related. (M-i-p-int describes, however, a form of direct instruction and is in many ways dissimilar to project-based learning as defined by Thomas [31].) We have sought to make M-i-p-int compatible with the general principles of effective instructional design identified by Merrill (Section 2.3).

M-i-p-int's dual emphasis on a challenging project and cognitive load management is a good match with the first weeks of CS1. This is the time when beginners should gain a concrete sense of the kinds of problems they will eventually be able to solve. On the other hand, before complete beginners can understand how even a small, quasi-realistic application program works, they will need to be introduced to many new topics that have complex relationships with each other (Section 2.1).

Figure 1: The GUI of the GoodStuff experience diary used in Prog1. A category of experiences (hotels, in this case) is displayed. The user can add and rate experiences. An almost too happy smile marks the user's favorite experience in the category.

The subsections below discuss the four components of M-i-p-int in more detail.

4.1 Motivate: Start with a "large" project

The *overarching project* that is central to the motivational component of M-i-p-int provides a theme that anchors the various subtopics that will be learned about. The overarching project should be sufficiently realistic for the students to feel like they will accomplish something relevant. In the case of the first weeks of Prog1, the overarching project is a lengthy worked example, a program whose implementation will be explained over a period of several weeks, with detours to other, smaller programs. For this purpose, we currently use a simple "experience diary application" called GoodStuff (Figure 1). While not extraordinarily nifty, this program has a number of desirable features such as the simplicity of its domain, which has an intuitive mapping to OOP using multiple classes, interacting objects, and collections — all topics that we wish to get to during the early parts of CS1. (GUIs we cover later. Some example programs, such as GoodStuff, have given GUIs.)

The overarching project may be initially introduced simply by presenting it, but its introduction may also involve some very strongly scaffolded activities that guide learners to participate in the project in a peripheral but reasonably authentic way (cf. [15]). In Prog1, students' first encounter with the GoodStuff project includes fixing a typographical "mistake" in a string literal within the given implementation (which they do not yet otherwise understand) as well as learning their way around the IDE, getting an overall sense of what programs look like to programmers, and observing the syntax for comments. This is in fact the students' first brush with program code — and one quite unlike *Hello, World!*

4.2 Isolate: Highlight necessary skills

The overarching project is complex from a beginner's point of view and requires many unfamiliar concepts and skills. For instance, to understand the domain model of the Good-

Stuff application, all the topics covered by the first quarter of Prog1 (see Section 3) are needed. Learners also need to develop multiple perspectives to this content (e.g., the source code, modeling, and object interaction perspectives; cf. [8]).

To avoid overwhelming the learner, specific topics are broken off from the overarching project. These smaller wholes are introduced with reference to the project but addressed separately. For instance, objects are introduced via modeling the domain of GoodStuff.

Spiral learning may be employed so that certain critical aspects of a concept are introduced first and others only after the introduction of other concepts. For instance, we introduce strings and collections without initially discussing their object nature, which becomes apparent only later after OOP has been introduced.

4.3 Practice: Work on individual skills

The practice component of M-i-p-int represents students working on the isolated topics, building knowledge about them through low-load practice, and reducing their future intrinsic cognitive load through this expanding knowledge. This practice targets not only the introduction of new concepts but the prevention of misconceptions, the development of a degree of automaticity in recurrent small subtasks, and a growing familiarity with tools and ways of working. It also has a motivational facet: as students are presented with cognitively manageable challenges, they gain mastery experiences and their *self-efficacy* [1] grows.

Below is a list of various types of tasks embedded into the learning materials of Prog1. Variability of practice is conducive to learning [19, 34], and these tasks are undertaken using different example programs and domains. Ultimately, practice must align with the overall learning goal. In the case of a programming course, practice must involve programming. But other activities can also help on the way towards that overall goal. Even otherwise meaningless "drills" can have a meaningful supporting role if used within a motivating context (as in 4C/ID; Section 2.3). The activities in the following list range from fairly authentic programming to decontextualized études.

Completion problems Students are given the design of a program, whose partially given implementation they must complete. In Prog1, completion problems are far more common than "implement the entire specification yourself" assignments, as they enable us to use a broader variety of interesting examples and promote code-reading. A fading amount of procedural information can be provided as guidance. Smaller projects can be completed while practicing for the overarching project. They may involve the integration of some previously isolated topics to each other.

Worked examples Students study fully given solutions.

REPL practice Students use a *REPL* (a read–eval–print loop, an interactive interpreter) as they practice on a particular topic. As discussed by Gries [11], a REPL can greatly help in avoiding the premature introduction of unnecessary concepts in the early stages of CS1. Also, when following the consume-before-produce principle (Section 2.3), such practice can often incorporate interesting functionality. For instance, as students in Prog1 practice calling functions, they use a given, tailor-made function to find out and report movie statistics from the Internet Movie Database. The implementation of some of the consumed functionality may be entirely beyond the goals of CS1.

REPL exploration Students use a REPL for free micro-scale exploration of a programming topic. They may be tasked to discover how integer division works, for instance, or a class from the standard library, or a given module of a completion problem.

Micro-level completion Students fill in the blanks in a given subprogram.

Code-reading Students trace or otherwise study given code. May involve visualizations and/or questions for the learner.

Debugging Students fix given programs which either have obvious compile/runtime errors or need to be tested first. Direct guidance may be provided on interpreting error messages, for instance.

Concept drills Students answer a battery of questions about a particular subtopic (e.g., reference semantics) on the basis of readings or REPL explorations. Feedback addresses misconceptions apparent in the answers.

Parsons problems Students construct programs by ordering jumbled code fragments [7]. (Currently planned, not implemented in Prog1.)

Reflection Students provide feedback, reflect on their progress, report problems, formulate questions. Over the course of Prog1, we require dozens of short reflections, comments, or questions from each student.

Supporting information Students read textbook-style text, study concept maps and other media, use a glossary, etc. To promote these more "passive" materials, all the other activities are embedded in them and often interweave with the text rather than merely appearing as end-of-chapter problems (cf. [14]).

4.4 Integrate: Bring skills to bear on project(s)

The fourth component of M-i-p-int involves the application of the isolated topics on the overarching project and other roughly similar projects. We highlight here three forms of useful integration.

Integration to the overarching project Once a concept has been practiced on, it is applied in the context of the overarching project. For instance, in Prog1, once the concept of selection has been initially covered, it is used to implement a particular method in the GoodStuff project. The integration may happen one or a few topics at a time (as we do in Prog1) or in one go once all the relevant topics have been covered.

Additional integrative practice In order to avoid piecemeal knowledge of isolated topics and to provide variable practice, students practice on other problems. In Prog1, once students are familiar with the entire domain model of GoodStuff, they work on several other domain models with fading scaffolding. Few if any entirely new concepts are introduced during these activities.

Creative integration Ideally, instruction includes the eventual creative application of skills to open-ended problems that relate to the learner's everyday life [19, 14]. In Prog1, we currently give few open-ended problems, due to budgetary constraints on assessment, and none in the first half of the course. Although our other programming courses compensate for this, we see creative practice as an aspect of Prog1 that calls for improvement.

5. HEAD STRAIGHT FOR OBJECTS

This section describes a framework that can be used to organize the topics taught near the beginning of an object-oriented CS1. We call this framework *Head Straight for Objects* (HSfO) and use it in Prog1. In the context of M-i-p-int, HSfO may be viewed as an outline of how one may go about introducing the individual topics needed for an overarching OOP project that spans the first weeks of CS1. It is also possible to use HSfO to structure the beginning of an object-oriented CS1 independently of M-i-p-int.

5.1 Motivating factors

There are two main motivating factors behind the HSfO approach. The first is our overall goal, which is to teach OOP. Since this is our primary CS1 paradigm, we wish to introduce students to it early, the sooner the better. To avoid a difficult paradigm shift in CS1, we want to avoid procedural decomposition; the first solutions to substantially sized projects that students see should be object-oriented. We wish to emphasize the modeling aspect of OOP as well as the collaborative interaction of objects at a high level of abstraction, and to orient students towards these perspectives as soon as we introduce OOP. The modeling and interaction aspects are best appreciated in the context of a "larger" project such as GoodStuff.

The second factor is that OOP involves many potentially difficult and interdependent concepts (Section 2.1). Students' cognitive load is potentially very high, and the effectiveness of their learning may be compromised, if all these concepts are introduced at once; there is also the risk of difficulties escalating due to a shaky understanding of the basics. We would prefer to mediate this cognitive load.

HSfO walks the tightrope between these two competing factors. It consists of two phases, *preparing for objects* and *introducing objects*.

5.2 Phase 1: Preparing for objects

Various concepts which are specific to OOP (e.g., classes) build on various other concepts which are not and which can be introduced in a simpler way independently of OOP (e.g., variables, parameters, state). The goal of the preparing for objects phase is to introduce non-OOP topics that have been *selected for the specific purpose of facilitating the learning of OOP immediately after*. More specifically, this phase aims to:

- introduce topics that are (also) present in OOP but without yet explaining them in OOP terms,
- draw attention to such critical aspects of these topics that are useful for learning OOP,
- provide practice with the topics so as to carry some of the cognitive load of introductory OOP, and
- address and prevent misconceptions related to the topics.

HSfO does not prescribe a particular pedagogy or ordering for specific concepts or even specify which concepts should be covered. Such choices are influenced by the programming language used, learning goals, and other factors. In Prog1, the preparing for objects phase covers expressions, values, types, variables, references, functions, and state — that is, items 1 to 3 in the topic list from Section 3.

During the preparing for objects phase, it is recommended to avoid content that is not helpful towards learning OOP. In Prog1, we initially steer clear of topics such as Booleans, selection, iteration, procedural decomposition, bit-level concepts, application entry, and I/O beyond `println` (cf. [13]). No algorithms of substantial complexity are implemented before objects are brought in (although students may *consume* such implementations, as discussed in Section 4.3 above). We also limit our coverage of each concept to its non-OOP aspects at this preparatory stage. For instance, strings and numbers, although represented as objects in Scala, are initially introduced in a way that does not reveal this aspect. The introduction of functions follows a similar rule of thumb; although Scala is fully object-based, it allows subprograms to be part of a package in a way that enables students to use them without being aware of OOP concepts.

During this phase, it may be useful to introduce a concept which is not in itself helpful for learning OOP fundamentals but which motivates the introduction of another concept that is. In our case, we introduce buffers (which, in Scala, are mutable collections similar to Java's `ArrayList`s), which we use to motivate the introduction of reference semantics, and functions that mutate state, which are needed for IOOP. Only a very small subset of buffer operations are covered that serve this particular purpose.

A fifteen-item list of concepts necessary for writing classes was quoted in Section 2.1. Of that list, our preparing for objects phase covers eleven concepts to useful degree, so only four OOP-specific ones (object, class, instantiation, constructor) are genuinely new to students in the second phase.

5.3 Phase 2: Introducing objects

The goals of the second part of HSfO are to introduce object-orientation, to integrate the previously learned concepts into OOP, and to provide efficient practice with OOP-specific core concepts. These goals may be reached in a number of ways. For illustration, we briefly describe the main steps in our current implementation of the introducing objects phase and consider how they build on the concepts introduced in the preparing for objects phase.

1. Big-picture introduction: 1a) Introduce the modeling aspect of OOP and the concepts of object and method. Discuss models of interconnected objects. 1b) Introduce the dynamic aspect of OOP. Discuss examples of how various objects pass messages and keep track of state in order to carry out tasks.
2. Consume singleton objects (i.e., individual objects not created via classes; see Figure 2): Relate the concept of method to the previously familiar concept of function. Students practice using various given singleton objects by calling their methods. Build on earlier concepts such as expressions, return values, etc.
3. Produce singleton objects: Students practice implementing singleton objects using a combination of previously learned techniques such as variables, assignment and formal parameters.
4. Motivate classes via the limitations of singleton objects. Introduce the concept of instantiation.
5. Consume classes: Students practice creating instances of given classes and manipulating references to them, proceeding from individual classes to composite ones. They build on prior understandings of methods, variables, references, state, etc.

```
// usage: myObject.method(123)
object myObject {
  val field1 = 10
  val field2 = 0.2
  def method(param: Int) = field1 * field2 * param
}

// usage: new MyClass(10, 0.2).method(123)
class MyClass(val field1: Int, val field2: Double) {
  def method(param: Int) = field1 * field2 * param
}
```

Figure 2: A singleton object and a class that resembles it, defined in Scala. (This example is for the reader's reference, not a part of Prog1.)

6. Produce classes: Students learn to initialize and use instance variables. This step builds on all the previous ones but introduces only the relatively challenging concept of instance initialization, as students are at this point able to implement singleton objects. (As Figure 2 shows, methods are implemented the same way for singleton objects and classes.)

After these steps, our students proceed to additional integrative practice and to topics such as selection that allow them to create richer methods. Students also gradually progress from individual to composite classes and one-to-many relationships implemented using collections.

5.4 A form of objects early

Many CS1 pedagogies are described as "objects first" or "objects early". These terms do not have unambiguous definitions [3], but the following ones will suffice here: *Objects early* refers loosely to any CS1 pedagogy which introduces objects during the early parts of CS1 and in which objects (and usually classes) have a leading role in structuring programs thereafter. *Objects first* refers to objects-early pedagogies in which students encounter objects at the very beginning.

The above definitions take the students' perspective, that is, they are descriptive of when students become aware of objects. HSfO does not introduce objects to students off the bat, and is, in these terms, an example of objects early but not of objects first.

We may also look at the matter from the perspective of instructional design. In HSfO, the selection of topics and activities for all the first stages of CS1 is guided by the goal of making the learning of OOP as effective as possible. In this sense, objects have first priority also in HSfO.

6. EXPLICIT PROGRAM DYNAMICS

As noted above in Section 2.1, students sometimes have a hard time getting past the more obvious static aspect of programs — program code — and come to reason about the dynamic runtime aspect; program dynamics may constitute a transformative threshold. A notional machine of some sort is present in any CS1 even if it is left implicit, but learners often have inadequate mental models and misconceptions of the notional machine and its components [28].

On this basis, we suggest the *principle of explicit program dynamics*: There should be a continuous and explicit emphasis on the execution-time dynamics of programs in CS1, and perhaps especially in the first few weeks.

In our teaching in Prog1, the principle of explicit program dynamics informs the practice component of M-i-p-int and both phases of HSfO. The principle is more generic, however, and may be applied in any CS1.

In itself, the principle of explicit program dynamics is agnostic as to how it should be implemented. Below, we describe our approach not prescriptively but as an example.

6.1 Statics vs. dynamics, a running theme

Here are a few features of our learning materials that can help tease apart the static and dynamic aspects of programs and particular programming concepts.

Terminology At the very beginning of the course, we discuss the word "program", which may refer to either the code or the process generated by running the code. We later make analogous points about more specific concepts such as function calls. Unlike in many CS1s, the words "static" and "dynamic" are used explicitly in discussing these aspects of programs at various points of the course.

Explicit state Mutable state and reference semantics are key to the dynamics of IOOP and part of the notional machine that we seek to introduce to students. We use a selection of early *transliminal concepts* — concepts that can lure students across a threshold to a new perspective [27] — to make state explicit and establish a need for a dynamic perspective to programs. Mutable collections and functions that mutate state, for instance, can serve this role, as their behavior cannot easily be explained in terms of program code only, or readily understood in terms of school mathematics. This motivates the introduction of new concepts and vocabulary for discussing the runtime aspects of state-based programs. Our treatment of state is very explicit: as soon as functions have been introduced, for example, a distinction is drawn between those that mutate state and those that do not, and students answer a few MCQs that prompt them to consider whether given functions affect state.[4]

"Low"-level program visualization From the preparing for objects phase of HSfO onwards, we use program visualization to introduce a notional machine that explains program dynamics in terms of concepts such as sequentiality, memory, references, and the call stack (Figure 3). This makes execution more tangible and provides a vocabulary for reasoning about it. We use such visualizations throughout Prog1 to provide a consistent platform in which new concepts can be integrated. The visualizations are used in a number of ways, as discussed in Subsection 6.2 below.

High-level program visualization In the introducing objects phase, and occasionally thereafter, we use more abstract visualizations of object interaction similar to the one shown in Figure 4. Using and contrasting different representations — as recommended by Merrill's principles — highlights the fact that there are multiple perspectives to program execution and may promote good *self-explanations* of examples [6].

[4]Scala has a style convention that distinguishes between subprograms that mutate state and ones that do not, making this distinction explicit in program code.

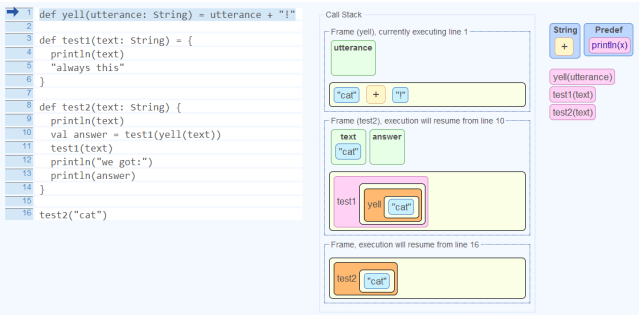

Figure 3: A snapshot of a "low"-level program animation of a toy program. Execution is paused just before concatenating the strings in the `yell` function. The stages of expression evaluation are visualized within each stack frame.

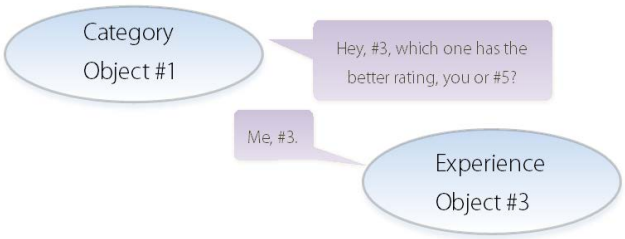

Figure 4: A fragment of a visualization of objects collaborating via messaging to provide a piece of functionality.

Singletons, classes, and instances The relationship between statically defined classes and their dynamically created instances is an area rich in beginner misconceptions. Instances are not directly visible in code as classes are, and Xinogalos [36], for instance, reports on a study in which students had more trouble with the concept of object than that of class. Singleton objects have the potential to enrich explicit instruction of the statics and dynamics of OOP concepts and to make it easier to focus on a single critical aspect at a time. In singleton objects, there is a straightforward one-to-one mapping from static code to runtime object and this can be used to introduce dynamic aspects such as object state before classes and instantiation are thrown into the mix. Singleton objects aside, another trick for exploring the dynamics of OOP is the use of structural recursion (as Caspersen, among others, has previously pointed out [5]).

6.2 Integrated and varied visualization

Program visualization is a technique that can be used to make program execution explicit [29]. Although such visualization is potentially very useful for CS1 students, visual representations of program execution are not trivial for beginners to interpret; students also sometimes fail to appreciate what it is that they are supposed to learn from the visualizations or how the visualized runtime concepts are relevant to programming practice [30]. Consequently, it makes sense to start with simple visualizations of simple programs early on and to grow the visual notation gradually as students encounter new constructs and notional machine elements. Moreover, the visualizations should not be a separate pedagogical module but tightly integrated with the overall pedagogy of the course.

In Prog1, we have embedded program visualizations into the course materials and devised a variety of activities that link the visualizations to the other content in different ways. This approach is intended to improve the perceived relevance and understandability of the visualizations and to enhance learning through the greater variability of learning tasks.

Here are some example uses for program visualization from various parts of Prog1.

Clarification of a concept Students view an animation of program execution that clarifies a recently introduced topic. The animation may be accompanied by questions that the visualization helps to answer or that guide the learner to understand the visualization. For instance, students may view the evaluation of nested function calls or the passing of a function as a parameter to a higher-order function.

Elaboration of a concept A visualization is used to guide students to draw new connections between concepts. For instance, MCQs prompt students to consider, with the aid of a visualization, whether formal parameters are local variables, and whether it is possible to have multiple distinct variables with the same name.

Clarification of a program "If you didn't understand that example program, you may view this animation of it."

Predictive tracing "Look at this program. Can you work out what it does? See the animation for the answer." Students may also be prompted for a prediction and receive feedback tailored to their prediction after viewing the visualization.

Code-writing hints "Implement method M. If you get stuck, view this animation that illustrates the execution of a working implementation but does not show the code. Try to write code that matches the visualization."

Bug analysis "Here is a program that crashes when run. Use this visualization to examine the conditions under which it crashes and answer some questions about the reasons."

Introduction of a construct Example: Students view a visualization of the internal behavior of a singleton object in which the keyword `this` is repeatedly used. They are asked to see if they can work out what the word means before proceeding to a section of the materials that discusses the matter in detail.

Motivating a forthcoming construct As above, but the construct is only introduced after the visualization. For instance, students view a visualization that illustrates how certain attributes might be associated with a class rather than its instances. This motivates the introduction of companion objects. (Companion objects are Scala constructs that fulfill a role similar to that of `static` members in Java.)

We are planning to introduce additional visualization-based activities as our browser-based visualization toolkit matures.

7. DISCUSSION

In this article, we have formulated three independent but compatible and complementary frameworks that may be used to guide the instructional design of introductory programming courses:

- Motivate–isolate–practice–integrate, a model of instruction in which topics are introduced through an overarching project but practiced separately before being integrated back to the project. The goal is to prevent cognitive overload, to address misconceptions, and to promote effective learning, all the while making deliberate practice attractive.
- Head Straight for Objects, an outline for objects-early courses that seeks to reconcile the goal of introducing object-oriented programming near the beginning of CS1 with a desire to mediate the intrinsic cognitive challenges of the paradigm and thereby to increase the effectiveness of learning OOP.
- The principle of explicit program dynamics, according to which the runtime aspect of programs should be explicitly and continuously addressed in CS1 so as to alleviate common obstacles to learning programming. Techniques towards this end may include terminology, program visualization, and the judicious introduction of language features.

These frameworks, although generic, have been developed in the context of a particular CS1 course, from which we have drawn examples of how the frameworks may be applied.

In more general terms, this article is an analytical exploration of the application of learning theory and findings from CER into the design of programming courses. The theories and models that we have applied are empirically grounded and so offer evidence-based suggestions for teaching. Moreover, having a theoretical foundation provides a vocabulary that facilitates the sharing and development of pedagogy and provides a framework for planning evaluations. Cognitive load theory, for instance, could be used in the future to analyze aspects of our design, as instruments for assessing load have been recently developed [21]. Concept inventories might also be useful in assessing learning outcomes and making comparisons to other contexts [33].

At the present time, we have no rigorous evaluation to report but can boast a high pass rate (89.1% among beginner programmers) and excellent student feedback. Since the introduction of the current course design in Prog1, feedback has reached an all-time high for our context: the average overall mark given by students to the course, out of five, was 4.31 (n=242, σ=0.76) at the end of the 2013 offering, with a low of 3.83 among students who major in industrial management (n=52, σ=0.78), and a high of 4.61 among information networks majors (n=28, σ=0.56).[5] Students commonly reported that we had made them work hard, but that they had enjoyed it and learned a lot as a result. Of course, these results are affected by numerous factors beyond the design principles presented in this article.

In closing, it is imperative to stress that our approach is designed for a particular set of goals and a particular context. As we mentioned in Section 3, our CS1 has been tailored to be part of a particular programming curriculum. Our course, Prog1, has a supportive role of providing effective education in specific fundamentals required by the other courses in the curriculum. As a result, a pedagogy based on direct instruction seems particularly appropriate. Different goals call for different pedagogies, however. A CS1 whose purpose is to teach beginners how to analyze ill-structured problem statements and design programs accordingly will need to be different from ours, as will a course that targets a paradigm other than IOOP (e.g., [10]). An introduction to computing whose main purpose is to motivate students and in which students do not operate under strict time pressure to reach specific goals may benefit from alternative approaches such as constructionism [12]. Even within the programming curriculum at Aalto University, there are other freshman-level programming courses with different kinds of goals and approaches. For example, one course that builds directly on Prog1 has looser goals, is more learner-directed, uses methods from problem-based learning as groups work on ill-structured challenges, and targets the improvement of communication skills, teamwork, and networking among peers. By using a combination of introductory-level courses based on direct instruction, such as the one we have discussed in this article, and less direct approaches, it is perhaps possible to gain the best of both worlds.

[5]Information networks is a multidisciplinary degree programme that combines computing with communications, psychology, and the social sciences.

8. REFERENCES

[1] A. Bandura. Self-efficacy. In V. S. Ramachaudran, editor, *Encyclopedia of Human Behavior*, volume 4, pages 71–81. Academic Press, New York, 1994.

[2] M. Barg, J. Kay, A. Fekete, T. Greening, O. Hollands, J. H. Kingston, and K. Crawford. Problem-based learning for foundation computer science courses. *Computer Science Education*, 10(2):109–128, 2000.

[3] J. Bennedsen and C. Schulte. What does "objects-first" mean? An international study of teachers' perceptions of objects-first. In R. Lister and Simon, editors, *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *CRPIT*, pages 21–29. Australian Computer Society, 2007.

[4] A. Berglund and R. Lister. Debating the OO debate: Where is the problem? In R. Lister and Simon, editors, *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *CRPIT*, pages 171–174. Australian Computer Society, 2007.

[5] M. E. Caspersen. *Educating Novices in the Skills of Programming*. PhD thesis, Department of Computer Science, University of Aarhus, 2007.

[6] J. L. Chiu and M. T. H. Chi. Supporting self-explanation in the classroom. In V. A. Benassi, C. E. Overson, and C. E. Hakala, editors, *Applying Science of Learning in Education: Infusing Psychological Science into the Curriculum*, pages 91–103. American Psychological Association, 2014.

[7] P. Denny, A. Luxton-Reilly, and B. Simon. Evaluating a new exam question: Parsons problems. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, pages 113–124. ACM, 2008.

[8] A. Eckerdal and M. Thuné. Novice Java programmers'

conceptions of "object" and "class", and variation theory. *SIGCSE Bulletin*, 37(3):89–93, 2005.

[9] K. A. Ericsson, R. T. Krampe, and C. Tesch-Römer. The role of deliberate practice in the acquisition of expert performance. *Psychological Review*, 100(3):363–406, 1993.

[10] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. The structure and interpretation of the computer science curriculum. *Journal of Functional Programming*, 14(4):365–378, 2004.

[11] D. Gries. A principled approach to teaching OO first. *SIGCSE Bulletin*, 40(1):31–35, 2008.

[12] M. Guzdial. Constructionism for adults (blog post). `http://computinged.wordpress.com/2014/05/23/constructionism-for-adults/`, 2014.

[13] M. Kölling. Using BlueJ to Introduce Programming. In J. Bennedsen, M. E. Caspersen, and M. Kölling, editors, *Reflections on the Teaching of Programming: Methods and Implementations*, pages 98–115. Springer, 2008.

[14] M. Kölling and D. J. Barnes. Apprentice-based learning via integrated lectures and assignments. In J. Bennedsen, M. E. Caspersen, and M. Kölling, editors, *Reflections on the Teaching of Programming: Methods and Implementations*, pages 17–29. Springer, 2008.

[15] J. Lave and E. Wenger. *Situated Learning: Legitimate Peripheral Participation*. Cambridge University Press, 1991.

[16] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas. A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bulletin*, 36(4):119–150, 2004.

[17] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. Ben-David Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin*, 33(4):125–180, 2001.

[18] J. Mead, S. Gray, J. Hamer, R. James, J. Sorva, C. St. Clair, and L. Thomas. A cognitive approach to identifying measurable milestones for programming skill acquisition. *SIGCSE Bulletin*, 38(4):182–194, 2006.

[19] M. D. Merrill. First principles of instruction. *Educational Technology Research and Development*, 50(3):43–59, 2002.

[20] J. H. F. Meyer and R. Land, editors. *Overcoming Barriers to Student Understanding: Threshold Concepts and Troublesome Knowledge*. Routledge, 2006.

[21] B. B. Morrison, B. Dorn, and M. Guzdial. Measuring cognitive load in introductory CS: Adaptation of an instrument. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 131–138, New York, NY, USA, 2014. ACM.

[22] J. L. Plass, R. Moreno, and R. Brünken, editors. *Cognitive Load Theory*. Cambridge University Press, 2010.

[23] A. Repenning. Programming goes back to school. *Communications of the ACM*, 55(5):38–40, 2012.

[24] A. Robins. Learning edge momentum: A new account of outcomes in CS1. *Computer Science Education*, 20(1):37–71, 2010.

[25] J. Sajaniemi and M. Kuittinen. From procedures to objects: A research agenda for the psychology of object-oriented programming in education. *Human Technology*, 4(1):75–91, 2008.

[26] B. Skudder and A. Luxton-Reilly. Worked examples in computer science. In J. Whalley and D. D'Souza, editors, *Proceedings of the 16th Australasian Conference on Computing Education (ACE '14)*, volume 148 of *CRPIT*, pages 59–64. Australian Computer Society, 2014.

[27] J. Sorva. Reflections on threshold concepts in computer programming and beyond. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 21–30. ACM, 2010.

[28] J. Sorva. Notional machines and introductory programming education. *ACM Transactions on Computing Education*, 13(2):1–31, July 2013.

[29] J. Sorva, V. Karavirta, and L. Malmi. A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education*, 13(4):1–64, Nov. 2013.

[30] J. Sorva, J. Lönnberg, and L. Malmi. Students' ways of experiencing visual program simulation. *Computer Science Education*, 23(3):207–238, 2013.

[31] J. W. Thomas. A review of research on project-based learning, 2000. Report prepared for the Autodesk Foundation. Retrieved from `http://www.bobpearlman.org/BestPractices/PBL_Research.pdf`.

[32] S. Tobias and T. M. Duffy. *Constructivist Instruction: Success or Failure?* Taylor & Francis, 2009.

[33] I. Utting, D. Bouvier, M. Caspersen, A. Elliott Tew, R. Frye, Y. Ben-David Kolikant, M. McCracken, J. Paterson, J. Sorva, L. Thomas, and T. Wilusz. A fresh look at novice programmers' performance and their teachers' expectations. In *Proceedings of the 2013 ITiCSE working group reports*, ITiCSE -WGR '13, pages 15–32. ACM, 2013.

[34] J. J. G. van Merriënboer and P. A. Kirschner. *Ten Steps to Complex Learning: A Systematic Approach to Four-Component Instructional Design*. Lawrence Erlbaum, 2007.

[35] J. J. G. van Merriënboer and H. P. M. Krammer. Instructional strategies and tactics for the design of introductory computer programming courses in high school. *Instructional Science*, 16(3):251–285, 1987.

[36] S. Xinogalos. Object oriented design and programming: An investigation of novices' conceptions of objects and classes. *ACM Transactions on Computing Education*, 2014.