Abstract

An approach to adding concurrency to an existing undergraduate course should integrate driving problems used in analogies, worked examples, and assignments; amplify the subject/topics of the existing course; and ensure that the addition does not have a negative impact on (a) conceptual focus, and (b) student learning, struggle, and engagement. When concurrency is added to a course covering object-based programming, five related design principles for meeting these, sometimes conflicting, requirements are: (a) analogies, worked examples, and assignments are all implementations of simulations of moving physical objects, (b) the user-interface components of the simulations are implemented manually using the MVC design pattern or created automatically, (c) assignment implementations are layered to follow the logical dependence among concepts, (d) the concurrency aspects of the functional components of the simulations are implemented using reusable loop and design patterns, and (e) students can experiment with concurrency extensions to implementations of worked examples and assignments. We followed these principles in multiple course offerings that added concurrency to two different courses on object-based programming. Our data-based evaluation of these offerings, using new inferencing algorithms, analyzed the number of posts and contributions to class discussion forums; the number of entries in class participation diaries; the number of times an automated test is run before it passes; and the percentage of attempts to implement different aspects of concurrency that yield no success. The results show that adding thread execution and creation, synchronization, and coordination has little or no significant effect on measures of engagement, learning, and struggle.

Keywords—education, simulations, animations, object-based programming, thread execution, thread creation, synchronized methods, wait, notify, threads, Java, interleaving, concurrency, command objects

1. Introduction

The motivation for teaching concurrent programming is relatively straightforward. It is replete with concepts that are difficult to self-learn and are the foundation for a variety of fields such as high-performance computing, real-time systems, operating systems, programming languages, distributed and cloud computing, and software engineering. Recent advances in computer hardware have increased the range of systems that make practical use of concurrency, and thus, increased its importance.

This increased importance highlights the need for teaching concurrent programming and the coupled concept of distributed computing, together often referred to as PDC (Parallel and Distributed Computing). Traditionally, concurrent programming has been taught mainly in upper-level undergraduate/graduate courses, such as operating systems and real-time systems, which are not required in many departments. Ghafoor et al [1], estimate that less than 10% of the universities and community colleges that offer CS/CE degrees cover concurrent programming at the undergraduate level.

To address this problem, the National Science Foundation has started a center to encourage new pedagogical methods for teaching concurrency [2], which, in turn, has produced a book on this topic [3]. In these works, concurrent programming has been introduced either in a special course on this concept [4-6] or added to one [7-10] or more [11, 12] courses addressing different subjects in computer science.

Arguably, the research resulting from these efforts has only scratched the surface in the field of concurrency education. Teaching of sequential programming is still an active research area, even though such programming has been extensively taught ever since computing was invented. As mentioned above, many of the recent efforts have added concurrent programming to courses that have traditionally addressed only sequential programming. As there are many ways to couple concurrency concepts with a course on sequential computing, the design space of concurrency education is larger than that of sequential-programming.

In this paper, we consider the introduction of concurrency in a course that covers object-based programming [13]. Such an introduction raises several questions:

- *Requirements*: What requirements should such an introduction be designed to meet? For instance, how should the complexity of the aspects of concurrency introduced compare to the complexity of existing topics in the course?
- *Principles*: What are some reusable principles for helping meet these requirements? For instance, should assignments on the concurrency topics be layered or independent?
- *Metrics*: What metrics can help determine how well the requirements are met? For instance, how should the complexity of concurrency and existing topics be compared?

- *Evaluation*: How well do course offerings that follow these principles meet these requirements based on these metrics? For instance, how do scores on concurrency topics compare to exam scores on existing non-concurrency topics?

We provide first-cut answers to these questions based on a decade of experiments with adding concurrency to two courses on object-based programming. Section 2 motivates and abstracts the main reusable concepts in our work: the requirements and principles shared by the discussed course offerings, and the metrics used to evaluate them. Using our course offerings as examples, Section 3 shows how our principles can be translated into a variety of concrete designs. Section 4 presents an analysis of data collected from these offerings. Section 5 evaluates our implementations using both data analyses and qualitative arguments. Section 6 concludes with a discussion of related work, contributions, and directions for future work.

A preliminary version of Section 3 has been previously presented in a workshop paper [14].

2. Overview of Main Concepts

2.1 Requirements

Two fundamental questions faced when extending an existing course with concurrency concepts are:

1) Which *aspects* of concurrent programming are taught?
2) What *problems* are used in real-world analogies, worked examples, and assignments?

Six requirements in resolving these questions are:

1) *Problem Matching*: The distance between problems used in real-world analogies and worked examples and assignments should be low so that the steps in the analogies guide the algorithms in the implementations. Problem matching is higher if a larger number of actions on the physical objects in analogies correspond to operations on the data structures of the worked examples and assignments.
2) *Learning Matching*: The learning from the aspects of concurrency introduced should be more than or on par with at least some existing topics of the extended course.
3) *Struggle Matching:* The amount students struggle with the aspects of concurrency introduced should be less than or on par with at least some existing topics of the extended course.
4) *Subject/Topic Amplification*: The addition of concurrency should amplify the understanding of the general subject of the course, and ideally, also specific existing topics covered by the course, thereby allowing the subject/topics to be understood in more depth.
5) *Engagement Matching*: The interest students show in learning concurrency should be more than or on par with what they show in at least some topics of the extended course.

6) *Conceptual Focus*: Worked examples and assignments should focus on the concepts taught rather than implementation details irrelevant to these concepts.

These requirements are, of course, related. The distance between the analogies and worked examples/assignments has an effect on how complex concurrency appears to the students, and thus on how much they learn from and struggle with the concepts. For example, if the real-world analogy is concurrent manual execution of cookbook recipes [4], and implementation of a worked example or assignment involves concurrent production and consumption of items [15, 16], then arguably, the students have to cover a large gap between analogy and implemented problem than they would if the analogy is concurrent manual sorting and the implementation involves parallel automatic sorting [8]. Similarly, introducing parallel sorting in a course that covers sorting [8] amplifies sorting, and is thus more likely to result in engagement, learning, and struggle matching than teaching it in a course on object-based programming. Implementing well-understood real-world analogies can make students anticipate the teaching of concurrency, thereby increasing engagement. For instance, if the students have been told that the overall goal is to create a simulation of multiple recipes being executed concurrently, then after they have learned to simulate one recipe execution, they are likely to anticipate, and even ask for, an abstraction that allows concurrent execution of recipes. On the other hand, requiring students to implement details of the real-world problem that are irrelevant to the course topic can negatively affect the conceptual focus requirement. For instance, if the course topic is algorithms, then asking students to manually implement the user-interface of a stovetop reduces conceptual focus.

Besides the two primary questions identified above, a course on concurrent programming, like any other course, must also address several secondary questions: What combination of live and recorded lectures and hands-on learning are used to communicate the concepts? Are there multiple-choice quizzes and/or written exams to test student understanding? Are assignments implemented in the classroom, with instantaneous instructor help, or at home with instructor help through discussion forums and office hours? Which, if any, of the homework assignments are optional, meant for more advanced and/or engaged students? What is the nature of allowed/encouraged peer collaboration?

How these questions are addressed influences both how well the requirements above are met, and more important, the data available to perform this evaluation. The answers to these questions define a large design space of approaches. The course offerings discussed here have experimented with different points in this space. Each of these offerings has added concurrency to an existing course on object-based programming. They share a common set of design principles for meeting the requirements when the course extended covers such programming.

2.2 Principles

Object-based programming [13] is often introduced by drawing an analogy between computational objects and familiar physical objects (e.g. cars). Object methods correspond to operations on these objects (e.g. accelerate and brake), object components correspond to physical components (e.g. chassis and tires), object classes correspond to blueprints, and class hierarchies correspond to physical taxonomies (e.g. cars are special cases of vehicles). Thus, students can use their understanding of the physical world to both understand aspects of object-based programming explained to them, and also anticipate and infer concepts not explained so far. The use of these analogies is not surprising as the first object-based language, Simula-67, was targeted at simulations of the physical world.

A fundamental insight in our work is that simulations of the real world also serve as powerful analogies for teaching the three main aspects of concurrent programming: threads (their creation and execution), synchronization, and coordination. To continue with the car example, multiple cars moving simultaneously corresponds to multiple threads executing concurrently, cars entering an intersection serially corresponds to unrelated threads synchronizing with each other to prevent unwanted interference, racing cars waiting for the start signal from an organizer corresponds to related thread coordinated by an external thread, and cars in a relay race waiting for team members' cars to arrive corresponds to threads exhibiting internal coordination.

A related insight is that these analogies can be converted into implemented simulations in which concurrent threads do not simply correspond to concurrent physical actions; they are implementation of such actions in the simulations. Such a conversion, of course, directly meets the driving-problem matching requirement. It also helps meet the complexity-matching requirement, as the simulation implementations can show correct and incorrect use of concurrent programming. If the simulation user-interface shows two cars colliding in an intersection or a driver in a relay not waiting for a teammate, then the programmer can infer incorrect use of thread synchronization and internal coordination, respectively, which, in turn, reduces the struggle involved in debugging the concurrent program. Debugging is known to be a difficult problem even in the single-threaded case [17, 18], and is particularly difficult in the concurrent case [17, 19, 20].

Thus, issues faced in creating, synchronizing, and coordinating concurrent activities in a virtual space are analogous to ones faced in creating, synchronizing and coordinating concurrent activities involving moving objects in a physical space. If a concurrent program simulates concurrent moving objects, then there is a one-to-one correspondence between these abstract computing issues faced by the application and concrete physical real-life concerns. Such a program can then be used as a worked example to explain the issues and computing concepts for resolving them. It can also be used as a programming assignment to exercise these concepts and identify and verify correct use of these concepts.

If the simulation implementations involve the creation of new kinds of objects or use of new design patterns, then the requirement of topic amplification is met. Design and other patterns [21] such as loop patterns [22] also reduce struggle as students have clearly articulated bases for their code design and implementation. Moreover, concept presentation, worked examples, and assignments can be layered, which increases (a) topic amplification, as it shows the use of lower layers in the explanation and implementation of higher layers, (b) conceptual focus, as details irrelevant to the higher layer can be the focus of the lower layer, and (c) engagement, as after implementing a layer that partly implements the real-world problem, students can anticipate and even demand concepts in higher layers needed to make a more full implementation of the problem. Layering is particularly suitable for teaching the three concurrency aspects, which are inherently layered. It is not possible to synchronize threads without creating them, and thread coordination requires synchronized queues. Thus, thread creation and execution is a prerequisite for synchronization, which in turn is a prerequisite for coordination. These three concepts can, thus, be incrementally introduced and exercised. Additionally, depending on the mix of students, one or more of the assignment components covering them can be made extra credit.

A barrier to taking the simulation approach is the high cost of implementing interactive user-interfaces [23], which interferes with the conceptual focus requirement. This barrier is the reason that in a course on object-based programming, typically, the analogies are physical objects such as cars, but the worked examples and assignments involve abstract data structures such as stacks and queues. There are three main approaches to overcome this barrier:

- *Simulation-Specific Scaffolding Code*: The instructor provides scaffolding code to implement the user-interface components of a simulation (worked example or assignment), with the students responsible for understanding and implementing only its functional components, whose interface is defined by the instructor. This approach puts overhead on the instructor each time a new kind of simulation is developed and also does not give students any flexibility in defining the nature of the functional components. In particular, they cannot add optional functional components in their assignments for which the corresponding user-interface components do not exist, limiting engagement. Implementations of real-word analogies provide ample opportunities to add such extensions, so an important benefit of using them is lost using this approach.
- *User-Interface Generation*: A state-of-the-art user-interface generator [24] is used to create the user-interface components of a simulation automatically from its functional components. Because of the automation it offers, such a tool limits the range of generated user-interfaces, but examples can be chosen from this range.
- *Object-Oriented User-Interface Implementation*: User-interface implementations in modern languages exercise several important concepts taught in courses on object-based programming such as inheritance and the observer pattern. Moreover,

these implementations can exercise an important design pattern identified for creating user-interfaces – the MVC (model-view-controller)] pattern [25] – also taught in some courses. In this approach, students manually implement the user-interface in layers to exercise concepts taught in the existing course on object-based programming.

This reasoning leads to the following related principles for adding concurrency to a course on object-based programming:

- *Moving-object simulations:* Analogies, worked examples, and assignments are implementations of simulations of moving physical objects.
- *Reusable patterns*: The concurrency aspects of the functional components of the simulations (worked examples and assignments) are implemented using general, reusable, patterns.
- *Concepts-based user-interface implementations*: The user-interface components of the simulations are either implemented manually using the MVC design pattern or generated automatically using a user-interface generator.
- *Assignment layering*: Assignment implementations are layered to follow the logical dependence among concepts covered in sequential and parallel programming.
- *Extendible implementations*: Students can experiment with extensions to implementations of worked examples and assignments.

2.3 Overview of Metrics

The requirements of learning, struggle, and engagement matching require metrics to perform the comparison. The data we have to compute such metrics are extracted from logged test executions, recorded in-class interactions, forum discussions, and quiz, assignment, and exam scores. Table 1 overviews the metrics we compute from these data. Higher (lower) values of the non-starred (starred) metrics are better.

Table 1 Metrics Overview

Metric	Explanation	Learning	Struggle	Engagement
Quiz Scores	Percentages on multiple-choice quizzes on the topic	Yes	Yes	Yes
Assignment-Topic Scores	Percentages on assignment work on the topic	Yes	Yes	Yes
Exam Scores	Percentages on written exam questions on the topic	Yes	Yes	Yes
Number of Test Attempts*	The number of times a test related to the topic was run unsuccessfully before its final run by a student		Yes	
All and Per-Student Topic Posts *	Number of posts on technical questions related to the topic made by all students and by each student		Yes	
Contributions Ratio	Number of contributions per technical post on the topic	Yes	Yes	Yes
Number of Recorded In-Class Answers (My Q/A)	Number of answers by a student to a question on the topic posed by the professor in class	Yes	Yes	Yes
Number of Recorded In-Class Interactions (Class Q/A)	Number of class interactions related to the topic recorded by a student	Yes		Yes

A single metric can influence how well multiple matching requirements are met. For instance, a higher value for in-class answers indicates higher learning and engagement and lower struggle. As our requirements involve comparisons rather than absolute values, a comparison of these metrics also indirectly performs the required matching. In particular, we can show that all of our matching requirements are met if we compare the metrics for each concurrency topic, C, with those for an existing topic, E, in the extended course, and show that the values of non-starred (starred) metrics for C are greater (less) than or equal to those for E. These metrics are discussed in more depth when we use them in data analysis.

Not all requirements are associated with metrics. As we show later, the requirements of problem matching, subject/topic amplification, and conceptual focus are evaluated by inspecting the design of the evaluated course offering.

3 Implementations of the Principles in Course Offerings

This section addresses how the principles above can be translated into concrete designs by illustrating the nature of analogy-based worked examples and layered assignments in our concurrency-based course offerings. Section 3.1 sets the context for the examples and assignments by identifying the courses that were extended and the specific reasons for adding concurrency to them. Section 3.2 describes a set of worked examples, a subset of which was used in every discussed offering. Section 3.3 presents layered assignments used in our most recent offerings, and Section 3.4 overviews how its correct working was assessed automatically by both instructors and students. Sections 3.5-3.8 overview the assignments in other offerings.

3.1 Context for Course Offerings

The requirements and principles above form the basis for the design of several offerings of two different courses on programming in Java taught in our department:

- *Introduction to Programming*: The course provided an introduction to programming in Java and was taken by both majors and non-majors. It was not required for our majors.
- *Foundations of Programming*: The course assumed students knew the basics of procedural programming such as functions, arrays, and loops. It focused on more advanced concepts found in Java such as objects, classes, interfaces, exceptions, assertions, generics, and design patterns such as the observer, factory, and MVC patterns [26]. This was a required course for our majors and a prerequisite for the data structures course.

Neither of these courses covered concurrency officially, but as with other courses, instructors teaching them typically created versions with material that went beyond the official requirements. In a recent course restructuring, a new version of *Introduction to Programming* has become a pre-requisite for the data structures course, which has become a prerequisite for a new version of *Foundations of Programming*. Based on the efforts and experience reported here, concurrency has now become a required topic in the new incarnation of *Foundations of Programming*.

As this work was done before the restructuring, we will refer to the two courses above as CS-1 and CS-1+. Many of our majors, including freshmen, took CS-1+ without having taken CS-1. Those who took both courses were exposed to objects and classes in both courses.

The motivation for adding concurrency to these two courses came from other members of our department. In 2007, a Ph.D. student was teaching a version of the CS-1 course developed by the first author to fulfill his teaching requirement. The instructor was interested in giving as a homework assignment, the "rabbit crossing the highway" game, which required dynamic thread creation. Animations were already a part of the course material to demonstrate the use of loops. The first author extended the course material, worked examples, and a tool he had built for teaching, to cover thread execution, creation, and synchronization. This version of the course, along with the game, was offered again by another Ph.D. student in 2011.

In 2011, in a faculty meeting, our department head and several faculty members lamented the fact that students could leave our program without knowing any concurrency and wondered if CS-1+ could overcome this problem. The first author added concurrency to versions of the course taught each fall from 2011 to 2018 except for 2014 when he was on research leave.

To accommodate the new material, some topics from the pre-concurrency offerings were removed. We identified two general criteria for removing material from a course that adds concurrency to an existing course: (a) it has the least relevance to creating simulation-based assignments; (b) it duplicates some of the concepts taught through concurrency. We removed from the previous course material (a) the use of assertions in proving program correctness and an in-depth discussion of shallow and deep copy, as they did not fit well with simulation-based layered assignments, and (b) an in-depth discussion of delegation vs inheritance and undo and redo, as these two topics duplicated some of the concepts learned through concurrency. As undo/redo fit in well with the simulation-based assignments, students were encouraged to learn it through recordings of lectures, and implement extra credit features that exercised it. All of the removed topics were discretionary, optional topics in the official course description. It was our judgment that learning increased by covering concurrency was more important than the learning lost by not covering the removed material. These sacrifices do not need to be made in the new incarnation of the course, as the new version no longer introduces objects and classes.

3.2 Converting Analogies to Worked Examples

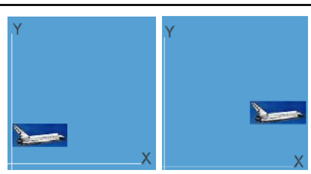
The worked example for each concurrency topic has been used unchanged for every course offering (of CS-1 and CS-1+) that covered that topic. In particular, the worked examples to demonstrate concurrent and synchronized thread execution were used in both CS-1 and CS-1+ offerings, which all covered these two topics; and the worked examples for coordination were used in all

CS-1+ offerings since 2012 but not in any (CS-1 or CS-1+) offering before that, which did not cover coordination. The worked examples for different concurrency topics, like the assignments, were layered.

In our worked examples, our example simulation is a shuttle object positioned in a Cartesian plane, which provides a command to change its position. This object is used to illustrate several non-concurrency concepts including objects, classes, object composition, and the observer and MVC patterns. To introduce concurrency, we extended this object with a command to animate its movement from the origin to its current position. For implementation reasons, the icon for a shuttle remains horizontal during its animation from the origin. In this section, we use different versions of this example to illustrate the four different aspects of concurrency we considered – threads, synchronization, external coordination, and internal coordination.

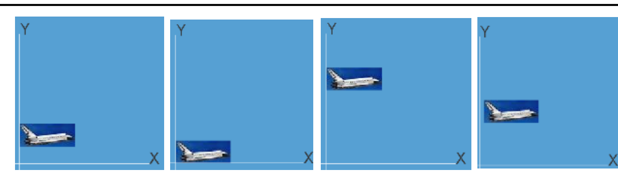
3.2.1 Execution and Creation of Threads

We explain the basic concept of thread creation and execution using two instances of the space shuttles whose current positions are different. Their animations (from origin to current position) can be executed sequentially by a single thread or concurrently by two different threads. Figures 1 (a, b) and 2 (a, b) show the difference in the behavior of the two alternate implementations when we transition from snapshot (a) to (b). In the sequential case, the Y coordinate of the left shuttle changes, while the right shuttle remains at its current position. In the concurrent/interleaved case, the Y coordinates of both shuttles change simultaneously, giving an appearance of concurrent movement. The first case simulates a single pilot flying both shuttles sequentially and the second one corresponds to two different pilots flying them concurrently. The second simulation of the two-pilot, two-shuttle real-world case motivates the need for multiple threads. Whether the underlying implementation correctly mimics this case by creating two independent thread activities is apparent in the external behavior of the shuttle. Students were exposed to the correct and incorrect use of concurrency abstractions, and their tangible effect on the simulations, through both lectures and exercises.



(a) Snapshot^1 (b) Snapshot^2

Figure 1: *Shuttle^1 moves along Y axis, Shuttle^2 does not animate*



(a) Snapshot^1 (b) Snapshot^2

Figure 2: *Shuttle^1 and, Shuttle^2 both move along Y axis*

The concept of thread creation and execution is the foundation of concurrency. While the real-world analogies of concurrent objects in motion simplify its explanation, students need to understand how to translate an analogy into a program in the given programming language. In our CS-1 offerings, this conversion was done using a home-grown declarative mechanism based on the Java **synchronized** keyword discussed later. In all of our CS-1+ offerings, Java concurrency abstractions were used directly.

Java provides multiple ways to do this translation. One approach is to create a subclass of the Java *Thread* class that (a) has the complete code of the animating shuttle, and (b) overrides the *run()* method of *Thread* to execute the animation loop. A concurrently animated virtual object, such as the shuttle, is then an instance of this subclass. Consistent with the subject/topic amplification requirement, we use a more sophisticated approach (from the point of object-based programming) involving the use of a new design pattern. The pattern, illustrated in Figure 3, replaces the virtual object in the approach above with four kinds of participating objects: an animated object, an animating object, a command object, and a thread object. It also includes a class or object that serves as a thread client.

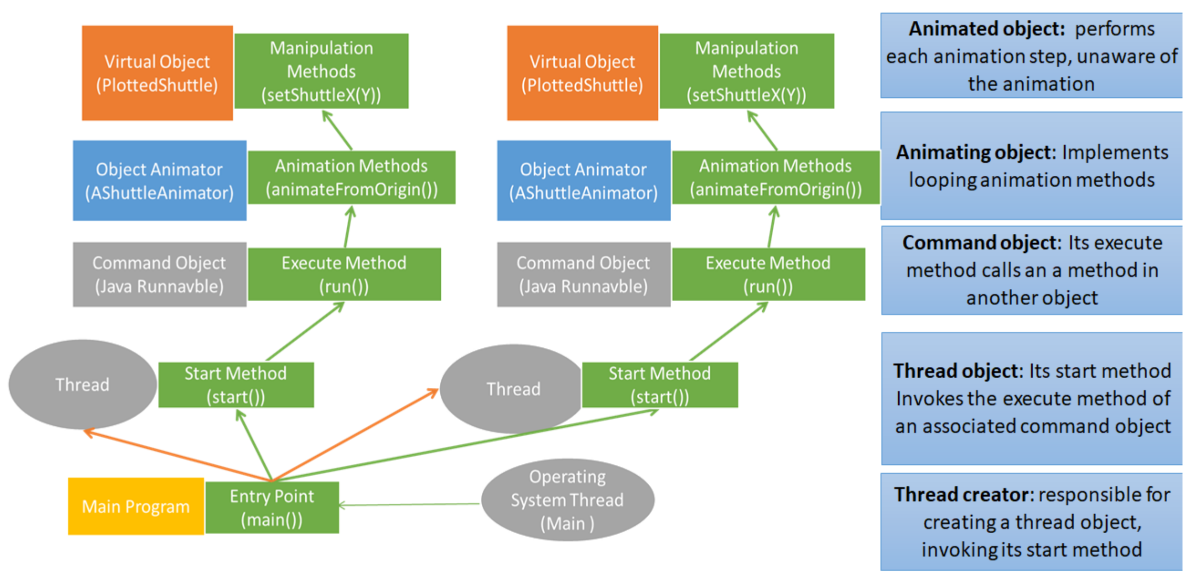


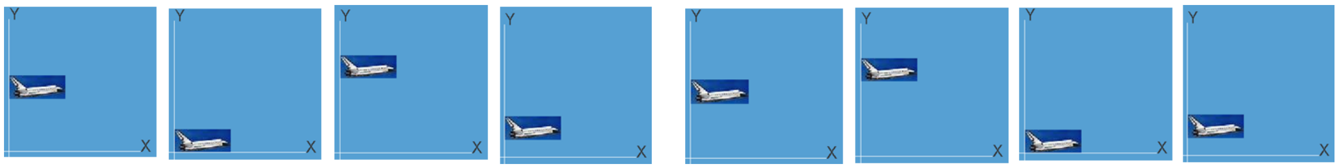
Figure 3: Concurrent Animations with a Design Pattern Amplifying the Separation of Concerns Principe

A virtual object simulates the physical object and provides methods to manipulate it in a virtual space. In our shuttle example, this object is an instance of *PlottedShuttle*, and it provides the *setShutleX()* and *setShuttleY()* methods to move its current position. An object animator provides one or more methods to animate the manipulation of the virtual object in animation loops. In our example, this is the *animateFromOrigin()* method, which animates the movement of the shuttle from the origin to its current position. For each animation method in the animator, a command object [27] is created to call the method. This abstraction was invented in the context of undo/redo. It provides an (a) *execute* method whose purpose is to call a method in another object to initially execute or redo the method [27] and (b) an *undo* method whose purpose is to undo the method invoked by execute. In Java, a command object without an undo method is used to create threads. Such an object is an instance of the Java *Runnable* interface, and its execute method is called the *run()* method. This method is automatically invoked by the *start()* method of a predefined system thread object, which, in Java, is an instance of the *Thread* class. The thread client is responsible for instantiating this class and invoking the *start()* method on the instance. The thread client may be the main thread created by the operating system when the entry point of the main program is executed, or a thread dynamically created during program execution. Two independent instances of this pattern are executed by the main thread to simulate the concurrent movements of two independent shuttles.

This design pattern serves not only to help students understand and apply the foundational concurrency concept of thread creation but also to amplify the programming concept of object composition. The classical examples of stacks and queues give few opportunities to illustrate interaction among multiple types of objects. Here, students see three kinds of programmer-defined objects - virtual object, animator, and command object - interacting with the powerful predefined thread object. To the best of our knowledge, the term command object has been used only in the context of undo/redo. By showing that it applies also to a Java *Runnable*, we bridged the gap between the two concepts, which allowed students to add an undo method to existing implementations of Java *Runnable* to support undo/redo as extra credit in their assignments. As we see later, their assignments were also augmented with an interpreter, which invoked the Java *run()* methods of the *Runnable* instances in the same or different threads to support a command-line interface. In this design pattern, the Java thread object delegates command execution to a programmer-defined command object. The alternative approach for thread creation in Java, identified earlier, uses inheritance instead of delegation, and is popular because of its simplicity. In this approach, the command object class is a subclass of the Java *Thread* class that overrides the *run()* method of the latter. Students learn that the delegation approach is superior as a command object can be used independently of a thread object (in undo/redo and command interpretation, for example). They learn that the IS-A relationship does not logically hold between a command object and a thread object, as a command object is not conceptually a thread object – the latter is a concurrent activity, while the former defines a sequence of actions executed by a concurrent or sequential activity. This delegation-based design pattern approach will also allow much of the code used in this worked example and student assignments to be reused in examples and assignments exercising more advanced aspects of concurrency, thereby supporting the layering principle.

3.2.2 Synchronized vs Unsynchronized Concurrency

Consider a variation of the thread example of Figure 2 in which the movement of a single shuttle is controlled by two different animation loops executed by two different threads, which take it on the Y-axis along the paths (y^{11}, y^{12}, …) and (y^{21}, y^{22}, …), respectively. Figures 4 (a..d) show that when the threads are not synchronized, the shuttle oscillates between the trajectories computed by the loops, taking y coordinates (y^{11}, y^{21}, y^{12}, y^{22}, ...). Figures 4(e..h) show that when the threads are synchronized, the shuttle first follows the trajectory computed by the first loop, taking Y positions (y^{11}, y^{12} ...), and then the trajectory computed



(a) Pos y^{11} (b) Pos y^{21} (c) Pos y^{12} (d) Pos y^{22} (e) Pos y^{11} (f) Pos y^{12} (g) Pos y^{21} (h) Pos y^{22}

Figure 4: Unsynchronized (a..d) vs Synchronized (e..h) Control of a Single Shuttle

by the second loop, taking Y positions (y^{21}, y^{22} , ...). These two virtual cases correspond to the real-world situation of two pilots, with different flight plans, controlling the same shuttle. The external behaviors of the two cases, motivate, explain and demonstrate the concept of synchronization.

CS-1 students learned only the **synchronized** keyword using this example. In CS-1+ offerings, a variation of the design pattern of Figure 3 is used to create the implementation of the synchronized and unsynchronized cases. The difference in this pattern is that the two thread command objects share a single object animator rather than operating on different animator objects (Figure 5), and make two independent calls to its animation method. In the synchronized case, the Java **synchronized** keyword is used to declare this method as synchronized. Students learn that a thread is (a) queued if it tries to execute a synchronized method on an object while another thread is executing such a method on the object, and (b) dequeued when it is at the front of the queue when another thread completes execution of a synchronized method in the object. They also learn how to determine which methods should be synchronized. In this example, synchronizing the execute methods of the command objects serves no purpose, as different objects are associated with different synchronization queues. Similarly, synchronizing the methods of the shared virtual object does not help as multiple loops can interleave their execution of them, causing interference. It is the animation method of the shared animator object that must be synchronized to prevent interference. Figure 5 also amplifies the teaching of the programming principle of separation of concerns, which says that functionality that can be independently changed should be implemented in separate classes. Separation of the command object and object animator functionality in different classes allowed us to animate the same object concurrently by two threads. Had this

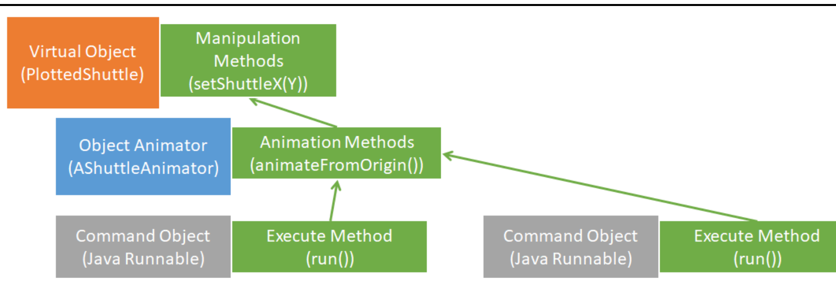


Figure 5: Two Thread Command Objects Sharing an Object Animator

functionality been combined into a single command/animation class, the animator would be bound to a single command object and associated thread, and its animate method would be executed only by that thread.

32.3 Coordination: External and Internal

Thread coordination was added to CS-1+ offerings in 2012. It is needed in related concurrent activities that accomplish some larger goal. The activities of related threads may be coordinated externally by a separate controller thread or internally by the threads themselves. Let us first consider the external case in the context of our shuttle example.

This case has an additional user-interface with two buttons, *Proceed* and *Proceed All,* for external control*,* and a visualization of a queue of waiting threads. We call it the *clearance manager interface* as it gives clearance for shuttles to begin their flights.

The simulation starts with two threads, animating independent shuttles, in the displayed queue, as shown in Figure 6(a), which simulates two physical shuttles queued to enter a physical launching pad. Figure 6(b) shows the effect of interactively pressing the *Proceed* button. The first thread is removed from the queue and starts animating its shuttle. Figure 6(c) shows that clicking the

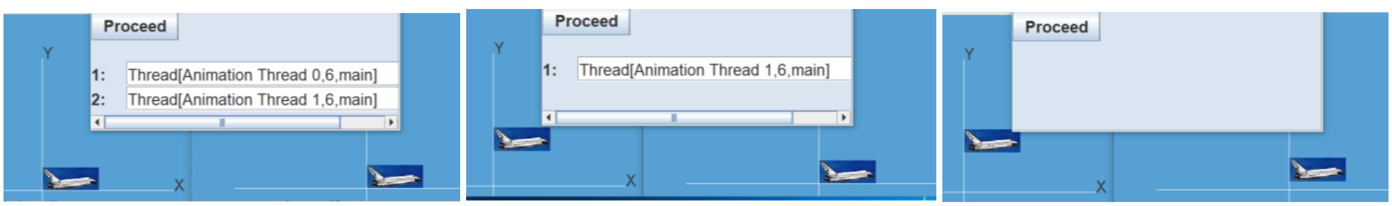


Figure 6 (a) Both Threads in Queue (b) First Thread Dequeued, Starting Animation (c) Second Thread Dequeued, Starting Animation

Proceed button again dequeues the second thread, which now starts animating the second shuttle. Clicking the *Proceed* button, of course, corresponds to an air traffic controller giving clearance to the next waiting shuttle to takeoff.

In a variation of the above scenario, the shuttles are again queued as before (Figure 7(a)). This time the *Proceed All* button is

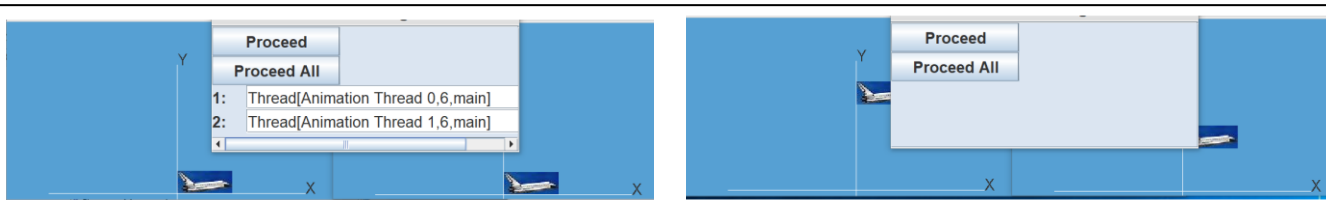


Figure 7 (a) Both Threads Waiting in Queue (b) All Waiting Threads Dequeued Starting Both Animations

interactively pressed to unblock all waiting threads simultaneously (Figure 7 (b)). Unblocking all waiting threads corresponds to signaling multiple shuttles at independent launching pads to start a race or a joint mission.

To support external coordination, we provide the predefined abstraction of a clearance manager, implemented by two classes, *AClearanceManager* and *ABroadcastingClearanceManager*. *AClearanceManager* provides a *waitForClearance()* method, which makes a synchronized call to the Java *wait()* method to block a thread. Similarly, it provides a *proceed()* method that makes a synchronized call to the Java *notify()* method. In addition, it maintains the queue of threads waiting to be notified. *ABroadcastingClearanceManager* is a subclass of *AClearanceManager,* and provides an additional *proceedAll* method, which makes a synchronized call to the Java *notifyAll()* method. Figures 6 and 7 show the user-interfaces of instances of the two clearance classes, which display the queue of threads blocked by the *waitForClerarance()* call on the clearance manager instance. The *Proceed and Proceed All* buttons in these user interfaces execute the corresponding methods of these two classes.

The design pattern used to implement the coordinated shuttles is the same as the one shown in Figure 3, as the animations involve different shuttles. The difference is in the loop patterns executed by the animation methods. The animation methods of the uncoordinated examples implement the change-sleep loop pattern of Figure 8. In this pattern, the methods implement one or more loops in which each iteration changes the animated object in some way and then sleeps for a specified time interval. In our shuttle example, the changes consist of moving the animated object, and a single animation method (*animateFromOrigin*) executes two loops to animate the shuttle in the Y and X directions respectively.

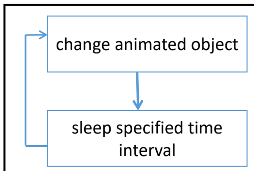


Figure 8 Change-Sleep

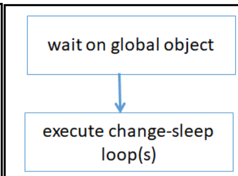


Figure 9 Wait-Loop

The animators in the external coordination examples use the wait-loop pattern of Figure 9. In this pattern, before executing one or more of their loops, an animator executes the *waitForClearance()* method of a global clearance manager, which is notified by the end-user by calling the *Proceed* or *Proceed All* button of the clearance manager. The interactive user can thus play the role of a controller determining when a waiting animator launches its shuttle.

The coordination above is external as an external human/thread, not responsible for flying/animating the shuttle, signals waiting pilots/threads to dequeue and start the flight/animation; it does not itself wait. In internal coordination, the threads/pilots that wait and animate/fly also do the signaling necessary to dequeue and unblock other threads/pilots. Figure 10 illustrates such coordination.

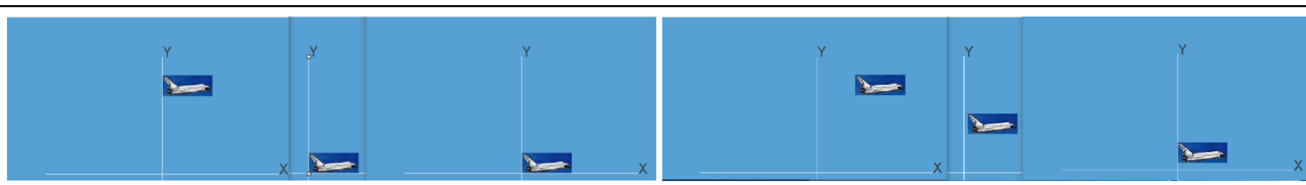


Figure 10 (a) First Thread Reserves/Locks Y-Axis, Other Threads Wait (b) Second Shuttle Finishes Ascent, Third Shuttle Starts Ascent

Threads now have to lock or reserve the Y-axis, which is initially unlocked. A thread that successfully reserves the axis unlocks the axis when its shuttle has climbed to its highest point and started its flight in the X-direction. In Figure 10(a), the thread animating the leftmost shuttle finds an unreserved Y-axis, and starts the ascent of its shuttle, while the other two threads wait. Once the shuttle of the first thread finishes its ascent, the thread signals it has unlocked the Y-axis, at which point the second thread starts animation of its shuttle, while the third thread waits. As soon as the second thread finishes the ascent of its shuttle, the third thread starts the ascent of its shuttle (Figure 10(b)).

The internal-coordination shuttle example uses the wait-loop-notify loop pattern of Figure 11. The animation method executes the *wait()* method, not on an instance of a predefined clearance manager, but a programmer-defined class. Next, it executes a sequence of locked animation actions in a loop, and then executes *notify()* on the global object on which it did the wait. In our shuttle example, it calls the *wait()* method before the loop that animates in the Y direction. After the loop is executed, it executes the *notify()* method on the same global object.

Coordination is the most complex of the concurrency aspects, as it is layered on top of synchronization and threads. Separating it into external and internal coordination and providing a predefined external coordinator allows students to incrementally learn its two components: a mechanism to block the current thread (Java *wait()*) at any point in its execution, and a mechanism to unblock such a blocked thread (Java *notify()*). Making the external coordinator interactive concretely simulates a human controller signaling a waiting shuttle to start, tying this concept to a familiar real-world situation.

Both forms of coordination teach students that a call to the Java *wait()* method of an object puts the current thread in a wait queue associated with the object, which is different from the object's synchronization queue, and that a call to the Java *notify()* method removes the thread at the front of this queue. They also amplify the principle of separation of concerns. It is because the virtual object and the object animator were separate that the same object could be animated by three kinds of animators, implementing the loop patterns of Figure 8, 9, and 11, respectively.

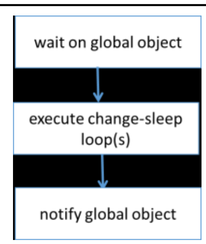


Figure 2 Wait-Loop-Notify

The loop and design patterns above, together with their implementation in the worked shuttle examples, provide the bases for implementing layered student assignments, discussed next.

3.3 Holy Grail Bridge Scene Simulation (CS-1+ 2013-2018)

The 2016 and later CS-1+ offering required students to, individually, implement the concurrency and other concepts in layered assignments, which together, simulated the bridge scene from the movie *Monty Python and the Holy Grail*. Assignments 1 to 9 implemented the static scene shown in the left picture in Figure 12. Assignment 10 forked threads that made avatars of Arthur, Galahad, Robin, and Lancelot move independently. To exercise synchronization, in Assignment 11, they synchronized accesses by multiple threads that manipulate the same knight. To exercise thread coordination, in Assignment 12, they created threads that made the knights (a) wait for clearance from a clearance manager (external coordination) and (b) march to a beat set by the guard (internal coordination). Animating threads were also created by some students (for no credit) to make the knights fall into the gorge on failing to answer a question. Assignments 10-12 covered several topics other than concurrency including exceptions, assertions, generics, recursive descent parsing, and undo/redo. Consistent with the goal of using concurrency to amplify the understanding of other programming concepts, the implementation of recursive descent parsing and undo/redo was integrated with the implementation of concurrency.

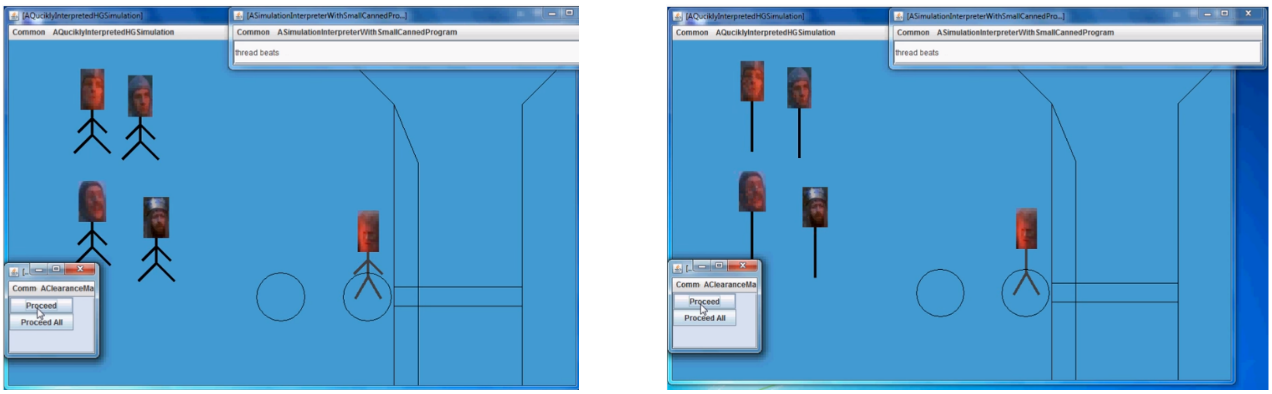


Figure 12 Holy Grail Bridge Scene: Knights Marching to a Beat Set by the Guard: Instructor Implementation

To exercise recursive-descent parsing, students parsed an interactive language allowing the end-user to manipulate avatars in the simulation. The language allowed the user to, for instance, enter: *rotateLeftArm arthur 60* and *rotateRightArm arthur 60* to make the left and right arms of the avatar of Arthur, respectively, rotate 60 degrees. Each type of command was associated with a Java *Runnable* command object whose *run()* method invoked a corresponding method on an avatar object. The command objects created for the *rotateLeftArm* and *rotateRightArm* command types were associated with two different implementations of the Java *Runnable* interface whose *run()* methods invoked operations in the specified avatar to rotate the left and right arms, respectively. The interactive language also allowed the end-user to define interactive procedures, illustrated in Figure 13. The first two commands interactively define two procedures that make the guard move his arms in and out, respectively. The third command defines a coordinating animation *beat* procedure, which (a) makes the guard moves his arms in, (b) signals all waiting threads to proceed so they can take a corresponding action, (c) sleeps for 3000 milliseconds, (d) makes the guard moves his arms out, (e) signals waiting threads to take corresponding actions, and (f) sleeps for 700 milliseconds. The fourth command defines a procedure called *beats* that repeats the beat 10 times. The fifth command executes the *beats* procedure in a new thread. When this

```
define guardArmsIn {rotateLeftArm guard 60 rotateRightArm guard -60 }
define guardArmsOut {rotateLeftArm guard -60 rotateRightArm guard 60
define beat {call guardArmsIn proceedAll sleep 3000 call guardArmsOut proceedAll sleep 700
define beats repeat +10 call beat
thread beats
```

Figure 13 Interactive Programming of Concurrent Animations procedure was executed, the previously started threads for the knights clapped to a beat set by the guard. Such interactive programming of concurrent animations of simulations of physical objects is in the spirit of programming in Alice [28]. The difference here is that students, using procedural abstractions, implemented an Alice-like environment providing interactive concurrent programming, while Alice programmers use a predefined environment. Allowing the end-user to create procedures and threads interactively was extra credit in Assignment 12. In this assignment, using recorded lectures from the 2013 offering of CS-1+, for extra credit, students could extend the command objects to also support command undo/redo.

The above is an example of internal coordination, programmed interactively. The assignment also included a programmed case of such coordination in which the knights marched to a beat set by the guard (Figure 12). In the programmed and interactive internal coordination examples, the animators of the coordinating guard and the coordinated knights essentially implement the loops of Figure 14 and 15, respectively. The leading avatar (guard) executes the change-notify-sleep loop pattern of Figure 14, while the following knight avatars such as Arthur and Galahad execute the wait-change loop of Figure 15. Both loops rely on a global instance of a student-created class that provides public methods to make synchronized calls to the *wait ()* and *notifyAll()* methods of this instance. The wait-change loop, in each animation step, performs a wait on a global object before the change to the animated object, and the change-notify-sleep loop makes a *notify()* or *notifyAll()* call in the global object between the change and sleep.

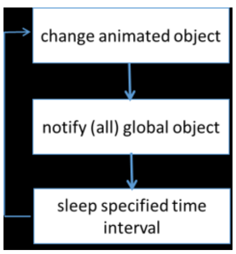


Figure 14 Change-Notify-Sleep

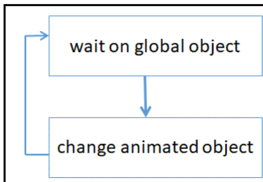


Figure 15 Wait-Change

Whether an aspect of concurrency was implemented correctly was evident to students by interacting with the simulation. Without multiple threads, the avatars would not animate concurrently. Without synchronization, concurrent animations of the same avatar would interfere with each other. Without external and internal coordination, the manipulation of different threads would not be correlated. Students were asked to write a main program that displayed the scene and concurrently performed animation operations on the scene objects to demonstrate the correct working of concurrency (and other aspects).

3.4 Automatic Assignment Assessment

In 2015 we developed Java JUnit checks (Figure 16) to automatically assess aspects of the bridge-scene assignments. Students submitted the assignments by uploading them to the Sakai learning management system. The instructors downloaded them and graded them by running both the programs and JUnit checks. They augmented the automatically computed grade with a manual grade to evaluate the source code, how well the features had been

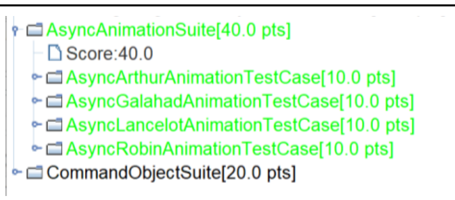


Figure 16 Example Tests

demonstrated by the main program, and correct any errors in the automatic checks themselves. In 2016, the automatic checks were also made available to the students to test their implementations before submission. They could either be executed by the students locally on their computer, in which case data about the test executions were logged on their computer, or on a server, in which case only the final output was logged. These tests could be run an unlimited number of times and gave students automated assignment feedback. Each assignment was broken into a suite of component tests and test suites, where a test suite could have multiple tests and suites in it. We embellished the JUnit framework with the user-interface shown in Figure 16, which allowed students to run individual tests or full suites.

Our exact tests and extensions to the JUnit framework are beyond the scope of this paper. What is important here is the role these tests played. Students could use them to identify requirements and bugs and determine the consequences of not meeting certain regular and extra credit requirements. Instructors used them to automate part of the grading process, making it easier to add extra-credit features. As we see later, the data produced by the instructor and student execution of these tests provide information about student struggle, learning, and engagement. Thus, the tests influenced the nature of the student and instructor experience and provided data about this experience.

3.5 ObjectEditor and Rabbit Game (CS-1 2007 and 2011)

As mentioned above, in our CS-1+ offerings, both worked examples and assignments were implemented from scratch. Our CS-1 offerings relied on a homegrown tool called ObjectEditor [24, 29, 30], which is based on the MVC design pattern. It automatically creates a user-interface, both view and controller, for any Java model that follows certain conventions or patterns in its method signatures. The view displays the model data, both graphically and textually, and the controller provides menu items and buttons to execute the methods of the model. To illustrate user-interface generation, consider the shuttle example. Its public methods are shown in the Java interface in Figure 17. ObjectEditor uses the nature of the method signatures (e.g. whether they are getters or setters and the types such as *CartesianPlane* referenced by then) in the interface to derive a tree of model properties, displayed in the left pane of Figure 18. Based on the nature of the signatures (e.g. whether they are getting or setting an X or Y coordinate), some of these properties are classified as graphical shapes, and others are recognized as textual properties. The graphical ones are displayed in the graphical pane (bottom-right pane), and the textual ones

```
public interface PlottedShuttle {
  CartesianPlane getCartesianPlane();
  ImageWithHeight getShuttleImage();
  int getShuttleX();
  void setShuttleX(int newVal);
  int getShuttleY();
  void setShuttleY(int newVal);
  void resetShuttle ();
}
```

Figure 17 Shuttle Interface

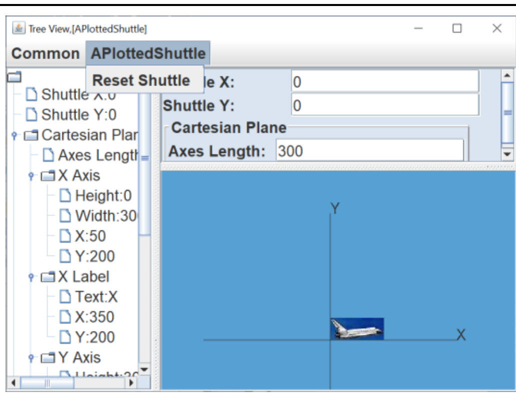


Figure 18 Generated User-Interface

are displayed in the textual pane (top-right pane). Methods such as *resetShuttle* that are not responsible for defining model properties can be invoked through a menu, as shown in Figure 18. ObjectEditor provides an API to hide the display of each of the three panes shown in Figure 18. CS-1 students relied only on ObjectEditor for the user-interface. The CS-1+ students used it as “training wheels”: they first used it to create a user-interface automatically, and next replaced this user-interface with an identical user-interface implemented manually. More ObjectEditor details are beyond the scope of this concurrency-oriented paper and are provided in previous papers [24, 29, 30]. What is important here is that it is possible to create an interactive simulation by defining only the model.

To support concurrency in CS-1, we adopted a declarative approach that leveraged the fact that methods could be invoked interactively through ObjectEditor. If a method had the Java keyword **synchronized**, ObjectEditor created a separate thread to execute it. Thus, in Figure 19, the *Start Game* menu item invoked the *startGame()* method in a separate thread if the method was synchronized. Students were taught the behavior of automatically created synchronized and unsynchronized threads through the worked shuttle example. They understood that the keyword **synchronized** made sense only if the method was invoked by a thread, and also that ObjectEditor used it to automatically create the threads for them. Thus, like other declarative mechanisms, this abstraction allowed students to determine what should be parallelized without worrying about how the concurrency was implemented. It did not allow them to create unsynchronized threads executing the same method.

In the CS-1 assignments, students were responsible for implementing the “rabbit crossing the highway” game. In the instructor’s implementation, the rabbit was expected to be visualized as a rectangular box (Figure 19). Executing the *Start Game* command was expected to start a new thread that animated the movement of the cars. Once the game was started, the game player could move the rabbit left, right, up, and down, using the menu items, which invoked corresponding methods on the rabbit virtual object. The game finished once the rabbit crossed the highway or was hit by a car. In all of our movement-based animations, a user could interactively change several parameters of the animation including the amount of time slept by the thread between object movements and the number of pixels moved by each step. The top window of Figure 19 shows how this was done in the rabbit assignment.

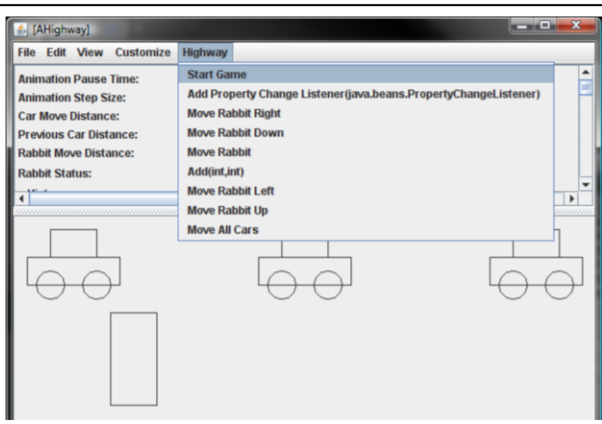


Figure 19 *Rabbit Crossing Highway (Instructor Implementation)*

Like other simulation-based worked examples and implementations, this game was extendible. Figure 20 shows a 2007 embellishment of the CS-1 assignment by a student. A comparison between this implementation and the bare-bones one of the instructor (Figure 19) provides a quick overview of these embellishments. The rectangular rabbit box is replaced by a rabbit image. More important, from a concurrency point of view, the assignment, as given, involved only two parallel activities – the moving rabbit and animating cars. If a rabbit crossed the highway successfully, this student added a second lane of moving *Banshee* cars (Figure 20 (b) and (c)). On another successful crossing, he added flying Warthog monsters to the mix (Figure 20 (d) and (e)). He simply added the functional components of the extensions – ObjectEditor took care of the user-interface.

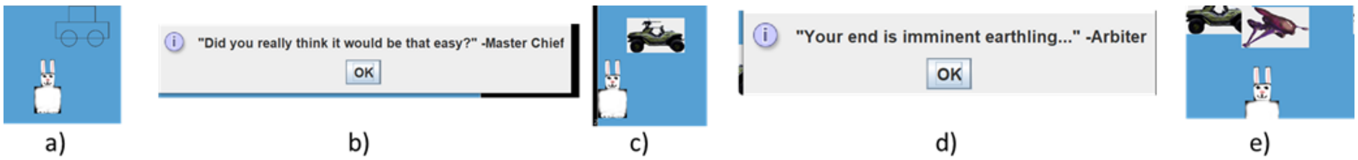


Figure 20 CS-1 Student Invention of Animations of Warthogs and Banshees

His inventions showed the possibility of students creating unanticipated threads. The CS-1 offerings did not cover coordination, but it is easy to imagine extensions to this assignment to exercise this concept. Multiple active threads can simulate the movement of cars in different, possibly intersecting, lanes. A traffic policeman or timer can externally control cars stopped at a stoplight. Internal coordination can be provided by simulating stop signs – when a car clears an intersection, it can signal other cars to proceed. Thus, this is another example showing the possibility of incorporating all aspects of concurrency in a simulated scene.

3.6 Halloween Simulation (CS-1+ 2011)

The first CS-1+ concurrency assignment, given in 2011, was a Halloween simulation. The student implementation of Figure 21 illustrates the required aspects. In the single-threaded assignments implemented before this assignment, students created a static scene consisting of a series of houses, each with a container full of candies. A person visiting these houses was represented by an avatar with a candy basket. Interactive commands were provided to move the avatar and take candies. In the concurrency assignment, the user could also switch between normal and animation mode. In the normal mode, the actions occurred instantaneously, while in the animation mode, certain actions were animated by a newly created thread. For instance, in the normal mode, the *take 1* command to take a single candy simply moved the position of a candy from its container to the basket

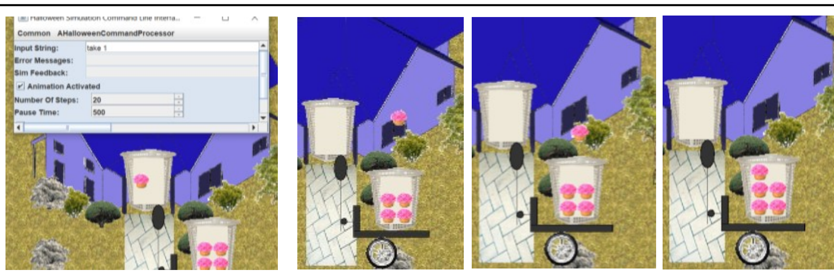


Figure 21 Halloween Simulation with Example Animation (Student Implementation)

using the Java user-interface thread, while in the animation mode, the candy movement was animated, as shown in Figure 21, by a thread created by the user-interface thread.

The student who created the implementation of Figure 21 again showed the possibility of adding threads to support several unanticipated animations. He invented an *eat* command, which animated the eating of the candy by gradually shrinking it, and then animated the effect of overeating of the candy by animating the person vomiting (Figure 22). He also added animations of moving cars on the street on which the houses were located (not shown). His implementation of the full scene, shown in Figures 21 and 22, randomly picked different images for houses, added a variety of bushes to the houses, and scrolled the street, adding and removing houses from the scene, in the direction the avatar moved. In contrast, the instructor's implementation simulated the street by making a static number of copies of a single house. The implementation of the student is particularly striking as he was an economics major. While not all students went as far in embellishing the scenes, to the best of our knowledge, it was rare for a student to not exercise some imagination in creating the simulation. The unrewarded embellishments in the discussed assignments were arguably indicative of the joy and sense of accomplishment students felt in creating the animated scenes and encouraged us to continue our experiments with animation-based concurrency introduction. The 2011 offering did not cover coordination, but this simulation also shows the possibility of accommodating it. Internal coordination could be added to this assignment by introducing multiple avatars, controlled by different animation threads, and (a) making an avatar wait for another avatar to finish taking candies from a house before it can take candies from that house, and (b) allowing multiple avatars to jointly move to another house, with a leading avatar determining the pace of the others. External coordination could be added by making an external agent such as a parent signal that candy could be taken by either one or all candy takers waiting for their turn.

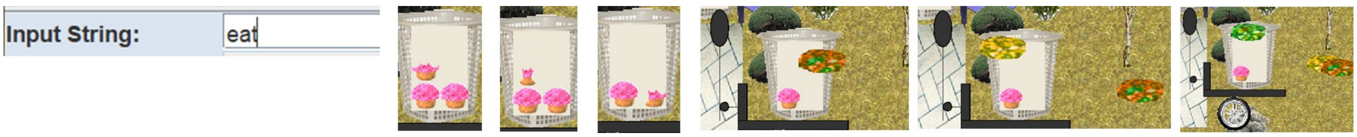


Figure 22 CS-1+ Student Invention of Animations of Eating and Overeating Halloween Candies

3.7 The Wizard of Oz Simulation (CS-1+ 2012)

The next offering of CS-1+, in 2012, covered concurrency more comprehensively, with multiple assignments exercising it. As in other offerings, students built a non-animating scene before the concurrency assignments. For the sake of variety, we experimented with a different simulation (Figure 23) - the movement of characters in the *The Wizard of Oz* movie, with liberties being taken with the story. Commands were provided to create separate threads that

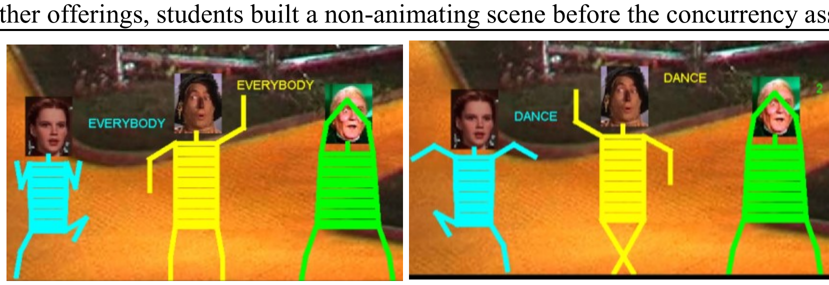


Figure 23 Example Coordinated Oz Animations (Student Implementation)

made each character dance and sing individually. As in the bridge scene project: (a) two threads animating the same avatar had to synchronize their manipulation of the avatar, (b) coordination was provided by making the animations wait for a *Proceed* command from the user or making one character (in this project, *Oz*) provide the beat for other characters (Figure 23), and (c) an interactive language allowed the end-user to create and coordinate threads. The instructor implementation of the Oz scene was like the bridge scene, using stick figures and simple arm rotation for dancing. The student Oz implementation of Figure 23 goes beyond the instructor implementation, both in terms of the avatars and their singing and dance moves. This was true of almost all implementations the first author had a chance to view in class or during office hours. Nonetheless, the 2013 and later offerings all required the implementation of the bridge scene simulation, as this scene was familiar to most students and they seem to have the most fun with it – especially the dialogs.

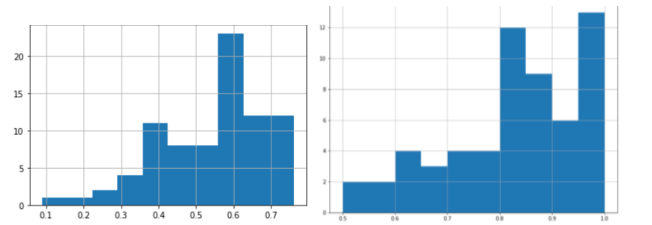
4. Data Analysis (CS-1+ 2012-2018*)*

It was not our goal to describe our multi-year experiment in a research paper when we started it, so we made no special effort to gather data for such a paper. The publication of results based on anonymized data gathered to evaluate and hence improve a course can be expected to do no harm, and for this reason, we have an approved IRB that allows such use. The second, third, and fourth authors led different aspects of the data gathering and processing component of our work, which as we see below, was an intellectual challenge in its own right.

We have no data to evaluate our CS-1 offerings or the CS-1+ 2011 offering. As we see below, we have varying amounts of data to evaluate later CS-1+ offerings. As each offering (for which we have data) differed in some way from other offerings, we keep the analysis of data for different offerings separate, except for Piazza data. Analysis of Piazza data required the use of natural language processing (NLP) techniques, which work better with more data. Therefore, we combine the posts and contributions for all offerings since 2012 in our Piazza data analysis.

We used the Piazza data we collected to evaluate if learning, struggle, and engagement of different concurrency topics matched that of other topics taught around the same time. Based on the comparison possible, the baseline is either (a) all non-concurrency topics, covered around that time (which we refer to as *other*) or (b) exceptions, a standard topic in object-based programming, taught around the same time. If the three requirements (learning, struggle, and engagement) are met, each concurrency topic introduced using our technique fits seamlessly with the baseline topics in the course, and thus, belonged to the course.

The assignment features and quiz, in-class and exam questions were of varying difficulty to help pinpoint a wide range of student learning levels, which is illustrated by histograms of the example exam and non-exam score distributions shown in Figure 24. Our non-exam distributions are skewed right (Figure 24(b)) rather than normal. Therefore, for these data, we used the Wilcoxon



(a) 2012 Midterm (b) 2016 Synchronization Quiz

Figure 24 Different Distributions for Exam and Other Scores

Signed-Rank Test to determine the p-value difference between groups of student metrics. For the exam distributions, we used the dependent t-test, as they conformed more to normal (Figure 24(a)). Unless otherwise stated, all scores were normalized to the percentage of the maximum score.

4.1 Assignment Topic Scores, Submission and Fail Ratios (CS-1+ 2015-2018)

In all offerings for which we have assignment data, concurrency was covered in the last three assignments, which we refer to as A10, A11, and A12. There was not a one-to-one mapping between assignments and topics- each assignment that covered a concurrency topic also covered several non-concurrency topics. For example, A10 included assertions and threads; A11 included recursive-descent parsing of interactive commands and thread synchronization; and A12 included (external and internal) coordination, exceptions, and undo/redo. Thus, we cannot use the Sakai Gradebook assignment scores to determine the impact of concurrency on assignment performance, even though we have such scores for all offerings since 2012.

Our automatic test suites, introduced in 2015, generated feedback files that gave per-suite scores. From these feedback files, we determined the percentage of the maximum score for each topic. To evaluate the learning and struggle matching requirements, we compared scores on concurrency topics from A10-12 for our 2015-2018 offerings with scores on other topics in these assignments. Thus, "other" was considered the baseline. For each year, we considered only those students who had submitted all three of the concurrency-covering assignments (A10, A11, and A12) as threads were covered in all three assignments. Many students did not submit A12, as most of its requirements were extra credit in

Table 2 Topic Score Comparisons, Submission Ratios, Fail Ratios

Comparison	Year	A10-12/A9 Submissions	P-Value	Means	Fail Ratio
Topic Scores Other vs Threads	**2015**	**63/76 (83%)**	**5.16e-12**	**59.0%, 99.6%, +40.6%**	0%
	2016	**36/59 (61%)**	**0.037**	**61.4%, 56.2%, -5.2%**	5.6%
	2017	24/70 (34%)	0.819	69.4%, 68.3%, -1.1%	0%
	2018	19/54 (35%)	0.212	76.2%, 77.6%, +1.4%	0%
Topic Scores Other vs Synchro-nization	**2015**	**63/76 (83%)**	**0.001**	**59.0%, 76.1%, +17.1%**	11.1%
	2016	36/59 (61%)	0.148	61.4%, 68.4%, +7.0%	8.3%
	2017	24/70 (34%)	0.797	69.4%, 66.7%, -2.7%	12.5%
	2018	19/54 (35%)	0.445	76.2%, 67.0%, -9.2%	10.5%
Topic Scores Other vs Coordination	**2015**	63/76 (83%)	**3.92e-7**	**59.0%, 67.1%, +8.1%**	1.6%
	2016	36/59 (61%)	**0.001**	**61.4%, 55.2%, -6.2%**	5.6%
	2017	24/70 (34%)	**0.005**	**69.4%, 53.9%, -15.5%**	0%
	2018	19/54 (35%)	**0.006**	**76.2%, 67.0%, -9.2%**	0%

all years. The results are shown in Table 2. All assignments had regular and extra-credit features on each topic, and we do not distinguish between these in our results.

As we see in the table, the data for different years show different trends. This can be explained by the fact that in 2016, we moved from a lecture-style model to a flipped class model, in which students collaboratively completed twenty hands-on exercises and associated quizzes (called praxes) [31], which they started in class and typically completed at home. No requirements from previous offerings were removed. Thus, this was the most intense semester. To explain its impact, we show in column 3 the submission ratio for each year, which is the number of students who submitted A10, A11, *and* A12 to the number of students who submitted A9, which was the assignment that completed the single-threaded version of the simulation. We see that in 2016 the number is much lower than in 2015. To accommodate concurrency, in all offerings we had tried to remove an equivalent amount of non-concurrency material from the covered concepts but not from the assignments. The 2016 offering amplified this original mistake, making it glaring. This was particularly unfortunate as the last three assignments had also many more features than the previous assignments, and thus weighed much more heavily in determining student grades. We corrected the overload mistake in 2017 and 2018 by making all three of the last three assignments, A10-A12, optional, which, of course, resulted in lowering the submission ratios even further. In all three years with praxes, students typically worked on the assignments during their Thanksgiving break and submitted them on the last day of classes. Given this, the submission ratios, especially in 2017 and 2018, are an indication of high engagement with the extra credit assignments. Students knew the extra credit assignments would reduce the pressure to study for the final exam, and those who did all (or even some) of them were implicitly indicating that putting work into them was more worthwhile. The vast majority of A students earned a course score of greater than 100%. For example, in 2018, the A threshold was 95%, and 25 out of the 34 students who received an A had a final course score greater than 100%, with the highest being 140%. Arguably, these students worked more for the learning than the grade. Many non-A students also submitted all three assignments, often covering many aspects of them. A notable case was a student who received 8% below the mean midterm score in fall 2018 but did enough work on the extra credit assignments to receive a 99% course score.

If assignment topic scores are to be interpreted in isolation, then arguably the 2015 offering provides the most reliable data about the means and p-values, as there were no quizzes or optional assignments, though implementation of internal coordination (programmed and interactive) was optional even that year. The data for this year show that the scores on the concurrency topics were significantly higher than those on other topics, and that the increase was highest for threads and lowest for coordination, which is consistent with the layering among the three concurrency topics.

The reason for the significant relative decreases in mean concurrency scores in other years can be explained by the hypothesis that (a) students picked what other topics they worked on based on the amount of time they had, and (b) there was nothing fundamental about their worse concurrency scores. To help validate this hypothesis, we developed an algorithm to determine the fail ratio for each topic, which is the number of students who tried to implement some feature associated with the topic but received a zero score from all tests on that topic. To determine if they attempted some work on that topic, we searched their source code for keywords associated with the topic: for example *Thread* and *sleep* for threads, *synchronized* for synchronization, and *wait*, *notify* for coordination. Their work was considered fruitful on a topic if their source code had the associated keyword and their feedback files showed a non-zero score for that topic. The fail ratio is the ratio of unfruitful attempts to total attempts. The last column of Table 2 gives the fail ratio for each concurrency topic each year.

The numbers are likely higher than actual fail ratios as our algorithm can label a student who has not tried as having tried. For instance, in all assignments that contributed to the thread fail ratio, the *sleep* call occurred in unused code unrelated to thread creation that had previously created an animation in the main thread. However, our classification algorithm used *sleep* as a keyword to determine if the student had tried to create threads, and thus wrongly misclassified these as failed attempts.

The fail ratios for synchronization are consistently high in all years. Unlike thread creation and coordination, synchronization is specified declaratively in Java by simply putting the **synchronized** keyword in a method declaration. One explanation for the result is that declarative mechanisms are easy to program but difficult to debug. Usually, students sorted out insurmountable debugging problems with help from instructors. However, the concurrency assignments could be submitted until the last day of classes, and, when they were optional, with no penalty. So it is possible that students who failed the synchronization tests worked at the last minute and thus did not have time to visit instructors. The "gotchas" reported in Figure 28 give some of the concrete reasons for these failures.

4.2 Test Attempts and Surmountable Difficulties (CS-1+ 2016-2018)

Test results from submitted code and fail ratios do not indicate all of the struggle students face with a task as students with the same assignment scores can struggle to a different extent. Arguably, struggle is also measured by the average number of times students ran a test to improve its score, which we refer to as the number of test attempts. Data logs from student tests, introduced in 2016, allowed us to make an inference of this number. Each time a student runs one or more tests, the logs

```
Read student's log data
AttemptCount ← 0
For each line in student's log
   AttemptCounter++
   If a test has moved into passing or partial
        Find all tests that have moved to passing or partial
        ProportionalAttempts ← AttemptCounter/Size of Moved Tests
        For each test that moved to passing or partial
                Attempt on Test += ProportionalAttempts
        AttemptCounter ← 0
```

Figure 25 Inferencing Attempts Per Test indicate the transitions made by the executed tests between four states: not tested, failed, partially passed, and passed. We used the algorithm sketched in Figure 25 to infer the number of attempts on each test made by each student. The algorithm keeps track of the number of tests executed between two positive runs, that is, runs in which an improvement to some test status occurs. It also keeps track of which tests are affected positively by a run. It evenly divides all of the runs since the last positive run to the newly improved tests.

Table 3 gives failed and partial pass data about the first 25 test runs of a student. We illustrate our algorithm using this trace. The first two runs cause no changes to the tests' state. On the third run, *AssertingBridgeSceneDynamicTestCase* moves from the failed category to partial passed. Thus, this test's attempt counter becomes 3 – the assumption is that the student was working on this test in this run and the two unsuccessful runs before it when no positive score transition occurred. The next three runs are unsuccessful, and on the fourth one, *AsyncRobinAnimationTestCase*, moves from failed to partial pass. Hence its attempt counter is set to 4. The next run is successful, moving *AsyncLancelotAnimationTestCase* from failed to partial pass. Its attempt counter is incremented by 1. The next run is again unsuccessful, and the one after that moves two tests, *AsyncGalahadAnimationTestCase* and *AsyncArthurAnimationTestCase* from failed to partial pass. The previous run and this run are now proportionally allocated to both of these tests, so the attempts counter for each is incremented by 1. The next run moves all asynchronous animation tests to failed, and so is a failed run. No progress is made in the next twelve runs. On the run after that, some new tests (not identified here) move from failed to pass. The thirteen runs since the last successful one are evenly divided among these tests. The run after that moves, again, the four asynchronous animation tests from failed to partial pass. Therefore, the attempts counter for each of these four tests is incremented by 0.25.

Table 3 Partial Pass and Failed Status from Example Student Test-Log

Run#	Partially Passed	Failed
3	AssertingBridgeSceneDynamicTestCase+	AsyncRobinAnimationTestCase, AsyncLancelotAnimationTestCase, AsyncArthurAnimationTestCase AsyncGalahadAnimationTestCase
7	AssertingBridgeSceneDynamicTestCase, AsyncRobinAnimationTestCase+	AsyncLancelotAnimationTestCase, AsyncArthurAnimationTestCase AsyncGalahadAnimationTestCase
8	AssertingBridgeSceneDynamicTestCase, AsyncRobinAnimationTestCase, AsyncLancelotAnimationTestCase+	AsyncArthurAnimationTestCase AsyncGalahadAnimationTestCase
10	AssertingBridgeSceneDynamicTestCase, AsyncRobinAnimationTestCase, AsyncLancelotAnimationTestCase, AsyncArthurAnimationTestCase+ AsyncGalahadAnimationTestCase+	
11	AssertingBridgeSceneDynamicTestCase	AsyncRobinAnimationTestCase-, AsyncLancelotAnimationTestCase-, AsyncArthurAnimationTestCase- AsyncGalahadAnimationTestCase-
24	AssertingBridgeSceneDynamicTestCase,	
25	AssertingBridgeSceneDynamicTestCase, AsyncRobinAnimationTestCase+, AsyncLancelotAnimationTestCase+, AsyncArthurAnimationTestCase+ AsyncGalahadAnimationTestCase+	

The fact that multiple tests could be executed in one run makes this algorithm error-prone as we do not really know which tests were being targeted by the student. A further threat to the validity of our metric is that if a student runs a test individually, the run is not recorded (due to implementation reasons), so we could be undercounting. Our conclusions from this metric assume that these factors balance out in our comparisons – that is, errors due to proportional allocation of runs and undercounting do not make a difference to the comparison of the aggregate numbers.

We classified each test in assignments A10-12 into one of four categories: threads, synchronization, coordination, and other. For each category, we found the average number of attempts on tests that passed fully or partially. The "other" category was used as a baseline. In each pairwise comparison, we only considered those students who had non-zero attempts in both of the compared categories.

Table 4 Number of Test Attempts

Comparison	Year	Num. Students	P-Value	Means
Other vs Threads	**2016**	**19**	**0.0158**	**3.13, 1.82, -1.31**
	2017	**16**	**0.013**	**2.52, 1.68, -0.8**
	2018	9	0.173	2.33, 1.79, -0.54
Other vs Synchro-nization	2016	19	0.601	3.13, 3.05, -0.08
	2017	16	1	2.52, 2.30, -0.22
	2018	9	0.085	2.33, 1.46, -0.87
Other vs Coordination	2016	19	0.904	3.13, 3.17, +0.04
	2017	16	0.379	2.52, 2.44, -0.08
	2018	9	0.953	2.33, 2.70, +0.37

Table 4 shows our results. We had test logs from only those students who chose to run them locally (some students relied completely on servers or on their own tests) and submit them with their code. We found that the only significant difference in the number of test attempts was between other and threads in the years 2016 and 2017, with threads having a much smaller number. This is consistent with the fact that other concurrency topics are layered on top of it and thus, it is arguably the simplest of these topics. The small number of students with logs in 2018 probably accounts for the fact that the mean difference for threads is not significant that year.

Figure 26 shows the distribution of the number of attempts in the four compared categories when the data for all three years are combined. Not surprisingly, this distribution is wide. The figure graphically shows that (a) in general, the attempt numbers for threads were relatively small compared to other concurrency and non-concurrency topics, but (b) that concurrency topics resulted in more high numbers (> 4), confirming the difficulty of debugging concurrent programs [17, 19, 20]. The fact that the means for concurrency topics were not larger, and in the case of threads, smaller, than non-concurrency topics is a strong argument for the principle of creating simulation-based projects that make many concurrency bugs apparent in the user-interface.

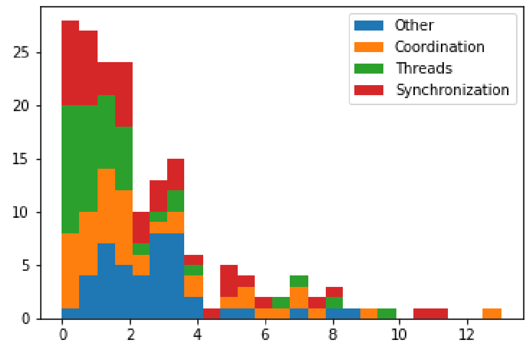


Figure 26 Distribution of Test Attempts, 2016-2018

4.3 Piazza and Insurmountable Difficulties (CS-1 + 2012-2018)

Not all bugs, of course, can be solved by students on their own, as illustrated by Figure 27, which shows an excerpt of a long discussion on coordination. In the first message, Student A describes the problem of not being able to start all waiting avatars simultaneously. The middle message is an example of one of the several messages Student B sent to try and help solve the problem. The last message describes the problem Student A discovers – doing a *notify()* rather than a *notifyAll()*, and thanks student B for his attempts to help.

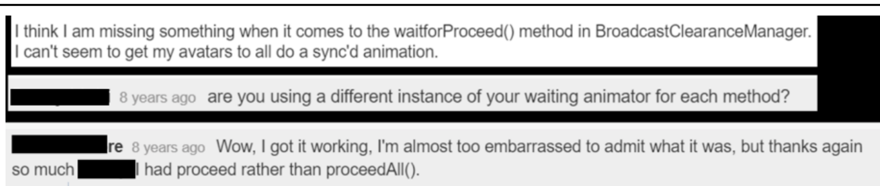


Figure 27 Flavor of Insurmountable Problem

Such online discussions have been shown to correlate with engagement, learning, and struggle [32-37]. Online discussions can be divided into structured discussions, in which instructors give students topics to discuss, and unstructured discussions, in which students start the thread [35]. Both kinds of discussions help students engage and actively think about the course [35]. We will refer to a discussion started with a student question as a question thread.

As the example above illustrates, (a) the number of question threads on a topic is an indicator of topic struggle (b) and the number of messages in a question thread is an indicator of topic engagement and learning. Piazza refers to messages that initiate a discussion thread as a post and all other messages (posts, responses, edits, follow-ups, and comments to follow-ups) as contributions. Unstructured student discussions in our courses almost always started with questions. Thus, the two metrics above are equivalent to (a) the number of posts per topic, and (b) the number of contributions per post. We determined these metrics for the baseline topic of exceptions and the three concurrency topics.

For each topic, we identified a set of keywords that was a superset of the one used in the source-code classification scheme. We used the NLP library, Spacy, to tokenize each Piazza post and contribution into individual words with accompanying semantic information. If a word was not a verb, noun, or proper noun, we threw it out because it was unlikely to be related to concurrency. We looked at the lemma (root) of each word and added the lemma to the set of meaningful words if it had not yet appeared in the body of the posts. For example, under this scheme, "threading" and "thread" will derive to the same lemma and count as the same word. If a meaningful word was in a set of keywords associated with a topic, then we said that the entire post was about that topic, otherwise, we classified it as *other*. If keywords for multiple topics appeared in it, then the post was categorized as the topic whose keyword was searched first. Consider the following example based on the fact the keyword **thread** was associated with threads and **synchronized** with synchronization. A post containing the sentence "The threads were not synchronized," would be incorrectly put in the threads category if the **thread** keyword was searched first. The keyword-based scheme also does not take context into account. For example, the sentence "look at the thread in @235 to solve the exception problem" would be classified in the thread category.

To determine the extent of such errors, we manually labeled all posts in the fall 2018 offering, and used it as ground truth to evaluate our classification scheme. These posts included both questions and student My Q/A and Class Q/A diaries, discussed later. Our evaluation of this scheme showed that its F-1 score was high (0.94). Therefore, we used the automatic classification on all of our course offerings since 2012 (when we started using Piazza) to determine the number of posts and contributions a student made relating to exceptions and each concurrency topic that had the Piazza tag/folder of A10, A11, and A12. Table 5 shows our results. We used a Web API to gather student posts and contributions. These data do not include how many times a post or contribution was read and thus did not count the large number of lurkers who never posted.

Table 5 2012-2018 Piazza Data Analysis

Topic	Posts	Contri-butions	Thread length
Exceptions	9	32	3.56
Threads	41	136	3.31
Synchronization	3	11	3.7
Coordination	15	90	6
Other	156	501	3.21

The table shows that for all topics, few insurmountable problems were posted on Piazza. This is not to say that the impact of their discussions was low, as it is common for multiple students to have the same insurmountable problem. Thus, the number is more an indication of how many different kinds of difficulties (that could be explained in a forum) occurred rather than how many students had insurmountable difficulties. Even, so the number seems intuitively low. This is partly by design. To reduce this number, students who had faced difficulties and instructors who had addressed them were encouraged to contribute to a list of "gotchas" in the assignment write-up stored in a Google document commentable by the whole class, where each item in the list was a symptom of a problem and its solution. Figure 28 shows some of the concurrency items in the fall 2018 A11 write-up. This list contained the contributions of students not only in the

Potential Gotchas

1. For the synchronize test suite, make sure that the animation was finished in *5000* milliseconds, so that two animations could be done in required 5 seconds. Otherwise, local check may give an error.
2. If you get the message Child 1 and Child 2 are not synchronized, it means two animations of the same avatar do not occur serially. This may happen if you (a) create a single animator for each avatar but do not use the synchronize keyword for the animation method or, (b) create multiple animators for each avatar.

Figure 28 Example Problems in Publically-Commentable Assignments Descriptions current offerings but also previous ones, and students were told to consult this list when they faced a problem.

Interestingly, though students were more successful with threads than other concurrency and non-concurrency topics in terms of the number of attempts, Table 5 shows that they made more posts on threads. One explanation is that insurmountable problems with threads were sorted out more by Piazza posts rather than interaction in office hours. The table also shows that coordination had the longest discussions, which is consistent with (a) the fact that it layered on top of other concurrency topics, (b) the intuition that it is full of subtleties, hence mistakes in it are hard to find, and (c) some of the large attempt numbers for it shown in Figure 26.

If some topics indeed required more posts and contributions, then it is useful to compare a student's fraction of the total posts and contributions on different topics, which assesses if a topic causes a change in student's discussion behavior. Using exceptions as a baseline, we found no statistically significant difference in post and contribution fractions when comparing each pair of topics. Typically, the articulate students post their insurmountable problems on Piazza. The others pay visits to instructors during office hours (if the problem has not been addressed already in Piazza or the "gotcha" list). Assuming the number of insurmountable problems per student in the same for both groups, our topic-based comparisons of Piazza posts may also apply to office visits, for which we have no data.

4.4 Exam Scores (CS-1+ 2012, 2015, 2016)

Scores on written exams (as opposed to multiple-choice quizzes) are perhaps the most standard mechanism for evaluating teaching approaches. By comparing scores on various concurrency topics with the average exam score or scores on other topics, we can obtain valuable data to evaluate the learning and struggle matching requirements. However, considering one particular exam in isolation does not provide a complete comparison, for two reasons. First, because of the limited amount of time available for a written exam, it cannot cover all course topics. Second, it is possible to ask questions of varying difficulty on any topic, and exams are designed to include questions of varying difficulty. Thus, it is unfair to compare the scores on an easy question on a topic with the scores on a hard question on another topic. Therefore, it is important to (a) consider exam data for all years together and (b) interpret these data relative to the kind of questions asked and the amount of time available.

All of our written exams were closed book and no collaboration was allowed. We have not kept exam data for our CS-1 offerings. We also do not have relevant exam data for some of our CS-1+ offerings. As assignments on concurrency were optional in 2017 and 2018, exams of these offerings did not include questions on it. The 2013 offering asked a concurrency question regarding the Java user-interface thread, which required knowledge of both concurrency and the Java AWT/Swing toolkit. As it was not a pure concurrency question, we ignored this exam.

Therefore, we consider scores on exams in the 2012, 2015 and 2016 offerings of CS-1+. The other topics included in the exams varied and included the factory, observer, and MVC design patterns, exceptions, assertions, preconditions, generics, and recursive-descent parsing. In 2012, the concurrency question was asked in a 75-minute midterm exam, while in 2015 and 2016, it was asked in a 3-hour final. The 2016 concurrency question required students to know the Java *Runnable* interface, implement a correct implementation of it, and use this implementation to correctly start a thread from the main program. The 2015 concurrency question gave students the code for a multi-threaded program. They were required to give two different possible results from the executions of the threads. The 2012 question also involved giving the output of instructor-defined multi-threaded code, which involved the use of Java **synchronized** keyword and the Java *sleep(), wait()*, and *notify()* calls. To get regular credit on the question, students were required to understand that (a) when one thread sleeps, other threads are scheduled, (b) a *notify()* on object *A* unblocks a *wait()* on *A*, and (c) a parent thread that starts children threads can terminate before the children threads. To get extra credit, they had to understand that if a thread executing a synchronized method in object *A* executes a *wait()* in a synchronized method in object *B*, the thread does not give up the synchronization lock on *A*, preventing a second thread from executing a synchronized method in *A* to unblock the first thread, resulting in a deadlock. Students had not been taught deadlock, and were expected to identify it from first principles, based on their knowledge of queue entry and exit semantics; so this question was made extra credit. None of these questions involved the creation or understanding of simulations or the use of any of the design and loop patterns used in the worked examples and assignments. Thus, they tested students' ability to apply the concurrency concepts in new patterns.

The 2015 and 2016 final exams were scanned and uploaded to Gradescope, which maintains scores on each question. We have total scores also from all (82) of the 2012 midterm exams, and it is their distribution that was displayed in Figure 24(a). Only 14 of the 82 2012 midterms were not collected by students; we have graded paper copies of only the uncollected exams. Table 6 shows the data from these exams. The 2012 row has a sub-row for the unreturned copies and one for the entire class. As we see, the averages of the two groups are close to each other.

Table 6 Exam Score Comparisons

	Topics (Points in Exam)	Number of Students	Concurrency Score	Non-Concurrency Score	Total Score	Concurrency vs Non-Concurrency: P-Value	Concurrency vs Total: P-Value
2012	Other Topics (60), Synchronization (2), Coordination (13), Deadlock (5)	14 82	77.6%	57.3% (-20.3%)	61.1% 57.8%	0.885	0.799
2015	Other Topics (125), Thread Creation and Scheduling (5)	95	90.1%	63.1% (-27.0%)	56.7%	**2.10e-15**	4.73e-20
2016	Other Topics (133), Thread Creation (17)	65	74.6%	63.9% (-10.7%)	64.9%	**5.87e-4**	3.93e-4

P-Value comparison shows that the concurrency vs non-concurrency differences were significant in 2015 and 2016 but not 2012. Arguably, the 2015 and 2016 concurrency questions were simpler than the 2012 question, which accounts for the fact that scores on them were better than those on the (a) non-concurrency questions given the same year, and (b) concurrency question given in 2012. Interestingly, even the scores on the difficult concurrency question of 2012 are better than those on the non-concurrency questions given the same year, which may be because some students were able to amplify their concurrency scores by up to 33.3% with answers to the extra credit deadlock question. Assuming students would allocate time to questions based on their points, students had 15 minutes for this complex question. 2012 was the first year coordination was introduced in CS-1+. Performance on this difficult concurrency question was a strong motivation for continuing with all three aspects of concurrency in the subsequent years.

Scores on closed-book, non-collaborative high-stake exams gauge not only learning but also the (a) ability to perform under pressure, and (b) alignment between the topics of the questions and the ones studied by the students. It was probably to avoid the risks posed by this pressure and alignment that motivated many to do the extra-credit assignment work in the course. Below, we discuss scores from two lower stake mechanisms for gauging learning based on instructor questions: quizzes and Q/A diaries.

4.5 Quiz Scores (CS-1+ 2016 - 2018)

As mentioned earlier, starting in 2016, topics covered in the course were associated with in-class hands-on exercises and open-book quizzes. These exercises involved experimentation with the worked examples associated with the topic. The concurrency exercises involved modifying the worked shuttle examples in prescribed ways and viewing and understanding the results of these modifications. The quizzes were deliberately low-stake to encourage collaboration and reduce stress. Unlike exams, they were comprehensive, covering most aspects of a topic. While they were open book and collaborative and hence right-skewed (Figure 24(b)), they were challenging, as shown by Figure 24(b) and the average scores reported in Table 7. Thus, these scores provide an important metric for determining the learning and struggle associated with the topic.

Our three concurrency topics were introduced in the last phase of the semester and, as mentioned above, exercised in the last three assignments. In the same phase and assignments, the topic of exceptions was introduced and exercised, respectively. Therefore, it was considered the base case in our quiz comparison. It was compared with quizzes on each of the concurrency topics: threads, synchronization, and coordination. When comparing two quizzes, we considered only students who received a non-zero score on *both* quizzes; the others were assumed to have not taken the quiz.

Overall, the mean differences in Table 7 are either (a) not significant or (b) small and not consistent over the years. The only significant differences are in the 2017 thread row and the 2018 coordination row, with the concurrency topics having higher means than the baseline. The 2018 result is surprising as intuition says that coordination is a more difficult topic than exceptions, which even CS-1 students are expected to understand to some extent. Figure 29 is an example of the difficulty and student performance of a coordination quiz question involving the relationship between coordination and synchronization. The high scores on it are particularly remarkable given that in 2018, assignments on concurrency were optional, and coordination was covered in the very last of these assignments. One explanation is that the visual and real-world nature of the worked examples used in the hands-on exercises made it easier to learn the basics of concurrency concepts than concepts, such as exceptions or recursion, that do not have ready real-world or visual manifestations.

Table 7 Quiz Score Comparisons

Comparison	Year	Number Students	P-Value	Means
Quiz Grade Scores Exceptions vs Threads	2016	57	0.091	75.1%, 81.8%,+6.7%
	2017	**76**	**0.023**	**69.2%, 74.5%, +5.3%**
	2018	67	0.349	77.2%, 76.7%, -0.5%
Quiz Grade Scores Exceptions vs Synchronization	2016	62	0.117	69.0%, 78.8%, +9.8%
	2017	71	0.159	74.0%, 66.9%, -7.1%
	2018	65	0.376	79.6%, 79.3%, -0.3%
Quiz Grade Scores Exceptions vs Coordination	2016	59	0.667	72.6%, 77.3%, +4.7%
	2017	-	-	-
	2018	**66**	**0.014**	**78.4%, 81.0%, +2.6%**

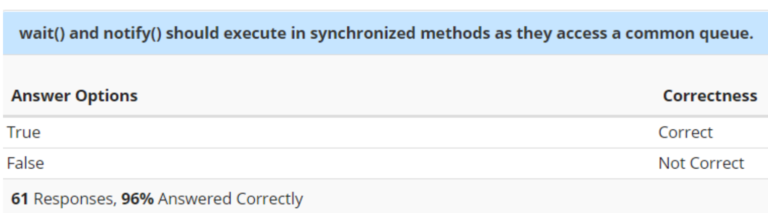


Figure 29 Flavor of Quiz Question and Student Performance

4.6 Analysis of Q/A Diaries (CS-1+ 2018)

In 2018, the instructor motivated the topic in class, assigned a praxis, and then asked for explanation-based answers after students submitted the quiz associated with the praxis. For extra credit, students could post private diaries of their interaction on Piazza by making two kinds of entries in these posts:

- My Q/A: For a question posed in class that they answered, wrongly or correctly, they could note the question and the answer they gave.
- Class Q/A: For each question posed, they could note the question and a correct answer.

Figure 30 gives the flavor of the Q/A for our three concurrency topics, threads, synchronization, and coordination, illustrating the breadth and depth of the posed questions and the internalization of the answers by the student.

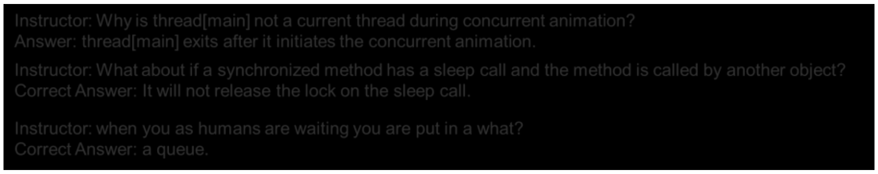


Figure 30 Flavor of Recorded Class Q/A

My Q/A diaries provide data about class participation [38], an important means for active learning and student engagement [39, 40], to promote which special tools and techniques [39, 40] have been researched. As mentioned in [41], its purpose can be to engage students with new material or material they have reviewed earlier. The post-quiz class participation of fall 2018 involved material review. Class Q/A diaries provide data about student note-taking, another important means for learning and engagement [42] shown to correlate with test performance [43], to promote which special tools and techniques have also been developed [43]. Thus, as the example in Figure 30 illustrates, (a) answering an in-class question indicates engagement, and answering it partly correctly also indicates learning, (b) not answering a question can indicate lack of engagement or struggle in understanding the question or formulating the answer, and (c) taking class notes on question and answer interactions indicates both learning and engagement.

Therefore, we computed metrics based on these diaries to evaluate the learning, struggle, and engagement matching requirements. We picked two dates when concurrency (all three subtopics) and exceptions, respectively, were the focus. For each day, we determined the maximum number of Class Q/A entries and My Q/A entries made by any student. To account for absences, we considered only students who posted both days. We used this value to normalize the number of entries of other students on that day – if a student made N entries and the maximum was M, then the normalized fraction assigned to that student was N/M. This was consistent with the way these diaries were graded. We used exceptions as a baseline. Table 8 shows that we found no statistical difference between the two compared topics. In this analysis, dividing concurrency into subtopics was not practical because of the small number of questions asked on each subtopic.

Table 8: Diary entries per student: Exceptions vs Concurrency

Diary	Number Students	P-Value	Means (Exceptions vs Concurrency
My Q/A	6	0.752	0.43, 0.35, -0.08
Class Q/A	21	0.508	0.39, 0.46, +0.07

5. Evaluation

The design principles and evaluation data presented above can be used to determine how well our approach meets our requirements.

5.1 Evaluation based on Principles

Let us first consider the requirements met to some extent by the design principles. Making simulations of moving objects a common theme for analogies, worked examples, and assignments meets the driving-problem matching requirement. As mentioned before, providing predefined scaffolding user-interface code dependent on a particular functional component would not have met the extendibility requirement. Implementing the user-interface components of these simulations either manually using the MVC design pattern or automatically using a user-interface generator allows the functional components of these user-interfaces to be changed, meeting the extendibility requirement. As mentioned above, these components were extended for no credit both in the CS-1 and CS-1+ offerings.

Amplification of the general subject of object-oriented programming is met by using a general design pattern for implementing concurrency aspects of the simulations. Amplification of the specific concept of command objects is met by basing the design pattern on this concept.

Conceptual focus is met by creating simulations whose user-interfaces can be generated using an available user-interface generator. If the user-interfaces are implemented manually, it is not out of necessity but to help students learn MVC. Requiring students to implement user-interface details only when they contributed to learning meets the conceptual focus requirement.

5.2 Evaluation based on Data Analysis

Our metrics-based analyses compare the concurrency topics with either the topic of exceptions or all other topics for which we have data. The data show little or no significant difference in some or all years, indicating that the concurrency topics fit seamlessly in the existing course.

6. Discussion

6.1 Ideas in Existing Work

Many aspects of our approach borrow ideas from previous work. Like us, others have added concurrency to existing courses (which include introduction to programming [7, 44], data structures [9, 10], and upper-level courses [45, 46]).

A course on parallelism must introduce the notion of concurrency and optionally abstractions for implementing it. At least one CS-0 course has focused only on introducing concurrency [1]. Presenting and explaining the behavior of the worked examples without the implementations, which is supported by our approach, is in the spirit of such an introduction. The use of the ObjectEditor-based declarative abstraction in CS-1 is in the spirit of concurrency courses that cover high-level/declarative abstractions such as message-passing [7], and fork-join and reduce [8, 10]. Use of Java procedural abstractions in CS-1+ is in the spirit of previous works that provide procedural thread-creation [9, 44].

A related issue is the programming language in which these abstractions are supported. These languages have included a visual programming language called Scratch [7], C [4, 10], Python [7, 8], and, as in our work, Java [9, 44].

There are two complementary reasons for making a program concurrent. The first is to make an algorithm faster but not easier to program by executing parallelized tasks concurrently. The second is to make an algorithm easier to program but not faster. In the second approach, the automatic context-switching offered by concurrency abstractions relieves the programmer from manually implementing context-switching. Most of the surveyed work has focused on performance, discussing parallelization of a variety of tasks such as scanning [8], sorting [8, 9], Monte Carlo Pi estimation [4], image processing [1, 4], printing [7], rendering [46] and computing heat flow diffusion and airflow [45], that can execute faster when multiple cores/processors are available. Arguably, this parallelization makes programming of these tasks more difficult as it requires the additional steps of thread creation and typically, synchronization and coordination. A striking example of the alternative approach (of not focusing on performance) is using parallel communicating threads to make it easier to program a person moving with a ball [7] – the person and ball can be represented by separate threads that communicate with each other. As these two threads are closely coupled and together represent one autonomous activity, parallelization would reduce performance due to the overhead of thread creation, context switching, and message passing. Other examples have taken a hybrid approach of focusing on both performance and programmability by targeting multiple autonomous activities such as the Conway Game of Life [4, 47], the Frogger™, Tetris and Centipede games, multiple independent ATM withdrawals [4, 44], and a single server connected to multiple clients [48]. Parallelization both saves programmers from implementing context-switching of such activities and increases performance when multiple cores are available. Our work falls in the hybrid camp,

Yet another aspect of teaching concurrency is the learning mode used by students to understand the concepts. The traditional approach is to use only lectures, live or recorded. There has previously been some work on hands-on introduction to concurrency [7] [1]. As mentioned above, we incorporated a hands-on approach in 2015.

Assessment of student assignments is an important issue in any course. Like [49], our offerings support automatic checking of concurrency requirements of programs.

A variety of metrics have been used to evaluate course effectiveness. The objective/quantitative metrics include course and lab sizes [1], and comparison of the average scores on concurrency labs and all labs [1]. The subjective/qualitative metrics include student assessment of learning gains [1, 12, 46], satisfaction [6], difficulty [44], engagement [44], and the years in the curriculum in which concurrency should be addressed [4]. Our work provides quantitative metrics that extend the average score comparisons. The course sizes are not relevant for our courses as they were required for a large subset of the students who took them.

6.2 Contributions

Requirements: While previous work has addressed the addition of concurrency in an existing course, to the best of our knowledge, this is the first paper that explicitly articulates a set of requirements that such an effort should meet. These requirements are abstract enough to be independent of the topics of the course being extended and how concurrency is added to it.

Some of them have been implicitly followed by one or more of the surveyed works. Their explicit articulation leads to (a) understanding of pedagogical approaches based on which subset of these requirements were considered by the approaches, (b) motivation for tools and techniques for satisfying one or more of these requirements, and (c) metrics to evaluate how well they are met. Based on our understanding of previous work on extending existing courses, learning, struggle and engagement matching, topic amplification, and conceptual focus have been common implicit requirements of this work. Parallelization of single-threaded algorithms was a common technique to provide topic amplification. We did not see in previous work any special tool invented to meet any of these requirements, nor did we see previous work that met the driving problem matching or extendible implementation requirements.

Metrics: We have also not seen in previous work the metrics used here to determine how well these requirements are met. These metrics are also abstract enough to be independent of the topics of the course being extended and how concurrency is added to it. Using inter-topic p-values to determine if concurrency had a significant effect on student scores in assignments, quizzes, and exams is a generalization of the idea in [1] of comparing concurrency lab averages with overall lab averages. Using inter-topic p-values to determine if concurrency had a significant effect on the number of student posts and contributions in discussion forums is a generalization of counting the total number of posts and contributions in two different offerings of a course on parallel and distributed computing [6]. The fail ratio and number of test attempts and in-class questions answered and recorded do not have analogs in previous metrics.

Inference Algorithms: Associated with these metrics are three new data mining algorithms for inferring them, which include the algorithm for determining (a) the number of test attempts when multiple tests pass together, (b) identifying unfruitful attempts from source code, (c) identifying the topic of a post and contribution when the associated tags are not fine-grained. While developed for our approach to concurrency, they apply to any course.

Principles and Patterns: Our innovation with a more narrow application is the set of principles for extending existing courses on object-based programming. Associated with these principles are the design and loop patterns we have identified. Our loop patterns apply to any procedural language, while the design patterns apply only to object-based languages.

Design of specific analogy-based worked examples and assignments: The innovations with the narrowest applications are the specific worked examples and assignments developed to meet these principles: the shuttle-based worked examples and the assignments simulating a rabbit crossing the road, Halloween candy foraging, and adaptations of scenes from two movies. They show that the principles can be translated into a variety of concrete designs.

Evaluation: The evaluation of the applications of these three principles (in our various offerings) showing that the requirements are met is an important contribution given the breadth of the requirements and metrics and breadth and depth of the covered concepts, assignments, quizzes, class interaction, and exams.

Experience: To help interpret and understand our results and give the reader a concrete feel for the student experience, we have given a flavor of exam and quiz distributions (Figure 24), student-invented concurrent simulations (Figures 20-23), the tests used to evaluate assignments (Figure 16), the attempts students made to overcome surmountable problems (Table 3), insurmountable problems (Figure 27) and "gotchas" (Figure 28) posted on Piazza and assignment descriptions, respectively, exam (Table 6) and quiz (Figure 29) questions, and class interaction, as recorded by students (Figure 30). The reported experience also includes the (a) assignment overload mistake we made, (b) how we fixed it with extra credit, and (c) based on our offerings, the incorporation of concurrency as a required element of the new incarnation of the Foundation of Programming course, which as mentioned above, would be taken after the data structures course and should be able to accommodate required concurrency assignments.

Taxonomy: Our final contribution is a new classification of previous work on extending existing courses with concurrency based (a) the subject of the extended course, (b) the concurrency abstractions introduced, (c) the programming language used, (d) whether performance or programmability was the motivation for introducing concurrency, (e) the nature of the worked examples and assignments, (f) whether the offering included hands-on learning and/or automatic grading, and (g) the metrics used to evaluate the offering. While the primary purpose of the taxonomy was to identify the bases and innovations of our work, arguably, like any new taxonomy, it increases understanding of the field.

6.3 Directions for Future work

A large variety of new directions are opened by this work, which include:

- *Additional requirements* based on implicit requirements in this and other work, (b) the use of existing metrics and associated algorithms in other concurrency-based course offerings collecting similar data, which can result in new evaluations and refinements of the associated algorithms.
- *Improvements to our algorithms* for inferring the topic addressed by natural language posts and source code.
- *New metrics* for determining if these requirements are met based on both data used here and data not considered here such as GitHub commits, chat logs and audio transcripts collected in Zoom sessions, the amount of peer interaction in collaborative work, and office-hour visits [50, 51].

- Use of our worked examples, assignments, and the clearance manager tool in courses on Java-based programming covering a *different a set of topics* and *targeting different kinds of student populations.* A specific example of such a course is our replacement of the CS-1+ course, which assumes an object-based introduction to programming and data structures.
- Implementations of the examples, assignments and tools in other object-based programming languages such as *Python* and *Typescript* – two languages that have also been used to teach CS-1 in our department.
- *New worked examples and assignments* to increase engagement/extensions and reduce plagiarism, based perhaps on cultural events other than “trick or treat”, popular games other than the rabbit crossing the highway, and popular movie scenes other than the two we have considered.
- *New principles* for meeting our requirements in extensions of courses on a variety of topics such as data science, object-based programming, and data structures;
- Evaluating the impact of *eliminating some of our requirements* such as the driving problem matching and extendible implementations requirements. For example, in a C course introducing concurrency using OpenMP, our driving example could still be used, even though it would not match other worked examples and assignments.
- *Additional dimensions* in our taxonomies such as the nature of hands-on or active learning in class and the length and nature of recorded videos available to students.

This paper provides the bases for pursuing these directions. Lecture recordings, assignment descriptions, quizzes, and source code of hands-on exercises, worked examples, automatic tests, and our inferencing algorithms are available on request.

Acknowledgments
This work was funded in part by NSF award OAC-1924059.

References
1. Ghafoor, S., D.W. Brown, and M. Rogers. *Integrating Parallel Computing in Introductory Programming Classes: An Experience and Lesson Learned*. in *Proceedings of the Euro-EDUPAR 2017 workshop of 23rd International European Conference on Parallel and Distributed Computing*. 2017.
2. Prasad, S.K., A. Gupta, A. Rosenberg, A. Sussman, and C. Weems. *CDER Center | NSF/IEEE-TCPP Curriculum Initiative," NSF/IEEE-TCPP Curriculum Initiative.* 2017; Available from: https://grid.cs.gsu.edu/~tcpp/curriculum/?q=node/21183.
3. Prasad, S.K., A. Gupta, A.L. Rosenberg, A. Sussman, and C.C. Weems, *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses*. 2015: Morgan Kaufmann Publishers Inc. 360.
4. Ko, Y., B. Burgstaller, and B. Scholz, *Parallel from the beginning: the case for multicore programming in thecomputer science undergraduate curriculum*, in *Proceeding of the 44th ACM technical symposium on Computer science education*. 2013, ACM: Denver, Colorado, USA. p. 415-420.
5. Grossman, M., M. Aziz, H. Chi, A. Tibrewal, S. Imam, and V. Sarkar, *Pedagogy and tools for teaching parallel computing at the sophomore undergraduate level.* J. Parallel Distrib. Comput., 2017. **105**(C): p. 18-30.
6. Malakar, P. *Experiences of Teaching Parallel Computing to Undergraduates and Post-graduates*. in *Proc. of EduHiPC-19 worksshop in IEEE HiPC*. 2019. Hyderabad: IEEE.

7. Bogaerts, S., *Hands-on Parallelism with no Prerequisites and Little Time Using Scratch*, in *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses, 1st Edition*, S. Prasad, A. Gupta, A. Rosenberg, A. Sussman, and C. Weems, Editor. 2019, Morgan Kaufmann.
8. Cormen, T.H., *Parallel Computing in a Python-Based Computer Science Course*, in *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses, 1st Edition*, S. Prasad, A. Gupta, A. Rosenberg, A. Sussman, and C. Weems, Editor. 2019, Morgan Kaufmann.
9. Grossman, D., *Fork-Join Parallelism with a Data-Structures Focus*, in *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses, 1st Edition*, S. Prasad, A. Gupta, A. Rosenberg, A. Sussman, and C. Weems, Editor. 2019, Morgan Kaufmann.
10. Adams, J.C., *Injecting parallel computing into CS2*, in *Proceedings of the 45th ACM technical symposium on Computer science education*. 2014, ACM: Atlanta, Georgia, USA. p. 277-282.
11. Bunde, D.P., *Modules for Introducing Threads*, in *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses, 1st Edition*, S. Prasad, A. Gupta, A. Rosenberg, A. Sussman, and C. Weems, Editor. 2019, Morgan Kaufmann.
12. Burtscher, M., W. Peng, A. Qasem, H. Shi, D. Tamir, and H. Thiry, *A Module-based Approach to Adopting the 2013 ACM Curricular Recommendations on Parallel Computing*, in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. 2015, ACM: Kansas City, Missouri, USA. p. 36-41.
13. Wegner, P. *Dimensions of Object-Based Language Design*. in *OOPSLA '87 Proceedings*. October 1987.
14. Dewan, P. *Visually Introducing Freshmen to Low-Level Java Abstractions for Creating, Synchronizing and Coordinating Threads*. in *Proc. of EduHiPC-19 worksshop in IEEE HiPC*. 2019. Hyderabad: IEEE.
15. Brown, R. and E. Shoop, *CSinParallel and Synergy for Rapid Incremental Addition of PDC Into CS Curricula*, in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. 2012, IEEE Computer Society. p. 1329-1334.
16. Brown, R. and E. Shoop, *Modules in community: injecting more parallelism into computer science curricula*, in *Proceedings of the 42nd ACM technical symposium on Computer science education*. 2011, ACM: Dallas, TX, USA. p. 447-452.
17. Brito, M.A.S., K.R. Felizardo, P.S.L. Souza, and S.R.S. Souza, *Concurrent Software Testing: A Systematic Review*, in *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*. 2011.
18. McCauley, R., S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander, *Debugging: A Review of the Literature from an Educational Perspective.* Computer Science Education, 2008: p. 67-92.
19. McDowell, C.E. and D.P. Helmbold, *Debugging concurrent programs.* ACM Computing Surveys (CSUR), 1989. **21**(4): p. 593-622.
20. Chen, J. and S. MacDonald, *Towards a better collaboration of static and dynamic analyses for testing concurrent programs*, in *Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging*. 2008, ACM: Seattle, Washington. p. 1-9.
21. Wallingford, E., *Using patterns in the CS curriculum.* Journal of Computing Sciences in Colleges, 2000. **15**(5): p. 235-237.
22. Astrachan, O. and E. Wallingford. *Loop Patterns*. in *PLoP*. 1998.
23. Myers, B.A. and M.B. Rosson. *Survey on User Interface Programming*. in *Proceedings SIGCHI'92: Human Factors in Computing Systems*. 1992.
24. Dewan, P. *How a Language-based GUI Generator Can Influence the Teaching of Object-Oriented Programming*. in *Proc. ACM SIGCSE*. 2012.
25. Krasner, G.E. and S.T. Pope, *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80.* Journal of Object-Oriented Programming, 1988. **1**(3): p. 26-49.
26. Dewan, P. *Teaching Inter-Object Design Patterns to Freshmen*. in *Proc. SIGCSE*. 2005.
27. Berlage, T., *A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects.* ACM Transactions on Computer-Human Interaction, 1994. **1**(3): p. 269-294.
28. Cooper, S., W. Dann, and R. Pausch, *Alice: a 3-D tool for introductory programming concepts.* J. Comput. Sci. Coll., 2000: p. 107-116.
29. Dewan, P. *Increasing the Automation of a Toolkit without Reducing its Abstraction and User-Interface Flexibility*. in *Proc. ACM EICS*. 2010.
30. Roussev, V., P. Dewan, and V. Jain. *Composable Collaboration Infrastructures based on Programming Patterns*. in *Procedings of ACM Computer Supported Cooperative Work*. 2000.
31. Dewan, P. *Discovery-based Praxes: Chanelling the User- Interface of an Industrial-Strength Programming Environment to Formally Teach Programming*. in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2017. Raleigh, NC: IEEE.

32. Andresen, M.A., *Asynchronous discussion forums: success factors, outcomes, assessments, and limitations.* Journal of Educational Technology & Society, 2009. **12**(1): p. 249-257.
33. Romero, C., M.-I. López, J.-M. Luna, and S. Ventura, *Predicting students' final performance from participation in on-line discussion forums.* Computers & Education, 2013. **68**: p. 458-472.
34. Rose Catherine, R.G., Karthik and D.R. Visweswariah. *Semisupervised answer extraction from discussion forums.* in *Proceedings of the Sixth International Joint Conference on Natural Language Processing,*. 2013.
35. Salter, N.P. and M.R. Conneely, *Structured and unstructured discussion forums as tools for student engagement.* Computers in Human Behavior, 2015. **46**: p. 18-25.
36. Dommett, E.J., *Understanding student use of twitter and online forums in higher education.* Education and Information Technologies, 2019. **24**(1): p. 325-343.
37. Vellukunnel, M., P. Buffum, K.E. Boyer, J. Forbes, S. Heckman, and K. Mayer-Patel. *Deconstructing the Discussion Forum: Student Questions and Computer Science Learning*. in *ACM SIGCSE 2017*. 2017.
38. Petress, K., *An operational definition of class participation.* College Student Journal, 2006. **40**(4): p. 821-824.
39. O'Connor, K., *Class participation: Promoting in-class student engagement.* Education, 2013. **133**(3): p. 340-344.
40. Simpson, C. and Y. Du, *Effects of learning styles and class participation on students' enjoyment level in distributed learning environments.* Journal of education for library and information science, 2004: p. 123-136.
41. Jones, R.C., *The "Why" of Class Participation: A Question Worth Asking.* College Teaching, 2008. **56**(1): p. 59-63.
42. Castelló, M. and C. Monereo, *Students' note-taking as a knowledge-construction tool.* L1-Educational Studies in Language and Literature, 2005. **5**(3): p. 265-285.
43. Kam, M., J. Wang, A. Iles, E. Tse, J. Chiu, D. Glaser, O. Tarshish, and J. Canny. *Livenotes: a system for cooperative and augmented note-taking in lectures*. in *Proceedings of the SIGCHI conference on Human factors in computing systems*. 2005.
44. Bruce, K.B., A. Danyluk, and T. Murtagh, *Introducing Concurrency in CS 1*, in *Proc. ACM SIGCSE'10*. 2010, ACM.
45. Geist, R., J.A. Levine, and J. Westall, *A problem-based learning approach to GPU computing*, in *Proceedings of the Workshop on Education for High-Performance Computing*. 2015, ACM: Austin, Texas. p. 1-8.
46. Lupo, C., Z.J. Wood, and C. Victorino, *Cross teaching parallelism and ray tracing: a project-based approach to teaching applied parallel computing*, in *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. 2012, ACM: Raleigh, North Carolina, USA. p. 523-528.
47. Eijkhout, V., *Parallel Programming Illustrated Through Conway's Game of Life*, in *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses, 1st Edition*, S. Prasad, A. Gupta, A. Rosenberg, A. Sussman, and C. Weems, Editor. 2019, Morgan Kaufmann.
48. Dewan, P. *The Structure of a Project-Based Course on the Fundamentals of Distributed Computing*. in *Proc. of EduHiPC-18 worksahop in IEEE HiPC*. 2018.
49. Aziz, M., H. Chi, A. Tibrewal, M. Grossman, and V. Sarkar, *Auto-grading for parallel programs*, in *Proceedings of the Workshop on Education for High-Performance Computing*. 2015, ACM: Austin, Texas. p. 1-8.
50. Carter, J., P. Dewan, and M. Pichilinani. *Towards Incremental Separation of Surmountable and Insurmountable Programming Difficulties*. in *Proc. SIGCSE*. 2015. ACM.
51. A. Smith, K.E. Boyer, J. Forbes, S. Heckman, and K. Mayer-Patel. *My Digital Hand: A Tool for Scaling Up One-to-One Peer Teaching in Support of Computer Science Learning*. in *ACM SIGCSE 2017*. 2017.