Visually Introducing Freshmen to Low-Level Java

Abstractions for Creating, Synchronizing and Coordinating Threads

Prasun Dewan
Department of Computer Science
University of North Carolina
Chapel Hill, USA dewan@cs.unc.edu

***Abstract*— We have developed and experimented with an approach to teach low-level Java concurrency abstractions in our first required course for CS majors, which assumes knowledge of procedural programming. The driving problems are visualized simulations of multiple physical objects in motion that may (a) be confined to a shared space and (b) coordinate with each other. Such simulations do not require any domain-specific knowledge such as sorting and image processing for driving problems and exercises, and their implementation demonstrates the benefits of object-based programming. They allow focus on both the performance and programmability benefits of concurrency, provide analogies for an abstraction-independent explanation of concurrency concepts, and can be used to incrementally motivate all low-level concurrency abstractions and visualize the effect of using and not using these abstractions. Layered simulation-based worked examples illustrating the abstractions were presented and easily understood in multiple offerings of a course that implemented this approach. Students implemented non-trivial assignments based on these abstractions, even when they were optional, did not face major obstacles because of visual error feedback, and were excited by concurrency as they felt it empowered them to implement arbitrary applications early.**

Keywords—education, simulations, animations, object-oriented programming, thread creation, synchronized methods, wait, notify, autonomous threads, Java, interleaving, concurrency

I. Introduction

The motivation for teaching concurrency 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 increase has spurred interest in pedagogical methods for teaching concurrency and the coupled concept of distributed computing, together often referred to as PDC (Parallel and Distributed Computing). The National Science Foundation has started a center to encourage such methods [1], which, in turn, has produced a book on this topic [2].

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. On the other hand, concurrency has been taught mainly in niche, typically nonrequired and/or graduate courses, such as operating systems and real-time systems. Many recent efforts have suggested adding concurrency earlier 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. Yet another argument for more work in this area is provided by Ghafoor et al [3], who estimate that less than 10% of the universities and community colleges that offer CS/CE degrees cover concurrency at the undergraduate level.

Our university traditionally covered concurrency only in one undergraduate course – the operating system course – which is not required and is typically taken by juniors and seniors. This paper describes the author's effort to introduce concurrency in the first required course for CS majors, referred to as our target course. It assumes knowledge of procedural programming.

In this paper, we focus on the contribution this effort makes to the field of concurrency education. The novelty of a pedagogical method can manifest itself in both the set of requirements it attempts to meet and the approach used to meet these requirements. In other words, the method can be distinguished by both *what* it is trying to cover and *how* it is meeting its goal. We address both aspects below.

We first develop a requirement space for describing a range of practical requirements existing PDC pedagogical methods

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

978-1-7281-4894-6/19/$31.00 ©2019 IEEE
DOI 10.1109/HiPCW.2019.00022

have been developed to meet. We then identify a new point in this space that, to the best of our knowledge, no published paper in this area has met. The remainder of the paper presents a key idea in our work to meet our goal –using simulations of multiple physical objects as driving problems in worked examples and student exercises.

II. Requirement Space and Goal

Concurrency can be introduced in a special course on this concept [4] or added to one [5-8] or more [9, 10] courses addressing different topics/domains in computer science.

The evolutionary approach has been taken in many efforts, and they can be distinguished by the topic of the course to which concurrency is added. Some introduce concurrency in the very first course on programming [5, 8]. Others have assumed prior experience with programming and have targeted CS2 [7, 8] or upper-level courses [11, 12]. Typically, CS2 is a data structure course, in which many have introduced concurrency[7, 8].

TABLE I. Domain vs. Concurrency Orientation

Orientation	
Concurrency Oriented	[4]
Domain-Oriented	[5-10](this paper)

TABLE II. Domains/Topics Embellished with Concurrency

Domain	
Intro. Programming	[5, 13]
Data Structures	[7, 8]
Object-Based Programming	(this paper)
Upper-Level Courses	[11, 12]

A course on parallelism must not only introduce the notion of concurrency but also abstractions for implementing it. Some concurrency courses cover high-level/declarative abstractions such as message-passing [5] and fork-join and reduce [6, 8]; while some address low-level/procedural ones such as explicit thread-creation [7, 13]. The low-level abstractions offer more flexibility but require more code to program.

TABLE III. Range of Concurrency Abstractions

Abstraction-Level	
Low-Level/Procedural	Threads, synchronization [7, 13] and this paper; Runnable interface, coordination (This paper)
High-Level/Declarative	[5-8]

A related issue is the programming language in which these abstractions are supported. These languages have included a visual programming language called Scratch [5], C [4, 8], Python [5, 6], and Java [7, 13].

TABLE IV. Languages Used for Programming Concurrency

Language Kind	
Visual	Scratch[5]
Procedural	C[4, 8]
Object-based	Python[5, 6]; Java [7, 13], this paper

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 this approach, the automatic context-switching offered by concurrency abstractions relieves the programmer from implementing it.

Most of the surveyed work has focused on performance, discussing parallelization of a variety of tasks such as scanning [6], sorting [6, 7], Monte Carlo Pi estimation [4], image processing [3, 4] and printing [5], 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 dual approach is using parallel communicating threads to make it easier to program a person moving with a ball [5] – the person and ball can be made separate threads that communicate with each other. As these threads are closely coupled and represent one autonomous activity, parallelization would reduce performance because of the overhead of thread creation and message passing.

Other examples have taken a hybrid approach by targeting multiple autonomous activities such as the Conway Game of Life [4, 14], and the Frogger™, Tetris and Centipede games, multiple independent ATM withdrawals [4, 13], and a single server connected to multiple clients [15]. Parallelization both saves programmers from implementing context-switching of such activities and increases performance when multiple cores are available.

TABLE V. Concurrenct Diriving Problems

Metric	**Driving Problems**
Performance	Scanning [6]; Sorting [6, 7]; Monte Carlo Pi estimation [4]; Image processing [3, 4]; Conway Game of Life [4, 14]; Frogger™, Tetris and Centipede games [13], Objects in Motion (this paper)
Programmability	Single ball-mover animation [5]; Conway Game of Life [4, 14]; ATM Withdrawals [4, 13], Frogger™, Tetris and Centipede games [13], Objects in Motion (this paper)

Our target course assumes knowledge of the basic procedural programming concepts of primitive types, arrays, loops and procedures, and is taken by many freshmen. It does not cover data structures, and goes beyond the basic concepts by teaching object-based programming in Java [16]. We designed

the concurrency addition to meet the following unique set of requirements:

1. Integration with object-based programming - the topic of the target course.
2. Coverage of all concurrency concepts provided by the Java *language* - thread creation by implementing a runnable interface or extending a thread class, synchronized methods, wait, notify, notifyAll - but not the higher-level Java library constructs such as fork-join and thread pools.
3. Focus on both performance and programmability benefits of concurrency.

To the best of our knowledge, no other published method on early concurrency meets requirements 1 and 2. Table III shows that coordination (notify and notifyAll) is not addressed by surveyed methods covering low-level concurrency abstractions. Table IV shows that none of these methods has considered adding concurrency to a course that focuses on object-based programming and assumes knowledge of procedural programming. In other words, we know of no other effort that has experimented with teaching all Java language abstractions in a course on object-oriented programming that focuses on both programmability and performance benefits. We describe below how we have met this unaddressed set of requirements.

III. Autonomous Interacting Objects in Motion

Ideally, a course introducing concurrency must have the following four components.

1. An *abstraction-independent explanation* of the concepts of interleaved and concurrent execution of multiple activities within a process, and synchronization and coordination of them.
2. *Layered explanations* of the *behavior* of a set of *thread abstractions* to support such activities.
3. *Layered worked-examples* that illustrate these abstractions by showing (a) how they can be programmed to implement realistic driving problems, and (b) the (programmability, performance and correctness) consequences of using and not using the abstractions for the problem.
4. *Layered student exercises* to use these abstractions.

Courses on operating system take the further step of also explaining the implementation of concurrency abstractions. The challenge in early introduction of concurrency is to include these four components without addressing the implementation of thread abstractions. Our key idea to meet this challenge was using as driving problems visual simulations of autonomous interacting objects in motion. This idea has been implicitly used in the design of several assignments (Table V). Here we explicitly articulate and motivate it as a foundation for concurrency analogies, driving problems, and exercises.

A. Rationale

There are many reasons for choosing such objects in a course introducing both object-based programming and concurrency.

Real-Life Analogies: These objects occur in real life; thus, they can be used for analogies to provide an abstractionindependent explanation of concurrency.

Prerequisite Free: Their parallel computer simulations can be motivated without requiring knowledge of other computerscience concepts such as image processing.

Demonstrate Benefits of Object-based Programming: Simulations of physical phenomenon are particularly suited for object-based programming (the first object-based language, Simula-67, was targeted at simulations), the subject of our target course.

Programmability and Performance: These objects represent multiple autonomous activities; thus, their parallelization offers both programmability and performance benefits.

Synchronization Illustration: By confining them to a shared space, synchronization constructs – in our course, Java synchronized methods - can be motivated and explained.

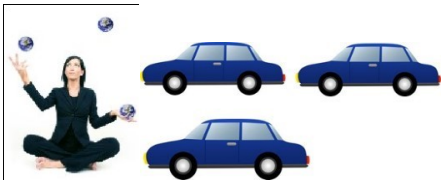
Coordination Illustration: By choosing cooperating objects, coordination constructs – in our course, Java wait, notify and notifyAll - can be motivated and explained.

Concurrency Visualization: The effect of using concurrency, synchronization and coordination mechanisms correctly can be visualized.

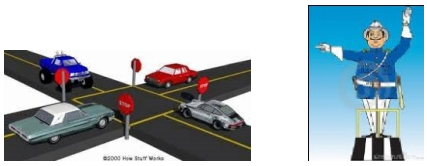
Layered, Incremental Introduction: It is possible to create related driving problems and exercises that require concurrency but not synchronization or coordination, and those that require synchronization but not coordination, allowing these three concepts to be introduced and implemented incrementally.

B. Analogies

Analogies involving physical objects in motion allowed us to meet the goal of providing an abstraction-independent explanation of concurrency. Two hands juggling three balls (Figure 1(a)) corresponded to two processors executing three threads. Preventing collision among three balls in a shared space (Figure 1(a)), two cars in the same lane (Figure 1(b), vehicles at an intersection (Figure 1(c)), and two cooperative runners exchanging batons (Figure 1(f)) corresponded to safe access to shared data by concurrent synchronized threads. Runners (Figure 1(e)) and vehicles (Figure 1(b)) allocated to different lanes corresponded to safe access to different data structures by unsynchronized threads. Baton exchange between two cooperating relay runners (Figure 1(f)) corresponded to thread coordination to achieve some joint task.



(a) (b)



(c) (d)

(e)

(f)

Fig. 1. Analogies involving Objects in Motion

C. Prerequisites and Layering of Worked Examples

All of our worked examples involve the animation of a space shuttle from the origin of a Cartesian Plane to a destination coordinate. (The icon for the shuttle remains horizontal through its flight.) This animation is implemented by an animator object that provides an animateFromOrigin method, which takes as arguments a shuttle object and its destination. The method moves the shuttle first in the Y direction and then in the X direction by calling the animateYFromOrigin and animateXFromOrigin methods, respectively. In some coordinated examples, the animators make use of two different objects, called "clearance managers", responsible for notifying waiting animators. Multiple implementations of the animator and clearance managers are used in the examples, which are related by inheritance. Also, the user-interface in the worked examples was implemented using the Model-View-Controller design pattern [16, 17]. Thread implementation through subclassing of the Thread class and implementation of the Runnable interface was covered. Therefore, the examples and underlying concepts were introduced after interfaces, inheritance and MVC are taught. With a different, less modular, coding style, they could be taught after interface or inheritance.

D. Sequential vs Concurrent Execution

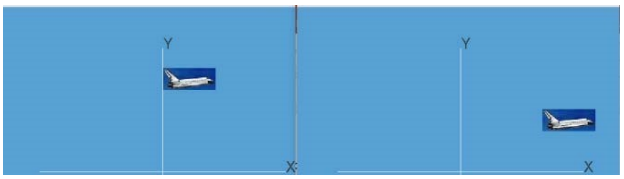
To illustrate the difference between sequential and concurrent method execution, we created a sequential and concurrent worked example. In the sequential example, a single thread executes animateFromOrigin sequentially in two instances of AShuttleAnimator to move two different shuttles. In the concurrent case, two different threads execute the method in two different animator instances controlling separate shuttles.

Figures 2 (a, b) and 3 (a, b) shows the difference in the behavior of the two programs. In the sequential case, the X coordinate of the left shuttle changes, while the right shuttle remains stationary, as we transition from snapshot (a) to (b). In the concurrent/interleaved case, the Y coordinates of both shuttles change in snapshots (a) and (b), giving an appearance of concurrent movement.

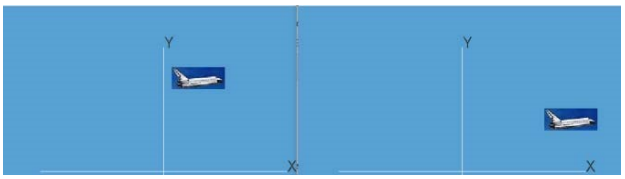
The console traces in Figure 2(c) and Figure 3(c) show the difference in algorithms in the two cases. In the sequential case, a single animator object (AShuttleAnimator@1901648626) executes the Y and X Loops in the main thread created automatically by Java. In the concurrent case, the main thread uses the Java Thread object to create and start two threads, [Shuttle Animation 1,5,main] and [Shuttle Animation 2,5,main], respectively, which use two different animator objects, to execute the animateFromOrigin method. As a result, the traces produced by the two executions are interleaved. Figure 3(c) also shows the asynchronous code execution possible with thread creation – the parent thread terminates with a message before the children it created finish execution.

E. Synchronized vs Unsynchronized Concurrency

To illustrate the need for thread synchronization, we create two instances of AShuttleAnimator that control a single shuttle, taking it along the paths (y^{11}, y^{12}, …) and (y^{21}, y^{22}, …), respectively. We fork from the main thread two threads that concurrently execute the animateFromOrigin method in the two animators.



a) Shuttle[1] at Position X^{11}, Shuttle[2] Stationary



b) Shuttle[1] at Position X^{12}, Shuttle[2] Stationary

```
Thread[main,5,main] started animating in Y Direction
in:AShuttleAnimator@1901648626
Thread[main,5,main] finished animating in Y Direction
in:AShuttleAnimator@1901648626
Thread[main,5,main] started animating in X Direction
in:AShuttleAnimator@1901648626
Thread[main,5,main] finished animating in X Direction
in:AShuttleAnimator@1901648626
Thread[main,5,main] started animating in Y Direction
in:AShuttleAnimator@2144912729
Thread[main,5,main] finished animating in Y Direction
in:AShuttleAnimator@2144912729
Thread[main,5,main] started animating in X Direction
in:AShuttleAnimator@2144912729
Thread[main,5,main] finished animating in X Direction
in:AShuttleAnimator@2144912729
Main terminates
```

c) Single Main Thread, Single Animator, Two Controller Shuttles

Fig. 2. Independent Serial Shuttle Animators

a) Shuttle[1] at Position Y^{11}, Shuttle[2] at Position Y^{21}

b) Shuttle[1] at Position Y^{12}, Shuttle[2] at Position Y^{21}

```
Thread:Thread[main,5,main] has started Thread[Shuttle Animation
1,5,main]
Thread:Thread[main,5,main] has started Thread[Shuttle Animation
2,5,main]
Main terminates
Thread[Shuttle Animation 2,5,main] started animating in Y Direction
in:AShuttleAnimator@673226183
Thread[Shuttle Animation 1,5,main] started animating in Y
Direction in:AShuttleAnimator@2114748546
Thread[Shuttle Animation 2,5,main] finished animating in Y
Direction in:AShuttleAnimator@673226183
Thread[Shuttle Animation 2,5,main] started animating in X Direction
in:AShuttleAnimator@673226183
Thread[Shuttle Animation 1,5,main] finished animating in Y
Direction in:AShuttleAnimator@2114748546
Thread[Shuttle Animation 1,5,main] started animating in X Direction
in:AShuttleAnimator@2114748546
```

```
Thread[Shuttle Animation 1,5,main] finished animating in X
Direction in:AShuttleAnimator@2114748546
```

(c) Two Animators, Two Created Interleaved Threads

Fig. 3. Independent Concurrent Shuttle Animators

Figure 4(a..d) shows that when the method is not synchronized, the shuttle oscillates between the trajectories computed by the animators taking y coordinates (y^{11}, y^{21}, y^{12}, y^{22}, ,…). Figure 5(a..d) shows that when the method is synchronized, the shuttle first follows the trajectory computed by the first animator, taking Y positions (y^{11}, y^{12} …), and then the trajectory computed by the second animator, taking Y positions (y^{21}, y^{22} , …). Figure 4(e) and 5(e) trace the fact that in both cases the same algorithm executes, involving two animators and threads. The difference is interleaving of Y movements in the unsynchronized case – the second shuttle starts movement in the Y direction before the first one finishes movement in that direction.

F. Internal vs External Coordination

Thread coordination involves concurrent activities that are related to each other and together accomplish some larger goal. As mentioned earlier, in our analogies of Figure 1, they correspond to runners racing competitively or cooperatively in a race (Figure 1(e, f)).

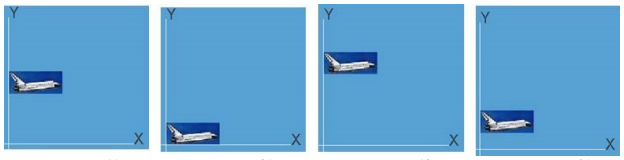
The activities of a set of related threads may be coordinated internally by each of the threads or be controlled externally by a single thread. In our analogies, internal control corresponds to two cooperating runners ensuring they coordinate their baton exchange themselves, while external control corresponds to a set of runners waiting for a whistle from a referee to start the race. In our shuttle example, this corresponds to a set of shuttles going on a joint mission. In the external case, their flights are controlled by methods of an external air traffic controller object being executed by a separate thread, or internally by the methods of their animators executed by different threads. In both cases, the animator methods perform blocking wait operations. In the internal case, they also execute notify or notifyAll, while in the external case, a thread manipulating the external controller (which itself does not wait) executes the unblocking operations

G. External Coordination

To support external coordination, we built a special visualized class called AClearanceManager, whose wait method can be called in a synchronized method to block a thread. An instance of this class displays the queue of threads that are waiting to be notified. The class also provides a Proceed button to interactively execute its notify method in a synchronized method called proceed. We also implemented a special case of this class, ABroadcastingClearanceManager, which provides an additional Proceed All button to call notifyAll in a synchronized method called proceedAll to unblock all waiting threads.

Figure 6 shows the use of a global AClearanceManager. Two separate shuttles are controlled by two different animators, which, this time, are instances of AShuttleAnimatorWatitingForClearance. They are like the ones we saw earlier, except that they execute the wait method in the AClearanceManager at the start of their animateFromOrigin method. Figure 6(a) shows them in the displayed queue. Figure 6(b) shows the effect of interactively pressing the Proceed button. The first animator is removed from the queue and starts moving its shuttle. Figure 6(c) shows that clicking the Proceed button again dequeues the second thread, which now starts animating the second shuttle.

Figure 6(c) shows the behavior of the wait calls executed by the animateFromOrigin method of the new animator. The execution by the first (second) thread is blocked until the first (second) execution of the notify call in the clearance manager by the AWT (user-interface) thread.

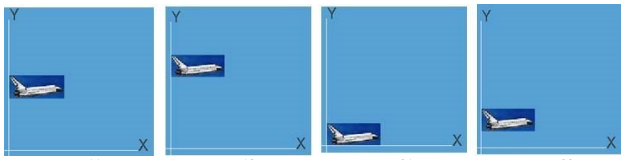


(a) At Pos y^{11} (b) At Pos y^{21} (c) At Pos y^{12} (d) At Pos y^{21}

```
Thread:Thread[main,5,main] has started Thread[Shuttle Animation
1,5,main]
Thread[Shuttle Animation 1,5,main] started animating in Y
Direction in:AShuttleAnimator@1817260044
Thread:Thread[main,5,main] has started Thread[Shuttle Animation
2,5,main]
Thread[Shuttle Animation 2,5,main] started animating in Y Direction
in:AShuttleAnimator@1817260044
Thread[Shuttle Animation 1,5,main] finished animating in Y Direction
in:AShuttleAnimator@1817260044
Thread[Shuttle Animation 1,5,main] started animating in X
Direction in:AShuttleAnimator@1817260044
Thread[Shuttle Animation 2,5,main] finished animating in Y
Direction in:AShuttleAnimator@1817260044
Thread[Shuttle Animation 2,5,main] started animating in X
Direction in:AShuttleAnimator@1817260044
Thread[Shuttle Animation 1,5,main] finished animating in X
Direction in:AShuttleAnimator@1817260044
```

(e) Two Threads, No Synchronization

Fig. 4. Single Shuttle Controlled by Two Unsynchronized Animators



(a)At Pos y^{11} (b) At Pos y^{12} (c) At Pos y^{21} (d) At Pos y^{22}

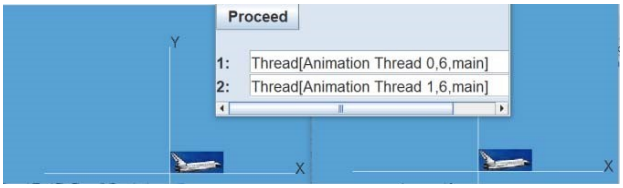
```
Thread:Thread[main,5,main] has started Thread[Shuttle Animation
1,5,main]
Thread[Shuttle Animation 1,5,main] started animating in Y
Direction in:ASynchronizedShuttleAnimator@1890996505
Thread:Thread[main,5,main] has started Thread[Shuttle Animation
2,5,main]
Thread[Shuttle Animation 1,5,main] finished animating in Y
Direction in:ASynchronizedShuttleAnimator@1890996505
Thread[Shuttle Animation 1,5,main] started animating in X
Direction in:ASynchronizedShuttleAnimator@1890996505
Thread[Shuttle Animation 1,5,main] finished animating in X Direction
in:ASynchronizedShuttleAnimator@1890996505
Thread[Shuttle Animation 2,5,main] started animating in Y
Direction in:ASynchronizedShuttleAnimator@1890996505
Thread[Shuttle Animation 2,5,main] finished animating in Y
Direction in:ASynchronizedShuttleAnimator@1890996505
Thread[Shuttle Animation 2,5,main] started animating in X
Direction in:ASynchronizedShuttleAnimator@1890996505
Thread[Shuttle Animation 2,5,main] finished animating in X Direction
in:ASynchronizedShuttleAnimator@1890996505
```

e) Two Threads, With Synchronization

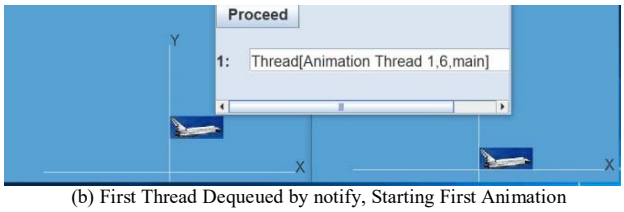
Fig. 5. Single Shuttle Controlled by Two Synchronized Animators

To illustrate notiifyAll, we created animators whose animateFromOrigin method performs a wait on ABroacastingClearanceManager. As before, the method executions by the two threads block. Figure 7(b) shows that when the Proceed All button is pressed, both threads are unblocked. Figure 7 (c) shows that the button press results in

ABroacastingClearanceManager calling notifyAll in the AWT thread, which causes all waiting threads to unblock from their wait calls.



(a) Both Threads Waiting in Queue



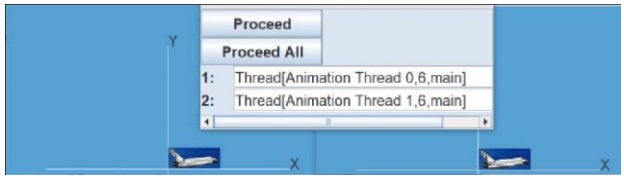
(b) First Thread Dequeued by notify, Starting First Animation

(c) Second Thread Dequeued by notify, Starting Second Animation

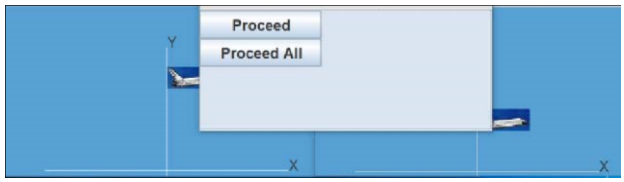
```
Thread[Animation Thread 0,6,main] before wait
Thread[Animation Thread 1,6,main] before wait
Thread[AWT-EventQueue-0,6,main] after notify
Thread[Animation Thread 0,6,main] after wait
Thread[Animation Thread 0,6,main] started animating in Y Direction
in:AShuttleAnimatorWatitingForClearance@1783146483
Thread[AWT-EventQueue-0,6,main] after notify
Thread[Animation Thread 1,6,main]:after wait
Thread[Animation Thread 1,6,main] started animating in
Y Direction in:AShuttleAnimatorWatitingForClearance@62182667
…
```

d) Traces of notify and wait calls

Fig. 6. Externally Coordinated Launch of a Single Shuttle at a Time



(a) Both Threads Waiting in Queue



(b) Both Threads Dequeued by notifyAll, Starting Both Animations

```
Thread[AWT-EventQueue-0,6,main]: after notifyAll
Thread[Animation Thread 0,6,main]: after wait
Thread[Animation Thread 1,6,main]: after wait
```

(c) notifyAll unblocks all waits

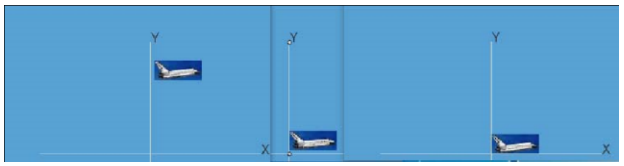
Fig. 7. Externally Coordinated Launch of Multiple Shuttles at a Time

H. Internal Coordination

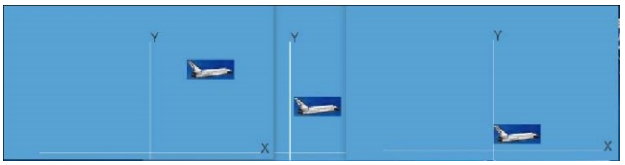
As mentioned earlier, the coordination in Figures 6 and 7 corresponds to flight takeoffs being controlled by an external agent such as an air traffic controller. Once in air, shuttles should be responsible for coordinating their trajectories. To simulate such internal coordination, we implemented the ALock class, which has a Boolean variable. It offers a lock operation that waits if the Boolean is true and unlock operation that invokes notify and sets the Boolean to false. We created a new kind of animator, AControlledShuttleAnimator. Before (after) calling animateYFromOrigin, the method calls the lock (unlock) operation on a lock. We created three instances of this class that use the same lock, and three threads to execute their animateFromOrigin method on three different shuttles. Figure 8(a..d) shows the coordination among the three animators.

(a) Three Shuttles Ready to Launch

(b) First Animator Gets Lock for Y Axis and Launches First Shuttle



(c) Second Animator Gets Released Lock and Launches Second Shuttle



(d) Third Animator Gets Released Lock and Launches Third Shuttle

```
Thread[Animation Thread 0,6,main] waiting for
lock:lectures.animation.threads.wait_notify.lock.ALock@5145d7f0
Thread[Animation Thread 0,6,main] got
lock:lectures.animation.threads.wait_notify.lock.ALock@5145d7f0
Thread[Animation Thread 0,6,main] started animating in Y
Direction in:AControlledShuttleAnimator@1248334686 Thread[Animation
Thread 1,6,main] waiting for
lock:lectures.animation.threads.wait_notify.lock.ALock@5145d7f0
Thread[Animation Thread 2,6,main] waiting for
lock:lectures.animation.threads.wait_notify.lock.ALock@5145d7f0
Thread[Animation Thread 0,6,main] finished animating in Y
Direction in:AControlledShuttleAnimator@1248334686
Thread[Animation Thread 0,6,main] returning from releaseLock
Thread[Animation Thread 0,6,main] started animating in X
Direction in:AControlledShuttleAnimator@1248334686
…
```

(e) Three lock operations, each unlock causes the oldest lock to unblock

Fig. 8. Internally-Coordinated Launch of Multiple Shuttles

The animateYFromOrigin method of all three animators execute the lock operation of the shared instance of ALock. The first animator gets the lock first, and keeps it until it reaches its highest Y position (Figure 8(b)). At this point, the method releases the lock, allowing the second waiting animator to start animating in the Y direction (Figure 8(c)), which releases its lock when it reaches its highest position, causing the last animator to start moving its shuttle (Figure 8(d)). The trace of

Figure 8(e) shows that each unlock causes the oldest lock invocation to unblock.

I. Project and Experience

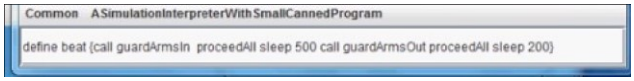
As mentioned earlier, our target course focuses on objectbased programming, whose benefits are manifested in large evolving programs. Therefore, the course requires a set of layered assignments that together implement a semester-wide project. The nature of the projects we have given is inspired by the Alice programming environment [18], which allows novice programmers to interactively implement and call procedures that create and manipulate objects in a predefined 3-D virtual environment. In our projects, students do not *use* a predefined Alice environment; instead, they *implement from scratch* an Alice-inspired project that provides commands to manipulate objects in a 2-D virtual environment. The projects have varied. For instance, one of the projects has been a Halloween simulation of "trick or treat" [15]. The most recent project simulates the bridge scene in the movie "Monty Python and the Holy Grail" in which King Arthur and his knights attempt to cross a bridge after answering questions posed by a guard.

A virtual environment is particularly well suited for concurrency assignments involving objects in motion. We illustrate using the Bridge Scene project. To exercise concurrency, students create animators that make avatars of Arthur, Galahad, Robin and Lancelot move simultaneously. To exercise synchronization, they synchronize access by multiple threads that manipulate the same knight. To exercise thread coordination, taking liberties with the bridge scene, they create threads and animators that make the knights clap to a beat set by the guard. The concurrency abstractions can be used directly in the implemented simulations or indirectly by end-users of the simulation who interactively call commands to create and coordinate threads.

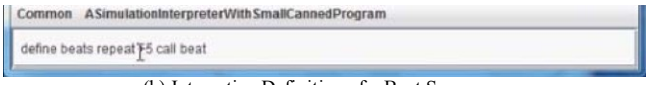
Figure 9(a..c) shows the end-user entering commands to create synchronized clapping. Figure 9(a) defines the beat procedure to simulate a single beat, which makes the guard tuck its arms in, sleep for 500ms, spread its arms out, call proceedAll in ABroadcastingClearanceManager, and then sleep for 200ms. Figure 9(b) defines the beats procedure to simulate multiple beats, which calls beat 5 times. Figure 9(c) invokes the beats procedure in a separate thread.

Before calling the beat procedure, the user calls an animation method for each knight, executed by a separate thread, that moves the knight's arms in and out in response to notifications from the clearance manager. Thus, when the beat procedure is called, the knights and guard move their arms in unison – when the guard moves his arms out (in), so do the knights, as shown in Figure 9(d) (Figure 9(e)).

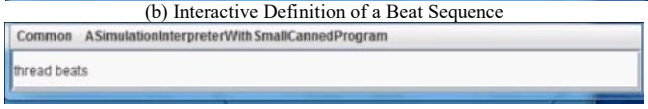
The author has taught the concurrency-augmented target course once almost every year since Fall 2012. Concurrency, synchronization, and coordination have been parts of the last three assignments (A10, A11, A12), respectively, which are also responsible for command parsing and interpretation (the bulk of the assignments), implementation and use of generic types, implementation and use of exception classes, and undo/redo. In all of these offerings, A1-A9 were required. In different offerings, some or all features of A10-A12 were made extra credit based on the quizzes added to the course.



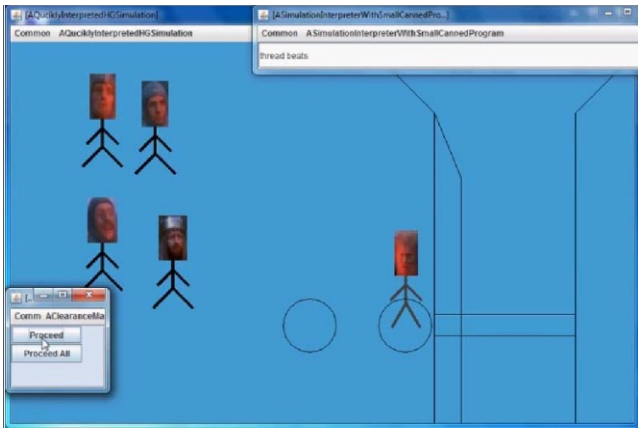
(a) Interactive Definition of a Single Beat



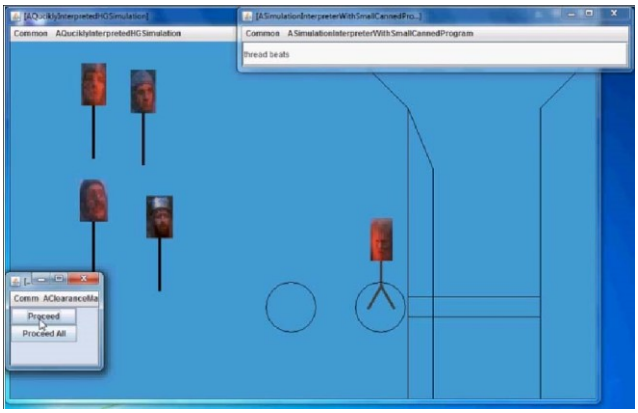
(b) Interactive Definition of a Beat Sequence



(c) Interactive Call to Execute Beat Sequence in a Thread



(d) Coordinated Guard and Knight Arms Out



(e) Coordinated Guard and Knight Arms In

Fig. 9. Knights Clapping to a Beat Set by the Guard

Table VI shows two data points. In Fall 2012, all three assignments were required, and almost all students who submitted the last concurrency-unware assignment also submitted the concurrency-aware ones. In Fall 2018, all three were optional. Though students had less time and incentive to implement the concurrency-aware assignments, almost half of them did so. Our experience shows that (a) the visual approach made it easy for students to understand concurrency and the Java language abstractions for supporting it, (b) the concurrency aspects of the assignments did not create major obstacles in any of our offerings, as they were applications of the concurrency patterns [8, 19] demonstrated in the worked examples and gave visual feedback in case of errors, and (c) students felt excited by concurrency as they believed its knowledge empowered them to implement any application. This excitement is consistent with that reported in other work [13].

TABLE VI. REQUIRED VS OPTIONAL CONCURRENCY ASSIGNMENTS

Semester	A8	A9	A10	A11	A12
F2018	61	59	45	33	28
F2012	73	70	67	67	67

IV. SUMMARY AND FUTURE WORK

The contributions of this paper are (a) a five-dimensional (Tables I to V) space that succinctly compares the goals of existing work on concurrency pedagogy, (b) identification of a new practical point in this space, (c) motivation of a visual approach to support this point, (d) and layered worked examples and project-based exercises that implemented this approach in multiple offerings of our target course.

While we have implemented our approach in a singlemodule course on Java object-based programming, the general idea is applicable in a variety of contexts. The presentation of this idea in this paper has shown no Java code. Thus, it can be applied to a course taught in other object-based languages by implementing the driving problems and clearance managers in these languages. It can also be applied to a course in other languages such as C by providing more labor-intensive code for creating worked examples and student exercises. It is particularly suitable for a multi-module introduction to concurrency, wherein our worked examples and assignments can be an additional module taught in Java or other languages.

It will be attractive to create such ports as future work, and to use our current examples directly in the large number of Javabased courses offered today that cover the prerequisite for our concurrency module – interfaces or inheritance. Future work can also address how the concurrency concepts and worked examples should be learned by students – should they participate in live lectures, watch videos, or do hands-on manipulation of worked examples [5], or use some mix of these techniques? It would also be useful to explore early introduction of starvation and deadlock. Further work is needed to develop techniques for automatic assessment of concurrent programs to both help grading of finished exercises and provide guidance while they are being developed

[1] Prasad, S.K., A. Gupta, A. Rosenberg, A. Sussman, and C. Weems. CDER Center | NSF/IEEE-TCPP Curriculum Initiative," NSF/IEEETCPP Curriculum Initiative. 2017; Available from: https://grid.cs.gsu.edu/~tcpp/curriculum/?q=node/21183.

[2] 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.

[3] 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.

[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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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. 523528.

[13] Bruce, K.B., A. Danyluk, and T. Murtagh, Introducing Concurrency in CS 1, in Proc. ACM SIGCSE'10. 2010, ACM.

[14] 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.

[15] 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.

[16] Dewan, P. Teaching Inter-Object Design Patterns to Freshmen. in Proc. SIGCSE. 2005.

[17] Krasner, G.E. and S.T. Pope, A Cookbook for Using the Model-ViewController User Interface Paradigm in Smalltalk-80. Journal of ObjectOriented Programming, 1988. 1(3): p. 26-49.

[18] Cooper, S., W. Dann, and R. Pausch, Alice: a 3-D tool for introductory programming concepts. J. Comput. Sci. Coll., 2000: p. 107-116.

[19] Keutzer, K., B. Massingill, T. Mattson, and B.S. 2010. A Design Pattern Language for Engineering (Parallel) Software: Merging the PLPP and OPL Projects. in 2010 Workshop on Parallel Programming Pattern (Carefree, AZ), ParaPLoP'10. 2010.