Managing Concurrency in Mobile User Interfaces with Examples in Android

Konstantin Läufer and George K. Thiruvathukal

Abstract In this chapter, we explore various parallel and distributed computing topics from a user-centric software engineering perspective. Specifically, in the context of mobile application development, we study the basic building blocks of interactive applications in the form of events, timers, and asynchronous activities, along with related software modeling, architecture, and design topics.

Relevant software engineering topics: software requirements: functional requirements (C), nonfunctional requirements (C) software design: user interface patterns (A), concurrency patterns (A), testing patterns (A), architectural patterns (C), dependency injection (C), design complexity (C); software testing: unit testing (A), managing dependencies in testing (A); cross-cutting topics: web services (C), pervasive and mobile computing (A)

Relevant parallel and distributed computing topics: algorithmic problems: asynchrony (C); architecture classes: simultaneous multithreading (K), SMP (K); parallel programming paradigms and notations: task/thread spawning (A); semantics and correctness issues: tasks and threads (C), synchronization (A); concurrency defects: deadlocks (C), thread safety/race conditions (A); cross-cutting topics: why and what is parallel/distributed computing (C), concurrency (A), nondeterminism (C)

Learning outcomes: The student will be able to model and design mobile applications involving events, timers, and asynchronous activities. The student will be able to implement these types of applications on the Android platform. The student will develop an understanding of nonfunctional requirements.

Context for use: A semester-long intermediate to advanced undergraduate course on object-oriented development. Assumes prerequisite CS2 and background in an object-oriented language such as Java, `C++`, or `C#`.

K. Läufer (✉) · G. K. Thiruvathukal
Department of Computer Science, Loyola University Chicago, Chicago, IL, USA
e-mail: laufer@cs.luc.edu; gkt@cs.luc.edu

© Springer International Publishing AG, part of Springer Nature 2018

S. K. Prasad et al. (eds.), *Topics in Parallel and Distributed Computing*,
https://doi.org/10.1007/978-3-319-93109-8_9

Background and Motivation

In this chapter, we will explore various parallel and distributed computing topics from a user-centric software engineering perspective. Specifically, in the context of mobile application development, we will study the basic building blocks of interactive applications in the form of events, timers, and asynchronous activities, along with related software modeling, architecture, and design topics.

Based on the authors' ongoing research and teaching in this area, this material is suitable for a five-week module on concurrency topics within a semester-long intermediate to advanced undergraduate course on object-oriented development. It is possible to extend coverage by going into more depth on the online examples [17] and studying techniques for offloading tasks to the cloud [19]. The chapter is intended to be useful to instructors and students alike.

Given the central importance of the human-computer interface for enabling humans to use computers effectively, this area has received considerable attention since around 1960 [26]. Graphical user interfaces (GUIs) emerged in the early 1970s and have become a prominent technical domain addressed by numerous widget toolkits (application frameworks for GUI development). Common to most of these is the need to balance ease of programming, correctness, performance, and consistency of look-and-feel. Concurrency always plays at least an implicit role and usually becomes an explicit programmer concern when the application involves processor-bound, potentially long-running activities controlled by the GUI. Here, long-running means anything longer than the user wants to wait for before being able to continue interacting with the application. This chapter is about the concepts and techniques required to achieve this balance between correctness and performance in the context of GUI development.

During the last few years, mobile devices such as smartphones and tablets have displaced the desktop PC as the predominant front-end interface to information and computing resources. In terms of global internet consumption (minutes per day), mobile devices overtook desktop computers in mid-2014 [5], and "more websites are now loaded on smartphones and tablets than on desktop computers" [14] as of October 2016. Google also announced [3] that it will be displaying mobile-friendly web sites higher in the search results, which speaks to the new world order. These mobile devices participate in a massive global distributed system where mobile applications offload substantial resource needs (computation and storage) to the cloud.

In response to this important trend, this chapter focuses on concurrency in the context of mobile application development, especially Android, which shares many aspects with previous-generation (and desktop-centric) GUI application frameworks such as Java AWT and Swing yet. (And it almost goes without saying that students are more excited about learning programming principles via technologies like Android and iOS, which they are using more often than their desktop computers.)

While the focus of this chapter is largely on concurrency within the mobile device itself, the online source code for one of our examples [19] goes beyond the on-device

experience by providing versions that connect to RESTful web services (optionally hosted in the cloud) [6]. We've deliberately focused this chapter around the on-device experience, consistent with "mobile first" thinking, which more generally is the way the "Internet of Things" also works [1]. This thinking results in proper separation of concerns when it comes to the user experience, local computation, and remote interactions (mediated using web services).

It is worth taking a few moments to ponder why mobile platforms are interesting from the standpoint of parallel and distributed computing, even if at first glance it is obvious. From an architectural point of view, the landscape of mobile devices has followed a similar trajectory to that of traditional multiprocessing systems. The early mobile device offerings, even when it came to smartphones, were single core. At the time of writing, the typical smartphone or tablet is equipped with four CPU cores and a graphics processing unit (GPU), with the trend of increasing cores (to at least 8) expected to continue in mobile CPUs. In this vein, today's–and tomorrow's–devices need to be considered serious parallel systems in their own right. (In fact, in the embedded space, there has been a corresponding emergence of parallel boards, similar to the Raspberry Pi.)

The state of parallel computing today largely requires the mastery of two styles, often appearing in a hybrid form: *task parallelism* and *data parallelism*. The emerging mobile devices are following desktop and server architecture by supporting both of these. In the case of task parallelism, to get good performance, especially when it comes to the user experience, concurrency must be disciplined. An additional constraint placed on mobile devices, compared to parallel computing, is that unbounded concurrency (threading) makes the device unusable/unresponsive, even to a greater extent than on desktops and servers (where there is better I/O performance in general). We posit that learning to program concurrency in a resource-constrained environment (e.g. Android smartphones) can be greatly helpful for writing better concurrent, parallel, and distributed code in general. More importantly, today's students really want to learn about emerging platforms, so this is a great way to develop new talent in languages and systems that are likely to be used in future parallel/distributed programming environments.

Roadmap

In the remainder of this chapter, we first summarize the fundamentals of thread safety in terms of concurrent access to shared mutable state.

We then discuss the technical domain of applications with graphical user interfaces (GUIs), GUI application frameworks that target this domain, and the runtime environment these frameworks typically provide.

Next, we examine a simple interactive behavior and explore how to implement this using the Android mobile application development framework. To make our presentation relevant to problem solvers, our running example is a bounded click

counter application (more interactive and exciting than the examples commonly found in concurrency textbooks, e.g., atomic counters and bounded buffers) that can be used to keep track of the capacity of, say, a movie theater.

We then explore more interesting scenarios by introducing timers and internal events. For example, a countdown timer can be used for notification of elapsed time, a concept that has almost uniquely emerged in the mobile space but has applications in embedded and parallel computing in general, where asynchronous paradigms have been present for some time, dating to job scheduling, especially for longer-running jobs.

We close by exploring applications requiring longer-running, processor-bound activities. In mobile app development, a crucial design goal is to ensure UI responsiveness and appropriate progress reporting. We demonstrate techniques for making sure that computation proceeds but can be interrupted by the user. These techniques can be generalized to offload processor-bound activities to cloud-hosted web services.[1]

Fundamentals of Thread Safety

Before we discuss concurrency issues in GUI applications, it is helpful to understand the underlying fundamentals of thread safety in situations where two or more concurrent threads (or other types of activities) access shared mutable state.

Thread safety is best understood in terms of *correctness*: An implementation is *correct* if and only if it conforms to its specification. The implementation is *thread-safe* if and only if it continues to behave correctly in the presence of multiple threads [28].

Example: Incrementing a Shared Variable

Let's illustrate these concepts with perhaps the simplest possible example: incrementing an integer number. The specification for this behavior follows from the definition of increment: *After performing the increment, the number should be one greater than before*.

Here is a first attempt to implement this specification in the form of an instance variable in a Java class and a `Runnable` instance that wraps around our increment code and performs it on demand when we invoke its run method (see below).

```
1 int shared = 0;
2
3 final Runnable incrementUnsafe = new Runnable() {
4   @Override public void run() {
```

[1] This topic goes beyond the scope of this chapter but is included in the corresponding example [19].

```
5     final int local = shared;
6     tinyDelay();
7     shared = local + 1;
8   }
9 };
```

To test whether our implementation satisfies the specification, we can write a simple test case:

```
1 final int oldValue = shared;
2 incrementUnsafe.run();
3 assertEquals(oldValue + 1, shared);
```

In this test, we perform the increment operation in the only thread we have, that is, the main thread. Our implementation passes the test with flying colors. Does this mean it is thread-safe, though?

To find out, we will now test two or more concurrent increment operations, where the instance variable shared becomes shared state. Generalizing from our specification, the value of the variable should go up by one for each increment we perform. We can write this test for two concurrent increments

```
1 final int threadCount = 2;
2 final int oldValue = shared;
3 runConcurrently(incrementUnsafe, threadCount);
4 assertEquals(oldValue + threadCount, shared);
```

where `runConcurrently` runs the given code concurrently in the desired number of threads:

```
1 public void runConcurrently(
2     final Runnable inc, final int threadCount) {
3   final Thread[] threads = new Thread[threadCount];
4   for (int i = 0; i < threadCount; i += 1) {
5     threads[i] = new Thread(inc);
6   }
7   for (final Thread t : threads) {
8     t.start();
9   }
10  for (final Thread t : threads) {
11    try {
12      t.join();
13    } catch (final InterruptedException e) {
14      throw new RuntimeException("interrupted during join");
15    }
16  }
17 }
```

But this test does not always pass! When it does not, one of the two increments appears to be lost. Even if its failure rate were one in a million, the specification is violated, meaning that *our implementation of increment is not thread-safe*.

Interleaved Versus Serialized Execution

Let's try to understand exactly what is going on here. We are essentially running two concurrent instances of this code:

```
1 /*f1*/ int local1 = shared;          /*f2*/ int local2 = shared;
2 /*s1*/ shared = local1 + 1;          /*s2*/ shared = local2 + 1;
```

(For clarity, we omit the invocation of `tinyDelay` present in the code above; this invokes `Thread.sleep(0)` and is there just so we can observe and discuss this phenomenon in conjunction with the Java thread scheduler.)

The instructions are labeled f_n and s_n for `fetch` and `set`, respectively. Within each thread, execution proceeds sequentially, so we are guaranteed that f_1 always comes before s_1 and f_2 always comes before s_2. But we do not have any guarantees about the relative order across the two threads, so all of the following interleavings are possible:

- $f_1\ s_1\ f_2\ s_2$: increments `shared` by 2
- $f_1\ f_2\ s_1\ s_2$: increments `shared` by 1
- $f_1\ f_2\ s_2\ s_1$: increments `shared` by 1
- $f_2\ f_1\ s_1\ s_2$: increments `shared` by 1
- $f_2\ f_1\ s_2\ s_1$: increments `shared` by 1
- $f_2\ s_2\ f_1\ s_1$: increments `shared` by 2

This kind of situation, where the behavior is nondeterministic in the presence of two or more threads is also called a *race condition*.[2]

Based on our specification, the only correct result for incrementing twice is to see the effect of the two increments, meaning the value of shared goes up by two. Upon inspection of the possible interleavings and their results, the only correct ones are those where both steps of one increment happen before both steps of the other increment.

Therefore, to make our implementation thread-safe, we need to make sure that the two increments do not overlap. Each has to take place *atomically*. This requires one to go first and the other to go second; their execution has to be *serialized* or *sequentialized* (see also [10] for details on the *happens-before* relation among operations on shared memory).

[2]When analyzing race conditions, we might be tempted to enumerate the different possible interleavings. While it seems reasonable for our example, this quickly becomes impractical because of the combinatorial explosion for larger number of threads with more steps.

Using Locks to Guarantee Serialization

In thread-based concurrent programming, the primary means to ensure atomicity is mutual exclusion by *locking*. Most thread implementations, including *p-threads (POSIX threads)*, provide some type of locking mechanism.

Because Java supports threads in the language, each object carries its own lock, and there is a `synchronized` construct for allowing a thread to execute a block of code only with the lock held. While one thread holds the lock, other threads wanting to acquire the lock on the same object will join the *wait set* for that object. As soon as the lock becomes available—when the thread currently holding the lock finishes the synchronized block—, another thread from the wait set receives the lock and proceeds. (In particular, there is no first-come-first-serve or other fairness guarantee for this wait set.)

We can use locking to make our implementation of increment atomic and thereby thread-safe [20]:

```
1  final Object lock = new Object();
2
3  final Runnable incrementSafe = new Runnable() {
4    @Override public void run() {
5      synchronized (lock) {
6        final int local = shared;
7        tinyDelay();
8        shared = local + 1;
9      }
10   }
11 };
```

Now it is guaranteed to pass the test every time.

```
1 final int threadCount = 2;
2 final int oldValue = shared;
3 runConcurrently(incrementUnsafe, threadCount);
4 assertEquals(oldValue + threadCount, shared);
```

We should note that thread safety comes at a price: There is a small but not insignificant overhead in handling locks and managing their wait sets.

The GUI Programming Model and Runtime Environment

As we mentioned above, common to most GUI application framework is the need to balance ease of programming, correctness, performance, and consistency of look-and-feel. In this section, we will discuss the programming model and runtime environment of a typical GUI framework.

In a GUI application, the user communicates with the application through input events, such as button presses, menu item selections, etc. The application responds to user events by invoking some piece of code called an *event handler* or *event listener*. To send output back to the user, the event handler typically performs some action that the user can observe, e.g., displaying some text on the screen or playing a sound.

The GUI Runtime Environment

Real-world GUI applications can be quite complex in terms of the number of components and their logical containment hierarchy. The GUI framework is responsible for translating *low-level events* such as mouse clicks and key presses to *semantic events* such as button presses and menu item selections targeting the correct component instances. To manage this complexity, typical GUI frameworks use a producer-consumer architecture, in which an internal, high-priority system thread places low-level events on an event queue, while an application-facing *UI thread*[3] takes successive events from this queue and delivers each event to its correct target component, which then forward it to any attached listener(s). The UML sequence diagram in Fig. 1 illustrates this architecture.

Because the event queue is designed to be thread-safe, it can be shared safely between producer and consumer. It coalesces and filters groups of events as appropriate, maintaining the following discipline:

- *Sequential (single-threaded)* processing: At most one event from this queue is dispatched simultaneously.
- *Preservation of ordering:* If an event A is enqueued to the event queue before event B, then event B will not be dispatched before event A.

Concretely, the UI thread continually takes events from the event queue and processes them. Here is the pseudo-code for a typical UI thread.

```
1 run() {
2   while (true) {
3     final Event event = eq.getNextEvent();
4     final Object src = event.getSource();
5     ((Component) src).processEvent(event);
6   }
7 }
```

[3]In some frameworks, including Java AWT/Swing, the UI thread is known as *event dispatch thread (EDT)*.

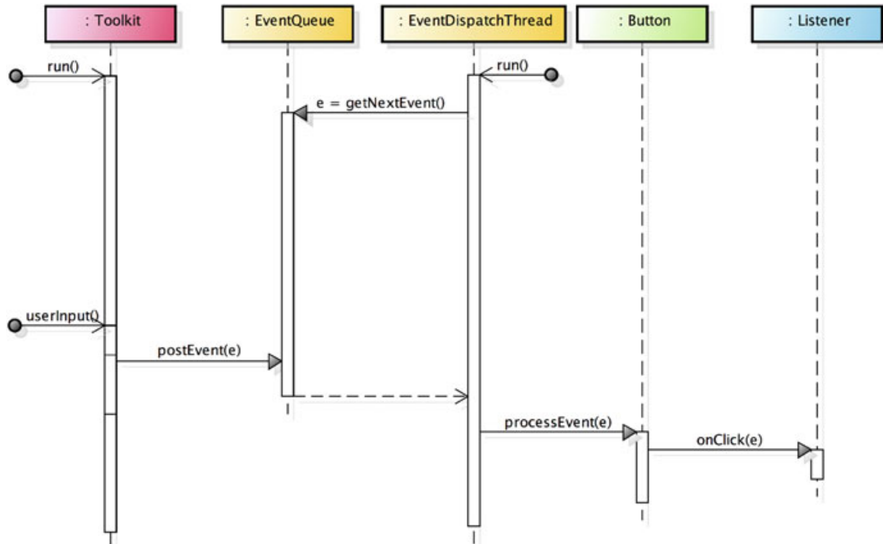

Fig. 1 UML sequence diagram showing the producer-consumer architecture of a GUI. Stick arrowheads represent asynchronous invocation, while solid arrowheads represent (synchronous) method invocation

The target component, e.g., `Button`, forwards events to its listener(s).

```
1 processEvent(e) {
2   if (e instanceof OnClickEvent) {
3     listener.onClick(e);
4   }
5   ...
6 }
```

While this presentation is mostly based on Java's AWT for simplicity, Android follows a similar approach with `MessageQueue` at the core and some responsibilities split between `Handler` and `Looper` instances [27].

This general approach, where requests (the events) come in concurrently, get placed on a request queue, and are dispatched sequentially to handlers, is an instance of the *Reactor design pattern* [30].

The Application Programmer's Perspective

Within the GUI programming model, the application programmer focuses on creating components and attaching event listeners to them. The following is a very

simple example of the round-trip flow of information between the user and the application.

```
1 final Button button = new Button("press me");
2 final TextView display = new TextView("hello");
3
4 increment.setOnClickListener(new OnClickListener() {
5   @Override public void onClick(final View view) {
6     display.setText("world");
7   }
8 });
```

The event listener mechanism at work here is an instance of the *Observer design pattern* [8]: Whenever the event source, such as the button, has something to say, it notifies its observer(s) by invoking the corresponding event handling method and passing itself as the argument to this method. If desired, the listener can then obtain additional information from the event source.

Thread Safety in GUI Applications: The Single-Threaded Rule

Generally, the programmer is oblivious to the concurrency between the internal event producer thread and the UI thread. The question is whether there is or should be any concurrency on the application side. For example, if two button presses occur in very short succession, can the two resulting invocations of `display.setText` overlap in time and give rise to thread safety concerns? In that case, should we not make the GUI thread-safe by using locking?

The answer is that typical GUI frameworks are already designed to address this concern. Because a typical event listener accesses and/or modifies the data structure constituting the visible GUI, if there were concurrency among event listener invocations, we would have to achieve thread safety by serializing access to the GUI using a lock (and paying the price for this). It would be the application programmer's responsibility to use locking whenever an event listener accesses the GUI. So we would have greatly complicated the whole model without achieving significantly greater concurrency in our system.

We recall our underlying producer-consumer architecture, in which the UI thread processes one event at a time in its main loop. This means that event listener invocations are already serialized. Therefore, we can achieve thread safety directly and without placing an additional burden on the programmer by adopting this simple rule:

> The application must always access GUI components from the UI thread.

This rule, known as the *single-threaded rule*, is common among most GUI frameworks, including Java Swing and Android. In practice, such access must happen either during initialization (before the application becomes visible), or

within event listener code. Because it sometimes becomes necessary to create additional threads (usually for performance reasons), there are ways for those threads to schedule code for execution on the UI thread.

Android actually *enforces* the single-threaded GUI component access rule by raising an exception if this rule is violated at runtime. Android also enforces the "opposite" rule: It prohibits any code on the UI thread that will block the thread, such as network access or database queries [11].

Using Java Functional Programming Features for Higher Conciseness

To ensure compatibility with the latest and earlier versions of the Android platform, the examples in this chapter are based on Java 6 language features and API. As of October 2017, Android Studio 3.0 supports several recently introduced Java language features, including *lambda expressions* and *method references*; for details, please see [13].

These features can substantially improve both the conciseness and clarity of callback code, such as runnable tasks and Android event listeners. For example, given the equivalence between a single-method interface and a lambda expression with the same signature as the method, we can rewrite `incrementSafe` from section "Using Locks to Guarantee Serialization" and `setOnClickListener` from section "The Application Programmer's Perspective" more concisely:

```
1 final Runnable incrementSafe = () ->
2   synchronized (lock) {
3     final int local = shared;
4     tinyDelay();
5     shared = local + 1;
6   };
```

```
1 increment.setOnClickListener(
2   (final View view) -> display.setText("world")
3 );
```

Single-Threaded Event-Based Applications

In this section, we will study a large class of applications that will not need any explicit concurrency at all. As long as each response to an input event is short, we can keep these applications simple and responsive by staying within the Reactor pattern.

We will start with a simple interactive behavior and explore how to implement this using the Android mobile application development framework [9]. Our running example will be a bounded click counter application that can be used to keep track of the capacity of, say, a movie theater. The complete code for this example is available online [18].

The Bounded Counter Abstraction

A *bounded counter* [16], the concept underlying this application, is an integer counter that is guaranteed to stay between a preconfigured minimum and maximum value. This is called the *data invariant* of the bounded counter.

$$min \leq counter \leq max$$

We can represent this abstraction as a simple, passive object with, say, the following interface:

```
1 public interface BoundedCounter {
2   void increment();
3   void decrement();
4   int get();
5   boolean isFull();
6   boolean isEmpty();
7 }
```

In following a *test-driven* mindset [2], we test implementations of this interface using methods such as this one, which ensures that incrementing the counter works properly:

```
1 @Test
2 public void testIncrement() {
3   decrementIfFull();
4   assertFalse(counter.isFull());
5   final int v = counter.get();
6   counter.increment();
7   assertEquals(v + 1, counter.get());
8 }
```

In the remainder of this section, we'll put this abstraction to good use by building an interactive application on top of it.

The Functional Requirements for a Click Counter Device

Next, let's imagine a device that realizes this bounded counter concept. For example, a greeter positioned at the door of a movie theater to prevent overcrowding would require a device with the following behavior:

- The device is preconfigured to the capacity of the venue.
- The device always displays the current counter value, initially zero.
- Whenever a person enters the movie theater, the greeter presses the *increment* button; if there is still capacity, the counter value goes up by one.
- Whenever a person leaves the theater, the greeter presses the *decrement* button; the counter value goes down by one (but not below zero).
- If the maximum has been reached, the *increment* button either becomes unavailable (or, as an alternative design choice, attempts to press it cause an error). This behavior continues until the counter value falls below the maximum again.
- There is a *reset* button for resetting the counter value directly to zero.

A Simple Graphical User Interface (GUI) for a Click Counter

We now provide greater detail on the user interface of this click counter device. In the case of a dedicated hardware device, the interface could have tactile inputs and visual outputs, along with, say, audio and haptic outputs.

As a minimum, we require these interface elements:

- Three buttons, for incrementing and decrementing the counter value and for resetting it to zero.
- A numeric display of the current counter value.

Optionally, we would benefit from different types of feedback:

- Beep and/or vibrate when reaching the maximum counter value.
- Show the percentage of capacity as a numeric percentage or color thermometer.

Instead of a hardware device, we'll now implement this behavior as a mobile software app, so let's focus first on the minimum interface elements. In addition, we'll make the design choice to disable operations that would violate the counter's data invariant.

These decisions lead to the three *view states* for the bounded click counter Android app (see Fig. 2: In the initial (minimum) view state, the decrement button is disabled. In the counting view state of the, all buttons are enabled. Finally, in the maximum view state, the increment button is disabled; we assume a maximum value of 10). In our design, the reset button is always enabled.

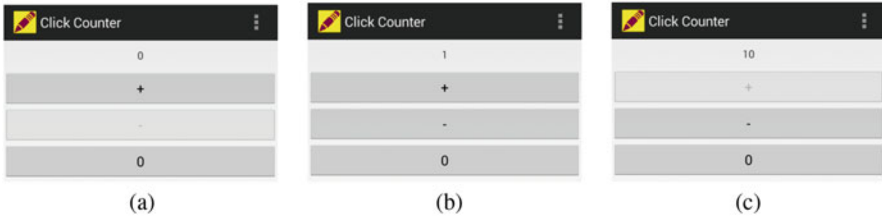

Fig. 2 View states for the click counter. (**a**) Minimum state. (**b**) Counting state. (**c**) Maximum state

Understanding User Interaction as Events

It was fairly easy to express the familiar bounded counter abstraction and to envision a possible user interface for putting this abstraction to practical use. The remaining challenge is to tie the two together in a meaningful way, such that the interface uses the abstraction to provide the required behavior. In this section, we'll work on bridging this gap.

Modeling the Interactive Behavior

As a first step, let's abstract away the concrete aspects of the user interface:

- Instead of touch buttons, we'll have *input events*.
- Instead of setting a visual display, we'll *modify a counter value*.

After we take this step, we can use a UML state machine diagram [29] to model the dynamic behavior we described at the beginning of this section more formally.[4] Note how the touch buttons correspond to events (triggers of *transitions*, i.e., arrows) with the matching names.

The behavior starts with the *initial pseudostate* represented by the black circle. From there, the counter value gets its initial value, and we start in the minimum state. Assuming that the minimum and maximum values are at least two apart, we can increment unconditionally and reach the counting state. As we keep incrementing, we stay here as long as we are at least two away from the maximum state. As soon as we are exactly one away from the maximum state, the next increment takes us to that state, and now we can no longer increment, just decrement. The system mirrors this behavior in response to the decrement event. There is a surrounding global state to support a single reset transition back to the minimum state. Figure 3 shows the complete diagram.

[4] A full introduction to the Unified Modeling Language (UML) [29] would go far beyond the scope of this chapter. Therefore, we aim to introduce the key elements of UML needed here in an informal and pragmatic manner. Various UML resources, including the official specification, are available at http://www.uml.org/. Third-party tutorials are available online and in book form.

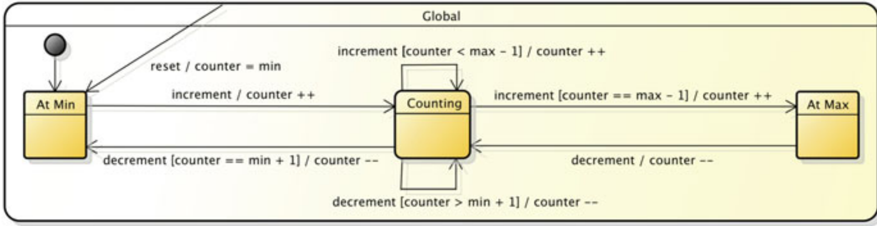

Fig. 3 UML state machine diagram modeling the dynamic behavior of the bounded counter application

As you can see, the three model states map directly to the view states from the previous subsection, and the transitions enabled in each model state map to the buttons enabled in each view state. This is not always the case, though, and we will see examples in a later section of an application with multiple model states but only a single view state.

GUI Components as Event Sources

Our next step is to bring the app to life by connecting the visual interface to the interactive behavior. For example, when pressing the increment button in a non-full counter state, we expect the displayed value to go up by one. In general, the user can trigger certain events by interacting with view components and other event sources. For example, one can press a button, swipe one's finger across the screen, rotate the device, etc.

Event Listeners and the Observer Pattern

We now discuss what an event is and what happens after it gets triggered. We will continue focusing on our running example of pressing the increment button.

The visual representation of an Android GUI is usually auto-generated from an XML source during the build process.[5] For example, the source element for our increment button looks like this; it declaratively maps the `onClick` attribute to the `onIncrement` method in the associated activity instance.

```
1 <Button
2   android:id="@+id/button_increment"
3   android:layout_width="fill_parent"
4   android:layout_height="wrap_content"
5   android:onClick="onIncrement"
6   android:text="@string/label_increment" />
```

[5]It is also possible—though less practical—to build an Android GUI programmatically.

The *Android manifest* associates an app with its main activity class. The top-level `manifest` element specifies the Java package of the activity class, and the activity element on line 5 specifies the name of the activity class, `ClickCounter-Activity`.

```
1  <manifest
2      xmlns:android="http://schemas.android.com/apk/res/android"
3      package="edu.luc.etl.cs313.android.clickcounter" ...>
4    ...
5    <application ...>
6      <activity android:name=".ClickCounterActivity" ...>
7        <intent-filter>
8          <action android:name="android.intent.action.MAIN" />
9          <category
10           android:name="android.intent.category.LAUNCHER" />
11       </intent-filter>
12     </activity>
13   </application>
14 </manifest>
```

An *event* is just an invocation of an *event listener* method, possibly with an argument describing the event. We first need to establish the association between an event source and one (or possibly several) event listener(s) by *subscribing* the listener to the source. Once we do that, every time this source emits an event, normally triggered by the user, the appropriate event listener method gets called on each subscribed listener.

Unlike ordinary method invocations, where the caller knows the identity of the callee, the (observable) event source provides a general mechanism for subscribing a listener to a source. This technique is widely known as the *Observer design pattern* [8].

Many GUI frameworks follow this approach. In Android, for example, the general component superclass is View, and there are various types of listener interfaces, including `OnClickListener`. In following the *Dependency Inversion Principle (DIP)* [24], the `View` class owns the interfaces its listeners must implement.

```
1  public class View {
2    ...
3    public static interface OnClickListener {
4      void onClick(View source);
5    }
6    public void setOnClickListener(OnClickListener listener) {
7      ...
8    }
9    ...
10 }
```

Android follows an event source/listener naming idiom loosely based on the JavaBeans specification [15]. Listeners of, say, the `onX` event implement the `OnXListener` interface with the `onX(Source source)` method. Sources of this kind of event implement the `setOnXListener` method.[6] An actual event instance corresponds to an invocation of the `onX` method with the source component passed as the source argument.

Processing Events Triggered by the User

The Android activity is responsible for mediating between the view components and the POJO (plain old Java object) bounded counter model we saw above. The full cycle of each event-based interaction goes like this.

- By pressing the increment button, the user triggers the `onClick` event on that button, and the `onIncrement` method gets called.
- The `onIncrement` method interacts with the model instance by invoking the increment method and then requests a view update of the activity itself.
- The corresponding `updateView` method also interacts with the model instance by retrieving the current counter value using the `get` method, displays this value in the corresponding GUI element with unique ID `textview_value`, and finally updates the view states as necessary.

Figure 4 illustrates this interaction step-by-step.

```
1  public void onIncrement(final View view) {
2    model.increment();
3    updateView();
4  }
5  protected void updateView() {
6    final TextView valueView =
7      (TextView) findViewById(R.id.textview_value);
8    valueView.setText(Integer.toString(model.get()));
9    // afford controls according to model state
10   ((Button) findViewById(R.id.button_increment))
11     .setEnabled(!model.isFull());
12   ((Button) findViewById(R.id.button_decrement))
13     .setEnabled(!model.isEmpty());
14 }
```

What happens if the user presses two buttons at the same time? As discussed above, the GUI framework responds to at most one button press or other event trigger at any given time. While the GUI framework is processing an event, it

[6] Readers who have worked with GUI framework that supports multiple listeners, such as Swing, might initially find it restrictive of Android to allow only one. We'll leave it as an exercise to figure out which well-known software design pattern can be used to work around this restriction.

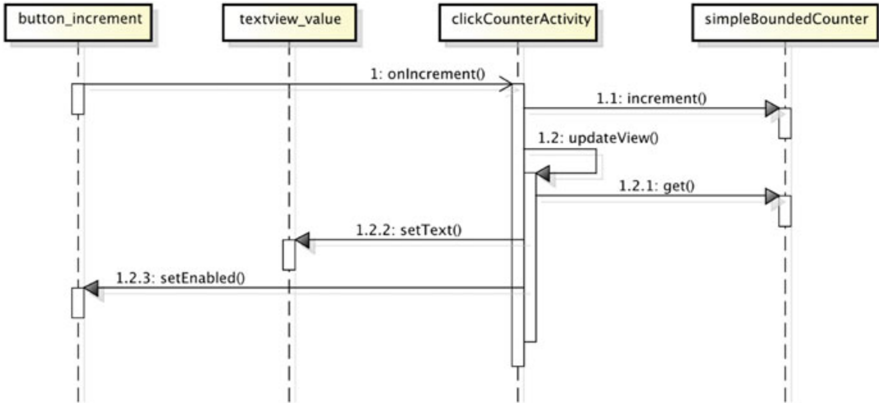

Fig. 4 Sequence diagram showing the full event-based interaction cycle in response to a press of the increment button. Stick arrowheads represent events, while solid arrowheads represent (synchronous) method invocation

places additional incoming event triggers on a queue and fully processes each one in turn. Specifically, only after the event listener method handling the current event returns will the framework process the next event. (Accordingly, activation boxes of different event listener method invocations in the UML sequence diagram must not overlap.) As discussed in section "Thread Safety in GUI Applications: The Single-Threaded Rule", this *single-threaded event handling* approach keeps the programming model simple and avoids problems, such as race conditions or deadlocks, that can arise in multithreaded approaches.

Application Architecture

This overall application architecture, where a component mediates between view components and model components, is known as *Model-View-Adapter (MVA)* [4], where the adapter component mediates all interactions between the view and the model. (By contrast, the *Model-View-Controller (MVC)* architecture has a triangular shape and allows the model to update the view(s) directly via update events.)

Figure 5 illustrates the MVA architecture. The solid arrows represent ordinary method invocations, and the dashed arrow represents event-based interaction. View and adapter play the roles of observable and observer, respectively, in the Observer pattern that describes the top half of this architecture.

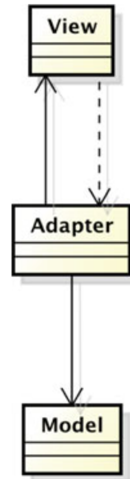

Fig. 5 UML class diagram showing the Model-View-Adapter (MVA) architecture of the bounded click counter Android app. Solid arrows represent method invocation, and dashed arrows represent event flow

System-Testing GUI Applications

Automated system testing of entire GUI applications is a broad and important topic that goes beyond the scope of this chapter. Here, we complete our running example by focusing on a few key concepts and techniques.

In system testing, we distinguish between our application code, usually referred to as the *system under test (SUT)*, and the *test code*. At the beginning of this section, we already saw an example of a simple component-level unit test method for the POJO bounded counter model. Because Android view components support triggering events programmatically, we can also write system-level test methods that mimic the way a human user would interact with the application.

System-Testing the Click Counter

The following test handles a simple scenario of pressing the reset button, verifying that we are in the minimum view state, then pressing the increment button, verifying that the value has gone up and we are in the counting state, pressing the reset button again, and finally verifying that we are back in the minimum state.

```
1  @Test
2  public void testActivityScenarioIncReset() {
3    assertTrue(getResetButton().performClick());
4    assertEquals(0, getDisplayedValue());
5    assertTrue(getIncButton().isEnabled());
6    assertFalse(getDecButton().isEnabled());
7    assertTrue(getResetButton().isEnabled());
8    assertTrue(getIncButton().performClick());
9    assertEquals(1, getDisplayedValue());
10   assertTrue(getIncButton().isEnabled());
```

```
11   assertTrue(getDecButton().isEnabled());
12   assertTrue(getResetButton().isEnabled());
13   assertTrue(getResetButton().performClick());
14   assertEquals(0, getDisplayedValue());
15   assertTrue(getIncButton().isEnabled());
16   assertFalse(getDecButton().isEnabled());
17   assertTrue(getResetButton().isEnabled());
18   assertTrue(getResetButton().performClick());
19 }
```

The next test ensures that the visible application state is preserved under device rotation. This is an important and effective test because an Android application goes through its entire lifecycle under rotation.

```
1  @Test
2  public void testActivityScenarioRotation() {
3    assertTrue(getResetButton().performClick());
4    assertEquals(0, getDisplayedValue());
5    assertTrue(getIncButton().performClick());
6    assertTrue(getIncButton().performClick());
7    assertTrue(getIncButton().performClick());
8    assertEquals(3, getDisplayedValue());
9    getActivity().setRequestedOrientation(
10     ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
11   assertEquals(3, getDisplayedValue());
12   getActivity().setRequestedOrientation(
13     ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);
14   assertEquals(3, getDisplayedValue());
15   assertTrue(getResetButton().performClick());
16 }
```

System Testing In and Out of Container

We have two main choices for system-testing our app:

- *In-container/instrumentation testing* in the presence of the target execution environment, such as an actual Android phone or tablet emulator (or physical device). This requires deploying both the SUT and the test code to the emulator and tends to be quite slow. So far, Android's build tools officially support only this mode.
- *Out-of-container testing* on the development workstation using a test framework such as *Robolectric* that simulates an Android runtime environment tends to be considerably faster. This and other non-instrumentation types of testing can be integrated in the Android build process with a bit of extra effort.

Although the Android build process does not officially support this or other types of non-instrumentation testing, they can be integrated in the Android build process with a bit of extra effort.

Structuring Test Code for Flexibility and Reuse

Typically, we'll want to run the exact same test logic in both cases, starting with the simulated environment and occasionally targeting the emulator or device. An effective way to structure our test code for this purpose is the xUnit design pattern *Testcase Superclass* [25]. As the pattern name suggests, we pull up the common test code into an abstract superclass, and each of the two concrete test classes inherits the common code and runs it in the desired environment.

```
1 @RunWith(RobolectricTestRunner.class)
2 public class ClickCounterActivityRobolectric
3 extends AbstractClickCounterActivityTest {
4   // some minimal Robolectric-specific code
5 }
```

The official Android test support, however, requires inheriting from a specific superclass called `ActivityInstrumentationTestCase2`. This class now takes up the only superclass slot, so we cannot use the Testcase Superclass pattern literally. Instead, we need to approximate inheriting from our `AbstractClick-CounterActivityTest` using delegation to a subobject. This gets the job done but can get quite tedious when a lot of test methods are involved.

```
1 public class ClickCounterActivityTest
2 extends ActivityInstrumentationTestCase2<ClickCounterActivity>
      {
3   ...
4   // test subclass instance to delegate to
5   private AbstractClickCounterActivityTest actualTest;
6
7   @UiThreadTest
8   public void testActivityScenarioIncReset() {
9     actualTest.testActivityScenarioIncReset();
10  }
11  ...
12 }
```

Having a modular architecture, such as model-view-adapter, enables us to test most of the application components in isolation. For example, our simple unit tests for the POJO bounded counter model still work in the context of the overall Android app.

Test Coverage

Test coverage describes the extent to which our test code exercises the system under test, and there are several ways to measure test coverage [31]. We generally want test coverage to be as close to 100% as possible and can measure this using suitable tools, such as JaCoCo along with the corresponding Gradle plugin.[7]

Interactive Behaviors and Implicit Concurrency with Internal Timers

In this section, we'll study applications that have richer, timer-based behaviors compared to the previous section. Our example will be a countdown timer for cooking and similar scenarios where we want to be notified when a set amount of time has elapsed. The complete code for a very similar example is available online [21].

The Functional Requirements for a Countdown Timer

Let's start with the functional requirements for the countdown timer, amounting to a fairly abstract description of its controls and behavior.

The timer exposes the following controls:

- One two-digit display of the form 88.
- One multi-function button.

The timer behaves as follows:

- The timer always displays the remaining time in seconds.
- Initially, the timer is stopped and the (remaining) time is zero.
- If the button is pressed when the timer is stopped, the time is incremented by one up to a preset maximum of 99. (The button acts as an increment button.)
- If the time is greater than zero and three seconds elapse from the most recent time the button was pressed, then the timer beeps once and starts running.
- While running, the timer subtracts one from the time for every second that elapses.
- If the timer is running and the button is pressed, the timer stops and the time is reset to zero. (The button acts as a cancel button.)

[7]More information on JaCoCo the JaCoCo Gradle plugin is available at http://www.eclemma.org/jacoco/ and https://github.com/arturdm/jacoco-android-gradle-plugin, respectively.

Fig. 6 View states for the countdown timer. (**a**) Initial stopped state with zero time. (**b**) Initial stopped state after adding some time. (**c**) Running (counting down) state. (**d**) Alarm ringing state

- If the timer is running and the time reaches zero by itself (without the button being pressed), then the timer stops counting down, and the alarm starts beeping continually and indefinitely.
- If the alarm is sounding and the button is pressed, the alarm stops sounding; the timer is now stopped and the (remaining) time is zero. (The button acts as a stop button.)

A Graphical User Interface (GUI) for a Countdown Timer

Our next step is to flesh out the GUI for our timer. For usability, we'll label the multifunction button with its current function. We'll also indicate which state the timer is currently in.

The screenshots in Fig. 6 show the default scenario where we start up the timer, add a few seconds, wait for it to start counting down, and ultimately reach the alarm state.

Modeling the Interactive Behavior

Let's again try to describe the abstract behavior of the countdown timer using a UML state machine diagram. As usual, there are various ways to do this, and our guiding principle is to keep things simple and close to the informal description of the behavior.

It is easy to see that we need to represent the current counter value. Once we accept this, we really don't need to distinguish between the stopped state (with counter value zero) and the counting state (with counter value greater than zero). The other states that arise naturally are the running state and the alarm state. Figure 7 shows the resulting UML state machine diagram.

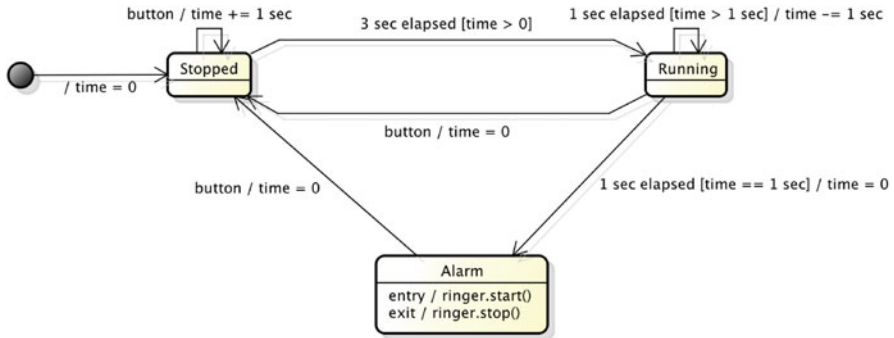

Fig. 7 UML state machine diagram modeling the dynamic behavior of the countdown timer application

As in the click counter example, these model states map directly to the view states shown above. Again, the differences among the view states are very minor and are aimed mostly at usability: A properly labeled button is a much more effective affordance than an unlabeled or generically labeled one.

Note that there are two types of (internal) timers at work here:

- *one-shot timers*, such as the three-second timer in the stopped state that gets restarted every time we press the multifunction button to add time
- *recurring timers*, such as the one-second timer in the running state that fires continually for every second that goes by

The following is the control method that starts a recurring timer that ticks approximately every second.

```
1  // called on the UI thread
2  public void startTick(final int periodInSec) {
3    if (recurring != null) throw new IllegalStateException();
4
5    recurring = new Timer();
6
7    // The clock model runs onTick every 1000 milliseconds
8    // by specifying initial and periodic delays
9    recurring.schedule(new TimerTask() {
10     @Override public void run() {
11       // fire event on the timer's internal thread
12       listener.onTick();
13     }
14   }, periodInSec * 1000, periodInSec * 1000);
15 }
```

Thread-Safety in the Model

Within the application model, each timer has its own internal thread on which it schedules the run method of its `TimerTask` instances. Therefore, other model components, such as the state machine, that receive events from either the UI and one or more timers, or more than one timer, will have to be kept thread-safe. The easiest way to achieve this is to use locking by making all relevant methods in the state machine object synchronized; this design pattern is known as *Fully Synchronized Object* [22] or *Monitor Object* [7, 28, 30].

```
1 @Override public synchronized void onButtonPress() {
2   state.onButtonPress();
3 }
4 @Override public synchronized void onTick() {
5   state.onTick();
6 }
7 @Override public synchronized void onTimeout() {
8   state.onTimeout();
9 }
```

Furthermore, update events coming back into the adapter component of the UI may happen on one of the timer threads. Therefore, to comply with the single-threaded rule, the adapter has to explicitly reschedule such events on the UI thread, using the `runOnUiThread` method it inherits from `android.app.Activity`.

```
1 @Override public void updateTime(final int time) {
2   // UI adapter responsibility
3   // to schedule incoming events on UI thread
4   runOnUiThread(new Runnable() {
5     @Override public void run() {
6       final TextView tvS =
7         (TextView) findViewById(R.id.seconds);
8       tvS.setText(Integer.toString(time / 10) +
9         Integer.toString(time % 10));
10    }
11  });
12 }
```

Alternatively, you may wonder whether we can stay true to the single-threaded rule and reschedule all events on the UI thread at their sources. This is possible using mechanisms such as the `runOnUiThread` method and has the advantage that the other model components such as the state machine no longer have to be thread-safe. The event sources, however, would now depend on the adapter; to keep this dependency manageable and our event sources testable, we can express it in terms of a small interface (to be implemented by the adapter) and inject it into the event sources.

```
1 public interface UIThreadScheduler {
2   void runOnUiThread(Runnable r);
3 }
```

Some GUI frameworks, such as Java Swing, provide non-view components for scheduling tasks or events on the UI thread, such as `javax.swing.Timer`. This avoids the need for an explicit dependency on the adapter but retains the implicit dependency on the UI layer.

Meanwhile, Android developers are being encouraged to use `Scheduled-ThreadPoolExecutor` instead of `java.util.Timer`, though the thread-safety concerns remain the same as before.

Implementing Time-Based Autonomous Behavior

While the entirely passive bounded counter behavior from the previous section was straightforward to implement, the countdown timer includes autonomous timer-based behaviors that give rise to another level of complexity.

There are different ways to deal with this behavioral complexity. Given that we have already expressed the behavior as a state machine, we can use the *State design pattern* [8] to separate state-dependent behavior from overarching handling of external and internal triggers and actions.

We start by defining a state abstraction. Besides the same common methods and reference to its surrounding state machine, each state has a unique identifier.

```
1  abstract class TimerState
2  implements TimerUIListener, ClockListener {
3
4    public TimerState(final TimerStateMachine sm) {
5      this.sm = sm;
6    }
7
8    protected final TimerStateMachine sm;
9
10   @Override public final void onStart() { onEntry(); }
11   public void onEntry() { }
12   public void onExit() { }
13   public void onButtonPress() { }
14   public void onTick() { }
15   public void onTimeout() { }
16   public abstract int getId();
17 }
```

In addition, a state receives UI events and clock ticks. Accordingly, it implements the corresponding interfaces, which are defined as follows:

```
public interface TimerUIListener {
  void onStart();
  void onButtonPress();
}

public interface ClockListener {
  void onTick();
  void onTimeout();
}
```

As we discussed in section “Understanding User Interaction as Events”, Android follows an event source/listener naming idiom. Our examples illustrate that it is straightforward to define custom app-specific events that follow this same convention. Our `ClockListener`, for example, combines two kinds of events within a single interface.

Concrete state classes implement the abstract `TimerState` class. The key parts of the state machine implementation follow:

```
// intial pseudo-state
private TimerState state = new TimerState(this) {
  @Override public int getId() {
    throw new IllegalStateException();
  }
};

protected void setState(final TimerState nextState) {
  state.onExit();
  state = nextState;
  uiUpdateListener.updateState(state.getId());
  state.onEntry();
}
```

Let’s focus on the stopped state first. In this state, neither is the clock ticking, nor is the alarm ringing. On every button press, the remaining running time goes up by one second and the one-shot three-second idle timeout starts from zero. If three seconds elapse before another button press, we transition to the running state.

```
private final TimerState STOPPED = new TimerState(this) {
  @Override public void onEntry() {
    timeModel.reset(); updateUIRuntime();
  }
  @Override public void onButtonPress() {
    clockModel.restartTimeout(3 /* seconds */);
    timeModel.inc(); updateUIRuntime();
  }
  @Override public void onTimeout() { setState(RUNNING); }
  @Override public int getId() { return R.string.STOPPED; }
};
```

Let's now take a look at the running state. In this state, the clock is ticking but the alarm is not ringing. With every recurring clock tick, the remaining running time goes down by one second. If it reaches zero, we transition to the ringing state. If a button press occurs, we stop the clock and transition to the stopped state.

```
1 private final TimerState RUNNING = new TimerState(this) {
2   @Override public void onEntry() {
3     clockModel.startTick(1 /* second */);
4   }
5   @Override public void onExit() { clockModel.stopTick(); }
6   @Override public void onButtonPress() { setState(STOPPED); }
7   @Override public void onTick() {
8     timeModel.dec(); updateUIRuntime();
9     if (timeModel.get() == 0) { setState(RINGING); }
10  }
11  @Override public int getId() { return R.string.RUNNING; }
12 };
```

Finally, in the ringing state, nothing is happening other than the alarm ringing. If a button press occurs, we stop the alarm and transition to the stopped state.

```
1 private final TimerState RINGING = new TimerState(this) {
2   @Override public void onEntry() {
3     uiUpdateListener.ringAlarm(true);
4   }
5   @Override public void onExit() {
6     uiUpdateListener.ringAlarm(false);
7   }
8   @Override public void onButtonPress() { setState(STOPPED); }
9   @Override public int getId() { return R.string.RINGING; }
10 };
```

Managing Structural Complexity

We can again describe the architecture of the countdown timer Android app as an instance of the Model-View-Adapter (MVA) architectural pattern. In Fig. 8, solid arrows represent (synchronous) method invocation, and dashed arrows represent (asynchronous) events. Here, both the view components and the model's autonomous timer send events to the adapter.

The user input scenario in Fig. 9 illustrates the system's end-to-end response to a button press. The internal timeout gets set in response to a button press. When the timeout event actually occurs, corresponding to an invocation of the `onTimeout` method, the system responds by transitioning to the running state.

By contrast, the autonomous scenario in Fig. 10 shows the system's end-to-end response to a recurring internal clock tick, corresponding to an invocation of the `onTick` method. When the remaining time reaches zero, the system responds by transitioning to the alarm-ringing state.

Fig. 8 The countdown timer's Model-View-Adapter (MVA) architecture with additional event flow from model to view

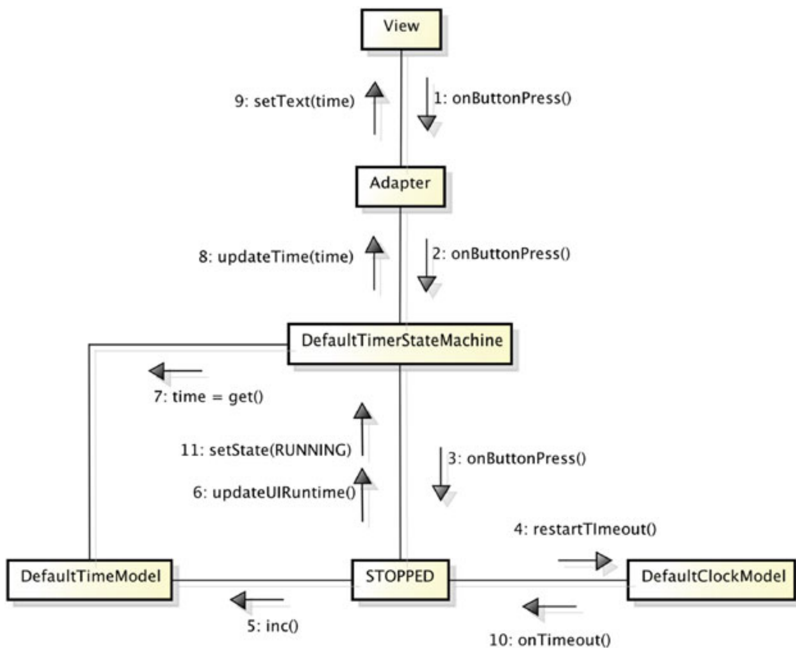

Fig. 9 Countdown timer: user input scenario (button press)

Testing GUI Applications with Complex Behavior and Structure

As we develop more complex applications, we increasingly benefit from thorough automated testing. In particular, there are different structural levels of testing:

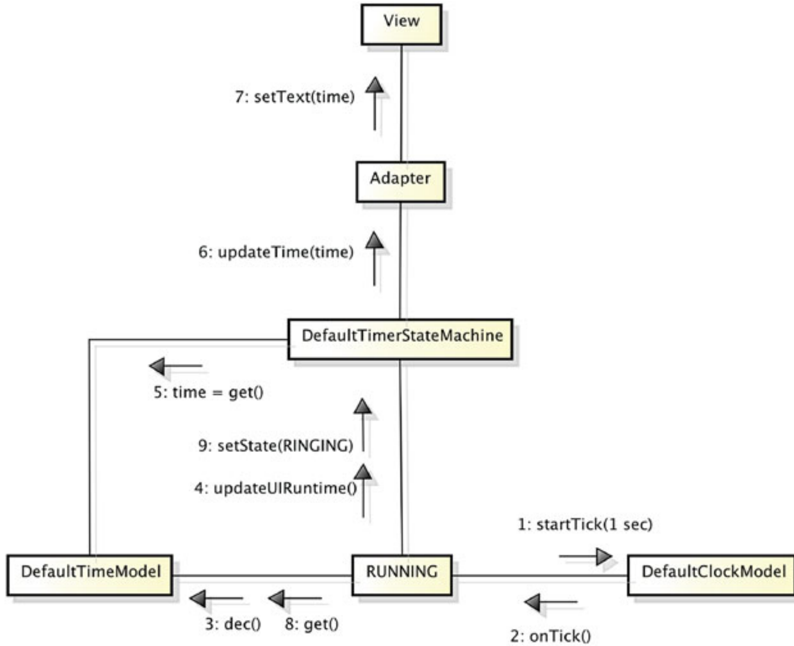

Fig. 10 Countdown timer: autonomous scenario (timeout)

component-level unit testing, integration testing, and system testing. Testing is particularly important in the presence of concurrency, where timing and nondeterminism are of concern.

In addition, as our application grows in complexity, so does our test code, so it makes sense to use good software engineering practice in the development of our test code. Accordingly, software design patterns for test code have emerged, such as the Testclass Superclass pattern [25] we use in section "Understanding User Interaction as Events".

Unit-Testing Passive Model Components

The time model is a simple passive component, so we can test it very similarly as the bounded counter model in section "Understanding User Interaction as Events".

Unit-Testing Components with Autonomous Behavior

Testing components with autonomous behavior is more challenging because we have to attach some kind of probe to observe the behavior while taking into account the presence of additional threads.

Let's try this on our clock model. The following test verifies that a stopped clock does not emit any tick events.

```
1 @Test
2 public void testStopped() throws InterruptedException {
3   final AtomicInteger i = new AtomicInteger(0);
4   model.setClockListener(new ClockListener() {
5     @Override public void onTick() { i.incrementAndGet(); }
6     @Override public void onTimeout() { }
7   });
8   Thread.sleep(5500);
9   assertEquals(0, i.get());
10 }
```

And this one verifies that a running clock emits roughly one tick event per second.

```
1 @Test
2 public void testRunning() throws InterruptedException {
3   final AtomicInteger i = new AtomicInteger(0);
4   model.setClockListener(new ClockListener() {
5     @Override public void onTick() { i.incrementAndGet(); }
6     @Override public void onTimeout() { }
7   });
8   model.startTick(1 /* second */);
9   Thread.sleep(5500);
10  model.stopTick();
11  assertEquals(5, i.get());
12 }
```

Because the clock model has its own timer thread, separate from the main thread executing the tests, we need to use a thread-safe `AtomicInteger` to keep track of the number of clock ticks across the two threads.

Unit-Testing Components with Autonomous Behavior and Complex Dependencies

Some model components have complex dependencies that pose additional difficulties with respect to testing. Our timer's state machine model, e.g., expects implementations of the interfaces `TimeModel`, `ClockModel`, and `TimerUIUpdateListener` to be present. We can achieve this by manually implementing a so-called *mock object*[8] that unifies these three dependencies of the timer state machine model, corresponding to the three interfaces this mock object implements.

[8] There are also various mocking frameworks, such as Mockito and JMockit, which can automatically generate mock objects that represent component dependencies from interfaces and provide APIs or domain-specific languages for specifying test expectations.

```
1  class UnifiedMockDependency
2  implements TimeModel, ClockModel, TimerUIUpdateListener {
3
4    private int timeValue = -1, stateId = -1;
5    private int runningTime = -1, idleTime = -1;
6    private boolean started = false, ringing = false;
7
8    public int     getTime()   { return timeValue; }
9    public int     getState()  { return stateId; }
10   public boolean isStarted() { return started; }
11   public boolean isRinging() { return ringing; }
12
13   @Override public void updateTime(final int tv) {
14     this.timeValue = tv;
15   }
16   @Override public void updateState(final int stateId) {
17     this.stateId = stateId;
18   }
19   @Override public void ringAlarm(final boolean b) {
20     ringing = b;
21   }
22
23   @Override public void setClockListener(
24       final ClockListener listener) {
25     throw new UnsupportedOperationException();
26   }
27   @Override public void startTick(final int period) {
28     started = true;
29   }
30   @Override public void stopTick() { started = false; }
31   @Override public void restartTimeout(final int period) { }
32
33   @Override public void reset() { runningTime = 0; }
34   @Override public void inc() {
35     if (runningTime != 99) { runningTime++; }
36   }
37   @Override public void dec() {
38     if (runningTime != 0) { runningTime--; }
39   }
40   @Override public int  get() { return runningTime; }
41 }
```

The instance variables and corresponding getter methods enable us to test whether the SUT produced the expected state changes in the mock object. The three remaining blocks of methods correspond to the three implemented interfaces, respectively.

Now we can write tests to verify actual scenarios. In the following scenario, we start with time 0, press the button once, expect time 1, press the button 198 times (the max time is 99), expect time 99, produce a timeout event, check if running, wait 50 s, expect time 49 (99–50), wait 49 s, expect time 0, check if ringing, wait 3 more seconds (just in case), check if still ringing, press the button to stop the ringing, and make sure the ringing has stopped and we are in the stopped state.

```
1  @Test
2  public void testScenarioRun2() {
3    assertEquals(R.string.STOPPED, dependency.getState());
4    model.onButtonPress();
5    assertTimeEquals(1);
6    assertEquals(R.string.STOPPED, dependency.getState());
7    onButtonRepeat(MAX_TIME * 2);
8    assertTimeEquals(MAX_TIME);
9    model.onTimeout();
10   assertEquals(R.string.RUNNING, dependency.getState());
11   onTickRepeat(50);
12   assertTimeEquals(MAX_TIME - 50);
13   onTickRepeat(49);
14   assertTimeEquals(0);
15   assertEquals(R.string.RINGING, dependency.getState());
16   assertTrue(dependency.isRinging());
17   onTickRepeat(3);
18   assertEquals(R.string.RINGING, dependency.getState());
19   assertTrue(dependency.isRinging());
20   model.onButtonPress();
21   assertFalse(dependency.isRinging());
22   assertEquals(R.string.STOPPED, dependency.getState());
23 }
```

Note that this happens in *"fake time"* (fast-forward mode) because we can make the rate of the clock ticks as fast as the state machine can keep up.

Programmatic System Testing of the App

The following is a system test of the application with all of its real component present. It verifies the following scenario in *real time*: time is 0, press button five times, expect time 5, wait 3 s, expect time 5, wait 3 more seconds, expect time 2, press *stopTick* button to reset time, and expect time 0. This test also includes the effect of all state transitions as assertions.

```
1  @Test
2  public void testScenarioRun2() throws Throwable {
3    getActivity().runOnUiThread(new Runnable() {
4      @Override public void run() {
5        assertEquals(STOPPED, getStateValue());
6        assertEquals(0, getDisplayedValue());
7        for (int i = 0; i < 5; i++) {
8          assertTrue(getButton().performClick());
9        }
10     }
11   });
12   runUiThreadTasks();
13   getActivity().runOnUiThread(new Runnable() {
14     @Override public void run() {
15       assertEquals(5, getDisplayedValue());
```

```
16       }
17     });
18     Thread.sleep(3200); // <-- do not run this in the UI thread!
19     runUiThreadTasks();
20     getActivity().runOnUiThread(new Runnable() {
21       @Override public void run() {
22         assertEquals(RUNNING, getStateValue());
23         assertEquals(5, getDisplayedValue());
24       }
25     });
26     Thread.sleep(3200);
27     runUiThreadTasks();
28     getActivity().runOnUiThread(new Runnable() {
29       @Override public void run() {
30         assertEquals(RUNNING, getStateValue());
31         assertEquals(2, getDisplayedValue());
32         assertTrue(getButton().performClick());
33       }
34     });
35     runUiThreadTasks();
36     getActivity().runOnUiThread(new Runnable() {
37       @Override public void run() {
38         assertEquals(STOPPED, getStateValue());
39       }
40     });
41 }
```

As in section "Understanding User Interaction as Events", we can run this test as an in-container instrumentation test or out-of-container using a simulated environment such as Robolectric.

During testing, our use of threading should mirror that of the SUT: The button press events we simulate using the `performClick` method have to run on the UI thread of the simulated environment. While the UI thread handles these events, we use `Thread.sleep` in the main thread of the test runner to wait in pseudo-real-time, much like the user would wait and watch the screen update.

Robolectric queues tasks scheduled on the UI thread until it is told to perform these. Therefore, we must invoke the `runUiThreadTasks` method *before* attempting our assertions on the UI components.

Keeping the User Interface Responsive with Asynchronous Activities

In this section, we explore the issues that arise when we use a GUI to control long-running, processor-bound activities. In particular, we'll want to make sure the GUI stays responsive even in such scenarios and the activity supports progress reporting and cancelation. Our running example will be a simple app for checking whether a number is prime. The complete code for this example is available online [19].

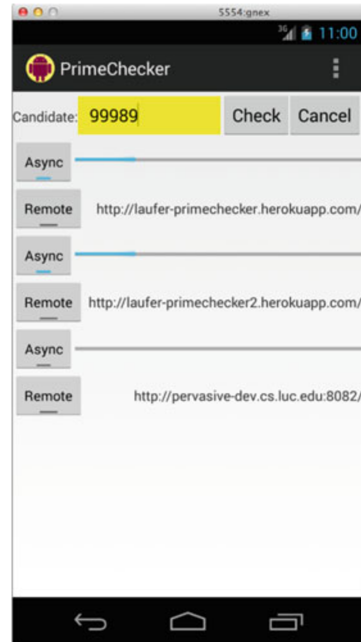

Fig. 11 Screenshot of an Android app for checking prime numbers

The Functional Requirements for the Prime Checker App

The functional requirements for this app are as follows:

- The app allows us to enter a number in a text field.
- When we press the *check* button, the app checks whether the number we entered is prime.
- If we press the *cancel* button, any ongoing check(s) are discontinued.

Figure 11 shows a possible UI for this app.

To check whether a number is prime, we can use this iterative brute-force algorithm.

```
1 protected boolean isPrime(final long i) {
2   if (i < 2) return false;
3   final long half = i / 2;
4   for (long k = 2; k <= half; k += 1) {
5     if (isCancelled() || i % k == 0) return false;
6     publishProgress((int) (k * 100 / half));
7   }
8   return true;
9 }
```

For now, let's ignore the isCancelled and updateProgress methods and agree to discuss their significance later in this section.

While this is not an efficient prime checker implementation, this app will allow us to explore and discuss different ways to run one or more such checks. In particular, the fact that the algorithm is heavily processor-bound makes it an effective running example for discussing whether to move such activities to the background (or remote servers).

The Problem with Foreground Tasks

As a first attempt, we now can run the isPrime method from within our event listener in the current thread of execution (the main GUI thread).

```
1   final PrimeCheckerTask t =
2     new PrimeCheckerTask(progressBars[0], input);
3   localTasks.add(t);
4   t.onPreExecute();
5   final boolean result = t.isPrime(number);
6   t.onPostExecute(result);
7   localTasks.clear();
```

The methods onPreExecute and onPostExecute are for resetting the user interface and displaying the result.

As shown in Table 1 below, response times (in seconds) are negligible for very small numbers but increase roughly linearly. "$\ll$1" means no noticeable delay, and "$*$" means that the test was canceled before it completed.

The actual execution targets for the app or isPrime implementation are

- Samsung Galaxy Nexus I9250 phone (2012 model): dual-core 1.2 GHz Cortex-A9 ARM processor with 1 GB of RAM (using one core)
- Genymotion x86 Android emulator with 1 GB of RAM and one processor running on a MacBook Air
- MacBook Air (mid-2013) with 1.7 GHz Intel Core i7 and 8 GB of RAM
- Heroku free plan with one web dyno with 512 MB of RAM

For larger numbers, the user interface on the device freezes noticeably while the prime number check is going on, so it does not respond to pressing the cancel button. There is no progress reporting either: The progress bar jumps from zero to 100 when the check finishes. In the UX (user experience) world, any freezing for more than a fraction of a second is considered unacceptable, especially without progress reporting.

Table 1 Response times for checking different prime numbers on representative execution targets

Execution target prime	Phone	Emulator	Computer	Web service
1013	$\ll 1$	$\ll 1$	$\ll 1$	$\ll 1$
10007	1	$\ll 1$	$\ll 1$	$\ll 1$
100003	3	1	$\ll 1$	$\ll 1$
1000003	27	6	$\ll 1$	1
10000169	$*$	60	2	2
100000007	$*$	$*$	8	8

Reenter the Single-Threaded User Interface Model

The behavior we are observing is a consequence of the single-threaded execution model underlying Android and similar GUI frameworks. As discussed in section "Thread Safety in GUI Applications: The Single-Threaded Rule", in this design, all UI events, including user inputs such as button presses and mouse moves, outputs such as changes to text fields, progress bar updates, and other component repaints, and internal timers, are processed sequentially by a single thread, known in Android as the *main thread* (or UI thread). We will continue to say *UI thread* for clarity.

To process an event completely, the UI thread needs to dispatch the event to any event listener(s) attached to the event source component. Accordingly, single-threaded UI designs typically come with two rules:

1. To ensure responsiveness, code running on the UI thread must never block.
2. To ensure thread-safety, only code running on the UI thread is allowed to access the UI components.

In interactive applications, running for a long time is almost as bad as blocking indefinitely on, say, user input. To understand exactly what is happening, let's focus on the point that events are processed sequentially in our scenario of entering a number and attempting to cancel the ongoing check.

- The user enters the number to be checked.
- The user presses the check button.
- To process this event, the UI thread runs the attached listener, which checks whether the number is prime.
- While the UI thread running the listener, all other incoming UI events—pressing the cancel button, updating the progress bar, changing the background color of the input field, etc.—are *queued* sequentially.
- Once the UI thread is done running the listener, it will process the remaining events on the queue. At this point, the cancel button has no effect anymore, and we will instantly see the progress bar jump to 100% and the background color of the input field change according to the result of the check.

So why doesn't Android simply handle incoming events concurrently, say, each in its own thread? The main reason not to do this is that it greatly complicates the design while at the same time sending us back to square one in most scenarios: Because the UI components are a shared resource, to ensure thread safety in the presence of race conditions to access the UI, we would now have to use mutual exclusion in every event listener that accesses a UI component. Because that is what event listeners typically do, in practice, mutual exclusion would amount to bringing back a sequential order. So we would have greatly complicated the whole model without effectively increasing the extent of concurrency in our system (see also section "Thread Safety in GUI Applications: The Single-Threaded Rule" above).

There are two main approaches to keeping the UI from freezing while a long-running activity is going on.

Breaking Up an Activity Into Small Units of Work

The first approach is still single-threaded: We break up the long-running activity into very small units of work to be executed directly by the UI thread. When the current chunk is about to finish, it schedules the next unit of work for execution on the UI thread. Once the next unit of work runs, it first checks whether a cancelation request has come in. If so, it simply will not continue, otherwise it will do its work and then schedule its successor. This approach allows other events, such as reporting progress or pressing the cancel button, to get in between two consecutive units of work and will keep the UI responsive as long as each unit executes fast enough.

Now, in the same scenario as above—entering a number and attempting to cancel the ongoing check—the behavior will be much more responsive:

- The user enters the number to be checked.
- The user presses the check button.
- To process this event, the UI thread runs the attached listener, which makes a little bit of progress toward checking whether the number is prime and then schedules the next unit of work on the event queue.
- Meanwhile, the user has pressed the cancel button, so this event is on the event queue *before* the next unit of work toward checking the number.
- Once the UI thread is done running the first (very short) unit of work, it will run the event listener attached to the cancel button, which will prevent further units of work from running.

Asynchronous Tasks to the Rescue

The second approach is typically multi-threaded: We represent the entire activity as a separate asynchronous task. Because this is such a common scenario, Android provides the abstract class `AsyncTask` for this purpose.

```
1 public abstract class AsyncTask<Params, Progress, Result> {
2   protected void onPreExecute() { }
3   protected abstract Result doInBackground(Params... params);
4   protected void onProgressUpdate(Progress... values) { }
5   protected void onPostExecute(Result result) { }
6   protected void onCancelled(Result result) { }
7   protected final void publishProgress(Progress... values) {
8     ...
9   }
10  public final boolean isCancelled() { ... }
11  public final AsyncTask<...> executeOnExecutor(
12      Executor exec, Params... ps) {
13    ...
14  }
15  public final boolean cancel(boolean mayInterruptIfRunning) {
16    ...
17  }
18 }
```

The three generic type parameters are `Params`, the type of the arguments of the activity; `Progress`, they type of the progress values reported while the activity runs in the background, and `Result`, the result type of the background activity. Not all three type parameters have to be used, and we can use the type `Void` to mark a type parameter as unused.

When an asynchronous task is executed, the task goes through the following lifecycle:

- `onPreExecute` runs on the UI thread and is used to set up the task in a thread-safe manner.
- `doInBackground(Params...)` is an abstract template method that we override to perform the desired task. Within this method, we can report progress using `publishProgress(Progress...)` and check for cancelation attempts using `isCancelled()`.
- `onProgressUpdate(Progress...)` is scheduled on the UI thread whenever the background task reports progress and runs whenever the UI thread gets to this event. Typically, we use this method to advance the progress bar or display progress to the user in some other form.
- `onPostExecute(Result)` receives the result of the background task as an argument and runs on the UI thread after the background task finishes.

Using AsyncTask in the Prime Number Checker

We set up the corresponding asynchronous task with an input of type `Long`, progress of type `Integer`, and result of type `Boolean`. In addition, the task has access to the progress bar and input text field in the Android GUI for reporting progress and results, respectively.

The centerpiece of our solution is to invoke the `isPrime` method from the main method of the task, `doInBackground`. The auxiliary methods `isCancelled` and `publishProgress` we saw earlier in the implementation of `isPrime` are for checking for requests to cancel the current task and updating the progress bar, respectively. `doInBackground` and the other lifecycle methods are implemented here:

```
1 @Override protected void onPreExecute() {
2   progressBar.setMax(100);
3   input.setBackgroundColor(Color.YELLOW);
4 }
5
6 @Override protected Boolean doInBackground(
7     final Long... params) {
8   if (params.length != 1)
9     throw new IllegalArgumentException(
10      "exactly one argument expected");
11  return isPrime(params[0]);
12 }
13
14 @Override protected void onProgressUpdate(
15     final Integer... values) {
16   progressBar.setProgress(values[0]);
17 }
18
19 @Override protected void onPostExecute(final Boolean result) {
20   input.setBackgroundColor(result ? Color.GREEN : Color.RED);
21 }
22
23 @Override protected void onCancelled(final Boolean result) {
24   input.setBackgroundColor(Color.WHITE);
25 }
```

When the user presses the cancel button in the UI, any currently running tasks are canceled using the control method `cancel(boolean)`, and subsequent invocations of `isCancelled` return `false`; as a result, the `isPrime` method returns on the next iteration.

How often to check for cancelation attempts is a matter of experimentation: Typically, it is sufficient to check only every so many iterations to ensure that the task can make progress on the actual computation. Note how this design decision is closely related to the granularity of the units of work in the single-threaded design discussed in section "Thread Safety in GUI Applications: The Single-Threaded Rule" above.

Execution of Asynchronous Tasks in the Background

So far, we have seen how to define background tasks as subclasses of the abstract framework class `AsyncTask`. Actually executing background tasks arises as an

orthogonal concern with the following strategies to choose from for assigning tasks to worker threads:

- *Serial executor:* Tasks are queued and executed by a single background thread.
- *Thread pool executor:* Tasks are executed concurrently by a pool of background worker threads. The default thread pool size depend on the available hardware resources; a typical pool size even for a single-core Android device is two.

In our example, we can schedule `PrimeCheckerTask` instances on a thread pool executor:

```
1   final PrimeCheckerTask t =
2     new PrimeCheckerTask(progressBars[i], input);
3   localTasks.add(t);
4   t.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, number);
```

This completes the picture of moving processor-bound, potentially long-running activities out of the UI thread but in a way that they can still be controlled by the UI thread.

Additional considerations apply when targeting symmetric multi-core hardware (SMP), which is increasingly common among mobile devices. While the application-level, coarse-grained concurrency techniques discussed in this chapter still apply to multi-core execution, SMP gives rise to more complicated low-level memory consistency issues than those discussed above in section "Fundamentals of Thread Safety". An in-depth discussion of Android app development for SMP hardware is available here [12].

Summary

In this chapter, we have studied various parallel and distributed computing topics from a user-centric software development perspective. Specifically, in the context of mobile application development, we have studied the basic building blocks of interactive applications in the form of events, timers, and asynchronous activities, along with related software modeling, architecture, and design topics.

The complete source code for the examples from this chapter, along with instructions for building and running these examples, is available from [17]. For further reading on designing concurrent object-oriented software, please have a look at [7, 22, 23, 28].

Acknowledgements We are grateful to our former graduate students Michael Dotson and Audrey Redovan for having contributed their countdown timer implementation, and to our colleague Dr. Robert Yacobellis for providing feedback on this chapter and trying these ideas in the classroom.

We are also grateful to the anonymous CDER reviewers for their helpful suggestions.

References

1. Kevin Ashton. That 'Internet of Things' thing. RFID Journal, http://www.rfidjournal.com/articles/view?4986, July 2009. Accessed: 2016-12-09.
2. Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.
3. Google Webmaster Central Blog. Rolling out the mobile-friendly update. https://webmasters.googleblog.com/2015/04/rolling-out-mobile-friendly-update.html, April 2015. Accessed: 2016-12-12.
4. Stefano Borini. Understanding model-view-controller. https://www.gitbook.com/book/stefanoborini/modelviewcontroller, 2016. Accessed: 2016-12-09.
5. Jemma Brackebush. How mobile is overtaking desktop for global media consumption, in 5 charts. Digiday, http://digiday.com/publishers/mobile-overtaking-desktops-around-world-5-charts/, June 2016. Accessed: 2016-12-10.
6. Jason H. Christensen. Using restful web-services and cloud computing to create next generation mobile applications. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 627–634, New York, NY, USA, 2009. ACM.
7. Thomas W. Christopher and George K. Thiruvathukal. *High Performance Java Platform Computing*. Prentice Hall PTR, Upper Saddle Ridge, NJ, 2000.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
9. Google. Android developer reference. http://developer.android.com/develop/, 2009–2018. Accessed: 2016-12-09.
10. Google. Memory consistency properties. https://developer.android.com/reference/java/util/concurrent/package-summary.html#MemoryVisibility, 2009–2018. Accessed: 2016-12-09.
11. Google. Processes and threads. http://developer.android.com/guide/components/processes-and-threads.html, 2009–2018. Accessed: 2016-12-09.
12. Google. SMP primer for Android. http://developer.android.com/training/articles/smp.html, 2009–2018. Accessed: 2016-12-09.
13. Google. Android Studio: Use Java 8 language features. https://developer.android.com/studio/write/java8-support.html#supported_features, 2017. Accessed: 2018-02-05.
14. The Guardian. Mobile web browsing overtakes desktop for the first time. https://www.theguardian.com/technology/2016/nov/02/mobile-web-browsing-desktop-smartphones-tablets/, November 2016. Accessed: 2016-12-10.
15. Graham Hamilton. JavaBeans specification. Technical report, Sun Microsystems, inc, 1997.
16. Per Brinch Hansen. *Operating System Principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973.
17. Konstantin Läufer, George K. Thiruvathukal, and Robert H. Yacobellis. Loyola University Chicago Computer Science COMP 313/413 course examples. https://github.com/lucoodevcourse/, 2012–2018.
18. Konstantin Läufer, George K. Thiruvathukal, and Robert H. Yacobellis. Loyola University Chicago Computer Science COMP 313/413 course examples: Click counter. https://github.com/lucoodevcourse/clickcounter-android-java, 2012–2018.
19. Konstantin Läufer, George K. Thiruvathukal, and Robert H. Yacobellis. Loyola University Chicago Computer Science COMP 313/413 course examples: Prime number checker. https://github.com/lucoodevcourse/primenumbers-android-java, 2012–2018.
20. Konstantin Läufer, George K. Thiruvathukal, and Robert H. Yacobellis. Loyola University Chicago Computer Science COMP 313/413 course examples: Simple threads. https://github.com/lucoodevcourse/simplethreads-java, 2012–2018.
21. Konstantin Läufer, George K. Thiruvathukal, and Robert H. Yacobellis. Loyola University Chicago Computer Science COMP 313/413 course examples: Stopwatch. https://github.com/lucoodevcourse/stopwatch-android-java, 2012–2018.

22. Doug Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.
23. Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.
24. Robert C. Martin and Micah Martin. *Agile Principles, Patterns, and Practices in C# (Robert C. Martin)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.
25. Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.
26. Brad A. Myers. A brief history of human-computer interaction technology. *interactions*, 5(2):44–54, March 1998.
27. Oracle. Java platform, standard ed. 8 API specification: Class EventQueue. http://docs.oracle.com/javase/8/docs/api/java/awt/EventQueue.html, 1993–2018. Accessed: 2016-12-09.
28. Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.
29. James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.
30. Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2000.
31. Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.