Com S 430
Spring 2018
Homework 3

General instructions

- When submitting modifications of existing code, **please do not reformat any sections of the code that you did not explicitly have to change**. We need to be able to diff your submission against the original to see exactly what was changed.
- Submit an archive on Canvas containing all classes originally posted as hw3.zip.

1) Modify `Histogram` so that the work is done by multiple threads. You should be able to observe a speedup that is nearly proportional to the number of threads (up to the number of cores you have, that is). One question to consider concerns the issue of shared access to the array 'result'. In theory, all threads could share it, since it isn't ever being modified. You may find that doing so impedes performance when the array is large. One common is to give each thread its own local copy of a sub-array, then aggregate the results at the end.

2) *Based on a true story*. An IDE such as Eclipse includes a feature that will provide an "outline view" of a source file when a node in the explorer pane is clicked. The outline is based on a model which is constructed from the abstract syntax tree produced by a parser. Similarly, the syntax tree might have to be constructed if the file is opened for editing. Constructing the parser and performing the parse are relatively expensive operations, so the syntax tree is only constructed the first time it is needed. Upon construction, the syntax tree is kept in a cache for reuse until the file is modified. (In real life, this has to be a least-recently-used cache of bounded size, but we can ignore that issue for now.)

Note that various threads may need access to the syntax tree, (e.g. the editor or the event handler for expanding the outline view) so the cache has to be thread-safe. (The syntax trees themselves can be considered to be effectively immutable.)

The sample code `SyntaxTreeManager` is a bare-bones implementation of a class for managing the store of already-constructed syntax trees. It suffers from the major defect that if one thread needs the tree for a file f that is already parsed while another thread is busy parsing a different file g, the first thread is blocked until the parse of g is finished.

Modify the implementation to address the following. See the class `Test` for a bit of test code.

1. Allow access to the cache and parsing to run concurrently, in the sense that a thread may get an existing syntax tree or initiate construction of a new one while other threads are parsing different files.

2. Make sure that two threads don't parse the same file at the same time: if one thread is already parsing a file, the second thread should block until the first thread completes, and then return the result.

Before you begin, **read JCIP section 5.6**, which solves a *very* similar problem. You do not have to follow exactly what the author does; you could consider making the `SyntaxTreeHolder` into a Future-like object, or you could completely replace it with a `FutureTask`.

Note that the operation of getting a syntax tree, which is now potentially blocking, should be *interruptible*. That is, you can't just use a sync lock on **`SyntaxTreeHolder`**.

3) The **`components`** package contains the basic ingredients for an actor-style component framework based on one-way message passing, as recently discussed in class. The basic rules of this design are that:

- Each component is single-threaded.
- A component only communicates with other components by passing messages.
- Sending a message never causes the sender to block.

In this package, there is a base **`Component`** type and a few different kinds of messages. The functionality of the Main program is like the client from Homework 2. If you run the **`SimpleServer`** from Homework 2 and then connect to it with the Main program provided here, it should work more or less like the client from in Homework 2.

You will need to the given modify code as in (a) and (b):

a) Implement the **`TimerComponent`** to appropriately handle **`SetTimeoutMessages`** and to send out **`TimeoutMessages`**. A **`SetTimeoutMessage`** contains a timeout value in milliseconds and an additional attribute called the "original ID". After receiving a **`SetTimeoutMessage`**, the **`TimerComponent`** should send a reply of type **`TimeoutMessage`** to the sender after the given number of milliseconds have elapsed.

The correlation ID in the **`TimeoutMessage`** should be the "original ID" attribute provided in the **`SetTimeoutMessage`**, rather than the ID of the **`SetTimeoutMessage`**. This choice may seem odd at first, but it is helpful when your component needs to be able to mark whether some task which it started takes too long and "times out".

Here is an example of a scenario in which having the "original ID" is useful:

- A component sends a message with some ID x (triggering some task that is, logically, only "relevant" for some period of time).
- This component saves a copy of the message, keyed on (i.e. saved under) x.
- This component then uses the TimerComponent to set a timeout for receiving a reply to message x.
- If a timeout occurs, the correlation ID for the TimeoutMessage will be x.
- Since the component saved a copy of the original message with ID x using the value x, it is easy for this component to find this x in its internal data structures and to mark it as timed-out.
- If a message correlated with x arrives at the component before a timeout has occurred, then the component knows that the task is still "relevant" and hasn't "timed out". Otherwise, the task has timed out and the message correlated with x should be ignored.

This is just one way of using a **`TimerComponent`**. The designer of a component using **`TimerComponent`** would prefer to get some other value as the correlation ID, it can set the originalID field any way that it wants.

Note that the `TimerComponent` does not have any built-in provision for sending periodic messages, but a component can get the same effect: if a component wants a periodic timeout, it can just re-send another `SetTimeoutMessage` whenever it receives a relevant `TimeoutMessage`.

There are a number of ways to implement `TimerComponent` so that it can both receive messages and also "wake-up" when necessary to send out `TimeoutMessages` at the right time. **Just remember the rule that a component has to be single-threaded**. This implies, for example, that you can't have a thread in a message-handling loop (as in `ThreadedComponent`), *and also* use `java.util.Timer`, because `java.util.Timer` starts another thread. A good design hint, in fact, is to *not* try to extend `ThreadedComponent` here. One idea that could work: use a single instance of `java.util.Timer`, and have the `send()` method schedule a new `TimerTask` with the appropriate delay. Similarly, you could use a `ScheduledThreadPoolExecutor` with one thread. Another idea: suppose you maintain a separate data structure containing timeout messages to be sent, and that the soonest message you need to send is t milliseconds in the future: then when reading from your input queue, call `poll()` with a t millisecond timeout. Another option is something like `java.util.concurrent.DelayQueue`.

b) Modify the `ClientComponent` class so that it sets a timeout for each request. Make `ClientComponent` display a message such as "request for id 42 timed out" if the result is not received from the server before this timeout.

Notes:

- The `ResultMessage` does not include the "key". This is intentional. The "key" represents part of the Client's local context that must be saved and later restored when the result message arrives. Keep track of requests that have been sent, and use the correlation ID to look them up.

- If you are confused by the double dispatch mechanism, I promise that you are not alone. This pattern was discussed in lecture, but please see the week 6 examples if you need a review or clarification. This pattern is tangential to the general topics of the course, but it is a useful pattern to know. (It is also known as the *visitor* pattern.) Note, however, that *you don't actually need to understand it* to do the homework, as long as you precisely follow these rules whenever you define a new type of message. Suppose you define a new type `FooMessage`:

1. include a `dispatch()` method in the `FooMessage` class
2. add a default `handleFoo()` method to the base `Component` class
3. add a component-specific `handleFoo()` method in any component that plans to receive a `FooMessage.`

4) Modify the contents of the `yahtzee` package to implement a "Yahtzee Cube". (Notice that the provided code duplicates some of the code in the `components` package which we saw in Problem 3 above.)

Your cubes must be designed in the following way:

- All cubes run identical code.
- Cubes should never send messages to each other directly; cubes should only
- communicate via broadcasts to the `Universe`.
- Cubes update their state based on messages broadcasted from the left and right neighbors,

transitioning through the phases described below

The Universe

Since we do not have actual physical cubes with radio transmitters, the provided code includes a class **`Universe`** to simulate the physical environment and a **`UniverseUI`** for us to view and interact with it. The **`Universe`** keeps track of each cube's location and simulates the radio transmissions. In particular, the methods

```
Universe#broadcastLeft(IMessage)
Universe#broadcastRight(IMessage)
```

are called by a cube to attempt to send a message to that cube's current left neighbors or its right neighbors, respectively. The **`Universe`** checks which (if any) cubes are within range of a broadcasting cube. If a neighboring cube is in range, the **`Universe`** invokes **`send()`** on that neighbor to deliver the message.

The **`Universe`** also simulates a way for each cube's display to be visible in the UI:

```
Universe.updateDisplay(String text)
```

The class **`Universe`** is fully implemented along with **`UniverseUI`**. Currently, when you start the application by running **`Universe`**, it allow you to see the cubes and to move them around with the mouse. However, they won't actually ever change their displays appropriately until you provide the missing functionality.

To do

The provided code includes a minimal skeleton for a **`Cube`** class and for **`TimerComponent`**, just so that everything will compile.

Start by copying over your **`TimerComponent`** implementation from Problem 3. Then define whatever additional message types you need to implement the **`Cube`** behavior.

The cubes should periodically recheck the status of their left and right neighbors by attempting to send some kind of a "ping" message to the left and to the right. A cube receiving a ping should normally broadcast a reply with a message conveying whatever information might be needed from the neighbors. If there is no ping response within a specified timeout period, the original sending cube can deduce that there is no neighbor to that side. Both the polling period and timeout value should be easily configurable; use the constants defined in the **`Cube`** skeleton. It is up to you to design the ping message and response types.

Cube Behavior

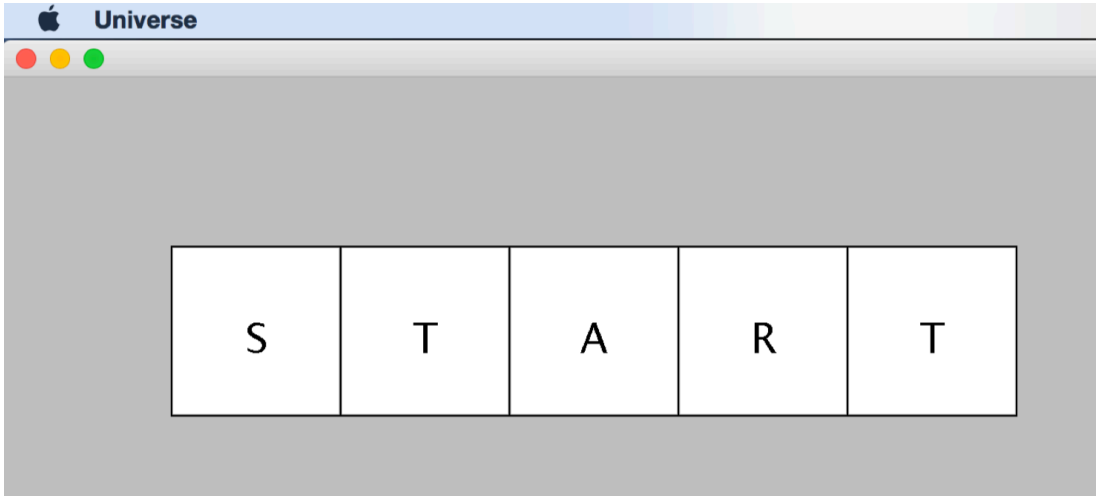
We are definitely not going to attempt to implement a whole Yahtzee game in this homework, but we can do the following as a start.

The behavior of a cube depends upon its position relative to other cubes, and upon the *phase* that it is in. A cube's phase proceeds in the sequence described below through a "starting", "generating", "scoring", and "done" phase. (Note there is an **`enum`** type **`Status`** might be useful,

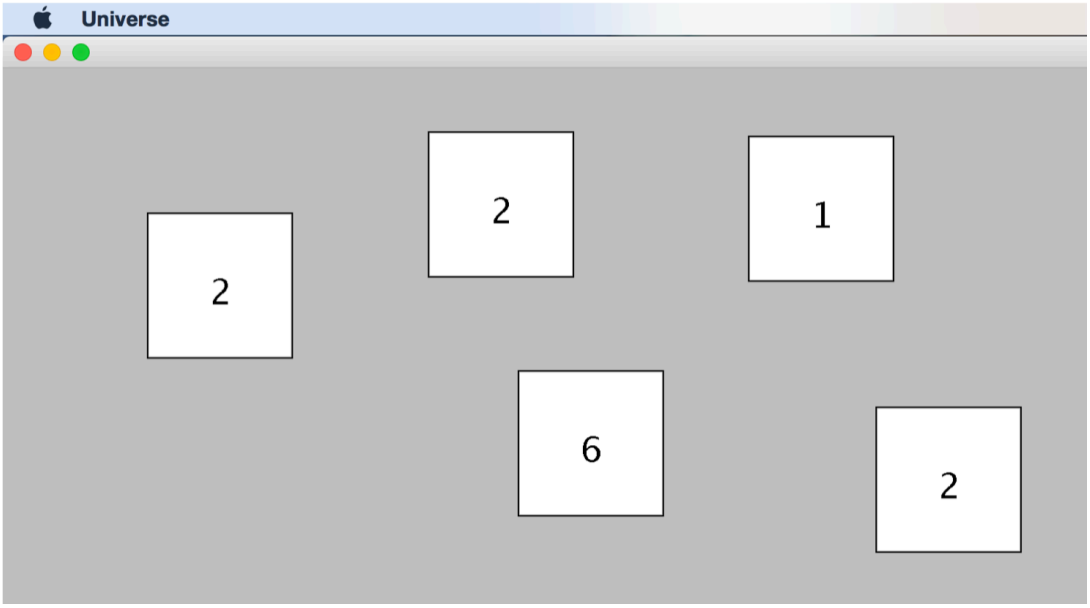
for modeling phases, but you are not required to use it.)

Initially each cube is in a "starting" phase, where the cubes are trying to spell out "START". Initially, they should all display a question mark, "?". As they are lined up into a group, their displays should spell out "S", "T", "A", "R", and "T" based on their relative positions within the group.

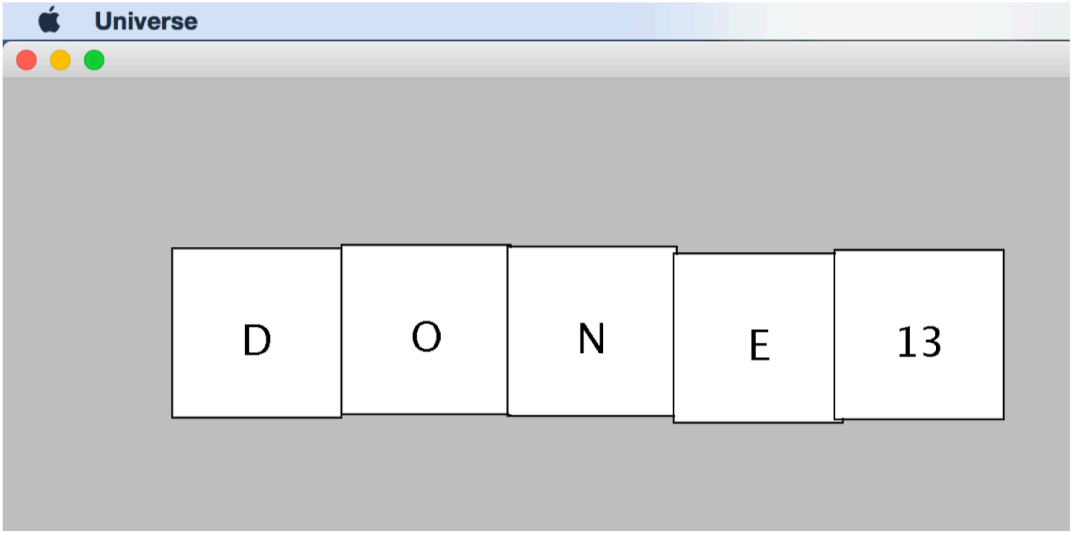
Note that a cube might be added to the lineup and then removed before fully spelling out "START". Your cubes should be responsive to such removals: isolated cubes should revert to "?"; any line of fewer than 5 cubes should only spell out a prefix of "START". (Make sure you handle the case where an "S" cube is removed from a line of cubes.)



Once the cubes have successfully spelled out "START", they should transition into a "generating" phase: as soon as any of the five cubes is moved apart from the others, it should generate and display a random number between 1 and 6 (inclusive).



After that, the cube goes into a "scoring" phase. When the cubes are lined up together again, each cube determines the sum of all the cubes' values, and then proceeds to a "done" phase. The rightmost cube displays the sum, and the remaining cubes spell out the word "DONE".



Finally, if a cube has gone through the "done" phase and is then moved apart from its neighbors, it returns to the "starting" phase.

When the correct string to display cannot be determined by a cube (e.g., what if it's in the "done" phase but not all neighbors are lined up yet) it should display a question mark.

If you missed class the day we demoed the cubes, you can find various demonstrations on YouTube by searching for "Yahtzee Flash", e.g. https://www.youtube.com/watch?v=bwujJDWL4E8.