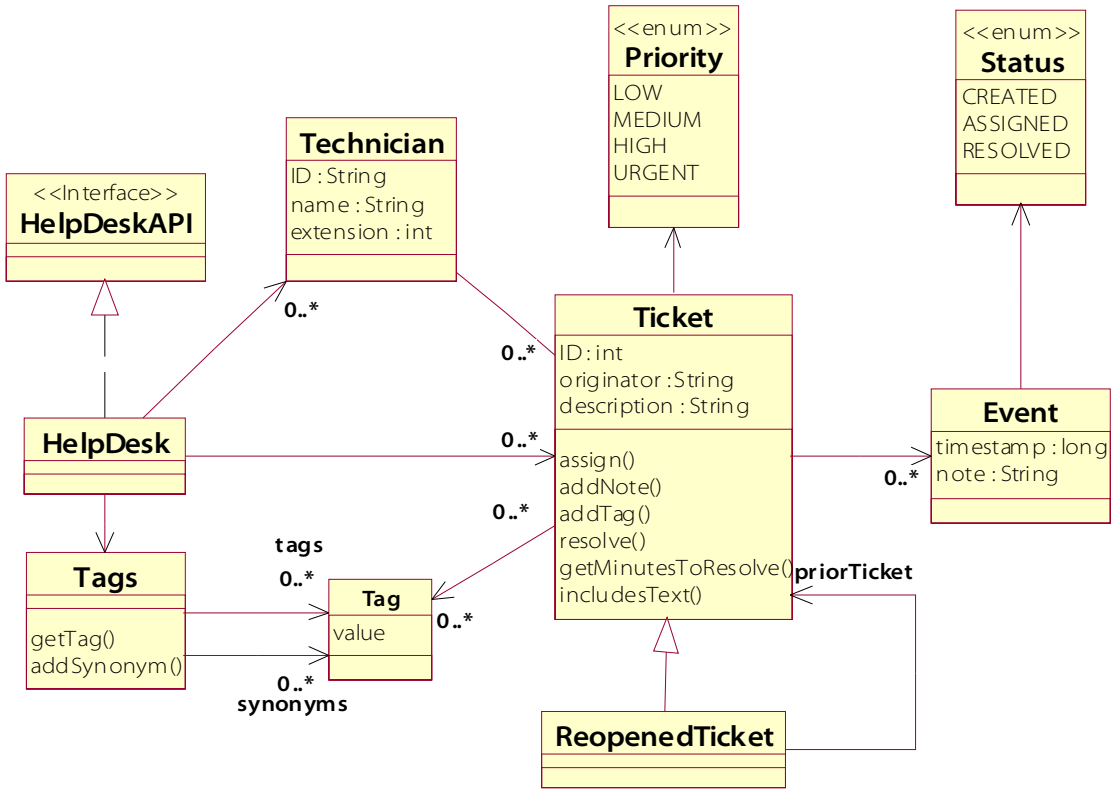
HelpDesk – Implementation

You'll implement the system, in phases. Sections later in this document provide further detail on each phase, but here is a summary:

1. HelpDesk and tickets
2. History of events per ticket
3. Assignment of tickets to technicians
4. Adding tags to tickets
5. Computing time to resolve tickets
6. Searching for tickets by included text
7. Re-opening tickets
8. Tag synonyms and preferred capitalization

At the end of the allotted time, you'll upload your code to the **CompletedWork** folder, under your name and the subfolder **HelpDesk/HelpDesk1**. You can feel free to upload copies of the code as complete specific phases; if you do this please create subfolders **HelpDesk/PhaseN** to indicate the completed phase number.

You may work with your own, approved design, or adopt the design shown in UML below.



The only hard requirements are that you implement the **HelpDeskAPI** and that you use the supporting types that are provided, as the API interface relies on them: **Ticket** and its two enumerated types **Priority** and **Status**, and **Tag**. The **Ticket** and **Tag** types are otherwise empty classes and, again, you can flesh them out as above or by your own design, as long as the API requirements are met.

There is also a **TestProgram** class that uses the API to drive a test scenario that aligns with the test data provided in **TestData.xlsx**. The **main** method of this class calls a **runSimulation** method to push the test data into the system, and then a series of test methods that query the system and assert expected results. Some of these test methods are fully implemented and some, because they'll ultimately rely on details of your design and implementation and not just on the pre-defined API, are partially implemented or are not implemented at all, and you will fill in test logic as you go.

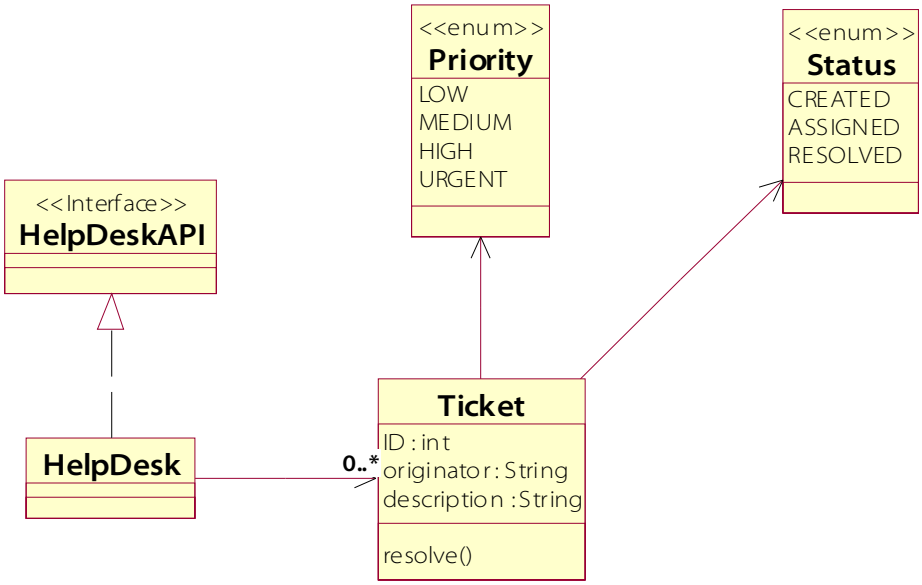
A separate file **TestProgram_Standard.txt** is also provided. This has more complete test logic, but not all of the passages are guaranteed to compile cleanly based solely on the starter types. If you stick closely to the standard design, all of the code here will build and run – or, you may just want to use this file as a reference for coding up your own test methods to your own design.

All test methods are invoked, and of course those that are implemented will fail at the start of the exercise, printing various "ASSERTION FAILED" messages; as you build up your system, part of the goal will be for one after another to pass. In all cases the absence of failure messages means success – no news is good news!

Phase 1: Tickets

In this phase you'll implement requirements #1, #2 (except for history), #3, #6 (except for history), #9, and #10. Your system should pass **test1_Tickets** at the completion of this phase.

In the standard design, this involves classes **HelpDesk**, **Ticket**, **Priority**, and **Status**.



Many of the **HelpDesk** and **Ticket** methods can be empty, or can return a hard-coded value. You are effectively ignoring events that are of any type other than creating and resolving tickets, and you are not yet supporting queries except by ID or status.

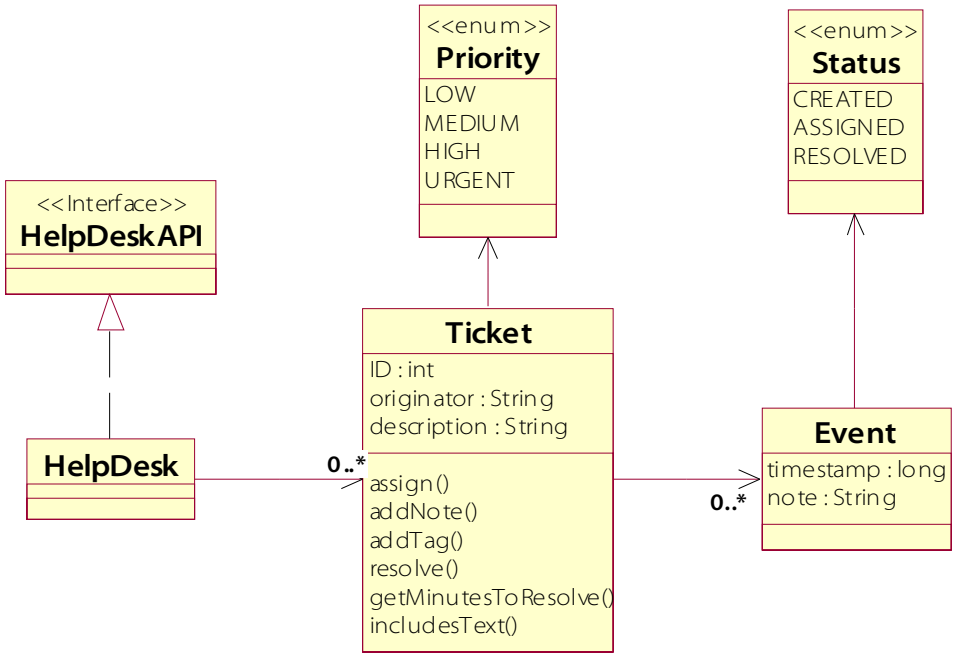
Note that you will write a stop-gap implementation of status for this phase, with a **status** field on the **Ticket** class and a getter that returns the value of that field. You'll automatically set ticket status to Status.ASSIGNED. In Phase 2 you'll implement history, and start to derive status from the history; and in Phase 3 you'll implement actual assignment.

The provided test method proves that you can find tickets by ID and that you're managing status and can query by status. It doesn't dig into the information model of a ticket, so you might want to add code that proves that you're tracking originator and description.

Phase 2: History

In this phase you'll implement requirements #2 (history), #4, and #7. Your system should pass **test2_History** at the completion of this phase – well, and you'll need to put some logic in there that actually tests your system!

In the standard design this means adding the **Event** class, and adding logic to the **Ticket** to generate and to track events.



Some events will have **newStatus** set to represent a state change in the ticket. Notice that the association between **Ticket** and **Status** is gone now, meaning that you will remove the **status** field, and rewrite **getStatus** to report the status of the most up-to-date **newStatus** value in the history. (Note that this is not the same as the status of the most recent event.

There is no specific text formatting required for reporting history. We're focusing on a data model here – not on UI. So your test logic should get one or more tickets, look at their history, and assert that the right values are found: time stamp, note, new status. If you want to print values to the console, develop **toString** methods, etc. for your own purposes, that is fine.

In order to support event timestamps, you could record the system time as each event is created. In practice, since the test program will run in just a few milliseconds, this wouldn't make for an effective simulation as reflected in the test data.

Instead, the provided **Clock** class can be primed with a simulated time by the test program. Instead of, for example, calling **System.currentTimeMillis** or creating a new Date object in order to get the system time, just call **Clock.getTime**, and record that value, which is a **long** number of milliseconds since the start of 1970 – something like this:

```
timestamp= Clock.getTime();
```

Whenever you need to format or print the time of an event, you can call **Clock.format**:

```
Clock.format(timestamp)
```

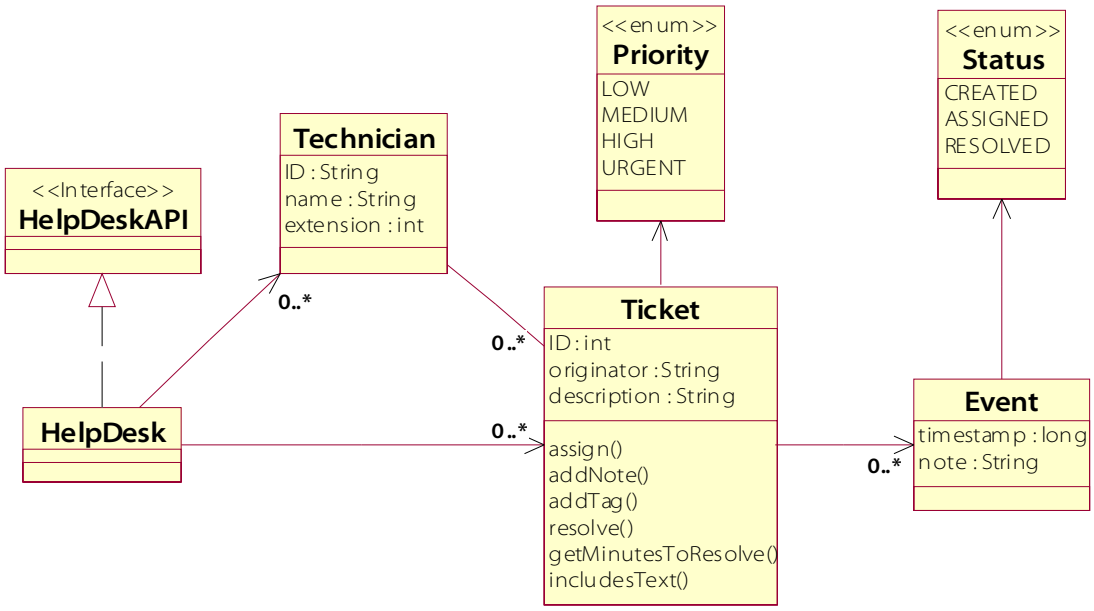
This method returns a string in the format "mm/dd/yyyy h:mm" representing the calendar date and time to the hour and minute, with the hours counted on a 24-hour cycle.

Phase 3: Assignment

In this phase you'll implement requirements #5, #11, and #13. Your system should pass **test3_Assignment** at the completion of this phase.

In the standard design this means adding the **Technician** class; adding logic to **HelpDesk** to build a staff of technicians, to select the next or least-busy technician for a new ticket, and to query tickets by assigned technician; adding logic to **Ticket** to assign to a specific technician and to generate an assignment event.

A **Ticket** will also need to assure that it is added to its assigned **Technician**'s list of active tickets when it is assigned; and to remove itself from the list when it is resolved. Notice a subtle distinction in the information model, which we use for different querying purposes: the **Technician**'s list of active tickets excludes resolved tickets, but even a resolved ticket will retain it's **technician** reference.

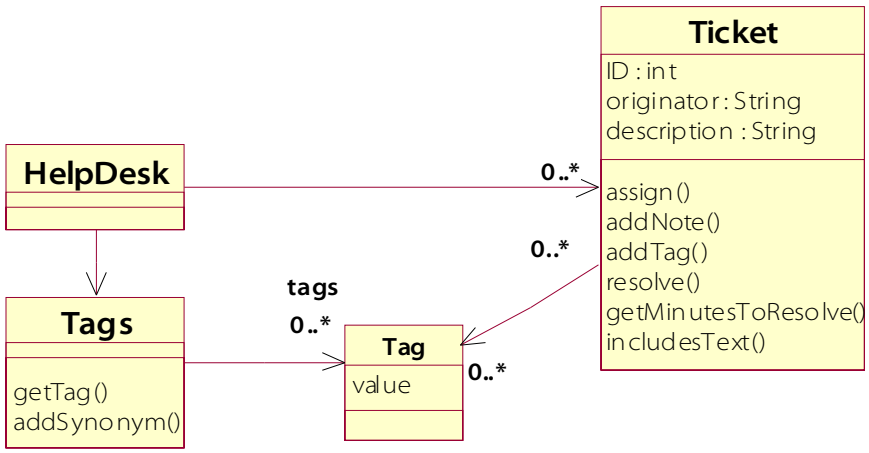


The prepared test method goes some way to prove that you are assigning tickets correctly, taking load into account. You might want to add some logic to check on specific ticket numbers and technicians, and to test requirement #13, ordering of tickets as returned from one or more of the **HelpDesk** query methods.

Phase 4: Tags

In this phase you'll implement requirements #8 and #12. Your system should pass **test4_Tags** at the completion of this phase.

In the standard design this means adding the **Tag** and **Tags** classes, and using it from **HelpDesk** and **Ticket**:



While it would be possible to manage tags as simple strings, and do all of the comparison and coercion to lower case inside the **Ticket** code, it's more robust to develop a separate class; and this class will be much better positioned to support the synonym and capitalization requirements that come up in later phases. For now, **Tag** can just hold a **value** and automatically convert that value to lower case in its constructor; then implement **equals** and **compareTo** to compare the tag **values** in a case-insensitive manner.

Tags holds a, and you can have a **SortedSet** of **Tag** objects, and implements **getTag** to either find an existing, matching tag for a given keyword, or to create one and add it to the set before returning it. (You don't need to implement **addSynonym** for now.

HelpDesk will hold a **Tags** repository, which it can create on its own. When there is a call to add tags to a ticket, call **getTag** to convert each given string to the **Tag** for that string, and then call **addTag** on the ticket with the given ID. **Ticket** will hold its own **SortedSet** of **Tag** objects, and **addTag** can just add to that.

Then implement finding tickets by tag(s), which should be straightforward given the data model you've now set up.

Phase 5: Time to Resolve

In this phase you'll implement requirements #14 and #15. Your system should pass **test5_TimeToResolve** at the completion of this phase. This is mostly just adding a method **getMinutesToResolve** to the **Ticket** class, to calculate the time for one ticket. Then **HelpDesk** must be careful to call this method only on resolved tickets! And take averages based on that sample.

Note the use of a **Map** with string keys (technician ID) and integer values (average time for that technician) as the return type for **HelpDeskAPI.getAverageMinutesToResolvePerTechnician**.

Phase 6: Text Search

In this phase you'll implement requirement #16. Your system should pass **test6_TextSearch** at the completion of this phase. Be sure to implement the core search logic in **Ticket**, in a method **includesText**; then let **HelpDesk** rely on that for its query method. This will put you a better position to extend and override ticket behavior in the following phase.

Phase 7: Reopened Tickets

In this phase you'll implement requirements #17-21. Your system should pass **test7_ReopenedTickets** at the completion of this phase.

In the standard design, the strategy is to extend **Ticket** in the **ReopenedTicket** class, as shown in the full UML diagram earlier in this document.

Add the **priorTicket** field, and override a few methods to meet the requirements, and implement **HelpDeskAPI.reopenTicket** to create an object of the derived type. As per the requirements, objects of this type will mostly behave as normal tickets, but then your overridden methods will jump in where different behavior is expected.

Note that the test method generates a few more events. These are not in the **runSimulation** method because many of the assertions in prior test methods would start to fail, once these re-opened tickets were introduced – even something as simple as ticket count will change. There is some test logic in this method,, but you'll need to add logic to assert correct history and tagging.

Phase 8: Tag Synonyms and Capitalization

In this phase you'll implement requirements #22-23. Your system should pass **test8_Synonyms** at the completion of this phase.

In the standard design, these features are implemented entirely in the **Tag** class, which now holds a set of all known **Tag** instances, along with a map of synonyms and a set of preferred capitalizations, and public methods that allow a caller to populate these "dictionaries".

When creating a tag, the caller must use a factory method getTag, which performs any synonym translation, and either pushes to preferred capitalization or to lower case as a default. The method then either finds an existing **Tag** instance that matches, or creates a new one. The result is something like an enumerated type, but one that allows new values to be added programmatically and can coerce almost-equal values to their correct forms.