Introduction

Please note that we base our code review on the version of Team 1's calendar application that was pushed to Github on Sunday, November 20, at 12:10 am.

Design

The domain model and the separation of responsibilities are organized in a way that allows outsiders to easily understand how the application works and which class is responsible for what. That means: Methods are where you expect them to be, classes interact how you expect them to interact.

The design of inheritance for different sorts of events and the usage of enums for different visibility types provides a solid ground for further extensions and is a good example of code re-using.

However, there are still some points of critique:

Use small controller

Sometimes it seems that the controller takes a bit much responsibility. For example when creating or editing an event the controller checks itself for the validity of the input and does not pass this to the model.

Handling of input errors

The way how input errors are resolved differs between classes and methods. For example in #showRegistration.html and #editProfile the built-in validation is used while in contrast in the #createEvent and #saveEditedEvent methods error handling is done with 'if' blocks.

Exceptions

You should not catch generic exceptions. Making the exception disappear can cause unpredictable behaviour in other parts of the application. Furthermore, most of the time the checked exception mechanism is circumvented in the application. This way failure handling can become hard as exceptions that were not expected are just ignored and the execution continues as if nothing happened. In the higher levels of the application this results in strange behaviour or runtime errors.Consider to catch each exception separately and to split them up in multiple try blocks. An even better solution is to throw a runtime exception, because the code is likely to evolve further in the future. This way a new sort of error falls though all catch blocks and is finally displayed indicating programming errors in modified code.
If the method can do some useful recovery actions, try to fix it making a checked exception for recoverable conditions. A fix action for example can include setting default values or returning to a stable state. Even re-throwing the exception can be regarded as some sort of fix if there is no need to catch the exception at this level or if it cannot be handled in a practical way at this point. If nothing can be done to deal with the error triggered, make it an unchecked exception so higher levels might eventually be able to deal with the exception. The same goes for a subroutine that is part of an external source or cannot be altered. It might be better to throw this exception to higher levels because at this point it cannot be repaired appropriately. Even if it occurs, it is due to a programming fault and handling it lies outside of the responsibility of the method using that subroutine.

Null values

With null as a return value the upper layer is likely to encounter the infamous null pointer exception if it tries to access properties of the object and does not check for a null value.
Either always check for a null value to be returned or return an error if the method above can handle the error in the subroutine in a meaningful way or display an error if continuation without valid data is not possible.

Coding Style

The code is in the most parts easy to read and the naming helps the reader to understand what is going on and which part is responsible for what task. Many methods have quite long names, which could be criticized; on the other hand these long names help to better understand what the purpose of a certain method is.

However, also here a few points of critique:

TODOs

Even if the code is very clean in the most parts, there are some parts that give us the impression that the code is not really ready for deployment. We noticed a lot of '//TODO' comments and many methods have some lines commented out. A drastic example is the #saveEditedEvent method in the Application class. Half of the method is commented out and there is a Boolean 'repeated' in it that is never used afterwards. Such obstacles can be overcome using settings in the IDE. But in the project specific preferences under Java, Compiler, Errors/Warnings nearly everything is set to ignore.

Returning of Lists

An important thing to take into account is the return of lists. They are returned directly without creating a new copy. Thus a reference that points to this list is handed over to the method. As a result it is impossible to deal with it because altering the list in other methods (even if it has been assigned to another local variable) will alter the original list too. Always return a **copy** like `return new LinkedList<Calendar>(this.calendars).`

Method naming

Even though most methods have names that help outsiders to understand the purpose of a method very quickly, there is one exception, namely the #getDatesFor method: The 'For' in the name suggests that the method is related to visibility privileges handling as other methods with 'For' are. However the requesting user is never used here. The only usage beside the test is in #showCalendar.html, that uses it to decide whether to display the date or not. Renaming the misleading method to something like 'getDatesIfNotCurrentDate', 'getDatesOrOnlyTimeIfDateEqualsToday', 'omitDatesForCurrentDay' or simply 'getDates' would be advisable.

Some other small things that could be optimized:

The #remove method in the IntervallEvent class contains more than 150 lines of code and many commented parts. Split it!
Use curly braces '{ }' after 'if' statements. This helps to prevent errors if at a later time more lines are added after the single existing line.
The three methods #removeObserve, #addObserve and #changeObservedCalendar in the User class look all almost the same. You should get rid of these code duplications by re-organizing these three methods.

JavaDoc

The JavaDoc written by Team 1 is very well ‚balanced' in the sense that detailed explanations are given for methods, whose purpose is not clear at first sight while easy understandable methods just carry a short description. The same is true for the description of the contracts between methods: If needed for a better understanding of the functioning of a method, detailed information is provided on what the expected values of the arguments of a certain method are.

The provided JavaDoc is very useful and a good example on how JavaDoc can help other programmers (us) to get a good understanding of foreign code within reasonable time.

Inexplicably and in a sharp contrast to the elsewhere very well documented methods, some methods carry ‘TODOs’ and other comments instead of neat JavaDoc. This strengthens the impression that some parts of this code are not completely ready yet for deployment.

Examples are the #getNameFor method in the Event class, where the comment above the method is ‘//TODO write javadocs’ or the #findEventByIdForUserOnDate method in the same class.

Another point, regarding the documentation is the usage of assertions. They do not work unless explicitly turned on with ’play run calendar -ea’. Mention that in the documentation.

Tests

The team wrote a lot of different tests. Almost every test tests only one single part of the code (or one scenario), as it should be. That means, there are clear and distinct test cases and the test data is well crafted and appropriate for the things to be tested.

Unfortunately, many test methods have names that do not tell what is tested in this method. Method names like #testGetNextRepetitionIntervall7() or #testRemove3() do not tell by themselves, what is tested in these methods and what the expected outcomes of the tests should be.

Another small point of critique: The #remove method in the IntervallEvent class is not tested at all. It seems to be an important method, so test it!