Hotel-project
Week 5-8: Implementation

Assignment

For the final part of the project you shall...

1. ...implement your hotel component(s) in Java
2. ...implement one or more test suites for your system and place these in components

Your system shall at least be able to handle the processes for: booking, checking in, and checking out, in a proper way. By proper is understood that your solution makes use of the designs you have done thus far in the course, and that you apply a good object-oriented approach. Your system is required to follow the models you have created in the last four weeks. As a part of this, you shall generate the structural code for your system component(s) from your class diagram. If you run into problems and have to make changes in your design, update your models as well!

Testing shall be separated from, rather than hard-coded into, the system. Provide one or more components that simulate the users of the system, and contain the tests. Remember to include tests for erroneous or undesired behaviour, in order to show that your system is robust! Additionally, try considering more advanced test cases, such as multiple actors trying to book rooms at the same time.

We advise you to use an iterative process during implementation. Time is short and you want to avoid spending the last week debugging. Our recommendation is to start with a small part of the system. Test and debug it before proceeding with the implementation of new features. You need test cases for the covered use cases when you demonstrate the system; defining them now is time well spent.

Do's and Don'ts

In the following, we have listed some things that you can do and some things to explicitly NOT do!

You may:

- ...ignore the persistence and the presentation layer.
  That means that your system does not need to have any form of graphical user interface, nor does its need persistent data.
- ...place initialisation code in your system component, in case you don't cover use cases for creating rooms and similar things that you need to demonstrate the three required use cases. However, for a higher grade than 3/G, your system shall be able to provide these functions through a component interface.

Do NOT use:

- ...one central object representing the system.
- ...huge interfaces with 10+ methods.
- ...classes, in your system component(s), not generated from your class diagram.
- ...any kind of access to the inside of a component from outside not going through a component interface.
- ...any kind of test cases inside your system component(s). An exception is initialisation code (see the list of allowed things above).

Checklist

- A Java implementation covering at least
  - the booking use case
  - the check in use case
  - the check out use case
- Test cases that show how the use cases are realised. For each use case, there should be at least one test case that violates the intended behavior.

Final Hand-in

The deadline for handing in the source code of the running system is 7th January 2016 09.00. If necessary, a delta of report updates may also be handed in at this time. Please consult the course homepage for further details.

Technical Guidelines

Programming Language

You shall implement your system in Java. Exceptions are only possible if you implement the code generation from your class diagram yourself. If you are really interested in doing this, contact your supervisor (and Grischa for technical information).

IDE

You may use the IDE of your choice for developing your system and test component(s). However, we only provide help for the Eclipse IDE (www.eclipse.org). Additionally, the code generation uses the Eclipse Modeling Framework. Therefore, the use of Eclipse is highly recommended.

Code Generation

As a starting point for your implementation, you are required to generate Java code from your class diagram. This is done using the Eclipse Modeling Framework (EMF). A brief introduction to code generation from Papyrus models using EMF is given at `http://youtu.be/nw182D69k0M`.

Some hints and recommendations:

EMF generates Interfaces for every class in your class diagram. Additionally, it creates classes implementing each of these interfaces (*.impl packages). Only make changes to the classes in the *.impl packages! Additionally, if you make changes, remember to change the '`@generated`' tag to '`@generated NOT`'. Every time you re-generate code from your models, take great care that you do not override any changes that are not backed up, or – preferably – under revision control (see next section). There are also two links on the course homepage (under Project) which explain in detail how the generated code looks like and should be handled.

Revision Control

We highly recommend the use of revision control systems, such as SVN or GIT, for improved collaboration and in order to avoid data loss. We will only provide minimal support for revision control, though. There is plenty of good information and tutorials on the internet for SVN and GIT. Free version control repositories are available, for example through `www.github.com` or `www.bitbucket.org`.

Components in Java

There are many ways to implement components in Java; going into details would probably require a course in itself. A rather rudimentary approach is to place each component in a different package, and in different Java projects in Eclipse.

In each of these components you will have your component interfaces and classes implementing these interfaces (so that they can be instantiated). The classes implementing the component interfaces can, for example, be implemented following the Singleton pattern. On the side of the component that is using (requiring) another component's interface, you can also have a corresponding class that handles calls to the other component interface. In other words, a "requires" interface conforming to the "provides" interface. This ensures that the only communication with another component's interface is going through a single, designated, class.

Within your component, all methods, apart from the ones in your component interfaces, should have package visibility or less. This ensures that no method can be directly accessed from outside the component. Your attributes should of course follow data encapsulation and should not even be directly accessible from other classes within the same package (component).

The aim is for you to be able to treat your system component(s) as a black box! This means that you can later package them as .jar files and simply import them in your testing components.