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 should be able to at least handle the processes for booking and checking in and out in a proper way. By proper it is understood that the solution shall make use of the designs you have done previously in the course and that you shall use 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. Additionally, your system is required to interface with the banking component introduced in the last assignment. If you run into problems and have to make changes in your design, update your models as well!

It is not ok to hard-code the testing into the system. Instead you shall provide one or more components that simulate users of the system and contain the tests. Remember to include tests for erroneous or undesired behaviour, in order to show us that your system is robust! Additionally, try to consider more advanced test cases, such as multiple actors trying to book rooms at the same time. If you want to get a higher grade than a 3/G, your system should be able to handle this as well!

While implementing, it is not advised to use a waterfall process. Time is short and you do not want to spend the last week debugging. Instead, we recommend that you use an iterative process where you start of with a small part of the system. Test and debug before you try to implement new features. You need test cases for the covered use cases when you demonstrate the system, so defining them now is not wasted time.

Do's and Don'ts

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

We do NOT want:

- ...one central object representing the system.
- ...huge interfaces with 10+ methods.
- ...classes in your system component(s) that are not generated from your class diagram.
- ...any kind of access to the inside of a component from outside that does not go 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 below).

On the other hand, you can:

- ...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 it have to persist its 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.

Checklist

- A Java implementation covering at least
  - the booking use case
  - the check-in use case
  - the check-out use case
- At least in one place of these use cases, your system has to use the banking component.
- Test-cases that show how the use cases are realised. There should be at least one test case for each use case that violates the intended behavior.

Final Hand-in

The hand-in date of your final report, the individual report, and the running system is in Week 2, 2015. As with the previous hand-ins, the deadline is 24 hours before your weekly supervision date (even though in Week 2, there is no more supervision).

Technical Guidelines

Programming Language

You shall implement your system in Java. Exceptions are not possible, unless you implement the SOAP/WSDL communication with the banking component and 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). I shortly introduce code generation from Papyrus models using EMF in http://youtu.be/nw182D69k0M.
Some hints or recommendations:
EMF generates Interfaces for every class in your class diagram. Additionally, it creates Classes implementing each of these interfaces (*.impl packages). Do only make changes to the classes in the *.impl packages! Additionally, if you make changes, do not forget 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. We are not going into any details here, as this would probably require a course in itself. A rather rudimentary approach is to place each component in different packages and, preferably, 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 actually 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 and conforms to the same interface. This ensures that the only communication with another component's interface is going through a single, destined class.

Within your component, all methods, apart from the ones in your component interfaces, should have package visibility (protected) 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 that you can treat your system component(s) as a black box! This means that you can later package them as .jar files and simply run import them in your testing components.

Banking Component

In the previous assignment, the banking component's interfaces have been shortly introduced. The administration interface *AdministratorProvides*, which has not been explained in detail, behaves analog to the customer interface: *makeDeposit* adds a non-negative sum to a given credit card account, given that the account exists. It returns the current balance after the deposit or *-1* in case the deposit fails. *getBalance* returns the current balance of an existing credit card account or *-1* in case it doesn't exist. *addCreditCard* adds a credit card account given that an account with that number does not yet exist. *removeCreditCard* removes a credit card account given that it exists.

On the implementation side, the component is a Java EE component

(JAX-WS) running on a GlassFish application server here at Chalmers. We provide a stub for accessing both the administration and the customer interfaces. On the course homepage, under *Project*, you will find a zip file for both interfaces. They contain a .jar file and a demo class. In Eclipse, right-click on your project and choose *Build Path* → *Configure Build Path*. In the following dialog, choose the *Library* tab and click *Add external JARs*. Then simply choose the jar file from your file system and confirm. The demo class shows the usage of the interface. Please note that the application server is only accesible from inside the Chalmers network. Therefore, either use a VPN connection from outside, or use a stub while outside the Chalmers network.

If you are interested in the technical details, you can contact Grischa!

Please observe that the system component is not allowed to access the administration interface. Only your testing components may communicate with it!

In case the bank decides to change their interface, we will notify you of the changes. Additionally, it might be that the banking component is unavailable at some times, due to maintenance or technical issues. In this case, simply replace the interface methods with stub code!