**Database Programming Assignment ("Lab PM")**

*Chalmers/GU TDA357/DIT620*  
LP2, 2018

* [Purpose](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/index.html#toc1)
* [Assignment submission and deadlines](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/index.html#toc2)
* [Introduction](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/index.html#toc3)
* [Domain description](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/index.html#toc4)
* [System Specification](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/index.html#toc5)
* [Task 1: Constructing the database](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/index.html#toc6)
* [Task 2: Database design](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/index.html#toc7)
  + [E-R model](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/index.html#toc8)
  + [Functional dependencies](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/index.html#toc9)
* [Task 3: Triggers](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/index.html#toc10)
* [Task 4: Front end application in JDBC](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/index.html#toc11)

*Modified by Jonas Duregård and Aarne Ranta from earlier years' specifications by many other teachers at Chalmers and GU*

**Purpose**

The purpose of this assignment is to give you hands-on experience with designing, constructing and using a database for a real-world-like domain. You will see all aspects of database creation, from understanding the domain to using the final database from external applications.

**Assignment submission and deadlines**

To pass the programming assignment, you must pass all five tasks described on this page. You will do the assignment in groups of two. You need to form groups, register in Fire and request a PostgreSQL account in the first week of the course.

You must submit your solutions through the Fire reporting system, where you can also see deadlines for each task (link on [main page](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/index.html)).

You must submit your group's solutions to each task by the given deadline. After submission, your assignment will be graded ("pass" or "reject") and you will receive comments on your solution (for tasks 1, 2, and 3). If your submission is rejected, you are allowed to refine your solution and re-submit it.

To pass the final part of the assignment, you must demonstrate your system to one of the teachers. Your files for task 4 must be uploaded to the Fire system **after** you have demonstrated your system, and **before** the task 4 deadline.

**Introduction**

In this assignment you will design and construct a database, together with a front end application, that handles university students and courses. You will do this in four distinct tasks:

1. Construction: create (a prototype of) the database and explore it with queries
2. Design: take a step back and redesign the database using more systematic approaches
3. Usage: define more constraints and triggers to maintain the database
4. Interface: write a wrapper program to permit database access without explicit use of SQL

All of the tasks are related to each other. They deal with the same database domain, and subsequent tasks build on earlier ones to varying degrees.

For each task you will hand in and get feedback on your results. Since errors in one task may propagate to the next one, it is wise to hand in your solutions early to get more chances for feedback. You can also ask assistants on lab sessions to have a quick look at your solution, if they are not to busy (not a guarantee that your submitted solution is accepted!).

Be sure to read through the full description of the assignment before you start since the requirements we place on the system must influence your initial design as well.

**Domain description**

The domain that you will model in this assignment is that of courses and students at a university. So as not to make the task too large and unspecified, you will here get a description of the domain that restricts the problem somewhat. Note that the described domain is not identical to Chalmers or GU.

The university for which you are building this system is organized into departments for employees, such as the Dept. of Computing Science (CS), and study programmes for students, such as the Computer Science and Engineering programme (CSEP). Programmes are hosted by departments, but several departments may collaborate on a programme, which is the case with CSEP that is co-hosted by the CS department and the Department of Computer Engineering (CE). Department names and abbreviations are unique, as are programme names but not necessarily abbreviations.

Each study programme is further divided into branches, for example CSEP has branches Computer Languages, Algorithms, Software Engineering etc. Note that branch names are unique within a given programme, but not necessarily across several programmes. For instance, both CSEP and a programme in Automation Technology could have a branch called Interaction Design. For each study programme, there are mandatory courses. For each branch, there are additional mandatory courses that the students taking that branch must read. Branches also name a set of recommended courses from which all students taking that branch must read a certain amount to fulfill the requirements of graduation, see below.

A student always belongs to a programme. Students must choose a single branch within that programme, and fulfill its graduation requirements, in order to graduate. Typically students choose which branch to take in their fourth year, which means that students who are in the early parts of their studies may not yet belong to any branch.

Courses are given by a department (e.g. CS gives the Databases course). Each course has a unique six character course code. All courses may be read by students from any study programme. Some courses may be mandatory for certain programmes, but not so for others. Students get credits for passing courses, the exact number may vary between courses (but all students get the same number of credits for the same course). Some, but not all, courses have a restriction on the number of students that may take the course at the same time. Courses can be classified as being mathematical courses, research courses or seminar courses. Not all courses need to be classified, and some courses may have more than one classification. The university will occasionally introduce new classifications. Some courses have prerequisites, i.e. other courses that must be read before a student is allowed to register to it.

Students need to register for courses in order to read them. To be allowed to register, the student must first fulfill all prerequisites for the course. It should not be possible for a student to register to a course unless the prerequisite courses are already passed. It should not be possible for a student to register for a course which they have already passed.

If a course becomes full, subsequent registering students are put on a waiting list. If one of the previously registered students decides to drop out, such that there is an open slot on the course, that slot is given to the student who has waited the longest. When the course is finished, all students are graded on a scale of 'U', '3', '4', '5'. Getting a 'U' means the student has not passed the course, while the other grades denote various degrees of success.

A study administrator can override both course prerequisite requirements and size restrictions and add a student directly as registered to a course. (Note: you will not implement any front end application for study administrators, only for students. The database must still be able to handle this situation.)

For a student to graduate there are a number of requirements they must first fulfill. They must have passed (have at least grade 3) in all mandatory courses of the study programme they belongs to, as well as the mandatory courses of the particular branch that they must have chosen. Also they must have passed at least 10 credits worth of courses among the recommended courses for the branch. Furthermore they need to have read and passed (at least) 20 credits worth of courses classified as mathematical courses, 10 credits worth of courses classified as research courses, and one seminar course. Mandatory and recommended courses that are also classified in some way are counted just like any other course, so if one of the mandatory courses of a programme is also a seminar course, students of that programme will not be required to read any more seminar courses.

**System Specification**

You will design and implement a database for the above domain, and a front end application intended for students of the university. Through the application they should be able to see their own information, register to, and unregister from courses.

Formally, your application should have the following modes:

- Info: Given a students national identification number, the system should provide

- the name of the student, the students national identification number (10 digits) and their university issued login name/ID (something similar to how the CID works for chalmers students) - the programme and branch (if any) that the student is following. - the courses that the student has read, along with the grade. - the courses that the student is registered to and waiting for (and their queue position if waiting). - whether or not the student fulfills the requirements for graduation

- Register: Given a student id number and a course code, the system should try to register the student for that course. If the course is full, the student should be placed in the waiting list for that course. If the student has already passed the course, or is already registered, or does not meet the prerequisites for the course, the registration should fail. The system should notify the student of the outcome of the attempted registration, and the reason for failure (if any). - Unregister: Given a student id number and a course code, the system should unregister the student from that course. If there are students waiting to be registered, and there is now room on the course, the one first in line should be registered for the course. The system should acknowledge the removed registration for the student. If the student is not registered to the course when trying to unregister, the system need not notify this, but no student from the waiting list (if applicable) should be promoted in that case.

**Task 1: Constructing the database**

Your task is to construct a first version the database by implementing the database schema in a database engine (PostgreSQL). The schema for this part is given in this file: [Abstract schema for database task 1](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/task1.txt)

Note that the schema is somewhat incomplete, but that will be fixed in part two of the assignmnent. You should implement the schema with CREATE TABLE statements that

* uses exactly the given table and attribute names
* express the same primary key and foreign key constraints
* use sensible types (types are not expressed in the abstract schema).   
  **Hint1:** The NUMERIC(X,Y) type can be used to specify types like "integers with X digits" see [this documentation](https://www.postgresql.org/docs/10/static/datatype-numeric.html" \l "DATATYPE-NUMERIC-DECIMAL). The CHAR(X) type is used for texts with X characters.  
  **Hint2:**For the position attribute, you may either use a timestamp, an auto-increasing (unique) number or the absolute position (there are advantages and disadvantages to each).
* no column may accept NULL values

Thus you should create all tables, marking key and foreign key constraints in the process, and you should also insert basic checks that ensure that only valid data can be inserted in the database. Examples of invalid data would be the grade '6', or a course that takes a negative number of students.

When you have created the tables, you should fill the tables with example data. Ordinarily, this is a time-consuming but important part of the development of a database. Having data in the database is crucial in order to properly verify that it behaves the way that you expect it to. The tables should be filled with enough data so that it is possible to test that your application can handle the various operations specified above. Just inserting tons of data is of no use if the data still doesn't test all parts of the database.

In this course however, we have decided to just give you a set of insert that should work (possibly with slight adjustments): [inserts.sql](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/inserts.sql). Note the various corner cases covered by the tests:

* A handful of students, at least one of which fulfils the requirements for graduation and a couple that do not for different reasons.
* A number of courses that test all of the various aspects a course. This includes classifications, mandatory, recommended etc.
* Two branches with the same name on different programs, each with at least one registered student and at least one mandatory course.
* At least three waiting students for two different (full) courses.
* At least one student that has not chosen a branch.
* A student that has not taken any courses.
* A student that has only failing grades.
* ...

Following the system specification, create these views:

* View: BasicInformation(idnr, name, login, program, branch) For all students, their national identification number, name, login, their program and the branch (if any). The branch column is the only column in any of the views that is allowed to contain NULL.
* View: FinishedCourses(student, course, grade, credits) For all students, all finished courses, along with their codes, grades (grade 'U', '3', '4' or '5') and number of credits. The type of the grade should be a character type, e.g. CHAR(1).
* View: PassedCourses(student, course, credits) For all students, all passed courses, i.e. courses finished with a grade other than 'U', and the number of credits for those courses. This view is intended as a helper view towards later views (and for task 4), and will not be directly used by your application.
* View: Registrations(student, course, status) All registered and waiting students for all courses, along with their waiting status ('registered' or 'waiting').
* View: UnreadMandatory(student, course) For all students, the mandatory courses (branch and programme) they have not yet passed. This view is intended as a helper view towards the PathToGraduation view, and will not be directly used by your application.
* View: PathToGraduation(student, totalCredits, mandatoryLeft, mathCredits, researchCredits, seminarCourses, qualified) For all students, their path to graduation, i.e. a view with columns for
  + student: the student's national identification number.
  + totalCredits: the number of credits they have taken.
  + mandatoryLeft: the number of courses that are mandatory for a branch or a program they have yet to read.
  + mathCredits: the number of credits they have taken in courses that are classified as math courses.
  + researchCredits: the number of credits they have taken in courses that are classified as research courses.
  + seminarsCourses: the number of seminar courses they have read.
  + qualified: whether or not they qualify for graduation. The SQL type of this field should be BOOLEAN (i.e. TRUE or FALSE).

**Hint1:** Make a query for the data of each column and when they all work, but them in a WITH clause and use a chain of (left) outer joins to combine them.  
**Hint2:** Use COALESCE to replace null values with 0 (e.g. COALESCE(totalCredits,0) AS totalCredits. Also, keep in mind that comparing null values with anything gives UNKNOWN!  
**Hint3:** A query containing student/classification/credit with a row for each classification of each course every student has passed may be useful.

Make sure that your views use the right names of columns! Use AS to name a column.

**Testing** The file [runtests.sql](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/runtests.sql) automatically runs the files for this assignment. The file [output.txt](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/output.txt) contains the expected output of doing this. Run the file (with \i runtests.sql in psql) and make sure your output matches the example (assuming you use the provided insert.sql file) and that you get no errors (order of rows may possibly vary). This is not a perfect guarantee that your code works as it should, but it does find several common errors.

**Deliverables**: For task 1, you should submit the following files through Fire:

* tables.sql: your SQL code for creating the tables.
* insert.sql: OPTIONAL, your SQL code containing the insert statements for the data IF IT DIFFERS FROM THE FILE YOU WHERE GIVEN. Also include a short comment next to every change you made to the file.
* views.sql: your SQL code for creating the listed views.
* explanation.txt OPTIONAL, this file is only required if you did not get all the tests provided to work exactly as intended, and should contain a short description of the problem. Submissions that do not contain this file and fail to pass the tests will be summarily rejected without comments.

Note that SQL code should be in plain text format. Make sure that PostgreSQL can execute your files on an empty database before you hand them in. Do not have the "delete everything"-statement in any of the files you submit.

**Task 2: Database design**

The task in this lab is to take a look back at the database implemented in Lab 1 and check how we should have done it. More precisely, we will

* apply our design skills to the domain description:
  + build an E-R model and derive a schema from it by the E-R to schema translation
  + analyse functional dependencies (FD) and derive a schema by normalization
  + compare the previously given schema ([task1.txt](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/task1.txt)) with the ones obtained by these methods and construct a final schema for the whole database (and translate it into SQL).

There are many interesting outcomes that may result from this study:

* + we may notice that E-R and FD give the same schema as task1.txt and conclude that we have done excellent work
  + find differences between E-R, FD and task1.txt that help complete each of them
  + find a very different design that is still justifiable

**E-R model**

First create an E-R diagram that correctly models the domain described in the [Domain description](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/index.html#toc4) above.

You can use any tool you like for this task, as long as you hand in your solution as an image in one of the formats .png, .jpg, .gif or .pdf and as a schema in a pure text file (in a format similar to [task1.txt](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/task1.txt)). We recommend you use [Dia](http://www.cse.chalmers.se/edu/year/2017/course/TDA357/HT2017/lab/dia.html).

You must in both cases use the translations specified in the lectures, and mark the keys and references properly.

**Hint**: if your diagram does not contain at least one weak entity, at least one ISA relationship, and at least one many-to-at-most-one relationship, you have probably done something wrong.

**Functional dependencies**

Your second task is to formulate all functional dependencies in the domain and derive a valid schema from them. This should be done mostly independently of the E-R design and of the schema in [task1.txt](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/task1.txt).

* + Identify the functional dependencies that you expect should hold for the domain:
    1. Collect all attributes (with unique names, e.g. sname for student names and cname for course name etc.) for the domain and put them into one table, Domain(....). You may look in the schema from your ER model for this.
    2. By reading the Domain Description, identify all functional dependencies in Domain(....). Do not just extract them from your ER-model!  
       Here is a non-exhaustive list of things you may want to consider when searching for functional dependencies:
       - department names and abbreviations
       - study programme names and abbreviations
       - branch names
       - student names and identification numbers
       - student programmes and branches
       - course names and codes
       - course classifications and prerequisites
       - course waiting lists (more than one FD involving position)
  + Use the dependencies to find keys and normal forms:
    1. Find all possible keys of Domain(....), as well as all BCNF violations.
    2. Decompose Domain(....) to distinct tables by using the BCNF normalization algorithm.
    3. Carefully mark keys and references in the resulting schemas.
  + Compare and discuss:
    1. Compare the BCNF schema with your E-R schema, asking in particular: are their constraints that only E-R or only FD can reveal?
    2. Are there any constraints that neither approach guarantee? (**Hint:** Are you sure a student can only attend one programme?).
    3. As a result of this, build a "perfect schema", based on the ER-schema but including unique constraints and at least one additional/modified reference constraint based on the functional dependencies.
    4. Compare this with the [schema](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/task1.txt) in Lab 1. If you made clever design choices, most of it should be similar but with additional constraints and relations.
    5. Translate the updated schema into SQL Tables and add required inserts etc. If you find that you will have to make changes to existing inserts/views, perhaps your design is not yet perfect.

**Hint**: the Query Converter (link on main page) can automate much of the analysis, as soon as you have identified the functional dependencies yourself. It can get slow when the schema is very large, but you can then just reload the page to kill the process.

**Hint**: The final schema should contain at least four additional relations compared to the schema in Task 1, and at least 3 UNIQUE constraints (one is only required for technical reasons to enable a constraint that prevents students from chosing a branch in the wrong program).

**Deliverables**: For Task 2, you should submit the following files through Fire

* + ER.png: your E-R diagram as a .png, .jpg, .gif or .pdf file
  + ER-schema.txt: the database schema derived from your E-R diagram, as a text file (no PDF, no Microsoft Word, no HTML!)
  + FD.txt: your Domain(...) schema together with the functional dependencies you have found (you can omit dependencies that can be derived from the ones you mention)
  + FD-schema.txt: the decomposed schemas using the BCNF algorithm
  + final-schema.txt: your final database schema
  + tables.sql, views.sql, inserts.sql: Your modified database code.

Criteria of acceptance:

* + ER.png: the E-R diagram must be syntactically correct and a reasonable model for the domain (and you cannot do this without weak entities and ISA entities!)
  + ER-schema.txt: the ER-schema must be correctly derived from the diagram
  + FD.txt: you should have found all and only the reasonable functional dependencies in the domain (derived ones can be omitted)
  + FD-schema.txt: the decompositions should be correct (this is partly automatic if you use Query Converter, but you should make sure to mark the keys and references)
  + design-report.pdf: the report should identify all cases where the ER and FD analyses lead to different schemas and, at least in some cases, argue why any of them is preferable. A similar comparison should be done with the [schema in Lab 1](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/task1.txt). In the simplest case, you can just say that the schema is exactly the same. You can say this if some names of attributes and tables are different, as long as the structure is the same.

**Task 3: Triggers**

When your tables and views are implemented in Task 2, the next task is to create two triggers to handle some key issues in registration and unregistration. Here is a piece of code to get you started on the first trigger:

[Postgresql trigger example](http://www.postgresql.org/docs/current/static/plpgsql-trigger.html#PLPGSQL-TRIGGER-EXAMPLE)

But first, you should define one more view that can be used by your triggers, and your application in Task 4:

* + View CourseQueuePositions(course,student,place): For all students who are in the queue for a course, the course code, the student's identification number, and the student's current place in the queue (the student who is first in a queue will have place "1" in that queue, etc.). This view is trivial if you store the positions directly in the database, but not if you store e.g. registration timestamps.

When a student tries to register for a course, it is possible that the course is already full, in which case the student should be put in the waiting list for that course. When a student unregisters, it might be that there is now room for some student who is in the waiting list, and who should then be registered for the course instead. Such things are typically handled via triggers. You should write two triggers:

* + When a student tries to register for a course that is full, that student is added to the waiting list for the course. Be sure to check that the student may actually register for the course before adding to either list, if it may not you should raise an error (use [RAISE EXCEPTION](http://www.postgresql.org/docs/current/static/plpgsql-errors-and-messages.html)). **Hint**: There are several requirements for registration stated in the domain description, and some implicit ones like that a student can not be both waiting and registered for the same course at the same time.
  + When a student unregisters from a course if the student was properly registered and not only on the waiting list, the first student (if any) in the waiting list should be registered for the course instead. **Note**: this should only be done if there is actually room on the course (the course might have been over-full due to an administrator overriding the restriction and adding students directly).

You need to write the triggers on the view Registrations instead of on the tables themselves (the view was built in task 1 above). (One reason for this is that we "pretend" that you only have the privileges listed under Task 4, which means you cannot insert data into, or delete data from, the underlying tables directly. But even if we lift this restriction, there is another reason for not defining these triggers on the underlying tables - can you figure out why?)   
**Hint1**: Write your triggers incrementally, first make a registration trigger that just always give an error and test that it works, then make it give an error only if the student is already registered (or waiting) and otherwise register it, then make it give a different error if the student is missing prerequisites, then make it put the student in a waiting list if the course is full, and so on.  
**Hint2**: One way to check if a course is full: Count the number of courses with that code and more students than its seat capacity. The count will be 1 or 0 (0 meaning either it has no limit or the limit is not exceeded).

**Testing** Make a file tests.sql with inserts and deletes that test all cases of your triggers. Write a short comment (line starting with --) over each insert/delete that states what is tested the expected outcome (change/error).   
**Hint1**: Your tests need to test every way a registration could fail, and the outcomes of successfull registrations/unregistrations. The latter includes:

* + registered to unlimited course
  + registered to limited course
  + waiting for limited course
  + unregistered from unlimited course
  + unregistered from limited course without waiting list
  + unregistered from limited course with waiting list
  + unregiestered from overfull course with waiting list

**Hint2**: Look at the list at the end of Task 4 for some ideas on what to test.

**Deliverables**. For task 3, you should hand in the following files:

* + setup.sql: SQL code that sets up your database for testing the triggers. This will normally be the concatenation of files tables.sql, insert.sql and views.sql from Task 2. We must be able to test your triggers by executing the files setup.sql and triggers.sql, in that order.
  + triggers.sql: SQL code for creating the two triggers.
  + test.sql The testing code for your triggers.

**Make sure that PostgreSQL can execute your files before you hand them in.**

**Task 4: Front end application in JDBC**

The last part of this assignment is to write an application that students can use to communicate with your database. This application should be a Java program that uses JDBC to connect to the PostgreSQL database to request and insert the proper data.

To your help when writing your application we provide you with a stub file that contains the code for connecting to PostgreSQL on the local system. It also contains hooks for the three operating modes of the application, and this is where you should insert your code. The idea is that you should not need to focus so much on the pure Java parts of the application, but rather get straight down to business with the database-interfacing code.

The stub file is here: [StudentPortal.java](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/StudentPortal.java).

The intended behavior of the program is that you use it from the command line, giving some student identification number as an argument (what exactly that is depends on your design). This corresponds to the student "logging on" to the portal. Once logged on, the student can choose one of the three modes "Information", "Register" or "Unregister". If the first is chosen, all information for that student should be printed. Exactly what information must be printed is given by the system requirements specified above. If one of the latter modes are chosen, the student will be prompted for a course to register to or unregister from, and the application should perform the requested operation and print the result (success, failure).

The stub file can be compiled and run as it is, only nothing will happen in any of the modes. Your task is thus to fill in the actual logic of these three tasks.

Running your application could look like this:

$> java StudentPortal 1234567890

Welcome!

Please choose a mode of operation:

? > i

Information for student 1234567890

-------------------------------------

Name: Emilia Emilsson

Student ID: emem

Line: Information Technology (IT)

Branch: Systems Development

Read courses (name (code), credits: grade):

Set Theory (MAT050), 5p: 5

Functional Programming (TDA450), 10p: 5

Object-Oriented Systems Development (TDA590), 10p: 4

Registered courses (name (code): status):

Databases (TDA356): registered

Algorithms (TIN090): waiting as nr 3

Seminar courses taken: 0

Math credits taken: 5

Research credits taken: 0

Total credits taken: 25

Fulfills the requirements for graduation: no

-------------------------------------

Please choose a mode of operation:

? > r TDA350

You are now successfully registered to course TDA350 Cryptograhy!

Please choose a mode of operation:

? > r TDA381

Course TDA381 Concurrent Programming is full, you are put in the

waiting list.

Please choose a mode of operaion:

? > quit

Goodbye!

$>

Note that the exact formatting is only a suggestion: you may choose to format your output differently as long as you give the proper information back to the user.

To get access to the PostgreSQL jdbc drivers from your application, you should download it from [https://jdbc.postgresql.org](https://jdbc.postgresql.org/) and import it into your java CLASSPATH.

Alternatively you can import the file into an Eclipse project if you are using Eclipse.

Your student application should behave as if it has **only** the following privileges:

* + SELECT ON Course
  + SELECT ON Student
  + SELECT ON BasicInformation
  + SELECT ON FinishedCourses
  + SELECT ON Registrations
  + SELECT ON CourseQueuePositions
  + SELECT ON PathToGraduation
  + INSERT ON Registrations
  + DELETE ON Registrations

We will check your submitted code to ensure that you adhere to these privileges, even though we cannot get the system to enforce them automatically.

Deliverables:

* + file StudentPortal.java with your additions, submitted via Fire
  + oral demonstration to a teacher in a lab session
  + filled [demo form](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/demo-form.pdf) shown at the oral demonstration

You must come to one of the supervision sessions and demonstrate your running application, and we will accept or reject it on the spot (pending the check of the submitted code for authority violations).

All lab sessions starting from Monday 5 March are available for grading, which should take around 10 minutes per group. In addition, we will arrange two extra lab sessions in the exam week (Monday and Thursday).

As you probably realize, if all of you wait until the last session, we will quite simply not have time for everyone, so come as early as possible!

Here is a list of what we will test your application for:

* + List info for a student.
  + Register the student for an unrestricted course, and show that they end up registered (show info again).
  + Register the same student for the same course again, and show that the program doesn't crash, and that the student gets an error message.
  + Unregister the student from the course, and then unregister again from the same course. Show that the student is unregistered.
  + Register the student for a course that they don't have the prerequisites for, and show that the registration doesn't go through.
  + Unregister the student from a restricted course that they are registered to, and which has at least two students in the queue. Register again to the same course and show that the student gets the correct (last) position in the waiting list.
  + Register, unregister and re-register the same student for the same restricted course, and show that the student is first removed and then ends up in the same position as before (last).
  + Finally, unregister a student from an **overfull** course, i.e. one with more students registered than there are places on the course (you need to set this situation up in the database directly). Show that no student was moved from the queue to being registered as a result.

Ensure that the data you have put into your system can handle all these cases. Please prepare before you ask us to check your application, so that running through these cases will be smooth. Please fill the following form before demonstration: [demo form](http://www.cse.chalmers.se/edu/course/TDA357/HT2018/lab/demo-form.pdf).

**You must demo your working project before or on the last lab session.**