# CS 583, Spring 2011 — Class Project

For the CS 583 class project you have one of two options:

1. Implement the default project described in more detail below, or
2. Find and implement an alternative project

To be able to exercise the second option, you must find a suitable project on your own and *convince me in advance* that it is a viable alternative, has the appropriate scope, and is overall well suited to demonstrate your functional programming proficiency. Please note that I consider option (2) to be the exception. Only if you can make a convincing case why you should *not* choose option (1) and why an alternative project would be so much better, will I approve it. I will set the bar deliberately very high. For example, if your project promises to yield a publishable paper, that would certainly be a good reason.

**IMPORTANT!** *You can choose option (2) only if your project has been explicitly approved by me.*

You can implement the class project alone or in team of two. *Note:* If you decide to go for a team solution, you must make clear the individual contribution of each team member.

## 1 Class Project: *Twizzer*

Your task is to write a Haskell program that implements a Twizzer system that allows creation of twizzes and the submission of solutions and reviews, much like what we have been simulating with the CoE TEACH system in our class so far.

The Twizzer system contains two different levels: (1) An *administrative level* for the creation of twizzes, the definition of deadlines for twizzes and reviews, transitioning between twizzer phases (tasks vs. reviews), managing users and assigning buddies for reviews, tabulating results, etc. and (2) a *user level* that allows users to submit solutions, see solutions of other users for which they have to prepare a review, submit reviews, view reviews of their own submissions, etc.

The twizzer system can be understood as a state transition system. Each twiz goes through different phases, and each phase gives users specific access to the system. The transition between the phases should be implemented through the invocation of a function from the administrator. Here is a summary of the different phases.

(1) *Setup.* In this initial phase the twiz is defined by the administrator, together with deadlines for twiz and review submissions and assignment of buddies.
(2) *Task.* During this phase, users can submit solutions. They cannot see other users' solutions.
(3) *Review*. In the review phase users can see the solutions of those other users whom they have been assigned to as review buddies, and they can submit reviews for those solutions. Users cannot see the solution or reviews of other users, and they cannot see the review for their own solution during this phase.
(4) *Closing.* In this phase, all users can see the reviews created by their buddies. As an optional feature, the administrator may be able to make public some solutions and reviews (to provide good and not so good examples).

This specification still leaves open many details. In particular, the review assignments (buddies), the reviews, and the published examples (of solutions and reviews) could all be kept anonymous, or the authors of reviews could be revealed to support accountability.

## 2 Implementation Details

It might be a good idea to implement your program in different stages to gradually evolve it. Here is a suggestion of how to proceed.

(1) Implement a program that uses only in-memory data structures. The user interface for this program is GHCi's command-line interface, that is, a user of the program enters functions you have defined in you program. The twiz solutions and reviews can be represented as strings, that is, the "uploading" happens by giving string arguments to functions. No data is saved when the program is terminated.
(2) Extend your program to make the Twizzer state persistent, that is, when the program starts the state is read in from a file, and before the program terminates, the state is saved in a file.
(3) Extend your programs to allow the uploading of solutions and reviews from files.
(4) The previous versions of the program were all non-distributive, that is, all events in the system had to be initiated within one and the same program run. For a realistic implementation this should be generalized so that different instances of the program can run and still have access to the same Twizzer state. This could be implemented using rudimentary file system commands or by adding a databases backend. Alternatively, one could also implement a web-based interface. A realistic distributed implementation would probably also need some form of user access control. (For either the database or web-based extension, see `haskell.org` for packages that support these kinds of programs.)

## 3 Deliverables

You must submit a working Haskell program together with some form of documentation (either as an ASCII or a pdf file) that contains the following.

- Instructions how to install and run the program
- Some example usage scenarios
- The contribution of each team member (in case of a team solution)
- An explanation of how and where you used functional programming concepts in the implementation (lazy evaluation, higher-order functions, type classes, etc.).
- Anything you think is cool about your implementation and that you want to highlight

## 4 Grading Criteria

Your solution will be graded for

- Correctness (Does the program behave as intended?)
- Functionality (Does the program provide all or most of the functions it is supposed to support?)
- Style (Does the program look like a functional program? Is it written in a functional style? Does it make use of functional programming idioms?)

Good style means, for example, to not re-implement existing higher-order functions, but to recognize their applicability and use them. Moreover, function definitions should be as general as possible (as is reasonable to expect within a given context), and repeated code should be avoided and instead be refactored into function definitions and their use. Where appropriate, type classes should be used to structure the implementation.

You should explain in the documentation of your submission what functional programming concepts (such as, higher-order functions or monads) you have have employed and how.