1. Design

1.1. Violation of MVC pattern

Pattern is used correctly in the sense of the mvc-Spring Framework. Data, functionality and representation are clearly separated.

1.2. Usage of helper objects between view and model

Whenever model data is added or edited, helper objects (pojos) are used to transfer the data from view to model.

1.3. Rich OO domain model

The real-world concepts of universities, courses and users (members) are translated very well in an OO environment in the code.

1.4. Clear responsibilities

There are many different ways to approach the separation of responsibilities. In this project the responsibilities are assigned very clearly.

1.5. Sound invariants

There are no invariants.
Example Proposition: Whenever the EditController is addressed there must be exactly one (not none, not several) registered user logged in.

1.6. Overall code organization & reuse, e.g. views

We like your url-mappings.
We can't find any redundancies.

2. Coding style

2.1. Consistency

The style is consistent throughout the whole project.

2.2. Intention-revealing names

Problem: *deleteProvidedCourse()* (CourseService, line 34)
Proposition: *deleteCourseFromMember()*

Problem: *save()* (CourseController, line 93)
Proposition: *addCourseToMember()*

Problem: *saveEditChange()* (MemberService, line 33)
Proposition: *saveChange()*
Problem: *deleteCourse()* (CourseController, line 135)
Proposition: *deleteCourseFromMember()*

Problem: *show()* (ProfileController, line 42)
Proposition: *showProfile()*

Problem: *edit()* (EditController, line 50)
Proposition: *getEditPage()*
Comment: Applies for every *RequestMethod.Get*. Easy to read if 'get' is somehow mentioned in the method names.

Problem: *saveChange()* (EditController, line 80)
Proposition: *editProfile()*

Problem: *isIsActivated()*
Proposition: pretty ugly indeed ;)

Since the *EditController* is responsible for editing explicitly a specific profile we would recommend naming it *EditProfileController*.

2.3. Do not repeat yourself

Problem: *index1()*
Where: *IndexController* line 22
Proposition: Delete, Same as *index()*. Value can be a list: *value = {"/", "/index"}*

Problem: When the *EditForm* is not valid a new view of edit is returned.
Where: *EditController,* line 79.
Proposition: You could simply redirect to *"/edit"*. This would require the method to return a string which is possible and easier.

Problem: *Securitycontextholder.getContext().getAuthentication()* is cast to member multiple times.
Where: *CourseController* line 36, 95, 113 for example.
Proposition: This could easily be refactored into a method called *getCurrentUser()*.

2.4. Exception, testing null values

Whenever something is *NULL* when entering a method exceptions will be thrown but not handled.
If you test your methods with *NULL*, most of them will crash. So we have to make sure your program will never call these methods with *NULL*. Why and how is this guaranteed? What happens if your DB is empty at the beginning? (We couldn't use some functionality until we manually added some entries in the DB)

2.5. Encapsulation

Problem: Method extractNames() is public and not used anywhere except for testing.
Where: *EditController*, line 107.
Proposition: Make it private.

2.6. Assertion, contracts, invariant checks

You assume that some Data is already provided (like universities or courses) but there are no contracts that ensure it.

2.7. Utility methods

There are very few utility methods. There is room for more.

3. Documentation

3.1. Understandable

Yes

3.2. Intention-revealing

Some docs (like in tests) don't actually explain anything and just make a sentence out of the method name. They are not necessary, the method name itself is suffiecient.

3.3. Describe responsibilities

Although the intentions are pretty clear, there are no contracts. And since responsibilities are part of the responsibility-driven-design-contracts we see room for improvement.

3.4. Match a consistent domain vocabulary

There is no need to say "This method …". Just start with a verb. Everyone knows what method is being described.
Otherwise it's well documented following the java standard documentation style.

4. Test

4.1. Clear and distinct test cases

Every major functionality-class its own test-class.

4.2. Number/coverage of test cases

Coverage is high but not perfect yet.
Especially EditController is lacking coverage.
Also there are no test for exceptions.

4.3. Easy to understand the case that is tested

Simple and well segregated unit tests provide an easy understanding of what is being tested.

4.4. Well-crafted set of test data

No repeating names, no redundancies.

4.5. Readability

As good as it gets with this kind of project.

5. ProfileController

Since all profile editing is handled in the EditController the becomeTutor() method would be better placed in the EditController.
Else the controllers are very well designed in our opinion.