Designing With GRASP

We will see some design patterns in our presentations.
But there is next to these design patterns also GRASP.

GRASP is an acronym for General Responsibility Assignment Software Patterns

And in English “to grasp” also means to “understand something” or “to see how something works”.

➔ GRASP allows you to work with patterns in order to design object-oriented software

You can see GRASP as a guideline: it helps you keep the main features of a good design in check. Focus is always on responsibility: which objects get what responsibilities?

GRASP has nine patterns:

Creator	*Controller*	*Pure Fabrication*
Information Expert	*High Cohesion*	*Indirection*
Low Coupling	*Polymorphism*	*Protected Variations*

Let’s have a closer look at a few:

Pattern: Creator

In a design, make sure to identify who “creates” other objects. This object has then the responsibility to create.

Example: When a student registers for a course. Which class will create the student objects? Is it some “Admin” class that knows the “Student” class and creates the objects?

Pattern: Information Expert

Make sure to identify who “knows” about other objects. Who has the information?

Example: In a student registry for courses, it would make sense to have a class that has all the information about the available courses in a “CoursesCatalogue” class. You would then have a dependency between “Catalogue” and singular “Course”, but the “Catalogue” would know more.

Pattern: Low Coupling

Make sure that you reduce the impact of change. If you have a class that is the “information expert”, you make sure that the information is concentrated in one class and not spread in too many classes.

Example: using interfaces which offer services / methods to another class guarantees a lower coupling.

Pattern: Controller

You know this from the MVC-principle: It makes sense to separate a user-interface from the logic of a program. Who should receive the message from the UI layer? This class is your controller. Make sure that your “menu” class does not deal with business logic.

Pattern: High Cohesion

The idea is to keep objects focused and understandable.
If you have a class that is doing many unrelated things, then they become hard to maintain and have too many responsibilities.

Example: We have a class called “Company”. This is responsible for a) knowing its employees and b) knowing its financial information. These two areas are not strongly related to each other. So the class “Company” should make sure that those tasks are delegated.
It’s like in the real world: if a person takes on too many responsibilities, the quality of work will get worse and you could say this person suffers from low cohesion.