# Cohesion and Coupling

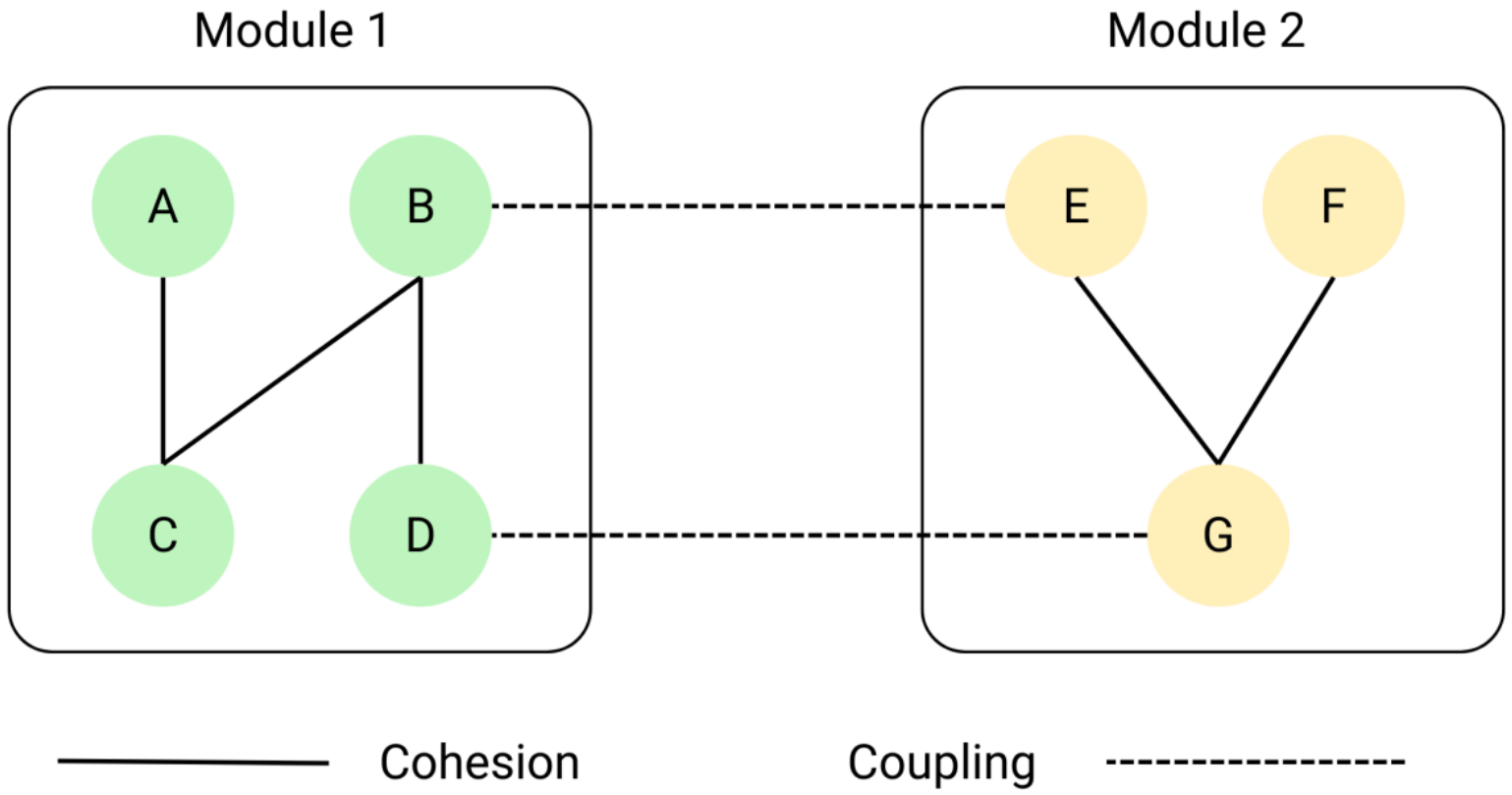
Kaustubh Kulkarni

# Cohesion

- Cohesion defines how effectively elements within a module or object work together to fulfil a single well-defined purpose.
- A high level of cohesion means that elements within a module are tightly integrated and working together to achieve the desired functionality.

# Coupling

- Coupling defines the degree of interdependence between software modules.
- Tight coupling means that modules are closely connected and changes in one module can affect others.
- On the other hand, loose coupling means that modules are independent and changes in one module have minimal impact on others.


Module 1
Module 2
A
B
C
D
E
F
G
Cohesion
Coupling

# What is the meaning of Low Cohesion?

- In low cohesion, a module or object within a system has multiple responsibilities that are not closely related i.e. a single module or object performs a diverse range of independent tasks.
- As a result:
  - This makes our code difficult to understand.
  - This can lead to confusion and make it challenging to modify or update the code.

# Example

- For example, let’s consider a class “StudentRecord” that has following responsibilities
- 1) Maintaining student information
- 2) Calculating student grades
- 3) Printing student transcripts
- 4) Sending email notifications to students.

# Initial Design

(Low Cohesion, High Coupling)

# Low Cohesion

- Maintaining student information is one responsibility, which might involve storing and updating student details.
- Calculating student grades is another, which involves processing grades data.
- Printing student transcripts is another, focusing on formatting and outputting information.
- Sending email notifications is yet another, involving communication tasks.
- If all these responsibilities are in one class, then the class has low cohesion because it handles many unrelated tasks.

# High Coupling

- If the StudentRecord class is tightly coupled with all the classes or methods that manage information, calculate grades, print transcripts, and send emails, changes in any of these methods or dependencies can cause issues across the entire system.
- For example, if the StudentRecord class directly interacts with a Printer class to print transcripts, then a change in the Printer class would require changes in StudentRecord as well.
- This is an example of high coupling.

- Refer `StudentRecord.java`

# Refactored Design

(High Cohesion, Low Coupling)

# High Cohesion

- Ideally, each of these responsibilities should be in its own class to increase cohesion.
- A StudentInfo class to handle student information.
- A GradeCalculator class to calculate grades.
- A TranscriptPrinter class to print transcripts.
- An EmailNotification class to handle email communication.

# Low Coupling

- To achieve low coupling, you would use interfaces or abstract classes, where StudentRecord would depend on abstractions rather than concrete implementations.
- This way, changes to the underlying implementations of email sending, transcript printing, etc., won't necessarily impact the StudentRecord class.

- Refer following files for refactored design:
  - `StudentInfo.java`
  - `GradeCalculator.java`
  - `TranscriptPrinter.java`
  - `SimpleEmailNotification.java`
  - `StudentRecordRefactored.java`

# Questions?