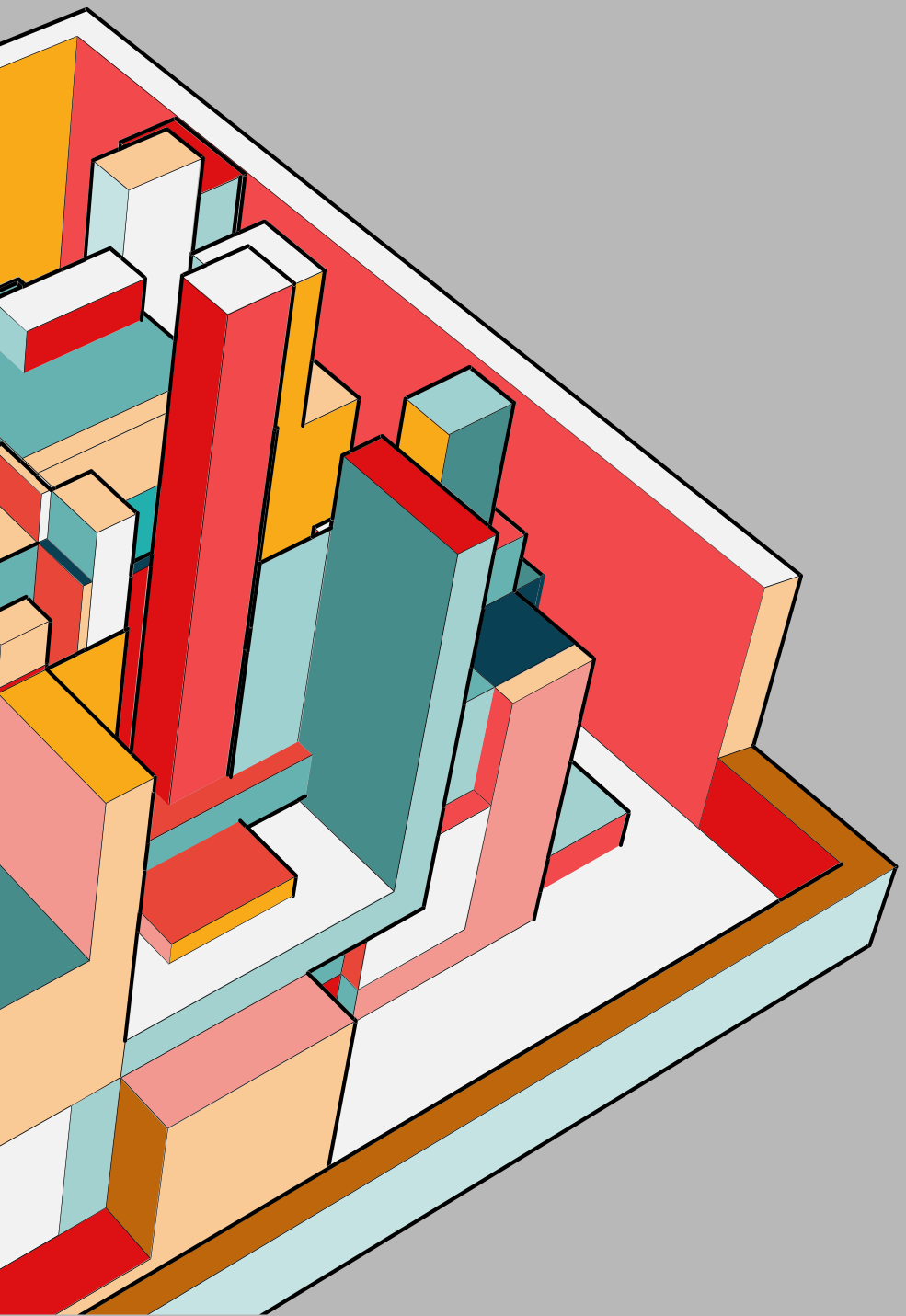


تصميم وهيكلة البرمجيات
ITSE411

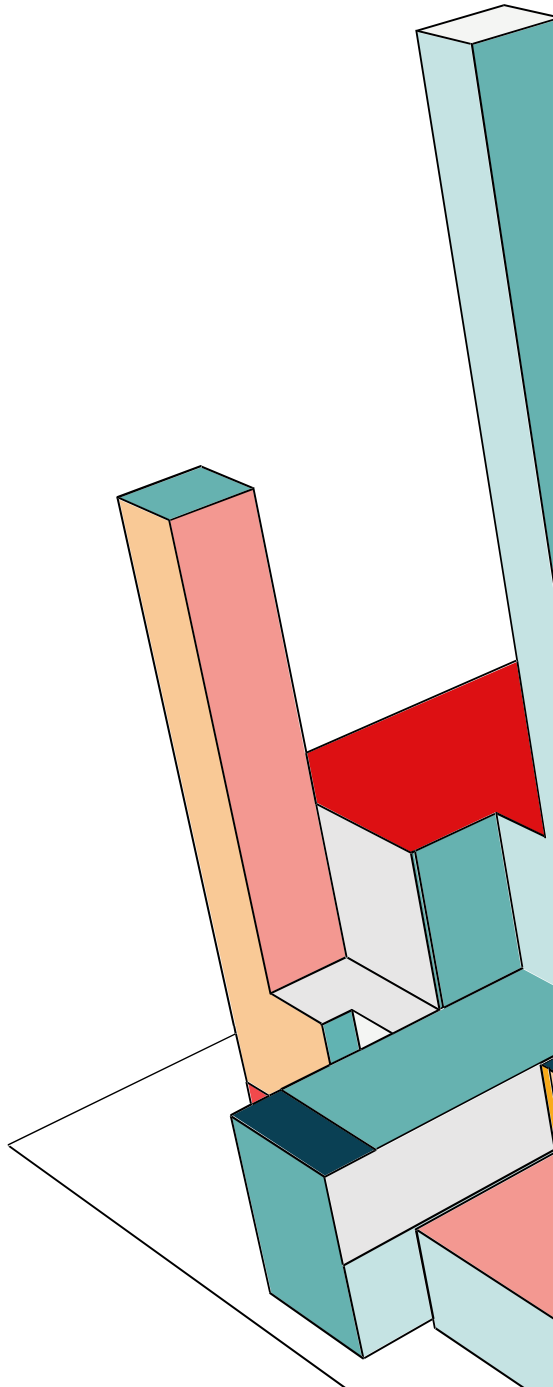
المحاضرة الثالثة

Design Concepts

بيرم زرتي - 2025

محاور المحاضرة

- Overview
- Modularization
- Cohesion
- Types of Cohesion
- Coupling
- Types of Coupling
- أمثلة



OVERVIEWS

- Why Software Design matters?

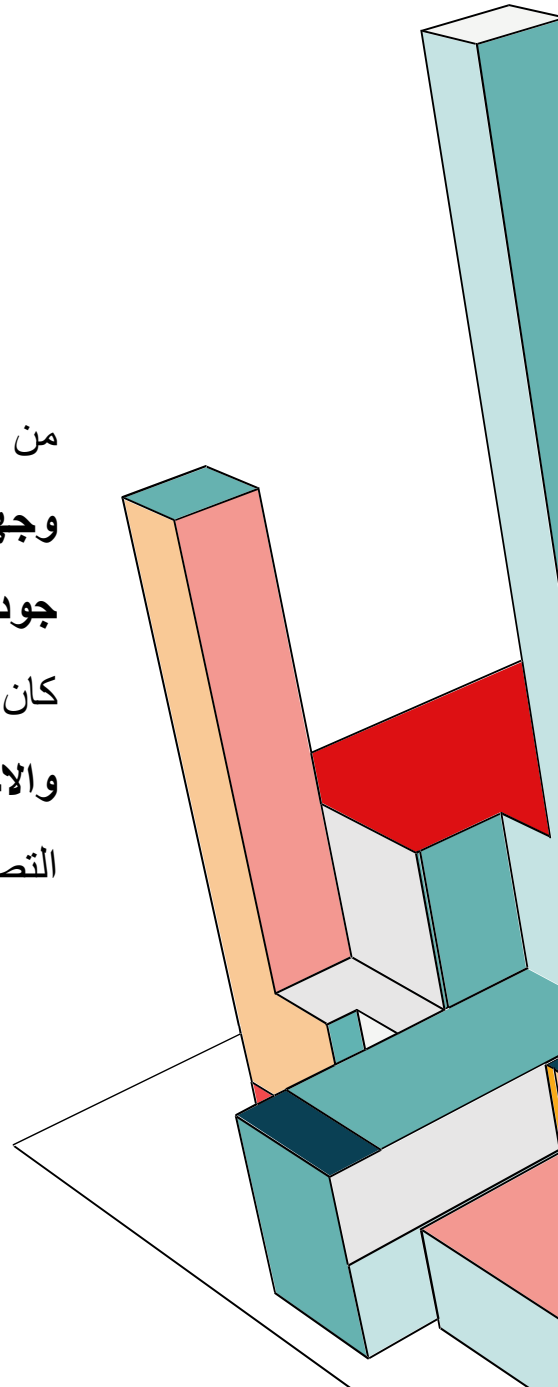
  Because it always takes more time to write something from scratch than changing something that is already there.

  With good design the time you spend to change (and test) something is also minimized

من المعروف أن **كتابة نظام برمجي من الصفر تستغرق وقتًا وجهدًا أكبر** مقارنة بتعديل نظام موجود مسبقًا. ومع ذلك، فإن **جودة التصميم** هي العامل الحاسم في مدى سهولة التعديل. فكلما كان التصميم جيدًا ومدروسًا، **قلّ الوقت اللازم لإجراء التعديلات والاختبارات**.

التصميم الجيد يُسهم في:

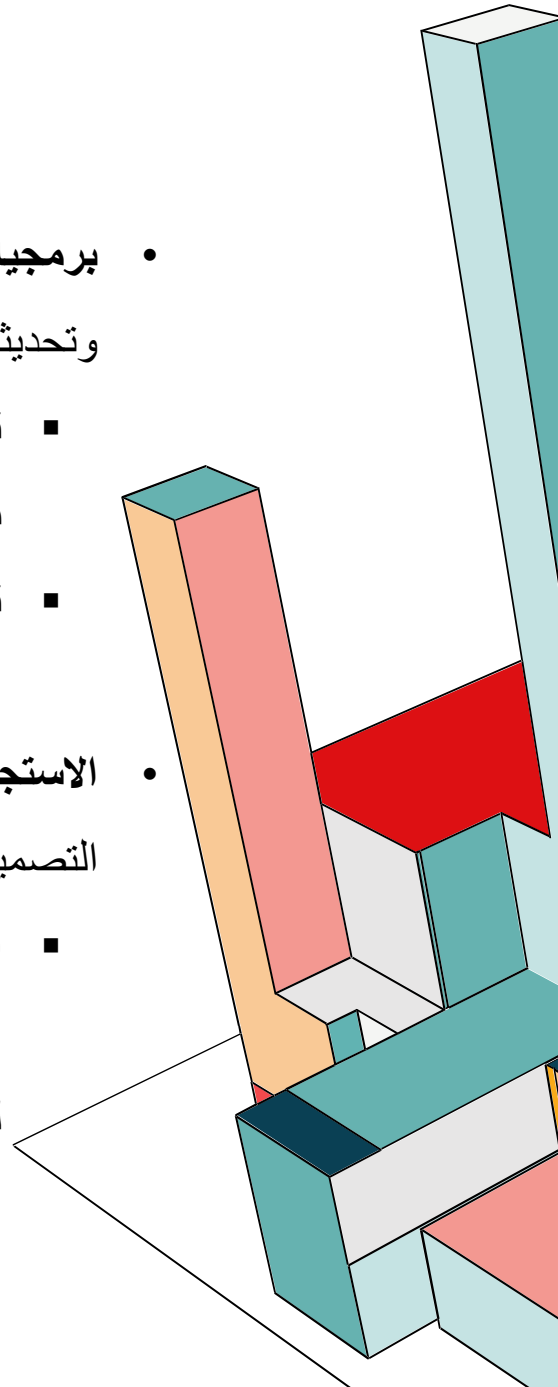
- تقليل التكرار والاعتماد المتبادل بين المكونات.
- تسهيل فهم النظام من قبل المطورين الجدد.
- دعم التوسع المستقبلي وتعديل المتطلبات بسهولة.
- تحسين جودة الاختبار وتقليل الأخطاء.



OVERVIEWS

- Why Software Design matters?
  - Maintainable software
    - Can implement new requirements with less cost
    - Can change existing implementation with less cost
  - Because requirement keep changing
    - Unit Test Programs also need to be changed
      - –They also need to have good design

- **برمجيات قابلة للصيانة** Maintainable Software يُسهل فهم النظام وتحديثه دون الحاجة لإعادة بناءه من الصفر.
  - **تنفيذ المتطلبات الجديدة بتكلفة أقل:** عند ظهور متطلبات جديدة، يمكن دمجها بسهولة ضمن التصميم الحالي دون تغييرات جذرية.
  - **تعديل التنفيذ الحالي بتكلفة أقل:** التصميم المرن يُتيح تعديل المكونات القائمة دون التأثير على باقي أجزاء النظام.
- **الاستجابة لتغير المتطلبات:** نظرًا لأن المتطلبات تتغير باستمرار، فإن التصميم الجيد يُمكّن النظام من التكيف بسرعة وفعالية.
  - **اختبارات الوحدة** Unit Tests **تحتاج إلى تصميم جيد أيضًا:** البرامج المخصصة للاختبار يجب أن تكون مصممة بشكل منظم لتواكب تغييرات النظام وتضمن دقة النتائج.



OVERVIEWS

- What is a bad designed code?

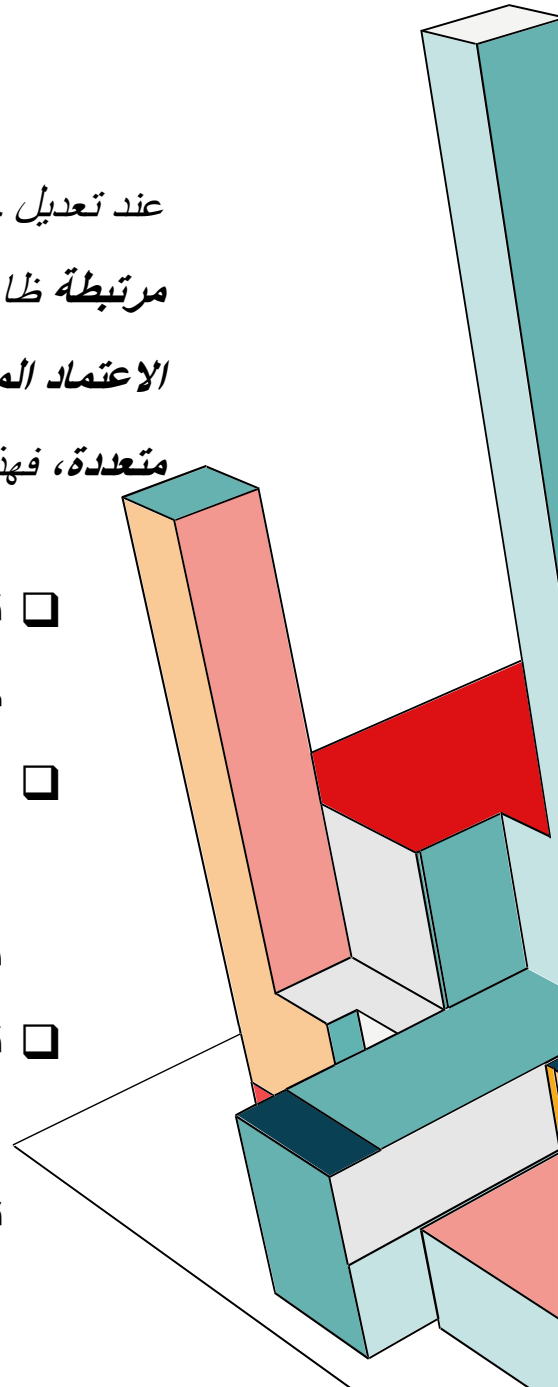
- If you change something
  - you break something else that is not related to the change
- if you change something
  - you need to change something else.

- How Software designers solve problems?
  - ❑ Dividing the problem
  - ❑ Isolating parts (each part has one goal/responsibility)
  - ❑ Reducing dependencies between different related elements

*عند تعديل جزء معين من الكود، قد يؤدي ذلك إلى **كسر وظائف أخرى غير مرتبطة** ظاهريًا بالتعديل، مما يدل على وجود ترابط غير محكوم بين المكونات.*

***الاعتماد المتبادل العالي:** إذا تطلب تعديل وظيفة واحدة **تعديل وظائف أخرى متعددة**، فهذا يشير إلى ضعف في الفصل بين المسؤوليات، ويزيد من تكلفة التغيير.*

- ❑ **تقسيم المشكلة** Divide and Conquer يتم تقسيم النظام إلى أجزاء صغيرة يمكن فهمها وتطويرها بشكل مستقل، مما يُقلل من التعقيد الكلي.
- ❑ **عزل الأجزاء** Isolation يتم تصميم كل جزء ليؤدي **مهمة واحدة فقط** Single Responsibility Principle ، مما يُسهل اختباره وتعديله دون التأثير على الأجزاء الأخرى.
- ❑ **تقليل الترابط** Low Coupling **وزيادة التماسك** High Cohesion تُصمم المكونات بحيث تكون مستقلة قدر الإمكان، مع تركيز كل مكون على وظيفة محددة



Dividing the problem

- Modularization

Isolating parts (each part has one goal/responsibility)

- High Cohesion. High cohesion is when you have a class that does a well-defined job. Low cohesion is when a class does a lot of jobs that don't have much in common.
- Cohesion refers to the degree to which the elements of an entity belong together.

Reducing dependencies between different related elements

- Lower Coupling. The goal of a loose/low coupling architecture is to reduce the risk that a change made within one element will create unexpected changes within other elements.
- Coupling refers to degree of interdependence between software entities.

تقسيم المشكلة Dividing the Problem يُعد تقسيم المشكلة إلى أجزاء صغيرة ومحددة الخطوة الأولى نحو تصميم فعال. هذا النهج يُقلل من التعقيد ويُسهل فهم النظام وتطويره تدريجيًا. التقسيم **إلى وحدات** Modularization يتم تنظيم النظام على شكل **وحدات مستقلة** Modules بحيث يمكن تطوير كل وحدة واختبارها وصيانتها بشكل منفصل.

العزل الوظيفي Isolating Parts يُصمم كل جزء في النظام ليؤدي **مهمة واحدة فقط**، وفقًا لمبدأ **المسؤولية الواحدة** Single Responsibility Principle، مما يُسهل فهمه وتعديله دون التأثير على باقي الأجزاء.

التماسك Cohesion يُشير إلى مدى ترابط عناصر الكيان الواحد.

الترابط Coupling يُشير إلى مدى اعتماد مكون على مكونات أخرى.

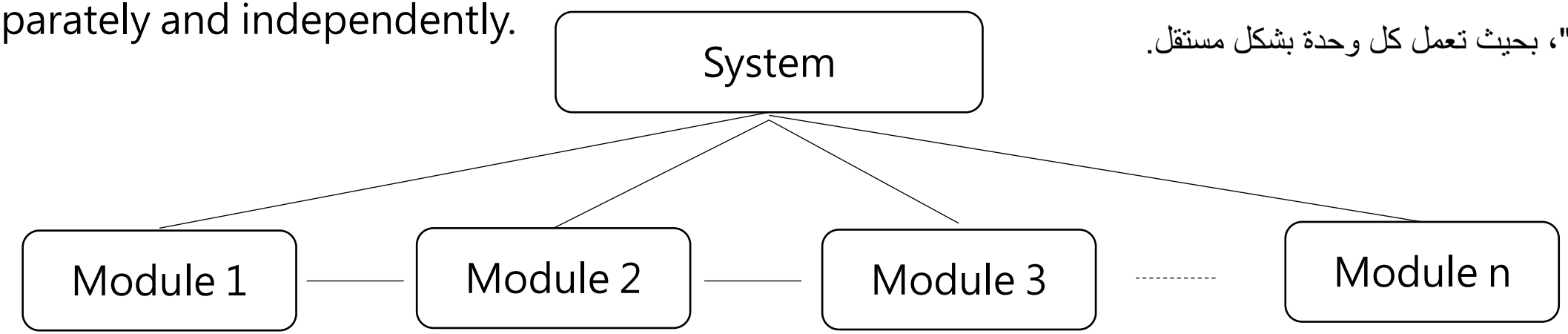
MODULARIZATION

Modularization is a technique to divide a software system into multiple independent modules, where each module works independently. Designers tend to design modules such that they can be executed and/or compiled separately and independently.

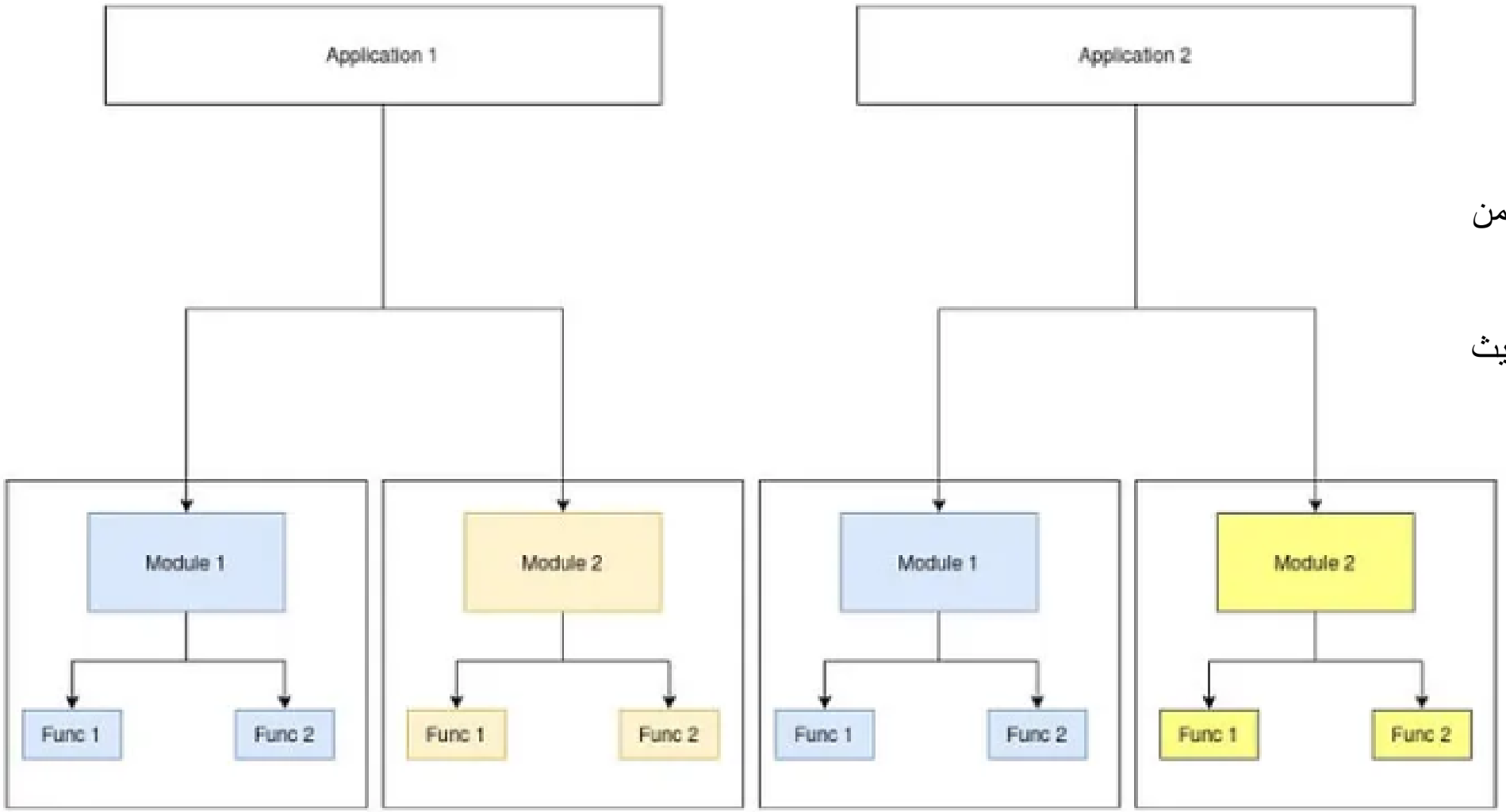
يُعد **التقسيم إلى وحدات** من التقنيات الأساسية في تصميم البرمجيات، ويهدف إلى **تفكيك النظام البرمجي إلى مجموعة من الوحدات المستقلة**، بحيث يمكن لكل وحدة أن تعمل بشكل منفصل عن الأخرى.

- يتم تصميم كل وحدة بحيث **تُنفذ وتُترجم** Compile **بشكل مستقل**.
- تُعزز هذه التقنية **قابلية إعادة الاستخدام**، و**سهولة الصيانة**، و**الاختبار المنفصل** لكل وحدة.
- تُقلل من التعقيد العام للنظام، وتُسهل فهمه وتطويره تدريجيًا.

مثال: في نظام إدارة مكتبة، يمكن تقسيم النظام إلى وحدات مثل "إدارة الكتب"، "إدارة المستخدمين"، "إدارة الإعارات"، بحيث تعمل كل وحدة بشكل مستقل.



MODULARIZATION



الهدف من التقسيم:

- **عزل الوظائف** بحيث تكون كل وحدة مسؤولة عن مهمة محددة.
- **تقليل الترابط** بين الوحدات، مما يُقلل من تأثير التغييرات.
- **تحسين التعاون بين فرق التطوير**، حيث يمكن توزيع العمل على فرق متعددة دون تعارض.

WHAT IS A MODULE?

A module refers to a self-contained and reusable piece of software that performs a specific set of related functions.

A module is a program unit with parts.

Immediate parts are those directly below the whole in the parts hierarchy.

Program

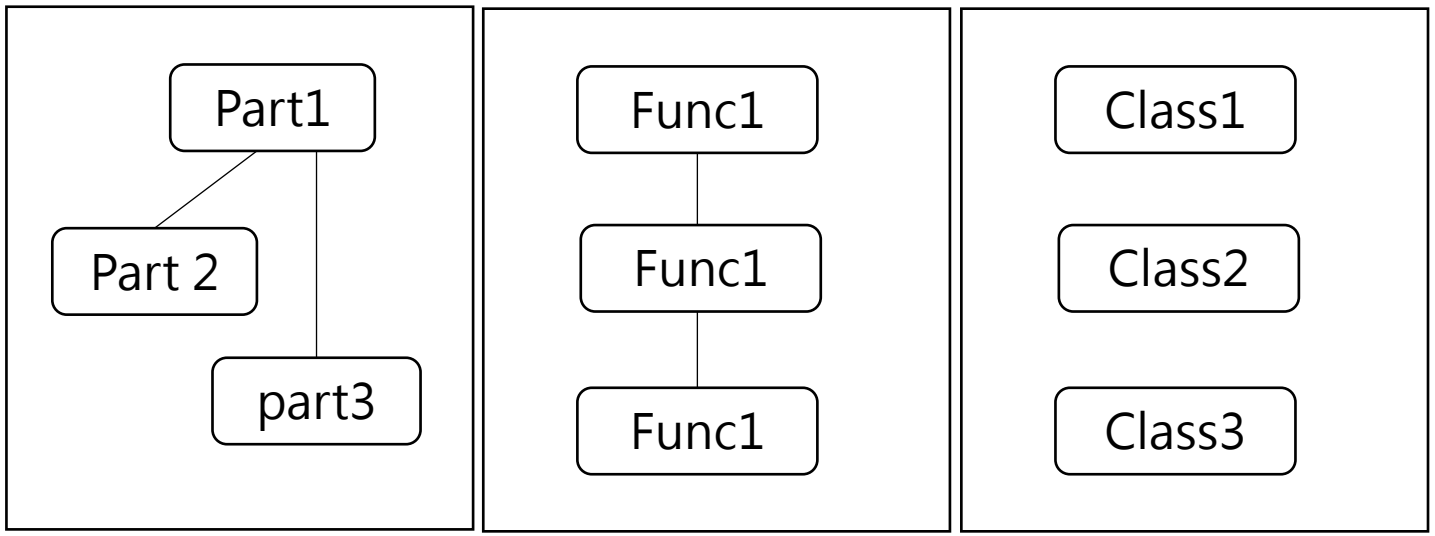
- Sub-programs or sub-systems
  - Packages, compilation units
    - Classes, functions
      - Attributes, operations, blocks
        - Lines of code

ما هي الوحدة البرمجية Module ؟

الوحدة البرمجية هي **جزء مستقل وقابل لإعادة الاستخدام من البرنامج**، يُنفذ مجموعة محددة من الوظائف ذات العلاقة. تُستخدم الوحدات لتقسيم النظام إلى أجزاء منظمة، مما يُسهل تطويره، اختباره، وصيانته.

خصائص الوحدة البرمجية:

- **مستقلة ذاتيًا:** يمكن تنفيذها أو ترجمتها Compile بشكل منفصل.
- **قابلة لإعادة الاستخدام:** يمكن استخدامها في أنظمة أو سياقات مختلفة.
- **تؤدي وظيفة محددة:** تركز على مجموعة مترابطة من المهام.



Module

Advantages of Modularization

There are many advantages of Modularization in software engineering.

Some of these are given below:

- Easier to understand each module and their purpose.
    - So easy to understand the system.
- Smaller components are easier to maintain,
    - System maintenance is easy
- Easier to reuse and refactor modules.
    - A module can be used many times as their requirements

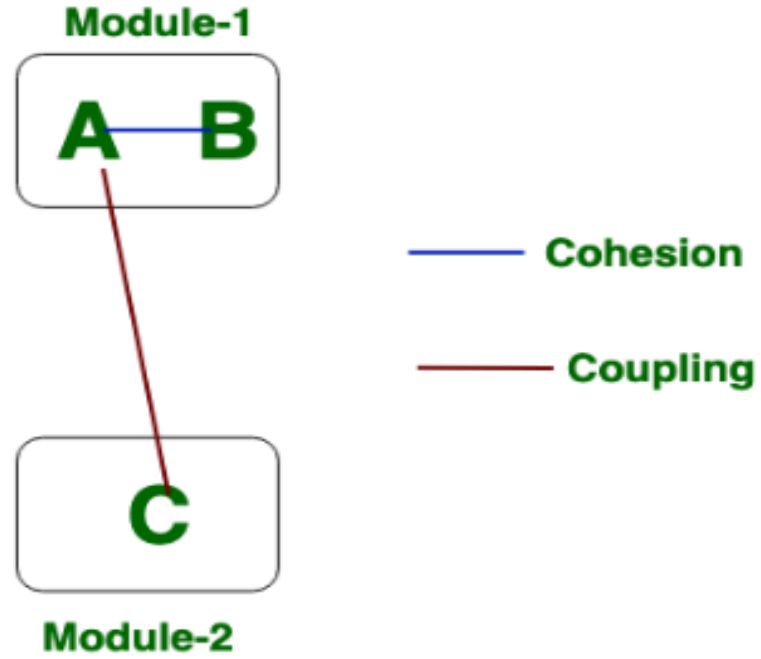
من مزايا التقسيم إلى وحدات

- **سهولة الفهم** يمكن فهم كل وحدة بشكل مستقل، مما يُسهل فهم النظام ككل، خاصة في الأنظمة الكبيرة والمعقدة.
- **سهولة الصيانة** المكونات الصغيرة تكون أسهل في التعديل والاختبار، مما يُقلل من تكلفة الصيانة ويُسرّع عملية التطوير.
- **إعادة الاستخدام** يمكن استخدام الوحدة نفسها في أكثر من مكان داخل النظام أو في أنظمة أخرى، مما يُقلل من التكرار ويُعزز الإنتاجية.
- **سهولة إعادة الهيكلة** يمكن تحسين أو تعديل تصميم الوحدة دون التأثير على باقي النظام، مما يُسهل التطوير المستمر وتحسين الأداء.
- **المرونة في التوسع** يمكن إضافة وحدات جديدة أو تعديل القائمة بسهولة، مما يُسهل التكيف مع المتطلبات المتغيرة.

SW DESIGN CONCEPTS

Coupling and Cohesion are two key concepts in software engineering that are used to measure the quality of a software system’ s design.

- **Cohesion** the extent to which a component has a single purpose or function
  - Usually, High cohesion is good
- **Coupling** the extent to which two components depend on each other for successful execution
  - Usually, Low coupling is good

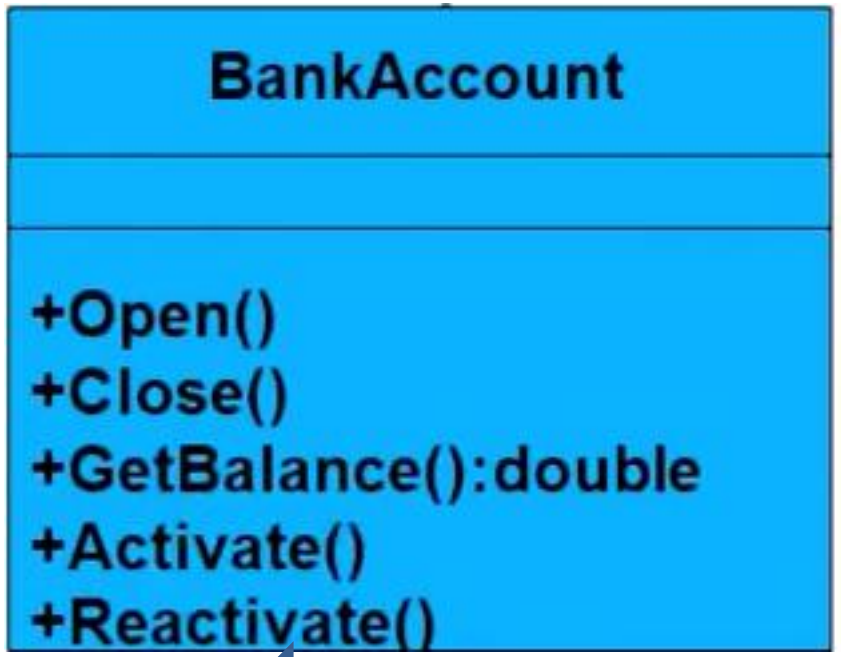


SW DESIGN CONCEPTS

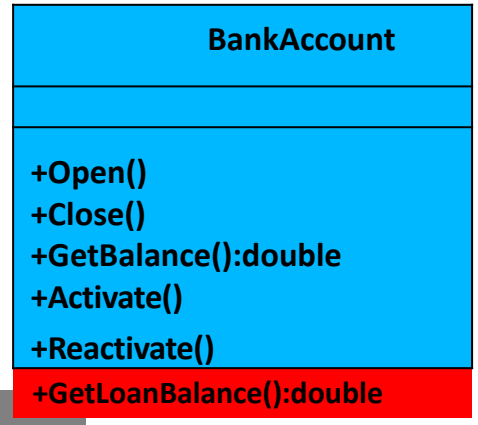
Cohesion

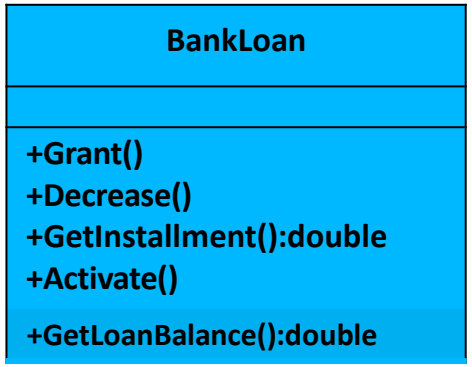
التماسك هو مقياس يُستخدم لتقييم مدى تركيز مكون برمجي (مثل الصف أو الوحدة) على **مهمة واحدة محددة** أو مجموعة مترابطة من الوظائف. كلما زاد التماسك، كان التصميم أكثر وضوحًا، وأسهل في الفهم والصيانة. يقيس التماسك إلى أي مدى تُجبر عناصر التصميم أو المعمارية على **التركيز على مسؤولية واحدة فقط**، دون تشتيت أو تداخل في الوظائف

Cohesion measures how much a design makes the design and architectural elements to be responsible of one and only.



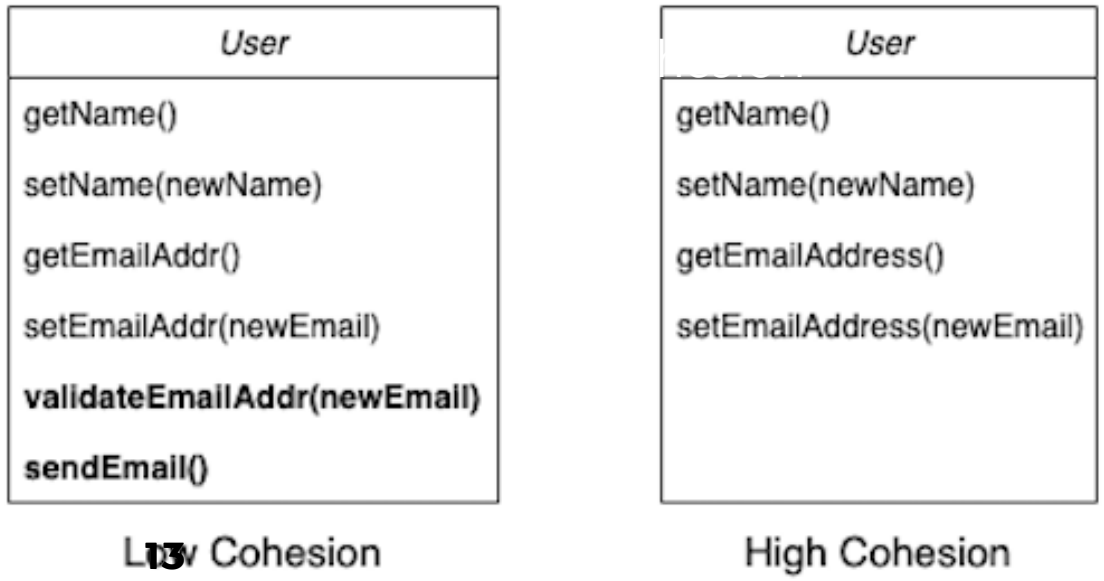
High Cohesion?!





SW DESIGN CONCEPTS

For example 2, a User class containing a method on how to validate the email address. User class can be responsible for storing the email address of the user but not for validating it or sending an email:



Low Cohesion

High Cohesion

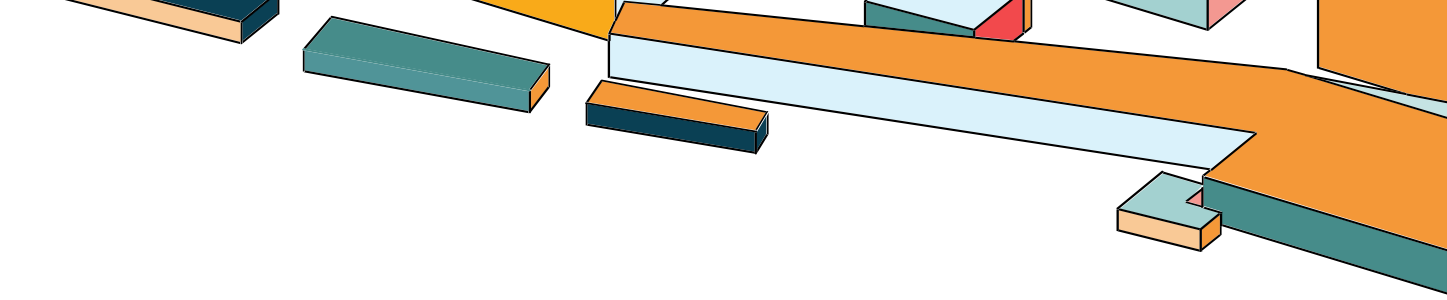
نفترض وجود صف يُدعى `User`، وظيفته الأساسية هي **تمثيل بيانات المستخدم**، مثل الاسم، البريد الإلكتروني، وكلمة المرور.

- **مسؤوليات مناسبة للصف** `User`:
  - تخزين البريد الإلكتروني.
  - إرجاع البريد الإلكتروني عند الطلب.
  - تحديث البريد الإلكتروني.
- **مسؤوليات غير مناسبة للصف** `User`:
  - التحقق من صحة البريد الإلكتروني Email Validation
  - إرسال بريد إلكتروني للمستخدم.

لماذا؟ لأن **التحقق من صحة البريد الإلكتروني** أو **إرسال الرسائل** هي وظائف لا ترتبط مباشرة بتمثيل بيانات المستخدم، بل ترتبط بوحدات أخرى مثل `EmailValidator`أو `EmailService`.

COHESION

- Refers to the degree to which the elements within a module or component of a software system are related to one another.
- It measures how closely the functions, procedures, or classes within a module work together to achieve a common purpose or goal.
- High cohesion is generally considered a desirable quality in software design because it leads to more maintainable, modular, and understandable code.
- Modules with high cohesion are easier to comprehend, modify as they encapsulate a specific and well-defined functionality



يُشير إلى **درجة العلاقة بين العناصر داخل الوحدة البرمجية**، مثل الدوال، الإجراءات، أو الأصناف.

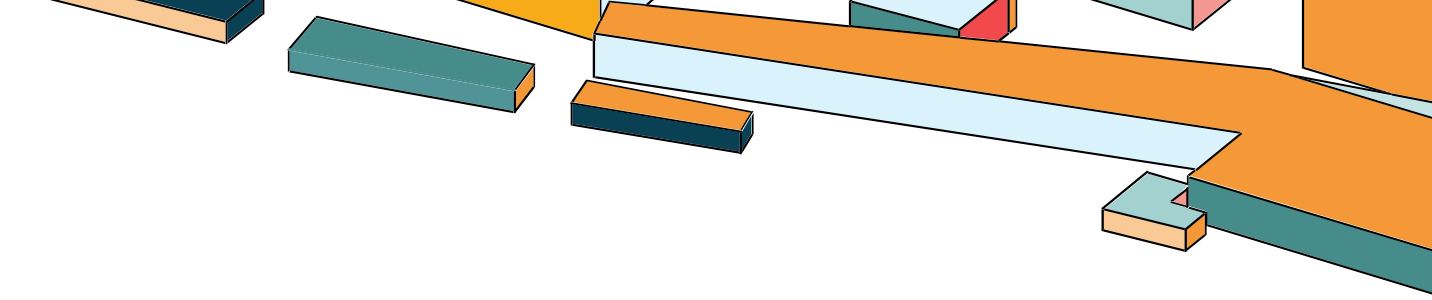
يُقاس من خلال مدى تعاون هذه العناصر لتحقيق **هدف مشترك وواضح** داخل الوحدة.

التماسك العالي يحدث عندما تكون جميع العناصر داخل الوحدة **مرتبطة وظيفيًا** وتُساهم في تحقيق وظيفة واحدة محددة. يُعد من **الصفات المرغوبة** في التصميم البرمجي لأنه يؤدي إلى:

- كود أكثر قابلية للفهم.
- سهولة في التعديل والصيانة.
- تصميم أكثر تنظيمًا وقابلية لإعادة الاستخدام.

التماسك العالي يُسهم في بناء وحدات برمجية واضحة، قابلة للصيانة، وسهلة الفهم، وهو من المبادئ الأساسية للتصميم الجيد.

TYPES OF COHESION



1. Functional Cohesion:

This type of cohesion occurs when all elements or tasks in a module contribute to a single well-defined function or purpose, and there is little or no coupling between the elements.

Functional cohesion is considered the most desirable type of cohesion as it leads to more maintainable and reusable code.

التماسك الوظيفي هو أحد أنواع التماسك في التصميم البرمجي، ويُعد **الأكثر مثالية** من حيث الجودة الهندسية. يحدث هذا النوع من التماسك عندما تُساهم جميع العناصر داخل الوحدة البرمجية (مثل الدوال أو الإجراءات) في **تحقيق وظيفة واحدة محددة وواضحة**.

- جميع المهام داخل الوحدة **مرتبطة بهدف مشترك**.
- لا يوجد ترابط أو اعتماد غير ضروري بين العناصر الداخلية.
- يُسهل فهم الوحدة واختبارها وتعديلها دون التأثير على أجزاء أخرى.
- يُعزز قابلية **إعادة الاستخدام** لأن الوحدة تؤدي وظيفة محددة يمكن توظيفها في سياقات متعددة.

التماسك الوظيفي هو المعيار الذهبي للتصميم الجيد، لأنه يُحقق وضوح الوظيفة، ويُقلل من التعقيد، ويُسهل التطوير المستقبلي. وهو مرتبط ارتباطًا وثيقًا بمبدأ المسؤولية الواحدة، الذي يُعد من أهم مبادئ التصميم الكائني Object-Oriented Design .

Note: **Single Responsibility Principle (SRP):** Ensure that each function or method within the module has a single responsibility or task.

FUNCTIONAL COHESION

For example, both functions are focused on calculations related to a circle - one calculates the area, and the other calculates the circumference. The module is functionally cohesive because both functions serve a common purpose of circle-related calculations.

لنفترض وجود وحدة برمجية تحتوي على دالتين:

- `calculateArea(radius)` - لحساب مساحة الدائرة.
- `calculateCircumference(radius)` - لحساب محيط الدائرة.

لماذا يُعد هذا المثال عالي التماسك؟

- كلا الدالتين تُركزان على **وظائف رياضية مرتبطة بالدائرة**.
- تخدمان **هدفًا مشتركًا** وهو إجراء عمليات حسابية خاصة بالدائرة.
- لا توجد وظائف غير مترابطة داخل الوحدة، مثل الرسم أو التحقق من صحة البيانات، مما يُعزز التماسك.

هذه الوحدة تُعد **وظيفيًا متماسكة** لأنها تُنفذ مجموعة من المهام المرتبطة ارتباطًا وثيقًا بهدف واحد محدد.

```python
# Example in Python
def calculate_area(radius):
    return 3.14 * radius * radius

def calculate_circumference(radius):
    return 2 * 3.14 * radius
```

Functional cohesion

Grouped related to operations of a task.

TYPES OF COHESION

2. Sequential Cohesion:

- Functions or methods within a module are arranged in a sequential order, where the output of one function is the input to the next.
- The output of an element is the input of other element in a module i.e., data flow between the parts.

التماسك التسلسلي هو نوع من أنواع التماسك في التصميم البرمجي، ويحدث عندما تُرتب الوظائف أو الإجراءات داخل الوحدة البرمجية **بشكل متسلسل**، بحيث يكون **ناتج وظيفة هو مدخل للوظيفة التالية**.

الوظائف داخل الوحدة مترابطة من خلال **تدفق البيانات**.

•كل وظيفة تُنفذ جزءًا من العملية، وتعتمد على نتائج الوظيفة السابقة.

مثال: وحدة تقوم بمعالجة بيانات نصية:

1. `readText()` - تقرأ النص من ملف.
2. `cleanText()` - تُنظف النص من الرموز غير المرغوبة.
3. `analyzeText()` - تُحلل النص وتُستخرج الكلمات المفتاحية.

في هذا المثال، كل وظيفة تعتمد على ناتج الوظيفة السابقة، مما يُحقق **تماسكًا تسلسليًا**.

فوائد التماسك التسلسلي:

•يُسهل تتبع تدفق البيانات داخل الوحدة.

•يُحافظ على تنظيم الوظائف ضمن سياق مشترك.

•يُعزز قابلية الفهم والاختبار، خاصة في العمليات متعددة الخطوات.

رغم أن التماسك التسلسلي يُعد أفضل من التماسك المنخفض، إلا أنه **أقل مثالية من التماسك الوظيفي**، لأن الوظائف قد تكون مترابطة بالبيانات فقط وليس بالهدف الوظيفي الكامل.

SEQUENTIAL COHESION

Example, with Order Processing: : functions represent the steps involved in processing an order: receiving the order, validating it, processing payment, and shipping the order.

```python
# Example in Python
def receive_order(order_details):
    # ...

def validate_order(order):
    # ...

def process_payment(order):
    # ...

def ship_order(order):
    # ...
```

لماذا يُعد هذا المثال تماسكًا تسلسليًا؟
كل وظيفة تُنفذ خطوة من خطوات معالجة الطلب. ناتج كل خطوة يُستخدم كمدخل للخطوة التالية. الوظائف مترابطة من خلال تدفق البيانات، لكنها لا تؤدي بالضرورة نفس الوظيفة.

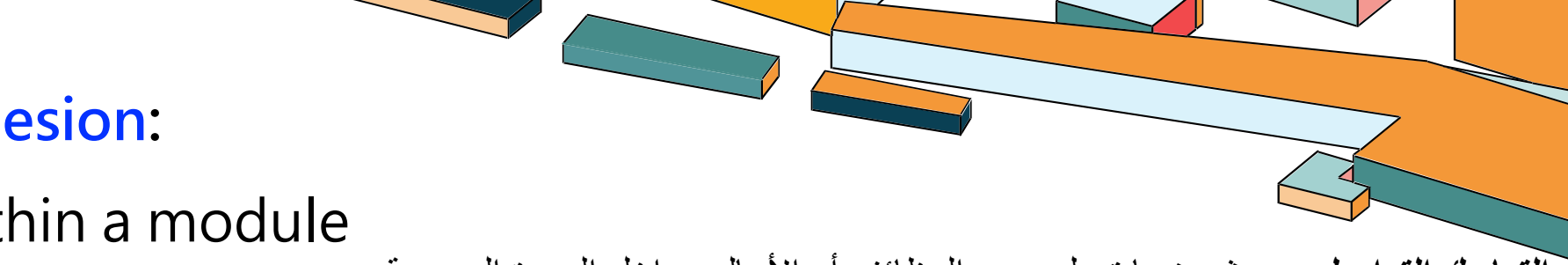
TYPES OF COHESION

3. Communicational Cohesion:

Functions or methods within a module operate on the same set of data.

Functions within a communicational cohesive module share data structures, variables, or parameters.

Changes to the structure or format of the shared data may affect multiple functions, potentially leading to maintenance challenges (**Drawbacks**).



التماسك التواصلي يحدث عندما تعمل جميع الوظائف أو الأساليب داخل الوحدة البرمجية على **نفس مجموعة البيانات المشتركة**، مثل المتغيرات أو الهياكل أو المعاملات.

خصائص التماسك التواصلي:

- الوظائف داخل الوحدة **تُشارك نفس البيانات**، لكنها قد تُنفذ مهام مختلفة.
- يُعد هذا النوع من التماسك **أفضل من التماسك الإجرائي أو المنطقي**، لكنه **أقل مثالية من التماسك الوظيفي**.

مثال: وحدة `CustomerProfile` تحتوي على:
- `updateAddress()` لتحديث عنوان العميل.
- `validatePhoneNumber()` للتحقق من رقم الهاتف.
- `formatProfile()` لتنسيق بيانات العرض.

جميع هذه الوظائف تعمل على نفس هيكل بيانات العميل، مما يُحقق تماسكًا تواصليًا.

التحديات Drawbacks

- **تغيير هيكل البيانات المشترك** قد يُؤثر على عدة وظائف داخل الوحدة، مما يزيد من **تكلفة الصيانة**.
- **الاعتماد على البيانات المشتركة** قد يُصعّب اختبار الوظائف بشكل مستقل.

COMMUNICATIONAL COHESION

Example, both functions operate on the same set of data (the list of numbers). The data is shared between the functions, and each function performs a different operation on that shared data..

```python
# Example in Python
def calculate_average(nums):
    total = sum(nums)
    count = len(nums)
    return total / count

def find_maximum(nums):
    return max(nums)
```

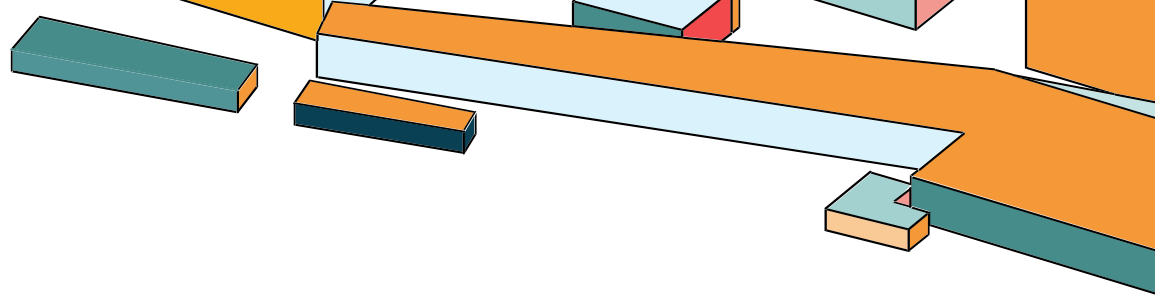
لماذا يُعد هذا المثال تماسكًا تواصليًا؟كلا الدالتين (calculate_average و find_maximum) تعملان على نفس مجموعة البيانات وهي numbers. كل وظيفة تُنفذ عملية مختلفة على نفس البيانات المشتركة.لا يوجد تسلسل مباشر بين الوظائف، لكن البيانات المشتركة تربط بينها.

ADVANTAGES OF HIGH COHESION

- **Improved readability and understandability**: High cohesion results in clear, focused modules with a single, well-defined purpose, making it easier for developers to understand the code and make changes.
- **Better error isolation**: High cohesion reduces the likelihood that a change in one part of a module will affect other parts, making it easier to isolate and fix errors.
- **Improved reliability:** High cohesion leads to modules that are less prone to errors and that function more consistently, leading to an overall improvement in the reliability of the system.

- **تحسين قابلية القراءة والفهم** الوحدات ذات التماسك العالي تُركز على وظيفة واحدة محددة. هذا يُسهل على المطورين **فهم الغرض من الوحدة**، مما يُسرّع عملية التطوير والتعديل.
- **عزل الأخطاء بشكل أفضل** عند حدوث خطأ، يكون من السهل **تحديد مصدره داخل الوحدة**. التماسك العالي يُقلل من احتمال أن تؤثر التغييرات في جزء من الوحدة على أجزاء أخرى غير مرتبطة.
- **تحسين الموثوقية** الوحدات المتماسكة تعمل بشكل أكثر **اتساقًا واستقرارًا**. تقل احتمالية ظهور الأخطاء الناتجة عن تداخل الوظائف أو ضعف التنظيم الداخلي. هذا يُؤدي إلى **تحسين موثوقية النظام ككل**.

DISADVANTAGES OF LOW COHESION



- **Increased code duplication:** Can lead to the duplication of code, as elements that belong together are split into separate modules.
- **Reduced functionality:** Can result in modules that lack a clear purpose and contain elements that don’ t belong together, reducing their functionality and making them harder to maintain.
- **Difficulty in understanding the module:** Can make it harder for developers to understand the purpose and behavior of a module, leading to errors and a lack of clarity.

- زيادة التكرار في الكود عند توزيع الوظائف ذات العلاقة على وحدات متعددة، قد يتم تكرار نفس الكود في أكثر من مكان. هذا يُصعّب عملية الصيانة ويزيد من احتمال حدوث أخطاء عند تعديل أحد النسخ دون الأخرى.
- ضعف الوظائف الوحدات التي تحتوي على عناصر غير مترابطة تفتقر إلى هدف واضح. هذا يُقلل من فعاليتها ويجعلها أقل قابلية لإعادة الاستخدام أو التطوير.
- **صعوبة الفهم عندما تكون الوحدة غير متماسكة،** يُصبح من الصعب على المطورين فهم الغرض منها أو كيفية عملها. هذا يُؤدي إلى ارتباك في الاستخدام، وزيادة احتمال ارتكاب الأخطاء أثناء التطوير أو الصيانة.

COUPLING

- In general, low coupling is desirable because it leads to more modular, maintainable, and flexible software. On the other hand, high coupling can make the system more rigid, harder to maintain, and less adaptable to changes.
- Coupling refers to the degree of interdependence between software modules.
- It measures how much one module knows about or relies on the internals of another module.
- High coupling means that modules are closely connected and changes in one module may affect other modules.
- Low coupling means that modules are independent and changes in one module have little impact on other modules.

الترابط هو مقياس يُستخدم لتحديد مدى **اعتماد وحدة برمجية على وحدات أخرى** داخل النظام.

يُعبر عن درجة التداخل أو التفاعل بين المكونات، ويُعد من العوامل الحاسمة في جودة التصميم. يُشير إلى **درجة الاعتماد أو المعرفة** التي تمتلكها وحدة برمجية عن تفاصيل وحدة أخرى. كلما زاد الترابط، زادت **الاعتمادية المتبادلة** بين الوحدات، مما يُصعّب تعديل النظام دون التأثير على أجزائه الأخرى.

الترابط العالي الوحدات تكون **مرتبطة ارتباطًا وثيقًا**.أي تغيير في وحدة قد يُؤثر على وحدات أخرى.يُؤدي إلى تصميم **صلب وغير مرن**، ويُصعّب الصيانة والتوسع.

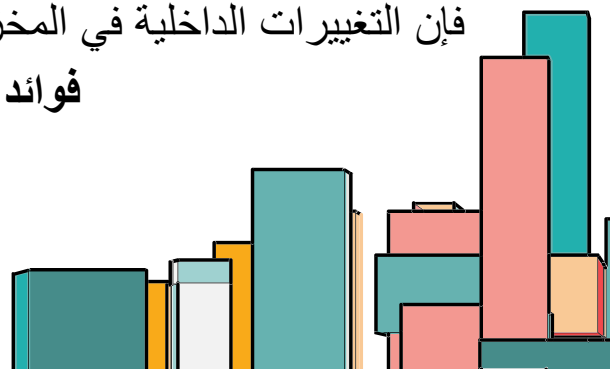
مثال: إذا كانت وحدة "الدفع" تعتمد مباشرة على تفاصيل تنفيذ وحدة "المخزون"، فإن أي تعديل في المخزون قد يُسبب خللًا في الدفع.

الترابط المنخفض الوحدات تكون **مستقلة نسبيًا**. يمكن تعديل وحدة دون التأثير على الأخرى. يُؤدي إلى تصميم **مرن، قابل للصيانة، وسهل التوسع**.

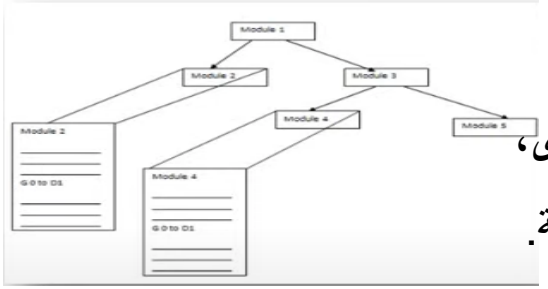
مثال: إذا كانت وحدة "الدفع" تتفاعل مع "المخزون" عبر واجهة مجردة Interface، فإن التغييرات الداخلية في المخزون لا تؤثر على الدفع.

فوائد الترابط المنخفض:

- **تحسين قابلية الصيانة**.
- **تقليل الأخطاء الناتجة عن التداخل**.
- **تعزيز إعادة الاستخدام**.
- **تسهيل اختبار الوحدات بشكل مستقل** ومستدام.



TYPES OF COUPLING

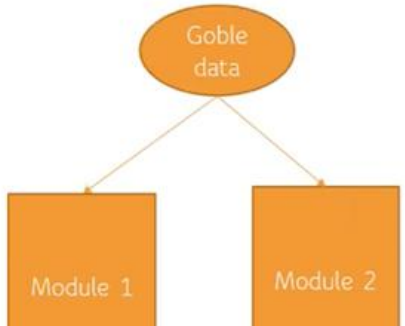


Type 1: Content Coupling

- Here, Two modules are connected as they share the same content like functions, methods.
- When a change is made in one module the other module needs to be updated as well.

Type 2: Common Coupling:

- Two modules are common coupled if they share information through some global data items.



الترابط بالمحتوى يحدث عندما **تصل وحدة برمجية مباشرة إلى محتوى وحدة أخرى**، مثل التعديل على متغيراتها الداخلية أو استدعاء وظائفها الخاصة بطريقة غير رسمية.

- يُعد **أقوى أنواع الترابط وأكثرها خطورة**، لأنه يُسبب:
  - **اعتمادًا مباشرًا** بين الوحدات.
  - **هشاشة في النظام**، حيث أن أي تغيير في وحدة يُجبر على تعديل الوحدة الأخرى.
  - **صعوبة في الصيانة والاختبار**.

النوع الثاني: الترابط المشترك يحدث عندما **تُشارك وحدتان أو أكثر بيانات عامة** Global Data. كل وحدة تعتمد على نفس المتغيرات أو الهياكل العامة، مما يُسبب:

- **تداخلًا غير مباشر** بين الوحدات.
- **صعوبة في تتبع مصدر التغييرات**.
- **احتمالية حدوث أخطاء غير متوقعة** عند تعديل البيانات المشتركة.

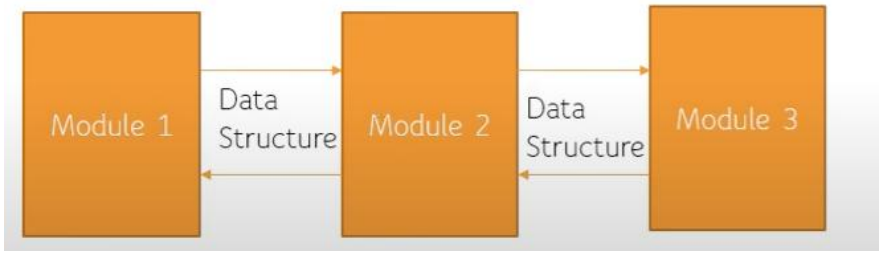
TYPES OF COUPLING

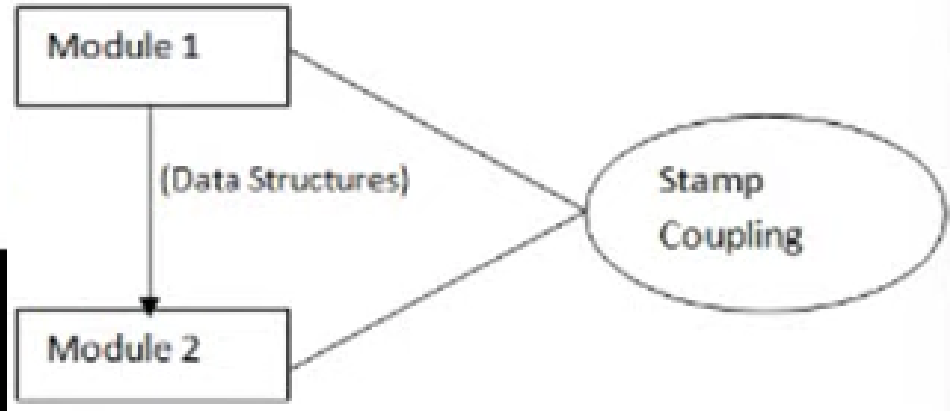
Type 3: Stamp Coupling:

Two modules are stamp coupled if they communicate using composite data items such as Complete Data Structure & objects. The complete data structure is passed from one module to another module.

Example:

الترابط بالطابع يحدث عندما تتبادل وحدتان برمجيتان **هياكل بيانات مركبة أو كائنات كاملة** (مثل كائنات أو سجلات)، حتى وإن كانت الوحدة المستقبِلة لا تحتاج إلى كل محتويات هذا الهيكل.






```
// Module A
public class ModuleA {
    public void processObject(MyObject obj) {
        // Code here
    }
}

// Module B
public class ModuleB {
    public void processObject(MyObject obj) {
        // Code here
    }
}
```

TYPES OF COUPLING

Type 4: Data Coupling

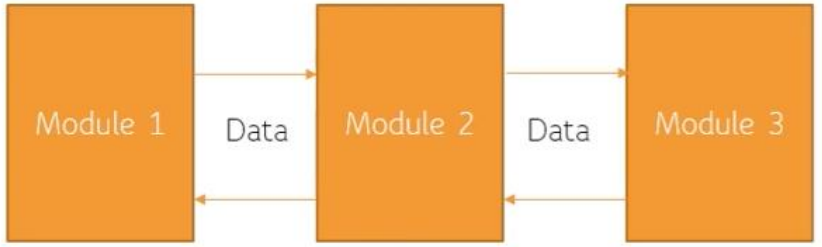
- When data are passed from one modules to another module via argument list or parameters through functional blocks.
- If the dependency between the modules is based on the data only, then the modules are said to be data coupled.

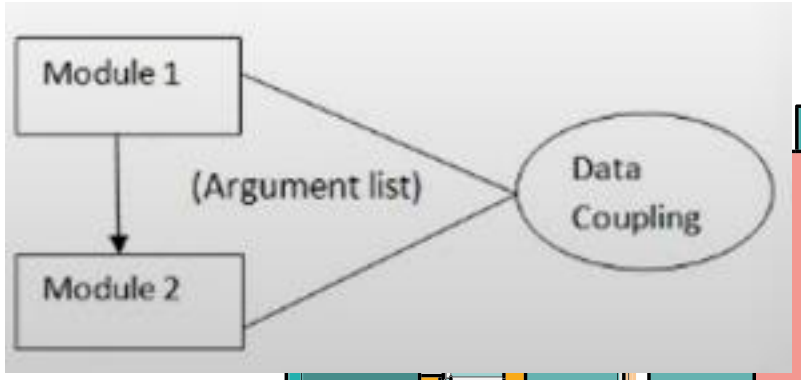
Example:

الترابط عبر البيانات يحدث عندما تتبادل الوحدات البرمجية المعلومات فيما بينها **عبر قائمة المعاملات أو الوسائط** Parameters، دون أن تعتمد على تفاصيل تنفيذ بعضها البعض.

- يتم تمرير **بيانات بسيطة أو أولية** مثل أرقام، نصوص، أو قيم منطقية.
- الوحدات تعتمد فقط على **البيانات المُرسلة**، وليس على البنية الداخلية للوحدة الأخرى.
- يُعد من **أنواع الترابط المنخفضة والمفضلة** لأنه يُعزز استقلالية الوحدات.

```
// Module A
public class ModuleA {
    public void processData(int data) {
        // Code here
    }
}

// Module B
public class ModuleB {
    public void processData(int data) {
        // Code here
    }
}
```

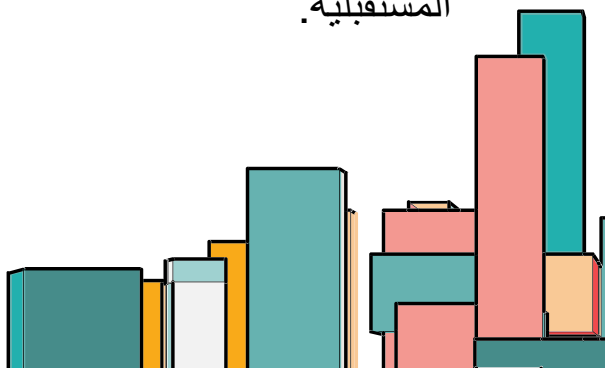




ADVANTAGES OF LOW COUPLING

- Improved maintainability: Reduces the impact of changes in one module on other modules, making it easier to modify or replace individual components without affecting the entire system.
- Enhanced modularity: Allows modules to be developed and tested in isolation, improving the modularity and reusability of code.
- Better scalability: facilitates the addition of new modules and the removal of existing ones, making it easier to scale the system as needed. Easier adaptability to new requirements.

- **حسين قابلية الصيانة** يقلل من تأثير التغييرات في وحدة على الوحدات الأخرى. يُسهل تعديل أو استبدال مكون معين دون التأثير على النظام بأكمله.
- **تعزيز التقسيم إلى وحدات** يُتيح تطوير واختبار الوحدات بشكل مستقل. يُعزز إعادة استخدام الكود في سياقات مختلفة.
- **تحسين قابلية التوسع** ُسهل إضافة وحدات جديدة أو إزالة وحدات قائمة دون تعقيد. يُساعد في **توسيع النظام تدريجيًا** حسب الحاجة. **سهولة التكيف مع المتطلبات الجديدة** يُتيح تعديل النظام ليتماشى مع التغيرات في المتطلبات أو البيئة التقنية. يُقلل من تكلفة التعديلات المستقبلية.



DISADVANTAGES OF HIGH COUPLING:

- Increased complexity: Increases the interdependence between modules, making the system more complex and difficult to understand.
- Reduced flexibility: makes it more difficult to modify or replace individual components without affecting the entire system.
- Decreased modularity: Makes it more difficult to develop and test modules in isolation, reducing the modularity and reusability of code.

زيادة التعقيد

الترابط العالي يُسبب **اعتمادًا متبادلًا بين الوحدات**.

أي تغيير في وحدة واحدة قد يتطلب تعديلات في وحدات أخرى.

يُصعّب فهم النظام ككل، خاصة في الأنظمة الكبيرة.

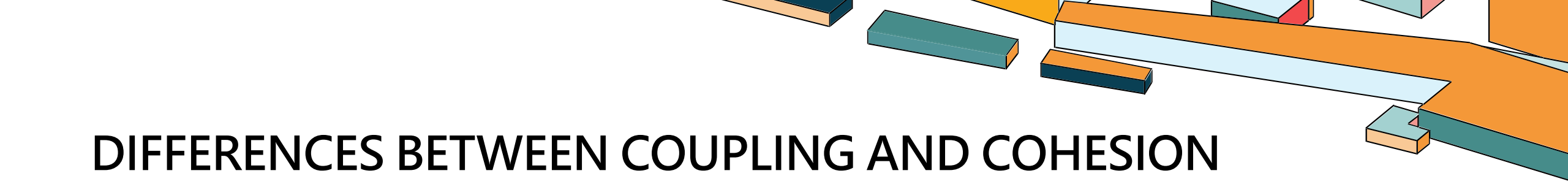
انخفاض المرونة

يُصبح من الصعب **تعديل أو استبدال مكون معين** دون التأثير على بقية النظام.

•يُعيق التكيف مع المتطلبات الجديدة أو التغييرات المستقبلية.

ضعف التقسيم إلى وحدات

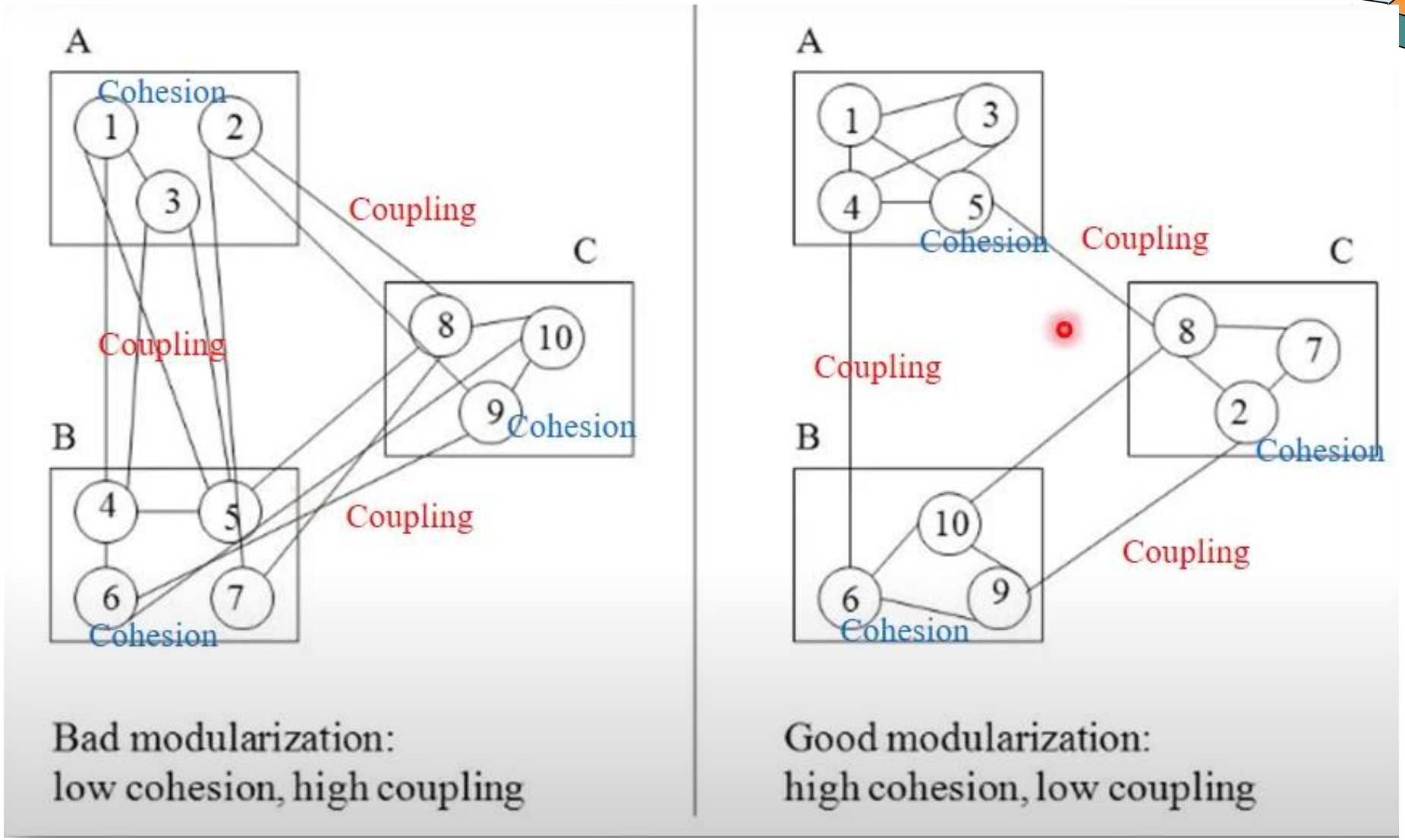
يُصعّب تطوير واختبار الوحدات بشكل مستقل. يُقلل من **قابلية إعادة الاستخدام** ويزيد من الاعتمادية غير المرغوبة.



DIFFERENCES BETWEEN COUPLING AND COHESION

Cohesion	Coupling
Cohesion represents the relationship within a module.	Coupling represents the relationships between modules.
Increasing cohesion is good for software.	Increasing coupling is avoided for software.
Cohesion represents the functional strength of module.	Coupling represents the independence among modules.
Highly cohesive gives the best software.	Whereas loosely coupling gives the best software.
In cohesion, the module focuses on a single thing.	In coupling, modules are connected to the other modules.
Cohesion is created between the same module.	Coupling is created between two different modules.
Types of Cohesion 1. Functional Cohesion. 2. Sequential Cohesion. 3. Communication Cohesion 4. Layer Cohesion.	Types of Coupling 1. Data Coupling 2. Stamp Coupling 3. Common Coupling. 4. External Coupling.

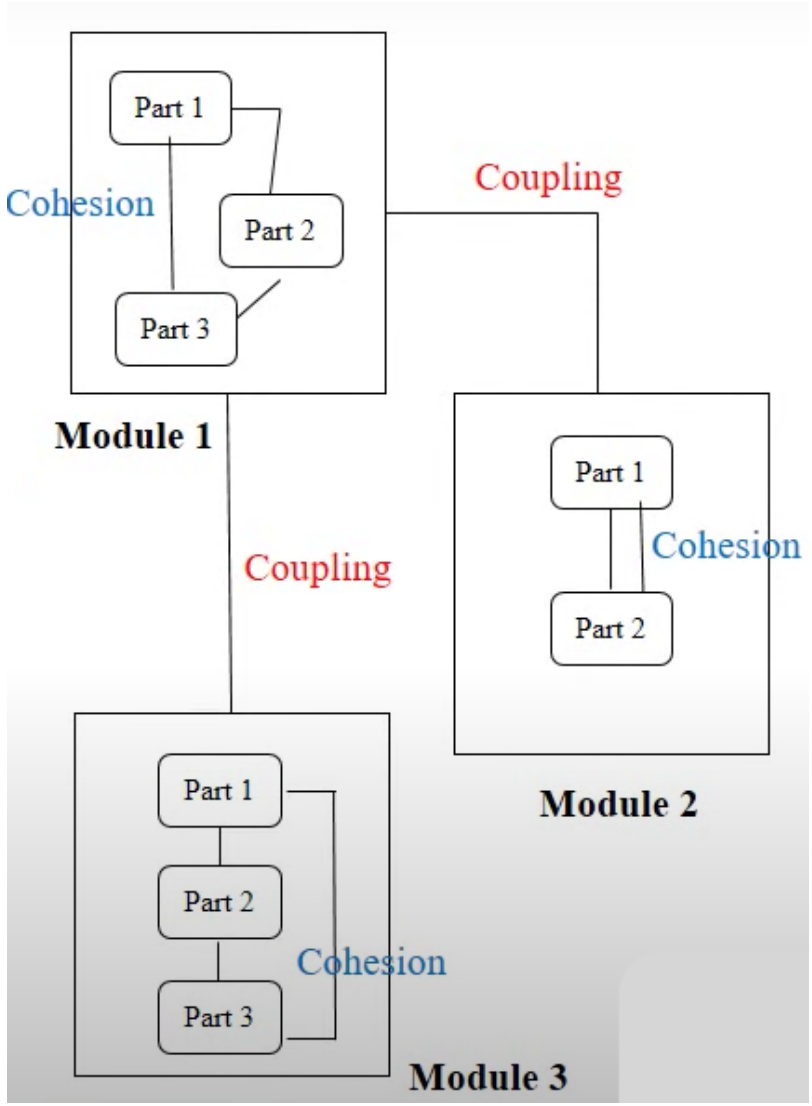
GOOD AND BAD SOFTWARE DESIGN



WHY HIGH COHESIVE & LOW COUPLING GENERATE GOOD DESIGN?

➢ **Due to Low Coupling**

- Readability: Modules are easy to understand not complex.
- Maintainability: Changes in one module little impact on other.
- Modularity: Enhance modules development.
- Scalability: Adding new module remove existing one easy.
- Testability: Modules are easy to test & debug.



EXAMPLE 1

Example

- *Let's consider a hypothetical example where we have a module responsible for handling customer data, including both personal information and purchase history. This module contains various functions, and each function deals with a different aspect of customer data:*
- *Class: handling customer data,*

```
class CustomerDataHandler:
    def get_personal_info(customer_id):
        # Code to retrieve and display personal
information

    def get_purchase_history(customer_id):
        # Code to retrieve and display purchase
history

    def update_personal_info(customer_id,
new_info):
        # Code to update personal information

    def process_payment(customer_id, amount):
        # Code to process a payment

    # ... other functions related to customer data
```

Low or High cohesion ??

EXAMPLE 1

Why is the CustomerDataHandler class not cohesive?

Answer*: deal with different aspects of customer data, such as personal information, purchase history, updating personal information, and processing payments. .*

Solution ?:

High cohesion !

Make three classes:

- ***class CustomerInfoHandler***
- ***class PurchaseHistoryHandler.***
- ***class PaymentProcessor.***

```
class CustomerInfoHandler:
    def get_personal_info(customer_id):
        # Code to retrieve and display personal
information

    def update_personal_info(customer_id, new_info):
        # Code to update personal information

    # ... other functions related to personal
information

class PurchaseHistoryHandler:
    def get_purchase_history(customer_id):
        # Code to retrieve and display purchase history

    # ... other functions related to purchase history

class PaymentProcessor:
    def process_payment(customer_id, amount):
        # Code to process a payment
    # ... other functions related to payment
processing
```

EXAMPLE 2

Example

In this example, the LibraryManager class contains functions related to adding/removing books, searching for books, calculating fines, and displaying member information

Low or High cohesion ??

```
class LibraryManager:
    def add_book(book):
        # Code to add a book to the library

    def remove_book(book_id):
        # Code to remove a book from the library
        # ...

    def search_book_by_title(title):
        # Code to search for a book by title
        # ...
    def calculate_fine(member_id, days_overdue):
        # Code to calculate the fine for a member
based on overdue days
        # ...

    def display_member_info(member_id):
        # Code to display information about a library
member
        # ...
    # ... other functions related to various concerns
```

EXAMPLE 2

Why is the LibraryManager class not cohesive?

Answer: *The functions cover a wide range of concerns, resulting in lower cohesion. The class is trying to manage various aspects of the library system, making it less maintainable and harder to understand.*

Solution ?:

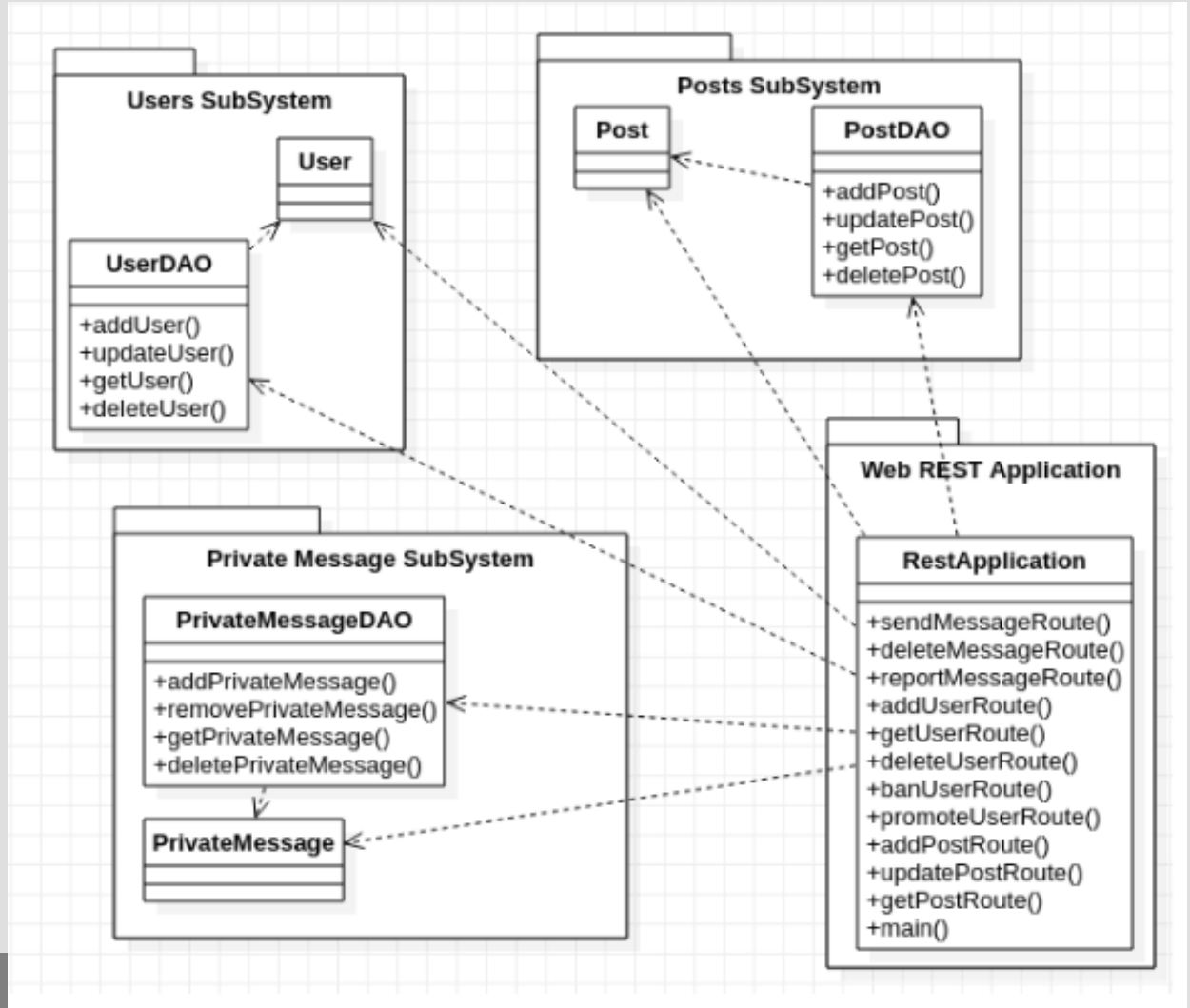
Separated concerns into three different classes:

- ***class BookManager***
- ***class FineCalculator.***
- ***class MemberInfoDisplay.***

```
class BookManager:
    def add_book(book):
        # Code to add a book to the library

    def remove_book(book_id):
        # Code to remove a book from the library

    def search_book_by_title(title):
        # Code to search for a book by title

class FineCalculator:
    def calculate_fine(member_id, days_overdue):
        # Code to calculate the fine for a member
based on overdue days

class MemberInfoDisplay:
    def display_member_info(member_id):
        # Code to display information about a
library member
```

EXAMPLE OF HIGH COHESION AND LOW COUPLING

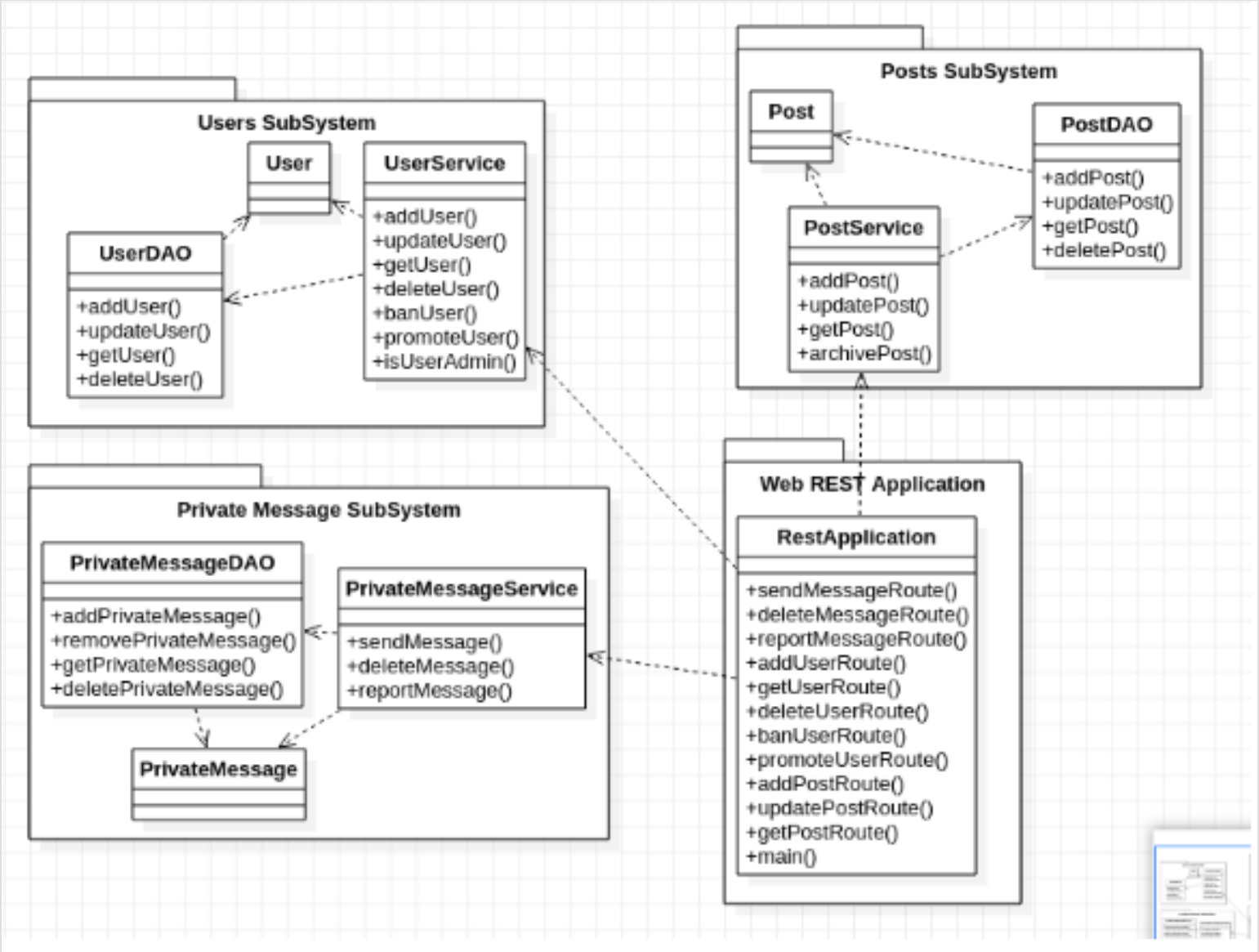
- A REST API that have to manage Users, Posts and Private Message between users



EXAMPLE OF HIGH COHESION AND LOW COUPLING

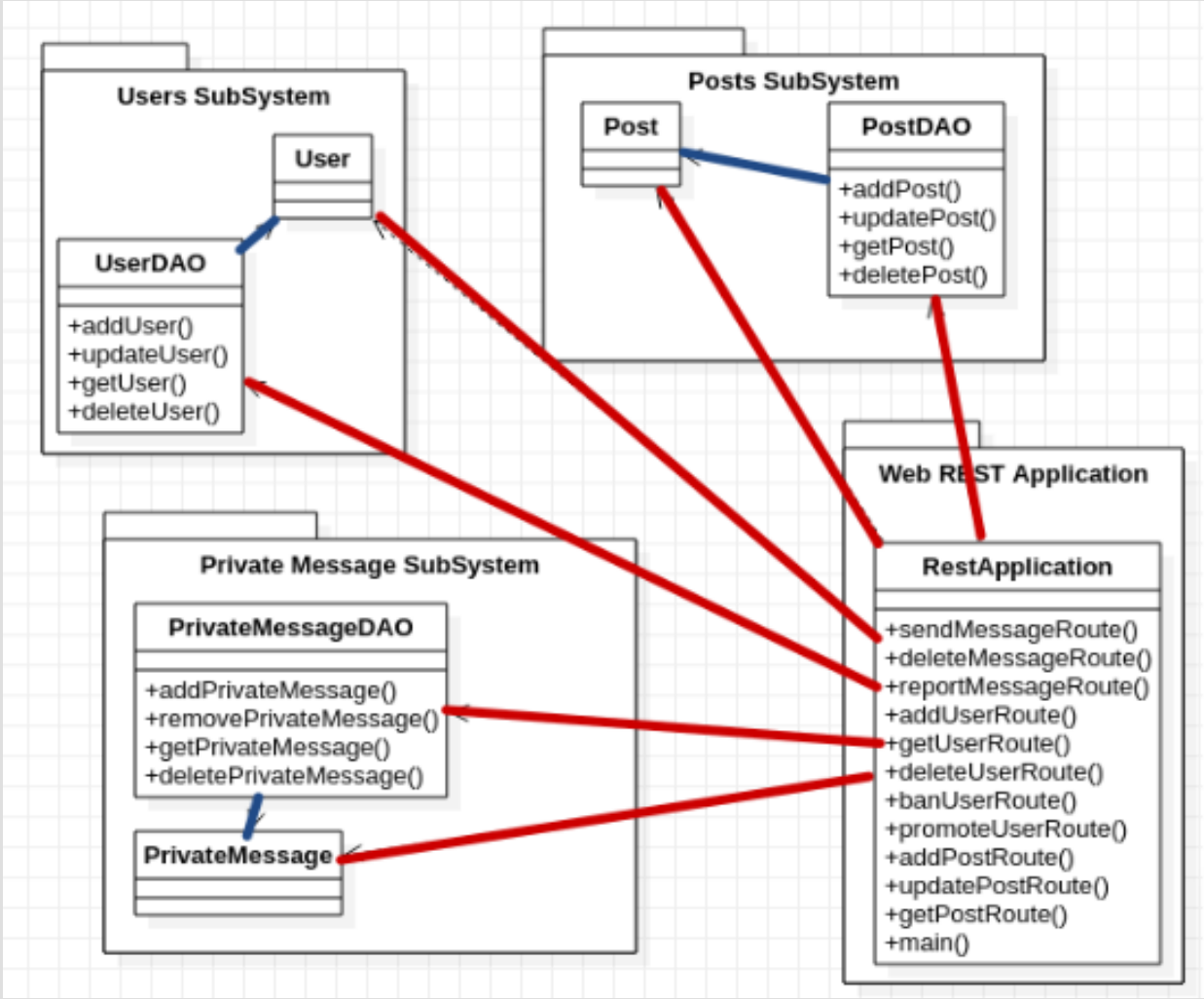
- the RestApplication class is managing the REST API requests correctly,
- but the issue here is that it also depends on all the other classes and it is managing the logic behind every single action (for example, the logic behind banning an user, reporting a message, sending a message, etc...).
- This is a clear example of **low cohesion and high coupling** because it depends heavily on internal classes of other modules.
- One bad thing about this is: what if some of the internal classes of the other modules changes? Then you will have to change the Application class to make this work

EXAMPLE OF HIGH COHESION AND LOW COUPLING



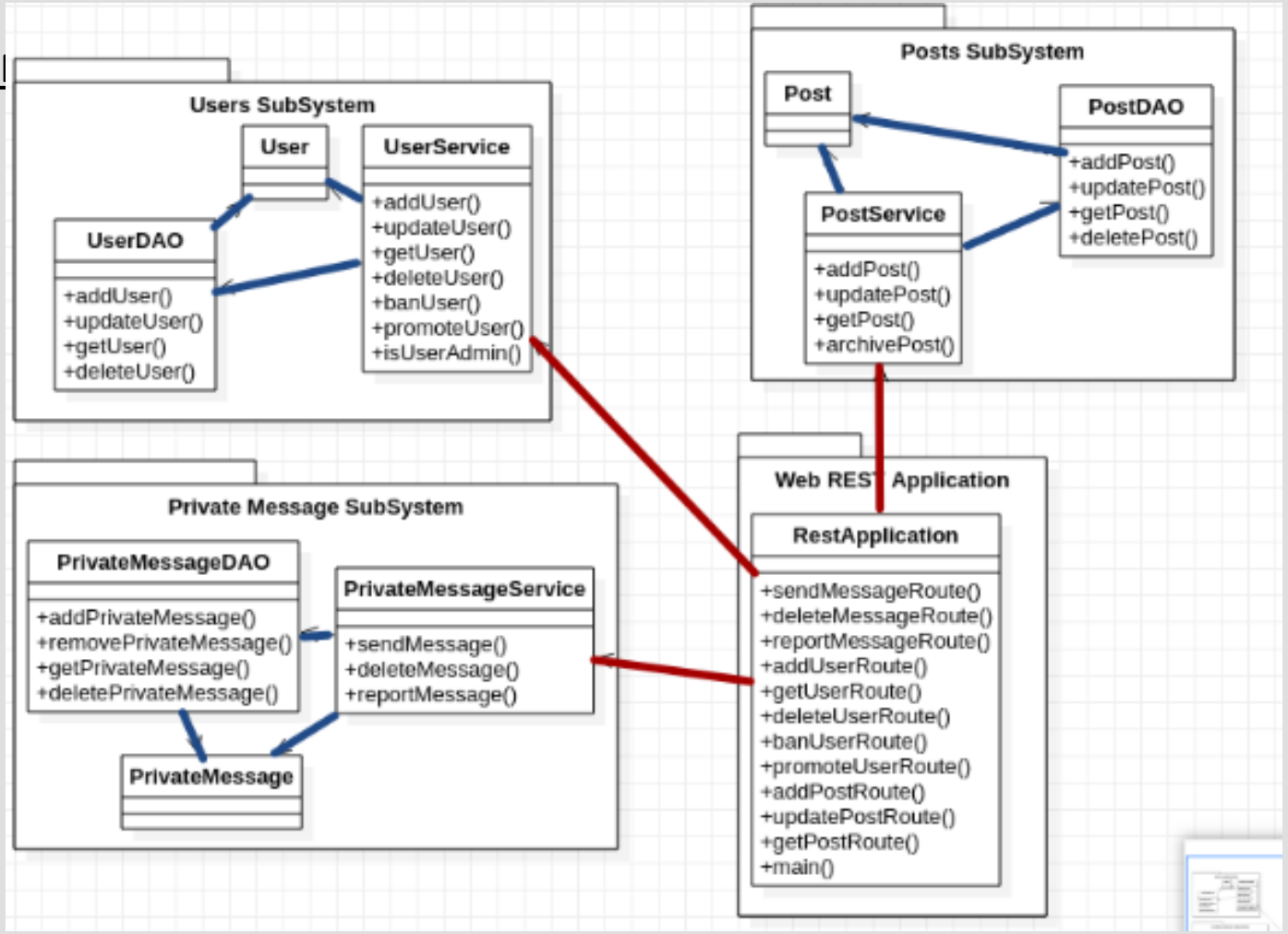
EXAMPLE OF HIGH COHESION AND LOW COUPLING

Low Cohesion (BLUE), High Coupling (RED) - BAD



EXAMPLE OF HIGH COHESION AND LOW COUPLING

High Cohesion (BLUE), Low Coupling (RED) - BETTE



THANK YOU

Bayram Zarty

Briram.zarty@gmail.com

APPENDIXES

REFERENCES

رقم المصدر	سنة النشر	المؤلف	عنوان المحتوى
1	2025	Hironori Washizaki	Guide to the Software Engineering Body of Knowledge v4.0
2	2023	Marwa Solla	Software architecture and design for modern large-scale systems
3	2024	akkah.com بكه للتعليم	الفرق بين المتطلبات الوظيفية وغير الوظيفية وتقنيات استنباطهما وأفضل الممارسات
4	2006	Christopher John Fox	Introduction to Software Engineering Design Processes, Principles, and Patterns with UML2
5	2025	Wikipedia	ISO/IEC 12207 - Wikipedia

ASSIGNMENT 2

- The previous questions helps in defining the scope of that project. Answers helps in having a clear consensus about if this production project could be conducted or not.
- If there some technical, legal, expertise- related, political, or other could arise, and couldn't be mitigated, then the production project is not feasible.
- If the dedicated budget was not reasonable at all then, Boeing will notify Libyan Airline that this plane couldn't be produced and delivered to Libya.
- In contrary, if the project shows feasibility then, the rest of information will help in defining the scope of the project.