Low Coupling

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.

An element with low (or weak) coupling is not dependent on too many other elements.

Types of coupling

Low coupling or weak coupling	High coupling or strong coupling
An element if does not depend on too many other elements like classes, subsystems and systems it is having low coupling.	A class with high coupling relies on many other classes. The Problem of high coupling are - Forced local changes because of changes in related classes. - Harder to understand in isolation.

Problem:

How to support low dependency, low change impact, and increased reuse?

Solution :

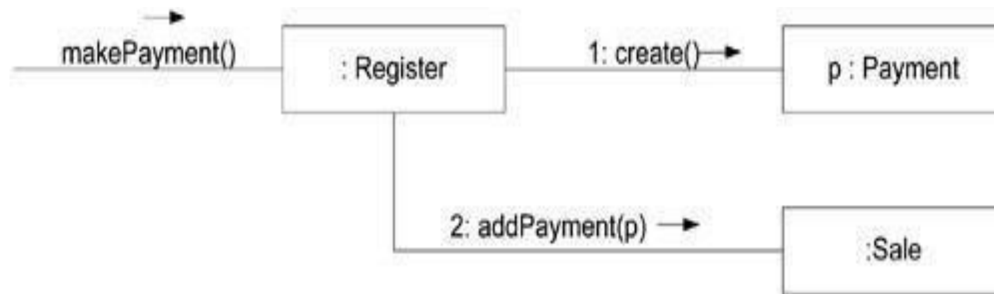
Assign a responsibility so that coupling remains low. Use this principle to evaluate alternatives.

Example:-

NextGen case Study

We have to create payment instance and associate it with sale.

DESIGN 1: Suggested by creator

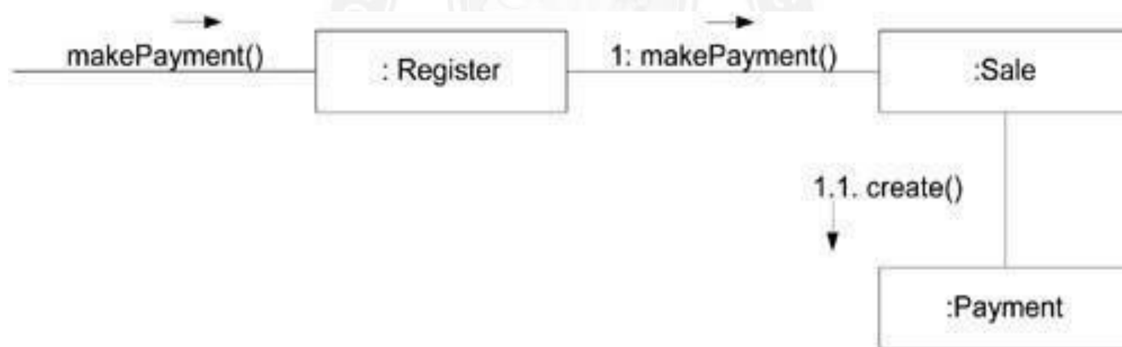


Register creates Payment

1) The Register creates the Payment and
2) It adds coupling of Register to Payment.

DESIGN 2: Suggested by low coupling

1) The Sale does the creation of a Payment and
2) It does not increase the coupling.



Sales creates Payment

Low coupling is an evaluation principle for evaluating all designs decisions. In object- oriented languages such as C++, Java, and C#, common forms of coupling from TypeX to TypeY include the following:

- TypeX has an attribute (data member or instance variable) that refers to a TypeY instance, or TypeY itself.
- A TypeX object calls on services of a TypeY object.

- TypeX has a method that references an instance of TypeY, or TypeY itself, by any means. These typically include a parameter or local variable of type TypeY, or the object returned from a message being an instance of TypeY.
- TypeX is a direct or indirect subclass of TypeY.
- TypeY is an interface, and TypeX implements that interface.

Contradictions

High coupling to stable elements and to pervasive elements is a problem. For example, a J2EE application can safely couple itself to the Java libraries

Benefits

- Not affected by changes in other components
- Simple to understand in isolation
- Convenient to reuse

HIGH COHESION

?Cohesion (or more specifically, functional cohesion) is a measure of how strongly related and focused the responsibilities of an element are. An element with highly related responsibilities that does not do a tremendous amount of work has high cohesion. These elements include classes, subsystems, and so on.

Solution

Assign a responsibility so that cohesion remains high. Use this to evaluate alternatives.

A class with low cohesion does many unrelated things or does too much work. Such classes are undesirable; they suffer from the following problems:

- hard to comprehend
- hard to reuse
- hard to maintain
- delicate; constantly affected by change

Low cohesion classes often represent a very "large grain" of abstraction or have taken on responsibilities that should have been delegated to other objects.

Example

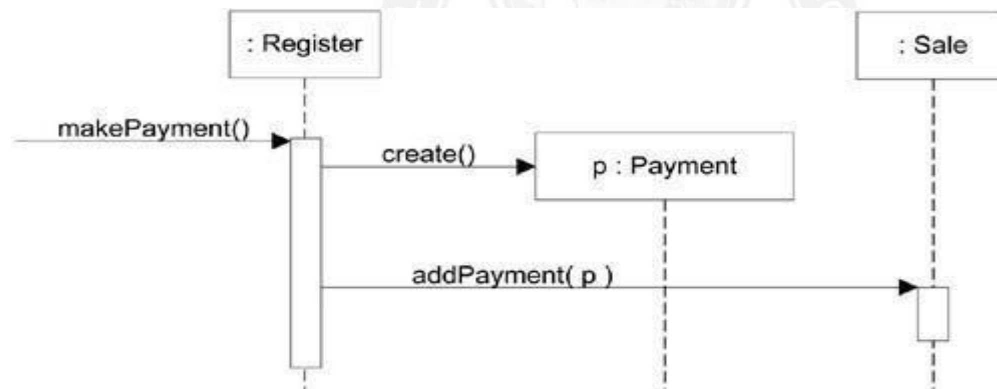
Create a payment instance and associate it with sale

DESIGN 1

- Register records a Payment in the real-world domain, the Creator pattern suggests

Register as a candidate for creating the Payment.

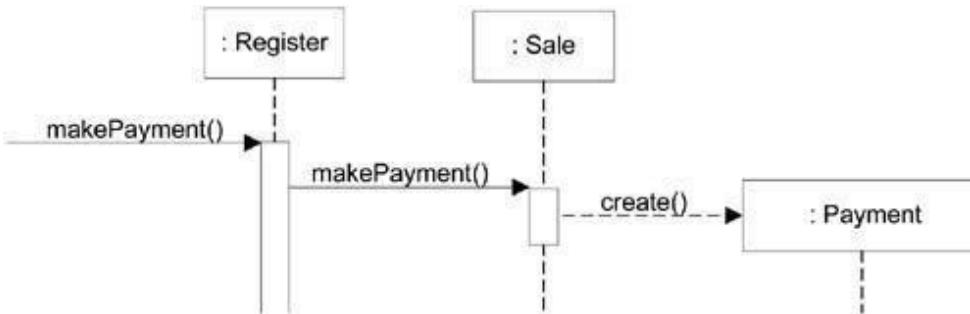
• The Register instance could then send an addPayment message to the Sale, passing along the new Payment as a parameter.



Register creates Payment

DESIGN 2

The second design delegates the payment creation responsibility to the Sale supports higher cohesion in the Register.



Sale creates Payment

In the second design, payment creation in the responsibility of sale.

It is highly desirable because it supports High Cohesion & Low Coupling

Scenarios of varying degrees of functional cohesion

1. Very low cohesion: A class is solely responsible for many things in very different functional areas.

   Ex : Assume the existence of a class called RDB-RPC-Interface which is completely responsible for interacting with relational databases and for handling remote procedure calls. These are two vastly different functional areas, and each requires lots of supporting code.

2. Low cohesion: A class has sole responsibility for a complex task in one functional area.

   Ex: Assume the existence of a class called RDBInterface which is completely responsible for interacting with relational databases. The methods of the class are all related, but there are lots of them, and a tremendous amount of supporting code; there may be hundreds or thousands of methods.

3. High cohesion: A class has moderate responsibilities in one functional area and collaborates with other classes to fulfill tasks.

Ex: Assume the existence of a class called RDBInterface that is only partially responsible for interacting with relational databases. It interacts with a dozen other classes related to RDB access in order to retrieve and save objects.

4. Moderate cohesion: A class has lightweight and sole responsibilities in a few different areas that are logically related to the class concept but not to each other.

Ex: Assume the existence of a class called Company that is completely responsible for (a) knowing its employees and (b) knowing its financial information. These two areas are not strongly related to each other, although both are logically related to the concept of a company.

Rule of thumb

A class with high cohesion has a relatively small number of methods, with highly related functionality, and does not do too much work. It collaborates with other objects to share the effort if the task is large.

Easy to maintain

- Understand and
- Reuse

Modular Design

Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules. Modular design creates methods and classes with single purpose, clarity and high cohesion.

Lower cohesion is had in

- Grouping responsibilities or code into one class or component.
- Distributed server objects.

Benefits

- Clarity and ease of comprehension of the design is increased.
- Maintenance and enhancements are simplified.
- Low coupling is often supported.
- Reuse of fine-grained, highly related functionality is increased because a cohesive class can be used for a very specific purpose