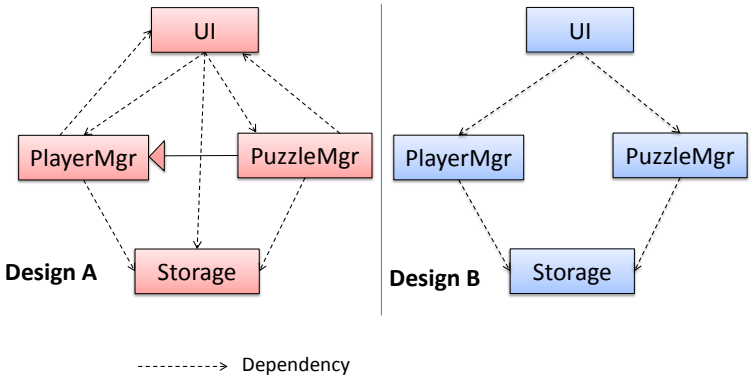
[L6P1]

To Tighten or Loosen: What Makes a Good Design

Coupling

Coupling is a measure of the degree of *dependence* between components, classes, methods, etc. Low coupling indicates that a component is less dependent on other components. In the case of high-coupling (i.e. relatively high dependency), a change in one component may require changes in other coupled components. Therefore, one should strive to achieve a low-coupled design.

In the example below, design A appears to have a higher coupling than design B.



To illustrate some examples of coupling, component A is coupled to B if,

- *component A* has access to the internal structure of *component B* (this results in a very high level of coupling);
- *component A* and *B* depend on the same global variable;
- *component A* calls *component B*;
- *component A* receives an object of c*omponent B* as a parameter or a return value;
- *component A* inherits from *component B*;
- *components A* and *B* are required to follow the same data format or communication protocol.

Highly coupled (also referred to as *tightly* coupled or *strongly* coupled) systems display the following disadvantages:

- A change in one module usually forces changes in other modules coupled to it (i.e. a ripple effect).
- Integration is harder because multiple components coupled with each other have to be integrated at the same time.
- Testing and reuse of the module is harder due to its dependence on other modules.

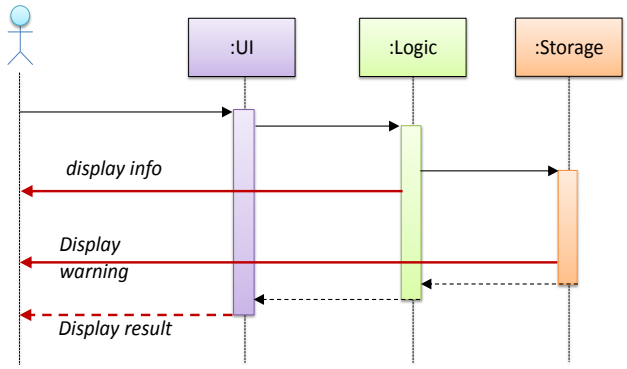
Cohesion

Cohesion is a measure of how strongly-related and focused the various responsibilities of a component are. A highly-cohesive component keeps related functionalities together while keeping out all other unrelated things. One should strive for high cohesion to facilitate code maintenance and reuse.

Cohesion can be present in many forms. For example,

- Code related to a single concept is kept together, e.g. the Student component handles everything related to students.
- Code that is invoked close together in time is kept together, e.g. all code related to initializing the system is kept together.
- Code that manipulates the same data structure is kept together, e.g. the GameArchive component handles everything related to the storage and retrieval of game sessions.

The components in the following sequence diagram show low cohesion because user interactions are handled by many components. Its cohesion can be improved by moving all user interactions to the UI component.



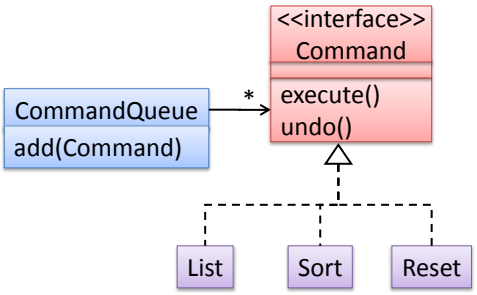
The following are some disadvantages of low cohesion (or "weak cohesion").

- Impedes the understandability of modules as it is difficult to express module functionalities at a higher level.
- Difficulty in maintaining modules because a localized adjustment in the requirements can result in changes spread across the system since requirement-related functionality is implemented across many components of the system.
- Modules become less reusable because they do not represent logical units of functionality.

Open-Closed Principle

While it is possible to isolate the functionalities of a software system into modules, there is no way to remove interaction between modules. When modules interact with each other, coupling naturally increases. Consequently, it is harder to localize any changes to the software system. In 1988, Bertrand Meyer proposed a guiding principle to alleviate this problem. The principle, known as the *open-closed principle*, states: "A module should be *open* for extension but *closed* for modification". That is, modules should be written so that they can be extended, without requiring them to be modified. In other words, changing what the modules do without changing the source code of the modules.

In object-oriented programming, these two seemingly opposing requirements can be achieved in various ways. This often requires separating the *specification (interface)* of a module from its *implementation*. For example, consider this following design.



The behavior of the CommandQueue class can be altered by adding more concrete Command subclasses. For example, by including a Delete class alongside List, Sort, and Reset, the CommandQueue can now perform delete commands without modifying its code at all. Indeed, its behavior was extended without having to open up and modify its code. Hence, it was open to extensions, but closed to modification.

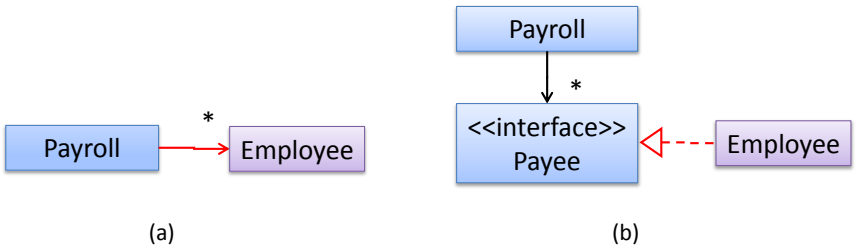
Another example of putting this principle into action is Java Generics (similar to C++ templates). The behavior of a template/generic class can be altered by passing it a different class as a parameter. In the code below, the ArrayList class behaves as a container of Students in one instance and as a container of Admin objects in the other instance, without having to change its code. That is, the behavior of the ArrayList class is extended without modifying its code.

```
ArrayList students = new ArrayList<Student>();
ArrayList admins = new ArrayList<Admin>();
```

Dependency Inversion Principle (DIP)

The *Dependency Inversion Principle* states that high-level modules should not depend on low-level modules. Both should depend on abstractions.

Consider the example below. In design (a), the higher level class Payroll depends on the lower level class Employee, a violation of DIP. In design (b), both Payroll and Employee depends on the Payee interface (not that inheritance is a dependency). Design (b) is more flexible (and less coupled) because now the Payroll class need not change when the Employee class changes.

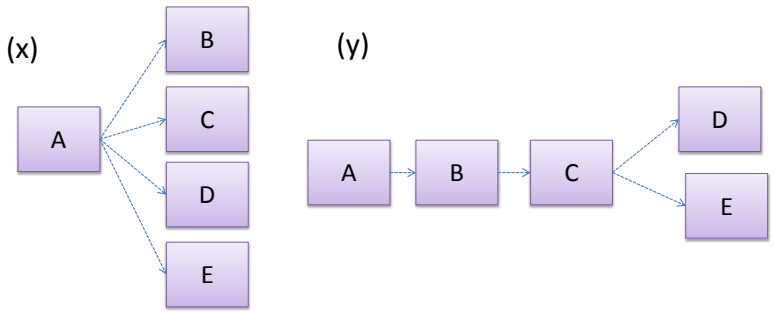


Worked examples

[Q1]
Explain the link (if any) between regression and coupling.

[A1]
When the system is highly-coupled, the risk of regression is higher too. When component A is modified, all components 'coupled' to component A risk 'unintended behavioral changes'.

[Q2]
Discuss the coupling levels of alternative designs x and y.



[A2]
Overall coupling levels in x and y seem to be similar (neither has more dependencies than the other). (Note that the number of dependency links is not a definitive measure of the level of coupling. Some links may be stronger than the others.). However, in *x*, *A* is highly-coupled to the rest of the system while *B, C, D*, and *E* are standalone (do not depend on anything else). In *y*, no component is as highly-coupled as *A* of *x*. However, only *D* and *E* are standalone.

[Q3]
Discuss the relationship between coupling and testability (testability is a measure of how easily a given component can be tested).

[A3]
Coupling decreases testability as it is difficult to isolate highly-coupled objects.

[Q4]
Compare the cohesion of the following two versions of the EmailMessage class. Which one is more cohesive and why?

```
// version-1
class EmailMessage {
	private String sendTo;
	private String subject;
	private String message;

	public EmailMessage(String sendTo, String subject, String message) {
		this.sendTo = sendTo;
		this.subject = subject;
		this.message = message;
	}

	public void sendMessage() {
		// sends message using sendTo, subject and message
	}
}

// version-2
class EmailMessage {
	private String sendTo;
	private String subject;
	private String message;
	private String username;

	public EmailMessage(String sendTo, String subject, String message) {
		this.sendTo = sendTo;
		this.subject = subject;
		this.message = message;
	}

	public void sendMessage() {
		// sends message using sendTo, subject and message
	}

	public void login(String username, String password) {
		this.username = username;
		// code to login
	}
}
```

[A4]
Version 2 is less cohesive because it is handling functionality related to login, which is not directly related to the concept of 'email message' that the class is supposed to represent. On a related note, we can improve the cohesion of both versions by removing the sendMessage functionality. Although sending message is related to emails, this class is supposed to represent an email message, not an email server.