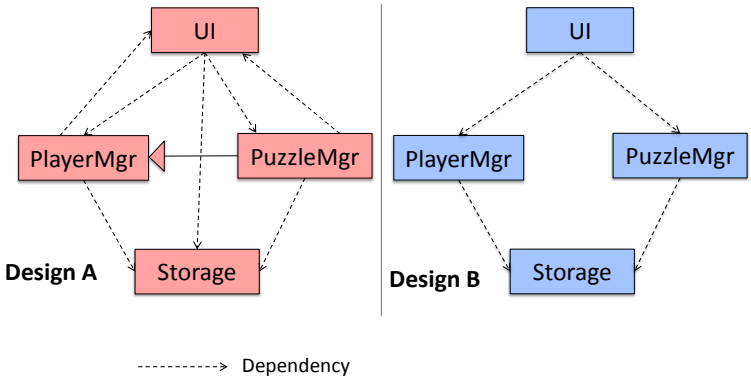
[Handout for L5P1]
To Tighten or Loosen: What Makes a Good Component

Coupling and Cohesion

Coupling is the degree to which components depend on each other. Low coupling means a component depends less on other components. In the case of high-coupling (i.e. relatively high dependency), a change in one component would require changes in other coupled components. Therefore, we should strive to achieve low coupling in our design.

For example, design A appears to have higher coupling than design B.



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

- *component A* has access to 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* have 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.
- 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 because it is dependent on other modules.

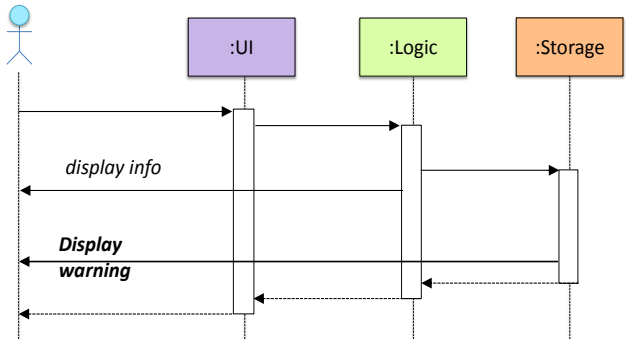
Cohesion is a measure of how strongly-related and focused are the various responsibilities of a component. A highly-cohesive component keeps together things that are related to each other while keeping out all other things. We should strive for high cohesion because having related things together and not mixing unrelated things make the code easier to maintain and reuse.

Cohesion can be present in many forms. For example,

- Code related to a single concept is kept together. e.g. 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. GameArchive component handles everything related to archiving and retrieving of game sessions.

The components in the following illustration shows low cohesion because user interactions are handled by many components. We can improve its cohesion by moving all user interactions to the UI component.



Disadvantages of low cohesion (or "weak cohesion") are:

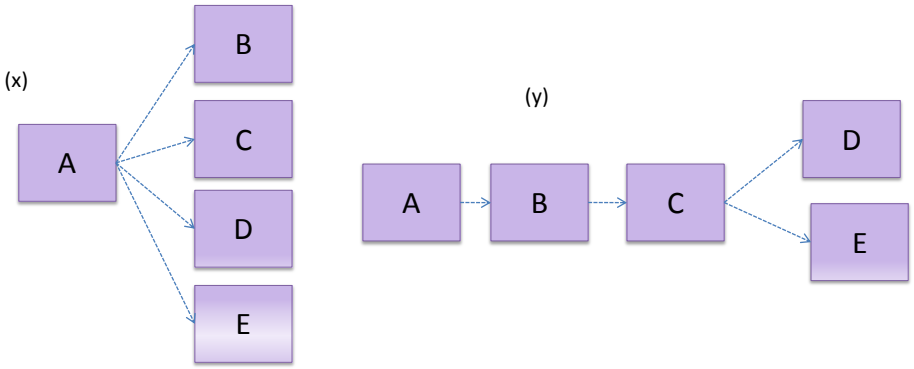
- More difficult to understand modules because it is difficult to express at a higher level which module has what functionality.
- Maintenance is more difficult because a localized change in the requirements can result in changes spread across the system because functionality related to that requirement can be implemented in many components of the system.
- Modules are less reusable because they do not represent logical units of functionality.

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 A risk 'unintended behavioral changes'.

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



(b) Discuss the relationship between coupling and testability (testability is a measure of how easy we can test a given component).

(c) 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 = to;
                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 = to;
                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
        }
}
```

[A2]

(a) 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.

When following design *x*, four developers can build *B, C, D, E* in parallel while frequently integrating with *A*. Working is parallel is harder if we follow design *y*.

However, it is difficult to state a definitive choice without knowing more about the context.

Also, note that dependency is not transitive, i.e. *A* does not depend on *C* although *A* depends on *B* and *B* depends on *C*.

b) Coupling decreases testability as it is difficult to isolate highly-coupled objects.

c) 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.

---End of Document---