Object-Oriented Design I

Single responsibility
High cohesion
Information expert
Low coupling
Law of Demeter
Dependency inversion
Up to this point, you have studied object-oriented design mostly at the class level.

- This set of skills needs to be expanded to design larger scale systems.
- You need to consider the interactions between classes and the effect of classes on other classes.
- The software engineering community has put forward sets of design principles to follow.
  - **SOLID** *(Bob Martin, Principles of OOD)*
  - **GRASP** *(Craig Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.)*

- We will look at some of these principles, along with the Law of Demeter, in two lessons.
SOLID and GRASP provide two sets of object-oriented design principles.

**SOLID**
- Single responsibility
- Open/closed
- Liskov substitution
- Interface segregation
- Dependency inversion

**GRASP**
- Controller
- Creator
- Indirection
- Information expert
- High cohesion
- Low coupling
- Polymorphism
- Protected variations
- Pure fabrication

Law of Demeter
The **Single responsibility** principle is perhaps the most important object-oriented design principle.

A class should have only a single responsibility.
The Single responsibility principle will lead to smaller classes each with less responsibility.

A class should have only a single responsibility.

- A class should have a single, tightly focused responsibility.
- This leads to smaller and simpler classes, but more of them.
  - Easier to understand the scope of a change in a class.
  - Easier to manage concurrent modifications.
  - Separate concerns go into separate classes.
- Helps with unit testing.
High Cohesion aims for focused, understandable, and manageable classes.

Assign responsibility so the cohesion of classes remains high.

- High cohesion leads to smaller classes with more narrowly defined responsibilities.
- This design goal should have a higher priority than most other goals.
High Cohesion vs. Low Coupling

- High cohesion will require coupling to more classes to get work accomplished.
  - Do not be afraid of requiring a few more relationships in the pursuit of improved cohesion.
Consider that you are implementing a library management system.

- You could place most of the functionality into a `LibraryManager` class.
- This class would have too many responsibilities.
  - *Maintaining the library catalog*
  - *Maintaining patron accounts*
  - *Scheduling library events*
Separate the concerns into more classes each with a single highly focused responsibility.
Make a class as simple as possible, but not simpler.

*Everything should be made as simple as possible, but not simpler.* - Einstein

- Aim to implement the behaviors that directly work with the class' attributes.
- Consider what clients will want to do with the attribute data—put those behaviors in the class.
- If you are a client doing processing with the attribute data, consider putting your operation in the class.
Some classes are more complex than you think they are.

- Consider that you are building an airline flight reservation system.
- A **Flight** entity will definitely be in the domain model and be a class in your implementation.
- You could consider this to be only a data holder class with no other behavior.

- This would lead to something like

```
Flight
- departure : unsigned int
- arrival : unsigned int
- destination : String
- origin : String
+ getDeparture() : unsigned int
+ getArrival() : unsigned int
+ getDestination() : String
+ getOrigin() : String
```
Student project code often does not do right by the client of their classes.

A client of **Flight** had this code:

- `flt.getDestination().equals("JFK")`
- `flt1.getOrigin().equals("ROC") && flt1.getDestination().equals("JFK")`
- `flt.getArrival() < flt.getDeparture()`
- `flt1.getArrival() + 60 < flt2.getDeparture()`

Why does the client of **Flight** have to do this "heavy-lifting"?

Note: Time is stored in 24 hour notation.
**Information Expert** looks to have behavior follow data.

Assign responsibility to the class that has the information needed to fulfill the responsibility.

- The first place to consider placing code that uses/processes attribute data is in the class that holds the attributes.
- Instead of the client of `Flight` implementing this:
  - `flt.getDestination().equals("JFK")`
  - `flt1.getOrigin().equals("ROC") && flt1.getDestination().equals("JFK")`
  - `flt.getArrival() < flt.getDeparture()`
  - `flt1.getArrival() + 60 < flt2.getDeparture()`
- Consider `Flight` as the Information expert
  - `boolean Flight.destinationIs(String airportCode)`
  - `boolean Flight.itineraryIs(String originCode, String destinationCode)`
  - `boolean Flight.arrivesNextDay()`
  - `boolean Flight.canConnectWith(Flight nextFlight)`
Low Coupling attempts to minimize the impact of changes in the system.

Assign responsibility so that (unnecessary) coupling remains low.

- Note the unnecessary word. Coupling is needed in your system.
- Resist lowering coupling simply to reduce the number of relationships.
  - A design with more relationships is often better than the design with fewer relationships.
  - You need to balance all the design principles.
  - Beginning designers often place low coupling at the top of their design principles list. It should be lower down!
The Law of Demeter addresses unintended coupling within a software system.

- Limit the range of classes that a class talks to
  - *Each unit only talks to its friends; don't talk to strangers.*
  - *Each unit only talks to its immediate friends; don't talk to friends of friends*
  - *Chained access exposes each intermediate interface*

- From a previous checkers project
  - `board.getPieceAt(i, j).getType()` ➔ *eliminate violation of Law of Demeter and reduce coupling for this class, i.e. may not need to know about Piece*
  - `board.getPieceTypeAt(i, j)` or *predicate*

- If you need to talk to something "far away"
  - Get support from your friend, i.e. new method
  - Get a new friend, i.e. new direct relationship
The *Dependency inversion* principle provides looser coupling between dependent entities.

High-level modules should not depend on low-level modules. Both should depend on abstractions.

- A common manifestation of this is *dependency injection*.
  - The low-level module does not have responsibility for instantiating a specific dependent class.
  - The high-level module *injects* the dependent element.
  - The low-level module is only dependent on the (high-level) abstraction not the (low-level) implementation.
  - The injection can be during construction, using a setter, or as a parameter in an operation method.
  - **Critical** for doing unit testing since we can inject test/mock objects.
Here is how an application's design might evolve to incorporate dependency injection.

- Higher level has responsibility for instantiation.
- Design does not adequately separate the concerns of catalog operations vs database operations.
- It will be difficult to unit test just catalog operations.

- Lower level has responsibility for instantiation.
- It will be difficult to unit test the Catalog class.

- Higher level has responsibility for instantiation of the specific database implementation.
- It injects this Database dependency into the Catalog when it is instantiated.
- Catalog only deals with higher level Database abstraction.
- Use test version of Database to test Catalog.