

Mutable vs Immutable

- classes shouldn't be mutable unless they have to be

Mutable vs Immutable

Use a mutable class only when object identity is important.

- “equivalent” means “same address”
 - two Players are different even if they have the same values for properties (name, score, etc)
 - use `==` to compare
- a Player retains a certain identity even when values of properties change (e.g. score)

Use a non-mutable class when object identity is not important.

- “equivalent” means “same values”
 - two Positions are the same if they have the same row and column
 - two Fractions are the same if they have the same value
- ★ implement and use `equals()` to compare
- Position is just a bundling of row, column into one unit for convenience
 - changing the value of the row or column changes the position
- Fraction is just a bundling of numerator and denominator
 - changing the value of one part changes the value of the fraction

Mutable Objects

```
public class Date {
    private int day_, month_, year_;

    public Date ( int day, int month, int year ) { ... }

    public int getDay () { ... }
    public int getMonth () { ... }
    public int getYear () { ... }

    public void setDate ( int day, int month, int year ) { ... }
}
```

- object identity is not important for dates
 - no reason to have `setDate`

Mutable Objects

```
public class Hand {
    private List<Card> cards_;

    public Hand ( List<Card> cards ) { cards_ = cards; }

    public void addCard ( Card card ) { ... }
    public Card removeCard ( int position ) { ... }
    public Card getCard ( int position ) { ... }
    public List<Card> getCards () { return cards_; }

    public int getNumCards () { ... }
    public int getPosition ( Card card ) { ... }
}
```

- object identity is important for hands
 - OK to have `addCard`, `removeCard`
- maintaining encapsulation
 - `getCards()`, passing a list of cards to the constructor breaks encapsulation
 - to fix, eliminate `getCards()` and have the constructor create an empty hand

Mutable Objects

Instead of passing or returning a mutable object –

- provide methods that do what you want
 - e.g. put `print()` into class containing collection instead of having a getter for the collection so the caller can iterate in order to print
- provide methods that allow the abstraction you need
 - e.g. instead of returning the list used to store a collection so you can iterate through, support iteration with `getNum()` and `get(i)` methods, where *i* is an index (0,1,2,...)
- provide access only through a non-mutable wrapper
 - e.g. an iterator

If you still need to pass or return mutable object, can you make a copy?

Loose Coupling

- components do not rely on the definitions of other components
 - a class should only have to change because something related to its internals changed, not because some other class changed
- advantages of loose coupling
 - reduces the ripple effect – limits the scope of changes
 - reduces system complexity – components are more independent and can be understood in isolation
 - facilitates reuse – limits specific dependencies
- coupling vs cohesion
 - *coupling* refers to interdependencies between modules
 - *cohesion* addresses relationships between elements within a module
 - decrease coupling by putting related things together
 - but, separating less tightly-related things to increase cohesion can also result in more tightly-coupled components

Guitar Store



```
public class Guitar {
    private String serialNumber_;
    private double price;
    private Builder builder_;
    private String model;
    private GuitarType type_;
    private Wood backWood_;
    private Wood topWood_;

    public Guitar ( String serialNumber,
                    double price,
                    Builder builder,
                    String model,
                    GuitarType type,
                    Wood backWood,
                    Wood topWood ) { ... }

    ...
}
```

What happens if we want to add “number of strings” as a property of guitars?

Guitar Store



What happens if we want to add “number of strings” as a property of guitars?

- add property to Guitar
 - add instance variable and getter to Guitar
 - update Guitar constructor
- update where Guitar constructor is used
 - add parameter to `addGuitar (Inventory)`
 - modify initialization in `FindGuitarTester` to add the number of strings
- update where (all) Guitar properties are used
 - modify main program (`FindGuitarTester`) to include the number of strings in the search
 - include new property in `search (Inventory)`

Both `Inventory` and `FindGuitarTester` are affected when `Guitar` is changed → not loosely coupled!

Where Coupling Occurs

- A has a method with a parameter or return type B
- A has an instance variable of type B
- A calls methods on B – cost varies with what changes
 - e.g. adding a method to B does not affect existing code (only new code added to use the new method)
 - e.g. changing the name of a method affects the callers, but only a search-and-replace on the name
 - e.g. removing a method or modifying the parameter list requires rewriting code in the callers
- A is a subclass of or implements B

tighter
coupling

Recognizing Coupling Issues

- warning signs
 - a class has an instance variable, parameter, or return type of a concrete class where an interface (or abstract class) would do
 - a chain of method calls drilling through several objects – `a.getB().getC().getD()`
 - changing one class forces you to touch several others
 - two classes keep changing together for unrelated reasons
- code smells
 - message chains – `a.getB().getC().getD()`
 - shotgun surgery – changing one class forces you to touch several others
 - feature envy – a method more interested in another class's data than its own
 - middle man – a class does nothing but forward calls to another class
- swap test
 - “if I replaced this dependency with a different implementation offering the same behavior, how much would I need to change?”

Achieving Loose Coupling

- the goal is *appropriate* coupling, not *zero* coupling
 - some interdependency between classes is necessary if they are to collaborate at all
- limit collaborators
 - only store or pass information that is needed for the task
 - bundle methods with the data they manipulate, so the caller doesn't have to provide as much information
- prefer whole-task methods
 - have methods perform the whole task rather than only support individual steps of the task
- only communicate through parameters and return values
 - do not add instance variables to a class for the purpose of communication between methods
- watch for sequence dependency
 - e.g. dependence on a particular sequence – `a()` must be called before `b()` because `a()` sets values `b()` needs
 - initialization should be done in the constructor whenever possible

Code to the Interface

- depend on an abstraction, not the concrete implementation

Why?

- reduces coupling
 - A depends only the concept rather than the specific realization, so A is isolated from changes in the specific realization
 - A depends only on the interface (one thing) instead of specific concrete classes (many things), so only have to change one thing (which constructor is called) to use a different implementation
- increases flexibility and reuse
 - A can work with any implementation of the interface – including classes not written yet!

Guitar Store



Fixing the guitar store –

- of course whatever uses the Guitar constructor will be affected
 - can't remove FindGuitarTester's dependence on Guitar's constructor
- Inventory's job is to manage the collection of guitars – that should not require knowledge of internals of Guitar
 - code to the interface

Guitar Store – Coding to the Interface



Fixing the guitar store –

- Inventory's job is to manage the collection of guitars – that should not require knowledge of internals of Guitar
 - addGuitar
 - collection management is adding Guitar object to collection → pass Guitar object instead of individual attributes
 - search
 - collection management is going through the collection of guitars
 - determining if searchGuitar matches a particular guitar in the inventory depends on the specific attributes of a guitar → outsource to Guitar (matches method) because Guitar knows about those attributes

Law of Demeter

- only talk to friends, not strangers – a method should use only
 - instance fields of its class
 - parameters
 - objects it constructs with new
- (and, in particular, it should not ask another object to give it part of its internal state to work on)
- avoid chains of dots – especially where the middle methods are getters
 - `a.b()`, not `a.b().c().d()`
 - also watch for hidden sequences – methods called on variables storing something just returned

Law of Demeter

Why?

- asking A to give you B means you depend on both A and B → increases coupling
- and often means
 - revealing that A stores B → violates information hiding
 - granting access to A's internal state → violates encapsulation, especially if B is a mutable object

How to fix? Options –

- instead of asking A to give you a reference to B so you can manipulate it, ask A to do the manipulation for you
- have B passed to you directly instead of A

Law of Demeter

Disadvantages.

- can result in a lot of delegation, with wrapper methods just to propagate calls
- can expand class interfaces with lots of special-purpose and wrapper methods
 - new methods may not really fit with A's purpose
- can limit reusability if A's interface is not complete
 - can use reference to B to do anything instead of being limited to what A supports

As with everything, balance is key.