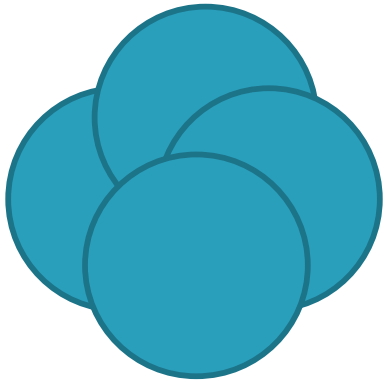
Refactoring Couplers

Andrejs Doronins
TEST AUTOMATION ENGINEER

Couplers

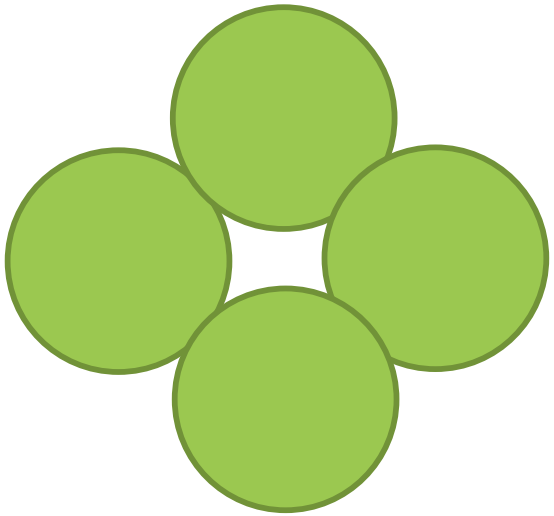
Code smells that result in tightly coupled classes.

Tight vs. Loose Coupling



Tight Coupling

Classes are so tied, that you cannot change one without changing the other.



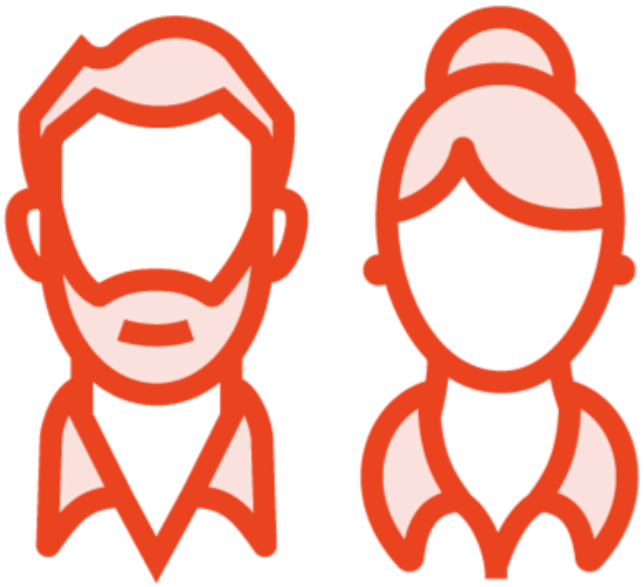
Loose Coupling

Change in one class requires no or minimum changes in other classes

To prevent coupling:

- Improve encapsulation
- Apply SRP principle

Couplers



Feature Envy

Inappropriate Intimacy

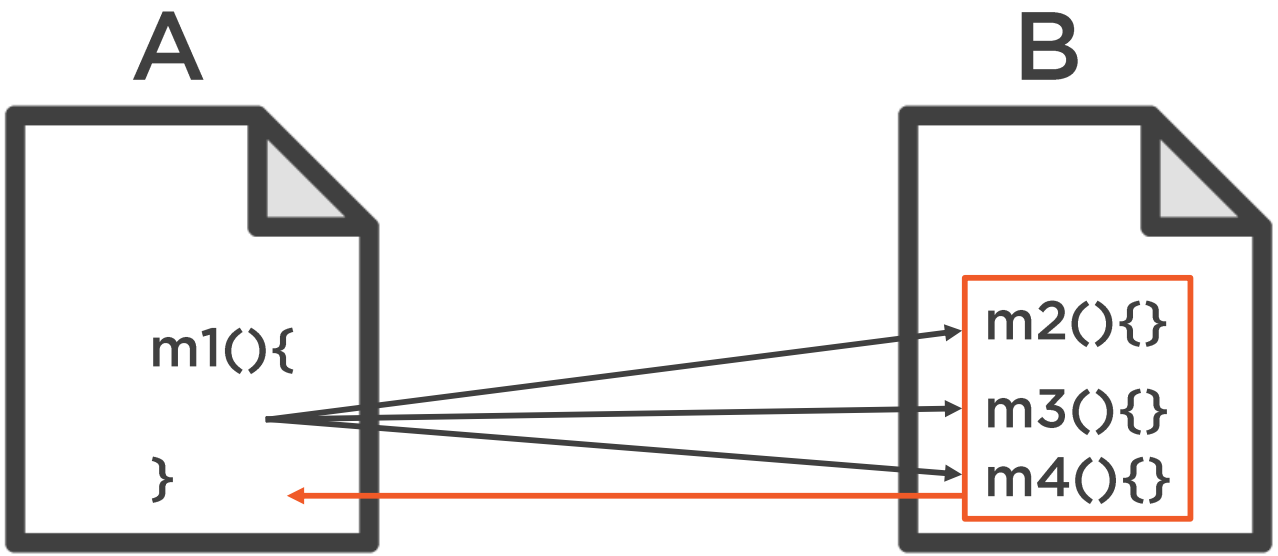
Excessive Exposure

Message Chains

Middle Man

Feature Envy

Class uses methods or accesses data of another class more than its own


A
B
m1(){
}
m2(){}
m3(){}
m4(){}

Feature Envy Issues

Hard to understand code that logically belongs elsewhere

May result in Shotgun Surgery

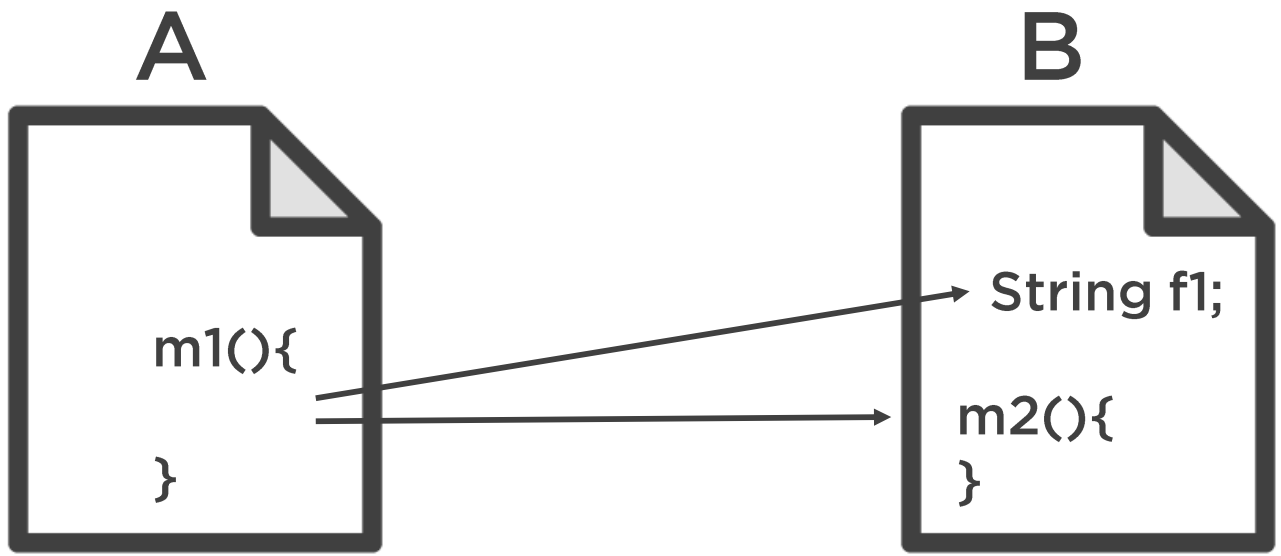
Benefits Achieved

Correctly encapsulated functionality

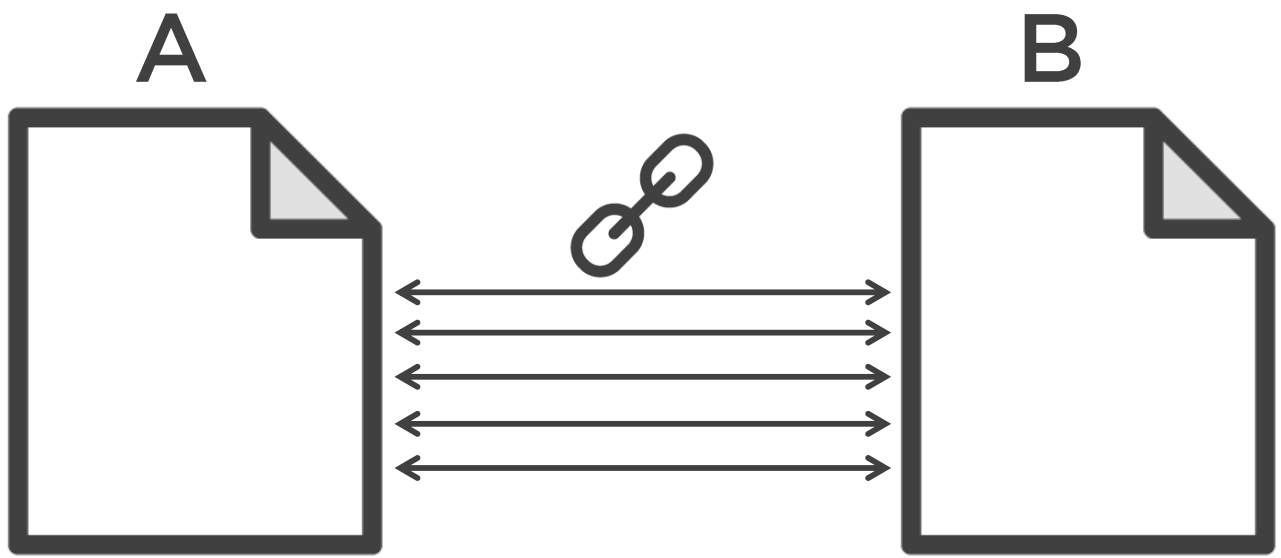
You find code where you expect it to be

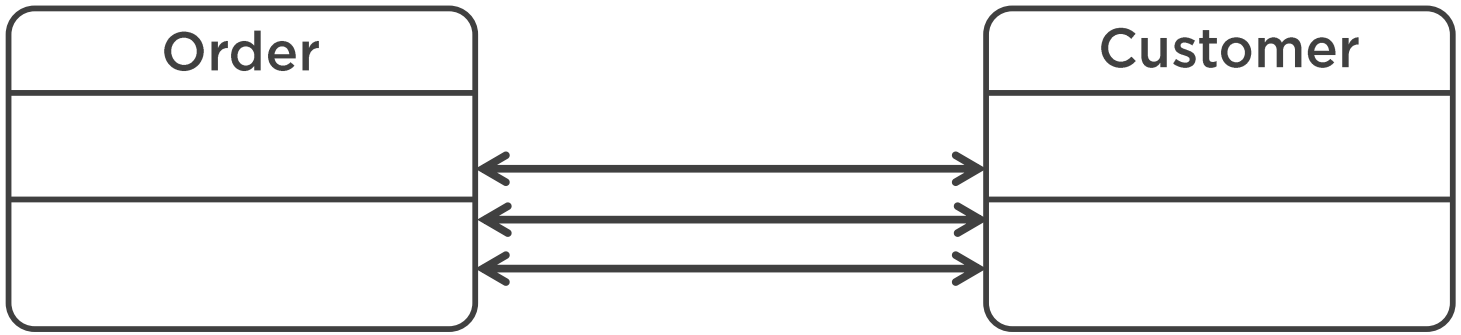
Inappropriate Intimacy

A class knows too much of (or has to deal with) internal details of another class


A
B
m1(){
}
String f1;
m2(){
}

Hide as much as possible (i.e. mark fields and methods private unless they are publicly needed)


A
B


Order
Customer

Excessive Exposure

Happens when a class or a module exposes too many internal details about itself

Inappropriate Intimacy
vs.
Excessive Exposure?

Inappropriate Intimacy

Usage of public fields instead of getters and setters

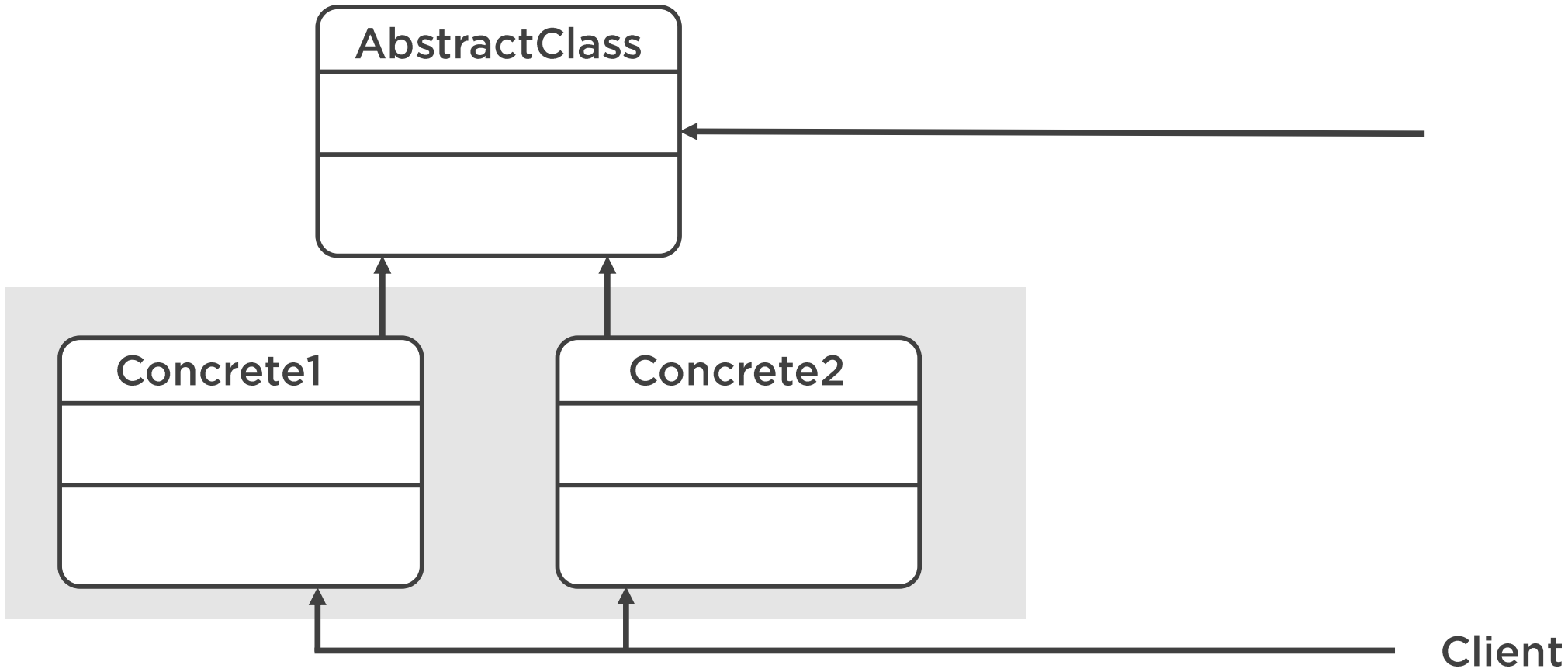
Too many links (e.g. method calls) between two classes

“You are with each other too much”

Excessive Exposure

A class forces you to know and care about its internal details and state

“You make me care too much”


AbstractClass
Concrete1
Concrete2
Client

Joshua Kerievsky

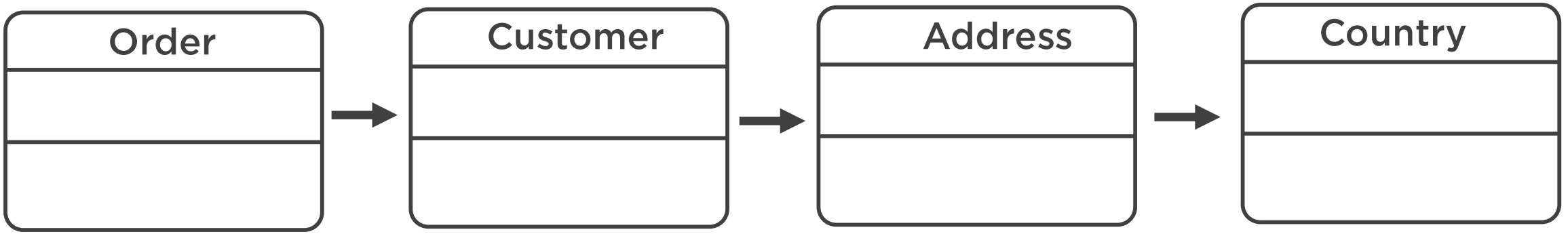
Book: Refactoring to Patterns

Message Chain

Code that calls (chains) multiple methods to get to the required data

Message Chain

```
customer.getAddress().getCountry().toString();
```


Order
Customer
Address
Country

Message Chain Issues

Must learn complex relationships between classes

Fragile

Before

```
customer.getAddress()
        .getCountry()
        .toString();
```

After

```
customer.getCountry();
```

Middle Man

A class that performs only one action, which is delegating work to other classes.

```
class SomeClass {
  OtherClass c;

void doThing(){
   c.doTheThing();
 }
void doAnotherThing(){
    // implementation
    }
}
```

```
class SomeClass {
  OtherClass c;

void doThing(){
   c.doTheThing();
 }
void doAnotherThing(){
    c.doThatOtherThing();
    }
}
```

Middle Man Issues

Has a cost, but brings no value

Middle Man Patterns

Facade

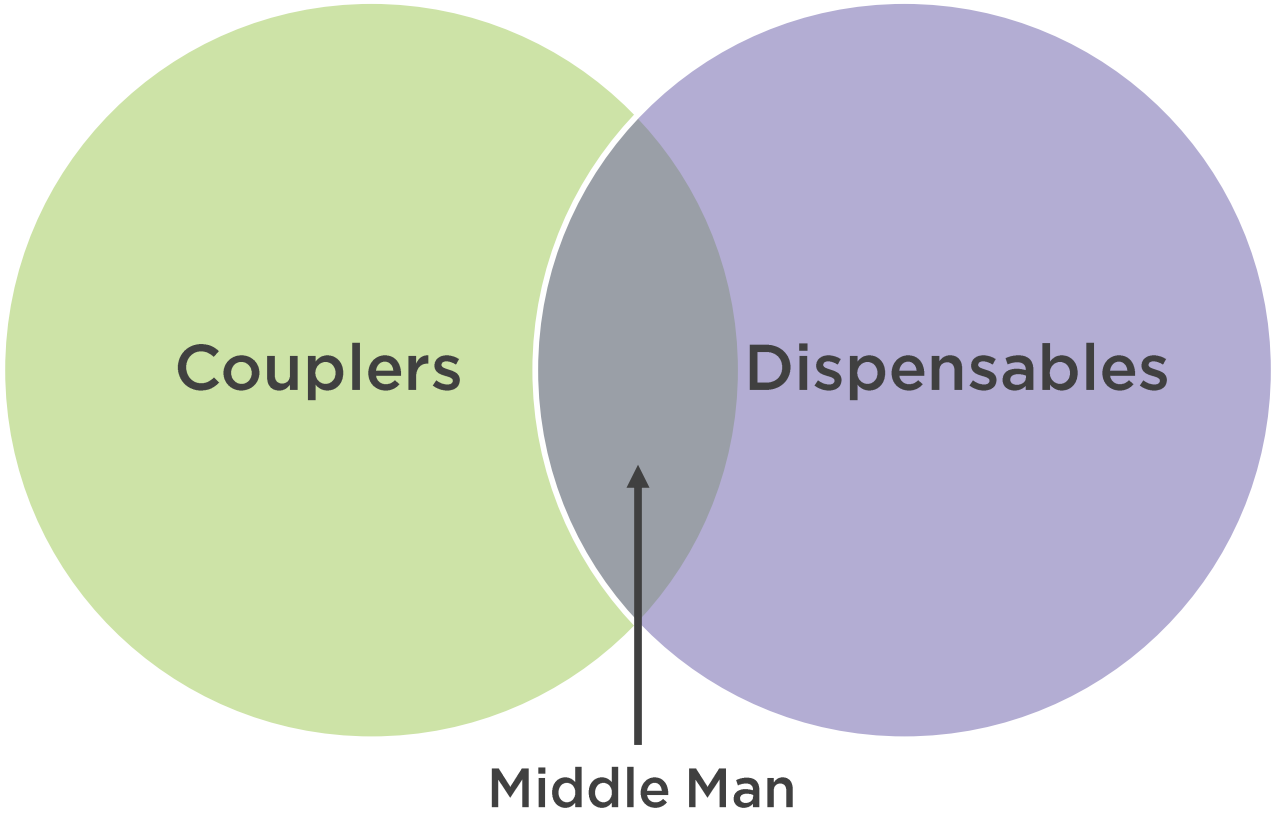
Provide a unified interface to a set of interfaces

Proxy

Provide a surrogate or placeholder for another object to control access to it

Adapter

Convert the interface of a class into another interface the clients expect


Couplers
Dispensables
Middle Man

Summary

- Keep closely related code in a single class or module
- Encapsulate and “hide” as much as possible
- Avoid message chains to keep code short and resistant to change
- Consider removing “delegation-only” classes