Recall: Design Patterns

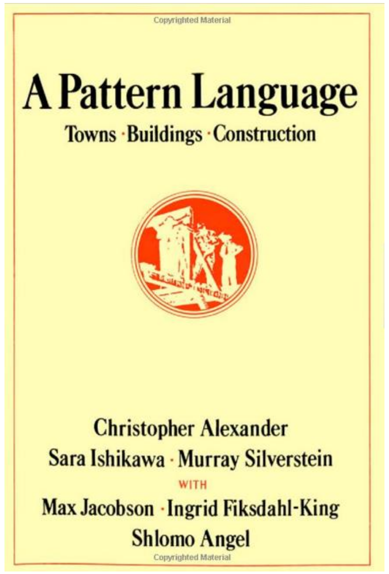
"A pattern describes a problem that occurs often, along with a tried solution to the problem" - Christopher Alexander, 1977

A Design Pattern describes parts of a problem/solution that are the same every time

Design Pattern ≠

- Specific classes or libraries
- Full design

Design Pattern = *template*



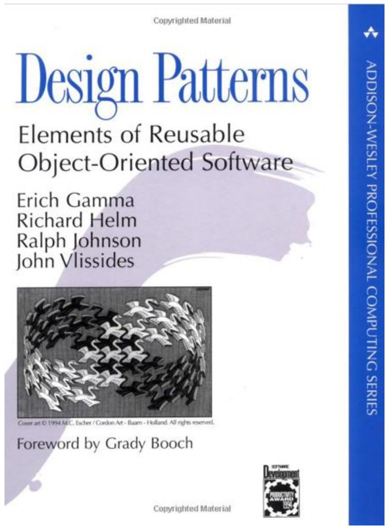
Copyrighted Material

A Pattern Language

Towns · Buildings · Construction

Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel

Copyrighted Material



Copyrighted Material

Design Patterns

Elements of Reusable Object-Oriented Software

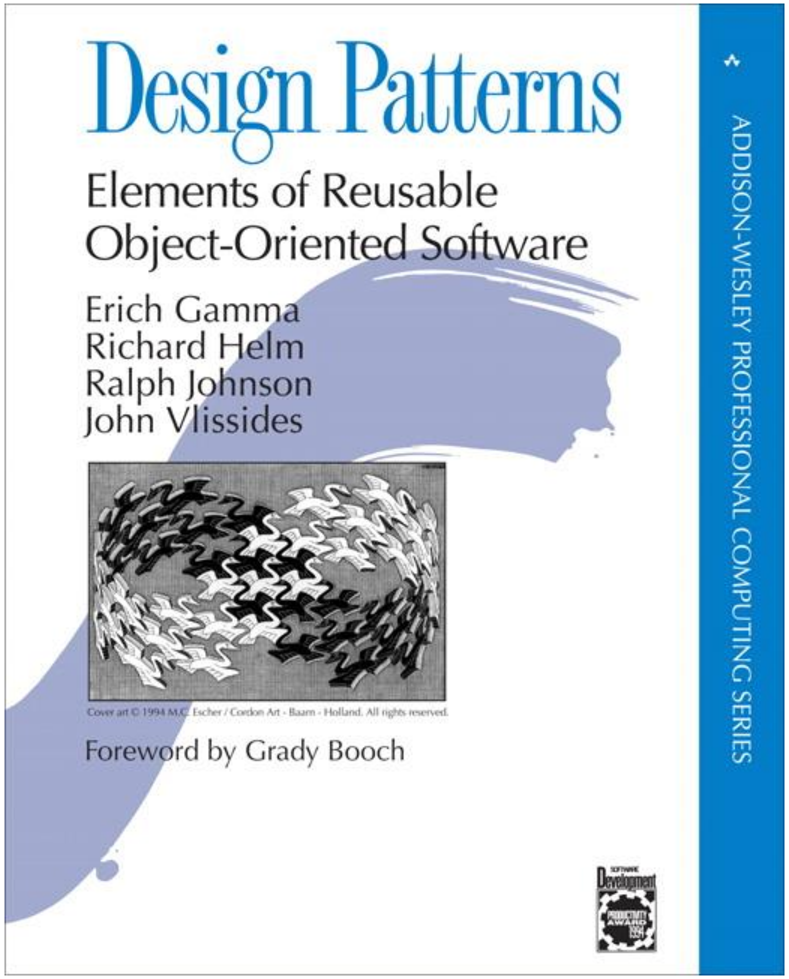
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Copyrighted Material

Design patterns in other contexts: The "gang of four"


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

- Creational
  Ways to create objects
- Structural
  Ways to combine/compose objects
- Behavioral
  Ways to communicate between objects

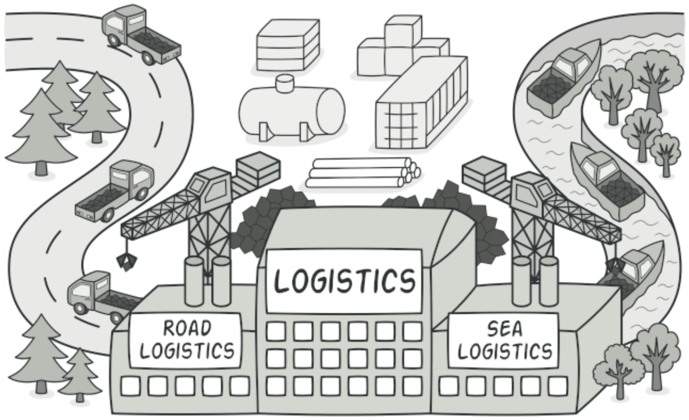
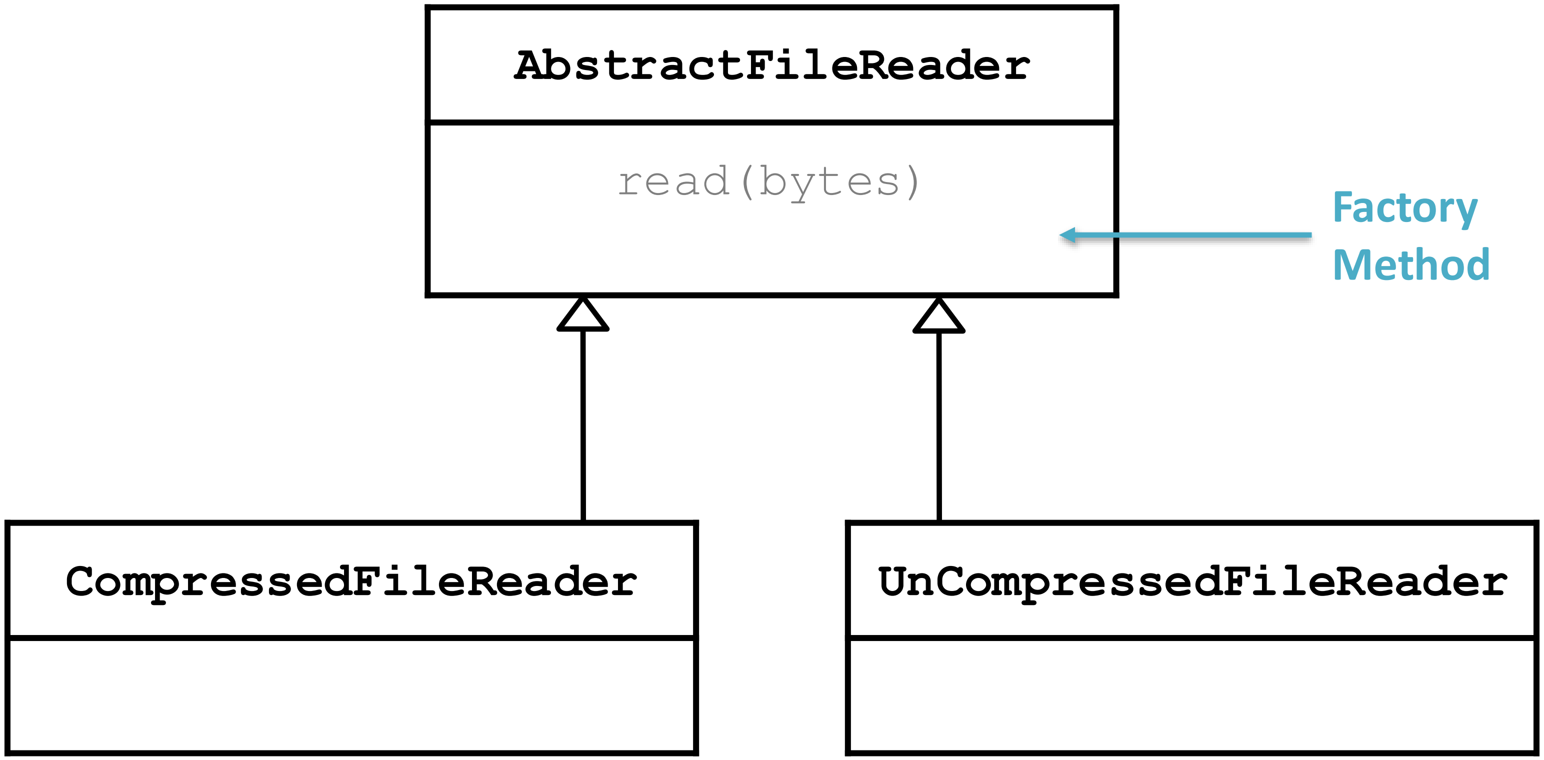


Image: https://refactoring.guru/design-patterns/factory-method

Example: I want to implement the same reader interface for compressed and uncompressed files so downstream consumers can work with either.



SOLID* OOP principles (CS312 version)

Motivation: minimize cost of change

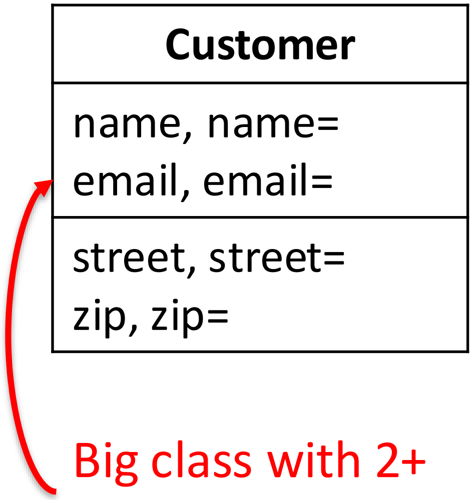
- Single Responsibility principle
- Open/Closed principle
- Liskov substitution principle
- Injection of dependencies
- Demeter principle

*Robert C. Martin

Single Responsibility Principle (SRP)

- A class should have *one and only one* reason to change
  - Each *responsibility* is a possible *axis of change*
  - Coupled changes are fragile
- What is a Classes' responsibility in ≤25 words?
- Example of many responsibilities: User model
  - A user is a moviegoer, and an authentication principal, and a social network member, etc.

Example: Extract classes in Model



Big class with 2+ responsibilities

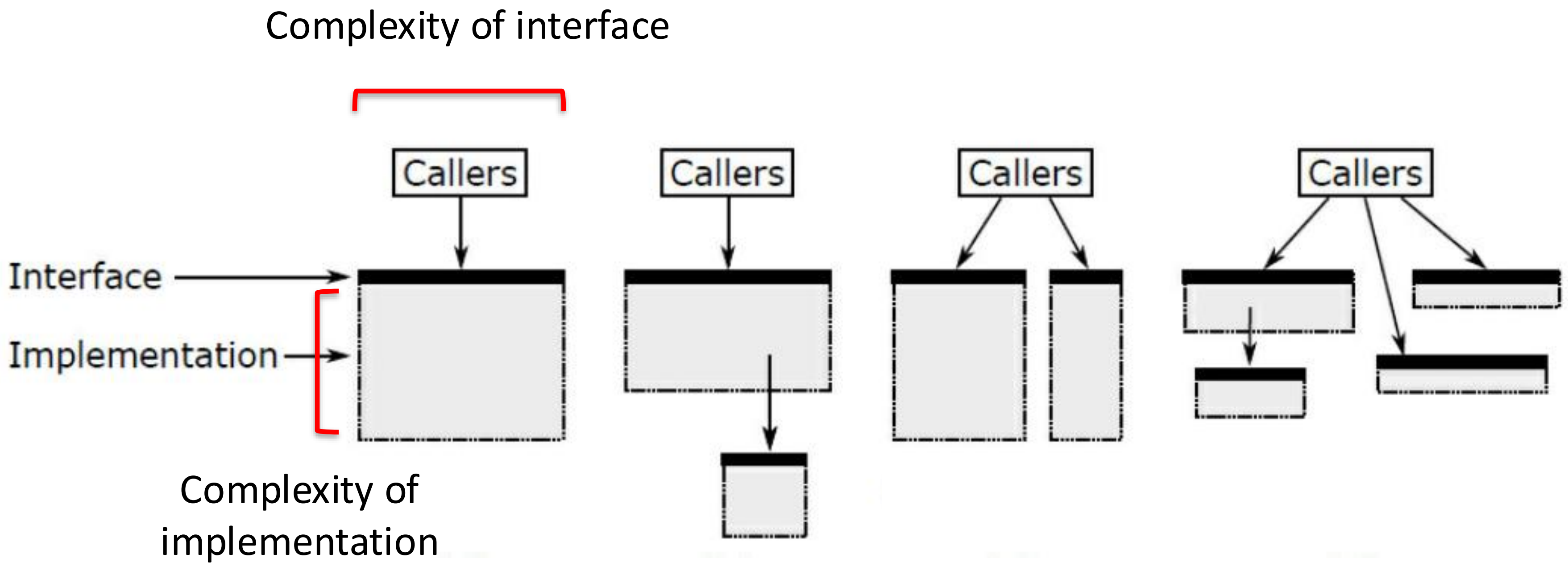
Mixins

```
const addrMix = Base => class extends Base {
  isValidZip() { … }
}
const idMix = Base => class extends Base {
  isVIP() { … }
}
class Customer extends addrMix(idMix(Model)) {
  …
}
```

Composition & Delegation

```
class Customer extends Model {
  get address() { return new Address(this); }
}
class Address {
  isValidZip() { … }
  get zip() { return this.customer.zip(); }
  …
}
```

Which of the following ways of splitting method is most likely to result in good design?



A. B. C.

Ousterhout. A Philosophy of Software Design

Summary: Single Responsibility Principle

What: A class should have exactly one responsibility or *reason to change*

Symptoms:

High LCOM (lack of cohesion of methods)

Long class with “cliques” of methods

Resolution:

Extract class(es)

Open/Closed Principle

Classes should be *open for extension,* but *closed for* ***source*** *modification*

```
class Report
  report() {
    if (this.format === "html") {
      new HtmlFormatter(this).report();
    } else if (this.format === "pdf") {
      new PdfFormatter(this).report();
    } …
  }
  …
}
```

Can’t extend (add new report types) without changing class code!

Extend the report generator

Provide mechanism for adding reporters

```
const reporters = {
  html: HtmlFormatter,
  …
}
export function registerReporter(name, klass) {
  reporters[name] = klass;
}

export default class Report
  report() {
    new reporters[this.format].report();
  }
  …
}
```

OCP In Practice

- You can’t close against *all* types of changes; you must choose and might be wrong
- Agile methodology can help *expose important types of changes early*
- Then you can try to close against *those types* of changes

Summary: Open/Closed principle

- **What:** *Extending* functionality of a class shouldn’t require *modifying* existing code, just *adding* to it
- **Symptoms:**

  Conditional statements based on class or other property that doesn’t change after assignment
- **Resolution:**

  Abstract factory pattern combined with…

  Template and strategy patterns (capture outline of algorithm’s steps, or of overall algorithm)

  Decorator (add behaviors to a base class)

Formalizing subtyping: Liskov Substitution Principle

Let φ(x) be a property provable about objects *x* of type *T*. Then φ(y) should be true for objects *y* of type *S* where *S* is a subtype of *T*.

Turing Award Winner
Barbara Liskov

TL;DR; A method that works on an instance of *type T*, should also work on any subtype of *T*

When a Square is not a Rectangle

```
class Rectangle {
  constructor(w, h) {
    this.w = w;
    this.h= h;
  }
  setWidth(w) { this.w = w; }
}

class Square extends Rectangle {
  constructor(side) {
    super(side, side);
  }
  setWidth(w) {
    this.w = w;
    this.h = h;
  }
}
```

Assumption is that changing width doesn't change height

Summary: Liskov Substitution principle

- **What:** Instance of subtype of type *T* can always be safely substituted for a *T*
- **Symptoms:**

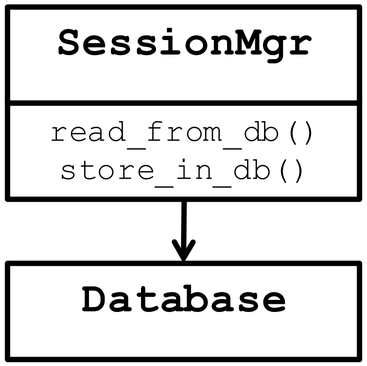
  Refused bequest: No meaningful way to implement a behavior of your superclass in a subclass
- **Resolutions:**

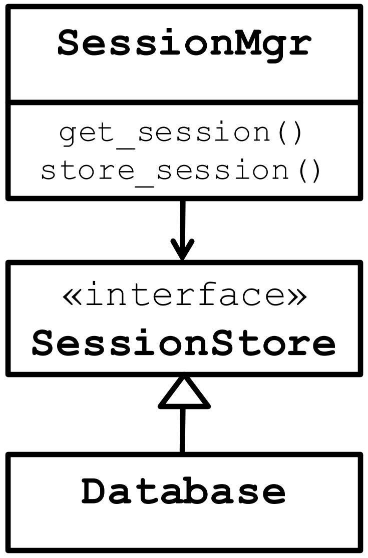
  Composition: Rather than *inheriting* from *T*, create class that has a *T* as a *component*

  Explicitly delegate method calls on *T* to component (inheritance is effectively implicit delegation)

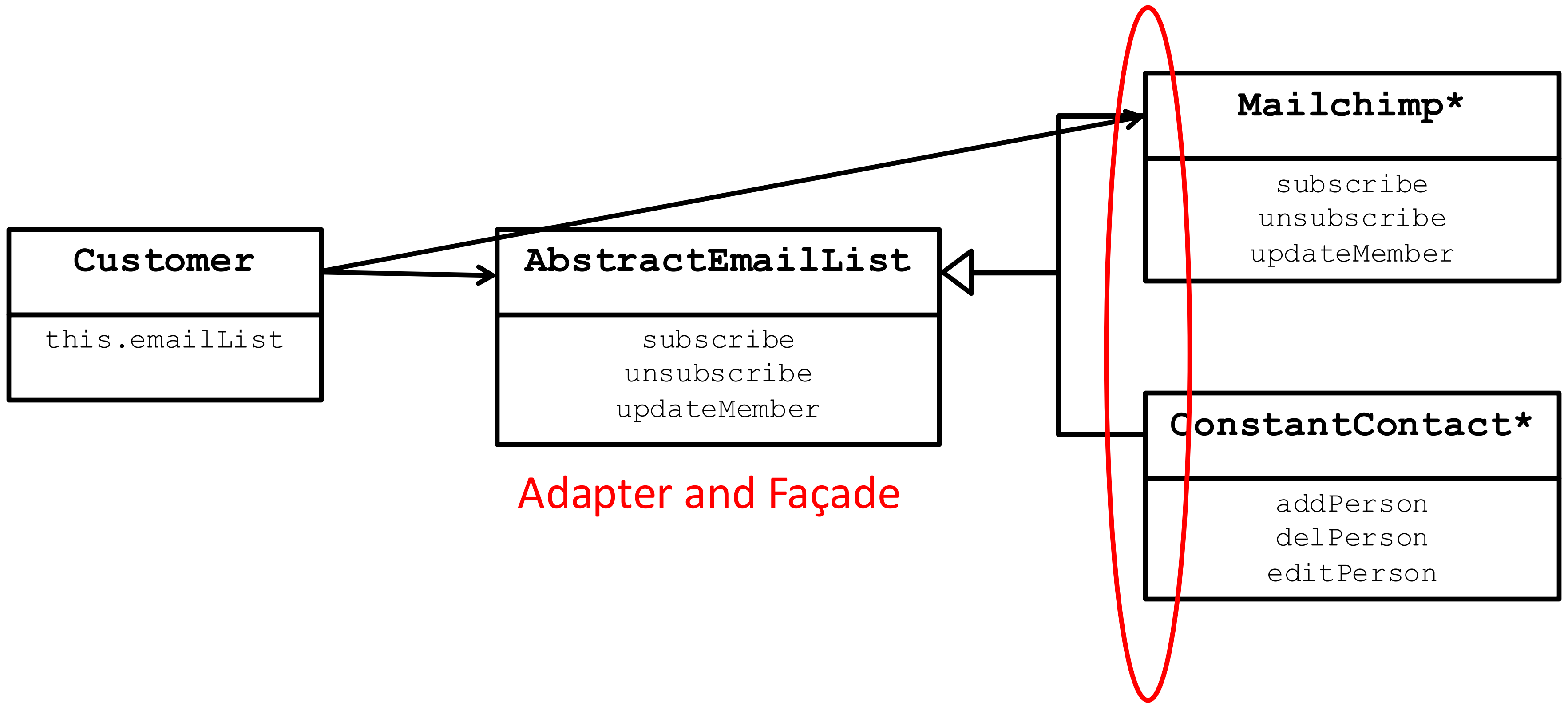
Dependency Inversion & Dependency Injection

- Problem: *A* depends on *B,* but *B's* interface & implementation can change, even if *functionality is* stable
- Solution: "Inject" an *abstract interface* that *A* & *B* depend on
  - If not exact match, Adapter/Façade
  - "Inversion": Now *B* and *A* depend on interface vs. *A* depending on *B*





DI example: Supporting external services



Adapter and Façade

Similar but not identical interfaces

*Totally made-up APIs

Summary: Injection of Dependencies Principle

- **What:** Rather than one class depending on another, have both depend on common interface
- **Symptom:**

  Classes depend on “concretions instead of abstractions”
- **Resolutions:**

  Dependency Inversion and Injection

  Adapter (convert one interface to another) and Façade (provide simplified interface)

Demeter Principle (Principle of least knowledge)

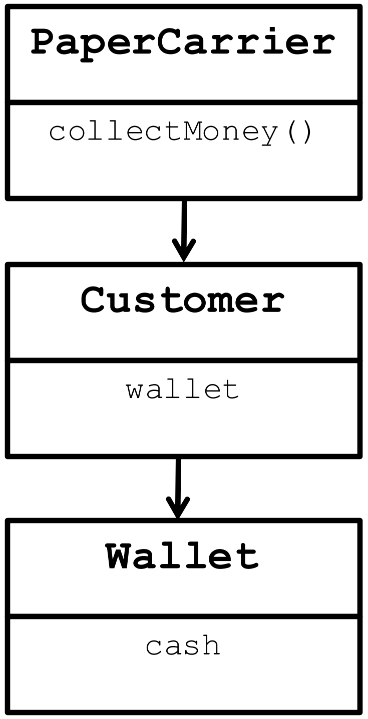
Only talk to your friends ... not strangers

You can call methods on:

- Yourself
- Your own instance variables, if applicable

But not on *the results returned by* those methods

Demeter example: Method/property chains



```
collectMoney() {
  this.customer.wallet.cash -= 10;
  this.collectedAmount += 10;
}
```

Imagine testing this code. You would need to:

1. Mock wallet with cash property
2. Mock customer with mock wallet

```
collectMoney() {
 this.collectedAmount += this.customer.pay(10);
}
```

Now just need one mock function

Summary: Demeter Principle

- **What:** Talk to friends & friends of friends; everyone else is a stranger
- **Symptoms:**

  Long chains of method calls, leading to *mock trainwrecks* in tests
- **Resolutions:**

  Replace method with delegate (e.g. wrap `customer.wallet.withdraw` in `Customer.pay`)

  Visitor pattern (separate traversal from computation)

  Observer pattern (be aware of important events)

Knex includes an "abstract" Client class for connecting with databases. Subclasses of Client exist for each database. The correct subclass is instantiated based on configuration in the knexfile. Which SOLID principles are illustrated by this example (as described here)?

A. Single Responsibility, Liskov Substitution, Dependency Inversion
B. Open/Closed, Liskov Substitution, Dependency Inversion
C. Open/Closed, Dependency Inversion, Demeter
D. All five

SOLID Caveat

- Designed for statically typed languages, so principles have more impact in that context
  - Designed, in part, to avoid changing type signatures, recompiling, etc.; not as relevant to JS.
- Use your judgment: Your goal is to *deliver working & maintainable code efficiently*

Summary

- Design patterns represent *successful solutions* to classes of problems
  - Reuse of design rather than reuse code or classes
- Can apply at many levels: architecture, design (GoF patterns), computation
- Separate what changes from what stays the same
  - *Program to interface, not implementation*
  - *Prefer composition over inheritance*
  - *Delegate!*
  - All 3 are made easier by duck typing (like in JS, Python, etc.)
- Much more to learn about — this is just a quick survey

Which of the following is true about SW architecture and design patterns in Plan & Document vs. Agile processes?

A. P&D's explicit design phase results in poor SW architecture with inappropriate use of design patterns
B. Agile prohibits doing any sort of high-level design, the code should just evolve
C. Agile can be dependent on developers' experience to plan/architect for functionality not yet implemented
D. None of the above are true

Imagine you are implementing a GUI text editor with multi-level undo/redo for both text and interface (e.g., cursor position, selection, etc.). Your current implementation has a Text class that manages the underlying text of the file, e.g., inserting and deleting text, and UI class that manages the GUI. What are some possible designs? Specifically, how could you implement undo by extending the existing Text class or with a separate class(es).

Which represents a better design in the context of the principles we discussed today?