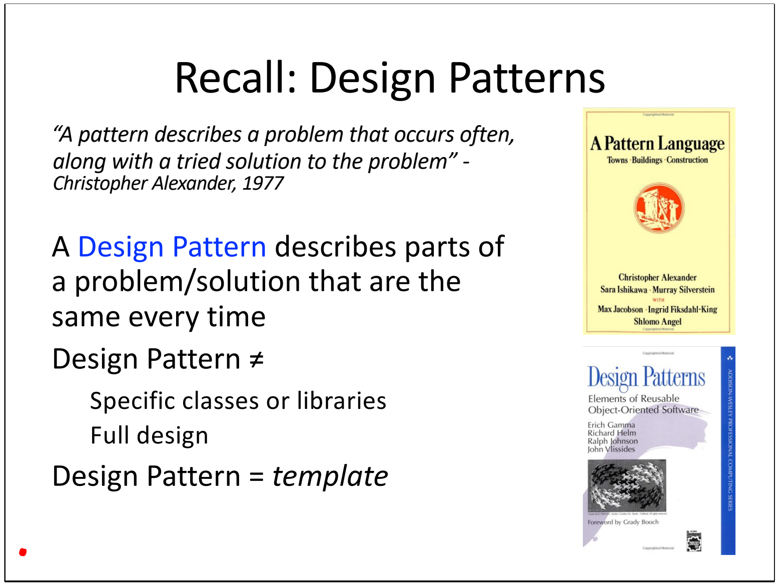


Recall: Design Patterns

"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

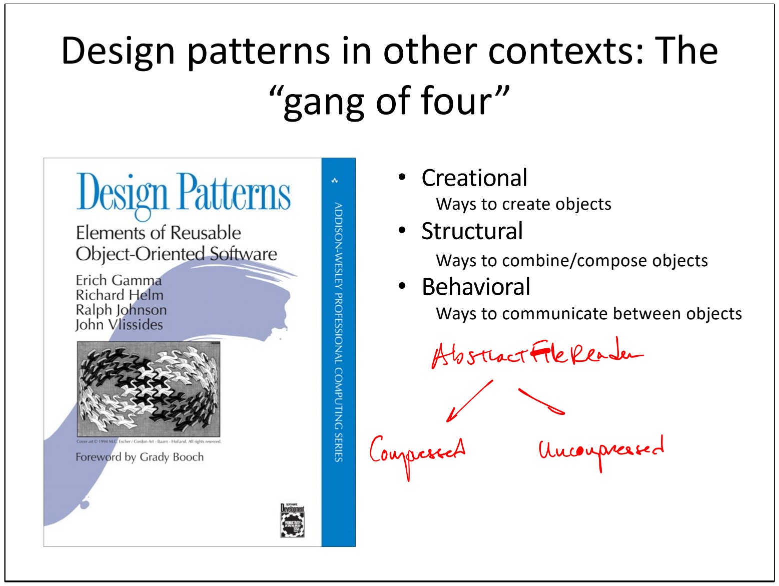
Design Pattern ≠

- Specific classes or libraries
- Full design

Design Pattern = *template*

A Pattern Language
Towns · Buildings · Construction
Christopher Alexander
Sara Ishikawa · Murray Silverstein
with
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel
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

A pattern is not a specific class or library, instead it is a template for designing a specific class or library. Over time as a SW developer, you will build up a repository of such patterns, helping you develop high-quality SW much more quickly (since you are not "starting from scratch"). You have already used many design patterns, at different scales, including the three-tier architecture pattern, model-view-controller (MVC) patterns, React's virtual-DOM based rendering approach and more. Today we will talk more about designs patterns and some other general design principles to keep in mind, especially when working with statically typed languages.



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

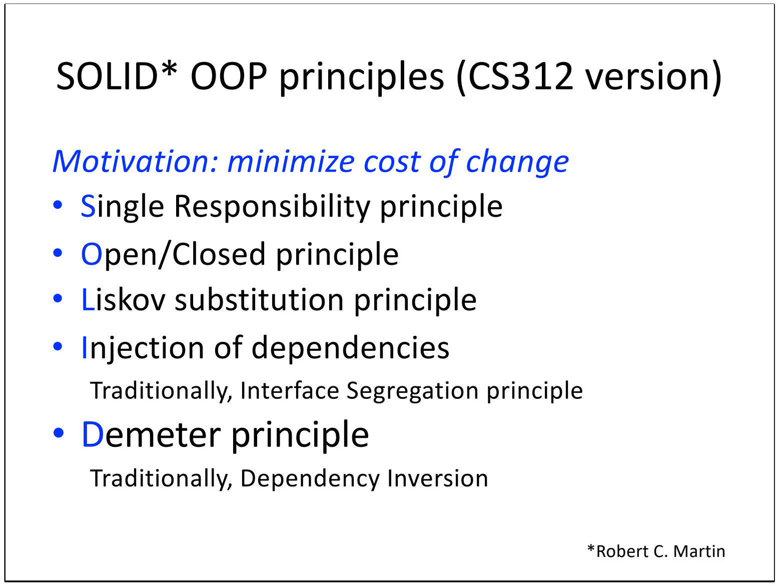
Abstract~~File~~Reader

Compressed

Uncompressed

A well-known example of design patterns comes from this very influential book on writing object-oriented software. The book describes a variety of techniques for dealing with common issues that come up in object-oriented design. An example would be the factory method, where you create a static method for creating and initializing new objects instead of a constructor. This allows you to swap in different subclasses depending on need, i.e., depending on the arguments. The caller does not need to know which sub-class should be created in any given context, that is the responsibility of the factory.

Imagine you are writing a class to work with both un-compressed and compressed files. We want to use the same interface in both contexts, that is the code that consumes the files shouldn't care whether the file is compressed or not. We can imagine creating a hierarchy with the AbstractFileReader as the base, with CompressedFileReader and UncompressedFileReader as the child classes. The factory method, which would return a AbstractFileReader*, would look at the filename, etc. and determine which class to create.



SOLID* OOP principles (CS312 version)

Motivation: minimize cost of change

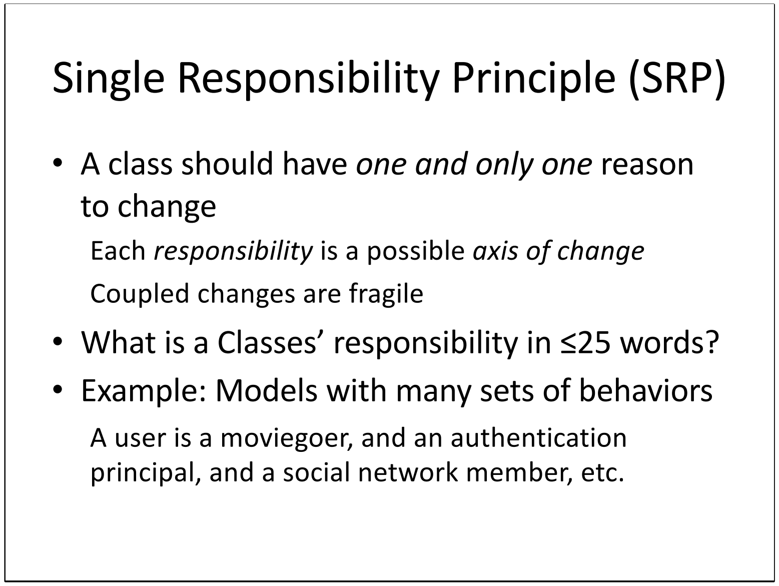
- Single Responsibility principle
- Open/Closed principle
- Liskov substitution principle
- Injection of dependencies

  Traditionally, Interface Segregation principle
- Demeter principle

  Traditionally, Dependency Inversion

*Robert C. Martin

SOLID provide a set of guidelines for defining classes in object-oriented programming (OOP) (and our code more generally). Unlike design patterns, SOLID is not a solution to a class of problems but instead a set of "cross-cutting" principles that inform how we write classes.



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

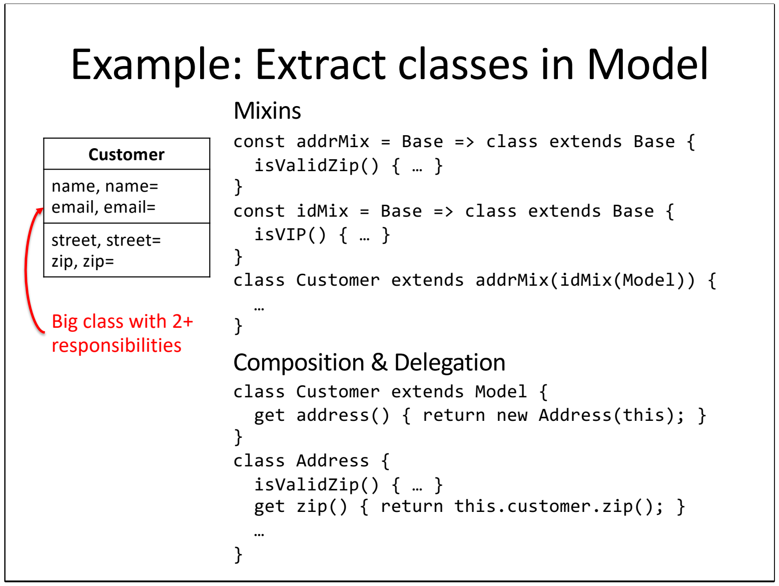
- What is a Classes' responsibility in ≤25 words?
- Example: Models with many sets of behaviors

  A user is a moviegoer, and an authentication principal, and a social network member, etc.

Change is not data changing, but requirements changing. "If you can think of more than one motive for changing a class, then that class has more than one responsibility." –Agile Software Development

Part of the craft of OO design is *defining* responsibilities and then sticking to them. This is a situation where the CRC cards can be helpful. We use that lo-fi approach to work out the responsibilities before writing any code.

In our example, we are caught between a rock and hard place (a common problem)… for efficiency we want to maintain this information in a single table, but the resulting model is overly complex. How can we handle such a situation?



Example: Extract classes in Model

Customer
name, name=
email, email=
street, street=
zip, zip=

Big class with 2+ responsibilities

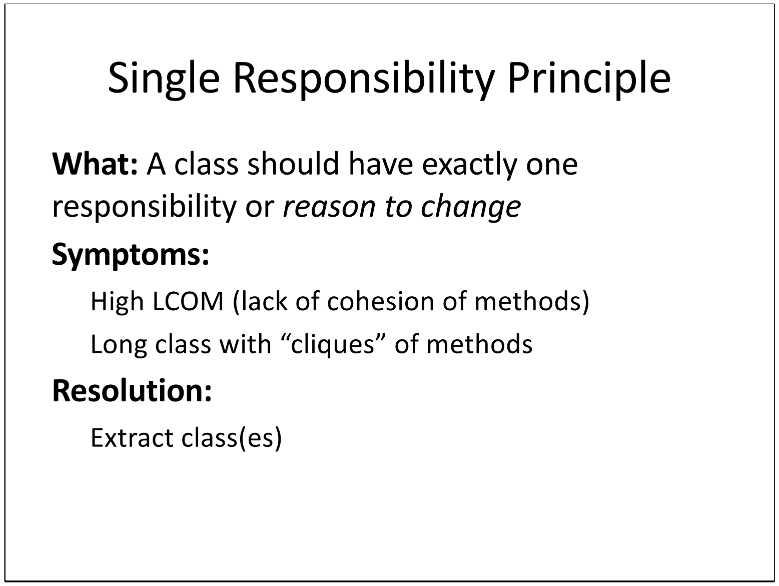
Mixins

```
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(); }
  …
}
```

Our solution is to have one table (for efficiency) but multiple responsibilities (Identity and address). We can use techniques for creating cliques of methods for each responsibility, i.e., having an id and having and address, to decouple those responsibilities for each other. The address clique would have methods like isValidZip, etc. relevant to addresses, while the identity clique would have methods like isVip, relevant to identity. Here are some examples of how we could do that in JavaScript. Notice that Customer has all the designed capabilities, but we have implemented those distinct responsibilities separately so that they can developed and tested in isolation and reused in other settings.



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)

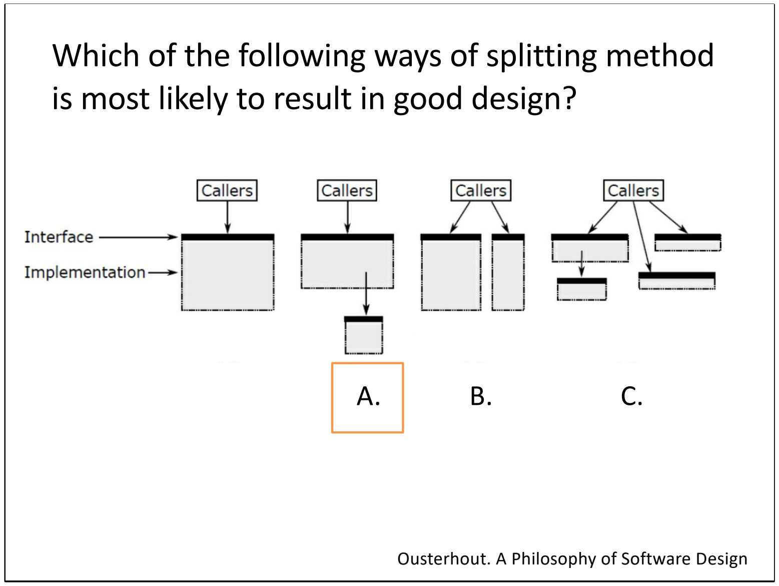
LCOM scores are are a measure of methods/variables. Many methods accessing different instance variables would have high LCOM score.

Doesn’t this create needless complexity? Possibly? “An axis of change is an axis of change only if the changes actually occur. It is not wise to apply the SRP, or any other principle for that matter, if there is no symptoms.” –Agile Software Development

Counter argument? The question we are asking is ”better together or better apart?”, i.e., should two pieces of functionality be implemented in the same or different places. The goal for the latter is to reduce overall complexity and improve modularity. But without care we can increase complexity because we are now managing multiple components, in possibly disparate locations (symptom, flipping between files). So, when might pieces of code be better together? When they share information, if they are always used together, they overlap conceptually, or if it would be hard to understand one without looking at the other.

Length is not the key determiner. Instead, our goal is a that class (method/function) ”should do one thing and do it completely.” We are aiming for simple interfaces so users don’t need to keep much in their head to use the code, and the code should be “deep”, i.e., the interface should be simpler than the implementation. If we satisfy those then length is not the issue.

Ousterhout, John K. . A Philosophy of Software Design, 2nd Edition Yaknyam Press. Kindle Edition.



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

Ousterhout. A Philosophy of Software Design

Answer: A

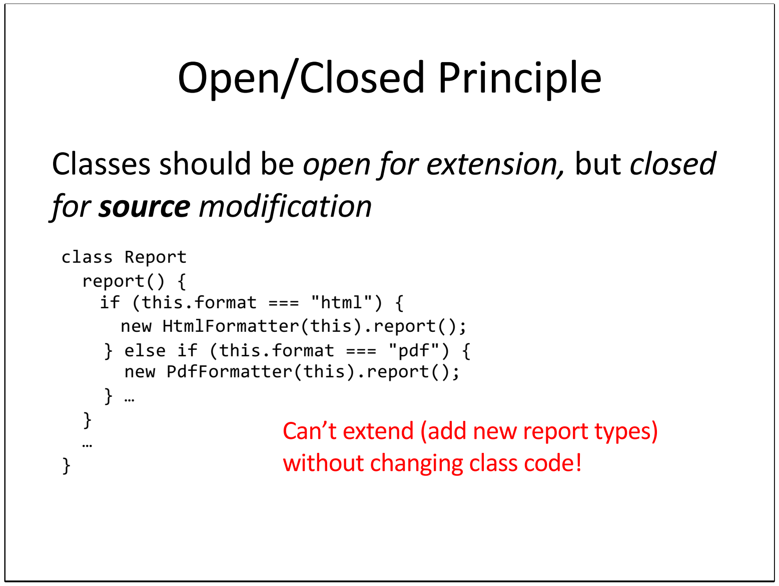
The interface remains the same, both are deep, that is the interface is simpler than the implementation. This makes sense if there is a separable subtask. Typically, such a subtask is a relatively general purpose and could be used elsewhere. Now each method has a different responsibility, and the parent does not need to know the details of the child and vice versa.

B can work but is fraught. If callers need to invoke both new methods that is probably a sign that they shared a responsibility. If, however, if most callers invoke one or the other that is a sign that they have different responsibilities and you have identified better abstractions. C is the least desirable as the caller now must deal with multiple methods, most of which are shallow, that is their interfaces and implementations are similarly complex.

Length is not the key determiner. Instead, our goal is a that class (method/function) ”should do one thing and do it completely.” We are aiming for simple interfaces so users don’t need to keep much in their head to use the code, and the code should be “deep”, i.e., the interface should be simpler than the implementation. If we satisfy those then length is not the issue.

Ousterhout, John K. . A Philosophy of Software Design, 2nd Edition Yaknyam Press.

Kindle Edition.



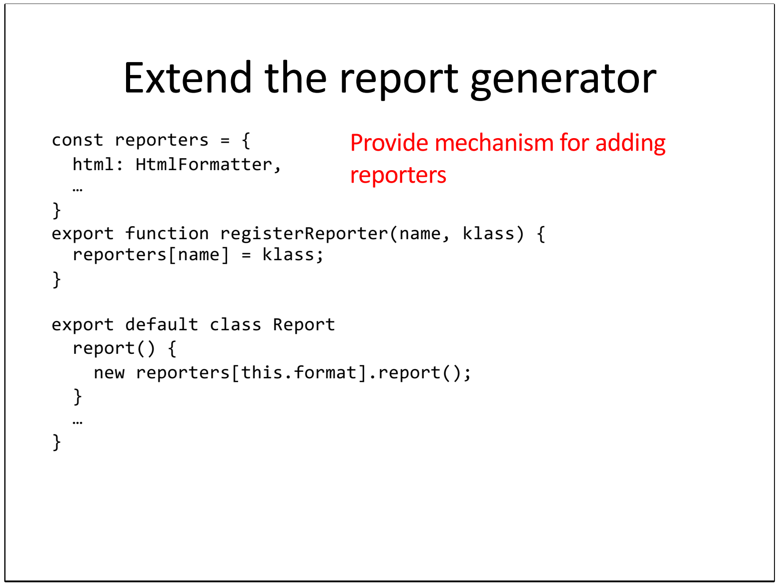
Open/Closed Principle

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!

What is the issue here? How could we update to observe the OCP?
“Open for extension”: The behavior of the module can be extended, e.g., we can add additional report types
“Closed for modification”: Extending the behavior does not result in changes to the existing source code



Extend the report generator

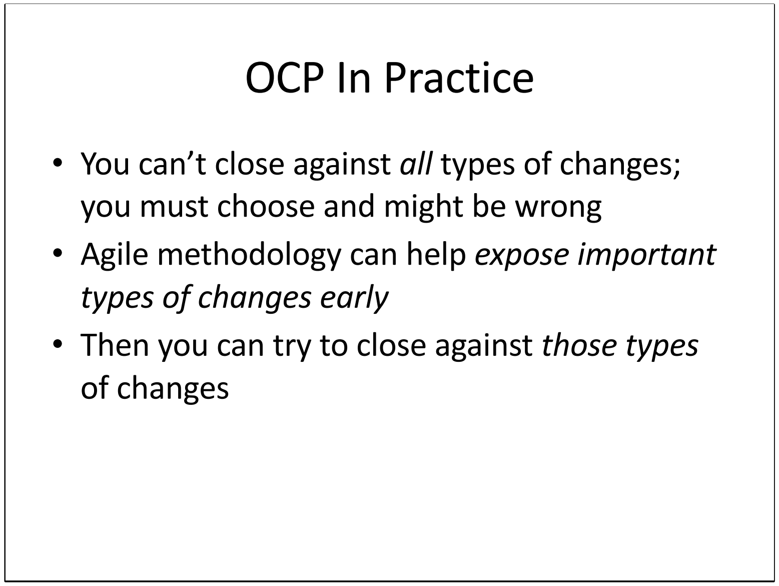
```
const reporters = {
  html: HtmlFormatter,
  …
}
export function registerReporter(name, klass) {
  reporters[name] = klass;
}

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

Provide mechanism for adding reporters

In OOP, the primary mechanism for extension is creating abstract class (interfaces), e.g., a Reporter interface, and then use polymorphic dispatch. Here we use duck typing in lieu of interfaces and go one step further and use a form of reflection to translate strings into classes (and a registry). `Report.report` is now "closed" to modification for adding report types.

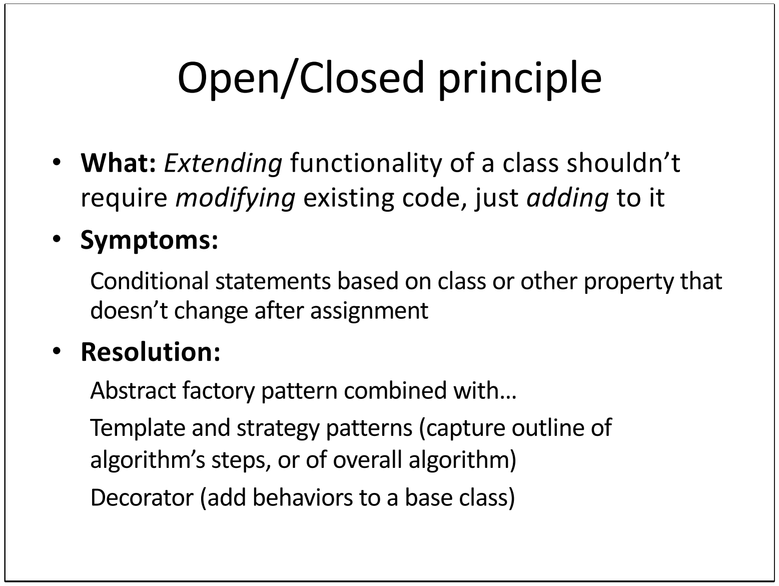
In some languages the registry would be unnecessary, as they have more reflection support (i.e., can obtain class by name).



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

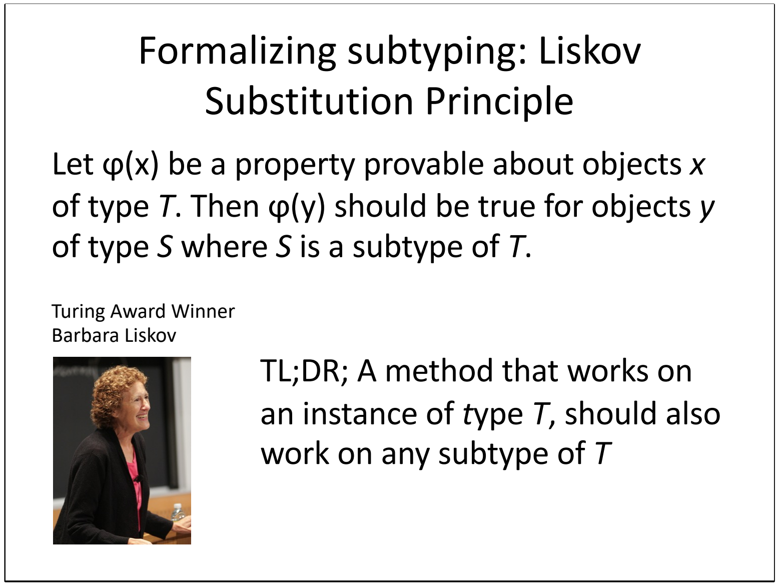
Close sounds ”negative” but remember that here it means enable extension without source modification (i.e., minimizing the cost of change).



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)

Template/strategy patterns use polymorphism to customize a common set of steps.



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*

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.

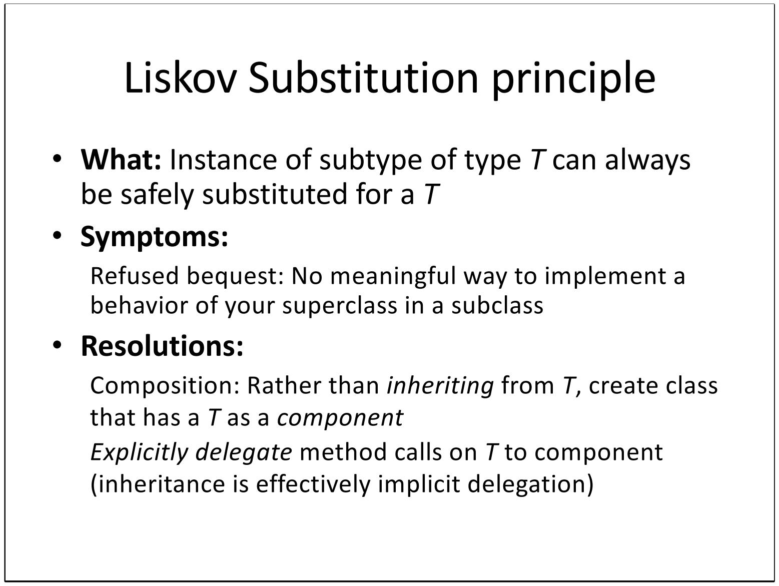
When a Square is not a Rectangle

```
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;
  }
}
```

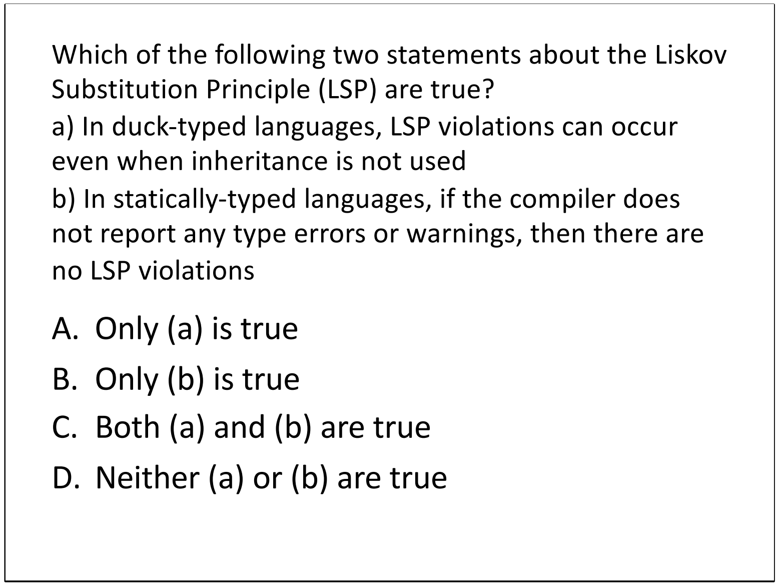
Assumption is that changing width doesn’t change height

✗



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)



Which of the following two statements about the Liskov Substitution Principle (LSP) are true?

a) In duck-typed languages, LSP violations can occur even when inheritance is not used

b) In statically-typed languages, if the compiler does not report any type errors or warnings, then there are no LSP violations

A. Only (a) is true

B. Only (b) is true

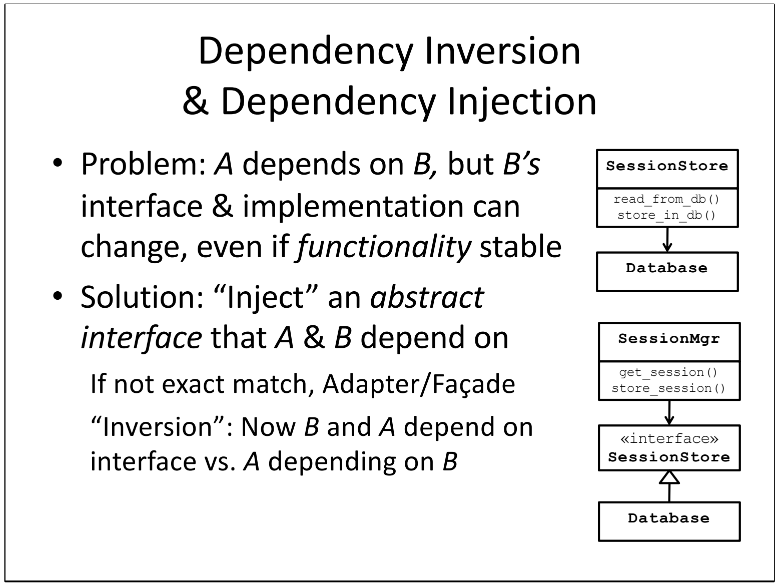
C. Both (a) and (b) are true

D. Neither (a) or (b) are true

Answer: A

LSP isn't necessarily tied to inheritance or class-based typing and thus applies to both duck-typed and statically-typed languages. Any time polymorphism is used (in whatever form) the LSP is applicable. But as we saw in our square/rectangle example, successfully compiling does not ensure there are no LSP violations.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license and https://stefanroock.wordpress.com/2010/11/08/the-liskov-substitution-principle-lsp-in-duck-typed-programming-languages/.



Dependency Inversion & Dependency Injection

- Problem: *A* depends on *B,* but *B's* interface & implementation can change, even if *functionality* 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*

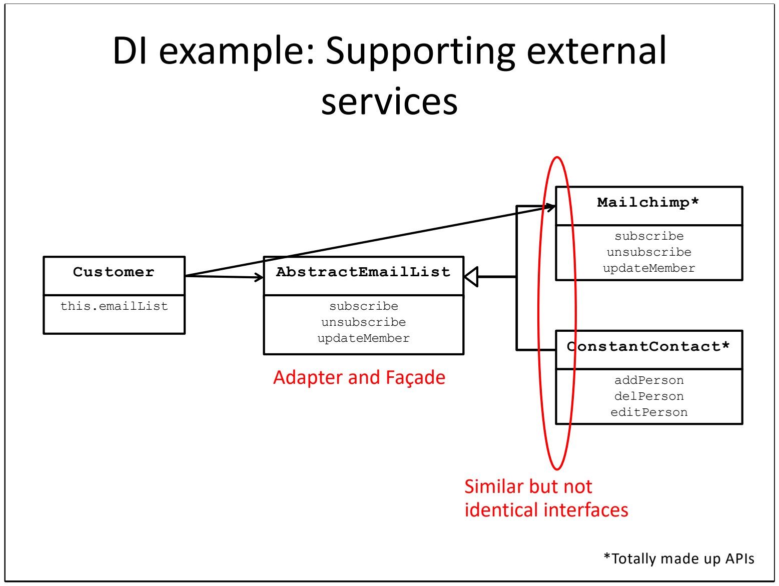
SessionStore
read_from_db()
store_in_db()
Database
SessionMgr
get_session()
store_session()
«interface»
SessionStore
Database

It seems natural to have a high-level module (SessionStore) depend on a low-level module (the DB), but that creates problematic coupling when B changes (e.g., using a different database). High-level modules should influence low-level modules, not the other way around. We solve this by ”inverting” the dependencies so that A uses an interface implemented by B (i.e., they both depend on the interface).

Injection is the means of supplying a valid implementation (often by supplying the implementation class as a constructor argument).

TL;DR; High-level classes shouldn’t create concrete instances of lower-level classes

Quick reminder about interfaces… In statically typed languages, interfaces are typically abstract classes (method signatures, but not implementations) that specify what methods a class provides (and you can reference an object through through interface type). In JS you can implement an interface by mixing in new methods.



DI example: Supporting external services

Customer

this.emailList

AbstractEmailList

subscribe
unsubscribe
updateMember

Adapter and Façade

Mailchimp*

subscribe
unsubscribe
updateMember

ConstantContact*

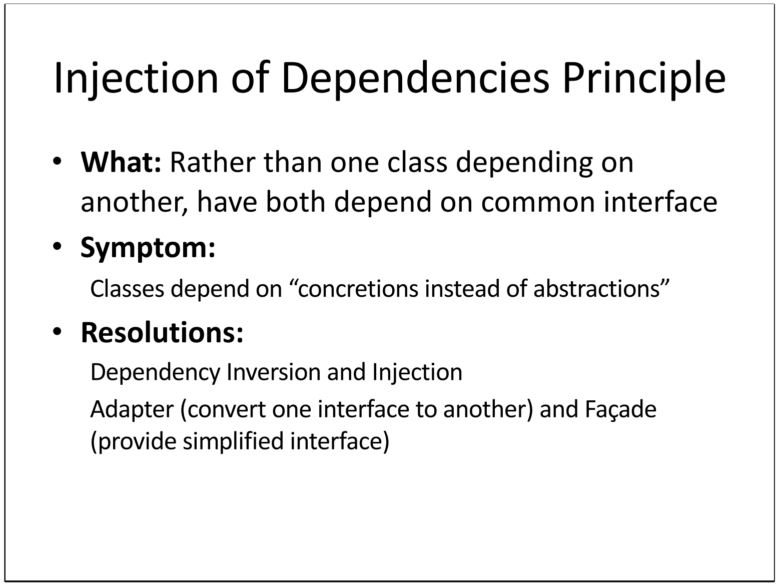
addPerson
delPerson
editPerson

Similar but not
identical interfaces

*Totally made up APIs

We want to use an external service for e-mail and may end up using different services in different contexts. Instead of tying our code to a specific “concrete” service, define an Abstract interface. In this case that interface will not exactly match the similar but not identical APIs. Interface serves as an adapter (and a façade – our interface likely only provides a subset of functionality of the e-mail services).

An example of DI in React are higher-order components, that is components that take other components as props.



Injection of Dependencies Principle

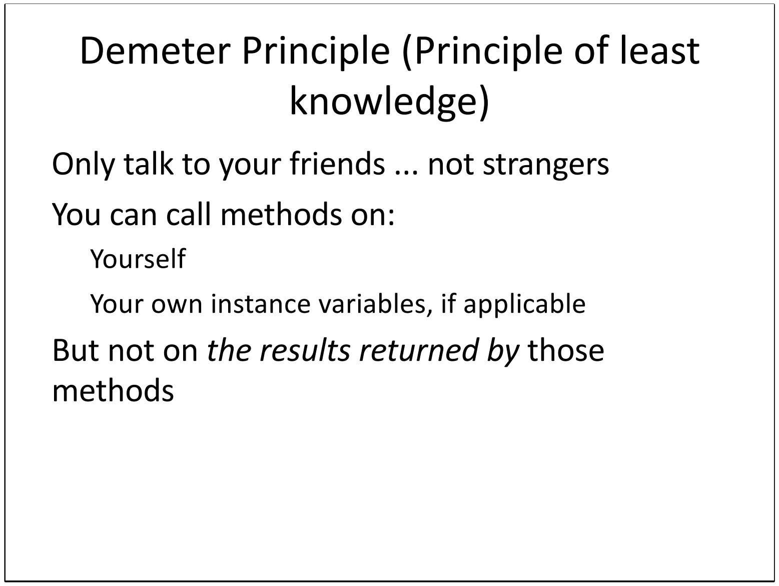
- **What:** Rather than one class depending on another, have both depend on common interface
- **Symptom:**

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

  Dependency Inversion and Injection

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

By concretion we mean specific implementation, i.e. MailChimp, instead of abstract interfaces, like AbstractEmailList. We often use the adapter pattern when creating a common abstract interface for multiple underlying implementations. Façade is used to provide a simplified version of an interface.



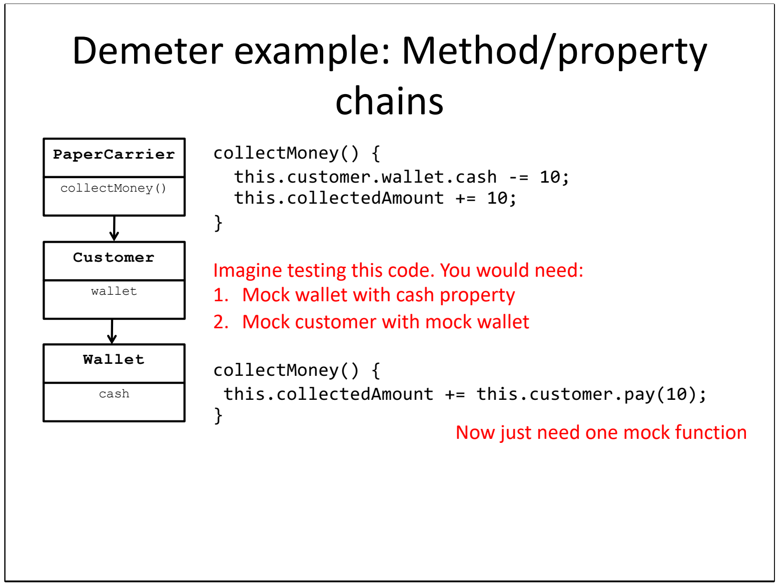
Demeter Principle (Principle of least knowledge)

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

PaperCarrier
collectMoney()
Customer
wallet
Wallet
cash

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

Imagine testing this code. You would need:

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

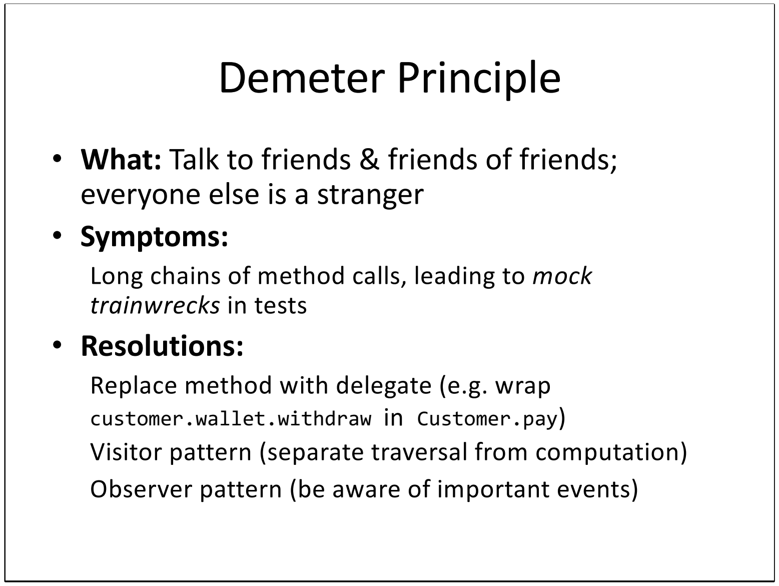
```
collectMoney() {
 this.collectedAmount += this.customer.pay(10);
}
```

Now just need one mock function

Where is the Demeter violation lurking here? Notice that we are accessing the cash property of the customer’s wallet property. Under the Demeter principle we are not allowed to go ”past” wallet.

Recall that mocks are synthetic implementations of methods, etc. use for testing, i.e. fake objects or methods with known properties or results.

The complexity of the mocking is an indication we have violated the Demeter principle. In our new version we call a method on on our instance variables (i.e. this.customer) but don’t access properties returned by those (i.e. don’t access the cash property on the wallet accessed via customer). As a result we don’t need to mock nearly as much.



Demeter Principle

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

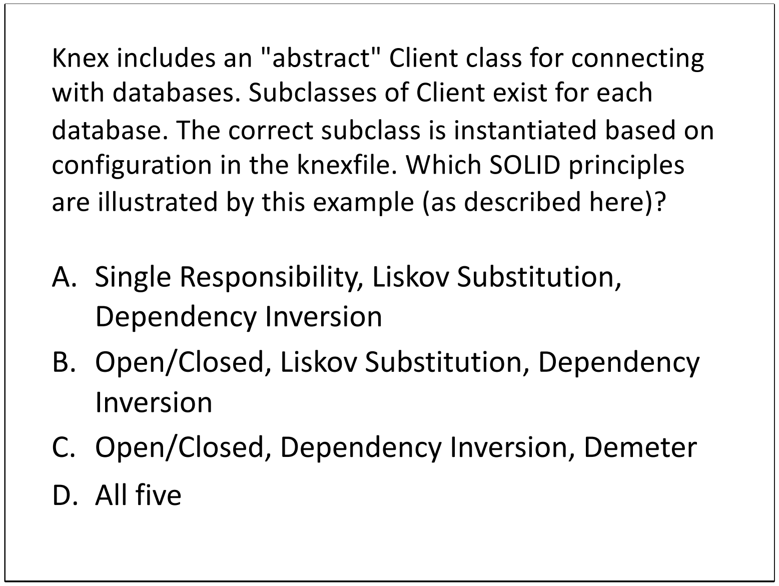
  Long chains of method calls, leading to *mock trainwrecks* in tests
- **Resolutions:**

  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)

A common example of this (that we accept) is parsing JSON responses in `fetch`. Mocking that is awkward (we have to create a mock response, returned by a mock fetch…). We utilize libraries to help us with that process! Or in the case of PA4, try to avoid mocking fetch entirely and test directly against a mock server.



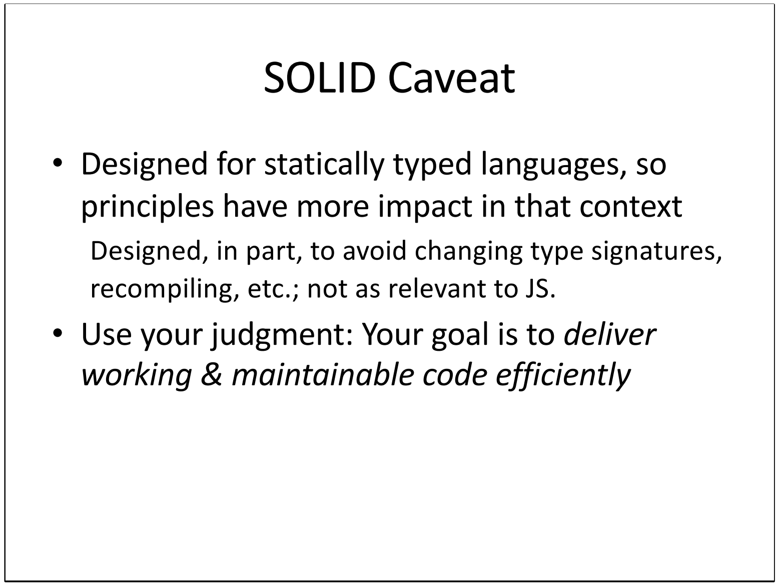
Knex includes an "abstract" Client class for connecting with databases. Subclasses of Client exist for each database. The correct subclass is instantiated based on configuration in the knexfile. Which SOLID principles are illustrated by this example (as described here)?

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

Answer: B

From the available description, knex demonstrates Open/Closed (open to extending databases without modification), Dependency Inversion (in defining the Client interface), and Liskov Substitution (each concrete implementation can stand in place of the abstract interface). It may also observe the other principles, but we don't know that from this information.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.

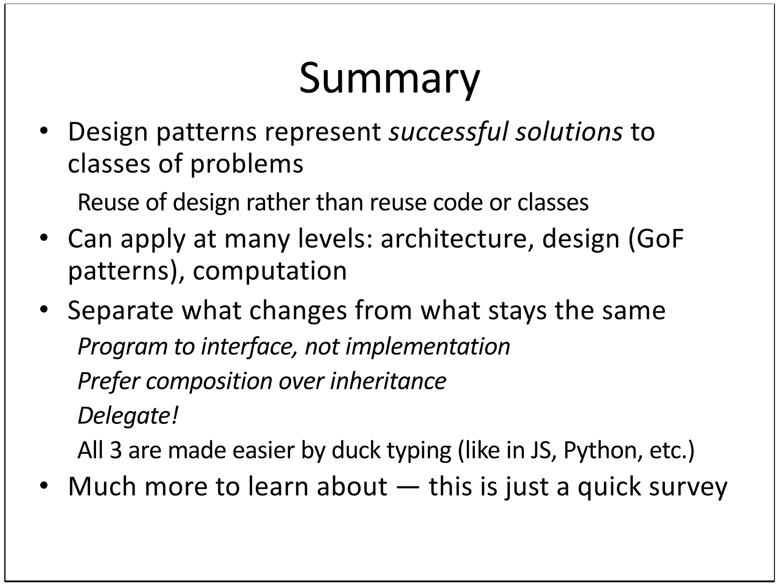


SOLID Caveat

- 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.

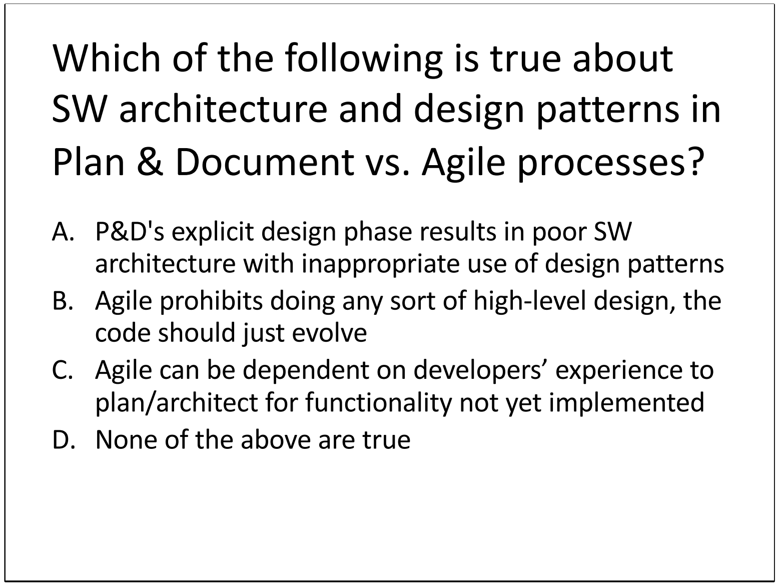
- Use your judgment: Your goal is to *deliver working & maintainable code efficiently*



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

Recall that our original goal was to minimize the cost of change. And we saw many guidelines for doing so. At their heart the guidelines (and with design patterns generally) is to separate the aspects that change (from problem to problem, or for a given application as it evolves) from those that don’t. By doing so, we achieve our goal of minimizing the cost of change.



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

Answer: C

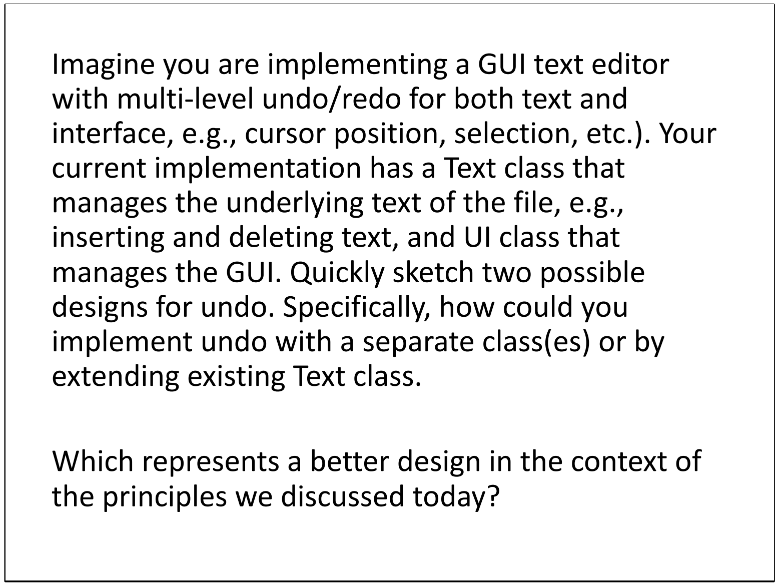
One of the criticisms of Agile is that encourages developers to start without any design and thus is too reliant on later refactoring. Agile doesn't preclude all design but depending on approach can be dependent on developers keeping past experiences in mind (something you might have experienced), that is anticipating future needs and writing code today accordingly. The issue with P&D is not that the design phase leads to poor architecture, but that the designed architecture is no longer the right approach as the application evolves.

“One of the risks of agile development is that it can lead to tactical programming. Agile development tends to focus developers on features, not abstractions, and it encourages developers to put off design decisions in order to produce working software as soon as possible. For example, some agile practitioners argue that you shouldn’t implement general-purpose mechanisms right away; implement a minimal special-purpose mechanism to start with, and refactor into something more generic later, once you know that it’s needed. […] This can result in a rapid accumulation of complexity.”

“It’s fine to put off all thoughts about a particular abstraction until it’s needed by a feature. Once you need the abstraction, invest the time to design it cleanly;”

Ousterhout, John K. . A Philosophy of Software Design, 2nd Edition (pp. 155-156). Yaknyam Press. Kindle Edition.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license



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. Quickly sketch two possible designs for undo. Specifically, how could you implement undo with a separate class(es) or by extending existing Text class.

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

Design 1: Maintain the undo log in the Text class, e.g., whenever you insert text add a corresponding operation to the internal list of changes. To undo the UI would invoke a method on the text class to revert the changes. "For entries related to text, it updated the internals of the text class; for entries related to other things, such as the selection, the text class called back to the user interface code to carry out the undo or redo."

Design 2: A separate History class that maintains the undo/redo list. Each actions adds a corresponding action to this list where the action is an object that implements a shared interface. The History class doesn't know the specifics of the individual actions, instead it just walks the list as needed invoking the actions. Each operation has a corresponding specialized undo action. "The text class might implement UndoableInsert and UndoableDelete objects to describe text insertions and deletions. Whenever it inserts text, the text class creates a new UndoableInsert object describing the insertion and invokes History.addAction to add it to the history list. The editor's user interface code might create UndoableSelection and UndoableCursor objects that describe changes to the selection and insertion cursor."

Design 1 violates Single Responsibility and Open/Closed. The Text class seems to have two distinct responsibilities, managing text and managing undo. And adding undoable actions unrelated to text requires modifying the text class. Design 2 is a better choice.

This example from adapted from John Ousterhout, and he talks about it in a different way, specifically as general purpose vs. special purpose, specifically writing “The key design decision was the one that separated the general-purpose part of the undo mechanism from the special-purpose parts, creating a separate class for the general-purpose part and pushing the special-purpose parts down into subclasses of History.Action. Once that was done, the rest of the design fell out naturally.”

Ousterhout, John K. . A Philosophy of Software Design, 2nd Edition (p. 46-48). Yaknyam Press. Kindle Edition.