Design Patterns Promote Reuse

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

- Christopher Alexander's 253 (civil) architectural patterns range from the creation of cities (2. distribution of towns) to particular building problems (232. roof cap)
- A pattern language is an organized way of tackling an architectural problem using patterns

Kinds of Patterns in Software

- Architectural ("macroscale") patterns
  - Model-view-controller
  - Pipe & Filter (e.g. compiler, Unix pipeline)
  - Event-based (e.g. interactive game)
  - Layering (e.g. SaaS technology stack)
- Computation patterns
  - Fast Fourier transform
  - Structured & unstructured grids
  - Dense linear algebra
  - Sparse linear algebra
- *GoF (Gang of Four) Patterns: structural, creational, behavior*

The Gang of Four (GoF)

- 23 *structural* design patterns
- description of communicating objects & classes
- captures common (and successful) solution to a *category* of related problem instances
- can be customized to solve a specific (new) problem in that category

- Pattern ≠
  - individual classes or libraries (list, hash, ...)
  - full design—more like a blueprint for a design

The GoF Pattern Zoo

Creation

1. Factory
2. ***Abstract factory***
3. Builder
4. Prototype
5. ***Singleton/Null obj***

Structural

6. ***Adapter***
7. ***Composite***
8. ***Proxy***
9. Bridge
10. Flyweight
11. ***Façade***
12. ***Decorator***

Behavioral

13. ***Observer***
14. Mediator
15. Chain of responsibility
16. Command
17. Interpreter
18. ***Iterator***
19. Memento (memoization)
20. State
21. ***Strategy***
22. ***Template***
23. Visitor

Meta-Patterns

Separate out the things that change from those that stay the same

1. Program to an Interface, not Implementation
2. Prefer composition & delegation over Inheritance

- delegation is about interface sharing, inheritance is about implementation sharing

Antipattern

- Code that looks like it should probably follow some design pattern, but it doesn't
- Symptoms:
- Viscosity (easier to do hack than Right Thing)
- Immobility (can't DRY out functionality)
- Needless repetition (comes from immobility)
- Needless complexity from generality

SOLID OOP principles
(Robert C. Martin, co-author of Agile Manifesto)

Motivation: minimize cost of change

- Single Responsibility principle
- Open/Closed principle
- Liskov substitution principle
- Injection of dependencies
- traditionally, Interface Segregation principle
- Demeter principle

Refactoring & Design Patterns

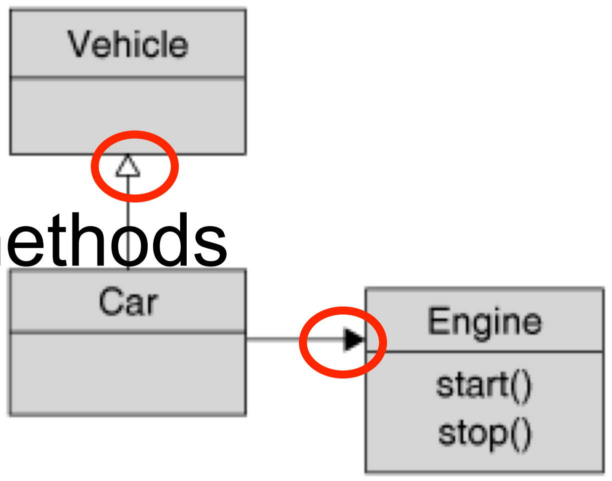
Methods within a class	**Relationships among classes**
Code smells	Design smells
Many catalogs of code smells & refactorings	Many catalogs of design smells & design patterns
Some refactorings are superfluous in Ruby	Some design patterns are superfluous in Ruby
Metrics: ABC & Cyclomatic Complexity	Metrics: Lack of Cohesion of Methods (LCOM)
Refactor by extracting methods and moving around code within a class	Refactor by extracting classes and moving code between classes
SOFA: methods are **S**hort, do **O**ne thing, have **F**ew arguments, single level of **A**bstraction	SOLID: **S**ingle responsibility per class, **O**pen/closed principle, **L**iskov substitutability, **I**njection of dependencies, **D**emeter principle

Which statement is FALSE?

- ☐ Software that uses more design patterns isn’t necessarily better.
- ☐ Well-designed software can evolve to the point where patterns become antipatterns.
- ☐ Trying to apply design patterns too early can be just as bad as applying them too late.
- ☐ Most design patterns are specific to a particular subset of programming languages.

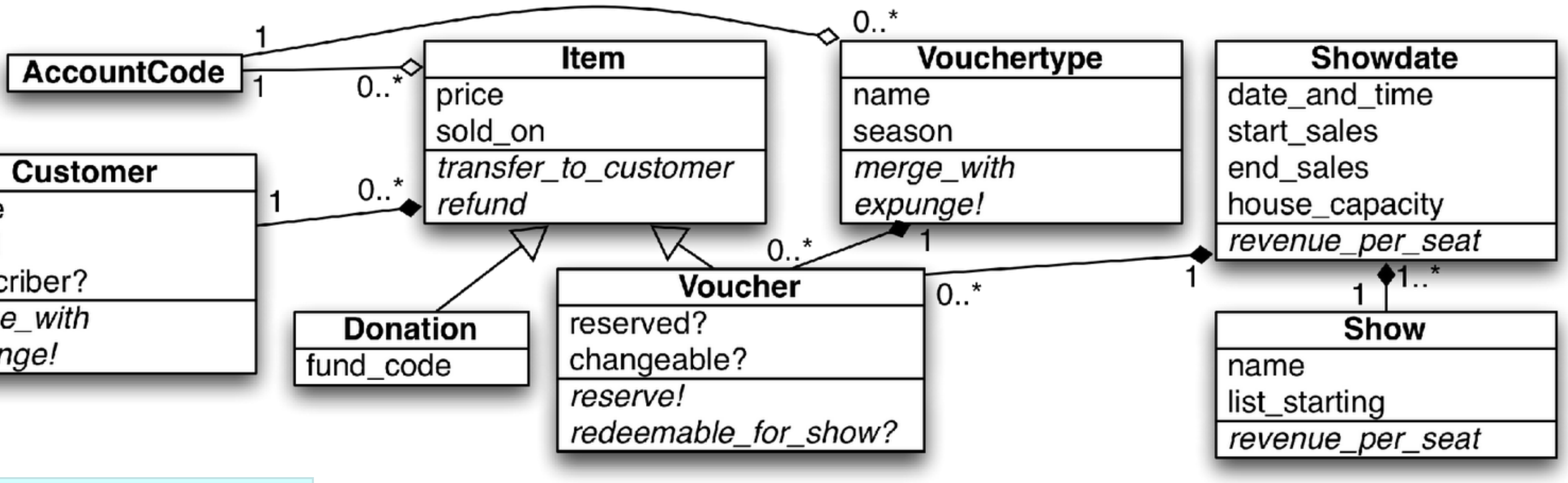
Just Enough UML

- Unified Modeling Language: notation for describing various artifacts in OOP systems
- One type of UML diagram is a *class diagram*, showing class relationships and principal methods:
- Car is a subclass of Vehicle
- Engine is a *component* of Car
- Engine class includes start(), stop() methods



Relationships

Aggregation



Composition

Inheritance

(Too Much UML)

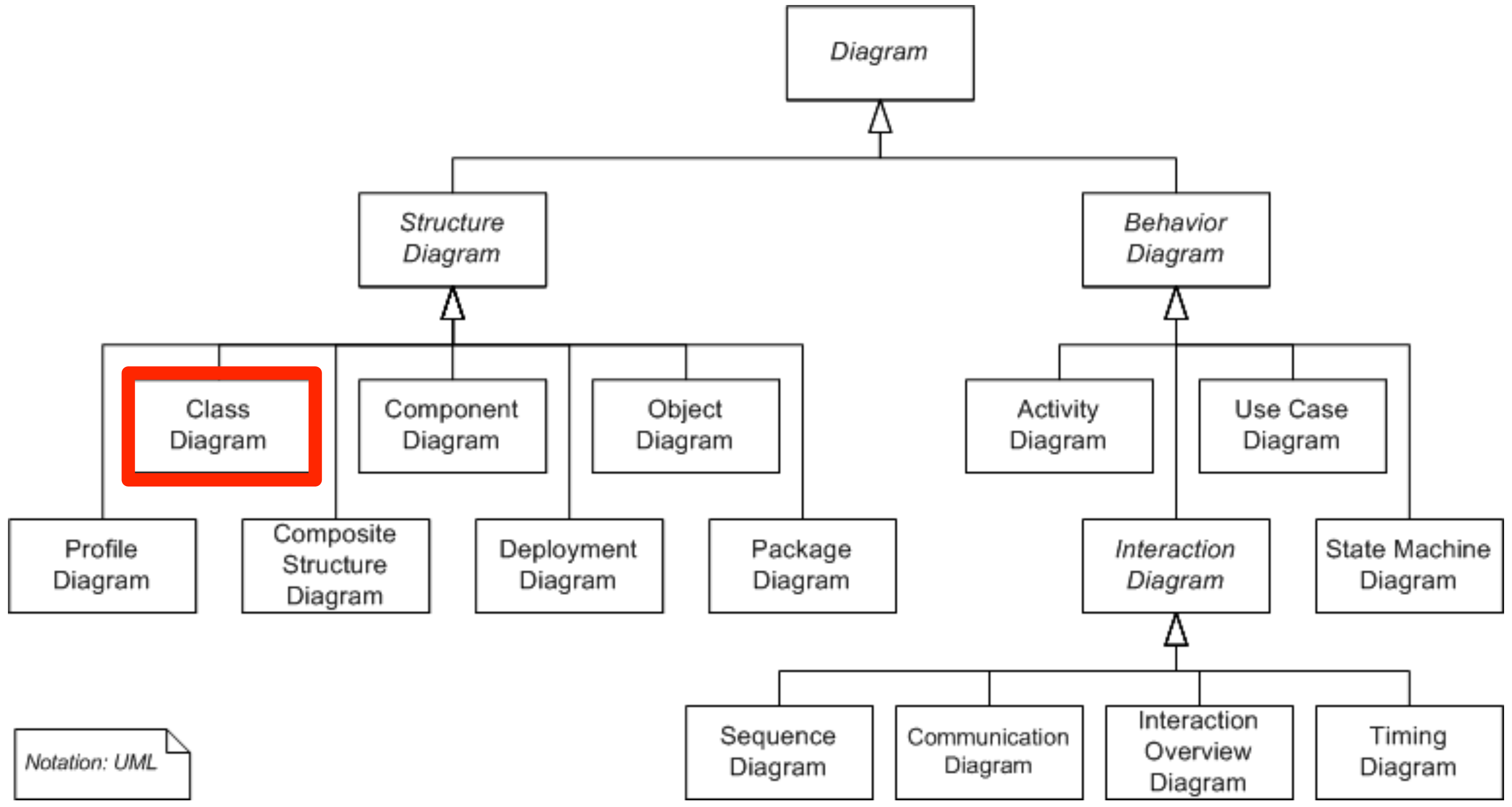
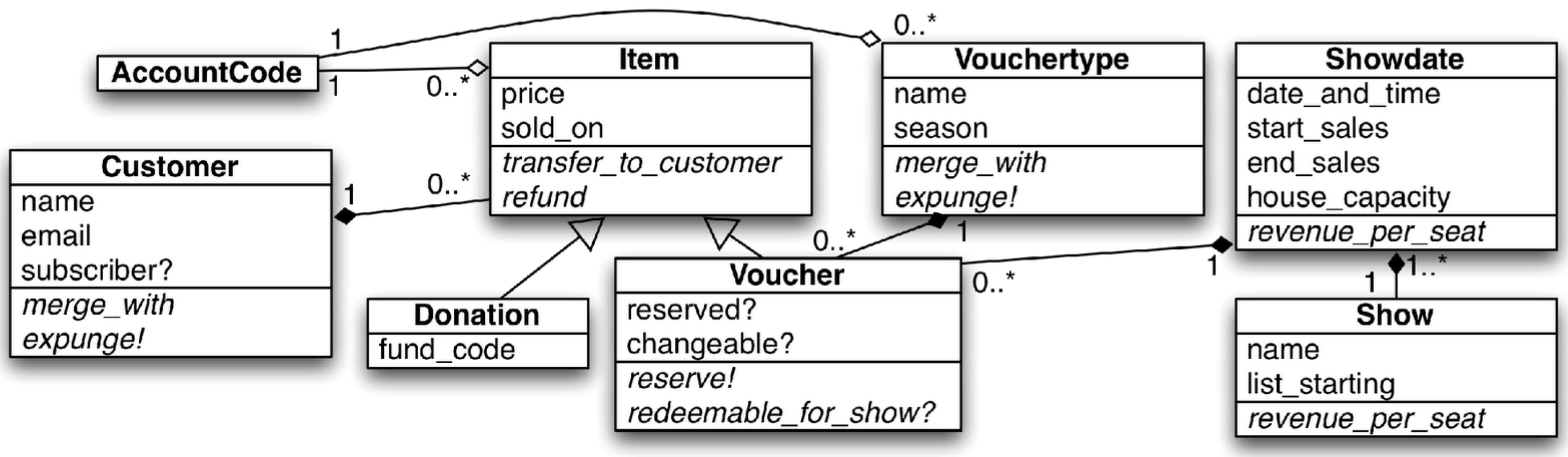


Diagram
Structure Diagram
Behavior Diagram
Class Diagram
Component Diagram
Object Diagram
Activity Diagram
Use Case Diagram
Profile Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Interaction Diagram
State Machine Diagram
Sequence Diagram
Communication Diagram
Interaction Overview Diagram
Timing Diagram
Notation: UML



Which AR relationship DOES NOT follow from this UML diagram:

☐ Show has many Vouchers, through Showdate

☐ Item belongs to Vouchertype

☐ Customer has many Donations

☐ Voucher belongs to Vouchertype

*

Single Responsibility Principle (SRP)

- A class should have *one and only one* reason to change
- Each *responsibility* is a possible *axis of change*
- Changes to one axis shouldn't affect others
- What is class's responsibility, in ≤25 words?
- Part of the craft of OO design is *defining* responsibilities and then sticking to them
- Models with many sets of behaviors
- eg a user is a moviegoer, and an authentication principal, and a social network member, ...etc.
- really big class files are a tipoff

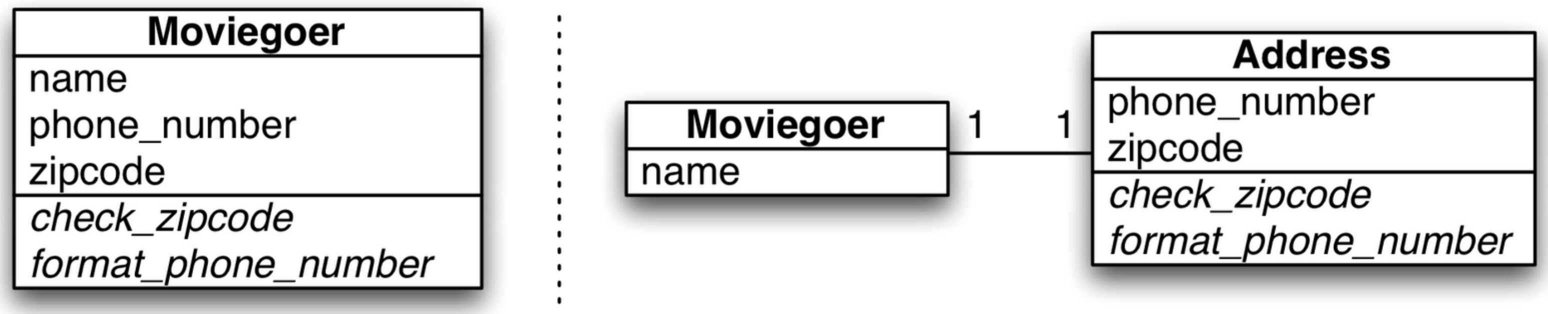
Lack of Cohesion of Methods

- Revised Henderson-Sellers LCOM=1–(sum(MV_i) / M*V) (between 0 and 1)
- M = # instance methods
- V = # instance variables
- MV_i = # instance methods that access the *i*'th instance variable (excluding "trivial" getters/setters)
- LCOM-4 counts # of connected components in graph where related methods are connected by an edge
- High LCOM suggests possible SRP violation

Do AR models violate SRP?

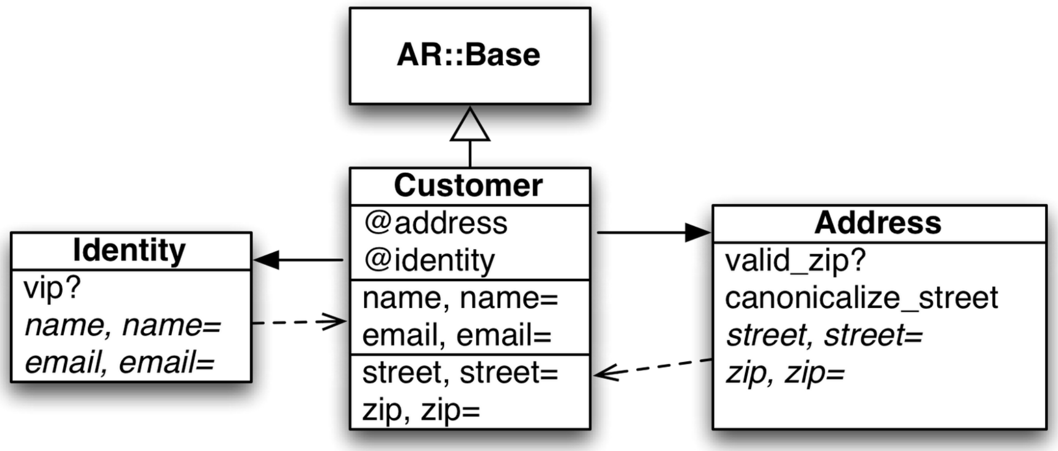
- They seem to mix behaviors in 1 class
- they know how to load, save, delete themselves
- they know about associations
- they know about validations
- all in addition to model-specific business logic
- Although most ActiveRecord behaviors are *included as modules*

Extract a module or class



- has_one or composed_of?
- Use composition & delegation?

http://pastebin.com/bjdaTWN8



http://pastebin.com/XESSSNb6

Which is true regarding the Active Record design pattern and the SRP?

- ☐ AR violates SRP, but the Rails designers are willing to pay the price for the added convenience
- ☐ AR doesn’t violate SRP, because the data storage behaviors are included as Modules
- ☐ The AR design pattern itself doesn't lead to SRP violations, but Rails' implementation does
- ☐ The AR design pattern *can* lead to SRP violations, but Rails' implementation avoids them

Open/Closed Principle *(ELLS §10.4)*

Armando Fox

© 2012 Armando Fox & David Patterson
Licensed under
Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported

Open/Closed Principle

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

```
class Report
  def output_report
    case @format
    when :html
      HtmlFormatter.new(self).output
    when :pdf
      PdfFormatter.new(self).output
```

- Can’t extend (add new report types) without changing Report base class
  - Not as bad as in statically typed language....but still ugly

Abstract Factory Pattern: DRYing out construction

•How to avoid OCP violation in Report constructor, if output type isn't known until runtime?

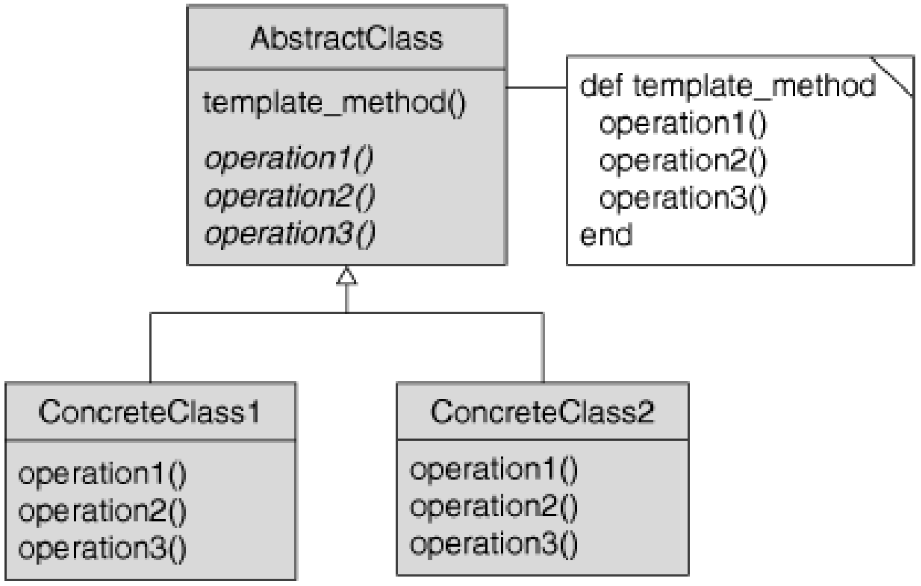
•Statically typed language: *abstract factory* pattern

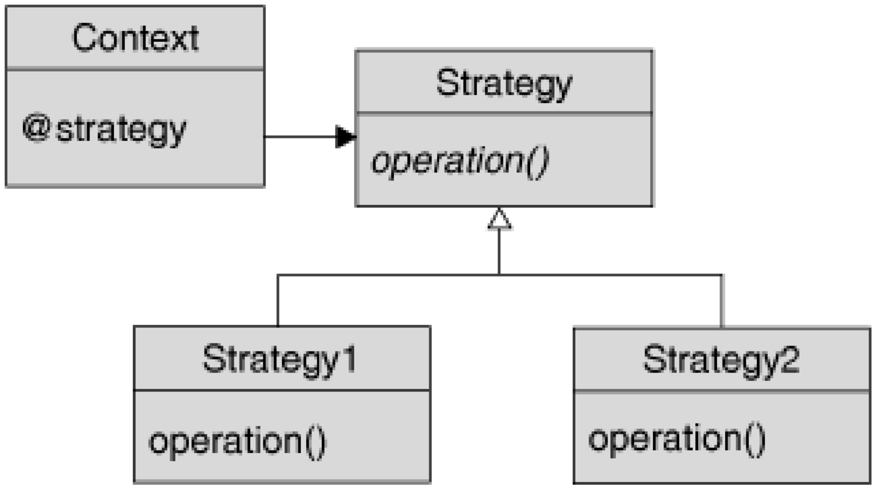
•Ruby has a particularly simple implementation of this pattern…

http://pastebin.com/p3AHMqHZ

Template Method Pattern & Strategy Pattern

- *Template method*: set of steps is the same, but implementation of steps different
- Typical implementation: inheritance, with subclasses overriding abstract methods
- *Strategy:* task is the same, but many ways to do it
- Typical implementation: composition



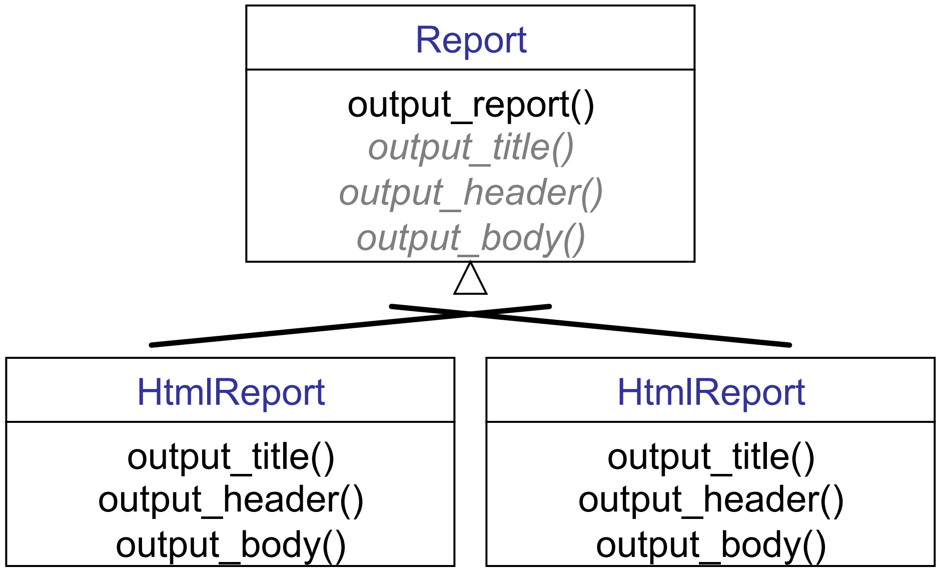


Report Generation Using Template

```
class Report
 attr_accessor :title, :text
 def output_report
  output_title
  output_header
  output_body
 end
end

class HtmlReport < Report
 def output_title ... end
 def output_header ... end
end
class PdfReport < Report
 def output_title ... end
 def output_header ... end
end
```

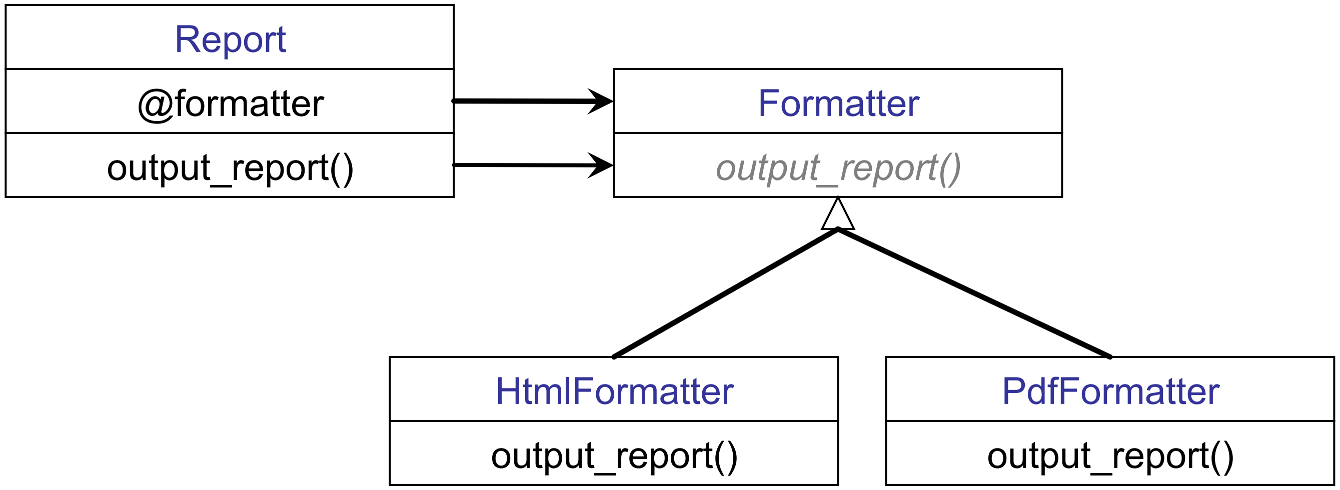
Template method stays the same;
helpers overridden in subclass



Report Generation Using Strategy

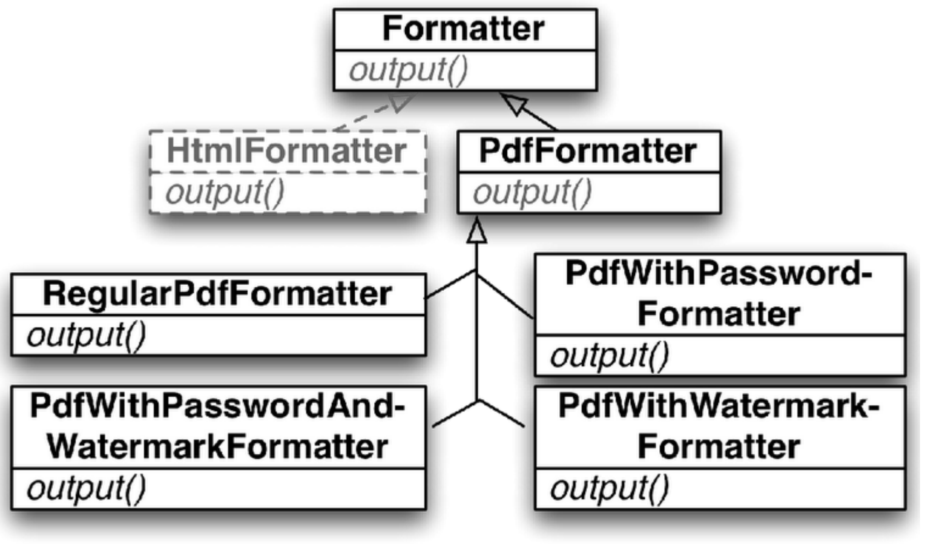
```
class Report
 attr_accessor :title, :text, :formatter
 def output_report
  @formatter.output_report
 end
end
```

Delegation
(vs. inheritance)



"Prefer composition over inheritance"

Decorator Pattern: DRYing Out Extension Points



Example in Rails: ActiveRecord scopes

Movie.for_kids.with_good_reviews(3)
Movie.with_many_fans.recently_reviewed

OCP In Practice

- Can’t close against *all* types of changes, so have to choose, and you might be wrong
- Agile methodology can help *expose important types of changes early*
  - Scenario-driven design with prioritized features
  - Short iterations
  - Test-first development
- Then you can try to close against *those types* of changes

Does the Associations functionality in Rails ActiveRecord adhere to OCP?

- ☐ Yes, because no source changes are needed in the models themselves to use Associations
- ☐ Yes, because we can change the type of relational database used by the app
- ☐ No, because we cannot add a new type of association relationship without modifying source
- ☐ No, because Associations provide functionality that's not necessarily related to the model's business logic