Liskov Substitution Principle *(ELLS §10.5)*

Liskov Substitution: Subtypes can substitute for base types

- Current formulation attributed to (Turing Award winner) Barbara Liskov

> "A method that works on an instance of *type T*, should also work on any *subtype of T*"

- Type/subtype != class/subclass

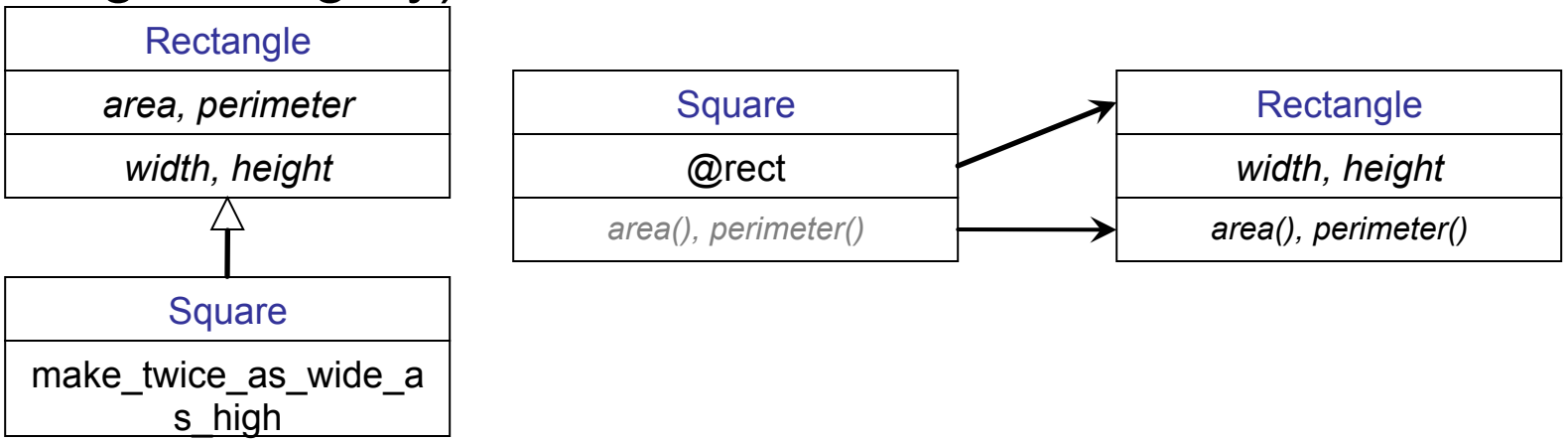
With duck typing, *substitutability* depends on how collaborators interact with object

- Let's see an example

http://pastebin.com/nf2D9RYj

Contracts

- "Prefer composition & delegation over inheritance"
- If can't express consistent assumptions about "contract" between class & collaborators, likely LSP violation
- Symptom: change to subclass requires change to superclass (shotgun surgery)



(a) In duck-typed languages, LSP violations can occur even when inheritance is not used
(b) In statically-typed languages, if the compiler reports no type errors/warnings, then there are no LSP violations

- ☐ Only (a) is true
- ☐ Only (b) is true
- ☐ Both (a) and (b) are true
- ☐ Neither (a) nor (b) is true

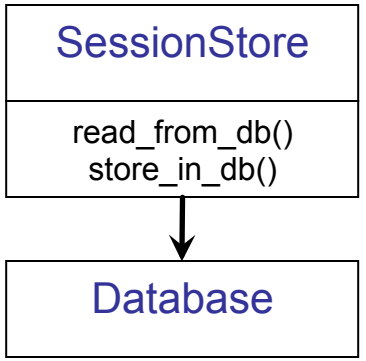
http://pastebin.com/rLguzt8X

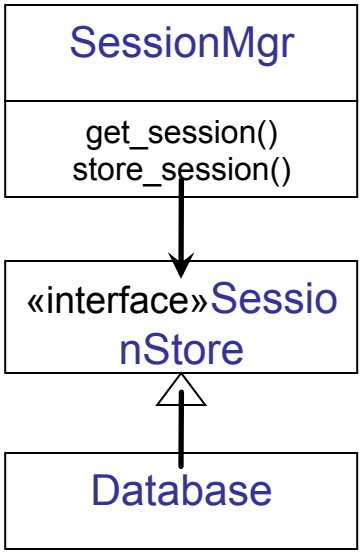
*

Dependency Injection *(ELLS §10.6)*

Dependency Inversion & Dependency Injection

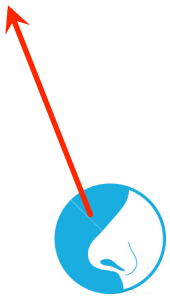
- Problem: *a* depends on *b*, but *b* interface & implementation can change, even if *functionality* stable
- Solution: "inject" an *abstract interface* that *a* & *b* depend on
- If not exact match, Adapter/Façade
- "inversion": now b (and a) depend on interface, vs. a depending on b
- Ruby equivalent: Extract Module to isolate the interface

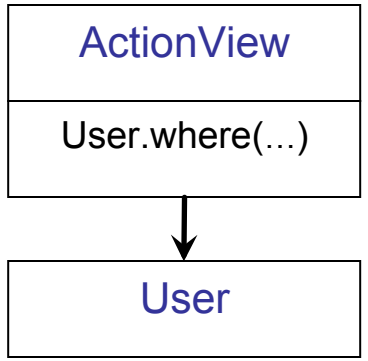




DIP in Rails: example

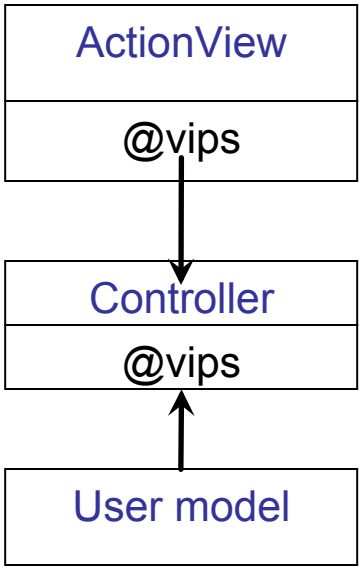
- What's wrong with this in a *view*:
  - @vips = User.where('group="VIP"')





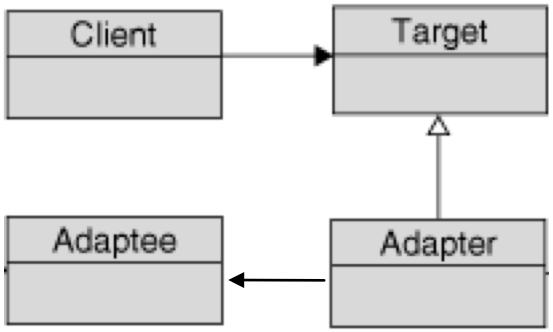
- A little better:
  - @vips = User.find_vips
- Happiness:

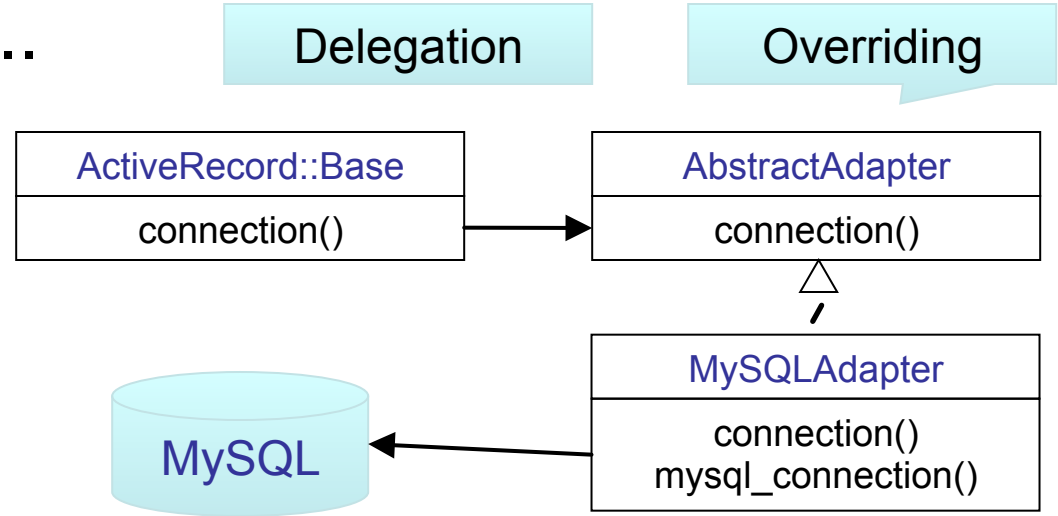
```
# in controller
@vips = User.find_vips
```



Injecting Dependencies with the Adapter pattern

- Problem: client wants to use a "service"...
- service generally supports desired operations
- but the API's don't match what client expects
- and/or client must interoperate transparently with multiple slightly-different services
- Rails example: database "adapters" for MySQL, Oracle, PostgreSQL, ...

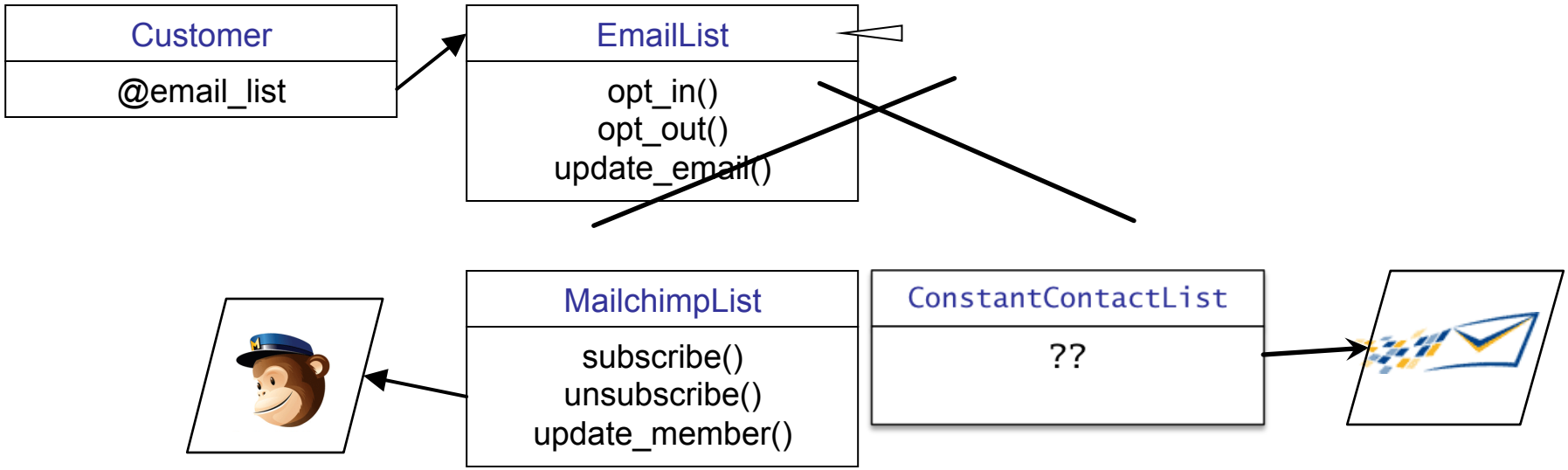




Example: Supporting External Services

- I use external services for email marketing
- Both have RESTful API's
- Similar features

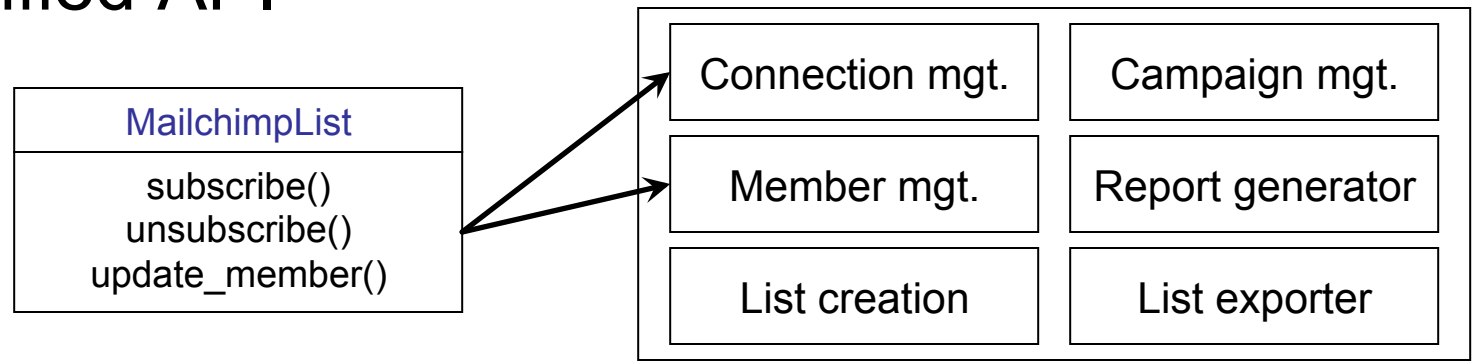
- Maintain multiple lists, add/remove user(s) from list(s), change user's opt-in status, ...



* Disclaimer: logos © or ™ their respective holders. This example is for education only and doesn't imply endorsement, etc.—duh.

Related: Façade

- In fact, we only use a *subset* of much more elaborate API's
- Initialization, list management, start/stop campaign...
- So our adapter is also a *façade*
- may *unify* distinct underlying API's into a single, simplified API

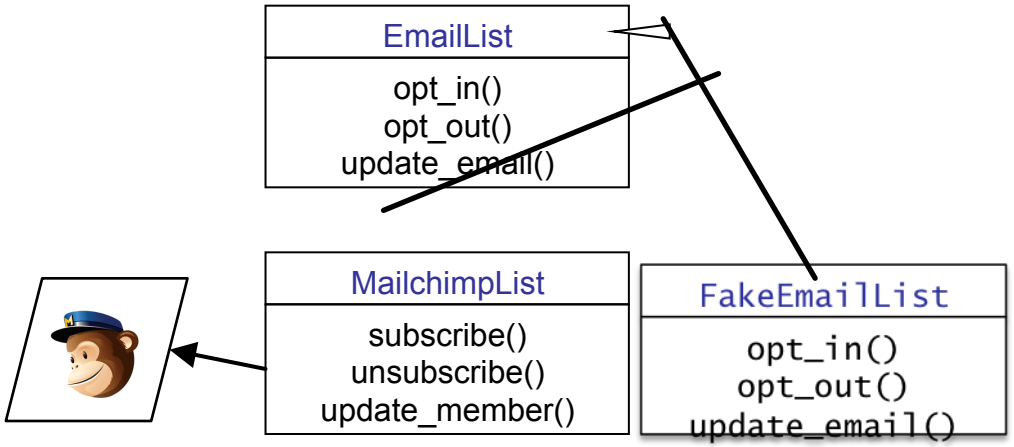


Related: Null object

- Problem: want *invariants* to simplify design, but app requirements seem to break this
- *Null object:* stand-in on which "important" methods can be called

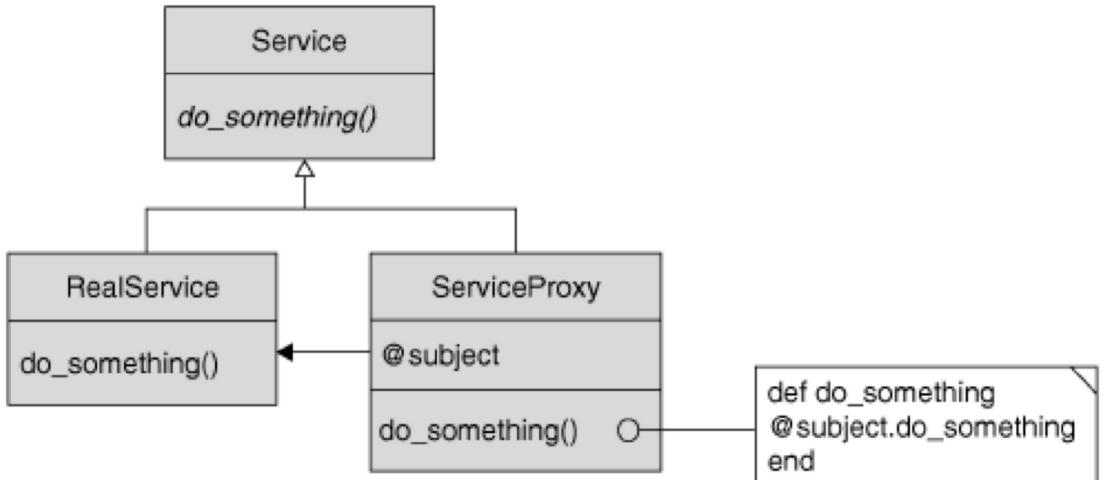
```
@customer = Customer.null_customer
@customer.logged_in?  # => false
@customer.last_name   # => "ANONYMOUS"
@customer.is_vip?     # => false
```

http://pastebin.com/RBuvPMkR



Related: Proxy

- Proxy implements *same methods* as "real" service object, but "intercepts" each call
- do authentication/protect access
- hide remote-ness of a service
- defer work (be lazy)
- Rails example: association proxies (eg Movie.reviews)



In RSpec controller tests, it's common to stub ActiveRecord::Base.find, an inherited method. Which statements are true of such tests:

- ☐ The controller under test is tightly coupled to the model
- ☐ In a static language, we'd have to use DI to achieve the same task in the testing framework.
- ☐ Both of the above are true
- ☐ Neither the first nor the second statement is true

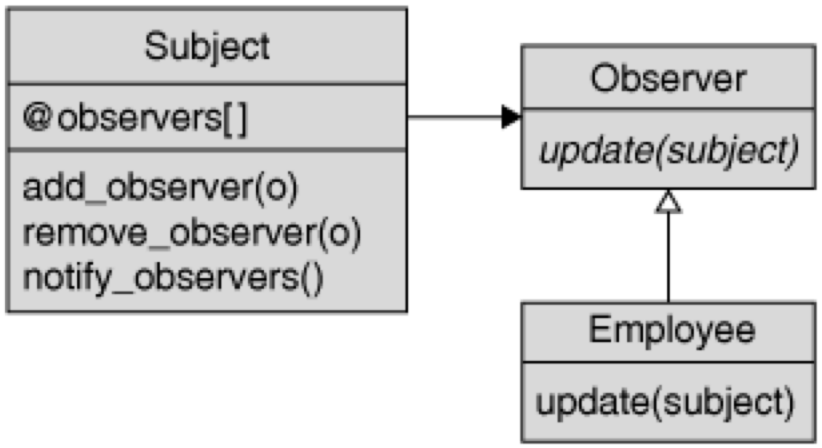
Demeter Principle + Example

- 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* them.
- Solutions:
- replace method with delegate
- Separate traversal from computation (Visitor)
- Be aware of important events without knowing implementation details (Observer)

http://pastebin.com/NRSkHstN

Observer

- Problem: entity O ("observer") wants to know when certain things happen to entity S ("subject")
- Design issues
- acting on events is O's concern—don't want to pollute S
- any type of object could be an observer or subject—inheritance is awkward
- Example use cases
  - full-text indexer wants to know about new post (e.g. eBay, Craigslist)
  - auditor wants to know whenever "sensitive" actions are performed by an admin



Example: Maintaining Relational Integrity

- Problem: delete a customer who "owns" previous transactions (i.e., foreign keys point to her)
- My solution: merge with "the unknown customer"
- ActiveRecord provides built-in hooks for Observer design pattern

```
class CustomerObserver < ActiveRecord::Observer
  observe :customer # actually not needed (convention)
  def before_destroy ... end
End

# in config/environment.rb
config.active_record.observers = :customer_observer
```

Suppose Order belongs to Customer, and view has `@order.customer.name`
… is this a Demeter violation?

- Yes...but probably reasonable to just expose object graph in the view in this case
- Yes...replace with Order#customer_name which delegates to Customer#name
- You can make a case for either of the above
- No…by using belongs_to we're already exposing info about the Customer anyway

A Few Patterns Seen in Rails

- Adapter (database connection)
- Abstract Factory (database connection)
- Observer (caching—Chapter 12)
- Proxy (AR association collections)
- Singleton (Inflector)
- Decorator (AR scopes, alias_method_chain)
- Command (migrations)
- Iterator (everywhere)
- Duck typing simplifies expressing and "plumbing" most of these by "weakening" the relative coupling of inheritance

SOLID Caveat

- Designed for statically typed languages, so some principles have more impact there
- "avoid changes that modify type signature" (often implies contract change)—but Ruby doesn't really use types
- "avoid changes that require gratuitous recompiling"—but Ruby isn't compiled
- Use judgment: goal is *deliver working & maintainable code quickly*

Summary

- Design patterns represent *successful solutions* to classes of problems
- Reuse of design rather than code/classes
- A few patterns “reified” in Rails since useful to SaaS
- 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
- Much more to read & know—this is just an intro

Rails' ActiveRecord module defines an AbstractAdapter for connecting to databases. Subclasses of AbstractAdapter exist for each database type and can be added for new databases; when the app starts, the correct one is instantiated based on **config/database.yml**.
Which SOLID principle is NOT illustrated by this example:

- Open/Closed Principle
- Dependency Injection
- Demeter Principle
- Liskov Substitution