Design Patterns (III)

Chung-Kil Hur

(Credit: Byung-Gon Chun & Many Slides from UCB CS169 taught by Armando Fox, David Patterson)

SWPP, CSE, SNU

SOLID OOP principles

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

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

Single Responsibility Principle (SRP)

- A class should have *one and only one* reason to change
- What is class's responsibility, in ≤25 words?
  - Part of the craft of OO design is *defining* responsibilities and then sticking to them
- Quantification: lack of cohesion of methods

Open/Closed Principle

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

```
public class Report {
  public void output(ReportData data) {
    switch (format) {
    case HTML:
      new HtmlFormatter().output(data)
    case PDF:
      new PdfFormatter().output(data)
    default: // no op
    }
  }
}
```

- Can't extend (add new report types) without changing Report base class

Design Patterns for OCP

- (Abstract) Factory
- Template: set of steps is the same, but implementation of steps different (Inheritance: subclasses override abstract "step" methods)
- Strategy: task is the same, but many ways to do it (composition: component classes implement whole task)
- Decorator: decorate a class or method by wrapping it in an enhanced version that has the same API, allowing us to compose multiple decorations as needed

Liskov Substitution Principle

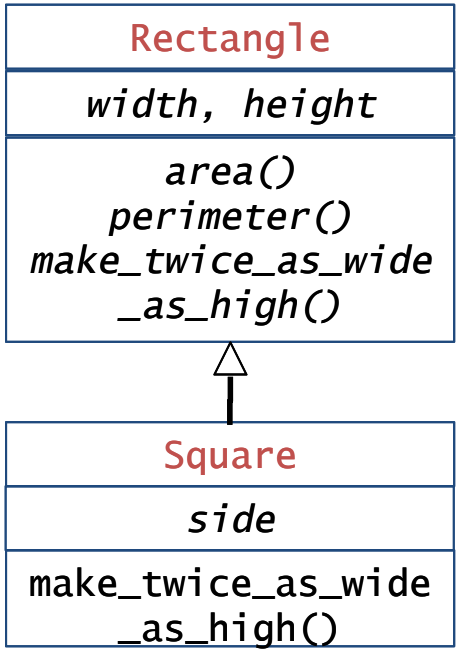
Liskov Substitution: Subtypes can substitute for base types

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

- Type/subtype ____ class/subclass

Contracts

- Composition vs. (misuse of) inheritance
- If can't express consistent assumptions about "contract" between class & collaborators, likely LSP violation

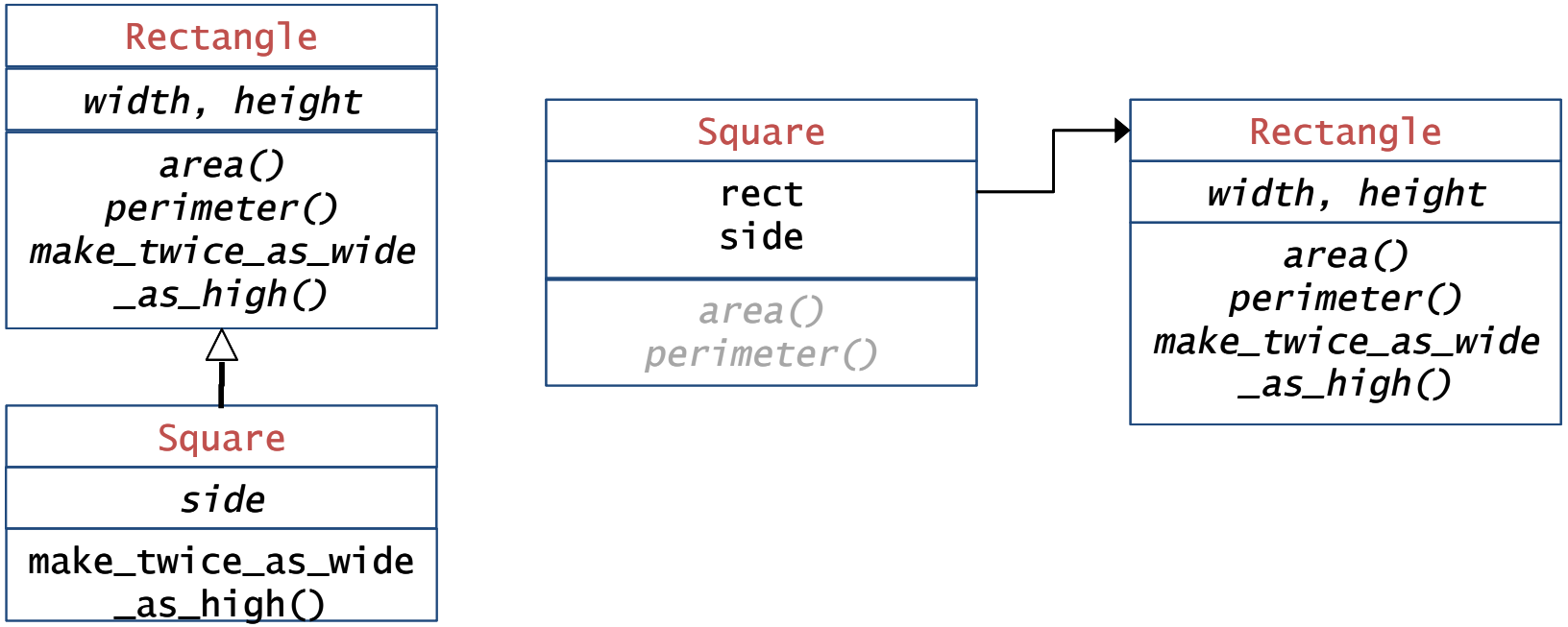


Symptoms

- Subclass destructively overrides a behavior inherited from the superclass
  - A refused bequest – a design smell that often indicates an LSP violation
  - Inheritance is all about implementation sharing; if a subclass won't take advantage of its parent's implementations, it might not deserve to be a subclass at all
- Forces changes to the superclass to avoid the problem (OCP violation)

LSP-Compliant Code

- Composition of classes rather than inheritance, achieving **reuse through delegation** rather than through subclassing



In statically-typed languages, if the compiler reports no type errors/warnings, then there are no LSP violations

- ☐ True
- ☐ False

Dependency Injection

Dependency Inversion & Dependency Injection

- Problem: *a* depends on *b,* but *b* implementation can change, even if *functionality* stable
- Solution: “inject” an *abstract interface* that *a* & *b* depend on
  - If not exact match, _____ pattern
  - “inversion”: now b (and a) depend on interface, vs. a depending on b

Example: Email Marketing

```
public class EmailList {
  MailerMonkey mailer;
  public EmailList() {
    mailer = new MailerMonkey();
  }
  public void sendEmail(Moviegoers mg) {
    mailer.sendEmail(mg);
  }
}
public class EmailListController {
  // ...
  public void advertiseDiscountForMovie(Moviegoers mg) {
    new EmailList().sendEmail(mg)
  }
}
```

Example: Email Marketing

- 1st problem: Moviegoers who are on the YourSpace social network can opt to have these emails forwarded to their YourSpace friends.

Dependency Injection Principle (DIP)

- Injecting an additional seam between two classes

- In statically compiled languages the DIP helps with testability

```
public class EmailList {
  private final GenericMailer mailer;
  public EmailList(GenericMailer mailer) {
    this.mailer = mailer;
  }
  public void sendEmail(Moviegoers mg) {
    mailer.sendEmail(mg);
  }
}
public class EmailListController {
  // ...
  public void advertiseDiscountForMovie(Moviegoers mg) {
    GenericMailer mailer = Config.hasYourSpace ? new YourSpace() :
        new MailerMonkey();
    new EmailList(mailer).sendEmail(mg);
  }
}
```

Example: Email Marketing

- 2nd problem: Your space exposes a different and more complex API than the simple sendEmail method provided by MailerMonkey
- But our EmailListController is already set up to call sendEmail on the mailer object

Adapter Pattern (#5)

```
public class YourSpaceAdapter implements GenericMailer {
  private final YourSpace ys = new YourSpace();
  public void sendEmail() {
    ys.authenticate(...);
    ys.sendMessage(...);
  }
}
public class EmailList {
  private final GenericMailer mailer;
  public EmailList(GenericMailer mailer) {
    this.mailer = mailer;
  }
  public void sendEmail(Moviegoers mg) {
    mailer.sendEmail(mg);
  }
}
public class EmailListController {
  // ...
  public void advertiseDiscountForMovie(Moviegoers mg) {
    GenericMailer mailer = Config.hasYourSpace() ? new YourSpaceAdapter() :
new MailerMonkey();
    new EmailList(mailer).sendEmail(mg);
  }
}
```

Adapter pattern:

convert an existing API into one that's compatible with an existing caller

Façade Pattern (#6)

- When the Adapter pattern not only converts an existing API but also simplifies it
- E.g., YourSpace provides many other YourSpace functions unrelated to email, but YourSpaceAdapter only adapts the email-specific part of that API, it's sometimes called the Façade pattern.

Example: Email Marketing

- 3rd problem: what if we want to disable email sending altogether from time to time

What if We Want to Disable Email Sending Altogether From Time To Time

- Naïve approach: move the logic for determining which emailer to use into a new Config class, but we still have to condition out the email-sending logic in the controller method if email is disabled

```
public class Config {
  public boolean isEmailEnabled() { ... };
  public GenericMailer emailer() { ... };
}
public class EmailListController {
  public void advertiseDiscountForMovie(Moviegoers mg) {
    if (Config.isEmailEnabled())
      new EmailList(Config.emailer()).sendEmail(mg);
  }
}
```

What if There Are Other Places Where a Similar Check Must Be Performed?

- The same condition logic would have to be replicated there (shotgun surgery)

Null Object Pattern (#7)

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

Null Object Pattern (#7)

```
public class Config {
  public GenericMailer emailer() {
    if (emailDisabled)
      return new NullMailer();
    else ...
  }
}
public class NullMailer {
  public void sendEmail(Moviegoers mg) { }
}
public class EmailListController {
  public void advertiseDiscountForMovie(Moviegoers mg) {
    new EmailList(Config.emailer()).sendEmail(mg);
  }
}
```

NullObject pattern

Proxy Pattern (#8)

- Interesting relative of the Adapter and Façade patterns
- One object “stands in” for another that has the same API
- Proxy implements *same methods* as “real” service object, but “intercepts” each call
  - The client talks to the proxy instead of the original object
  - The proxy may forward some requests directly to the original object but may take other actions on different requests
    - Caching for performance
    - Sending emails while disconnected from the Internet

Class Diagrams: Without or With DI

Singleton: Ensure there's only one of something

- Technically, a class that provides only 1 instance, which anyone can access

The use of FakeWeb to stub external SOA requests in testing is an example of which design pattern?

- ☐ Null object
- ☐ Proxy
- ☐ Adapter
- ☐ Façade