Object-Oriented Design

You know the software is rotting when it starts to exhibit any of the following odors

- **Rigidity** – The system is hard to change because every change forces many other changes to other parts of the system.
- **Fragility** – Changes cause the system to break in places that have no conceptual relationship to the part that was changed.
- **Immobility** – It is hard to disentangle the system into components that can be reused in the other systems.
- **Viscosity** – Doing things right is harder than doing things wrong.
- **Needless Complexity** – The design contains infrastructure that adds no direct benefit.
- **Needless Repetition** – The design contains repeating structures that could be unified under a single abstraction.
- **Opacity** – It is hard to read and understand. It does not express its intent well.

Agile Software Development: Principles, Patterns, and Practices ch7 p88

Simple Design

- Extreme Programming's principle of *Simple Design*, in order of importance:
  1. Runs all the tests
  2. Contains no duplication
  3. Expresses the intent of the programmer
  4. Minimizes the number of classes and methods
- Truly agile teams don't allow the software to rot.
  - Keeping the design clean and simple *at all times* is the only way to go fast. It makes the design flexible and easy to change.
  - Making a mess will always slow you down the next time that you need to read or change the same code.

Clean Code ch12
Extreme Programming Explained (1st Ed.) ch10 p57
http://xprogramming.com/xpmag/expEmergentDesign
Agile Software Development: Principles, Patterns, and Practices ch7 p90

Architecture

> "Architecture is about the important stuff. Whatever that is."

> "Architecture is the decisions that you wish you could get right early in a project." Why do people feel the need to get some things right early in the project? The answer, of course, is because they perceive those things as hard to change. So you might end up defining architecture as "things that people perceive as hard to change."

- Big Design Up Front (BDUF), i.e. spinning system designs based on untested hypotheses for many months, is harmful. BDUF inhibits adapting to change.
- Little/Enough Design Up Front (LDUF/EDUF) is good. With a big project spending a week or even a month thinking about the important things is nothing wrong.
- Agile was a response to BDUF, but not DUF. Far from "design nothing," the XP strategy is "design always."
- Whatever designs have been done up front, they should always be open for change. Let the tests drive the system architecture.

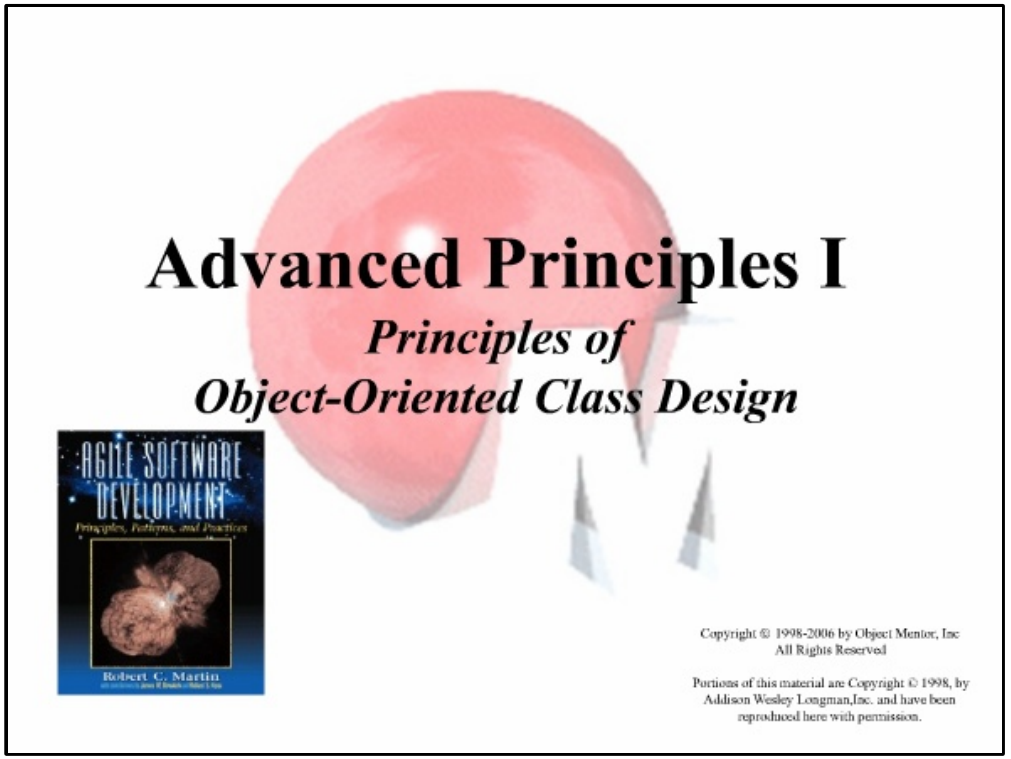
http://www.ibm.com/developerworks/java/library/j-eaed1/
http://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf
Clean Code ch11 p166
http://blog.objectmentor.com/articles/2009/04/25/the-scatology-of-agile-architecture

Principles of Modular Design

- **Cohesion** – higher is better
  - Cohesion is a measure of how strongly-related or focused the responsibilities of a single module are.
- **Coupling** – lower is better
  - Coupling or dependency is the degree to which each program module relies on each one of the other modules.

http://en.wikipedia.org/wiki/Cohesion_(computer_science)
http://en.wikipedia.org/wiki/Coupling_(computer_science)
Clean Code ch10 p140
http://reborg.tumblr.com/post/82362851/clean-code-cheat-sheet

Principles of Object-Oriented Design



http://www.infoq.com/presentations/principles-agile-oo-design (30 min)

SOLID

Software development is not a Jenga game.

http://www.lostechies.com/blogs/derickbailey/archive/2009/02/11/solid-development-principles-in-motivational-pictures.aspx
http://www.globalnerdy.com/2009/07/15/the-solid-principles-explained-with-motivational-posters/

SOLID Principles

- SRP: Single Responsibility Principle
- OCP: Open Closed Principle
- LSP: Liskov Substitution Principle
- ISP: Interface Segregation Principle
- DIP: Dependency Inversion Principle

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
http://www.hanselminutes.com/default.aspx?showID=163
http://blog.objectmentor.com/articles/2009/02/12/getting-a-solid-start
http://www.lostechies.com/blogs/chad_myers/archive/2008/03/07/pablo-s-topic-of-the-month-march-solid-principles.aspx
http://www.lostechies.com/content/pablo_ebook.aspx

Single Responsibility Principle

Just because you *can* doesn't mean you *should*.

http://www.lostechies.com/blogs/derickbailey/archive/2009/02/11/solid-development-principles-in-motivational-pictures.aspx
http://www.globalnerdy.com/2009/07/15/the-solid-principles-explained-with-motivational-posters/

Single Responsibility Principle

A class should have only one reason to change.

- Each responsibility should be separated to its own class, because each responsibility is an axis of change. Change to one responsibility may break others in the same class.
- Following the SRP leads to small, highly focused, cohesive classes, which each do only one thing. Each of them will be easy understand and change.
  - Example: In a payroll application, it's wrong to have an Employee class which (1) calculates pay, (2) produces a report, and (3) saves itself to database. Each of them should be separated into its own class: EmployeePayCalculator, EmployeePayrollReport, EmployeeRepository.

http://www.objectmentor.com/resources/articles/srp.pdf
Agile Software Development: Principles, Patterns, and Practices ch8 p95
Clean Code ch10 p128

http://www.lostechies.com/blogs/derickbailey/archive/2009/02/11/solid-development-principles-in-motivational-pictures.aspx
http://www.globalnerdy.com/2009/07/15/the-solid-principles-explained-with-motivational-posters/

Open Closed Principle

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

- Make classes depend on abstract interfaces instead of concrete classes. Inject new behaviour into existing classes by providing a different implementation of the interface.
- When the OCP is applied well, new behaviour can be added by creating a new class, without needing to change existing classes. Breaking existing code will be avoided.
  - Realistically will not always be possible, but we can try to approximate it by anticipating what will change.

http://www.objectmentor.com/resources/articles/ocp.pdf
Agile Software Development: Principles, Patterns, and Practices ch9 p99

Anticipating Future Changes

- "Since closure cannot be complete, it must be strategic. That is, the designer must **choose the kinds of changes** against which to close his design. He must guess at the most likely kinds of changes, and then construct abstractions to protect him from those changes."
- "Also, conforming to the OCP is *expensive.* It takes **development time and effort** to create the appropriate abstractions. Those abstractions also **increase the complexity** of the software design."

Agile Software Development: Principles, Patterns, and Practices ch9 p105

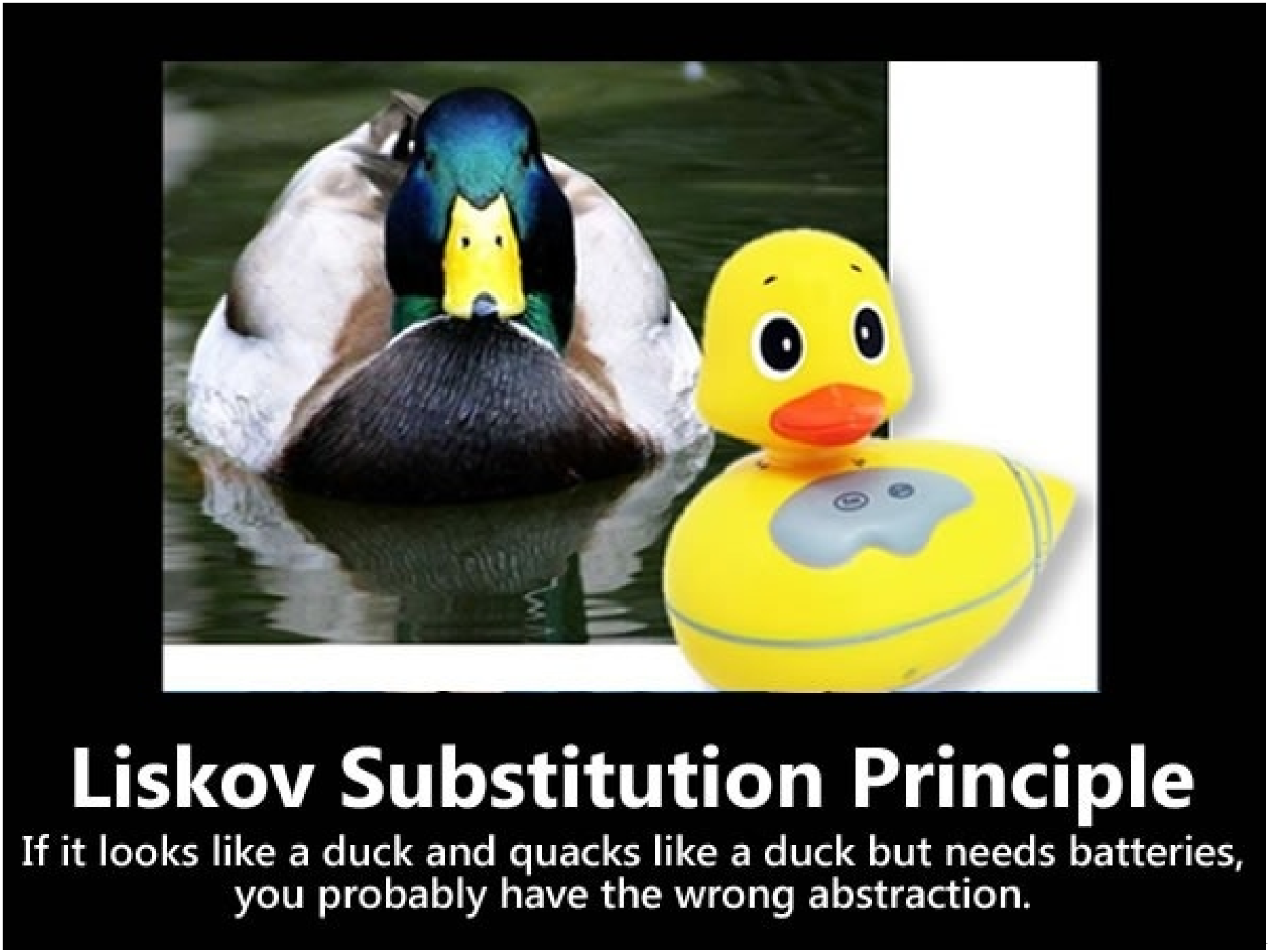
Anticipating Future Changes

- Do **not** put hooks in for changes that *might* happen. Instead, *wait until the changes happen!*

  "Fool me once, shame on you. Fool me twice, shame on me."

- Initially write the code expecting it to not change. When a change occurs, implement the abstractions that protect from future changes *of that kind.*
- It's better to take the first hit as early as possible. We want to know what kind of changes are likely before going too far in the development.
  - Use TDD and listen to the tests. Develop in short cycles. Develop features before infrastructure. Develop the most important features first. Release early and often.

Agile Software Development: Principles, Patterns, and Practices ch9 p105



http://www.lostechies.com/blogs/derickbailey/archive/2009/02/11/solid-development-principles-in-motivational-pictures.aspx
http://www.globalnerdy.com/2009/07/15/the-solid-principles-explained-with-motivational-posters/

Liskov Substitution Principle

Subtypes must be substitutable for their base types. / Functions that use references to base classes must be able to use objects of derived classes without knowing it.

- The rule which must be followed when inheriting from another class/interface.
- All assumptions that functions make about the base class, must be valid for all derived classes. If there may exist code which breaks when it is passed a subclass, then the LSP has been violated.
  - Example: Given a class Rectangle, should a class Square inherit from it? After all, square *IS-A* rectangle?

http://www.objectmentor.com/resources/articles/lsp.pdf
Agile Software Development: Principles, Patterns, and Practices ch10 p111

```
public class Rectangle {
    private int width;
    private int height;

    public int getWidth() {
        return width;
    }
    public void setWidth(int width) {
        this.width = width;
    }
    public int getHeight() {
        return height;
    }
    public void setHeight(int height) {
        this.height = height;
    }
}
```

```
public class Square extends Rectangle {

    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    public void setHeight(int height) {
        super.setWidth(height);
        super.setHeight(height);
    }
}
```

The Problem

```
public void g(Rectangle r) {
    r.setWidth(5);
    r.setHeight(4);
    assert r.area() == 20;
}
```

- "The *behaviour* of a `Square` object is not consistent with g's expectation of the behaviour of a `Rectangle` object. Behaviorally, a `Square` is **not** a `Rectangle`, and it is *behaviour* that software is really all about."
- "The LSP makes it clear that in OOD, the IS-A relationship pertains to *behaviour* that can be reasonably assumed and that clients depend on."

Agile Software Development: Principles, Patterns, and Practices ch10 p116

Interface Segregation Principle

You want me to plug this in *where?*

http://www.lostechies.com/blogs/derickbailey/archive/2009/02/11/solid-development-principles-in-motivational-pictures.aspx
http://www.globalnerdy.com/2009/07/15/the-solid-principles-explained-with-motivational-posters/

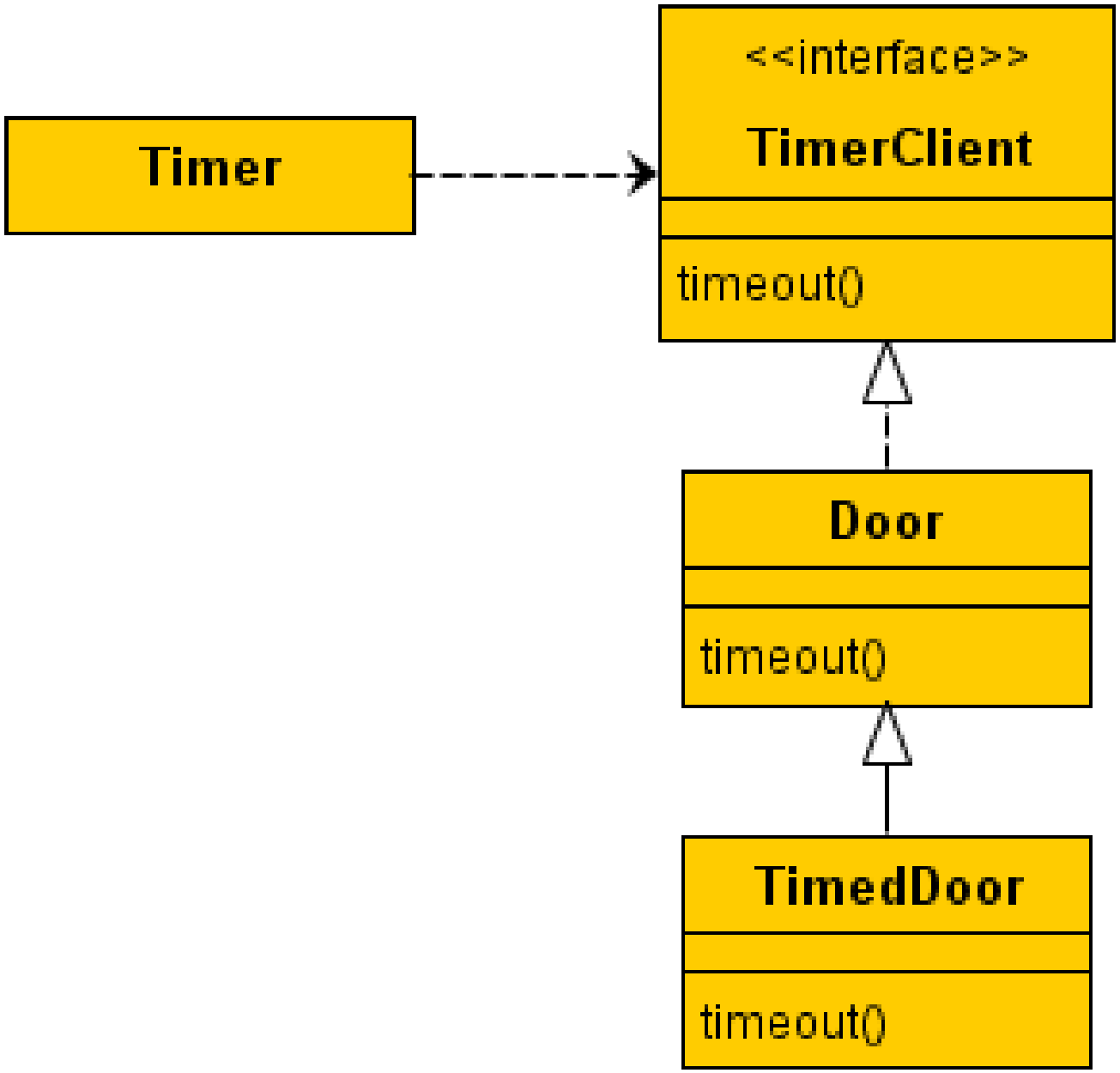
Interface Segregation Principle

Clients should not be forced to depend upon interfaces that they do not use.

- Classes should not have "fat" non-cohesive interfaces. The interfaces should be broken into groups of methods, so that each group serves a different set of clients.
- When one set of clients forces new requirements and changes to the interface, we do not want unrelated clients to be affected by those changes. Following the ISP isolates different groups of clients from one another.

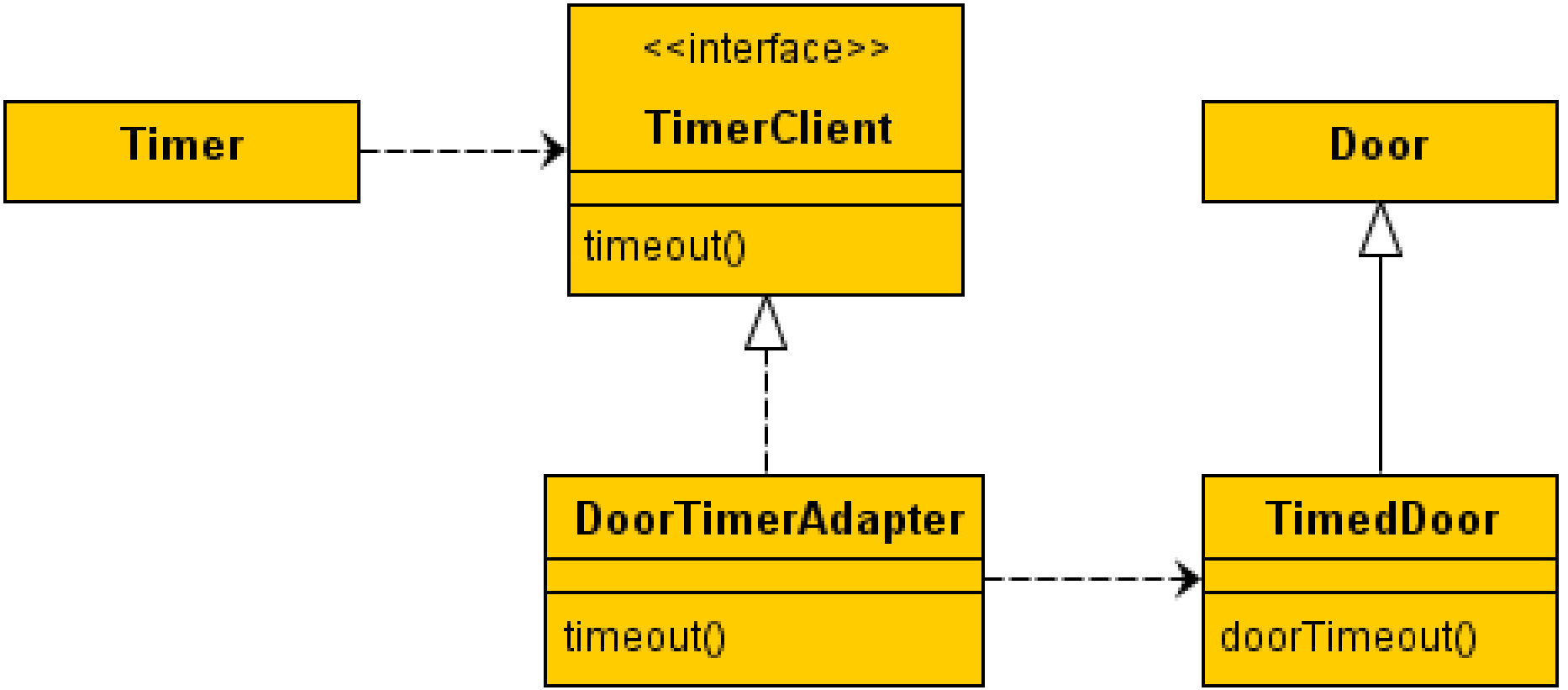
http://www.objectmentor.com/resources/articles/isp.pdf

Agile Software Development: Principles, Patterns, and Practices ch12 p135



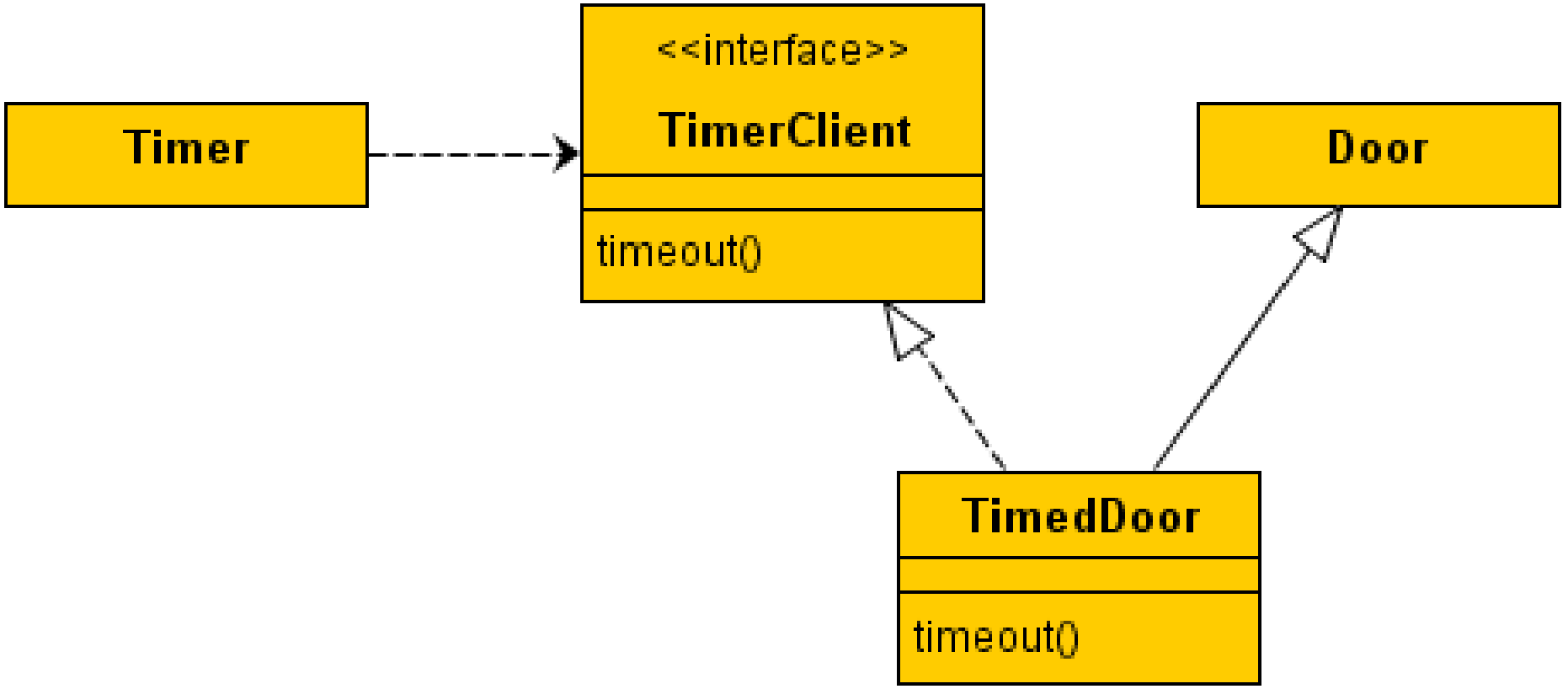
Agile Software Development: Principles, Patterns, and Practices ch12 p136

Separation through delegation



Agile Software Development: Principles, Patterns, and Practices ch12 p138

Separation through multiple inheritance



Agile Software Development: Principles, Patterns, and Practices ch12 p139

Dependency Inversion Principle

Would you solder a lamp directly
to the electrical wiring in a wall?

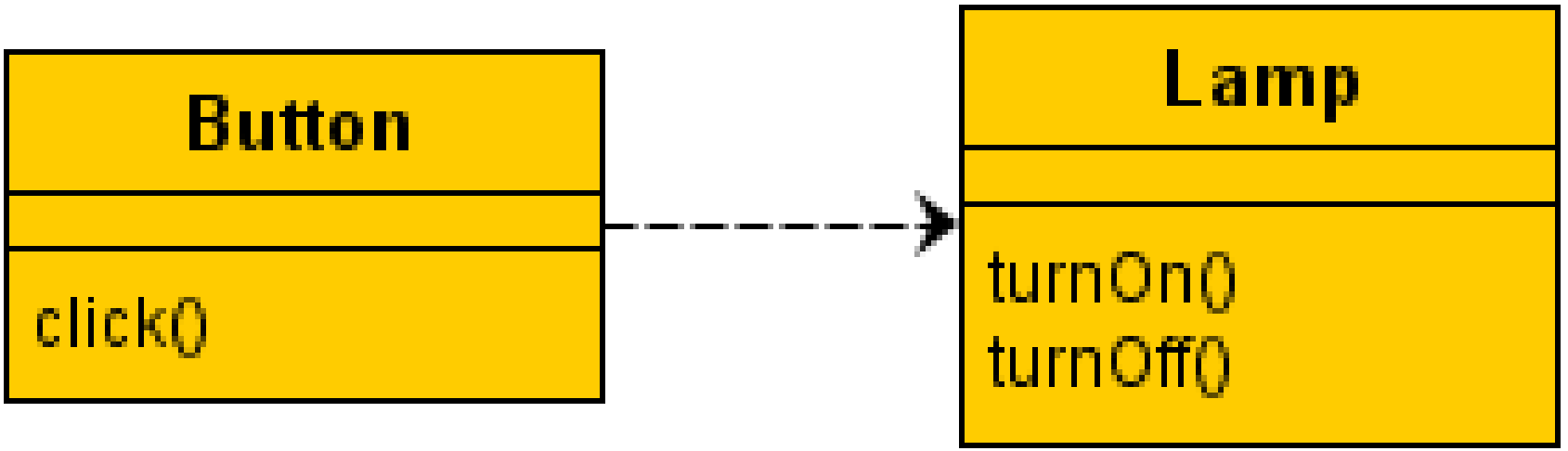
http://www.lostechies.com/blogs/derickbailey/archive/2009/02/11/solid-development-principles-in-motivational-pictures.aspx
http://www.globalnerdy.com/2009/07/15/the-solid-principles-explained-with-motivational-posters/

Dependency Inversion Principle

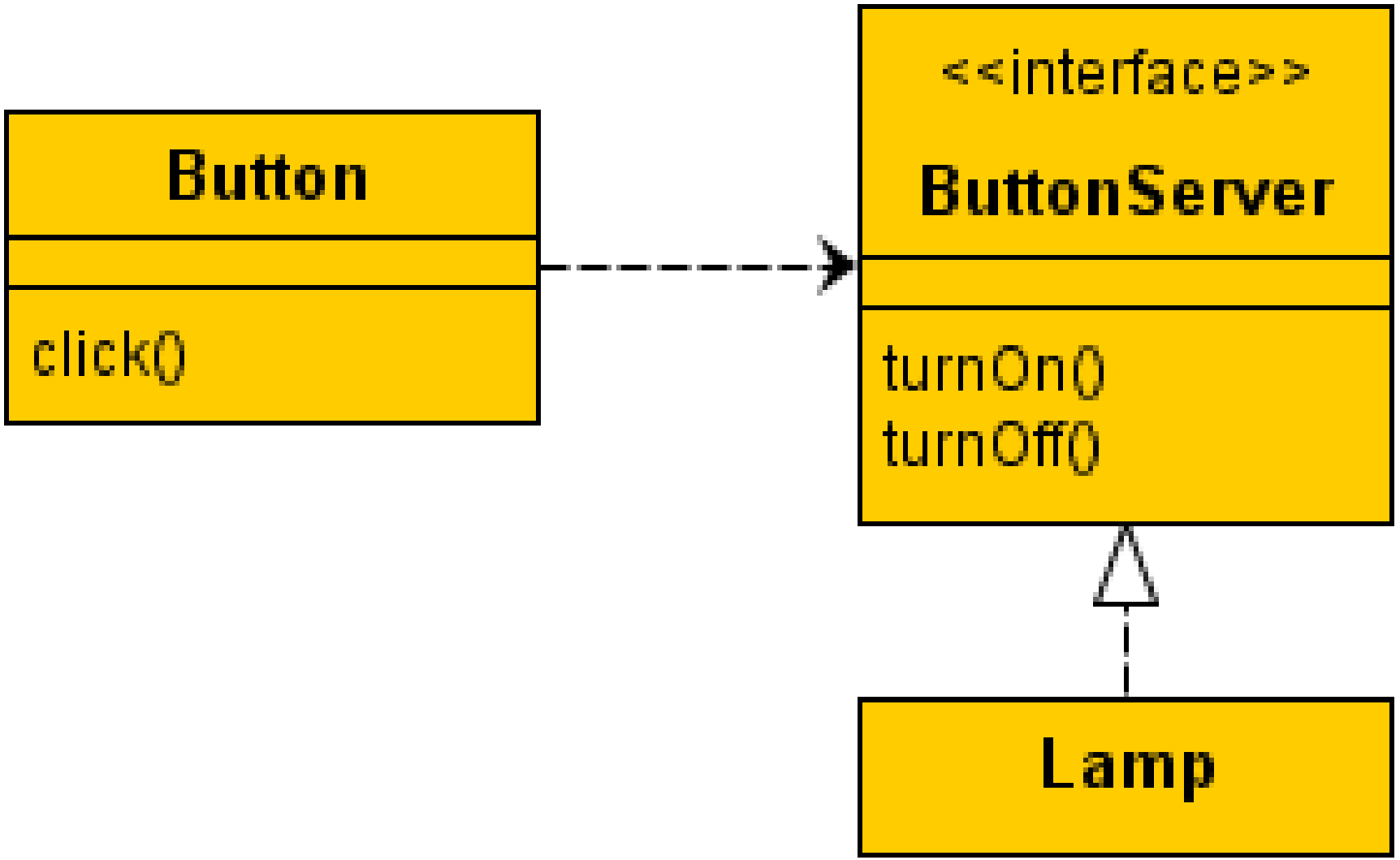
A. *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
B. *Abstractions should not depend on details. Details should depend on abstractions.*

- When high-level modules depend on the lower level modules, *changes to the lower level modules can force the higher level modules to change.* It is the high-level, policy-setting modules that ought to be influencing the low-level, detailed modules, not the other way round!
- It is the high-level, policy-setting modules that we want to be able to reuse. We are already quite good at reusing low-level modules as libraries. When high-level modules depend on low-level modules, *it becomes very difficult to reuse those high-level modules in different contexts.*

http://www.objectmentor.com/resources/articles/dip.pdf
Agile Software Development: Principles, Patterns, and Practices ch11 p127



Agile Software Development: Principles, Patterns, and Practices ch11 p130



Agile Software Development: Principles, Patterns, and Practices ch11 p131

Course Material

- *Clean Code* chapter 10: Classes
- *Clean Code* chapter 11: Systems
- *Clean Code* chapter 12: Emergence
- http://www.ibm.com/developerworks/java/library/j-eaed1/
- http://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf
- http://blog.objectmentor.com/articles/2009/04/25/the-scatology-of-agile-architecture
- http://en.wikipedia.org/wiki/Cohesion_(computer_science)
- http://en.wikipedia.org/wiki/Coupling_(computer_science)
- http://reborg.tumblr.com/post/82362851/clean-code-cheat-sheet
- http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod (all linked articles)
- http://www.infoq.com/presentations/principles-agile-oo-design
- http://www.hanselminutes.com/default.aspx?showID=163
- http://blog.objectmentor.com/articles/2009/02/12/getting-a-solid-start