Slides by G. Kulczycki

Component-Level Design

Based (somewhat) on chapter 14 of Software Engineering: A Practitioner's Approach by Roger Pressman and Bruce Maxim

Component Design Principles

Design Principle and Design Patterns
by Robert C. Martin
www.objectmentor.com

What is a Component?

- A modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces

– OMG UML Specificaiton

Component Views

- OO View – A component is a set of collaborating classes.
- Conventional View – A component is a functional element of a program that incorporates processing logic, the internal data structures required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.

What Goes Wrong with Software?

- Why does software deteriorate?
- How does software deteriorate?
- Can we prevent (or slow) the deterioration?

Symptoms of Rotting Design

- Rigidity – tendency for software to be difficult to change
- Fragility – tendency for software to break when it is changed
- Immobility – inability to reuse software from other projects
- Viscosity – easier to hack software than to keep design

Changing Requirements

The immediate cause of the degradation of the design is well understood. The requirements have been changing in ways that the initial design did not anticipate.

Often these changes need to be made quickly, and may be made by engineers who are not familiar with the original design philosophy. So, though the change to the design works, it somehow violates the original design.

Bit by bit, as the changes continue to pour in, these violations accumulate until malignancy sets in.

Martin, DP[2]

Changing Requirements

However,
we cannot blame
the drifting of the
requirements for the
degradation of the design.
We, as software engineers,
know full well that
requirements
change.

Indeed,
most of us realize
that the requirements
document is the most volatile
document in the project. If our
designs are failing due to the
constant rain of changing
requirements, it is our
designs that are at
fault.

Martin, DP[2]

Dependency Management

- Software rot is the result of improper dependencies between modules
- Dependency among modules must be managed – use design principles

Principles of OO Class Design

- Open-Closed Principle
- Liskov Substitution Principle
- Dependency Inversion Principle
- Interface Segregation Principle

Open-Closed Principle

"A module should be open for extension, but closed for modification."

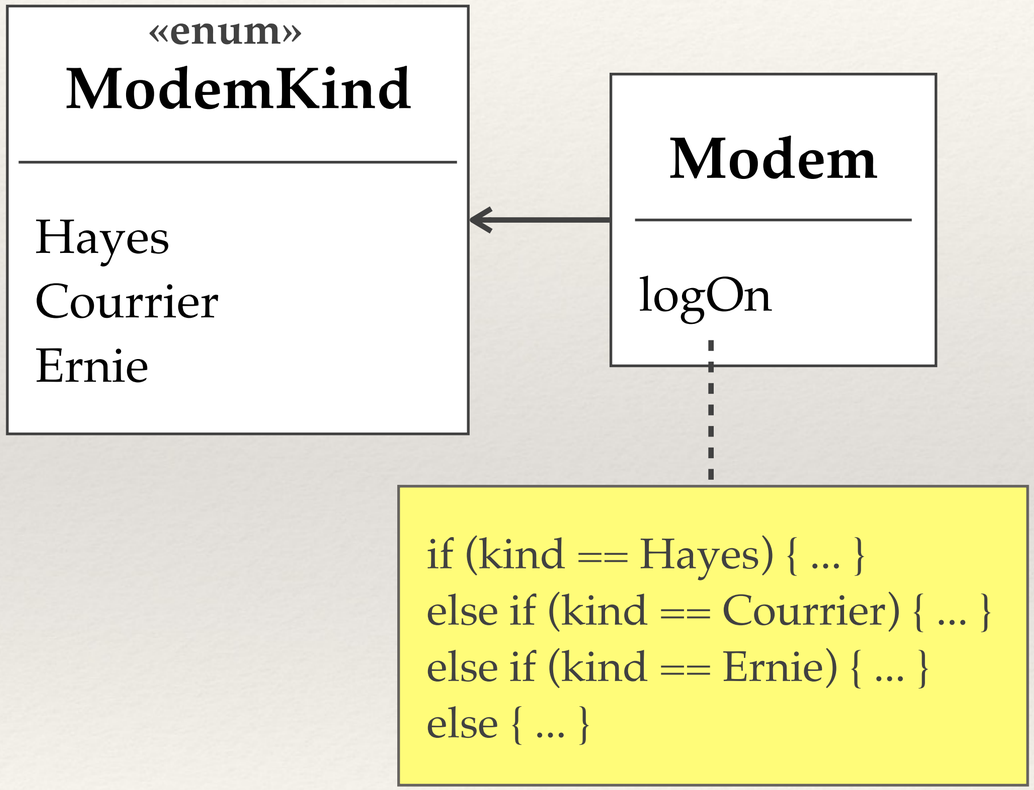
Of all the principles of object-oriented design, this is the most important.

We should write our modules so that they can be extended, without requiring them to be modified.

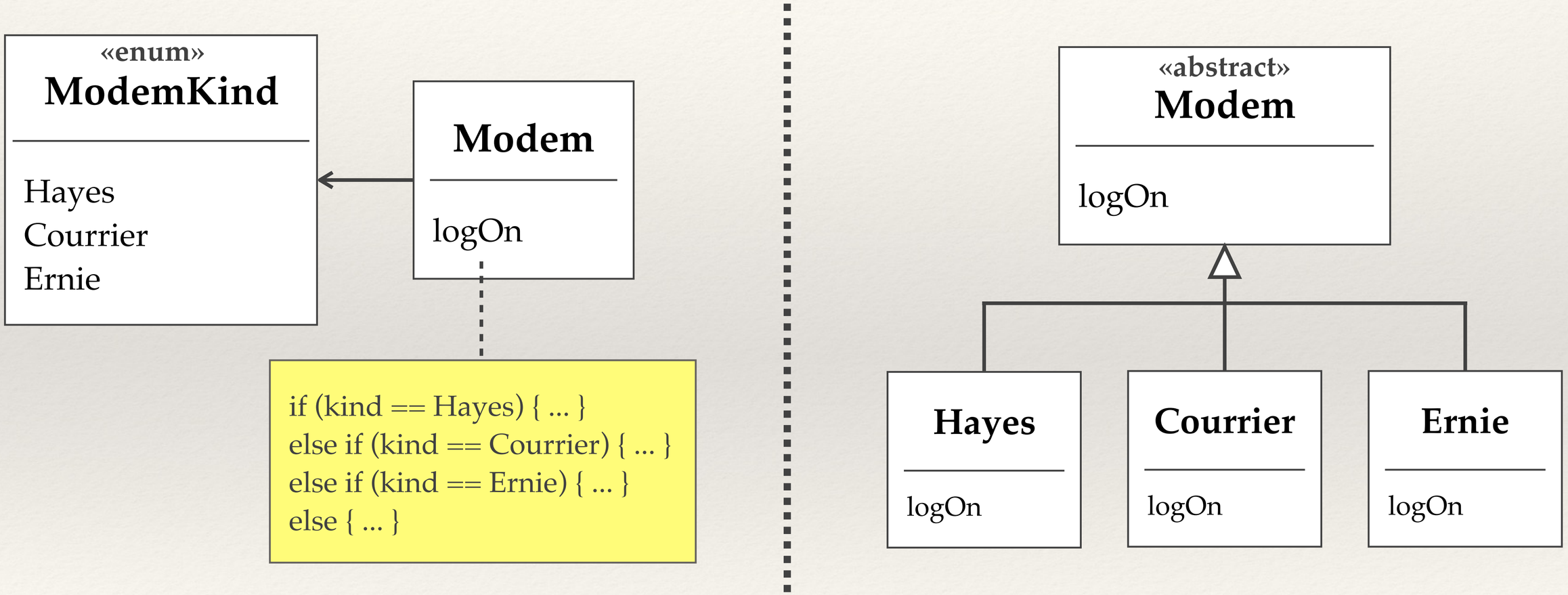
In other words, we want to be able to change what the modules do, without changing the source code of the modules.

Martin, DP[2]

Open-Closed Principle



Open-Closed Principle



Pause and Think

What refactoring did we see in the
Open-Closed example?

Final Thoughts on the OCP

Abstraction is the key to the OCP.

If you don't have to change working code, you aren't likely to break it.

Martin, DP[2]

Design by Contract

- The relationship between a class and its clients can be viewed as a formal agreement, expressing each party's rights and obligations.

```
public E remove(int index)
// REQUIRES that the specified index is in the range
//           0 ≤ index < size()
// ENSURES the element at the specified position in this list
//         is removed, subsequent elements are shifted to the left
//         (1 is subtracted from their indices), and the element
//         that was removed is returned
```

Design by Contract

- Java documentation for the list remove method

```
/**
 * Removes the element at the specified position in this list.
 * Shifts any subsequent elements to the left (subtracts one from
 * their indices). Returns the element that was removed from the list.
 *
 * @throws IndexOutOfBoundsException
 *     if the specified index is out of range
 *     ( index < 0 || index >= size( ) )
 */
public E remove(int index) throws IndexOutOfBoundsException
```

Pause and Think

In the contract implied by the Java documentation, who is responsible for checking if the index is out of range?

```
/**
 * Removes the element at the specified position in this list.
 * Shifts any subsequent elements to the left (subtracts one from
 * their indices). Returns the element that was removed from the list.
 *
 * @throws IndexOutOfBoundsException
 *     if the specified index is out of range
 *     ( index < 0 || index >= size( ) )
 */
public E remove(int index) throws IndexOutOfBoundsException
```

Design by Contract

A pre-condition is a statement of how we expect the world to be before we execute an operation.

We might define a pre-condition for the “square root” operation of input ≥ 0.

Such a pre-condition says that it is an error to invoke “square root” on a negative number and that the consequences of doing so are undefined.

Fowler, UML

Design by Contract

On first glance, this seems a bad idea, because we should put some check somewhere to ensure that “square root” is invoked properly.

The important question is: Who is responsible for doing so?

The pre-condition makes it explicit that the caller is responsible for checking.

Fowler, UML

Design by Contract

Without this explicit statement of responsibilities, we can get either too little checking (because both parties assume that the other is responsible) or too much (both parties check).

Fowler, UML

Too much checking is a bad thing, because it leads to lots of duplicate checking code, which can significantly increase the complexity of a program.

Being explicit about who is responsible helps to reduce this complexity.

The danger that the caller forgets to check is reduced by the fact that assertions are usually checked during debugging and testing.

Design by Contract

Partial procedures lead to programs that are not robust. A robust program is one that continues to behave reasonably even in the presence of errors. A program like this is said to provide graceful degradation.

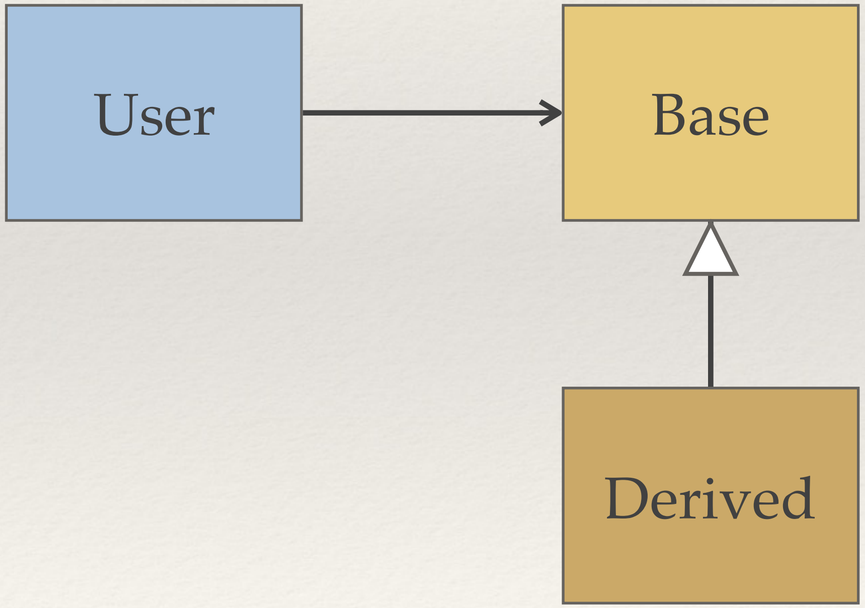
Liskov, PD in Java

Liskov Substitution Principle

"Subclasses should be substitutable for their base classes."

A user of a base class should continue to function properly if a derivative of that base class is passed to it.

Martin, DP[2]

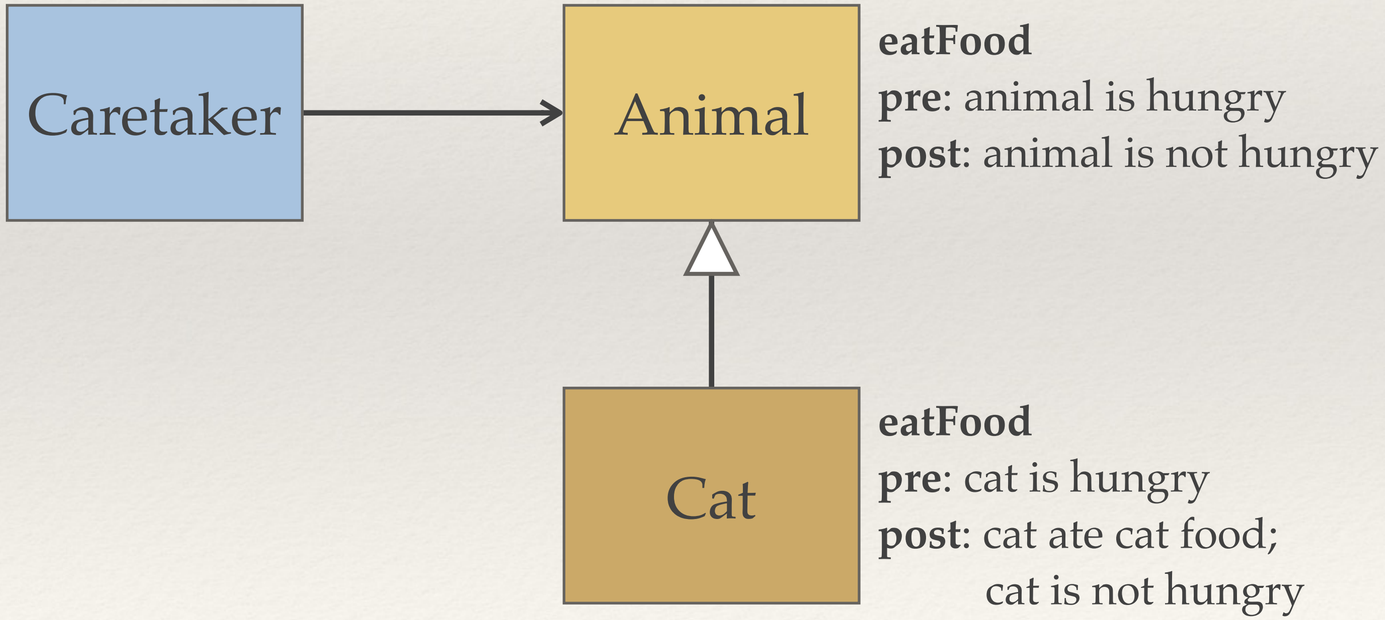


Liskov Substitution Principle

"Subclasses should be substitutable for their base classes."

A user of a base class should continue to function properly if a derivative of that base class is passed to it.

Martin, DP[2]

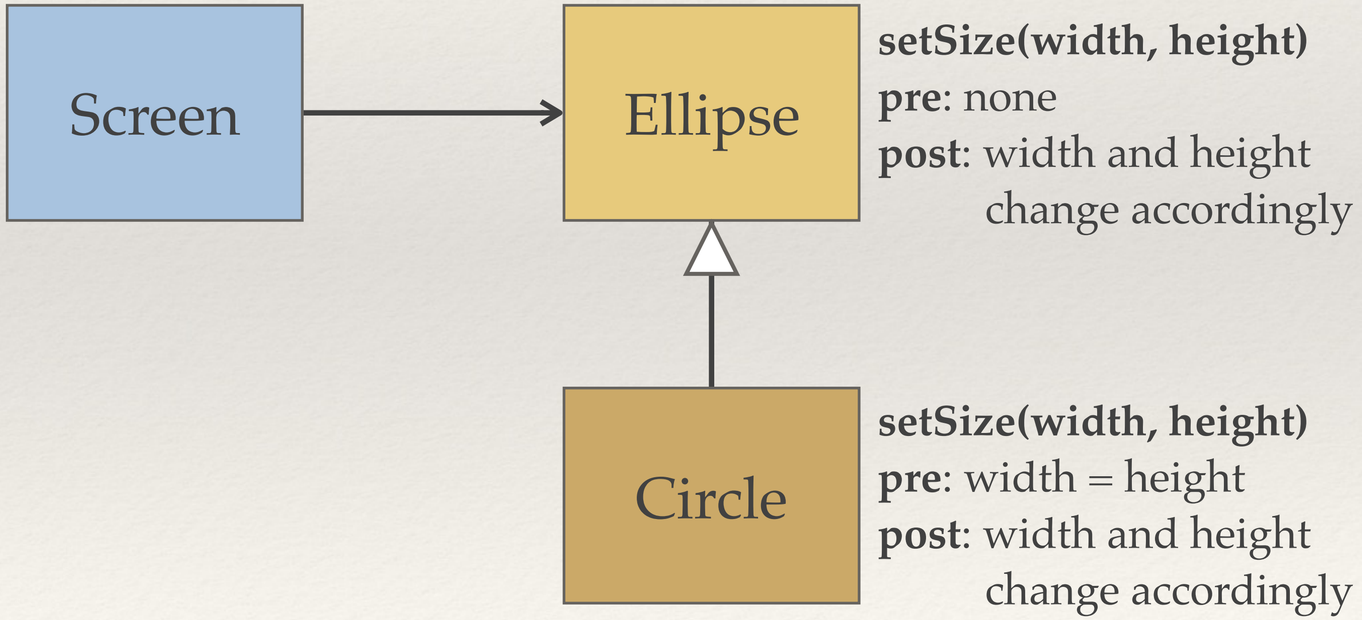


Liskov Substitution Principle

"Subclasses should be substitutable for their base classes."

A user of a base class should continue to function properly if a derivative of that base class is passed to it.

Martin, DP[2]

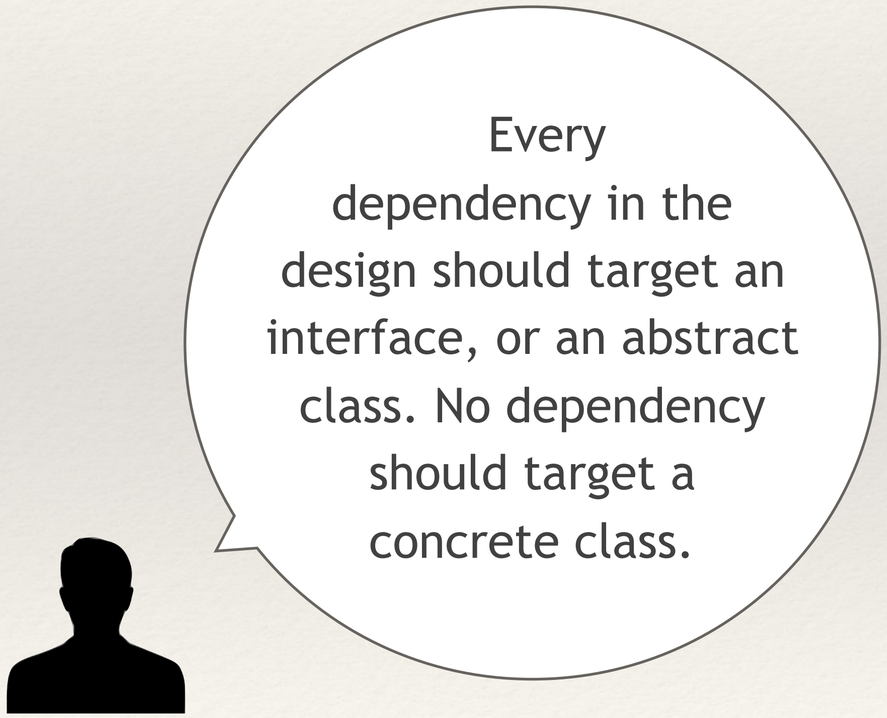


Liskov Substitution Principle

- In terms of contracts, a derived class is substitutable for its base class if
  1. Its preconditions are no stronger than the base class method
  2. Its postconditions are no weaker than the base class method
- In other words, derived methods should ***expect no more and provide no less***

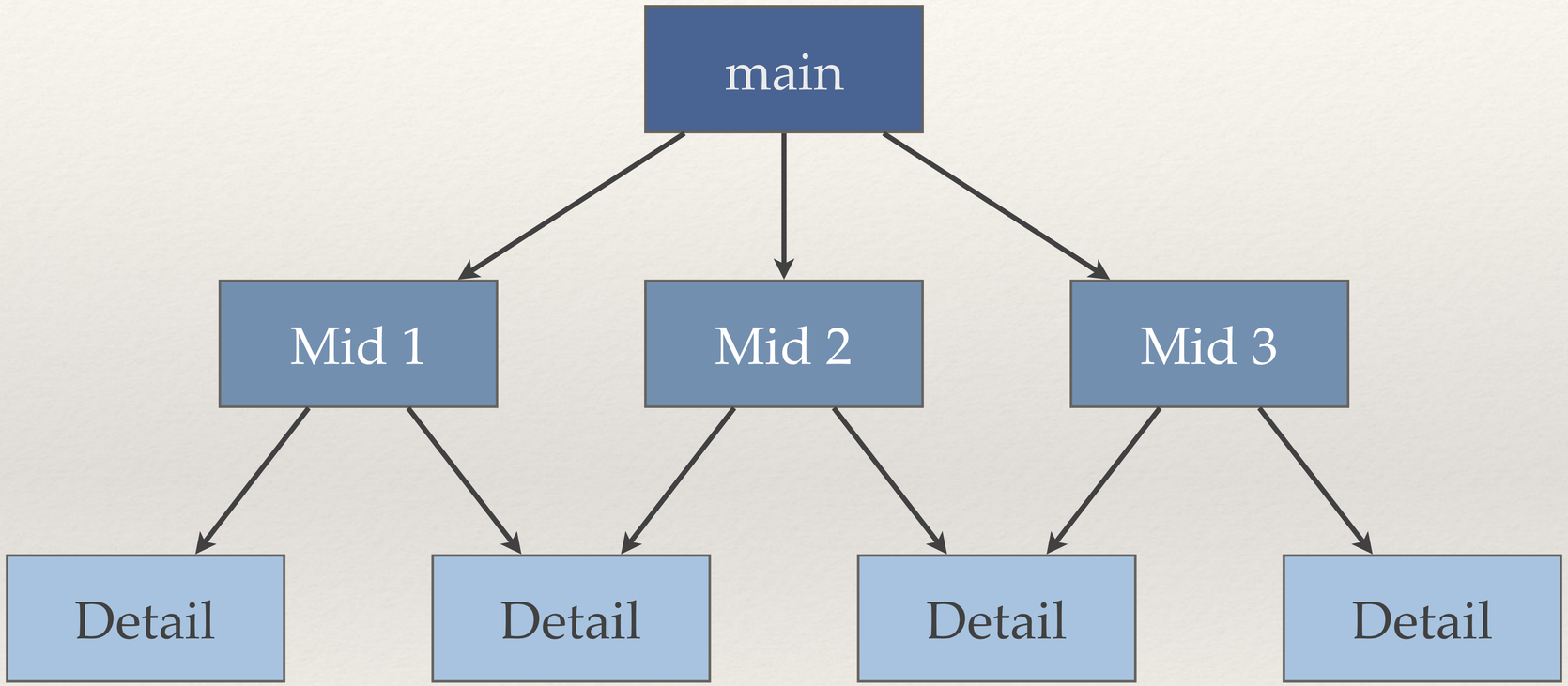
Dependency Inversion Principle

"Depend upon abstractions. Do not depend upon concretions."



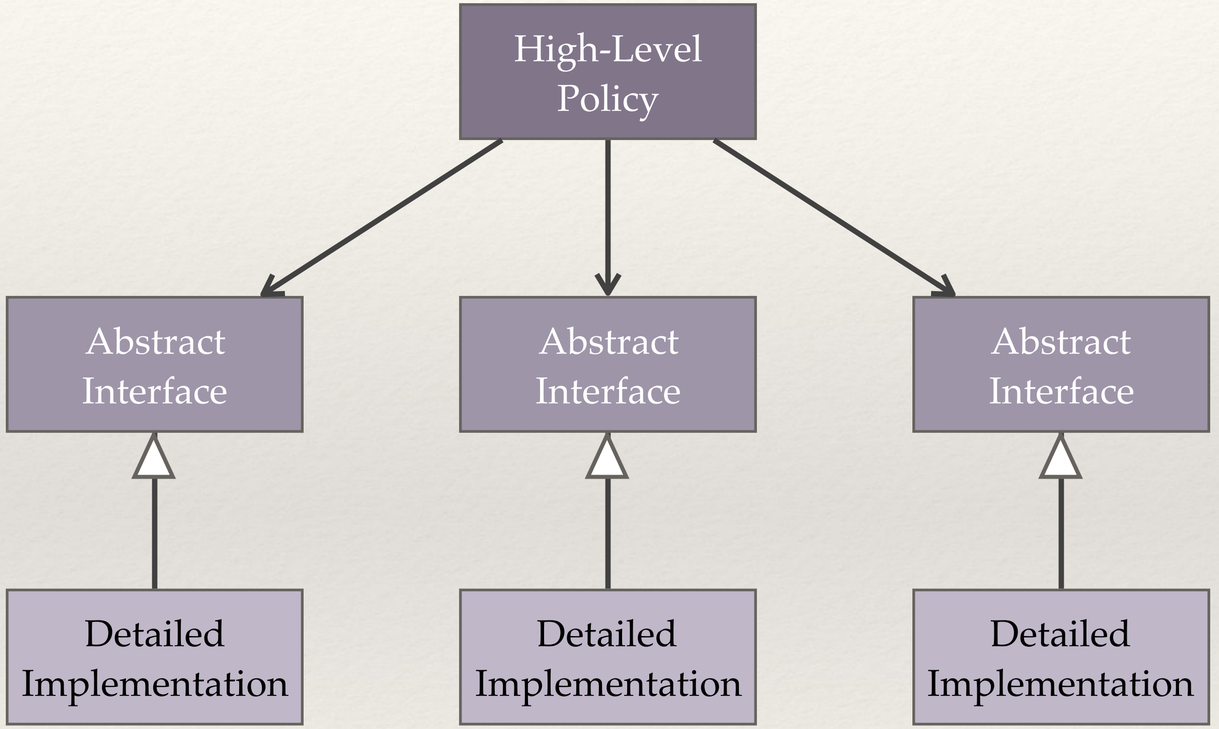
Martin, DP[2]

Dependency Inversion Principle



Dependency Structure of a Procedural Architecture

Dependency Inversion Principle



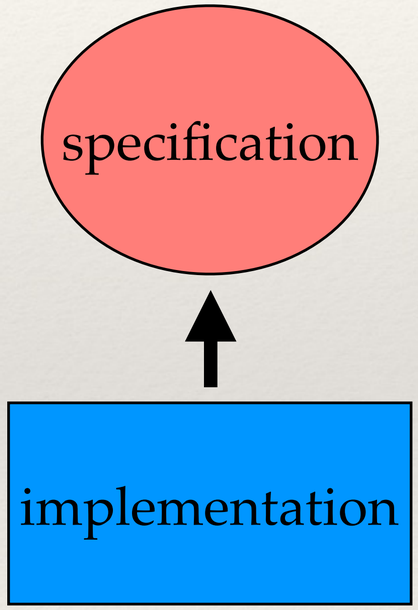
Dependency Structure of an Object-Oriented Architecture

Software Components

A component should be usable solely on the basis of its specification.

B. Meyer & C. Szyperski

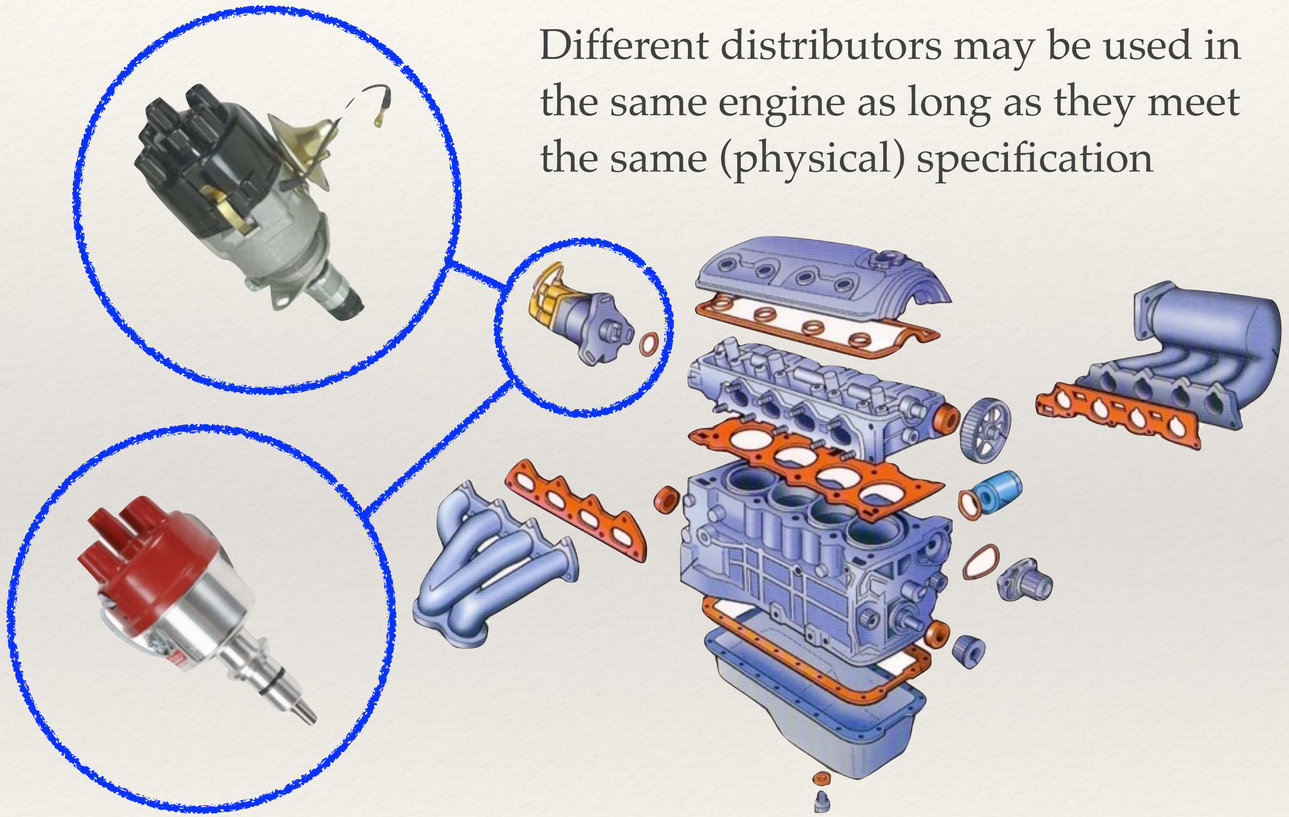
Software Components



abstraction

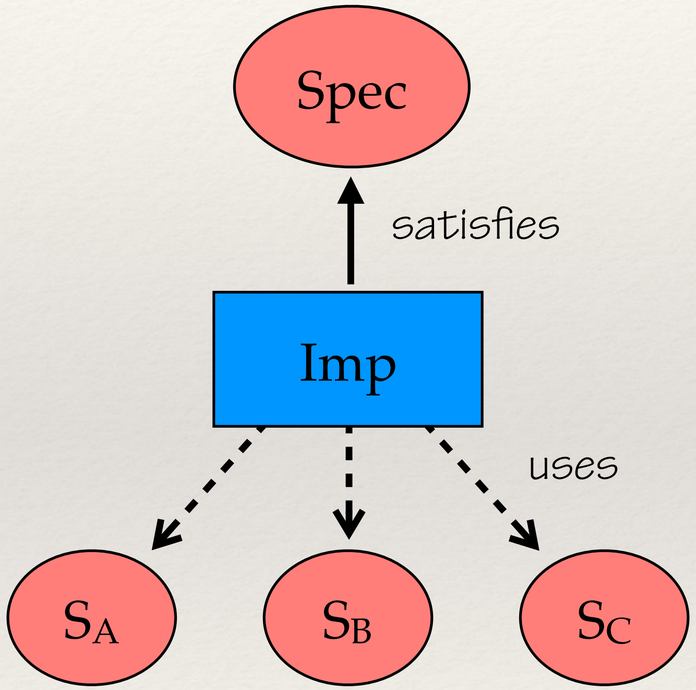
concretion

Modular (Component) Reasoning

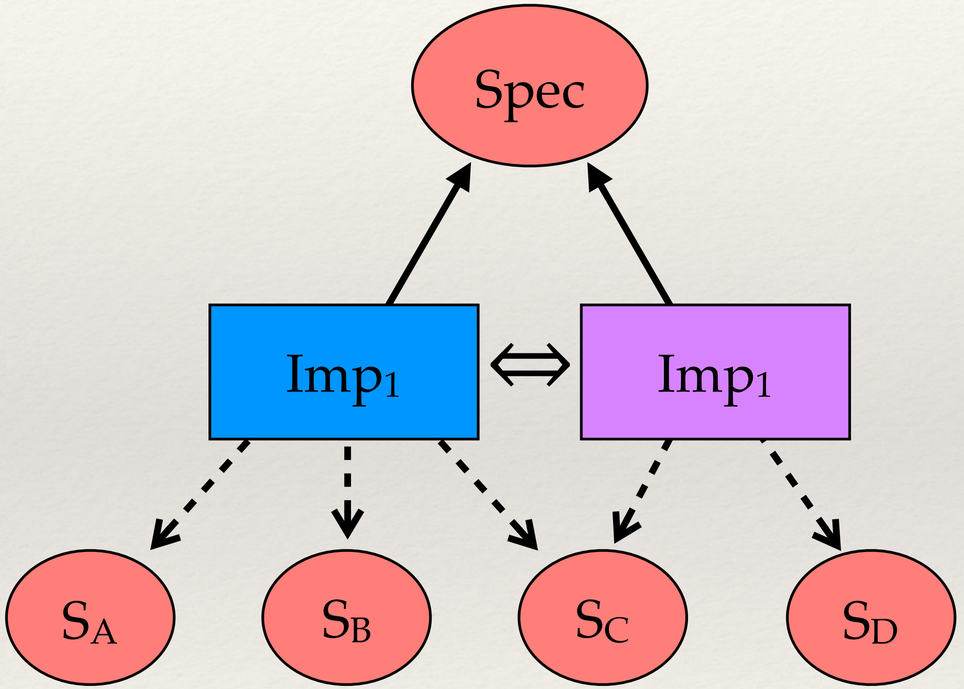


Different distributors may be used in the same engine as long as they meet the same (physical) specification

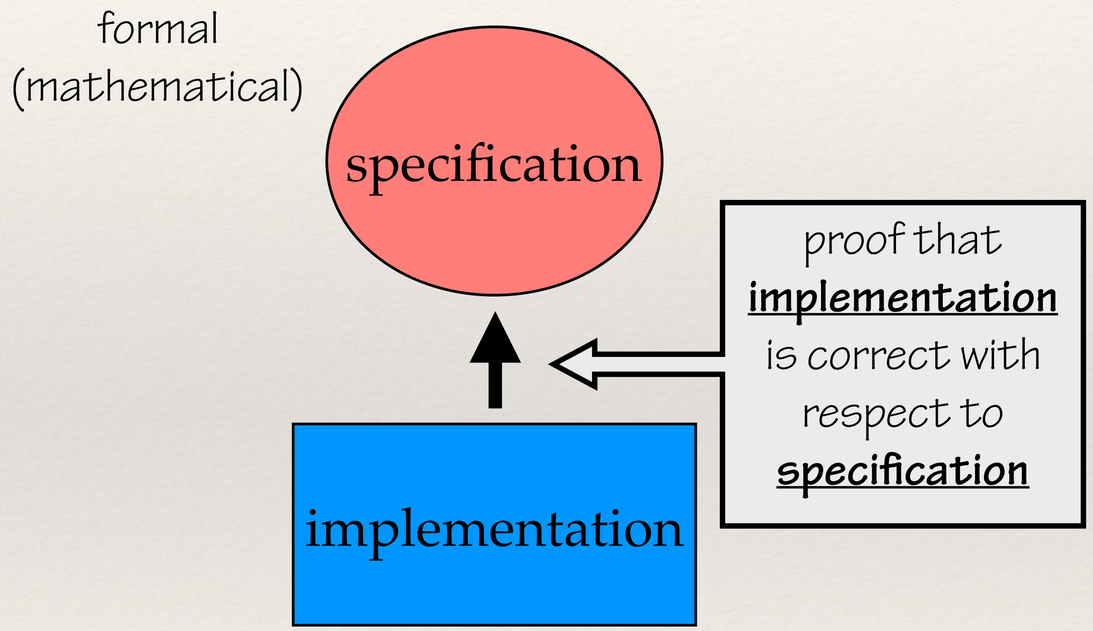
Modular Reasoning



Modular Reasoning



Verified Software Components



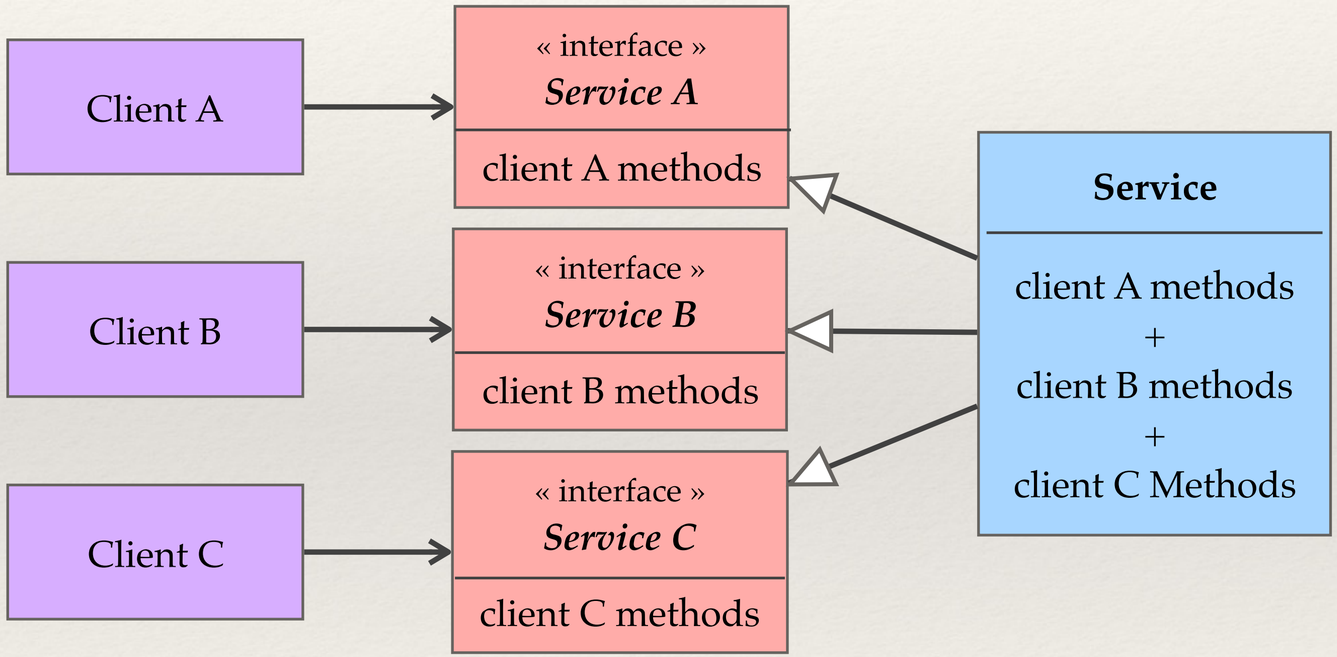
Interface Segregation Principle

"Many client specific interfaces are better than one general purpose interface."

If you have a class that has several clients, rather than loading the class with all the methods that the clients need, create specific interfaces for each client and multiply inherit them into the class.

Martin, DP[2]

Interface Segregation Principle



Cohesion and Coupling

Functional Independence

- COHESION – The degree to which a module performs one and only one function.
- COUPLING – The degree to which a module is connected to other modules in the system.

Package Cohesion Principles

- The Release Reuse Equivalency Principle
  The granule of reuse is the granule of release.
- The Common Closure Principle
  Classes that change together, belong together.
- The Common Reuse Principle
  Classes that aren't reused together should not be grouped together.

Package Coupling Principles

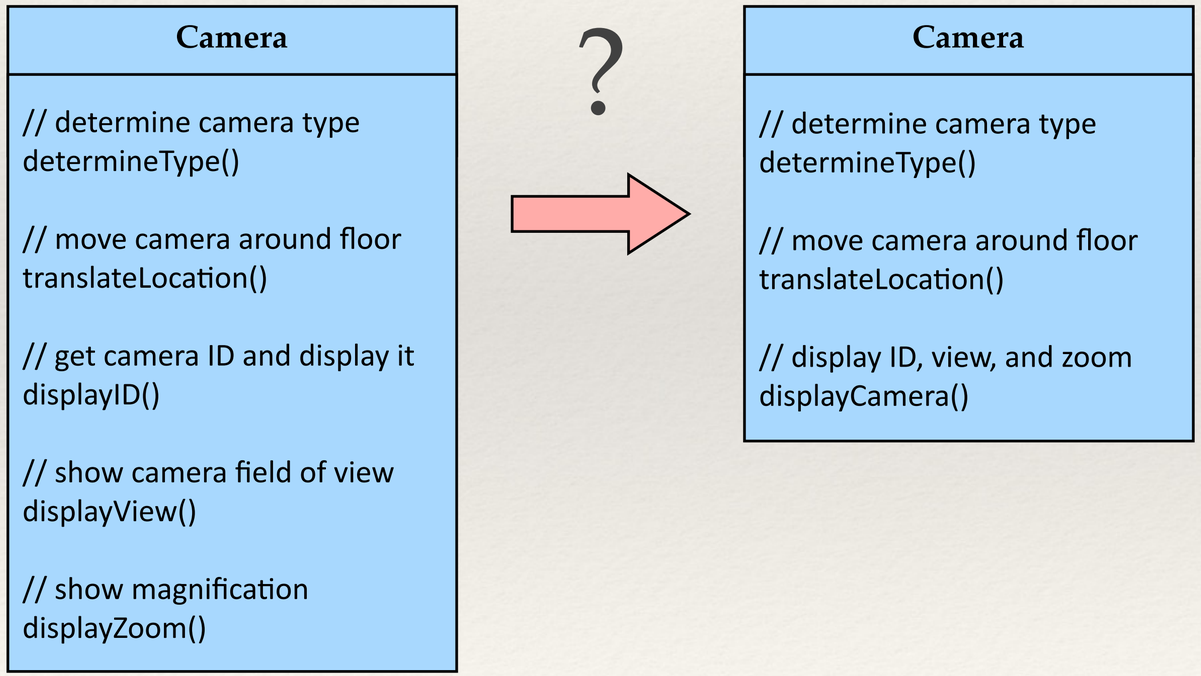
- The Acyclic Dependencies Principle
  The dependencies between packages must not form cycles.
- The Stable Dependencies Principle
  Depend in the direction of stability.
- The Stable Abstractions Principle
  Stable packages should be abstract packages.

Cohesion

- The “single-mindedness of a module”
- A component should contain only attributes and operations that are closely related to one another and to the component itself.
- Types of cohesion include:
  - ***functional***
  - ***layer***
  - ***communicational***

Functional Cohesion

- A function should perform only one task



Pause and Think

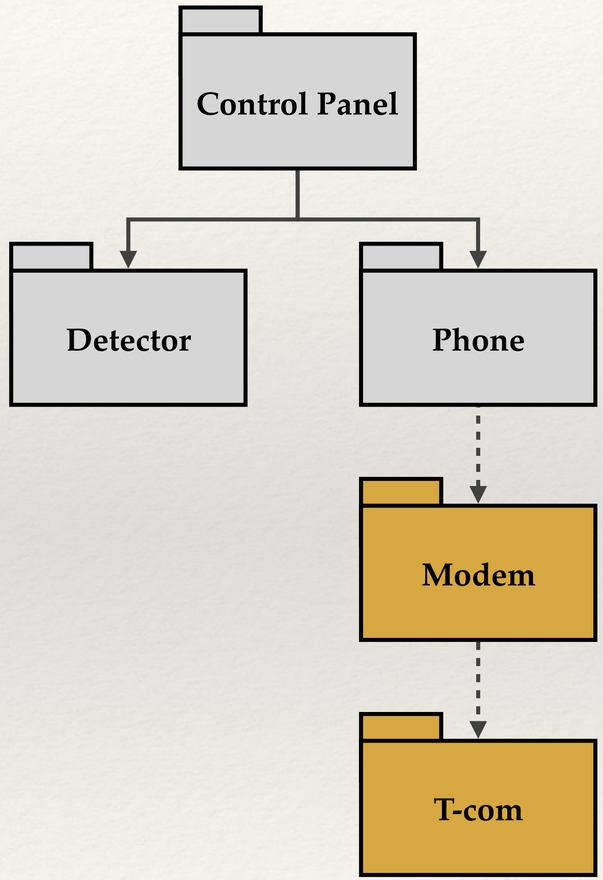
How can combining

displayID
displayView
displayZoom

into a single method
(displayCamera)
be problematic?

Layer Cohesion

- A higher layer in a system can access a lower layer, but not the other way around



Communicational Cohesion

- All operations that access the same data are defined within one component.
- Often, such components focus solely on the data in question, accessing and storing it.
- Example: A StudentRecord class that adds, removes, updates, and accesses various fields of a student record for client components.

Coupling

- A qualitative measure of the degree to which classes or components are connected to each other.
  - *content coupling* AVOID!
  - *common coupling* Use Caution
  - *routine call coupling, type-use coupling, import coupling* Be Aware of

Content Coupling

- Occurs when one component "surreptitiously modifies data that is internal to another component"
- Violates information hiding

```
public class StudentRecord {

    private String name;
    private int[ ] quizScores;

    public String getName() {
        return name;
    }

    public int getQuizScore(int n) {
        return quizScores[n];
    }

    public int[] getAllQuizScores() {
        return quizScores;
    }

    ...
```

Pause and Think

What is wrong with this class?

```
public class StudentRecord {

    private String name;
    private int[ ] quizScores;

    public String getName() {
        return name;
    }

    public int getQuizScore(int n) {
        return quizScores[n];
    }

    public int[] getAllQuizScores() {
        return quizScores;
    }

    ...
```

Common Coupling

- Components use the same global variable

```
package farwest;

import Environment.setup;

public class MyClass {

    public void doSomething() {
        // do something
        // use setup
    }

    public void doSomethingElse() {
        // do something else
        // modify setup
    }

    ...
```

```
package fareast;

import Environment.setup;

public class YouClass {

    public void doSomething() {
        // do something
        // use setup
    }

    public void doSomethingElse() {
        // do something else
        // modify setup
    }

    ...
```

Routine Coupling

- Some types of coupling occur routinely in object-oriented code.