COM S 362
Object-Oriented Analysis & Design

SOLID
Open-Closed and Liskov Substitution

Reading

Robert C. Martin. Clean Architecture, 2018.

- Ch. 8 and 9, pp. 69-82

SOLID

- SOLID
  - **S**ingle Responsibility Principle
  - **O**pen-Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inversion Principle
- Design principles assembled by Robert Martin (“Uncle Bob”), although some originally developed by others
- Goal: make mid-level (module level) software structures that
  - Tolerate change
  - Are easy to understand
  - Are reusable in many software systems

The Single Responsibility Principle

- We will not spend time on this principle because it is similar to Separation of Concerns, which we have looked at in detail
- Separation of Concerns
  - Separate an application into distinct sections, so each section addresses a separate concern
- Single Responsibility Principle
  - *“A module should have one, and only one, reason to change.”*
  - *“A module should be responsible to one, and only one, actor.”*
- The goal is the same for both, but Single Responsibility gives more direct guidelines on when to separate: if more than one actor's requirements can cause change to a class, then it is not sufficiently separated

The Open-Closed Principle

- Coined by Bertrand Meyer in 1988
  - *“A software artifact should be open for extension but closed for modification.”*
- Translation: we want to be able to add (extend) new features without having to rewrite (modify) existing code

How to Achieve Open-Closed?

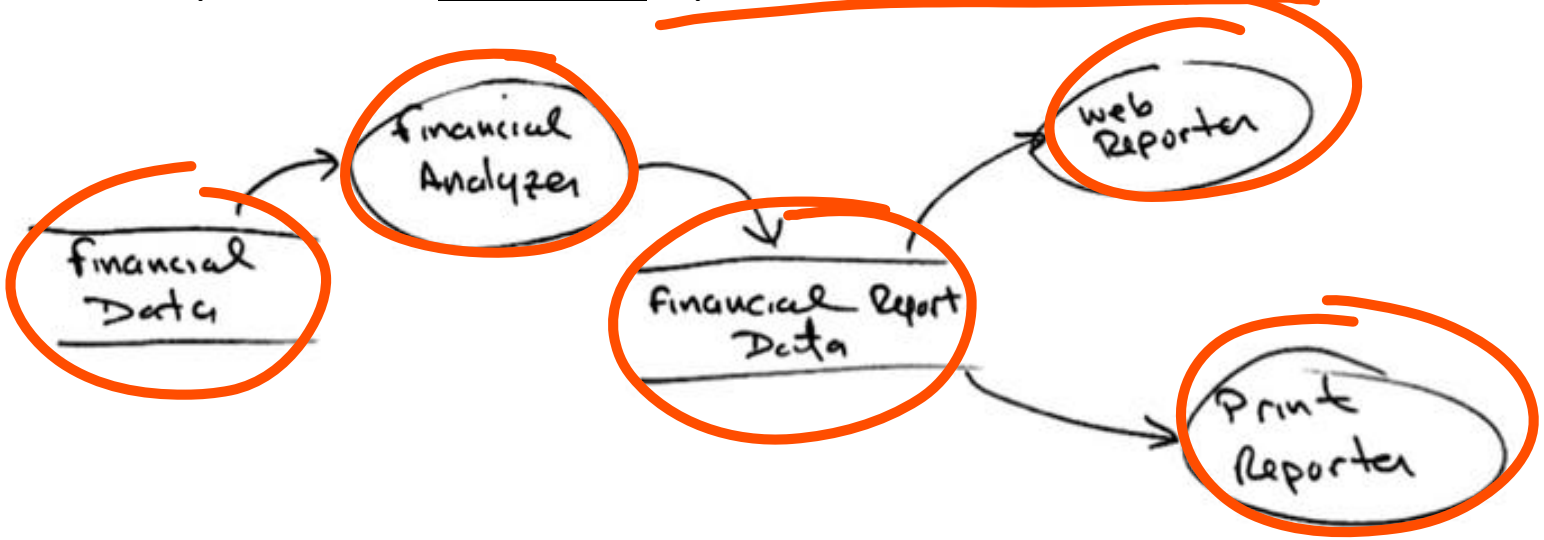
- Separate the things that change for different reasons (i.e., Separation of Concerns/Single Responsibility Principle)
- Organize dependencies (i.e., Dependency Inversion Principle)
  - Protect higher-level components from changes to lower-level components
  - If component A should be protected from changes in component B, then component B should depend on component A

Example: Financial Reporting

- We have a system that displays a financial summary on a web page. The data on the page is scrollable, and negative numbers are rendered in red, for example, -1000.
- New requirement: Stakeholders have asked that the same information be turned into a report to be printed on a black-and-white printer. The report should be properly paginated, with appropriate page headers, page footers, and column labels. Negative numbers should be surrounded by parentheses, for example, (1000).
- How much code rewrite will there be? Is it possible to design a systems in which the answer to most new requirements is near zero rewrite?

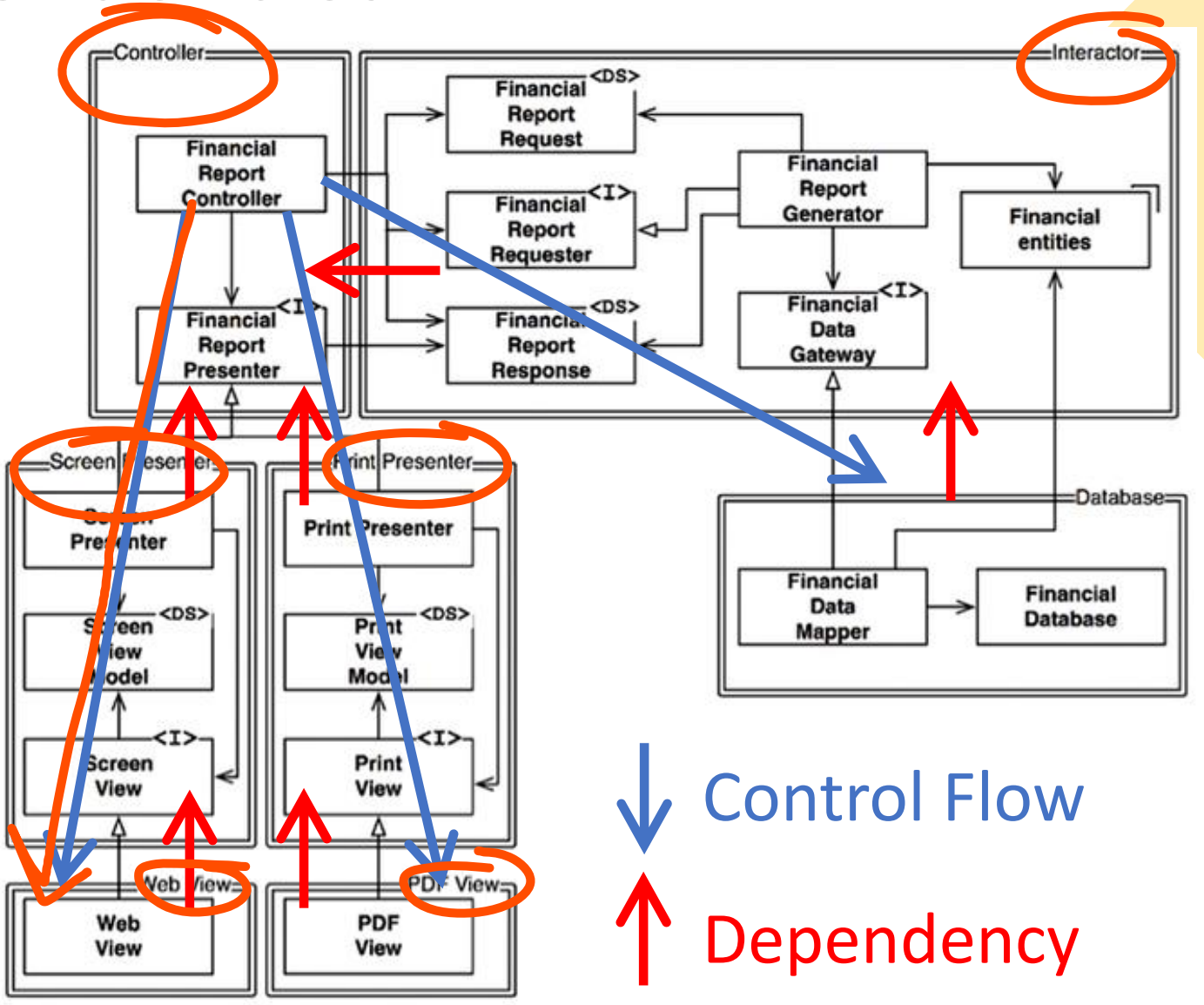
Example: Separate Concerns

- First step to achieve an open-closed design is separation of concerns/single responsibility
- Every component has one primary responsibility – one and only one reason to change
  - FinancialData – responsible for knowing financial records
  - FinancialAnalyzer – responsible for generating report data from financial records
  - FinaincialReportData – responsible for knowing data generated for a particular report
  - WebReporter – responsible for generating a web version of the report
  - PrintReporter – responsible for generating a printable version of the report



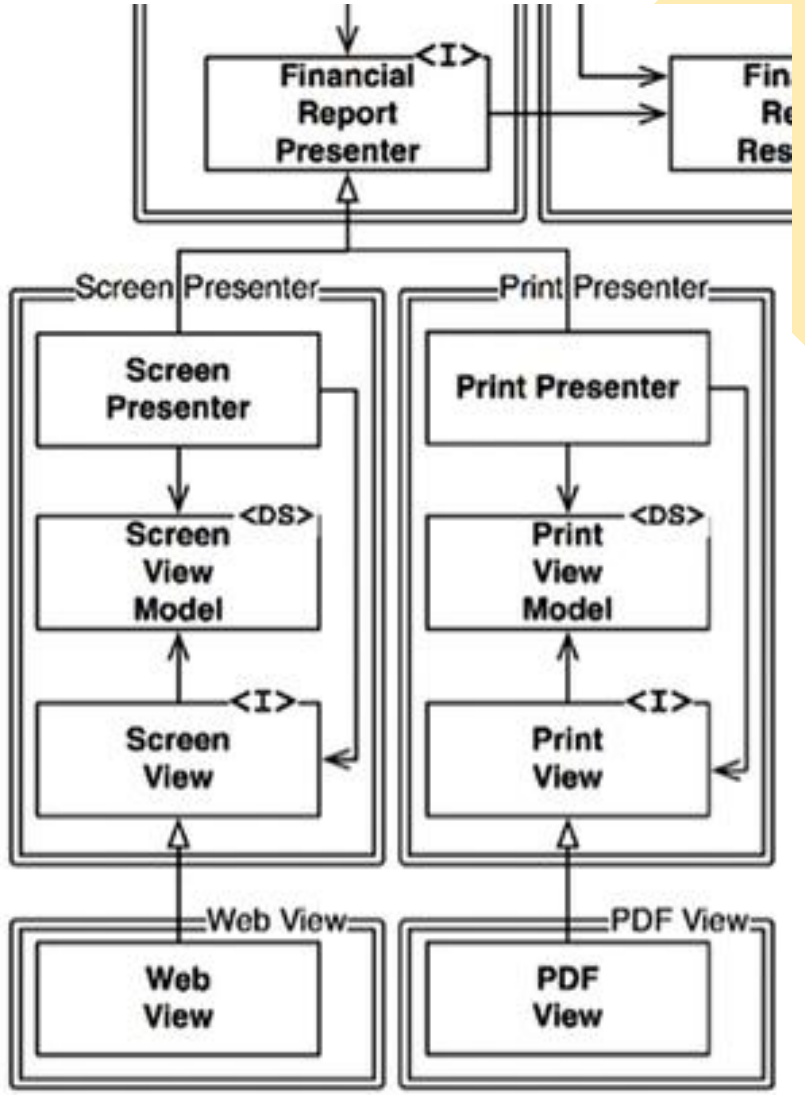
Example: Using Dependencies

- The goal of dependency inversion is to make dependencies point up
- We will find out how this is achieved later
- Up means
  - from low-level to high-level abstractions
  - against the flow of control



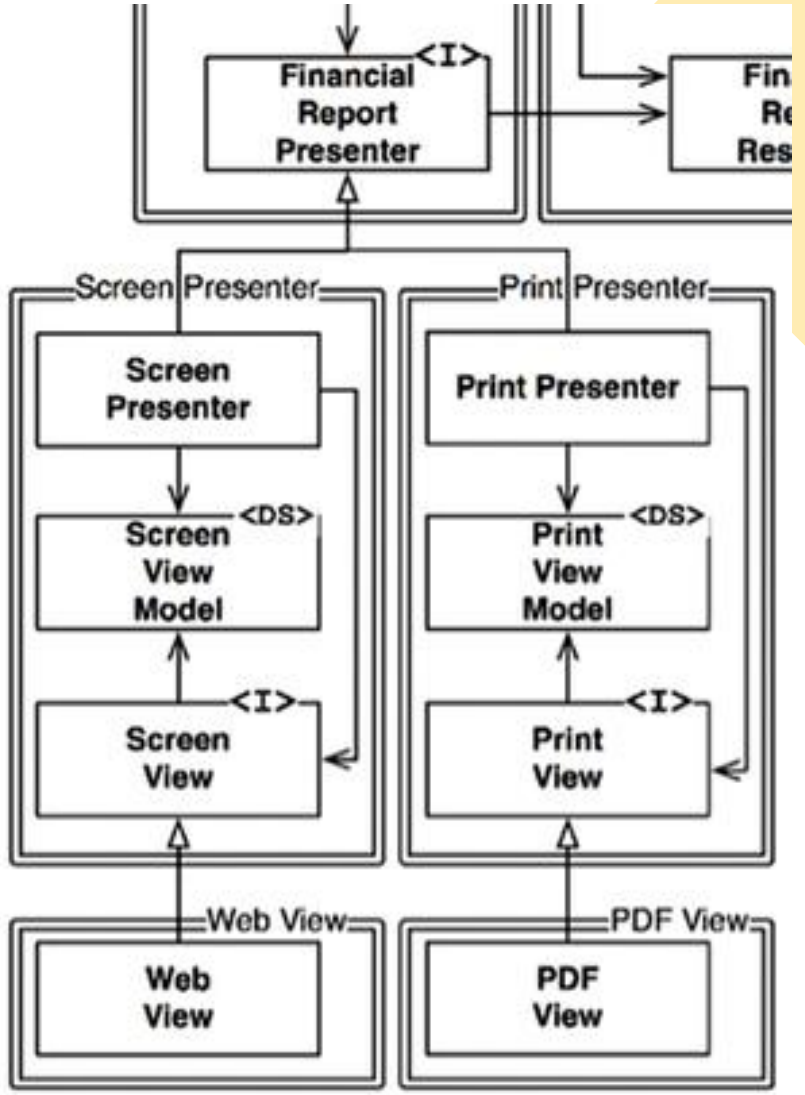
Example: Dependency Impact on Open-Closed

- At class level
  - Protect FinalcialReportController from changes to ScreenPresenter
- At component level
  - Protect Controller from changes to Presenter
  - Protect Presenter from changes to View



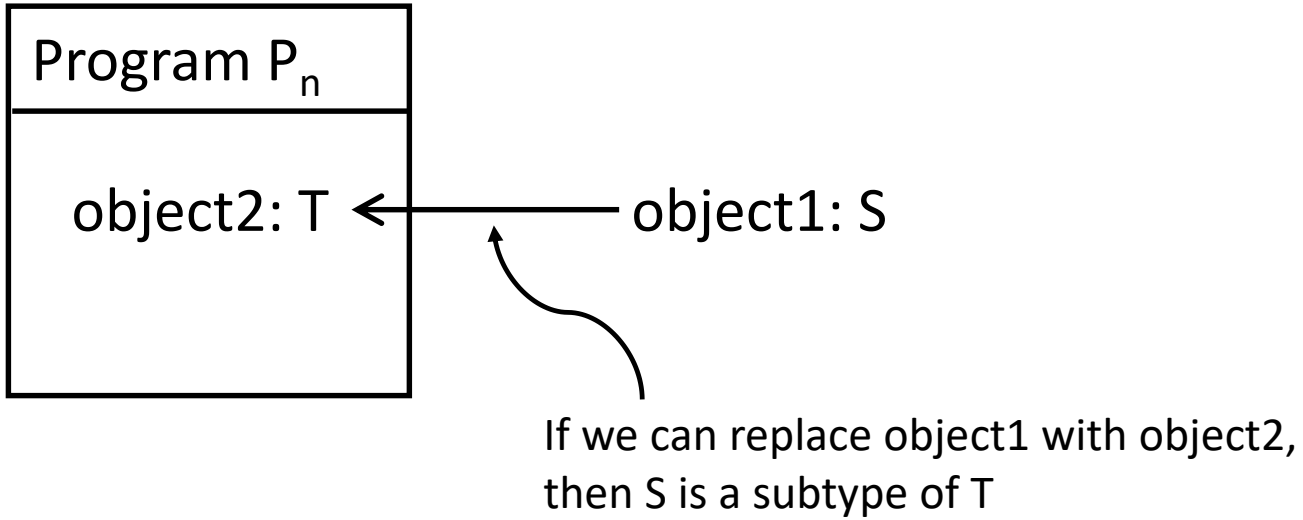
Example: Adding New Report Formats

- Interfaces become **extension points**
  - Places where code can be extended without requiring modification
  - Put at the boundaries of components
  - More about this with Dependency Inversion
- Now when a stakeholder wants their reports to have negative numbers rendered in red with parenthesis, e.g., (1000) and presented in an Excel file, there are only extensions not modifications



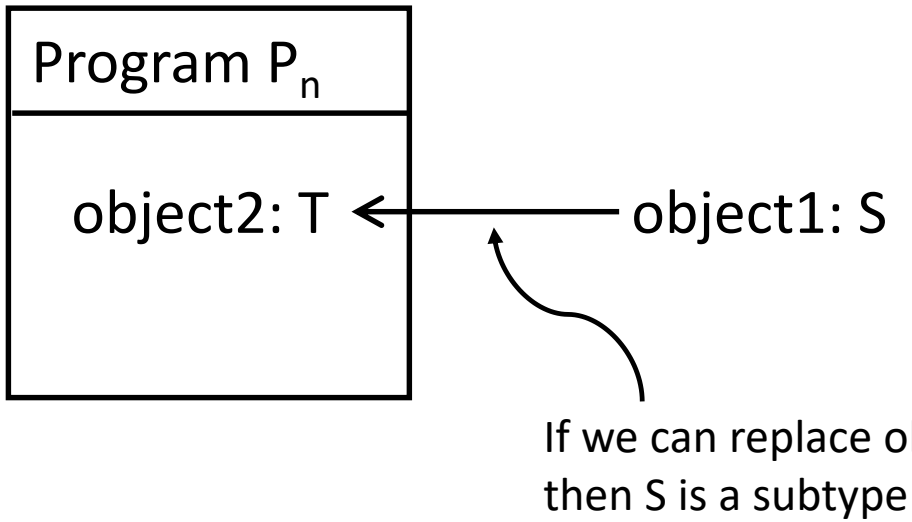
The Liskov Substitution Principle

- Created by Barbara Liskov in 1988
  - *"If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T."*



The Liskov Substitution Principle

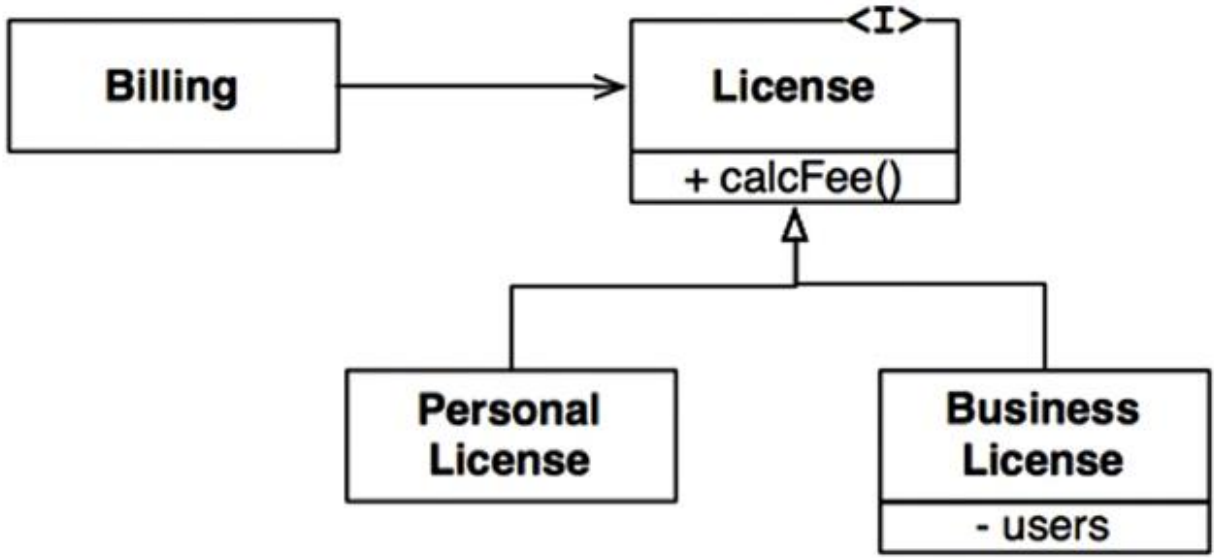
- Created by Barbara Liskov in 1988
  - *"If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T."*



Wait… this sounds like an overly complicated description of **polymorphism**.

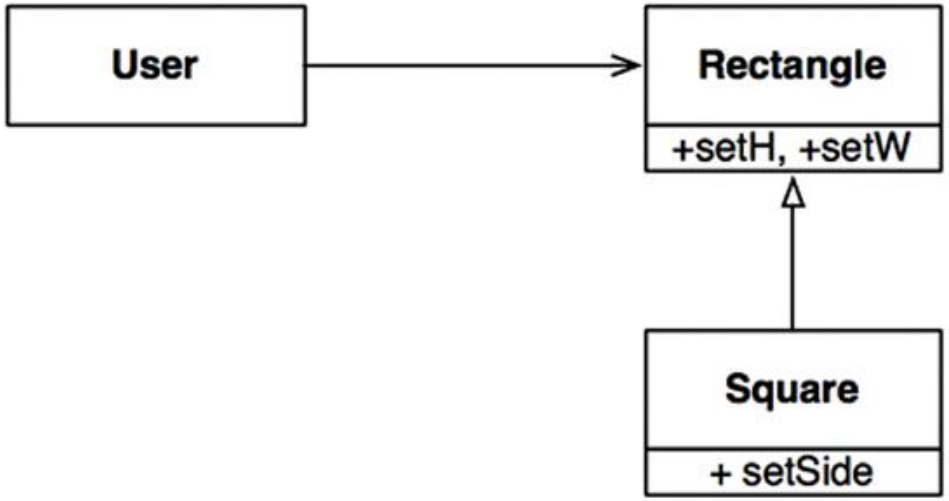
Relation to polymorphism

- Liskov substitution is closely related to polymorphism
  - Subtypes are similar to subclasses
  - In the example below, Billing only depends on the License interface, therefore the subclasses PersonalLicense and BusinessLicense can be freely substituted



When Polymorphism Fails

- But not all polymorphism passes the Liskov test
  - Consider the classic Square/Rectangle Problem




```
// code in User class
Rectangle r = new Rectangle();
r.setW(5);
r.setH(2);
Assert(r.area() == 10);
```

- Square is a type of Rectangle, it obeys the rules of inheritance
- But when we use polymorphism to substitute Square for Rectangle in User, the program fails

Liskov Principle Applied Generally

- In a general sense, we can view Liskov Principle as: use interfaces in a way that allow for free substitution of different implementations
- When we violate Liskov Principle
  - Substitutions require modifying the code of the users of the interface, a clear violation of Open-Close Principle
  - It results in messy logic to handle special cases
  - It produces unmaintainable code like the following

```
if (obj instanceof Square) {
    // do this
} else if (obj instanceof Rectangle) {
    // do that
} else { // TODO: if you create a new Shape class don't forget to add it here
    // oops, unknown shape
}
```