Design Principles

SOLID Design Principles, Part 1

SOLID Design Principles

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

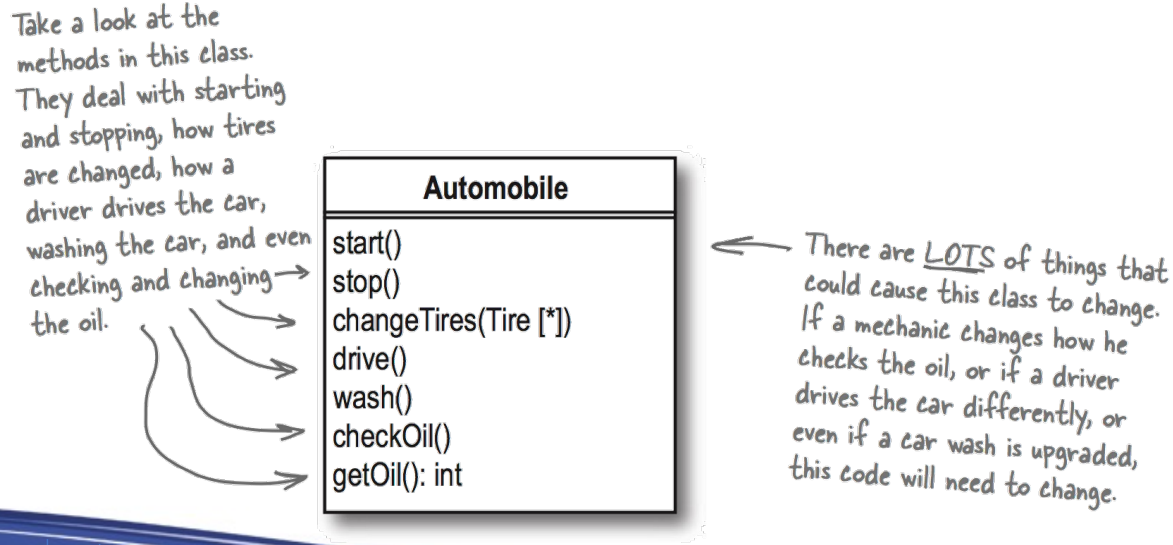
SOLID: Single Responsibility Principle

Design Principle:
Single Responsibility Principle

Every object in a system should have a single responsibility, and all the object's services should be focused on carrying out that single responsibility.

SOLID: Single Responsibility Principle

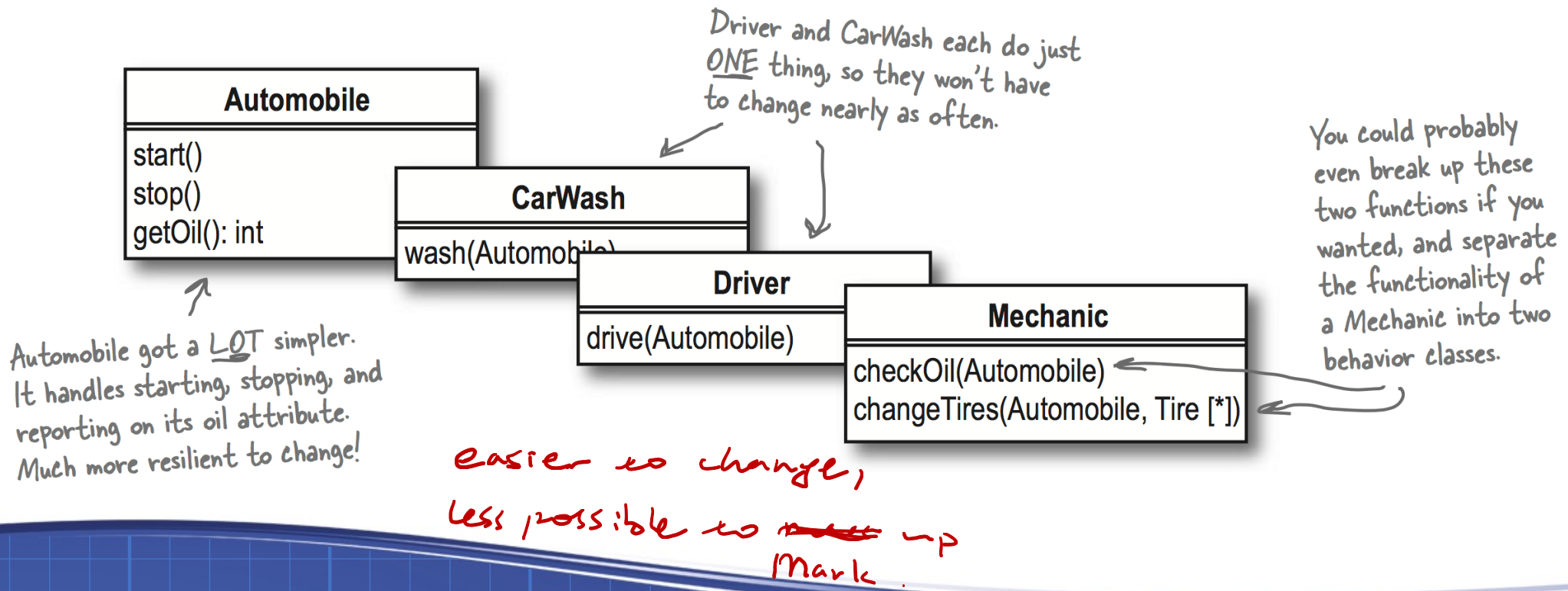
- Every object in a system should have a single responsibility
- Another way to think about a responsibility is as a reason to change



SOLID: Single Responsibility Principle

- When a class has more than one reason to change, it might be trying to do too much
- In such a case, we should try to break the class into multiple classes where each individual class has a single responsibility and thus has only one reason to change

SOLID: Single Responsibility Principle



SOLID: Single Responsibility Principle

- Benefits:
  - Doing this minimizes the chance that a class will need to be changed by reducing the number of things in the class that can change
  - Generally, this also results in high cohesion as the elements of the class belong together in a stronger way
  - Achieving higher cohesion generally increases reusability, robustness, understandability, and so on

S**O**LID: Open/Closed Principle

Design Principle:
Open/Closed Principle

Classes should be open for extension, and closed for modification.

S**O**LID: Open/Closed Principle

- **Closed for modification:** The source code of our classes is to be treated as immutable … no one should be allowed to modify it
- Changing existing code can introduce new bugs
- If we need a different behaviour, we should extend the class

rather than modifying the original source code.

S**O**LID: Open/Closed Principle

- **Open for extension:** Behavioural changes that may be required should be accomplished through inheritance or other means (e.g. the Observer pattern … more on this later)

- We should not touch our existing, well-tested code!

SOLID: Open/Closed Principle

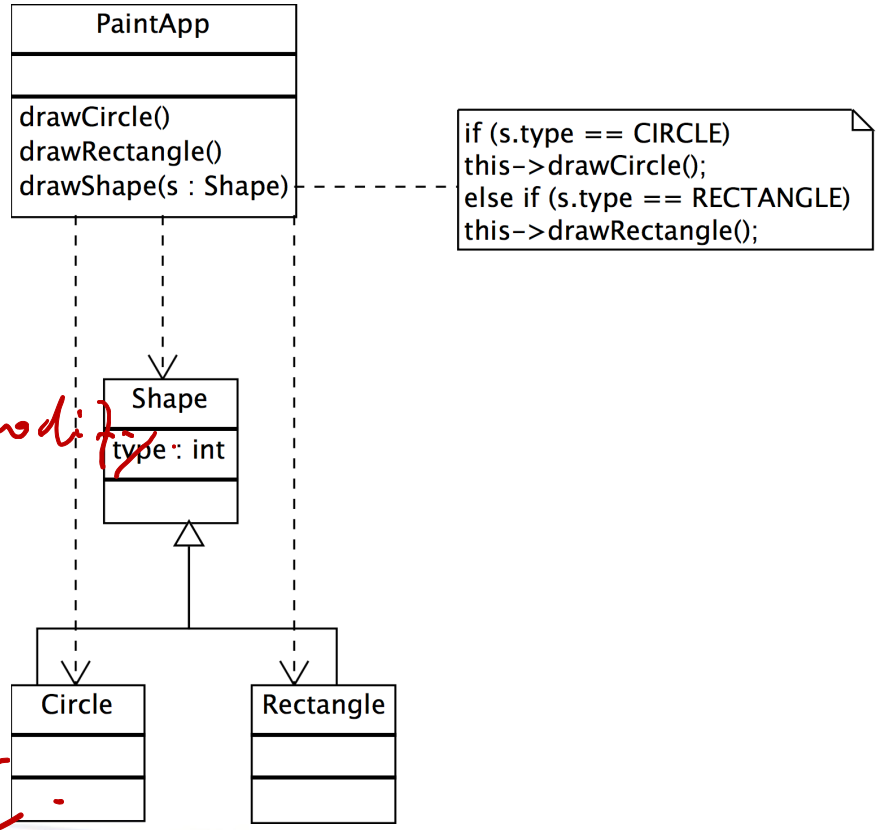
You open classes by allowing them to be subclassed and extended.

You close classes by not allowing anyone to touch your working code.

SOLID: Open/Closed Principle

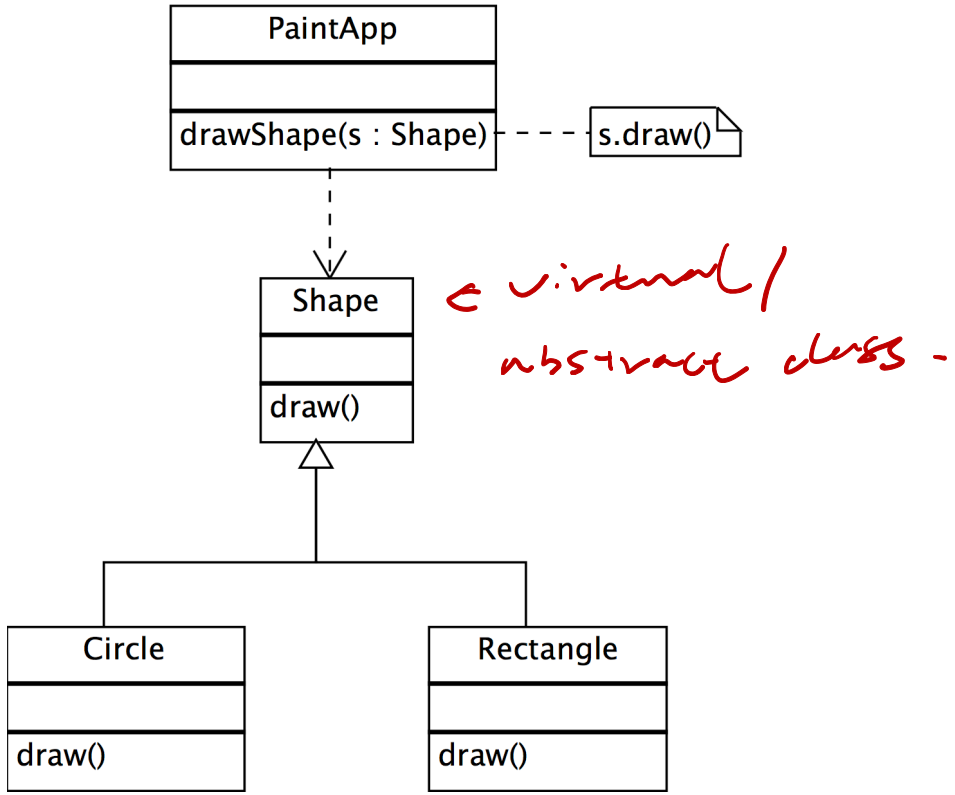
- This example violates the open/closed principle
- What happens when we need to add a new `Shape` subclass? goes PaintApp and modify
- We would need to change `PaintApp` to accommodate the new shape

it is not ideal. Not closed for modification.



SOLID: Open/Closed Principle

- A better approach is to refactor the code and take advantage of the inheritance hierarchy that is in place
- We can now create new `Shape` subclasses without requiring changes to `PaintApp`



S**O**LID: Open/Closed Principle

- Benefits:
  - This effectively allows the behaviour of a class to be modified without touching its source code
  - In doing so, we don't risk breaking well-tested code
  - Instead, we only modify existing code to fix errors; new/modified features require extension

SOLID: Liskov Substitution Principle

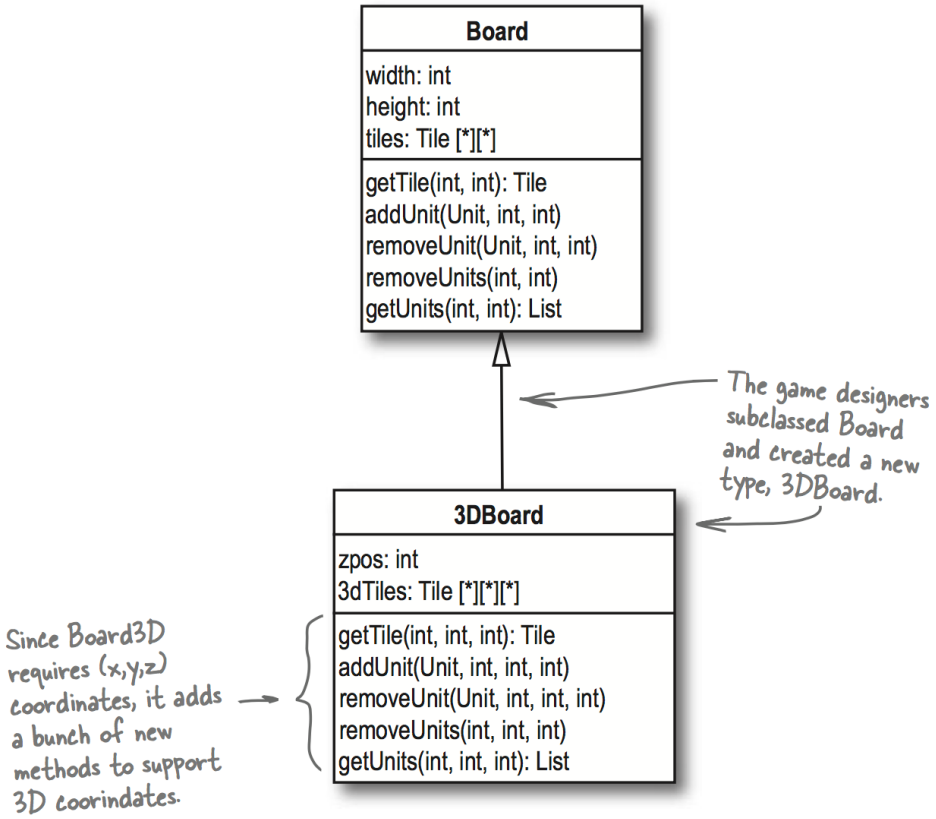
Design Principle:
Liskov Substitution Principle

Subtypes must be substitutable for their base types.

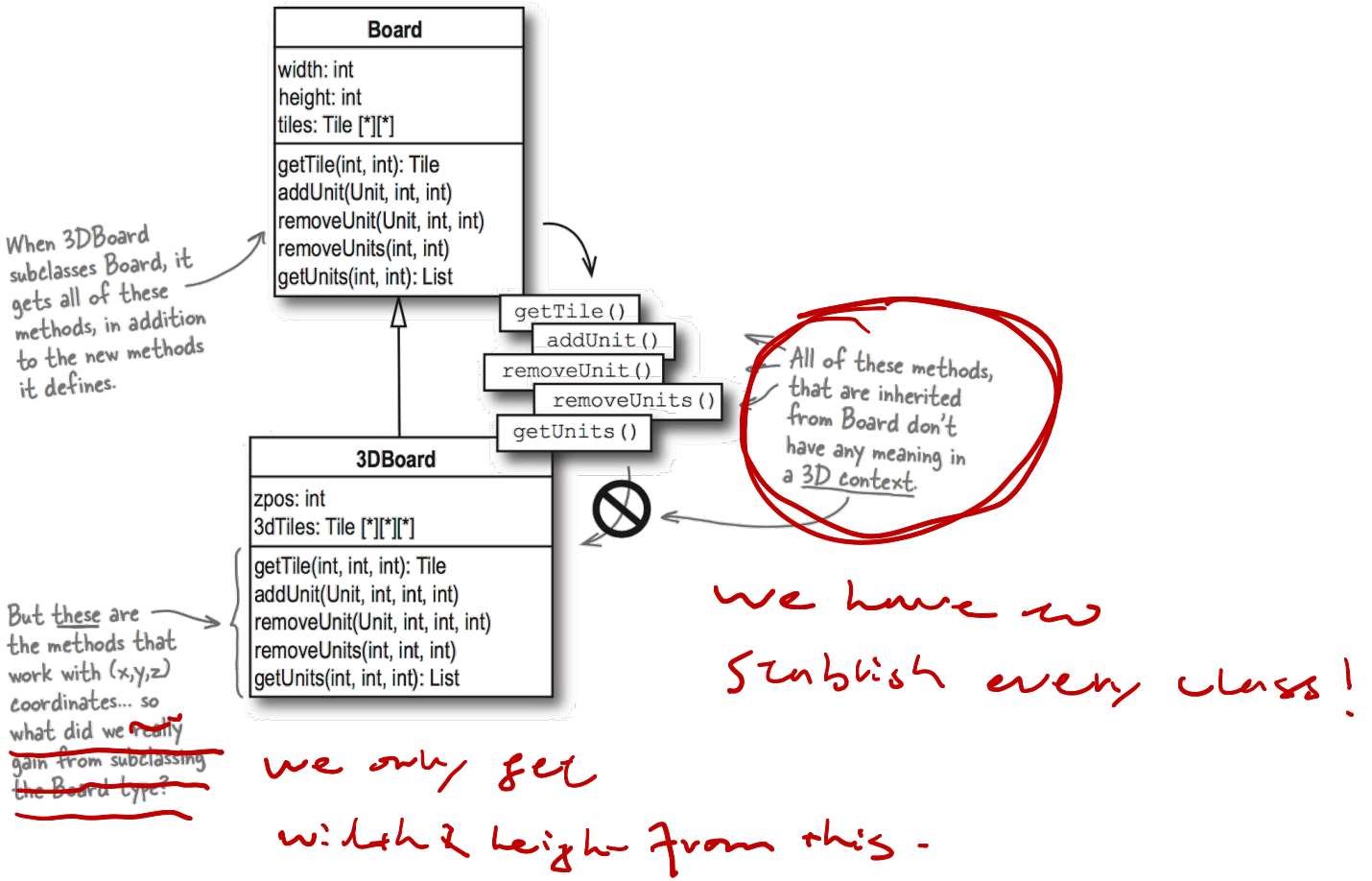
SOLID: Liskov Substitution Principle

- The Liskov Substitution Principle is all about well-designed inheritance
  - When you inherit from a base class, you must be able to substitute your subclass for that base class without affecting program correctness
  - If not, you are misusing inheritance

SOLID: Liskov Substitution Principle



SOLID: Liskov Substitution Principle



SOLID: Liskov Substitution Principle

- The compiler will allow us to substitute `3DBoard` for `Board` just fine:

```
Board* board = new 3DBoard();
```

base class

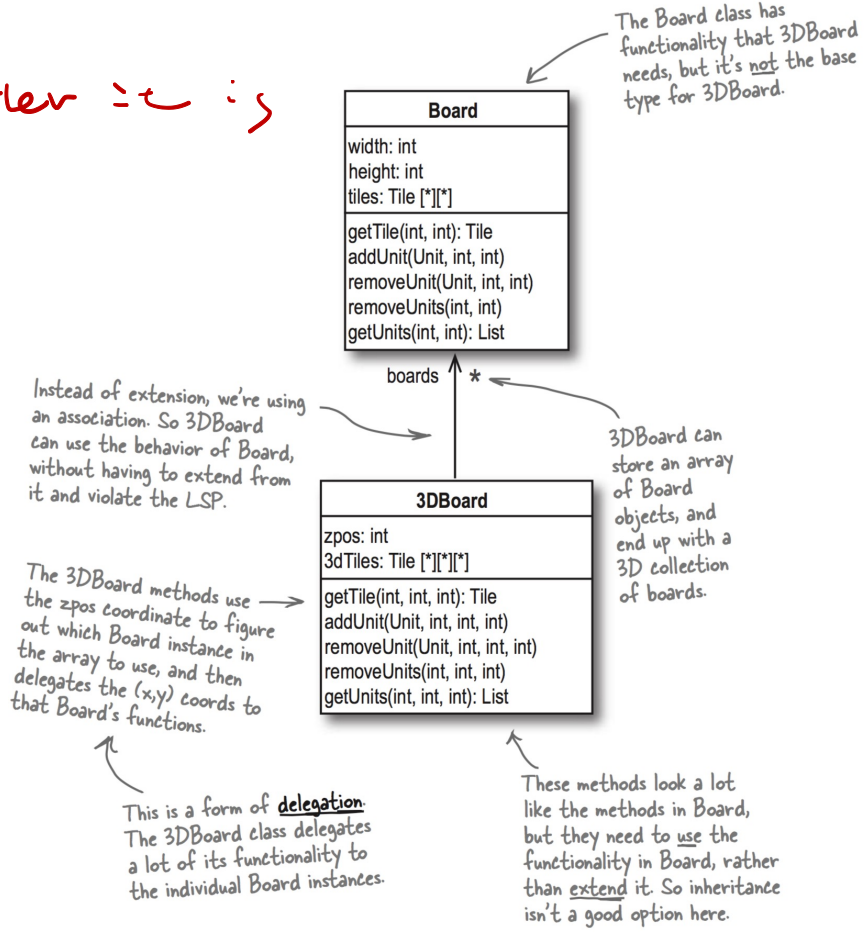
- But, `3DBoard` cannot really stand in for `Board` without affecting program correctness

```
List units = board->getUnits(8, 4);
```

- What does this method mean on `3DBoard`?
  - The Liskov Substitution Principle states that any method on `Board` should be usable on `3DBoard` without affecting correctness
  - `3DBoard` really is not substitutable for `Board`: none of the methods on `Board` will work in a 3D environment

SOLID: Liskov Substitution Principle

use the depthiL
to determine whether it is
a 3D-board.



SOLID: Liskov Substitution Principle

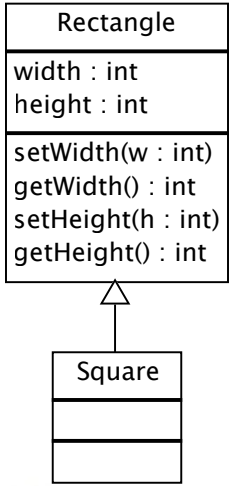
- Suppose we have a class `Rectangle` in our system …

```
class Rectangle {
public:
  void setWidth(int w) {
     this->width = w;
  }
  void setHeight(int h) {
     this->height = h;
  }

// ...
protected:
  int width;
  int height;
};
```

SOLID: Liskov Substitution Principle

- A few months later we realize we need to add a `Square`
  - Does the following hierarchy violate the Liskov Substitution Principle?
  - After all a square IS-A rectangle …



SOLID: Liskov Substitution Principle

- This one is more subtle and requires more consideration
- The first clue that something is wrong:
  - A `Square` does not need both `width` and `height` members, but it inherits them anyways
  - Really, a `Square` only needs a single side length
  - It's not a big deal though … memory is cheap these days, we have lots of RAM, and we won't be making tons of `Square` objects anyways

SOLID: Liskov Substitution Principle

- The second clue:
  - The `setWidth` and `setHeight` methods inherited from `Rectangle` will not be appropriate for `Square`
  - That's okay; we can override them …

```
class Square : public Rectangle {
public:
   void setWidth(int w) {
      this->width = this->height = w;
   }
   void setHeight(int h) {
      this->width = this->height = h;
   }
};
```

SOLID: Liskov Substitution Principle

- Third clue:
  - Our function `f` works for `Rectangle` but not for `Square`

```
void f(Rectangle& r)
{
   r.setWidth(32);  // calls Rectangle::setWidth
}
```

SOLID: Liskov Substitution Principle

- No worries … we can fix this too by making `setWidth` and `setHeight` virtual in the `Rectangle` base class
- The fact that we have to violate our Open/Closed Principle to make this work is another hint that something is wrong …

```
class Rectangle {
public:
  virtual void setWidth(int w) {
     this->width = w;
  }
  virtual void setHeight(int h) {
     this->height = h;
  }
};
```

SOLID: Liskov Substitution Principle

- Fourth clue:
  - Our function `g` works for `Rectangle` but not for `Square`

```
void g(Rectangle& r)
{
  r.setWidth(5);
  r.setHeight(4);
  assert(r.getWidth() * r.getHeight() == 20);
}
```

- The Liskov Substitution Principle says that anywhere we can use the base type, we should be able to use the subclass type without affecting program correctness … does this hold true here?

SOLID: Liskov Substitution Principle

- What gives? In real life, a square IS-A rectangle
- A `Square` object, though, is not a `Rectangle` object
  - In the end, the behaviour of a `Square` is not consistent with the behaviour of a `Rectangle`
  - Behaviour is what software is all about
- Conclusion: IS-A does not tell the whole story
- We should use inheritance when one object behaves like another, rather than just when the IS-A relationship applies