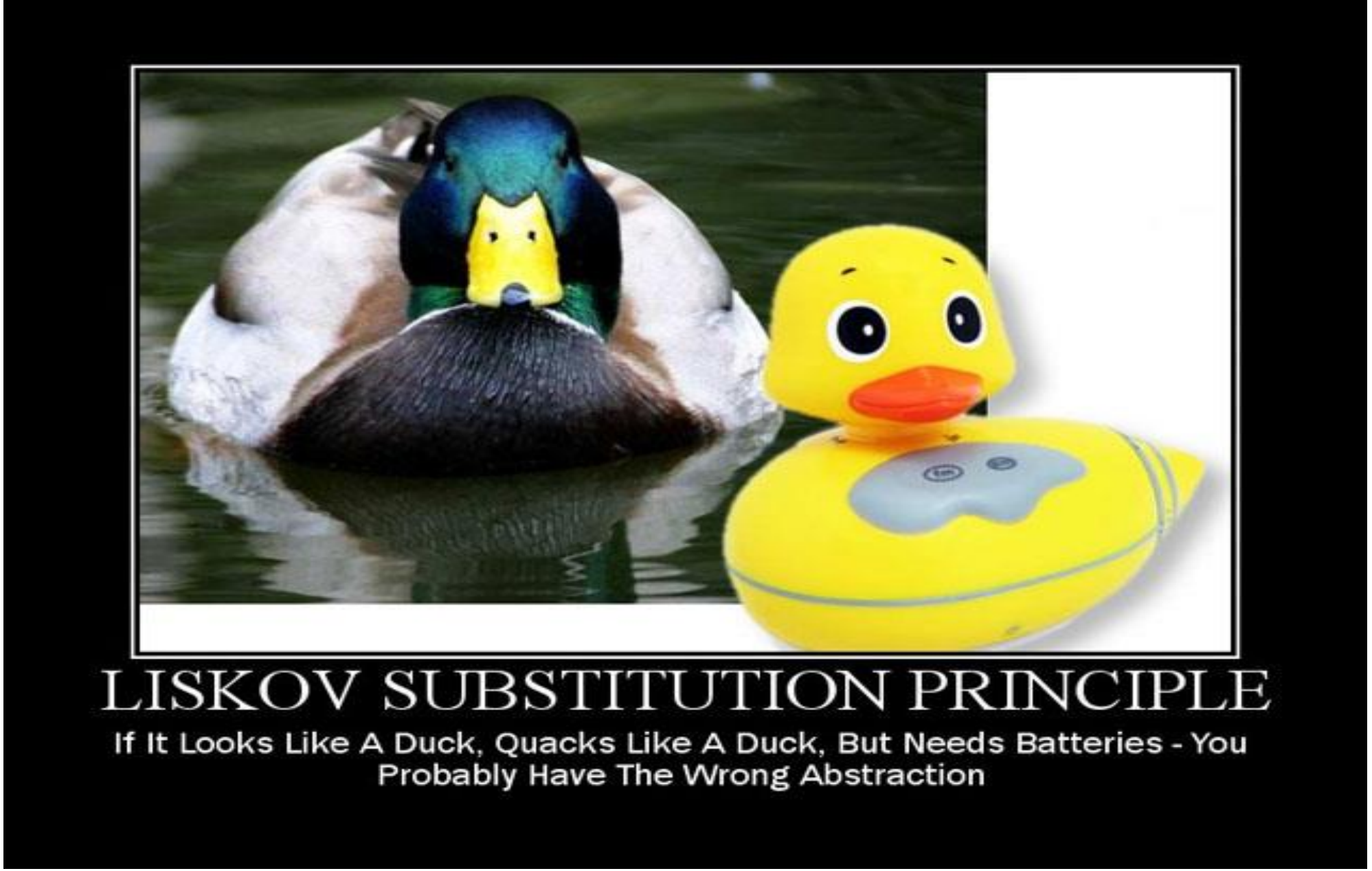
```
CLASS BALL EXTENDS THROWABLE {}
CLASS P{
  P TARGET;
  P(P TARGET) {
     THIS.TARGET=TARGET;
  }
  VOID AIM(BALL BALL) {
    TRY {
      THROW BALL;
    }
    CATCH (BALL B){
      TARGET.AIM(B);
    }
  }
  PUBLIC STATIC VOID MAIN(STRING[] ARGS) {
    P PARENT=NEW P(NULL);
    P CHILD=NEW P(PARENT);
    PARENT.TARGET=CHILD;
    PARENT.AIM(NEW BALL());
  }
}
```

SOLID

"The critical design tool for software development is a mind well educated in design principles"

THE SOLID ACRONYM

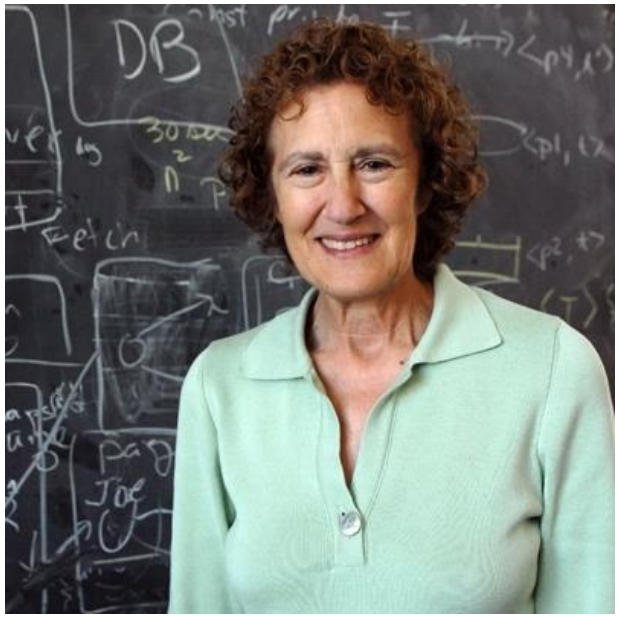
- The SOLID acronym is composed of the first letters of 5 design principles:
  - **S** Single Responsibility Principle (SRP)
  - **O** Open Closed Principle (OCP)
  - **L** Liskov's Substitution Principle (LSP)
  - **I** Interface Segregation Principle (ISP)
  - **D** Dependency Inversion Principle (DIP)


LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

LIVSKOV'S SUBSTITUION PRINCIBLE (LSP)

- LSP: *"In a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of that program"*

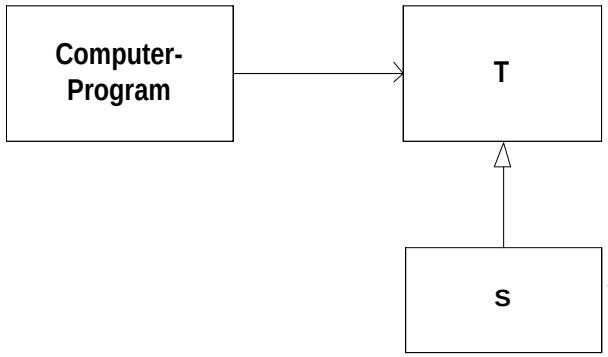
  Barbara Liskov, 1987



LSP - EXPLAINED

- LSP: *In a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of that program*

ComputerProgram uses T and thus expects some specific behavior of T



We can extend ("sub-type") T in a derived class, S, and use it in ComputerProgram instead of S.

However, we must make sure that the behavior that ComputerProgram expects of T is also implemented in S.

If we don't, ComputerProgram is broken because of something changed outside it!

So according to LSP, subtyping should not mean *IS-A* but should mean *IS-SUBSTITUTABLE-FOR*.

LSP: THE CIRCLE-ELLIPSIS PROBLEM

Is a circle an ellipsis?

PUTTING IT ANOTHER WAY

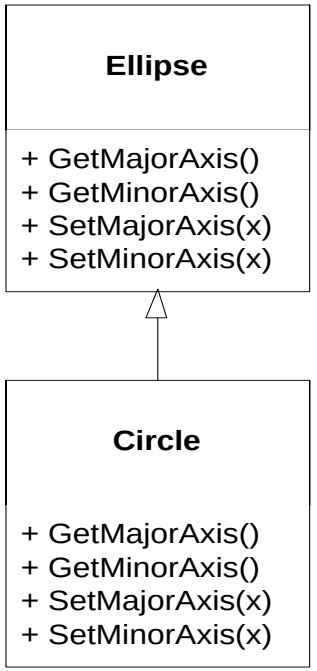
"A routine declaration of a derivative may only replace the original precondition with one **equal or weaker**, and the original postcondition with one **equal or stronger**"

Bertrand Meyer, Design By Contract

Postcondition for `Ellipse.SetMajorAxis(x)`:
`a==x && b == old.b`

Weaker postcondition!

Postcondition for `Circle.SetMajorAxis(x)`:
`a==x && b==x`



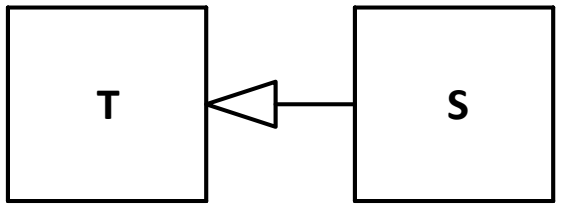
PRE- AND POSTCONDITIONS

"A routine declaration of a derivative may only replace the original precondition with one **equal or weaker**, and the original postcondition with one **equal or stronger**"

Bertrand Meyer, Design By Contract

Stronger precondition

```
void setValue(int val)
{
  Assert(val <= 10)
  ...
}
```




```
void setValue(int val)
{
  Assert(val <= 5)
  ...
}
```

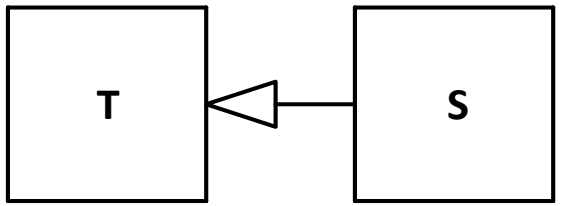
PRE- AND POSTCONDITIONS

"A routine declaration of a derivative may only replace the original precondition with one **equal or weaker**, and the original postcondition with one **equal or stronger**"

Bertrand Meyer, Design By Contract

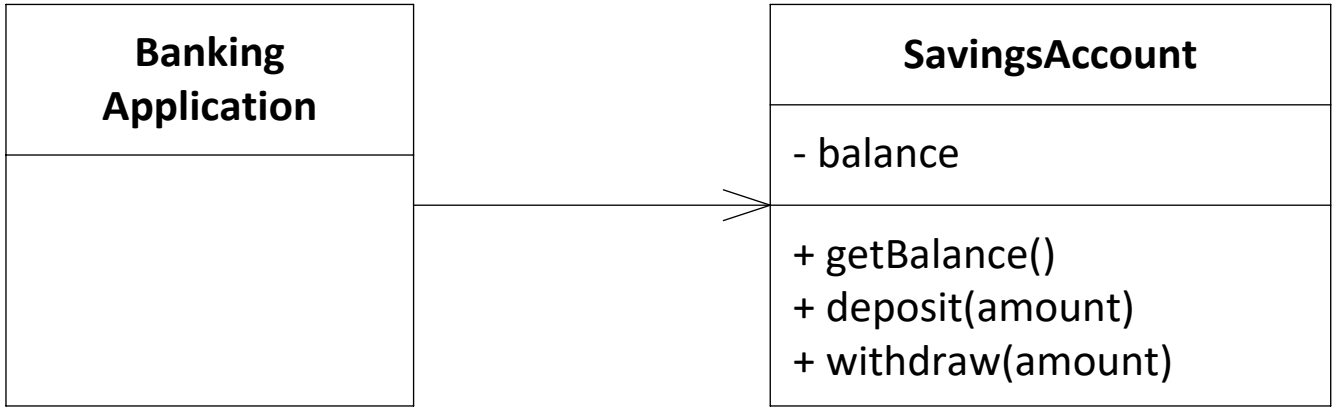
Weaker precondition

```
void setValue(int val)
{
  Assert(val <= 10)
  ...
}
```

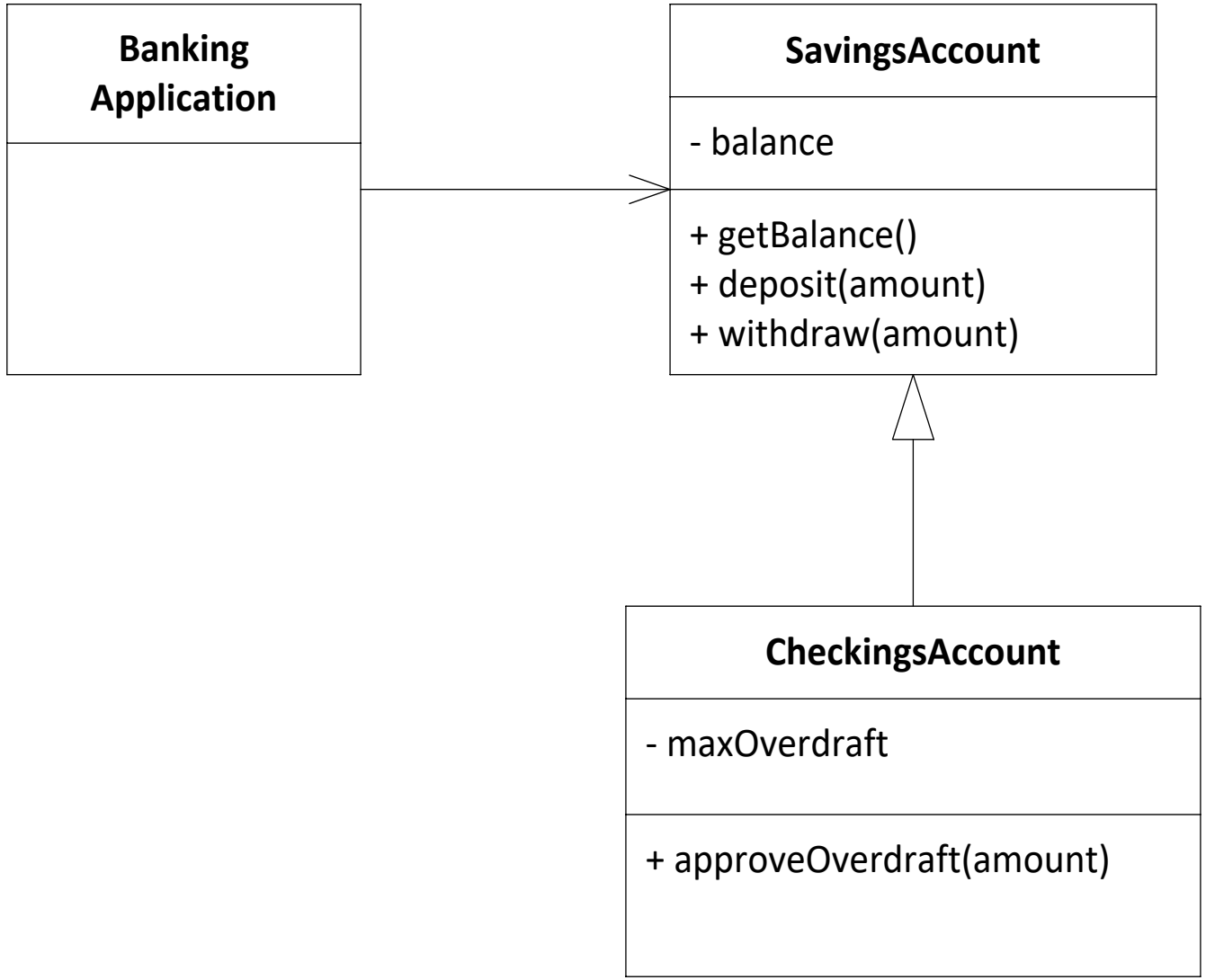



```
void setValue(int val)
{
  Assert(val <= 15)
  ...
}
```

ACCOUNT EXAMPLE



EXTENDING ACCOUNT



Discussion:

Is there a problem?

SUBCLASSING

When you subclass, think about the class assumptions:

- What are the pre- and post conditions for the superclass, in the context it is used?
- What are the class invariants?

- Take care not to break the code by violating those assumptions.

- Think *IS-SUBSTITUTABLE-FOR* instead of *IS-A*

ALTERNATIVE

- Alternative to SOLID because
  - SOLID is hard to apply
    - SRP – vague
    - OCP – replace old code
    - LSP – no surprises
    - ISP – everything is better than one object/interface
    - DIP – reuse is overrated

Instead → Write simple code

CUPID

- **C**omposable
- **U**nix philosophy
- **P**redictable
- **I**diomatic
- **D**omain-based

SOLID - RECAB

- SOLID: 5 principles for good OO design
- S: SRP – Each class/module should only have a single responsibility
- O: OCP – Classes should be open for extension but closed for modification
- L: LSP – Any client of a class should be able to use subclasses of that class with no problems

SOLID - RECAB

- I: ISP – Clients should not be forced to depend on methods they do not use

- D: DIP
  - 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.


ANY
QUESTIONS?

AARHUS
UNIVERSITY

REFERENCES

XKCD: https://xkcd.com/1188/

Babara Livskov: https://news.mit.edu/2009/turing-liskov-0310

Questions: http://sourcesofinsight.com/questions-and-answers-on-the-top-10-leadership-lessons/