Principles of Software Construction: Objects, Design, and Concurrency (Part 1: Designing Classes)

Class-level reuse with inheritance
Behavioral subtyping

Charlie Garrod **Bogdan Vasilescu**

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Homework 1 due tonight 11:59pm
  - Everyone must read and sign our collaboration policy before we can grade
- Reading assignments:
  - Optional due today (UML class diagrams)
  - Mandatory due Tuesday (immutability, defensive copying)
- Homework 2 released tomorrow

Key concepts from Tuesday

Key concepts from Tuesday

- Design for Change such that
  - Classes are *open for extension* and modification without invasive changes
  - Classes encapsulate details likely to change behind (small) stable interfaces
- Design for Division of Labor such that
  - Internal parts can be *developed* independently
  - Internal details of other classes do not need to be *understood*, contract is sufficient
  - Test classes and their contracts separately (unit testing)

Subtype Polymorphism

- A type (e.g. Point) can have many forms (e.g., CartesianPoint, PolarPoint, …)

- Use interfaces to separate expectations from implementation

- All implementations of an interface can be used interchangeably

- This allows flexible **change** (modifications, extensions, reuse) later without changing the client implementation, even in unanticipated contexts

Check your Understanding

```
interface Animal {
    void makeSound();
}
class Dog implements Animal {
    public void makeSound() { System.out.println("bark!"); }
}
class Cow implements Animal {
    public void makeSound() { mew(); }
    public void mew() {System.out.println("Mew!"); }
}

1 Animal a = new Animal();
2 a.makeSound();

3 Dog d = new Dog();
4 d.makeSound();

5 Animal b = new Cow();
6 b.makeSound();
7 b.mew();
```

- What happens?

Static types vs dynamic types

- Static type: how is a variable declared
- Dynamic type: what type has the object in memory when executing the program (we may not know until we execute the program)

```
Point createZeroPoint() {
        if (new Math.Random().nextBoolean())
                return new CartesianPoint(0, 0);
        else    return new PolarPoint(0,0);
}
Point p = createZeroPoint();
p.getX();
p.getAngle();
```

JUnit

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class AdjacencyListTest {
       @Test
       public void testSanityTest(){
              Graph g1 = new AdjacencyListGraph(10)
              Vertex s1 = new Vertex("A");
              Vertex s2 = new Vertex("B");
              assertEquals(true, g1.addVertex(s1));
              assertEquals(true, g1.addVertex(s2));
              assertEquals(true, g1.addEdge(s1, s2));
              assertEquals(s2, g1.getNeighbors(s1)[0]);
       }

       @Test
       public void test….

       private int helperMethod…
}
```

Set up tests

Check expected results

Write testable code

```
//700LOC
public boolean foo() {
    try {
        synchronized () {
            if () {
            } else {
            }
            for () {
                if () {
                    if () {
                        if () {
                            if ()?
                            {
                                if () {
                                    for () {
                                    }
                                }
                            }
                        } else {
                            if () {
                                for () {
                                    if () {
                                    } else {
                                    }
                                    if () {
                                    } else {
                                        if () {
                                        }
                                    }
                                    if () {
                                        if () {
                                            if () {
                                                for () {
                                                }
                                            }
                                        }
                                    } else {
                                    }
                                }
```

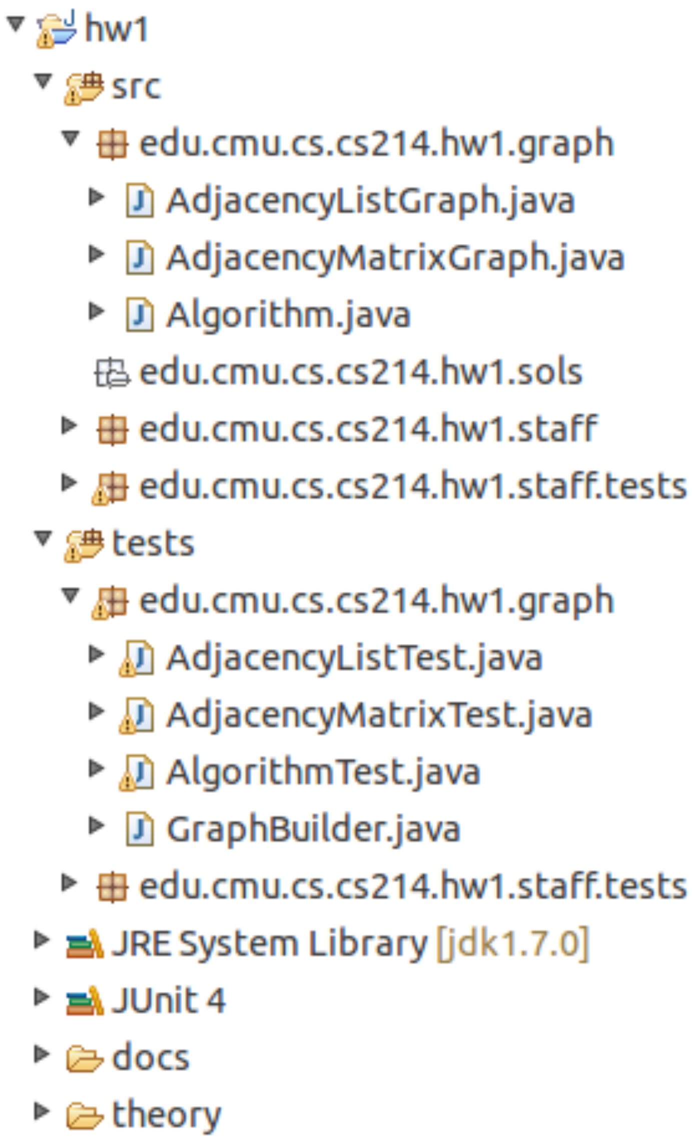
Unit testing as design mechanism

* Code with low complexity
* Clear interfaces and specifications

Source: http://thedailywtf.com/Articles/Coding-Like-the-Tour-de-France.aspx

Test organization

- Conventions (not requirements)
- Have a test class FooTest for each public class Foo
- Have a source directory and a test directory
  - Store FooTest and Foo in the same package
  - Tests can access members with default (package) visibility



Selecting test cases: common strategies

- Read specification
- Write tests for
  - Representative case
  - Invalid cases
  - Boundary conditions
- Are there difficult cases? (error guessing)
  - Stress tests?
  - Complex algorithms?
- Think like an attacker
  - The tester’s goal is to find bugs!
- Prevent regressions

When should you stop writing tests?

- When you run out of money…
- When your homework is due…
- When you can't think of any new test cases...
- The *coverage* of a test suite
  - Trying to test all parts of the implementation
  - Statement coverage
    - Execute every statement, ideally
    - Compare to: method coverage, branch coverage, path coverage

Code coverage metrics

- Method coverage – coarse
- Branch coverage – fine
- Path coverage – too fine
  - Cost is high, value is low
  - (Related to *cyclomatic complexity*)

Method Coverage

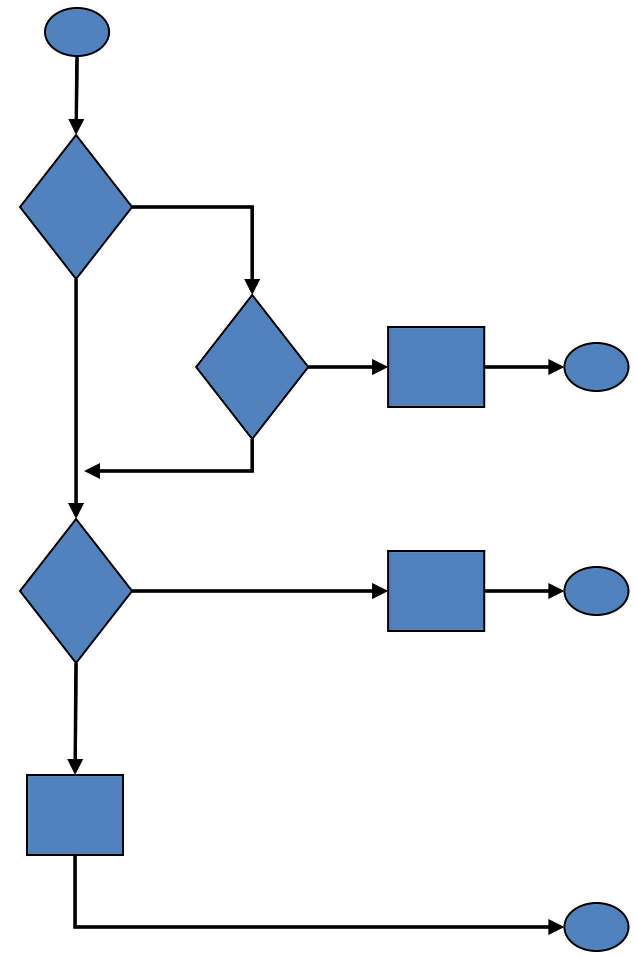
- Trying to execute each method as part of at least one test

```
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney= (Money)anObject;
        return aMoney.currency().equals(currency())
                        && amount() == aMoney.amount();
    }
    return false;
}
```

- Does this guarantee correctness?

Structure of Code Fragment to Test

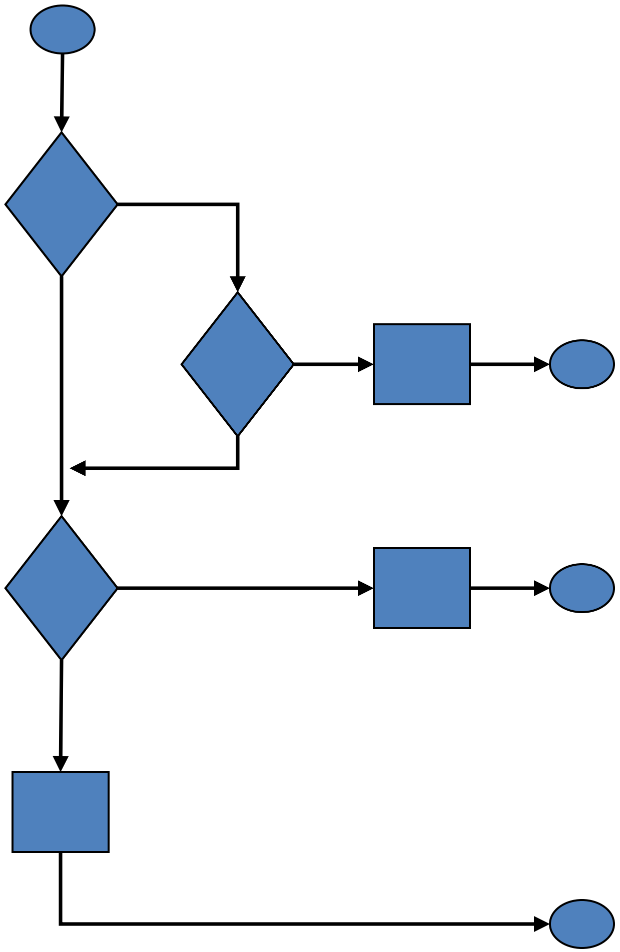
```java
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney= (Money)anObject;
        return aMoney.currency().equals(currency())
                        && amount() == aMoney.amount();
    }
    return false;
}
```



Flow chart diagram for
`junit.samples.money.Money.equals`

Statement Coverage

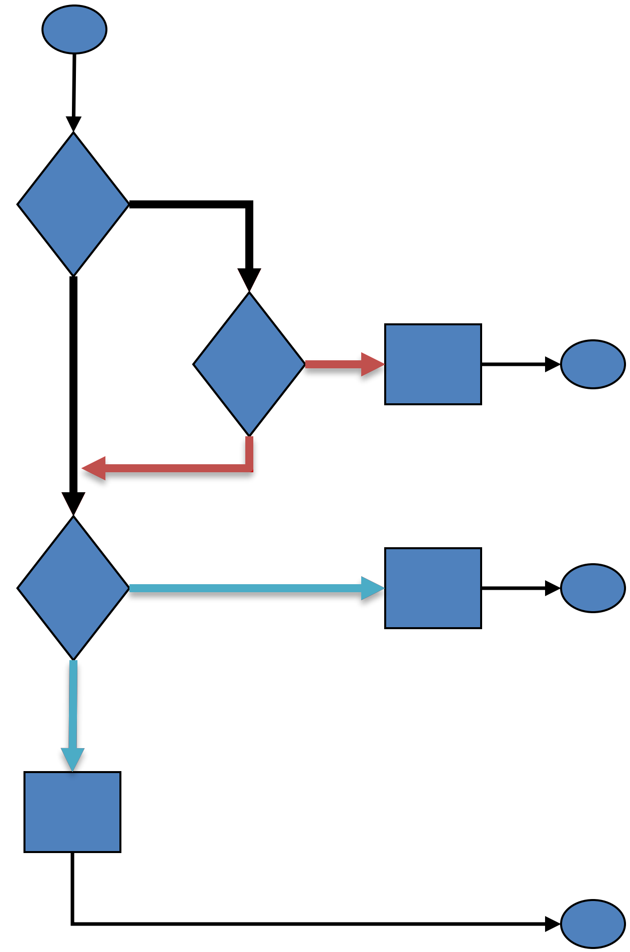
- Statement coverage
  - What portion of program statements (nodes) are touched by test cases



```
}
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney= (Money)anObject;
        return aMoney.currency().equals(currency())
                        && amount() == aMoney.amount();
    }
    return false;
}
```

Branch Coverage

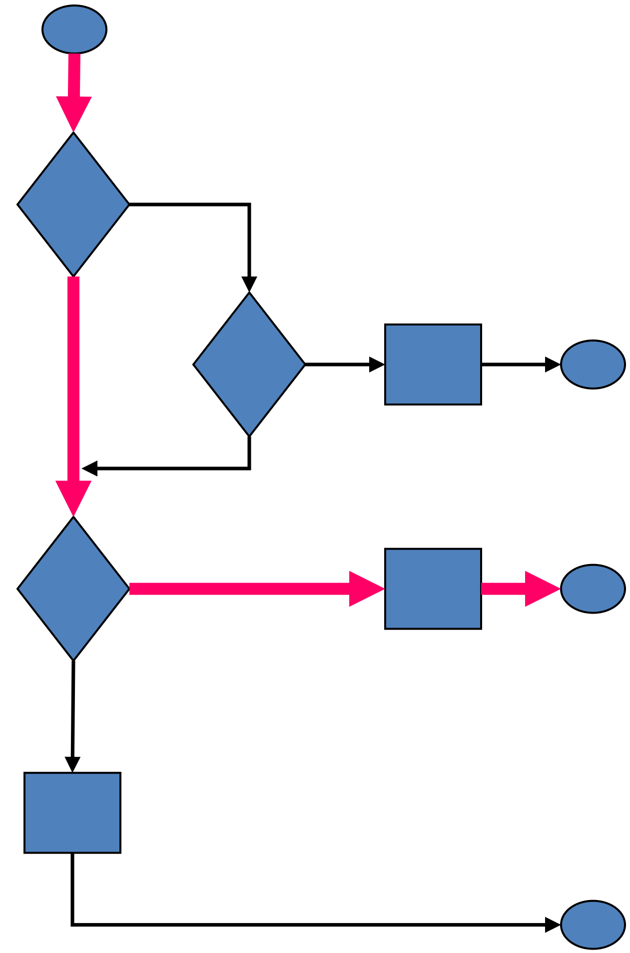
- Branch coverage
  - What portion of condition branches are covered by test cases?
  - Multicondition coverage – all boolean combinations of tests are covered



```java
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney= (Money)anObject;
        return aMoney.currency().equals(currency())
                            && amount() == aMoney.amount();
    }
    return false;
```

Path Coverage

- Path coverage
  - What portion of all possible paths through the program are covered by tests?



```java
}
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney= (Money)anObject;
        return aMoney.currency().equals(currency())
                            && amount() == aMoney.amount();
    }
    return false;
}
```

Packages

All
net.sourceforge.cobertura.ant
net.sourceforge.cobertura.check
net.sourceforge.cobertura.coveragedat
net.sourceforge.cobertura.instrument
net.sourceforge.cobertura.merge
net.sourceforge.cobertura.reporting
net.sourceforge.cobertura.reporting.htr
net.sourceforge.cobertura.reporting.htr
net.sourceforge.cobertura.reporting.xm
net.sourceforge.cobertura.util

All Packages

Classes

AntUtil *(88%)*
Archive *(100%)*
ArchiveUtil *(80%)*
BranchCoverageData *(N/A)*
CheckTask *(0%)*
ClassData *(N/A)*
ClassInstrumenter *(94%)*
ClassPattern *(100%)*
CoberturaFile *(73%)*
CommandLineBuilder *(96%)*
CommonMatchingTask *(88%)*
ComplexityCalculator *(100%)*
ConfigurationUtil *(50%)*
CopyFiles *(87%)*
CoverageData *(N/A)*
CoverageDataContainer *(N/A)*
CoverageDataFileHandler *(N/A)*
CoverageRate *(0%)*
ExcludeClasses *(100%)*
FileFinder *(96%)*
FileLocker *(0%)*
FirstPassMethodInstrumenter *(100%)*
HTMLReport *(94%)*
HasBeenInstrumented *(N/A)*
Header *(80%)*

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Compl
All Packages	55	75%	1625/2179	64%	472/738	
net.sourceforge.cobertura.ant	11	52%	170/330	43%	40/94	
net.sourceforge.cobertura.check	3	0%	0/150	0%	0/76	
net.sourceforge.cobertura.coveragedata	13	N/A	N/A	N/A	N/A	
net.sourceforge.cobertura.instrument	10	90%	460/510	75%	123/164	
net.sourceforge.cobertura.merge	1	86%	30/35	88%	14/16	
net.sourceforge.cobertura.reporting	3	87%	116/134	80%	43/54	
net.sourceforge.cobertura.reporting.html	4	91%	475/523	77%	156/202	
net.sourceforge.cobertura.reporting.html.files	1	87%	39/45	62%	5/8	
net.sourceforge.cobertura.reporting.xml	1	100%	155/155	95%	21/22	
net.sourceforge.cobertura.util	9	60%	175/291	69%	70/102	
someotherpackage	1	83%	5/6	N/A	N/A	

Report generated by Cobertura 1.9 on 6/9/07 12:37 AM.

Check your understanding

- Write test cases to achieve 100% line coverage but not 100% branch coverage

```
void foo(int a, int b) {
    if (a == b)
        a = a * 2;
    if (a + b > 10)
        return a - b;
    return a + b;
}
```

Check your understanding

- Write test cases to achieve 100% line coverage <u>and also</u> 100% branch coverage

```
void foo(int a, int b) {
    if (a == b)
        a = a * 2;
    if (a + b > 10)
        return a - b;
    return a + b;
}
```

Check your understanding

- Write test cases to achieve 100% line coverage and 100% branch coverage and 100% path coverage

```
void foo(int a, int b) {
    if (a == b)
        a = a * 2;
    if (a + b > 10)
        return a - b;
    return a + b;
}
```

Coverage metrics: useful but dangerous

- **Can give false sense of security**
- Examples of what coverage analysis could miss
  - – Data values
  - – Concurrency issues – race conditions etc.
  - – Usability problems
  - – Customer requirements issues

Testing, Static Analysis, and Proofs

- Testing
  - Observable properties
  - Verify program for one execution
  - Manual development with automated regression
  - Most practical approach now
  - Does not find all problems (unsound)

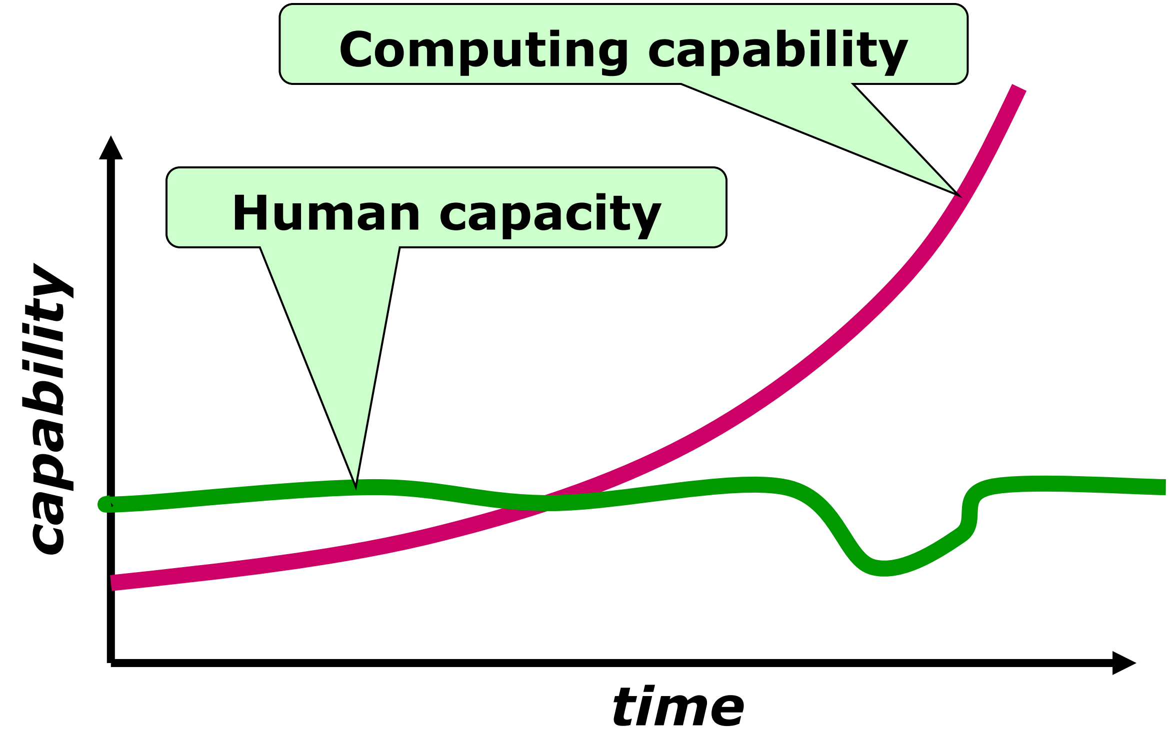
- Static Analysis
  - Analysis of all possible executions
  - Specific issues only with conservative approx. and bug patterns
  - Tools available, useful for bug finding
  - Automated, but unsound and/or incomplete

- Proofs (Formal Verification)
  - Any program property
  - Verify program for all executions
  - Manual development with automated proof checkers
  - Practical for small programs, may scale up in the future
  - Sound and complete, but not automatically decidable

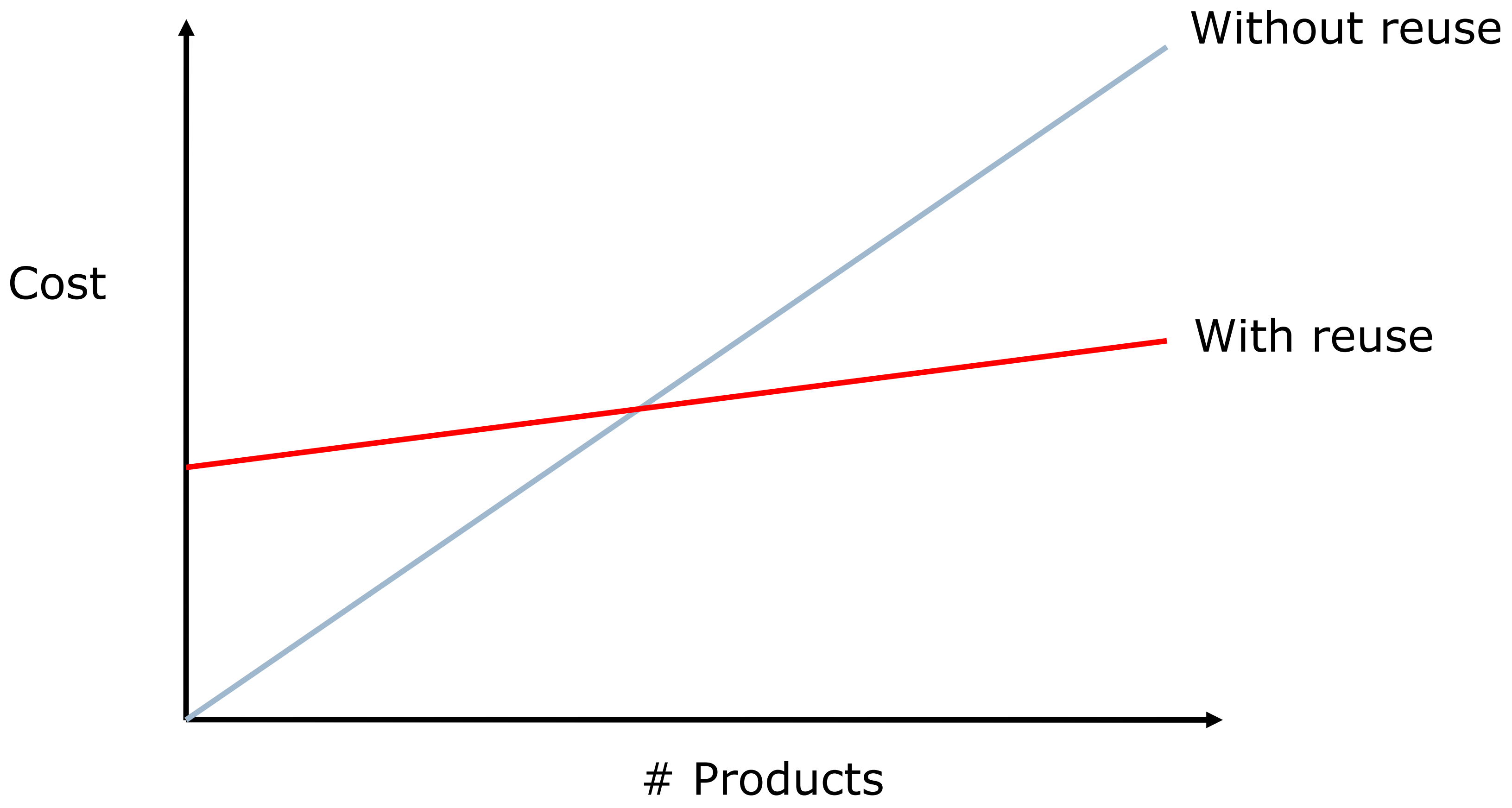
What strategy to use in your project?

DESIGN FOR REUSE

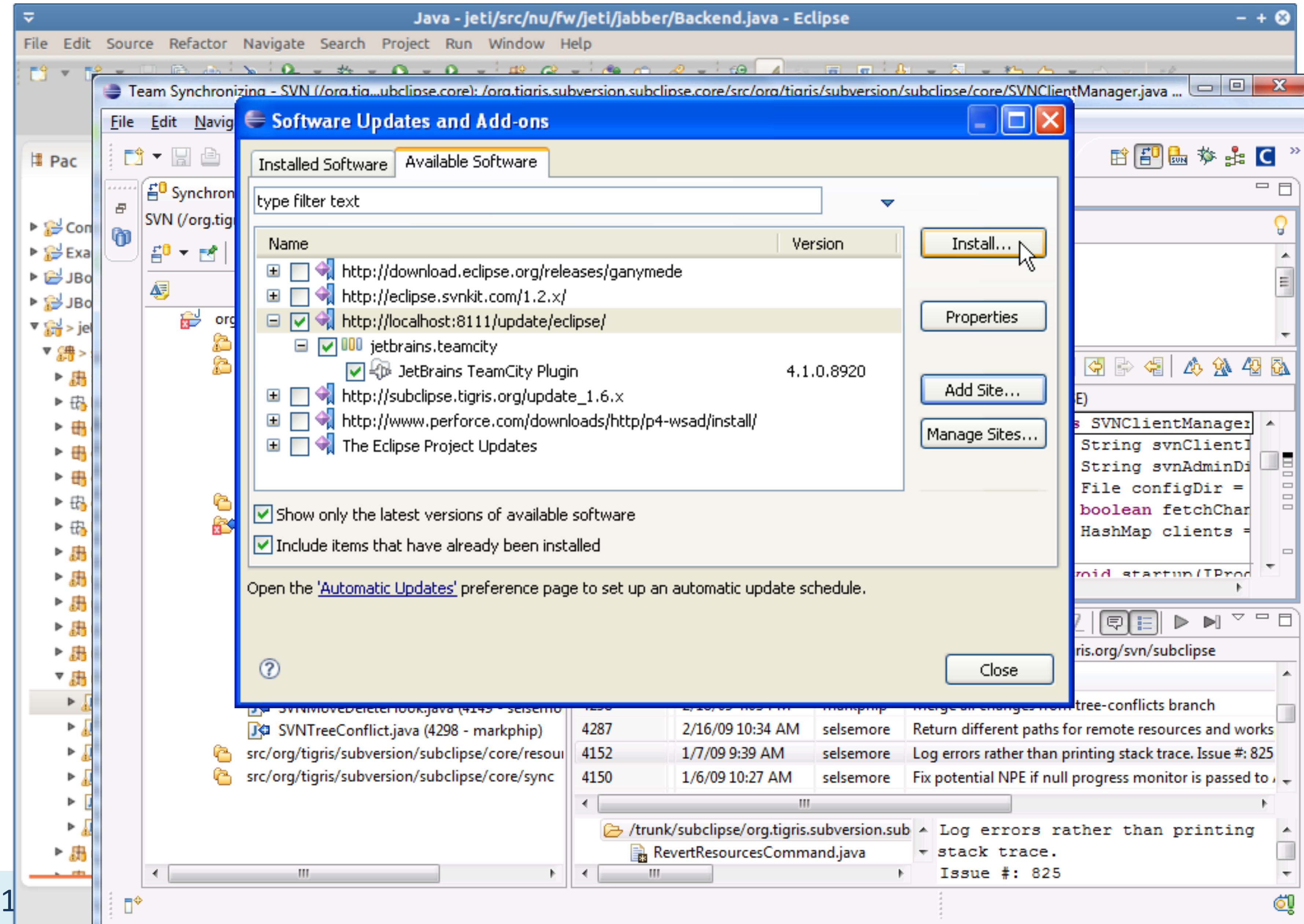
The limits of exponentials



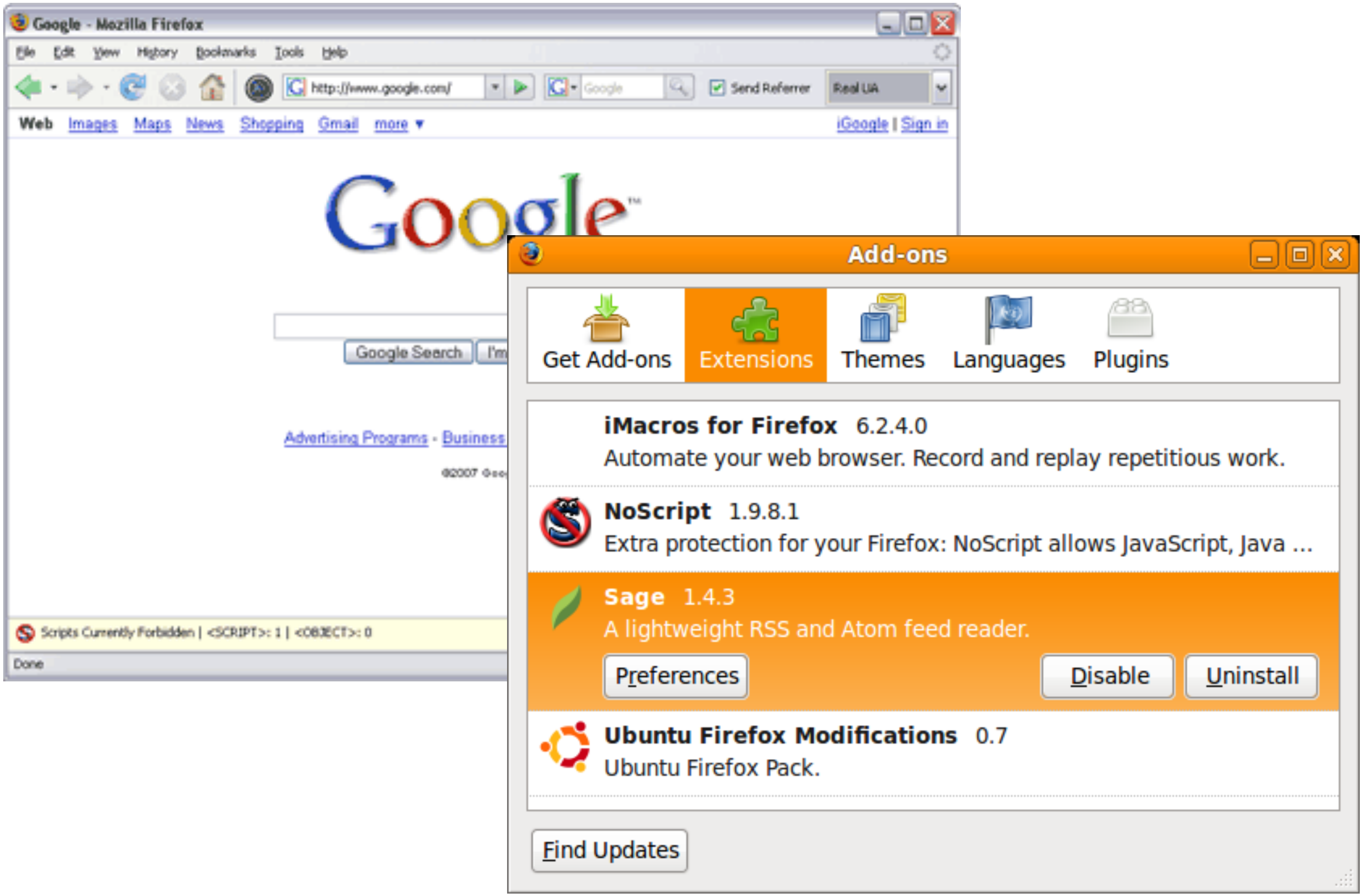
The promise of reuse:



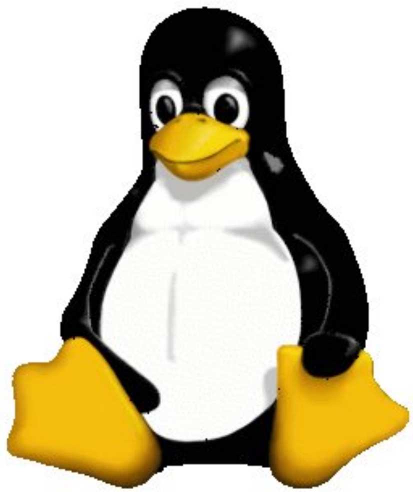
Reuse: Family of development tools



Reuse: Web browser extensions



Reuse and variation: Flavors of Linux



Today: Class-level reuse with inheritance

- Inheritance
  - Java-specific details for inheritance
- Behavioral subtyping: Liskov's Substitution Principle

- Next week:
  - Delegation
  - Design patterns for improved class-level reuse
- Later in the course:
  - System-level reuse with libraries and frameworks

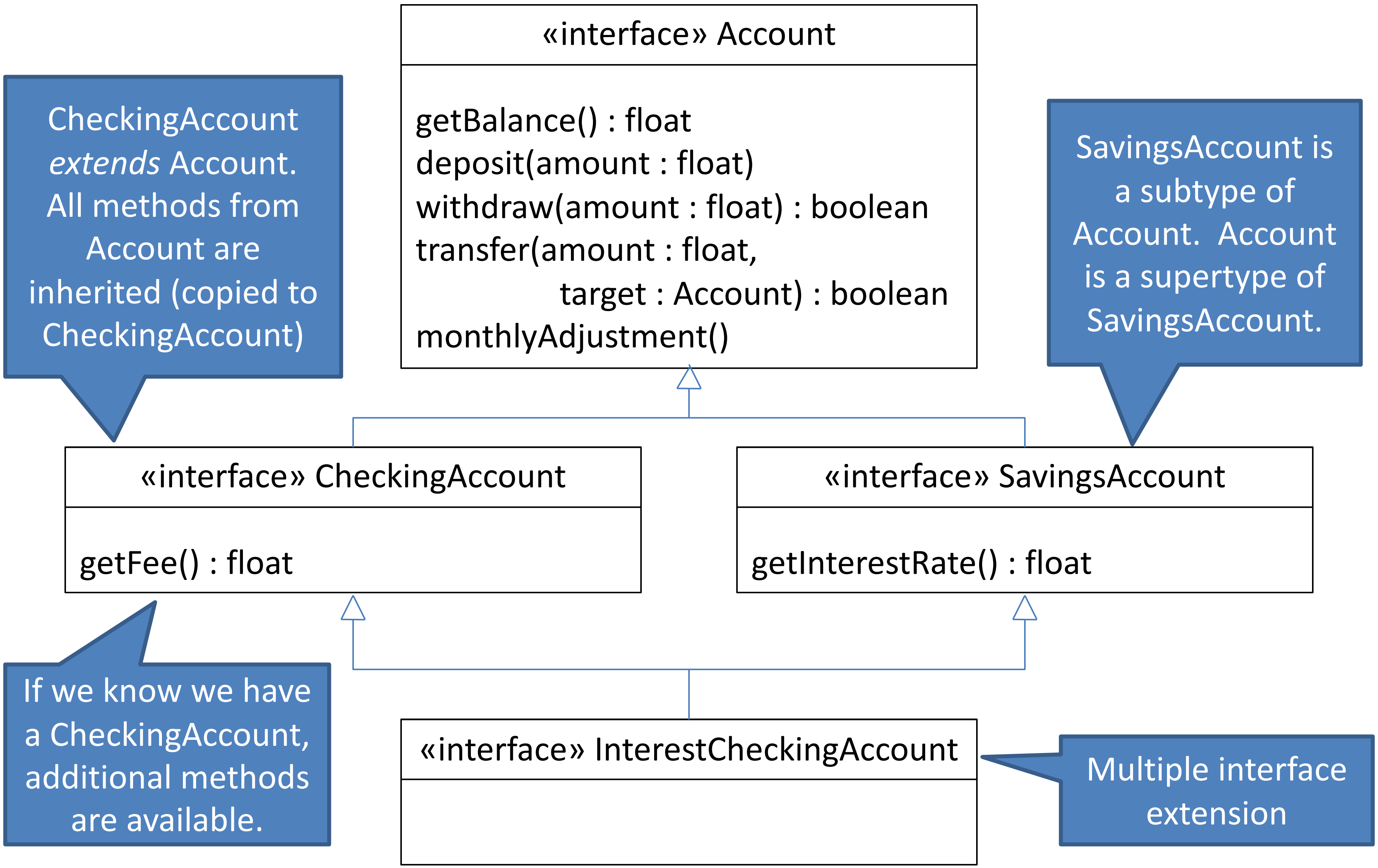
IMPLEMENTATION INHERITANCE AND ABSTRACT CLASSES

Variation in the real world: types of bank accounts

```
«interface» CheckingAccount
---------------------------
getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float,
           target : Account) : boolean
getFee() : float
```

```
«interface» SavingsAccount
--------------------------
getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float,
           target : Account) : boolean
getInterestRate() : float
```

Better: Interface inheritance for an account type hierarchy



Interface inheritance for an account type hierarchy

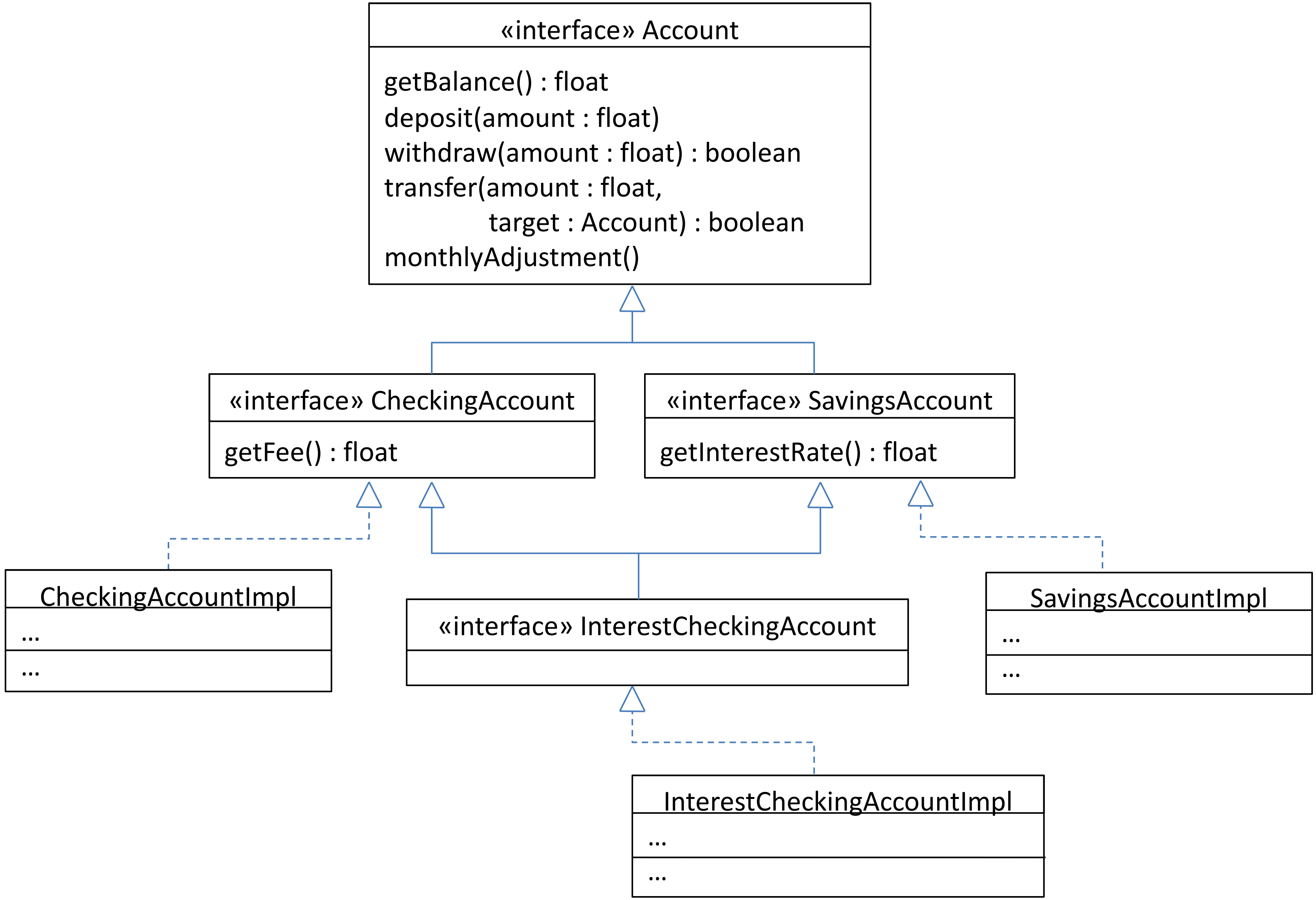
```
public interface Account {
    public long getBalance();
    public void  deposit(long amount);
    public boolean withdraw(long amount);
    public boolean transfer(long amount, Account target);
    public void monthlyAdjustment();
}

public interface CheckingAccount extends Account {
    public long getFee();
}

public interface SavingsAccount extends Account {
    public double getInterestRate();
}

public interface InterestCheckingAccount
                      extends CheckingAccount, SavingsAccount {
}
```

The power of object-oriented interfaces

- Subtype polymorphism
  - Different kinds of objects can be treated uniformly by client code
  - Each object behaves according to its type
    - e.g., if you add new kind of account, client code does not change:

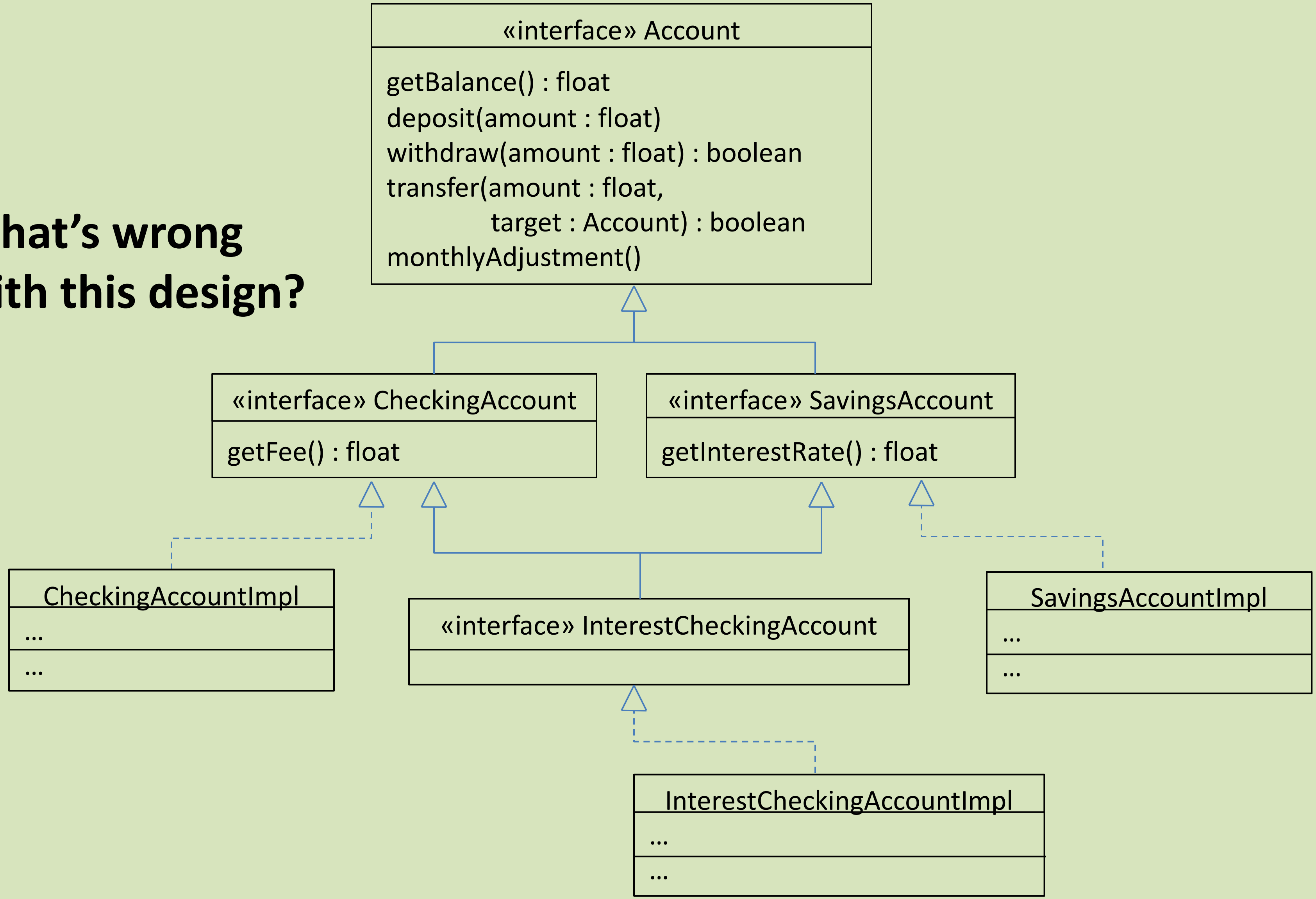
```
If today is the last day of the month:
   For each acct in allAccounts:
      acct.monthlyAdjustment();
```

Implementation inheritance for code reuse



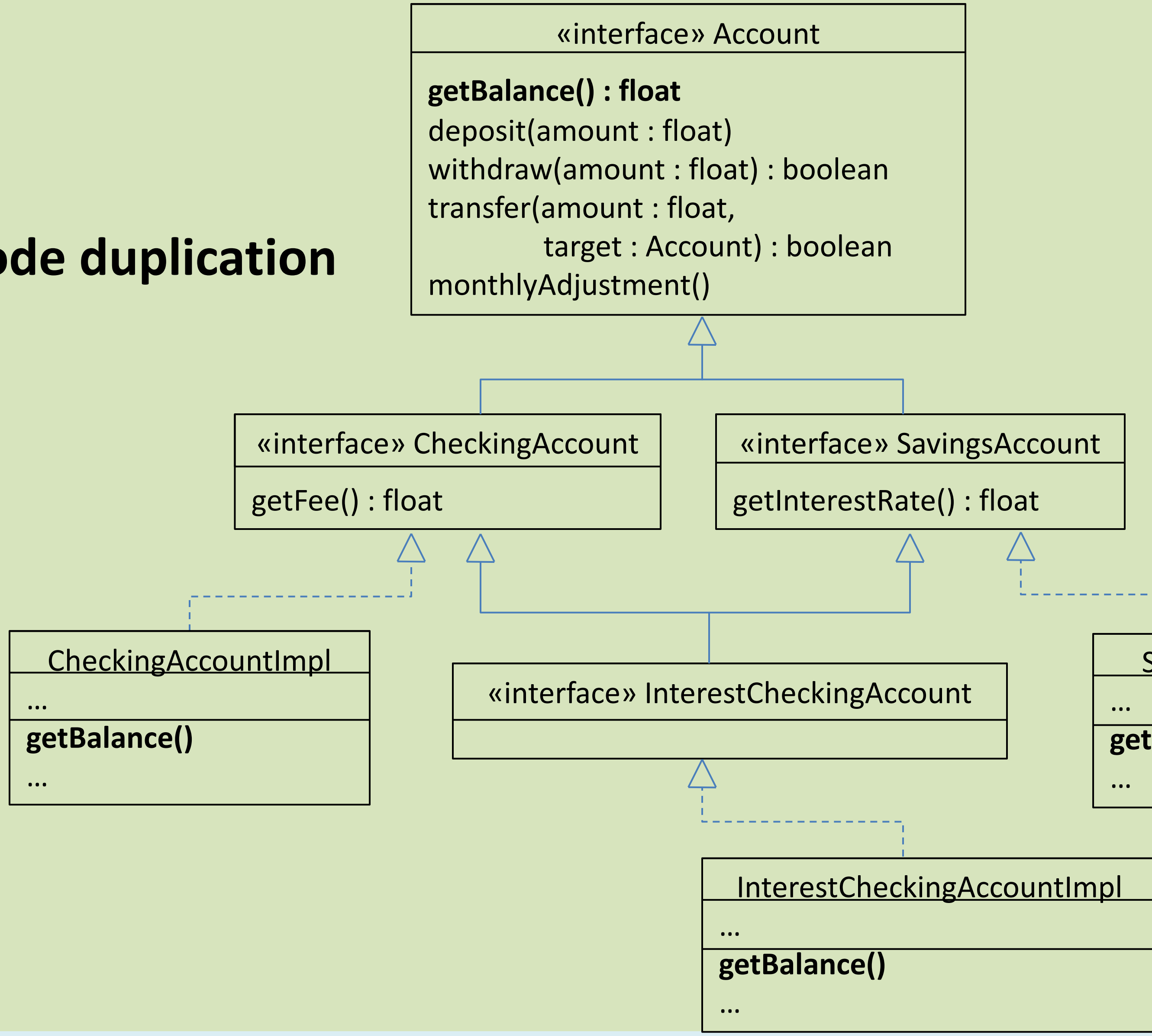
Implementation inheritance for code reuse

What's wrong with this design?



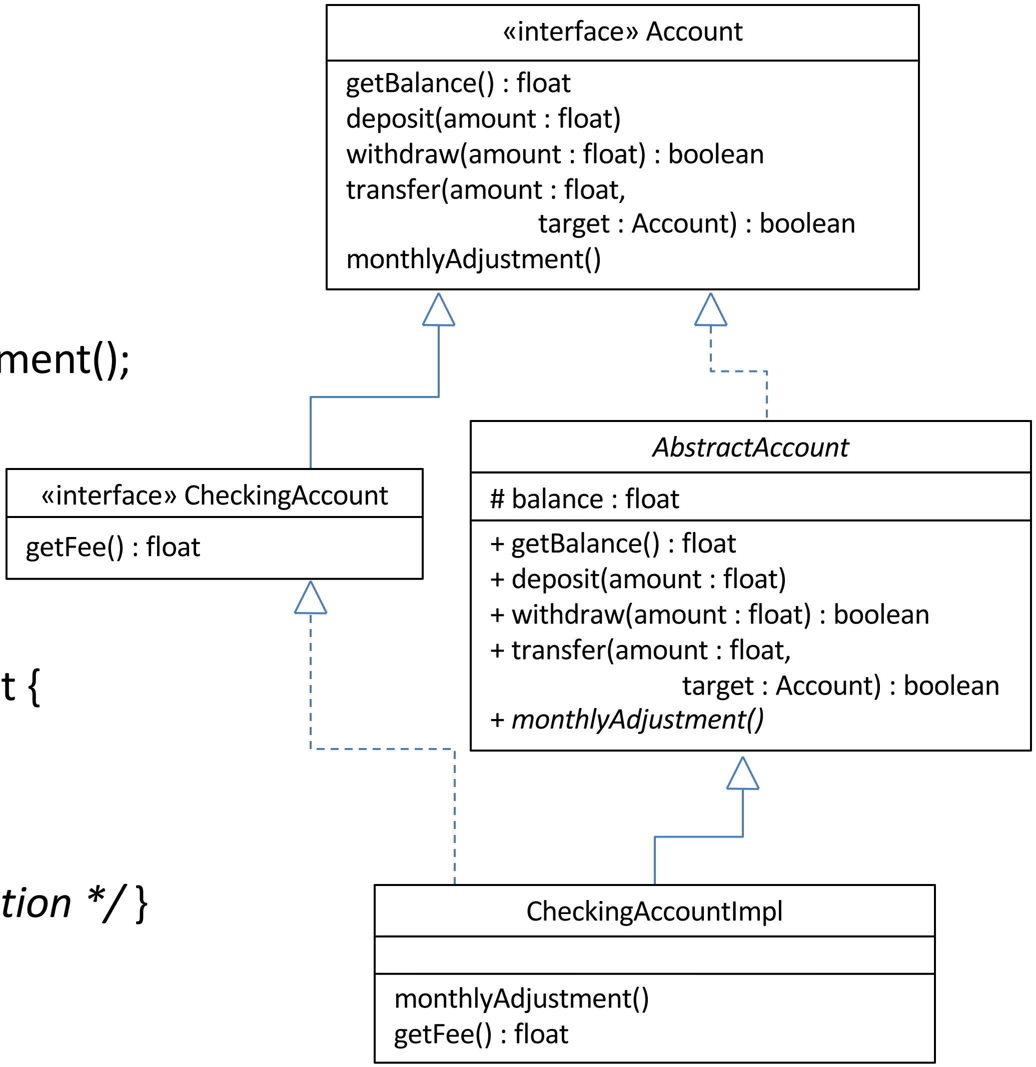
Implementation inheritance for code reuse

Code duplication



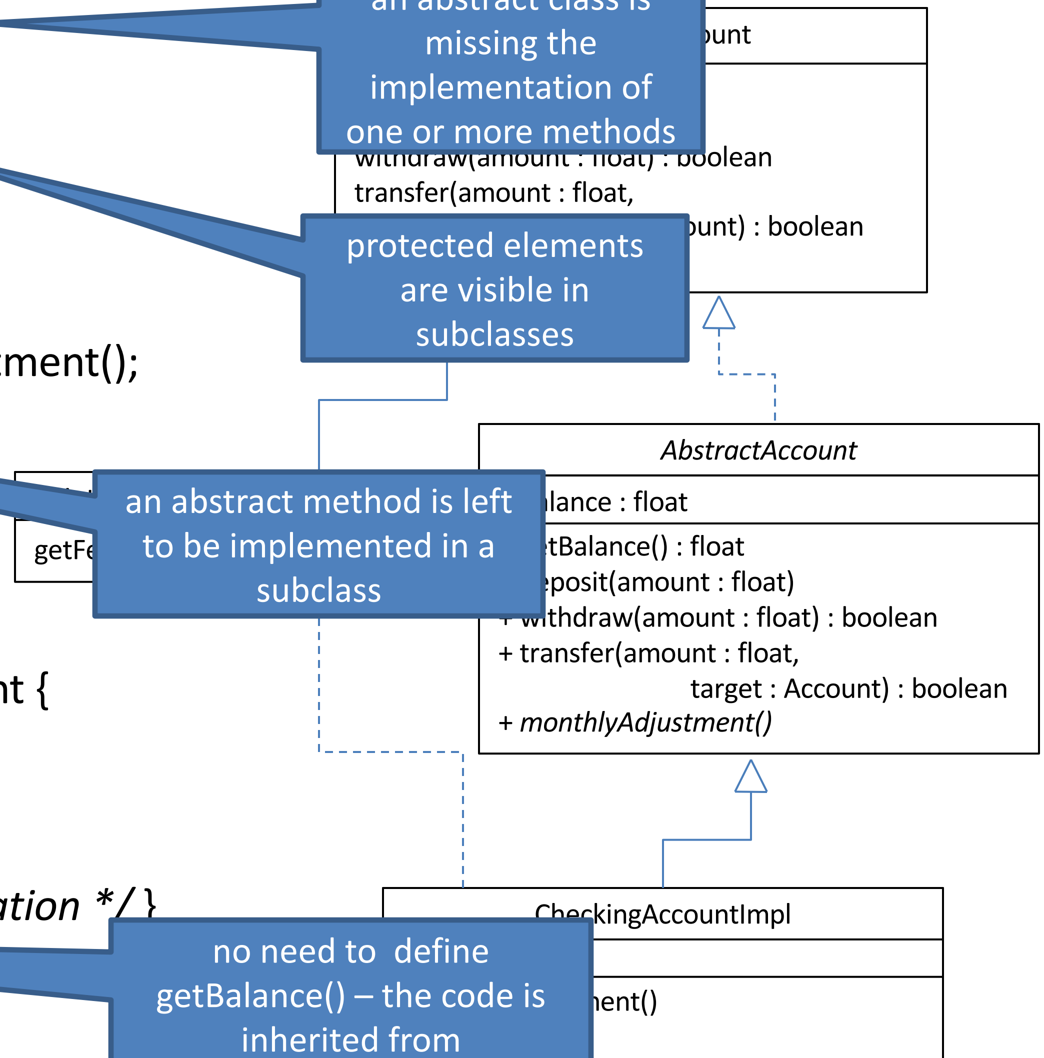
Better: Reuse abstract account code

```
public abstract class AbstractAccount
          implements Account {
    protected float balance = 0.0;
    public float getBalance() {
          return balance;
    }
    abstract public void monthlyAdjustment();
    // other methods…
}

public class CheckingAccountImpl
          extends AbstractAcount
          implements CheckingAccount {
    public void monthlyAdjustment() {
          balance -= getFee();
    }
    public float getFee() { /* fee calculation */ }
}
```



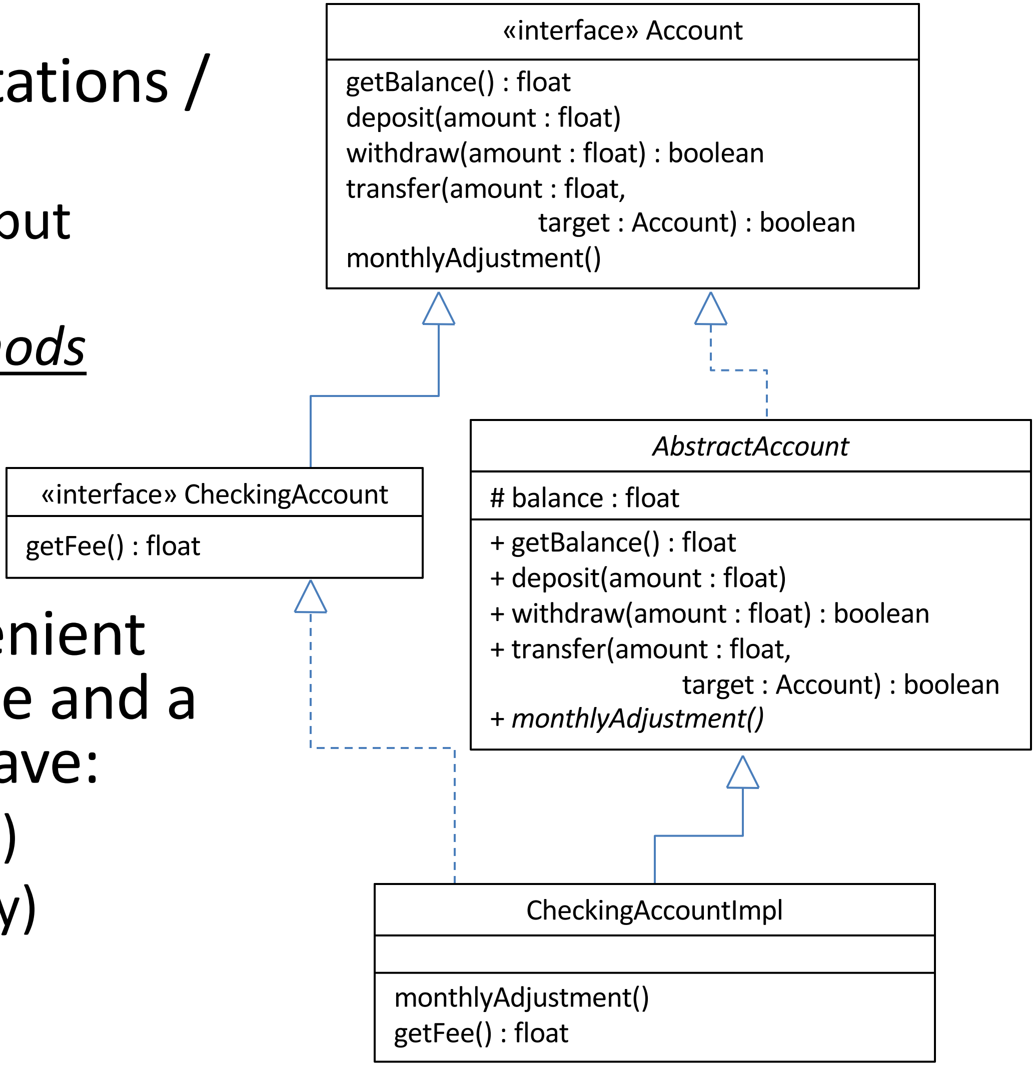
Better: Reuse abstract account code

```
public abstract class AbstractAccount
            implements Account {
    protected float balance = 0.0;
    public float getBalance() {
            return balance;
    }
    abstract public void monthlyAdjustment();
    // other methods..
}

public class CheckingAccountImpl
            extends AbstractAcount
            implements CheckingAccount {
    public void monthlyAdjustment() {
            balance -= getFee();
    }
    public float getFee() { /* fee calculation */ }
}
```



Interfaces vs Abstract Classes vs Concrete Classes

- An *interface* defines expectations / commitment for clients
  - Java: can declare methods but cannot implement them
  - Methods are *abstract methods*

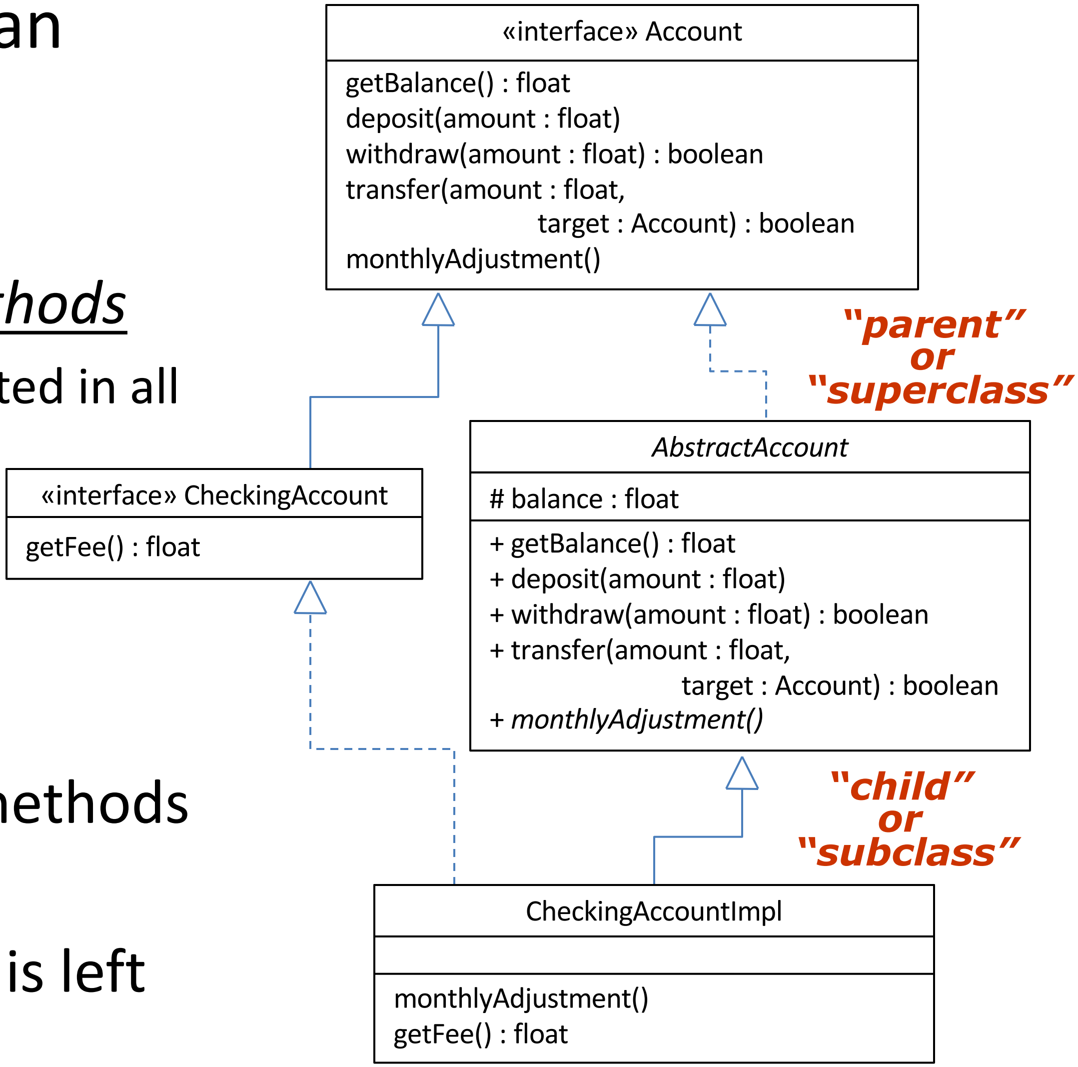


- An ***abstract class*** is a convenient hybrid between an interface and a full implementation. Can have:
  - Abstract methods (no body)
  - Concrete methods (w/ body)
  - Data fields

Interfaces vs Abstract Classes vs Concrete Classes

- Unlike a concrete class, an *abstract class* ...
  - <u>*Cannot be instantiated*</u>
  - <u>*Can declare abstract methods*</u>
    - Which *must* be implemented in all *concrete* subclasses

- An abstract class may <u>*implement*</u> an interface
  - But need not define all methods of the interface
  - Implementation of them is left to subclasses



Aside: Inheritance and subtyping

- Inheritance is for ***code reuse***
  - Write code once and only once
  - Superclass features implicitly available in subclass

```
class A extends B
```

- Subtyping is for ***polymorphism***
  - Accessing objects the same way, but getting different behavior
  - Subtype is substitutable for supertype

```
class A implements I
class A extends B
```

CLASS INVARIANTS

58

Recall: Data Structure Invariants (cf. 122)

```
struct list {
   elem data;
   struct list* next;
};
struct queue {
   list front;
   list back;
};

bool is_queue(queue Q) {
   if (Q == NULL) return false;
   if (Q->front == NULL || Q->back == NULL) return false;
   return is_segment(Q->front, Q->back);
}
```

Recall: Data Structure Invariants (cf. 122)

- Properties of the Data Structure
- Should always hold before and after method execution
- May be invalidated temporarily during method execution

```
void enq(queue Q, elem s)
//@requires is_queue(Q);
//@ensures is_queue(Q);
{ … }
```

Class Invariants

- Properties about the fields of an object
- Established by the constructor
- Should always hold before and after execution of public methods
  - May be invalidated temporarily during method execution

Class Invariants

- Properties about the fields of an object
- Established by the constructor
- Should always hold before and after execution of

rari

```
public class SimpleSet {

    int contents[];
    int size;

    //@ ensures sorted(contents);
    SimpleSet(int capacity) { … }

    //@ requires sorted(contents);
    //@ ensures sorted(contents);
    boolean add(int i) { … }

    //@ requires sorted(contents);
    //@ ensures sorted(contents);
    boolean contains(int i) { … }
}
```

```
public class SimpleSet {

    int contents[];
    int size;

    //@invariant sorted(contents);

    SimpleSet(int capacity) { … }

    boolean add(int i) { … }

    boolean contains(int i) { … }
}
```

BEHAVIORAL SUBTYPING

“SHOULD I BE INHERITING FROM THIS TYPE?”

Behavioral subtyping (Liskov Substitution Principle)

Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

Barbara Liskov

- Applies to specified behavior:
  - Same or stronger invariants
  - Same or stronger postconditions for all methods
  - Same or weaker preconditions for all methods

- e.g., Compiler-enforced rules in Java:
  - Subtypes can add, but not remove methods
  - Concrete class must implement all undefined methods
  - Overriding method must return same type or subtype
  - Overriding method must accept the same parameter types
  - Overriding method may not throw additional exceptions

This is called the *Liskov Substitution Principle*.

Behavioral subtyping in a nutshell

- If `Cowboy.draw()` overrides `Circle.draw()` somebody gets hurt!

Car is a behavioral subtype of Vehicle

```
abstract class Vehicle {
    int speed, limit;

    //@ invariant speed < limit;








    //@ requires speed != 0;
    //@ ensures speed < \old(speed)
    void brake();
}
```

```
class Car extends Vehicle {
    int fuel;
    boolean engineOn;
    //@ invariant speed < limit;
    //@ invariant fuel >= 0;

    //@ requires fuel > 0 && !engineOn;
    //@ ensures engineOn;
    void start() { … }

    void accelerate() { … }

    //@ requires speed != 0;
    //@ ensures speed < \old(speed)
    void brake() { … }
}
```

- **Subclass fulfills the same invariants (and additional ones)**
- **Overridden method has the same pre and postconditions**

Hybrid is a behavioral subtype of Car

```
class Car extends Vehicle {
     int fuel;
     boolean engineOn;
     //@ invariant fuel >= 0;

     //@ requires fuel > 0 && !engineOn;
     //@ ensures engineOn;
     void start() { … }

     void accelerate() { … }

     //@ requires speed != 0;
     //@ ensures speed < old(speed)
     void brake() { … }
}
```

```
class Hybrid extends Car {
     int charge;
     //@ invariant charge >= 0;

     //@ requires (charge > 0 || fuel > 0)
                         && !engineOn;
     //@ ensures engineOn;
     void start() { … }

     void accelerate() { … }

     //@ requires speed != 0;
     //@ ensures speed < \old(speed)
     //@ ensures charge > \old(charge)
     void brake() { … }
}
```

- **Subclass fulfills the same invariants (and additional ones)**
- **Overridden method start has weaker precondition**
- **Overridden method brake has stronger postcondition**