Designing for Testability

your code is awesome, but I can't test it…

What’s testable design?

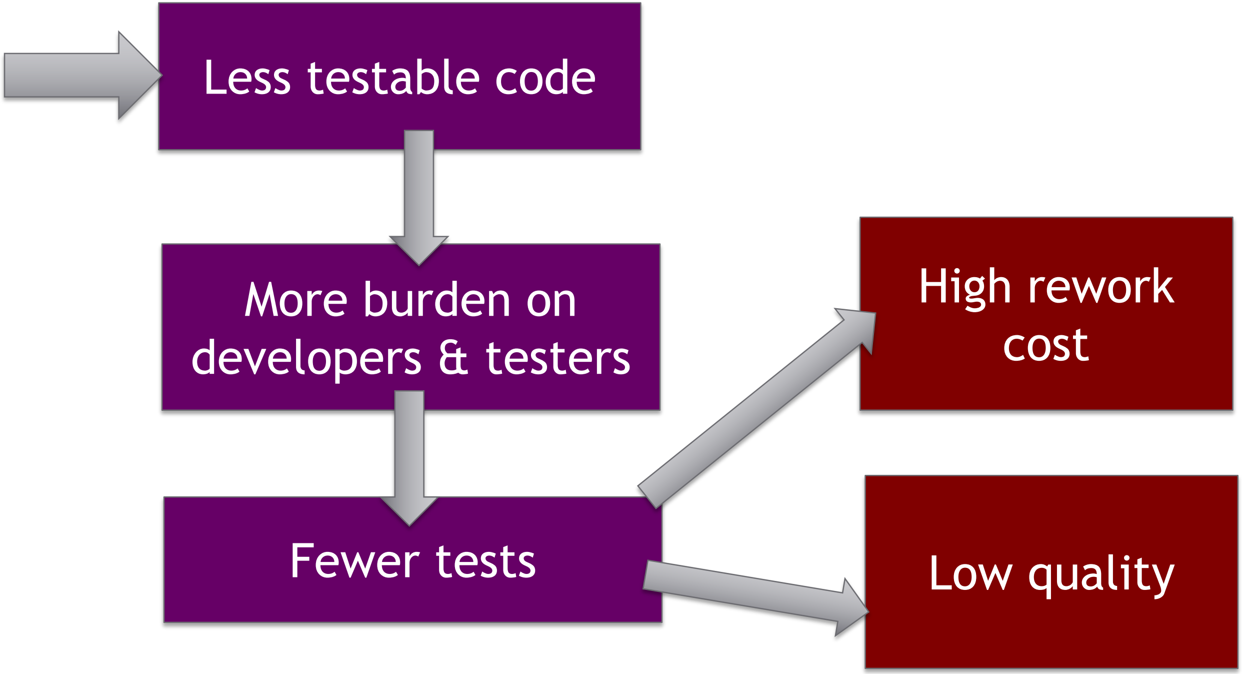
Testability: whether software can easily be tested

Testable design: design that improves/favors testability

“a given piece of code should be easy and quick to write a unit test against”

Roy Osherove, *The Art of Unit Testing with Examples in .NET*

Why testability?



Less testable code

More burden on developers & testers

Fewer tests

High rework cost

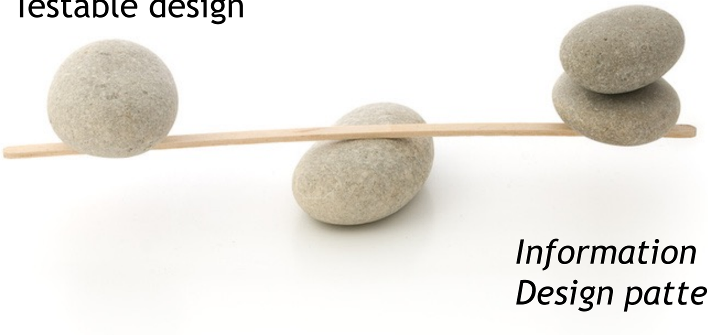
Low quality

Is testability easy and free?

It’s a matter of balance: sometimes testability clashes with design principles

Testable design

Design principles



Information hiding?
Design patterns?

“Oh no, this is a really complex private class, it doesn’t seem to be working, and I cannot test it!”

“Oh no, this class is a Singleton and it has nondeterminism, too bad because I cannot replace it with a deterministic dummy class in my tests!”

Testable Design Principles

Modularity

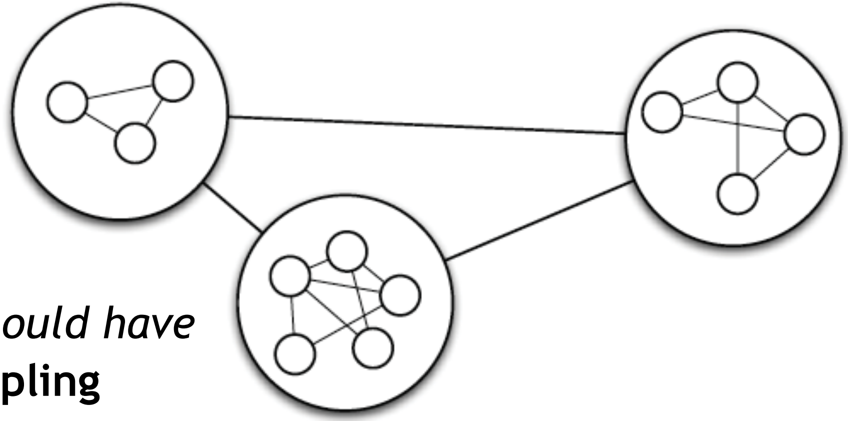
(Modular designs are easier to test)

SOLID

(SOLID designs are easier to test)

Modularity

Partitioning software into separate components in such a way that dependencies among components are minimized while relatedness of elements within components are maximized



Modules should have

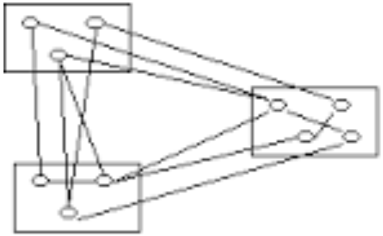
- **Low coupling**
- **High cohesion**

Koskela 2013

OO: component = class

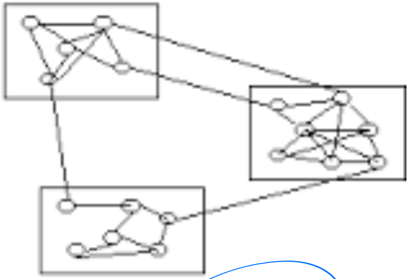
Modularity - Which system is better designed? Which is easier to test?

Low cohesion
High coupling



A

High cohesion
Low coupling



[✓ B]

OO: boxes = classes; circles = methods; lines = dependencies

Keeping designs SOLID

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Applies to OO software in particular…

but generalizable…

SOLID principles: attributed to Robert C. Martin

Single Responsibility Principle (SRP)

“There should not be more than one reason for a class to change.”

Classes should be

- *small*
- *focused*
- *cohesive*

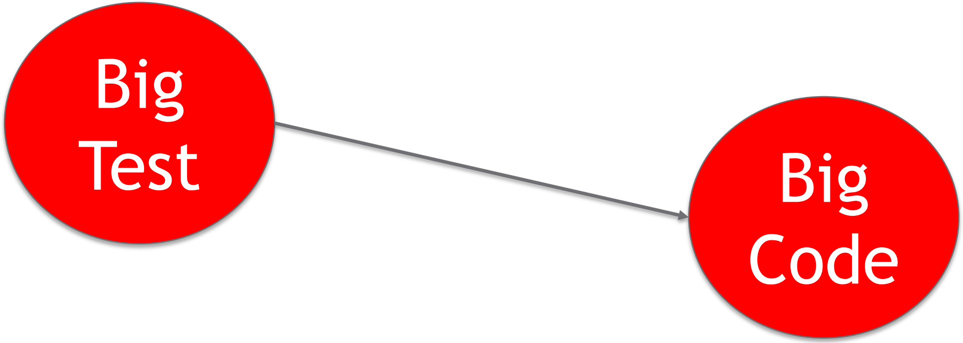
Methods should be

- *small*
- *focused*
- *simple*

How can tests profit from SRP?

"Tests should have a single reason to fail" ("Single Purpose")

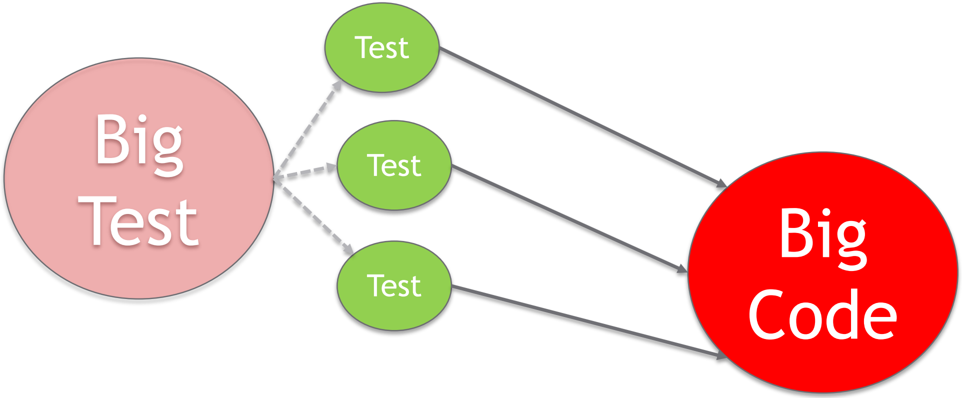
- Tests can themselves adopt SRP (already know)
- It's easier to focus tests when production code obeys SRP



How can tests profit from SRP

“Tests should have a single reason to fail”

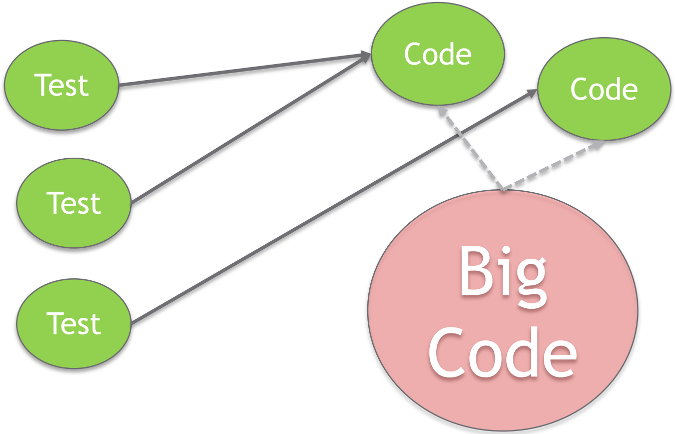
- Tests can themselves adopt SRP
- It’s easier to focus tests when production code obeys SRP



How can tests profit from SRP

"Tests should have a single reason to fail"

- Tests can themselves adopt SRP
- **It's easier to focus tests when code obeys SRP**



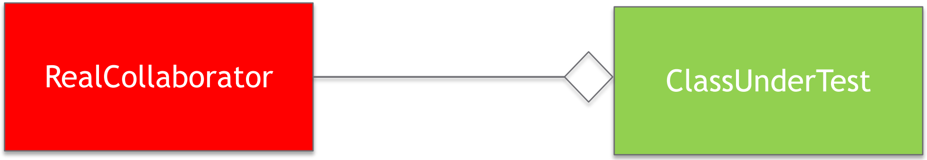
Open-Closed Principle (OCP)

“Designs should be **open** for extension, but **closed** for modification.”

- *For new functionality:* we should be able to change what a class does without changing its source code
  - E.g., rather than *copy-paste-tweak*, we reuse through
    - ***inheritance*** *by modifying inherited behavior only*
    - ***composition*** *by adding an instance of the class we want to reuse to another class*

How can tests profit from OCP

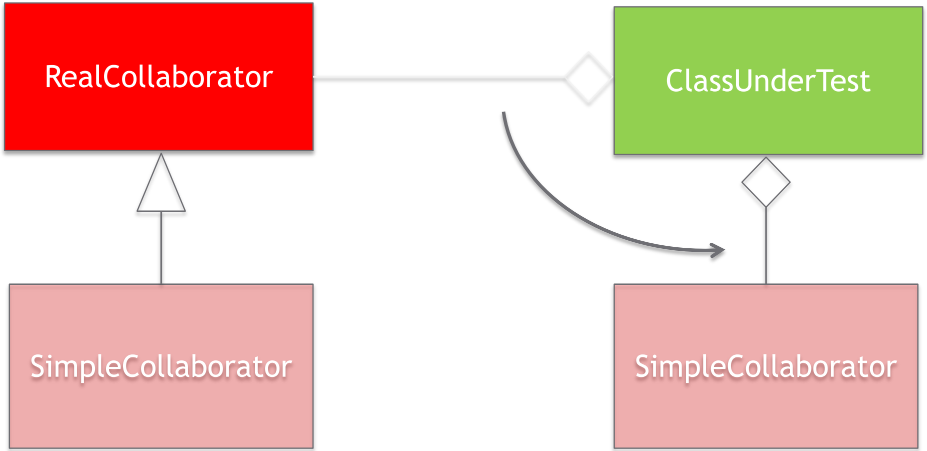
- Expensive
- Slow
- Complex
- Non-deterministic
- Available only in production environment



How can tests profit from OCP

- Expensive
- Slow
- Complex
- Non-deterministic
- Available only in production environment

- Cheap
- Fast
- Simple
- Deterministic
- Available in test environment

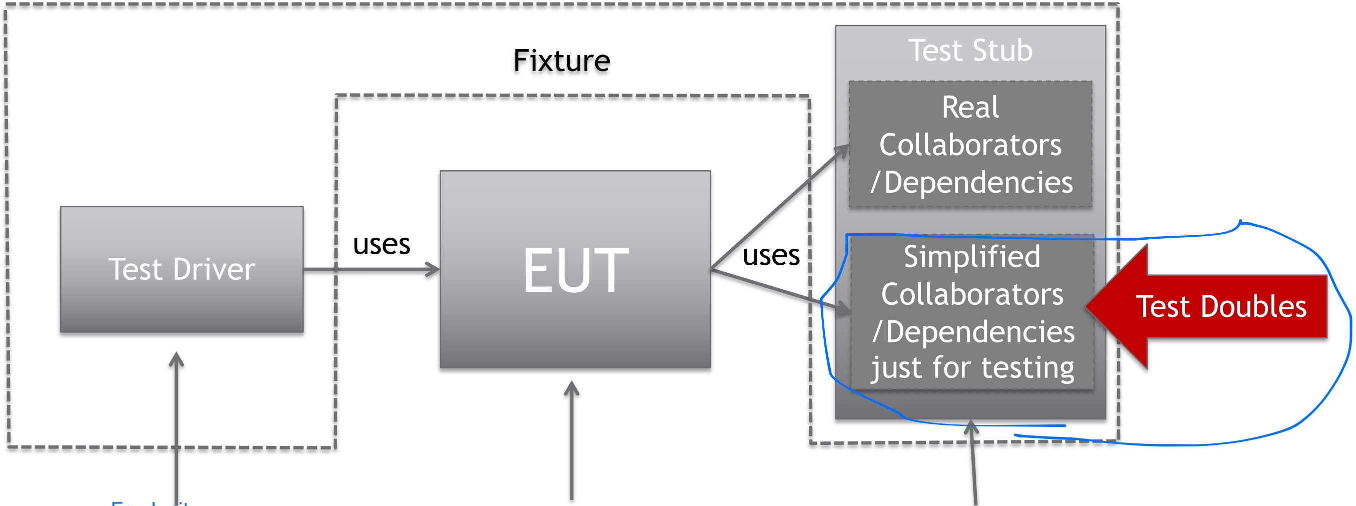


Create a simpler version of RealCollaborator by subclassing and use it in the test

How can tests profit from OCP

Test doubles!

Forward reference: Test Doubles?



Fixture
Test Stub
Real Collaborators /Dependencies
Test Driver
uses
EUT
uses
Simplified Collaborators /Dependencies just for testing
Test Doubles

- Eg: Junit

Unit testing framework + other plumbing needed to test EUT

Class/unit we're testing

Classes we're not testing alone, but we need for testing the EUT

A lot more on test doubles later!

Test Double typical example: strategy 1

A test double that emulates a persistent object…

```
public class FakeUserRepo extends UserRepo {
   // an in-memory fake user repo
   private Collection<User> users = new ArrayList<User>();
   @Override
   public void save(User user) {
      if (findById(user.getId()) == null) {
                 users.add(user);
       }
   }
   @Override
   public User findById(long id) {
      for (User user: users) {
         if (user.getId() == id) return user;
      }
      return null;
   }
}
```

EUT depends on UserRepo

- *that has real DB access*

FakeUserRepo *behaves like a real* UserRepo*, but much faster, and it doesn't touch the real DB!*

(it's not persistent, but it doesn't need to be for testing the EUT)

So the test substitutes in the fixture the UserRepo *instance with a* FakeUserRepo *instance*

Test Double typical example: strategy 2

A test double that emulates a persistent object…

```
public class FakeUserRepo implements IUserRepo {
   // an in-memory fake user repo
   private Collection<User> users = new ArrayList<User>();

   public void save(User user) {
      if (findById(user.getId()) == null) {
                 users.add(user);
       }
   }

   public User findById(long id) {
      for (User user: users) {
         if (user.getId() == id) return user;
      }
      return null;
   }
}
```

EUT depends on UserRepo, *which*

- *has real DB access*
- *implements interface* IUserRepo

FakeUserRepo *behaves like a real UserRepo, but much faster, and it doesn't touch the real DB!*

(it's not persistent, but it doesn't need to be for testing the EUT)

So the test substitutes in the fixture the UserRepo instance with a FakeUserRepo instance

How can tests profit from OCP

Test doubles!

A test can substitute a test double in the fixture without having to change the original/real collaborator/dependency

A test double can be created by (a) inheriting from the real collaborator and changing its behavior without changing the real collaborator or (b) implementing the same interface as the real collaborator

Or (c) using a mocking framework (will see this next week)

Liskov's Substitution Principle (LSP)

"Subclasses should be substitutable for their base classes."

- If B inherits from A
  - B is more specialized than A
  - Anything that is a B is also automatically an A
- then code that uses/expects an instance of class A should continue to function properly if passed an instance of class B (but not vice versa)

B is-an A

A is-not-a B

First implication of LSP for testing

Test doubles can be created using LSP

“Test double and real class extend or implement the same superclass or interface; they can be substituted for superclass in any (?) context that expects the superclass/interface.”

Koskela 2013

First implication of LSP for testing

Test doubles can be created using LSP

"Test double and real class extend or implement the same superclass or interface; they can be substituted for superclass in a test's context that expects the superclass/interface."

Koskela 2013

Second implication of LSP for testing

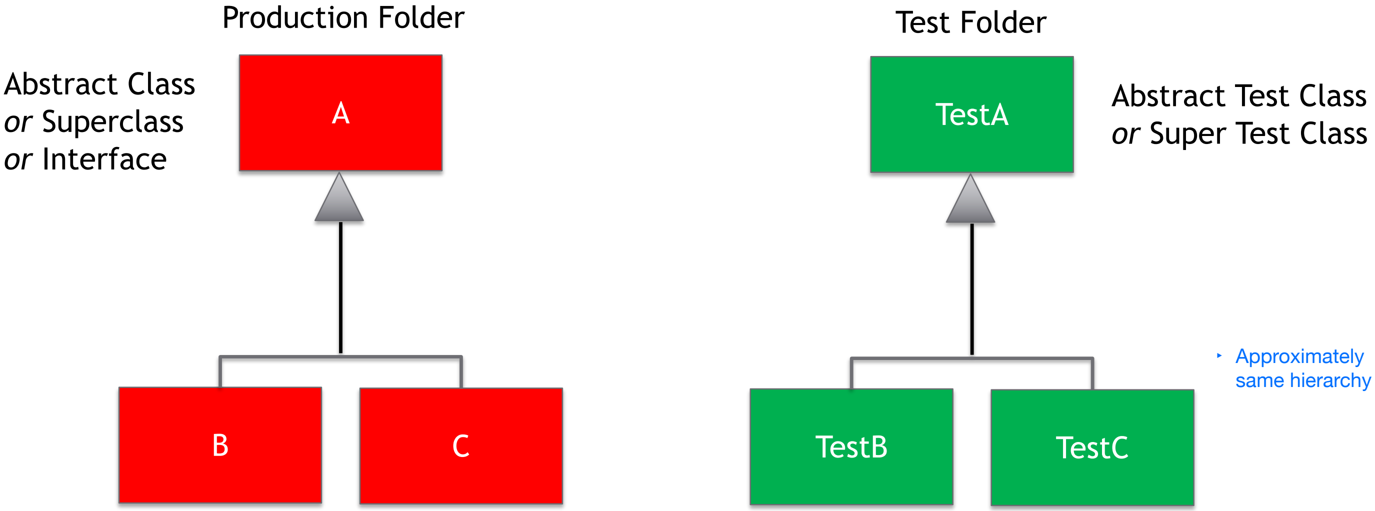
Reusable tests… *LSP supports writing tests for interfaces*

"Class hierarchies that follow LSP contribute to testability by enabling the use of *contract tests* — tests written for an interface can be executed against all implementations of that interface."

See example in A1

Koskela 2013

Reusable Tests with LSP



- Approximately same hierarchy

See example in A1

Interface Segregation Principle (ISP)

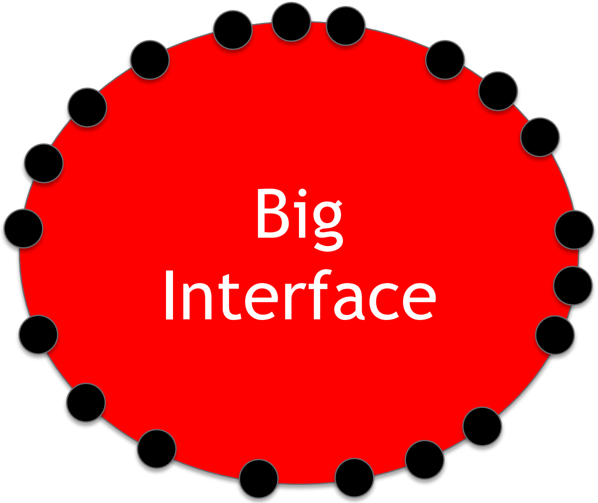
“Many client-specific interfaces are better than one general purpose interface.”

- Keep interfaces small and focused

Interface = public API of a class

What are ISP’s implications for testing?

How does a test class look like for a class with this interface?
How easy is it to create a test double this?



© 2023 Hakan Erdogmus

What are ISP’s implications for testing?

- *Small interfaces allow us to organize our tests as smaller, more focused test classes (taking us back to SRP)*
- *Small interfaces improve testability by making it easier to write and use test doubles*

“One test might want three collaborators A, B, C that need to be substituted by test doubles. With each collaborator having its own small interface, it’s straightforward to implement the test doubles.”

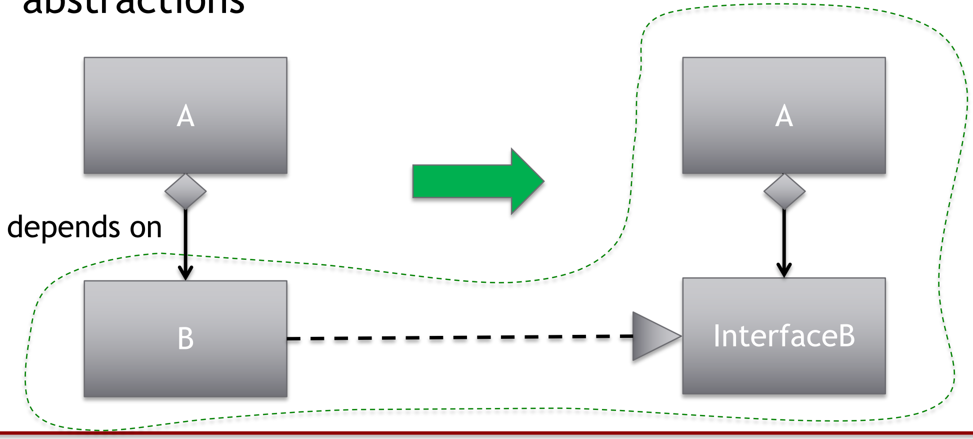
Otherwise, we may need lots of code in a single test double -- it can become a complete mess!

Koskela 2013

Dependency Inversion Principle (DIP)

"Code should depend on abstractions, not on concretions."

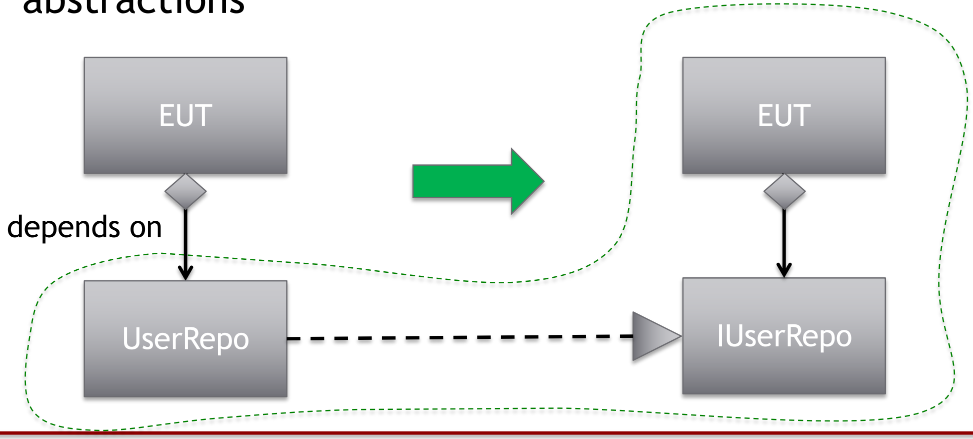
- High-level classes, *especially if they are complex*, should not directly depend on low-level classes that *are likely to change*: instead both should depend on abstractions



Dependency Inversion Principle (DIP)

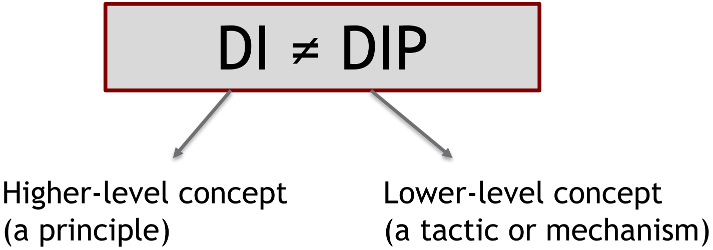
"Code should depend on abstractions, not on concretions."

– High-level classes, *especially if they are complex*, should not directly depend on low-level classes that *are likely to change*: instead both should depend on abstractions



Dependency Injection (a mechanism) *frequently used with* DIP (a principle)

“a class shouldn’t instantiate its own collaborators, but rather have their interfaces passed in”



DI ≠ DIP

Higher-level concept
(a principle)

Lower-level concept
(a tactic or mechanism)

- DI is a way of implementing DIP

Dependency injection example

```
// interface IDatabase;

public class ClassWeAreTesting {
  private IDatabase myDatabase;

  public ClassWeAreTesting() {
    myDatabase = new RealDatabase();
  }

  public ClassWeAreTesting(IDatabase useThisDatabaseInstead) {
    myDatabase = useThisDatabaseInstead;
  }

  public void doStuff() {
    ...
    myDatabase.getData();
    ...
  }
}
```

Default constructor uses a real database for production environment

Constructor injector for testing

`RealDataBase` and `DatabaseDouble` implement `IDatabase`

Now my test's fixture can *inject* an `IDatabase` ***double*** into `ClassWeAreTesting`:

```
    IDataBase dBDouble = new DatabaseDouble(); // implements IDataBase
    ClassWeAreTesting eut = new ClassWeAreTesting(dBDouble);
```

Testability Inhibitors

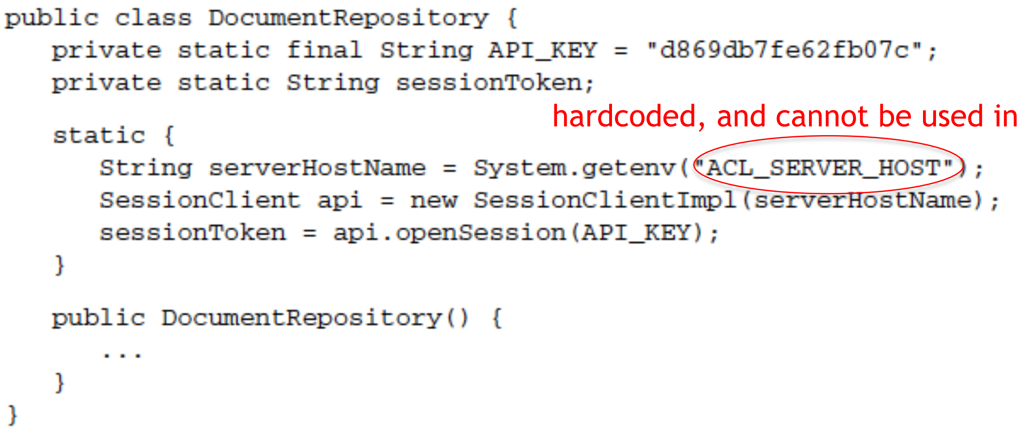
Restrictions on…

- instantiation
- invocation
- observation
- substitution
- overriding

Instantiation: Can’t instantiate a class

May happen because of production-environment dependencies…

A class constructor that relies on a production-environment variable:



```
public class DocumentRepository {
    private static final String API_KEY = "d869db7fe62fb07c";
    private static String sessionToken;

    static {
        String serverHostName = System.getenv("ACL_SERVER_HOST");
        SessionClient api = new SessionClientImpl(serverHostName);
        sessionToken = api.openSession(API_KEY);
    }

    public DocumentRepository() {
        ...
    }
}
```

hardcoded, and cannot be used in tests

Koskela 2013

Invocation: Can't invoke a method

- **Private** methods not accessible to test classes
  - *Don't test it*
  - *Change visibility (protected or package)* This is the easy way
  - *Use reflection at runtime to change visibility (undesirable because because it can make tests brittle and very complicated)*
  - *Test it indirectly (already doing...)*

- **Opaque** methods whose signatures make it hard to guess what they expect without documentation
  - Ex: computeScore(int i, Object mtd, Object aggr)

Observation: Can’t observe the outcome

- **Void** methods with no return value
- **Internal state** not visible/accessible to clients
- **Side effects** that need to be checked, but are hidden
- Object **interactions** that need to be verified, but are hidden

  (more on object interactions in Test Doubles)

Substitution: Can’t substitute a collaborator

- Hard-coded collaborators

```
Collaborator slowCollab = new Collaborator()
slowCollab.doStuff();
```

Hardcoded collaborator: can’t substitute a double

- Solution: dependency injection

Substitution: Can’t override method

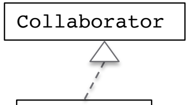
Sometimes rather than substituting a *whole* double for a collaborator, we may simply want to use a ***partial*** double

- *Subclass and override the behavior that the test needs to be changed*

Production code

```
// class ClassUnderTest
…
private static final
     void Collaborator
          getCollaborator() {
     ...
}
```

Test code



Collaborator

```
@TEST
public void test() {
     final Collaborator collab = new TestDouble();
     ClassUnderTest o = new ClassUnderTest() {
          @Override
          private static final void getCollaborator() {
               return collab;
          }
     };
     …
```

Is this possible?

Substitution: Can't override method

`private, static, final` *modifiers prevent overriding*

Production code

```
// class ClassUnderTest
…
private static final
     Collaborator
          getCollaborator() {
     ...
}
```

Test code

```
@TEST
public void test() {
     final Collaborator collab = new TestDouble();
     ClassUnderTest o = new ClassUnderTest() {
          @Override
          private static final Collaborator getCollaborator() {
               return collab;
          }
     };
     …
```

Testability Guidelines

… *concerning the use of* …

- private methods
- final methods
- static methods
- instantiation
- logic in constructors
- singletons
- composition vs. inheritance
- wrapping

Avoid complex private methods

Private methods may be warranted for encapsulation to promote information hiding and prevent unwanted manipulation of an object's state

Testability compromise: decompose private methods with complex logic to **green** and **red** methods

- **green: simple private methods** (no need to test directly), and
- **red: complex public methods** that isolate the complexity (need to test)

Avoid final methods

Final methods provide immutability and protection from unwanted overriding by a subclass

- *some design patterns (e.g., Template Method) advocate the use of final methods*

No easy answer

- Weigh cost of wrong usage against benefits of testability
- *Remember: testing may reveal wrong usage*

In this course: **testability > wrong usage**

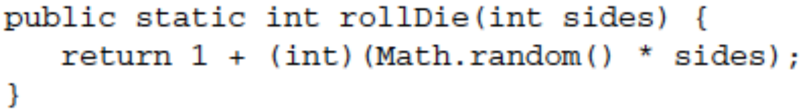
Avoid static methods

Static methods are generally reserved for

- orphaned methods (don’t relate to a specific instance)
- utility methods
- ~~global access~~

*Cannot be overridden**
(although can be hidden/masked if subclasses redefine them)

Example:



```
public static int rollDie(int sides) {
    return 1 + (int)(Math.random() * sides);
}
```

Nondeterministic: will need to be substituted in tests
→ *forget about “static” and make it an instance method*

Remember: polymorphism is about instances, not static class properties

What about reflection?

Java Reflection API can remove **private** and **final** modifiers…

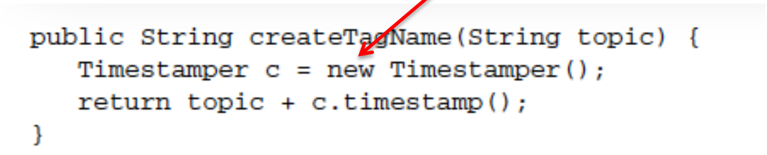
Should you use it?

- No. We don't want the test to be more complex than the production code

Use “new” with care

- Only instantiate objects you won’t want to substitute with doubles
- Otherwise treat them as dependencies
  - ??
  - ??

hardcoded, and *non-deterministic*



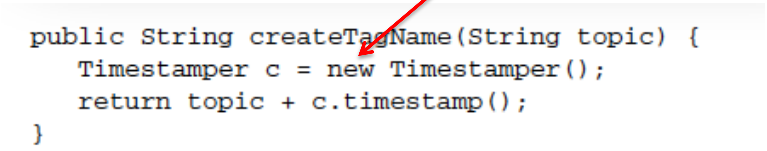
```
public String createTagName(String topic) {
   Timestamper c = new Timestamper();
   return topic + c.timestamp();
}
```

How can you fix this code to be able to test `createTagName`*?*

Use "`new`" with care

- Only instantiate objects you won't want to substitute with doubles
- Otherwise treat them as a dependencies
  - pass as a parameter
  - inject dependency

hardcoded, and *non-deterministic*



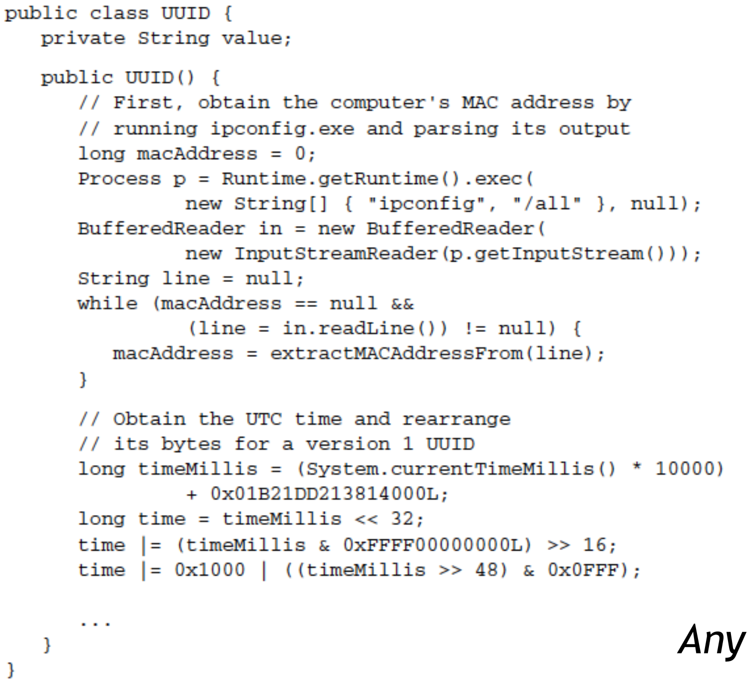
```
public String createTagName(String topic) {
   Timestamper c = new Timestamper();
   return topic + c.timestamp();
}
```

You could make the `Timestamper` object a parameter of this method or an injectable instance variable of the method's class so you can create fake timestamps that don't rely on the actual date and time of day

Avoid logic in constructors

- Constructors are hard to bypass in test code
- At least one superclass constructor will be triggered
- Move test-preventing code out of the constructor

Non-bypassable logic in constructors

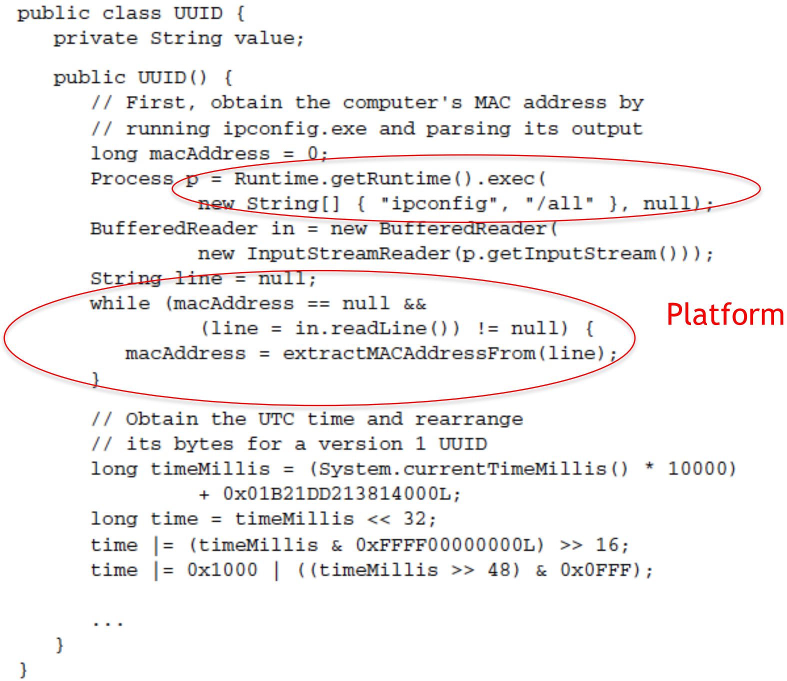


```
public class UUID {
   private String value;

   public UUID() {
      // First, obtain the computer's MAC address by
      // running ipconfig.exe and parsing its output
      long macAddress = 0;
      Process p = Runtime.getRuntime().exec(
               new String[] { "ipconfig", "/all" }, null);
      BufferedReader in = new BufferedReader(
               new InputStreamReader(p.getInputStream()));
      String line = null;
      while (macAddress == null &&
               (line = in.readLine()) != null) {
         macAddress = extractMACAddressFrom(line);
      }

      // Obtain the UTC time and rearrange
      // its bytes for a version 1 UUID
      long timeMillis = (System.currentTimeMillis() * 10000)
               + 0x01B21DD213814000L;
      long time = timeMillis << 32;
      time |= (timeMillis & 0xFFFF00000000L) >> 16;
      time |= 0x1000 | ((timeMillis >> 48) & 0x0FFF);

      ...
   }
}
```

Any problems?

Koskela 2013

Non-bypassable logic in constructors

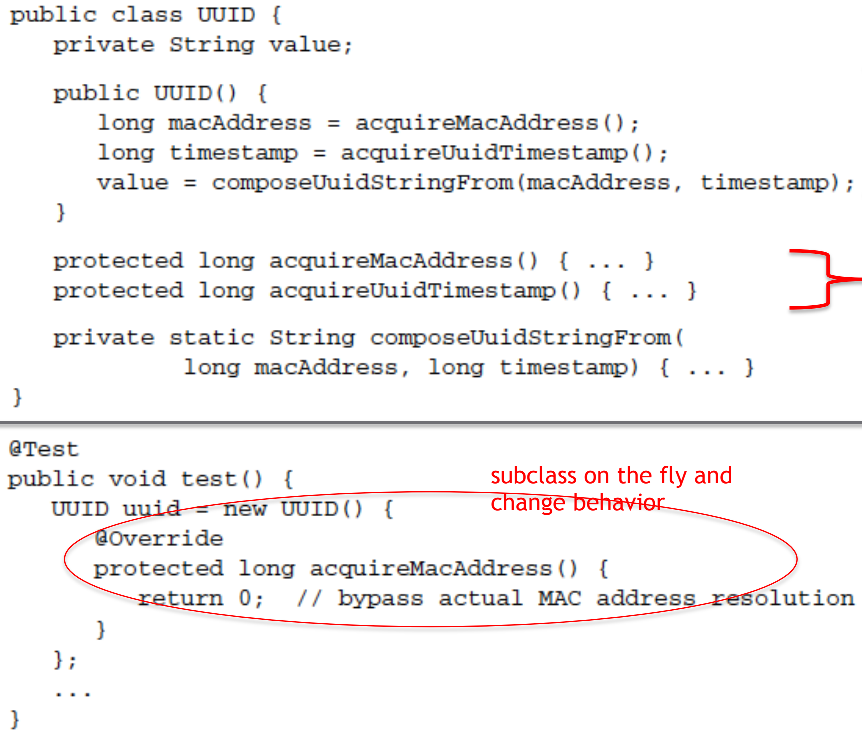


```
public class UUID {
   private String value;

   public UUID() {
      // First, obtain the computer's MAC address by
      // running ipconfig.exe and parsing its output
      long macAddress = 0;
      Process p = Runtime.getRuntime().exec(
               new String[] { "ipconfig", "/all" }, null);
      BufferedReader in = new BufferedReader(
               new InputStreamReader(p.getInputStream()));
      String line = null;
      while (macAddress == null &&
               (line = in.readLine()) != null) {
         macAddress = extractMACAddressFrom(line);
      }

      // Obtain the UTC time and rearrange
      // its bytes for a version 1 UUID
      long timeMillis = (System.currentTimeMillis() * 10000)
               + 0x01B21DD213814000L;
      long time = timeMillis << 32;
      time |= (timeMillis & 0xFFFF00000000L) >> 16;
      time |= 0x1000 | ((timeMillis >> 48) & 0x0FFF);

      ...
   }
}
```

Platform dependence!

Koskela 2013

Avoiding non-bypassable logic in constructors



```
public class UUID {
   private String value;

   public UUID() {
      long macAddress = acquireMacAddress();
      long timestamp = acquireUuidTimestamp();
      value = composeUuidStringFrom(macAddress, timestamp);
   }

   protected long acquireMacAddress() { ... }
   protected long acquireUuidTimestamp() { ... }

   private static String composeUuidStringFrom(
            long macAddress, long timestamp) { ... }
}
```

```
@Test
public void test() {
   UUID uuid = new UUID() {
      @Override
      protected long acquireMacAddress() {
         return 0;  // bypass actual MAC address resolution
      }
   };
   ...
}
```

much simpler

platform-dependent
complexity moved to
overridable methods

subclass on the fly and
change behavior

Avoid the Singleton pattern

Singleton pattern ensures a class has *one and only one* instance

but it uses private static constructs, which prevent tests from creating doubles

Conundrum:

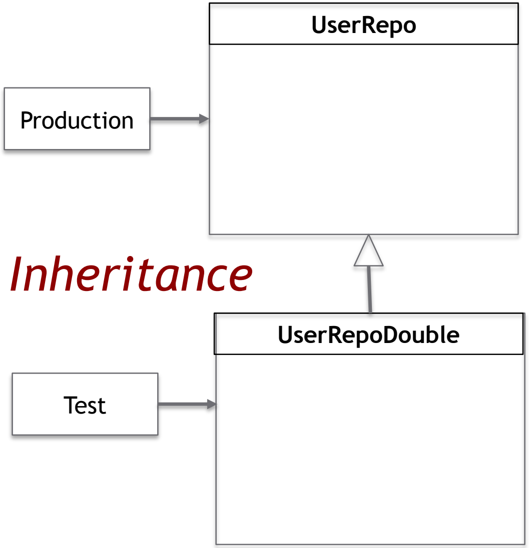
Pervasive, centralized, shared runtime resources should conceptually be singletons…

Unfortunately, these are precisely the objects for which we want to create test doubles

See workaround in Appendix (weakens the pattern for testing)

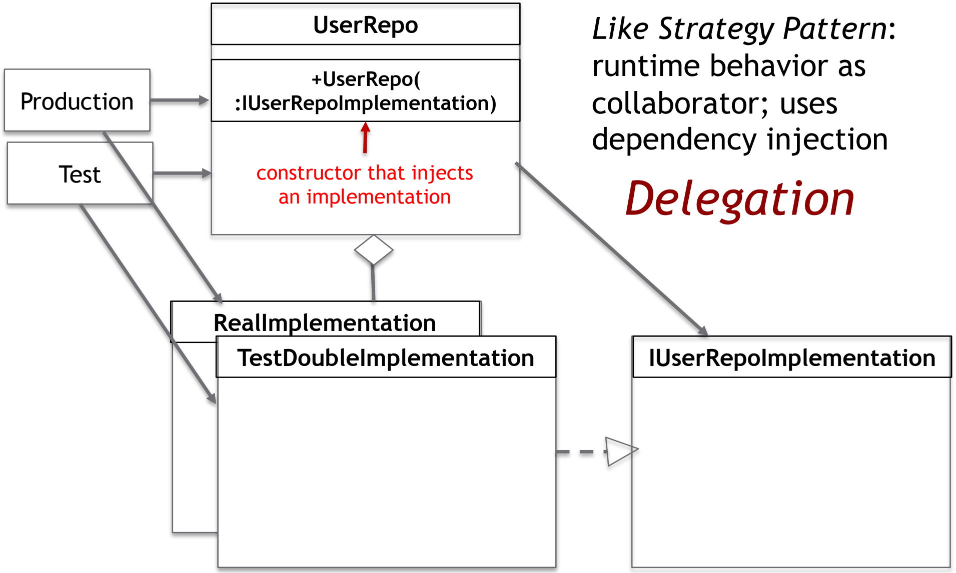
Favor Composition over Inheritance for better testability

Strategy 1: Less work, less flexible



Inheritance

Strategy 3: More work, more flexible, more obvious for testing (similar to Strategy 2, but uses composition)



Like Strategy Pattern: runtime behavior as collaborator; uses dependency injection

Delegation

See Appendix on how inheritance can lead to LSP violations!

Wrap external libraries (*aka*, introduce a middle man)

Sometimes we can't change the design…

- Avoid inheriting from external untestable code
- Isolate those pieces you can't change or test, such as…
  - external libraries
  - legacy components

  within *thin wrappers** that you can control…
- Redirect calls to those pieces to the new wrappers
- You can create doubles for the wrappers (you control them)

The **Adapter** design pattern allows you to do this…

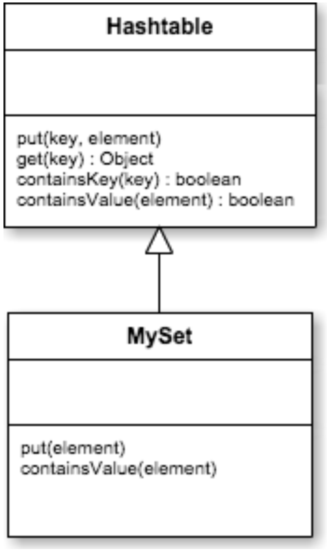
**translates from one API to another using delegation*

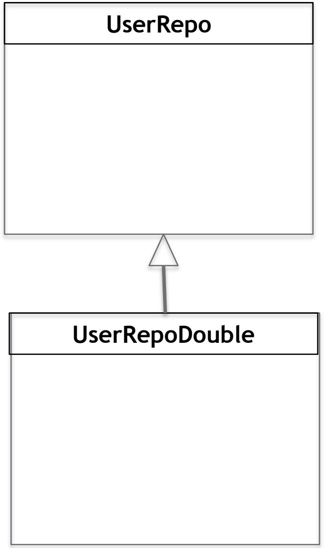
Testability: key messages

- *Main obstacles for testable code are:*
  - – Inability to instantiate a class
  - – Inability to invoke a method
  - – Inability to observe a method’s outcome or side effects
  - – Inability to substitute a test double
  - – Inability to override a method
- *Tradeoffs between testability and other design principles can be settled with some compromises*
- *Dependency injection is a friend of testability*

Appendix

Delegation/composition prevents implementation inheritance, an LSP violation!

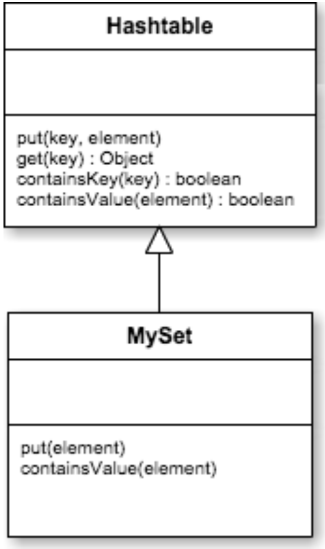


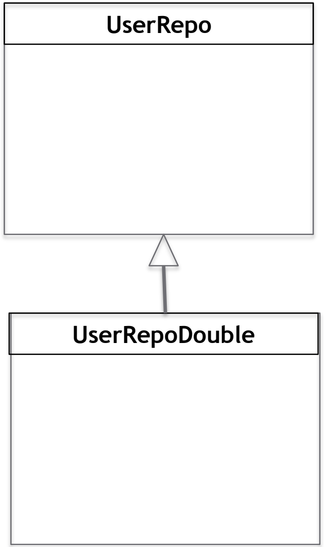


Implementation inheritance

A misuse of inheritance

Violates LSP!





Better!

In test’s context, ok, but not in general!

Singleton example

Singleton

```
public class Clock  {
   private static final Clock singletonInstance = new Clock();

   // private constructor prevents instantiation from other classes
   private Clock() { }

   public static Clock getInstance() {
      return singletonInstance;
   }
   …
}
```

If you want to use a Clock double in testing the Log class, you can't!

Client of Singleton

```
public class Log {
   public void log(String message) {
      String prefix = "[" + Clock.getInstance().timestamp() + "] ";
      logFile.write(prefix + message);
   }
}
```

Dealing with singletons

- Avoid singletons altogether and rely on *convention*

or

- Use ***weak singletons*** and rely on *less convention*

Interface that declares timestamp() for DI

not final

```
public class Clock implements IClock {
     private static IClock singletonInstance = new Clock();

     static installClockDoubleForTesting(IClock clockDouble) {
              singletonInstance = clockDouble;
     }

     private Clock() { }

     public static IClock getInstance() {
          return singletonInstance;
     }

     public LocalTime timestamp() {
          LocalTime.now();
     }

}
```

- *convention: setter injector,* only test classes in same package should call this method
- clockDouble defines timestamp to return a constant LocalTime rather than the actual local timestamp

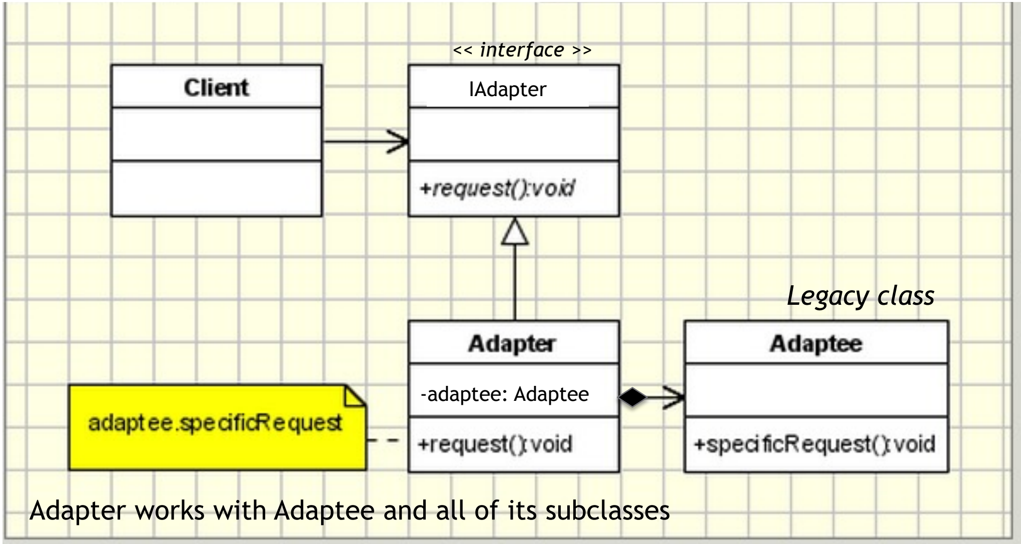
convention: I mean "*unenforceable but clear usage rules that all devs and testers should obey*"

Usage of weak singleton

```
@Test
public void usingClockDoubleInThisTest() {
  IClock clockDouble = new IClock{) {
    public LocalTime timestamp() {
        return LocalTime.NOON;
    }
  }
  Clock.installClockDoubleforTest(clockDouble);
  … // invoke the tested behavior here
}
```

Sidetrack: Adapter design pattern

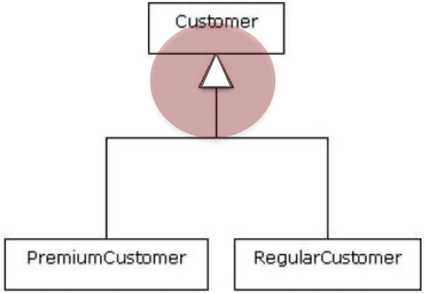
Convert the interface of a legacy class into a different interface expected by the client so that the legacy class can be replaced easily in the future

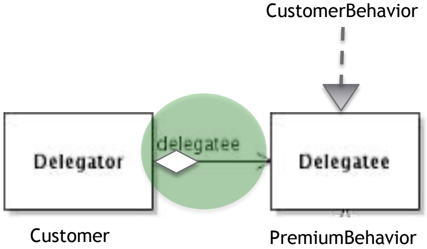


Adapter works with Adaptee and all of its subclasses

http://www.oodesign.com/adapter-pattern.html

Sidetrack: inheritance vs. delegation/composition





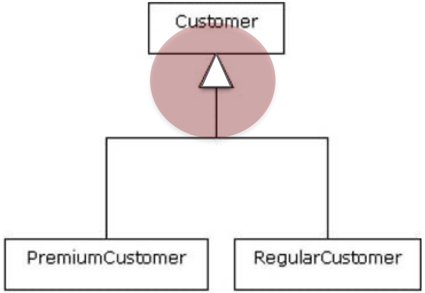
Strategy Pattern

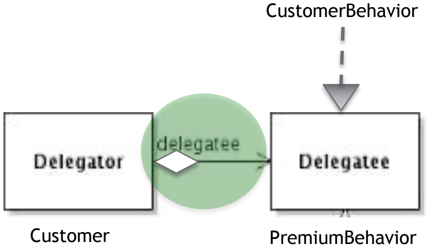
Inheritance

Delegation/Composition

Hierarchy

Sidetrack: inheritance vs. delegation/composition





Inheritance

Easy reuse with subtyping

Does not need extra structure - succinct

Introduces strong dependency

Behavior fixed at runtime

Sometimes too powerful

Delegation/Composition

Reuse without subtyping

Needs extra structure – a bit of overhead

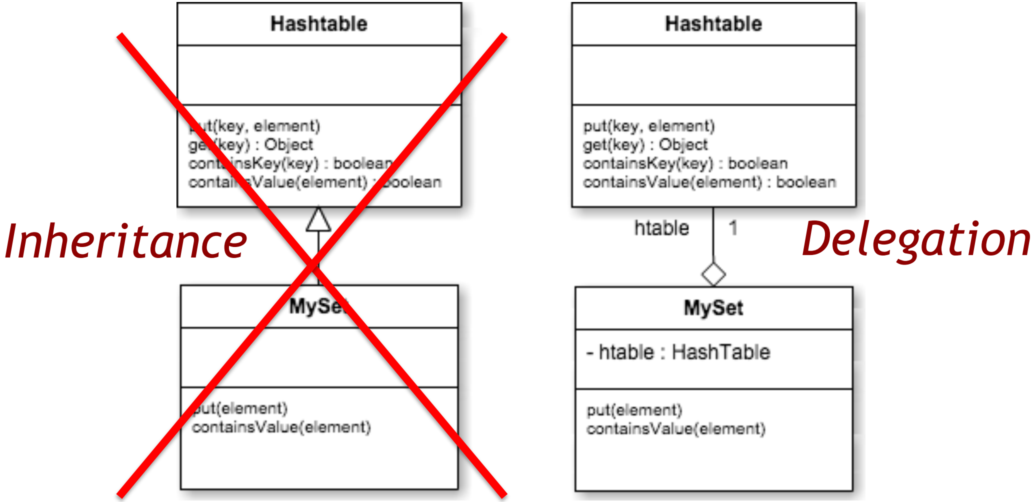
Introduces weak dependency

Can change behavior at runtime

Often what we need

Delegation/Composition in General

Adopted when reuse through inheritance is not the best option



Does not violate LSP!

Favor composition/delegation over direct inheritance

- Inheritance is sometimes too powerful or restricting
  - may straight-jacket you: *forever bound to superclass* (*pray that its API doesn't change*) – *changes in superclass trigger test refactoring*
  - use only to harness power of polymorphism (always *remember LSP* not to abuse inheritance)

For ***reuse***, prefer composition/delegation

- Can reuse various implementations vs. just super-methods
- Can change implementations at runtime
- Enables dependency injection
- Makes your code more testable – just inject a test implementation!

Substitution: Can’t substitute a collaborator

- Chain method calls (with deep-level access)

```
getCollaborator().getComplexData().findComplexObject().askForComplexStuff()
```

Homework: Think about why this isn’t good for testing?
(reason on next slide)

Substitution: Can’t substitute a collaborator

- Hard-coded collaborators

```
Collaborator collab = new Collaborator()
collab.doStuff();
```

can’t substitute a double

- Chain method calls (with deep-level access)

```
getCollaborator().getData().findObject().askForStuff()
```

This object needs a double

ok

may need a double

deep

may need a double

deeper

Demeter’s Law: an object should access only to itself, parameters, objects instantiated by it, immediate collaborators, instance variables, globals

Deep access is an anti-pattern, a “code smell”: violates Demeter’s **Law** (exception: fluid interface)