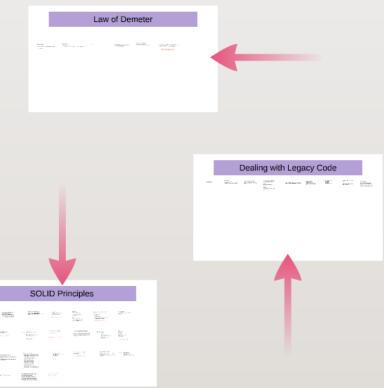
CS1699: Lecture 22 - Writing Testable Code

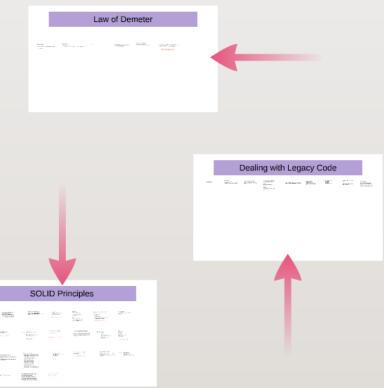






CS1699: Lecture 22 - Writing Testable Code







Defining Testable Code

In one sense, all code is testable, since we can provide input and observe output.



Testable code: Code for which it is easy to perform automated tests at various levels of abstraction, and track down errors when tests fail.



for. Throught, writing poing to salik above good code... and by disking act, we'll assess anothey governable code!

TWO FOR THE PRICE OF ONE!

In this lecture, we'll cover-1. Basic Strategy for Testability

The DRY Concept
 SOUD Principles
 Law of Demeter
 Minimizing Mutable Gi

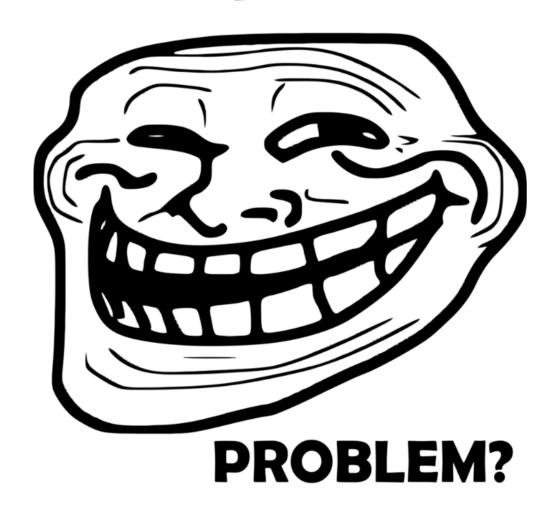
You only mean to make it testable, but it turns out to also be maintainable and VERY easy to integrate with."

Chris Umbel. Software Engine

"Testable code is one of those funny things.



In one sense, all code is testable, since we can provide input and observe output.



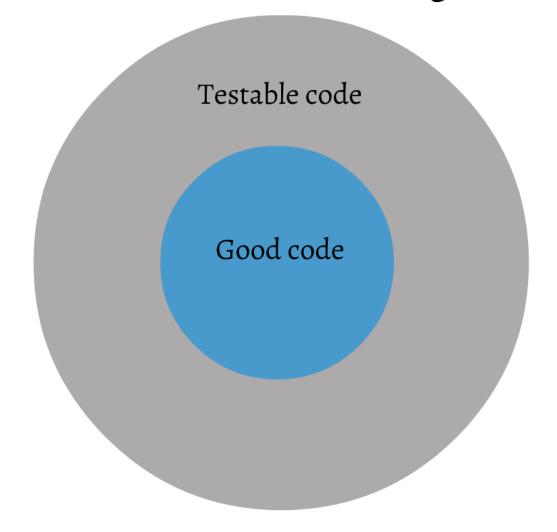


Testable code: Code for which it is easy to perform automated tests at various levels of abstraction, and track down errors when tests fail.



Good code is testable code.

Not all testable code is good code.





Tonight, we're going to talk about good code... and by doing so, we'll automatically get testable code!

TWO FOR THE PRICE OF ONE!



In this lecture, we'll cover -

- 1. Basic Strategy for Testability
- 2. The DRY Concept
- 3. SOLID Principles
- 4. Law of Demeter
- 5. Minimizing Mutable Global State
- 6. Dealing with Legacy Code



"Testable code is one of those funny thir You only mean to make it testable, but it turns out to also be maintainable and Vi easy to integrate with."

-Chris Umbel, Software Engineer



The Basic Strategy for Testability

Key concept:

Segment code, make things repeatable.

The more parts of the system your code relies upon to execute properly, the more difficult it is to test.

```
public in getNumllooksfort userNum) [
String dh - DatabaseTastury getDbi) namekoString));
DatabaseConnector dhe - new DraibaseConnector(dbi);
Schema schema: SchemaSingleton getSchema();
Über user;
try [
user - UserLookup getUser(userNum)(o); toUser();
[atah [Exception 0] [user = milh.];
dbc useSchema(schema);
return dbc.get("BoolsCus");where("User = " + user.toString()).
```

Think about everything that depends on to execute properly.

Try to minimize these external dependencies. How could we do this?

Good testing and good code involves keeping concerns separate, as much as possible.

This will not only make testing easier, but code comprehension easier!

Pure vs Impure functions

MINIMIZE MUTABLE STATE!

Functional Programming, especially in a language like Haskell, does much of what we're talking about it automatically as part of the language.



Key concept:

Segment code, make things repeatable.



The more parts of the system your code relies upon to execute properly, the more difficult it is to test.



```
public int getNumBooks(int userNum) {
   String db = DatabaseFactory.getDb().name.toString();
   DatabaseConnector dbc = new DatabaseConnector(db);
   Schema schema = SchemaSingleton.getSchema();
   User user;
   try {
    user = UserLookup.getUser(userNum)[0].toUser();
   } catch (Exception e) { user = null; }
   dbc.useSchema(schema);
   return dbc.get("BooksOut").where("User = " +
user.toString());
```



Think about everything that depends on to execute properly.

Try to minimize these external dependencies. How could we do this?



```
public int getNumBooks(String userName,
  DatabaseConnector dbc) {
  return dbc.get("BooksOut").where("User = " +
  userName);
}
```



Good testing and good code involves keeping concerns separate, as much as possible.

This will not only make testing easier, but code comprehension easier!



Pure vs Impure functions

MINIMIZE MUTABLE STATE!



Functional Programming, especially in a language like Haskell, does much of what we're talking about it automatically as part of the language.



DRYing up Code

DRY = Don't Repeat Yourself

```
public inti] addarrays[mt] the, let] thei {
    inti] issReams = even intilla-integrit;
    for (or i ot.) i = that length; in-i} {
        lothering | in-intill + rint[];
    }
    integrit | in-intill + rint[];
    import | j.*;
    import | j.*;
    public intill | publithAddition() intill | line, intill | rinternal motion intill | publithAddition() intill | line, intill | rinternal | public | line, intill | rinternal | line, intill | line, intil
```

```
Why is this had?

1. Thrice as many hosts

2. Thrice as many places to make errors

3. Which is the correct one to one?

4. Should codebase
```

A more insidious example...

name = db.where("user_id = " + id_num).get_names(0)
...later ...
name = db.find(id).get_names.first

Why not make a getName(id) method?

If multiple pieces of code are doing the same thing, consider creating a method/function for it.



DRY = Don't Repeat Yourself



Simple Example

```
public int[] addArrays(int[] lhs, int[] rhs) {
 int[] toReturn = new int[lhs.length];
 for (int j=0; j < lhs.length; j++) {
   toReturn[i] = Ihs[i] + rhs[i];
 return toReturn;
import fj.*;
public int[] zipWithAddition(int[] lhs, int[] rhs) {
 return zipWith(lhs, rhs, add);
```



Why is this bad?

- 1. Twice as many tests
- 2. Twice as many places to make errors
- 3. Which is the correct one to use?
- 4. Bloated codebase



A more insidious example...

name = db.where("user_id = " + id_num).get_names[0]

... later ...

name = db.find(id).get_names.first



Why not make a getName(id) method?



If multiple pieces of code are doing the same thing, consider creating a method/function for it.



SOLID Principles

A mnemonic for the "five key principles" of object-oriented Single Responsibility Principle S Single Responsibility Principle O Open/Glosed Principle L Liskov Substitution Principle Clesses should be open for extraction, but clessed to modification. Other code smells: 1. Many methods 2. Many attributes I Interface Segregation Principle
D Dependency Inversion Principle Difficult to comprehend what class does
 Methods don't seem related Why does this make testing easier? Liskov Substitution Principle Once complete, code modification in a given module ('slear') should not more energitic fix defects. A class B which is a subclass of class A, should implement any method in A while meeting all Why does this make our code easier to test? public interface BankInterface [public void transferMoneyIntraBank(); public void transferMoneyInterBank(); How does this help for testing? public void allocateMortgage(); public void transferMortgage(); In practice, this means lots of small interlaces, not one big one. public void setupHeloc(); public void withdrawGash(); public void depositCheck(); public void depositCash(); public Bank[] getBankBranches(); public Employee() getBankEmployees(); S Single Responsibility Principle O Open/Closed Principle L Liskov Substitution Principle

I Interface Segregation Principle D Dependency Inversion Principle



A mnemonic for the "five key principles" of object-oriented design.



- S Single Responsibility Principle
- O Open/Closed Principle
- L Liskov Substitution Principle
- I Interface Segregation Principle
- **D** Dependency Inversion Principle



Single Responsibility Principle

A class should have a single responsibility.

That responsibility should be entirely encapsulated by the class.



```
public class Stuff {
  public void printMemo() { ... }
  public int numCats(String breed) { ... }
  public String getName() { ... }
  public void haltSystem(int exitCode) { ... }
}
```

What is Stuff's single responsibility?



```
public class Cat {
 public String getName() { ... }
 public String getBreed() { ... }
 public Currency getRentalCost() {...}
 public int rent() { ... }
public class RentACatSystem {
 public void startSystem() { ... }
 public void haltSystem(int exitCode) { ... }
 public void forceShutdown() { ... }
```



Describe the class. If you can't do it without using "and", you are probably violating the Single Responsibility principle.

Other code smells:

- 1. Many methods
- 2. Many attributes
- 3. Difficult to comprehend what class does
- 4. Methods don't seem related

Why does this make testing easier?



Open / Closed Principle

Classes should be open for extension, but closed to modification.



Add features by subclassing, not adding code.

Once complete, code modification in a given module ("class") should not occur except to fix defects.



```
public class Printer {
  private void formatDocument() { ... }
  public void printDocument() { ... }
  public void printToPDF() { ... }
}
```

Violation of Open/Closed Principle!



```
abstract class Printer {
                   private void formatDocument() { ... }
Better way:
                 public class PhysicalPrinter extends Printer {
                   private void printDocument() { ... }
                 public class PdfPrinter extends Printer {
                   public void printPdf() { ... }
```



If your classes keep getting bigger with each commit, you may be violating the Open/Closed Principle.

This is a really good reason for using abstract classes and interfaces.

Why does this make our code easier to test?



Liskov Substitution Principle

A class B which is a subclass of class A, should implement any method in A while meeting all invariants.



Example:

```
public class Shape {
  Location loc;
  Color color;
public class Rectangle extends Shape {
  public double length; public double height;
public class Square extends Shape {
  public double size;
```



```
public class Square {
  public Location loc;
  public Color color;
  public double size;
}
public class Rectangle extends Square {
  public double length;
  public double height;
}
```

What's wrong with this?



Liskov Substitution means that you can mock without fear of causing issues.



Interface Segregation Principle

Clients should depend on methods that they do not use.

In practice, this means lots of small interfaces, not one big one.



```
public interface BankInterface {
  public void transferMoneyIntraBank();
  public void transferMoneyInterBank();
  public void allocateMortgage();
  public void transferMortgage();
  public void setupHeloc();
  public void withdrawCash();
  public void depositCheck();
  public void depositCash();
  public void authenticate();
  public Bank[] getBankBranches();
  public Employee[] getBankEmployees();
```

```
public interface AtmInterface {
  public void withdrawCash();
  public void depositCash();
  public void depositCheck();
  public void authenticate();
}
Better
```



If you find yourself not using all of the methods of an interface from another class, consider splitting up the interfaces for different roles.

How does this help for testing?



Dependency Inversion Principle

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.



```
Example public class Aviary {
    public void buyCockatiel();
    public void buyGreyParrot();
    public void buyYellowBelliedSapSucker();
}
```



```
public class Aviary {
  public void buyBird(Bird b);
}
```

Note that abstractions/interfaces are not enough! For example, DataRecord.setTransactionRollbackTimeout();

Leaky abstractions are bad.



Allows for dependency injection!



- S Single Responsibility Principle
- O Open/Closed Principle
- L Liskov Substitution Principle
- I Interface Segregation Principle
- **D** Dependency Inversion Principle



Remember - these are principles, NOT laws.

Use common sense when applying.



Law of Demeter

The Law of Demotor

Never call a resthod on an object you get from another rall.

" Not an actual law.

Example: pig_latin_name = Db.petTable("Users").lookup(id).translate("Pig.Latin")

See complay with yourself.
You can play with your sen mayor discrease controls them up flow any play with any ot charmon given anyou.
And you can play with any you to made you self.

Sear that the descriptions (FFHE deficit) biff with a beside that some chars: Sear - MANT between charge above highly replanded; it

If you have a long line of dot-whatevers, you may be violating the Law of Demeter.

How does this help us test?



The Law of Demeter*

Never call a method on an object you got from another call.

* Not an actual law.



Example:

```
pig_latin_name =
```

Db.getTable("Users").lookup(id).translate("Pig Latin")



Better:



You can play with yourself.

You can play with your own toys (but you can't take them apart),

You can play with toys that were given to you.

And you can play with toys you've made yourself.



Note that this doesn't count IF THE OBJECTS RETURNED ARE ALL THE SAME CLASS!

foo = "BLAH ".toLowerCase().substring(2).replace('a', 'b').trim



If you have a long line of dot-whatevers, you may be violating the Law of Demeter.

How does this help us test?



Dealing with Legacy Code

It's difficult.

May pieces of advice: 1. Write tests as you go along 2. Legit for enoing su can start TDBring from a given point. on't let the manaxs of already-existing code reliese you up and mana you not add your even Seams are places in the code where you can after behavior without code modification. Example:

If SEAM public void printbac@rintor p. args) (p.printlaggic }

| NO MAM public valid printfine20 (Printer p = new Printer(DEFASET ARG p.printf); } No TUFs inside TUCs

entitationally Proteste roding the statistate as to the Statistate the Commission across the retinor's which is decided by a GAR applicates) TUC II Yest-Unified by Constitute Princip methods Final wathods Final stance. Constitutions / Desiruations "Working with Legacy Code" by Michael Feathers

Mischeneck jectments combes across articles/festable laws off Even't be discoveraged.

The hard gast of software engineering is gotting large systems to work correctly, busing with legacy code and adding testilities on of the control of the control

s was easy, everybeely would be doing it



It's difficult.



Key pieces of advice:

- 1. Write tests as you go along
- 2. Look for seams
- 3. Move/Create TUFs so that they're not inside TUCs



You can start TDDing from a given point.

Don't let the morass of already-existing code swallow you up and make you not add your own tests.



Seams are places in the code where you can alter behavior without code modification.

Example:

```
// SEAM
public void printDoc(Printer p, args) {
    p.print(args);
}

// NO SEAM
public void printDoc2() {
    Printer p = new Printer(DEFAULT_ARGS);
    p.print();
}
```



No TUFs inside TUCs



TUF = Test-Unfriendly Feature

Accessing the database Writing to the filesystem Communicating across the network Side effect-ful code (e.g. GUI updates) etc.



TUC = Test-Unfriendly Construct

Private methods
Final methods
Final classes
Constructors / Destructors



"Working with Legacy Code" by Michael Feathers

http://www.objectmentor.com/resources/articles/TestableJava.pdf



Don't be discouraged.

The hard part of software engineering is getting large systems to work correctly. Dealing with legacy code and adding testing is part of this.

If this was easy, everybody would be doing it!



CS1699: Lecture 22 - Writing Testable Code



