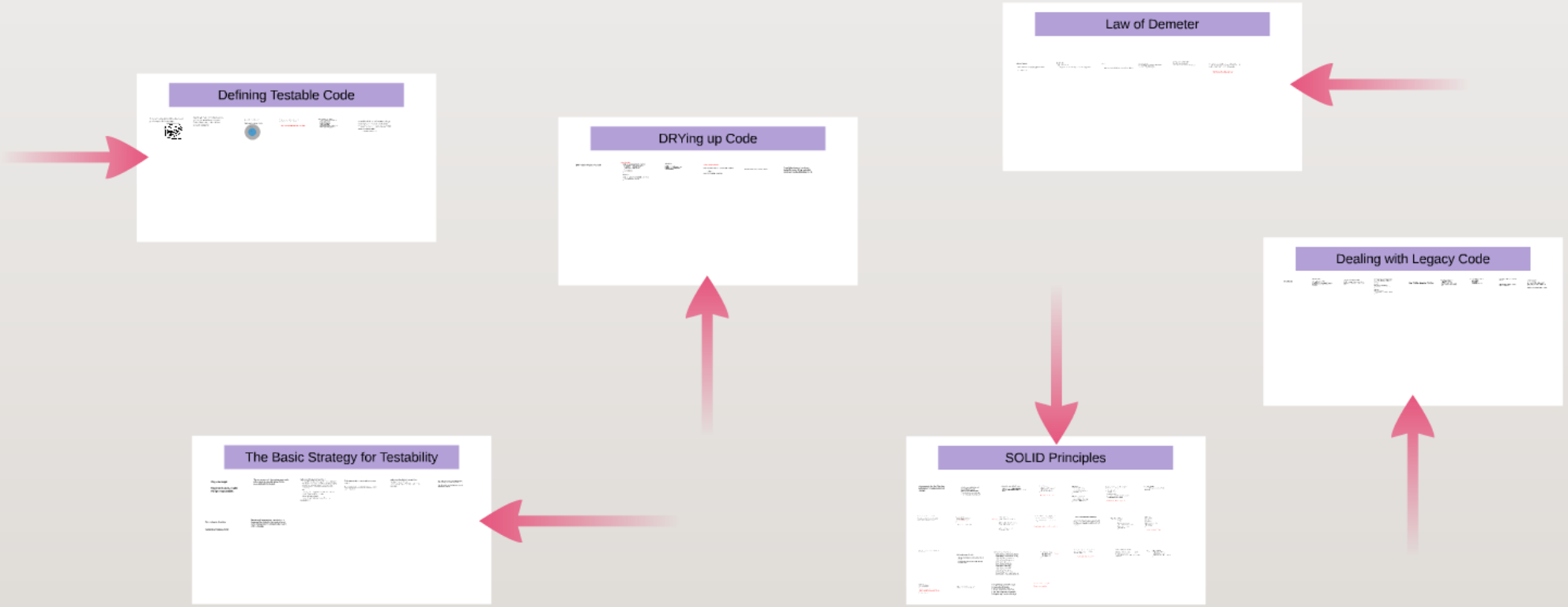
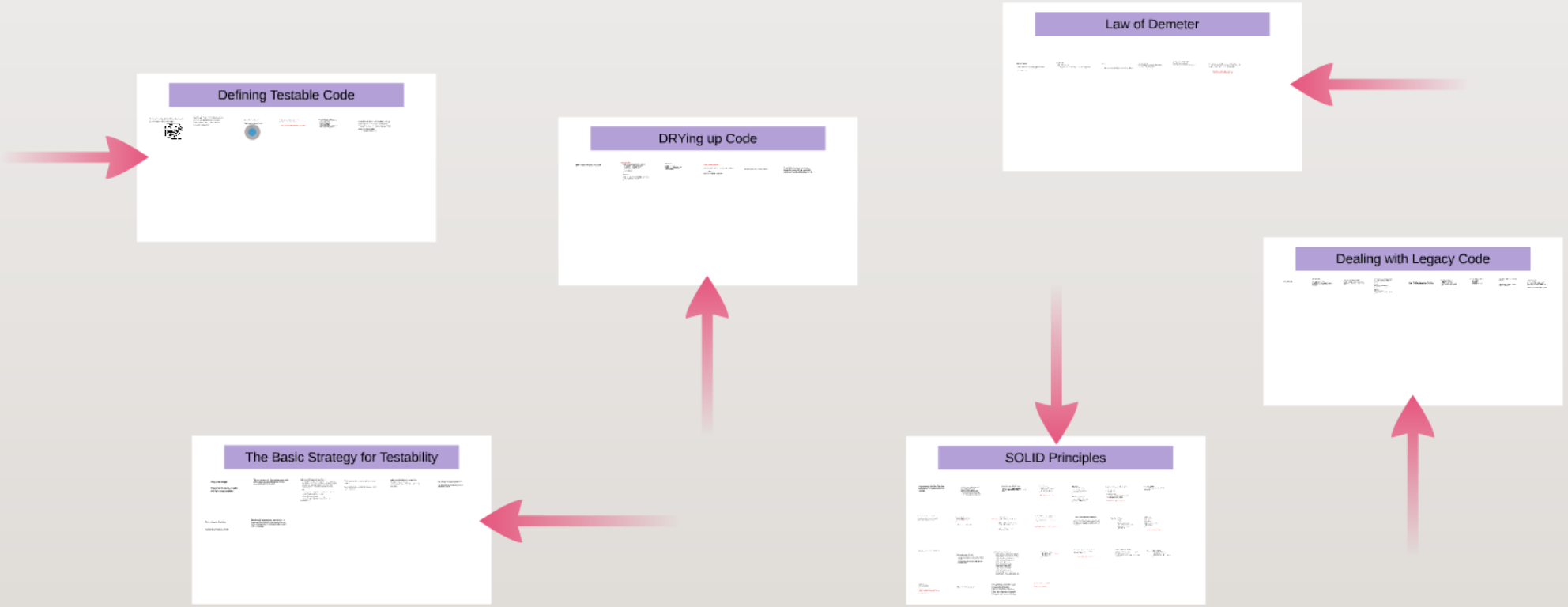
CS1699: Lecture 22 - Writing Testable Code



Prezi

CS1699: Lecture 22 - Writing Testable Code



Prezi

Defining Testable Code

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

PROBLEM?

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!

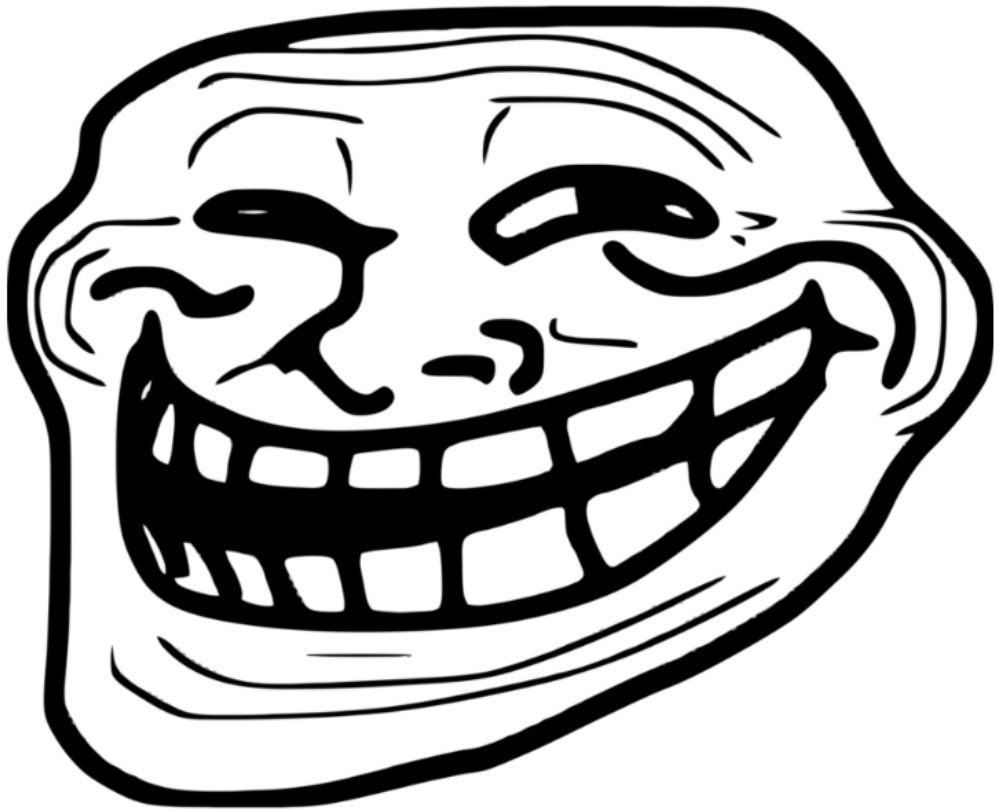
In this lecture, we'll cover -
1. Basic Strategy for Testability
2. The DRY Concept
3. SOLID Principles
4. Law of Demeter
5. Minimizing Mutable Global State
6. Dealing with Legacy Code

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

-Chris Umbel, Software Engineer

Prezi

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



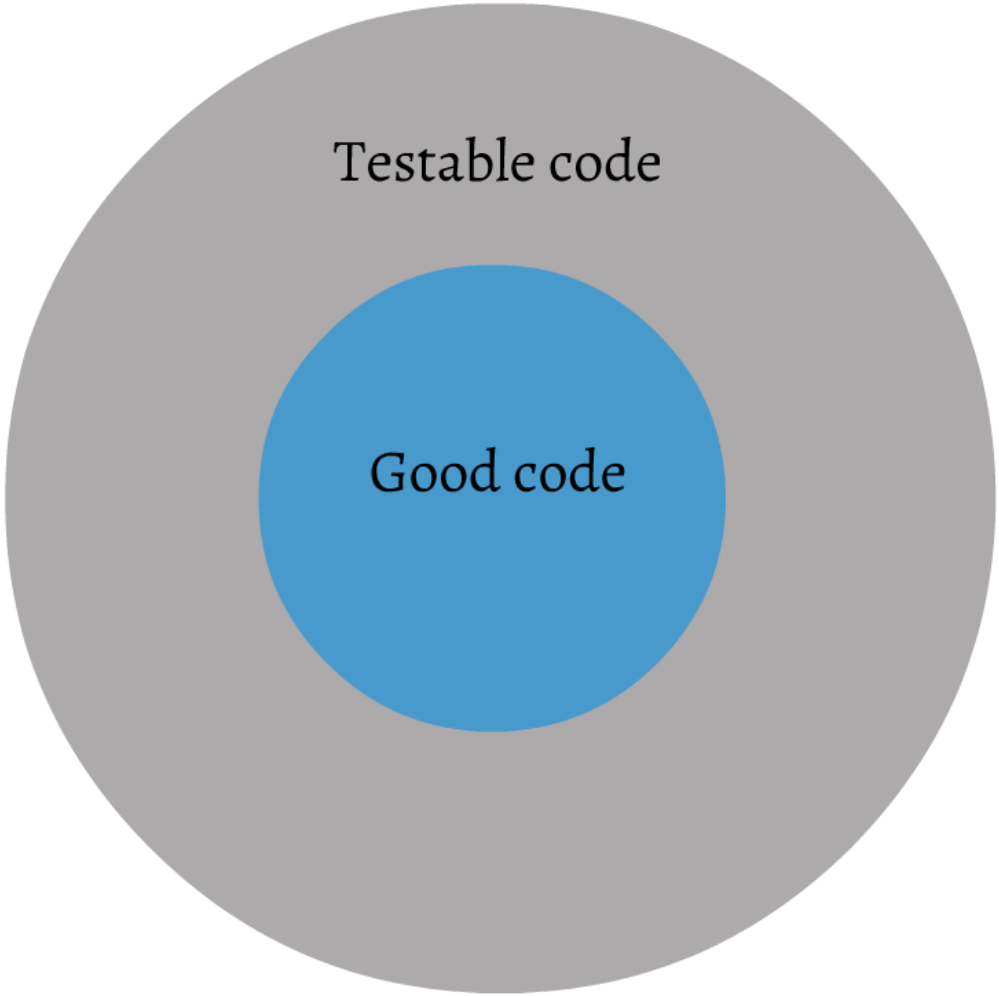
Prezi

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

Prezi

Good code is testable code.

Not all testable code is good code.



Prezi

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!

Prezi

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**

Prezi

"Testable code is one of those funny thin
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

Prezi

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 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.

Prezi

Key concept:

Segment code, make things repeatable.

Prezi

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

Prezi

```
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());
}
```

Prezi

Think about everything that depends on to execute properly.

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

Prezi

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

Prezi

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

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

Prezi

Pure vs Impure functions

MINIMIZE MUTABLE STATE!

Prezi

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

Prezi

DRYing up Code

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[j] = lhs[j] + rhs[j];
  }
  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.

Prezi

DRY = Don't Repeat Yourself

Prezi

Simple Example

```
public int[] addArrays(int[] lhs, int[] rhs) {
  int[] toReturn = new int[lhs.length];
  for (int j=0; j < lhs.length; j++) {
    toReturn[j] = lhs[j] + rhs[j];
  }
  return toReturn;
}

import fj.*;
...
public int[] zipWithAddition(int[] lhs, int[] rhs) {
  return zipWith(lhs, rhs, add);
}
```

Prezi

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

Prezi

A more insidious example...

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

        ... later ...

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

Prezi

Why not make a getName(id) method?

Prezi

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

Prezi

SOLID Principles

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 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() { ... }
}
```

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.

Better way:

```
abstract class Printer {
  private void formatDocument() { ... }
}

public class PhysicalPrinter extends Printer {
  private void printDocument() { ... }
}

public class PdfPrinter extends Printer {
  public void printPdf() { ... }
}
```

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:

What's wrong with this?

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();
}
```

Better

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

Allows for dependency injection!

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

Prezi

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

Prezi

S Single Responsibility Principle

O Open/Closed Principle

L Liskov Substitution Principle

I Interface Segregation Principle

D Dependency Inversion Principle

Prezi

Single Responsibility Principle

A class should have a single responsibility.
That responsibility should be entirely encapsulated by the class.

Prezi

```
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?

Prezi

```
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() { ... }
}
```

Prezi

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?

Prezi

Open / Closed Principle

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

Prezi

Add features by subclassing, not adding code.

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

Prezi

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

Violation of Open/Closed Principle!

Prezi

Better way:

```
abstract class Printer {
  private void formatDocument() { ... }
}

public class PhysicalPrinter extends Printer {
  private void printDocument() { ... }
}

public class PdfPrinter extends Printer {
  public void printPdf() { ... }
}
```

Prezi

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?

Prezi

Liskov Substitution Principle

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

Prezi

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;
}
```

Prezi

```
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?

Prezi

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

Prezi

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.

Prezi

```
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();
}
```

Prezi

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

Better

Prezi

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?

Prezi

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.

Prezi

Example

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

Prezi

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

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

Leaky abstractions are bad.

Prezi

Allows for dependency injection!

Prezi

S Single Responsibility Principle

O Open/Closed Principle

L Liskov Substitution Principle

I Interface Segregation Principle

D Dependency Inversion Principle

Prezi

Remember - these are principles, NOT laws.

Use common sense when applying.

Prezi

Law of Demeter

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:

```
pig_latin_name = PigLatinizer.pig_latinize(name)
```

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.

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

How does this help us test?

Prezi

The Law of Demeter*

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

* Not an actual law.

Prezi

Example:

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

Prezi

Better:

```
pig_latin_name = PigLatinizer.pig_latinize(name)
```

Prezi

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.

Prezi

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

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

Prezi

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

How does this help us test?

Prezi

Dealing with Legacy Code

It's difficult.

Key pieces of advice:

1. Write tests as you go along
2. Look for seams
3. Know/Create TUFs (Test-Unfriendly Features) so that they're not inside TUCs (Test-Unfriendly Constructs)

You can start TDDing from a given point.

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

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

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
- 99 things in the filesystem
- Communicating across the network
- Sets affect AJ code (e.g. GUI operation)
- 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/WorkingWithLegacyCode.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!

Prezi

It's difficult.

Prezi

Key pieces of advice:

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

Prezi

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.

Prezi

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();
}
```

Prezi

No TUFs inside TUCs

Prezi

TUF = Test-Unfriendly Feature

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

Prezi

TUC = Test-Unfriendly Construct

- **Private methods**
- **Final methods**
- **Final classes**
- **Constructors / Destructors**

Prezi

"Working with Legacy Code" by Michael Feathers

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

Prezi

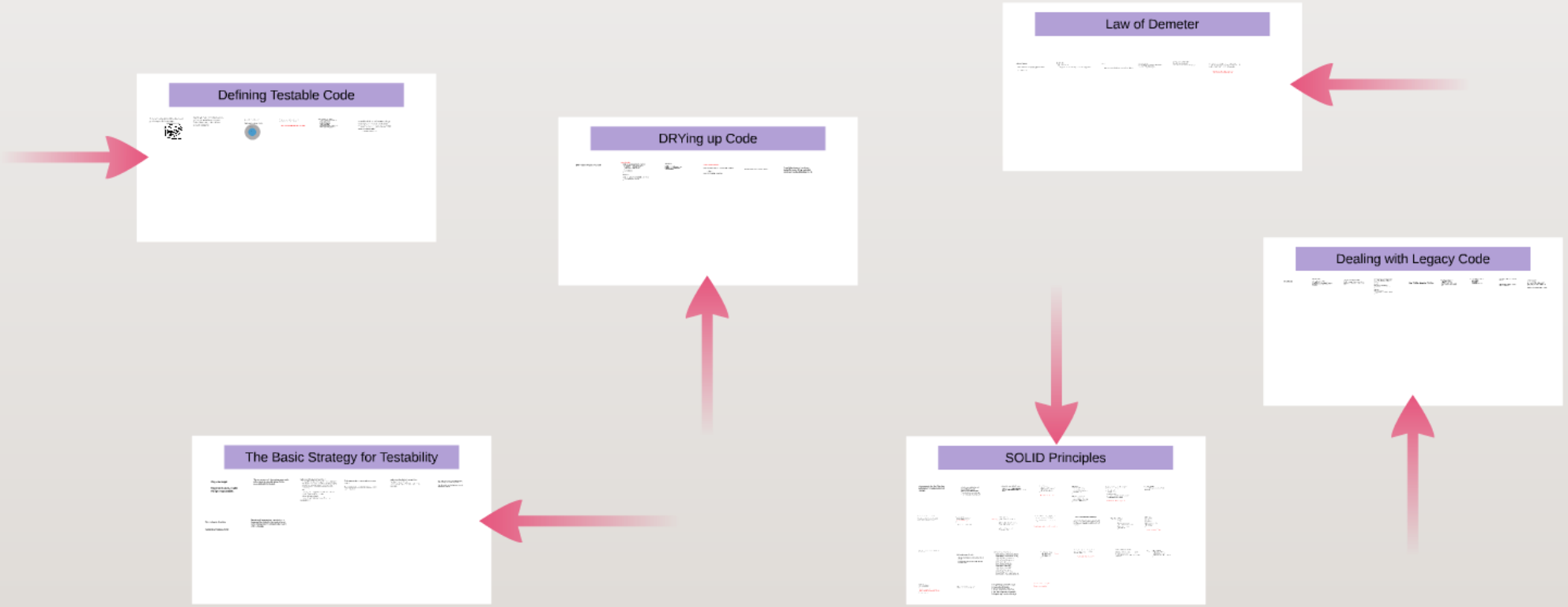
Don't be discouraged.

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!

Prezi

CS1699: Lecture 22 - Writing Testable Code



Prezi