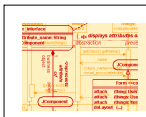
Design Patterns and Architecture

https://holub.com/patterns
https://holub.com/patterns/book.pdf

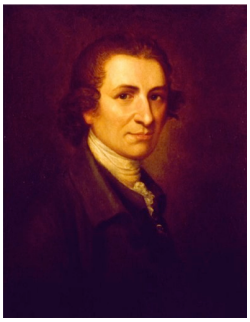
Allen Holub
www.holub.com
allen@holub.com
@allenholub

http://holub.com/slides

1



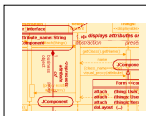
Familiar ≠ Correct



"A long habit of not thinking a thing wrong, gives it a superficial appearance of being right, and raises at first a formidable outcry in defense of custom."

-Thomas Paine

2

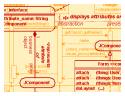


Dunning/Kruger effect

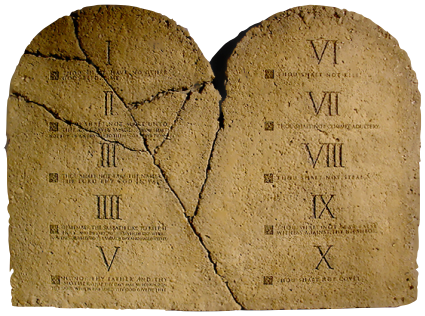
The skills that enable one to construct a grammatical sentence are the same skills necessary to recognize a grammatical sentence, and thus are the same skills necessary to determine if a grammatical mistake has been made.

Kruger and Dunning: Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments.

3



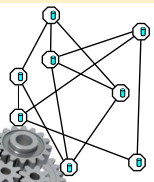
Basic Principles.



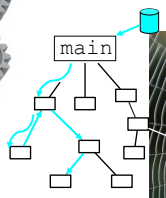
4

OO ≠ Procedural

- Cloud of peers.
- Messages flow; data stays put.



- Centralized control.
- Data passed between functions.



5

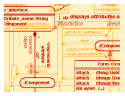
What does it *do*?

Objects are defined by what they *do*, not what they contain.

- ➡ objects ≠ data + functions.
- ➡ objects have *responsibilities*, not data.

Hide the way the object does the work (*Encapsulation*).

6



Delegation

**Ask for help,
not information.**

Don't ask an object to give you the information you need to do the work — ask the object that has the information to do the work for you.

7

Orthogonality

Changes to an object/class should not impact other objects/classes.

No side effects!

8

Holub Replacement Principal

You must be able to replace the implementation of a class without affecting the clients.

9

Symptoms of bad design

- Rigidity - hard to change.
- Fragility - easy to break.
- Immobility - hard to reuse.
- Viscosity - easier to hack than fix properly
- Complexity
- Repetition - duplicate code / bad structure
- Opacity - hard to understand.

10

S.O.L.I.D. Principles

Single Responsibility
Open Closed
Liskov Substitution
Interface Segregation
Dependency Inversion

11

Single Responsibility Principal

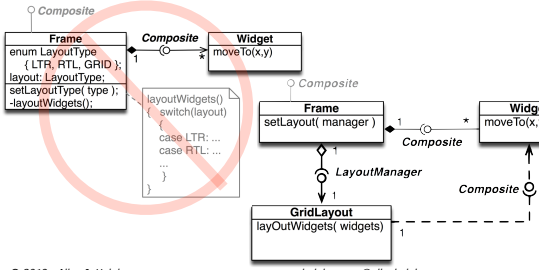
12

13

Open-Closed Principle

Open for extension;
Closed for modification.

Changes shouldn't ripple out to dependent classes/objects/modules



14

Fool me twice, shame on me.

"There's an old saying in Tennessee—I know it's in Texas, probably in Tennessee—that says, fool me once, shame on—shame on you. Fool me—[long pause]—you can't get fooled again."

G. W. Bush, Nashville, Tenn., Sept. 17, 2002

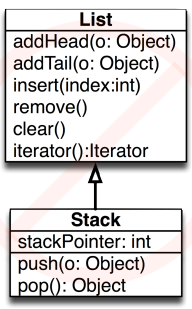
Go with the simplest solution first, but if you have to change it, do it right!

15

Liskov Substitution Principle

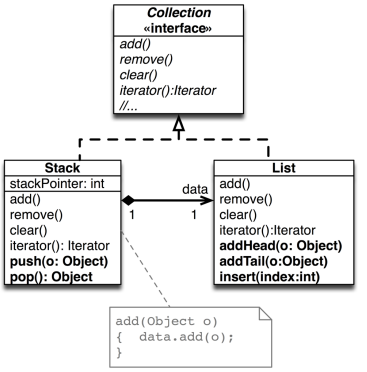
Subtypes must be substitutable for their base types.

```
f( List l )
{
  l.clear();

  l.addHead("x");
}
Stack aStack...;
f(aStack);
```



Context Matters!

An LSP-compliant Stack



List contains methods meaningful to a List.

Stack contains methods meaningful to a Stack.

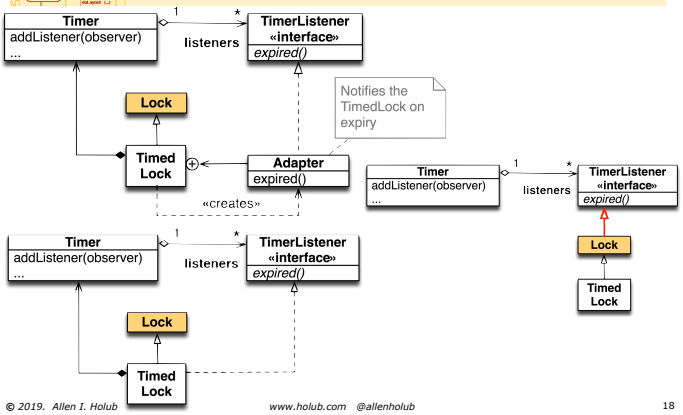
Interface-Segregation Principle

Avoid FAT interfaces

Don't force clients to depend on methods they do not use.

- minimize coupling
- maximize cohesion

Fixing ISP problems



Dependency Inversion

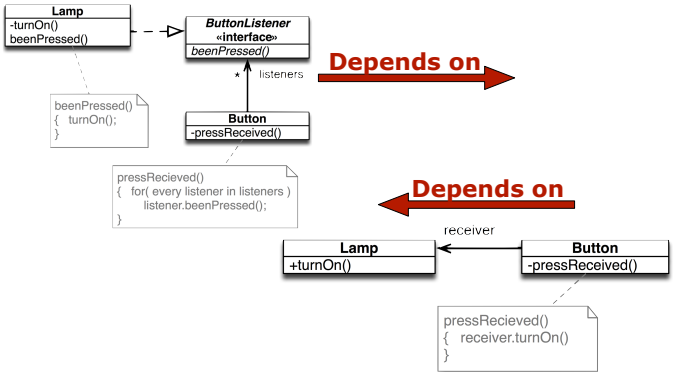
High-level modules should not depend on low-level modules.

They should both depend on abstractions.

The details should be in the implementation, not the abstraction.

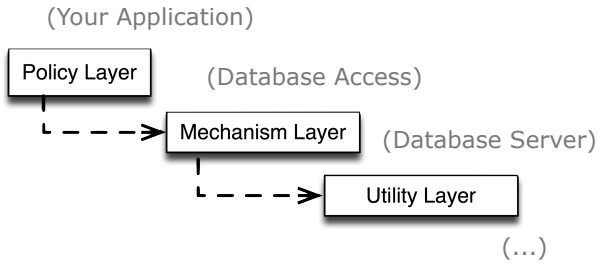
19

The Hollywood Principle: Don't call us, we'll call you



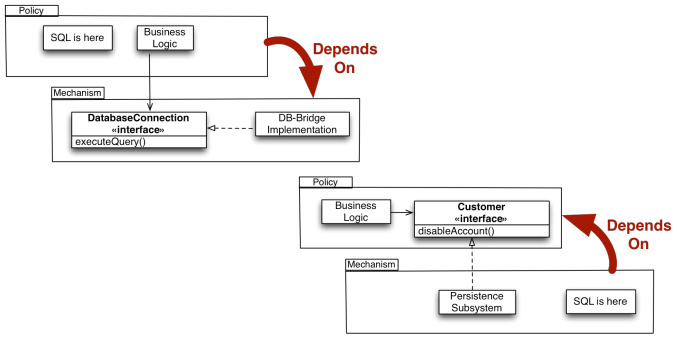
20

Module-level Dependancies



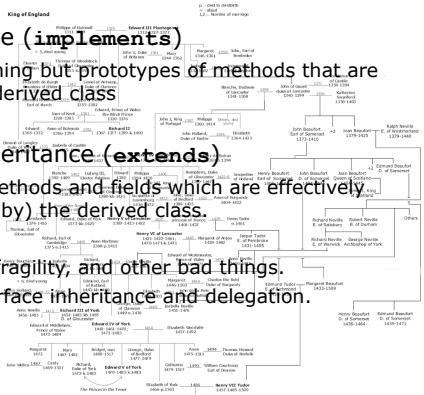
21

DI Example ⋄ Persistence



Two kinds of Inheritance

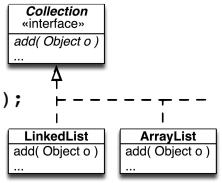
- Interface inheritance (**`implements`**)
  - The base class is nothing but prototypes of methods that are implemented by the derived class
- Implementation inheritance (**`extends`**)
  - The base class has methods and fields which are effectively part of (are inherited by) the derived class.
  - **Too-Tight coupling**!
    - Increases rigidity, fragility, and other bad things.
  - Can replace with interface inheritance and delegation.



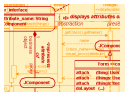
Interfaces ⇒ Flexibility

```
LinkedList list = new LinkedList();
g( list );

g( LinkedList list )
{   list.add( ... );
    g2( list )
}

Collection list = new LinkedList();
g( list );

g( Collection list )
{   list.add( ... );
    g2( list )
}
```

Changing the list type impacts `g()`.



Changing the list type ***doesn't*** impact `g()`.

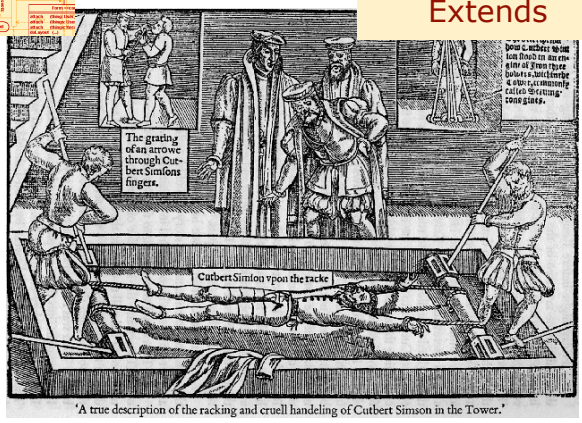


Design patterns add even more flexibility

```
void f2()
{   Collection c = new HashSet();
    //...
    g2( c.iterator() );
}

void g2( Iterator i )
{   while( i.hasNext() ;)
        do_something_with( i.next() );
}
```


25

Problems with Extends



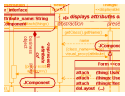
'A true description of the racking and cruell handeling of Cutbert Simson in the Tower.'

26

class != data + methods.

Classes are defined by their responsibilities, not their contents.

27



Roles, not Actors

"Managers" are not "employees" if their responsibilities do not overlap .

28

Behavior is everything!

IS-A

really means:

Behaves exactly like...

29

When "is-a" fails.

- Manager *is an* Employee in every sense, but is categorized differently.
- Manager authorizes time sheet, Employee fills it out.

✓ Mangers do a little more than normal Employees.

Additional responsibilities ⇒ implementation Inheritance.

Non-overlapping responsibilities ⇒ distinct classes

Identical responsibilities ⇒ identical classes

30

Fragility

- Derived classes often depend on base class behaving in a certain way.
- If you change the behavior of a base-class method, you can break the derived class.
- This base-class change is often an IMPROVEMENT.

31

Consider this code

```
class Stack extends ArrayList
{
  private int stackPointer = 0;


  public void push( Object article )
  {   add( stackPointer++, article );
  }


  public Object pop()
  {   return remove( --stackPointer );
  }
```


32

So what's wrong?

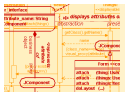
- What if a user leverages inheritance and uses the `ArrayList`'s `clear()` method to pop everything off the stack:

```
Stack aStack = new Stack();
 aStack.push("1");
 aStack.push("2");
 aStack.clear();
aStack.addHead();
aStack.removeHead();
```

- `stackPointer` still points at `stack[1]`.

The stack now holds garbage.

33



Liskov Substitution Principle

34

Subtypes must be substitutable for their base types.

It must make sense to send any/all base-class messages to subclass objects.

How about using encapsulation?

35

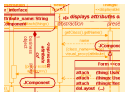
```
class Stack extends ArrayList
{   private int stackPointer  = 0;
    private ArrayList theData = new ArrayList();
    public void push( Object article )
    {   theData.add( stackPointer++, article );
    }
    public Object pop()
    {   return theData.remove( --stackPointer );
    }
    public void pushMany( Object[] articles )
    {   for( int i = 0; i < o.length; ++i )
            push( articles[i] );
    }
}
```

There's no `clear()` [that's good]. **But … →**

Now we'll extend to add behavior

36

```
class MonitorableStack extends Stack
{   private int highWaterMark = 0;     ←Added
    private int currentSize;
    @Override public void push( Object article )
    {   if( ++currentSize > highWaterMark )
            highWaterMark = currentSize;
        super.push(article);
    }
    @Override public Object pop()
    {   --currentSize;
        return super.pop();
    }

    public int maximumSizeSoFar()  ←This is NEW
    {   return highWaterMark;
    }
    ← pushMany() IS INHERITED                →
}
```



Someone *improves* the base class

```
class Stack
{   private int stackPointer = -1;
    private Object[] stack = new Object[1000];
    public void push( Object article )
    {   //...
    }
    //...
    public void pushMany( Object[] articles )
    {   assert (stackPointer + articles.length) < stack.length;
        System.arraycopy(articles, 0, stack, stackPointer+1,
                                              articles.length);
        stackPointer += articles.length;
    }
}
```

No longer calls `push()`

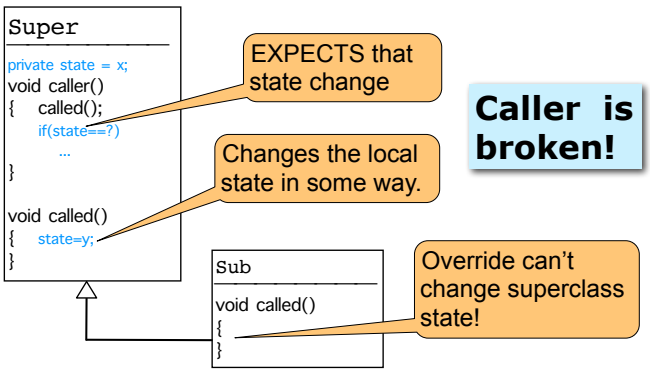
But...

```
MonitorableStack myStack = new MonitorableStack();
myStack.pushMany(new String[]{“a”, “b”, “c”});
int size = myStack.maximum_size_so_far();
```

What’s the value of `size`?

0

A common variant



Caller is broken!

Let's fix it!

```
interface Stack
{   void push( Object o );
   Object pop();
    void pushMany( Object[] source );
}

 class SimpleStack implements Stack
 {   private int stackPointer  = 0;
     private ArrayList theData = new Arr
     public void push( Object article )
     {   the_data.add( stackPointer++, a
     }
     public Object pop()
     {   return theData.remove( --stackP
     }
     public void pushMany( Object[] arti
     {   for( int i = 0; i < o.length; i
```

Exactly like earlier Stack class

Fixed version

```
class MonitorableStack implements Stack
{  private SimpleStack stack = new SimpleSta
   private int highWaterMark = 0, currentSize
   public void push( Object o )
    {   if( ++currentSize > highWaterMark )
            highWater_mark = currentSize;
        stack.push(o);
    }
    //...
    public void pushMany( Object[] source )
    { if( current_size + source.length > highWaterMark )
            highWaterMark = currentSize + source.length;
        stack.pushMany( source );
    }
    //...
}
```

Because we're using interface inheritance,

We delegate to `SimpleStack`, which could be a base class, but isn't

and we're forced to implement `push_many()`,

which also delegates.

Delegation/Inheritance pattern

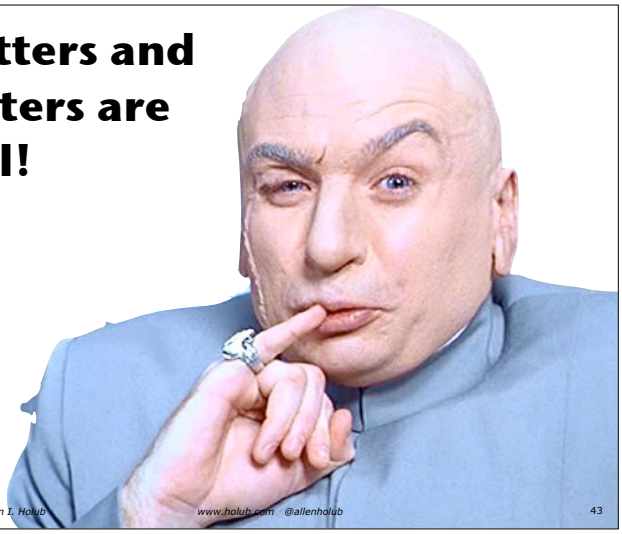
Rather than:

```
class Simple{ void f(){ /*...*/ }     }
class Specialization extends Simple{ /*...*/ }
```

Use:

```
interface Simple
{   void f();
    static class Implementation implements Simple
    {   void f(){/*reasonable default implementation*/}
    }
}
class Specialization implements Simple
{   Simple delegate = new Simple.Implementation();
    void f(){ delegate.f(); }
}
```

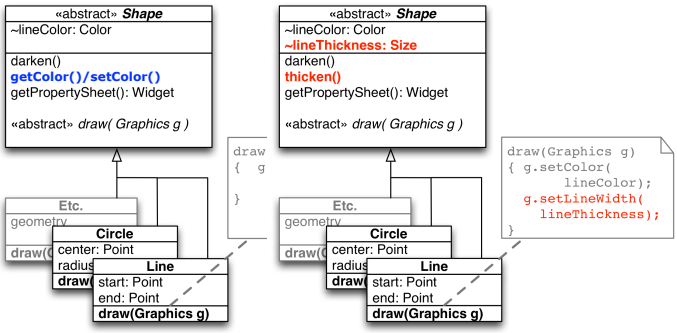
Getters and setters are evil!



43

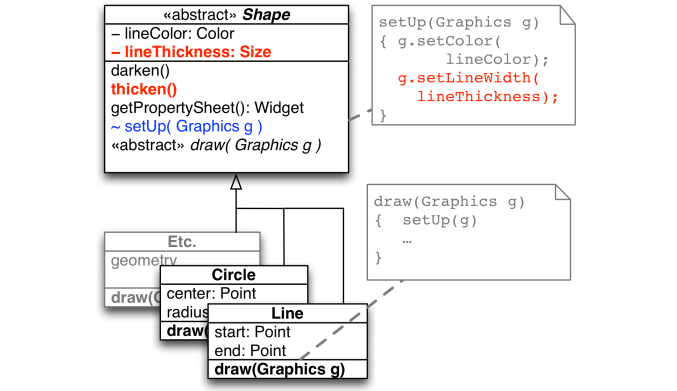
A Challenge

Add new property without changing derived classes!

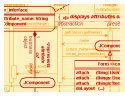


44

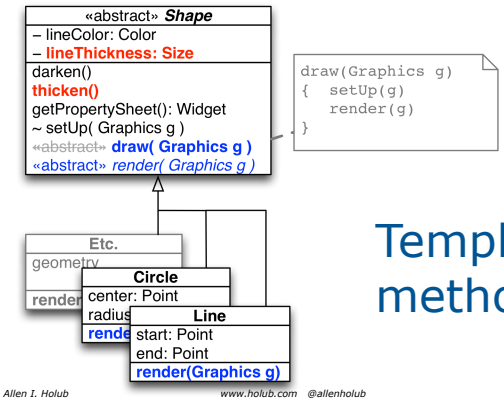
A solution (delegation)



45



A design-pattern solution



Template method

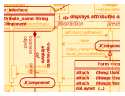
Do not ask for the information that you need to do the work.

Ask the object that has the information to do the work for you.

Delegation

The object that has the information should do the work!

```
System.out.println( someString );

new PrintWriter(
 new BufferedWriter(
  new OutputStreamWriter(System.out )))
                .println( someString );

someString.print( System.out );
```

Ask for help, not information.

```
class Money
{
  private double value;
  public double getValue()          { return value; }
  public void   setValue(double v)  { value = v; }
  //...
}
```

49

Ask for help, not information.

```
Customer remoteCustomer = getRemoteCustomer();
Money  request    = ...;
Money  balance    = remoteCustomer.getBalance();
double balanceVal = balance.getValue();
double requestVal = request.getValue();

if( balance.getCurrency() != EURO )
  balanceVal =
     CurrencyConverter.convert( balance.getValue(),
                        balance.getCurrency(), EURO);
if( requested.getCurrency() != Currency.EURO )
  requestVal =
     CurrencyConverter.convert( requested.getValue(),
                        requested.getCurrency(), EURO);
if( requestVal < balanceVal)
  dispenseFunds( requested );
```

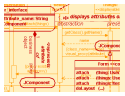

50

Ask for help, not information.

```
Customer remoteCustomer = getRemoteCustomer();
Money requested         = ...;

if( remoteCustomer.yourBalanceIsAtLeast(requested) )
  dispenseFunds( requested );
```

```
class Money
{ private double   value;
  private Currency currency;
  public boolean largerThan(Money m)
  {
     if( currency != m.currency )
        m=currency.covertToYourCurrency(m);
     return value > m.value ;
  }
  public Money  addTo   ( Money m )   {...}
  public Writer printTo ( Writer out ){...}
  public String toXML   ()            {...}
}
```


51



Properties are Worse

```
string x = someObject.AProperty;
```

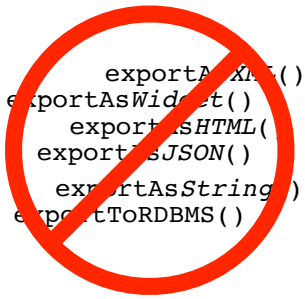
```
class Foo
{   public int MyProp
  { get{ return readValueFromFile("myProp"); }
    set{ modifyFile("myProp", value); }
  }
}
```

Could crash the server!

Throws exception when it can't open, parse, etc.

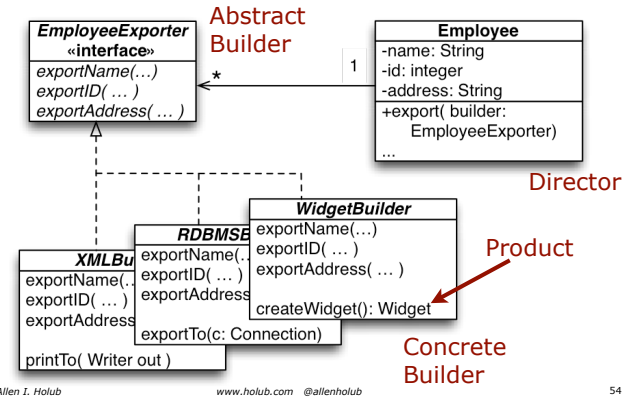
52

"Draw yourself" isn't practical

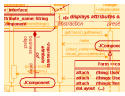


53

The *Builder* design pattern



54



Building output

```
    JSONBuilder exporter = new   JSONBuilder();
Employee.exportTo( exporter );
exporter.printTo( response );
  HTMLBuilder exporter = new   HTMLBuilder();
Employee.exportTo( exporter );
exporter.printTo( response );
 RDBMSBuilder exporter = new  RDBMSBuilder();
Employee.exportTo( exporter );
exporter.storeIn( database );
WidgetBuilder exporter = new WidgetBuilder();
Employee.exportTo( builder );
someFrame.add( builder.getComponent() );
```


55

Building objects

```
 JSONImporter importer = new  JSONImporter(stream);
Employee fred = new Employee( importer );
```

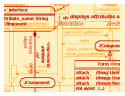
```
RDBMSImporter importer = new RDBMSImporter(stream);
Employee fred = new Employee( importer );
```


56

A *Director*

```
public class Employee
{   private Name        name;
    private EmployeeId  id;
    private Money       salary;

    public interface Exporter
    {   void addName    ( String name   );
        void addID      ( String id     );
        void addSalary  ( String salary );
    }

    public void export( Exporter builder )
    {   builder.addName  ( name.toString()   );
        builder.addID    ( id.toString()     );
        builder.addSalary( salary.toString() );
    }
```

57



A *Director*

```
public interface Importer
{   String provideName();
    String provideID();
    String provideSalary();
    void   open();
    void   close();
}

public Employee( Importer builder )
{ builder.open();
  this.name = new Name      (builder.provideName() );
  this.id   = new EmployeeId(builder.provideID()   );
  this.salary = new Money   (builder.provideSalary(),
                             new Locale("en","US"));
  builder.close();
 }
 //...
}
```


Build a *Product* (HTML)

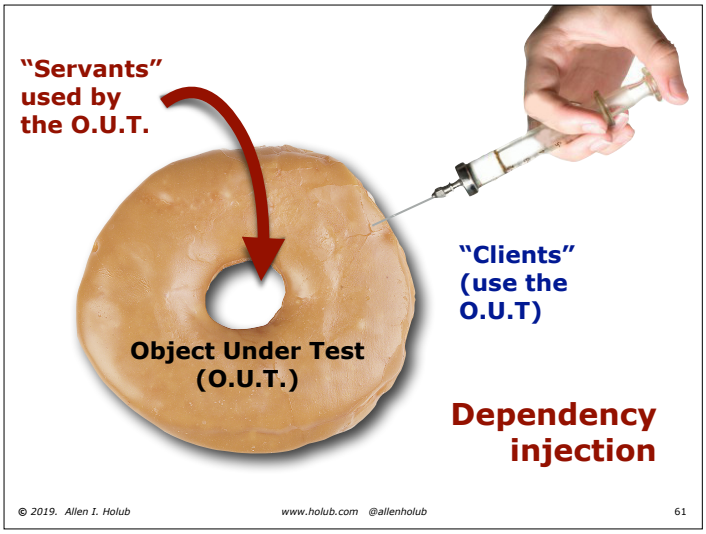
```
HTMLExporter implements Employee.Exporter
{   private final String       HEADER = "<table border=\"0\">\n";
    private final StringBuffer out    = new StringBuffer(HEADER);

    public void addName( String name )
    {   out.append("\t<tr><td>Name:</td><td>" );
        out.append("<input type=\"text\" name=\"name\"value=\"");
        out.append( name );
        out.append("\"></td></tr>\n" );
    }
    public void addID    ( String        ) { /*.. .*/ }
    public void addSalary( String salary ) { /*.. .*/ }
    String getHTML()
    {   out.append("</table>");
        String html = out.toString();
        out.setLength(0);                    // erase the buffer
        out.append(HEADER);
        return html;
    }
}
```

```
HTML Exporter e = new HtmlExporter;
someEmployee.export( e );
someStream.print( e.getHTML() );
```


Build an object (from HTML form)

```
class HTMLImporter implements Employee.Importer
{   ServletRequest request;
    public void open() { /*nothing to do*/ }
    public void close(){ /*nothing to do*/ }
    public HTMLImporter( ServletRequest request )
    {   this.request = request;
    }
    public String provideName()
    {   return request.getParameter("name");
    }
    public String provideID()
    {   return request.getParameter("id");
    }
    public String provideSalary()
    {   return request.getParameter("salary");
    }
}

    Employee e =
          new Employee( new HTMLImporter(request) );
```

61



62

So, how do you test when you don't know what's in the object?

63

Decouple tests from the code.

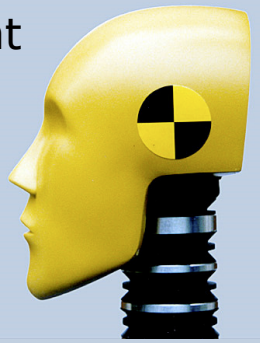
Holub Replacement Principle:

You should be able to replace a class's implementation without impacting the clients.

64

The Replacement Principle applies here, too.

The test is a "client."



Tests should be abstract

65

Your tests should know as little as possible about how the object works.

Test by looking at external behavior. Was the email actually sent? Did the tail wag?

Don't assume implementation

66

```
Dollar d …;

assertEqual(d.value(), 5.0);
```

Assumes value is a float!

Train Wreck

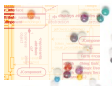
67

```
dog.getBody().getTail().wag();
```

Use this instead

68

```
dog.expressHappiness();
```



The Donut

69

"Servants" used by the O.U.T.

"Clients" (use the O.U.T)

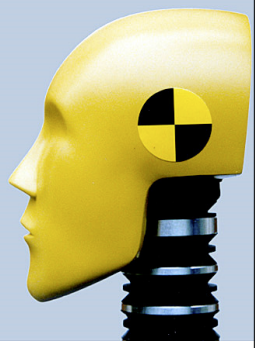
Object Under Test (O.U.T.)

Test using mock objects.

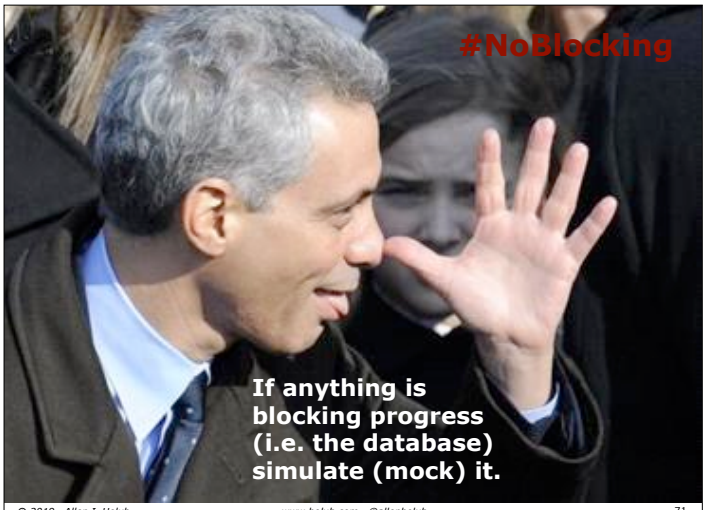
Simulate the clients and servants.

Infer behavior of the O.U.T. by looking at how it interacts with the mocks

Write your own, or use a mocking framework.



70



71

```
class Employee ( ){
   private String name;
   private Connection con = new JSONDbConnection();
   private URL location   = new URL("JSONDb://...");
   public Employee( EmployeeID id ){
      con.open(location);
      JSON result = con.query(id);
      name = result("name");
      con.close();
   }
}

Employee fred = new Employee( FREDS_ID );
```


72

73

```
class Employee ( ){
   private String name;
   private Connection con;
   public Employee( EmployeeID id, Connection con){
       this.con = con;
       JSON result = con.query(id);
       name = result("name");
   }
}
Connection con = new JSONDbConnection();
URL location   = new URL("JSONDb://...");
con.open(location);
Employee fred = new Employee(FREDS_ID , con );
con.close();
```

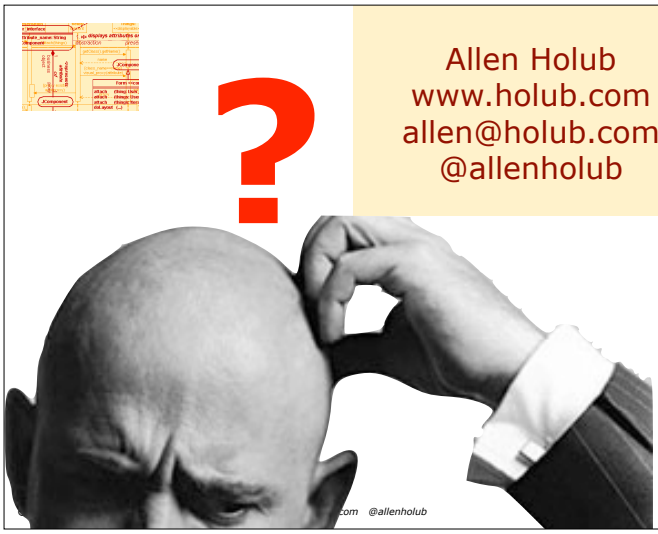

74

```
class Employee ( ){
   private String name;
   private Connection con;
   public Employee( EmployeeID id, Connection con){
       this.con = con;
       JSON result = con.query(id);
       name = result("name");
   }
}
class ConnectionMock extends JSONDbConnection{
   public open(URL location){ log(location); }
   public close(){ }
   public JSON query(EmployeeID id){
      return new JSON("{ 'name':'Fred' }");
   }
}
Connection con = new ConnectionMock();
URL location   = new URL("JSONDb://...");
con.open(location);
Employee fred = new Employee(FREDS_ID, con );
con.close();
```


75

```
class Employee ( ){
   private String name;
   private Connection con;
   public Employee( EmployeeID id, Connection con){
       this.con = con;
       JSON result = con.query(id);
       name = result("name");
   }
}

class ConnectionMock extends JSONDbConnection( ){
   public ConnectionMock(JSONDbConnection wrapped){
       this.wrapped = wrapped;
   }
   public open(URL location){ log(location);
                              wrapped.open(location);}
   public close(){ wrapped.close(); }
   public JSON query(EmployeeID id){
      return wrapped.query(id);
   }
}
```

```
class Employee ( ){
   private String name;
   private Connection con;
   public Employee( EmployeeID id, Connection con){
       this.con = con; = ConnectionFactory.create();
       JSON result = con.query(id);
       name = result("name");
   }
}

class ConnectionMock extends JSONDbConnection( ){
   …
}

class ConnectionFactory {
    public static Connection create(){
        Connection actual = new JSONDbConnection();
        return !TESTING ? actual :
                          new ConnectionMock(actual);
}
```

Allen Holub
www.holub.com
allen@holub.com
@allenholub