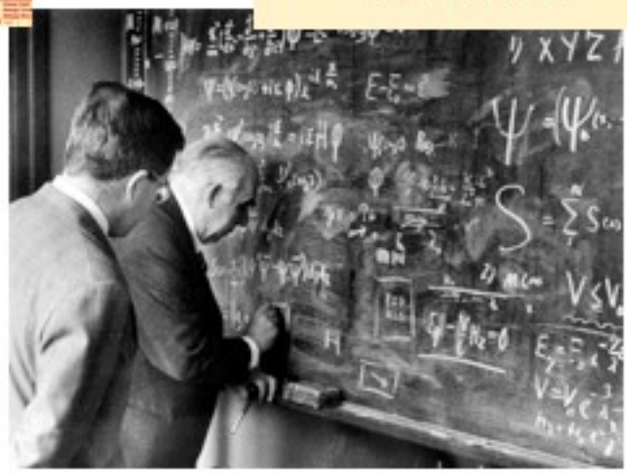
Agile Architecture

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

The Problem



Aage & Niels Bohr. (sciencephoto.com)

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

There are many badly designed libraries with millions of users (Struts, Spring, EJB, ...)
Just because it looks like X, doesn't mean that it's "good."

People don't know what they don't know

...consider the ability to write grammatical English. 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. In short, the same knowledge that underlies the ability to produce correct judgment is also the knowledge that underlies the ability to recognize correct judgment. To lack the former is to be deficient in the latter.

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

 www.holub.com 4

Justin Kruger and David Dunning's original journal article is at http://www.apa.org/journals/psp/psp7761121.html. There's a copy on my web site at http://www.holub.com/goodies/DunningKruger.pdf .
It's easier to get David Dunning's book: *Self-Insight: Roadblocks and Detours on the Path to Knowing Thyself (Essays in Social Psychology)*. ISBN-10: 1841690740.
I've gotten death threats when I've written about this stuff!
Even experienced programmers may know nothing about design.

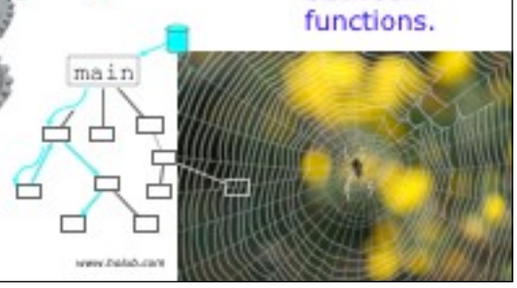
Basic Principles.

 www.holub.com 5

OO ≠ Procedural

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

- Centralized control.
- Data passed between functions.



 www.holub.com

A central controller is a "bad smell"
No "God classes."

Abstract = Flexible

- **The less you know about how it works, the better.**
- Program in terms of the abstraction, not the concrete implementation.

 www.holub.com 7

The Three Wise Monkeys carved on a stable housing sacred horses at Tōshōgū shrine, Nikkō, Japan. Photo Copyright © 2003 David Monniaux (http://en.wikipedia.org/wiki/File:Hear_speak_see_no_evil_Toshogu.jpg) Use List, not LinkedList. Use Composite, not Button.

What is an object?

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*).

 www.holub.com 8

The fact that this box contains a woman is irrelevant to the outside world, and will not impact the interface to box: (open(), close(), etc.)

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.

 www.holub.com 9

Orthogonality

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

No side effects!

Holub Replacement Principal

You should be able to radically change a class without affecting any of the objects that use that class.

The “objects that use the class” are usually called “client” objects.

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.

By “complexity,” I mean “needless complexity.” Sometimes, things are just complicated.
Viscosity can apply to projects as well. Multi-day builds, hours required for testing, everything is difficult.
Repetition (duplicate code) implies that you’re not using derivation or design patterns appropriately. The problem is not just identical code, but similar code as well. Often, results from cut-paste-and-modify strategies for code development.

S.O.L.I.D. Principles

Single Responsibility
Open Closed
Liskov Substitution
Interface Segregation
Dependency Inversion

 www.holub.com

From Robert C. Martin, *Agile Software Development, Principles, Patterns, and Practices*, ISBN: 0135974445
Closet by Livio DeMarchi

Single Responsibility Principal

A class should have only one reason to change.

A Manager authorizes time sheets.
A Peon fills them out.

No shared operations, so they're distinct classes, with minimal coupling.

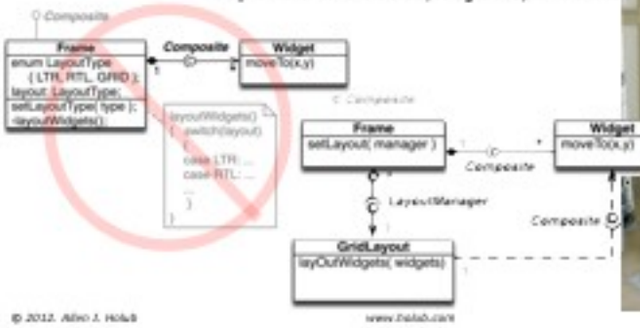
 www.holub.com 14

The fact that managers and peons are both employees irrelevant.

Open-Closed Principle

Open for extension;
Closed for modification.

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



 www.holub.com 15

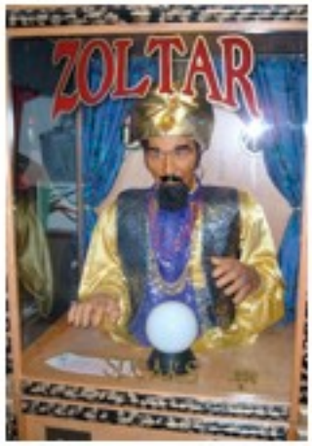
We're trying for classes that are stable in the face of change, so they'll last longer than the first change in requirements.
Should NEVER need to modify base class to add a derived class.
Ideally, adding an extension should not require that another class be recompiled.
Note the direction of the dependencies out of Frame. Frame can change without affecting other classes.

Problems with OCP

You can't predict the future!

You risk adding complexity that solves nonexistent problems

Sometimes, to make one class less rigid, you need to make another more so



 www.holub.com 16

It's better to err on the side of simplicity, at least for the initial implementation of a class. Add interfaces, etc., when they're required.
One can "stimulate" the change to find out where the flexibility is needed. (TDD, short cycles, work on disparate stories that leverage the same classes).

Fool me once, shame on you. 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."

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

 www.holub.com 17

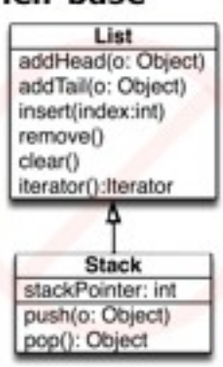
An agile approach mandates going with the simple solution, but the "right" solution often isn't simple. Don't add unnecessary complexity right off the bat, but if the simplistic solution needs refactoring, do it right.

Liskov Substitution Principle

Subtypes must be substitutable for their base types.

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

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

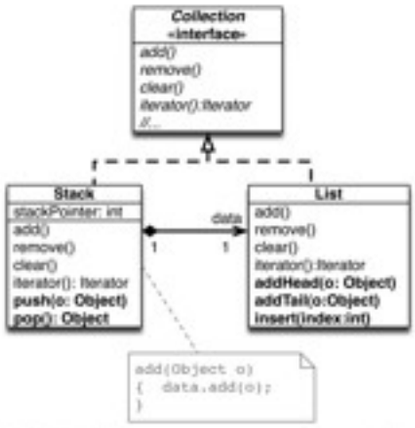


Context Matters!

 www.holub.com 18

Barbara Liskov, Data Abstraction and Hierarchy. SIGPLAN Notices, 23.5 (May, 1988)
Corollary: subtypes must be substitutable for each other.
Is-a FAILS, here. A Stack is not the same thing as a list!
Inverse: If you never use inherited methods, then you shouldn't be using inheritance (e.g. Window/Dialog)
In example, you could implement clear() in the Stack, but addHead() has no meaning whatever to a stack!

An LSP-compliant Stack



List contains methods meaningful to a List.

Stack contains methods meaningful to a Stack.

Interface-Segregation Principle

Avoid FAT interfaces

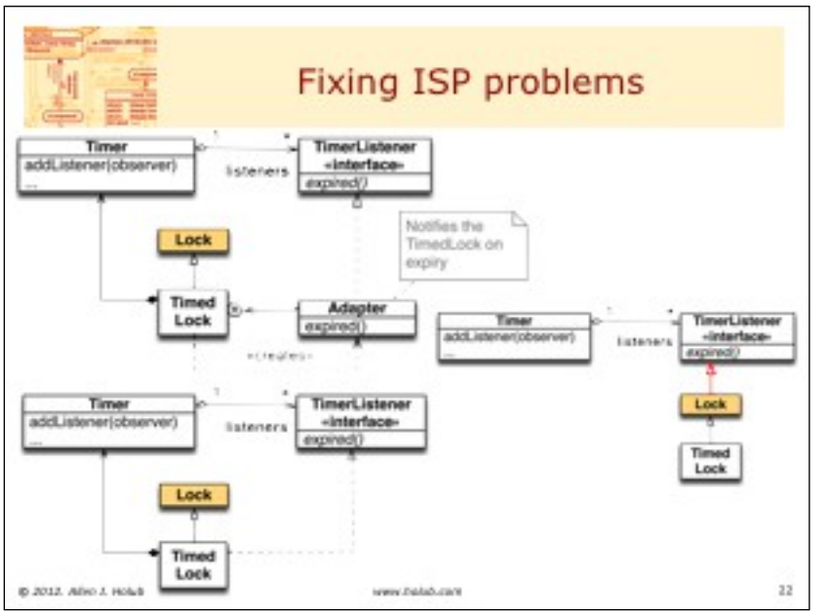
Don't force clients to depend on methods they do not use.

- minimize coupling
- maximize cohesion

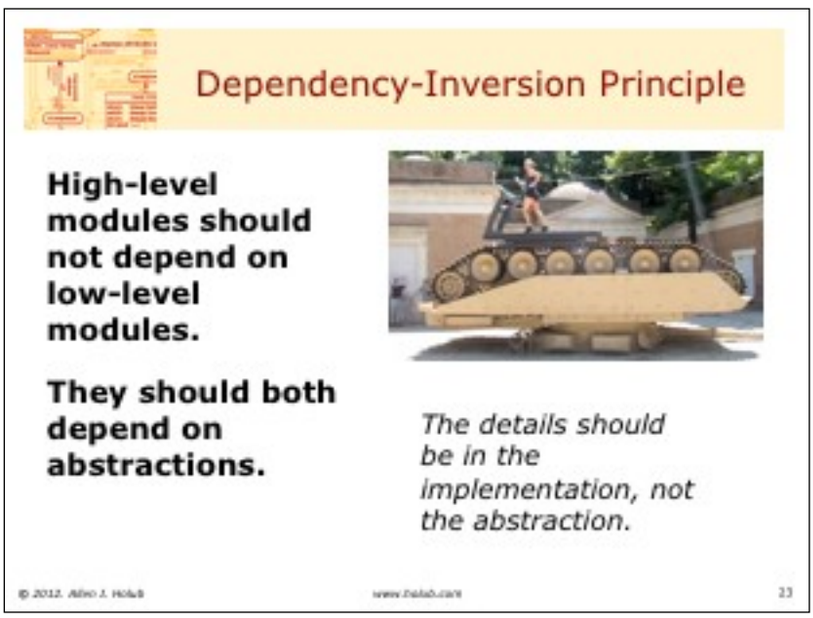
ISP Bad Smells

- "Extends" adds generic capability.
- Empty implementations of interface methods.
- Base-class overrides do nothing but throw exceptions.
- Changing an interface method affects classes that don't use that method.
  - A recompile counts as an effect.

Some of these symptoms are also symptoms of LSP



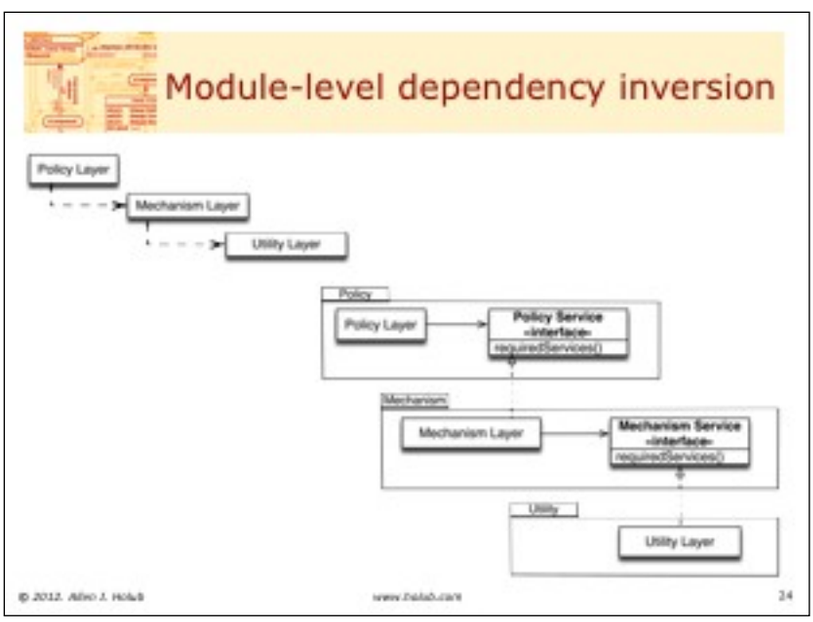
The timer illustrates the Observer pattern, but the real issue is how the pattern is applied.



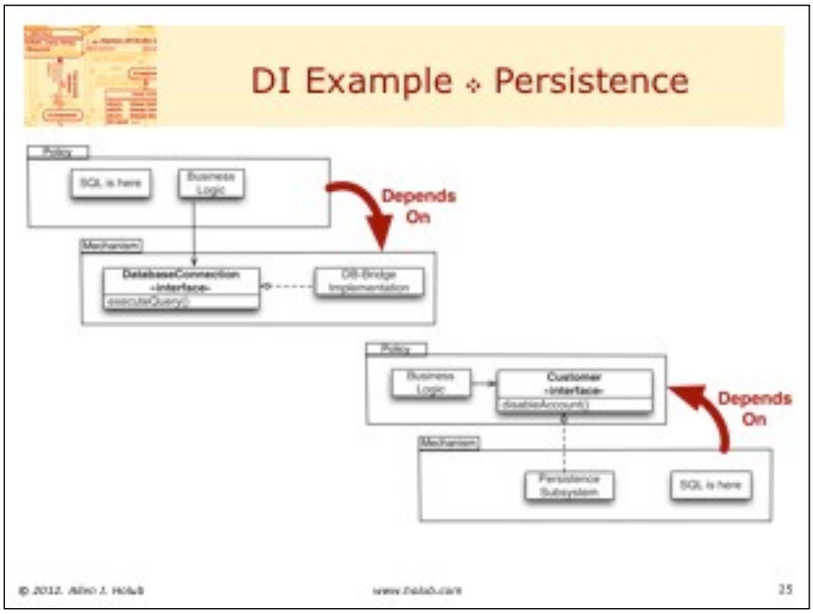
Jennifer Allora and Guillermo Calzadilla's Tank

Treadmill @ 2011 Venice Biennale: http://

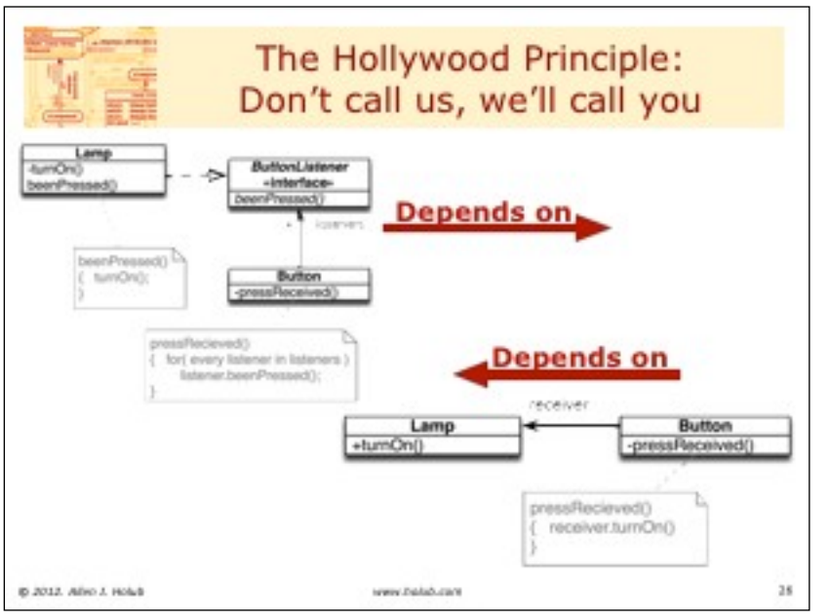
www.youtube.com/watch?v=-0Dmptetj1s



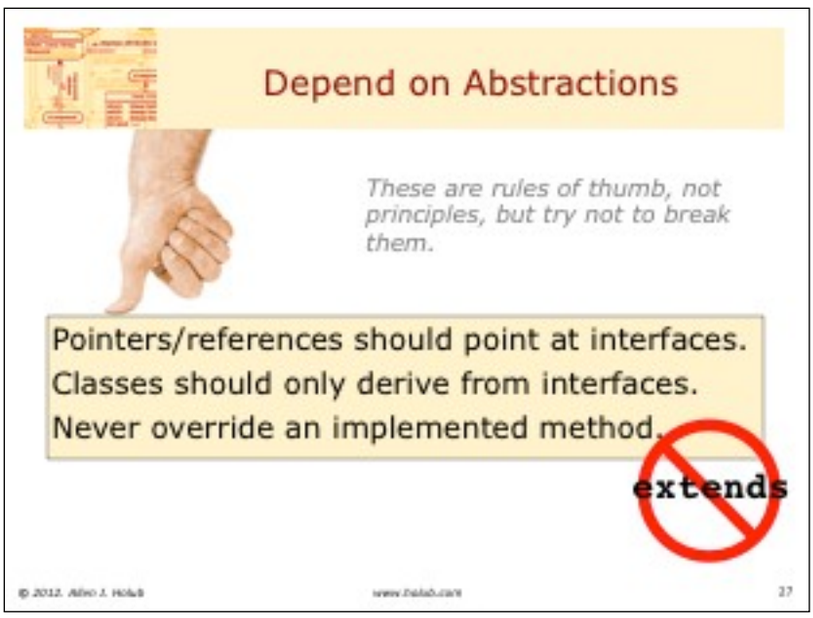
In the first (procedural) version, dependancies go down. If you change something in a lower layer, the higher layer has to change.
In the DIP version, messages flow in the same direction, but dependancies go up: if you change an interface, then the lower version has to change.



In the first (procedural) version, dependancies go down. If you change something in a lower layer, the higher layer has to change.
In the DIP version, messages flow in the same direction, but dependancies go up: if you change an interface, then the lower version has to change.

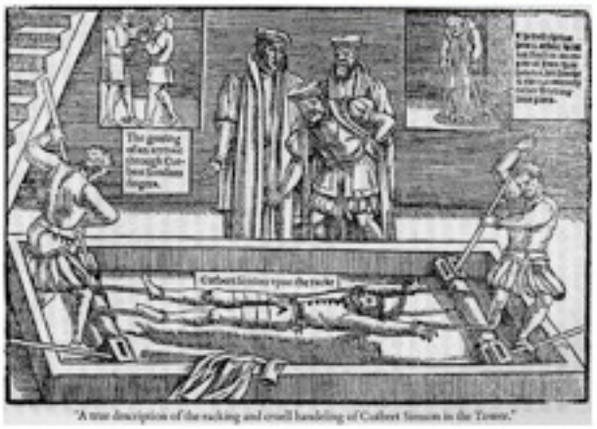


Notice how the dependency relationships are inverted between the first and second (listener) version.
Notice how the "policy" method (turnOn) went from public to private when when we introduced the interface.
This particular example is the "Observer" design pattern.



Rule 2 implies that you should NOT use "extends" unless there's no alternative.
The obvious exception to rule 3 is the Template Method pattern where the base-class method defines a reasonable default.
Otherwise, if you find yourself overriding an implemented method, that method should probably be defined in a common interface.
Various design patterns make it easier to follow these rules. For example, Abstract Factory lets you create an object that implements some interface without knowing the actual concrete class of the object.

Problems with Extends



"A true description of the racking and cruell handeling of Cuthert Simson in the Tower."

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() );
}
```


Behavior is everything!

IS-A

really means:

Behaves exactly like...

When designing, two classes are the same (or two objects are members of the same class) when they behave identically. That's the only meaningful criterion. Attributes are irrelevant.

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.

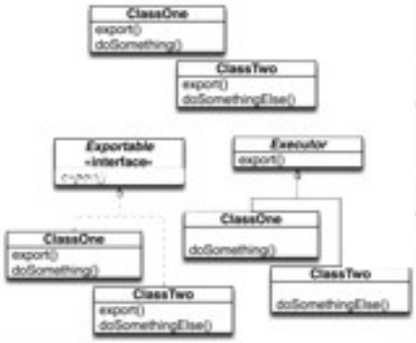
Additional responsibilities ⇒ implementation Inheritance.

Non-overlapping responsibilities ⇒ distinct classes

Identical responsibilities ⇒ identical classes

There are three possibilities, but only one implies inheritance.
The fact that Managers are Employees in real life is immaterial.
What we care about is the role that they take on in the context of the program's problem domain.

Class normalization



When two classes have a common subset of operations, and those operations are implemented identically, factor those operations into a shared superclass, otherwise they probably should be defined in an interface implemented by both classes.

Fragile base classes

- The main problem with implementation inheritance is "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.

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 );
    }

    public void push_many( Object[] articles )
    {   for( int i = 0; i < articles.length; ++i )
            push( articles[i] );
    }
}
```

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();
```

Pop Quiz

Which principle does Stack break?

Liskov Substitution

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

The stack now holds garbage.

How about using encapsulation?

```
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

```
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

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
```

Exactly like earlier Stack class

Fixed version

```
class MonitorableStack implements Stack
{  private SimpleStack stack = new SimpleStac
   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.

The Monitorable stack USES a a SimpleStack, it IS NOT a SimpleStack.

Delegation/Inheritance pattern

Rather than:

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

Use:

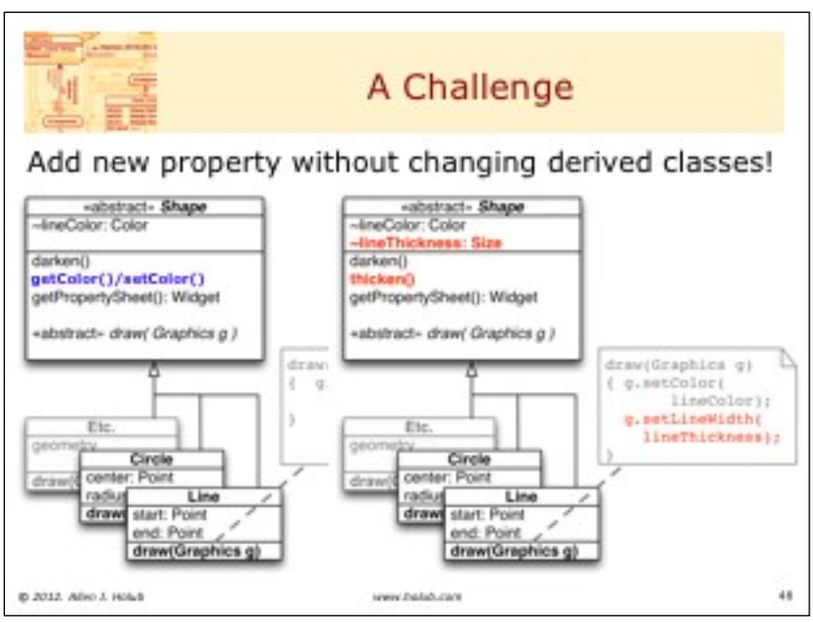
```
interface Simple
{   void f();
    static class Implementation implements Simple
    {   void f(){/*reasonable default implementation*/}
    }
}
class Specialization implements Simple
{   Simple delegate = new Simple.Implementation();
    void f(){ delegate.f(); }
}
```


*Create an interface, not a class.
*If you would normally inherit base-class methods, provide a default implementation of the interface that implements those methods that would have been implemented at the base-class level.
*Instead of extending a base class, implement the interface.
*For every interface method, delegate to a contained instance of the default implementation.

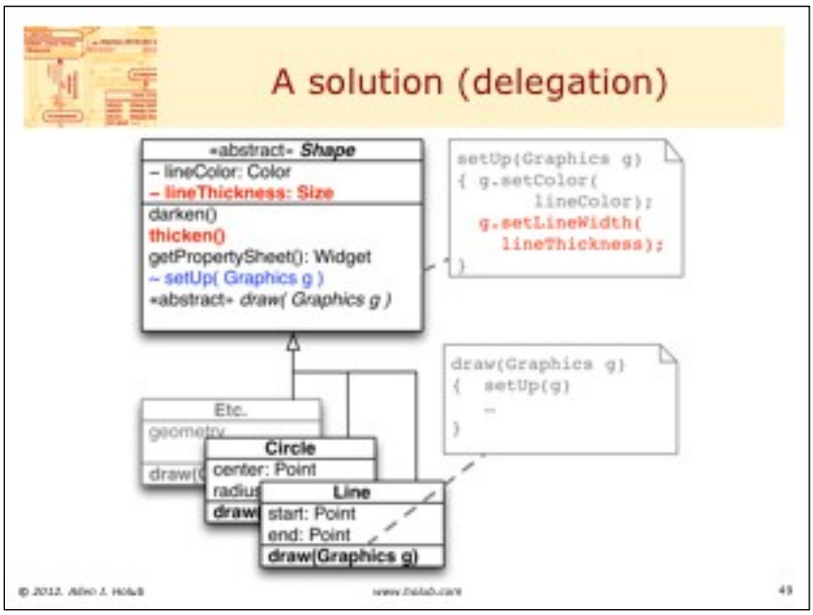
A few observations

- At any time in the future, anyone can add a method to a base class (e.g. **clear()**) that might break the derived class.
- Avoid "Framework" architectures. (in which you must use implementation inheritance to customize base-class behavior)
- Since you can implement as many interfaces as you like, you can use the "inheritance" pattern to implement multiple inheritance in Java.

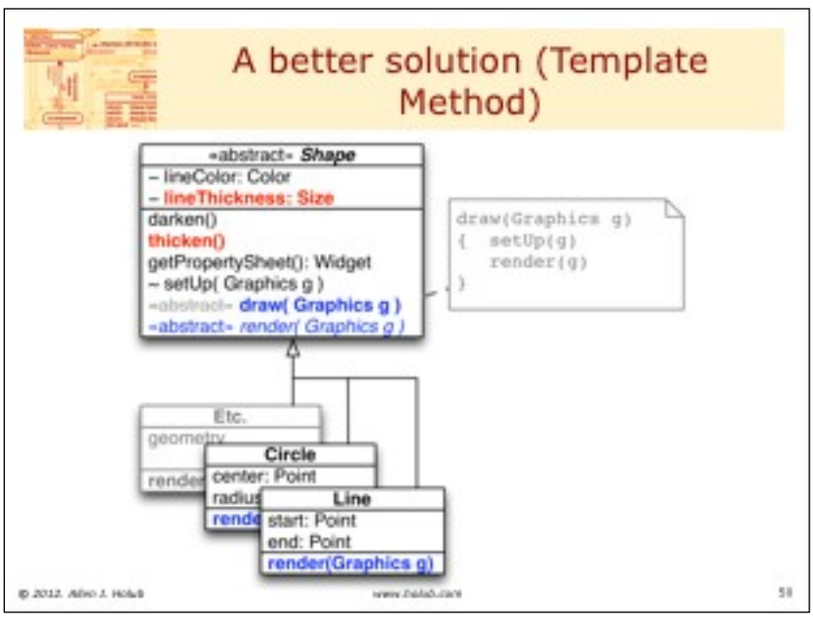
Protected fields/methods give you too much access to implementation. Use only for "Template Methods."
Avoid overriding base-class methods (unless you're implementing an interface).
Avoid virtual (Java: make as many as possible final) methods.
Don't leverage the base-class implementation.



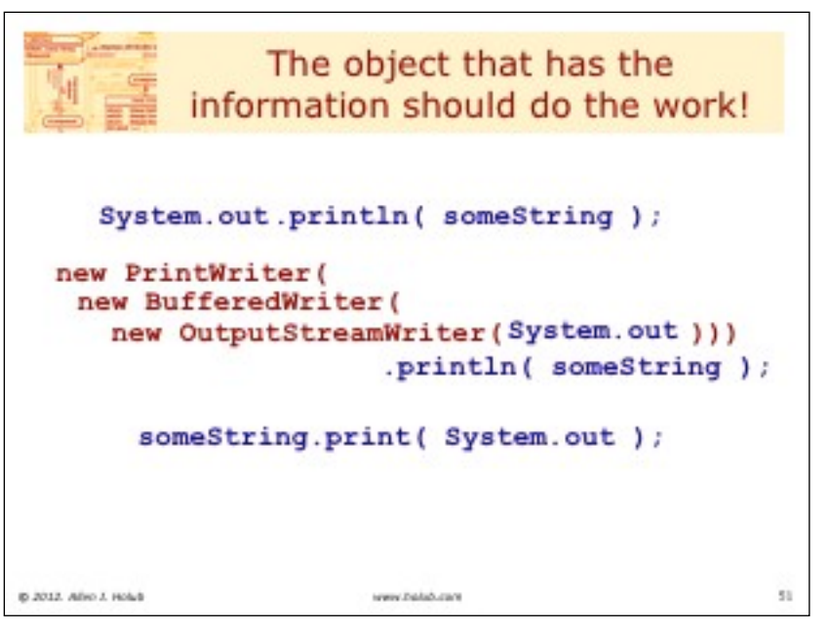
- get/set methods (or protected Color), are the problem: Adding thickness requires modification to every derived class (to getThickness()).
- You don't need a get/set Color if the object provides it's own property sheet.



Simple delegation (ask the object that has the information to do the work) solves the problem.
No getColor(). Color can be private!
BUT --- it's too easy to forget to call setUpGraphics().



The Template-Method design pattern is better.



The top line is Java 1.0
The problem is the getBytes() call in String.
Because the object-with-the-information (the string) should do the work (print itself) rule wasn't followed, the red version is needed to support unicode.
But, if the rule had been followed (3rd version), no changes would have been necessary at the client level.

Ask for help, not information.

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

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 );
```


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   ()            {...}
}
```


By not exposing customer balance, you don't need to do any coversions at all!
This philosophy apples all the way down.
Money doesn't expose a value either.
Adding currency to the money class (in blue) doesn't affect the outside world.
Other methods (addTo, printTo, etc.) work the same way.

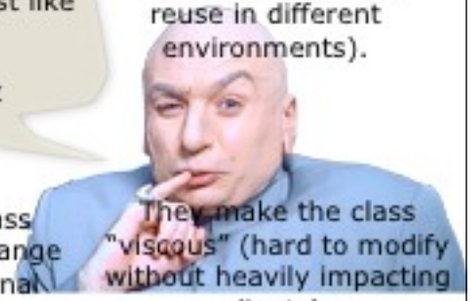
The obvious conclusion

Getters and Setters are evil!

They expose implementation (just like public fields)

They make the class "immobile" (hard to reuse in different environments).

They encourage indirect coupling (like global variables)



They make the class "rigid" (you can't change it because of external dependancies).

They make the class "viscous" (hard to modify without heavily impacting clients).

 www.holub.com 55

Procedural programmers sees nothing wrong, People blindly copy the idiom without considering the consequences.

Many books recommend putting mutators and accessors on all fields!

JavaBeans introduced the getter/setter "design pattern" because it was "easy." (There's a better alternative, called a BeanCustomizer, but nobody uses it. @annotations are better)

Protect yourself from changes to the Model!

- Every object builds it's own user interface.
- The larger UI is a composite.
  - individual objects construct subcomponents.
- When the class changes, the UI code is right there.
- Change the business object, the UI changes. Everywhere.

Ask: who created that part of the UI?

 www.holub.com 56

Compare this situation with a VB UI. When the model changes, you need to identify hundereds of places in the UI where the change impacts the screen, and fix each one separately.

But the business object needs to be decoupled from the user interface!

- Why?
  - Agile projects are not generic!
  - When the business object changes, the UI usually changes as well.
  - There's usually one best way to present an object, even if the object will be reused.
  - There are architectural solutions if you really need the flexibility.
  - Coupling with the OS is handled with an abstraction layer.

 www.holub.com 57

Compare this situation with a VB UI. When the model changes, you need to identify hundereds of places in the UI where the change impacts the screen, and fix each one separately.

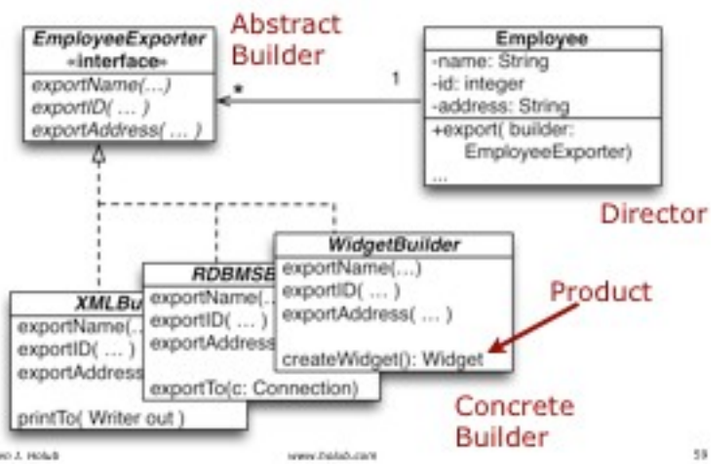
The notion of generic component architectures built on general business objects is discredited. See the IBM "San-Francisco" project.

Building for the general case is NOT agile. You can create a generic class by using an existing class in a new project and adding methods, etc., as needed, but that makes for a lot of bloat.

Bloat is bad.

"Draw yourself" isn't practical

The *Builder* design pattern



Building objects

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

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


All builders implement the same interface, so are interchangeable to an Employee. Construction isn't mentioned as an application of Builder in GoF "Design Patterns" book, but it seems reasonable to me.

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() );
```


All builders implement the same interface, so are interchangeable to an Employee.

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() );
    }
```

↓

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* (Swing UI)

```
class JComponentExporter implements Employee.Exporter
{ private String name, id, salary;

  public void addName  ( String name   ){ this.name = name;}
  public void addID    ( String id     ){ this.id = id;    }
  public void addSalary( String salary ){this.salary=salary;}

  JComponent getJComponent()
  {   JComponent panel = new JPanel();
      panel.setLayout( new GridLayout(3,2));
      panel.add( new JLabel("Name:  ") );
      panel.add( new JLabel( name ) );
      panel.add( new JLabel("Employee ID:"));
      panel.add( new JLabel( id ) );
      panel.add( new JLabel("Salary:"));
      panel.add( new JLabel( salary ));
      return panel;
  }
}
```

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;
    }
}
```

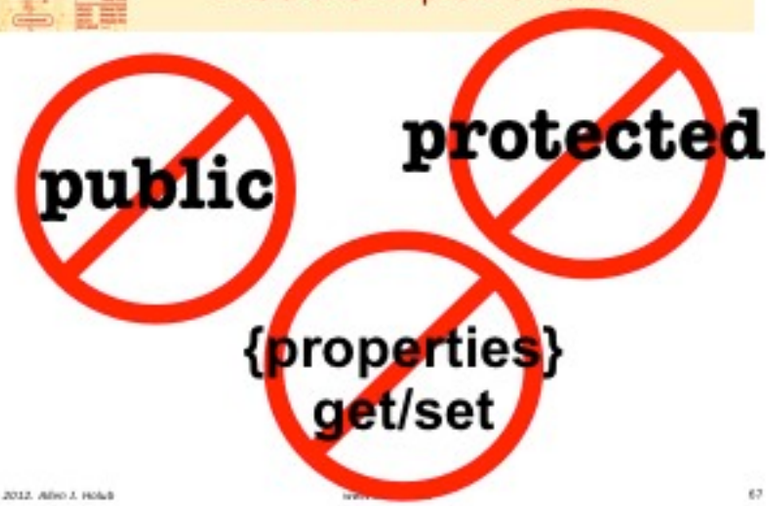
```
HTML Exporter e = new HtmlExporter;
someEmployee.export( e );
someStream.print( e.getHTML() );
```


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) );
```

Hide the implementation!



Domain-level methods are public. Nothing else!
Avoid accessors/mutators (C# properties).
Avoid protected (or package-level) access.

Design by Responsibility

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

Focus on *responsibilities*

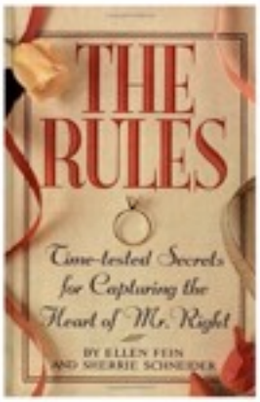
- **Ask the object that has the information to do the work for you**.
  - then you don't need to "get" anything.
- Develop code using Agile Object Modeling:
  - Develop from User Stories (Use-case scenarios).
  - Model the problem domain, not the computer.
  - Use a CRC-card approach (but not the cards)
  - Build around the messages, not the class structure.
- TAKE A CLASS!

References

- *These slides*
  - *http://www.holub.com/publications/notes_and_slides*
- *Why extends is evil: Improve your code by replacing concrete base classes with interfaces*
  - http://www.javaworld.com/javaworld/jw-08-2003/jw-0801-toolbox.html
- *Why getter and setter methods are evil: Make your code more maintainable by avoiding accessors*
  - http://www.javaworld.com/javaworld/jw-09-2003/jw-0905-toolbox.html
- *More on getters and setters: Build user interfaces without getters and setters*
  - http://www.javaworld.com/javaworld/jw-01-2004/jw-0102-toolbox.html

So, what's a mother to do?

- **Design is a series of trade-offs.**
- The get/set idiom has a significant cost.
- But if you must, you must.
  - Return interfaces if possible.
  - May be required for generic code, UI Toolkits, OS-level wrappers, etc. (anywhere where you don't know how an object is used).
  - If you don't know how an object is used, you're usually doing something wrong!



What's this all mean?

There is no such thing as perfect

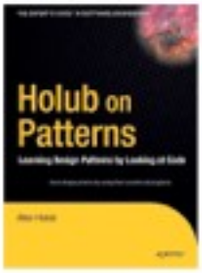
- Design is a series of trade-offs.
- Assess risk, then make reasonable decisions.
  - If you use implementation inheritance, then you run the risk of a fragile-base-class related bug.
  - If you expose implementation (with getters and setters) then you run the risk of a change to the exposing class rippling out to the entire program, with concomitant maintenance headaches.
  - ***That might be okay. Use your brain!***

There's often a better solution

- Approach popular libraries with skepticism
  - Use them, but don't hold them out as a model of good design.
- There's almost always a way to do it "right."
  - Move the work into the class that has the information needed to do the work.
  - Replace implementation inheritance with interface inheritance.
- You will learn to think in an OO way with enough practice.
- Study design.

Shameless self-promotion

The first couple chapters discuss these issues in depth.



?

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