# TDD, Refactoring and **Dependency Injection**: Agile's answer to "Big Up-Front Architecture" (BUFA)

*May 5th, 2006, Agile India Conference*

*Paul Hammant,*
*ThoughtWorks, Inc*
*www.paulhammant.com*

# The problem?

Previous-era architects and stake-holders suggest that the only way to write software was after an exhaustive period of design.
Latterly known as -
"big up front architecture".

Craig Larman's keynote cited plenty of evidence for this amongst other things being both problematic and perpetuated as a fact even today.

# Embrace Change?

Agile suggests embracing change is the key to success ..

.. yet how do we convince process and control-drunk stake-holders that we are adept at embracing design change while building complex applications without a detailed prescriptive architecture?

# Agile has always had some answers

- Test Driven Development (TDD).
- Refactoring makes design changes cheap
- Continuous Integration Testing (CIT)

(cheekily ignoring the other agile practices)

We're going to introduce each briefly - they are really worth presentations in their own right

# Test Driven Development

- You write the unit-test **BEFORE** you write the implementation code - no exceptions.
- As a practice it helps drive design
- Many other methodologies cherry-pick from Agile .... but never TDD
- Also read up on Behaviour Driven Development

MathTestCase.java

```
import junit.framework.TestCase;

public class MathTestCase extends TestCase {

    public void testAddition() {
        long result = Math.round(1.1);
        assertEquals(1L,result);
    }
    public void testAddition2() {
        long result = Math.round(1.6);
        assertEquals(1L,result);
    }

}
```

} bad test method naming by the way

Run – MathTestCase

Done: 2 of 2 Failed: 1(0.07 s)

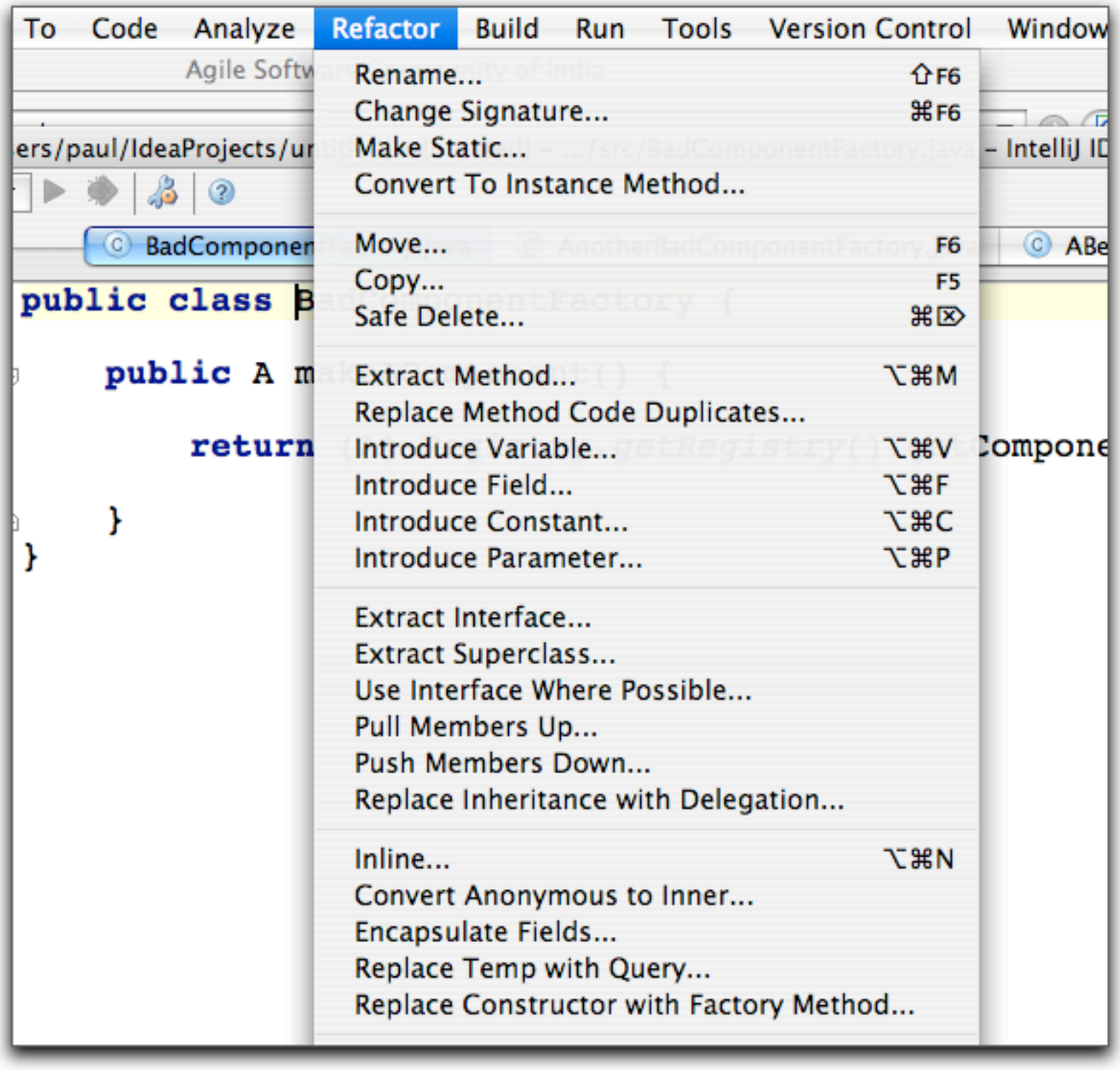
- MathTestCase
  - testAddition (MathTestCase)
  - testAddition2 (MathTestCase)

Output | Statistics

```
/System/Library/Frameworks/JavaVM.framew

junit.framework.AssertionFailedError: ex
    at MathTestCase.testAddition2(MathTe
    at sun.reflect.NativeMethodAccessorI
    at sun.reflect.NativeMethodAccessorI
    at sun.reflect.DelegatingMethodAcces
    at com.intellij.rt.execution.junit2.


Process finished with exit code 255
```

MathTestCase

# Refactoring




- smart functions in IDE for change lots of code at once.
- guaranteed to be error free
- for Java and C# .....
- ... else do it by hand :-(

# Continuous Integration Testing (CIT)

- Ensures changes 'here' don't undo something 'there'
- Provides early warning system for any build issue as well as history
- Creates team excitement about *working* builds
- website makes project status or progress visible to all

ThoughtWorks open sourced a tool called CruiseControl some years ago

# Dependency Injection: Components evolving or emerging over time...

# 5 second introduction to Dependency Injection

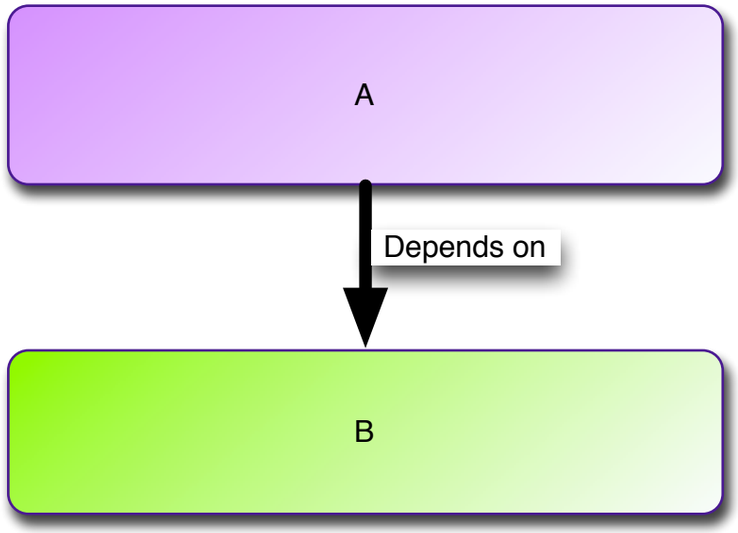
```java
public class A {

    private final B b;

    public A(B b) {
        this.b = b;
    }

    // other methods

}
```

```java
public class B {

    // methods ....

}
```

imagine components ‘A’ and ‘B’. A depends on B and declares B in its constructor to make that declarative and thus clear

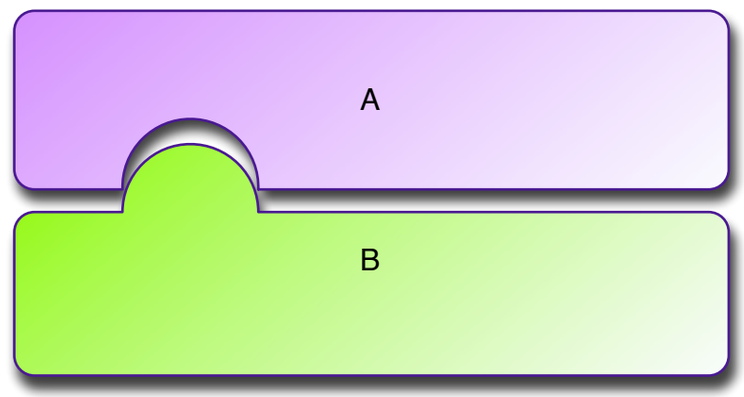
**simple stuff!**

# First, we're going to talk about component dependencies

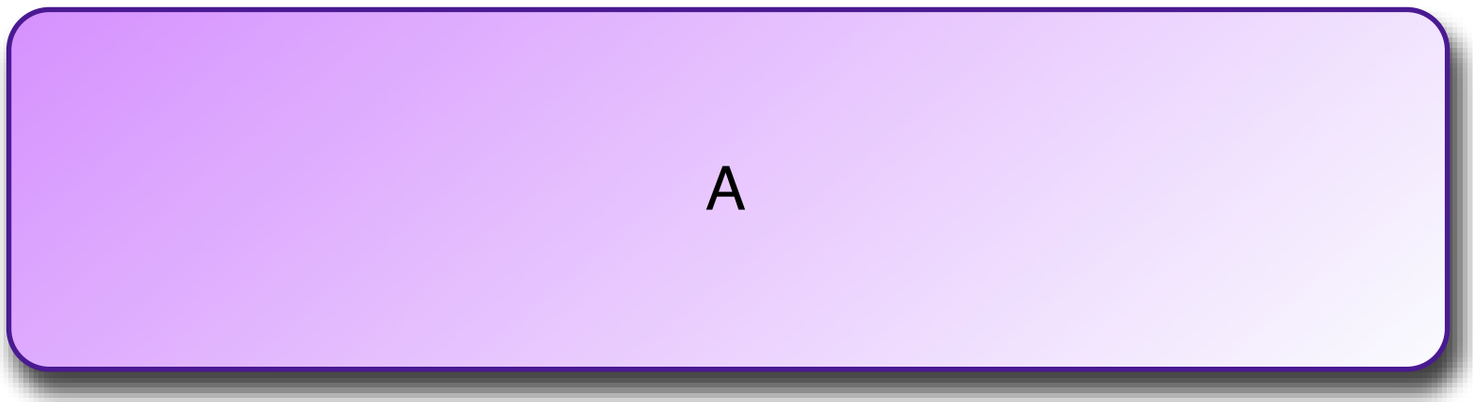



Rather than use arrows (which could be confusing if there are lots of them criss-crossing) ...

... we're going to use shapes. In this case the semi-circle implies a component need in A, that B can provide.



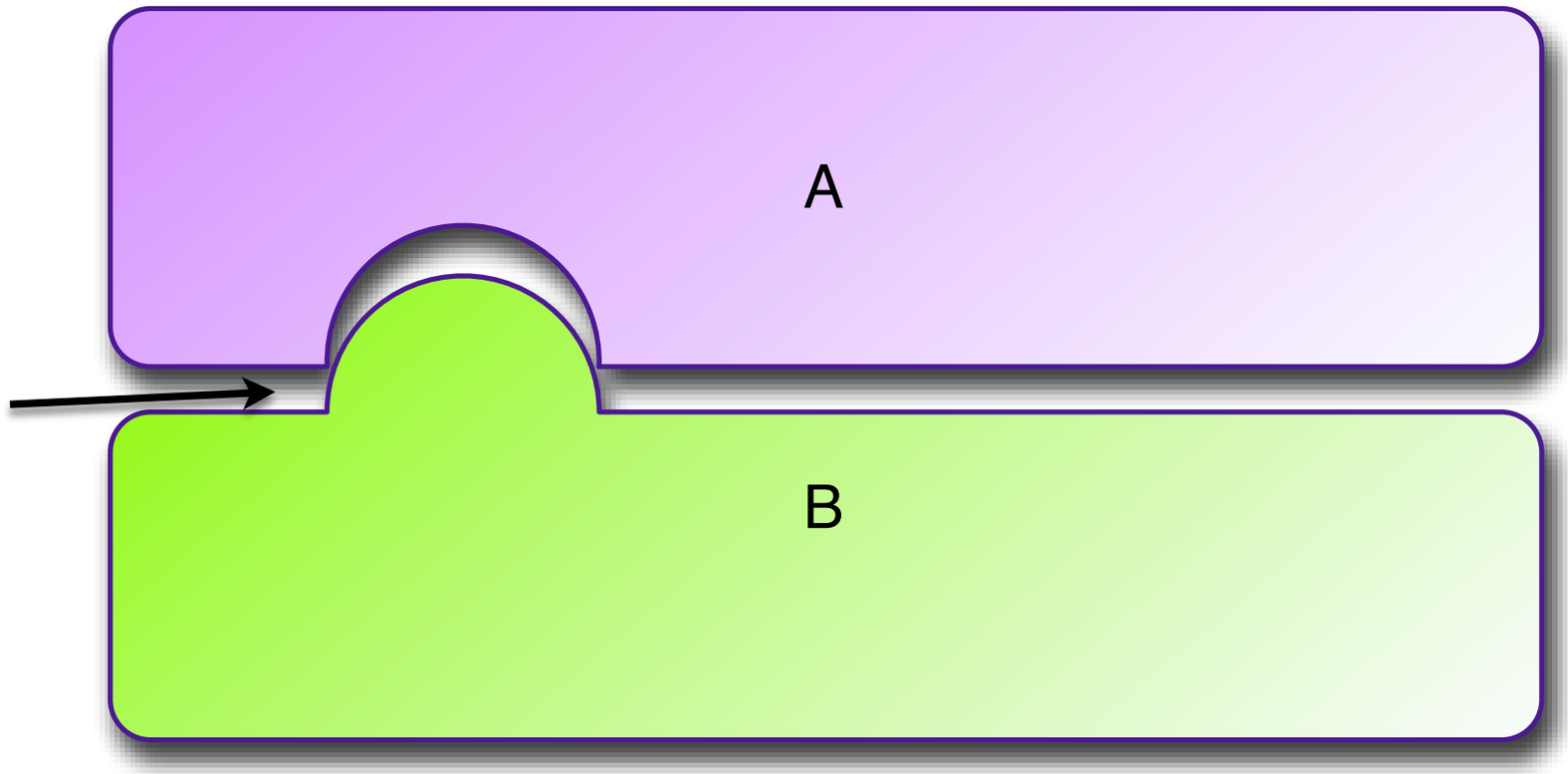
# presenting component A




But wait - we notice after writing it that there is
lots of *other* functionality inside it.
Maybe, we feel, too much.
Maybe we should separate into two components ....

# we split code between A & B

A is dependent on B remember?




Chg #1

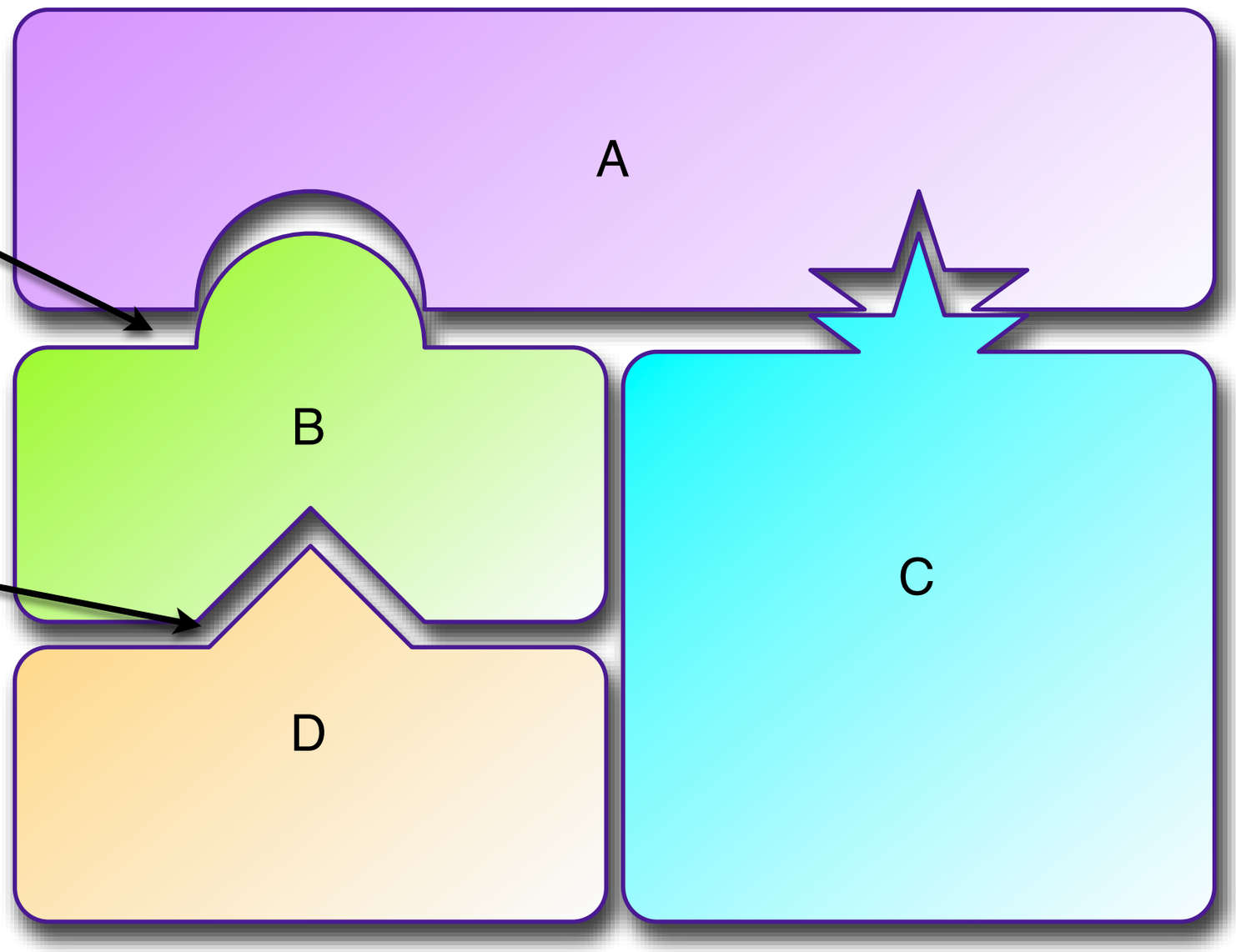
Still it does not feel right
A feels good, but B is too fat.
It's doing too many things ....

# Comps C & D Introduced

A is dependent on B

B is dependent on D

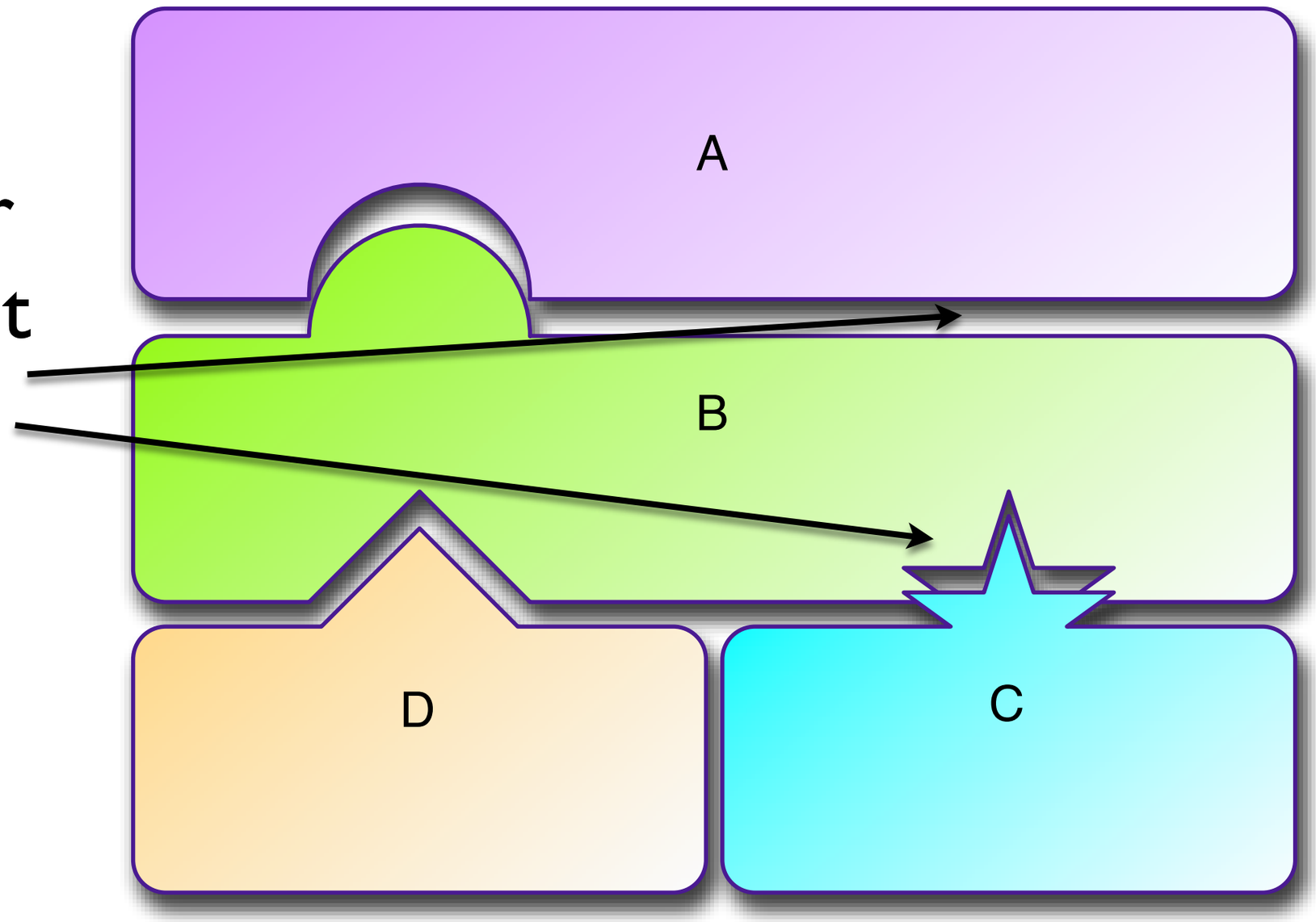
Chg #2




But later we have small rethink ....

# responsibilities rethought

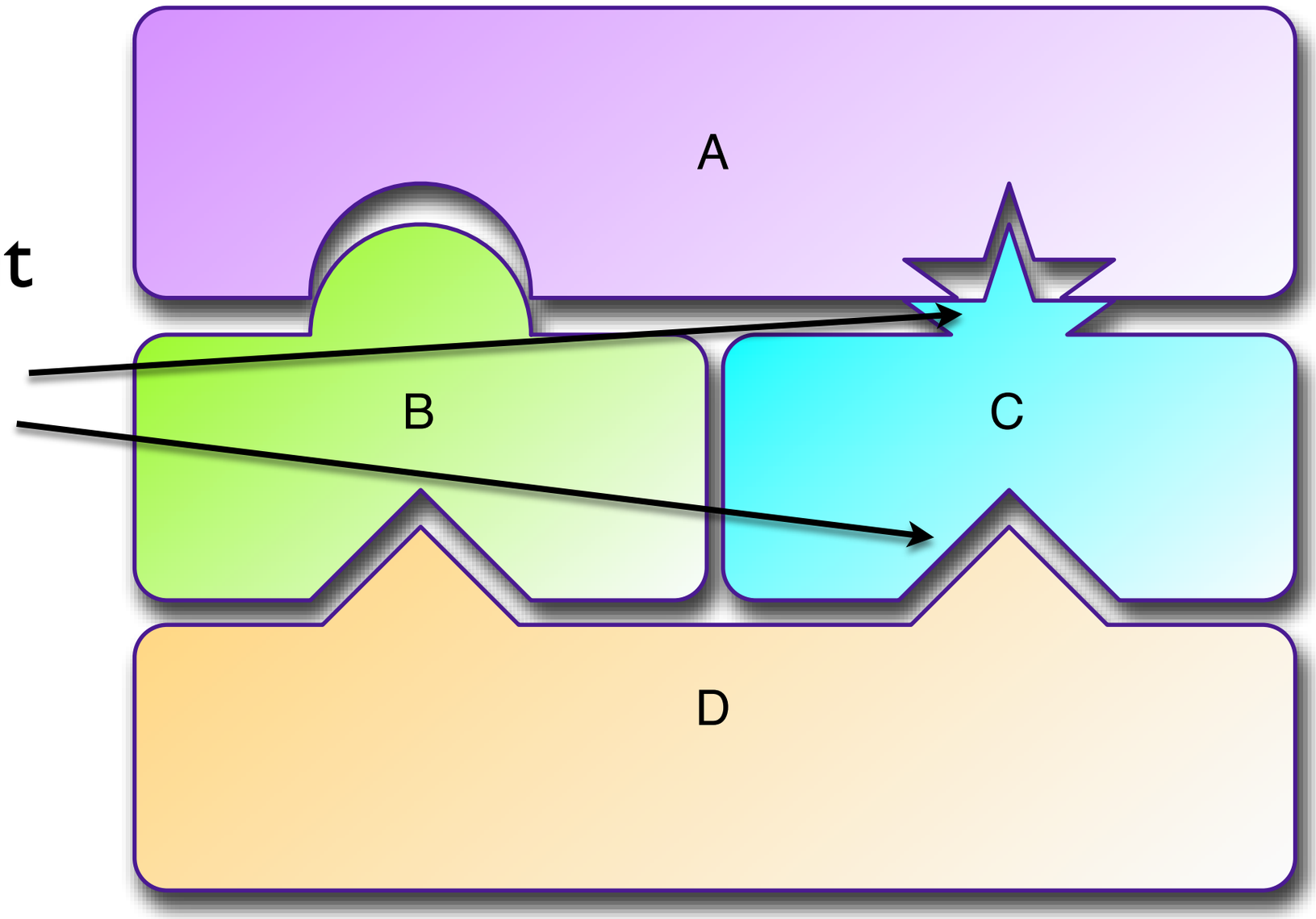
A is
no longer
dependent
on C,
B is




Chg
#3

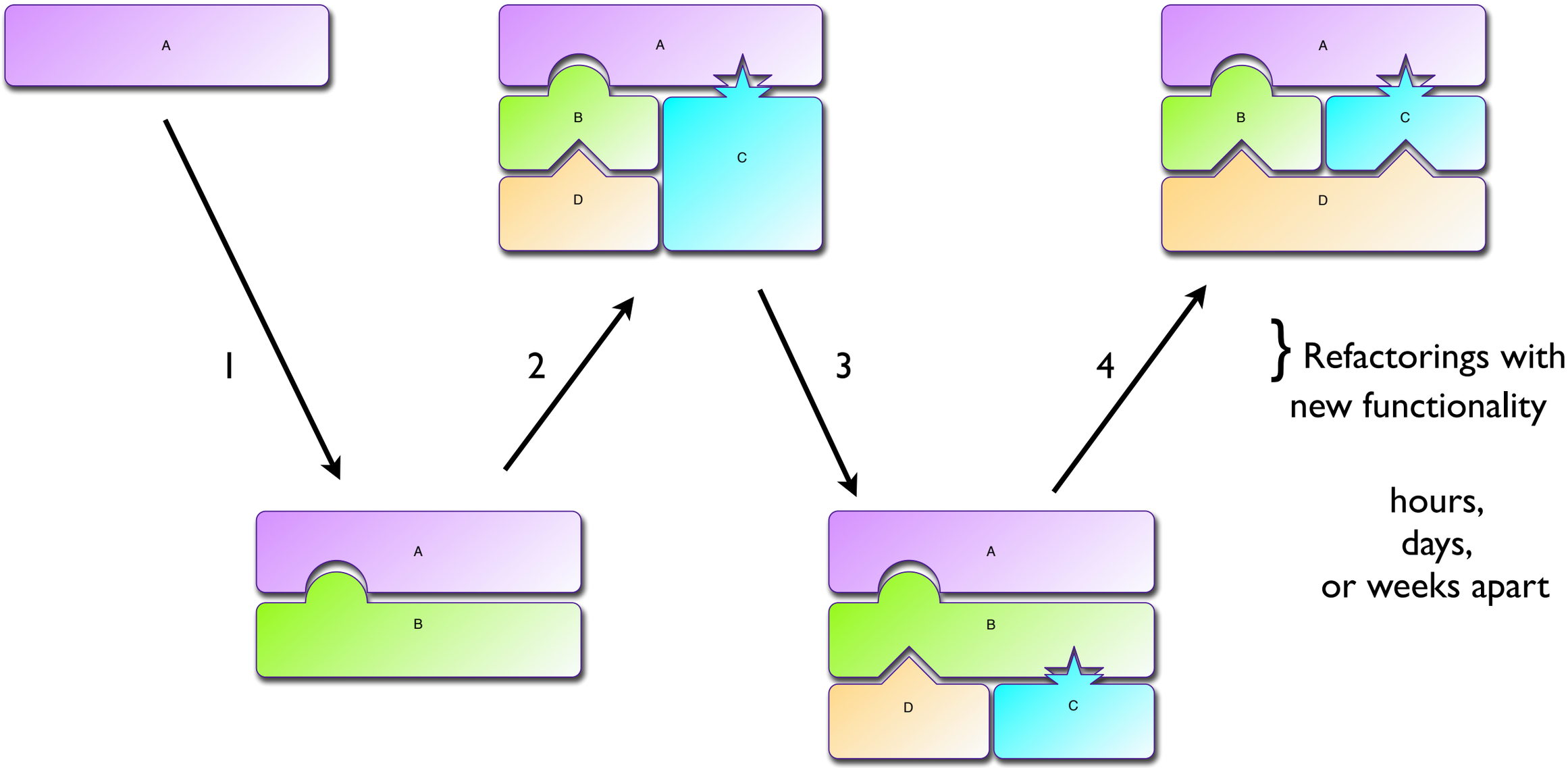
# then perhaps once more

A is again dependent on C, and C on D



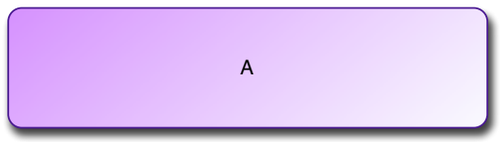

Chg #4

# the evolution recapped

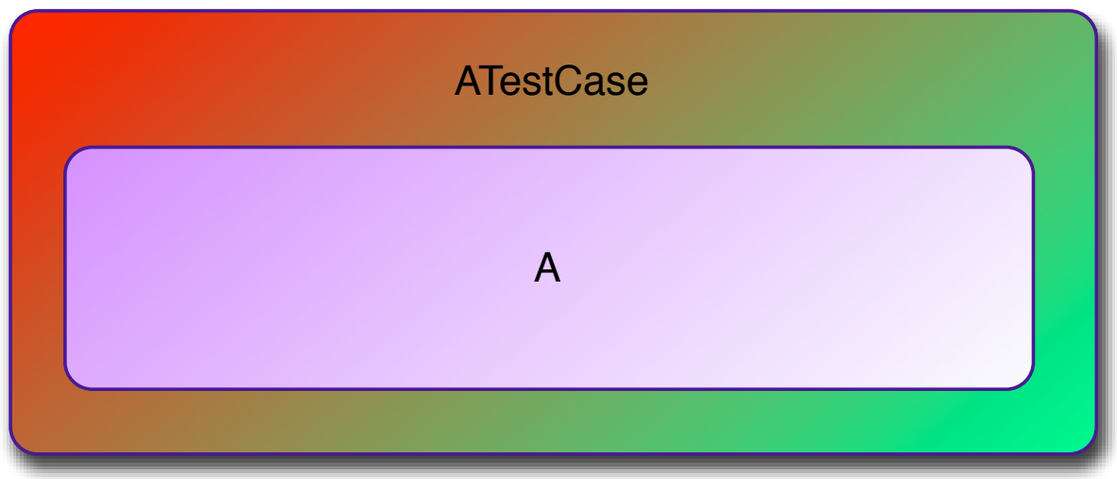


Was that the right way to illustrate an Agile component evolution?

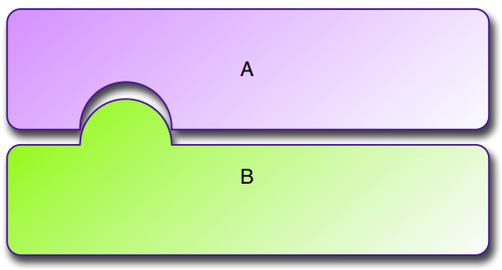
# No, we would have done it TDD



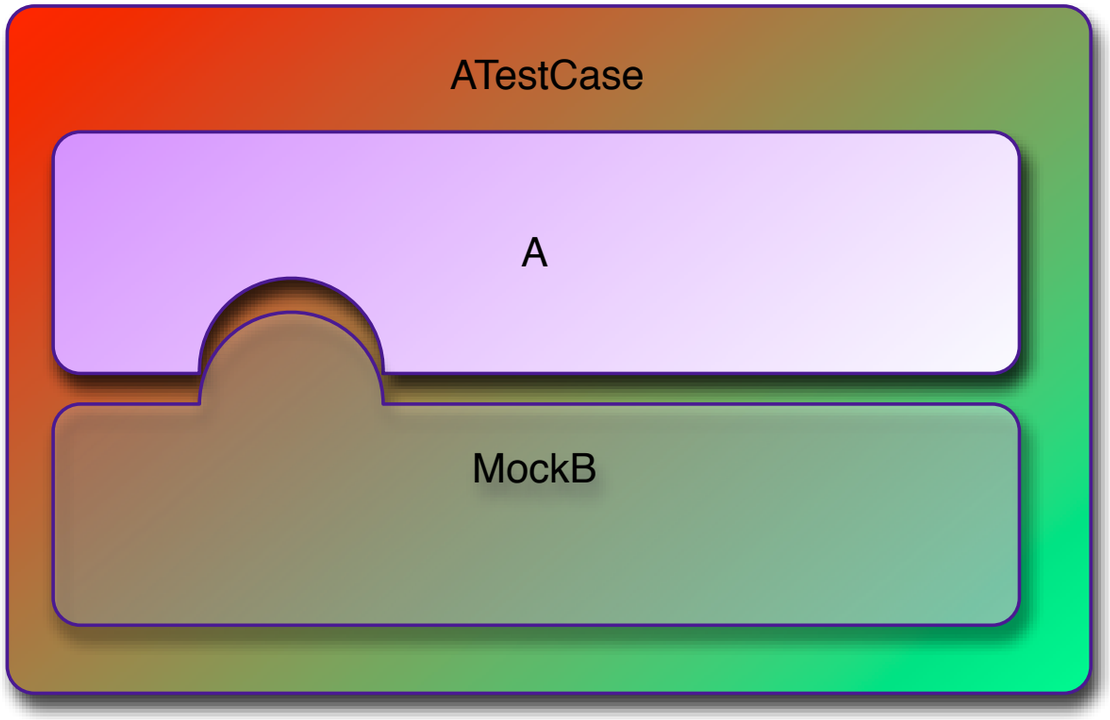

- is actually made like so ....

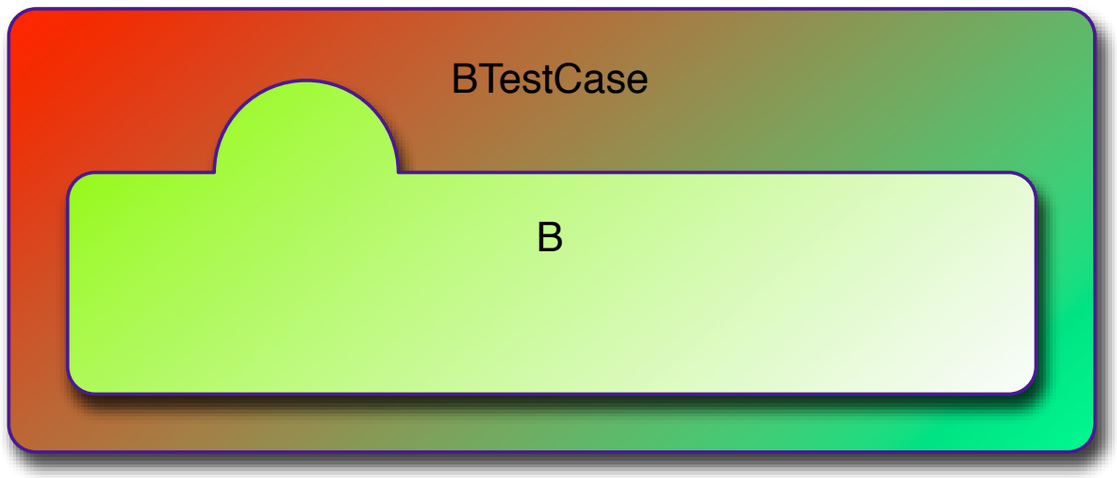


# unit-test code is refactored too ...

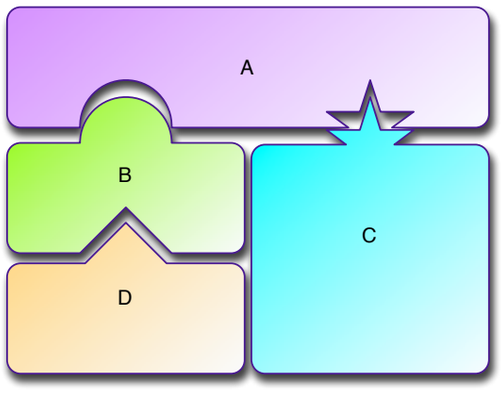



- is actually made like so ....

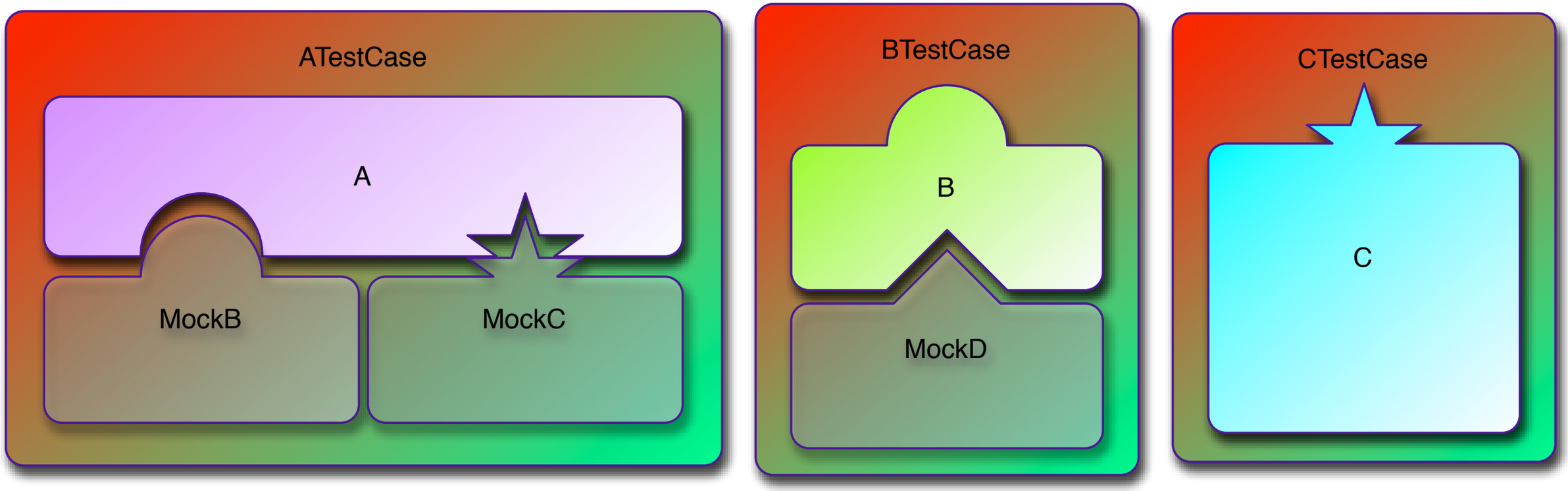


# and so on ...




- is actually made
  like so ....



# Mocking

- Read up on JMock for Java
- And NMock for C#
- Mocking (or stubbing) helps drive design
- Also see RhinoMocks for .Net and EasyMock for both as tools that are alternatives

# Things to remember when making components

# #1: You can over-use containers/frameworks...

# Consider just A and B

```java
public class A {

    private final B b;

    public A(B b) {
        this.b = b;
    }

    // other methods

}
```

```java
public class B {

    // methods ....

}
```

yeah, they are a bit light, but they are representative of bigger components

# bad: singleton registry

```
public class BadComponentFactory {

    public A makeAComponent() {

        return (A) Registry.getRegistry().getComponent(A.class);

    }
}
```

oops - ‘component registry can be synonymous with component container or framework’

# bad: same thing, but strongly typed

```
public class AnotherBadComponentFactory {

    public A makeAComponent() {

        return Registry.getRegistry().getAComponent(A.class);

    }
}
```

# good: passing a reference

```
public class ABetterComponentFactory {

    private final Registry registry;

    public ABetterComponentFactory(Registry registry) {
        this.registry = registry;
    }

    public A makeAComponent() {

        return registry.getAComponent(A.class);

    }
}
```

# good: short lifetime container/components

```
public class MaybeBetterStillComponentFactory {

    private final Registry registry;

    public MaybeBetterStillComponentFactory(Registry registry) {
        this.registry = registry;
    }

    public A makeAComponent() {

        // 'A' may, unlike 'B', have a short lifetime.
        Registry transientReg = new Registry(registry);
        transientReg.register(A.class);
        return transientReg.getAComponent(A.class);

    }
}
```

# good: do you need a registry at all?

```
public class GoodComponentFactory {

    private final A a;

    public GoodComponentFactory(A a) {
        this.a = a;
    }

    public A makeAComponent() {

        return a;

    }
}
```

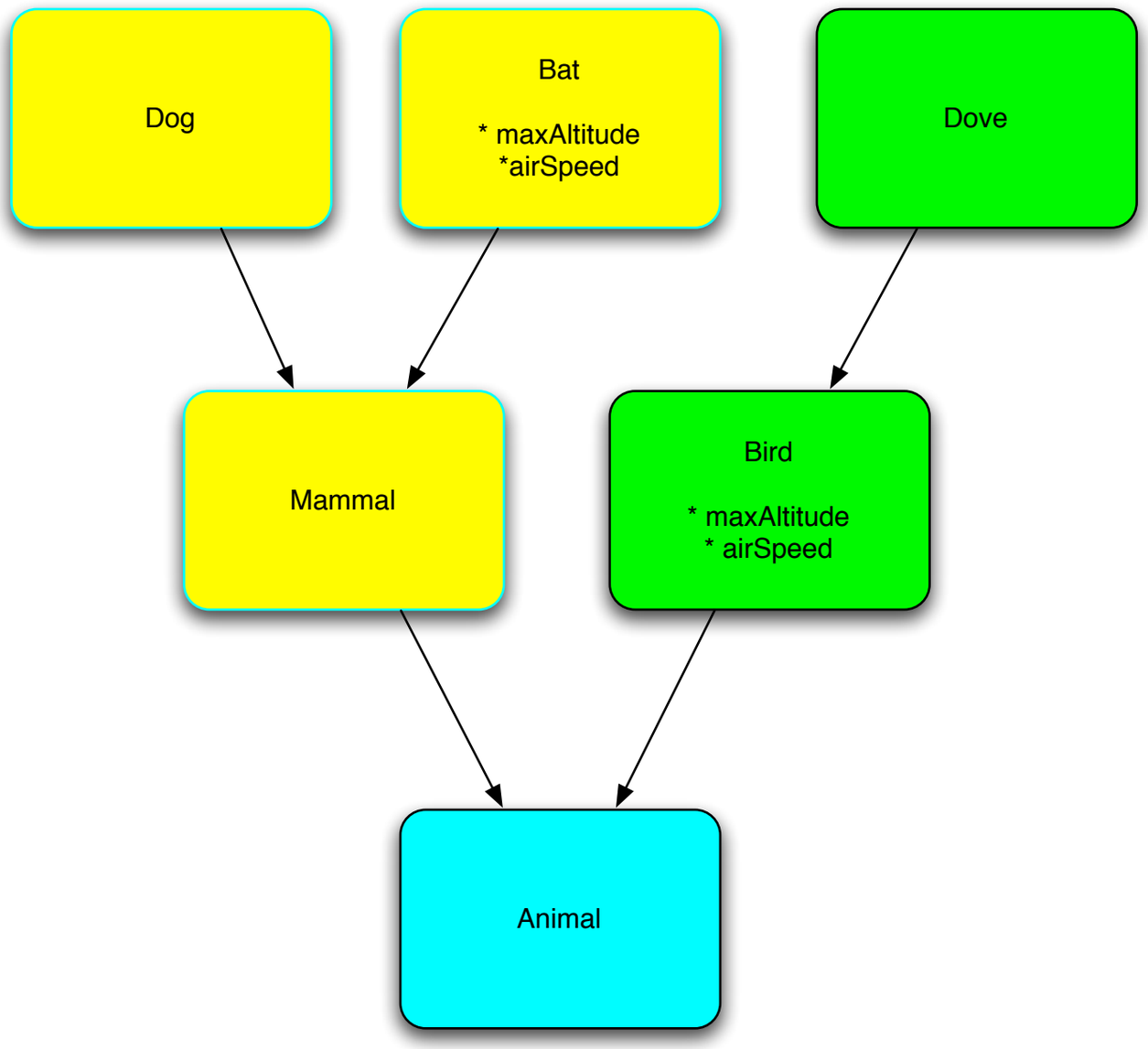
# good: without registry, with short lifetime

```
public class AnotherGoodComponentFactory {

    private final B b;

    public AnotherGoodComponentFactory(B b) {
        this.b = b;
    }

    public A makeAComponent() {

        // 'A' may, unlike 'B', have a short lifetime.
        return new A(b);

    }
}
```

# #1 recap: if you can exist without a registry/container/framework do so

# #2: Composition is much better than Inheritance...

# inheritance models are not perfect




Bat cannot leverage Bird functionality

# composition models have less emergent limitations

- Bat **has a** FlightCapability
- Dove **has a** FlightCapability

.... is better than .....

- Bad **is a** FlyingCreature
- Dove **is a** FlyingCreature

# #3: Interface/ Implementation separation

# A poor example..

```java
public interface FlightCapable {

    int getMaxAltitude();
    int getTopSpeed();
    int getRange();

}
```

```java
public class FruitBat implement

    FlightCapable flightCapable

    public FruitBat(FlightCapab
        this.flightCapable = fl
    }

    public int getMaxAltitude()
        return flightCapable.ge
    }

    public int getTopSpeed() {
        return flightCapable.ge
    }

    public int getRange() {
        return flightCapable.getRange();
    }

    // more methods .....

}
```

```java
public class FlyingMammal implements FlightCapable {

    private int maxAltitude, topSpeed, range;

    public FlyingMammal(int maxAltitude, int topSpeed, int range) {
        this.maxAltitude = maxAltitude;
        this.topSpeed = topSpeed;
        this.range = range;
    }

    public int getMaxAltitude() {
        return maxAltitude;
    }

    public int getTopSpeed() {
        return topSpeed;
    }

    public int getRange() {
        return range;
    }
}
```

# Terms to search for in Google India

PicoContainer, The Spring Framework, Dependency Injection, Lightweight Components, EJB 3.0, Domain Driven Design, JBehave, JUnit, JMock, NMock, EasyMock, Rhino.mocks, Continuous Integration Testing, CruiseControl, Design Patterns, Refactoring

# #4: Avoid meta-data (XML etc) wherever you can where it encodes functionality

Thanks to Ward Cunningham for the idea:

"Dependency Injection
is a key element
of agile architecture*"

* a second hand and paraphrased from his
'Agile vs Traditional panel'
at a Testing conference a year or so ago.

# Thanks for coming!

# Questions?