**Double Dispatch is a Code Smell**

<https://lostechies.com/derekgreer/2010/04/19/double-dispatch-is-a-code-smell/>

If you’re using [Double Dispatch](http://en.wikipedia.org/wiki/Double_dispatch) in your code, this may be a symptom of an underlying design issue which may impact the maintainability of your application.  Due to the fact that Double Dispatch is at times confused with a form of the [Strategy Pattern](http://en.wikipedia.org/wiki/Strategy_Pattern), an overview may be in order to elaborate on this assertion further.

**What is Double Dispatch?**

Technically, Double Dispatch refers to a technique used in the context of a polymorphic method call for mitigating the lack of [multimethod](http://en.wikipedia.org/wiki/Multimethods) support in programming languages.  More simply, Double Dispatch is used to invoke an overloaded method where the parameters vary among an inheritance hierarchy.  To explain fully, let’s start with a review of [polymorphism](http://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming).

**Polymorphism**

In the following example, a hierarchy of shapes are defined with each of the derived types overloading a base virtual Draw() method.  Next, a console application is used to define a list of each of the shapes and iterate over each shape in the collection calling the Draw() method of each item in the list:

import java.util.ArrayList;  
import java.util.List;  
class Shape  
 {  
 public void Draw()  
 {  
 System.*out*.println("A shape is drawn.");  
 }  
 }  
  
 class Polygon extends Shape  
 {  
 public void Draw()  
 {  
 System.*out*.println("A polygon is drawn.");  
 }  
 }  
  
 class Quadrilateral extends Polygon  
 {  
 public void Draw()  
 {  
 System.*out*.println("A quadrilateral is drawn.");  
 }  
 }  
  
 class Parallelogram extends Quadrilateral  
 {  
 public void Draw()  
 {  
 System.*out*.println("A parallelogram is drawn.");  
 }  
 }  
  
 class Rectangle extends Parallelogram  
 {  
 public void Draw()  
 {  
 System.*out*.println("A rectangle is drawn.");  
 }  
 }  
  
 class Program  
 {  
 public static void main(String[] args)  
 {  
 List<Shape> shapes = new ArrayList<Shape>();  
 shapes.add( new Shape() );  
 shapes.add( new Polygon() );  
 shapes.add( new Quadrilateral() );  
 shapes.add( new Parallelogram());  
 shapes.add( new Rectangle());  
  
  
 for(Shape shape : shapes)  
 {  
 shape.Draw();  
 }  
 System.*out*.println("................................");  
 }  
 }

The following lines are printed to the console upon running the application:

A shape is drawn.

A polygon is drawn.

A quadrilateral is drawn.

A parallelogram is drawn.

A rectangle is drawn.

Note that the proper Draw() method is called for each item in the collection.  In most object-oriented languages, this polymorphic behavior is achieved through the use of a [virtual table](http://en.wikipedia.org/wiki/Virtual_table) consulted at run-time to derive the proper offset address for an object’s method.  This behavior is referred to as "[Dynamic Dispatch](http://en.wikipedia.org/wiki/Dynamic_dispatch)" or "Single Dispatch".  So, how does this relate to Double Dispatch?  To answer this question, let’s next review [method overloading](http://en.wikipedia.org/wiki/Function_overloading).

**Method Overloading**

In the following example, our Shape class is redefined to have two overloaded Drawmethods: one with a parameter of type Surface and one with a parameter of typeEtchASketch:  
class Surface {  
}  
  
class EtchASketch extends Surface {  
}  
  
class Shape {  
 public void Draw(Surface surface) {  
 System.*out*.println("A shape is drawn on the surface with ink.");  
 }  
  
 public void Draw(EtchASketch etchASketch) {  
 System.*out*.println("The knobs are moved in attempt to draw the shape.");  
 }  
}  
  
class Program {  
 public static void main(String[] args) {  
 Shape shape = new Shape();  
 shape.Draw(new Surface());  
 shape.Draw(new EtchASketch());  
 }  
}

When executed, the following lines are printed to the console:

A shape is drawn on the surface with ink.

The knobs are moved in attempt to draw the shape.

Note that the parameter type determines which Draw() method is invoked.

But what happens if we change the Main() method to the following?

class Program {  
 public static void main(String[] args) {  
 Shape shape = new Shape();  
 Surface surface = new Surface();  
 Surface etchASketch = new EtchASketch();  
  
 shape.Draw(surface);  
 shape.Draw(etchASketch);  
 }  
}

Executing this produces the following:

A shape is drawn on the surface with ink.

A shape is drawn on the surface with ink.

What happened?  The issue here is that the method to call was determined statically at compile time based upon the reference type, not at run-time based upon the object type.  To resolve this issue, another technique is needed … Polymorphic Static Binding.

**Polymorphic Static Binding**

Polymorphic static binding is a technique where static method invocations are determined at run-time through the use of polymorphism.  This can be demonstrated in our example by adding a new Draw(Shape shape) method to theSurface and EtchASketch types which call shape.Draw() with a reference to the current object:

class Surface {  
  
 public void Draw(Shape shape) {  
 shape.Draw(this);  
 }  
}  
  
class EtchASketch extends Surface {  
  
 public void Draw(Shape shape) {  
 shape.Draw(this);  
 }  
}

To invoke the correct Shape.Draw() method, our console application needs to be modified to call the the method indirectly through a Surface reference:

class Program {  
 public static void main(String[] args) {  
 Shape shape = new Shape();  
 Surface surface = new Surface();  
 Surface etchASketch = new EtchASketch();  
  
 surface.Draw(shape);  
 etchASketch.Draw(shape);  
 }  
}

For double dispatch example for my understanding, the complete code is given below.

package com.ddlab.rnd.type2;  
  
class Surface {  
  
 public void Draw(Shape shape) {  
 shape.Draw(this);  
 }  
}  
  
class EtchASketch extends Surface {  
  
 public void Draw(Shape shape) {  
 shape.Draw(this);  
 }  
}  
  
class Shape {  
  
 public void Draw(Surface surface) {  
 System.*out*.println("A shape is drawn on the surface with ink.");  
 }  
  
 public void Draw(EtchASketch etchASketch) {  
 System.*out*.println("The knobs are moved in attempt to draw the shape.");  
 }  
}  
  
class Program {  
 public static void main(String[] args) {  
 Shape shape = new Shape();  
 Surface surface = new Surface();  
 Surface etchASketch = new EtchASketch();  
  
 surface.Draw(shape);  
 etchASketch.Draw(shape);  
 }  
}

Upon executing the application again, the following lines are now printed:

A shape is drawn on the surface with ink.

The knobs are moved in attempt to draw the shape.

This example achieves the desired result by effectively wrapping the static-dispatched method invocation (i.e. Shape.Draw()) within a virtual-dispatch method invocation (i.e. Surface.Draw() and EtchASketch.Draw()).  This causes the staticShape.Draw() method invocation to be determined by which virtual Surface.Draw()method invocation is executed.

Although the above example now contains a method invocation using a reference to the current object as the method parameter (often seen with Double Dispatch), it should be noted that Double Dispatch has yet to be demonstrated.  Thus far, only one level of virtual dispatching has been used.  To demonstrate Double Dispatch, the techniques from both the polymorphism example and the polymorphic static binding example need to be combined as seen in the next section.

**Double Dispatch**

The following example contains a hierarchy of Surface types and a hierarchy of Shapetypes.  Each Shape type contains an overloaded virtual Draw() method which contains the logic for how the shape is to be drawn on a particular surface.  The example console application uses the polymorphic static binding technique to ensure the proper overload is called for each surface type:

import java.util.ArrayList;  
import java.util.List;  
class Surface {  
 public void Draw(Shape shape) {  
 shape.Draw(this);  
 }  
}  
  
class EtchASketch extends Surface {  
 public void Draw(Shape shape) {  
 shape.Draw(this);  
 }  
}  
  
class Shape {  
 public void Draw(Surface surface) {  
 System.*out*.println("A shape is drawn on the surface with ink.");  
 }  
  
 public void Draw(EtchASketch etchASketch) {  
 System.*out*.println("The knobs are moved in attempt to draw the shape.");  
 }  
}  
  
class Polygon extends Shape {  
 public void Draw(Surface surface) {  
 System.*out*.println("A polygon is drawn on the surface with ink.");  
 }  
  
 public void Draw(EtchASketch etchASketch) {  
 System.*out*.println("The knobs are moved in attempt to draw the polygon.");  
 }  
}  
  
class Quadrilateral extends Polygon {  
 public void Draw(Surface surface) {  
 System.*out*.println("A quadrilateral is drawn on the surface with ink.");  
 }  
  
 public void Draw(EtchASketch etchASketch) {  
 System.*out*.println("The knobs are moved in attempt to draw the quadrilateral.");  
 }  
}  
  
class Parallelogram extends Quadrilateral {  
 public void Draw(Surface surface) {  
 System.*out*.println("A parallelogram is drawn on the surface with ink.");  
 }  
  
 public void Draw(EtchASketch etchASketch) {  
 System.*out*.println("The knobs are moved in attempt to draw the parallelogram.");  
 }  
}  
  
class Rectangle extends Parallelogram {  
 public void Draw(Surface surface) {  
 System.*out*.println("A rectangle is drawn on the surface with ink.");  
 }  
  
 public void Draw(EtchASketch etchASketch) {  
 System.*out*.println("The knobs are moved in attempt to draw the rectangle.");  
 }  
}

class Program {  
 public static void main(String[] args) {  
 Surface surface = new Surface();  
 Surface etchASketch = new EtchASketch();  
  
 List<Shape> shapes = new ArrayList<Shape>();  
 shapes.add(new Shape());  
 shapes.add(new Polygon());  
 shapes.add(new Quadrilateral());  
 shapes.add(new Parallelogram());  
 shapes.add(new Rectangle());  
  
 for (Shape shape : shapes) {  
 surface.Draw(shape);  
 etchASketch.Draw(shape);  
 }  
 }  
}

Executing this example produces the following:

A shape is drawn on the surface with ink.

The knobs are moved in attempt to draw the shape.

A polygon is drawn on the surface with ink.

The knobs are moved in attempt to draw the polygon.

A quadrilateral is drawn on the surface with ink.

The knobs are moved in attempt to draw the quadrilateral.

A parallelogram is drawn on the surface with ink.

The knobs are moved in attempt to draw the parallelogram.

A rectangle is drawn on the surface with ink.

The knobs are moved in attempt to draw the rectangle.

In the above example, virtual dispatch occurs twice for each call to one of the Surfacereferences: Once when the Surface.Draw() virtual method is called and again when either calls the Shape.Draw() overloaded virtual method.  Note again that while the second virtual dispatch is based on the type of Shape instance, the overloaded method called is still determined statically based upon the reference type.

**Consequences**

So, what’s wrong with Double Dispatch?  The problem isn’t so much in the technique, but what design choices might be leading to reliance upon the technique.  Consider for instance the hierarchy of shape types in our Double Dispatch example.  What happens if we want to add a new surface?  In this case, each of the shape types will need to be modified to add knowledge of the new Surface type.  This violates the[Open/Closed Principle](http://en.wikipedia.org/wiki/Open_Closed_Principle), and in this case in a particularly egregious way (i.e. It’s violation is multiplied by the number of shape types we have).   Additionally, it violates the [Single Responsibility Principle](http://en.wikipedia.org/wiki/Single_responsibility_principle).  Changes to how shapes are drawn on a particular surface are likely to differ from surface to surface, thereby leading our shape objects to change for different reasons.

The presence of Double Dispatch generally means that each type in a hierarchy has special handling code within another hierarchy of types.  This approach to representing variant behavior leads to code that is less resilient to future changes as well as being more difficult to extend.

**Conclusion**

Since Double Dispatch is a technique for calling virtual overloaded methods based upon parameter types which exist within an inheritance hierarchy, its use may be a symptom that the Open/Closed and/or Single responsibility principles are being violated, or that responsibilities may otherwise be misaligned.  This is not to say that every case of Double Dispatch means something is amiss, but only that its use should be a flag to reconsider your design in light of future maintenance needs.

import java.util.ArrayList;

import java.util.List;

class Circle {

}

class SmallCircle extends Circle {

}

class BigCircle extends Circle {

}

class Shape {

public void draw(Circle circle) {

System.out.println("... Circle ...");

}

public void draw(BigCircle circle) {

System.out.println("... Big Circle ...");

}

public void draw(SmallCircle circle) {

System.out.println("... Small Circle ...");

}

}

public class Test1 {

public static void main(String[] args) {

Shape shape = new Shape();

List<Circle> circles = new ArrayList<Circle>();

circles.add(new Circle());

circles.add( new BigCircle());

circles.add( new SmallCircle() );

for( Circle circle : circles )

shape.draw(circle);

}

}

----------------------------

Output

... Circle ...

... Circle ...

... Circle ...

**To resolve the above problem, make small change**

package com.ddlab.rnd.type3;

import java.util.ArrayList;

import java.util.List;

class Circle {

public void draw(Shape shape) {

shape.draw(this);

}

}

class SmallCircle extends Circle {

public void draw(Shape shape) {

shape.draw(this);

}

}

class BigCircle extends Circle {

public void draw(Shape shape) {

shape.draw(this);

}

}

class Shape {

public void draw(Circle circle) {

System.out.println("... Circle ...");

}

public void draw(BigCircle circle) {

System.out.println("... Big Circle ...");

}

public void draw(SmallCircle circle) {

System.out.println("... Small Circle ...");

}

}

public class Test1 {

public static void main(String[] args) {

Shape shape = new Shape();

List<Circle> circles = new ArrayList<Circle>();

circles.add(new Circle());

circles.add(new BigCircle());

circles.add(new SmallCircle());

for (Circle circle : circles) {

// shape.draw(circle); //Do not call like this

circle.draw(shape);

}

}

}

Let us take another example like banking to open different accounts like Saving, Demat, Loan etc.

**public class** Account {  
 **public void** open() {  
 System.***out***.println(**"Basic Account opened ..."**);  
 }  
}

**public class** DematAccount **extends** Account {  
}

**public class** LoanAccount **extends** Account {  
}

**public class** SavingsAccount **extends** Account {  
}

**public class** Bank {  
  
 **public void** openAccount(Account act) {  
 System.***out***.println(**"Basic account ...."**);  
}  
  
 **public void** openAccount(SavingsAccount act) {  
 System.***out***.println(**"Savings account ...."**);  
}  
  
 **public void** openAccount(DematAccount act) {  
 System.***out***.println(**"Demat account ...."**);  
}  
  
 **public void** openAccount(LoanAccount act) {  
 System.***out***.println(**"Loan account ...."**);}  
}

**import** java.util.ArrayList;  
**import** java.util.List;  
**public class** Test {  
  
 **public static void** main(String[] args) {  
 List<Account> actList = **new** ArrayList<Account>();  
  
 actList.add( **new** Account());  
 actList.add( **new** SavingsAccount());  
 actList.add( **new** DematAccount());  
 actList.add( **new** LoanAccount());  
  
 Bank bank = **new** Bank();  
 **for**( Account act : actList)  
 bank.openAccount(act);  
 }  
}

**OUTPUT**

Basic account ....

Basic account ....

Basic account ....

Basic account ....

Let us modify the above code using visitor pattern.

**public class** Account {  
  
 **public void** open(Bank bank) {  
 System.***out***.println(**"Basic Account opened ..."**);  
 bank.openAccount(**this**);  
 }  
}

**public class** DematAccount **extends** Account {  
 **public void** open(Bank bank) {  
 System.***out***.println(**"Demat account opened .."**);  
 bank.openAccount(**this**);  
 }  
}

**public class** LoanAccount **extends** Account {  
 **public void** open(Bank bank) {  
 System.***out***.println(**"Loan account opened .."**);  
 bank.openAccount(**this**);  
 }  
}

**public class** SavingsAccount **extends** Account {  
 **public void** open(Bank bank) {  
 System.***out***.println(**"Savings account opened .."**);  
 bank.openAccount(**this**);  
 }  
}

**public class** Bank {  
 **public void** openAccount(Account act) {  
 System.***out***.println(**"Basic account ...."**);  
 }  
  
 **public void** openAccount(SavingsAccount act) {  
 System.***out***.println(**"Savings account ...."**);  
 }  
  
 **public void** openAccount(DematAccount act) {  
 System.***out***.println(**"Demat account ...."**);  
 }  
  
 **public void** openAccount(LoanAccount act) {  
 System.***out***.println(**"Loan account ...."**);  
 }  
}

**import** java.util.ArrayList;  
**import** java.util.List;  
**public class** Test {  
 **public static void** main(String[] args) {  
 List<Account> actList = **new** ArrayList<Account>();  
  
 actList.add( **new** Account());  
 actList.add( **new** SavingsAccount());  
 actList.add( **new** DematAccount());  
 actList.add( **new** LoanAccount());  
  
 Bank bank = **new** Bank();  
 **for**( Account act : actList) {  
*// bank.openAccount(act);//Do not call like this* act.open(bank);  
 }  
 }  
}

**OUTPUT**

Basic Account opened ...

Basic account ....

Savings account opened ..

Savings account ....

Demat account opened ..

Demat account ....

Loan account opened ..

Loan account ....

Let us see one peculiarity in the following example.

import java.util.ArrayList;

import java.util.List;

class SavingAccount {

}

class DematAccount extends SavingAccount {

}

class Bank {

public void open(SavingAccount act ) {

System.out.println("... Opening Saving Account ...");

}

public void open(DematAccount act ) {

System.out.println("... Opening Demat Account ...");

}

}

public class Test {

public static void main(String[] args) {

List<SavingAccount> actList = new ArrayList<SavingAccount>();

Bank bank = new Bank();

actList.add( new SavingAccount());

actList.add( new DematAccount());

for( SavingAccount act : actList ) {

bank.open(act);

}

}

}

Here the output is given below.

**... Opening Saving Account ...**

**... Opening Saving Account ...**

Now let me modify the code and see the output below.

import java.util.ArrayList;

import java.util.List;

class SavingAccount {

public void open() {

System.out.println("... Opened Saving Account Successfully ...");

}

}

class DematAccount extends SavingAccount {

public void open() {

System.out.println("... Opened Demat Account Successfully ...");

}

}

class Bank {

public void open(SavingAccount act ) {

System.out.println("... Opening Saving Account ...");

act.open();

}

public void open(DematAccount act ) {

System.out.println("... Opening Demat Account ...");

act.open();

}

}

public class Test {

public static void main(String[] args) {

List<SavingAccount> actList = new ArrayList<SavingAccount>();

Bank bank = new Bank();

actList.add( new SavingAccount());

actList.add( new DematAccount());

for( SavingAccount act : actList ) {

bank.open(act);

}

}

}

Here the output is ... Opening Saving Account ... ... Opened Saving Account Successfully ... ... Opening Saving Account ... ... Opened Demat Account Successfully ...

Now my question is I am getting the result what I am expecting, why should I go for Visitor pattern, in the above code even if it displays "Saving Account", but it properly executes "Demat Account Code" portion.

Answer by StackOverflow.

Java's calls are polymorphic on objects. act.open() can call the subclass method because the runtime has the object to call it on, but bank.open(act) can't call the method taking a subclass parameter because **bank** is the object on which polymorphism is decided not **act** and**bank** has an **open()** method for **SavingAccount** so that is what has to be called.