Compose Method **

It isn't easy to understand your method's logic

Transform the logic into a
small number of intention-revealing steps
at the same level of detail

```
public boolean contains(Component c) {
   Point p = c.getLocation();
   int locX = new Double(p.getX()).intValue();
   int locY = new Double(p.getY()).intValue();
   boolean completelyWithin =
               (locX >= coords[0] &&
               locY >= coords[1] &&
               (locX+CardComponent.WIDTH) <= coords[2]) &&
               (locY+CardComponent.HEIGHT) <= coords[3];
   if (completelyWithin) return true;

   locX = locX+CardComponent.WIDTH;
   locY = locY+CardComponent.HEIGHT;
   boolean partiallyWithin =
               (locX > coords[0] &&
               locY > coords[1] &&
               (locX < coords[2]) &&
               (locY < coords[3]));

   return partiallyWithin;
}
```

```
public boolean contains(Component c) {
   return containsEntireCard(c) || containsRightCorner(c) ;
}
```

Motivation

Kent Beck once said that some of his best patterns are those that he thought someone would laugh at him for writing. *Composed Method* [Beck] may be such a pattern. A Composed Method is a small, simple method that is easy to understand. Do you write a lot of Composed Methods? I like to think that I do, but I often find that I don't, at first. So I have to go back and refactor to this pattern. When my code has many Composed Methods, it tends to be a easy to use, easy to read and easy to extend.

I gave this refactoring two asterisks because I find myself aggressively refactoring to it *often*. For example, just the other day I was debugging a method in some code I'm writing. The method, called `contains()`, wasn't very complex, but it was complex enough that I had to think about how it was doing its job. I knew this method would be easier to debug if I refactored it first. But my ego wasn't ready for that, just then: I just wanted to get rid of the bug. So, after writing an automated test to demonstrate the bug, I wrote new code in the `contains()` method to fix the bug. That code didn't fix the bug and after two more failed attempts, I was ready to refactor. It wasn't difficult to transform `contains()` into a Composed Method. But after doing so, it was so much easier to follow the logic. And moments after the refactoring, I found and fixed my bug.

Communication	Duplication	Simplicity
It may be clear *what* a method does but not *how* the method does what it does. Make the "how" easy to understand by clearly communicating every logical step. You'll often implement this refactoring using Extract Method [Fowler].	Duplicate code, whether blatant or subtle, clutters a method's logic. Remove the duplication to make the code smaller and simpler. Doing so often reveals further refactoring opportunities.	Composed Methods often read like English. If your method has too many lines of code, such that you can't easily explain how it does its job, simplify it by extracting logic till it is a Composed Method.

Mechanics

This is one of the most important refactorings I know of. Conceptually, it is also one of the simplest. So you'd think that this refactoring would lead to a simple set of mechanics. In fact, just the opposite is the case. While the steps themselves aren't complex, there is no simple, repeatable set of those steps. There are, however, certain guiding ideas to get your code to be a Composed Method, some of which are:

- *Small Methods* – Composed Methods are rarely more than 10 lines of code, and are usually more like 5.

- *Duplication Removal* – Reduce the amount of code in the method by getting rid of blatant and/or subtle code duplication.

- *Communicate Intention* – do so with the names of your variables and methods, and by making your code simple.

- *Simplicity* – there are many ways to skin a cat. Refactor to the way that is most simple and that best communicates your intention. Simple methods may not be the most highly optimized methods. Don't worry about that. Make your code simple and optimize it later.

- *Similar Levels* – when you break up one method into chunks of behavior, make the chunks operate at similar levels. For example, if you have a piece of detailed conditional logic mixed in with some high-level method calls, you have code at different levels. Push the detail into a new or existing high-level chunk.

- *Group Related Code* – Some code is simply hard to extract into its own method. You can easily see a way to extract part of the code, but the rest remains in the original method. You now have *code at different levels*. In addition, because you have an unnatural split between related fragments of code, your code is harder to follow. In general, look for ways to *group* related code fragments, even if they aren't obvious at first.

Let's now look at three examples of refactoring to Composed Method:

Example 1

I'll start with the game example from the code sketch above. We begin with a single bulky method, called `contains()`, which figures out whether a Component is fully or partially contained within a rectangular area:

```
public boolean contains(Component c) {
   Point p = c.getLocation();
   int locX = new Double(p.getX()).intValue();
   int locY = new Double(p.getY()).intValue();
   boolean completelyWithin =
     (locX >= coords[0] &&
     locY >= coords[1] &&
     (locX+CardComponent.WIDTH) <= coords[2]) &&
     (locY+CardComponent.HEIGHT) <= coords[3];
   if (completelyWithin) return true;

   locX = locX+CardComponent.WIDTH;
   locY = locY+CardComponent.HEIGHT;
   boolean partiallyWithin =
     (locX > coords[0] &&
     locY > coords[1] &&
     (locX < coords[2]) &&
     (locY < coords[3]));

   return partiallyWithin;
}
```

Before we get into the refactoring, let's look at one of six test methods for the `contains()` method. The following method tests to see if a card is initially contained within the first player's play area, then moves the card out of the first player's play area and follows that with another test:

```
public void testCardOutOfPlayAreaOne() {
    Hand hand = (Hand)explanations.getCurrentPlayer().getHand();
    Card card = (Card)hand.elements().nextElement();
    CardComponent c = new CardComponent(card,explanations);
    PlayerArea area = explanations.getPlayerArea(0);
    explanations.moveCard(c, area.upperLeft());
    assertEquals("area contains card", true, area.contains(c));

    explanations.moveCard(c, CardComponent.WIDTH + 10, CardComponent.HEIGHT + 10);
    assertEquals("area does not contain card", false, area.contains(c));
}
```

The above test, and the other five tests, all pass (or "run green') before I begin refactoring. I run these tests after each of the small steps I am about to do below.

To begin, my first impulse is to make the `contains()` method *smaller*. That leads me to look at the conditional represented by the variable, `completelyWithin`:

```
   boolean completelyWithin =
     (locX >= coords[0] &&
     locY >= coords[1] &&
     (locX+CardComponent.WIDTH) <= coords[2]) &&
     (locY+CardComponent.HEIGHT) <= coords[3];
```

While that variable helps make it clear what the conditional logic does, the `contains()` method would be smaller and easier to read if this fragment were in it's own method. So I start with an Extract Method:

```
public boolean contains(Component c) {
    Point p = c.getLocation();
    int locX = new Double(p.getX()).intValue();
    int locY = new Double(p.getY()).intValue();
    if (completelyWithin(locX, locY)) return true;

    locX = locX+CardComponent.WIDTH;
    locY = locY+CardComponent.HEIGHT;
    boolean partiallyWithin =
            (locX > coords[0] &&
            locY > coords[1] &&
            (locX < coords[2]) &&
            (locY < coords[3]));
    return partiallyWithin;
}
```

```
private boolean completelyWithin(int locX, int locY) {
    return  (locX >= coords[0] &&
             locY >= coords[1] &&
            (locX+CardComponent.WIDTH) <= coords[2]) &&
            (locY+CardComponent.HEIGHT) <= coords[3];
}
```

Next, after seeing a similar temporary variable, called `partiallyWithin`, I do another Extract Method:

```
public boolean contains(Component c) {
    Point p = c.getLocation();
    int locX = new Double(p.getX()).intValue();
    int locY = new Double(p.getY()).intValue();
    if (completelyWithin(locX, locY)) return true;
    locX = locX+CardComponent.WIDTH;
    locY = locY+CardComponent.HEIGHT;
    return partiallyWithin(locX, locY);
}

private boolean partiallyWithin(int locX, int locY) {
    return (locX > coords[0] &&
             locY > coords[1] &&
            (locX < coords[2]) &&
            (locY < coords[3]));
}
```

The `contains()` method is now smaller and simpler, but it still seems cluttered with variable assignments. I notice that the assignments to `locX` and `locY` are performed simply for use by the new methods, `completelyWithin()` and `partiallyWithin()`. I decide to let those methods deal with the `locX` and `locY` assignments. The easiest way to do this is to just pass the `Point` variable, `p`, to each of the methods:

```
public boolean contains(Component c) {
    Point p = c.getLocation();
    if (completelyWithin(p)) return true;
    return partiallyWithin(p);
}
```

Now, the `contains()` method is really looking smaller and simpler. I feel like I'm done. But then I look at that first line of code:

```
Point p = c.getLocation();
```

The level of that code seems wrong – it is a detail, while the rest of the code in the method represents core pieces of logic. The two methods I'm calling each need the `Point` variable. But each of those methods could easily obtain the `Point` variable if I just sent them `Component c`. I consider doing that, but then I worry about violating the rule of doing things *once and only once*. For if I pass variable `c`, the `Component`, to each method, each method will have to contain code to obtain a `Point` from `c`, instead of just getting one passed in directly.

Hmmmm. What is my real goal here? Is it more important to get the levels of the code right or to say things once and only once? After some reflection, I realize that my goal is to produce a method that can be read and understood in seconds. But as it stands, that first line of code takes away from the readability and simplicity of the method. I push down the code to obtain a `Point` into the two called methods and end up with the following:

```
public boolean contains(Component c) {
    return completelyWithin(c) || partiallyWithin(c);
}
```

I've now turned 17 lines of code into 1 line, which calls two methods. My tests pass and I'm content.

Example 2

```
public static Vector wrap(String s) {
   Vector wrapVector = new Vector();
   String words;
   String word;
   int lastPos;
   do {
      if (s.length() > 16) {
         words="";
         word="";
         lastPos=0;
         for (int i=0;i<16;i++) {
            if (s.charAt(i)==' ' || s.charAt(i)=='-') {
               words+=word+s.charAt(i);
               lastPos = i+1;
               word="";
            } else word+=s.charAt(i);
         }
         if (lastPos==0) {
            // Rare case that there was no space or dash, insert one and break
            words+=word+"-";
            lastPos=16;
         }
         wrapVector.addElement(words);
         s = s.substring(lastPos, s.length());
      }
   } while (s.length() > 16);
   if (s.length()>0) wrapVector.addElement(s);
   return wrapVector;
}
```

```
public static Vector wrap(StringBuffer cardText) {
   Vector wrapLines = new Vector();
   while (cardText.length() > 0)
      wrapLines.addElement(extractPhraseFrom(cardText));
   return wrapLines;
}

private static String extractPhraseFrom(StringBuffer cardText) {
   String substring = "";
   String word="";
   final int MAX_CHARS = Math.min(MAX_LINE_WIDTH, cardText.length());
   for (int i=0; i<MAX_CHARS; i++) {
      word += cardText.charAt(i);
      if (cardText.charAt(i)==' ' ||
          cardText.charAt(i)=='-' ||
          cardText.toString().endsWith(word)) {
         substring += word;
         word="";
      }
   }
   if (substring.length() == 0)
      substring=word+"-";
   cardText.delete(0, substring.length());
   return substring;
}
```

In a game I've been writing with a friend, text needs to be displayed on graphical cards. The text is typically too long to fit on one line of each card, so it must be displayed on multiple lines of each card. To enable this behavior, we test-first programmed a `wrap()` method. Here are a few of the tests:

```
public void accumulateResult(String testString) {
    int i = 0;
    for (Enumeration e = CardComponent.wrap(testString).elements();e.hasMoreElements();)
        result[i++] = (String)e.nextElement();
}

public void testWrap() {
    accumulateResult("Developers Misunderstand Requirements");
    assertEquals("First line","Developers ",result[0]);
    assertEquals("Second line","Misunderstand ",result[1]);
    assertEquals("Third line","Requirements",result[2]);
}
public void testWrap2() {
    accumulateResult("Stories Are Too Complex");
    assertEquals("First line","Stories Are Too ",result[0]);
    assertEquals("Second line","Complex",result[1]);
}
public void testWrap3() {
    accumulateResult("Intention-Revealing Code");
    assertEquals("First line","Intention-",result[0]);
    assertEquals("Second line","Revealing Code",result[1]);
}
```

With these tests in place, I can work on refactoring the big fat method shown below:

```
public static Vector wrap(String s) {
    Vector wrapVector = new Vector();
    String words;
    String word;
    int lastPos;
    do {
        if (s.length() > 16) {
            words="";
            word="";
            lastPos=0;
            for (int i=0;i<16;i++) {
                if (s.charAt(i)==' ' || s.charAt(i)=='-') {
                    words+=word+s.charAt(i);
                    lastPos = i+1;
                    word="";
                } else word+=s.charAt(i);
            }
            if (lastPos==0) {
                // Rare case that there was no space or dash, insert one and break
                words+=word+"-";
                lastPos=16;
            }
           wrapVector.addElement(words);
           s = s.substring(lastPos, s.length());
        }
    } while (s.length() > 16);
    if (s.length()>0) wrapVector.addElement(s);
    return wrapVector;
}
```

The first thing I notice is that we have some blatant duplicate logic: the line, `s.length() > 16`, appears in a conditional statement at line 6 and at the end of the while statement. No good. I experiment with removing the duplication by using a `while` loop instead of a `do..while` loop. The tests confirm that this experiment works:

```
public static Vector wrap(String s) {
    Vector wrapVector = new Vector();
    String words;
    String word;
    int lastPos;
    while (s.length() > 16) {
        words="";
        word="";
        lastPos=0;
        for (int i=0;i<16;i++)
```

```
            if (s.charAt(i)==' ' || s.charAt(i)=='-') {
                words+=word+s.charAt(i);
                lastPos = i+1;
                word="";
            } else word+=s.charAt(i);
        if (lastPos==0) {
            // Rare case that there was no space or dash, insert one and break
            words+=word+"-";
            lastPos=16;
        }
        wrapVector.addElement(words);
        s = s.substring(lastPos, s.length());
    }
    if (s.length()>0) wrapVector.addElement(s);
    return wrapVector;
}
```

Next I notice more duplication. At two places in the middle of the method, the code says:

`word+=s.charAt(i)`.

By consolidating this logic, I see a way to simplify a conditional statement:

```
for (int i=0;i<16;i++) {
    word+=s.charAt(i);  // now we say this only once
    if (s.charAt(i)==' ' || s.charAt(i)=='-') {
        words+=word;
        lastPos = i+1;
        word="";
    } // else statement is no longer needed
}
```

Additional duplicate logic doesn't jump out at me just yet, so I continue to look (I know it is there!). I wonder about the variable, `lastPos`. What does it store? Can I figure out what the value of `lastPos` would be, without having to declare and set a variable for it? After a little bit of study, I try some experiments. Gradually it dawns on me that `words.length()` contains the exact value as that held by `lastPos`. This allows me to get rid of another variable, and all of the assignments to it:

```
public static Vector wrap(String s) {
    Vector wrapVector = new Vector();
    String words;
    String word;
    while (s.length() > 16) {
        words="";
        word="";
        for (int i=0;i<16;i++) {
            word+=s.charAt(i);
            if (s.charAt(i)==' ' || s.charAt(i)=='-') {
                words+=word;
                word="";
            }
        }
        if (words.length() == 0)  // if no space or dash, insert one
            words+=word+"-";
        wrapVector.addElement(words);
        s = s.substring(words.length(), s.length());
    }
    if (s.length()>0) wrapVector.addElement(s);
    return wrapVector;
}
```

The code is definitely getting smaller and more manageable. But the body of the while method still seems big and bulky. I decide to *Extract Method* [Fowler]:

```
public static Vector wrap(String s) {
    Vector wrapVector = new Vector();
    String words;
```

```
    while (s.length() > 16) {
        words = extractPhraseFrom(s);
        wrapVector.addElement(words);
        s = s.substring(words.length(), s.length());
    }
    if (s.length()>0) wrapVector.addElement(s);
    return wrapVector;
}

private static String extractPhraseFrom(String cardText) {
    String phrase = "";
    String word="";
    for (int i=0;i<16;i++) {
        word += cardText.charAt(i);
        if (cardText.charAt(i)==' ' || cardText.charAt(i)=='-') {
            phrase += word;
            word="";
        }
    }
    if (phrase.length() == 0) // no found space or dash, insert dash
        phrase+=word+"-";
    return phrase;
}
```

We're making progress. But I'm still not happy with the `wrap()` method: I don't like the fact that the code is adding elements to the `wrapVector` both inside and outside the while loop and I also don't like the mysterious line that changes the value of the String "s" (which is a bad name for a variable that holds on to a card's text):

```
s = s.substring(words.length(), s.length());
```

So I ask myself how I can make this logic clearer? Given some card text, I would like my code to show how the text is broken up into pieces, added to a collection and returned. I decide that the best way to achieve this objective is to push all code that is responsible for creating a "phrase" into the `extractPhraseFrom()` method. I hope to end up with a while loop that has one line of code.

My first step is to rename and change the type of the `String` variable, `s`. I call it `cardText` and change it to be `StringBuffer`, since it will be altered by the `extractPhraseFrom()` method. This change requires that I make all callers of `wrap()` pass in a StringBuffer instead of a String. As I go about doing this work, I see that I can also get rid of the temporary variable, `word`, leaving the following:

```
public static Vector wrap(StringBuffer cardText) {
    Vector wrapVector = new Vector();
    while (cardText.length() > 16) {
        wrapVector.addElement(extractPhraseFrom(cardText));
        cardText.delete(0, words.length());
    }
    if (cardText.length()>0) wrapVector.addElement(cardText.toString());
    return wrapVector;
}
```

Now I must figure out how to push the fragmented pieces of phrase-construction logic down into the `extractPhraseFrom()` method. My tests give me a lot of confidence as I go about this work. I go for the low-hanging fruit first: the code that deletes a substring from cardText can easily be moved to `extractPhraseFrom()`, which yields the following:

```
public static Vector wrap(StringBuffer cardText) {
    Vector wrapVector = new Vector();
    while (cardText.length() > 16)
        wrapVector.addElement(extractPhraseFrom(cardText));
    if (cardText.length()>0) wrapVector.addElement(cardText.toString());
    return wrapVector;
}
```

Now, I've just got the line of code after the while loop to worry about:

```
if (cardText.length()>0) wrapVector.addElement(cardText.toString());
```

How can I get that code to live in the `extractPhraseFrom()` method? I study the while loop and see that I'm looping on this magic number, 16. First, I decide to make a constant for that number, called MAX_LINE_WIDTH. Then, as I continue to study the loop, I wonder why the `wrap()` method has two conditionals fragments that check `cardText.length()`, (one in the while loop and one after the while loop). I want to remove that duplication. I decide to change the `while` loop to do its thing while `cardText.length() > 0`.

This last change requires a few changes to the `extractPhraseFrom` method to make it capable of handling the case when a line of text isn't greater than 16 characters (now called `MAX_LINE_WIDTH`). Once the tests confirm that everything is working, wrap() now feels like a true Composed Method. Here's what it now looks like:

```
public static Vector wrap(StringBuffer cardText) {
    Vector wrapLines = new Vector();
    while (cardText.length() > 0)
        wrapLines.addElement(extractPhraseFrom(cardText));
     return wrapLines;
}

private static String extractPhraseFrom(StringBuffer cardText) {
    String phrase = "";
    String word="";
    final int MAX_CHARS = Math.min(MAX_LINE_WIDTH, cardText.length());
    for (int i=0; i<MAX_CHARS; i++) {
        word += cardText.charAt(i);
        if (cardText.charAt(i)==' ' || cardText.charAt(i)=='-' ||
             cardText.toString().endsWith(word)) {
             phrase += word;
             word="";
        }
    }
    if (phrase.length() == 0)
        phrase=word+"-";
    cardText.delete(0, phrase.length());
    return phrase;
}
```

You may wonder now about the `extractPhraseFrom` method. Is it Composed? Not really. In the real program, we refactored it as well, but I'll spare you the details here.

Example 3

```
private void paintCard(Graphics g) {
   Image image = null;
   if (card.getType().equals("Problem")) {
      image = explanations.getGameUI().problem;
   } else if (card.getType().equals("Solution")) {
      image = explanations.getGameUI().solution;
   } else if (card.getType().equals("Value")) {
      image = explanations.getGameUI().value;
   }
   g.drawImage(image,0,0,explanations.getGameUI());

   if (highlight)
      paintCardHighlight(g);
   paintCardText(g);
}
```

```
private void paintCard(Graphics g) {
   paintCardImage(g);
   paintCardHighlight(g);
   paintCardText(g);
}
```

The above, original `paintCard()` method isn't long, nor is it complicated. It paints a card image, checks a flag to see if it must paint a card highlight, and then paints text onto the card. Painting the card highlight and card text are performed by the methods, `paintCardHighlight()` and `paintCardText()`. But the code that paints the card image lives not in a separate method but in the `paintCard()` method itself. So? Well, consider the refactored version of `paintCard()`. I can look at the refactored version and know what it does in 2 seconds, while I have to spend a few brain cycles to figure out what the previous version does. Trivial difference? *No*, not when you consider how much simpler an entire system is when it consists of *many* composed methods, like `paintCard()`.

So what was the *smell* that led to this refactoring? *Code at different levels*: raw code mixed with higher-level code. When the method contains code at the same levels, it is easier to read and understand. As the guidelines in the mechanics section, above, say, Composed Methods tend to have code at the *same level.*

Implementing this refactoring was incredibly easy. I did Extract Method [Fowler] as follows:

```
private void paintCard(Graphics g) {
    paintCardImage(g);
    if (highlight)
        paintCardHighlight(g);
    paintCardText(g);
}

private void paintCardImage(Graphics g) {
   Image image = null;
   if (card.getType().equals("Problem")) {
      image = explanations.getGameUI().problem;
   } else if (card.getType().equals("Solution")) {
      image = explanations.getGameUI().solution;
   } else if (card.getType().equals("Value")) {
      image = explanations.getGameUI().value;
   }
   g.drawImage(image,0,0,explanations.getGameUI());
}
```

To finish this refactoring, I took the sole conditional statement in the method (`if (highlight)…`) and pushed it down into the `paintCardHightlight()` method. Why? I

wanted the reader to simply see three steps: paint image, highlight image and paint card text. The detail of whether or not we do highlight the card isn't important to me – the reader can find that out if they look. But if that confuses other programmers, I'd be happy to see the method name changed or the conditional brought back to the `contains()` method.

```
private void paintCard(Graphics g) {
    paintCardImage(g);
    paintCardHighlight(g);
    paintCardText(g);
}
```