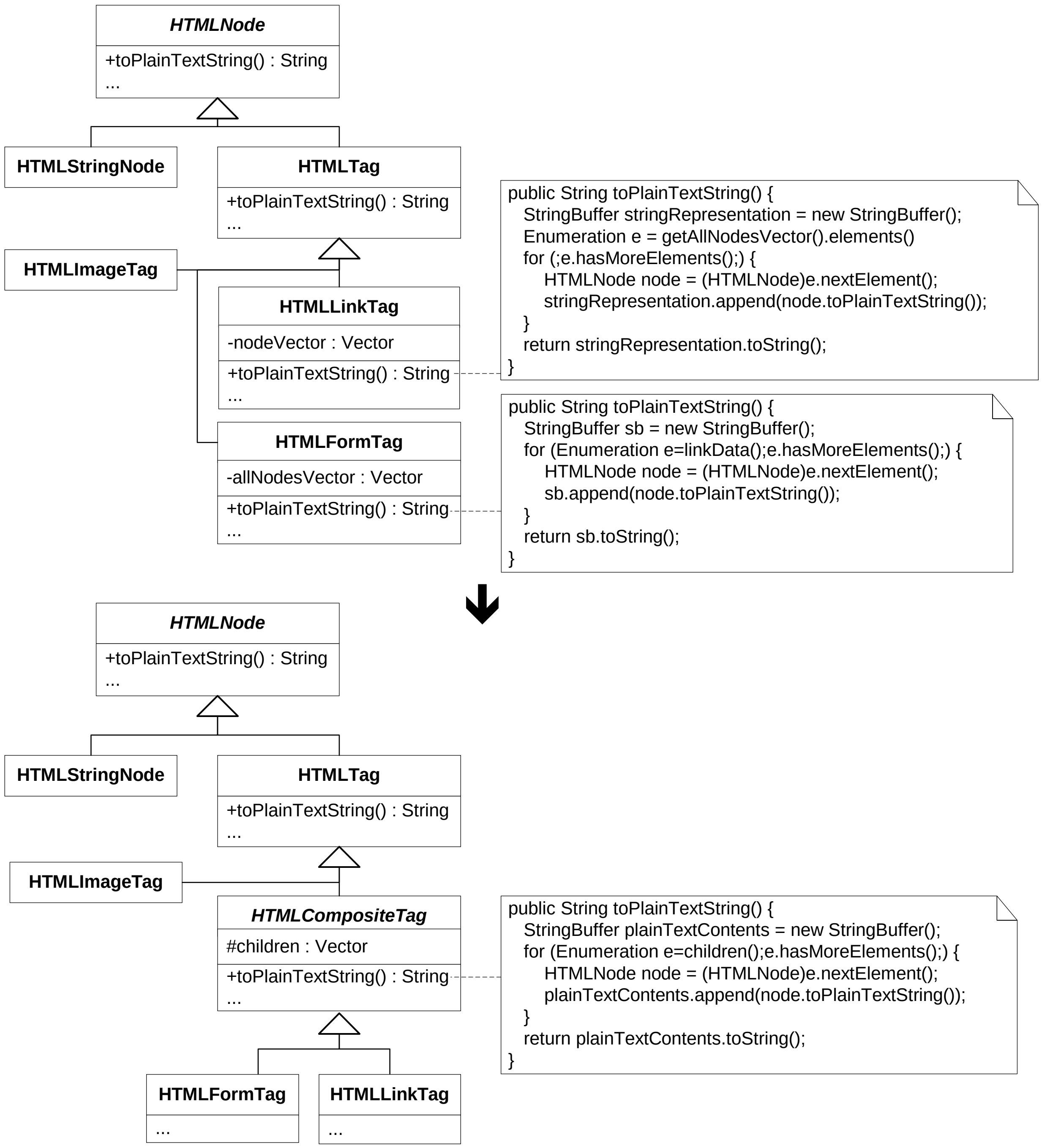
Extract Composite

Classes in a hierarchy have duplicate fields and logic
for storing and processing children from the same hierarchy

Create a Composite superclass and
move the duplicated fields and logic to it



```
public String toPlainTextString() {
   StringBuffer stringRepresentation = new StringBuffer();
   Enumeration e = getAllNodesVector().elements()
   for (;e.hasMoreElements();) {
      HTMLNode node = (HTMLNode)e.nextElement();
      stringRepresentation.append(node.toPlainTextString());
   }
   return stringRepresentation.toString();
}
```

```
public String toPlainTextString() {
   StringBuffer sb = new StringBuffer();
   for (Enumeration e=linkData();e.hasMoreElements();) {
      HTMLNode node = (HTMLNode)e.nextElement();
      sb.append(node.toPlainTextString());
   }
   return sb.toString();
}
```

```
public String toPlainTextString() {
   StringBuffer plainTextContents = new StringBuffer();
   for (Enumeration e=children();e.hasMoreElements();) {
      HTMLNode node = (HTMLNode)e.nextElement();
      plainTextContents.append(node.toPlainTextString());
   }
   return plainTextContents.toString();
}
```

Motivation

In *Extract Superclass [F]*, Martin Fowler explains that if you have two or more classes with similar features, it makes sense to put the common features into a superclass. This refactoring is similar only it's concerned with the case when the common features duplicated across subclasses are begging to be refactored into a Composite [GoF].

I often encounter subclasses in hierarchies that store collections of children and have methods for reporting information about those children. When the children being collected happen to be classes in the same hierarchy, there's a good chance that much duplicate code could be removed by refactoring to a Composite.

Duplication removal is at the heart of this refactoring and *Extract Superclass [F]*. When you have the type of child-related code I described above, following the mechanics for either refactoring will lead you to the creation of a Composite. So why did I choose to write this special case refactoring? Mostly because I think it's useful to give folks examples of duplication that are more subtle than others and to demonstrate different ways to remove the duplication. In addition, it would be excellent if the narrower mission of this refactoring (i.e., the removal of duplicate code related only to collections of children and related methods) made it easier for tools vendors to automate this refactoring.

Communication	Duplication	Simplicity
To-do	To-do	To-do

Mechanics

Since this refactoring is a generalized case of Martin Fowler's refactoring, *Extract Superclass [F]*, it's mechanics are similar. The main difference is the extent to which you'll extract code to a superclass. For this refactoring, the extracted code is the child-handling logic that is similar across classes in a hierarchy. Once you finish this refactoring, you can continue to pull common functionality up into your newly created Composite by following the mechanics in *Extract Superclass [F]*.

1. Create a *composite*, an abstract superclass, named to reflect that it will contain children (e.g. `CompositeTag`).

2. Make each *child-container* (i.e. a class in the hierarchy that contains duplicate child-handling code) a subclass of *composite*.

3. In a *child-container*, find a child-processing method that is *purely-duplicated* or *partially-duplicated* across the *child-containers*. A *purely-duplicated* method will have the same method body with the same or different method names across *child-containers*. A *partially-duplicated* method will have a method body with common *and* uncommon code and the same or different method names across *child-containers*.

   Whether you've found a *purely-duplicated* or *partially-duplicated* method, if its name isn't consistent across *child-containers*, make it consistent by applying *Rename Method [F]*.

   For a *purely-duplicated* method, move the child collection field referenced by the method to the *composite* by applying *Pull Up Field [F]*. Rename this field if its name doesn't make sense for all *child-containers*. Now move the method to the *composite* by applying *Pull Up Method [F]*. If the pulled-up method relies on contructor code still residing in *child-containers*, pull up that code to the *composite*'s constructor.

   For a *partially-duplicated* method, see if the method body can be made consistent across all *child-containers* using *Substitute Algorithm [F]*. If so, refactor it as a *purely-duplicated* method. Otherwise, extract the code that is common across all *child-container* implementations using *Extract Method [F]* and pull it up to the composite using *Pull Up Method [F]*. If the method body follows the same sequence of steps, some of which are implemented differently, see if a skeleton of the method body can be pulled up using *Form Template Method (149).*

   ✓ Compile and test after every one of the above-mentioned refactorings.

4. Repeat step 3 for child-processing methods in the child-containers that contain *purely-duplicated* or *partially-duplicated* code.

5. Check each client of each *child-container* to see if it can now communicate with the *child-container* using the composite interface. If it can, make it do so.

Example

I've been refactoring some code on an open-source project that was started by my colleague, Somik Raha. The project is an HTML parser (see http://sourceforge.net/projects/htmlparser). When the parser parses a piece of HTML, it identifies and creates objects representing HTML tags and pieces of text. For example, here's some HTML:

```
<HTML>
    <BODY>
        Hello, and welcome to my web page! I work for
        <A HREF="http://industriallogic.com">
            <IMG SRC="http://industriallogic.com/images/logo141x145.gif">
        </A>
    </BODY>
</HTML>
```

Given such HTML, the parser would create objects of the following types:

`HTMLTag` (for the `<BODY>` tag)
`HTMLStringNode` (for the `String`, "Hello, and welcome…")
`HTMLLinkTag` (for the `<A HREF="…">` tag)

You might wonder what the parser does with the `<IMG SRC…>` tag? That tag, which represents an `HTMLImageTag`, is treated as a child of the `HTMLLinkTag`. When the parser notices that the link tag contains an image tag, it constructs and gives one `HTMLImageTag` object as a child to the `HTMLLinkTag` object.

Additional tags in the parser, such as `HTMLFormTag`, `HTMLTitleTag` and others, are also child-containers. As I studied some of these classes, it didn't take long to spot duplicate code for storing and handling child nodes. For example, consider the following:

```
public class HTMLLinkTag extends HTMLTag...
    private Vector nodeVector;

    public String toPlainTextString() {
        StringBuffer sb = new StringBuffer();
        HTMLNode node;
        for (Enumeration e=linkData();e.hasMoreElements();)
        {
            node = (HTMLNode)e.nextElement();
            sb.append(node.toPlainTextString());
        }
        return sb.toString();
    }

public class HTMLFormTag extends HTMLTag...
    protected Vector allNodesVector;

    public String toPlainTextString() {
        StringBuffer stringRepresentation = new StringBuffer();
        HTMLNode node;
        for (Enumeration e=getAllNodesVector().elements();e.hasMoreElements();) {
            node = (HTMLNode)e.nextElement();
            stringRepresentation.append(node.toPlainTextString());
        }
        return stringRepresentation.toString();
    }
```

Since `HTMLFormTag` and `HTMLLinkTag` both contain children, they both have a `Vector` for storing children, though it goes by a different name in each class. Since both classes need to support the `toPlainTextString()` operation, which outputs the non-HTML-formatted text of the tag's children, both contain logic to iterate over their children and produce plain text. Yet the code to do this operation is nearly identical in these classes! In fact, there are several nearly-identical methods in the child-container classes, all of which reek from duplication. So let's apply *Extract Composite* to this code:

1. I must first create an abstract class that will become the superclass of the *child-container* classes. Since the child-container classes, like `HTMLLinkTag` and `HTMLFormTag`, are already subclasses of `HTMLTag`, I create the following:

```
public abstract class CompositeTag extends HTMLTag {
    public CompositeTag(
        int tagBegin,
        int tagEnd,
        String tagContents,
        String tagLine) {
        super(tagBegin, tagEnd, tagContents, tagLine);
    }
}
```

2. Now I make the child-containers subclasses of `CompositeTag`:

```
public class HTMLFormTag extends CompositeTag

public class HTMLLinkTag extends CompositeTag
```

and so on…

(Note, for the remainder of this refactoring, I'll only show code from two *child-containers*, `HTMLLinkTag` and `HTMLFormTag`, even though there are others in the code base).

3. I look for a *purely-duplicated* method across all *child-containers* and find `toPlainTextString()`. Since this method has the same name in each *child-container*, I don't have to change its name anywhere. My first step is to pull up the child Vector that stores children. I do this using the `HTMLLinkTag` class:

```
public class HTMLLinkTag extends CompositeTag...
    ~~private Vector nodeVector;~~

public abstract class CompositeTag extends HTMLTag...
    protected Vector nodeVector;  // pulled-up field
```

Since I want `HTMLFormTag` to use the same newly-pulled-up `Vector`, `nodeVector` (yes, it's an awful name, I'll change it soon), I rename its local child `Vector` to be `nodeVector`:

```
public class HTMLFormTag extends CompositeTag...
    ~~protected Vector allNodesVector;~~
    protected Vector nodeVector;
    ...
```

And then I delete this local field (since `HTMLFormTag` inherits it):

```
public class HTMLFormTag extends CompositeTag...
    ~~protected Vector nodeVector;~~
```

Now I can rename `nodeVector` in the composite:

```
public abstract class CompositeTag extends HTMLTag...
```

```
    ~~protected Vector nodeVector;~~
    protected Vector children;
```

I'm now ready to pull up the `toPlainTextString()` method to `CompositeTag`. My first attempt at doing this using the automated refactoring tool in Eclipse fails, because the two methods aren't identical in `HTMLLinkTag` and `HTMLFormTag`. The trouble is, `HTMLLinkTag` gets an iterator on its children by means of the method, `linkData()`,while `HTMLFormTag` gets an iterator on its children by means of the `getAllNodesVector().elements()`:

```
public class HTMLLinkTag extends CompositeTag
    public Enumeration linkData()
    {
        return children.elements();
    }

    public String toPlainTextString()...
        for (Enumeration e=linkData();e.hasMoreElements();)
            ...

public class HTMLFormTag extends CompositeTag...
    public Vector getAllNodesVector() {
        return children;
    }
    public String toPlainTextString()...
        for (Enumeration e=getAllNodesVector().elements();e.hasMoreElements();)
            ...
```

To fix this problem, I must create a consistent method for getting access to a `CompositeTag`'s children. I won't bore you with the steps. I end up with:

```
public abstract class CompositeTag extends HTMLTag...
    public Enumeration children() {
        return children.elements();
    }
```

…and a version of `toPlainTextString()` that's now identical in `HTMLLinkTag` and `HTMLFormTag`:

```
public String toPlainTextString() {
    StringBuffer plainTextContents = new StringBuffer();
    HTMLNode node;
    for (Enumeration e=children();e.hasMoreElements();) {
        node = (HTMLNode)e.nextElement();
        plainTextContents.append(node.toPlainTextString());
    }
    return plainTextContents.toString();
}
```

The automated refactoring in Eclipse now lets me easily pull up `toPlainTextString()` to `CompositeTag`. I run my tests and everything is ok.

4. In this step I repeat step 3 for additional methods that may be pulled-up from the *child-containers* to the *composite*. There happen to be several of these methods. I'll show you one that involves a method called `toHTML()`. This method outputs the HTML of a given node. Both `HTMLLinkTag` and `HTMLFormTag` have their own implementations for this methods. To implement step 3, I must first decide if `toHTML()` is *purely-duplicated* or *partially-duplicated*.

Here's a look at how `HTMLLinkTag` implements the method:

```
public class HTMLLinkTag extends CompositeTag
```

```
    public String toHTML() {
        StringBuffer sb = new StringBuffer();
        putLinkStartTagInto(sb);
        //sb.append(tagContents.toString());
        HTMLNode node;
        for (Enumeration e = children();e.hasMoreElements();) {
            node = (HTMLNode)e.nextElement();
            sb.append(node.toHTML());
        }
        sb.append("</A>");
        return sb.toString();
    }

    public void putLinkStartTagInto(StringBuffer sb) {
        sb.append("<A ");
        String key,value;
        int i = 0;
        for (Enumeration e = parsed.keys();e.hasMoreElements();) {
            key = (String)e.nextElement();
            i++;
            if (key!=TAGNAME) {
                value = getParameter(key);
                sb.append(key+"=\""+value+"\"");
                if (i<parsed.size()-1) sb.append(" ");
            }
        }
        sb.append(">");
    }
```

After creating a buffer, putLinkStartTagInto(…) deals with getting the contents of the start tag into the buffer, along with any attributes it may have. The start tag would be something like `<A HREF="…">` or `<A NAME="…">`., where `HREF` or `NAME` represent attributes of the tag. The tag could have children, such as an `HTMLStringNode`, as in `<A HREF="…">I'm a string node</A>` or child `HTMLImageTags`. Finally there is the end tag, `</A>`, which must be added to the result buffer before the HTML representation of the tag is returned.

Let's now see how `HTMLFormTag` implements this method:

```
public class HTMLFormTag extends CompositeTag...
    public String toHTML() {
        StringBuffer rawBuffer = new StringBuffer();
        HTMLNode node,prevNode=null;
        rawBuffer.append("<FORM METHOD=\""+formMethod+"\" ACTION=\""+formURL+"\"");
        if (formName!=null && formName.length()>0)
            rawBuffer.append(" NAME=\""+formName+"\"");
        Enumeration e = children.elements();
        node = (HTMLNode)e.nextElement();
        HTMLTag tag = (HTMLTag)node;
        Hashtable table = tag.getParsed();
        String key,value;
        for (Enumeration en = table.keys();en.hasMoreElements();) {
            key=(String)en.nextElement();
            if (!(key.equals("METHOD")
               || key.equals("ACTION")
               || key.equals("NAME")
               || key.equals(HTMLTag.TAGNAME))) {
                value = (String)table.get(key);
                rawBuffer.append(" "+key+"="+"\""+value+"\"");
            }
        }
        rawBuffer.append(">");
        rawBuffer.append(lineSeparator);
        for (;e.hasMoreElements();) {
            node = (HTMLNode)e.nextElement();
            if (prevNode!=null) {
                if (prevNode.elementEnd()>node.elementBegin()) {
                    // Its a new line
                    rawBuffer.append(lineSeparator);
                }
            }
            rawBuffer.append(node.toHTML());
            prevNode=node;
        }
        return rawBuffer.toString();
    }
```

This implementation has some similarities and differences to the `HTMLLinkTag` implementation. Therefore, according to the definition in the mechanics, `toHTML()` should be treated as a *partially-duplicated* child-container method. That means that my next step is to see if I can make one implementation of this method by applying the refactoring, *Substitute Algorithm [F]*.

It turns out that I can. It is easier than it looks because both versions of `toHTML()` essentially do the same three things: output the start tag along with any attributes, output any child tags, output the closed tag. Knowing that, I arrive at one method for dealing with the start tag, which I pulled-up to *composite*:

```
public abstract class CompositeTag extends HTMLTag...
    public void putStartTagInto(StringBuffer sb) {
        sb.append("<" + getTagName() + " ");
        String key,value;
        int i = 0;
        for (Enumeration e = parsed.keys();e.hasMoreElements();) {
            key = (String)e.nextElement();
            i++;
            if (key!=TAGNAME) {
                value = getParameter(key);
                sb.append(key+"=\""+value+"\"");
                if (i<parsed.size()) sb.append(" ");
            }
        }
        sb.append(">");
    }

public class HTMLLinkTag extends CompositeTag...
    public String toHTML() {
        StringBuffer sb = new StringBuffer();
        putStartTagInto(sb);
        ...


public class HTMLFormTag extends CompositeTag
    public String toHTML() {
        StringBuffer rawBuffer = new StringBuffer();
        putStartTagInto(rawBuffer);
        ...
```

I perform similar operations to make a consistent way of obtaining HTML from child nodes and from an end tag and all of that work enables me to pull-up one generic `toHTML()` method to the composite:

```
public abstract class CompositeTag extends HTMLTag...
    public String toHTML() {
        StringBuffer htmlContents = new StringBuffer();
        putStartTagInto(htmlContents);
        putChildrenTagsInto(htmlContents);
        putEndTagInto(htmlContents);
        return htmlContents.toString();
    }
```

To complete this part of the refactoing, I'll continue to move child-related methods to the `CompositeTag`, thought I'll spare you the details.

5. The final step involves checking clients of *child-containers* to see if they can now communicate with the child-containers using the `CompositeTag` interface. In this case, there are no such cases in the parser itself, so I am finished with the refactoring.