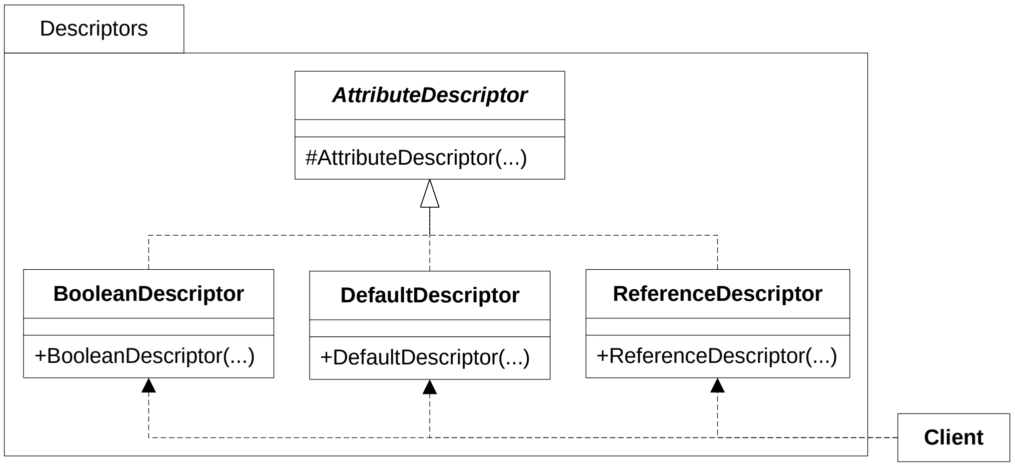
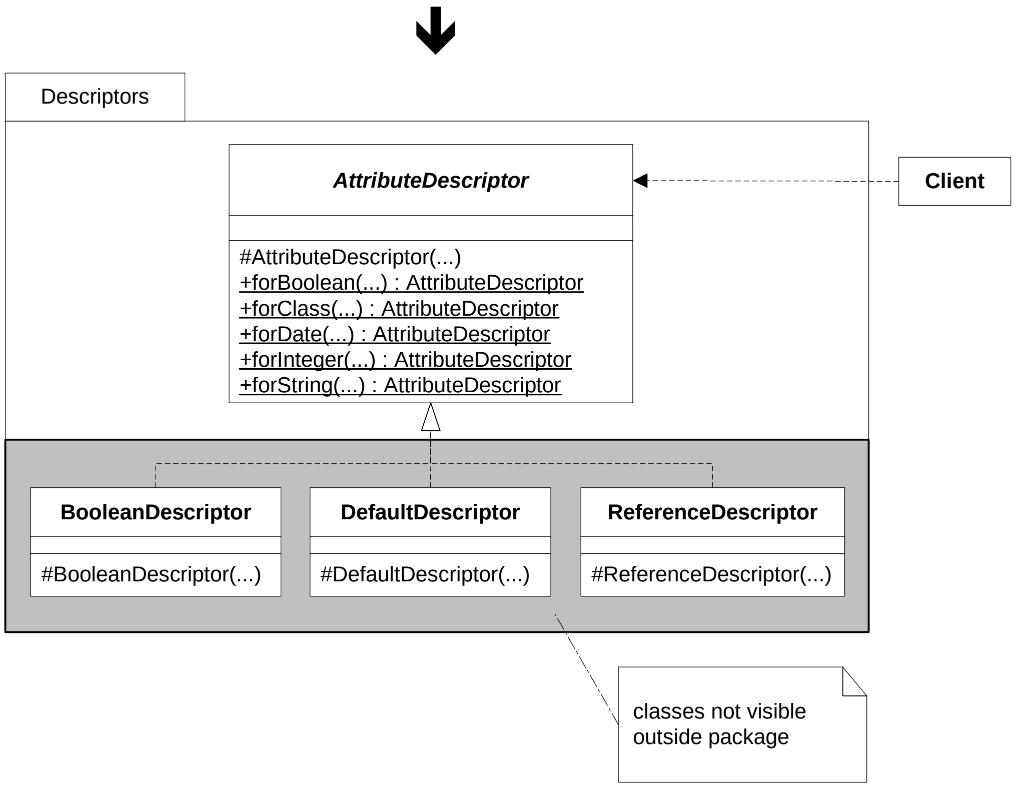
Encapsulate Subclasses with Creation Methods

Frequency: Common

Clients directly instantiate subclasses that
live in one package and implement one interface

*Make the subclass constructors non-public and let clients
create subclass instances using superclass Creation Methods*




Descriptors
AttributeDescriptor
#AttributeDescriptor(...)
+forBoolean(...) : AttributeDescriptor
+forClass(...) : AttributeDescriptor
+forDate(...) : AttributeDescriptor
+forInteger(...) : AttributeDescriptor
+forString(...) : AttributeDescriptor
Client
BooleanDescriptor
#BooleanDescriptor(...)
DefaultDescriptor
#DefaultDescriptor(...)
ReferenceDescriptor
#ReferenceDescriptor(...)
classes not visible outside package

Motivation

A client's ability to directly instantiate subclasses is useful so long as the client needs to know about the very existence of those subclasses. But what if the client doesn't need that knowledge? What if the subclasses lived in one package, implemented one interface and those conditions weren't likely to change? In that case, clients outside the package could not directly instantiate subclasses, but would instead obtain instances via superclass Creation Methods, all of which would use the common interface as a return type.

There are several motivations for doing this. First, it provides a way to rigorously apply the mantra, *separate interface from implementation* [GoF], by ensuring that clients interact with subclasses via their common interface. Second, it provides a way to reduce the "conceptual weight" [Bloch] of a package by hiding classes that don't need to be publicly visible outside their package. And third, it simplifies the construction of available *kinds* of subclass instances by making the set available through intention-revealing Creation Methods.

Despite these good things, some folks have reservations about applying this refactoring. I address and respond to their concerns below:

1. They don't like having a dependency cycle - i.e. having to add new Creation Methods to a superclass just because they create a new subclass or add/modify a subclass constructor. While this sounds like a real problem, in practice I don't find that it is difficult to update the superclass, particularly because by the time I apply this refactoring, there aren't many new subclasses or constructors to add.

2. They don't like mixing Creation Methods with implementation methods on a superclass. I don't have a problem doing this, unless the Creation Methods just make it too hard to see what the superclass does, in which case I would *Extract Creation Class (20)*.

3. They don't like giving a superclass knowledge of its subclasses. Somewhere they learned that this was a bad idea and they usually have some good C++ stories to tell about why, but when I point out that this refactoring happens within the context of one package with subclasses that implement one interface, they usually quiet down.

4. They don't like this refactoring in the context of code that gets handed off as object code, since programmers who must use the object code won't be able to add or modify subclasses or Creation Methods when needed. I'm more sympathetic to this reservation. If extensibility within the package is necessary and users don't have source code, I would not encapsulate the subclasses, but would instead provide a Creation Class for common instances.

The sketch at the start of this refactoring gives you a glimpse of some object-relation mapping code. Before the refactoring was implemented, programmers (including myself) occasionally instantiated the wrong subclass or the right subclass with slightly incorrect arguments (for example, passing in a primitive Java `int`, rather than an `Integer` object). The refactoring reduced bug creation by encapsulated the knowledge about the subclasses and producing a single place to get a variety of well-named subclass instances.

Communication	Duplication	Simplicity
When you expect client code to communicate with subclasses via one inteface, your code needs to communicate this. Public subclass constructors don't help, since they allow clients to couple themselves to subclass types. Communicate your intentions by protecting the subclass constructors, creating subclass instances via superclass Creation Methods and making the return type for the instances be the common interface.	Duplication isn't an issue with this refactoring.	Making subclases publicly visible when you want clients to interact with them via one interface isn't simple: it invites programmers to directly instantiate and couple themselves to subclass types and it communicates that it is ok to extend the public interface of individual subclasses. Simplify this by making it impossible to directly instantiate subclasses and by offering up instances via superclass Creation Methods.

Prerequisites

- Your subclasses have the same public interface as their superclass.

  This is essential because after the refactoring, all client code will interact with subclass instances via their superclass interface.

- Your subclasses reside in the same package.

- Code within the package may be modified, when programmers must extend it.

Mechanics

1. Write an intention-revealing Creation Method on the superclass for a *kind* of instance that a subclass constructor produces. Make the return type for the method be the type of the superclass and make the method's body be a call to the subclass constructor.

2. For the kind of instance chosen, replace all calls to the subclass constructor with calls to the superclass Creation Method.

3. Compile and test.

4. Repeats steps 1 and 2 for any other kinds of instances that may be created by the subclass constructor.

5. Declare the subclass constructor to be non-public (i.e. protected or package-protected).

6. Compile.

7. Repeat the above steps until every subclass constructor is non-public and all available subclass instances may be obtainted via superclass Creation Methods.

Example

1. We begin with a small hierarchy of classes that reside in a package called `descriptors`. The classes assist in the object-relation mapping of database attributes to instance variables:

```
package descripors;

public abstract class AttributeDescriptor {
    protected AttributeDescriptor(…)

public class BooleanDescriptor extends AttributeDescriptor {
    public BooleanDescriptor(…) {
        super(…);
    }

public class DefaultDescriptor extends AttributeDescriptor {
    public DefaultDescriptor(…) {
        super(…);
    }

public class ReferenceDescriptor extends AttributeDescriptor {
    public ReferenceDescriptor(…) {
        super(…);
    }
```

The abstract `AttributeDescriptor` constructor is protected, and the constructors for the three subclasses are public. Let's focus on the `DefaultDescriptor` subclass. The first step is to identify a kind of instance that can be created by the `DefaultDescriptor` constructor. To do that, I look at some client code:

```
protected List createAttributeDescriptors() {
    Vector result = new Vector();
    result.add(new DefaultDescriptor("remoteId", getClass(), Integer.TYPE));
    result.add(new DefaultDescriptor("createdDate", getClass(), Date.class));
    result.add(new DefaultDescriptor("lastChangedDate", getClass(), Date.class));
    result.add(new ReferenceDescriptor("createdBy", getClass(), User.class,
        RemoteUser.class));
    result.add(new ReferenceDescriptor("lastChangedBy", getClass(), User.class,
        RemoteUser.class));
    result.add(new DefaultDescriptor("optimisticLockVersion", getClass(), Integer.TYPE));
    return result;
}
```

Here I see that `DefaultDescriptor` is being used to represent mappings for Integers and Dates. It may also be used to map other types, but I must focus on one kind of instance at a time. So I decide to write a Creation Method to produce attribute descriptors for Integers:

```
public abstract class AttributeDescriptor {
    public static AttributeDescriptor forInteger(...) {
        return new DefaultDescriptor(...);
    }
```

I make the return type for the Creation Method an `AttributeDescriptor` because I want clients to interact with all `AttributeDescriptor` subclasses via the `AttributeDescriptor` interface and because I want to hide the very existence of `AttributeDescriptor` subclasses from anyone outside the `descriptors` package.

If you do test-first programming, you would begin this refactoring by writing a test to obtain the `AttributeDescriptor` instance you want from the superclass Creation Method.

2. Now client calls to create an Integer version of a `DefaultDescriptor` must be replaced with calls to the superclass Creation Method:

```
protected List createAttributeDescriptors() {
    List result = new ArrayList();
    result.add(AttributeDescriptor.forInteger("remoteId", getClass()));
    result.add(new DefaultDescriptor("createdDate", getClass(), Date.class));
    result.add(new DefaultDescriptor("lastChangedDate", getClass(), Date.class));
    result.add(new ReferenceDescriptor("createdBy", getClass(), User.class,
        RemoteUser.class));
    result.add(new ReferenceDescriptor("lastChangedBy", getClass(), User.class,
        RemoteUser.class));
    result.add(AttributeDescriptor.forInteger("optimisticLockVersion", getClass()));
    return result;
}
```

3. I compile and test that the new code works.

4. Now I continue to write Creation Methods for the remaining kinds of instances that the `DefaultDescriptor` constructor can create. This leads to 2 more Creation Methods:

```
public abstract class AttributeDescriptor {
    public static AttributeDescriptor forInteger(...) {
        return new DefaultDescriptor(...);
    }
    public static AttributeDescriptor forDate(...) {
        return new DefaultDescriptor(...);
    }
    public static AttributeDescriptor forString(...) {
        return new DefaultDescriptor(...);
    }
```

5. I now declare the DefaultDescriptor constructor protected:

```
public class DefaultDescriptor extends AttributeDescriptor {
    protected DefaultDescriptor(…) {
        super(…);
    }
```

6. I compile and everything goes according to plan.

7. Now I repeat the above steps for the other `AttributeDescriptor` subclasses. When I'm done, the new code:

- gives access to `AttributeDescriptor` subclasses via their superclass
- ensures that clients obtain subclass instances via the `AttributeDescriptor` interface
- prevents clients from directly instantiating `AttributeDescriptor` subclasses
- communicates to other programmers that `AttributreDescriptor` subclasses are not meant to be public – the convention is to offer up access to them via the superclass and a common interface.