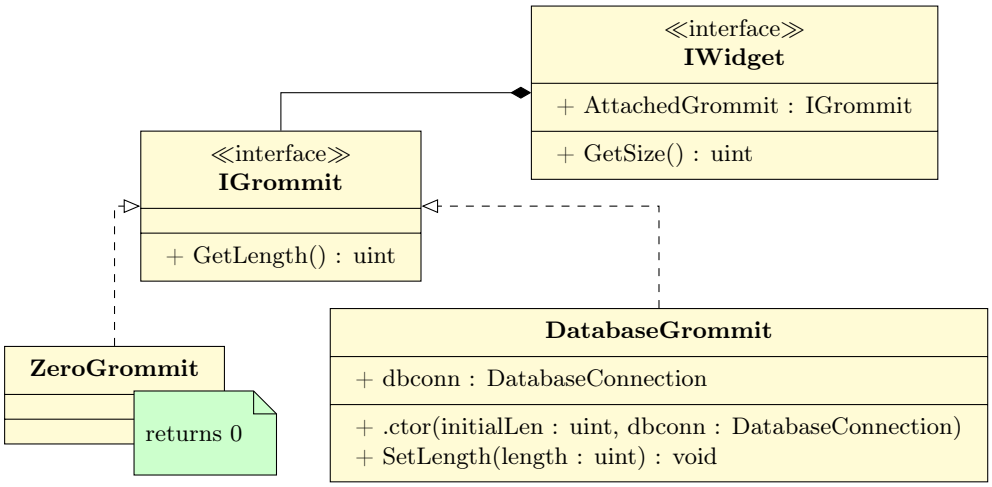
Downcasting considered harmful

Downcasting (casting a base type to a derived type) is bad news. The cast's safety relies on human knowledge of the code, not compile-time guarantees. It is better to give as much information to the compiler as possible, so that it can check casts before they become a problem.

1 Widget and grommit classes

Presented here are a couple of grommit types, and an interface for the widgets which use them.



A very basic widget might be:

Listing 1: Widget1.cs

```cs
public class Widget1 : IWidget {
    public IGrommit AttachedGrommit { get; }

    public Widget1() {
        AttachedGrommit = new DatabaseGrommit(0, DatabasePool.Get());
    }

    public uint GetSize() {
        return AttachedGrommit.GetLength();
    }
}
```

2 Grommit-modifying widget

Now suppose we need a kind of widget which can modify its grommit's length. This is no problem, since we're using a `DatabaseGrommit`, and that already exposes a means of modifying its length.

Unfortunately the class was written to use a downcast to access the `IGrommit` in the `AttachedGrommit` property as a `DatabaseGrommit`:

Listing 2: Widget2a.cs

```
public class Widget2 : IWidget {
    public IGrommit AttachedGrommit { get; }

    public Widget2() {
        AttachedGrommit = new DatabaseGrommit(0, DatabasePool.Get());
    }

    public uint GetSize() {
        return AttachedGrommit.GetLength();
    }

    // ... much further down the file ...

    public void ApplyPlumbus(IPlumbus plumbus) {
        uint newSize = plumbus.Radius;
        ((DatabaseGrommit)AttachedGrommit).SetLength(newSize);
        // ^-- downcast! --^
    }
}
```

This is bad because we've thrown away important type information that the compiler could have used to benefit us. First we store the `DatabaseGrommit` in a slot of type `IGrommit`. Later, we pull the value out again and manually (uh-oh!) reapply the `DatabaseGrommit` type. Since that cast will only be checked at runtime, it is now possible for us to make mistakes that the compiler would otherwise have caught easily.

3 Everything goes horribly wrong

A well-meaning developer notices that `Widget1`'s grommit only ever has length zero, so it can be replaced with a `ZeroGrommit` to save an expensive database query. Since `Widget2` appears to be doing the same thing, they also replace that grommit in the same way. `Widget2` now looks like this:

Listing 3: Widget2b.cs

```
public class Widget2 : IWidget {
    public IGrommit AttachedGrommit { get; }

    public Widget2() {
        AttachedGrommit = new ZeroGrommit(); // <-- modified line
    }

    public uint GetSize() {
        return AttachedGrommit.GetLength();
    }

    // ... much further down the file ...

    public void ApplyPlumbus(IPlumbus plumbus) {
        uint newSize = plumbus.Radius;
        ((DatabaseGrommit)AttachedGrommit).SetLength(newSize);
    }
}
```

This will compile quite happily, but will fail at runtime in `ApplyPlumbus`. If this is a rarely-used and untested method, the error might not be noticed for some time.

4 Correcting the problem

How do we fix this? Firstly, notice that the contract of `IWidget` forces the return type of `AttachedGrommit` to be `IGrommit`, so we can't simply change the type of this property. However, we *can* implement the property with an explict backing field (instead of the implicit one we have currently). We can freely set the type of that field to `DatabaseGrommit`, meaning we keep the type information around for the compiler. Meanwhile,

`AttachedGrommit.get` is resolved by an implicit upcast from our `DatabaseGrommit` to the return type of `IGrommit`. Upcasts are always safe.

Here's the improved version of `Widget2`:

Listing 4: Widget2c.cs

```
1  public class Widget2 : IWidget {
2      private readonly DatabaseGrommit attachedGrommit;
3      public IGrommit AttachedGrommit {
4          get { return attachedGrommit; }
5      }
6
7      public Widget2() {
8          attachedGrommit = new DatabaseGrommit(0, DatabasePool.Get());
9      }
10
11     public uint GetSize() {
12         return AttachedGrommit.GetLength();
13     }
14
15     // ... much further down the file ...
16
17     public void ApplyPlumbus(IPlumbus plumbus) {
18         uint newSize = plumbus.Radius;
19         attachedGrommit.SetLength(newSize);
20     }
21 }
```

Notice that if our well-meaning developer comes along and tries to replace the constructor's `DatabaseGrommit` with a `ZeroGrommit`, the code will fail to compile due to the invalid assignment on line 8. If they persist in their folly and change the type of the `attachedGrommit` backing field, the compilation error in `ApplyPlumbus` will reveal what the original developer was trying to do.

Appendices

A Supporting types

Listing 5: IWidget.cs

```
public interface IWidget {
    IGrommit AttachedGrommit { get; }

    uint GetSize();
}
```

Listing 6: IGrommit.cs

```
public interface IGrommit {
    uint GetLength();
}
```

Listing 7: ZeroGrommit.cs

```
public class ZeroGrommit : IGrommit {
    public uint GetLength() {
        return 0;
    }
}
```

Listing 8: DatabaseGrommit.cs

```
public class DatabaseGrommit : IGrommit {
    private DatabaseConnection dbconn;
    private System.Guid ident;

    public DatabaseGrommit(uint initialLen, DatabaseConnection dbconn) {
        this.dbconn = dbconn;
        this.ident = dbconn.CreateGrommit(initialLen);
    }

    public uint GetLength() {
        return dbconn.QueryGrommitLength(ident);
    }

    public void SetLength(uint length) {
        dbconn.ModifyGrommitLength(ident, length);
    }
}
```

Listing 9: IPlumbus.cs

```
public interface IPlumbus {
    uint Radius { get; }
}
```