The Smalltalk Meta Object Protocol  
(part II)

In this part of the exercise we will explore a few applications of Smalltalk's Meta Object Protocol.

**Application 1: Visitor Design Pattern**

The first application is to have a simple tool to automatically generate an implementation of the Visitor design pattern for a given class hierarchy.

Implement a class **AbstractVisitor**. Make it possible to send a message to this class using as argument at least the root class of a certain class hierarchy. In response to this message the classes in that hierarchy should be given appropriate **acceptVisitor:** methods and a new visitor subclass should be created that has the necessary template visitation methods. This is illustrated in the figure below:
As a sidetrack exploration consider for a moment that the Visitor design pattern is also interesting because at its lowest level it makes use of a common idiom to extend polymorphism in a way. This idiom also occurs in the methods in the protocol 'arithmetic' in subclasses of Number in Smalltalk. Take a look at these methods. Describe the idiom you see here and in the Visitor design pattern. What problem does it solve? What name would you give it?

Application 2: Message Interception

We want to have a system for intercepting messages on objects so that we can trace these messages, check for pre- and post-conditions etc.

Step 1: The LoggingWrapper

Implement a class LoggingWrapper which provides the following:

- A way of installing a new wrapper on another object and having all references that point to the object now point to the wrapper (Hint: there's a message become: that can be helpful here.)
- A way of uninstalling the wrapper of an object.
- A message interception so that whenever a message gets sent to the object via the wrapper, the wrapper should first log the message to the Transcript and then forward the message to the object.

Step 2: The Pre- and Post-conditions Wrapper

Implement a class PrePostWrapper which like the LoggingWrapper can be installed, uninstalled and intercepts messages on the wrapped object. Refactor if possible to get a small wrapper framework.

When intercepting a message, the PrePostWrapper should first check a number of pre-conditions before forwarding the message, after the message has been executed a number of post-conditions should be checked. These
pre- and post-conditions can be specified in the form of Smalltalk blocks. It should thus be possible to do something like:

```smalltalk
temp := PrePostWrapper newOn: anObject
temp controlMessage:#foo withPre: [ ...condition... ]
                  withPost: [ ...condition... ].
temp controlMessage:#bar withPre: [ ...condition... ]
                  withPost: [ ...condition... ].
```

When a pre- or post-condition fails, an exception should be raised.

Ensure that the post-conditions are also checked even when forwarding the message to the object results in some exception. (Hint: take a look at the BlockClosure class)

**Step 4: Evaluation**

2. What problems can you identify with pre- and post-conditions implemented using these wrappers? Are all message sends intercepted? What about performance?

**Step 5: Method Wrappers**

Get the Method Wrappers package from the POOL Store. Take a look at its implementation and some of the included examples.

Use Method Wrappers to (re)implement the **PrePostWrapper**.

3. Do the same problems as with the previous implementation still arise? Why (not)?
4. Compare the performance of your own wrapper implementation with the Method Wrappers implementation by timing the time needed to send a large number of messages to a wrapped object. What can you conclude? What is your explanation?

5. Can you make something like Method Wrappers in Java? Why (not)?