Modular Exceptions

A bachelor thesis project
by: Patrick Indermühle
supervised by: Prof. Dr. Oscar Nierstrasz

Conventional exception handling

- Done through try-catch blocks
- *aFunction(){*

```
    Try{
        //Do stuff here
    } catch(Exception e){
        //Handle exceptions here
    }
}
```

Disadvantages

- Clutters the code
- Often not reusable
- Requires manual implementation

Method wrapping

- Many languages support treating methods as objects
- Using a wrapper for exception handling
- *Class Wrapper{*

```
    execute(){
        Try{
            wrappedMethod.execute()
        }Catch(Exception e){
            //Handle exceptions here
        }
} }
```

Advantages

- Can keep exception handling outside of code
- Highly reusable/modular
- Much faster to implement

Additional advantages

- Can also be used to handle non exception related things such as..
- …preventing the execution of a method in case…
  - …it would create an invalid object
  - …it would change data to an invalid state
  - …its parameters are null

Implementing Modular Exceptions

- Multiple approaches
- Deeper look at wrapper objects in Smalltalk
- In Smalltalk all methods are objects
- Any objects can serve as a method
- Must implement run:with:in: among others
- Can replace a method with an object and keep the old implementation
- Result: Complete control over execution of the old method

Implementation of the wrapper

```
Object subclass: #ModularWrapper
    instanceVariableNames: 'wrappedMethod wrappedClass selector'
    classVariableNames: ''
    package: 'ModularExceptionPackage'

install
    wrappedMethod := wrappedClass lookupSelector: selector.
    wrappedClass addSelector: selector withMethod: self

uninstall
    wrappedMethod methodClass methodDictionary
    at: wrappedMethod selector
    put: wrappedMethod.
```

Implementation of the wrapper

```
doesNotUnderstand: aMessage
    ^wrappedMethod perform: aMessage selector withArguments: aMessage arguments

selector: aSelector
    selector := aSelector

selector
    ^selector

wrappedClass: aClass
    wrappedClass := aClass.

run: aSelector with: arguments in: aReceiver
    self inform: 'Modular wrapper was triggered'.
    ^self callOldMethodOn: aReceiver withArgs: arguments

callOldMethodOn: aReceiver withArgs: arguments
    ^aReceiver withArgs: arguments executeMethod: wrappedMethod
```

Implementation of the wrapper

- Class method:

```
installOn: aClass selector: aSelector
    |newInstance|
    newInstance := self new.
    newInstance selector: aSelector.
    newInstance wrappedClass: aClass.
    newInstance install.
    ^newInstance.
```

- Usage:

```
wrapper := ModularWrapper installOn: someClass selector: #someMethod.
```

Result

- Wrapper object takes the place of the old method
- Wrapper is triggered when method is called
- Old method is saved in the wrapper
- Can use the old method if needed or wanted
- Exception handling can be done in the wrapper while old method stays the same

Reflectivity

- Smalltalk is a reflective language
- Reflectivity = can recompile methods at runtime
- This allows method wrapping at runtime
- Can implement exception handling when an exception occurs

Possible future

- Implementation of Modular Exceptions in Java
  - Can be done through annotations
  - Or aspect oriented programming
- Tools to automatically add Modular Exceptions
- Tools to analyze currently deployed Modular Exceptions

The End