Treating Interfaces as Components

Manuel Schwarzinger
schwarzinger@racon-linz.at

Joachim Fröhlich
joachim.froehlich@acm.org
The Problem

- The Problem is not new!
  - The old problems and dreams are still with us.
  - Only the Words are new.

  *D. Parnas about Component-Oriented Design*

- Component Benefits have been known for years
  - exchangeable software units with clear interfaces
  - parallel development
  - ...

- .NET enables COP but does not enforce it
- Interplay between OOP and COP is still not clear
- COP requires interface-based programming
- Programmers have just mastered classes
The Goal

Components that

• are easy to use and implement
  … even for highly specialized application domains

• offer sharp interfaces
  … also with object-oriented techniques, but without classes

• adhere to the information hiding principle
  … of course

• decouple functional components completely
  … well, bind them well-defined

• allow (re)configurations
  … after development, before and during runtime
Treating Interfaces as Components

The Goal (2)

We have a dream of

- sharp component (interface) specifications
- systematic unit, component and integration tests
- separate component implementations
- flexible configurations of system families
- hiding specific, platform-dependent component models

⇒ We look for stability to be agile!
The Idea (1)

Interfaces carry software architectures, not components
- stable interfaces lead to stable architectures
- … still not new, but still newsworthy

Connectors are
Components with only systems of semantically related interfaces

… well, with almost only interfaces
Connectors may bee seen as clutches with
- driving parts: functional \textit{client} components
- driven parts: functional \textit{provider} components
Connectors may offer services
… like antishock protection and gear changing
… i. e. logging, profiling, protocol checks …
The Idea (2)

Schematic representation of a connector with a system of call and callback interfaces

Characteristics:
- functional components do not depend on each other
- functional components depend on connectors
- connectors do not depend on functional components
The Idea (3)

Systems full of Connectors

components may be clients and providers
A Solution

1. Design the Interfaces

2. Compile functional components and interfaces independently

   ```
   csc /out:Connector.dll /t:library ...
   csc /out:Provider.dll /t:library /r: Connector.dll
   csc /out:Client.exe /t:exe /r:Connector.dll ...
   ```

3. Plug together functional components
   … before runtime e.g. simply with app.config

   ```
   <configuration>
   <appsettings>
     <add key="ConnectorSubsystemX" value ="Provider.dll"/>
   ...
   </appsettings>
   </configuration>
   ```

4. Run the application
   … without changing a line of code
A Solution (2)

Tasks of a connector:

- Combine semantically coherent interfaces
- Establish the connection between a client and a provider

… without violating the information hiding principle

A Connector has at least one class

… the *provider-independent connector class*

Each provider has exactly one

… *provider-specific connector (sub)class*

… per supported connector

Exactly one object per provider-specific connector class

Clients use this object as a kind of factory
A Solution (3)

Connectors may factor out nonfunctional services
... from functional components.

*Light* connectors:
- only contain declarations of application-specific functions
  ... organized in interfaces

*Heavy* connectors:
- wrap interfaces in proxy classes
- hook in services like logging, profiling ...
- wrap call interfaces on the way *out*
- wrap callback interfaces on the way *in*

Connectors that may carry several providers: *Multitions*
... in contrast to *Singletons* (light and heavy) discussed so far
Review

We have been using connectors

- for a generic log analyzer
- in several student projects: mobile cash box, p2p chat ...

Our experience with others:

- heavy usage of OO hinders understanding of connectors
- need about 2 common programming sessions for confidence

Results compared to the goals

+ Ease of use
+ Sharp interfaces, information hiding principle
+ Complete separation of functional components
+ Hiding platform specifics
-/+ Dynamic exchange of components
Outlook

Focus: Experiments with Connectors of various expansion stages
• Load and unload providers dynamically
  … technically it works with Assemblies in AppDomains
  … and replacing interfaces with pure abstract classes
  … but what to do with stateful component instance?
• Expand on mechanisms for protocol checking
• Choose providers dynamically
  … according to an application/client-specific strategy
• Open programs for dynamic configuration from outside
• Monitor functional components
• …
We do not pretend to have achieved perfection – but we do have a system – and it works.

*Michael Rennie*

in „The Day The Earth Stood Still“ 1951