# Adapters / WCF Services Design Guideline

## Observations / Current Situation

Activities were used as the building blocks for composing functionality, which lead to:

Activity

WorkflowService

Adapter

* WorkflowFoundation being a mandatory element for implementing functionality
* Too much functionality is actually coded in Xaml just to avoid going back to c# world on a lower level again or because other low-level activities had to be reused (vicious circle)
* Even for normal WCF services, VS’s source code navigation features end right below the interface level
  + Mainly a tooling issue, however it is pretty sure that this will not be fixed by Microsoft
  + Applies as well for refactoring an existing xaml base
* Poor ‘Separation of Concerns’
  + To avoid navigating multiple xaml files, too much functionality was packed into a single xaml file
  + Fixing this is hard because of the missing refactoring tools
* Pickbranches to implement wide Contract of a WorkflowService, clutter the screen and result in cumbersome navigation in the editor

## Design Goals

1. Reduce Complexity
2. Improve "Separation of Concern"
3. Improve Functional Cohesion
4. Testability
   1. Allow tests to be simple
   2. Do not require spinning up massive infrastructure for running low-level tests
5. Improve Developer Productivity
   1. Minimize technology breaks
6. Use DependencyInjection as a means to realize above points
   1. Avoid making DI a key player or complexity driver

## DependencyInjection

* we start with using MEF as our DI framework (yes, MEF is not officially DI)
* multiple other alternatives available that are superior with regards to:
  + performance
  + usability
  + features (lifecycle scopes)
* isolate DI framework from the rest of the code base so that it can be replaced with reasonable impact
  + do not have MEF-specific types injected or MEF-specific helper classes  
    (usage of MEF Attributes are OK because of their optional character)

### Single “Composition Root” per Application

One principle when reading about DependencyInjection is to have only a single “CompositionRoot” (see <http://blog.ploeh.dk/2011/07/28/CompositionRoot/> for definition of CompositionRoot).

It is the one place where we wire things together and lifecycle is defined.

* in WF the term application maps to Service/Adapter
  + => we do not share instances across different service types
  + => service registrations can be shared

WCF

Svc 1

WCF

Svc 2

Composition

Container

Composition

Container

Container

Configuration

* use “Ambient Context” for Cross-Cutting Concerns
  + avoid that cross-cutting concerns bloat constructors
  + identify types that qualify as cross-cuttings

### Lifecycle of Stateful Components

* for objects that are “expensive” to create like client proxies to other WCF services (e.g. DRS) for normal WCF services following lifestyles make sense
  + call
  + session
  + singleton (thread safety required)
* use Decorator Pattern for disposable types
* Lifecycle Scopes (per call, per session)

#### MEF and transient disposable Objects

* MEF feels responsible for objects implementing IDisposable even if they are registered as nonshared
* MEF therefore holds a reference to them until the Container is disposed
* This might cause an unanticipated resource leak if the container is long-living and a lot of those transient instance are created

### DependencyInjection with Workflow

* WorkflowFoundation does not allow creating activities through external parties like DI containers (maybe Arpad has an idea)
* This leaves us with 2 possible approaches:
  + Use “property injection” to inject dependencies
  + Use WorkflowFoundations Extension concept

**Goals:**

* If a simple solution is not possible, revert to Extensions approach
* A major investment into developing a DI-based solution or adapting the existing code-base is not intended

**Reference Information**

<http://www.planetgeek.ch/2010/12/06/dependency-injection-with-windows-workflow-foundation-4-introduction/>

### Cohesion

Definition in Wikipedia: <https://en.wikipedia.org/wiki/Cohesion_(computer_science)>

In Workflow functionality is spread over a huge number of `Activities`. An `Activity` corresponds to a method on an interface or operation on a service. Activities that belong semantically together would have to be grouped through the namespace they live in.

However, there is no easy way to distinguish between high-level and low-level Activities as well as appropriate context in which a Activity can be used.

**Functional Cohesion over Logical Cohesion**

Activities are grouped along logical boundaries (accessing db, ...) rather than functionality.

There are a lot of similarly named `Activities` where a consumer has a hard time to find out what the `Activity` is doing under the hood and

in which context (e.g. Torus, Cube or via) the `Activity` are intended to be used.

### Testability

## Normal vs. long running Operations

Timeouts on client side can be configured per ServiceContract and can be adapted per client proxy instance or on the ServiceHost per binding. This means that all operations on a given ServiceContract must be similar regarding their maximum answering time (return a result / error).

Mixing short and long-running operations on the same contract requires cranking up the timeouts for all operations, which has additional negative impact:

* Client logic must be explicitly adapted as well (e.g when waiting for calls coming back using Task.WhenAll() and having an overall timeout specified)
* Time until a connection problem is detected increases

It is better to make this different timing behavior explicit at the interface by modelling it differently.

specified per client proxy instance need to be "consistent" with service-side timeouts:

* clients should not timeout, because there is no background information we can present to the user
* if server timeout, we at least send info which underlying operation did not complete

If there are operations that cannot complete with the typical 30s (WCF default) or are known to have a large variance for their response times,

The interface to the operation needs to modelled differently:

Task<RequestHandle> rqh = client.BeginOpen();

client.SubscribeToChangeNotifications(rqh);

//or

//RequestHandle already allows to get change notifications / completion state

## Connection Reestablishment

* callback channels are also reestablished
* risk of missed callbacks! Resync necessary
  + more details required how to do Resync
  + Queries return most recent information,
  + events are executed async and lag behind

## Flowing Transactions between Services

* avoid between client and server