Our goal is simply to truly support “programming against abstraction”, with specializations (and as much covariance as possible), more specifically in the context of database interaction.

# Abstraction: Contract and Ambient Contract

The word “Contract” refers to an interface or an abstract base class: something that states what it does without any clue on the actual code that does it (the implementation).

An “Ambient Contract” is a Contract that is unique in a Context. One can consider it as a “pseudo-singleton”: only one implementation Type can exist in a Context for an Ambient Contract. A simple way to declare an Ambient Contract is to support the marker interface IAmbientContract.

An Ambient Contract can be specialized:

public interface ICommunicationService : IAmbientContract

{

void SendMail( MailInfo m );

}

public interface ICommunicationPlusService : ICommunicationService

{

void SendViaSMS( MailInfo m );

}

In a given Context, the final implementation of both services is guaranteed to be the same object: any sub-part of the System that depends on ICommunicationService will work with the most specialized implementation.

# Dependencies

Implementations depends on Abstractions.

# Lifetime

How a Contract is implemented SHOULD have not impact the Contract itself. Unfortunately, as Joel Spolsky explained in 2002 in "[The Law of Leaky Abstractions](http://www.joelonsoftware.com/articles/LeakyAbstractions.html)", this is easier said than done.

One classical cause of leaky abstraction is related to objects’ lifetime management (see for instance <http://stackoverflow.com/questions/2634675/ioc-containers-and-idisposable>).

Lifetime control is generally

# Stateless vs. Stateful Contracts.

Can a Contract impacts Lifetime? If yes, is it possible to exhibit one or more conditions that will constrain its implementation lifetime?

A “pure service” (think IMailService) does not exhibit state: it offers a collection of “behaviors” through functions. Such Contract is stateless in the sense that it does not constrain its implementations to keep any state in them (some implementations may hold a state, but this is an implementation issue). These kind of Contracts are often implemented by singleton whenever possible for performance/optimization reasons, but can perfectly be implemented by transient objects (as façades on actual mailer infrastructure).

If a Contract exhibits a state, it seems logical object that dependends

Let’s take this ICurrentUser Contract that seems to be “bound to a state”:

public interface ICurrentUser

{

string UserName { get; }

void Login( string userName );

void Logout();

}

Nothing tells us how it has to be implemented: it actually can be implemented either as a stateful object or as a stateless façade based on other services, repositories or ambient contexts.

**Important:** The fact that the information exposed by a Contract can change at any time has nothing to do in this discussion. I consider this as being a “transactional” concern that is not related to the structure of the System (think “repeatable read”).

Does it mean that stateless/full is a useless concept when thinking in terms of “Contracts”? Is it an implementation concern only?

Actually not.

We will consider a Contract as “stateful” if it exposes a way to publish its state change[[1]](#footnote-1).

|  |  |
| --- | --- |
| Stateless Contract | Stateful Contract |
| public interface ICurrentUser  {  string UserName { get; }  void Login( string userName );  void Logout();  } | public interface ICurrentUser  {  string UserName { get; }  void Login( string userName );  void Logout();  event EventHandler UserNameChanged;  } |

A “Stateful Contract” is a Contract that exposes at least one event.

A “Stateless Contract” does not expose any event.

A “Stateful Contract” implementation must have a greater life time than any of the implementation that depend on it.

A “Stateless Contract” can be considered as having absolutely no impact on lifetime constraints: a Singleton implementation CAN depend on a Transient one (I know this may seem rather counterintuitive to almost everybody).

To catch the point, simply consider that a Proxy is generated. This Proxy’s lifetime is the same as the dependent object (a Singleton for instance) and each of its methods and properties obtain the actual implementation, relays the call, and release it (or not depending on the container’s instance management strategy).

This is simply an adapter, a lifetime adapter. It can be costly of course and

🡺 An abstraction does not carry per se any lifetime constraint.

Strong requirement: class B { void Construct( IA a ) {} }

This carries two constraints:

LT(Impl(IA)) >= LT(B)

Init(Impl(IA)) <= Init(B)

Some scopes can be subordinated to other ones:

Session => Application

Request => Application

Application => Machine

There is no systematic subordination between Session and Request since Session may not exist.

Some scopes are totally independent from each other. A User scope for instance binds Contracts to the current user identity. A Thread scope binds Contracts to the current thread.

1. Here .Net events are used, but any observer/observable or publisher/subscriber pattern between the two instances can be considered. [↑](#footnote-ref-1)