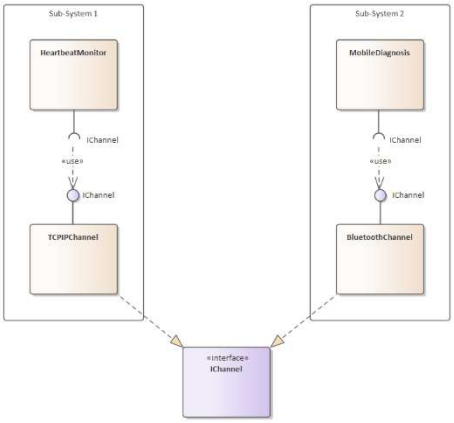
When dependency injection pattern can turn into anti-pattern

Since the beginning of the 2000's dependency injection as one of the inversion of control methodologies had become more and more popular. Already 2004, Martin Fowler wrote a very good article that putted it in a broader context of inversion of control methodologies and explained strengths and weaknesses in various scenarios (Fowler, 2004). Over the years several frameworks had been developed that providing container-based dependency injection solutions. Examples therefor are Microsoft Managed Extensibility Framework or autofac.

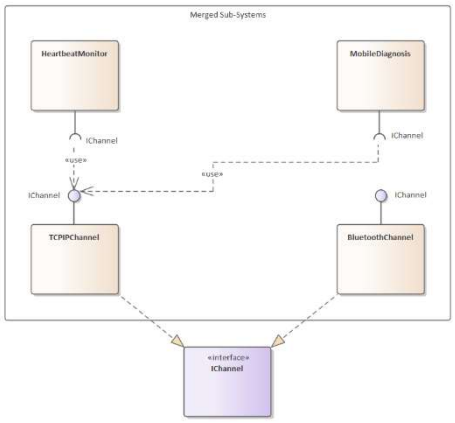
All the frameworks are less or more powerful and it is easy to wire components to build up more complex sub-systems. As far as the sub-system is encapsulated everything works well and nothing is to be said against this powerful approach.

Problems will rise when the enclosing application becomes more complex and developers want to build up the whole system at once. The typical approach here is to equip each sub-system with a specific builder method, that register its components in one single dependency injection container. This is quite seductive because the sub-system is already prepared for that. But at this point the problems come up.

Good developers equipping the components with abstraction in the form of interfaces and often the interfaces are well abstracted and can also be used in different contexts. But what happens when we now put all these things together. One component can use a communication interface for TCP/IP based communication another one uses Bluetooth. Both components are well designed and using the same abstract communication interface. That works well in their sub-system because the right component is registered for this sub-system. The following figure shows that situation.

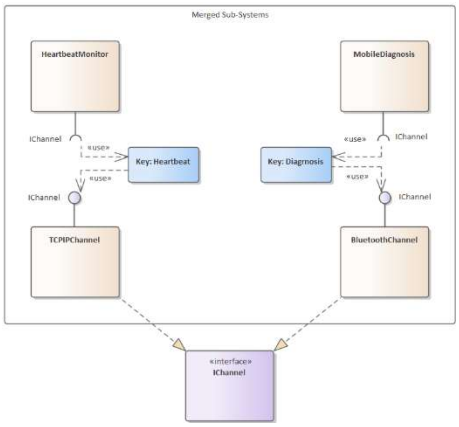


But what happens when we put both sub-systems in one single container? The following figure shows that situation.



Upps, what happens now? MobileDiagnosis is wired to TCPIPChannel. This was not our intention. The reason therefor is that the dependency container can only resolve one interface implementation. Depending on the framework and the settings that we have choosen for the framework, the already existing registration is overwritten or even few better the framework rejected the registration. Remember that both previously defined sub-systems are registered in one single dependency injection container.

The obvious solution for the frameworks that I know, is to use keyed registration. That looks like in the following figure.



This works for now. But if we try to merge further sub-systems based on a hierarchical approach I would say “Keep the finger crossed!”, that we have only one Heartbeat or Diagnosis key.

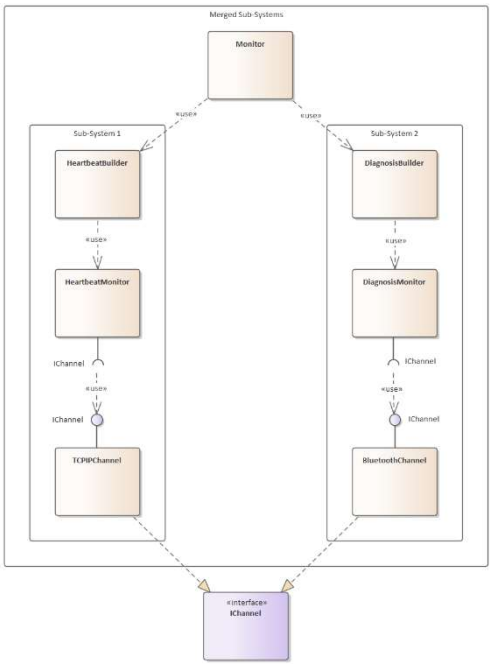
That example does not say that dependency injection containers are devil stuff and should not be used - in the opposite. But as with every tool you must use it in the right context. Otherwise we turn dependency injection into an anti-pattern. Use a hammer for a nail and not for a screw. In the example above I would say don’t use dependency injection containers to merge context / lifetime scopes. For our screw we can use a screwdriver. But what is the way to bring different contexts / lifetime scopes together?

We don’t need to reinvent the wheel. The builder design pattern already introduced by Gamma et. al., is exactly what we need and what dependency injection containers use too. But we need to adapt it to the additional requirement of merging contexts / lifetime scopes. How could it look like?

1. Put your dependency injection container of your sub-system into its own builder class.
2. The builder class provides a set of parameters that represent the variable elements in the dependency injection container of the encapsulated sub-system.

3. If the builder method is called it injects the provided parameter values into the internal dependency injection container and requests the class from the dependency injection container.
4. Wire the components of the sub-ordinated dependency injection container with the build method of the builder instance.

The following figure shows that approach.



Conclusion

Inversion of control and dependency injection are quite powerful techniques. They can be used to create complex object structures in an easy, adaptable and extensible way. But they have restricted capabilities in regards to the management of sub-lifetime scopes, in particular with multiple instances of services providing the same interface. With the introduction of an additional layer, based on the builder design pattern, between the different sub-lifetime scopes this limitation can be easily overcome.

An example of the above scenarios can be found here:
https://github.com/HeikoOehme69/Examples/tree/main/DI-Anti-Pattern-1

Fowler, M. (23. 01 2004). *Inversion of Control Containers and the Dependency Injection pattern*. Von martinFowler.com: https://martinfowler.com/articles/injection.html abgerufen