Discussion of Dependency Inversion, Dependency Injection, Dependency Lookup, Configurable Receiver and Inversion of Control

Alistair Cockburn
Humans and Technology Technical Report 2023.02 (v4d, 2023-06-02)

"Dependency" refers ambiguously to a compile-time or run-time dependency. "Inverting" something says to do the *"not"* of some other, unnamed thing. As a consequence, dependency inversion, dependency injection, dependency lookup and inversion of control are often mixed together in incorrect ways.

Let's work through them all.

1. Dependency Inversion Principle (compile-time topic)

The *Dependency Inversion Principle* refers to compile-time dependencies between two elements. "Inversion" in the name refers to the formerly dominant hierarchical decomposition techniques in which abstract decisions were higher up in the hierarchy and depended on the concrete implementations that were lower down. The principle says: "Do the opposite of that."

Element A has a *compile-time* dependency on element B if it needs B to be present for its (A's) compilation. If B's implementation changes, A has to be recompiled.

Noting *dependency* here as a compile-time dependency, the dependency inversion principle says:

> The Dependency Inversion Principle:
>
> A. HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS.
>
> B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.

[https://web.archive.org/web/20110714224327/http://www.objectmentor.com/resources/articles/dip.pdf, https://en.wikipedia.org/wiki/Dependency_inversion_principle]

From Bob Martin's original article:

> *One might question why I use the word "inversion". Frankly, it is because more traditional software development methods, such as Structured Analysis and Design, tend to create software structures in which high level modules depend upon low level modules, and in which abstractions depend upon details. Indeed, one of the goals of these methods is to define the subprogram hierarchy that describes how the high level modules make calls to the low level modules.*

Note here the date of the article: 1996. People were still predominantly using structured analysis and structured design. In those, one started with an abstract statement of policy at a higher level in the hierarchy and detailed that down to some specific implementation at a lower level in the hierarchy. "Higher" and "lower" levels made sense to talk about, and "more abstract" and "less abstract" similarly.

This changed with OO languages. As there is no hierarchical decomposition, there is no obvious "higher" and "lower" level in an OO design. What remain is the thought that there is a policy decision, like "notify people when the situation changes", and various execution possibilities, like telephones, pagers, emails, etc. Although it is not as obvious as it was before, we can apply the thought, "There are various ways to do that" to get to what Bob Martin refers to as "lower level."

Thus, if

there are various ways to do that

then apply this design idea.

In fact, you will apply the *Configurable Receiver* pattern: Define the policy object's required interface, add an instance variable to hold the receiver at run time, design the configurator to provide the receiver to use at run time, and go.

In his example of a button telling a lamp to turn on and off, is a button higher-level than a lamp? Hardly. Is the button setting a policy decision that lamps implement? Not really. Here, he considers the button being used for many devices, such as a hot tub or a radio, so "there are various things to turn on and off" becomes the direction of the principle.

Leaving aside higher and lower levels, if we want the button to control various things, then we might ignore the phrase "dependency inversion", but focus on the key recommendation: "both depend upon abstractions."

Also, programming language matters. Bob Martin writes about C++:

> *The definition of a class, in the .h module, contains declarations of all the member functions and member variables of the class. This information goes beyond simple interface. All the utility functions and private variables needed by the class are also declared in the .h module.*

For that reason, he uses abstract classes with no implementation details. Languages like Java, have interfaces which are the equivalent. And of course, dynamic languages need none of this, since they don't declare interfaces, so the entire matter is simply: Don't hard-code the receiver class.

Should, in a different situation, we need the lamp to be controlled by other things, like a dimmer switch, or voice, or signals from external devices, it becomes even more unclear which is higher level, and in particular, who should own which interface.

In this case, the design question is: Which class owns the interface definition?

When working in a large project, one solution is to put the interface definition in a separate module, referenced by two different teams, and the interface module becomes the shared agreement of the interface between the teams.

In May, 2023, I asked Bob Martin to comment on the above text. He replied:

> *Nowadays I define level the way Page-Jones defined it so long ago: distance from IO.*

Relating the dependency inversion principle to the Configurable Receiver pattern, the dependency inversion principle has the sender declaring a *required* interface so that receivers can be changed with the minimal amount of recompilation. The dependency inversion principle mentions the reasons to choose this design and describes the required interface, but does not mention the configurator.

2. Dependency Injection (run-time topic)

"Dependency injection" seems to have two origins, with slightly different meanings.

As far as I can find or have been told (thanks Chris Carroll), the phrase "dependency injection" was coined in 2003 for the Spring Framework, and specifically included in its scope constructors, factories, and service locators. [https://spring.io/blog/2006/11/09/spring-framework-the-origins-of-a-project-and-a-name] and [https://docs.oracle.com/javaee/7/api/javax/inject/package-summary.html]:

> *This package specifies a means for obtaining objects in such a way as to maximize reusability, testability and maintainability compared to traditional approaches such as constructors, factories, and service locators (e.g., JNDI). This process, known as dependency injection, is beneficial to most nontrivial applications.*

Martin Fowler in his excellent 2004 article [https://martinfowler.com/articles/injection.html] excludes service locator from his use of dependency injection. He writes:

> *Injection isn't the only way to break this dependency, another is to use a service locator. ... Dependency injection and a service locator aren't necessarily mutually exclusive concepts.*

Although I personally like the way Spring introduced the phrase "dependency injection," Fowler's use has become more common. Wikipedia at this moment says [https://en.wikipedia.org/wiki/Dependency_injection]:

> *dependency injection is a design pattern in which an object or function receives other objects or functions that it depends on ... Fundamentally, dependency injection consists of passing parameters to a method.*

That is how I will use the phrase.

What turns out to be significant is that there is an unmentioned third element, C, which passes B as an argument to A. So, C has to know both A and B to start off with. C gets to know both A and B, then passes B in to A for future use.

In the end, *Dependency Injection* is a way of saying *how* A comes to know of B: C tells A. It says nothing about what A does with B afterwards.

Important note: When I write "C passes B as an argument to A", I very specifically wish to finesse just how and when that is done. It might be done in A's constructor, or in a setter, or in any other way you can think of. Just that C gets that information to A.

3. Dependency Lookup (run-time topic)

Dependency Lookup is the other way for element A to get B's identity at run time: A asks some third party C for that information.

From [http://xunitpatterns.com/Dependency%20Lookup.html]:

> *... a "component broker" that returns to us a ready to use object.*

As with dependency injection there is a third element C that knows which B to use at that moment, but in this case, A asks C for the information as needed.

Aside from hard-coding, there are two ways for A to learn of B:

- A asks C about B (dependency lookup)
- C tells A about B (dependency injection)

Common to the *Dependency Injection* and *Dependency Lookup* is that they describe only *how* A obtains the B's identity. They say nothing about what A does with it afterwards. This is relevant when analyzing *Inversion of Control*.

4. Inversion of Control (run-time topic)

Inversion of Control is a run-time concept that is unrelated to any of the patterns described so far. It is often misrepresented. Even the Wikipedia entry had to be updated to correct the errors previously there [https://en.wikipedia.org/wiki/Inversion_of_control].

This idea was first publicized in a 1985 paper describing the Mesa system, using the phrase "Hollywood's Law" [https://digibarn.com/friends/curbow/star/XDEPaper.pdf]:

> Don't call us, we'll call you *(Hollywood's Law). A tool should arrange for Tajo to notify it when the user wishes to communicate some event to the tool, rather than adopt an "ask the user for a command and execute it" model*

"Inversion of control" was used in passing in the 1988 paper, "Designing Reusable Classes" by Ralph Johnson and Brian Foote [http://www.laputan.org/drc/drc.html]:

> *One important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in coordinating and sequencing application activity. This inversion of control gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application.*

Note here that what they are describing has nothing in common with what we have been talking about so far in terms of dependency inversion, injection or lookup. It is a totally unrelated concept.

In *Inversion of Control*, element A registers interest in a topic with element B, or the injection framework does that registration. Then when B has something of interest for element A, B calls or sends a message to A.

Note this is always a two-step process:

1. A registers or gets registered with B to set up a callback location.
2. When B detects a relevant event, it calls back to that callback location.

A suitable alternative word would be “callback.”

The "inversion" mentioned here is a reference to a “normal” call sequence, where A calls B to render a service, and A is in control of the call timing.

In the new situation, once B has A's id, B takes control of the timing, and calls A when something important happens. Hence, "inversion of control."

Inversion of control is a characteristic of frameworks, as opposed to libraries.

- When using library element B, A calls B to perform some task.
- When using framework element B, B calls element A for the specialized behavior needed to make the framework fit that situation.

This mechanism is widely used with UI frameworks, event systems, and ASP.NET. In each, the framework "wakes up" our object to handle some event.

Here is how .NET uses inversion of control (from “Dependency Injection in .NET”, Mark Seemann):

> *The term Inversion of Control (loC) originally meant any sort of programming style where an overall framework or runtime controlled the program flow. According to that definition, most software developed on the .NET Framework uses loC.*
>
> *When you write an ASP.NET application, you hook into the ASP.NET page life cycle, but you aren't in control-ASP.NET is.*
>
> *When you write a WCF service, you implement interfaces decorated with attributes.*
>
> *You may be writing the service code, but ultimately, you aren't in control- WCF is.*

> *These days, we're so used to working with frameworks that we don't consider this to be special, but it's a different model from being in full control of your code.*
>
> *This can still happen for a .NET application most notably for command-line executables. As soon as Main is invoked, your code is in full control. It controls program flow, life-time, everything. No special events are being raised and no overridden members are being invoked.*

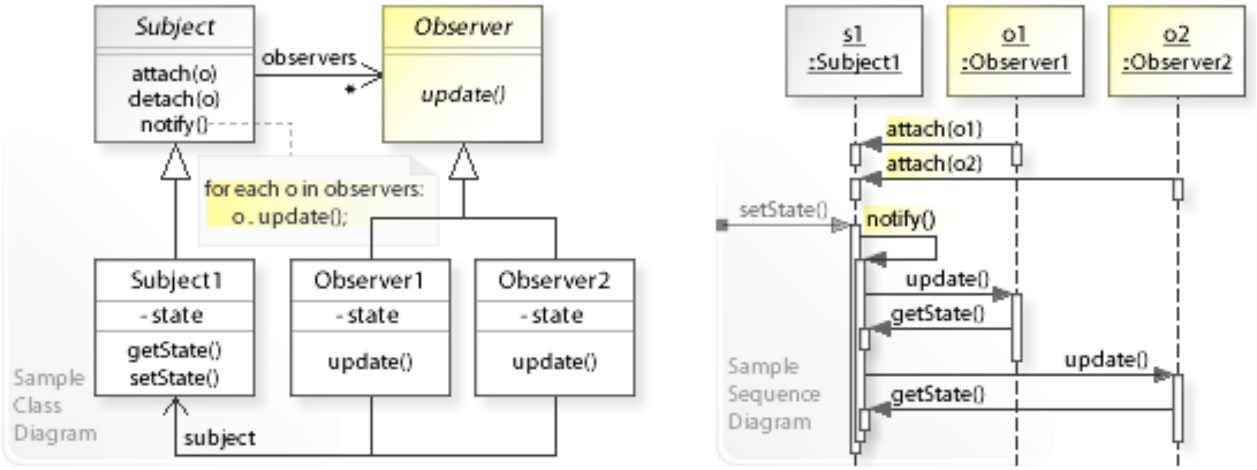
I hope you see his clear description of "normal" verses "inverted" control. If you write Main, you are in "normal" control. If you write an ASP.NET application, you do the two steps mentioned:

- First, hook into the ASP.NET page life cycle.
- Then, ASP.NET takes control and calls your code when events warrant it.

"Normal" control and "inversion of control" can be used in alternation. I like to think of inversion of control as setting a callback, where the callback code just continues the conversation between A and B, filling in some information B needs.

For tracing the behavior of A and B in inversion of control, a nice, simple example to walk through is the Observer pattern. In this example, the observer is A and the subject is B. [https://en.wikipedia.org/wiki/Observer_pattern].

Notice in the following that there is no element C that has to introduce them to each other. How A comes to know about B is not part of the *Inversion of Control* pattern.



The Observer pattern (source: https://en.wikipedia.org/wiki/Observer_pattern)

In the figure, we first see each observer (element A) attaching itself to the subject (B), "normal control". Later, (B) calls each (A) back to say that something has changed. In the third step, the observer (A) calls the subject (B) in "normal control" again to ask for some specific information.

Only the step in which B calls back to A is the "inversion of control" we are referring to. What makes the subject calling the observer an "inversion of control" is the observer

(A) sitting idle with respect to the subject (B) until something of interest happens and the subject takes the initiative to call the observer.

Here is the example of *Inversion of Control* from Wikipedia.
[https://en.wikipedia.org/wiki/Inversion_of_control]

> *A web application registers the endpoints it listens on with a web application framework, and then lets control pass to the framework. For instance, this example code for an Asp.NetCore web application creates an web application host, registers an endpoint, and then passes control to the framework:*

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/", () => "Hello World!");
app.Run();
```

Don't confuse *Inversion of Control* with *Dependency Injection* or *Dependency Lookup*, they have nothing to do with each other.

- *Dependency Injection* and *Dependency Lookup* talk about *how* element A comes to know of element B, namely via some element C. A will then call or send a message to B in the usual way, not using inversion of control. A controls the timing of the call.
- *Inversion of control* talks about who is in control of the timing of their interaction: If A is in control, then it's a "normal control" situation, if B is in control of the timing, then it is an "inversion of control" situation. There is no element C in the picture.

A can come to be registered with B in any manner: hardcoded, dependency injection, dependency lookup or injection framework.

5. Configurable Receiver (compile-time and run-time)

The *Configurable Receiver* pattern says (in extract):
[https://alistaircockburn.com/Articles/Configurable-Receiver]

Configurable Receiver (*Behavioral*)

Set or alter a receiver at run time.

> Arranging for a receiver to be set at run time affects both the source code structure and the run-time behavior. This pattern addresses both.

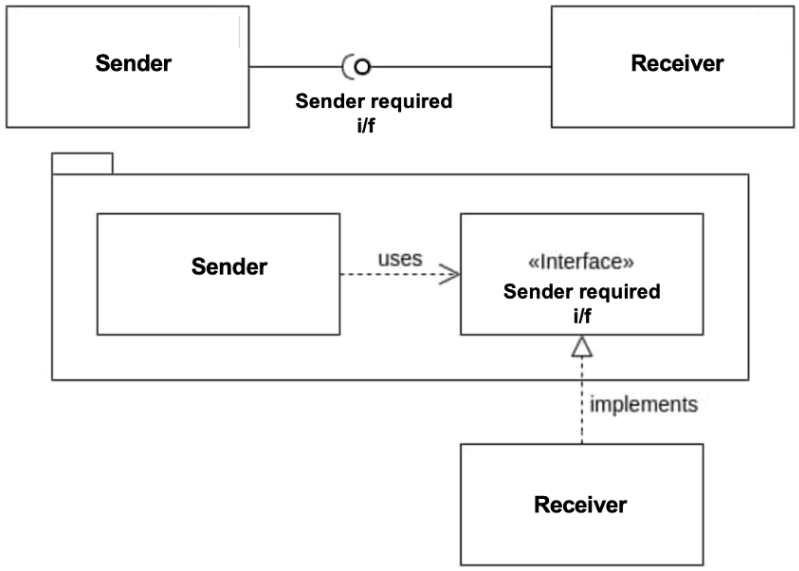
Configurable Receiver subsumes *Dependency Injection* and *Dependency Lookup*.

For a sending function or object to call or send a message to a receiver, it must know the receiver's identity. If that is written in the source code, then the sender has a compile-time dependency on the receiver.

What we are looking for is a way to structure the source code so that receiver can be chosen at run time, without changing the sender's source code.

[Aside: You will notice this is exactly what the *Dependency Inversion Principle* is asking for.]

In some languages, compile-time dependencies matter. The figure shows the sender defining and owning its required interface. The receivers depend on the sender, the sender doesn't depend on any receiver. This is what we are after.



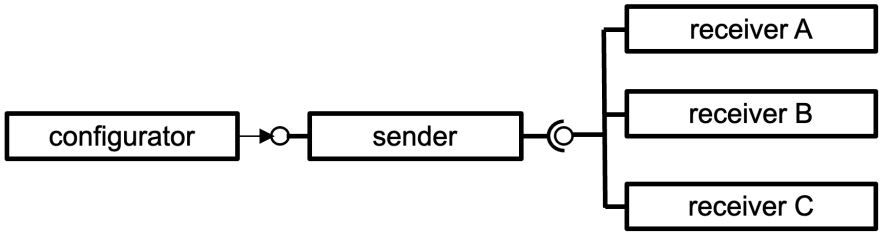
The sender owns the interface; receivers can be in different modules.

In some languages, the required interface is just whatever calls the sender makes. As no interface declarations are needed in such languages, the rule is simply: "*Don't hard-code the receiver*."

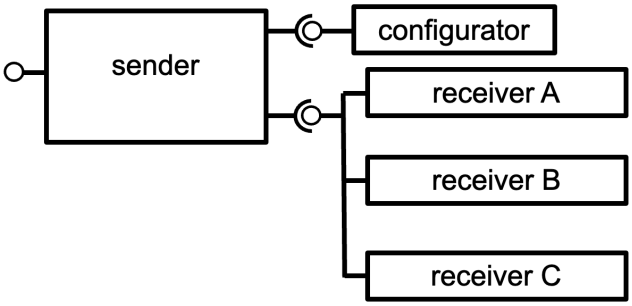
At run time, something—a "configurator"—must tell the sender what receiver to use. Having a configurator is therefore part of the pattern, although the specific design of the configurator is outside the pattern.

The configurator may provide the sender with the receiver in one of two ways, making for two possible views of the pattern as implemented:

Choice 1: The configurator tells the sender what receiver to use.



Choice 2: The sender asks the configurator which receiver to use.



In the first case, the configurator uses dependency injection to set the receiver. The second case uses dependency lookup: the configurator may be called a "service locator" or "broker".

Configurable Receiver implements the *Dependency Inversion Principle* at compile time. The run-time uses either *Dependency Injection* or *Dependency Lookup*, depending on the programmer's needs. In this way, *Configurable Receiver* subsumes *Dependency Injection* and *Dependency Lookup*.

As before, *Inversion of Control* is completely unrelated.

6. Relating them all

I found it useful to put into tables the different issues they are each dealing with. Some are compile-time while others are run-time. Here are the tables:

	Compile time	Run time
Dependency Inversion Principle (DIP)	✓	
Dependency Injection (DI)		✓
Dependency Lookup (DL)		✓
Configurable Receiver (CR)	✓	✓
Inversion of Control (IoC)		✓

At Run Time:	A controls call timing	B controls call timing
An element C informs A of B	DI, DL, CR	
A comes to know B, but B doesn't need to know A	DI, DL, CR	
B knows A		IoC

The "dependency" in *Dependency Inversion Principle* refers to compile-time dependency. All the others are run-time topics. *Configurable Receiver* covers both.

The second table shows three issues in play at run time.

- In *Dependency Injection, Dependency Lookup, Configurable Receiver*, there must be a third element C that informs A about B. This is not required in *Inversion of Control*.
- In *Dependency Injection, Dependency Lookup, Configurable Receiver*, A calls B whenever it wants, B does not call back. In *Inversion of Control*, B calls A back when it wants. That call is the "inversion of control."
- In all cases, A has to know B in order to make the first call. But in *Inversion of Control,* B also has to know of A in order to make the callback. That makes it different: both A and B have to know the other.

In short:

- *Dependency Inversion Principle* is a compile-time recommendation on how to structure the source code so that receivers can be set at run time without having to recompile the sender:
  → In the source code, make the sender dependent on an interface that gets implemented by the allowed receivers.
- *Dependency Injection* says how a sender comes to know of a receiver at run time:
  → A configurator tells the sender what receiver to use.
  How they interact after that is outside the pattern.
- *Dependency Lookup* says how a sender comes to know of a receiver at run time:
  → The sender asks a configurator what receiver to use.
  How they interact after that is outside the pattern.

- *Configurable Receiver* covers both compile-time and run-time issues.
  → At compile time, the sender defines and owns a required interface that every receiver must implement. (*Dependency Inversion Principle*)
  → At run time, the sender either asks or is told by a configurator what receiver to use (*Dependency Injection* or *Dependency Lookup*).
  How they interact after that is outside the pattern.
- *Inversion of Control* is about who controls the timing of the interactions:
  → Element A registers or gets registered with element B to set up a callback.
  → Later, B calls A to tell it or ask it something.
  How B comes to know about A is outside the pattern.

7. Common errors

Thinking *Dependency Injection* is *Inversion of Control*

People erroneously think that *Dependency Injection* is related to *Inversion of Control.*

> *In Java world Spring framework is very popular. It's responsible for DI (among other things) and what enables DI is called an "IoC container". Thus people used to Spring framework, who can't imagine DI done differently, may confuse DI with IoC.*
> [https://twitter.com/DmytroPolovynka/status/1661508626626822150]

Note, in contrast, the very explicit descriptions given earlier:

> *This package specifies a means for obtaining objects in such a way as to maximize reusability, testability and maintainability compared to traditional approaches such as constructors, factories, and service locators (e.g., JNDI). This process, known as dependency injection, is beneficial to most nontrivial applications.*
> [https://docs.oracle.com/javaee/7/api/javax/inject/package-summary.html]

> *dependency injection is a design pattern in which an object or function receives other objects or functions that it depends on ... Fundamentally, dependency injection consists of passing parameters to a method.*
> [https://en.wikipedia.org/wiki/Dependency_injection]

There is nothing there about registering A with B and then waiting for B to call A back.

In the other direction. *Inversion of Control* doesn't require *Dependency Injection*:

> *The term Inversion of Control (loC) originally meant any sort of programming style where an overall framework or runtime controlled the program flow. According to that definition, most software developed on the .NET Framework uses loC.*
>
> *When you write an ASP.NET application, you hook into the ASP.NET page life cycle, but you aren't in control-ASP.NET is.*
>
> *When you write a WCF service, you implement interfaces decorated with attributes.*

You may be writing the service code, but ultimately, you aren't in control- WCF is. [“Dependency Injection in .NET”, Mark Seemann]

In *Inversion of Control*, B has to come to know of A. This can be done with an explicit call from A to B (even hardcoded), or through a reflective framework.

In other words, you don’t have to use *Dependency Injection* to get *Inversion of Control* and you don’t have to use *Inversion of Control* with *Dependency Injection*.

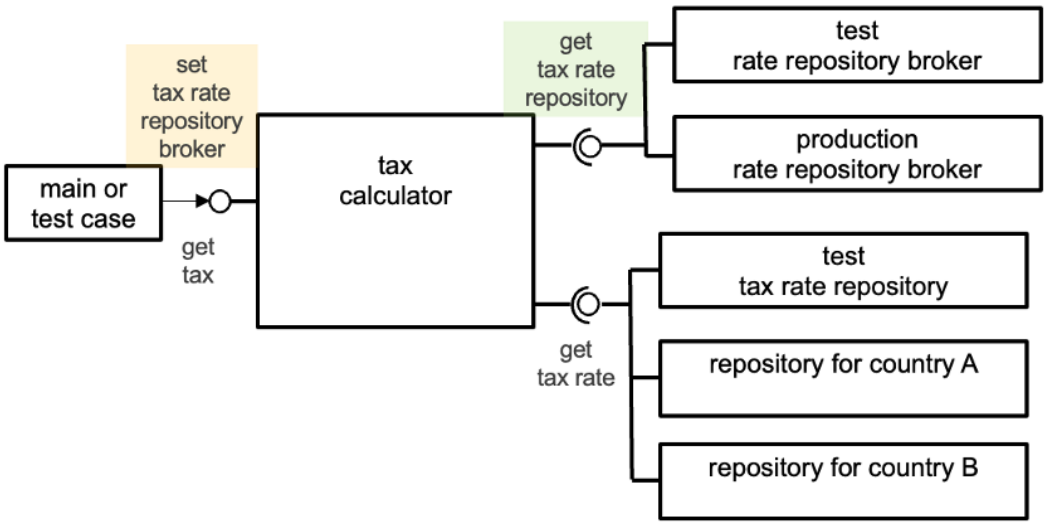
They are two independent constructs.

The 2nd error: Thinking *Dependency Lookup* broker must be hardcoded

The other common error I see in the literature is saying that in *Dependency Lookup*, the link from the sender to the service locator must be hard-coded. Even searching various expert sources, many said to hard-code the service locator or broker to the app.

This should patently make no sense to you, but for the doubting, consider example 6 from the *Configurable Receiver* article:
[https://alistaircockburn.com/Articles/Configurable-Receiver].

> Different tax rate sources are used in different countries. The calculator asks the RateRepositoryBroker what tax rate repository to use each time. We could hard code the link from TaxCalculator to RateRepositoryBroker, but we let Main (the configurator-configurator in this case) supply RateRepositoryBroker to the TaxCalculator at the start.
>
> This example illustrates *Dependency Injection* as Main passes the RateRepositoryBroker to TaxCalculator, and *Dependency Lookup* with TaxCalculator asking the RateRepositoryBroker each time which rate repository to use.



Main provides a broker to use to look up receivers.

If you really are going to connect to the tax offices of different countries, that rate repository broker is a key source of errors. You will want to test with a test double for the broker. Then, at production time, use the production broker with the real country tax sources, without having to recompile the tax calculator app.

For long-lived objects, it might be that technology evolves while the system is running, and a different service locator needs to be used. You will need a way to send in a new broker.

Thus, it make sense to apply *Configurable Receiver* pattern to the rate repository broker and not hard code it.

Martin Fowler describes how the Avalon framework supports the same design: [https://martinfowler.com/articles/injection.html]

> *Dependency injection and a service locator aren't necessarily mutually exclusive concepts. A good example of using both together is the Avalon framework. Avalon uses a service locator, but uses injection to tell components where to find the locator.*
>
> *Berin Loritsch sent me this simple version of my running example using Avalon.*

```
public class MyMovieLister implements MovieLister, Serviceable {
  private MovieFinder finder;
  public void service( ServiceManager manager ) throws ServiceException {
    finder = (MovieFinder)manager.lookup("finder");
  }
```

8. Final thoughts

"Dependency" and "inversion" have become a part of our industry's language and culture. However, given the number of errors of interpretation I see even in expert sources, it is time to reconsider them, or at least to clean up our use of the terms.

"Dependency" refers ambiguously to a compile-time or run-time dependency. Therefore, I beg everyone to make clear which you mean when you use the word.

"Inversion" only tells us we want the opposite of something, but then doesn't even say what that was that we didn't want. We prefer names that say what it *is* we are after.

Thus the name *Configurable Receiver* (thanks Daniel Terhorst-North).

I like to say "callback" instead of *Inversion of Control*, when I can.

I prefer to say "pass in an argument" for *Dependency Injection*, because that is normal programming action.

I prefer to say "broker" or "service locator" for *Dependency Lookup*, as in "Use a broker to select the relevant country tax source."

An example of speaking/writing in this way, for is the sentence earlier:

> This example illustrates *Dependency Injection* as Main passes the RateRepositoryBroker to TaxCalculator, and *Dependency Lookup* with TaxCalculator asking the RateRepositoryBroker each time which rate repository to use.

I would prefer to write:

> In this example, Main passes the RateRepositoryBroker to TaxCalculator, then the TaxCalculator asks the RateRepositoryBroker each time which rate repository to use.

I won't change everyone's language with this article, but perhaps some people will make fewer mistakes with these phrases.

Best wishes managing your dependencies :).

Alistair Cockburn
Humans and Technology Technical Report 2023.02