Configurable Receiver Subsumes Dependency Injection and Dependency Lookup

Alistair Cockburn
Humans and Technology Technical Report 2023.01 (v21d, 2023-06-04)

Configurable Receiver (*Behavioral*)

Set or alter a receiver at run time.

Arranging for the 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*.

1. Motivation and informal structure

Whether functions, objects, or systems, a *sender* needs to call or send a message to a *receiver*. We sometimes want to set the receiver at run time. For example:

- To develop a system with test data, then put it into production using production data without having to change the source code, but just to restart the system and set whichever data supplier we need during initialization.
- To evolve the system, using perhaps data files to start with and evolving to different databases over time.
- To change receivers in real time based on the data being handled.

For the receiver to be configurable at run time, we need to add a configurator (Figure 1). The specific design for the configurator is outside the pattern, as will be discussed.

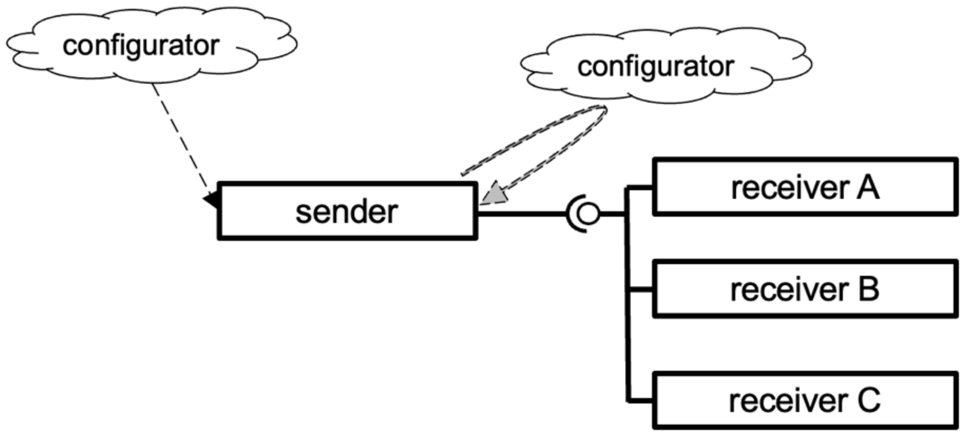


Figure 1. Informal view of Configurable Receiver, showing two choices for the placement of the configurator.

2. When to use the pattern

This pattern trades complexity for flexibility. Here, I present that trade using *Indications, Counterindications, Complicating side effects*, and *Overdose effect* (the idea that too much of a good thing is not a good thing).

Indications:

- You want to be able to set the receiver during execution, whether at initialization, in real time per individual data, or over a period of years as technologies shift.
- You want to be able to replace production connections with test harnesses, and back again, without changing the sender's source code.
- In all cases, you want to avoid having to change the source code and then rebuild the system every time you make these shifts.

Counterindications:

- You don't see those things in your future, and you don't want to add complexity to your system unnecessarily.
- Not every receiver of every message needs to be configurable, so you choose the moments at which to implement this pattern.

Complicating Side Effects:

- Since the call will be determined at run time, it adds one more layer of indirection.
- You must introduce a *configurator* that will provide the receiver to the sender, and you will have to walk through all the design choices that surround the configurator.
- Either the sender will have a provided interface for the configurator to pass in the receiver, or the sender makes an extra call to the configurator to get the receiver.
- Either the sender has one more local variable to hold the receiver, or must get the receiver every time.
- In statically type-checked languages, the sender must declare the interface that all intended receivers must implement. (This is not needed in dynamic languages.)
- In those statically type-checked languages, declaring that interface adds complexity to the source code folder structure.

Overdose Effect:

- If you put Configurable Receivers everywhere, your code becomes unnecessarily cluttered.

3. Implementing the Pattern

Source code structure

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.

Often, programmers hard code the intended receiver or its concrete class. To change the receiver, the source code has to be changed and recompiled. This is slow, error-prone, blocks migration to newer technologies, and makes testing harder.

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.

In Figure 1, using UML notation, the socket to the right of the sender indicates its *required* interface, the interface any suitable receiver must implement. The socket indicates that the sender "owns" the interface.

The ball to the left of the receivers indicates their *provided* interface, what any client of the receiver must call. The ball in the socket shows that the provided interface matches the required interface. The multiple receivers wired to the ball show that they all implement the same provided interface and can all be used there.

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

In other languages, compile-time declarations and dependencies matter. Figure 2 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.

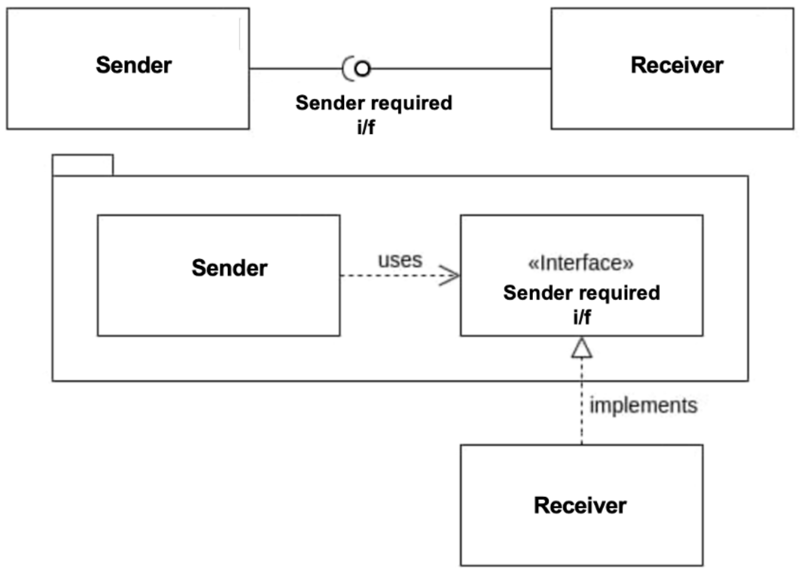


Figure 2. The sender owns the interface; receivers can be in different modules.

Just to show a common source-code structure that is useful in other places but does not have the property we want just now, Figure 3 shows the receiver defining and owning the interface. Here the sender has a compile-time dependency on the receiver. To use a different receiver we have to change the sender's source code and recompile. Not what we want here.

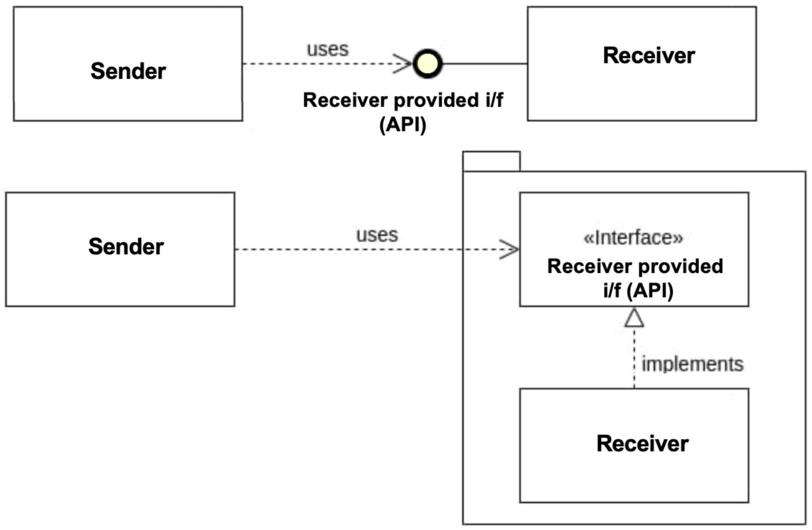


Figure 3. Not what we are after just now: The receiver owns the interface, the sender has a compile-time dependency on the receiver.

The configurator

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.

Here are some design decisions that have to be made around the configurator:

- Is the configurator hard-coded to know the sender, or is that connection made at run time?
- Does the configurator drive the sender or does the sender ask the configurator?
- Does the sender allow the configurator to set the receiver just once on initialization, or can the receiver be changed during execution?

These and other design decisions related to the configurator are not part of the Configurable Receiver pattern.

Run-time structure and behavior

The configurator may provide the sender with the receiver in one of two ways, making for two possible views of the pattern as implemented (Figures 4a and 4b):

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

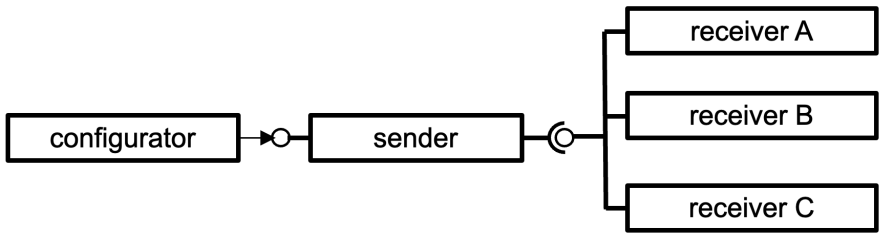


Figure 4a. The configurator tells the sender which receiver to use

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

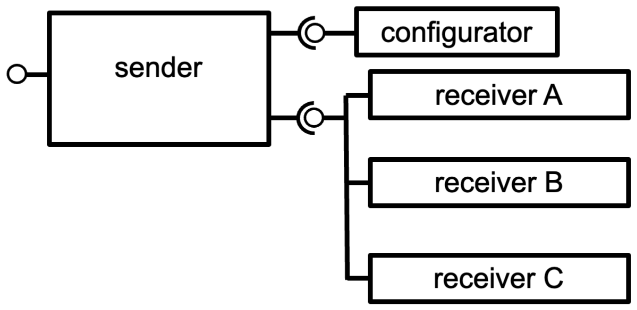


Figure 4b. The sender asks the configurator which receiver to use

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". In effect, the first design decision for the configurator is to choose between dependency injection and dependency lookup.

You may notice a problem in the second case: How did the sender get the id of the configurator?

The relation between the sender and the configurator in this second case is exactly a copy of the sender-receiver relation that this pattern is about. Walking through "*Indications and Counterindications*," you may decide that implementing Configurable Receiver here is not worth the trouble, and you hard-code the configurator in the sender.

On the other hand, you may want to indeed repeat the pattern for the configurator. This would be appropriate for testing receivers that change dynamically during normal execution, for updating a fixed receiver in long-running systems, or when writing a library function that will use a service locator and you need the client to be able to set what locator to use.

In such cases, you would introduce a (pardon the phrase) "configurator-configurator".

To get out of the recursion here, you might use the first method for the configurator-configurator—have it pass in the desired configurator to the sender at start-up and not change it again—and the second method for the sender-configurator pair: (Figure 5).

One implementation of Figure 5 is shown in sample code #4 and another in the Known Uses: Avalon framework.

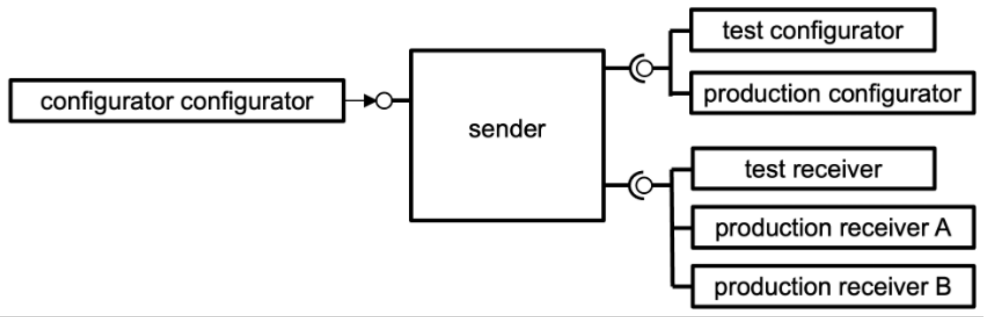


Figure 5. The configurator-configurator sends in a configurator to use as a service locator or broker for which receiver to use.

4. Sample Code

Here are examples creating and using a Configurable Receiver, moving from the simplest situation, using a function as the sender, to more complicated examples involving repositories. They are in different languages, to show the differences when using dynamic and statically type-checked languages:

1. A Configurable Receiver for objects, passing in the receiver at object creation time (in Ruby)

2. A Configurable Receiver for objects, passing in the receiver at object creation time (in Java)

3. A Configurable Receiver for objects, in which the configurator passes in the receiver at any time, to be used until further notice (in Java)

4. A Configurable Receiver in which the configurator is a repository and a configurator-configurator sets the configurator (in Ruby)

(1) A Configurable Receiver for objects, passing in the receiver at object creation time (in Ruby)

Here is an example of the configurator passing the receiver to the sender at object creation time. For this example, I imagine many future technologies for the tax rate repository: a test harness, a file, a database, or else perhaps direct connections to the tax office of various countries, hence needing different adapters.

For early development, I use an in-code tax rate repository with just one, fixed, tax rate.

Main is the configurator. It creates the receiver, FixedTaxRateRepository, and sends it to the sender object, the TaxCalculator, as part of the constructor.

I show Ruby code first, because as we don't have to declare the interfaces, the pattern is easier to see:

———

```
class TaxCalculator
  def initialize( tax_rate_repository )
    @tax_rate_repository = tax_rate_repository
  end

  def tax_cn( amount )
    amount * @tax_rate_repository.tax_rate( amount )
  end
end

class FixedTaxRateRepository
   def tax_rate( amount )
     0.15
   end
end

tax_rate_repository = FixedTaxRateRepository.new
my_calculator = TaxCalculator.new( tax_rate_repository )
puts my_calculator.tax_on( 2000 )
```

———

Thank you, ChatGPT.

(2) A Configurable Receiver for objects, passing in the receiver at object creation time (in Java)

Here is the same example, using Java to show the interfaces being declared.

———————————————————————————————————

```
interface ForGettingTaxRates {
   double taxRate(double amount);
}

class TaxCalculator {
   private ForGettingTaxRates taxRateRepository;

   public TaxCalculator(ForGettingTaxRates taxRateRepository) {
      this.taxRateRepository = taxRateRepository;
   }

   public double taxOn(double amount) {
      return amount * taxRateRepository.taxRate( amount );
   }
}

class FixedTaxRateRepository implements ForGettingTaxRates {
   public double taxRate(double amount) {
      return 0.15;
   }
}

class Main {
   public static void main(String[] args) {
      ForGettingTaxRates taxRateRepository = new FixedTaxRateRepository();
      TaxCalculator myCalculator = new TaxCalculator( taxRateRepository );
      System.out.println( myCalculator.taxOn( 2000 ) );
   }
}
```

———————————————————————————————————

Thank you, ChatGPT.

(3) A Configurable Receiver for objects, in which the configurator passes in the receiver at any time, to be used until further notice (in Java)

In this example, we move the setting of the receiver into a public function that can be called at any time. This is useful when the receiver might be changed dynamically.

```
interface ForGettingTaxRates {
   double taxRate(double amount);
}

class TaxCalculator {
   private ForGettingTaxRates taxRateRepository;

   public void setTaxRateRepository(ForGettingTaxRates taxRateRepository) {
      this.taxRateRepository = taxRateRepository;
   }

   public double taxOn(double amount) {
      return amount * taxRateRepository.taxRate( amount );
   }
}

class FixedTaxRateRepository implements ForGettingTaxRates {
   public double taxRate(double amount) {
      return 0.15;
   }
}

class Main {
   public static void main(String[] args) {
      ForGettingTaxRates taxRateRepository = new FixedTaxRateRepository();
      TaxCalculator myCalculator = new TaxCalculator();
      myCalculator.setTaxRateRepository( taxRateRepository );
      System.out.println(myCalculator.taxOn( 2000 ));
   }
}
```

Thank you, ChatGPT.

(4) A Configurable Receiver in which the configurator is a repository and a configurator-configurator sets the configurator (in Ruby)

The configurator-configurator example, matching Figure 5. (A second example is shown in the Avalon framework, in Known Uses.)

In Figure 6, 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.

Choosing Ruby this time because the intention is easier to see.

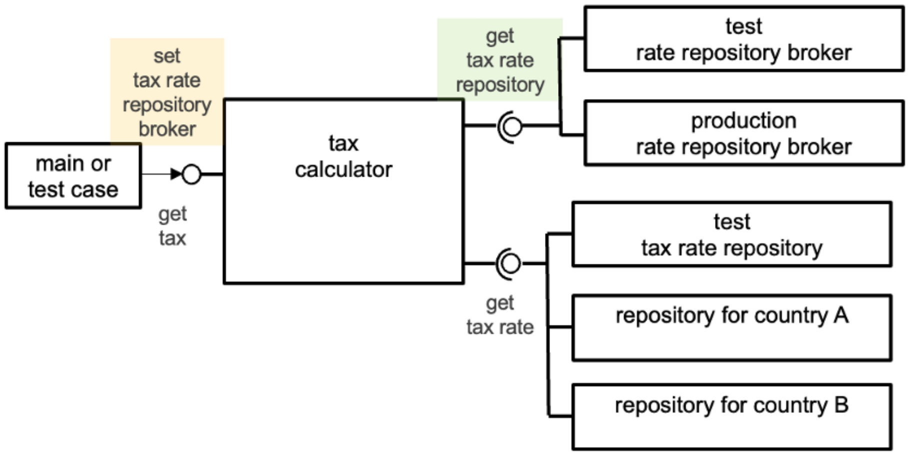


Figure 6. Main provides a broker to use to look up receivers.

```
_____________________________________________________
class RateRepositoryBroker
 def initialize
  @tax_rate_repository_FR = TaxRateRepositoryFR.new
  @tax_rate_repository_US = TaxRateRepositoryUS.new
 end
 def repository_for( country )
  if country == "US"      return @tax_rate_repository_US
  elsif country == "FR"  return @tax_rate_repository_FR
  else  return nil
  end
 end
end
```

```
class TaxCalculator

  def initialize( repository_broker )
    @my_rate_repository_broker = repository_broker
  end

  def tax_on( country, amount )
    tax_rate_repository = @my_rate_repository_broker.repository_for( country )
    amount * tax_rate_repository.tax_rate( amount )
  end

end


class TaxRateRepositoryFR
  def tax_rate( amount )
    0.30
  end
end

class TaxRateRepositoryUS
  def tax_rate( amount )
    0.15
  end
end

my_tax_rate_broker = RateRepositoryBroker.new
my_calculator = TaxCalculator.new( my_tax_rate_broker )
puts my_calculator.tax_rate( "FR", 2000 )
puts my_calculator.tax_rate( "US", 2000 )
```

———————————————————————————————————————

5. Known Uses

The Spring framework

Spring supports all variants of *Configurable Receiver*. From the original article [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.*

A note on vocabulary. That article says "dependency injection" includes dependency lookup. In his 2004 article [https://martinfowler.com/articles/injection.html] Martin Fowler kept dependency lookup separate from dependency injection.

The Avalon framework

The Avalon framework supports the configurator-configurator described in Figure 5, where the repository broker is not hard coded in the sender, but is passed in to the sender's constructor. From Martin Fowler [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");
   }
```

The Ports & Adapters pattern (Hexagonal architecture)

See [https://alistair.cockburn.us/hexagonal-architecture/]

The Ports & Adapters pattern requires all ports to be owned by the application, so they can all be connected at run time.

- Each *driving* port is published as the application's *provided* interface, making that port configurable with regard to its senders (and using the compile-time dependency structure shown in Figure 3).
- Each *driven* port is published as the application's *required* interface, making that port configurable with regard to its receivers (using the compile-time dependency structure shown in Figure 2).

Ports & Adapters does not say what form of configurator should be used. Any of the ones described in Configurable Receiver may be used.

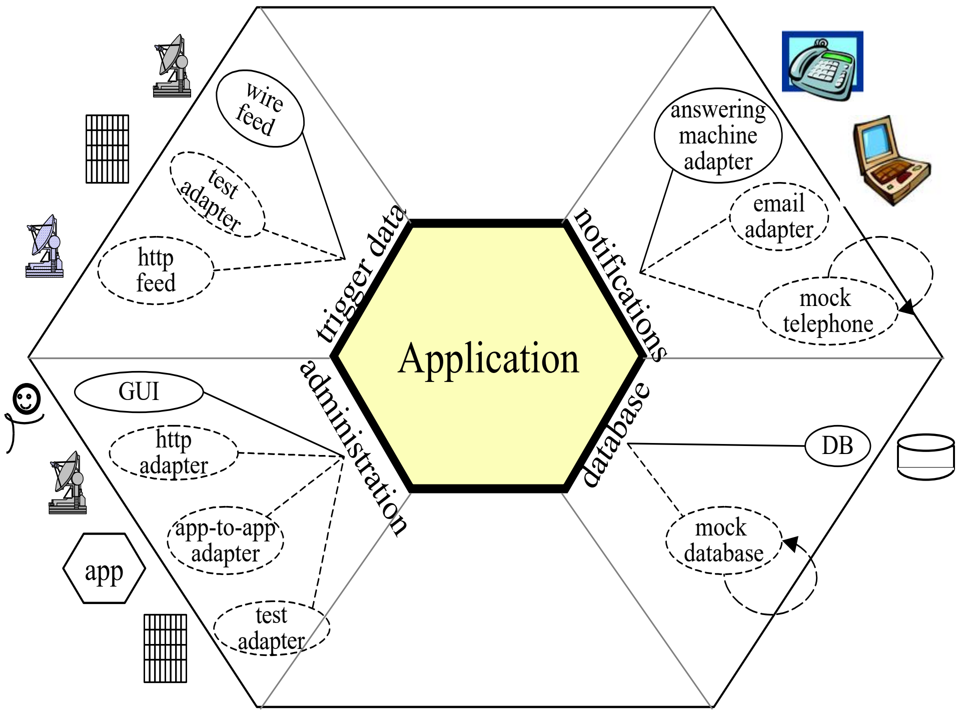


Figure 7. Ports & Adapters as known use of Configurable Receiver

6. Related Patterns

The Strategy pattern

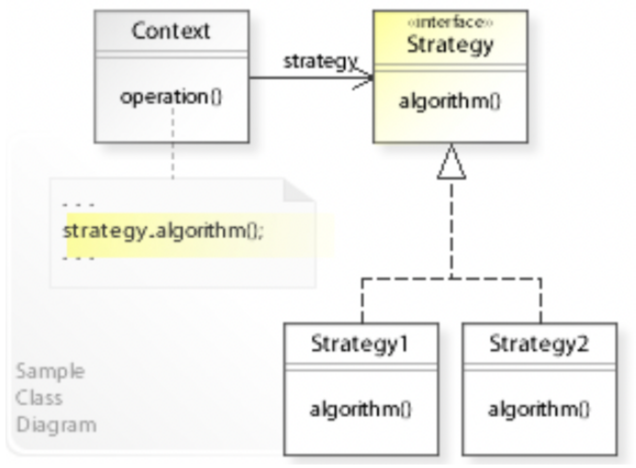


Figure 8: The Strategy pattern
(source: https://en.wikipedia.org/wiki/Strategy_pattern)

The Strategy pattern says that the "context" object has in its hands one of a set of possible objects that all respond to the same function call. The context object defines its required interface. Then, any of the conforming strategy objects can be used.

The Strategy pattern does not show the configurator. That is considered outside the pattern definition. Any of the configurators described in this article may be used.

Dependency Injection

From [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.*

In *Dependency Injection* the configurator tells the sender what receiver to use.

Dependency Lookup

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

> *We avoid hard-coding the names of classes on which we depend into our code because static binding severely limits our options for how the software is configured as it runs. Instead, we hard-code that name of a "component broker" that returns to us a ready to use object.*

In *Dependency Lookup* the sender asks the configurator what receiver to use.

Here they say the sender contains a hard-coded reference to the broker. That is not required (see [https://springframework.guru/service-locator-pattern-in-spring/] and Known Uses: Avalon). The key phrase is "returns to us a ready-to-use object."

7. Discussion of Dependency Inversion, Injection, Lookup

Configurable Receiver makes use of all three, *dependency inversion principle, dependency injection, dependency lookup*. As these terms confuse many people, this section clarifies the relationship. Additionally, *Inversion of control* is often conflated with those three patterns, although it is unrelated.

Part of the confusion is that "dependency" refers ambiguously to a compile-time or run-time dependency. Then, "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

This discussion here is subset [https://alistaircockburn.com/Articles/Discussion-of-dependency-injection-etc]. Please refer to that for the longer discussion.

1. Dependency Inversion Principle (a 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.

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 the design idea.

In fact, you will apply the Configurable Receiver pattern, as described later: 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: Who 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 different receivers can be used 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 (a run-time topic))

From [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.*

Element A has a *run-time* dependency on element B if it needs B to be present at run time to receive a call from A. If B is deleted before A calls it, A's call fails.

In this case, there is a third element, C, which knows about B and passes B as a argument to A, whether in A's constructor or through some other interface. Once A has that information, A has a run-time dependency on B; B had better not get deleted before A calls on it.

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.

3. Dependency Lookup (a 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.*

Leaving aside the unnecessary writing in that entry about hard coding things, this pattern says that there is a third element C that knows which B to use at that moment.
A asks C for the information as needed.

In the text, they suggest that the sender contains a hard-coded reference to the broker.
This is not necessary. A may come to know of C in any of the three ways: hard-coded, dependency injection, or dependency lookup.

In general, there are two ways for A to learn of B, when it is not hard-coded:

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

In terms of the Configurable Receiver pattern, C is the "configurator" in both cases. It has the magical information of which B A needs to use.

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

4. Inversion of Control (a 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.*

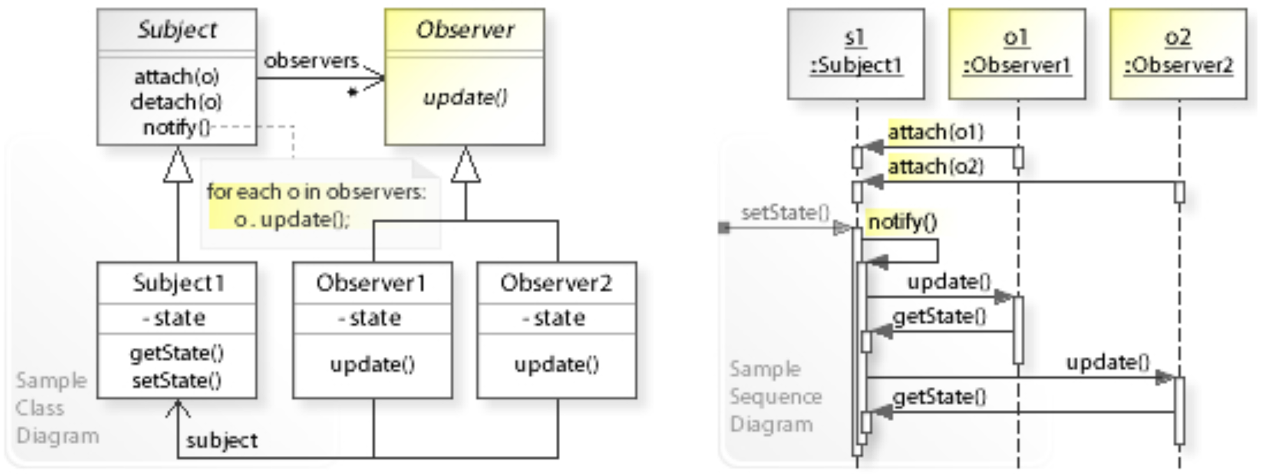
I hope you see his clear description of “normal” verses “inverted” control. If you write Main, you are in “normal” control. If you write an ASP.NET application, you do the two steps mentioned:

- 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();
```

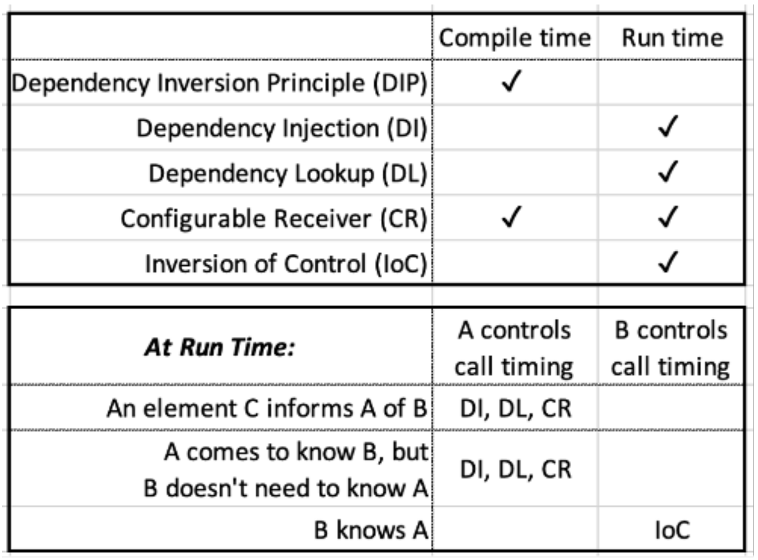
Don't confuse *Inversion of Control* with *Dependency Injection* or *Dependency Lookup*, they have nothing to do with each other.

- *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. 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.

8. Final thoughts on the writing and the naming of the pattern

There is a time when what one is doing so new that the obvious way to describe it is as "not the mainstream thing". Over time, we find a term for what-it-is instead of what-it-isn’t and the old term drops out of use.

- *“Horseless carriages”:* Early cars were called "horseless carriages". That became irrelevant once cars took over the road, and the term was dropped. Even "auto-mobile" was once hyphenated, as an attempt to distinguish it. “Car” was originally written "motor car". And so on.
- *“Inversion”:* "Inversion" is a reference to a previous something, meaning “not that”. Except, it even doesn't say what it is not. For this reason, I avoid “inversion”.

  Notes: In the Mesa paper, they stated, "Don't call us, we'll call you", with a slight indication of what had once been normal but wasn't being used, then focusing on what they *do* use. Johnson and Foote wrote "inversion of control" only as a passing phrase, not as a principle or a pattern name.
- *“Dependency”:* Does "dependency" refer to compile-time or run-time dependency? Compile-time dependency is important, as it affects build times. Run-time dependency is important if lifetimes are the worry: we worry that A will call B but B might get deleted before that. But which one is usually not stated.

  Even at run time, we are not passing in a “dependency”, we are passing in an actual object.

Thus, I avoid the terms “dependency” and “inversion” to the extent possible.

- *“Configurable Receiver”:* Daniel Terhorst-North finally noticed the thing we are actually working on: the receiver, that which should receive a future message. Hence the constructive name: *Configurable Receiver*.

Hopefully saying you will “use a Configurable Receiver” is clear. Your colleague might ask “Why?” and “How?” and “Is it worth it?” and a fruitful dialog follows.

Best wishes managing your dependencies :).

With many thanks to Juan Manuel Garrido de Paz, Chris Carroll, Valentia Cupać, and Martin Fowler for their close reading of the pattern and their detailed improvements.

Alistair Cockburn
Humans and Technology Technical Report 2023.01