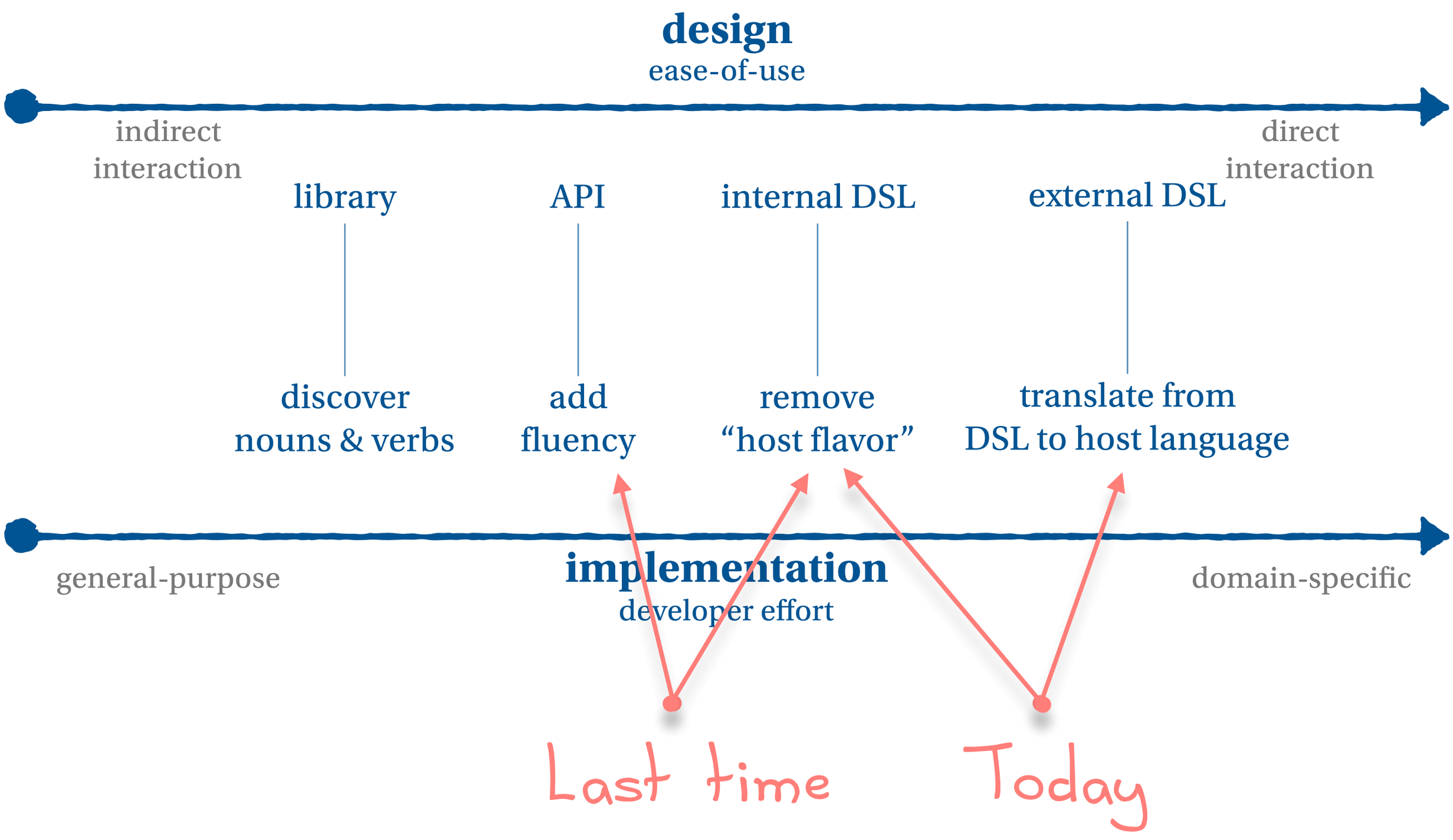
Today: implementation strategies



Recap: Removing the host flavor

We can omit the `.` from our method calls.

```
1  package pioneer.pictures
2
3  import pioneer.resource
4  import Picture._
5
6  object PictureProgram extends App {
7    load(resource("/image.png")).flipHorizontal().grayScale().rotateLeft().save("output.png")
8  }
9
```

These programs are equivalent!

```
1  package pioneer.pictures
2
3  import pioneer.resource
4  import Picture._
5
6  object PictureProgram extends App {
7    load(resource("/image.png")) flipHorizontal() grayScale() rotateLeft() save("output.png")
8  }
9
```

Recap: Removing the host flavor

```
1   package pioneer.pictures
2
3   import pioneer.resource
4   import Picture._
5
6   object PictureProgram extends App {
7     load(resource("/image.png"))
8     flipHorizontal()
9     grayScale()
10    rotateLeft()
11    save("output.png")
12  }
13
```

each line is a *statement*

```
1   package pioneer.pictures
2
3   import pioneer.resource
4   import Picture._
5
6   object PictureProgram extends App {(
7     load(resource("/image.png"))
8     flipHorizontal()
9     grayScale()
10    rotateLeft()
11    save("output.png")
12  )}
13
```

the entire program is one *expression*

Internal DSL
case study: time

Creating a class for the time domain

```
package time

class Time(val hours: Int, val minutes: Int, val seconds: Int)
```

↑ language implementation

↓ language use

```
package pioneer

import time.Time

object Program extends App {
  val time1 = new Time(12,  0, 30)
  val time2 = new Time(12,  0, 30)

  println(s"The time is $time1")
  println(s"time1 == time2: ${time1 == time2}")
}
```

```
The time is time.Time@fe18270
time1 == time2: false
```

Case classes are about data

They give us a `toString` method, an equality method, and we don't have to use `new` to create objects of case classes.

```
package time

case class Time(hours: Int, minutes: Int, seconds: Int)
```

↑ language implementation

↓ language use

```
package pioneer

import time.Time

object Program extends App {
  val time1 = Time(12,  0, 30)
  val time2 = Time(12,  0, 30)

  println(s"The time is $time1")
  println(s"time1 == time2: ${time1 == time2}")
}
```

```
The time is Time(12,0,30)
time1 == time2: true
```

Let's introduce a new behavior

```
package time

case class Time(hours: Int, minutes: Int, seconds: Int) {
  def addSeconds(moreSeconds: Int): Time = {
    val rawSeconds = seconds + moreSeconds
    val newSeconds = rawSeconds % 60

    val rawMinutes = minutes + rawSeconds / 60
    val newMinutes = rawMinutes % 60

    val rawHours = hours + rawMinutes / 60
    val newHours = rawHours % 24

    Time(newHours, newMinutes, newSeconds)
  }
}
```

↑ language implementation

↓ language use

```
package pioneer

import time.Time

object Program extends App {
  val time1 = Time(12, 0, 30)
  val time2 = Time(12, 0,  0).addSeconds(30)

  println(s"The time is $time1")
  println(s"time1 == time2: ${time1 == time2}")
}
```

```
The time is Time(12,0,30)
time1 == time2: true
```

Operators are just methods!

We can write a method whose name is +.

```
package time

case class Time(hours: Int, minutes: Int, seconds: Int) {
  def addSeconds(moreSeconds: Int): Time = {
    …
  }

  def +(moreSeconds: Int): Time = addSeconds(moreSeconds)
}
```

↑ language implementation

↓ language use

```
package pioneer

import time.Time

object Program extends App {
  val time1 = Time(12, 0, 30)
  val time2 = Time(12, 0,  0).+(30)

  println(s"The time is $time1")
  println(s"time1 == time2: ${time1 == time2}")
}
```

```
The time is Time(12,0,30)
time1 == time2: true
```

Removing the host flavor

For methods that take one argument, we can remove the parentheses from the call.

```
package time

case class Time(hours: Int, minutes: Int, seconds: Int) {
  def addSeconds(moreSeconds: Int): Time = {
    …
  }

  def +(moreSeconds: Int): Time = addSeconds(moreSeconds)
}
```

↑ language implementation

↓ language use

```
package pioneer

import time.Time

object Program extends App {
  val time1 = Time(12, 0, 30)
  val time2 = Time(12, 0,  0) + 30

  println(s"The time is $time1")
  println(s"time1 == time2: ${time1 == time2}")
}
```

```
The time is Time(12,0,30)
time1 == time2: true
```

Uh oh, there's a problem

We can add seconds to time; but we can't add time to seconds.

```
package time

case class Time(hours: Int, minutes: Int, seconds: Int) {
  def addSeconds(moreSeconds: Int): Time = {
    …
  }

  def +(moreSeconds: Int): Time = addSeconds(moreSeconds)
}
```

↑ language implementation

↓ language use

```
package pioneer

import time.Time

object Program extends App {
  val time1 = Time(12, 0, 30)
  val time2 = 30 + Time(12, 0,  0)

  println(s"The time is $time1")
  println(s"time1 == time2: ${time1 == time2}")
}
```

error!

```
The time is Time(12,0,30)
time1 == time2: true
```

Literal extension

(Seeming to) add new behavior to built-in objects (such as `Ints`)

```
package time

case class Time(hours: Int, minutes: Int, seconds: Int) {
 …
}

object Time {
  implicit class IntTime(val seconds: Int) {
    def +(time: Time): Time = time + seconds
  }
}
```

implicit conversion from Int to IntTime

↑ language implementation

↓ language use

```
package pioneer

import time.Time

object Program extends App {
  val time1 = Time(12, 0, 30)
  val time2 = 30 + Time(12, 0,  0)

  println(s"The time is $time1")
  println(s"time1 == time2: ${time1 == time2}")
}
```

now it works!

```
The time is Time(12,0,30)
time1 == time2: true
```

Traits

Traits are another way to extend programs, by "mixing in" data and behavior.

```scala
package time

trait RichTime {
  val midnight = Time(0, 0, 0)
  val noon = Time(12, 0, 0)
}
```

↑ language implementation

↓ language use

```scala
package pioneer

import time.{RichTime, Time}

object Program extends App with RichTime {
  val time1 = noon
  val time2 = 30 + noon

  println(s"The time is $time1")
  println(s"time1 == time2: ${time1 == time2}")
}
```

adds vals from RichTime to Program

```
The time is Time(12,0,30)
time1 == time2: true
```

Postfix operators

A way to add more fluency to the time domain

```scala
package time

trait RichTime {
  val midnight = Time(0, 0, 0)
  val noon = Time(12, 0, 0)

  implicit class TimeUnits(val seconds: Int) {
    def minutes: Int = seconds * 60
  }
}
```

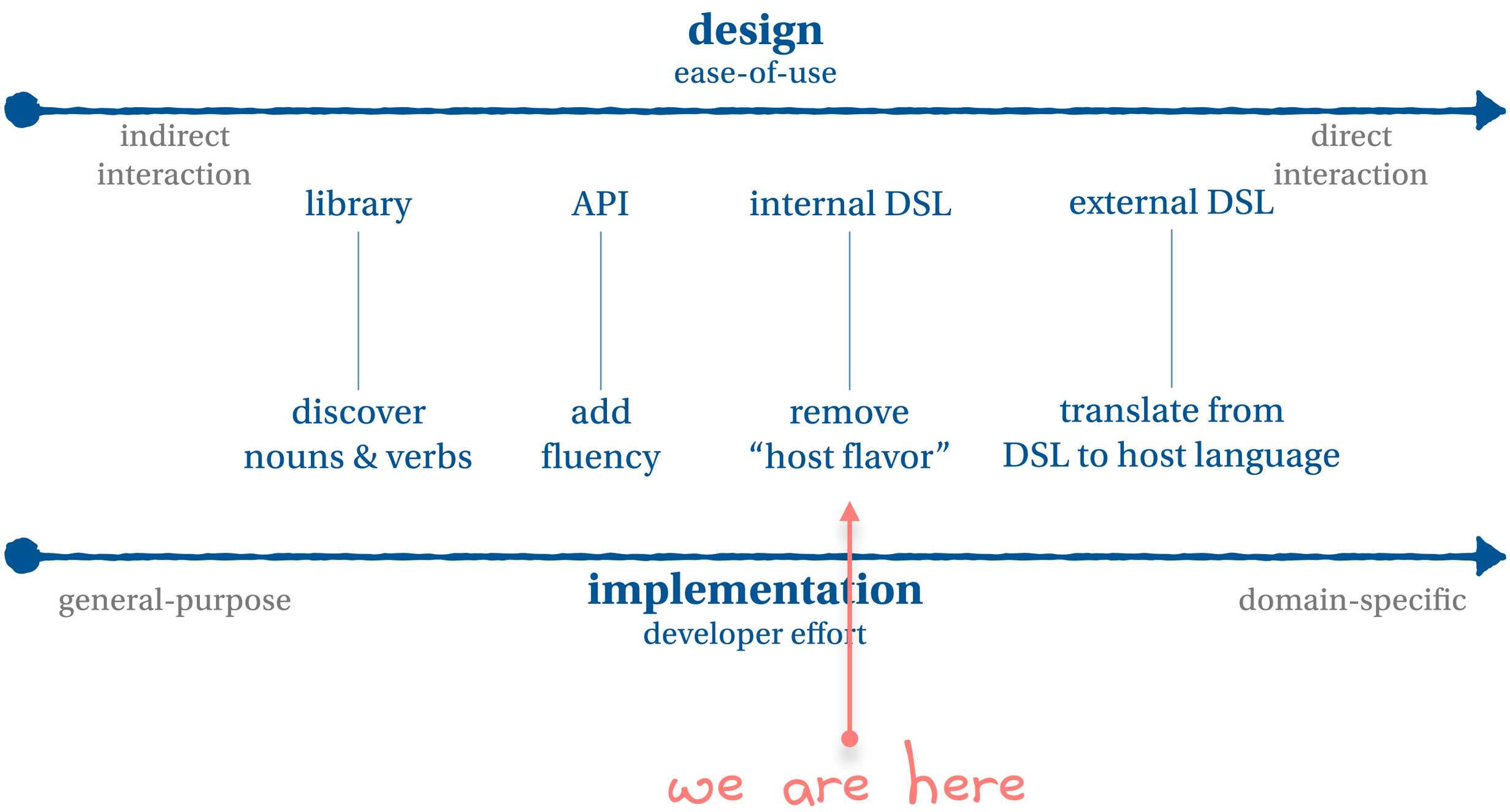
↑ language implementation

↓ language use

```scala
package pioneer

import time.{RichTime, Time}
import scala.language.postfixOps

object Program extends App with RichTime {
  val time1 = Time(12, 30, 0)
  val time2 = noon + (30 minutes)

  println(s"The time is $time1")
  println(s"time1 == time2: ${time1 == time2}")
}
```

```
The time is Time(12,0,30)
time1 == time2: true
```

DSL implementation strategies



Techniques for adding fluency

Most general-purpose languages support these features.

names including Unicode	sin(Θ) **ASK:** If the DSL supports Unicode, how will the user write programs?

whitespace	computer();
	processor();
	cores(2);
	disk();
	size(150);

function composition	computer(processor(cores(2)), disk(size(150))
method chaining	computer() .processor() .cores(2) .disk() .size(150) .end();

Techniques for **hiding the host language**

These features tend to be language-specific. Some languages support this ability more than others.

infix operators	set1 union set2 salaries map giveRaise
(re-)defining operators	set1 ∪ set2 set1 + set2 Different host languages gives us different amounts of control over precedence and associativity.
pre- and postfix operators	~1 i++
literal extension	3 little pigs

DSL implementation strategies