How does a programmer think about code

Szymon Rodziewicz

Who am I

- I contribute to the Scala Compiler, Scaladoc, and I was coordinating the Scala Toolkit project
- Ex Scala Compiler team, now Data Platform Engineer

“Designing programming languages only marginally involves empirical evidence [...] Instead, experience and plausibility are used”

N. Peitek et al., "A Look into Programmers’ Heads," in IEEE Transactions on Software Engineering, vol. 46, no. 4, pp. 442-462, 1 April 2020, doi: 10.1109/TSE.2018.2863303.

Scenario for today

We are tasked with designing good* and simple** language or API

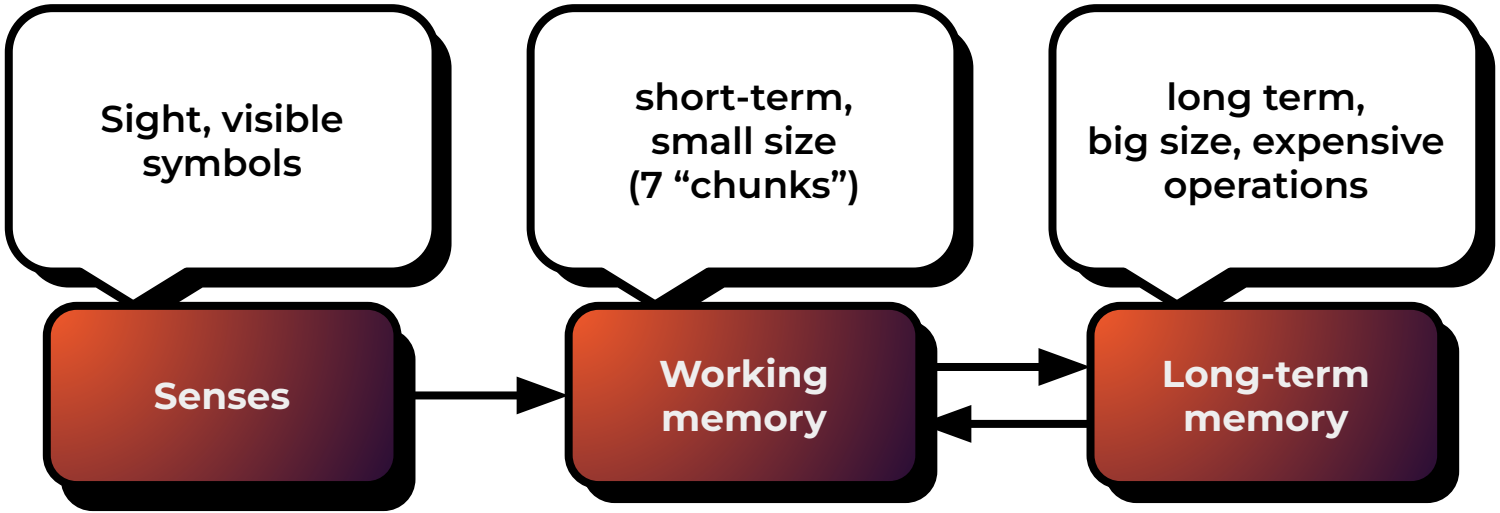
It is very hard to define “simple”

We can start with cognition

Our cognition is the foundation of the way we and our users write and read code

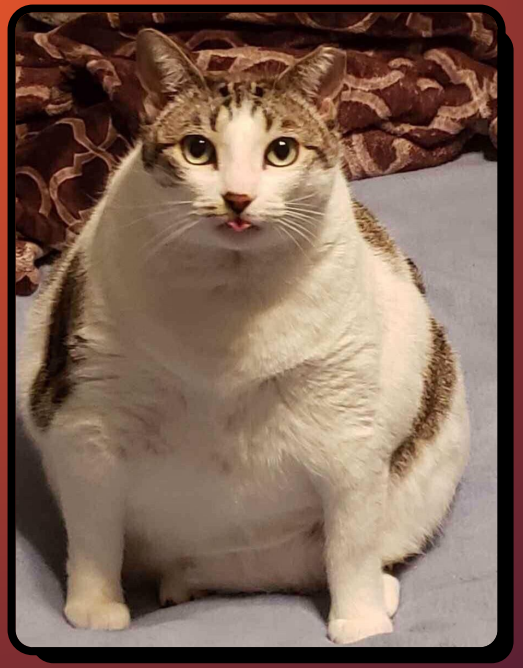
We can use it as a model to define simplicity.

Simplified model of cognition



What are chunks?

- The piece of information in working memory.
- We group information into chunks.



Do not confuse with chonks

Experiment!

Remember this digits sequence.
The more digits in correct order, the better.

2, 3, 1, 2, 1, 7, 1, 3, 2, 2, 1, 0, 1, 8, 2, 8

Experiment!

How many do you remember?

Experiment! – Second part

Remember these dates:

23.12.1713

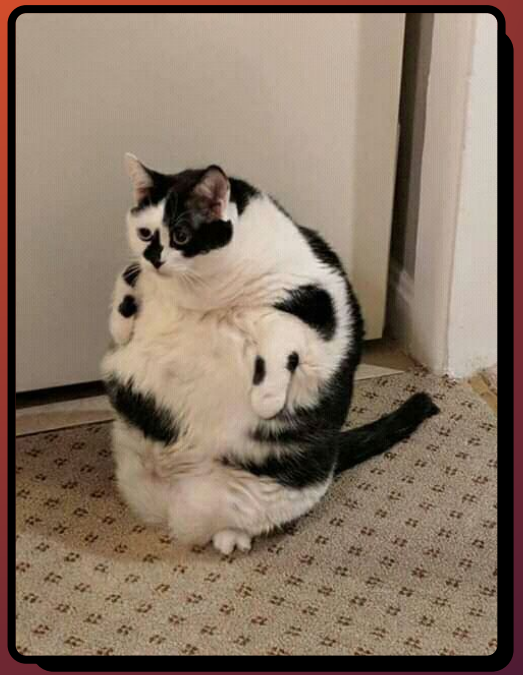
22.10.1829

Experiment! – Second part

How many do you remember now?

Conclusions

- Chunks allow us to group information and comprehend it as a whole
- It’s the unit that we operate on in our working memory



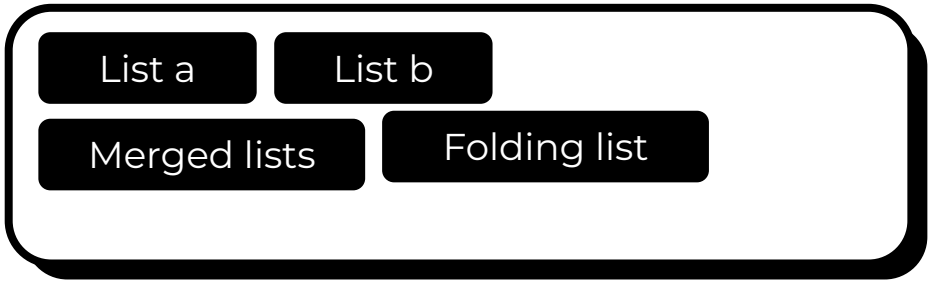
Chunks in programming - Scala

```
(((1 :: 2 :: 3 :: Nil) ++: (4 :: 5 :: 6 :: Nil)) /: 1)(_ + _)
```

- Semantics unknown - we have to store symbols as chunks.
- **Too much for working memory!**

Chunks in programming - Scala

```
val a = List(1,2,3)
val b = List(4,5,6)
val concat = a ++ b
concat.foldLeft(1)(_ + _)
```



- We are able to understand this code

Cognitive Load

- Amount of information you process in your working memory - **chunks**
- In our model - main limitation in code comprehension.

Simplicity can be described with cognitive load

Understanding the code

While trying to understand the code, we apply two approaches:

- Bottom-up
- Top-down

Top-down

- Perceiving meaning as it appears to be
- We go into details only when we must

Top-down example

```
@main
def main =
    val personJson = fetchJson()
    val person = processJson(personJson)
    doStuff(person)

def doStuff(person: Person) =
    sendMail(person.mail, "Hello")
```

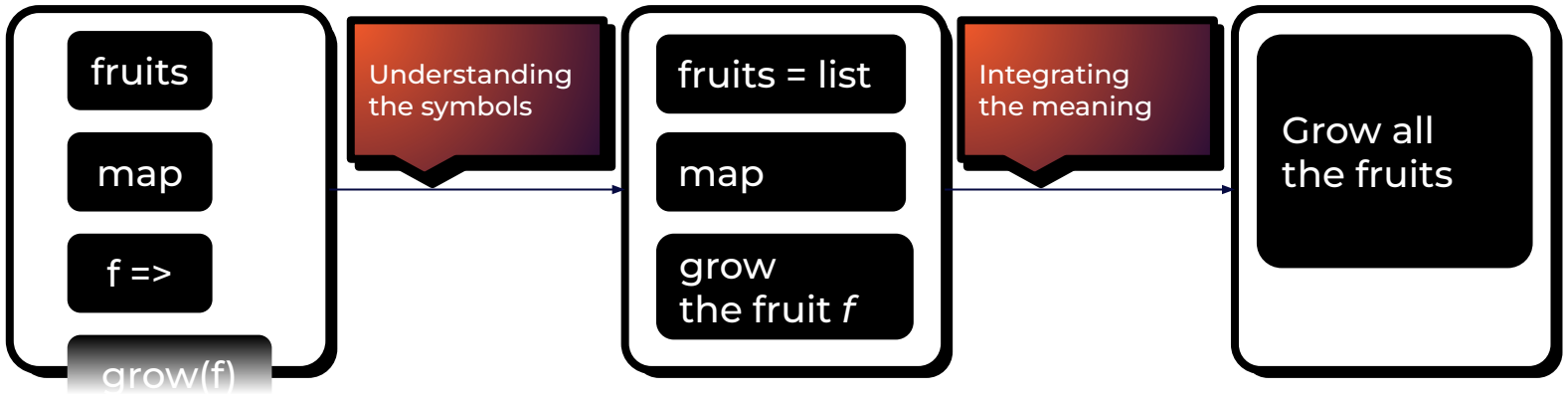
Bottom-up

- Merging symbols into meaningful chunks
- Used when debugging
- Also applied to the “harder” pieces of code

Bottom-up – Example

```
fruits.map(f => grow(f))
```

Chunks:

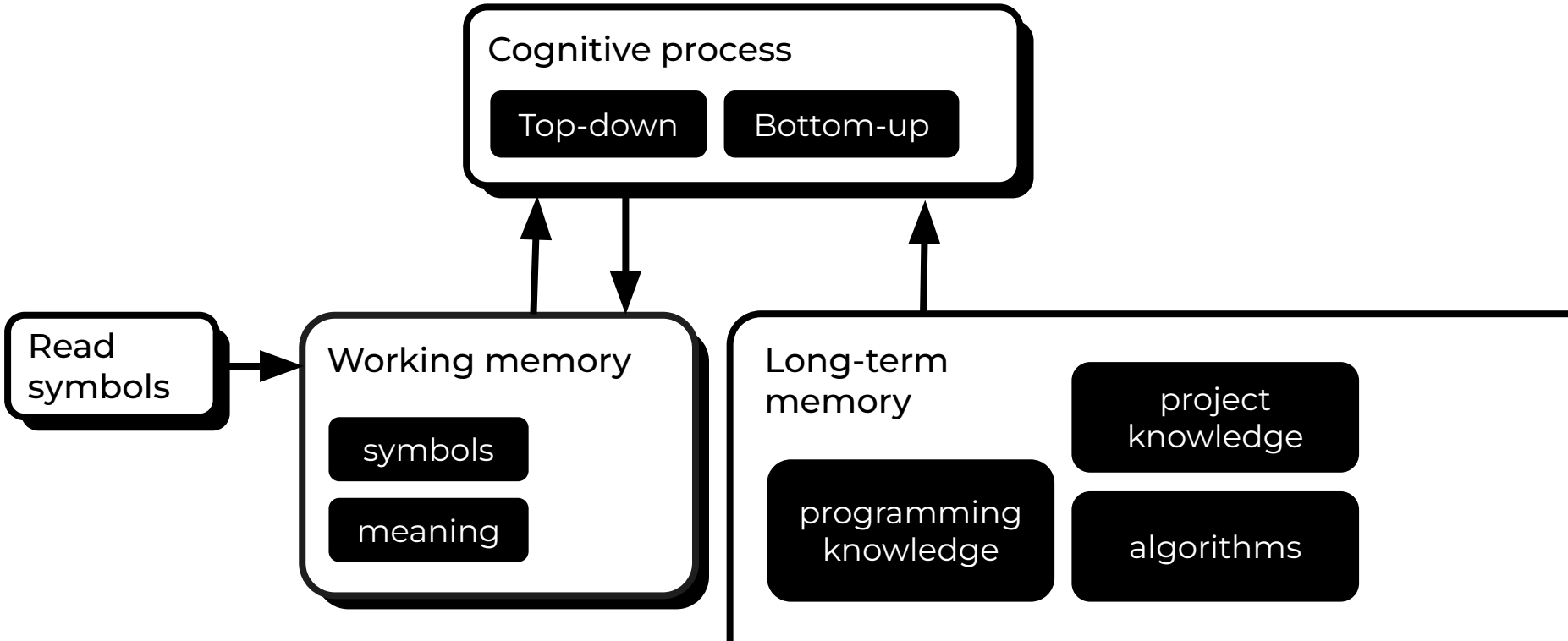


Too much for working memory

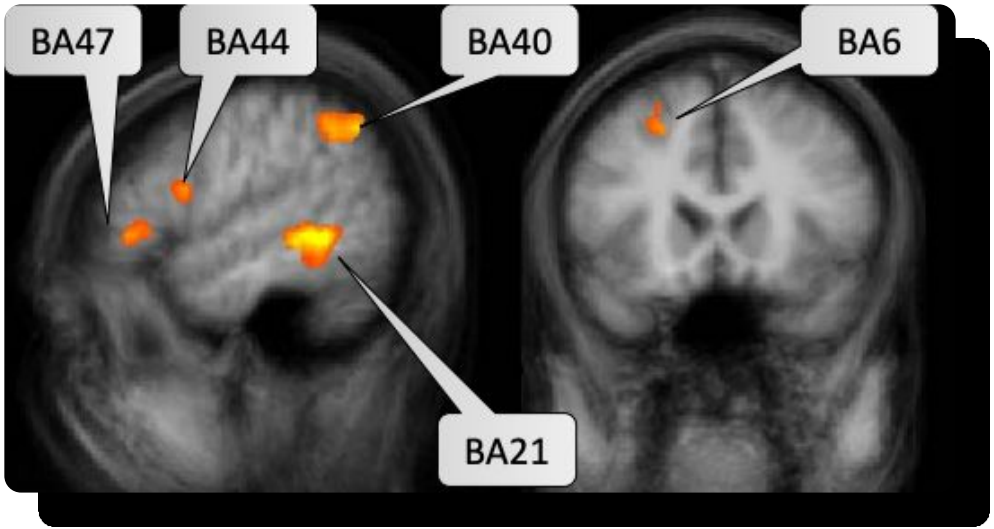
Fits in the working memory

Working memory has space for further code

How do we read code (simplified)

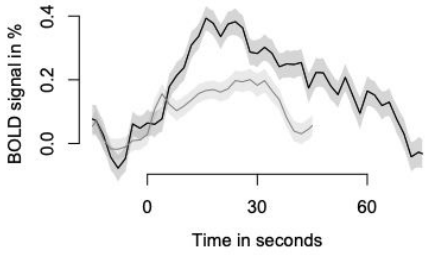


fMRIs



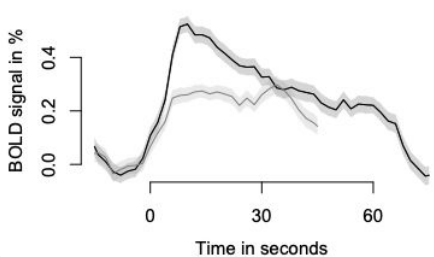
Blood oxygen level in brain while comprehending code

BA 40: Inferior parietal lobule
(Talairach coord.: -51, -49, 41; cluster size: 3368)

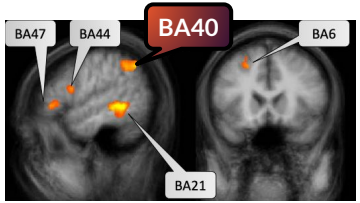


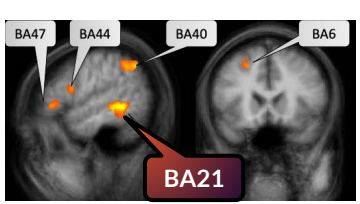
Working memory
Verbal/numeric
Problem solving

BA 21: Middle temporal gyrus
(Talairach coord.: -55, -39, -2; cluster size: 4746)



Semantic memory retrieval
Categorization





Proposed model

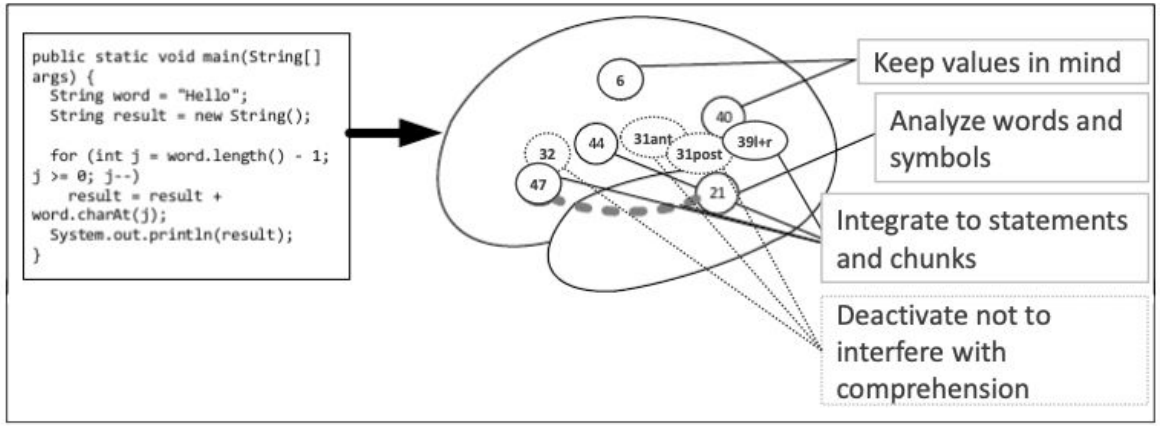


Fig. 10. Visualization of how bottom-up program comprehension might take place.

Take care of your colleagues' brain!

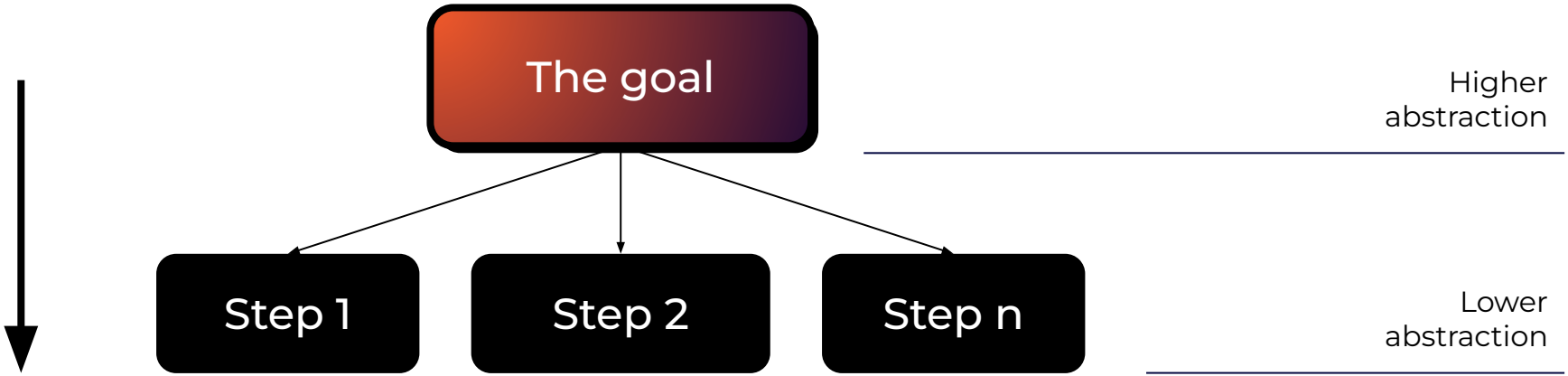
- Don't fill their working memory
- Allow top-down comprehension when possible
- Support swift bottom-up comprehension

Writing the code

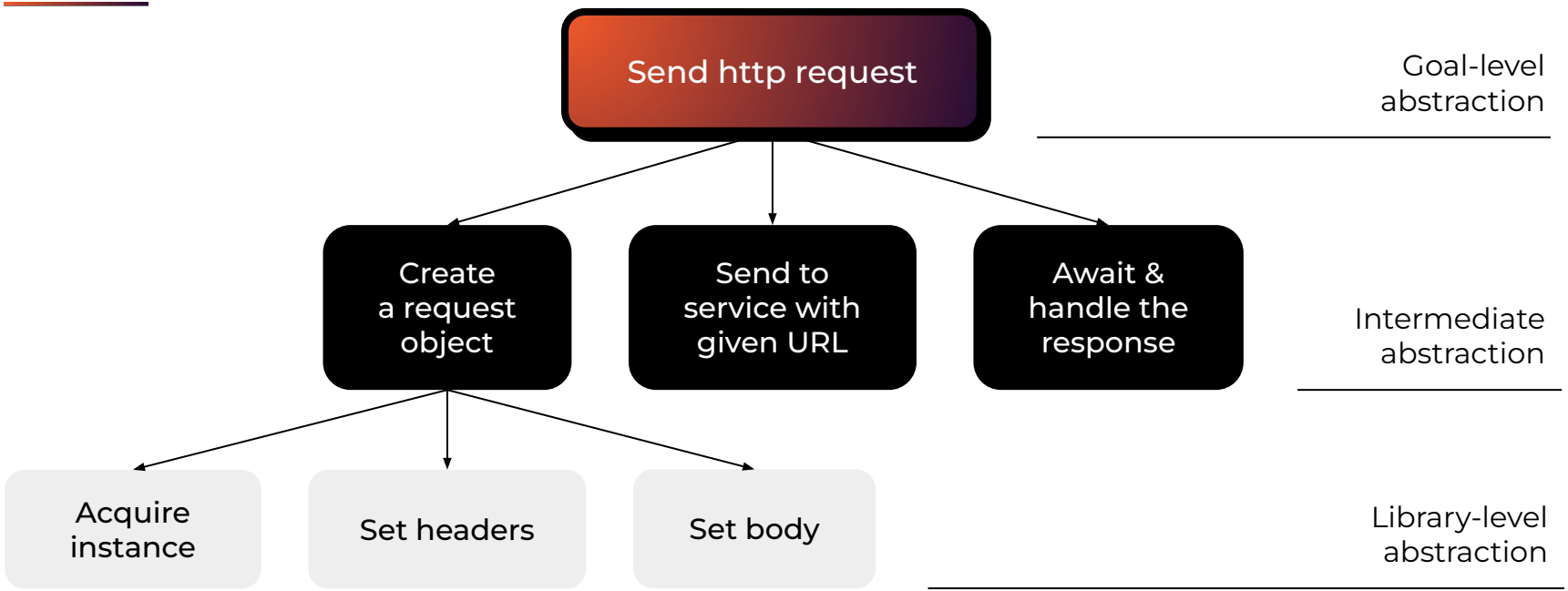
- Start with a goal (chunk)
- Finish with the code - structure of chunks, reducible to the goal chunk

Writing the code

- Chunking in the reverse direction



Example


Send http request
Create a request object
Send to service with given URL
Await & handle the response
Acquire instance
Set headers
Set body
Goal-level abstraction
Intermediate abstraction
Library-level abstraction

Abstraction rule of thumb

Avoid chunks that consist of more than 4-5 lower-level chunks

That will allow the user to reason about your abstraction with more ease!

Now let’s apply it

Scala Toolkit

Goal

- Ecosystem of battle-tested libraries
- Prioritising ease of use, good developer experience
- Really scalable experience - good for newcomers and experienced developers

Cognitive Dimensions

Cognitive Dimensions

- Framework to assess the cognitive load of a given code.
- Set of dimensions to assess cognitive load.

First step - Think about the target user!

- Code quality is subjective and the code should be tailored to the user's needs.
- **Target for Scala Toolkit**: Scripts, prototypes, simple services programmed by users that are not required to have a deep understanding of Scala language.

Second step - Define a test scenario

- Test case should be a description of a problem one is trying to solve
- For example: Read the whole file to find the word occurring most frequently

1. The Abstraction Level

The question:

What abstraction level would feel natural for our target user? Is the code written on this level of abstraction?

Example - Abstraction Level

```
extension s(s: String)
  def readFile[T[_]](using reader: FileReader[T]): T[String] = reader.readFile(s)

type Id[T] = T
trait FileReader[T[_]]:
  def readFile(path: String): T[String]

val syncFileReader: FileReader[Id] = path ⇒ ??? // read file synchronously
val asyncFileReader: FileReader[IO] = path ⇒ ??? //read file asynchronously
```

Excessive abstraction over the execution for our target user.

Cost of abstraction level

- **Long-term memory** - We require understanding of the given abstraction, i.e. over the execution model.
- **Working memory** - The selected model of execution has to be kept in memory.

Cost of abstraction level

- **Top-down and Bottom-up** - We need to constantly take the execution model into consideration during the cognitive process.

Clever abstraction is cool, but concentrate on its purpose and consequences

2. Role Expressiveness

The question:

Without experience working with the code, can one quickly recognize what each part of the code does?

Example - Role Expressiveness

```
!("http://example.com" ?% "user" =% user &% data)
```

Notations should be built on pre-existing knowledge of user:

```
post("http://example.com".withParam("user" -> user).withBody(data))
```

3. Visibility

The question:

How easy is it to discover this notation and follow its rules without changing context?

Other cognitive dimensions

- Consistency
- Domain Correspondence
- Conceptual Similarity to Ecosystem
- And others

Other analysis methods

- Language Level
- Structural measures
- Many others (for libraries): tests, responsiveness and availability of the maintainers, documentation, popularity, dependencies, dependencies stability, small size, API stability, versioning schema, cross-platform support, ...

Let’s apply it in Scala Toolkit

Applying cognitive dimensions

API Usability Rating

Process: Library Usability Measures

	Cognitive Dimensions						
Library	Abstraction Level	Consistency	Conceptual Similarity	Visibility	Domain Correspondence	Role Expressiveness	Language level
foo	0	1	1	1	1	1	0
bar	0	1	1	1	1	1	1

[Proposal] Add type hierarchy of requests and backends #1703

Merged adamw merged 34 commits into `softwaremill:master` from `adpi2:toolkit` 2 weeks ago

Process piping #200

Open szymon-rd wants to merge 6 commits into `com-lihaoyi:main` from `szymon-rd:process-pipeing`

Scala Toolkit

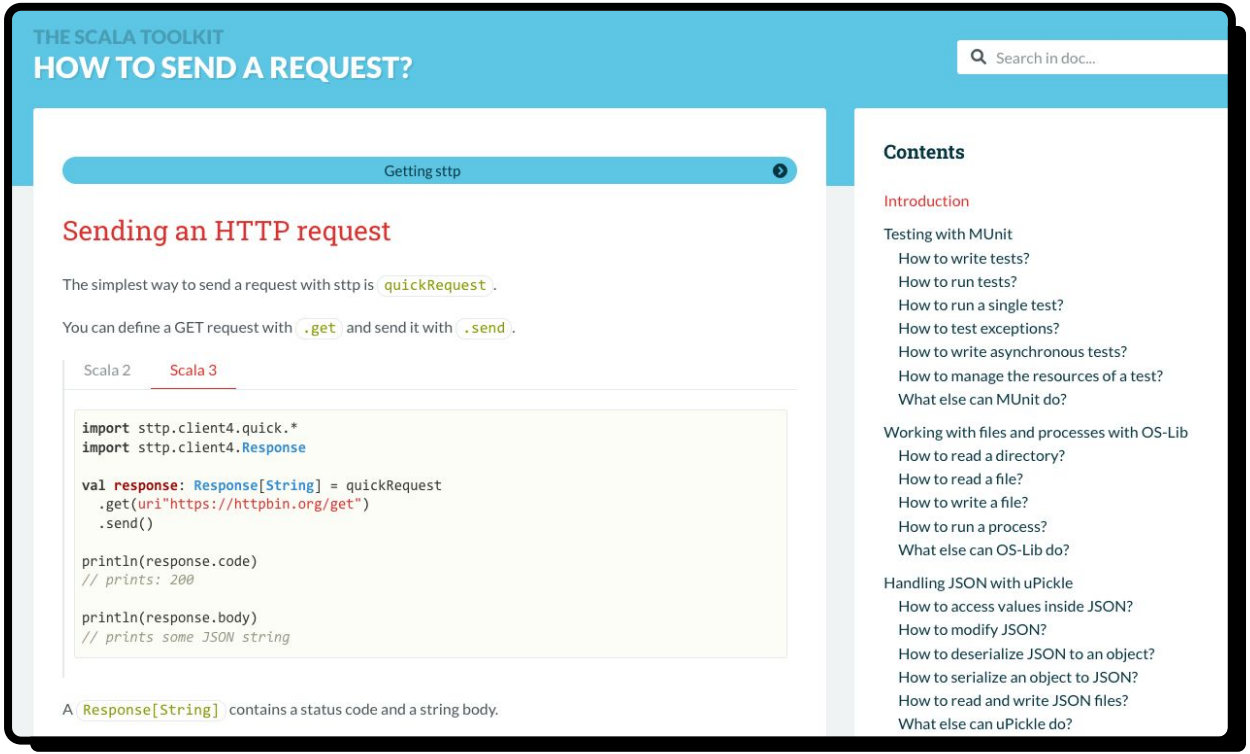
- Selected libraries:
  - JSON with *upickle*
  - HTTP with *sttp*
  - Files and shell with *os-lib*
  - Testing with *munit*

Apply to the whole experience

```
//> using toolkit "latest"
import sttp.client3._, sttp.client3.upicklejson._, upickle.default._

case class PetOwner(name: String, pet: String) derives ReadWriter
val petOwner = PetOwner("Peter", "Toolkitty")
val client = SimpleHttpClient()
val request = basicRequest.post(uri"https://example.com/").body(petOwner)
val response = client.send(request)
```

Q&A-like tutorials



Compatibility taken seriously

- Strict tests run on the whole Toolkit dependency graph
- Ensuring semver compliance and generating clear diffs

Compatibility taken seriously

Changelog for toolkit-test 0.2.1

Changes to direct dependencies

- Updated `org.scala-lang:toolkit_2.13:0.2.0` from `0.2.0` to `0.2.1` under `org.scala-lang:toolkit-test_2.13:0.2.1`
- Updated `org.scalameta:munit_2.13:1.0.0-M6` from `1.0.0-M6` to `1.0.0-M7` under `org.scala-lang:toolkit-test_2.13:0.2.1`

Changes to transitive dependencies

- Updated `com.softwaremill.sttp.client4:core_2.13:4.0.0-M1` from `4.0.0-M1` to `4.0.0-M2` under `org.scala-lang:toolkit_2.13:0.2.1`

Full dependency tree

- org.scala-lang:toolkit-test_2.13:0.2.1
  - org.scala-lang:toolkit_2.13:0.2.1
    - com.lihaoyi:os-lib_2.13:0.9.1
      - com.lihaoyi:geny_2.13:1.0.0
    - com.lihaoyi:upickle_2.13:3.1.0
      - com.lihaoyi:ujson_2.13:3.1.0

Typelevel Toolkit

- Toolkit is a standard rather than a single tool
- Typelevel created their own Toolkit already

Take part in the Toolkit

- Create issues with proposal in the Scala Toolkit github repo
- Take on tasks in Toolkit libraries
- Take part in the discussion on Discord

World with empirically-based APIs and language design

References

- N. Peitek et al., "A Look into Programmers' Heads," in IEEE Transactions on Software Engineering, vol. 46, no. 4, pp. 442-462, 1 April 2020, doi: 10.1109/TSE.2018.2863303.
- Anna A Ivanova, et al., (2020) Comprehension of computer code relies primarily on domain-general executive brain regions, eLife 9:e58906
- Siegmund, et al., (2017). Measuring neural efficiency of program comprehension. 140-150. 10.1145/3106237.3106268.
- Huang, et al., (2019). Distilling Neural Representations of Data Structure Manipulation using fMRI and fNIRS. 396-407. 10.1109/ICSE.2019.00053.

References

- Fakhoury, et al., (2020). Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization. Empirical Software Engineering. 25. 10.1007/s10664-019-09751-4.
- Shneiderman, Ben & Mayer, Richard. (1979). Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. International Journal of Parallel Programming. 8. 219-238. 10.1007/BF00977789.
- Maskeri, Girish & Kak, Avinash. (2015). Some structural measures of API usability. Software: Practice and Experience. 45. 10.1002/spe.2215.

References

- Maskeri, Girish & Kak, Avinash. (2015). Some structural measures of API usability. Software: Practice and Experience. 45. 10.1002/spe.2215.

Thank you for your attention!

Szymon Rodziewicz