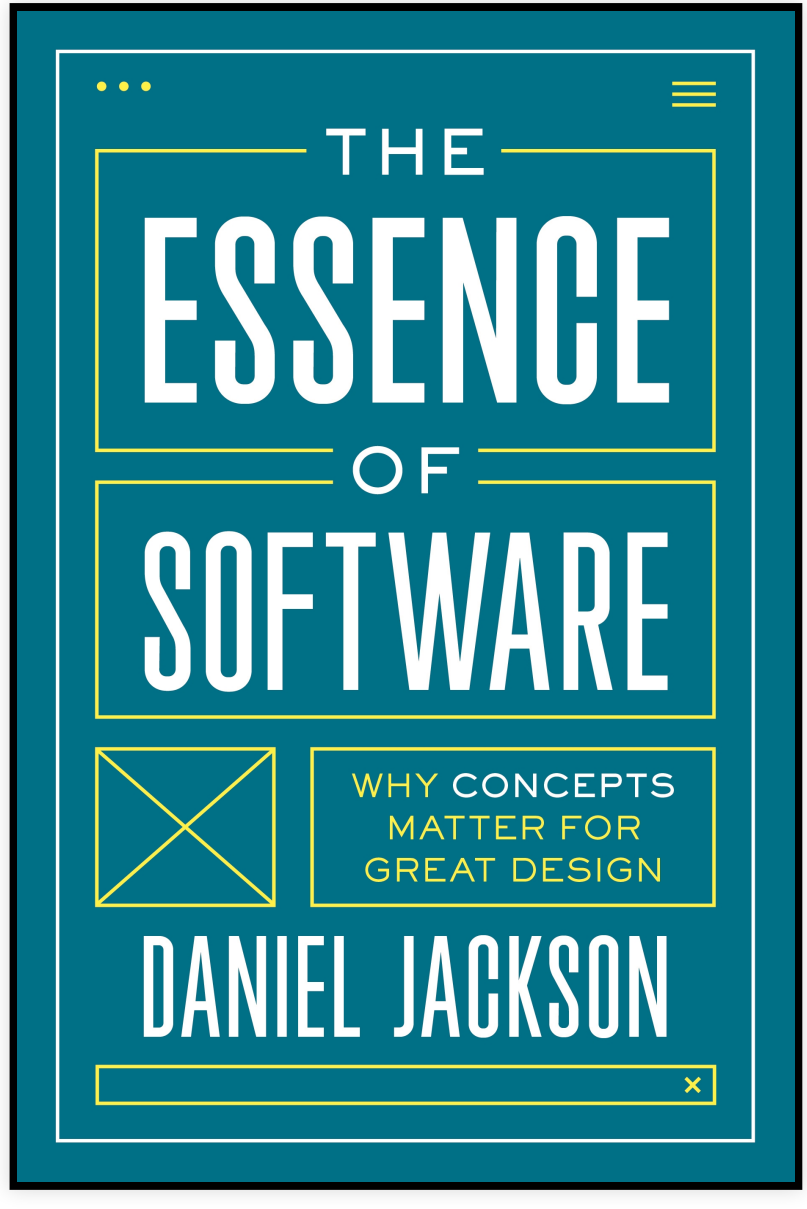
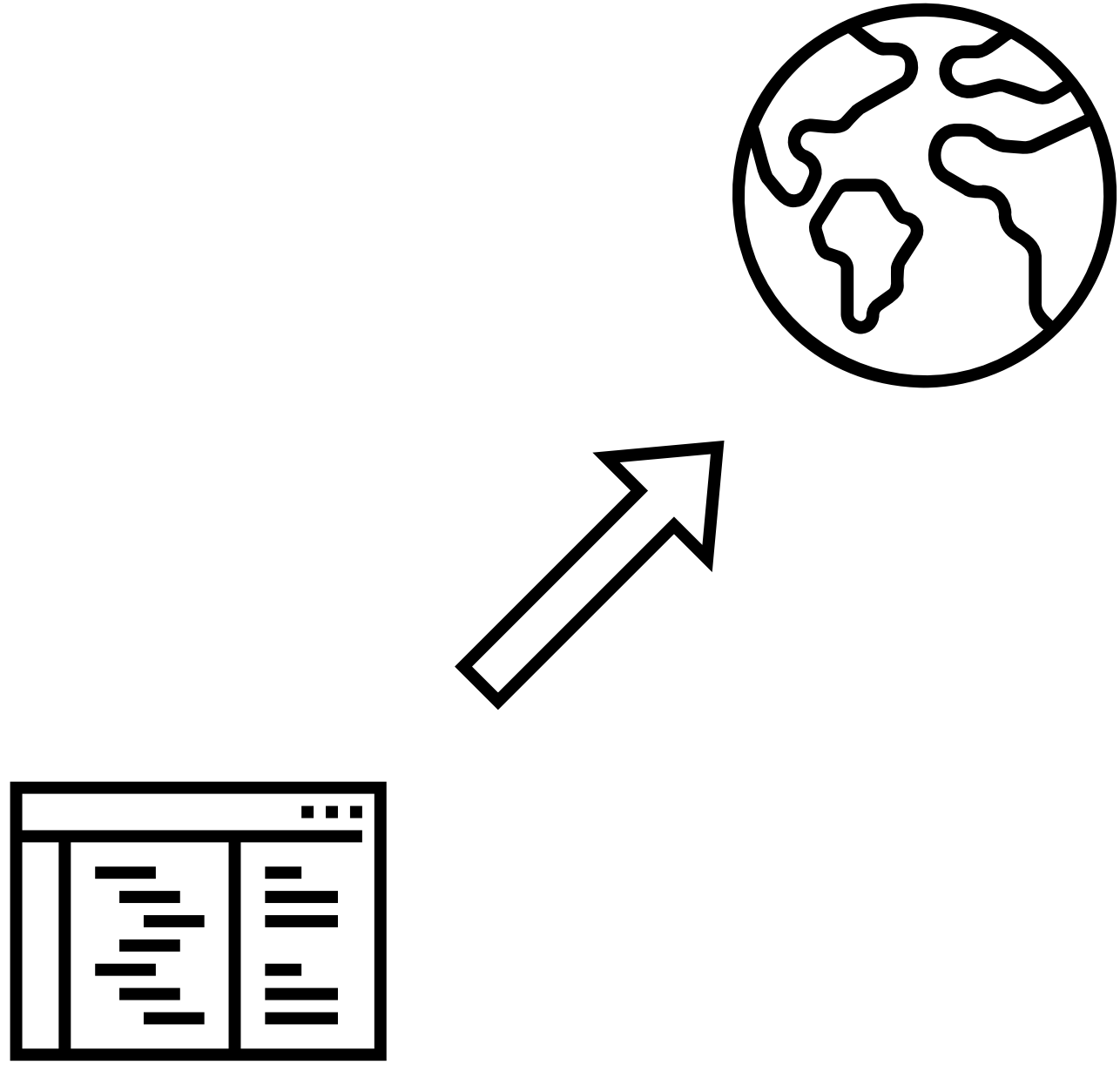
Types ⇔ Design

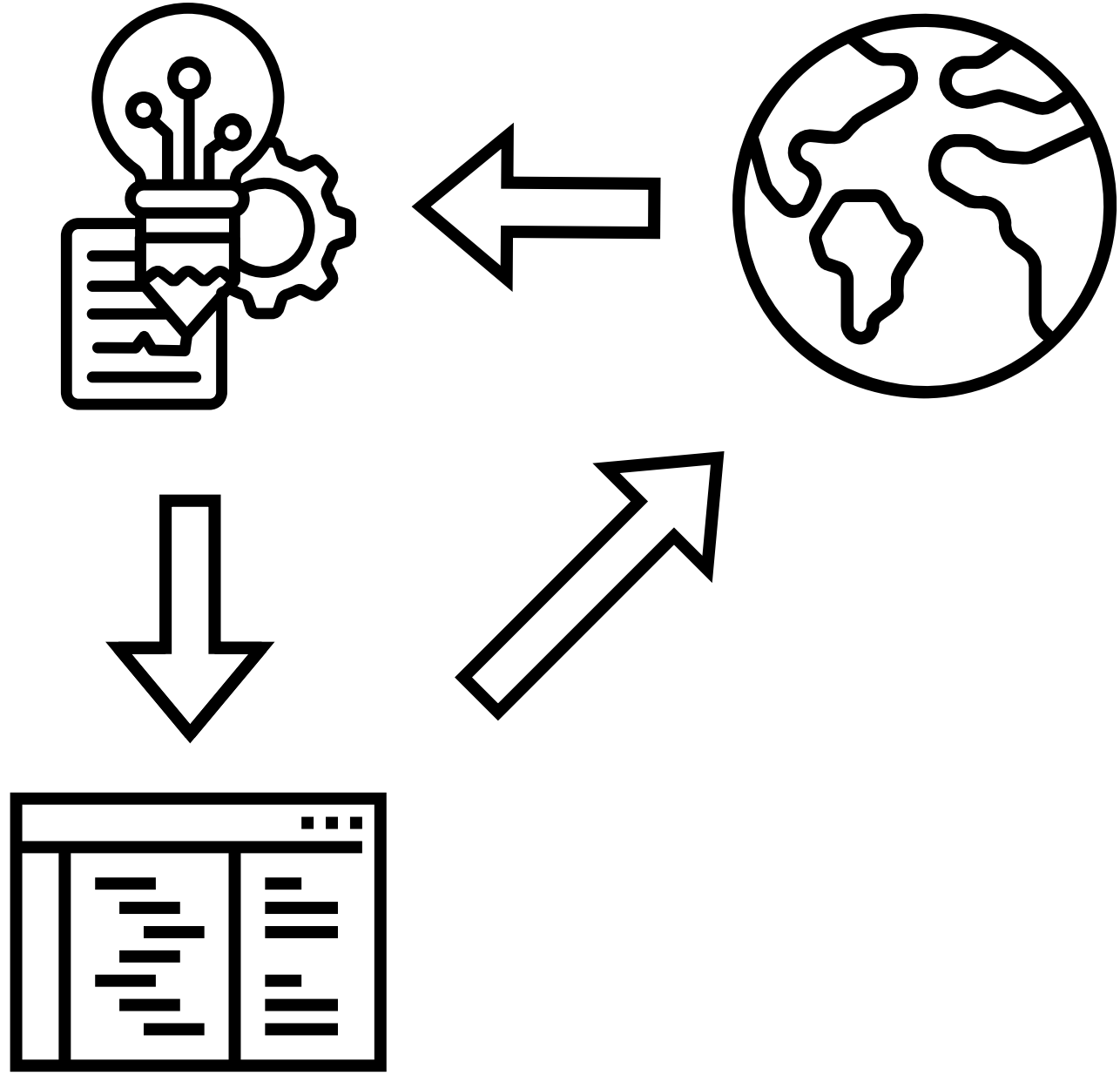
Tikhon Jelvis

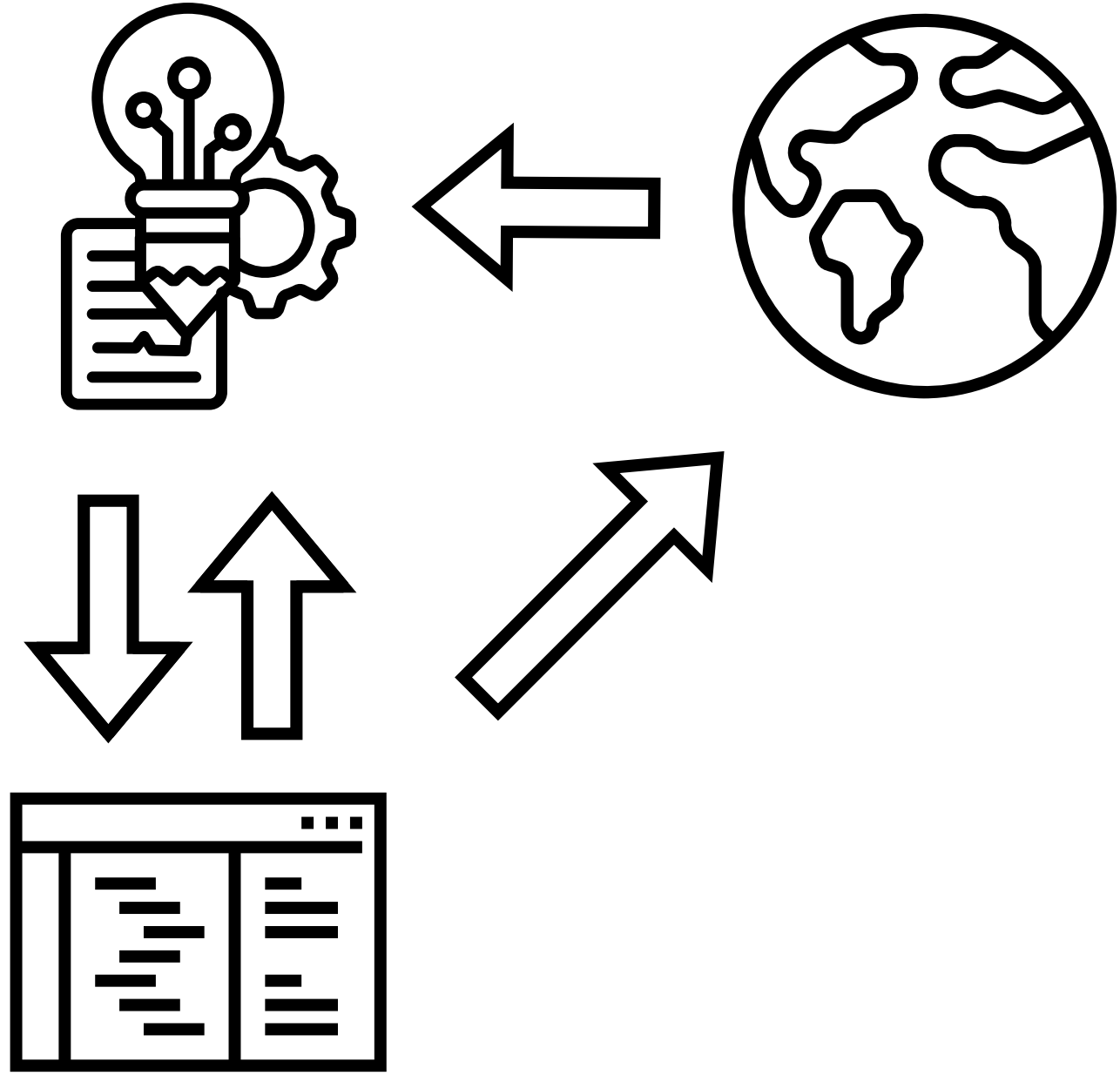
What *is* code design?


A PHILOSOPHY OF SOFTWARE DESIGN
JOHN OUSTERHOUT
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


THE
ESSENCE
OF
SOFTWARE
WHY CONCEPTS MATTER FOR GREAT DESIGN
DANIEL JACKSON

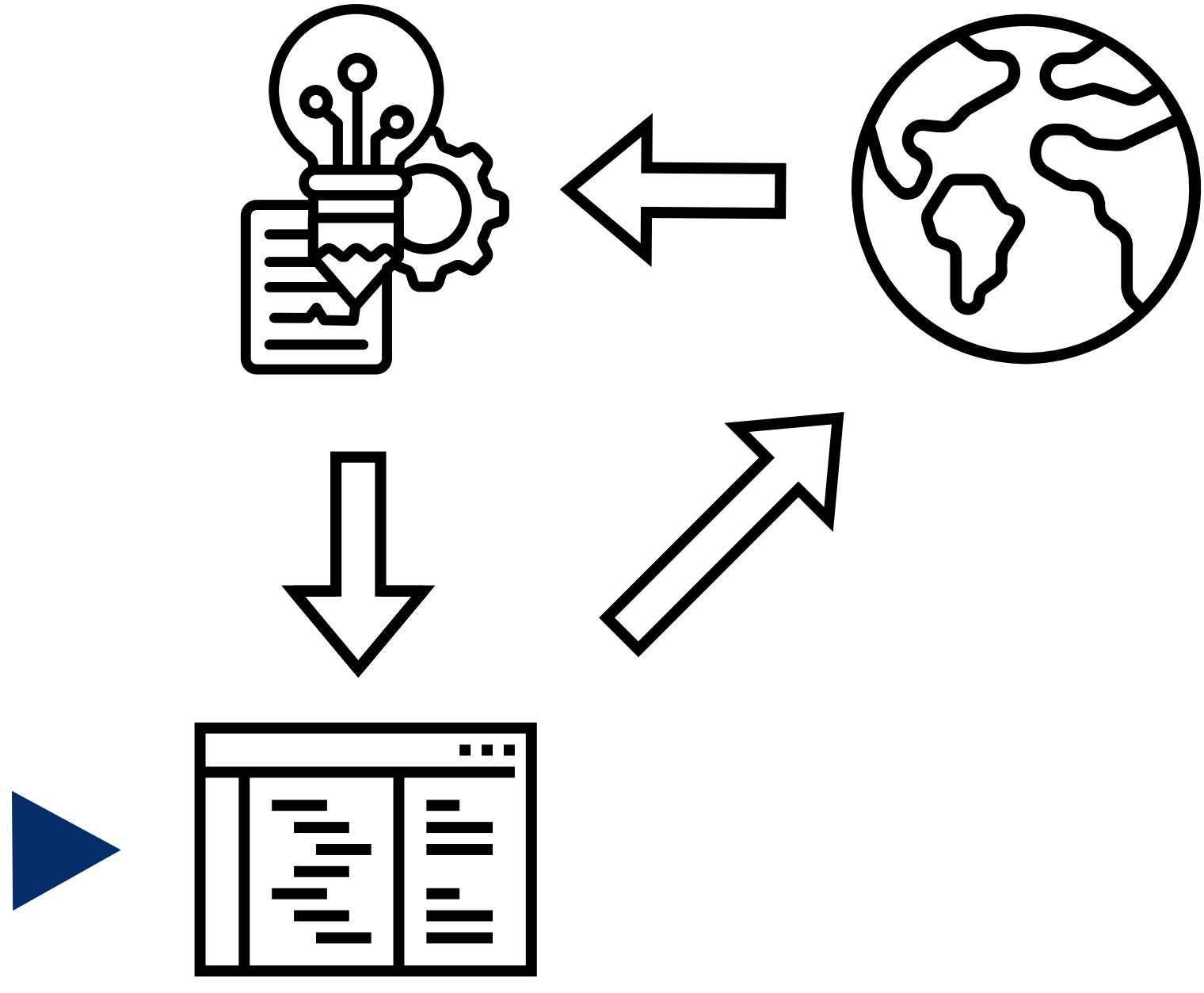


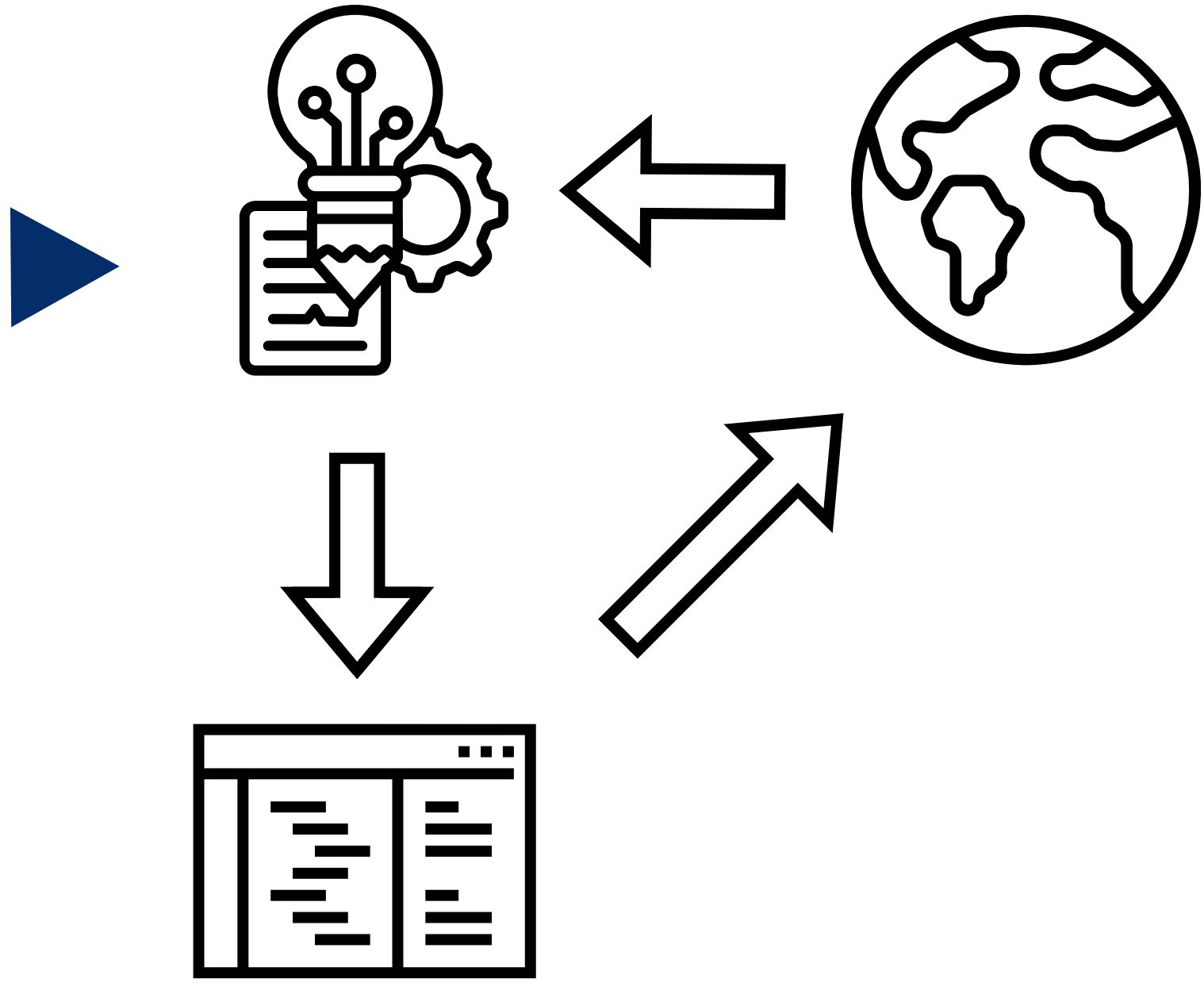


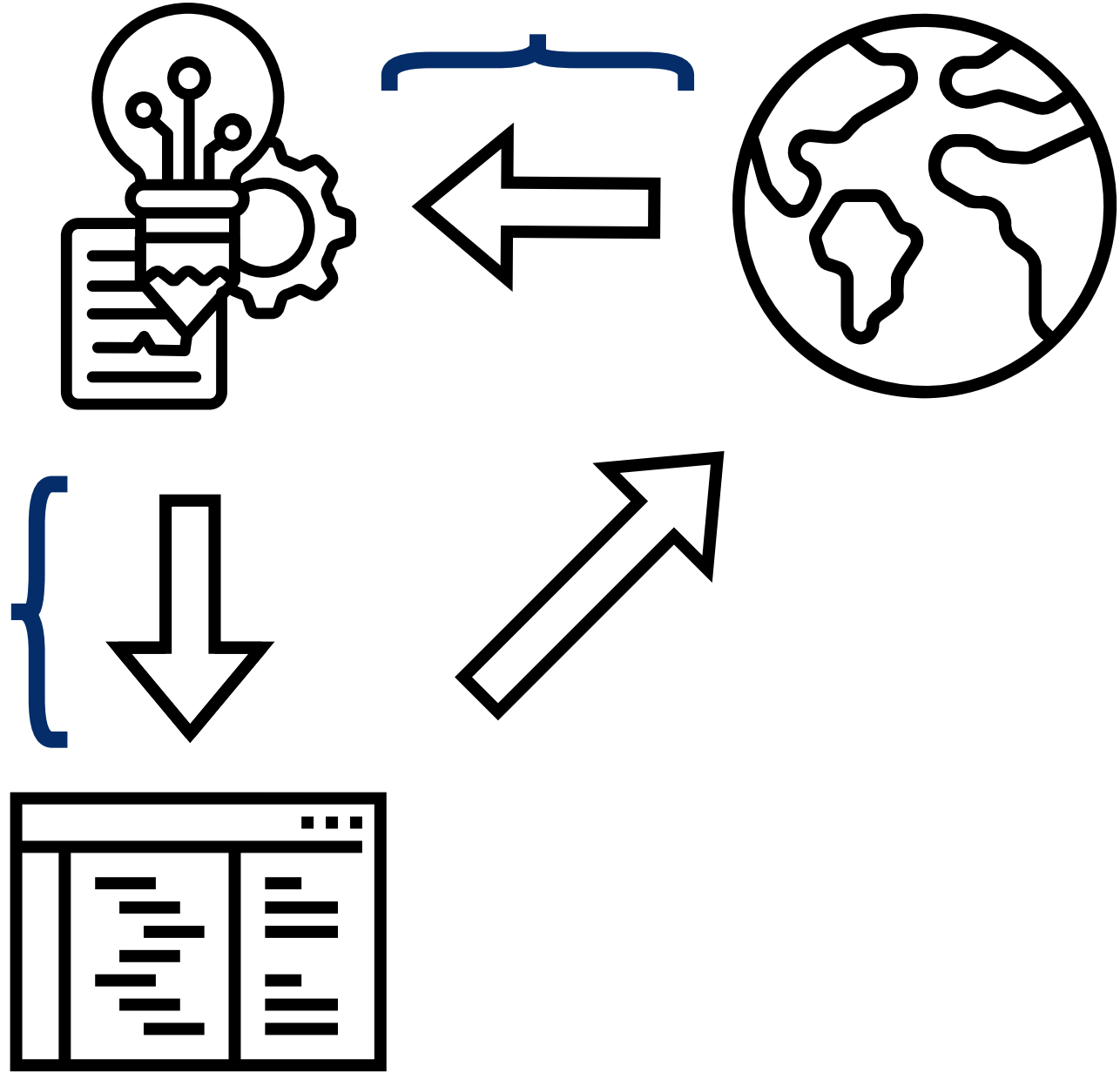


“Programming *is* planning.”

“Programming *is* planning.”
Programming *is* design.







Posted by u/Barafu 3 years ago

1.7k

How to delete all your files

A coworker has shown me a trick today. Imagine you have a nfs share with a folder "Documents". You want to copy them to your machine. So you type

```
cd /mnt/nfs/Documents
rsync -r * ~/Documents
```

And bam!!! - all you local documents are deleted. Why? Because the share contains a file named exactly "`--delete`", and since it gets sorted first, rsync thinks it to be an argument and deletes everything.

Lesson1: never start file names with a dash.

Lesson2: always use `--` to explicitly separate paths from arguments in scripts.

Lesson3: never use only `*` as a wildcard. A `./*` would save you in most cases, though commands that accept arguments between file names can still be tricked.

287 Comments Share Save •••

Comment as tikhonjelvis

What are your thoughts?

```
rsync -r --delete ~/path/a ~/path/b
```

```
rsync -r --delete ~/path/a ~/path/b
```

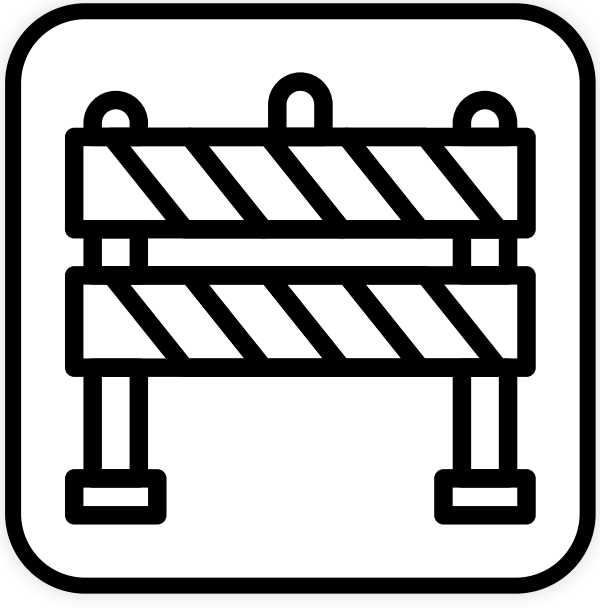
```
rsync -r --delete ~/path/a ~/path/b
```

```
rsync -r * ~/out
```

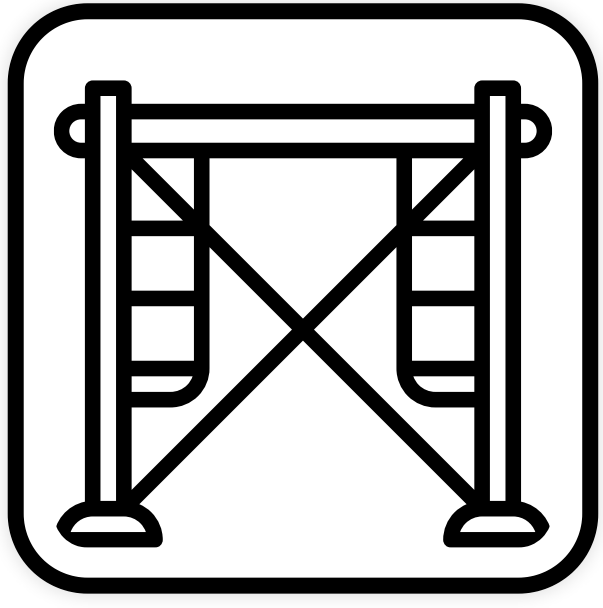
```
rsync -r ~/path/a ~/path/b
```


/home/tikhon/tmp:
bypass_block.png
--delete
foo
robot.png

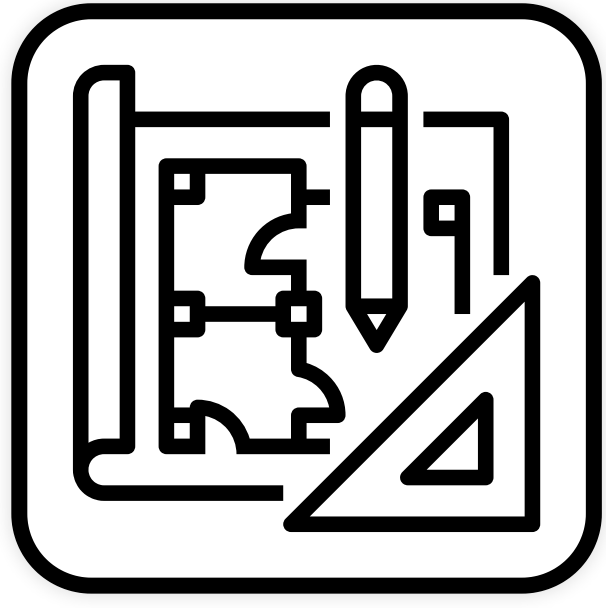
What are types *for*?



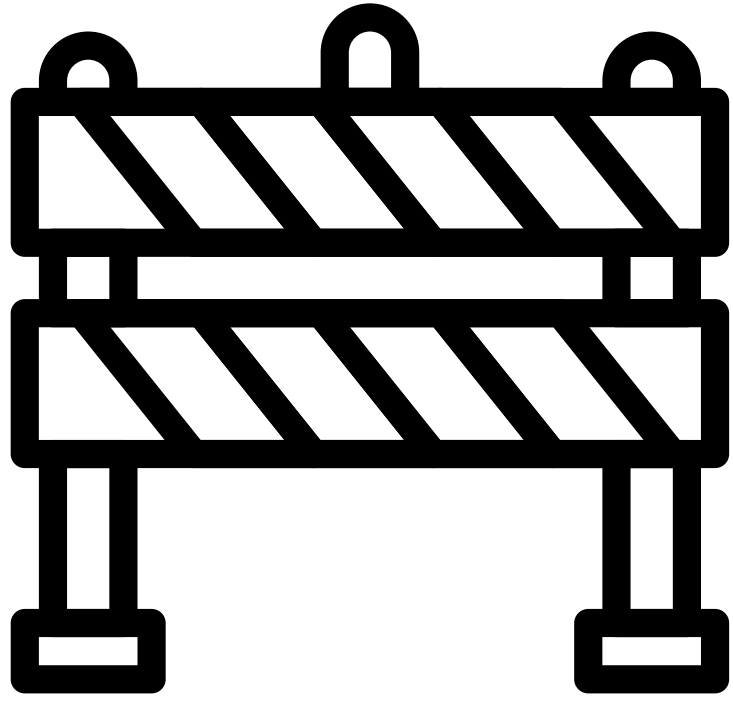
bugs



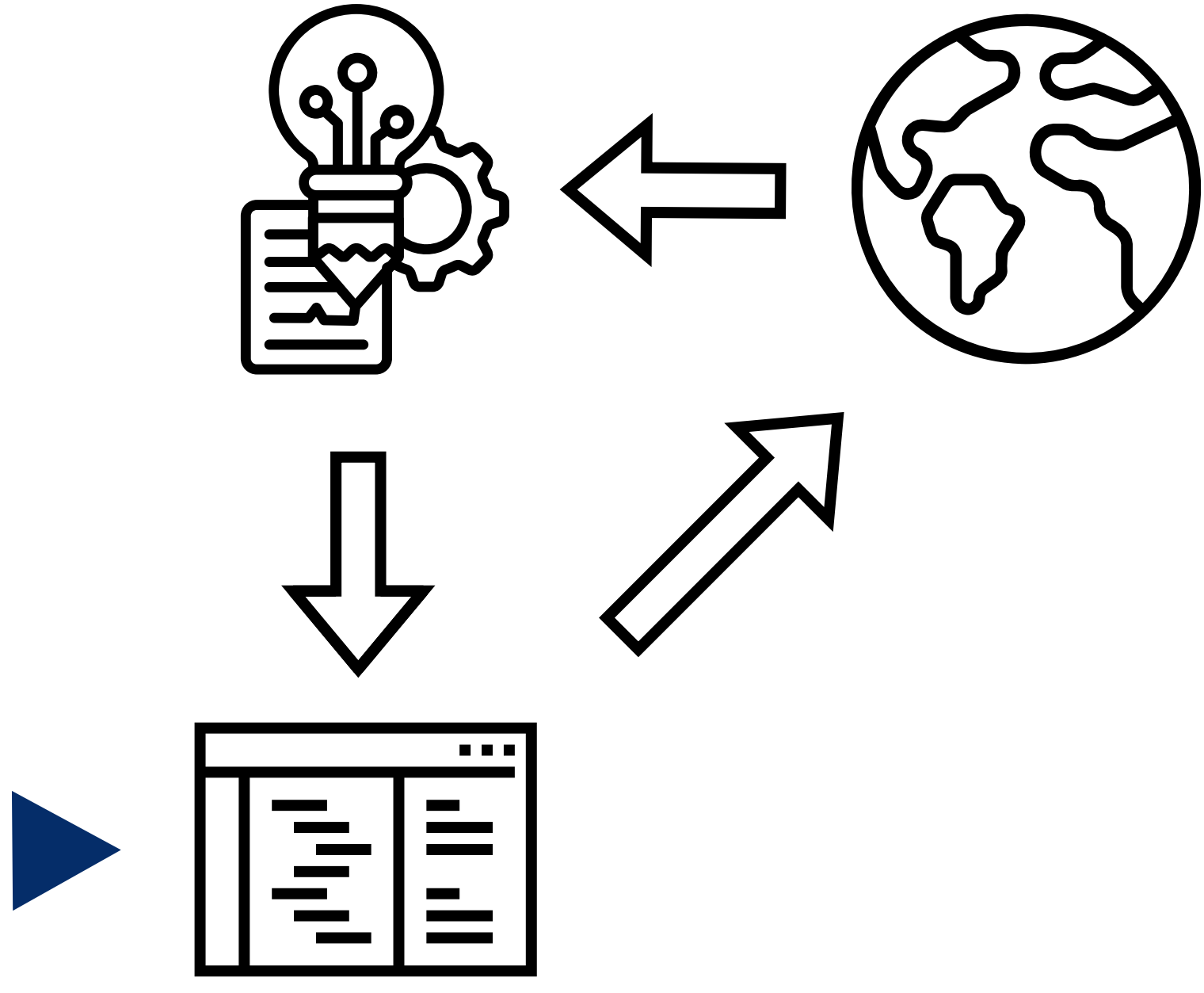
structure

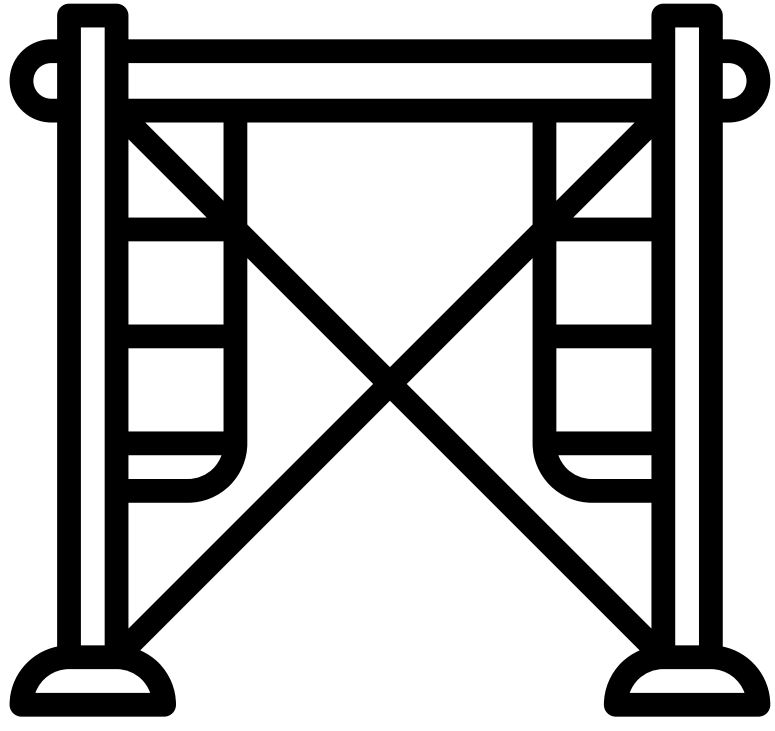


design

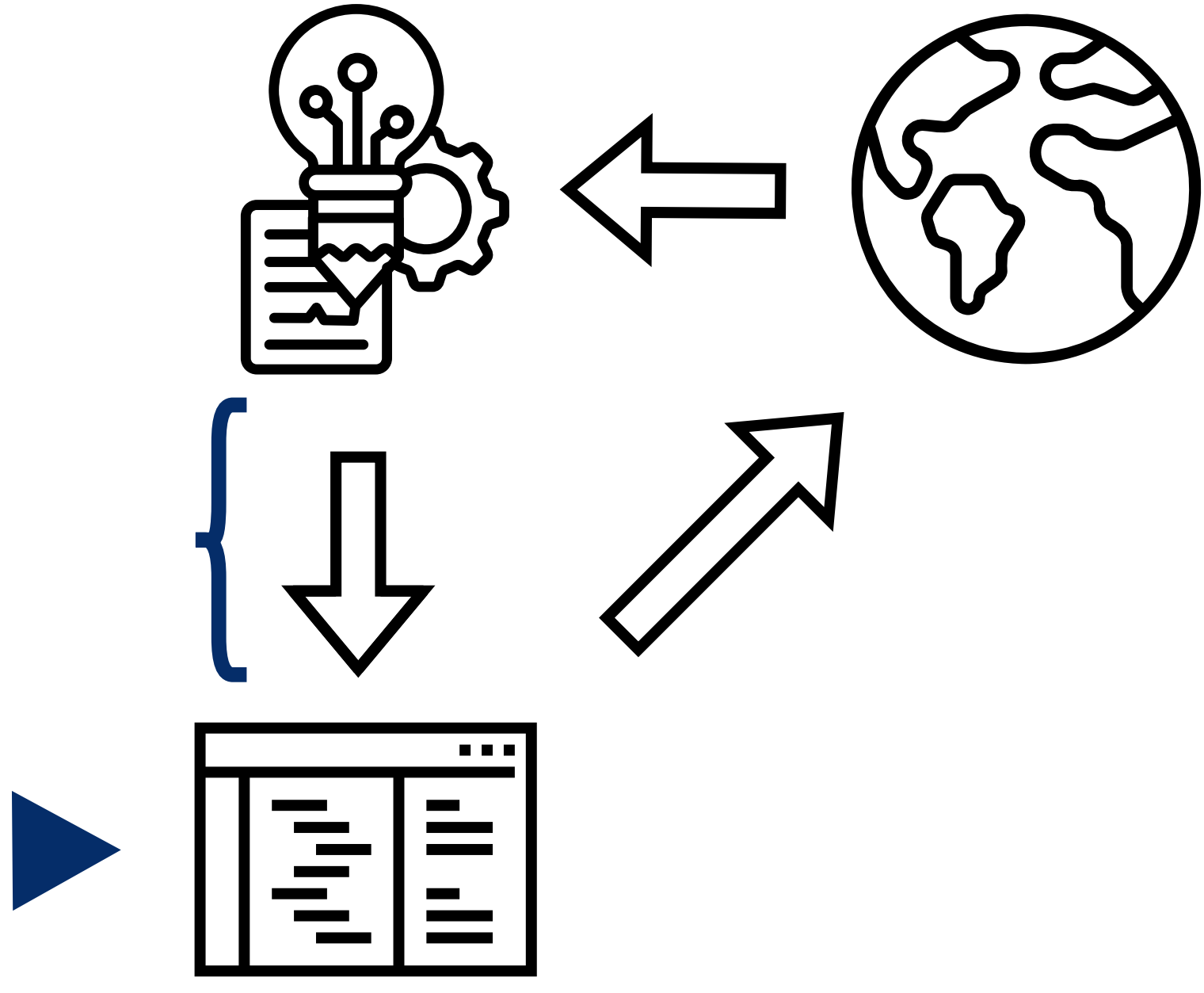


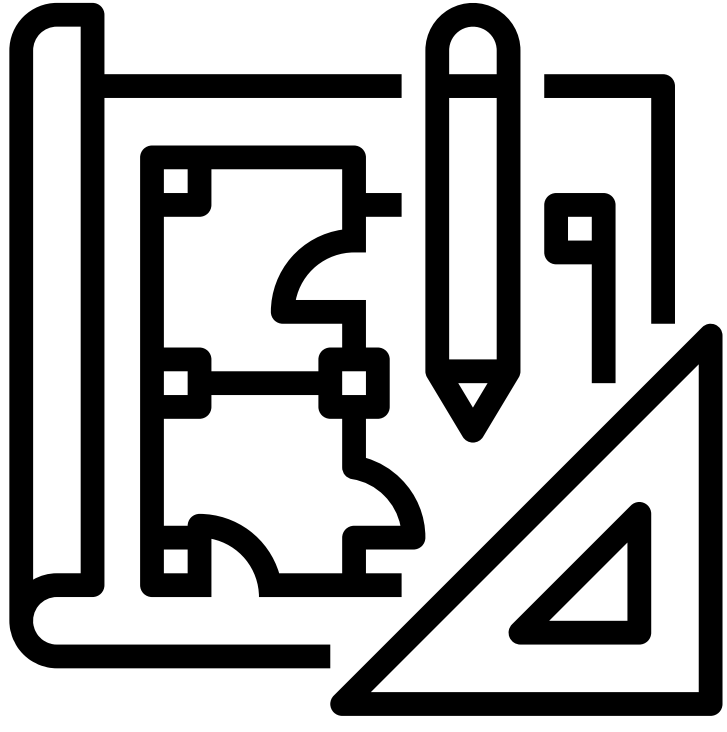
“Types catch mistakes”



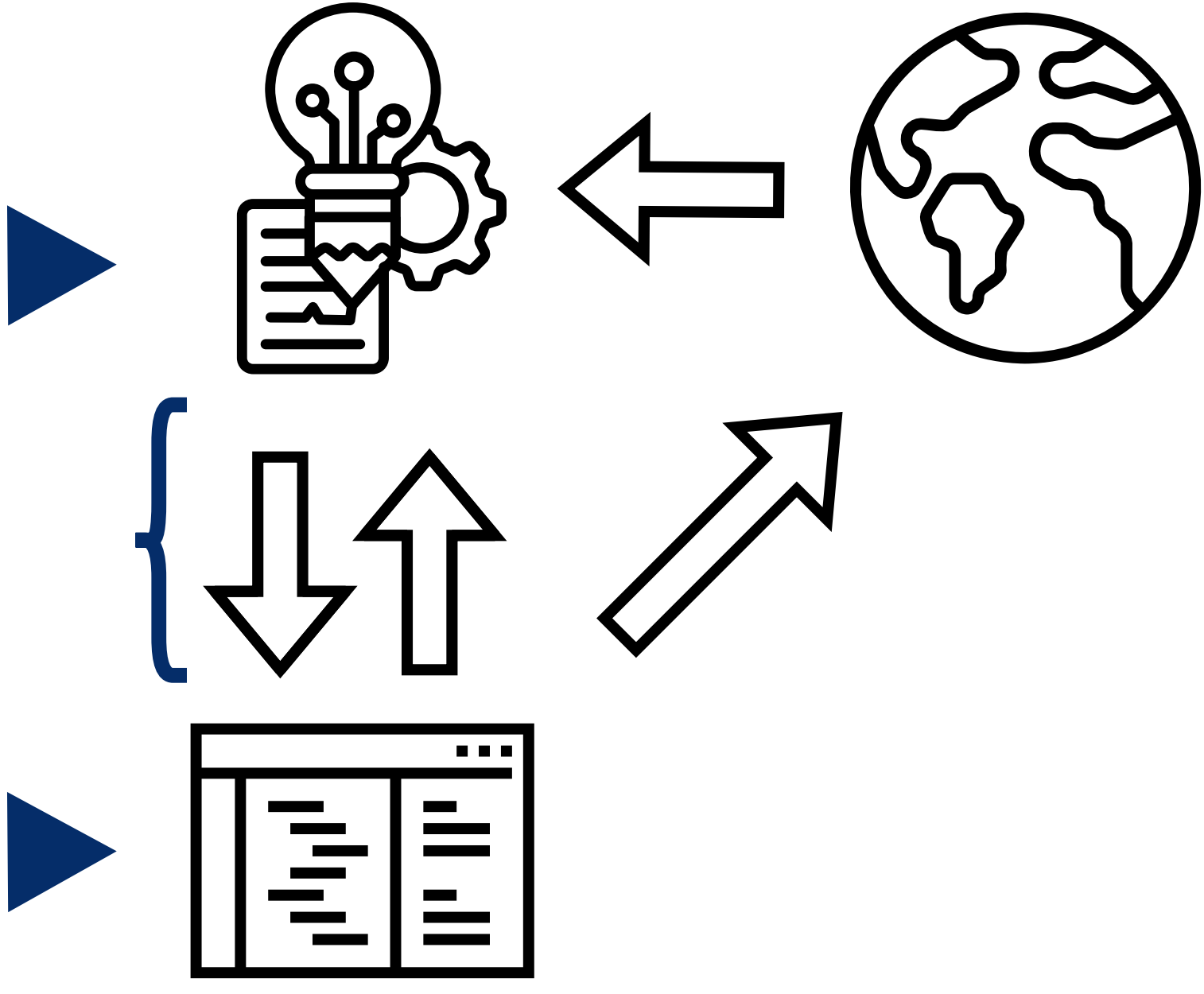


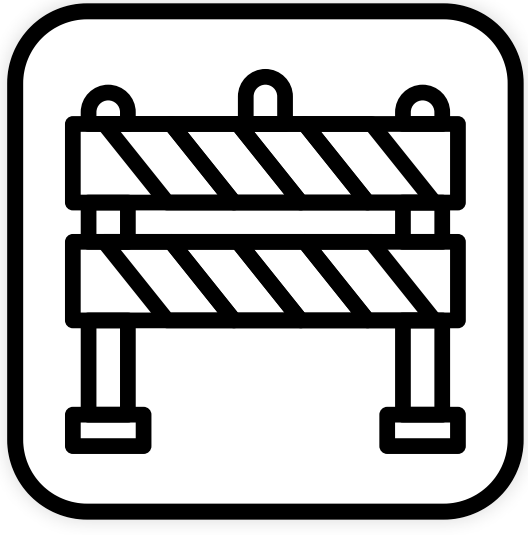
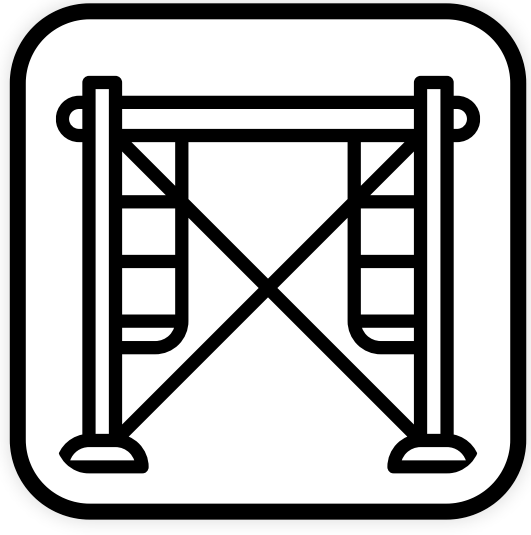
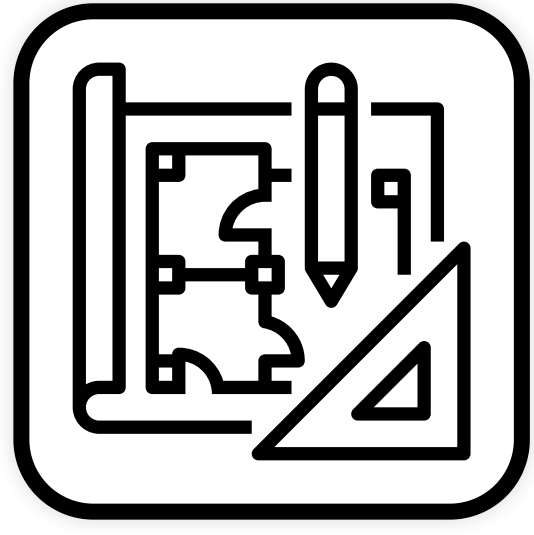
“Types structure my code”





“Types help me think”

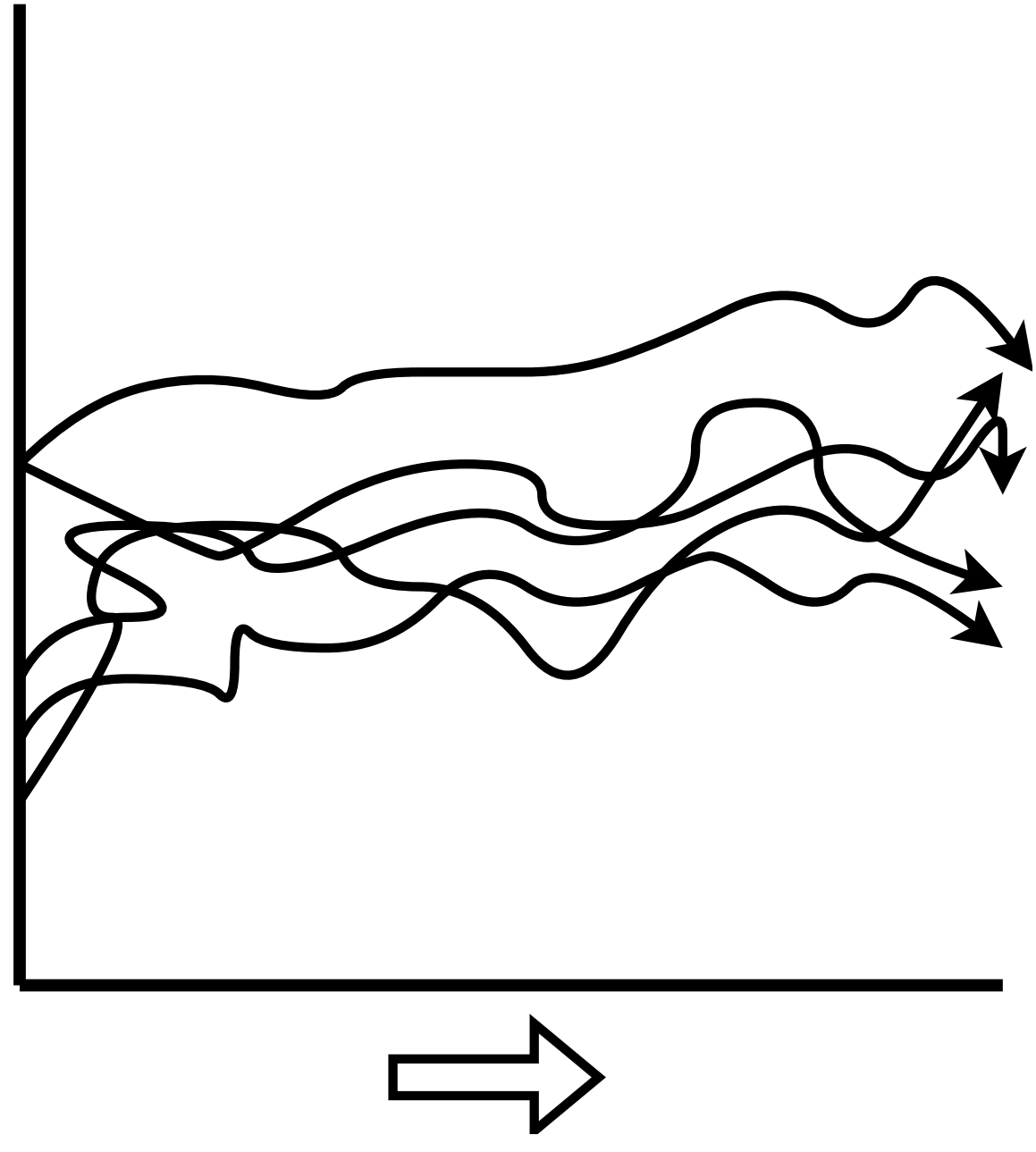


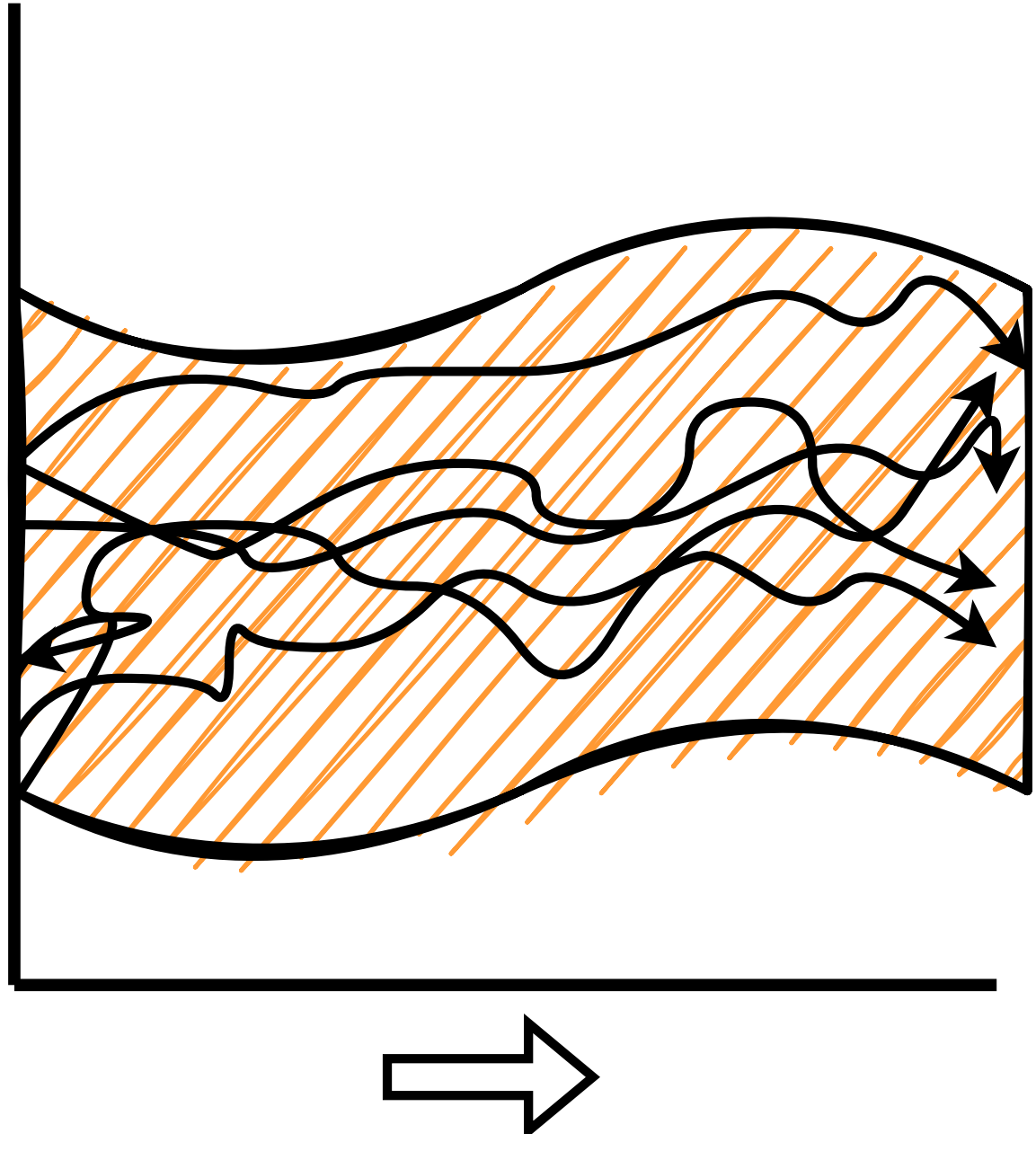




Information hiding:
abstract over **implementation**

::

Static types:
abstract over **runtime**





a different sort of abstraction

```
encode :: Image RGB -> ByteString
encode = _
```

```haskell
encode :: Image RGB -> ByteString
encode = _
```

```haskell
colorSpace :: Image RGB -> Image YCbCr
colorSpace = _
```

```haskell
downsample :: Image YCbCr
           -> Downsampled
downsample = _
```

```haskell
blocks :: Downsampled
       -> [Downsampled]
blocks = _
```

```haskell
data Image space = Image
  { pixels :: Vector space }

data Downsampled = Downsampled
  { y  :: Vector Word8
  , cb :: Vector Word8
  , cr :: Vector Word8
  }
```

FOUNDATIONS OF REINFORCEMENT LEARNING WITH APPLICATIONS IN FINANCE

Ashwin Rao • Tikhon Jelvis

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Markov Decision Process

A Markov decision process (MDP) is a Markov reward process with decisions. It is an *environment* in which all states are Markov.

Definition

A *Markov Decision Process* is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- $\mathcal{P}$ is a state transition probability matrix, $\mathcal{P}^{a}_{ss'} = \mathbb{P}\left[S_{t+1} = s' \mid S_t = s, A_t = a\right]$
- $\mathcal{R}$ is a reward function, $\mathcal{R}^{a}_{s} = \mathbb{E}\left[R_{t+1} \mid S_t = s, A_t = a\right]$
- γ is a discount factor $\gamma \in [0, 1]$.

Markov Process

A Markov process is a memoryless random process, i.e. a sequence of random states $S_1, S_2, \ldots$ with the Markov property.

Definition

A *Markov Process* (or *Markov Chain*) is a tuple $\langle \mathcal{S}, \mathcal{P} \rangle$

- $\mathcal{S}$ is a (finite) set of states
- $\mathcal{P}$ is a state transition probability matrix,
  $\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$

Markov Reward Process

```
data MarkovProcess m s = MP {
  step :: s -> m s
}

data MarkovRewardProcess m s = MRP {
  step :: s -> m (r, s)
}

data MDP m a s = MDP {
  step :: s -> a -> m (r, s)
}

type Policy s a = s -> a
```

```haskell
data MarkovProcess m s = MP {
  step :: s -> m s
}

type MarkovRewardProcess m s =
  MarkovProcess (WriterT Reward m) s
```

```haskell
data MDP m a s = MDP {
  step :: s -> a -> m s
}

apply :: MDP m a s
      -> Policy s a
      -> MarkovProcess m s
```

```
          ▽
data MDP m a s = MDP {
  step :: s -> a -> m s
}
```


▽
data MDP m a s = MDP {
step :: s -> a -> m s
}

```
                ▽
data MDP m a s = MDP {
  step :: s -> a -> m s
}
```

So, what did static typing get us?

simplify our **conceptual model**

simplify our **conceptual model**

feedback **before we had runnable code**

simplify our **conceptual model**

feedback **before we had runnable code**

interactive support as we program

simplify our **conceptual model**

feedback **before we had runnable code**

interactive support as we program

high-level guide to our conceptual model

Where does this take us?

Type-Driven Development

Type-Driven Development

Concepts ⇒ Types ⇒ Code

How Statically-Typed Functional Programmers Write Code

JUSTIN LUBIN, University of California, Berkeley, USA
SARAH E. CHASINS, University of California, Berkeley, USA

How working statically-typed functional programmers write code is largely understudied. And yet, a better understanding of developer practices could pave the way for the design of more useful and usable tooling, more ergonomic languages, and more effective on-ramps into programming communities. The goal of this work is to address this knowledge gap: to better understand the high-level authoring patterns that statically-typed functional programmers employ. We conducted a grounded theory analysis of 30 programming sessions of practicing statically-typed functional programmers, 15 of which also included a semi-structured interview. The theory we developed gives insight into how the specific affordances of statically-typed functional programming affect domain modeling, type construction, focusing techniques, exploratory and reasoning strategies, and expressions of intent. We conducted a set of quantitative lab experiments to validate our findings, including that statically-typed functional programmers often iterate between editing types and expressions, that they often run their compiler on code even when they know it will not successfully compile, and that they make textual program edits that reliably signal future edits that they intend to make. Lastly, we outline the implications of our findings for language and tool design. The success of this approach in revealing program authorship patterns suggests that the same methodology could be used to study other understudied programmer populations.

CCS Concepts: • **Human-centered computing** → **HCI theory, concepts and models**; • **Software and its engineering** → **Functional languages**.

Additional Key Words and Phrases: static types, functional programming, grounded theory, need-finding, interviews, qualitative, quantitative, mixed methods, randomized controlled trial

ACM Reference Format:
Justin Lubin and Sarah E. Chasins. 2021. How Statically-Typed Functional Programmers Write Code. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 155 (October 2021), 30 pages. https://doi.org/10.1145/3485532

1 INTRODUCTION

Statically-typed functional programming languages like Haskell, OCaml, Elm, F#, and SML offer features and norms—like expressive type systems, strong static guarantees, and an emphasis on small and reusable functions free of side effects—that differentiate them from other classes of languages. These attributes are different enough from those found in more mainstream languages that they engender distinct modes of interaction between statically-typed functional programmers and their language, environment, and tooling. The aim of this work is to understand how the specific affordances of statically-typed functional programming affect the way programmers author code. The end goal is to deepen our understanding of an understudied programmer population, lay the foundation for evidence-based design of useful and usable languages and tools, and elucidate tacit community knowledge, which could ease the onboarding of new members to the community.

Authors' addresses: Justin Lubin, justinlubin@berkeley.edu, University of California, Berkeley, Berkeley, California, USA; Sarah E. Chasins, schasins@cs.berkeley.edu, University of California, Berkeley, Berkeley, California, USA.

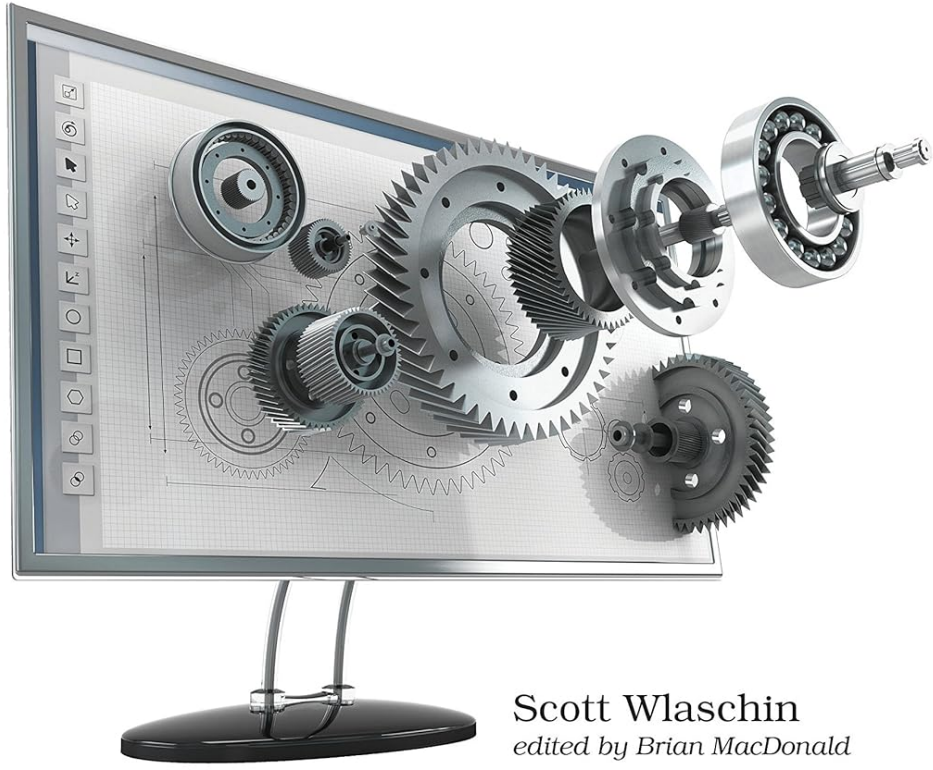
2475-1421/2021/10-ART155
https://doi.org/10.1145/3485532

Domain Driven Design?

The Pragmatic Programmers

Domain Modeling Made Functional

Tackle Software Complexity with Domain-Driven Design and F#



Scott Wlaschin
edited by Brian MacDonald

Dependent Types?

Type-Driven Development with Idris

Edwin Brady

MANNING

LLM Code Generation

Today: boilerplate, bad code, no design, no verification

Tomorrow: better tools, good design, verification?

Our tools should help us think

Created by Tikhon Jelvis.