Refactoring with stratified design

Outline

- Some background
- Stratified design as a concept
- A somewhat modified example from a real-world API

stratum is latin

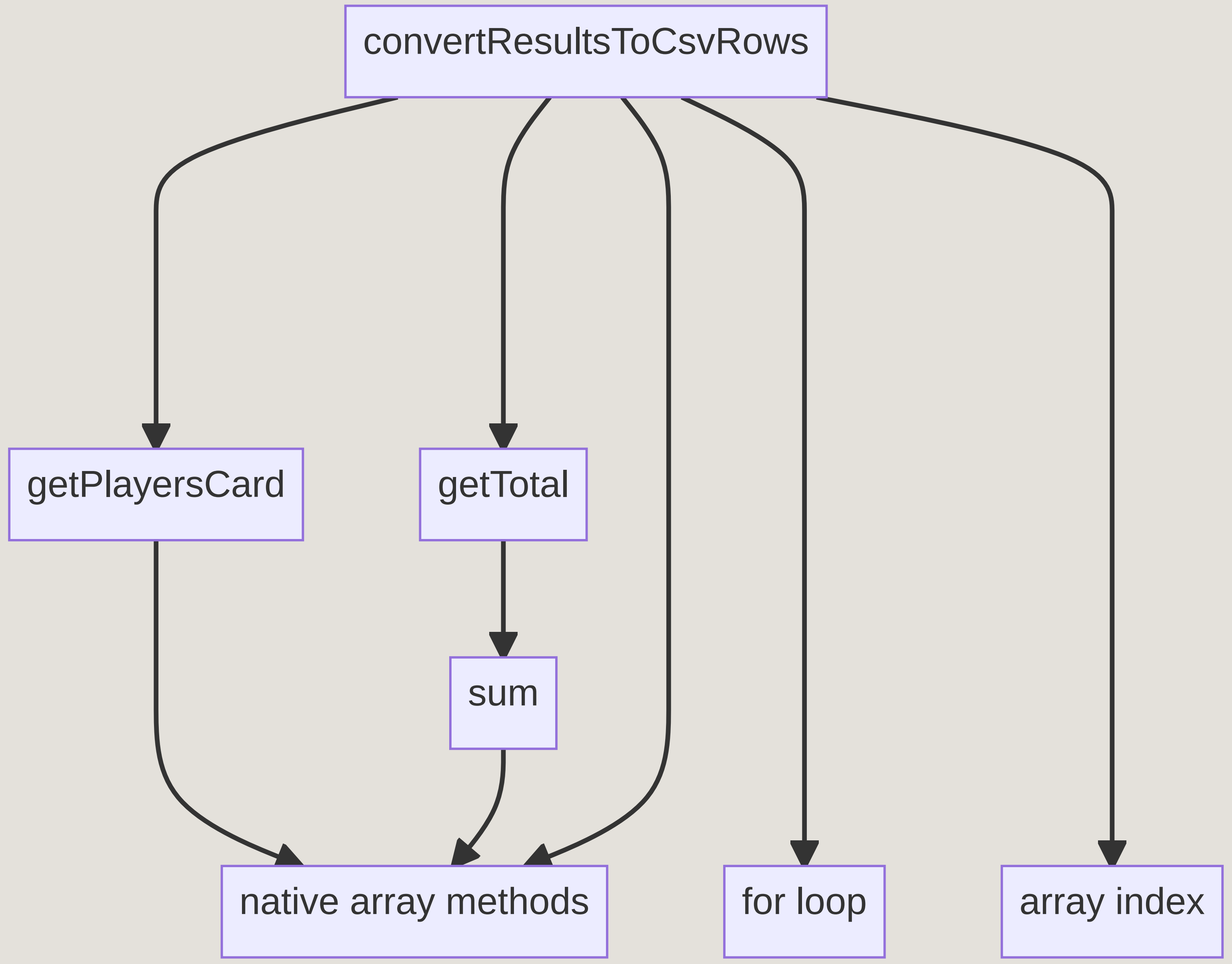
What’s the point?

- Readability, maintainability, testability etc
- Ability to work on the correct level of detail
- Power of conceptualization

```
const users: Player = [
  { name: "Paul", id: 1, strokes: undefined, rank: undefined },
  { name: "Ricky", id: 2, strokes: undefined, rank: undefined },
];
const holes: Hole = [
  { no: 1, par: 3 },
  { no: 2, par: 3 },
  { no: 3, par: 4 },
];
const scoreCards: ScoreCard = [
  { userId: 1, strokes: [2, 2, 4] },
  { userId: 2, strokes: [2, 2, 3] },
];
```

```
import sum from "mathlib";

const getTotal = (results: number[]) => {
  return sum(results);
};

const getPlayersCard = (player: Player, scoreCards: ScoreCa
  return scoreCards.find((scoreCard) => scoreCard.playerId 
};

const convertResultsToCsvRows = (players: Player[]): string
  let rows: string[] = [];
  for (let player of players) {
    const card = getPlayersCard(player, scoreCards);
    let row: ScoreRow[] = [];
    let results: number[] = [];
    for (let i = 0; i < holes.length; i++) {
      const hole = holes[i];
      const strokes = card.strokes[i];
      const par = holes[i].par;
      results.push(strokes - par);
    }
    const total = getTotal(results);
    rows.push([...results, total].join(","));
  }
  return rows;
};
```

- native language features
- generic function
- specific functions of domain X

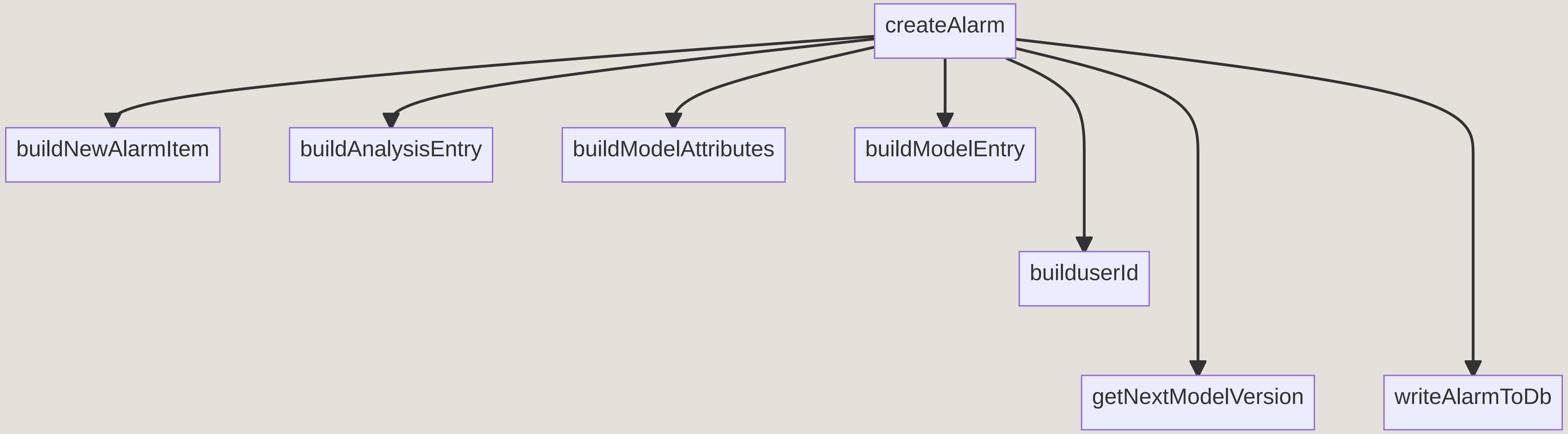


Examples from a KOA api

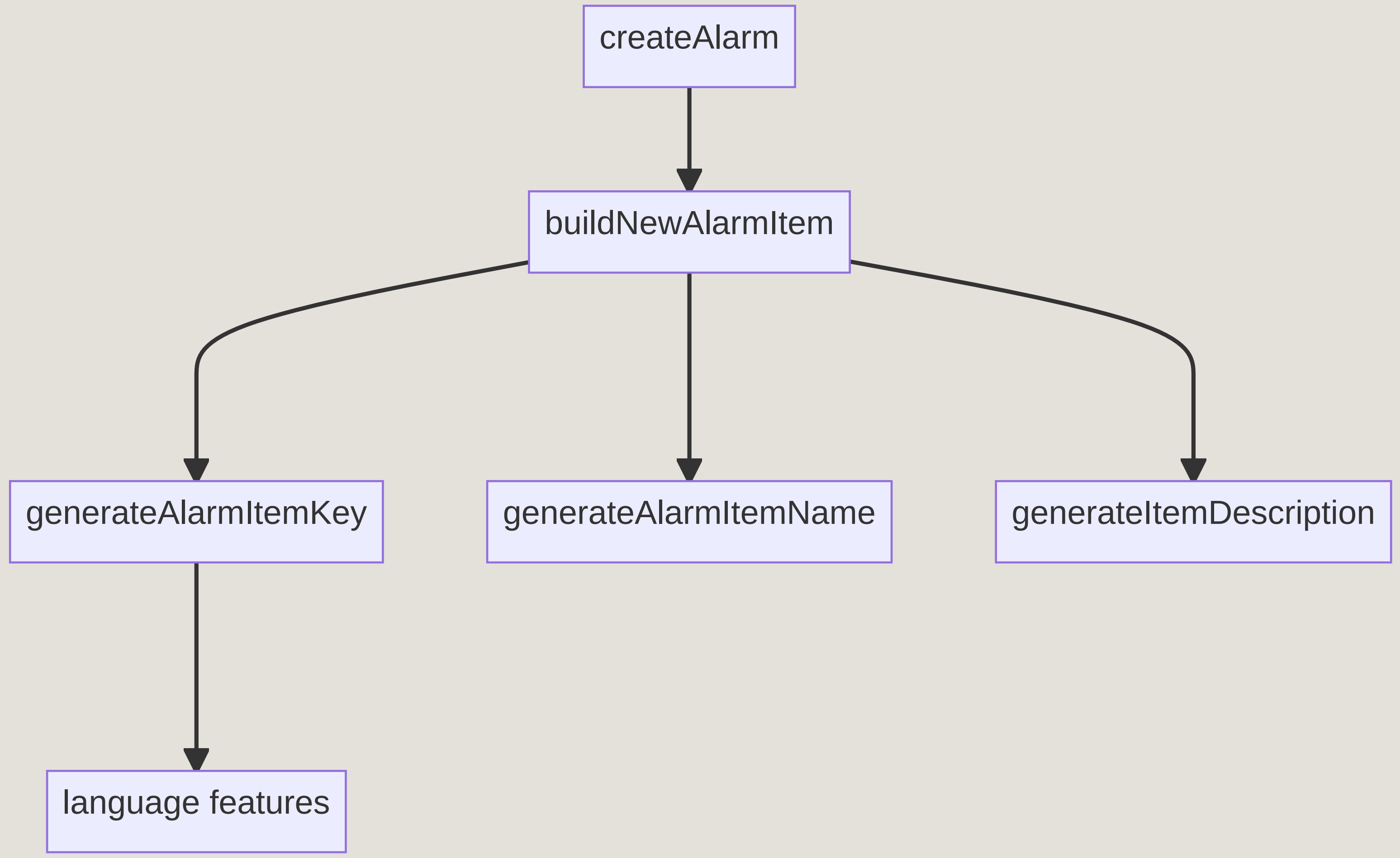
```
router.post(
  "/api/alarms",
  async (ctx) => {
    const alarmProps = validatePostAlarmProps(ctx.request.body);
    await createAlarm(alarmProps);
    ctx.status = 200;
    ctx.body = { message: "Alarm created" };
  },
);
```

```
router.post("/api/items", authenticationRequired, logQueryDetails, async (ctx: PublicContext) => {
  const table: string = 'items'

  const rp: { [key: string]: any } = ctx.request.body;
  const { params } = rp as { id: string; parentId: string };

  const key = getKey(id, parentId);

  if (key) {
    const tags: { [key: string]: string }[] = [];
    const searchTerms = Object.keys(rp)
      .filter((key) => !["id", "parentId"].includes(key))
      .reduce((obj: { [key: string]: string }, key) => {
        obj[key] = rp[key];
        return obj;
      }, {});


    const result = await connector.select("*").from(table).where({ id_key: key });

    for (const item of result) {
      const tagAttributes = tag["tag_attributes"] ?? {};
      const valmetTagInfo = tagAttributes["valmet_tag"] ?? {};
      const matching: boolean[] = [];
```

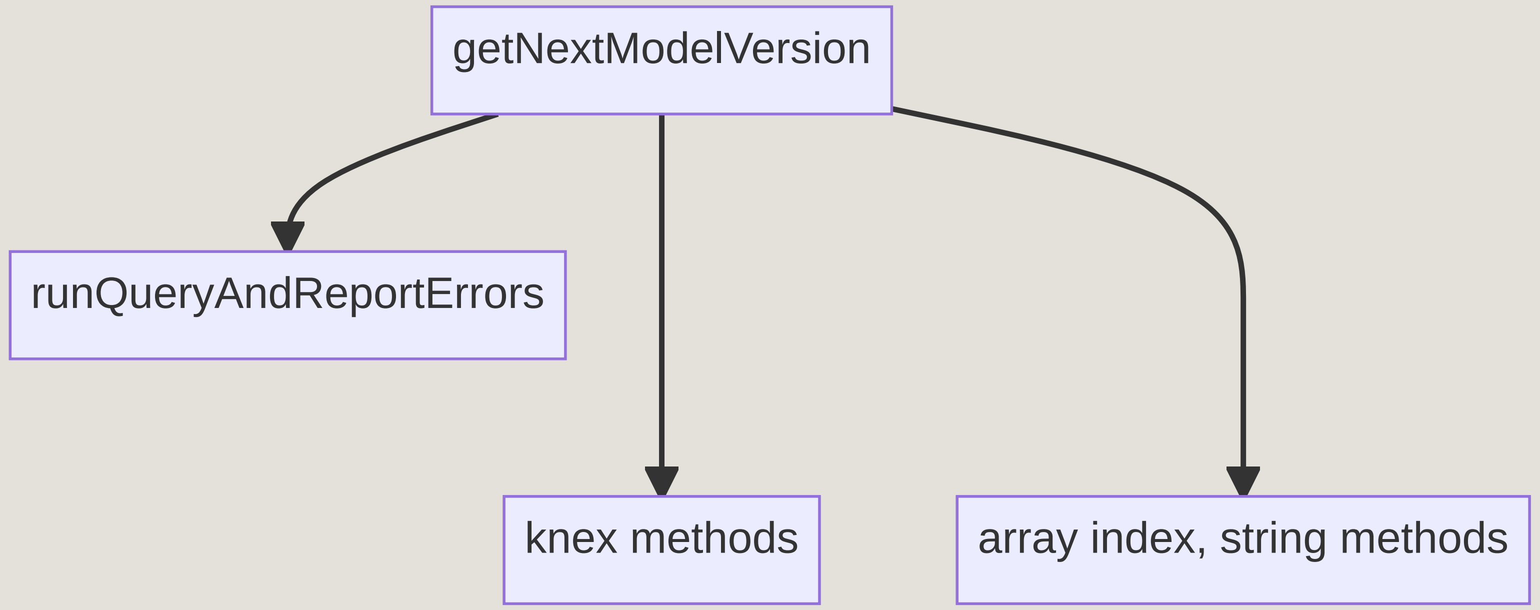
```
export const createAlarm = async (inputProps: FormattedAlarm, id: string, parentId: string, token: string, connector: Kr
  const userId = builduserId(token);
  const alarmItem = buildNewAlarmItem(inputItem, inputProps, id, parentId);
  const analysisEntry = buildAnalysisEntry(inputItem, userId);
  const nextModelVersion = await getNextModelVersion(analysisEntry?.code, connector).catch(handleDbError);
  const modelAttributes = buildModelAttributes(inputProps.alarmSeverity, inputProps.warningSeverity, inputItem);
  const modelEntry = buildModelEntry( analysisEntry?.analysis_code);
  await writeAlarmToDb(connector, modelEntry, alarmItem, analysisEntry, tagModelEntries, idate);
};
```


createAlarm
buildNewAlarmItem
buildAnalysisEntry
buildModelAttributes
buildModelEntry
builduserId
getNextModelVersion
writeAlarmToDb

```
export const buildNewAlarmItem = (
  inputItem: ItemMetadata | undefined,
  inputProps: FormattedAlarm,
  id: string,
  parentId: string,
): ItemMetadata | undefined => {
  if (!inputItem) return;
  const { limitType, alarmDescription } = inputProps;
  const key = generateAlarmItemKey(inputItem, id, parentId,
  const name = generateAlarmItemName(inputItem);
  const description = generateItemDescription(inputItem, al
  return {
    item_key,
    item_name,
    item_description,
    uby: userId,
    iby: userId,
    item_type: "AlarmItemType",
  };
};
```



```
export const getNextModelVersion = async (code: string | un
  if (!code)  return;
  const selectedColumns: (keyof Model)[] = ["col1"];
  const query = () =>
    connector
      .select<Partial<Model>[]>(selectedColumns)
      .from(tableName)
      .where("xxx", code)
      .orderByRaw("length(col1) DESC")
      .orderByRaw("col1 DESC")
      .limit(1);
  const result = await runQueryAndReportErrors(query);
  if (!result?.length) return "1";
  const latestVersion = result[0].col1;
  const nextVersion = Number(latestVersion) + 1;
  return nextVersion.toString();
};
```



```
router.post("/api/items", authenticationRequired, logQueryD
  const table: string = 'items'

  const rp: { [key: string]: any } = ctx.request.body;
  const { params } = rp as { id: string; parentId: string }

  const key = getKey(id, parentId);

  if (key) {
    const tags: { [key: string]: string }[] = [];
    const searchTerms = Object.keys(rp)
      .filter((key) => !["id", "parentId"].includes(key))
      .reduce((obj: { [key: string]: string }, key) => {
        obj[key] = rp[key];
        return obj;
      }, {});



    const result = await connector.select("*").from(table).

    for (const item of result) {
      const tagAttributes = tag["tag_attributes"] ?? {};
      const valmetTagInfo = tagAttributes["valmet_tag"] ??
      const matching: boolean[] = [];
```

- Tools to conceptualize "you should make this more readable"
- Aiming for straightforward implementations
- Helping future readers with the level of detail needed

Literature

- Normand, Eric 2021: Grokking simplicity. Manning.
- Płachta, Michał 2022: Grokking functional Programming. Manning
- Abelson, Harold & Sussman, Gerald 1985: Structure and Interpretation of Computer Programs