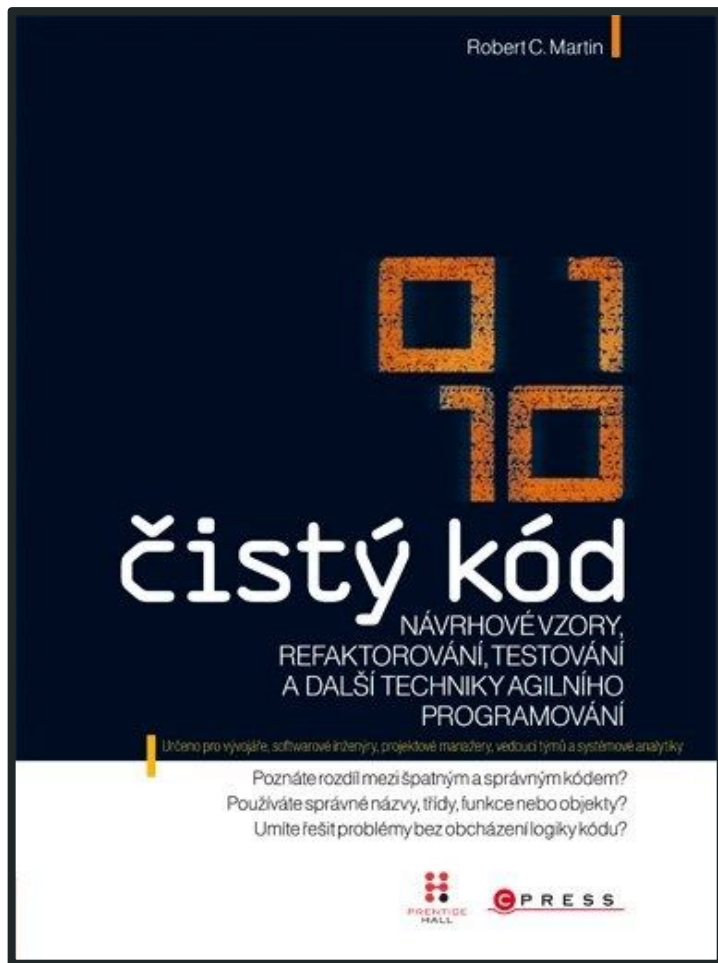


Clean Code a best practices v době Copilota



Postupné zlepšování

Smysluplná jména

Zpracování chyb

Souběžnost

Formátování

Vývoj

Funkce

Třídy

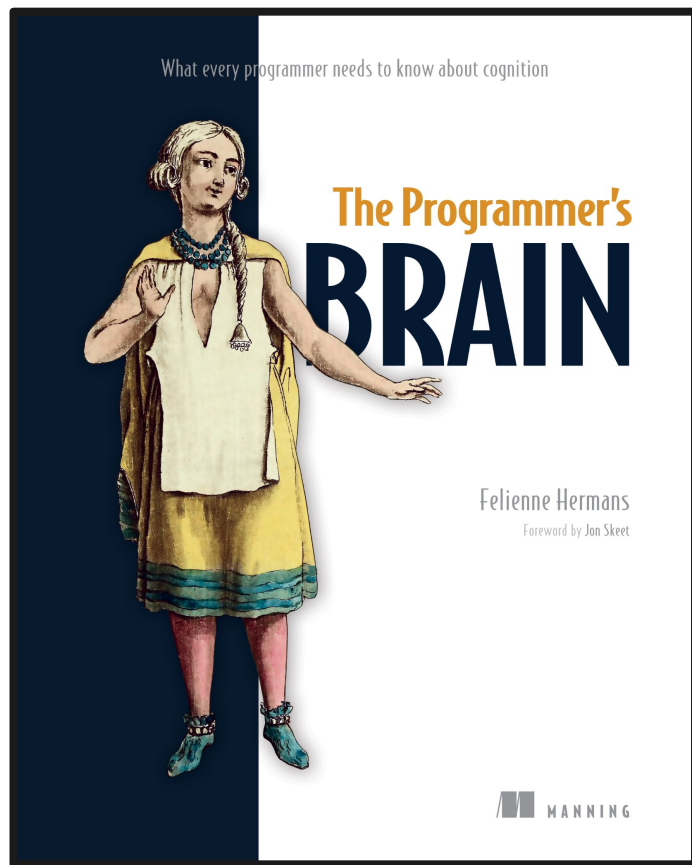
Hranice

Objekty a datové struktury

Jednotkové testy

Komentáře

Systemy



<https://www.manning.com/books/the-programmers-brain>

Postupné zlepšování

Smysluplná jména

Zpracování chyb

Souběžnost

Formátování

Vývoj

Funkce

Třídy

Hranice

Objekty a datové struktury

Jednotkové testy

Komentáře

Systemy

Různé druhy nejasností v kódu

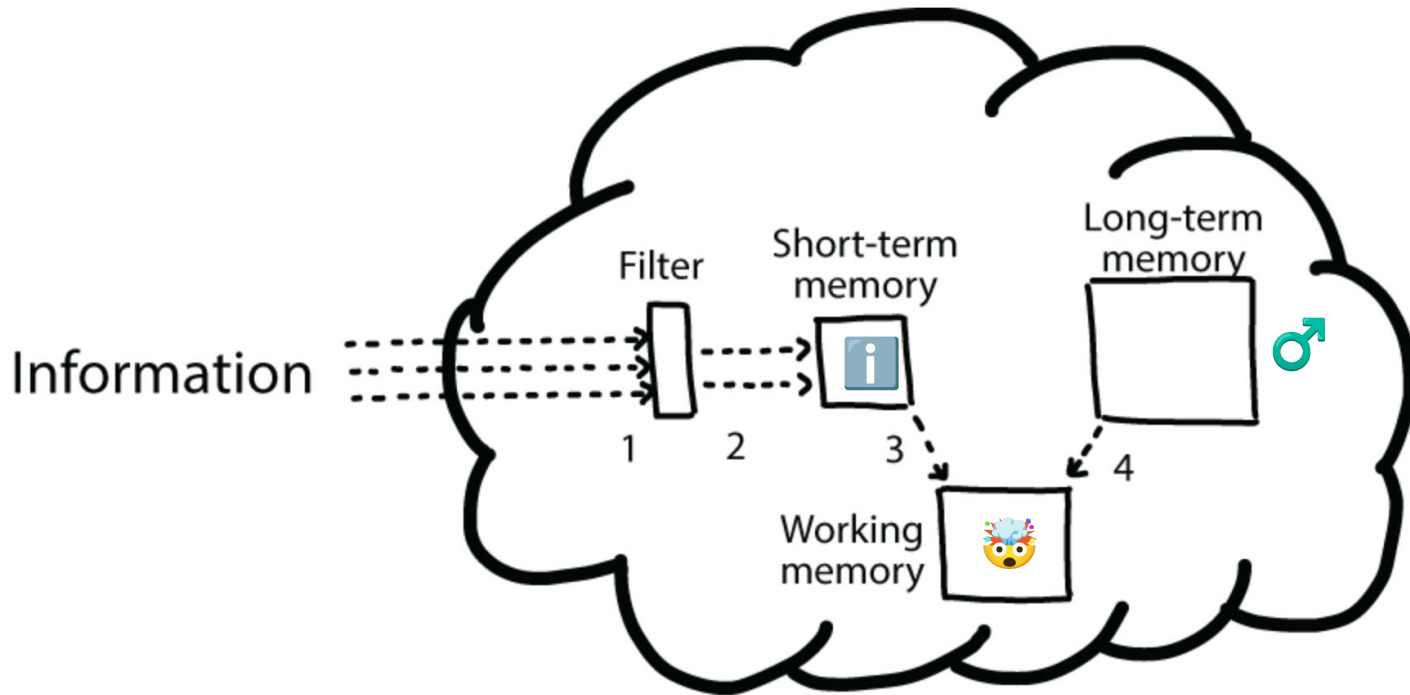
- Nedostatečné znalosti 🙋
- Nedostatek informací ⓘ
- Nedostatek
“výpočetního výkonu” 🤯

```
2 2 2 2 2 τ n
```

```
public class BinaryCalculator {  
    public static void main(Integer n) {  
        System.out.println(Integer.toBinaryString(n));  
    }  
}
```

```
1 LET N2 = ABS (INT (N))  
2 LET B$ = ""  
3 FOR N1 = N2 TO 0 STEP 0  
4     LET N2 = INT (N1 / 2)  
5     LET B$ = STR$ (N1 - N2 * 2) + B$  
6     LET N1 = N2  
7 NEXT N1  
8 PRINT B$  
9 RETURN
```

Kognitivní proces



Prohlédněte si tento kód a zkuste si ho zapamatovat

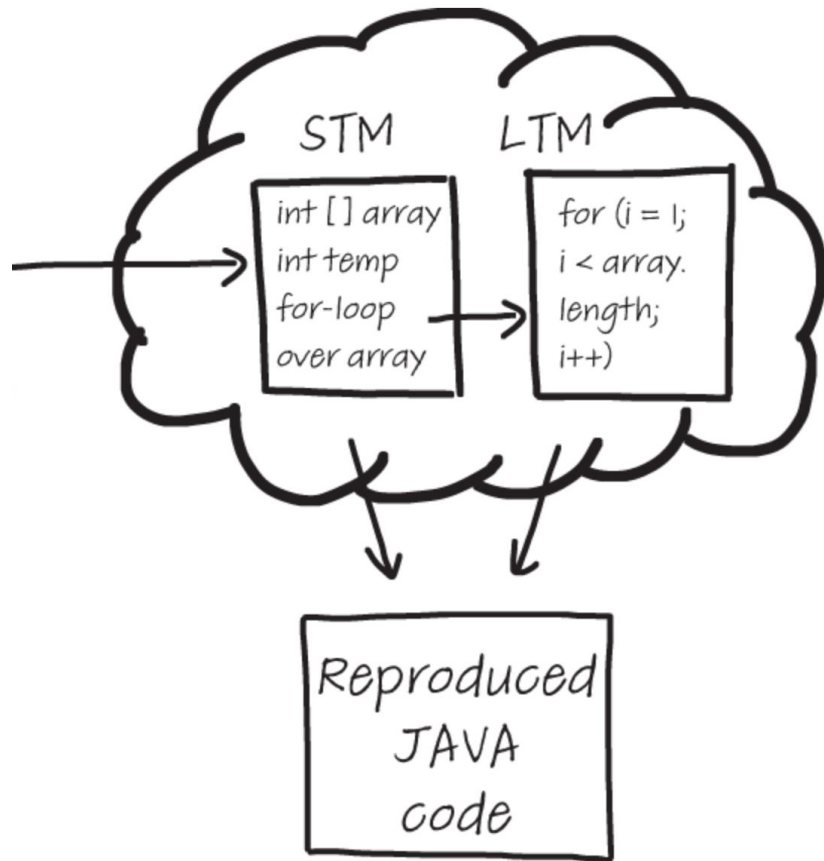
```
public class InsertionSort {  
    public static void main (String [] args) {  
        int [] array = {45,12,85,32,89,39,69,44,42,1,6,8};  
        int temp;  
        for (int i = 1; i < array.length; i++) {  
            for (int j = i; j > 0; j--) {  
                if (array[j] < array [j - 1]) {  
                    temp = array[j];  
                    array[j] = array[j - 1];  
                    array[j - 1] = temp;  
                }  
            }  
        }  
        for (int i = 0; i < array.length; i++) {  
            System.out.println(array[i]);  
        }  
    }  
}
```

Prohlédněte si tento kód a zkuste si ho zapamatovat

- `class InsertionSort`
- `int [] array = {45,12,85,35,89,39,...}`
- `int tpm`
- `for-loop i`
- `for-loop j`
- `if (... < ...)`
- `swap`
- `print in for`

Co se děje ve vašem mozku?

```
public class InsertionSort {  
    public static void main (String [] args) {  
        int [] array = {45,12,...};  
        int temp;  
        for (int i = 1; i < array.length; i++) {  
            for (int j = i; j > 0; j--) {  
                if (array[j] < array [j - 1]) {  
                    // swap j with j - 1  
                    temp = array[j];  
                    array[j] = array[j - 1];  
                    array[j - 1] = temp;  
                }  
            }  
        }  
        //print array  
        for (int i = 0; i < array.length; i++) {  
            System.out.println(array[i]);  
        }  
    }  
}
```



Proč je čtení neznámého kódu těžké?

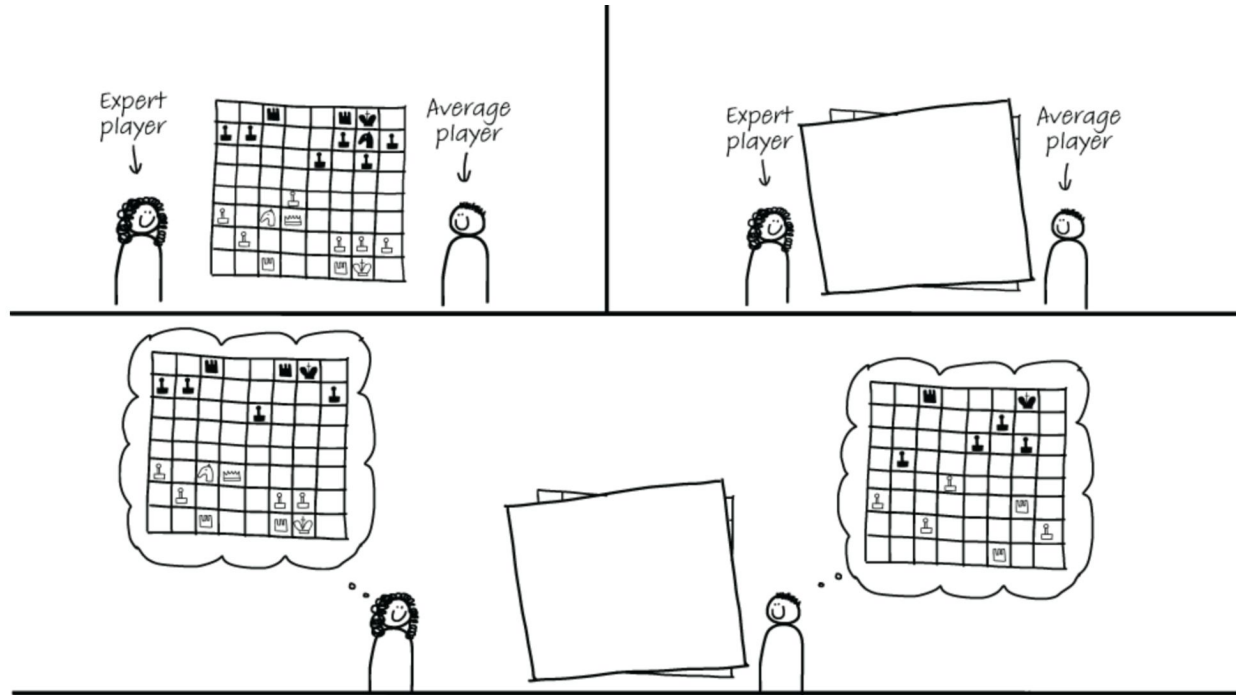
```
void execute(int x[]){
    int b = x.length;

    for (int v = b / 2 - 1; v >= 0; v--)
        func(x, b, v);

    // Extract elements one by one
    for (int l = b-1; l > 0; l--)
    {
        // Move current to end
        int temp = x[0];
        x[0] = x[l];
        x[l] = temp;

        func (x, l, 0);
    }
}
```

Lze trénovat krátkodobou paměť?



Chunking theory - De Groot's chess experiment

Kolik si toho z té věty dokážete zapamatovat?

o|z o\c<o v|z 3

abk mrtpi gbar

cat loves cake



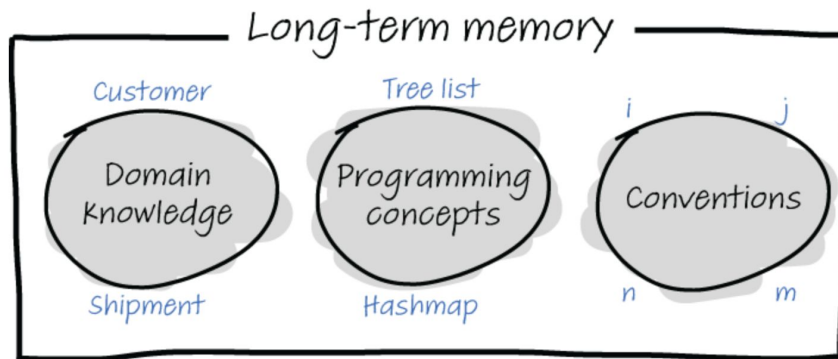
Chunking ve zdrojovém kódu

```
public class InsertionSort {  
    public static void main (String [] args) {  
        int [] array = {45,12,...};  
        int temp;  
        for (int i = 1; i < array.length; i++) {  
            for (int j = i; j > 0; j--) {  
                if (array[j] < array [j - 1]) {  
                    // swap j with j - 1  
                    temp = array[j];  
                    array[j] = array[j - 1];  
                    array[j - 1] = temp;  
                }  
            }  
        }  
        //print array  
        for (int i = 0; i < array.length; i++) {  
            System.out.println(array[i]);  
        }  
    }  
}
```

Pojmenování

Pojmenování

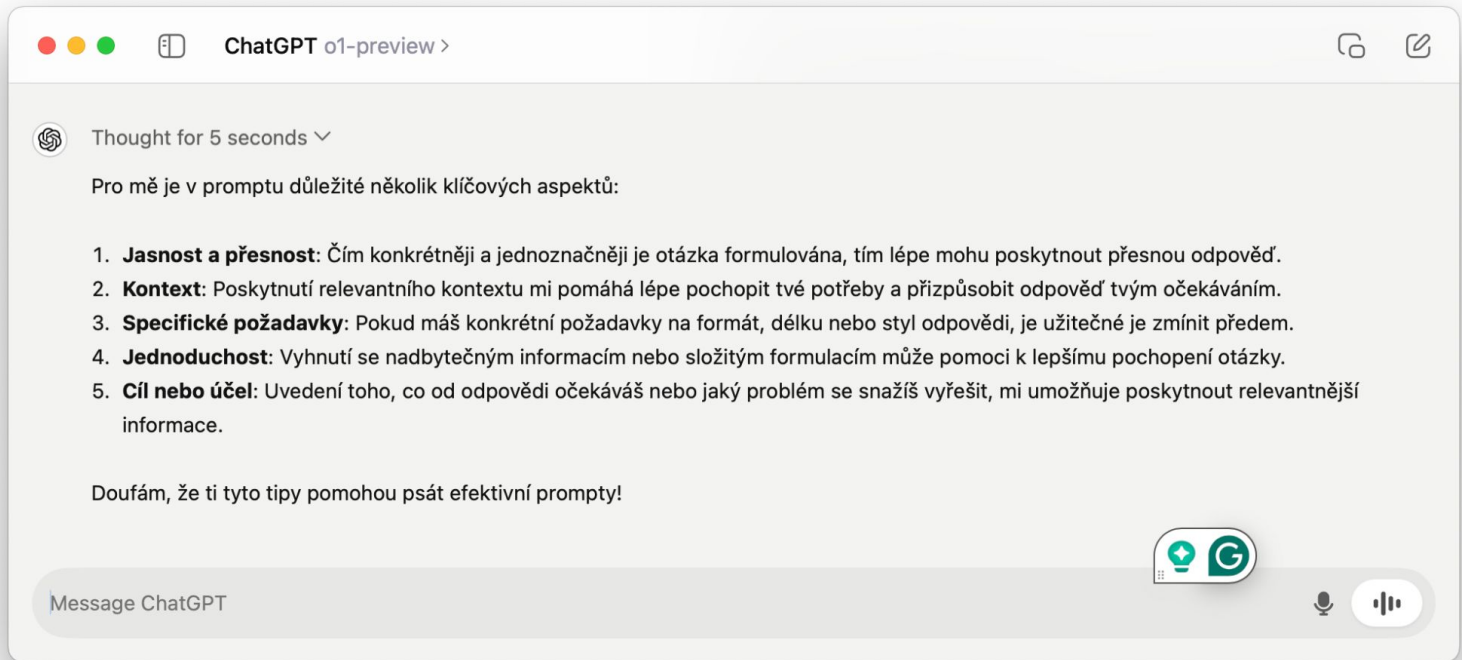
- Jsou smysluplná 
- Dají se přečíst/vyslovit
- Jdou vyhledat
- Nejsou zakódovaná
- Používají pojmy z domény řešení
- Používají “oborové pojmy” 
- Mají jedno slovo pro jeden pojem
- Nelžou



Čitelnost



Kontext



Čitelnost



Kontext



Majáky v kódu

- Jednoduché

- komentář (node in tree)
- Názvy proměnných (root, tree)

- Složené

- self.left & self.right
- for cyklus

```
# A class that represents a node in a tree
```

```
class Node:
```

```
    def __init__(self, key):  
        self.left = None  
        self.right = None  
        self.val = key
```

```
# A function to do in-order tree traversal
```

```
def print_in_order(root):  
    if root:
```

```
        # First recur on left child  
        print_in_order(root.left)
```

```
        # then print the data of node  
        print(root.val)
```

```
        # now recur on right child  
        print_in_order(root.right)
```

```
print("Contents of the tree are")  
print_in_order(tree)
```


Pojmenování - výzkumy

Jeden ze čtyř code review obsahoval **připomínky týkající se pojmenování** a **připomínky k názvům identifikátorů** se objevily v 9 % případů.

Počáteční pojmenování má dlouhodobý dopad – v jedné code base se **pojmenování nezlepší s věkem** kódu.

Kód se špatnými názvy obsahuje více chyb.

Formátování

Formátování

```
public class InsertionSort {
    public static void main (String [] args) {
        int [] array = {45,12,...};
        int temp;
        for (int i = 1; i < array.length; i++) {
            for (int j = i; j > 0; j--) {
                if (array[j] < array [j - 1]) {
                    // swap j with j - 1
                    temp = array[j];
                    array[j] = array[j - 1];
                    array[j - 1] = temp;
                }
            }
        }
        //print array
        for (int i = 0; i < array.length; i++) {
            System.out.println(array[i]);
        }
    }
}
```

Formátování

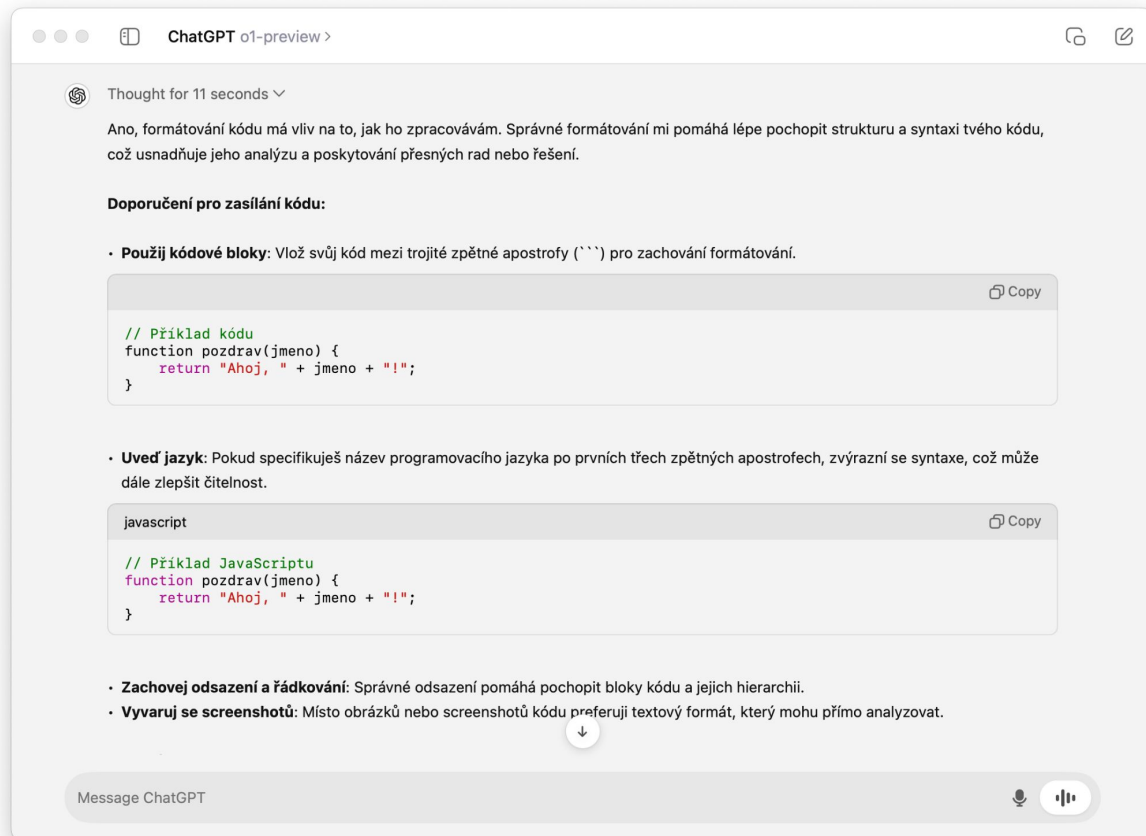
Vyřešený problém!

Namísto diskusí o formátování máme “opinionated” nástroje

- [Prettier](#), [eslint](#)
- [biome.js](#)

- Existují [pluginy pro prettier](#) na různé jazyky - php, ruby, kotlin, java, rust, elm,...
- Každý jazyk má specifické formátory (ktlint, ktfmt)

Formátování a ChatGPT



Paradigmata

Paradigmata

Objektově orientované programování

- Zapouzdření (data + kód)
- Dědičnost
- Polymorfismus
- Abstrakce

Další principy:

- S.O.L.I.D
- Návrhové vzory
 - Singleton, Factory, Observer, Strategy, Adapter, Visitor,...
- Demeters rule
- Tell, don't ask
- ...

Funkcionální programování

- Pure functions
- Imutabilita
- High order functions
- Rekurze
- Lazy evaluation
- **Rozlišujte**
 - **Data**
 - **Calculations**
 - **Actions**

Data - Calculations - Actions

- Actions

- Časově závislé
- Počet volání je důležitý
- Např. odeslání emailu, zápis do DB,...

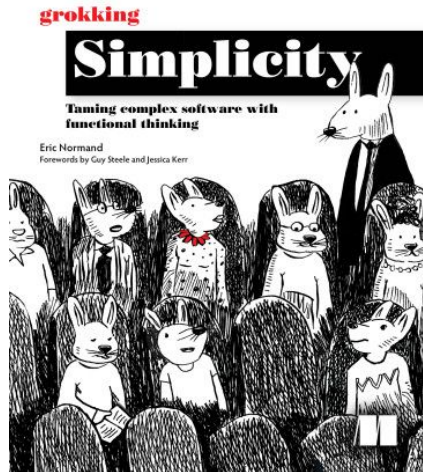
- Calculations

- Výpočty, filtrace, mapování dat
- Nejsou závislé na čase
- Počet volání nehraje roli

- Data

- Faktické informace
- Vyžadují interpretaci
- Serializovatelné

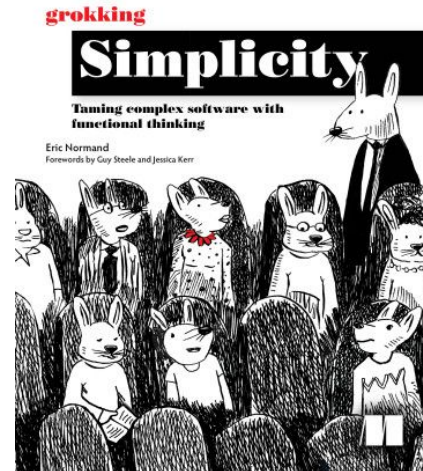
```
const shoppingCart = {  
  items: [{  
    product: {....}  
    count: 2  
  }]  
}
```



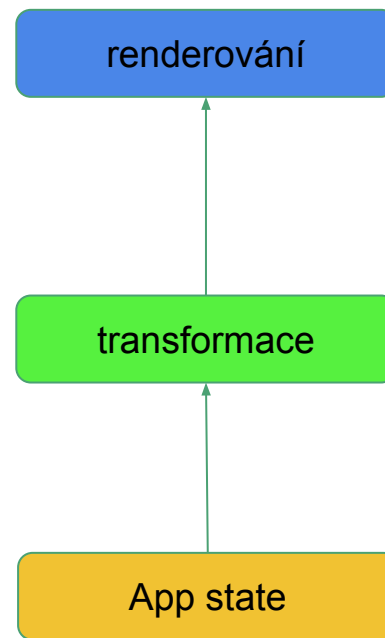
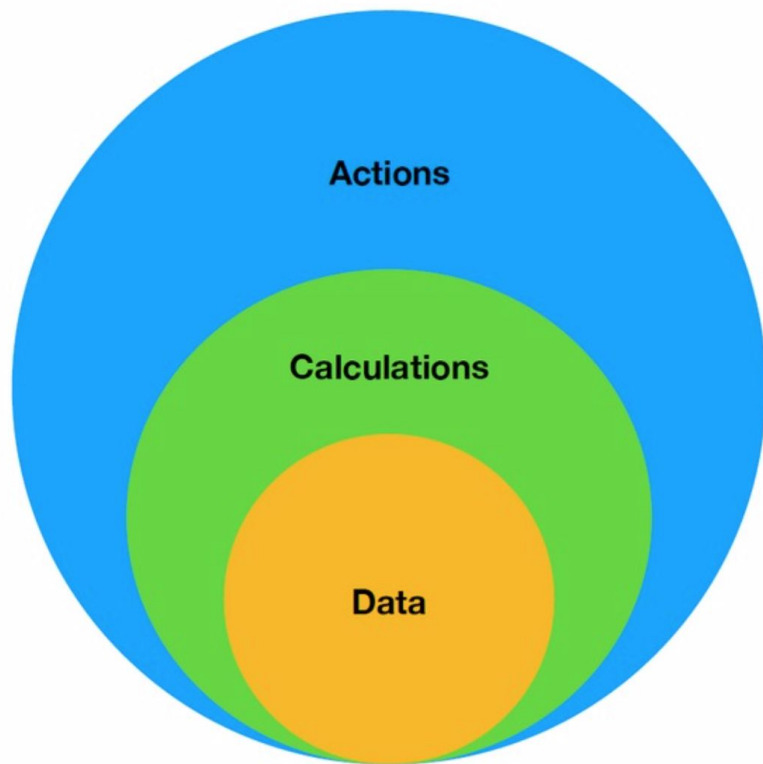
<https://www.manning.com/books/grokking-simplicity>

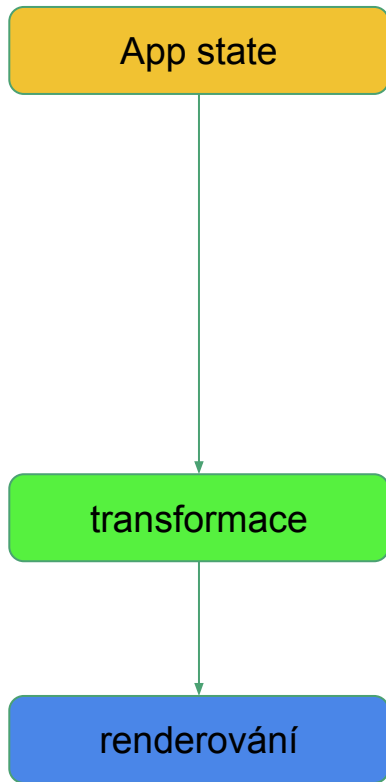
Data - Calculations - Actions

- Actions
 - Bezpečná změna stavu
 - Garantované pořadí
 - Spustí se ve správný čas a správném počtu
- Calculations
 - Správnost “logiky”
 - Testovací strategie
 - Výsledku výpočtu můžeme nahradit hodnotou (Referential transparency)
- Data
 - Správné struktury pro efektivní přístup a zpracování
 - Obsahují vše co je potřeba pro operace?
 - Dlouhodobý záznam



Data - Calculations - Actions





```
const state = {  
  todos: [  
    {label: 'koupit mléko', complete: false}  
    {label: 'umýt auto', complete: true}  
  ]  
  filter: {  
    completed: false  
  }  
}
```

```
getTodosByFilter(state)
```

```
const Todos = ({visibleTodos}) => <List items={visibleTodos} />
```

Funkce

Funkce

```
function sendOrderConfirmation(order) {  
  const emailTemplate = `  
    Dear {{customerName}},  
    Thank you for your order #{{orderId}} ....`;    Data uvnitř funkce  
  
  const orderDate = new Date(order.orderDate).toLocaleDateString();  
  
  const orderItems = order.items.map((item, index) => {    Řeší více věcí  
    return `${index + 1}. ${item.name} - Quantity: ${item.quantity} - Price: ${item.price}`;  
  }).join('\n');  
  
  const emailContent = emailTemplate  
    .replace('{{customerName}}', order.customerName || 'Customer')  
    .replace('{{orderId}}', order.orderId)  
    .replace('{{orderDate}}', orderDate)  
    .replace('{{orderItems}}', orderItems)  
    .replace('{{totalPrice}}', order.totalPrice.toFixed(2));  
  
  sendEmail(order.customerEmail, 'Your Order Confirmation', emailContent);  
}
```

Různá úroveň abstrakce

Funkce

```
const confirmationEmailTemplate = `
  Dear {{customerName}},
  Thank you for your order #{{orderId}} ....`;

function prepareOrderEmail(confirmationEmailTemplate, order) {
  const emailContent = ...

  return {
    subject: 'Your Order Confirmation',
    body: emailContent
  }
}

function sendOrderConfirmation(order) {
  const email = prepareOrderEmail(confirmationEmailTemplate, order)
  sendEmail(order.customerEmail, email.subject, email.body);
}
```

Clean code:

- Malá
 - Dělá jen jednu věc
 - Má jednu úroveň abstrakce
-

Eric Normand

- Data
- Calculations
- Actions

Počet argumentů funkce

Objektově orientované programování

- 0 - ideální
 - Pracujeme s daty objektu
- 1
 - `fileExists(fileName)`,
 - `fileOpen(fileName)`
- 2
 - `assertEquals(expected, actual)`
- 3
 - `assertEquals(message, expected, actual)`
- Výstupní argumenty
 - Ideálně žádný

Funkcionální programování

- 0 - nejspíš impure funkce
 - Action?
 - Čte globální data?
- 1
 - `fileExists(fileName)`,
 - `fileOpen(fileName)`
- 2
 - `getIn(path, object)`
- 3
 - `getIn(path, object, defaultValue)`
 - `setInPath(path, value, object)`
- Výstupní argumenty
 - Funkce typicky něco vrátí, pokud ne, jde o Action

Argumenty funkce

Boolean parametrům se vyhněte

```
const price = applyDiscount(order, true);
```

```
const price = applyDiscount(order, { isVip: true });
```

```
const price = applyDiscount(order, { isPartner: true });
```

```
const price = applyDiscount(order, { is...: true });
```

```
const price = applyDiscount(order, { priceList: PriceList.VIP });
```

```
const price = applyDiscount(order, PriceList.VIP);
```



Argumenty funkce

Preferujte object argument pro související data

```
const packageSize = derivePackageSize(width, height);
```

```
const packageSize = derivePackageSize(packageDimensions);
```

```
const packageDimensions = {width: 123, height: 456, depth: 789};
```

```
const packageSize = derivePackageSize(packageDimensions);
```



Kontext

Funkce - Single return

```
function calculateDiscount(price, coupon) {  
  let discount = 0;  
  
  if (coupon === "DISCOUNT10") {  
    discount = price * 0.10;  
  } else if (coupon === "FREESHIPPING") {  
    discount = 0;  
  } else if (coupon === "BLACKFRIDAY") {  
    discount = price * 0.50;  
  } else {  
    discount = 0;  
  }  
  
  return discount;  
}
```

Funkce - Early return

```
function calculateDiscount(price, coupon) {  
    if (coupon === "DISCOUNT10") {  
        return price * 0.10;  
    } else if (coupon === "FREESHIPPING") {  
        return 0;  
    } else if (coupon === "BLACKFRIDAY") {  
        return price * 0.50;  
    } else {  
        return 0;  
    }  
}
```

Potřebujeme

else if?

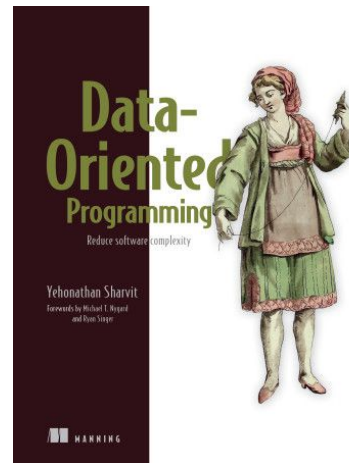
Funkce - Early return

```
function calculateDiscount(price, coupon) {  
    if (coupon === "DISCOUNT10") {  
        return price * 0.10;  
    }  
    if (coupon === "FREESHIPPING") {  
        return 0;  
    }  
    if (coupon === "BLACKFRIDAY") {  
        return price * 0.50;  
    }  
    return 0;  
}
```

Objekty a datové struktury

Data-Oriented Programming

- Oddělte data a kód
- Data reprezentujte generickými strukturami
- Data jsou imutabilní
- Oddělte schéma dat od reprezentace



Pozor: Data-oriented design je něco jiného

<https://www.manning.com/books/data-oriented-programming>

Klasické OOP

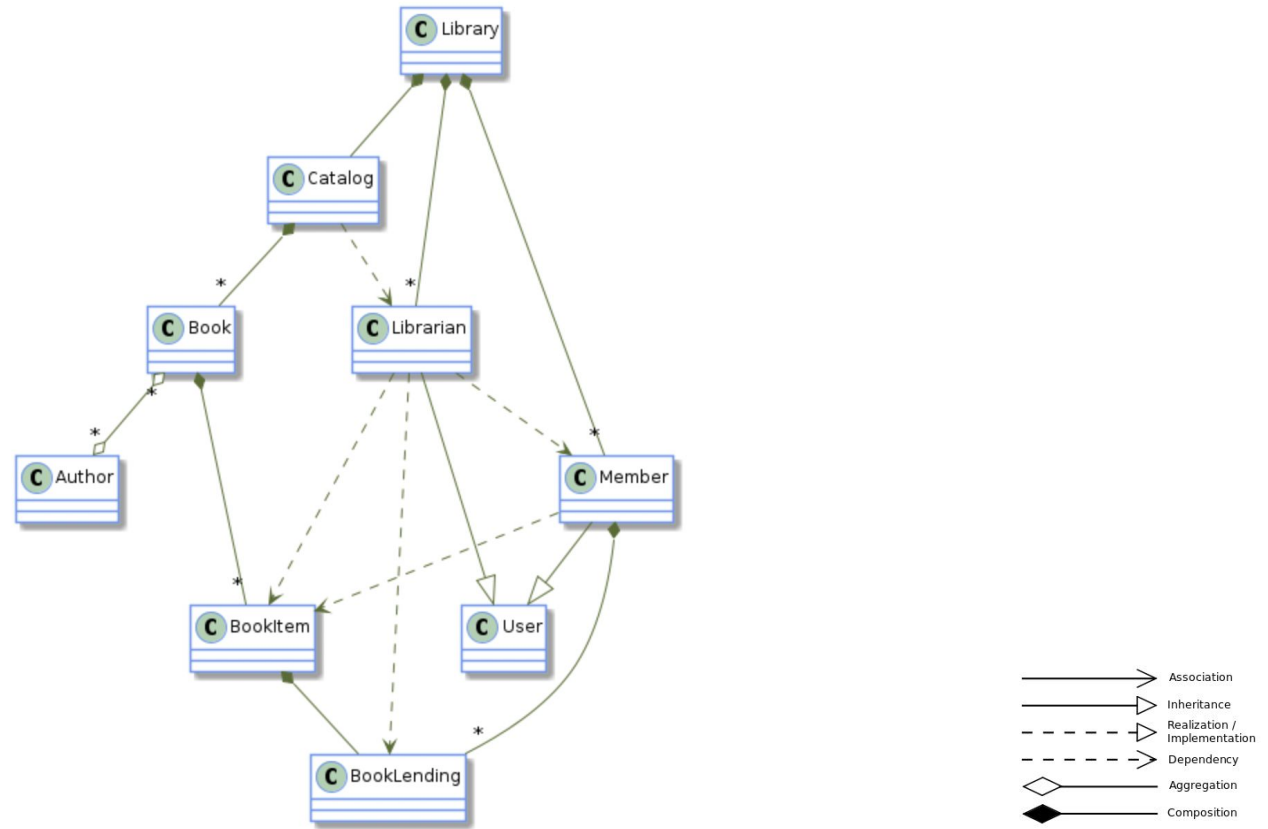


Figure 1.10. A class diagram overview for a Library management system

Oddělte kód a data

```
function createAuthorData(firstName, lastName, books) {  
  return {  
    firstName: firstName,  
    lastName: lastName,  
    books: books  
  };  
}  
  
function fullName(data) {  
  return data.firstName + " " + data.lastName;  
}  
  
function isProlific (data) {  
  return data.books > 100;  
}  
  
var data = createAuthorData("Isaac", "Asimov", 500);  
fullName(data);  
// → Isaac Asimov
```

```
class AuthorData {  
  constructor(firstName, lastName, books) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
    this.books = books;  
  }  
}  
  
class NameCalculation {  
  static fullName(data) {  
    return data.firstName + " " + data.lastName;  
  }  
}  
  
class AuthorRating {  
  static isProlific (data) {  
    return data.books > 100;  
  }  
}  
  
var data = new AuthorData("Isaac", "Asimov", 500);  
NameCalculation.fullName(data);  
// → "Isaac Asimov"
```


Oddělte kód a data

Výhody

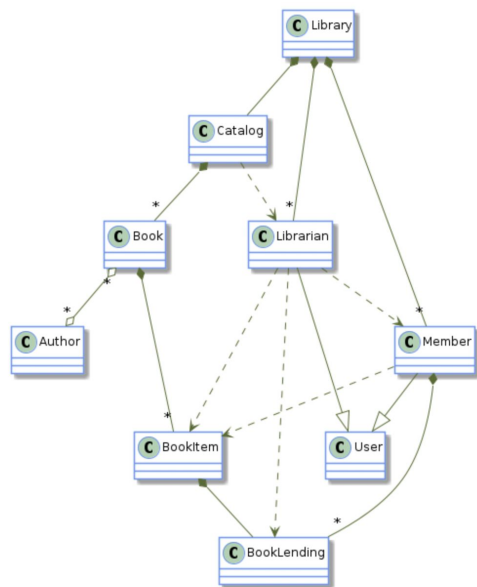
- Kód lze opakovaně použít v různých kontextech
- Kód lze testovat izolovaně
- Systémy bývají méně složité

Cena

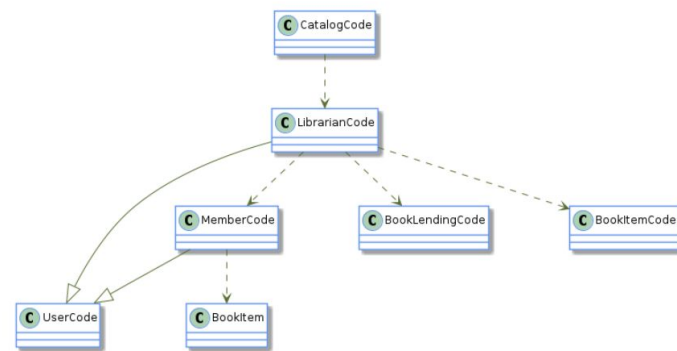
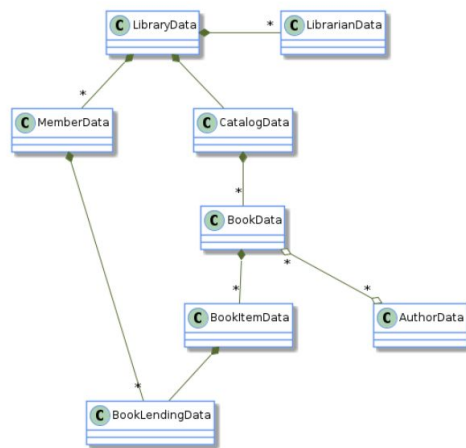
- Žádná kontrola nad tím, jaký kód má přístup k jakým datům
- Systémy jsou tvořeny více entitami

Více entit - méně complexity

OOP



DOP



Schémata dat

- Json schéma
 - Je podporované i na dalších platformách
- ZOD
 - Pokud se nechcete upsat při psaní schémat
 - Je převoditelný na JSON schéma - knihovna `zod-to-json-schema`
 - Velmi dobře podporuje TypeScript

Zod schéma

```
import { z } from 'zod';

const PersonSchema = z.object({
  id: z.number(),
  name: z.string(),
  age: z.number().gte(0).lte(150),
  email: z.string().email(),
});

const validationResult = PersonSchema.safeParse({
  id: 1,
  name: 'Alice',
  age: 25,
  email: 'alice@example.com',
});

if (!validationResult.success) { // TS typy fungují správně
  console.log(validationResult.data.name, validationResult.data.email)
}
```

Manipulace se schématem

```
const PersonWithAddressSchema = PersonSchema.merge(AddressSchema);
```

```
const BasicParsonSchema = PersonSchema.pick({  
  name: true,  
  email: true,  
});
```

```
const PersonWithouAgeSchema = PersonSchema.omit({  
  age: true,  
});
```

```
const stringOrNumber = z.union([z.string(), z.number()]);
```

Zod to TS

```
import { z } from 'zod';
```

```
const PersonSchema = z.object({  
  id: z.number(),  
  name: z.string(),  
  age: z.number(),  
  email: z.string().email(),  
});
```

```
const Person = z.infer<typeof PersonSchema>;
```

Schémata, typy a jejich použití

Druh validace dat	Účel	Prostředí
Hranice	Bezpečnost	Produkce i dev
Uvnitř	Usnadnění vývoje	dev

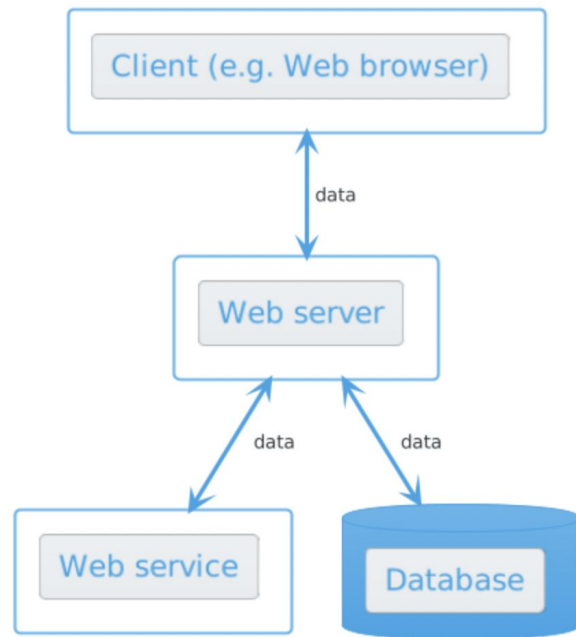
Výhody validace uvnitř systému - input/output funkcí

- Generování schema-based unit testů
- Generování diagramů

Statické typy jsou vlastně jen “dev experience”

- Použije se jen ve vývoji
- Runtime bezpečnost vám nezajistí

Schéma je “silnější” a ideálně by mělo sloužit jako zdroj, ze kterého se typy odvodí



Make impossible states impossible

```
fetchData<T>(url: string): Promise<{ data?: T; error?: string; isLoading: boolean }>
```

vrací objekt, který může volitelně obsahovat vlastnosti data, error a isLoading

- { isLoading: true }
- { isLoading: false, data: {...} }
- { isLoading: false, error: 'Some error' }

Ale také:

- { isLoading: true, data: {...} }
- { isLoading: true, error: 'Some error' }
- { isLoading: true, data: {...}, error: 'Some error' }

https://www.youtube.com/watch?v=lcgmSRJHu_8&ab_channel=elm-conf

Make impossible states impossible

```
type LoadingState = { status: 'loading' };  
type SuccessState<T> = { status: 'success'; data: T };  
type ErrorState = { status: 'error'; error: string };
```

```
type FetchState<T> = LoadingState | SuccessState<T> |  
ErrorState;
```

```
fetchData<T>(url: string): Promise<FetchState<T>>
```

Stavy, které nemohou nastat nejsou validní ani z pohledu typů

https://www.youtube.com/watch?v=lcgmSRJHu_8&ab_channel=elm-conf

Model & typy

- Použití schémat posouvá váš focus na data
- Uvažujte o tom co je “zdroj pravdy”
 - Schéma modelu
 - DB schéma
 - BE schéma
- Ze “zdroje pravdy” odvozujte
 - Další omezení schématu
 - Schémata pro konkrétní případy
 - Typy
- Schémata uplatňujte na hranici systému, např.
 - Volání REST api
 - Form data
 - Databáze (pokud ji sdílíte s jinou službou)
- Zvažte jestli použití schémat uvnitř systému nepřinese další benefity

Komentáře

Významové komentáře

- Komentuje proč, ne jak.
 - Co se děje by mělo být zřejmé z názvu?
 - Jak se to děje vidíte v kódu?
 - Proč je často informace, kterou občas musíme dodat komentářem.
Používej je střídavě a tam, kde je to skutečně nutné.
- Komentáře je třeba aktualizovat!



Procesní komentáře

```
// TODO: Implementovat cache pro zrychlení načítání dat  
function getDataFromServer() { ... }
```

```
// FIXME: po aktualizaci knihovny můžeme opravit typy  
function fetchData() { ... }
```

TS Komentáře

```
const value: number = 42;
```

```
// @ts-expect-error JS accepts number in parseInt  
const parsedValue = parseInt(value);
```

- Preferujte @ts-expect-error před @ts-ignore
 - @ts-expect-error “svítí” jako chyba, pokud není potřeba

Dokumentační komentáře

- JavaDoc
 - když píšete knihovnu
- Neopakujte se, generujte
 - <https://api-extractor.com/>

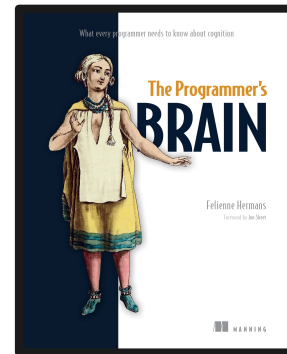
Špatné komentáře

- Nadbytečné, duplicitní
- Zadání pro Copilota
- Log/historie, deníček, návštěvní kniha
- Zakomentované řádky kódu
- Link na issues

Tipy na závěr

Jak zlepšit porozumění kódu?

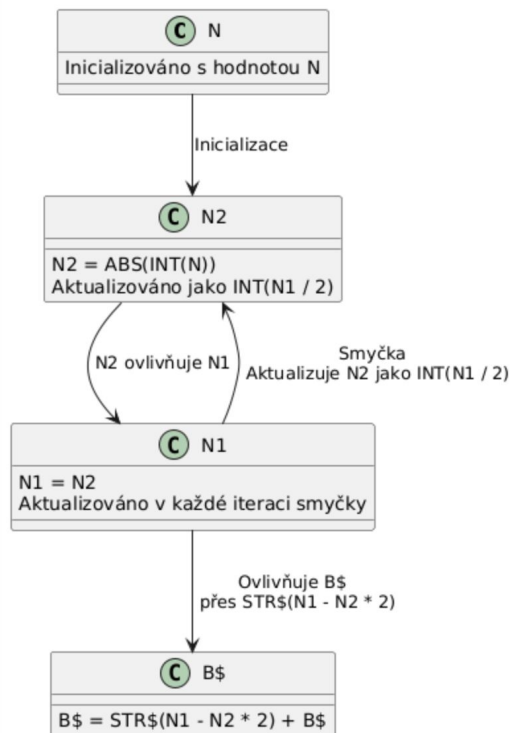
- Určete příčinu nejasností
 - Znalosti (LTM)
 - Čtěte kód
 - Učte se syntax, vzory,...
 - Kognitivní zátěž
 - Refaktorujte, zjednodušujte
 - Nakreslete si graf závislostí, stavové tabulky,...
 - Sumarizujte, vizualizujte



Jak zlepšit porozumění kódu s ChatGPT

Vytvoř graf závislostí mezi proměnnými v tomto kódu a výsledek vrať jako PlantUML

```
1 LET N2 = ABS (INT (N))
2 LET B$ = ""
3 FOR N1 = N2 TO 0 STEP 0
4   LET N2 = INT (N1 / 2)
5   LET B$ = STR$ (N1 - N2 * 2) + B$
6   LET N1 = N2
7 NEXT N1
8 PRINT B$
9 RETURN
```



Jak zlepšit porozumění kódu s ChatGPT

Pomož mi porozumět kódu.

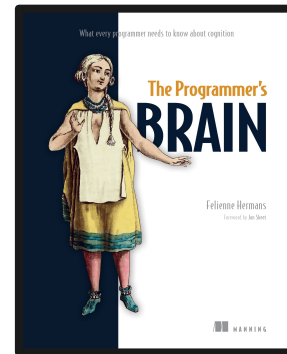
Vytvoř tabulku proměnných a jejich hodnot v jednotlivých krocích - začneme s $N = 7$

```
1 LET N2 = ABS (INT (N))
2 LET B$ = ""
3 FOR N1 = N2 TO 0 STEP 0
4     LET N2 = INT (N1 / 2)
5     LET B$ = STR$ (N1 - N2 * 2) + B$
6     LET N1 = N2
7 NEXT N1
8 PRINT B$
9 RETURN
```

Krok	N1	N2	B\$	Popis
1	-	7	""	Počáteční stav, N2 se nastaví na absolutní hodnotu N
2	7	7	""	Smyčka začíná s $N1 = N2$, což je 7
3	-	3	"1"	$N2 = \text{INT}(7 / 2) = 3$, B\$ se nastaví na $\text{STR}\$(7 - 3 * 2) = "1"$
4	3	3	"11"	$N2 = \text{INT}(3 / 2) = 1$, B\$ se aktualizuje na $\text{STR}\$(3 - 1 * 2) = "11"$
5	1	1	"111"	$N2 = \text{INT}(1 / 2) = 0$, B\$ se aktualizuje na $\text{STR}\$(1 - 0 * 2) = "111"$
6	0	0	"111"	Smyčka končí, protože $N1 = 0$
7	-	-	"111"	B\$ se vypíše, což je binární reprezentace čísla 7 (v binární soustavě 111)

Pár poznámek k chunkingu

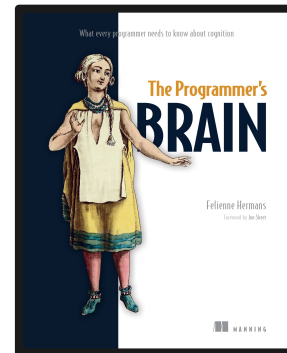
- Použití návrhových vzorů
 - Zlepšuje schopnost chunkování.
 - Pomáhá zpracovat kód rychleji, pokud čtenář zná návrhové vzory.
 - Při neznalosti návrhové vzory čitelnost snižují.
- Komentáře
 - High-level komentáře mohou programátorům pomoci rozdělit větší části kódu
 - Low-level komentáře mohou zatěžovat proces chunkingu.



Známé problémy pohledem kognitivní zátěže

Refactoring: Martin Fowler - Code smells

- Dlouhý seznam parametrů, složitý switch:
 - Přetížení kapacity pracovní paměti
- God class, dlouhé funkce/metody:
 - Nemožnost efektivního chunkingu
- Duplicity v kódu
 - Chunking selhává



Závěr

- Dbejte na srozumitelnost
- Dobré názvy pomohou tobě, kolegovi i umělé inteligenci
- Jednodušší je lepší - data > calculation > action
- Principy funkcionálního programování vedou k jednoduššímu kódu
- Odvozuj a generuj místo psaní
- Typy používej jako výhodu, ne jako bič
- Hledejte nástroje co udělají práci za vás
 - LLM nástroje jsou v této oblasti velmi schopné

- Napoprvé to nebude - začněte a postupně vylepšujte

Dotazy a připomínky?

Zdroje

Knihy:

- [Clean Code](#)
- [Programmer's Brain](#)
- [Grokking Simplicity](#)
- [Data oriented programming](#)

Video

- [Make impossible states impossible](#)