Specifikace, Návrhy specifikací, Web API, Apiary, Swagger

- Proč specifikace?
  - častou příčinou chyb chování je nedorozumění na rozhraní dvou částí kódu
  - nedokumentované rozhraní je v mysli vývojáře, druhý to může vnímat jinak
  - je to dokumentace pro klienta, který rozhraní využívá
    - nemusí zkoumat kód
  - odstínění pro vývojáře kódu
    - má možnost měnit kód, aniž by musel předávat jako dokumentaci nový kód klientům
  - umožňuje decoupling
  - představuje společný kontrakt, který musí obě strany dodržet

Behaviorální ekvivalence

- otázka, chovají-li se dva zdrojové kódy stejně
- možnost výměny kódu za jiný beze změny chování systému
- není obecně testovatelná
- roli mohou hrát:
  - množina vstupních parametrů
  - kontext, ve kterém kód běží
  - specifikace by měla obsahovat popis těchto vlivů:
    - preconditions (závazek pro klienta)
    - postconditions (závazek pro implementátora)

- může být:
  - **reflexivní**: E ~ E
  - **symetrická**: if E1 ~ E2 then E2 ~ E1
  - **transitivní**: if E1 ~ E2 AND E2 ~ E3 THEN E1 ~ E3

Preconditions/postconditions

- **preconditions** – popisují podmínky pro vstupní hodnoty
- **postconditions** – popisují, jakým způsobem má být vymezena implementace

static int find(int[] arr, int val)
requires: val occurs exactly once in arr
effects: returns index i such that arr[i] = val

```
/**
 * Find a value in an array.
 * @param arr array to search, requires that val occurs exactly once
 *            in arr
 * @param val value to search for
 * @return index i such that arr[i] = val
 */
static int find(int[] arr, int val)
```

Silnější vs. slabší specifikace

- v případě změny implementace nebo vlastní specifikace potřebujeme určit, jak bezpečné je provést změnu v implementaci (vzhledem k již existujícím klientům)
- mějme dvě specifikace – S1, S2
    - S2 je **silnější nebo stejná** jako S1, když:
        - preconditions pro S2 jsou slabší či stejné jako pro S1
        - a zároveň postconditions pro S2 jsou silnější nebo stejné jako pro S1, pro stavy, které uspokojí preconditions pro S1
    - v takovém případě impl. S2 dokáže uspokojit i potřeby klientů využívajících S1
        - lze tedy S1 nahradit S2
- **příklad**:

```
static int find_ExactlyOne(int[] a, int val)
requires: val occurs exactly once in a
effects: returns index i such that a[i] = val
```

… může být nahrazena (má slabší precondition):

```
static int find_OneOrMore,AnyIndex(int[] a, int val)
requires: val occurs at least once in a
effects: returns index i such that a[i] = val
```

… a ta může být nahrazena (má silnější postcondition):

```
static int find_OneOrMore,FirstIndex(int[] a, int val)
requires: val occurs at least once in a
effects: returns lowest index i such that a[i] = val
```

Výjimky

Null reference

- reference na objekty a pole mohou mít speciální hodnotu – null
    - neukazuje na žádný objekt
    - nelze pro primitivní objekty
- při použití je potencionální výjimka NullPointerException v runtime
- lépe se jim vyhýbat (především na rozhraní)
- good practice – pro každou operaci implicitně nepovolovat null v parametrech ani návrat. h.
- povoluje-li některý parametr null, nutné explicitně zmínit ve specifikaci (totéž pro návrat. h.)

Specifikace výjimek

- antipattern – předávání informace o chybě pomocí návratové hodnoty
- lepší je, když bude vracet výjimku

```
LocalDate lookup(String name) throws NotFoundException {
...
if ( ...not found... ) throw new NotFoundException();
...
```

- mohou se vyskytnout při chybě nebo jako speciální výsledek
- **checked výjimky** – pro speciální výsledky
- **unchecked výjimky** – pro chybové stavy

- **antipattern**:
  - nepoužívat tam, kde se nejedná o výjimečné stavy

```
try {
int i = 0;
while (true)
        a[i++].f();
} catch ArrayIndexOutOfBoundsException e) { }
```

  - lepší například for cyklus přes prvky pole
  - AIOoBException může navíc nastat jindy než očekáváme – těžko odhalitelné

Specifikace metod mutujících data

- metoda, která mění vstupní data

```
static boolean addAll(List<T> list1, List<T> list2)
requires: list1 != list2
effects: modifies list1 by adding the elements of list2 to the end of it, and returns true if
list1 changed as a result of call
```

- pro specifikaci rozhraní je velmi vhodná konvence jako pro null
  - není-li explicitně uvedeno v rozhraní, metoda neprovádí změnu ve vstupních datech
- popis mutace vstupních parametrů by se vždy měl explicitně objevit ve specifikaci rozhraní

Význam specifikace

- **předchází chybám**
  - jasná dokumentace předpokladů, na kterých staví klient i implementátor
  - chyby často vznikají z nesprávně pochopeného rozhraní
  - lze použít i strojem kontrolovaný popis ve specifikaci a výjimky

- **srozumitelnost**
  - krátká jednoduchá specifikace je srozumitelnější než vlastní implementace
  - ostatní vývojáři nemusí studovat kód pro jeho použití

- **připravenost na změny**
  - zavádí kontrakt mezi různými částmi kódu
  - ty se pak mohou rozvíjet nezávisle

Přesnost popisu specifikace

- Popisuje spec. pouze jeden možný výstup pro daný vstup, nebo dává volnost vývojáři vybrat si z množiny možných hodnot?

- **deterministic** – jednoznačně určený výstup

```
static int findExactlyOne(int[] arr, int val)
requires: val occurs exactly once in arr effects:
returns index i such that arr[i] = val
```

```
static int findOneOrMore,AnyIndex(int[] arr, int val)
requires: val occurs in arr
effects: returns the first index i such that arr[i] = val
```

- **undetermined** – není jednoznačně určený

```
static int findOneOrMore,AnyIndex(int[] arr, int val)
requires: val occurs in arr
effects: returns index i such that arr[i] = val
```

Deklarativní vs. operativní specifikace

- **operativní** – specifikace popisuje kroky implementace, dává předpis pro vývojáře

- **deklarativní** – nepopisuje detaily interních kroků implementace, popisuje výstup na základě vstupů do metody

- ve většině případů je preferovaný deklarativní přístup
    - obvykle kratší
    - lépe srozumitelná
    - neodhaluje vnitřní implementaci

Dobrý návrh specifikace rozhraní

- specifikace je klíčový kontrakt mezi implementátorem a klientem
- měla by být srozumitelná
- měla by být koherentní – metody by měly mít vždy jednu jasně definovanou zodpovědnost
- návratová hodnota by měla nést jasnou a jednoznačnou informaci
- specifikace by měla využívat abstraktní typy

Testování a specifikace

- testy musí splňovat podmínky specifikace
- znalost implementace může být vhodným doplňkem pro testování
    - může pomoci s testováním specifických podmínek (neuvedených ve specifikaci)
- testy nesmí být podřízeny implementaci, ale výhradně specifikaci rozhraní

```
static int find(int[] arr, int val)
        requires: val occurs in arr e
        ffects: returns index i such that arr[i] = val

int[] array = new int[] { 7, 7, 7 };
assertEquals(0, find(array, 0)); // bad test case: violates the spec
assertEquals(7, array[find(array, 7)]); // correct
```

Webové API

- API na webovém serveru či klientu
- server-side Web API sestává z jednoho či více endpointů
- request-response koncept
- datový formát JSON nebo XML
- SOAP vs. REST
- většina řešení je postavena na HTTP(S) a využívá HTTP response kódů
- web řešení využívají spíše RESTful web resources než SOAP web services

Web services, SOAP a WSDL

- **SOAP** – základní vrstva komunikace mezi webovými službami
- **WSDL** – popisuje rozhraní (Web Services Description Language)
    - **abstrakci** – datové typy (XML schema), message, operation
    - **přenosový protokol** – konkrétní popis formátu a endpointů služby

REST (Representational State Transfer)

- většinou využívá **HTTP** (ale ne nutně)
- REST API navrženy na základě **resources** (objekty, data či služby, ke kterým může klient přistupovat)
- každý resource má identifikátor (**URI**)
- klient si se službou vyměňuje reprezentaci resource (většinou pomocí **JSON**)
- REST API používá jednotný interface, aby byl zajištěn **decoupling klienta a serveru**
    - **HTTP slovesa** v hlavičce – GET, PUT, POST, DELETE, PATCH
- **bezestavový** model

REST Maturity Level

Level 0 (The Swamp of POX) – definuje jedno URI pro všechny operace, operace jsou POST volání

Level 1 (Resources) – separátní URI pro jednotlivé resource

Level 2 (HTTP verbs) – HTTP metody pro definici operací na resource

Level 3 (Hypermedia Controls) – hypermedia odkazy, HATEOAS (Hypertext As The Engine Of Application State)

Open API

- definuje standard pro popis rozhraní pro REST API, nezávisle na programovacím jazyku
- umožňuje stroji i člověku prohlížet a porozumět službě a jejímu významu
    - aniž by musel zkoumat zdrojový kód nebo sledovat síťový provoz
- dobře napsaná specifikace → snadné porozumění a interakce
- může být použita pro:
    - generování klientského i serverového kódu pro různé programovací jazyky
    - generování testovacích nástrojů a testovacích případů
    - generování dokumentace

Swagger

- framework pro definici rozhraní dle specifikace OpenAPI
- nejen vytvoření popisu, pokrývá celý životní cyklus:
    - **návrh** (Swagger Editor)
    - **dokumentace** (Swagger UI)
    - **testování** (Swagger Inspector)
    - **deployment** (Swagger CodeGen)

Swagger Editor

- strukturovaná definice rozhraní
- JSON, YAML
- editační okno, okno s průběžným zobrazením generované dokumentace
- možnost testování specifikace
- možnost generování kódu serverové i klientské části
- generování do velkého množství programovacích jazyků a frameworků

Model

- definice datových struktur

- primitivní typy (Integer, String,…)
- datové struktury

REST rozhraní

- definice operací na resource
- odkaz na datové struktury pomocí REF

Swagger CodeGen

- umožní generovat kostru:
    - aplikace pro implementaci služby
    - klienta, který službu bude volat

Swagger Inspector

- umožní spustit simulaci API
- zobrazuje REST request a response

APIARY

- nástroj pro definici rozhraní, podobné jako Swagger