# Technológiák

## Tiszta kód

A tiszta kód írásának talán legfontosabb célja, hogy bárki is vegye a keze közé a jövőben az általunk írt forrást, azt érezze, ez a rendszer a keze alá dolgozik, segíti az áttekintést, a megértést. Utat mutat az új belépőnek, hogyan is kell a rendszerhez illő további kódokat előállítani.

**Elnevezések:**

* kommunikálja jól a szándékot
* ne legyen félrevezető, bizonytalan jelentésű
* hosszabb elnevezések kis eltéréssel
* helyesírás
* kiejthető (az esetleges szóbeli kommunikáció miatt)
* típusra utalást NE
* probléma, megoldás domén elnevezések (account, JobQueue)
* ne mókázzunk (whack vs kill)
* class: főnév (kerüljük a túl általánost: pl. Manager, Info, Data)
* metódus: ige vagy igei alak
* adott koncepciót egységesen ugyanavval (retrieve, fetch, get)
* több koncepcióra ugyanazt ne
* elegendő kontextust kapjon (number: phone or people?)
* de többet ne
* ésszerűen minél rövidebb

**Metódus:**

* rövid
* részekre szedés
* követhető szerkezet
* kevés behúzási (indent) szint
* rövid sorok
* egy dolgot csináljon
* ugyanazon absztrakciós szinten maradjon (details vs essential concepts)
* The Stepdown rule – Lépésenkénti haladás
* argument: az ideális, ha nincs; háromnál több kétséges
* output argument: nehéz megérteni
* flag argument: ronda, jelzi, h nem egy dolgot csinál
* side effect: ne!
* command-query separation: vagy-vagy, ne mindkettőt
* exception: részesítsük előnyben a visszatérési értékkel szemben
* try-catch: a blokkok tartalma külön fgv
* hibakezelés: „egy dolog”
* enum hibatípus: dependency magnet
* DRY: don’t repeat yourself

**Switch statement:**

* veszélyes
* hosszú
* túl sokat csinálhat
* Open Closed Principle sérülhet
* -> Abstract Factory-ba rejtés

**Komment:** legtöbbször a szükséges rossz

* a nem pontos rosszabb, mint a hiányzó
* gyakran a rossz kódot próbálja kompenzálni
* jó:
  + jogi
  + informatív: pl. regexp magyarázata formátummal
  + szándék magyarázata
  + tisztázás (pl. a < b)
  + következményekre figyelmeztetés
  + TODO
  + megerősítés
  + API doc!
* rossz:
  + érthetetlen motyogás
  + redundáns (a kód OK)
  + félrevezető
  + kötelező jelleggel letudott
  + zaj (semmitmondó)
  + függvény vagy változó helyett
  + pozíció jelzés (forráson belüli szekcionálás)
  + kiiktatott kód!túl sok infó
  + API doc nem publikus kódra

**Kódformázás**

* függőleges:
  + kisebb fájlokat általában könnyebb megérteni
  + újság metafóra
  + szellős vs tömör (térköz)
  + távolság
  + változó deklaráció: közel a felhasználáshoz
  + példányváltozó: az osztály tetején
  + függő metódusok
  + koncepcionális affinitás (az összetartozó részel közelsége)
  + rendezés
* vízszintes
  + a rövidebb jobb (a technológia megengedne hosszút is..)
  + szellős vs tömör (térköz)
  + alignment
  + indent
* konvenció -> egyezség!

**Objektumok és adatstruktúrák**

* objektum:
  + absztrakció: megvalósítás elrejtése
  + adatreprezentáció
  + adatmanipuláló függvények
* adatstruktúra
  + adatok feltárása
  + nincsenek adatmanipuláló függvények
* Demeter törvénye: egy modulnak nem szabad tudnia az általa manipulált objektumok belsejéről (a.getB().getC() )
* hibrid: kerülendő
* a.getC() vs a.doSomethingOnC()

**Hibakezelés**

* fontos
* ne zavarja a programlogikát
* lsd metódus
* checked vs unchecked: pros vs cons
* Special Case Pattern
* return no null: SCP, NullObject Pattern etc

**Határvonalak**

* 3rd party kód: korlátozzuk/csomagoljuk be
* felfedező tesztesetek (ne a kódban)
* nem kiforrott elképzelések IF mögé rejtése
* világos definíció

**Unit teszt**

* TDD három törvénye -> rengeteg kód, tiszta!!
  + először hibázó teszt kód, utána a tesztelendő kód
  + csak annyi teszt kódot szabad írni, amennyi elegendő egy hibára; a fordítási hiba is hiba
  + csak annyi tesztelendő kódot szabad írni, amennyi a hibázó tesztet megjavítja
* test enable –ilities
* olvashatóság
* domain-specific test language
* hatékonyság nem szempont
* assert/teszt (TDD) vs 1 koncepció/teszt
* FIRST: fast, independent, repeatable, self-validating, timely

**Class**

* elrejtés
* kicsi
* egyetlen felelősség (Single Responsibility Principle)
* kohézió: példányváltózók száma vs az egyes metódusok által használtak száma
* a fentiből -> sok kis osztály
* változásra tervezés
* változástól elhatárolás: DIP

**Rendszerek**

* „Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build, and test”
* a rendszer konstruálásának és futásának szétválasztása
  + lusta inicializálás
  + factories
  + DI (ellentmondásos)
* separation of concerns
* cross-cutting concerns: AOP, Java Proxy
* döntés optimalizálás: ki, mikor
* standardok: demonstrálható értéket adjanak
* DSL!!!

**Emergence (kialakulás)**

* Simple Design
  1. minden teszt futtatása
  2. nincs duplikáció
  3. kifejezi a programozó szándékát
  4. class/method száma minimalizált
* 2-4: refaktorálás!

**Többszálúság**

* nehéz és komplex
* „mit” „mikor” szétválasztása
* nem mindig teljesítménynövelő
* rendszerint mély hatással van a rendszer struktúrájára
* a konténerek sem mentenek meg tőle
* bonyolítja a kódot
* hibák nehezen felderíthetők
* stb.

**Gyanús kód** (code smell - Martin Fowler)

* rossz kommentek
* környezet:
* egyetlen lépésben végre nem hajtható build
* egyetlen lépésben végre nem hajtható teszt
* lsd. metódus problémák
* általános
* egyetlen forrásfájlban több nyelv
* nyilvánvaló viselkedés nem implementált
* nem korrekt viselkedés határvonalon: minden eset lefedése
* felülírt biztonsági pontok (pl. nem működő teszteset kiiktatása)
* DRY megsértése
* hibás absztrakciós szintű kód
* ősosztály függése a leszármazottól
* túl sok információ (modul API)
* zűrzavar, döglött kód
* nagy függőleges szeparáció
* inkonzisztencia (kövesd a konvenciókat)
* mesterséges (szükségtelen?) csatolás
* funkció-irígység (Demeter?)
* selector argument (flag argument)
* homályos szándék
* rosszul elhelyezett felelősség
* alkalmatlan static
* nem érthető algoritmus (tesztek lefutása nem elég)
* if/else, switch/case
* elnevezés nélküli mágikus konstans
* precizitás hiánya
* a struktúra megelőzi a konvenciót
* feltételes szerkezet elrejtése
* negatív feltétel
* rejtett időbeli csatolás
* önkényesség

## REST

A REST (Representational State Transfer) egy szoftverarchitektúra típus elosztott hipermédia rendszerek számára. A kifejezést Roy Fielding vezette be és definiálta 2000-ben a doktori disszertációjában. (Fielding, 2000).

Egy REST architektúrát követő rendszernek öt (opcionálisan hat) követelménynek kell megfelelnie, melyek a következők:

**Kliens-szerver architektúra**

A kliensek el vannak különítve a szerverektől egy egységes interfész által. Az érdekeltségek ilyen nemű szétválasztása azt jelenti, például, hogy a kliensek nem foglalkoznak adattárolással, ami a szerver belső ügye marad, és így a kliens kód hordozhatósága megnő. A szerverek nem foglalkoznak a felhasználói felülettel vagy a kliens állapotával, így a szerverek egyszerűbbek és még skálázhatóbbak lehetnek. A szerverek és kliensek áthelyezhetőek és fejleszthetőek külön-külön is, egészen addig amíg az interfész nem változik meg.

**Állapotmentesség**

Az állapotmentesség azt jelenti, hogy a szerveren nem tárolják a kliens állapotát a kérések között. Minden egyes kérés bármelyik klienstől tartalmazza az összes szükséges információt a kérés kiszolgálásához, és minden állapotot a kliens tárol. A szerver lehet állapottartó. A megszorítás

This constraint induces the properties of visibility, reliability, and scalability.

**Gyorsítótárazhatóság**

Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

A gyorsítótárazás lehetővé teszi, hoigy egyes kliens-szerver interakciót elkerüljünk, növeli a válaszidőt, a teljesítményt és a skálázhatóságot.

**Egységes interfész**

The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components ([Figure 5-6](http://www.ics.uci.edu/%7Efielding/pubs/dissertation/rest_arch_style.htm#fig_5_6)). By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. Implementations are decoupled from the services they provide, which encourages independent evolvability. The trade-off, though, is that a uniform interface degrades efficiency, since information is transferred in a standardized form rather than one which is specific to an application's needs. The REST interface is designed to be efficient for large-grain hypermedia data transfer, optimizing for the common case of the Web, but resulting in an interface that is not optimal for other forms of architectural interaction.

Az egységes interfész kliens és szerver között egyszerűsíti és kettéválasztja az architektúrát, és lehetővé teszi, hogy egymástól függetlenül fejlődjenek az egyes részek. Az interfész négy irányadó elve alább kerül részletezésre.

**Réteges felépítés**

By restricting knowledge of the system to a single layer, we place a bound on the overall system complexity and promote substrate independence

Egy kliens általában nem tudja megmondani, hogy direkt csatlakozott-e a végpont szerverhez, vagy közvetítő segítségével. A közvetítő szerverek megnövelhetik a rendszer skálázhatóságát terheléseloszlás kiegyenlítéssel és megosztott gyorsítótárak használatával.

**Code-on-Demand** (opcionális)

The final addition to our constraint set for REST comes from the code-on-demand style of [Section 3.5.3](http://www.ics.uci.edu/%7Efielding/pubs/dissertation/net_arch_styles.htm#sec_3_5_3) ([Figure 5-8](http://www.ics.uci.edu/%7Efielding/pubs/dissertation/rest_arch_style.htm#fig_5_8)). REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility. However, it also reduces visibility, and thus is only an optional constraint within REST.

A REST eredetileg a HTTP keretein belül lett leírva, de nem korlátozódik erre a protokollra

|  |  |
| --- | --- |
| HTTP ige | jelentése a CRUD terminológiában |
| POST | Erőforrás létrehozása |
| GET | Erőforrás olvasása |
| PUT | Módosít egy erőforrást |
| DELETE | Töröl egy erőforrást |

<http://martinfowler.com/articles/richardsonMaturityModel.html>

Dr. Leonard Richardson put together a maturity model that interprets various levels of compliance with RESTful principles, and grades them. It describes 4 levels, starting at **level 0**. Martin Fowler [has a very good write-up on the maturity model](http://martinfowler.com/articles/richardsonMaturityModel.html)

* **Level 0**: the Swamp of POX - at this level, we’re just using HTTP as a transport. You could call SOAP a **Level 0** technology. It uses HTTP, but as a transport. It’s worth mentioning that you could also use SOAP [on top of something like JMS](http://www.w3.org/TR/soapjms/) with no HTTP at all. SOAP, thus, is not RESTful. It’s only just HTTP-aware.
* **Level 1**: Resources - at this level, a service might use HTTP URIs to distinguish between nouns, or entities, in the system. For example, you might route requests to /customers, /users, etc. XML-RPC is an example of a **Level 1** technology: it uses HTTP, and it can use URIs to distinguish endpoints. Ultimately, though, XML-RPC is not RESTful: it’s using HTTP as a transport for something else (remote procedure calls).
* **Level 2**: HTTP Verbs - this is the level you want to be at. If you do **everything** wrong with Spring MVC, you’ll probably still end up here. At this level, services take advantage of native HTTP qualities like headers, status codes, distinct URIs, and more. This is where we’ll start our journey.
* **Level 3**: Hypermedia Controls - This final level is where we’ll strive to be. Hypermedia, as practiced using the [HATEOAS](http://en.wikipedia.org/wiki/HATEOAS) ("HATEOAS" is a truly welcome acronym for the mouthful, "Hypermedia as the Engine of Application State") design pattern. Hypermedia promotes service longevity by decoupling the consumer of a service from intimate knowledge of that service’s surface area and topology. It **describes** REST services. The service can answer questions about what to call, and when. We’ll look at this in depth later.



<https://en.wikipedia.org/wiki/HATEOAS>

http://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/

## OpenAPI (swagger)

The OpenAPI Specification is an API description format or API definition language. Basically, an OpenAPI Specification file allow you to describe an API including (among other things):

* General information about the API
* Available paths (/resources)
* Available operations on each path (get /resources)
* Input/Output for each operation

https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md

## Spring MVC

## Spring HATEOAS

http://projects.spring.io/spring-hateoas/