55. TDD, REFAKTORIZACE, VLASTNICTVÍ A SPRÁVA ZDROJOVÉHO KÓDU V TÝMU

- návrh SW udává z čeho se bude skládat, jak jednotlivé prvky budou na sobě závislé, jaká bude celková struktura, organizace a hierarchie SW architektury a jaká bude vnitřní struktura prvků
- udává chování a funkčnost budoucího systému a jak spolu budou jednotlivé elem. komunikovat a jaké budou mít závislosti a třeba také to jaký bude tok v tom systému a v jakých stavech se může systém nacházet

→ z toho vychází postup a principy implementace, které by se měly návrhu držet a nějakým způsobem jej reflektovat a naplnit

- při realizaci/vývoji/implementaci se může narazit na nějaké problémy s HW, architekturou atd. a je někdy nutné návrh upravit → výsledný návrh se pak může dost lišit
- mohou se využívat různé přístupy z Agile, MDD, RDD, a také UP(RUP) a extrémního programování
  ↳ responsibility

Vývoj řízený testem (TDD)

- udává postup vývoje kdy se nejprve vytvoří testy které specifikují požadovanou funkčnost a schopnosti systému
- implementuje se potom cíl těchto testů do takového stavu aby všechny testy prošly
- nejprve jsou testy, pak implementace
- testy vycházejí ze specifikace, analýzy a hlavně z návrhu
- testy tedy vyjadřují funkčnost SW, jeho omezení a to, jak má fungovat
- implementace poté plní požadavky dané testy
- výhodou je i to že jakmile je nějaká část implementována tak je rovnou i otestována
- pokud se SW rozšiřuje nebo refaktoruje, tak je možné vždy ověřit zda splňuje testy
- pro programátory je "návykové" vidět zelené fajfky u všech testů
- může se třeba jednat rovnou o akceptační testy které se implementací splňují
- využívá Agilní návrh a vývoj a také UP (RUP)
- je to i automatizovaná a opakovatelná verifikace – že SW splňuje model/návrh
- změny v systému / SW jsou ihned testovatelné
- vytvoří se pro danou třídu testovací třída která obsahuje testovací metody pro všechny veřejné metody původní třídy a slouží k jejímu testování – assertEmpty, assertZer
- dokud testy nejsou splněny, upravuje se implementace tak aby byly ....

Refaktorizace

- rozčlenění, restrukturování a přeskupení kódu tak, aby byla zaručena lepší přehlednost, pochopitelnost, čitelnost, debuggovatelnost, rozšiřitelnost a další
- dělá se například ve chvíli kdy se má SW rozšiřovat nebo přidávat další moduly nebo když se opravují chyby
- zlepšuje a usnadňuje práci v týmu
- kód který umí číst PC umí psát každý, ale dobrý programátor píše kód, kterému rozumí lidé
- velmi obsáhlá řešení se dělí na více menších (SOLID)
- obsáhlé třídy jsou rozděleny do více menších
- obsáhlé metody jsou rozděleny a vytvořeny privátní
- řeší se části kde metody mají zbytečně moc parametrů
- řeší se opakování stejných částí kódu – vytvoří se z toho jedna metoda a ta se volá napříč.
- snažíme se aby byly jednotlivé prvky znovupoužitelné
- co nejlepší využití OO paradigmat a principů
- může využívat RDD, GRASP a SOLID principy a cleancode
- dokumentace tříd, metod a bloků kódu

⇒ refaktorizací se nemění funkčnost SW ale struktura kódu a uspořádání = restrukturalizace zdrojového kódu bez změny chování

- v dnešní době obsahují IDE nástroje pro refaktorizaci – JetBrains atd..
- často používaná data mohou být jako objekty
- odstraňuje silné závislosti tříd/metod

⟶ je dobré mít testy a ověřovat jestli refaktorizace nezměnila funkčnost (chování)

- problémem mohou být databáze a jejich silná provázanost s daty a aplikační logikou ⇒ je dobré vytvořit mezivrstvu a když se vyžádá nějaký refaktoring tak se jen upraví vrchní rozhraní té mezivrstvy

Verzování

- využívá se třeba Git, github, bitbucket a další ...
- zjednodušuje správu kódu, kontrolu, navracení se, kontrolu práce, práci v týmech, code review, schvalování změn a mnoho dalšího
- máme třeba produkční verzi kódu MASTER, vývojovou DEVELOP, verzi které se připravují do produkce (RELEASE), verzi ve které je vyvíjena nějaká nová funkčnost (FEATURE), nebo verzi ve které se opravují chyby objevené v produkci (BUG-FIX, HOT-FIX) a verzi kde se refaktoruje (CLEAN-UP)
- provádí se merge změn mezi větvemi, cherry pick, pull requests, commits, tagy a další

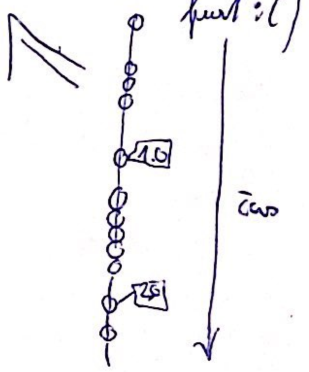
  ⇒ Git

- vlastnictví kódu ←
  - žádné – všichni mohou dělat vše – nikdo nenese zodpovědnost
  - individuální (silné) – změny na určité části kódu může dělat jen ten kdo je za něj zodpovědný
  - slabé – změny může dělat i někdo jiný než vlastník ale musí mít od vlastníka svolení
  - sdílené – kód je všech a každý by měl přispívat

– kód se nachází v nějakém společném repositáři ale je zde více větví (v gitu, jinak řekněme verzí)
– přístup k repositáři může být veřejný a nebo jen pro nějakou skupinu spolupracovníků které v něm mohou mít různě definovaná práva
– repositář či více repositářů může být zahrnuto v nějakém projektu a firmě či jinak
– každý projekt/SW může mít svůj repositář se zdrojovými kódy
– společné vlastnictví kódu je v malých a středních firmách ideální
– kód v repositáři je měněn využitím těch dříve uvedených principů (master, develop, feature, release, hot-fix, atd...)

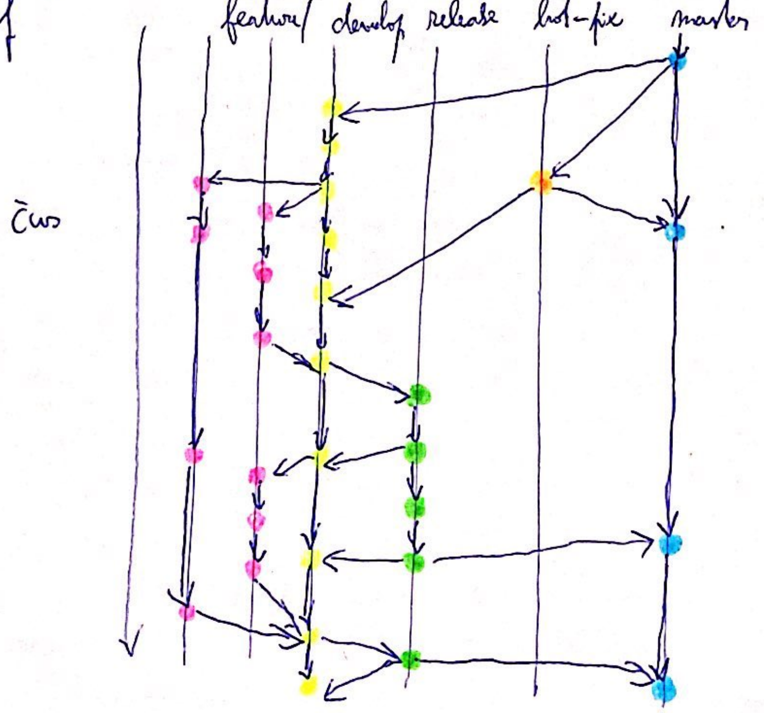
Sdílený kód s lineární historií

– historie kódu je jen přímka bez větvení
– než vývojář ze společného kódu (base) oddělí kterékoli část a upraví a mergne zpět, tak musí udělat rebase a nikdo pak nesmí nic mergnout než to mergne on jinak by musel dělat zase rebase
– přehlednost a jednoduchost správy kódu :)
– existuje jen jedna varianta sdíleného kódu a ta musí být perfektní :(
– nelze ukládat rozpracované věci – jen finální :(a další nevýhody (release first :(
→ jedna větev git



Sdílený kód s větvenou historií

– zmíněná klasika
– používá se nejvíc
– více větví (git) – master, release, develop, feature, hot-fix, ..
– více verzí kódu najednou
– vývoj třeba i v několika alternativních směrech
– spolupráce na sdíleném kódu
– každá větev svůj účel
– kontrola před přijetím změn
– acyklický graf


feature/
develop
release
hot-fix
master
čas