Anti-vzory a zavedené praktiky při vývoji software

Marek Rychlý

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů

Přednáška pro AIS
12. prosince 2019

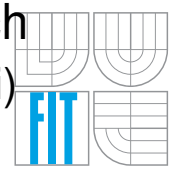
Obsah

Anti-vzory (Anti-patterns)

- Anti-vzor popisuje něco, čeho je třeba se vyvarovat.
  (při analýze a návrhu SW či UI, programování, procesním řízení, atp.)
- Znalost anti-vzorů znamená nedělat chybná rozhodnutí.
  (přestože tato rozhodnutí mohou zdánlivě vypadat jako správná)
- Anti-vzory zavádí „slovník“, který usnadňuje komunikaci.
  (jako to dělají vzory, kde např., když se řekne „observer“, každí ví, co to znamená)
  - použitelné při představování návrhů řešení,
  - při diskuzi o navrhovaném řešení,
  - v dokumentaci pro nastínění možných problémů,
  - pro refactoring stávajícího řešení (návrhu, kódu, atp.), . . .
- Anti-vzorů je velké množství (větší než v případě vzorů).
  (následující představují některé z důležitých anti-vzorů pro návrh/vývoj SW)

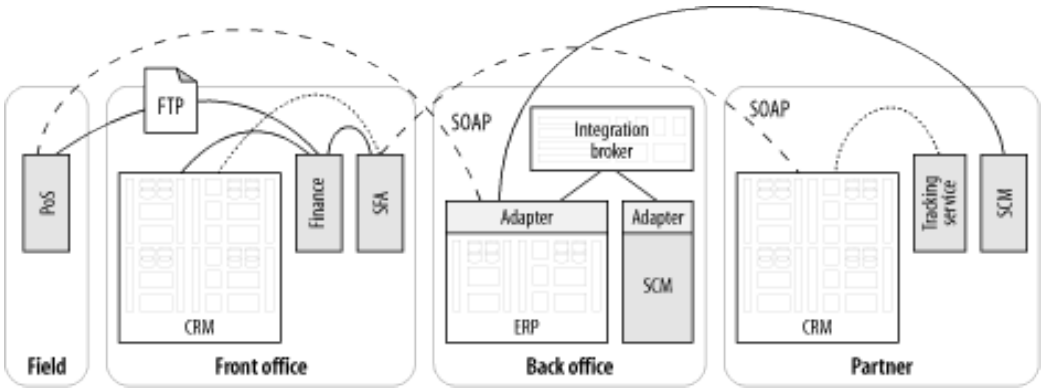
Design Anti-pattern: Abstraction Inversion

- Potřeba použít něco, co je schováno za abstrakcí, vně této abstrakce.
  (abstrakce v popisu výše může být např. rozhraní třídy)
- Příznak:
  Řešení jednoduchých věcí pomocí složitých, které jediné jsou dostupné.
  (např. zpětné počítání výsledků jednoduché operace z výsledků složité)
- Příklad:
  Třída „Product“ má metodu „getSuplierName“, avšak nikoliv metodu „getSuplier“. Tedy vrací pouze název dodavatele, nikoliv jeho ID či objekt (dle druhu úložiště). Bude-li někdo potřebovat jinou vlastnost dodavatele, než jméno, bude muset zbytečně vyhledat dodavatele (dle jména).
  (tedy dělat něco, co již prováděno a schováno uvnitř metody „getSuplierName“)
- Příčina:
  vysokoúrovňová abstrakce; orientace na použití, nikoliv na obecný návrh
- Řešení:
  rozdělení složitých metod na jednodušší, publikování i těch jednoduchých
  (tak, aby klient mohl při nespokojenosti se složitou metodou použít její dílčí části)
- Souvisí s návrhovými vzory: Facade, Strategy



Design AP: Big Ball of Mud / Accidental Architecture

- Řešení je nějak slepené dohromady, bez jasné architektury.
  (tak, jak se časem přilepovali nové a nové kousky)
- Příznak:
  Pro rozšíření nelze použít stávajících možností, nebo je jejich více.
  (tj. opět potřeba přilepit kousek navíc, nebo existuje několik částí řešící to samé)
- Příčina:
  (neřízený) inkrementální vývoj (produkčního) systému, málo času
- Řešení:
  refaktorizace, jasná rozhraní, necyklické závislosti, volné vazby



(diagram převzat z „David A. Chappell: Enterprise Service Bus, 2004“)

Design Anti-pattern: Magic push-button

- Stisk tlačítka v UI provede mnoho z vnějšku špatně viditelných akcí.
  (obsluha tlačítka je „black-box“ pro uživatele či pro programátora)
- Příznak:
  Uživatel dlouho vyplňuje formulář a pak stiskne tlačítko. Metoda řadiče obsluhy UI tlačítka v programu je příliš komplikovaná.
  (tedy metoda provádí business logiku, místo aby to jen delegovala dále)
- Příklad:
  Třída „MailFormController“ má metodu „sendButtonClick“, která ověří správnost zadaných adres příjemců, sestaví MIME email s přílohami, podepíše ho pomocí certifikátu, naváže komunikace s SMTP serverem a email odešle, a nakonec zobrazí výsledek operace uživateli.
- Příčina:
  vývoj započatý programováním UI, neobjektový návrh, postupné (neřízené) přidávání činností provedených po stisku tlačítka
- Řešení:
  refaktorovat, rozdělit prováděné aktivity, zavést objekty vykonavatelů
- Souvisí s návrhovými vzory: delegation, chain of responsibility

Design Anti-patterns

další, dříve nezmíněné, návrhové anti-vzory

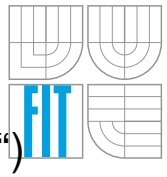
- Database-as-IPC
  (použití databáze pro komunikace sub-systémů, ne pro persistenci dat)
- Gold plating
  (neustálé a zbytečné vylepšování určité části, přestože již byla dobře použitelná)
- Inner-platform effect / Soft code
  (zbytečně velká možnost přizpůsobit systém, prostředky přizpůsobení nabídnuté uživateli připomínají prostředky pro vývoj systému; např. uživatel může upravovat UI v HTML, programovat vlastní funkce v zabudovaném skriptovacím jazyce, atp.)
- Interface bloat / Fat interface
  (interface s mnoha nesouvisejícími operacemi, musí se zbytečně implementovat)
- Race hazard / Race condition
  (bude-li akce vykonávána souběžně s jinou, bude nedeterministický výsledek)

Object-oriented Anti-pattern: Anemic Domain Model

- Akce s doménovými objekty se provádí z vnějšku, nikoliv jejich metodami.
  (doménové objekty jsou jen datové záznamy, nic nevykonávají)
- Příznak:
  Doménové objekty mají jen atributy, případně gettery/settery. Existují objekty pro manipulaci/operace s jednotlivými doménovými objekty.
- Příklad:
  Třída „BankovníÚčet“ má jen metody „getStavÚčtu“ a „setStavÚčtu“. Existuje třída „OperátorÚčtů“, která má následující metody s parametry objektů „BankovníÚčet“: „vkladNa“, „výběrZ“, „převodMezi“, „zrušení“, „přidáníDisponentaK“, atp.
- Příčina:
  neobjektový návrh, chybné přiřazení zodpovědností, nevhodná platforma/rámec
- Řešení:
  refaktorovat, přesunout metody dle (správných) zodpovědností

Object-oriented Anti-pattern: Call Super

- Nadtřída vyžaduje od podtříd volání její metody, kterou budou přepisovat.
  (počítá s tím, že podtřídy ve své metodě přepisující metodu nadtřídy zavolají tuto přepisovanou metodu nadtřídy a něco udělají s výsledkem, který vrátí)
- Příznak:
  Časté volání metod nadtřídy (super) v podtřídách, zejména uprostřed či na konci těla jejich přepisujících metod (tedy ne na začátku).
- Příklad:
  Třída „VýběrBankomatem“ má metodu „strženáČástkaZÚčtu“, která bude přepsána pro výběry s poplatkem, ale i nadále používána. Její pod-třída „VýběrSPoplatkem“ ji přepíše tímto: zavolá původní, aby zjistila částku, k ní připočítá 5% poplatek a výsledek bude návratová hodnota.
- Příčina:
  chybný návrh míst pro rozšíření nadtřídy (neřízené rozšíření)
- Řešení:
  nová abs. metoda nadtřídy pro přidání funkc. do přepisované metody
  (rozšíření ne přepisem, ale implementací abs. metody; např. nová abs. metoda „procentaPoplatku“ použitá v „strženáČástkaZÚčtu“, obojí v „VýběrBankomatem“)



Object-oriented Anti-pattern: God Object

- Objekt má či implementuje příliš mnoho atributů či metod.
- Příznak:
  Velké množství atributů a z vnějšku viditelných metod.
- Příklad:
  Třída „OdešliZprávu“ původně uměla posílat emaily. Pak se přidalo posílání zpráv SMS a instant messaging. Pak se doplnilo šifrování a podepisování zpráv a další.
- Příčina:
  inkrementální vývoj, chybná dekompozice na třídy, chybné přiřazení zodpovědností
- Důsledek:
  pokud chce někdo použít nějakou metodu, musí použít celý objekt
- Řešení:
  refaktorovat, rozdělit na menší třídy/objekty, které se rozšiřují a skládají
  (rozdělit dle funkcionality, závislostí, použití, atp.; objektový návrh)

Object-oriented Anti-patterns

další, dříve nezmíněné, objektově orientované anti-vzory

- BaseBean
  (použití metody poskytované potřebnou třídou tím, že ji z této třídy podědíme, aniž by platilo NašeTřída „is a“ PoděděnáTřída; tj. použití dědičnosti místo delegace)
- Circle-ellipse problem
  (objekty „Circle“ jsou specializací „Elipse“, protože kružnice je elipsa, jenže pak nemohou implementovat její metody specifické jen pro elipsy, např. „stretchX“)
- Circular dependency
  (cyklické závislosti mezi objekty; je třeba rozbít, např. použitím vzoru Observer)
- Object cesspool
  (objekty vrácené a znovu půjčené do/z pool nemají výchozí stav; správce poolu je zodpovědný za to, že stav půjčovaných objektů bude výchozí, tj. resetuje vracené)
- Object orgy
  (nedodržování zapouzdřenosti objektů vede k tomu, že nelze „design by contract“)
- Sequential coupling
  (metody třídy musí být volány v určitém pořadí; vhodné zafixovat volání v tomto pořadí pomocí vzoru Template method, tedy přidáním metody volající tu sekvenci)

Programming Anti-patterns (a další)

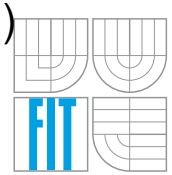
- Busy waiting
  (aktivní čekání, kontrola něčeho v cyklech, namísto použití událostí/observer)
- Cut/Copy and paste programming / Cargo cult programming
  (znovupoužití kopírováním či vytržení z kontextu velmi ztíží údržbu kódu; pokud programátor navíc kopírovanému nerozumí (cargo cult) je to vážný problém)
- Functional decomposition
  (skládání programu z funkcí/metod, nikoliv z objektů; chybný objektový návrh)
- Golden hammer
  (oblíbená technologie je použita i v nevhodných případech; např. CLI apl. v PHP)
- Lava flow / Dead code / Boat anchor
  (opuštěný/nepoužívaný kód „pro jistotu“ ponechán v projektu, nikdo neví k čemu)
- Mushroom management
  (vývojáři jsou někde zavřeni „ve tmě“ a krmeni, bez interakce s uživateli)
- Soft code / Hard code
  (uložení business logiky do konfigurace a ne do kódu, nebo naopak zafixování proměnných vlastností prostředí do kódu, místo do konfigurace, je špatně)

Anti-pattern: Dependency Hell

- Kvůli fci z knihovny a závislostem je potřeba hodně dalších knihoven.
  (peklo je to hlavně, je-li potřeba jen pár jednoduchých funkcí)
- Příznak:
  Velké množství závislostí, neúměrná velikost knihoven.
- Příčina:
  chybné členění fcí do knihoven, chybný návrh, nepoužití volných vazeb
- Řešení:
  jiná knihovna nebo vlastní implementace (zkopírovat implementaci)

Vlastnictví kódu

- Vlastnictví právní (majetek) vs. zodpovědnosti (správa).
  (nás bude zajímat to druhé, tedy vlastník se o kód stará, zodpovídá za něj)
- Vlastnictví kódu se řeší při týmové spolupráci.
  (kdo může měnit který kód používaný v projektu)
- Jednotkou vlastnictví kódu může být komponenta, modul, třída, metoda, soubor, nebo jakýkoliv jiný logický celek.
- Různé přístupy:
  - žádné vlastnictví kódu
    (všichni programátoři mohou měnit jakoukoliv oblast kódu)
  - silné/individuální vlastnictví kódu
    (jednoznačný vlastník, jedinec; programátor má může měnit je to, co vlastní)
  - slabé vlastnictví kódu
    (opět vlastník-jedinec; lze měnit cokoliv, se souhlasem/informováním vlast.)
  - společné/kolektivní vlastnictví kódu (extrémní programování)
    (každý by měl vylepšovat jakýkoli kód, kdykoliv je k tomu příležitost)

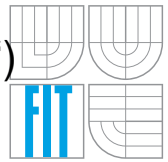


Žádné vlastnictví kódu

- všichni programátoři mohou měnit jakoukoliv oblast kódu
  (a to kdykoliv to potřebují, např. pro kompatibilitu s jiným kódem)
- může být výhodou v malých projektech, protože je to rychlé
- třeba dávat dobrý pozor na zachovávání účelu a konzistenci kódu
  (neměl by se měnit podle aktuálních potřeb na úkor jeho skutečného zaměření)
- nikdo není zodpovědný za kvalitní design
  (vizte anti/vzory diskutované v první části přednášky)

Silné/individuální vlastnictví kódu

- jednoznačný vlastník (jedinec) pro každou část kódu
- programátor má může měnit je to, co sám vlastní
- potřebuje-li změnit, co nevlastní, musí změnu požádat vlastníka
  (vlastník zkontroluje/opraví/zamítne požadavek a změnu v kódu sám provede)
- lepší kontrola prováděných změn a jasná zodpovědnost za změny
- nevýhodou je nákladnost provedení rozsáhlých změn
  (např. v případě refaktorování kódu, které mění kód vlastněný mnoha vlastníky)
- pro obtížnost změn ztratí členové týmu často motivaci k provádění změn, takže se typicky omezí refaktorování
- silné vlastnictví kódu může být vhodné při stabilním kódu
  (tedy kódu, který se moc často nemění, např. je v módu „údržba“, nikoliv „vývoj“)



Slabé a společné vlastnictví kódu

Slabé vlastnictví kódu:

- opět jednoznačný vlastník (jedinec) pro každou část kódu
- programátor může sám měnit cokoliv, ale musí požádat vlastníka o souhlas nebo ho aspoň informovat – dle úvahy či pravidel
  (programátor žádající změnu by měl být schopen odhadnout její závažnost)
- odstraňuje nákladnost provádění změn v silném vlastnictví kódu

Společné vlastnictví kódu:

- vhodné v případě, že programátoři zvládají extrémní programování
  (zejména jsou tedy schopni se svědomitě starat o všechen kód)
- vhodné spíše pro menší týmy, kdy vlastnictví kódu tvořeného týmem je společné, zatímco vztahy mezi týmy jsou řízené silným/slabým vlastnictvím kódu spravovaným jednotlivými týmy

Zavedené praktiky při vývoji software

- Vývojem software se zde myslí práce vývojáře (developera).
  (tedy především programování a návrh dílčích řešení, nikoliv celkové architektury)
- Tým pracuje na stejném projektu, potřebují společně měnit kód.
- Předpokládáme slabé případně společné vlastnictví kódu.
  (tyto jsou v praxi malých a středních týmů většinou nejvhodnější)
- Vyvíjený kód je uložen ve společném repozitáři a měněn…
  - implementací nových funkcí (tj. „feature“)
  - opravou chyb způsobujících problémy (tj. „bug-fix“)
  - rychlým řešením chyb-incidentů (tj. „hot-fix“)
  - laděním kódu pro vydání (tj. „release“)
  - refaktorizací kódu (tj. „clean-up“)
  - atd.

Základní aktivity při práci se sdíleným kódem

- Na projektu je potřeba pracovat souběžně.
  (tj. řešit mnoho na předchozí straně uvedených typů změn současně)
- Na sdíleném/stejném kódu nelze pracovat v reálném časem.
  (více vývojářů nemůže současně upravovat stejný kód, problém kompatibility)
- Proto jsou změny prováděny v dávkách (tj. aplikací „patch“).
- Běžný postup:
  1. vývojář začne pracovat na kódu v určitém stavu (tj. „base“)
  2. vývojář vyvíjí/implementuje (tj. „coding“)
  3. občas vývojář aktualizuje svoji (měněnou) kopii kódu dle aktuálního společného kódu a vyřeší případné nekompatibility (tj. „rebase“)
  4. až je hotovo, tak vývojář promítne svoje změny do aktuálního společného kódu (tj. „merge“)

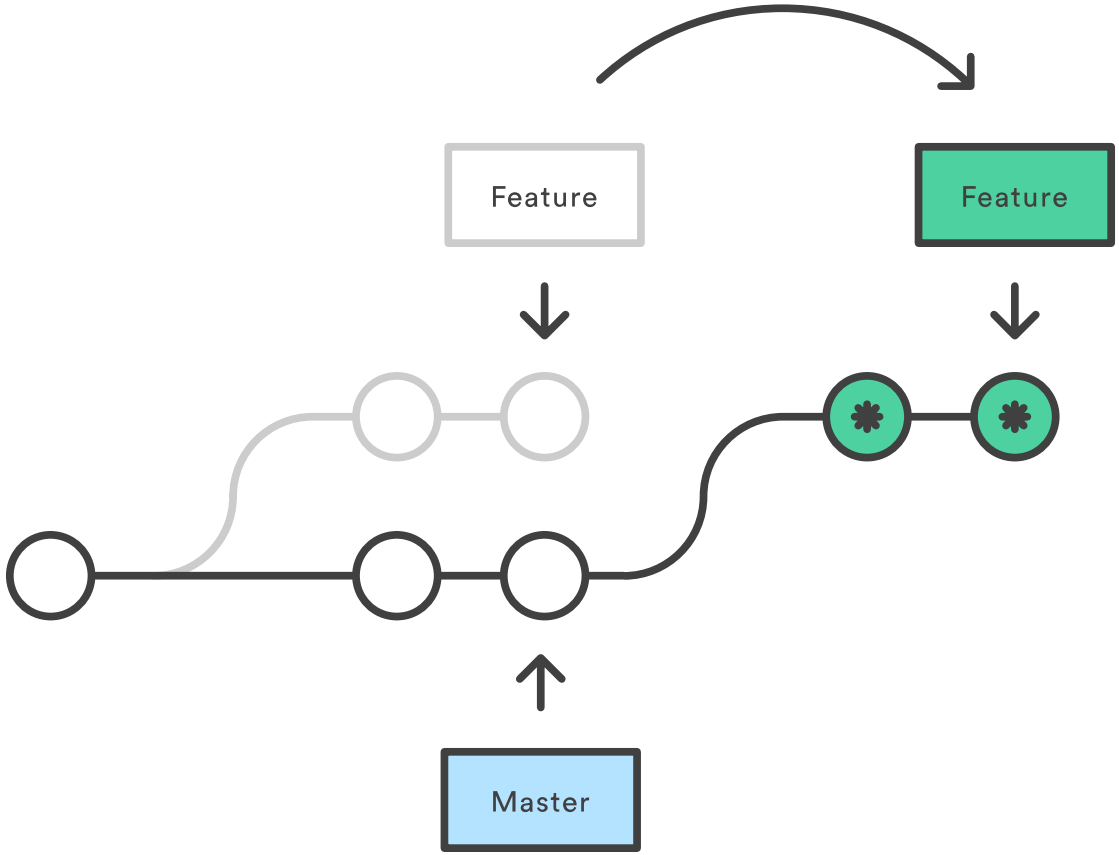
Sdílený kód s lineární historií

- Historie sdíleného kódu je vidět jako přímka, bez větvení.
  (tj. nikdy veřejně neexistovalo více variant sdíleného kódu)
- Vývojář musí před aplikací své změny na společný kód provést „rebase“, tj. sladit změnu s aktuálním stavem sdíleného kódu.
  (pak aplikuje změnu bez konfliktů a tedy pouze posune stav sdíleného kódu)
- Výhodou je přehlednost a jednoduchost správy kódu.
  (každý okamžitě ví, jaký je aktuální stav)
- Nevýhody však převažují:
  - existuje jen jedna varianta sdíleného kódu, musí být vždy perfektní
    (zejména musí jít vždy přeložit, spustit, případně také uvolnit jako „release“)
  - na společný kód lze aplikovat pouze hotové změny (důsledek)
    (těžko zveřejníme k posouzení/pomoci/diskuzi nedokončenou změnu)
  - vývojáři těžko spolupracují, rozpracované kopie kódu se nesdílí
    (sdílený kód existuje pouze v jedné variantě; spolupráci si musí řešit sami)
  - mezi „rebase“ změny a její aplikací se sdílený kód nesmí změnit
    (jinak musí být další rebase; není tedy čas na diskuzi o změně)

Sdílený kód s větvenou historií

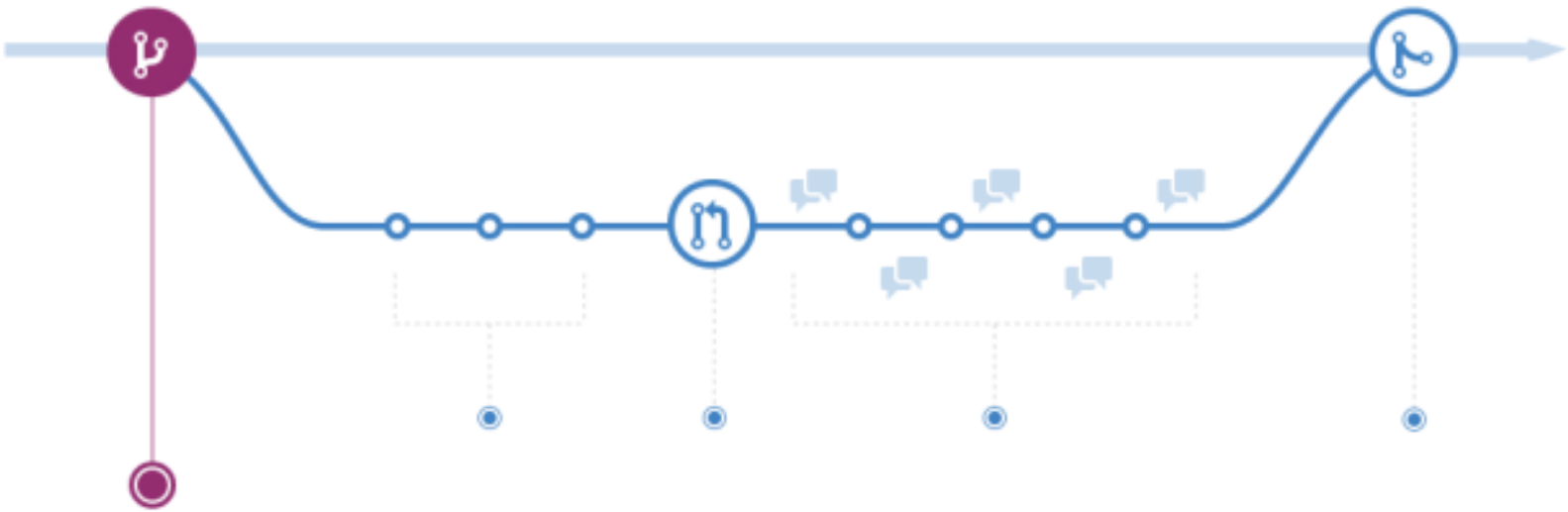
- Historie sdíleného kódu je vidět jako acyklický graf.
  (tj. veřejně existovalo více variant/větví sdíleného kódu)
- Každá větev (tj. „branch“) může mít svůj účel, např.
  - stabilní/produkční kód bez známých chyb (tzv. „master“ větev)
  - sdílený aktuálně vyvíjený kód (tzv. „development“ větev)
  - kód v přípravě na konkrétní „release“
    (odštěpením z „development“, tj. zmrazením, dokončený půjde do „master“)
  - sdílený kód pro implementaci rysu, opravu chyby, čištění, atp.
    (feature, bug/hot-fix, clean-up, atp.; odštěpený z a tekoucí do různých větví)
- Vývojáři mohou lépe spolupracovat na sdíleném kódu.
  (větve, kde se spolupracuje, se sdílí)
- Vlastníci kódu mohou lépe kontrolovat přijetí změn.
  (není třeba spěchat s aplikací změny; ta postupně probublává větvemi)

GIT Rebase



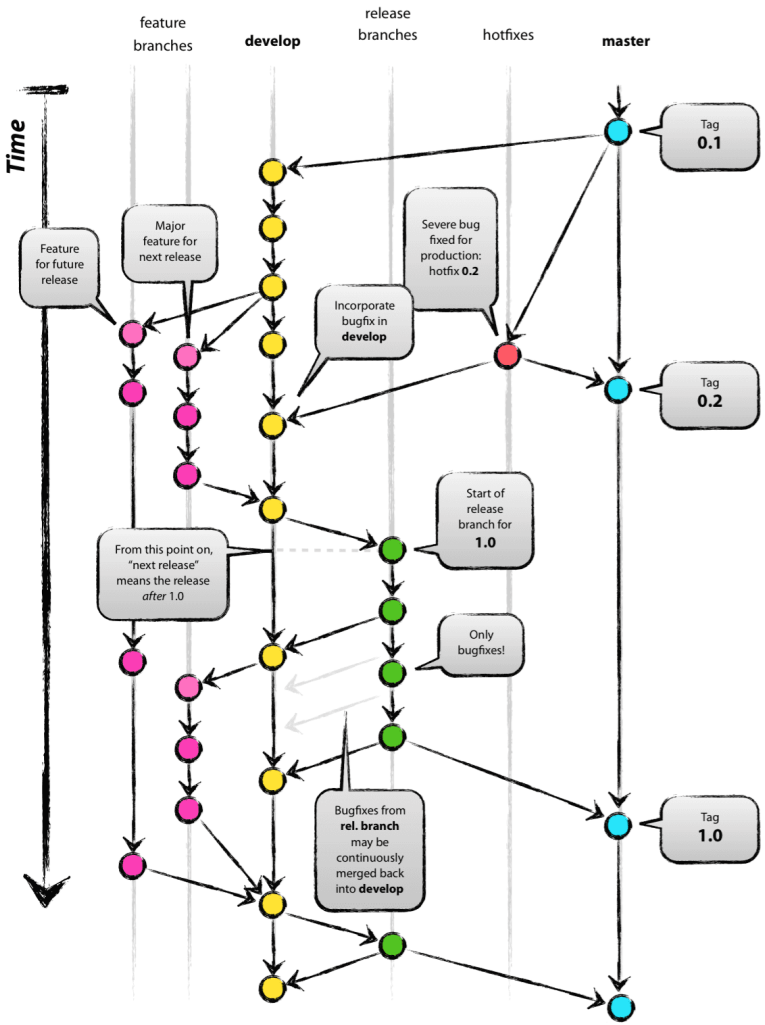
(převzato z „Rewriting history“)

GIT Model by GitHub



(převzato z „Understanding the GitHub Flow“)

GIT Model by nvie



(převzato z „A successful Git branching model“)

Literatura

Brown, W. H., Malveau, R. C., McCormick, H. W. S., and Mowbray, T. J. (1998).

AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.

John Wiley & Sons, Inc., New York, NY, USA, 1st edition.

Chacon, S. (2009).

Pro Git.

Apress, Berkely, CA, USA, 1st edition.

Šimonek, J. (2014).

Průzkum vlastnictví kódu ve velké organizaci.

Master's thesis, Vysoké učení technické v Brně, Fakulta informačních technologií.