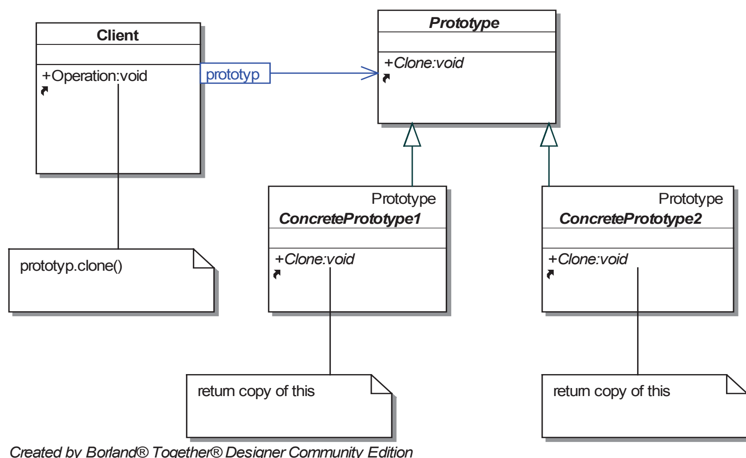
Struktura vzoru



Obrázek 88 Návrhový vzor prototyp

- Prototype – deklaruje rozhraní pro vlastní kopii
- ConcretePrototype – implementuje operaci pro vlastní kopírování
- Client – vytváří nový objekt požádáním prototypu o naklonování

Použití

- třídy, jejichž instance se mají vytvořit, jsou specifikovány za běhu
- nechceme budovat třídní hierarchie továrny souběžnou s hierarchií produktů
- instance třídy mají několik málo různých kombinací stavu, může být výhodnější vytvořit prototypy a ty klonovat

Ostatní

- přes manažera se dobře přidávají a odstraňují seznamy tříd v runtime
- vhodné použít např. při konfiguraci systému

Další příklad - grafický editor hudby

- vybírání not a jiných prvků notace graficky znázorněných a manipulace s nimi
- dva stromy dědičnosti Tool – manipulace s notacemi a Graphic – samotné notace
- sloučení dvou hierarchií dohromady – ne dědění (velký počet tříd), ale skládání objektů, tj. např. zápis noty: třída MoveTool a nový objekt typu GraphicNota jako atribut s rolí theGraphic
- volba třídy GraphicNota – vzor prototype, tj. ve třídě Graphic abstraktní metoda Clone() aneb výběr konkrétní třídy rozhraní Graphic převeden na výběr prototypické instance

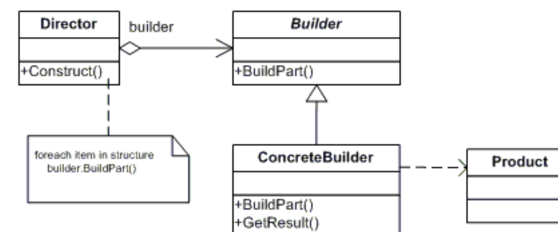
Související vzory

- Abstraktní továrna a Prototyp - konkurenční vzory, lze je však používat i společně – abstraktní továrna může ukládat sadu prototypů
- Skladba, Dekorátor

Stavitel (Builder)

- klasifikace: tvořivý, objekt
- smysl: Odděluje konstrukci složitého objektu od jeho reprezentace, lze pak vyrábět různé reprezentace
- motiv:
    - co má/nemá klient vidět při skládání složitých objektů pomocí jeho částí
    - konverze dokumentu z jednoho typu (třeba RTF) do několika jiných typů, počet konverzí je otevřený
    - parser čte RTF dokument, klient zvaný Director převezme výsledek od parseru a posílá požadavky, co se má stavět (jiný dokument) rozhraní Builder, Builder převezme požadavky a provede se samotná konstrukce nového dokumentu
    - Director neví, jaká je reprezentace objektu za rozhraním Builder
    - Po ukončení konstrukce objektu se požádá Builder o vydání poskládaného objektu

Struktura vzoru



Obrázek 89 Návrhový vzor Builder

– Builder – specifikuje rozhraní pro stavbu částí výsledného objektu
– ConcreteBuilder – implementuje rozhraní Builder, konstruuje konkrétní objekt daného Produktu, zavádí operaci pro návrat výsledku
– Director – zkonstruuje objekt použitím rozhraní Builder
– Product – reprezentuje skládaný konstruovaný objekt

Použití

- abstract faktory – operace CreateProduct() – návratová hodnota = nový produkt, u Builderu také žádost o nový produkt, ale dělá se „per partes“

- výsledné produkty nemusí patřit do stejného stromu dědičnosti
- nový složený objekt potřebuje nový builder
- možné žádosti klienta o mezivýsledky

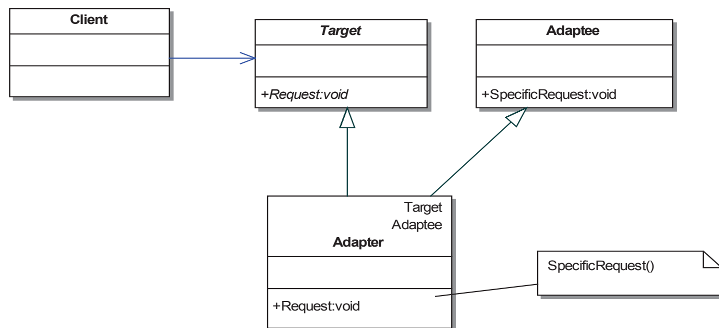
Související vzory

- Abstraktní továrna – také konstruuje komplexní objekty, ale najednou, vrací produkt okamžitě
- Skladba (Composite) – často výsledek konstrukce Stavitele

Adaptér (Adapter)

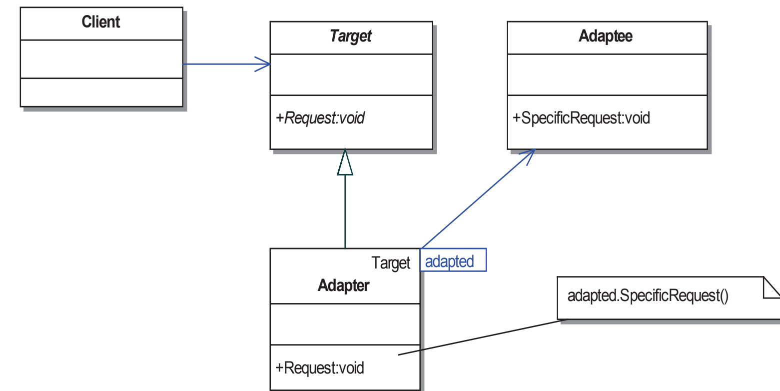
- klasifikace: strukturální, objekt nebo třída
- alias: Obal (Wrapper)
- smysl:
    - objekt – objekt jako spojka mezi klientem očekávajícím určité rozhraní a neznámým rozhraním, aneb klient může používat neznámé rozhraní
    - třída – zavádí třídu spojující neznámé a očekávané rozhraní, aneb klient může používat neznámé rozhraní
- motiv:
    - situace, kdy máme k dispozici třídu, která se nehodí do již existujících struktur, tj. např. podporuje rozhraní, které v existující struktuře vypadá jinak
    - jak zapojit cizí prvky do existující aplikace
    - adapter přizpůsobuje nekompatibilní rozhraní ke klientovi

Struktura vzoru třídní adaptér



Obrázek 90 Návrhový vzor adaptér - třídní verze

Struktura vzoru objektový adaptér



Obrázek 91 Obrázek 90 Návrhový vzor adaptér - objektová verze

Třída a objekt varianta, častěji užívaná údajně objekt varianta

- Target – definuje rozhraní používané Clientem
- Client – spolupracuje s objekty vyhovující cílovému rozhraní Target
- Adaptee – existující rozhraní, které potřebuje přizpůsobení
- Adapter – přizpůsobuje rozhraní Adaptee na rozhraní Target

- třídní varianta – používá vícenásobnou dědičnost, Adapter umí SpecificRequest(), protože je dědicem Adaptee
- objektová varianta – používá objektovou skladbu – zavolání požadavku Request() rozhraní Target se přesměruje na vložený objekt s rolí adapted

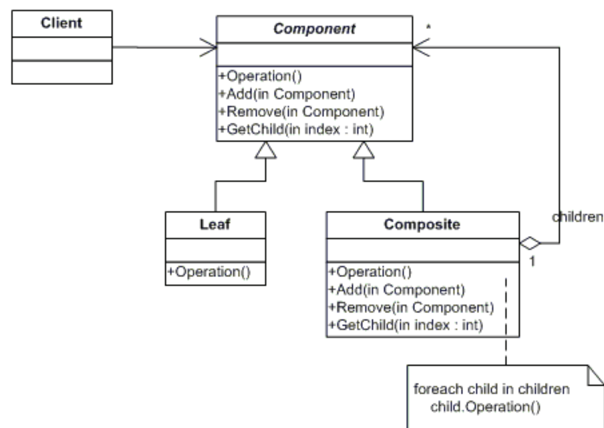
Použití

- objekt adaptér pracuje s objektem ze třídy Adaptee a s jeho dědici (stejné rozhraní), ale neumožňuje jednoduše přepsat jeho metody
- není určeno, co všechno má adaptér provádět, může operace pouze přejmenovat, může také přidat další funkcionalitu
- podoba se vzory Most (Bridge) a Zástupce (Proxy)

Skladba (Composite)

- klasifikace: strukturální, objekt
- smysl: skládá prvky do stromové struktury, k prvkům stromu přistupuje jednotným způsobem
- motiv:
    - obrazce se společnou abstraktní třídou Obrazec (metoda spoctiObsah()) viz PT
    - složený obrazec začleněn do stromu (člen rodiny obrazců, implementuje rozhraní Obrazec, rekurzivní volání) – prvky strom nejsou unifikované (listy a vnitřní uzly)
    - transparentní řešení – všechny možné operace do abstraktní třídy CObrazec (i operace pro správu dětí – pridej(), odeber(), zjistiDeti()) – unifikace prvků

Struktura vzoru



Obrázek 92 Návrhový vzor Composite

Použití

- maximalizace rozhraní Component – výhody a nevýhody (není třeba downcasting vs. nutnost ošetřit nesmyslnost operací u Leafu -celkově výhodnější)
- parent reference

Související vzory

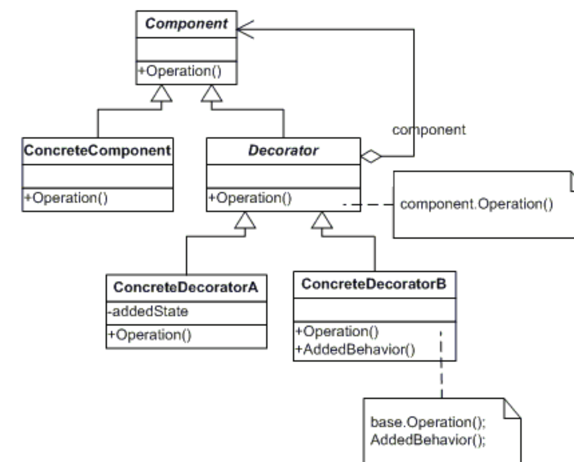
- Iterator pro průchod stromem
- Decorator někdy prvek Kompositu
- Chain of responsibility a odkaz na rodiče
- Visitor na prvku stromu

Dekoratér (Decorator)

- klasifikace: Strukturální, Objekt
- smysl: Dynamické přidání další funkcionality k objektu, zavedení flexibilní alternativy k dědění
- alias: Obal (Wrapper)

- motiv:
    - chceme přidat funkcionalitu k nějaké třídě – možné řešení je dědičnost – nevýhodou statická vazba a omezená možnost kombinací
    - jiné řešení – uzavření komponenty do jiného objektu, tzv. dekorátoru, využití skladby, výsledná struktura připomíná zřetězený seznam, nutné stejné rozhraní pro propojení mezi prvky reprezentované jedinou metodou operation()

Struktura vzoru



Obrázek 93 Návrhový vzor Decorator

- všechny prvky seznamu jsou chápány jako Component
- Component – rozhraní pro objekty, k nimž lze dynamicky přidávat funkcionalitu
- ConcreteComponent – definuje objekt, k němuž lze přidat funkcionalitu, koncový prvek seznamu
- Decorator – odkaz na objekt Component a definice rozhraní vyhovující rozhraní Component, mohou mít za sebou další jeden prvek, Component nebo další Decorator
- ConcreteDecorator – přidává funkcionalitu a stav

Důsledky

- větší tvárnost než statická dědičnost

- postupné přidávání funkcionality tj. není nutné v horní části hierarchie dědičnosti předvídat všechna možná chování
- Decorator a Component nejsou identické – dekorátor se chová jako průhledný obal
- mnoho malých objektů – obtížná laditelnost

Použití

- dynamické přidávání funkcionality jednotlivým objektům
- když je rozšíření pomocí tvorby podtříd nepraktické – masivní nárůst podtříd
- např. streamy v Javě

Související vzory

- podobný Adaptéru a Zástupci, protože za ním také kompatibilní rozhraní
- z matematického hlediska degenerovaný Composite
- podobný vzor Strategie

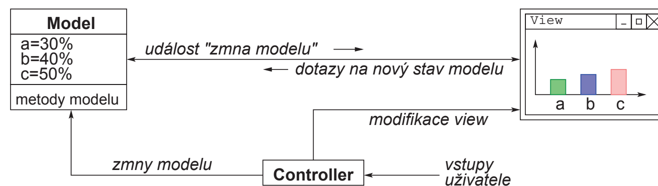
Vztah vydavatel-předplatitel

- angl. publisher-subscriber, subject-observer, broadcasting, model-view
- příklad architektury Model/View/Controller (MVC)

- Model/View/Controller

- funkčnost aplikace rozdělíme na 3 typy objektů, mezi objekty je vztah vydavatel-předplatitel
- "model" je objekt aplikace, zapouzdřuje data, která mají být zobrazena
- "view" (česky "náhled") je prezentace modelu na obrazovce
    - ve chvíli, kdy dojde ke změně stavu modelu, oznámí to model všem "view", které na něm závisejí; view zjistí od modelu nové hodnoty a znázorní je
    - způsobů prezentace může být více, je možné ho změnit např. z tabulky na koláčový graf, aniž by to ovlivnilo model nebo controller
- "controller" definuje způsob, jak uživatelské rozhraní reaguje na vstup
    - způsob reakce na uživatelský vstup je možné změnit, aniž by to ovlivnilo model nebo view; například místo klávesových zkratek použijeme výběr z menu

-



Obrázek 94 MVC

- příklad MVC = použití návrhového vzoru vydavatel-předplatitel
    - pokud je UI dobře navržené, view zná přímo model, ale obráceně to neplatí
    - pokud se model změní, vyvolá metodu "update()" všech view
    - view se pak zeptá modelu na konkrétní věci, které ho zajímají, a zpracuje je
- tento způsob komunikace se v OO systémech vyskytuje dostatečně často = návrhový vzor - nazveme "vydavatel-předplatitel"
    - vydavatel = instance, v níž může událost vzniknout, tj. v našem případě model
    - předplatitel = instance, která má zájem reagovat na určitou událost
        - předplatitel si u vydavatele zaregistruje metodu, která má být v důsledku události vyvolána, v případě MVC se nazývá "update()"
        - předplatitelů může být i více
- nastane-li událost, vydavatel vyvolá registrované metody předplatitelů
- vztah závislosti je možné implementovat různým způsobem, např. model bude mít v sobě instanční proměnnou ukazující na závislé view, nebo systém může obsahovat sdílený slovník závislostí apod.

- hlavní výhoda návrhového vzoru - model nezávisí na počtu a druhu view - model se může zabývat pouze svými objekty bez vztahu k uživatelskému rozhraní
- view se naopak zabývají znázorněním informací, k tomu musejí znát svůj model a způsob, jak z něj získat informace

Použití

- návrhový vzor vydavatel-předplatitel lze použít, kdykoli potřebujeme oznámit změnu stavu předem neznámému počtu objektů

Právník (lawyer)

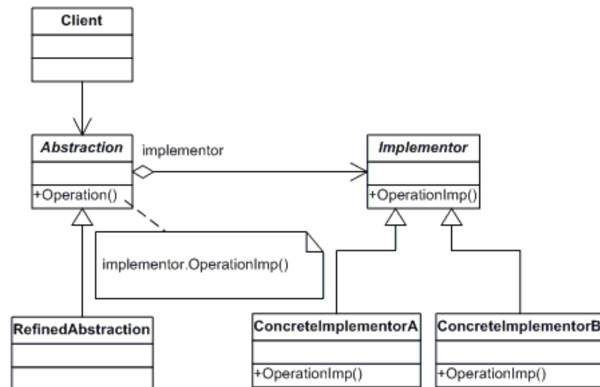
- občas dva objekty pracují společně, ale ani jednoho nechceme komplikovat tím, že by měl přímou znalost druhého (podobné Mediatoru)
- např. ikona a model, se kterým ikona souvisí
- chceme oddělit "model" od "view", není dobré dát modelu přímý odkaz na jeho ikonu
- ikona je tak jednoduchá, že od ní neočekáváme, že by znala objekt, který reprezentuje, tj. chceme je od sebe oddělit, ale na druhou stranu chceme model informovat o tom, co se děje s ikonou

- vytvoříme třetí objekt, nazývaný "právník", který zná ikonu i objekt, který ikona reprezentuje
- uživatelské rozhraní manipuluje s právníky místo s ikonami, resp. může prostřednictvím právníků komunikovat jak s ikonou, tak s modelem

Most (Bridge)

- klasifikace: strukturální, objekt
- smysl: oddělení abstrakce a implementace, lze je nezávisle měnit
- motiv:
    - řešení vztahu mezi abstrakcí a implementací děděním (abstraktní a implementační vrstva) -> mohou vznikat efekt „množení tříd“ (explosion of subclasses)
    - lepší řešení: zavedení abstraktní třídy implementace, ta se spojí (přemostí) s vrcholovou abstraktní třídou – vznikne objektová vazba
    - lze jednoduše přidat novou implementaci stejně jako rozšířit strom abstrakce nezávisle na sobě

Struktura vzoru



Obrázek 95 Návrhový vzor Bridge (Most)

- Abstraction - vrchol stromu abstrakce, udržuje referenci na objekt typu Implementor
- RefinedAbstraction – speciální abstrakce
- Implementor – rozhraní pro implementační třídy, může být rozdílné oproti rozhraní Abstraction
- ConcreteImplmentor – konkrétní implementace

Použití

- fyzické oddělení abstrakce a implementace přes rozhraní (u implementace použití komponentových technologií)
- nezávislost rozšíření stromů abstrakce a implementace
- řešení chyb souvisejících s problematickým použitím dědičnosti (množení tříd)

Související vzory

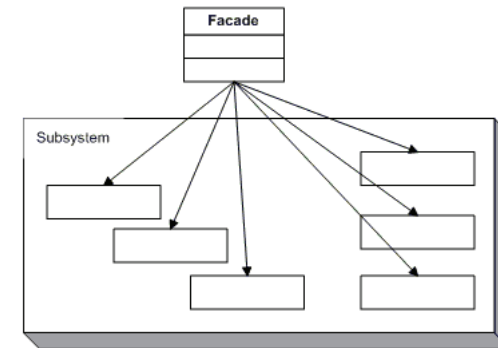
- volba třídy implementace – Abstraktní továrna, Prototyp

Fasáda (Facade)

- Klasifikace: strukturální, objekt
- Smysl: zjednodušuje přístup klienta k subsystému přes jeden nebo více rozhraní

- Motiv:
    - Užívaný zejména u komponentové technologie
    - Je lepší nabídnout klientovi jedno (popřípadě více) rozhraní místo celou množinu tříd – je to pro něj jednodušší
    - Hlavně pro vnitřně složité subsystémy – rozhraní = pohled na subsystém z pohledu případu užití (ostatní funkcionality se odstíní)

Struktura vzoru



Obrázek 96 Návrhový vzor Fasáda

Použití

- odstínění „nepotřebných“ tříd z pohledu případu užití
- sdílení instancí (sdílení objektů v paměti)
- pozor na příliš velké omezení přístupu k podsystému

Související vzory

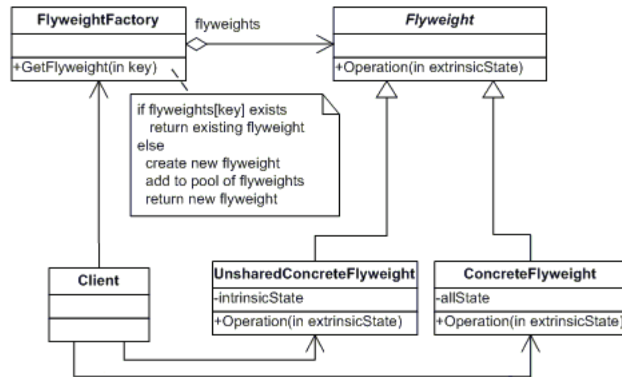
- vzory, které odstiňují volání tříd v konstrukci objektů -> jednoduchost fasády

Muší váha (Flyweight)

- Klasifikace: Strukturální, objekt
- Smysl: systémy s velkým počtem objektů – řešení sdílením
- Motiv:

- editor textu (fonty, velikost písma, pozice znaku) – text se skládá ze sloupců, sloupec z řádek, řádka z písmen obsahujících např. pozici a font -> příliš mnoho potenciálních objektů v paměti
- řešení: každý jeden znak dokumentu nebude objekt v paměti -> problém je nutné „odlehčit“
- u objektů, jejichž počet chceme redukovat, určíme stavy objektů a zavedeme je jako vstupní parametry požadovaných operací (u znaku např. operace Draw())
- různé hodnoty vstupních parametrů -> jiný kontext použití objektu v paměti (sdílený objekt), řídící proces a datový kontext
- u editoru počet objektů = počet možných znaků (číselník znaků), operace Draw() s parametry např. pozice a font (kontext), řídící objekt mapper (na obrázku je to přímo klient) – načte z dat id znaku, pozici a font, požádá číselník o vydání objektu s daným id, zavolá operaci Draw(pozice, font)
- znak je jako objekt použit tolikrát, kolikrát se v dokumentu vyskytuje
- kontext nemusí být předáván přímo z dat do číselníku, mohou existovat další meziúrovně (nesdílené FLYWEIGHT), u editoru např. instance řádku
- nesdílené FLYWEIGHTS – vznikají a zanikají jako objekty, nárůst objektů v aplikaci, ale stále podstatně méně než např. všechny znaky dokumentu

Struktura vzoru



Obrázek 97 Návrhový vzor Flyweight

- Flyweight – operace pro přijetí a zpracování kontextu
- ConcreteFlyweight – sdílené objekty číselníku, vnitřní stavy nezávislé na kontextu
- UnsharedConcreteFlyweight – nesdílené objekty číselníku přidávané (a odebírané) za běhu
- FlyweightFactory – číselník objektů, zodpovídá za správu objektů
- Client (Mapper) – zná referenci do číselníku a kontext

Použití

- není vhodné používat v případě malého počtu objektů, v případě, že není co sdílet (těžko se vyrábí vnější stavy), když se nám nehodí ztráta reference
- pro aplikace s velkým počtem objektů (ale pozor na režii kolem mapování – výpočet stavu, získání sdíleného objektu z číselníku)

Související vzory

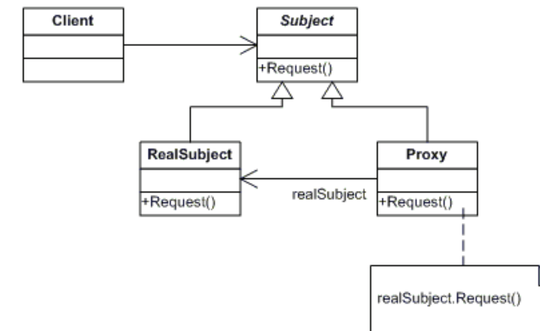
- Composite (listy jsou sdílené Flyweights, vnitřní uzly nesdílené Flyweights)

Zástupce (Proxy)

- Klasifikace: Strukturální, objekt
- Smysl: objekt zástupce, který kontroluje přístup k objektu
- Motiv:

- Přístup ke vzdálenému objektu na jiném stroji, klient chce s objektem pracovat jako by byl lokální
- Řešení: objekt zástupce
  - stejné rozhraní jako vzdálený objekt
  - uchovává technologii propojení mezi stroji

Struktura vzoru



Obrázek 98 Návrhový vzor Zástupce (Proxy)

Použití

- remote proxy (součást vývojových prostředků – Java, .NET)
- protection proxy – má zabudováno volání přístupových práv s předmětem práv „povolení operací objektu“, po vyhodnocení práva pokračuje/ nepokračuje operací původního objektu
- virtual proxy – nahradí původní objekt a provádí operace až ve chvíli, kdy jsou skutečně třeba (např. upload dat), když operace není třeba, je za ním prázdný objekt
- smart reference – řízený přístup k objektům (např. zamykání objektů)

Související vzory

- Adapter – dvě různá rozhraní
- Decorator – stejné rozhraní v řetězu za sebou, přidává funkcionalitu

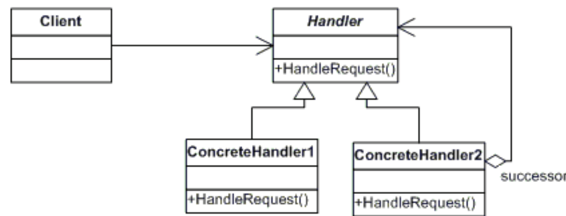
Řetěz odpovědnosti (Chain of responsibility)

- klasifikace: chování, objekt
- smysl: měnitelný řetěz objektů propojených stejným rozhraním k předání a zpracování požadavku

Motiv

- Zpracování Helpu, každý prvek GUI – požadavek na Help, pokud prvek help má, provede se, pokud nemá, pošle tento požadavek jinému objektu (rodič v GUI), může doputovat až na vrchol hierarchie

Struktura vzoru



Obrázek 99 Řetěz odpovědnosti

- Handler – společné rozhraní pro prvky stromu
- ConcreteHandler – objekty z řetězu, některé prvky ukončují řetěz vlastní funkcionalitou, některé posílají požadavek dál

Použití

- dynamický řetěz – konfigurace v runtime (pozor na zpracování na konci řetězu, zacyklení, vkládání zbytečných prvků na konec)

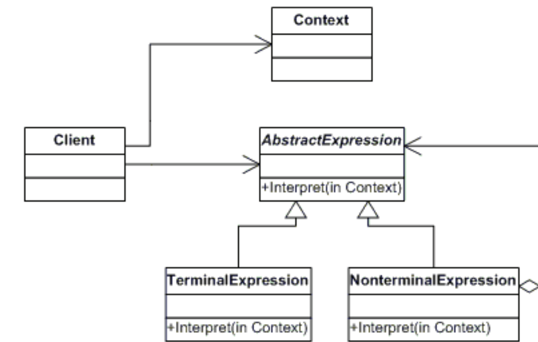
Související vzory

- vazba pro následovníka pro Řetěz odpovědnosti už je např. přirozeně u Compositu (reference na rodiče), viz GUI a Help

Interpreter

- Klasifikace: chování, třída
- Smysl: reprezentace pravidel gramatiky modelem tříd, zavedení odpovídajícího interpretu pravidel operacemi objektů těchto tříd
- Motiv
    - Definice jednoduché gramatiky na metaúrovni jako vztahy mezi třídami, její interpretace na konkrétních instancích (vztahy mezi instancemi)
    - Všechny třídy patří do téhož stromu dědičnosti

Struktura vzoru



Obrázek 100 Návrhový vzor Interpreter

Použití

- Není vhodný pro složité gramatiky, každé pravidlo zavádí třídu
- Vzor neukazuje, jak vybudovat strom objektů daného výrazu, ale jak jej „využít“

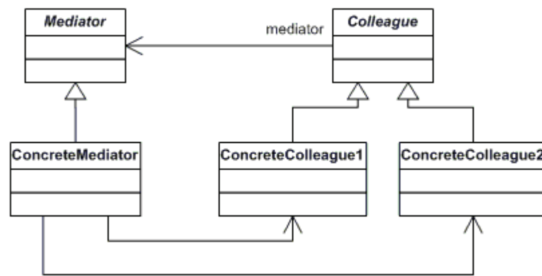
Související vzory

- Strom reg. výrazu jako Composite (průchod Iterátor)
- Non-terminální symboly – možnost optimalizace přes Flyweight

Mediator

- Klasifikace: chování, objekt
- Smysl: zavedený objekt je prostředníkem mezi jinými objekty (tyto pak nemusí mít mezi sebou přímé vazby), odděluje je a zprostředkovává mezi nimi komunikaci
- Alias: Controller
- Motiv:
    - spolupráce objektů v GUI, změna jednoho prvku -> nutnost změny dalších prvků, když je prvků mnoho -> nepřehlednost
    - řešení – objekt „vidící“ všechny komunikující objekty (přes objektové reference) - mediátor, zároveň všechny objekty „vidí“ mediátor, při změně prvku v GUI volá prvek specifickou operaci v Mediátoru, který pak v rámci operace volá ostatní prvky
    - často skryto v GUI - automatické zavedení vzoru, formulář zároveň mediátor – např Delphi, jiné řešení - listener s registrací – Java
    - složitější konstrukce – přímé volání operace mediátoru (ne přes vyvolání události) – každý prvek musí obsahovat instanci mediátoru a vidí jeho rozhraní – polymorfní operace ColleagueChanged(Colleague colleague), colleague jako vstupní parametr (musíme vědět, co a jak se změnilo) je obecný prvek

Struktura vzoru



Obrázek 101 Návrhový vzor Mediator

Použití

- systém událostí přehlednější než přímé volání rozhraní mediátoru
- formuláře jako mediátory (možnost přesunutí obslužného kódu do jiných formulářů (tzv. Controllerů – je to mediátor)

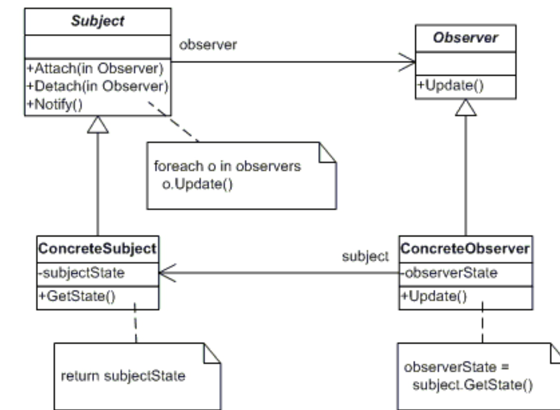
Související vzory

- Command – pvky při požadavku na spolupráci nevolají rozhraní mediátoru, ale MyCommand. execute(), každá třída Commandu obaluje jednu metodu Mediátoru (tříd Commandů = počet metod Mediatoru)
- Observer – událostní systém

Observer

- klasifikace: chování, objekt
- smysl: možnost sledování změny objektu; když objekt změní stav, ostatní objekty na to zareagují, avšak není přímá vazba od sledovaného objektu k těmto objektům
- Motiv
  - obdoba událostního systému (Java - listener)
  - první krok: objekt se spolu se svojí operací zaregistruje ke službě události jiného objektu, druhý krok: dojde k události, všechny registrované operace se zavolají
  - nutnost kompatibility volání zaregistrovaných operací -> vzor Observer

Struktura vzoru



Obrázek 102 Návrhový vzor Observer

- Abstraktní třída Subjekt – implementované operace pro přidání/odebrání objektu Observer a volání operace Update() všech zaregistrovaných Observerů = operace Notify()
- Observer – polymorfní operace Update()
- ConceretObserver – implementuje operaci Update(), udržuje referenci na ConcreteSubject, může si vyžádat stav ConcreteSubjectu
- ConcreteSubject - v rámci operace SetState() dojde zavolání operace Notify(), tj. k události a následovnému zavolání Update()

Použití

- dopředu není jasné, kdo má sledovat změnu sledovaného objektu
- konkrétní Subject neví, koho obsluhuje, konkrétní Observer je odstíněn od Subjectu
- pokud je jasné, kdo na změnu bude reagovat, a nebude se to měnit, vzor se nepoužívá, raději přímé volání daných objektů
- pozor na zacyklení událostí (více vrstev Observerů)
- při volání operace Notify() je třeba, aby konkrétní Subject byl v konzistentním stavu (Observer může načíst nekonzistentní stavy -> průšvih)
- vzor neřeší multithreading
- registrace observera a asociativní třída – dvojice Subject + Observer

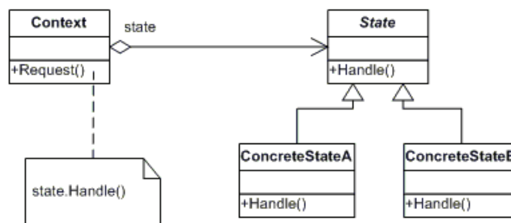
Související vzory

- Command – podobná funkcionalita a její obecné spouštění (nikoli sledování změny)

Stav (State)

- klasifikace: Chování, objekt
- smysl: změna stavu objektu prováděna výměnou objektu reprezentujícího stav
- motiv
  - vnitřní stav objektu
  - volání operací na základě vnitřního stavu objektu – implementace s if nebo switch může být nepřehledná
  - Řešení: abstraktní třída State, polymorfní operace zpracování stavu, dědicovové třídy State() (počet stavů = počet dědiců), přepnutí stavu = výměna objektu typu State()

Struktura vzoru



Obrázek 103 Návrhový vzor Stav (State)

- Context – rozhraní pro klienta, zná stavy, přesměruje požadavek na chování v daném stavu na objekt state
- State – abstraktní třída pro chování ve stavu
- ConcreteState – implementace chování v konkrétním stavu

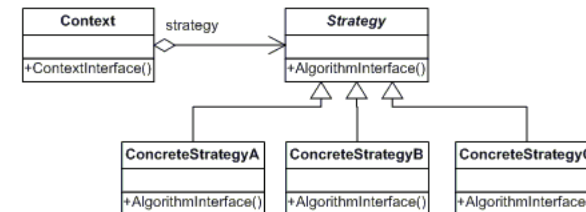
Použití

- přehlednost a flexibilita – nový stav= nový potomek třídy State
- menší riziko chyb a nesmyslných mezistavů
- přechody mezi stavy, nejlépe umí dědicové přímo v metodách Handle(), nutno mít referenci (zpětnou) na rozhraní objektu typu Context
- stavové atributy společné pro všechny stavy je lépe umístit do objektu Contextu (State má na Context referenci) nebo úplně mimo + svázat přes asociaci, nikoli do abstraktní třídy State

Strategie (Strategy)

- Klasifikace: Chování, objekt
- Smysl: Definuje množinu algoritmů, které lze za běhu vyměnit
- Motiv:
  - máme k dispozici více algoritmů vyhledávání dat, požadavek na použitý algoritmus dán obsluhou za běhu programu, neflexibilní řešení: přepínač
  - jiné řešení: rozhraní Strategy s operací např. SearchAlgorithm(), dědicové ji přepisují, výběr algoritmu na základě výběru instance třídy Strategy

Struktura vzoru



- Strategy – rozhraní pro všechny operace (podporované algoritmy)
- ConcreteStrategy – implementace konkrétních operací
- Context – udržuje referenci na objekt Strategy, konfigurace objektem ConcreteStrategy, může definovat rozhraní, aby mohl objekt Strategy přistupovat k datům Contextu

Související vzory

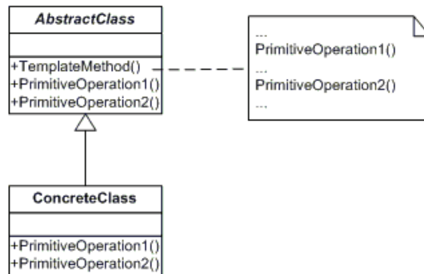
- Podobný vzoru State
- Strategy – sdílený algoritmus -> Singleton nebo Flyweights

Šablona (Template Method)

- klasifikace: chování, třída
- smysl: existence scénáře na abstraktní úrovni, obsahuje polymorfní metody, potomci je přepisují a volají scénář
- motiv
  - dokument u factory method – na úrovni předka operace (scénář) openDocument(), obsahuje polymorfní operaci createDoc(), potomci scénář využijí, operace přepíší

```
public void openDocument() {
        myDoc = createDoc();
        docs.add(mydoc);
        myDoc.open();
}
```

Struktura vzoru



Obrázek 104 Návrhový vzor Template method

- AbstractClass
  - obsahuje TemplateMethod() – scénář
  - obsahuje primitivní operace (nemusí být nutně abstraktní – implicitní chování)
  - samotná třída nemusí tedy být abstraktní
- ConcreteClass – přepisuje polymorfní operace

Použití

- přesně definované postupy – předefinování operací a zavolání TemplateMethod()

Související vzor

- Factory method

Návštěvník (Visitor)

- klasifikace: chování, objekt
- smysl: zavádí operaci, která účinkuje na prvky objektové struktury, přitom se nemění kód ve třídách těchto prvků
- motiv
  - strom obrazců, obrazců je hodně, potřebujeme přidat operaci spoctiObvod(), nechceme se nám ale zasahovat do kódu třídy každého obrazce
  - řešení: nová operace (pseudooperace) bude využívat již existující operace obrazce (obrazce musí umět operace jako getPolomer(), getDelkaStranyA() apod.), aneb z návratových hodnot zavedených operací musíme umět získat vše potřebné pro nově zavedenou operaci (pseudooperaci)

- celkový postup

- každý prvek stromu obrazců bude mít polymorfní operaci pro přijetí Visitora např. accept (CVisitor visitor), tj. vlastní implementaci této operace

- *každá pseudooperace bude reprezentována jednou třídou Visitora* (Visitoři tvoří strom dědičnosti na vrcholu s abstraktní třídou Visitor), tj. pro obrazce jedna třída Visitora např. VisitorObvod a abstraktní třída Visitor (typ vstupního parametru operace accept)-> *tj. obrazec umí přijmout libovolného Visitora*

- existence dvojitého polymorfismu: každý Visitor má přesně tolik operací, kolik jednotlivých obrazců ve stromu (operace jsou polymorfní, každý Visitor si je implementuje) může navštívit

- šablona Visit<Konkrétní obrazec> se vstupním parametrem = členem, kterého hodlá Visitor navštívit (Visitor si tak může vyžádat vše potřebné), tj. např. operace
  visitKruh (Ckruh kruh)
  visitObdelnik (CObdelnik obdelnik)
  visitTrojuhelnik (CTrojuhelnik trojuhelnik)

- ve výsledku: *přijetí Visitora daným konkrétním obrazcem se převede na zavolání visitorské návštěvy tohoto konkrétního obrazce*

- Např. kruh implementuje

```
public void accept (Visitor visitor) {
    visitor.visitKruh(this);
}
```

- obdélník implementuje

```
public void accept (Visitor visitor) {
    visitor.visitObdelnik(this);
}
```

- složený obrazec implementuje

```
public void accept (Visitor visitor) {
    for each item in obrazce item.accept(visitor);
}
```

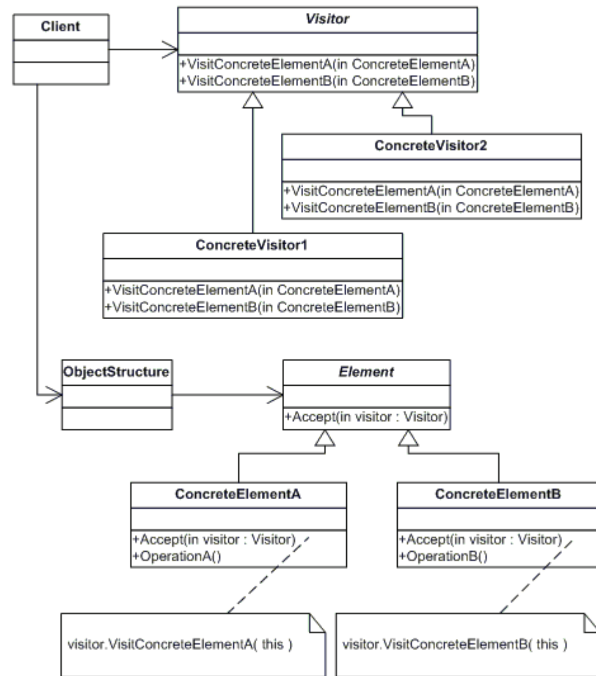
Scénář práce s Visitorem

- vybere se pseudooperace (tj konkrétní třída Visitora)
- zavolá se operace obrazce accept(visitor), vstupním parametrem objekt vybrané třídy Visitora
- vybraný obrazec vrací řízení visitorovi např. visitor.visitKruh(this)
- uvnitř visitora máme k dispozici obrazec, provedeme nad ním pseudooperaci, kterou daný visitor umí (tj. např. spočtení obvodu kruhu)
- výsledek převezme visitor do sebe a může jej na požádání vydat klientovi, který spouští accept()
- Visitor může postupně projít celou strukturou a posbírat všechny údaje
- Je možné flexibilně vyrobit dalšího visitora

Použití visitora např.

VisitorObvod myVisitor = new VisitorObvod(); /v_obvod = 0;
myObrazec.accept(myVisitor);
obvod = myVisitor.getObvod();

Struktura vzoru



Obrázek 105 Návrhový vzor Visitor

Použití

- flexibilita operací
- sběr údajů

Související vzory

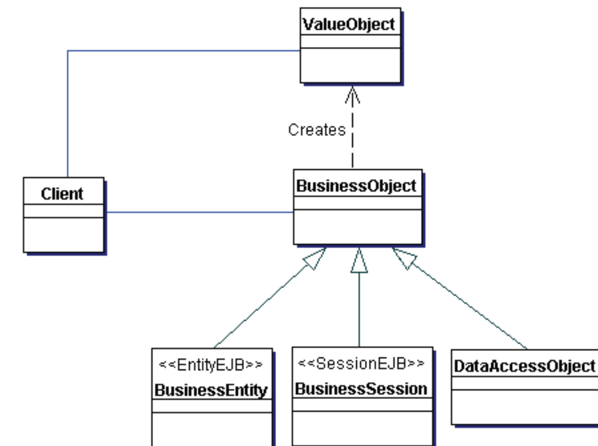
- zpracování Compositu
- projití struktury s využitím Iterátoru

Transfer Object (J2EE pattern)

- kontext: klient si potřebuje vyměňovat data s komponentou
- motiv:
    - komponenty na straně serveru poskytují data, tj. dávají k dispozici přístupové metody, každé volání metody klientem je potenciálně vzdálené, pokud je volání mnoho a klienti žádají jednoduché atributy, klesá výkonnost celé aplikace
    - řešení: aplikační data jsou zapouzdřena do Transfer objectu, manipulace s tímto objektem pak probíhá přes jedinou metodu, klient žádající o data obdrží celý Transfer object poté, co je tento objekt zkonstruován

    - využití Transfer objectu je spojeno s mnoha strategiemi

Struktura vzoru



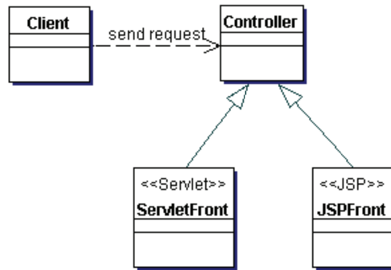
Obrázek 106 Návrhový vzor Transfer Object

Front Controller (J2EE pattern)

- kontext – mechanismus řízení požadavků na úrovni prezentační vrstvy, řízení a koordinace zpracování uživatelských požadavků, tento mechanismus může být centralizovaný/decentralizovaný
- motiv
    - pokud uživatel může přistupovat k prezentační vrstvě bez „průchodu“ centralizovaným mechanismem řízení požadavků, je často nutné duplikovat kód jednotlivých view, navíc dochází ke smísení obsahu a řízení daného view

- řešení: použití Controlleru jako prvotního bodu kontaktu s požadavkem, controller spravuje další zpracování požadavku (např. přihlášení, výběr korektního view apod.)
- pro implementaci Front Controlleru existuje několik strategií

Struktura vzoru

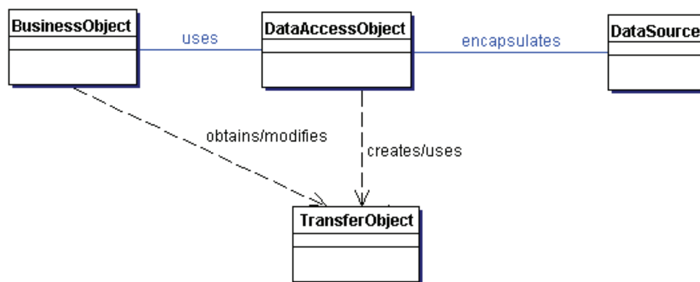


Obrázek 107 Návrhový vzor Front Controller

Data Access Object (DAO) (J2EE pattern)

- kontext – přístup k úložišti dat závisí na typu daného úložiště (relační databáze, soubor,…) a jeho implementaci
- motiv
    - potřebujeme získávat/ukládat data z/do persistentních datových úložišť, tyto se mohou měnit, mají různá API, klient chce však používat jednotné rozhraní
    - řešení: Data Access object zapouzdřující přístup k datovému zdroji, DAO zařizuje veškerou komunikaci s datovým zdrojem ohledně ukládání/získávání dat (je to takový Adaptér)

Struktura vzoru



Obrázek 108 Návrhový vzor Data Access Object

Stabilita a dynamika firmy, podpora a aktivita lidí

- obrázky v následující kapitole jsou převzaty z publikací J.Plamínka
- třetí a čtvrté patro pyramidy vitality
- podpora (akceptace) lidí nutná pro efektivní fungování cyklického modelu řízení
- spontánní aktivita lidí nutná pro dynamiku firmy

Motivace

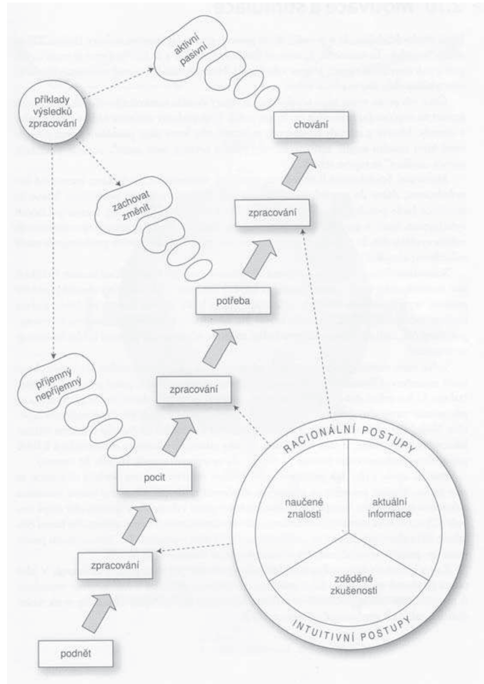
- třetí patro pyramidy kultury
- intuitivní monitorování a vyhodnocování stavu, ve kterém se nacházíme – aktuální pocit – vyvolání potřeby (příjemný stav zachovat, nepříjemný změnit)
- potřeba = příčina chování = motiv
- motivem chování je naplňování potřeb (nejzákladnější potřeba = příjemný pocit) viz Maslowova pyramida potřeb
- prožívání pocitů – pozadí učení
- stav naplnění potřeb + příjemný pocit = spokojenost (trvá jen omezenou dobu, buď tento stav ukončí vnější podmínky, nebo odezní sám od sebe – nutné, abychom se nepřestali vyvíjet)
- aktuální pocit souvisí s rovnováhou systému člověk, chování souvisí se stabilitou
- podněty, pocity, potřeby, chování – kauzální řetězec, do kterého vklíněny způsoby zpracování (způsob reakce člověka na podnět)
- ovlivňování podnětů (např. zadávání úkolů) a způsobů, jak je lidé zpracují (přijmou) = motivace (působí „zevnitř“) a stimulace (působí „zvnějšku“)

- úkol: najít impuls, který dokáže konkrétního člověka inspirovat k práci na naplňování firemních cílů a myšlenek

- motivace
    - činnost, kterou vyžadujeme, dáváme do souvislosti s existujícími vnitřními potřebami člověka
    - působí i bez našeho vlivu tak dlouho, dokud je v souladu s aktuálními motivy člověka
    - vyžaduje, abychom uměli odhadnout aktuální motivy lidí
    - podstatně složitější nástroj než stimulace
- stimulace
    - účinná tak dlouho, dokud působí jako podnět
    - jakmile přestaneme investovat prostředky (čas, úsilí, prostředky), žádoucí lidská činnost se zastaví
- při budování spodních pater pyramidy vitality vystačíme ze stimulací, při budování horních pater se zvyšuje podíl motivační složky (v dynamické fázi se očekává chování zčásti založené na „sebemotivaci“ – vhodné podněty dodává i systém firemních myšlenek)
- pozor na nebezpečí, kdy motivace a stimulace působí proti sobě

Principy motivace

- máme konkrétní úkol a konkrétního člověka – obvykle „do sebe vzájemně nezapadají“ -> lepší strategie: přizpůsobit úkol člověku (zadat jej formou, která mu vyhovuje, uzpůsobit podmínky pro plnění úkolu jeho požadavkům)
- Zlaté pravidlo motivace: Nepřitesávejte lidi k obrazu jejich úkolů, ale snažte se spíše přizpůsobit úkoly lidem a jejich aktuálním motivům

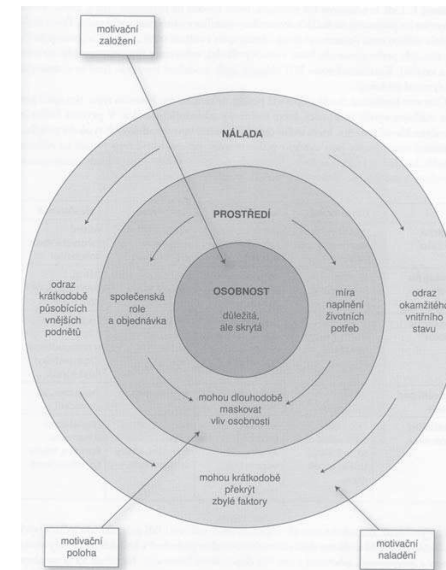


Obrázek 109 Cesta od podnětu k chování

- Manažerský úkol: ze dvou vstupů, které si nemusejí vyhovovat (člověk a úkol) provést transformaci do dvou výstupů (požadovaný výsledek a lidská spokojenost)
- Lidská spokojenost má komplexní charakter (nejméně tři vrstvy: věcná – týká se výsledku, procesní - týká se spravedlnosti procesu, osobní – souvisí s důstojností rolí, kterou člověk hraje) – je třeba nabídnout spokojenost alespoň ve dvou oblastech

Vrstvy motivace

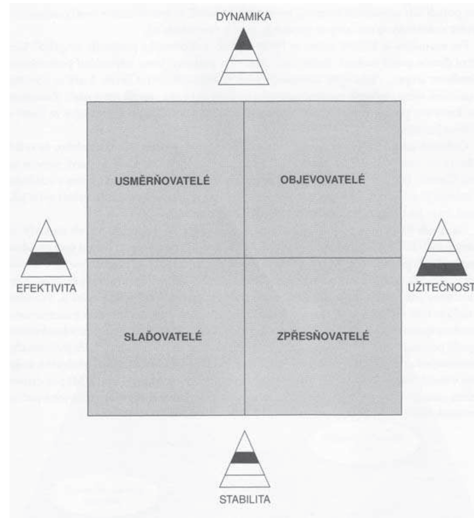
- Motivační založení
  - prakticky neměnná charakteristika lidské osobnosti (během života se mění pomalu a neochotně)
  - můžeme poznat, pochopit, použít
  - může být maskováno a překryto vnějšími podmínkami, „společenskou objednávkou“, rolemi v životě (spolupracovník, rodinný příslušník, soused)
  - pokud je v rozporu se „společenskou objednávkou“, projevuje se zejména v krizových situacích nebo při spontánních reakcích
- motivační poloha
  - souvisí s životními a pracovními podmínkami; prostřednictvím podmínek ji můžeme ovlivňovat
  - za určitých okolností může dominovat a potlačit vliv podnětů souvisejících s motivačním založením
  - dominuje, když okolí naléhavě vyžaduje po člověku specifické chování (dané např. sociální rolí)
- motivační naladění
  - okamžitý stav vnitřních pohnutek
  - rychle přichází a odeznívá, nemá smysl jej významněji řešit



Obrázek 110 Motivační pole

Posuzování motivačního založení

- při posuzování inspirace základními vitálními znaky
- škály dynamika-stabilita – volba mezi rizikem a jistotou
- škála užitečnost-efektivita – volba mezi „účelem“, a „prostředky“, preference cílů (věci, výsledky, úkoly) vs. cest (procesy, metody, lidé a vztahy)
- typy motivačního založení (Obrázek 111 Typy motivačního založení a hlavní rysy základních motivačních typů lidí



Obrázek 111 Typy motivačního založení

Tabulka 7 Hlavní rysy motivačních typů lidí

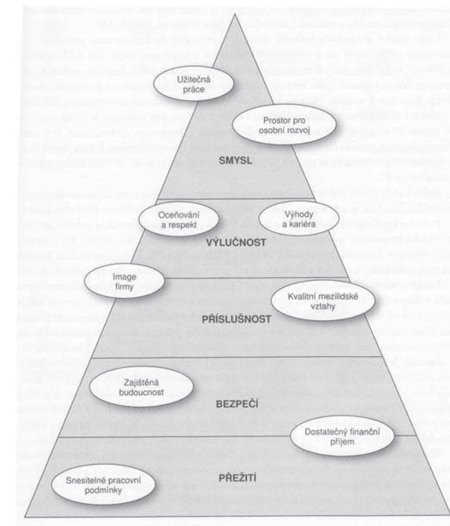
	DYNAMIKA		
EFEKTIVITA	– sebeprosazování – vůle vést lidi – vůle koordinovat činnosti – testování schopností druhých lidí – snaha být středem zájmu – poměřování a třídění lidí	– samostatnost jednání – aktivita při řešení problémů – netrpělivost – hlad po informacích – nezávislost a špatná zvladatelnost – racionální inteligence	UŽITEČNOST
	– vůle naslouchat a chápat – empatie a sociální inteligence – ochota uznat odlišný názor – potřeba sociálního kontaktu – preference společné práce – tolerance odlišností	– vyžadování pořádku – skrývání emocí – přesné vyjadřování – sklony analyzovat – tendence nepodceňovat detaily – sklony dotahovat činnosti	
	STABILITA		

Tabulka 8 Potřeby, reakce, jednání pod zátěží - motivační typy

	Objevovatel	Usměrňovatel	Slaďovatel	Zpřesňovatel
Typická potřeba	Vnitřní sebepro-sazení: překoná-vání výzev	Vnější sebepro-sazení: ovlivňo-vání lidí	Vnější zakotvení: příznivé prostředí	Vnitřní zakotvení: vlastní dokonalost
Reakce na pochvalu	Já vím. Samozřejmě, že to vyšlo.	Nebylo to lehké. Ukážu ti, jak jsem to udělal.	Pochval i ostatní. Jsi taky dobrý.	Děkuji. Udělal jsem, co jsem mohl.
Reakce na kritiku	Já vím. S tím už nic nenaděláš.	(Bagatelizace) Takhle otázka nestojí. Kdo vlastně jsi, že mi…	(Přijímá) Chápu tě. Asi jsem tě zklamal.	Spravedlivou: (Lítost) Nespravedlivou: (Diplomaticky) Když myslíš.
Jednání pod zátěží	Oživení Vyšší výkon	Často převádí zátěž na jiné	Příliš nereaguje	Velký stres, až zhroucení
Motivující formulace	Těžký úkol. Ještě nikdo to nedokázal. Udělej to, jak chceš.	Jsme na tobě závislí. Jsi výborný organizátor. Nechci ti do toho mluvit.	Budeš součástí týmu. Pomůžeš, kdyby se nepodhodli.	Napsal jsem ti instrukce. Poradím, kdyby to nebylo jasné.

Motivační poloha

- povahy a důsledky vnějších vlivů působících na člověka
- Maslowova pyramida – člověk je citlivý na podněty na svém patře pyramidy (podněty na úrovni jiných pater si často překládá do svého patra) – působíme na něj příslušnými stimuly



Obrázek 112 Pyramida stimulů

Typologie osobnosti

- Lidská rozmanitost – není úplně nahodilá
- „škatulkování“ lidských druhů
- Hippokrates, Galen – sangvinik, cholerik, flegmatik, melancholik
- Kategoriální pojetí (neexistence smíšených typů)

- Člověk je v jistém směru
- Jako ostatní lidé
- Jako někteří lidé
- Jako nikdo jiný

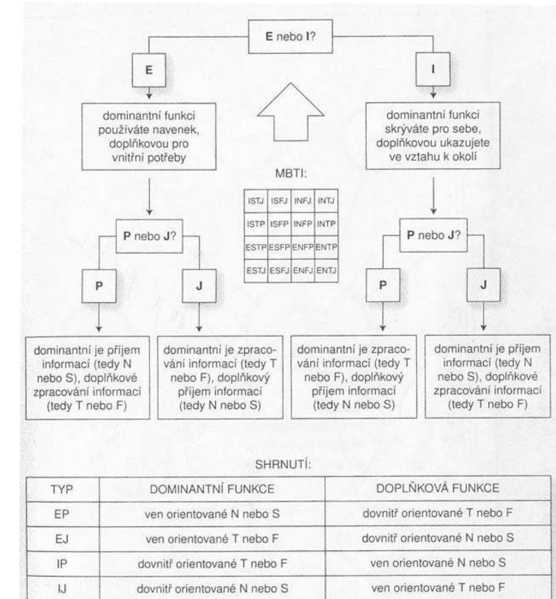
- Carl Gustav Jung
- každý vidí realitu z vlastního pohledu
- stejný soubor instinktů, funkcí, ale
- psychologické typy podle preferencí – něčemu dáváme přednost
- kombinace preferencí – osobnost

- oblíbené funkce je snadné používat, protože příslušná část mozku je k jejich používání vyvinuta – návyk, zkušenost

- Jsme motivováni apely na naše nejrozvinutější centra

- Lidé se také liší ve způsobu zpracování informací -> MBTI (Myers-Briggs Type Indicator)

- někde nekriticky populární, někde téměř neznámá
- zjištění nijak nehodnotí, neexistují dobré a špatné výsledky
- nástroj pro zjišťování předpokladů, neříká, že to tak určitě bude
- nehodnotí schopnosti a dovednosti, ale preference a typy
- žádná z preferencí není nadřazena jiné
- není černobílým pohledem
- čtyři dvojice parametrů - poměr
    - odkud čerpáme podněty (I - introverze vs. E -extroverze)
    - jak podněty přijímáme (S – smysly vs. N – Intuice)
    - jak podněty zpracováváme (F- citem vs. T- rozumem)
    - jak se rozhodujeme (P-vnímající vs. J- usuzující)

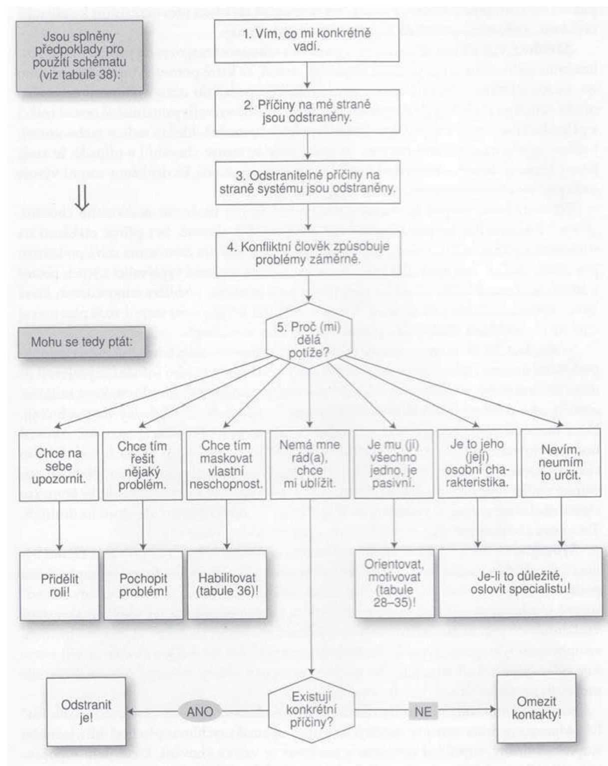


TYP	DOMINANTNÍ FUNKCE	DOPLŇKOVÁ FUNKCE
EP	ven orientované N nebo S	dovnitř orientované T nebo F
EJ	ven orientované T nebo F	dovnitř orientované N nebo S
IP	dovnitř orientované T nebo F	ven orientované N nebo S
IJ	dovnitř orientované N nebo S	ven orientované T nebo F

Obrázek 113 Klíč k určování významu funkcí a MBTI

Konfliktní chování

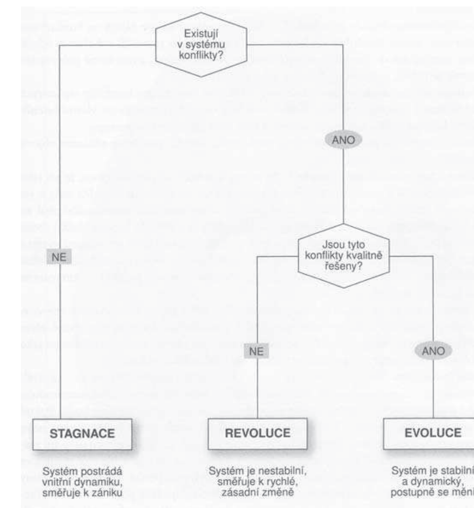
- „standardní konfliktní lidé“ a nestandardní situace
- Neměňte lidi, odstraňujte potíže (např. „stojí mne to příliš času“, „ostatní se nedostanou ke slovu“, **ne** např. „neměl by myslet tolik na sebe“)
- Příčiny hledejte nejdříve u sebe (např. „nemám podobné problémy i s jinými lidmi“) a v systému (např. úkolujete člověka, kterého nemůžete odměňovat)
- Ujistěte se, že konfliktní člověk ví o problémech, které způsobuje
- Hledejte příčiny konfliktního chování a snažte se nabídnout něco, co je může odstranit
- Pokud si nevíte rady a problémy jsou závažné, obraťte se na specialistu



Obrázek 114 Odstraňování příčin konfliktního jednání

Konflikty ve skupině

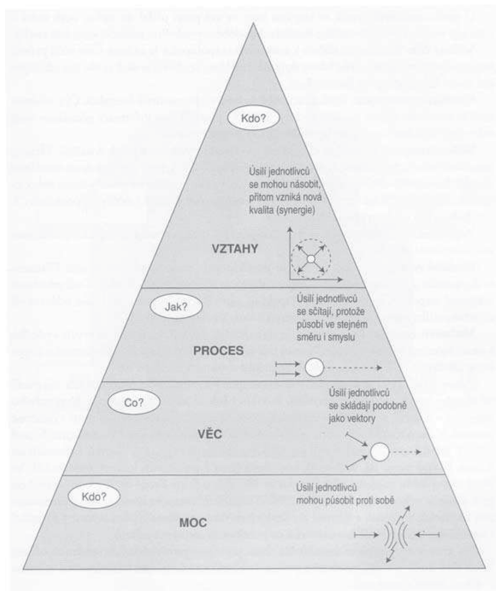
- Vnášejí do systému nerovnováhu – dynamizují jej, palivo do motoru vývoje – nemá smysl je přehlížet ani je násilně odstraňovat
- Několik typů konfliktů – pro vývoj skupiny důležitý konflikt mezi zájmy jednotlivých členů skupiny a zájmy celku (úspěšné zvládnutí = efektivní tým)
- Vyplatí se konflikty ve skupině řešit včas (dokud je napadena rovnováha, nikoliv již stabilita)
- - metodický důsledek můžeme (i tvrdě) prosazovat své nebo skupinové věcné zájmy, ale musíme přitom chránit mechanismy a procesy, jimiž je možné k dohodě dojít (můžete se hádat, ale neměli byste spolu přestat mluvit) – neúcta k rovnováze, hluboký respekt ke stabilitě
- Věcné jádro konfliktu – problém, vztahový (emociální) obal – spor
- Depersonifikace sporu – postup k racionálnímu jádru (které nemusí existovat)
- Mediace konfliktů – neutrální zprostředkovatel
- Princip racionalizace – konflikty řešte na úrovni zájmů
- Princip - snažte se objevit co nejvíce zájmů (čím více zájmů, tím větší šance, že se zájmy na obou stranách konfliktu budou lišit a vyjednávací prostor, a tím i šance na spolupráci se zvětší)



Obrázek 115 Konflikty v systému

Vznik a vývoj týmu

- Dvě hlavní překážky – souvisí s dominantními konflikty ve skupině
    - Napětí mezi zájmem jednotlivce a zájmem celku
    - Vyrovnání se s rozdíly mezi jednotlivci (vnitřní diverzita skupiny)
- Úspěšné zvládnutí těchto překážek – specifické vlastnosti každého člena skupiny plně využity v kontextu specifických vlastností ostatních členů skupiny
- Spontánně vznikající tým - postupně mocenské rozdělení vlivu ve skupině, věcné aspekty, procesní aspekty, vztahové aspekty
- „řízený" vývoj týmu – stejná cesta, rychlejší zrání skupiny v tým



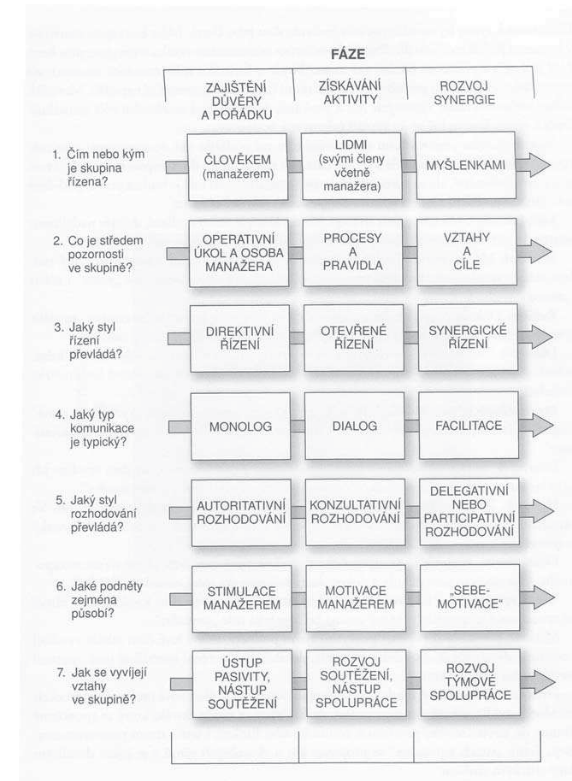
Obrázek 116 Spontánní zrání skupiny

Vlastnosti týmu

- Sdílené cíle (klíč k nastartování spolupráce)
- Kvalitní komunikace (efektivní, levná, přesná,…)
- Sdílené cesty (metody práce založené na sdílené soustavě pravidel)
- Rozdělení rolí (jednotlivci si hledají specifické a do určité míry autonomní role)
- Kvalitní vztahy (odborný i lidský respekt, ochota sdílet mimopracovní problémy)
- Možnosti rozvoje – dynamika týmu, osobní rozvoj jeho členů, prožití společného úspěchu a společného poučení z neúspěchu

Rozvoj týmu

- Péče o diverzitu (v lidech a rolích) – zvládnutí rozdílů mezi členy týmu, efektivní využívání těchto rozdílů, učení se z vlastních výsledků, vzájemná pomoc – její forma není rozhodující, smyslem není řešit problém, ale poskytnout jiné názory, podněty
- Péče o sdílení (cílů a cest) – analogie k vedení z řízení firmy, sdílení vize, vlastní cíle a cesty, funkční zpětné vazby… nejlepším prostředkem pro hledání jsou společné diskuse (neformální diskuse - zkušenosti, formální diskuse o konkrétních problémech – postupný posun: schopnost prezentovat, schopnost naslouchat a chápat, schopnost pomáhat – nejlepší nástroje sebeřízení týmu)
- Proces přeměny skupiny v tým

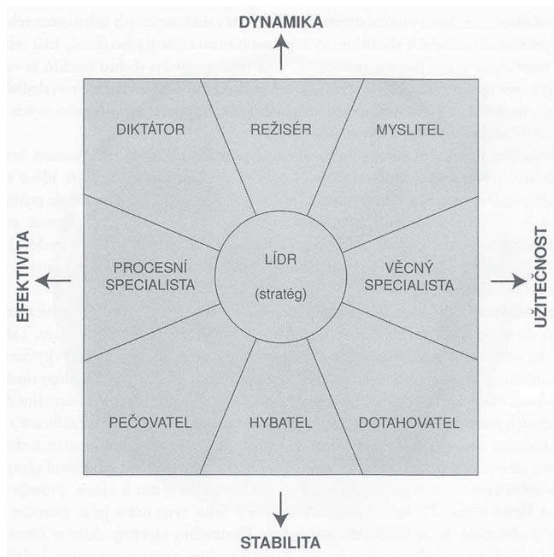


Obrázek 117 Proces přeměny skupiny na tým

Rozdělení rolí v týmu

- Lidé si v týmu přirozeně hledají svoje místa
- Nejde o formální role (manažer, sekretářka), ale o spontánní vykonávaní potřebných funkcí týmu (např. rozdělení úkolů, dotahování věcí do konce)
  - Lídr (stratég) – komplexní myšlení, dokáže nadchnout ostatní, univerzální, všestranný
  - Myslitel – racionální inteligence, kreativní, hravý, nezávislý
  - Režisér – uvedení myšlenek, strategií do praxe, určení taktiky, rozdělení operativních úkolů
  - Diktátor – problémový typ, použitelný zejména v krizi
  - Procesní specialista – vyniká při selhání standardních procesů, tvůrce a objevitel originálních metod, postupů a zdrojů
  - Pečovatel – stará se o pohodu v týmu, nemá velký tah na branku, vyniká při zvládání komunikačních a vztahových poruch

- Hybatel – ve standardních podmínkách motor výkonu týmu, uskuteční to, co připraví myslitelé, lídři a režiséři
    - Dotahovatel – puntičkář se systematickým myšlením, stará se o to, aby se na nic nezapomnělo
    - Věcný specialista – zná a zvládá svůj obor, v rámci své specializace spolehlivý
- Myslitel, lídr a režisér poskytují týmu podněty
- Role v levé části diagramu vynikají v nestandardních situacích
- Hybatelé, dotahovatelé a věcní specialisté zjišťují pravidelný chod týmu
- Strach z prázdna – u skutečného týmu se vždy najde člověk, který se spontánně přesune do neobsazeného prostoru a vezme na sebe funkce, které s tímto prostorem souvisejí



Obrázek 118 Týmové role

Potenciál vs. výkon

- potenciál – lidské zdroje
- výkon – skutečná práce
- kompetence (způsobilost lidí)
- harmonie potenciálu a výkonu
- rozdíl mezi potenciálem výkonem může být obrovský
- podceňování/přeceňování výkonu/potenciálu
- nevyužívaný potenciál leží ladem (malý výkon) – může se i snižovat
- soustředění na výkon – není péče o potenciál – budoucí problémy
- správa lidských zdrojů = udržitelný rozvoj

Time management

- Organizace času
- 1. generace – výsledek přehled úkolů (př. Seznam nákupu) – co máme dělat
- 2. generace – seznam úkolů v čase – co a kdy máme dělat
- 3. generace – přiřazování priorit činnostem – snaha zachytit rozsáhlejší projekty, týmovou práci, pomůcky – co, kdy a jak máme dělat
- 4. generace
    - prosperita začíná tím, že se cítíte dobře
    - vnitřní orientace na prožívání, vnější na výsledky
    - jakýkoli úspěch je podmíněn osobní kvalitou
    - pozornost a stálá jemná změna, „Ponechejte věcem volný průběh a budou se vyvíjet špatně“
    - jemné a trvalé posilování všech aspektů života

Stres

- Symbol (někdy snad až požadavek) současnosti
- Stres = zátěž/odolnost
    - ovlivňování zátěže
    - ovlivňování odolnosti
- myšlení – ustupuje do pozadí, pokud mozkový kmen vyhodnotí situaci jako ohrožující (automatická stresová reakce) – ohrožující je každá nová situace
- hodnocení situace – limbický systém (pomezí kmene a kůry) – rozhoduje o povaze stresu
- distres – nové a neznáme situace + známé, reálně nebezpečné situace
- eustres – známé pozitivní podněty
- rozhodující motiv lidské činnosti – příjemné pocity a vyhýbání se nepříjemným pocitům (0. patro Maslowovy pyramidy) – a také podstata učení

Vedení, řízení

- Vedení – přemýšlení, výběr strategie, musí být poměrně jednoduchá a přímočará, aby se dala reálně praktikovat
- Řízení – rozhodování, delegování, rutinní a tvořivá práce, jak u vybraného skutečně zůstat, jak sladit s protichůdnými nároky ostatních, nepředvídatelnými problémy
- P. Drucker: Výsledky jsou děláním správných věcí, nikoli děláním věcí správně
- Paretův princip – 80% vs. 20%
- J.A. Komenský: Všeliké kvaltování toliko pro hovada dobré jest.
- Lepší je nepřítelem dobrého – dokonalost je drahá a často k neunesení

Literatura:

[1] Jiří Plamínek: Sebeřízení: praktický atlas managementu cílů, času a stresu, Grada, Praha, 2004.
[2] Jiří Plamínek: Vedení lidí, týmů a firem: praktický atlas managementu, Grada, Praha, 2005.
[3] Michal Čakrt: Typologie pro manažery: kdo jsem já, kdo jste vy?, Management Press, Praha, 2000.
[4] Petr Pacovský: Člověk a čas, Grada, Praha, 2006.

Testování

Verifikace a validace

- verifikace = ověření, zda produkt dané fáze vývoje SW odpovídá konceptuálnímu modelu (např. zda kód odpovídá návrhu apod.) - tj. odpověď na otázku: vytvářím produkt správně?
- validace = vyhodnocení SW na konci procesu vývoje SW, abychom zajistili splnění požadavků na SW - tj. odpověď na otázku: vytvářím správný produkt?
- verifikace a validace - široké téma, dále pouze následující tři oblasti:
    - automatická statická analýza - používá se nejčastěji pro kontrolu zdrojových textů SW systému, případně kontrola modelů apod.
    - inspekce - ruční kontrola artefaktů SW procesu, typicky prováděná skupinou 3 až 5 lidí
    - testování - spouštění programu s takovými daty, abychom v programu odhalili defekty

Základní termíny:

- omyl (error) - chybná úvaha nebo překlep vývojáře, vede k jednomu nebo více defektům
- defekt (fault, bug, defect) - rozdíl mezi chybným programem a jeho správnou verzí
- symptom (symptom, failure, run-time fault) - pozorovatelné chybné chování programu; defekt se při konkrétním běhu může projevit žádným, jedním nebo více symptomy

Automatická statická analýza

- používají se programy pro automatickou kontrolu modelů nebo zdrojových textů
    - překladač jazyka Java – silná typová kontrola,
    - slabě typované jazyky (např. C) - statické analyzátory (code checkers), např. lclint(1) - detekuje neinicializované proměnné, odchylky od standardů apod.
- další kontroly - mnoho programů používá pro detekci podezřelých míst heuristiky
    - nevýhoda: pokud zdrojový text neodpovídá heuristikám zabudovaným v programu, mohou tyto programy produkovat falešná chybová hlášení
    - příklad program: Jlint pro jazyk Java

Inspekce a procházení programů

- používají se při přezkoumávání (review) DSP, detailního návrhu, kódu (provádějí se před testováním programu, může jim předcházet statická analýza)
- zahrnují čtení dokumentu nebo programu týmem např. 3 nebo 4 lidí (jeden z nich je autor) s cílem nalézt defekty (nikoli jejich řešení)
- pro inspekce kódu:
- výhody:
    - bývají poměrně efektivní (najdou 30% až 70% defektů detailního návrhu a kódování)
    - úsilí bývá přibližně poloviční oproti ekvivalentnímu otestování na počítači (za předpokladu, že nemáme testy již připravené, jinak mohou testy běžet automaticky)
    - cena opravy defektu bývá nižší než při testování na počítači (je známá přesná příčina defektu, zatímco testování na počítači najde pouze symptomy)
    - nalézá jiné typy defektů než klasické testování, tj. je s ním komplementární (je vhodné provádět obojí)
- nevýhody:
    - je třeba získat s inspekcemi zkušenost
    - experimenty ukazují, že individuální čtení kódu je naopak méně efektivní než individuální testování

Faganovské inspekce kódu

- zahrnují procedury a techniky pro skupinové čtení kódu, poprvé využity ve firmě IBM
- tým provádějící inspekci - obvykle 4 osoby
    - moderátor
        - distribuuje materiály pro schůzku, plánuje a vede schůzku, zaznamenává defekty, zajišťuje jejich opravy
        - schopný programátor, nikoli však autor programu;
        - nemusí mít ani detailní znalosti programu, jehož inspekce se provádí
    - programátor - autor programu
    - návrhář (pokud je odlišný od programátora)
    - specialista na testování

Před schůzkou:

- moderátor (v dostatečném čase předem) rozdistribuuje program a specifikaci návrhu programu, účastníci se mají s materiálem seznámit

Na schůzce:

- optimální doba inspekce cca 90 -120 min, bez přerušení

1. programátor požádán o vysvětlení logiky programu příkaz po příkazu
    - během programátorova proslovu mají ostatní účastníci otázky, jejichž cílem je zjistit, zda se v kódu nacházejí defekty
    - velkou část defektů najde programátor během výkladu (samotné čtení kódu před posluchači je efektivní - technika pro hledání defektů)
    - moderátor zodpovědný za to, že se účastníci zaměří na vyhledávání defektů, nikoli na jejich řešení
2. program analyzován vzhledem k seznamu obvyklých programátorských chyb (seznam byl vytvořen např. v průběhu předchozích inspekcí)

Po skončení schůzky:

- programátor dostane seznam defektů
    - pokud je nalezeno více defektů nebo pokud některý defekt vyžaduje podstatný zásah do programu, může se domluvit nová inspekce po opravě programu
    - defekty jsou analyzovány a kategorizovány, použijeme je pro zpřesnění seznamu obvyklých programátorských chyb použitých v bodě (2)

Další poznámky:

- při většině inspekcí se projde cca 150 příkazů za hodinu
- vedlejším efektem zpětná vazba týkající se programátorského stylu, výběru algoritmů a programovacích technik

- identifikace částí, které obsahují více defektů
    - defekty se vyskytují ve shlucích, pravděpodobnost existence dalších defektů v dané sekci programu (např. podprogramu) je přímo úměrná počtu defektů v příslušné sekci již nalezených
    - pokud jsou v některé sekci nalezeny defekty, měli bychom se na ní více zaměřit, např. při testování

- inspekce fungují jen v případě, že k nim všichni účastníci mají patřičný přístup (programátor může chápat inspekci jako útok na svou osobu nebo jako pomoc ke zlepšení kvality svého kódu)

Příklad obvyklých programátorských chyb (pro jazyk C):

- data
    - je proměnná inicializována?
    - jsou odkazy do pole v rámci definovaných mezí pole?
    - nenastává při indexování pole chyba off-by-one?
    - ukazuje ukazatel na alokovanou paměť?
    - pokud čteme záznam ze souboru, má proměnná správný typ?
- chyby výpočtu
    - jsou v kódu výpočty se smíšenými typy (např. sčítání float a int)?
    - je do kratší hodnoty (např. int) přiřazována hodnota s delší reprezentací?
    - je možné přetečení nebo podtečení během výpočtu?
    - může nastat případ, že dělitel je 0?
    - jaké jsou důsledky nepřesností reálné aritmetiky?
    - atd.
- řízení toku
- rozhraní
- vstup a výstup
- ostatní

Procházení kódu (walkthroughs)

- technika detekce defektů pomocí skupinového čtení kódu (podobně jako inspekce)- v podrobnostech se liší
- schůzka 3 až 5 lidí, 1 až 2 hodiny, nemá být přerušena
    - moderátor - podobně jako v případě inspekcí
    - sekretář - zaznamenává všechny nalezené defekty
    - tester
    - programátor - autor kódu
    - role ostatních členů týmu není ustálená, doporučuje se např. zkušený programátor, začínající programátor (má zatím nezkalený pohled), osoba, která bude provádět údržbu apod.

Před schůzkou: materiál předem

Schůzka: - hra na počítač:

- tester přijde na schůzku s malým počtem papírových testovacích případů - vstupy a očekávané výstupy programu (podprogramu)
- tým provádí testovací případ, stav programu (hodnota proměnných) zaznamenává na tabuli nebo na papír - v případě nejasností se ptá programátora na logiku programu a na předpoklady (většina defektů je nalezena otázkami, nikoli testovacími případy)

Po schůzce: obdobně jako u inspekcí - tj. programátor dostane seznam defektů, opraví

Další poznámky:

- opět podstatný přístup - tým by měl hodnotit program, nikoli toho, kdo program napsal (defekty nejsou způsobeny neschopností programátora, ale nutný důsledek nedokonalých metod programování a složitosti problému)

Testování

- testování = spouštění programu se záměrem najít v něm defekty (tj. snažíme se, aby se projevily symptomy případných defektů)

Black box a white box testování

- dva základní přístupy k testování - "black box" a "white box" testování

Black box testování (také názvy: functional, data-driven, input/output driven testing)

- tester na program pohlíží jako na černou skříňku s danou specifikací, vnitřní struktura a vnitřní funkce programu ho nezajímají
- hledá případy, ve kterých se program nechová podle specifikace
- pro nalezení všech defektů by bylo nutné otestovat program se všemi možnými vstupy (platnými i neplatnými), což je prakticky nemožné (např. překladač jazyka C bychom museli otestovat se všemi platnými i neplatnými programy)
- Otázka: jak maximalizovat počet defektů nalezený konečným počtem testovacích případů? -> k programu nemůžeme přistupovat čistě jako k černé skříňce, ale musíme učinit rozumné předpoklady o jeho vnitřním chování

White box testování (také: glass-box, clear-box, logic-driven testing)

- testovací data se odvozují z programové logiky

- pro úplné otestování programu bychom potřebovali pomocí testovacích případů otestovat všechny možné logické cesty v programu (analogie otestování programu se všemi možnými vstupy, viz výše)

- ale - dva zásadní problémy:

1. počet logických cest je i v malých programech příliš velký – například

```
for (i=0; i<100; i++) {
    if (podmínka)
        příkaz1;
    else
        příkaz2;
}
```

   - za předpokladu nezávislosti podmínek 2^100 logických cest, kterými může být vykonán (ve skutečnosti podmínky nebudou nezávislé, takže cest bude méně)

2. i po otestování všech logických cest mohou v programu zůstat nenalezené defekty - některé logické cesty mohou chybět a nemusejí být nalezeny defekty citlivé na data, např.

```
if ((a - b) < epsilon) ...  // místo: if (abs(a - b) < epsilon)
```

- při black-box i white-box testování se budou testovací případy skládat z popisu vstupních dat a z popisu správného výstupu pro daná vstupní data - program nebo jeho část spustíme se vstupními daty, porovnáme předpoklad se skutečným výstupem (nejlépe automaticky)

- testovací případy mají obsahovat platné i neplatné vstupy
- testovací případy je třeba uchovávat, protože je můžete znovu potřebovat (např. pro otestování programu po změně)
- je nutné také zkontrolovat, zda program neprovádí nechtěné vedlejší efekty (zápisy do databáze apod.)

Návrh testovacích případů

- úplné otestování programu není možné ->pro testování je velmi podstatný návrh efektivních testovacích případů, tj. hledáme podmnožinu všech testovacích případů, která má největší pravděpodobnost nalézt většinu defektů
- první nápad - náhodně vybraná podmnožina všech možných vstupů - pravděpodobně jedna z nejhorších možností - malá pravděpodobnost být optimální podmnožinou nebo alespoň se jí blížit

- použitelné metody - kombinace myšlenek black box a white box testování
- existuje několik metodik, každá má své silné a slabé stránky - tj. každá detekuje/přehlédne jiné typy defektů -> vhodné připravovat testovací případy pomocí více metod

Rozdělení vstupů do ekvivalentních tříd

- dobrý testovací případ bude mít dvě vlastnosti:
  - bude vyvolávat co nejvíc vstupních podmínek, tím omezí celkový počet potřebných testovacích případů
  - bude pokrývat určitou množinu vstupních hodnot

Množinu vstupů pak rozdělíme do tříd ekvivalence tak, abychom mohli rozumně předpokládat, že test nějaké reprezentativní hodnoty v dané třídě je ekvivalentní testu kterékoli další hodnoty

- z těchto úvah je odvozena metodologie pro black-box testování známá jako equivalence partitioning = rozdělení do tříd ekvivalence - nejprve identifikujeme třídy ekvivalence a pak definujeme testovací případy

identifikace tříd ekvivalence

- vezmeme každou vstupní podmínku a podle ní rozdělíme množinu všech vstupních hodnot do dvou nebo více podmnožin

- existují dva typy tříd ekvivalence - platné (reprezentující platné vstupy) a neplatné (reprezentující chybné vstupní hodnoty)
- rozdělení do tříd ekvivalence je heuristický proces, můžeme využít následujících doporučení:
  - pokud vstupní podmínka specifikuje interval hodnot (např. rok může být mezi 1900 a 2050), pak máme jednu platnou třídu ekvivalence (hodnoty 1900 až 2050) a dvě neplatné třídy ekvivalence (hodnoty < 1900, hodnoty > 2050)
  - pokud vstupní podmínka specifikuje množinu vstupních hodnot a pokud lze předpokládat, že každá z nich bude obsluhována jinak (např. "vlak", "autobus"), bude jedna platná třída ekvivalence pro každý prvek množiny; přidáme jednu neplatnou třídu ekvivalence pro další prvek množiny (např."letadlo")
  - pokud vstupní podmínka specifikuje situaci, která "musí nastat", např. první znak identifikátoru musí být písmeno, bude jedna platná třída ekvivalence (první znak je písmeno) a jedna neplatná třída ekvivalence (není písmeno)
  - pokud je důvod předpokládat, že prvky nějaké třídy ekvivalence nejsou obsluhovány stejně, rozdělte třídu do menších tříd ekvivalence

definice testovacích případů

- pro každou neplatnou třídu ekvivalence vytvoříme samostatný testovací případ (to je nutné, abychom otestovali každou podmínku kontrolující neplatný vstup); testovací případy budou typicky obsahovat:
    - příliš málo dat nebo žádná data
    - příliš mnoho dat
    - neplatná data (např. negativní počet zaměstnanců)
- pro platné třídy ekvivalence vytváříme testovací případy, pokrývající co nejvíce platných tříd ekvivalence; testovací případy budou typicky obsahovat:
    - nominální případy = běžné nebo očekávané hodnoty
    - minimální normální konfiguraci (např. jediný zaměstnanec)
    - maximální normální konfiguraci (pokud ji umíme určit)

- je velmi vhodné testovat hraniční hodnoty tříd ekvivalence vstupních hodnot – např. pokud je platný vstup -1.0 až +1.0, pak vytvoříme testovací případy pro vstupy -1.0, +1.0, -1.0001, +1.0001 (stejně dobré-nutné otestovat hranice výstupních hodnot)

- analýza hraničních hodnot je jedna z nejužitečnějších technik pro návrh testovacích případů, ve skutečnosti ale vyžaduje více přemýšlení, než to vypadá na první pohled.

Příklad (bankomat)

Např. SW bankomatu bychom mohli otestovat následujícími testovacími případy:

1. Vadná karta, konec.
2. Platná karta, chybné PIN, konec.
3. Platná karta, platné PIN, konec.
4. Výběr platné částky, dotaz na zůstatek.
5. Výběr neplatné částky.
6. Neplatný dotaz na zůstatek.

White-box testování

- využíváme znalost implementace
- obvykle pro testování relativně malých částí programu (podprogramy v modulu, metody třídy) - tj. testování jednotek
- pokud jsou moduly integrovány do systému, složitost narůstá tak, že jsou strukturální techniky prakticky neproveditelné
- se zvyšováním rozsahu projektu testy jednotek zabírají menší podíl na celkovém času vývoje (cca od 35% pro malé systémy až po cca 8% pro velké systémy)

- pokud známe strukturu implementace, můžeme pro testování použít třídy ekvivalence a testovat běžné a hraniční podmínky se znalostí kódu

- vytváření testů si vynucuje dobrý návrh - má-li být kód testovatelný, je nutné, aby moduly/třídy byly volně vázané, nevyžadovaly složitou inicializaci apod.

- z kódu můžeme určit také další typ hraniční podmínky, který nastává, pokud dochází ke kombinaci vstupních hodnot - například pokud podprogram hodnoty násobí, testovací případy mohou zahrnovat dvě velká kladná čísla, dvě velká záporná čísla apod.

Pokrytí kódu testovacími případy

- při white-box testování nás zajímá, do jaké míry testovací případy pokrývají zdrojový text programu - praktické metody testují pouze rozumnou podmnožinu cest v programu

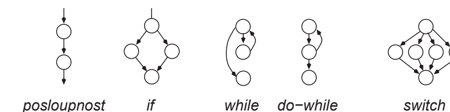
- je dobrým kritériem alespoň jedno vykonání každého příkazu? (statement coverage, pokrytí všech příkazů)

např.if (a>1) and (b=0) then x = x/a;

- oba příkazy (if a přiřazovací příkaz) by bylo možné pokrýt jediným testovacím případem (a=2, b=0) -> pak neodhalíme defekty
    - místo "and" má být "or",
    - místo "a>1" má být "a>=1" nebo "a>0"

- závěr: podmínka pokrytí všech příkazů je nutná, ale nikoli postačující

- kritérium pokrytí zesílíme - je dobrým kritériem pokrytí všech rozhodovacích příkazů tak, aby byly vykonány všechny jejich větve?
    - musíme vytvořit tolik testovacích případů, aby se v každém příkazu "if" vykonala alespoň jednou větev při podmínce "false" a alespoň jednou větev při podmínce "true" atd.
    - pro složitější podprogramy se vyplatí modelovat cestu podprogramem pomocí orientovaného grafu popisujícího možný tok řízení v podprogramu



Obrázek 119 Tok řízení v podprogramu

- uzly reprezentují příkaz nebo část příkazu (přiřazovací příkazy, volání podprogramů, případně podmínky rozhodovacích příkazů)
- každý pár uzlů, pro který je možný přenos řízení, je spojen hranou

- nezávislá cesta = musí procházet alespoň jednu novou hranu v grafu, tj. provést jednu nebo více nových podmínek
- existují extra případy (podprogram neobsahuje rozhodování, má více vstupních bodů apod.) - připojujeme ještě podmínku pokrytí všech příkazů
- kritérium "pokrytí všech rozhodovacích příkazů" by postačovalo, pokud bychom měli vždy jedinou podmínku v rozhodovacím příkazu, ale např.

pro kód "if (a>1) and (b=0) then x = x/a" je to stále ještě slabé kritérium

- pokud otestujeme pomocí (a=2, b=0) a (a=2, b=1), neodhaleny by zůstaly defekty jako "místo a>1 má být a>=1"

->rozumným kritériem je **pokrytí všech kombinací podmínek v rozhodovacím příkazu** (multiple condition coverage)

- oproti branch coverage navíc vyžaduje, aby byly otestovány všechny kombinace hodnot logických operandů v podmínce
- pak kód "if (a>1) and (b=0) then x = x/a"

můžeme otestovat čtyřmi testovacími případy:

- . (a=2, b=0) => obě podmínky jsou true, větev "then" se provede
- . (a=2, b=1) => true, false, větev "then" se neprovede
- . (a=1, b=0) => false, true, větev "then" se neprovede
- . (a=1, b=1) => false, false, větev "then" se neprovede

- testovací případy není vhodné generovat strojově z kódu, ale můžeme si pomoci nástroji generujícím "nápady na testovací případy" z výrazů

- další obvyklé kritérium - **pokrytí smyček** – vyžaduje 3 testovací případy:

  - tělo smyčky se nevykoná, tj. při prvním vyhodnocení bude test "false"
  - tělo smyčky se vykoná právě jednou, tj. při prvním vyhodnocení bude test "true", poté "false"
  - tělo smyčky se vykoná více než jednou, tj. test bude "true" nejméně dvakrát

- další možné kritérium - all-du-path
  - jedna cesta pro každou definici-použití: pokud je proměnná definována v jednom příkazu a použita v jiném, cesta by měla procházet oběma příkazy

- pro určení pokrytí velkých sad testů spouštěných na celé systémy jsou užitečné tyto podmínky:

  - pokrytí podprogramů - zda byl podprogram vyvolán alespoň jednou
  - pokrytí volání - zda každý podprogram volal všechny podprogramy, které může volat

- testy prováděné bez měřeného pokrytí kódu typicky pokrývají pouze 55% kódu

(Grady 1993)

- pro zjištění, které cesty byly vykonány, se používají dynamické analyzátory programu - při překladu je ke každému příkazu připojen kód, který počítá, kolikrát byl daný příkaz vykonán (tzv. instrumentace)
- po běhu můžeme zjistit, které části programu nebyly pokryty příslušným testovacím případem

Strategie testování

- testování by mělo být předem naplánováno spolu s celým SW procesem
- pro zvýšení efektivity testování je vhodné provádět také inspekce kódu

- testování by mělo začínat na úrovni jednotek (procedur, tříd - funkce každé jednotky se ověří samostatně) a postupovat směrem k větším celkům (podsystémům, celému systému)

- testování jednotek provádí obvykle ten, kdo danou část napsal; testování větších celků provádí nebo alespoň řídí specialista - tester (rozsáhlejší software testuje nezávislá testovací skupina)

Poznámka (psychologický problém testování)

- pro řadu programátorů je obtížné testovat vlastní programy, protože
  - jejich zájem je spíše ukázat, že jejich program neobsahuje defekty a pracuje podle požadavků zákazníka.
  - bývá obtížné přepnout se z kódování programu (proces konstrukce) na testování (destrukci).
  - program obsahuje chyby způsobené nepochopením specifikace, které programátor nemůže sám odhalit.
- proto je vhodné přenechat testování sestaveného programu někomu jinému, např. nezávislému testovacímu týmu. Programátor pak je zodpovědný za otestování jednotek, které vytvořil (procedur, funkcí, tříd).

Testování by mělo probíhat postupně současně s implementací systému v následujících krocích:

- testování jednotek (unit testing) - testujeme nejmenší jednotky návrhu, např. procedury nebo funkce; pro testování můžeme používat white-box techniky

- integrační testování (integration testing) - sestavujeme software, spolu s tím testujeme defekty týkající se rozhraní mezi jednotlivými částmi
- validační testování (validation testing) - po integraci se zaměřujeme na funkce viditelné uživatelem
- testování systému (system testing) - pokud SW je pouze jednou součástí většího celku, účelem je otestovat celek; např. zátěžové testování, zotavení po závadě apod.

Testování jednotek

- pojmem "jednotka" se v případě konvenčně napsaného SW obvykle myslí procedura, funkce, nebo nejmenší samostatně přeložitelná jednotka zdrojového textu (čili neexistuje všeobecně přijímaná definice)
- jednotka se testuje samostatně, okolní jednotky jsou nahrazeny ovladači testů (řídí testovanou jednotku) nebo testovacími maketami (nahrazují jednotky volané z testované jednotky)
- používají se již probrané white-box techniky

- pro objektově-orientovaný software se za jednotku považuje třída
- při testování třídy bychom měli provést:
  - samostatné otestování každé metody
    - některé metody lze testovat až po předchozím vyvolání jiných metod, např. po inicializaci objektu
    - pokud používáme dědičnost, musíme testovat i všechny zděděné operace (mohou obsahovat předpoklady o dalších operacích a atributech, které ale mohly být potomkem změněny; obdobně musíme znovu otestovat potomky při změně rodiče)
  - testovat průchod všemi stavy objektu, případně simulace všech událostí, které způsobují změnu stavu objektu
    - pokud jsme vytvořili stavový diagram objektu, můžeme z něj určit posloupnost přechodů, které chceme testovat, a najít posloupnost událostí, které ji způsobí
  - testovat nastavení všech atributů objektu

Nástroje JUnit a JUnitDoclet

- pomůcka pro testování jednotek v jazyce Java - knihovna JUnit, která poskytuje nástroje pro spouštění testovacích případů a umí graficky i textově zobrazit výsledky testů. Podobné nástroje existují i pro mnoho dalších programovacích jazyků.

- pro jednotkové testy je důležité mít také zavedené konvence, například:
  - pro každou třídu "Třída" vytvoříme třídu "TřídaTest"
  - pro každou veřejnou metodu vytvoříme testovací metodu "testJménoMetody()"
  - testovací metody můžeme vytvořit i pro ty soukromé metody, které obsahují nějaký složitější algoritmus
  - testujeme vždy správné chování i chování při chybách, testovací metody můžeme někdy rozdělit do dvou skupin:
    - testJménoMetody - testuje obvyklé chování
    - testJménoMetodyFailures - testuje chování při chybách
  - někdy můžeme navíc vytvořit testovací třídu pro testování spolupráce tříd

- s knihovnou JUnit souvisí pomocný nástroj JUnitDoclet (plug-in do Eclipsu), umí vygenerovat kostru testovacích případů pro JUnit.

Poznámka (jednotkové testy a makety)

- na to, aby kód (vytvářené třídy) byl testovatelný, je třeba pamatovat už při návrhu; nejlepší je navrhovat testy současně s kódem (či dokonce nejdříve testy, potom kód – tzv. programování řízené testy)

- pokud metoda "foo()" třídy A používá metodu "bar()" třídy B, máme dvě možnosti:
  - pokud je třída B dostatečně samostatná a dobře testovatelná, využijeme jí přímo - v testu se jí dotážeme na výsledek operace "bar()"
  - pokud je třída B špatně testovatelná (např. ve skutečnosti tiskne na tiskárnu, chová se nedeterministicky apod.), použijeme místo ní v testu jednoduchou maketu, které se můžeme dotázat na výsledek operace, aplikačnímu kódu obvykle předáváme maketu jako argument
- jednotkové testy většinou netestují spolupráci mezi třídami; proto jsou zapotřebí také ostatní typy testů

Integrační testování

- po otestování individuálních komponent musíme komponenty integrovat - sestavit do částečného nebo úplného systému
- výsledek musíme otestovat na problémy, které vznikají interakcí komponent
- (velký třesk) - po otestování jednotlivých modulů je z nich v jediném kroku sestavena aplikace
- použitelné pouze pro malé programy, ale i u nich mohou nastat problémy, pro větší systémy nejméně efektivní způsob integrace = vysoká pravděpodobnost neúspěchu
- hlavním problémem je lokalizace defektů, protože vztahy mezi komponentami mohou být značně složité
  - proto často pro integraci a testování doporučuje inkrementální přístup
  - nejprve integrujeme minimální konfiguraci systému, otestujeme
  - k systému přidáváme inkrementy, po každém přidání systém otestujeme
  - pokud nastaly problémy, budou pravděpodobně (ale ne nutně) způsobeny přidáním posledního inkrementu

- ve skutečnosti to není tak jednoduché, protože některé vlastnosti budou rozptýleny do několika komponent, vlastnost můžeme otestovat až po integraci těchto komponent => při plánování testů je třeba počítat s časovým plánem na dokončení modulů

- pokud má důležitý modul neočekávané problémy, může se tím zdržet celá integrace (programátor řeší problém, zatímco všichni ostatní na něj čekají)

Testování rozhraní

- cílem testování rozhraní je detekovat defekty, které mohou vzniknout chybnou interakcí mezi moduly nebo podsystému nebo chybným předpokladem o rozhraní
- testování rozhraní je obtížné, protože některé typy defektů se projeví pouze za neobvyklých podmínek
- obecná pravidla (volně podle Sommerville 2001):
  - v testovaném kódu najděte všechna volání externích komponent
  - navrhněte množinu testů tak, aby externí komponenty byly volány s parametry, které jsou extrémy jejich rozsahu (např. prázdný řetězec, dlouhý řetězec, který by mohl způsobit přetečení apod.)
  - navrhněte testy, které by měly způsobit neúspěch externí komponenty (zde často chybné předpoklady)
  - v systémech s předáváním zpráv použijte zátěžové testování (viz dále)
  - pokud spolu komponenty komunikují prostřednictvím databáze, sdílené paměti apod., navrhněte testy, ve kterých se bude lišit pořadí aktivace komponent; testy mohou odhalit implicitní předpoklady programátora o pořadí, v jakém pořadí budou sdílená data produkována a konzumována

- mnoho defektů rozhraní odhalí statické testy
  - např. silná typová kontrola v překladačích jazyka Java
  - pro slabě typované jazyky (např. C) by se před integračním testováním měly použít statické analyzátory (code checkers)
- některé inspekce se mohou zaměřit také na rozhraní komponent a jejich předpokládané chování

Validační testování

- začíná tam, kde končí integrační testování
- testujeme, zda SW splňuje požadavky uživatele

- akceptační testování - test zda produkt splňuje požadavky zadavatele, testuje zadání na reálných datech, testuje pouze externí chování systému, vnitřní strukturu přitom ignorujeme
  - obsah testu má specifikovat zadavatel
  - obsahem by měly být instance případů použití (tj. obsah funkčního testu je výhodné specifikovat v souvislosti se sběrem požadavků)
  - snaha o zautomatizování, abychom mohli spouštět po změnách aplikace

Alfa a beta testování

- pro generické produkty není většinou možné vykonat přejímací testování u každého zákazníka, proto alfa a beta testování
- alfa testy: na pracovišti, kde se SW vyvíjí (známé prostředí) - testuje uživatel, vývojoví pracovníci ho sledují a zaznamenávají problémy
- beta testy: testují vybraní uživatelé ve svém prostředí (vývojářům neznámém) - defekty ohlášené uživateli jsou opraveny => finální produkt

Testování systému

- účelem otestovat celý systém, jehož je SW součástí
- mívají různý účel, např. otestovat vlastnosti jako je výkonnost, kompatibilita, bezpečnost, instalovatelnost, spolehlivost apod.

Testování výkonnosti

- v mnoha typech systémů je nepřípustné, aby SW nesplňoval požadavky na výkonnost (zejména v řídicích systémech)
- v takovém případě by se výkonnost měla testovat ve všech krocích včetně jednotkových testů
- pomocné procedury monitorují dobu vykonání apod.

Zátěžové testování

- obvykle se používají testy, kde se zátěž postupně zvyšuje, dokud není výkonnost systému neakceptovatelná nebo dokud systém nehavaruje
- zátěž = množství dat, frekvence požadavků, data, která jsou extrémně náročná na zpracování
- je vhodné určit části kódu, které mohou být problematické při velké zátěži, zátěžové testy navrhnout tak, aby pokrývaly především tyto části kódu
- ověření, zda havárie systému nepoškodí data apod.
- může odhalit některé defekty, které se normálně neprojeví
- důležité zejména u internetových aplikací, v distribuovaných systémech apod., kde se vysoce zatížené systémy mohou zahltit, protože si vyměňují mnoho koordinačních dat, čímž se opět zvyšuje zátěž systému atd.

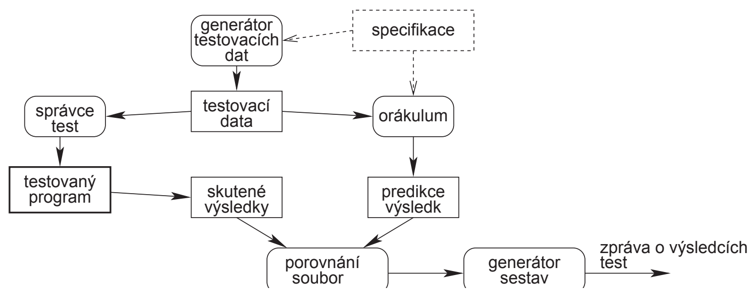
Testování zotavení systému po havárii

- mnoho systémů se musí být schopno zotavit po havárii v předepsaném čase, jinak hrozí značné finanční ztráty
- při testování zotavení systému způsobujeme různé havárie systému a ověřujeme, zda se systém zotavil správně a v časovém limitu

Nástroje pro testování SW

- rozsáhlé systémy - cena testování až 50% ceny vývoje, mohou existovat stovky až tisíce testovacích případů -> automatizace testů
- automatizace testů snižuje cenu změn - programátoři se nemusejí tolik obávat, že změnou zanesou do kódu defekt, protože testy by defekt (s určitou pravděpodobností) odhalily

- často následující uspořádání:



Obrázek 120 Nástroje pro testování - uspořádání

- správce testů (test manager) - řídí běh testů
- generátor testovacích dat (test data generator) - generuje testovací vstupní data pro testovaný SW
- orákulum (oracle) - generuje předpokládané výstupní hodnoty (nový program, prototyp SW, předchozí verze SW – tzv. regresivní testování, SW vytvořený konkurencí apod.)
- program pro porovnání souborů (file comparator) - porovná výsledky skutečného běhu s předpokládanými hodnotami vygenerovanými orákulem; často lze použít univerzální programy
- generátor zpráv (report generator) - umožňuje definovat a generovat zprávy o výsledcích testů

- progresivní fáze testování přidává a testuje nové fce (nově přidané nebo modifikované moduly a jejich rozhraní s již integrovanými moduly)
- regresivní fáze testuje důsledky změn v již integrovaných částech.

Nástroje pro zachycení a pozdější přehrání testů (capture-replay tool)

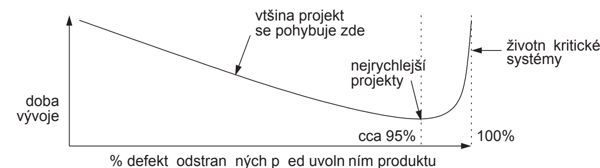
- informace protékají SW systémem - na vhodné místo toku dat aplikací vložíme nástroj, který tekoucí data zaznamená
- tester spustí/vykoná a zaznamená testovací případy
- později může zaznamenané testovací případy spustit znovu (regresivní testování)

- komerční capture-replay nástroje pro zachycení/přehrání stisků kláves a pohybů myši, obsahující i nástroje pro zachycení a porovnání výstupů na obrazovku - používané pro GUI aplikace
- pro většinu ostatních účelů (testování zapouzdřených systémů apod.) je většinou nutné vytvořit si vlastní SW pro podporu testování

Defekty v číslech

- podle kvality vývoje asi 10 až 50 defektů na 1000 řádek kódu před testováním
- testy najdou obvykle méně než 60% defektů, proto se kombinují s inspekcemi
- „paradox“: defekty jsou častěji v testovacích případech než v testovaném kódu (McConnell 1993)
- pro kritické systémy - kombinace formálních metod vývoje, inspekcí a testování

Doba vývoje vs. počet defektů



Obrázek 121 Doba vývoje a počet defektů

Ladění

- testováním nalezneme defekty, následuje proces ladění (angl. debugging)
- ladění = identifikace příčiny defektu (90% času) a její oprava (10% času)
- v 5 krocích –
  - stabilizace symptomu,
  - nalezení příčiny,
  - oprava,
  - otestování opravy defektu,
  - vyhledání obdobných defektů

- stabilizace symptomu
  - potřebujeme, aby se defekt projevoval spolehlivě, hledáme testovací případ, který symptom reprodukuje
  - testovací případ co nejvíce zjednodušíme, aby se defekt ještě projevoval (často již můžeme vytvářet hypotézu, proč defekt nastává)
  - defekty, které se neprojevují spolehlivě, (např. neinicializované proměnné, neplatný ukazatel, chyby časového souběhu), se snažíme zviditelnit (např. před spuštěním programu paměť zaplníme náhodnými hodnotami apod.)

- nalezení příčiny symptomu
  - zúžení podezřelé části kódu
  - použití ladícího programu nebo ladících výpisů - sledujeme, kde nastane symptom
  - dobře se vyspat

- oprava defektu
  - u nalezeného defektu bývá oprava poměrně jednoduchá,
  - vysoké nebezpečí zanesení dalšího defektu (podle některých studií více než 50%)
  - větší šanci provést opravu správně mají programátoři s celkovou znalostí programu, je třeba rozumět jak problému, tak opravovanému programu

- otestování opravy defektu
    - defekty je nutné opravovat po jednom, opravy po jedné otestovat
    - poté program otestovat jako celek (regresivní testy), ukáží se případné vedlejší efekty opravy
    - pro případ defektu v opravě je vhodné mít uchovánu předchozí verzi

- hledání obdobných defektů

Defenzivní programování

- zabezpečení definovaného chování při chybných vstupech, zamezení propagace chyb z podprogramu ven
- nástroje: kontroly vstupních parametrů
    - makro assert() v C, aserce v Javě), (programming by contract)
    - výjimky v Javě, C++, Delphi, Pythonu apod. - konstrukce typu try-catch a try-catch-finally:

```
try {
   foo();                          // zde může nastat výjimka
} catch (SomethingWentWrongException e) {  // pokud nastane výjimka,
   System.out.println("some error");     //  provede se toto
} finally {                          // nakonec se vždy provede
   dispose();                        //  toto
}
```

- reakce na chybu
    - fce vrátí speciální návratový kód
    - programová výjimka, způsob propagace - podle stanovených konvencí
    - nastavení defaultní hodnoty vstupu nebo defaultního stavu
- o způsobu reakce na chybu by se mělo rozhodnout na úrovni architektury, aplikace by se měla ve všech svých částech chovat konzistentně
- během vývoje potřebujeme, aby defekty byly co nejlépe viditelné, ve výsledném produktu se naopak snažíme, aby defekty co nejméně rušily, a proto:

- ponecháváme kód, který kontroluje významné defekty; pokud aplikace hlásí interní chybu, měla by také oznámit, jakým způsobem jí uživatel může ohlásit
- „zrušíme“ kód testující nepodstatné defekty,
- „zrušíme“ kód, který způsobuje havárie,
- ponecháme kód, který umožní přijatelné ukončení aplikace při chybě (např. s uložením dat)
- rušení kódu neprovádíme fyzicky (při ladění ho budeme opět potřebovat), ale např. pomocí preprocesoru vynecháme tělo procedury Assert, využijeme verzování apod.

Agile Testing
@ Kerio Technologies

Zdeněk Samuel, QA Manager

KERIO Connect. Communicate. Collaborate. Securely.

Agenda

- About Kerio and myself

- Testing in general
- Agile testing
- Testing in stabilization phase
- Supportive systems
- Quality measurement

- Opportunities

KERIO Connect. Communicate. Collaborate. Securely.

About …

KERIO Connect. Communicate. Collaborate. Securely.

About Kerio

>200 employees worldwide (Pilsen, San Jose, Cambridge, New York, Cologne, Moscow, …)

~100 developers in Pilsen

Kerio Connect (~39 000 active installations)

Kerio Control (~22 000 active installations) + HW

Kerio Workspace (since March 2011)

Kerio Operator (since January 2011) + HW

Agile development (Scrum)

Shrink-wrapped products not contract development

About myself

QA manager – member of the Development Management Group

- Quality Assurance
- Lead the QA and DOC team
- Test design

Team leader of testers

Technical support engineer

11 years in Kerio

Testing in general

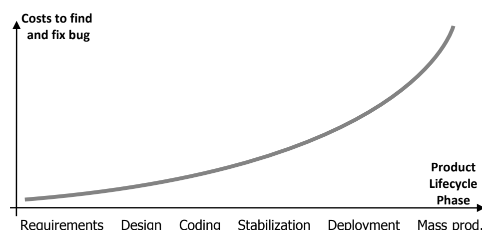
Why to test?

Kerio philosophy

- Quality. Simple. Stable. Secure. SMB. Chanel.

Perfect development process is not possible!

- Functionality
- Usability
- Stability
- Security
- Performance
- Compatibility
- Testing decreases costs



How much testing?

- Risk-based, business decision
- **It’s not possible to guarantee the software has no faults!**

And at Kerio?

- QA is independent team of testers as a part of whole development
- 10 testers, 6 sw engineers in AutoTests team
- => rather “better testing” than “more testing”

Psychology of testing

- Primary intent is to find faults
- It's based on different mindset of a tester and a developer

Testing is "against nature"

- It brings bad news
- Successful test discovers bugs

A good tester must be polite to developers

- It protects against developer's demotivation

KERIO Connect. Communicate. Collaborate. Securely.

Agile testing

KERIO Connect. Communicate. Collaborate. Securely.

SW development

Agile development

- Feature driven development → Date driven development
- Regular betas
- Regular automatic builds / tests
- Scrum is used in development

An initial documents for QA

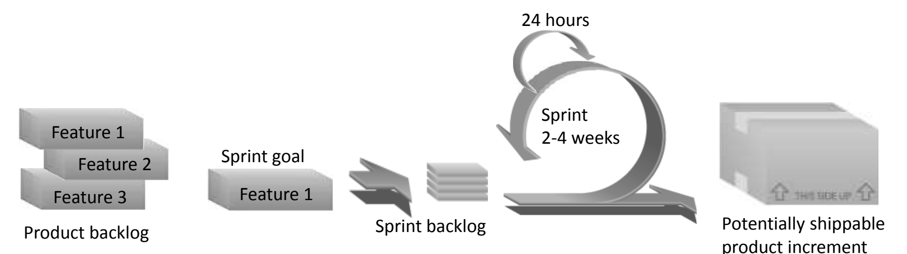
- Roadmap
- Vision Scope Document
- "Functional specification"

KERIO Connect. Communicate. Collaborate. Securely.

Scrum 1/2

Characteristic

- Iterative incremental process of software development
- Produces demonstrable working software every iteration
- Results-, commitment-, business- oriented
- Self-organizing teams
- No special training – easy to start



Scrum 2/2

Roles
- Product owner
- Scrum Master
- Team (tester included)
- Customer

Ceremonies
- Sprint planning
- Sprint review
- Sprint retrospective
- Daily scrum meeting

Artifacts
- Product backlog
- Sprint backlog
- SCRUM board

Testing in SCRUM 1/3

- New features testing

Tester’s roles and responsibilities
- Member of SCRUM team
- Participate on all ceremonies
- Participate on design
- Create verification / acceptance criteria (test cases)
- Test all features (in the same sprint)
- Using daily / hourly builds
- Reporting bugs

Testing in SCRUM 2/3

Acceptance / validation criteria
- User stories
- Instead of specifications

Test design
- The new test cases are created for acceptance / regression automatic tests

Black-box, functional tests
- Based on nescience of code and knowledge of model
- Specification (or requirements), installation packages

Testing in SCRUM 3/3

Acceptance testing
- Against to acceptance / verification criteria
- Feature is done / not done (acceptable by “customer”)

Exploratory testing
- An approach in software testing with simultaneous learning, test design and test execution
- Requirements and specifications are incomplete
- Most experienced tester

Other techniques 1/2

Test-Driven development / Unit testing

- Test is written before the feature has been implemented
- The developer must understand the specification and the requirement clearly
- More effective and readable code
- Quick detection of regressions
- Easy for refactoring

- Additional code

Pair programming

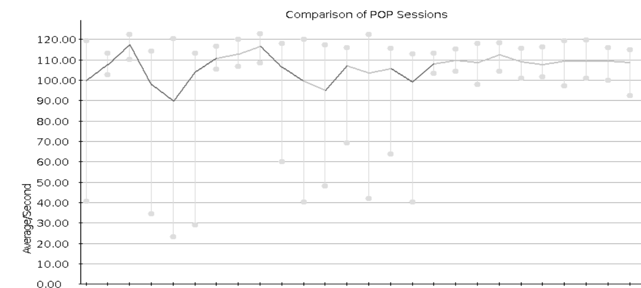
Code review

KERIO Connect. Communicate. Collaborate. Securely.

Other techniques 2/2

Performance tests

- Maximum firewall’s throughput
- HW requirements for X clients
- Synchronization time
- Slamd, Avalanche - with night builds



KERIO Connect. Communicate. Collaborate. Securely.

Testing in stabilization phase

KERIO Connect. Communicate. Collaborate. Securely.

Stabilization phase

All features have been coded in this phase

- Bug fixing
- Optimizations, improvements

Restricted commits to repository

- Repairs only
- Reviewed code only (by developers)

Not really agile

- 1 – 2 months
- Main goal is to cut down this time as much as possible

Testing in stabilization phase 1/2

Bugs verification

- All resolved bugs must be verified

Acceptance tests

- The test cases are grouped in test suites
- Tester says yes/no
- Test protocol – accessible to whole company

System tests

- It's not possible to test whole product in all supported platforms
- Installations and basic tests on all supported platforms

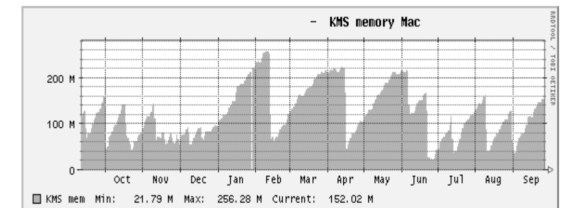
Testing in stabilization phase 2/2

Internal testing

- Our employees use our products from alpha
- Feedback from real environment
- Multilevel structure of real servers

Partners testing

- Betatesting
- Field-Testing



Other QA activities

Release test

- Have we published a new version successfully?
- Web, update checker

Updates monitoring and technology tests

- New versions of the operating systems, browsers, email clients, anti-viruses, …
- Test of platform-dependant features (installation, ssl, authentication, …)

Supportive systems

Supportive systems 1/2

KMS 6.4.0 - Test detail

Product:	Kerio MailServer 6.4.0	Overall:	99% - 217 of 220
Component:	WM, WMM	Original Time Estimate:	33:00
System:	Safari 2	Time Actually Taken:	30:10
Created:	2007-02-19 08:13:31 by Zdeněk Samuel	Time Left:	0:30

Test	Result	Bug		Build	
WM - E-mail searching	✗✓				
WM - Quick search	✓			2847	
WM - Basic search	✓			2847	
WM - Advanced search	✗✓	~~23059~~		2847	
WM - Basic actions on contacts	✗✓				
WM - New contact	✓			2847	
WM - Categories	✓			2847	
WM - Edit Contact	✗✓	~~23057~~		2847	
WM - Private	✓			2847	
WM - Mass actions on contacts	✓				
WM - Send mail to contact	✓			2847	
WM - Forward	✓			2847	
WM - Delete	✓			2847	

Supportive systems 2/2

AutoBuild system

- Regularly builds new versions from source code
- Developer knows if his commit contains any fatal error
- QA has a build with all repairs
- Unsuccessful build = urgent priority to fix it
- Vitally important system

AutoTest

AutoTests in general 1/2

What does it mean “AutoTest”?

- Tool which helps us to test more and better
- Script simulates tester’s work

Why we need AT?

- Many supported environments (operating systems, browsers, MS Outlooks)
- Cumulative products
- => impossible to test all features with every release

Unsuccessful test = urgent priority to fix it

AutoTests in general 2/2

Advantages

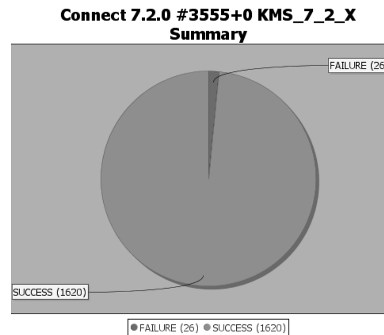
- Quickness
- Reduces time for testing
- Testing can be more effective
- Work exactly, well and they can work still around
- The best tool for regression tests

Disadvantages

- Scripts creating and maintenance => bugs in code, increases costs
- Can’t test everything

Kerio AutoTest System

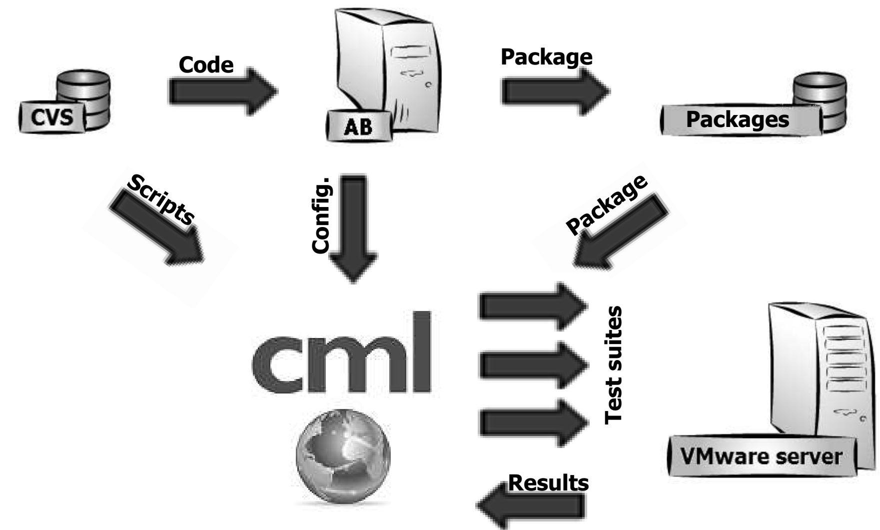
- Java application
- Virtualization
- Multiplatform
- Extendable
- Tests planning
- Statistics / trends
- …



Test	3595+0	3587+0	3578+0	3568+0	3555+0	3541+0	3526+0	3517+0	3508+0
Copy Files	●	●	●	●	●	●	●	●	●
Install	●	●	●	●	●	●	●	●	●
Check	●	●	●	●	●	●	●	●	●
SetDebugLog	●	●	●	●	●	●	●	●	●
CheckIcons	●	●	●	●	●	●	●	●	●

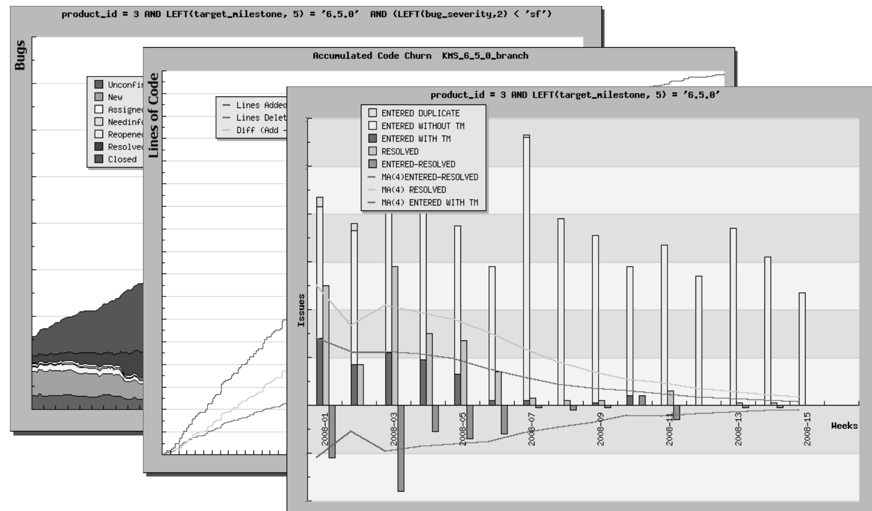
KERIO
Connect. Communicate. Collaborate. Securely.

AutoBuild/Test framework



KERIO
Connect. Communicate. Collaborate. Securely.

Quality measurement



KERIO
Connect. Communicate. Collaborate. Securely.

Results