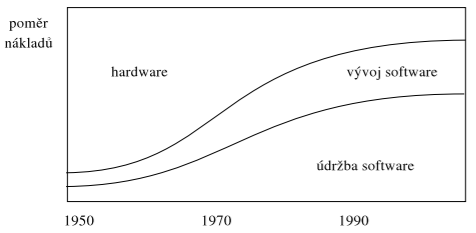
# Softwarové inženýrství

* Systematický přístup k vývoji, nasazení a údržbě softwaru
* Inženýrská disciplína zabývající se praktickými problémy vývoje rozsáhlých softwarových systémů
* Proč vytváříme software?
  + **Zlepšení služeb** – informační systémy
  + **Snížení nákladů** – řízení výroby
  + **Nemožnost řešení bez použití počítačů** – předpověď počasí
* Je nutné zlepšovat vlastnosti SW, hlavně jeho spolehlivost, bezpečnost a použitelnost
* Je potřeba zvyšovat produktivitu vývoje SW

## Počátek SW inženýrství

* Počátek – 60. léta 20. století
* Problémy při vývoji větších programů
* Zavedení pojmů softwarové inženýrství a softwarová krize na konferencích v letech 1968-1969
* SW krize se projevovala (a stále projevuje)
  + Neúnosným prodlužováním a prodražováním projektů
  + Nízkou kvalitou výsledných produktů
  + Problematickou údržbou a inovacemi
  + Špatnou produktivitou práce programátorů
  + Řada projektů končila neúspěchem
* První krok k metodickému přístupu k programování – strukturované programování
* Průměrný SW projekt v porovnání s původním plánem **stál o 89 % více, trval 2,22 krát déle** a poskytuje pouze **61 % funkčnosti**. => Průměrný projekt byl tedy téměř 7 krát horší, než se původně plánovalo!

## Hlavní cíle SW inženýrství

### Management projektu

* Řízení životního cyklu projektu
* Dosažení požadovaného výsledku v požadovaném čase
* => efektivní práce s časem a tedy i s náklady

### Techniky

* Analýzy
* Návrhu
* Programování
* Testování

### Vlastnosti SW inženýra

* Základní báze znalostí
* Schopnost aplikovat znalosti
* Schopnost vyhledávat nové informace a osvojit si nové znalosti

## Problémy při vývoji softwaru

Podstatné, vnitřní, nevyhnutelné problémy:

### Složitost

* Žádné dvě části nejsou stejné
* Složitost je zdrojem dalších problémů jako např. komunikace v týmech
* Je náročné pochopit všechny možné stavy systému
* Problémy s úpravami a rozšířeními

### Přizpůsobivost

* Když se něco změní, měl by se přizpůsobit software a ne naopak

### Nestálost

* Mění se okolí a mění se i software (nejde o nahrazení novým)
* Přibývají požadavky na úspěšně používaný software
* Software přežívá hardwarové prostředky

### Neviditelnost

* Neexistuje přijatelný způsob reprezentace softwarového výrobku, který by pokryl všechny aspekty
* Dokonce ani nejsme schopni určit, co v dané reprezentaci chybí
* Syndrom 90 % hotovo: při posuzování hotové části se nevychází z hotového, ale z odpracovaného (např. podle plánu)

Problémy, které se nemusí projevit vždy:

### Práce v týmu

* Problémy s organizací práce na velkých softwarových projektech
* Problémy s plánovaním procesu tvorby softwaru
* Komunikační problémy jsou jedním z hlavních zdrojů chyb v programech
* Extrémní odchylky v produktivitě mezi jednotlivými programátory, až 1:20

### Nízká znovupoužitelnost při tvorbě softwaru

* V procesu tvorby softwaru je málo standardů a většinou se software tvoří od začátku. S každým programem se vymýšlí už vymyšlené.
* Málo produktů se sestavuje z už existujících součástí.

### Problém míry

* Metody použitelné na řešení malých problémů se nedají přizpůsobit na řešení velkých (složitých) problémů.

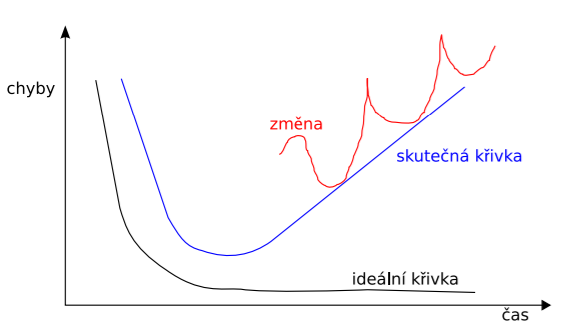
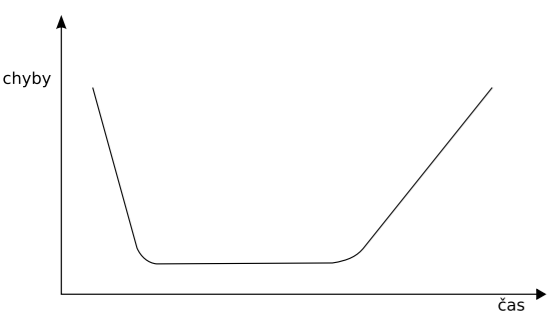
### Tvorba dokumentace

* Tvorba dokumentace je podobná tvorbě vlastního programu.
* Enormní rozsah dokumentace co do kvantity i rozmanitosti
* Problémy s udržováním aktuálnosti dokumentace vzhledem ke změnám softwaru
* Problémy s konzistencí a úplností dokumentace

### Náchylnost softwaru k chybám

* Hodně chyb se projeví až při provozu (a ne při vývoji)
* Odstraňování chyb vede k návratu v etapách vývoje softwaru

### Způsob stárnutí softwaru

* Software se fyzicky neopotřebuje. ALE: Přidávání nových funkcí ve spojení s častými opravami chyb vede k postupné degradaci struktury a k snižování spolehlivosti softwarových systémů.

*Obrázek SEQ Obrázek \\* ARABIC 1 - Typická chybová křivka hardwaru*

*Obrázek SEQ Obrázek \\* ARABIC 2 - Typická chybová křivka softwaru*

### Specifikace požadavků

* Problematická komunikace s uživatelem
* Nejasná a neúplná formulace požadavků spojená s neucelenou představou uživatele o výsledném softwarovém systému
* Nejednoznačnost spojená s častou specifikací požadavků v přirozeném jazyce

Tvorba softwaru je tvůrčí proces, software nelze vyrábět.

## Příčiny zastavení softwarových projektů

* Neúplnost nebo nejasnost požadavků
* Nedostatek zájmu a podpory ze strany uživatele
* Nedostatek zdrojů, tj. podhodnocený rozpočet a krátké termíny
* Nerealistické očekávání
* Malá podpora od vedení dodavatele nebo odběratele
* Změna požadavků a specifikace
* Nedostatečné plánování
* Vyvíjený systém už není potřeba

## Pár postřehů Freda Brookse

* Přidáním dalších pracovníků do zpožděného projektu se tento projekt ještě více zpozdí
* Napsání překladače Algolu zabere 6 měsíců nezávisle na tom, kolik ho vytváří programátorů
* Efekt (syndrom druhého systému – při návrhu druhé verze systému hrozí rizika:
  + Příliš složitý a neefektivní systém
  + Nepoužití nových technologií

## Rozvoj SW inženýrství

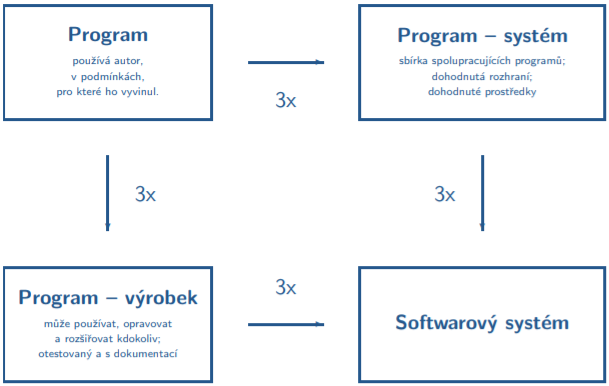
* Výzkum programovacích praktik
  + Uvědomění si lidského faktoru, práce v týmu
  + Podpora řízení tvorby SW
  + Modulární programování
  + Návrhové vzory
* Výzkum metodik
  + Vnímání životního cyklu vývoje SW
  + Strukturované metody, datově a procesně orientované metody, OO metody, agilní metodiky
  + Výzkum modelovacích jazyků (dnes UML)
* Zabezpečení kvality
  + Systematické testování, formální ověřování
* Metody návrhu založené na modelech
  + Transformace modelů do programu

## Metodiky vývoje softwaru

* Disciplinovaný proces nad vývojem softwaru s cílem zajistit tento vývoj více predikovatelný a efektivnější
* Věnují se různým aspektům, které ovlivňují vývoj softwarového produktu, včetně samotného procesu tvorby softwaru
* Zahrnují proces vývoje, nástroje, způsoby využití, plánování

Pozor na terminologii!

* Metoda – postup pro dosažení určitého cíle
* Metodika – souhrn doporučených praktik a postupů
* Metodologie – nauka o metodách, jejich tvorbě a použití
* Metodika vývoje softwaru = Software Development Methodology

Softwarový produkt

### Program

* funkční část produktu

### Softwarový produkt

* sbírka počítačových programů, procedur, pravidel a s nimi spojená dokumentace
* zahrnuje např.: požadavky, specifikace, popisy návrhu, zdrojové texty, testovací data, příručky

### Aktéři ve vývoji softwarového produktu (softwaru)

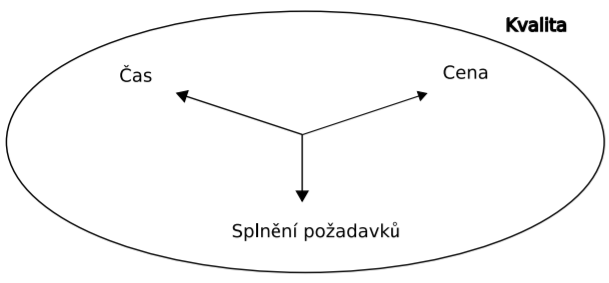
* Zákazník – sponzoruje vývoj SW, specifikuje požadavky na SW
* Dodavatel – vyvíjí systém, má závazky vůči zákazníkovi, komunikuje s uživatelem (testování,.)
* Uživatel – testuje a používá systém, upřesňuje požadavky na SW

### Typy softwarových produktů

#### Generické

* Software se prodává libovolnému zájemci (krabicový software).
* Musí být velice důkladně otestován, protože opravy chyb jsou vzhledem k velkému rozšíření drahé.

#### Zákaznické (na objednávku)

* Software se vytváří na základě požadavků pro konkrétního zákazníka.
* Většinou pro specializované aplikace, pro které vhodný generický software neexistuje.
* Cena zákaznického softwaru je výrazně vyšší.
* Dvě možnosti jeho tvorby:
  + Zadáním zakázky SW firmě
  + V rámci vlastní firmy

*Obrázek SEQ Obrázek \\* ARABIC 3 - Kvalita SW produktů*

## Proces vývoje softwaru

Je proces, ve kterém

* Se potřeby uživatele transformují na požadavky na SW
* Požadavky na SW se transformují na návrh
* Návrh se implementuje
* Implementace se otestuje
* A nakonec předá uživateli.

SW proces definuje *kdo*, dělá *co* a *kdy* => *jak* dosáhnout požadovaného cíle.

## Životní cyklus softwaru

### Životní cyklus

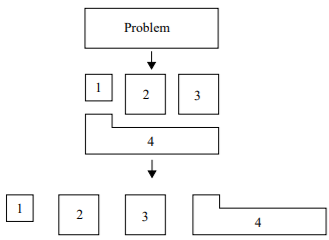
* Rozděluje proces vývoje softwaru na za sebou jdoucí období
* Pro každé období stanovuje cíl
* Období = etapa životního cyklu softwaru

### Činnosti spojené s vývojem softwaru

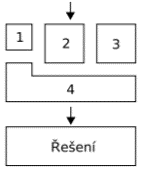
* Analýza a specifikace požadavků (8 %)
* Architektonický a podrobný návrh (7 %)
* Implementace (12 %)
* Integrace a testování (6 %)
* Provoz a údržba (67 %)

Úsilí věnované pečlivé analýze a návrhu se vrátí úsporou nákladů později.

## Dekompozice složitých problémů

* Rozdělení (dekompozice) složitějšího problému na jednodušší (lehčí zvládnutí problému)
* Rozhraní podsystémů

### Dekompozice přináší:

* Lépe zvládnutelné podsystémy
* Soustředění pozornosti na jeden podsystém
* Prezentovatelnost dílčího problému bez rušivých vlivů
* Podsystémy se mohou vyvíjet nezávisle
* Skutečně velké systémy se bez dekompozice nedají zvládnout

### Zvýšená pozornost

* Koordinace tvorby rozhraní
* Integrace a testování podsystémů

# Etapy životního cyklu softwaru

## Analýza a specifikace požadavků

* Získávání, analýza, definování a specifikace požadavků => transformace neformálních požadavků uživatele do strukturovaného popisu požadavků
* Cílem je identifikovat požadavky uživatele, ne návrh, jak je realizovat
* Provedení studie vhodnosti, identifikace a analýza rizik
* Plánování akceptačního testování

## Architektonický návrh

* Ujasnění koncepce systému
* Dekompozice systému
* Definování vztahů mezi částmi systému
* Specifikace funkcionality a ohraničení podsystémů
* Plánování testování systému
* Plánování nasazení systému do provozu, dohoda o postupu nasazování podsystémů, dohoda o plánu zaškolování uživatelů

## Podrobný návrh

* Podrobná specifikace softwarových součástí
* Specifikace algoritmů realizujících požadované funkce
* Specifikace rozhraní pro jednotlivé součásti
* Specifikace logické a fyzické struktury údajů, které zpracovává příslušná součást
* Specifikace způsobu ošetřování chybových a neočekávaných stavů
* Plán prací při implementaci součásti
* Plán testování součásti, návrh testovacích dat
* Specifikace požadavků na lidské zdroje (odhad trvání a nákladů projektu)

## Implementace a testování součástí

* Programová realizace softwarových součástí
* Vypracování dokumentace k součástem
* Testování implementovaných součástí
* Začátek školení budoucích uživatelů

## Integrace a testování systému

* Spojení součástí do podsystémů
* Testování podsystémů
* Integrace podsystémů do celého systému
* Testování podsystémů a celého systému
* Oprava nalezených chyb, návraty k etapě implementace

## Akceptační testování a instalace

* Testování systému uživatelem
* Operace přebírání SW produktu
* Školení používání systému, nasazení systému

## Provoz a údržba

* Zabezpečení provozu softwaru
* Řešení problémů s nasazením softwaru
* Řešení problémů s používáním softwaru
* Opravy, rozšiřování, přizpůsobování softwaru podle požadavků okolí

# Model životního cyklu softwaru

* Definuje etapy vývoje softwaru a jejich časovou následnost
* Pro každou etapu definuje nutné činnosti
* Pro každou etapu definuje její vstupy a výstupy
* Nedefinuje délku trvání kroků a jejich rozsah
* Každá etapa vytváří reálné výstupy
* Správnost každé etapy lze vyhodnotit

Rozdíly v modelech jsou zejména v definování etap a jejich posloupnosti.

## Vodopádový model životního cyklu softwaru

* Životní cyklus jde postupně od první etapy až do poslední
* Následující etapa začne až po ukončení předcházející
* Možnost návratu k předchozí etapě

### Vlastnosti

* Lineární (sekvenční) model, intenzivně používán v 70. letech
* Cílem bylo zavést do vývoje řád umožňující řešit náročnější problémy
* Dekompozice, kontrola výstupů etap => snížení počtu chyb
* Uživatel se účastní pouze při definování požadavků a zavádění

### Výhody

* Jednoduché na řízení
* Při stálých požadavcích: nejlepší struktura výsledného produktu

### Nevýhody

* Zákazník není schopen předem stanovit (přesně!) všechny požadavky
* Při změnách požadavků dlouhá doba realizace
* Zákazník vidí spustitelnou verzi až v závěrečných fázích projektu => odhalení nedostatků ve specifikaci požadavků příliš pozdě (validace)

# Přístupy k analýze a návrhu

Základní přístupy k analýze a návrh:

* Strukturovaný – systém je chápán jako kolekce funkcí (procesů) operujících nad daty
* Objektově orientovaný – systém je chápán jako kolekce vzájemně komunikujících objektů

## Strukturovaný přístup k analýze a návrhu

### Konceptuální model

* Vyjadřuje podstatu systému
* Říká, co má systém dělat
* Obsahuje sémantický model dat
* Vymezuje, co budeme sledovat, ne jak to budeme realizovat

### Logický model

* Definuje, jak bude konceptuální struktura dat implementována
* Modely: lineární, síťový, relační, objektově orientovaný..

### Fyzický model

* Model fyzického uspořádání dat (soubory..)

## Základní konceptuální modely

### Funkční (procesní) modelování

* Základní model strukturované analýzy
* Ukazuje funkce systému, toky dat mezi systémem a okolím a mezi funkcemi, data ukládaná v systému
* Diagram datových toků (Data Flow Diagram – DFD)

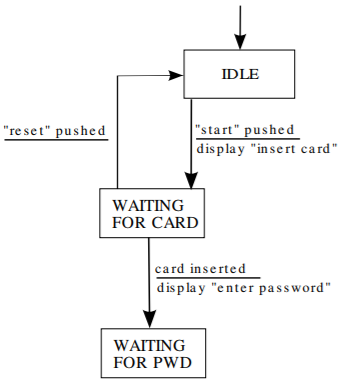
### Minispecifikace

* Popis funkcí (procesů) – co dělají

### Datové modelování

* Ukazuje entity aplikační domény zpracovávané systémem a statické vztahy mezi nimi (typicky perzistentní data ukládaná v databázi)
* Důležitý model datově intenzivních aplikací
* Zásadní význam pro návrh databáze
* Diagram entit a vztahů (Entity Relationship Diagram – ERD)

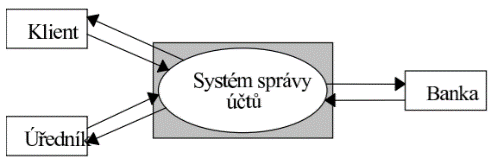
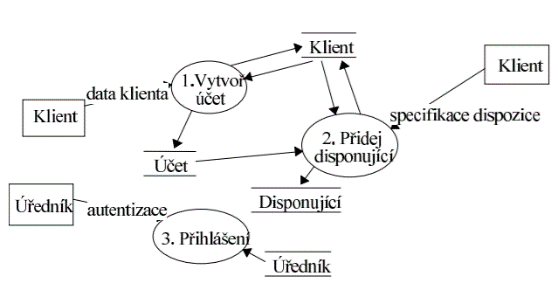
### Datový slovník

* Obsahuje specifikace prvků modelů
* Notace pro specifikaci informačního obsahu prvků DFD a ERD

### Stavový diagram (State – Transition Diagram – STD)

* Modeluje dynamické chování systému nebo jeho částí
  + Stavy – zachycují určitou situaci (počáteční, koncové)
  + Přechody – změny stavů
  + Podmínky – externí události ovlivňující proveditelnost přechodů
  + Akce – události jako komunikace, výpočet
* Teoretický koncept: konečný automat
* UML: stavový diagram

### Data Flow Diagram (DFD)

* Je technika používaná při strukturované analýze a návrhu pro specifikaci chování systému
* Je hierarchický model, který ukazuje funkce systému, toky dat mezi systémem a okolím a toky dat mezi funkcemi a datovými sklady
* Je tedy blíže návrhu
* Je doplněn minispecifikacemi

### Datové modelování

Cíle návrhu datově intenzivních systémů

* Mít v systému všechna potřebná data
* Nemít v systému žádná nepotřebná data
* Vyjádřit vztahy mezi daty
* Popsat transformaci dat v systému
* ER diagram

# Diagramy

Diagramy struktury – diagram tříd, diagram objektů, diagram seskupení

Diagramy chování – diagram případů užití, stavový diagram, diagram aktivit

Diagramy interakce – sekvenční diagram, diagram komunikace

# Diagramy struktury

## Diagram tříd

* Zobrazuje třídy a statické vztahy mezi nimi

### Vztahy mezi třídami

* Zobecnění (generalization)
* Asociace (association)
* Závislost (dependency)
* Realizace (realization)

### Zobecnění – dědičnost

* Vztah genegeralizace/specializace mezi třídami
* Odvozená třída sdílí atributy, chování, vztahy a omezení obecnější třídy
* Odvozená třída může přidávat a modifikovat atributy a chování

#### Přepisování (overriding)

* **změna definice operace (metody) definované v třídě T v odvozené třídě**
* **aby mohl potomek přepsat (předefinovat) operaci, musí mít operace shodnou signaturu**
* **např. třída Tvar a operace kreslit a obsah**

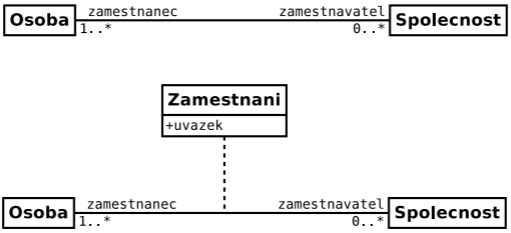
#### Polymorfismus

* **stejné operace (stejná signatura) s rozdílnou implementací**
* **např. kreslit, obsah**

### Asociace

* asociace slouží k zachycení vztahů mezi třídami (jejich instancemi)

Vlastnosti asociace:

* Objekt má ve vztahu svou roli
* Asociace má své násobnosti (mohutnosti)
  + Násobnost je odrazem cíle modelu
  + Bez této znalosti nelze určit špatnou/dobrou násobnost
* Asociace má svůj název
  + Název může být sloveso nebo podstatné jméno
  + V případě slovesa se často označuje směr vazby
* Vyjadřuje proměnlivý vztah mezi objekty (instancemi tříd)
  + Každé spojení váže instanci jedné třídy s instancí druhé třídy
  + Počet spojení se v čase může měnit
  + V OO návrhu lze asociaci povýšit na třídu (asociační třída)

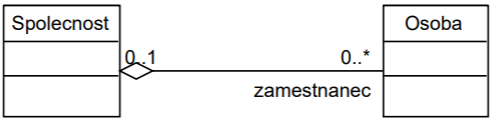
#### Asociační třída

* Přiřazení atributů asociaci
* Asociace Zamestnani, atribut uvazek

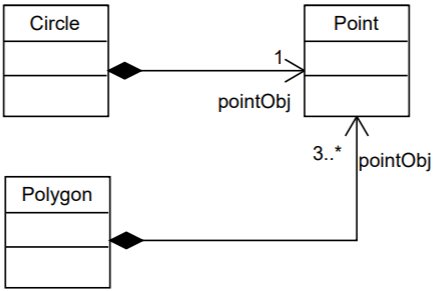
#### Asociace vyššího stupně

* Binární asociace (vztah dvou tříd, resp. jejich instancí)
* N-ární asociace (vztah více tříd, resp. jejich instancí)
  + Jsou méně časté
  + Většinou se dají převést na binární asociace
  + Pokud ne, bývá nutné povýšit asociaci na třídu

### Asociace celek/část – Agregace

* Celek je seskupen z více částí
* Celek = agregační (seskupený) objekt
* Část celku = konstituční (tvořící) objekt

Vlastnosti agregace:

* Seskupený objekt může existovat bez svých konstitučních objektů
* Konstituent (konstituční objekt) může být součástí více seskupení
* Implicitní násobnost se nedá předpokládat
* Asociace agregace nemívá název (vyjadřuje vztah má)
* Agregace bývají homeometrické (tj. konstituenti patří do téže třídy)

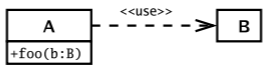
### Asociace celek/část – Kompozice

* Celek je složen z více částí
* Celek = kompozitní (složený) objekt
* Část celku = komponentní (složkový) objekt

Vlastnosti kompozice:

* Složený objekt neexistuje bez svých komponent
* Komponenta (komponentní objekt) může být součástí pouze jedné kompozice
* Implicitní násobnost každé složky je 1
* Asociace kompozice nemívá název
* Kompozice bývají heterometrické (tj. komponenty patří do různých tříd)

### Závislost (dependency)

* Vyjadřuje jiné různé vztahy mezi objekty či třídami
* Typ závislosti se označuje pomocí stereotypů

Nejběžnější typ stereotypu – používání <<use>>

* A (klient) -> B (dodavatel)
  + Metoda třídy A potřebuje argument třídy B
  + Metoda třídy A vrací hodnotu třídy B
  + Metoda třídy A používá objekt třídy B, ale ne jako atribut
* Závislost bez uvedeného stereotypu se považuje za používání

#### Typy závislostí (stereotypy)

«instantiate» / «create» - klient vytváří instance dodavatele

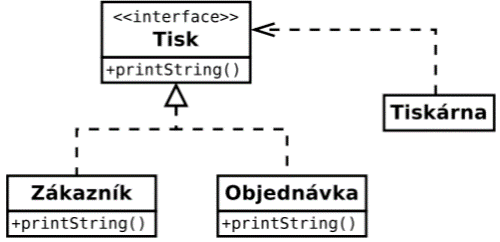
«trace» - klient realizuje dodavatele, vazba mezi elementem v různých modelech

«refine» - klientská třída poskytuje detailnější informace než dodavatel

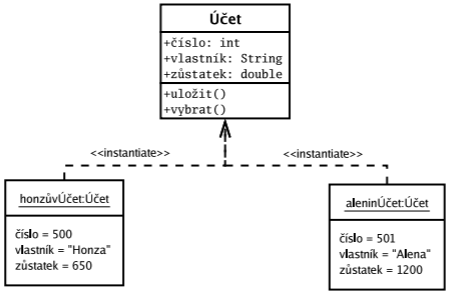
«send» - operace klienta zasílá signál příjemci

«call» - klientská třída volá operaci dodavatele

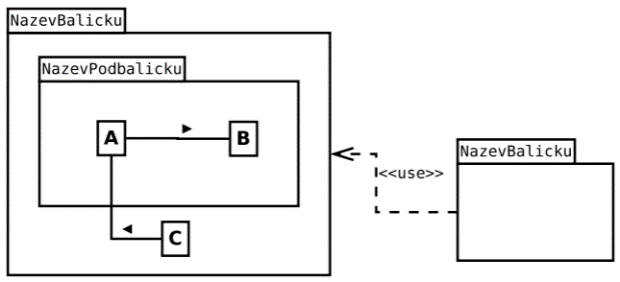
### Realizace

* Vztah mezi třídou a rozhraním
* Třída implementuje všechny operace (metody) z daného rozhraní
* Objekt používající rozhraní pak umí používat i jeho implementační třídy
* Omezuje počet vazeb mezi třídami

## Diagram objektů

* Je úzce svázán s diagramem tříd
* Znázorňují objekty a jejich relace v určitém čase
* Relace jsou dynamické (nemusí trvat po celou dobu existence objektů)
* Mezi objekty existuje spojení (spojení = instance vztahu asociace)

## Diagram seskupení

* Seskupení sémanticky souvisejících elementů
* Definuje sémantické hranice modelu
* Umožňují souběžnou práci v etapě návrhu
* Poskytují zapouzdření prostoru jmen

Balíčky mohou obsahovat

* Případy užití
* Analytické třídy
* Realizace případů užití
* Další balíčky

# Diagramy chování

## Diagram případů užití

#### Use-case-driven approach (např. metodika RUP)

* Klíčovými aktivitami při specifikaci požadavků je nalezení účastníků a nalezení případů užití
* K tomu se využívají diagramy případů užití doplněné o detaily (specifikace) případů užití

### Prvky diagramu užití

* Hranice systému
* Účastník (aktor) – subjekt, který se systémem pracuje (může mít i speciální podobu, např. čas nebo jiný systém)
* Případ užití – funkce, kterou systém vykonává jménem jednotlivých účastníků nebo v jejich prospěch
* Interakce – ukazuje účast aktora na provádění případu užití

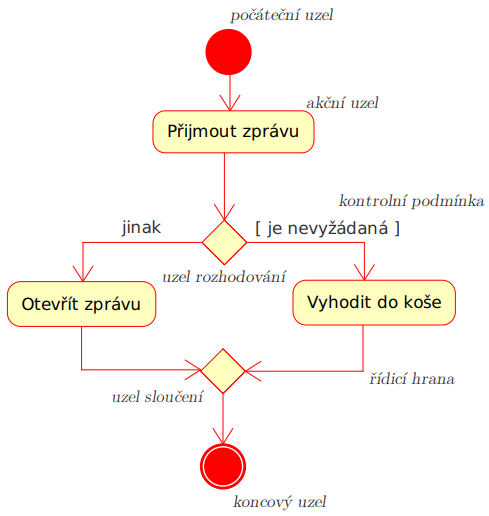
### Detail (specifikace) případu užití

* Konkretizace (specifikace) případu užití
* Neexistuje standard, většinou se však využívá tabulka
* Případ užití má svůj:
  + Název
  + Jedinečný identifikátor
  + Specifikaci
* Specifikace případu užití má:
  + Vstupní podmínky
  + Tok událostí
  + Následné podmínky
  + Alternativní toky

### Pokročilé techniky modelování případu užití

* Zobecnění účastníka
* Zobecnění případu užití
* Relace <<include>>
* Relace <<extend>>
* Pokročilé techniky používejte pouze pro zjednodušení modelu
* Případ užití je způsob zápisu požadavků. Musí být tedy čitelný i pro uživatele. Je proto žádoucí, aby byl co nejjednodušší.
* Uživatelé těžko chápou zobecnění účastníka.
* Význam relace <<extend>> často nechápou ani analytici či návrháři.
* Závěr: pokročilé techniky používejte co nejméně!

## Diagram aktivit (Activity Diagram)

* Reprezentují objektově orientované vývojové diagramy
* Modelování obchodních procesů, účastní se více objektů
* Lze je připojit k libovolnému modelovanému elementu
  + Případ užití, třída, komponenty

### Využití diagramu aktivit

* Modelování scénářů případů užití
* Modelování detailů operace nebo algoritmů
* Modelování obchodního procesu

### Prvky diagramu

* Uzly
  + Akční uzly: modelují aktivitu
  + Řídící uzly: modelují rozhodování, počáteční uzel, koncový uzel
  + Objektové uzly: modelují objekty podílející se na aktivitách
* Hrany
  + Řídící hrany: modelují přechody mezi uzly
  + Objektové hrany: modelují cesty objektů mezi uzly

### Možnosti modelování

* Tok událostí (včetně toku dat)
* Rozhodování
* Větvení a spojení
* Iterace
* Paralelní toky

## Stavový diagram

* Modelování životního cyklu jednoho reaktivního objektu
* Vycházejí ze stavového automatu (Harel)
* Mohou modelovat dynamické chování těchto reaktivních objektů
  + Třídy, resp. Instancí tříd (nejčastější), případy užití, podsystémy, systémy

### Reaktivní objekt

* Reaguje na vnější události
* Životní cyklus je modelován jako řada stavů, přechodů a událostí volání a změn
* Chování je důsledkem předchozího chování (následný stav závisí na aktuálním stavu)

# Diagramy interakce

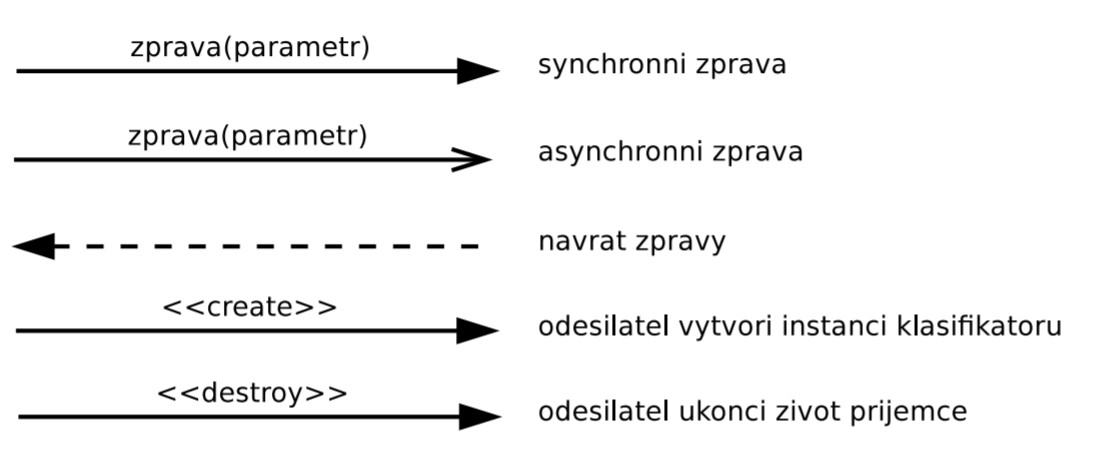
* Popisují spolupráci objektů
* Typicky modelují chování jednoho případu užití

**Čára života**

* Zastupuje jednoho účastníka interakce (objekt)

**Zprávy**

* Komunikace mezi účastníky interakce



## Sekvenční diagram (Sequence Diagram)

* Zdůrazňují časově orientovanou posloupnost předávání zpráv mezi objekty (chronologie zasílání zpráv)
* Bývají přehlednější a srozumitelnější než diagramy komunikace
* Každá čára života (objekt) je zobrazena s časovou osou

## Diagram komunikace (Communication Diagram)

* Zdůrazňují strukturální vztahy mezi objekty
* Výhodné pro rychlé zobrazení komunikace mezi objekty
* Objekty jsou spojeny linkami (komunikačními kanály)
* Zprávy jsou řazeny podle hierarchického číslování

# UML v etapách vývoje softwaru

## Tvorba analytických modelů

* Zaměřuje se na otázku ***co***, otázka ***jak*** je úkolem návrhu
* Zobrazuje důležité věci (objekty, vztahy) z problémové domény
  + Třídy Zákazník, Košík..
  + Ale třída pro přístup k databázím patří do návrhu (netýká se problémové domény)
* Analytické třídy
* Use Case diagramy

**Specifikace případů užití**

* + Diagramy aktivit
  + Stavové diagramy

**Realizace případů užití**

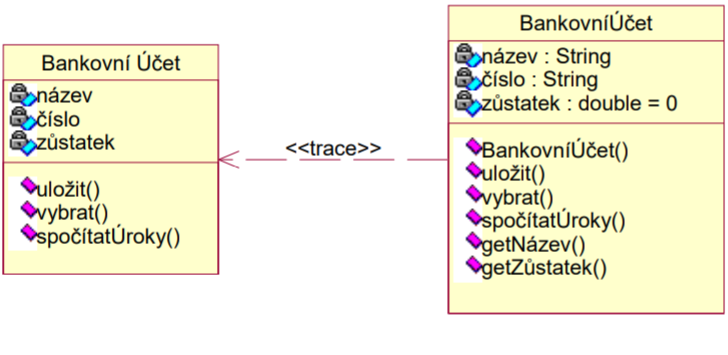
* + Modelují interakce analytických tříd
  + Identifikují zasílané zprávy mezi objekty (instancemi tříd)

## Tvorba návrhových modelů

* Vychází z výstupů etapy analýzy
* Zaměřuje se na otázku jak, věnuje se detailům
* Specifikace modelů je na takové úrovni, že je lze přímo implementovat
* Upřesňování analytických diagramů
  + Návrhové třídy
  + Realizace případů užití

## Návrhové třídy

* Specifikace návrhových tříd je na takovém stupni, že je lze přímo implementovat
* Upřesňování analytických tříd
* Využití tříd z doménového řešení, dále knihovny, vrstva aplikačního serveru, GUI..



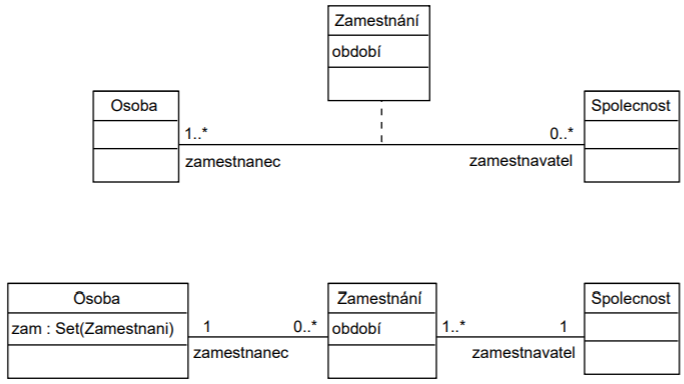
Vlevo analytická třída, vpravo návrhová třída.

* Návrhový model interakce obsahuje třídy i jako SprávceDB nebo GUI oproti analytickému modelu interakce (viz prezentaci 5, slide 19,20)

## Upřesnění analytických relací

### Asociace

* Agregace vs kompozice – upřesnění vztahu celek/část
* Asociace povýšená na třídu – návrh asociačních tříd
* Asociace typu 1:N – realizace (nejčastěji kolekce)



### Omezení (Constrains)

* Definují omezující podmínky
* Rozšiřují sémantiku elementu (např. OCL)

### Stereotypy (Stereotypes)

* Definují nový element na základě stávajícího elementu
* Stereotyp má svou sémantiku

### Označené hodnoty (tagged values)

* {tag1 = hodnota1, tag2 = hodnota2}
* Většinou se přidružují k stereotypu, vyjadřují vlastnosti nových elementů

## Omezení a dotazy nad modely

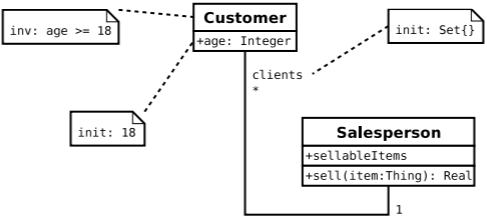
Modelovací techniky UML

* Nedokáží zachytit všechny závislosti mezi elementy graficky
* Řeší se poznámkou se slovním spojením
* Slovní popis je nedostatečný
  + Není vždy jednoznačný, může být různě pochopen
  + Komplikuje automatické konverze



* Jednoznačný jazyk => **OCL**

# Object Constraint Language

* Speciální formální jazyk pro UML (deklarativní, nikoliv programovací)
* Původně vytvořen jako obchodní modelovací jazyk v IBM roku 1995
* Součástí OMG standardů pro UML
* Definuje omezení, podmínky a dotazy nad UML modely => zpřesňování modelů
* Je formální deklarativní jazyk navržený pro návrháře => nevyžaduje se silný matematický základ
* Spojený s dalšími metamodely definovanými OMG
* Umožňuje transformace modelů

## Využití OCL

* Specifikace podmínek pro vykonání metod
* Specifikace invariantů tříd
* Specifikace počátečních hodnot atributů
* Specifikace těla operace
* Specifikace omezení

### Typy omezení

* Invariant – podmínka, která musí být vždy splněna všemi instancemi
* Precondition – omezení, které musí být pravdivé před provedením operace
* Postcondition – omezení, které musí být pravdivé těsně po ukončení operace
* Guard – omezení, které musí být pravdivé před provedením přechodu mezi stavy

### Základní knihovna

* Typy: Boolean, Integer, Real, String
* Kolekce: Collection, Set, Ordered Set, Bag, Sequence
* Operace: and, or, size, includes, count

# Objektově orientované modelování v UML

## Jazyk UML

* Unified Modelling Language
* Inspirován existujícími analytickými jazyky a modely
* Základní modelovací jazyk metodiky RUP

## Stavební bloky jazyka UML

### Předměty (Things)

* Samostatné prvky modelu
* Např. třída, případ užití, aktéři, stav, poznámka (anotace)

### Vztahy (Relationships)

* Určují vzájemnou souvislost předmětů
* Např. závislost, asociace, agregace, kompozice, zobecnění, realizace

### Diagramy (Diagrams)

* Pohledy na modely UML, kolekce předmětů a vztahů
* **Analytické diagramy** – co bude systém dělat
* **Návrhové diagramy** – jak to bude systém dělat

## Společné mechanismy jazyka UML

### Ornamenty (Adornments)

* Každý prvek modelu má svůj symbol (např. třída), který může být obohacen různými ornamenty (např. atributy, operace)
* Obvykle není potřeba vždy zobrazovat všechny podrobnosti, některé ornamenty mohou být skryty (různé pohledy na systém)

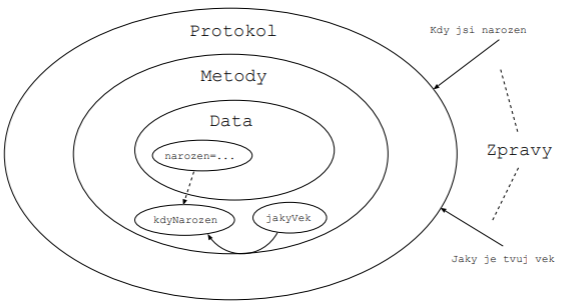
### Mechanismy rozšířitelnosti

* Omezení (constaints)
  + Definují pravidla, která musí být vyhodnocena jako pravdivá
  + Textový řetězec uzavřený do složených závorek {}
  + Jazyk OCL (Object Constraint Language)
* Stereotypy (stereotypes)
  + Definuje nový prvek, který je založen na existujícím prvku
  + Název stereotypu se většinou uzavírá do dvojitých závorek <<název>>
  + Musí se definovat sémantika nového prvku – podpora CASE nástroje, textová dokumentace, metamodel UML…

# Objektově orientované modelování

**Objektově orientovaný přístup** k modelování a vývoji systémů

* Kolekce vzájemně komunikujících objektů
* Soubor objektově orientovaných prostředků (objekty, třídy, UML) a metodik (RUP)
* Vykazuje vyšší stabilitu navrhovaných prvků z pohledu měnících se požadavků
* *Objektový návrh nutně neimplikuje objektovou implementaci!*

**Objekt** reprezentuje entitu riešeného problému

* Má jasně vymezenou roli (zodpovědnost)
* Zná sám sebe (identita)
* Uchovává data (stav)
* Má metody (chování)
* Umí zpracovávat a posílat zprávy (protokol)

## Rozhraní objektu

### Operace vs. Metoda

* Operace reprezentuje abstraktní pohled na chování objektu
* Metoda implementuje operaci
* Signatura operace = název, typ návratové hodnoty, typy všech stejně seřazených argumentů

### Rozhraní objektu

* Množina operací, které určují chování objektů a jeho vztah k jiným objektům
* Pouze definuje, *co* objekt umí (nabízí), nedefinuje *jak*
* Způsob provedení operace závisí na její implementaci (metodě)

## Třídy objektů – vlastnost objektové orientace

* Seskupení objektů do tříd podle podobnosti (typu)
* Třída je
  + generická definice (šablona) pro množinu objektů stejného typu
  + množina objektů se stejným chováním a stejnou množinou atributů
* Objekt (konkrétní jedinec) je instancí třídy

## Vlastnosti objektové orientace

### Abstrakce (Abstraction)

* Vytvářený systém objektů je abstrakcí řešeného problému (zjednodušený pohled na systém bez ztráty jeho významu)
* Analýza problému => klasifikace do abstraktních struktur => objekty / třídy
* Klasifikace je založena na rozpoznávání podobností v řešené problematice

### Zapouzdření (Encapsulation)

* Seskupení souvisejících idejí (data, funkcionalita) do jedné jednotky
* Seskupení operací a atributů do jednoho typu objektu (třídy) – stav je dostupný či modifikovatelný pouze prostřednictvím rozhraní
* Omezení externí viditelnosti informací nebo implementačních detailů

### Polymorfismus (Polymorphism)

* Znalost třídy (objektu), jak provést určitou operaci, která může být obecně společná pro více tříd (objektů)
* Stejné rozhraní může být implementováno různými třídami (objekty)
* Stejná operace s jedním názvem může mít více implementací
* Různé implementace operace ze stejného rozhraní by měly zachovávat stejnou sémantiku

### Dědičnost

* Definuje a vytváří třídy (objekty) na základě již existujících tříd (objektů)
  + Možnost sdílení chování bez nutnosti reimplementace
  + Možnost rozšíření chování
* Mezi třídami (objekty) vzniká hierarchický vztah podle dědičnosti (strom)

## Další pojmy

**Identita a shoda objektů**

* Stav objektu = aktuální hodnoty všech atributů
* Každý objekt je jedinečný bez ohledu na stav a třídu
* Shodnost je vázána na stavy objektů
* Objekty, které nejsou identické, mohou být shodné

**Časná vazba**

* Implementace operace (metoda) je vybrána v době kompilace

**Pozdní vazba (dynamická vazba)**

* Implementace operace (metoda) se vybere za běhu podle skutečně dosazeného objektu
* => viz polymorfismus

# Objektově orientovaný návrh a programování

Znovupoužitelnost?

* Zajištění znovupoužitelnosti => obecný návrh
* Zajištění aplikovatelnosti na řešený problém => specifický návrh

… přesto

* Proč nevyužít řešení, které již fungovalo
* Taková řešení jsou výsledkem mnoha pokusů a používání
* => vzory pro řešení stejných typů problémů (návrhové vzory)

## Principy objektově orientovaného návrhu

### Problémy se špatným návrhem

* Změna v softwaru je náročná a vyžaduje úpravy na mnoha místech
* Změna způsobí problémy v jiných, mnohdy nesouvisejících částech softwaru
* Vyčlenit část softwaru pro znovupoužitelnost je náročnější než tuto část vytvořit znovu

### Principy objektově orientovaného návrhu architektury

* Předkládají vhodné postupy pro návrh architektury
* Vhodné z pohledu údržby a rozšiřitelnosti architektury systému

Single Responsibility Principle (SRP)

Open Closed Principle (OCP)

Dependency Inversion Principle (DIP)

Principy návrhů balíčků (komponent)

Acyclic Dependencies Principle (ADP)

Liskov Substitution Principle (LSP)

Don’t Repeat Yourself (DRY)

### Single Responsibility Principle (SRP)

* Třídy by měly mít jedinou zodpovědnost, jediný důvod ke změně
* Zodpovědnost (responsibility)
  + Závazek nebo povinnost prvku něco dělat nebo něco vědět
  + Akce/znalost může prvek dělat/mít přímo nebo využívat jiné prvky (koordinace činností, agregace dat)
  + Zodpovědnost != metoda, metody jsou implementovány, aby byla splněna zodpovědnost

### Open Closed Principle (OCP)

* Třída by měla být otevřená pro rozšíření ale uzavřena pro modifikace
* Třída by měla být rozšířitelná bez nutnosti modifikace kódu

### Dependency Inversion Principle (DIP)

* Závislost na abstraktním ne na konkrétním
* Závislost by měly směřovat jedním směrem, a to od konkrétního k abstraktnímu
* Závislost by měly směřovat ke společným rozhraním a abstraktním třídám

Důsledky

* Redukce závislosti v kódu
* Abstraktní rozhraní se mění mnohem méně než konkrétní implementace => závislý kód se nemusí měnit tak často
* Snadná možnost nahradit jednu implementaci za jinou

Balíčky

* Třídy a rozhraní řešící komunikaci dáme do jednoho balíčku
* Třídy obsluhující klienta patří do jiného balíčku

### Principy návrhu balíčků (komponent)

Release Reuse Equivalency Principle (REP)

* + Granularita znovupoužitelnosti je shodná s granularitou uvolnění nové verze

Common Closure Principle (CCR)

* + Třídy, které se mění společně, patří k sobě

Common Reuse Principle (CRP)

* + Třídy, které nejsou znovupoužívány společně, by neměly patřit k sobě

Nelze dodržet všechny principy

* REP a CRP usnadňují vývoj s využitím znovupoužitelnosti
* CCP usnadňují práci při údržbě

### Acyclic Dependencies Principle (ADP)

* Závislosti mezi balíky nesmí tvořit cykly
* Minimální počet závislostí mezi balíky => jednodušší údržba a uvolňování nových verzí (menší počet závislých balíků pro testování)
* Závislosti s cykly => velký počet závislých balíků

### Rozhraní versus implementace

* Signatura
  + Deklaruje formální podobu operace (název, typy)
  + Lze zkontrolovat překladačem
* Kontrakt
  + Deklaruje sémantiku operace a její podmínky (preconditions..)
  + Nelze zkontrolovat překladačem
  + Kontraktem se rozumí podmínky definované rozhraním
  + Rozhraní definuje precondition a postcondition
  + Odvozené třídy musí kontrakt dodržet
* Implementace
  + Realizuje operace definované signaturami a kontrakty
  + Implementace by se měla skrývat

Návrh a implementace s využitím rozhraní

* Vymezujte se k rozhraní, nikoliv k implementace
* Nevyužívejte znalosti implementačních detailů, ty se mohou měnit!

Příklady vytvoření nové třídy s využitím dědičnosti

* Přímočaré použití, jednodušší úprava metod
* Těsná vazba s nadřazenou třídou, problém narušení zapouzdření

### Liskov Substitution Principle (LSP)

* Odvozené třídy by měly být zaměnitelné za bázové třídy
* Uživatelé bázové třídy by měli být schopni pokračovat bez chybného chování i při nahrazení odvozenou třídou

### Don’t Repeat Yourself (DRY)

* Neopakujte stejný kód na různých místech
* Problémy s modifikací a udržovatelností
* Opakující se kód => samostatná metoda
* Soubor opakujících se metod => vytvoření obecnější třídy

# Návrhové vzory

* Základní sada řešení důležitých a stále se opakujících návrhů
* Usnadňují znovupoužitelnost
* Umožňují efektivní návrh (výběr vhodných alternativ, dokumentace)

## Návrhový vzor

* Vzor je šablona pro řešení, nikoliv implementace problému!
* Každý vzor popisuje problém, který se neustále vyskytuje, a jádro řešení daného problému
* Umožňuje jedno řešení používat mnohokrát, aniž bychom to dělali dvakrát stejným způsobem

## Prvky návrhového vzoru

* Název – krátký popis (identifikace) návrhového problému
* Problém – popis, kdy se má vzor používat (vysvětlení problému, podmínky pro smysluplné použití vzoru)
* Řešení - popis prvků návrhu, vztahů, povinností a spolupráce
  + Nepopisuje konkrétní návrh, obsahuje abstraktní popis problému a obecné uspořádání prvků pro jeho řešení
* Důsledky – výsledky a kompromisy (vliv na rozšiřitelnost, přenositelnost,…)
  + Důležité pro hodnocení návrhových alternativ – náklady a výhody použití vzoru

## Vzory se mohou týkat

* Tříd – zabývají se vztahy mezi třídami a podtřídami (vztah je fixován)
* Objektů – zabývají se vztahy mezi objekty, jdou dynamičtější

## Základní rozdělení vzorů

* Tvořivý – zabývá se procesem tvorby objektů (Singleton, Abstract Factory)
* Strukturální – zabývá se skladbou tříd či objektů
* Chování – zabývá se způsoby vzájemné interakce mezi objekty nebo třídami (Command)
  + zabývá se způsoby rozdělení povinností mezi objekty nebo třídy

## Jedináček (Singleton)

### Účel

* třída může mít pouze jednu instanci
* tvořivý vzor – objekty

### Motivace

* nutnost mít pouze jednu instanci (např. tiskové fronty)
* při pokusu o vytvoření nové instance se vrátí již existující

### Důsledky

* řízený přístup k jediné instanci
* zdokonalování operací (dědičnost)
* usnadňuje změnu v návrhu (variabilní počet instancí)

## Abstraktní továrna (Abstract Factory)

### Účel

* vytváření příbuzných nebo závislých objektů bez specifikace konkrétní třídy
* tvořivý vzor – objekty

### Motivace

* např. změna vzhledu sady grafických nástrojů

### Důsledky

* izoluje konkrétní třídy – klient pracuje pouze s rozhraním
* usnadňuje výměnu produktových řad (např. změna vzhledu)
* podpora zcela nových produktových řad je obtížnější

## Command

### Účel

* zapouzdření požadavků nebo operací
* vzor chování

### Motivace

* zaslání požadavku na obecné úrovni, aniž známe konkrétní protokol
* podpora undo operací

### Důsledky

* reprezentuje jeden provedený příkaz
* umožňuje uchovávat stav klienta před provedením příkazu

## Observer

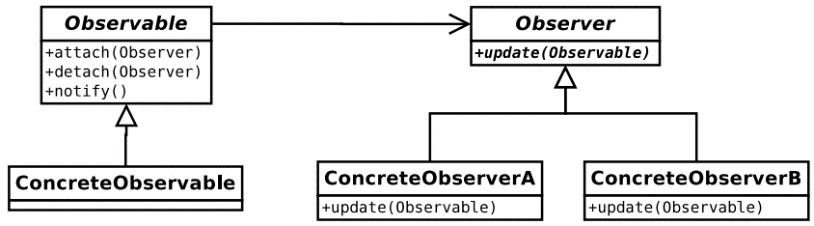
### Účel

* definuje závislost 1 ku N mezi objekty
* vzor chování

### Motivace

* Při změně stavu objektu jsou automaticky informovány všechny závislé objekty

### Důsledky

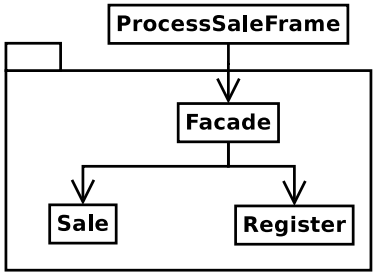
* Konkrétní klient nemusí znát závislé objekty 

## Fasáda (Facade)

### Účel

* Zjednodušení komunikace mezi prvky systému
* Zjednodušení práce se složitějšími požadavky a systémy

### Motivace

* Jednoduché rozhraní ke komplexnímu systému
* Prostředník oddělující implementační třídy a jejich použití (=vrstvy) 

### Důsledky

* Zjednodušené rozhraní
* Možnost výměny vrstvy za fasádou beze změny uživatelských tříd

Příbuzné vzory: Abstraktní továrna (Abstract Factory), Prostředník (Mediator)

# Architektonické vzory

## Návrh architektury

* zaměřuje se na otázku, jak má být sytém organizován
* vytváří se na počátku vývoje, v iterativním vývoji většinou po první iteraci
* spojuje návrh se specifikací požadavků
* identifikuje komponenty, jejich vztahy a komunikaci

## Vztah mezi specifikací a architekturou

* ideálně by se specifikace neměla architekturou zabývat
* nerealistický požadavek
* dekompozice je důležitá pro organizaci specifikace a rozdělení práce na specifikaci požadavků
* dekompozice do komponent či podsystémů je základem abstraktního návrhu architektury

## Architektonické vzory

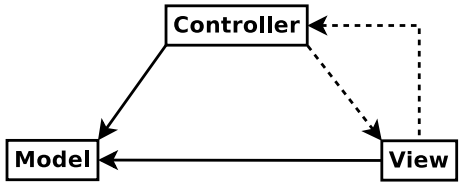
* abstraktní popis dobrých vyzkoušených praktik
* ověřeno na různých systémech a v různých prostředích
* každý vzor by měl obsahovat informace o vhodnosti použití, slabě a silné stránky

### Přehled architektonických vzorů

* Model-View-Controller
* Vrstvená architektura
* Klient-Server

## Model-View-Controller

### Konceptuální model

* Model – zapouzdřuje data a stav aplikace, informuje View o změnách stavu
* View – zobrazuje model, vyžaduje změny modelu, posílá uživatelské události Controlleru
* Controller – zajišťuje změny modelu na základě uživatelských akcí a změny View na základě změny modelu, vybírá Views

#### Konkrétní pohled

* Model – databáze, business logika
* View – dynamické stránky, formuláře
* Controller – zpracování http protokolu, validace dat

### Popis

* Odděluje prezentaci a interakci od systémových dat

### Kdy použít

* Různé způsoby zobrazení a interakce nad stejným modelem
* Budoucí požadavky na zobrazení a interakce nejsou známé

### Výhody

* Data mohou být měněna nezávisle na jejich reprezentaci (pohledu) a naopak
* Podpora prezentace dat různými způsoby

### Nevýhody

* Navýšení režie pro jednoduché modely a interakce

MVC odděluje model a pohled na model. Jak ale zajistit změnu pohledu při změně modelu, pokud model nic neví o pohledu ani kontroleru? => návrhový vzor Observer

### MVC s využitím vzoru Observeru

## Vrstvená architektura

### Koncept

* Rozdělení systémů do vrstev
* Každá vrstva odděluje elementy systému a lze je modifikovat nezávisle
* Přidání či změna vrstvy je možná bez modifikace vrstev nižší úrovně
* Lepší podpora inkrementálního vývoje – vrstvenou architekturu lze snadněji upravovat

### Popis

* Organizace systému do vrstev, každá vrstva má přidělenu svou zodpovědnost (funkcionalitu)
* Vrstva poskytuje služby nadřazené vrstvě, nejnižší vrstva reprezentuje jádro systému

### Kdy použít

* Přidání nových vlastností nad již existujícím systémem
* Vývoj rozdělen do několika týmů, každý se věnuje jedné vrstvě
* Požadavek na víceúrovňovou bezpečnost (security)

### Výhody

* Nahrazení celé vrstvy za novou
* Redundantní služby (autentizace apod.) na každé vrstvě pro zvýšení spolehlivosti

### Nevýhody

* Čisté oddělení vrstev je v praxi náročné
* Vrstva vyšší úrovně může potřebovat komunikovat s vrstvami nižší úrovně přímo, ne jen prostřednictvím bezprostředně navazující vrstvy
* Výkon aplikace – potřeba vícenásobná interpretace požadavků na různých vrstvách může zpomalovat výkon aplikace

## Klient-Server

### Popis

* Funkcionalita je rozdělena do služeb, každá služba (či množina služeb) je poskytována nezávislým serverem
* Klient je uživatel služeb, přistupuje na servery

### Kdy použít

* Data ve sdílené databázi musí být přístupná pro velký počet lokací (konkrétních míst)
* Servery mohou být replikovány – lze využít, pokud je zatížení systému proměnlivé

### Výhody

* Servery mohou být distribuovány na síti
* Služby jsou dostupné všem klientům a nemusí být implementovány všemi uzly

### Nevýhody

* Služba je jeden bod na síti, je náchylnější na útoky typu denial of service
* Výkon aplikace je těžko predikovatelný, závisí na vytížení sítě
* Problémy se správou, pokud jsou servery vlastněny jinou organizací

# Analýza a specifikace požadavků

* Scénář = textová strukturovaná specifikace případu užití

### Modely jazyka UML používané při specifikaci požadavků

#### Modelování dat

* ER Diagram – strukturovaný model dat
* Class Diagram – objektově orientovaný model dat a protokolu (rozhraní, zodpovědnosti)

#### Modelování funkčních požadavků

* Data Flow Diagram (DFD) – specifikace chování systému, strukturovaný přístup
* Diagram případů užití (Use Case Diagram) – specifikuje možnosti použití systému, jazyk UML
* Diagram aktivit (Activity Diagram)
* Stavový diagram (State Diagram)

## Činnosti spojené s vývojem softwaru

* Analýza a specifikace požadavků (8 %)
  + Cíl: Stanovení služeb, které zákazník požaduje od systému, a vymezení podmínek jeho vývoje a provozu.
  + Transformace neformálních požadavků uživatele do strukturovaného popisu požadavků
  + Zdůraznění požadavků uživatele, ne jak toho docílit (realizovat)
  + Provedení studie vhodnosti, identifikace a analýza rizik
  + Získávání, analýza, definování a specifikace požadavků
  + Plánování akceptačního testování
* Architektonický a podrobný návrh (7 %)
* Implementace (12 %)
* Integrace a testování (6 %)
* Provoz a údržba (67 %)

## Zainteresované strany

### Stakeholder

* (původně) dočasný držitel peněz či majetku
* Člověk nebo skupiny lidí, bez jejichž podpory by organizace přestala existovat
* (obecně chápán jako) zainteresované strany v projektu – zákazník, uživatel, analytik, návrhář, tester, manažer

Je důležité:

* Zapojit nejen zákazníka, ale všechny zainteresované strany (stakeholders)
* Tyto strany na začátku identifikovat – pokud analytik v průběhu tvorby požadavků zjistí, že existuje ještě někdo, kdo by se měl k něčemu vyjádřit, zdržuje to průběh projektu
* Pokud se to nezjistí, může být v požadavcích chyba

## Typy požadavků

### Obchodní požadavky (Business Requirements)

* Proč zákazník potřebuje systém => pochopení a definování cílů a smyslu projektu
* Zaměřeno na obchodní cíle (úspora nákladů, času)
* Př.: Úspora nákladů při práci s chemikáliemi

### Uživatelské požadavky (User Requirements)

* Úlohy, které uživatel se systémem provádí => co je možné se systémem dělat, use cases
* Př.: zjistit dostupnost chemikálie na skladě nebo v laboratoři

### Funkční požadavky (Functional Requirements)

* Chování systému v různých podmínkách => co musí být realizováno, aby mohly být vykonány úlohy (user requirements), a tím splněny obchodní požadavky (business requirements), diagram aktivit
* Př.: co vše je potřeba pro zjištění dostupnosti chemikálie

### Nefunkční požadavky (Nonfunctional Requirements)

* Vlastnosti a charakteristiky, které musí systém splňovat, a omezení, která musí respektovat

#### Požadavky na provoz systému

* Statické – např. počet uživatelůd
* Dynamické – např. čas odezvy, počet transakcí na jednotku času

#### Požadavky na výsledný systém

* Počítačové vybavení – např. HW náročnost
* Programové vybavení – např. operační systém, programovací jazyky
* Vyvíjený software – např. efektivnost, spolehlivost, odolnost vůči chybám, přenositelnost, bezpečnostdoméno

#### Požadavky na vývojový proces

* Dodržování norem
* Odevzdání systému

#### Požadavky na rozhraní

* Software 🡪 uživatel
* Software 🡪 jiné součásti systému (HW, SW)

#### Externí požadavky

* Legislativní požadavky – např. ochrana informací

Měritelnost požadavků

## Metody získávání informací

* Kvalitní získávání informací o problémové oblasti a požadavcích snižuje riziko vytvoření systému, který nebude vyhovovat potřebám uživatele.
* Důležitá je motivace ze strany zákazníka (uživatele).
* Pro analytika jsou nutné komunikační schopnosti i zkušenosti.

### Interview (orientační, strukturované)

* Základní běžná forma zjišťování potřeb zákazníka
* Orientační – první setkání, získat základní přehled
* Strukturované – připravené otázky, získat hlubší představu
* Nejen naslouchat, ale navrhovat alternativy

### Dotazníky

* Lze obsáhnout skupinu lidí
* Dobře připravené otázky s předem definovaným způsobem vyhodnocení

### Pracovní setkání (workshop, elicitation meeting)

* Skupina lidí (stakeholders) vyjednává o požadavcích a pracuje společně na specifikaci požadavků
* Menší skupiny jsou efektivnější, různá setkání s různou skupinou lidí

### Pozorování prací u zákazníka

* Prostá specifikace nemusí být úplná, některé detaily nemusí být zachyceny, neboť jsou „zjevné“ (nemusí pro každého)
* Umožňují lépe pochopit aktivity a procesy, ověřit získané informace, odhalit dosud neznámé informace
* Časově náročné, není vhodné pro všechny projekty a uživatele

### Další metody

* Studium dokumentů
* Přímá účast na pracech zákazníka
* Analýza existujícího softwarového systému

## Problémy při specifikaci požadavků

### Přirozená neúplnost a nepřesnost

* Nejasná a neúplná formulace požadavků zákazníkem
* Neucelená představa uživatele o výsledném softwarovém systému
* Problém rozhodování, jaké požadavky už nezačleňovat do specifikace
* Pro komunikaci se používá přirozený jazyk, který je nejednoznačný

### Nedostatek znalostí

* Vývojář (analytik) se neorientuje v doménové problematice analyzované oblasti, nezná terminologii
  + Specialista na doménovou oblast ve vývojovém týmu
* Zákazník se neorientuje v problematice vývoje softwaru, nezná terminologii
  + Vyčleněný člověk od zákazníka (orientuje se ve vývoji, zaškolení)

### Nekonzistence požadavků

* Různí uživatelé mají různé požadavky a priority
* Různé požadavky uživatele a zákazníka (objednavatele)
* Požadavky jsou mnohdy rozporné

### Další problémy

* Špatná predikovatelnost dopadu nového systému na organizaci, kde se nasadí
  + Otázka naplnění obchodní požadavků
* Problémy s testováním a validací požadavků
  + Zapojení zákazníka
  + Prototypování, pravidelná setkání

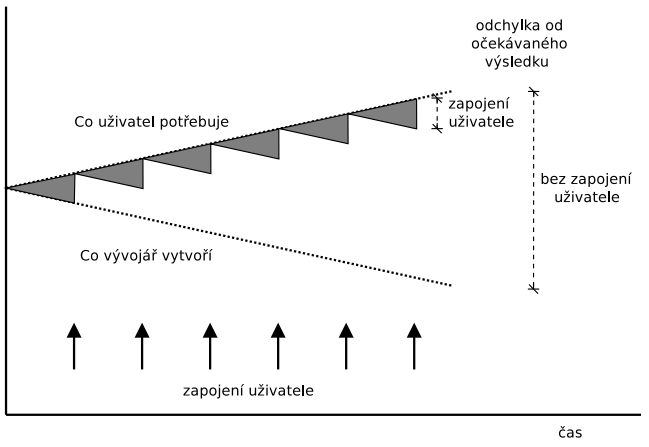
### Problémy plynoucí z použití přirozeného jazyka

* Vyřazení – př. *Používají systém k výpůjčkám knih.* Kdo?
* Deformace, zkreslení – př. *Čtenáři si nemohou půjčit další knihu, dokud nevrátí knihy s prošlou výpůjční lhůtou.* Když je zaplatí, tak mohou!
* Zobecnění – př. *Každý, kdo si chce vypůjčit knihu, musí mít průkazku.* A co výpůjčky mezi knihovnami?

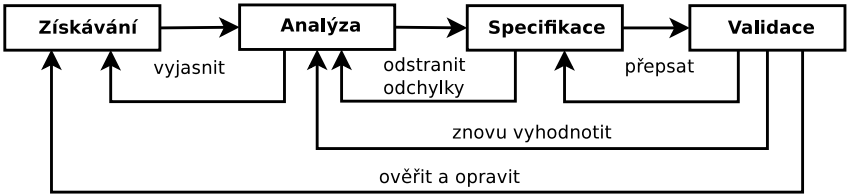
### Slovníček pojmů

* Zachycuje obchodní jazyk (terminologii) pro daný projekt
* Řeší synonyma (slova téhož nebo podobného významu) výběrem nejčastějšího
* Řeší homonyma (slova s odlišným významem znějící stejně) definicí jejich významu.

## Zapojení uživatelů

* Zapojení uživatele/zákazníka do procesu tvorby je důležité

## Postup při specifikaci požadavků

* Tvorba požadavků je iterativní proces.

### Získání informací

* Definice cílů projektu
* Identifikace uživatelských požadavků
* Interview, pozorování práce

### Analýza požadavků

* Studie vhodnosti = odhad, zda je reálné vytvořit systém s danými vlastnostmi za daných podmínek, musí být provedena rychle a levně
* Zkoumání současného stavu
* Modelování, prototypování

### Specifikace požadavků

* Transformace informací z analýzy do dokumentu
* Specifikace nefunkčních požadavků

### Validace požadavků

* Vyhodnocení požadavků, simulování, prototypování
* Definování kritérií pro akceptování produktu

## Prototypování

### Motivace

* Uživatelé mají problémy s přesnou specifikací svých požadavků, avšak dokáží lépe formulovat požadavky v reakci na práci s „reálným“ produktem => prototypem
* Prototypování dokáže ověřit správnost návrhu v počátečních fázích => validace požadavků (snižují se rizika projektu)
* Může nastínit různé alternativy návrhu

### Prototyp

* Částečná implementace produktu
* Prezentuje vnější rozhraní systému
* Prototyp je většinou implementován rychle s cílem demonstrovat potenciální rozhraní či chování systému, není kladen důraz na kvalitu návrhu a programování

### Problém dokončení prototypu

* Není kladen důraz na kvalitu návrhu a implementace prototypu => prototyp je špatně udržovatelný
* Po vyjasnění specifikace a návrhu se prototyp dále nepoužívá
* Pokud se prototypy od začátku realizují plnohodnotně, lze je využít v implementace (např. metodika RAD)

## Dobrá specifikace požadavků

**Specifikace by měla být:**

### Seřazená podle důležitosti

* Poznačte si datum vytvoření požadavku
* Seskupte požadavky do tříd důležitosti

### Sledovatelná

* Původ (smysl) požadavku je jasný
* Na každý požadavek je možné se odkazovat v další dokumentaci

### Modifikovatelná

* Struktura a styl specifikace je konzistentní a bez redundancí
* Snadné úpravy a doplňování požadavků

### Jednoznačná

* Neumožňuje více interpretací
* Požadavky pište jasně a jednoznačně (přirozený jazyk je zdrojem nejednoznačnosti)

### Úplná

* Obsahuje všechny důležité požadavky a definice reakcí systému na všechny třídy vstupních údajů
* Specifikujte situace, ve kterých se porušuje akceptovatelné chování

### Konzistentní

* Požadavek není v rozporu s jinými požadavky
* Buďte konzistentní v používání názvů

### Verifikovatelná

* Existuje proces kontroly, zda software splňuje požadavek
* Měřitelnost splnění požadavků

**Během procesu specifikace požadavků:**

### Udržujte specifikace čitelnou pro zákazníka

* Zákazník se musí umět orientovat ve specifikaci

### Ve specifikaci nenavrhujte řešení

* Cílem specifikace je získat úplný a správný pohled na potřeby zákazníka
* Realizace požadavků je záležitostí dalších etap

### Validujte požadavky

* Prototyp snižuje riziko špatného pochopení požadavků
* Slabá specifikace => špatný odhad nákladů

### Zainteresujte uživatele

* Uživatel se musí podílet na procesu formování a validace požadavků
* Nechte si výsledek zkontrolovat a potvrdit druhou stranou

## Dokumentace specifikace požadavků

### Dokumentace spojená se specifikací požadavků

* Různorodá, od textu až po formální specifikace
* Čím formálnější podoba, tím méně lidí je schopno tuto podobu akceptovat
* Praxe ukazuje, že pro mnoho softwarových projektů se používá kombinace strukturovaného jazyka, vizuálních modelů a dalších prezentačních technik (tabulky, matematické výrazy)
* Přiměřený rozsah dokumentace!

### Formy dokumentace požadavků

* Strukturovaný text
* Tabulky
* Vizuální modely
* Spustitelné modely
* Formální modely

# Komplexní modelování v procesu vývoje

## Pojmy

### Problémová doména

* Reprezentuje reálný systém (system-as-is), jehož model máme vytvořit a následně implementovat
* Z problémové domény vycházejí obchodní požadavky, uživatelské požadavky, funkční a nefunkční požadavky

### Doména řešení

* Reprezentuje vyvíjený systém (system-to-be), který odpovídá doménovému systému
* Modely systému, návrh, způsob řešení

## Konceptuální modely

### Doménový model

* Zachycuje entity a pojmy problémové domény
* Diagram analytických tříd

### Model architektury

* Zachycuje dekompozici systému a jeho budoucí architekturu
* Diagram tříd

### Modely chování

* Zachycují uživatelské a funkční požadavky
* Mohou modelovat i některé nefunkční požadavky (doba odezvy)
* Diagramy případů užití, aktivit a stavový diagram

### Modely interakce

* Zachycují interakci modelovaných elementů, např. objektů a aktérů participujících na případu užití
* Sekvenční diagram, diagram komunikace

### Modely struktury

* Zachycují strukturální vazby mezi elementy systému
* Modely reflektují principy návrhu architektury
* Diagram návrhových tříd

### Datový model

* Zachycuje perzistentní data systému
* „odlehčený“ diagram tříd, ERD

## Doménový model

* Model konceptuálních (analytických) tříd
* Nalezení abstrakcí v problémové doméně (např. Zákazník)
  + Abstrakce by měly odpovídat problémové doméně (slovníček pojmů)
  + Model případů užití používá pojmy z doménového modelu
* Zobrazovat jen věci podstatné z hlediska problémové domény
  + Třídy Zákazník, Košík, atp., ale třída pro přístup k databázi patří do domény řešení
* Doménový model není datový model, konceptuální třída nemusí mít atributy ani nemusí reprezentovat perzistentní data
* Doménový model pojmenovává koncepty doménového systému a zmenšuje tak propast mezi softwarovou reprezentací a naším mentálním modelem systému

## Konceptuální třída

* Obsahuje jen nejpodstatnější atributy a operace
* Obsahuje malou a správně definovanou množinu odpovědností
* Obsahuje minimum vazeb na jiné analytické třídy

### Hledání konceptuálních tříd

* Využití existujících modelů
* Využití seznamu kategorií
* Analýza podstatných jmen => třídy, atributy
* Analýza sloves => odpovědnosti tříd
* Metoda CRC štítků (Class, Responsibility, Collaborators)
  + Štítek reprezentuje třídu
  + Obsahuje seznam odpovědností
  + Obsahuje seznam spolupracovníků (jiné třídy) – hledání vztahů

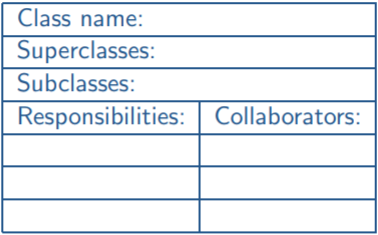
### Co by měly konceptuální třídy splňovat

* Třída má 3 až 5 odpovědností
* Každá třída spolupracuje s jinými třídami (není osamocena)
* Pozor na příliš mnoho malých tříd nebo malý počet obsáhlých tříd
* Pozor na hlubokou hierarchii ve stromu dědičnosti (typicky 3 a více úrovní)
  + Může signalizovat nevhodné použití dědičnosti
* Název třídy by měl vymezovat její účel (NakupniKosik, NavstevnikWeboveStranky)

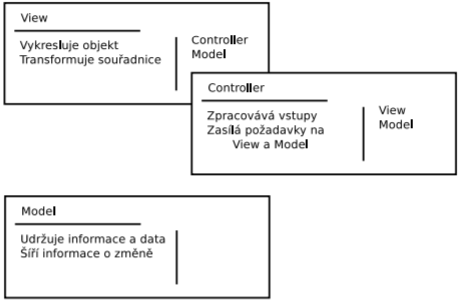
### Další

* Dvě konceptuální třídy, které spolu souvisejí, spojujeme asociací, nikoliv atributy (tzv. cizími klíči)
* Konceptuální model nezachycuje statická data, ale objekty, které reprezentují běh aplikace. (konceptuální model vs datový model)

## CRC Cards

* Představena Kent Beckem a Wardem Cunninghamem v roce 1989
* Původně pro výuku OO paradigmat
* Identifikace tříd, jejich zodpovědností a spolupracujících tříd
* Bez počítačové podpory, flexibilní práce

### Příklad – MVC

* View a Controller se překrývají, existuje úzká spolupráce
* View a Controller jsou umístěny nad Model, neboť Model neiniciuje žádnou spolupráci
* Uspořádání karet často reflektuje princip probublávání abstraktnějších konceptů na vrchol

## Modely interakce

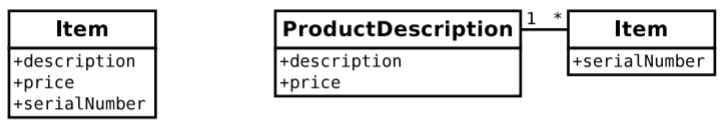
* Modelují interakce konceptuálních tříd – možnost nalezení nových konceptuálních tříd
* Identifikují zasílané zprávy mezi objekty (instancemi tříd)
  + Nalezení klíčových operací a atributů konceptuálních tříd a vztahů mezi konceptuálními třídami
* Během procesu modelování se mohou aktualizovat stávající doménový model a modely chování
* Obdobně je aplikováno i na návrhové diagramy (diagramy struktury, stavové diagramy)

## Modely struktury

* Modely návrhových tříd
* Vychází z doménového modelu, modelů chování a interakce
* Seskupení tříd reflektuje zvolenou architekturu

**Ukázkový příklad – viz přednáška 6, slide 28**

## Description Classes

* Obsahuje informace popisující jiné objekty. Typicky se jedná o otázku, za určitou skupinu atributů vyjmout a modelovat jako samostatnou třídu.
* Item představuje jednu položku zboží, jeden skutečný kus, má svůj popis, cenu a sériové číslo
* Totéž zboží má více reálných kusů, informace se tedy duplikují a pokud neextistuje na skladě reálný kus, nejsou informace žádné
* Společné informace modelujeme pomocí Description class

## Modely interakce

### Sekvenční diagram

* Zobrazuje objekty systému, externí aktéry a interakci mezi nimi
* Zachycuje události pro jeden scénář případu užití, vychází se z jeho inspekce
* Interakce jsou zachyceny pomocí zasílání zpráv

### Systémový sekvenční diagram

* Zobrazuje systém jako černou skříňku
* Důležitá součást analýzy chování systému – identifikuje události přicházející do systému (viz obrázek, přednáška 6, slide 39)

## Modely chování

* Diagramy případů užití a scénáře jsou hlavním způsobem zachycení chování systému. V některých případech je vhodné použít podrobnější popis.
* Diagram aktivit – popisuje scénář prostřednictvím toku událostí, lze zachytit i události
* Stavový diagram – popisuje změny objektu doménového modelu v reakci na události
* Operační kontrakty – popisuje změny objekty doménového modelu pomocí preconditions a postconditions)

# Implementace softwaru

* Je proces transformace návrhu jednotlivých modulů (návrhových podsystémů) a jejich vzájemných vazeb do programové realizace
* Výstupem etapy implementace je spustitelný software (softwarový produkt).

### Podíl implementace na celkovém objemu prací v životním cyklu softwaru se snižuje:

* Zavedením vysokoúrovňových jazyků (větší míra abstrakce)
* Využíváním integrovaných vývojových prostředí
* Využíváním pokročilých prostředků trasování a ladění programů
* Generováním aplikací z modelů
* Vývojem prostředků spolupráce aplikací (middleware)
* Rozšířením a rozvojem OO/AO/komponentního přístupu
* Znovupoužitelností (využití internetu)

## Vlastnosti softwarového produktu

Pro implementaci potřebujeme jasné cíle – všechna kritéria nelze splnit.  
Např. použitelnost x bezpečnost, efektivnost x udržovatelnost

Kromě funkčnosti nás zajímají další vlastnosti softwaru, které se projeví až při nasazení softwaru.

### Použití

* **Správnost** – míra, do jaké software vyhovuje specifikaci
* **Použitelnost** – úsilí, které je nutné vynaložit na to, aby se dal software používat. Zahrnuje i srozumitelnost výstupu programu.
* **Efektivnost aplikace** – doba odezvy, požadavky na paměť..
* **Efektivnost procesu tvorby programy** – čas potřebný na vývoj, náklady
* **Bezpečnost** – míra odolnosti vůči neoprávněným zásahům do systému
* **Spolehlivost** – pravděpodobnost, že software bude v daném čase vykonávat zamýšlenou funkci

### Přenos

* **Přenositelnost** – úsilí, které je nutné pro přenos softwaru z jedné platformy na jinou
* **Interoperabilita** – úsilí, které je potřebné k zajištění spolupráce systému s jinými systémy
* **Znovupoužitelnost** – míra, do jaké je možné jednotlivé části softwaru znovu použít v dalších aplikacích

### Změny

* **Udržovatelnost** – úsilí, které je potřeba vynaložit na další vývoj a údržbu softwaru podle měnících se potřeb zákazníka a také v důsledku měnícího se okolí (např. změna legislativy.) Zahrnuje i **čitelnost** a **pochopitelnost** zdrojového kódu programu.
* **Testovatelnost** – úsilí nutné pro testování vlastností softwaru, např. zda se chová správně.
* **Dokumentovanost** – míra, do které jsou všechna rozhodnutí při vývoji zdokumentována a kontinuita dokumentace v průběhu všech etap vývoje.

# Programovací jazyky

## Výběr programovacího jazyka

Kritéria výběru vhodného programovacího jazyka:

* Zkušenosti programátorů s daným jazykem
* Vhodnost jazyka pro příslušnou aplikaci, rozsah projektu
* Dostupnost podpůrných prostředků pro vývoj systémů v daném jazyku
* Rozšířenost jazyka
* Požadavky na přenositelnost
* Použitelnost na vybraném výpočetním prostředí
* Existující knihovny a možnosti znovupoužití
* Cena vývojového prostředí
* Budoucí strategie, orientace organizace na určité vývojové prostředí
* Požadavky zákazníka

## Vlastnosti jazyků

### Při programování v malém se zajímáme o:

* **Vlastnosti jazyka**: jednoduchost, srozumitelnost, side effects
* **Syntax jazyka**: konzistentnost, jednoduchost, možnost tvorby čitelných a snadno udržovatelných programů
* **Datové typy**: statické x dynamické, elementární x strukturované
* **Řídící konstrukce**: posloupnost, výběr, cyklus, rekurze, backtracking
* **Čitelnost**: možnosti formátování, syntax, pojmenování identifikátorů

### Při programování ve velkém se zajímáme o:

* **Podporu abstrakce**: procedury, funkce, generické typy údajů
* **Plánovací mechanizmy**: dělení práce, harmonogram, zdroje
* **Prostředí**: týmová tvorba softwaru, efektivní kompilace, podpora integrace systému, uchování a identifikace verzí

## Generace jazyků

### Generace

* Programování přímo v binárním kódu

### Generace

* Asemblery, symbolické vyjádření binárních instrukcí (1 ku 1)

### Generace

* Strukturované programování
* Strojově nezávislé jazyky
* Jeden příkaz se transformuje do 5-10 instrukcí v binárním kódu
* Procedurální jazyky: Fortran, Pascal, C

### 3,5. Generace (OO jazyky)

### Generace

* Neprocedurální jazyky, vizuální jazyky, doménově specifické jazyky
* Snaha o zjednodušení programování, využívají vestavěné funkce/komponenty (definuje se, co je třeba vykonat, ne jak)
* Nemožnost ovlivnit zabudovaný způsob realizace funkcí
* Jeden příkaz se přeloží do cca 30-50 instrukcí v binárním kódu (často méně efektivní realizace kódu)
* SQL, MATLAB, doménově specifické jazyky
* „End-User Programming“ – Microsoft Excel

### Generace

* Neprocedurální jazyky
* Definují se objekty, pravidla, omezení, kritéria pro řešení, postup řešení pak hledá stroj
* Umělá inteligence, neuronové sítě

## Paradigmata programovacích jazyků

### Imperativní x deklarativní

* Imperativní (Fortran, Algol, Ada, C, Pascal, Java, C++)
* Deklarativní (Prolog, Lisp, Haskell)

### Procedurální x funkcionální

* Procedurální (Algol, Ada, C)
* Funkcionální (Lisp, Haskell, Scheme)

### Objektově orientované

* Class-based (Simula, Smalltalk, Java, C++, C#)
* Prototype-based (Self, Io, Prothon)

### Logické (prolog)

### Paralelní (MPI, Shared-Memory, CUDA)

## Typy, kontrola typů

### Význam typování

* Určit sémantický význam elementů (hodnoty v paměti) => víme jaké operace je možné provést, můžeme provádět kontrolu typové konzistence

### Staticky typované jazyky

* K typové kontrole dochází v době kompilace
* Jazyky C++, Java

### Dynamicky typované jazyky

* K typové kontrole dochází v době běhu programu
* Jazyky Smalltalk, Self, Python, Lisp
* Dynamická typová kontrola probíhá u všech jazyků, avšak jako dynamicky typované se označují ty, které nemají statickou kontrolu
* Některé staticky typované jazyky (C++, Java, C#) umožňují dynamické přetypování, čímž částečně obcházejí statickou typovou kontrolu

### Silně a slabě typované jazyky

* Tyto pojmy dostávaly různé významy
* Bývá obtížné porozumět, co konkrétní autor míní užitím těchto pojmů
* Silně typované => silná omezení na kombinace typů, zamezení kompilace či běhu kódu, který může obsahovat nekorektní data (nekompatibilní typy)
* Slabě typované => slabá omezení na kombinace typů (obsahují např. implicitní přetypování)
* Silně typované jazyky bývají nazývány typově bezpečné (type safe)

# Strategie implementace

* Postup, jakým se realizují jednotlivé softwarové součásti a odevzdávají na testování
* Částečná závislost na architektuře a strategii návrhu
* Potřeba inkrementálního (postupného) vývoje
* Strategie implementace zpravidla podmiňuje strategii testování

## Implementace zdola-nahoru

* Systém je možné předvádět až po jeho úplném dokončení
* Možnost přímého použití odladěných modulů nižších úrovní
* Chyby v logice se identifikují až v etapě integračního testování
* Testování modulů na nižších úrovních: potřeba speciálních modulů (simulace chování/dat nižších úrovní)
* Testování modulů jednotlivě je jednodušší než testování logiky celého systému

## Implementace shora-dolů

* Možnost demonstrace systému poměrně brzy
* Včasná identifikace nejzávažnějších chyb
* Logika systému se ověřuje několikrát (testování celého systému)
* Testování systému: potřeba simulačních modulů (simulace práce podsystémů)
* Nedá se použít, pokud se požaduje implementace některých modulů nejnižší úrovně na začátku (např. výstupní sestavy)
* Testování logiky systému je náročnější než testování modulů jednotlivě

**V praxi se používá kombinace přístupu zdola-nahoru a shora-dolů.**

# Dobré programátorské praktiky

* Komentáře
* Jednoduchost
* Přenositelnost
  + Žádné magické konstanty (cesty, soubory, adresy)
* Jednotný programátorský styl (house style)
  + Pojmenování
  + Odsazování
  + Bílé znaky
  + Délka řádků
  + Způsob ošetření chyb

## Nástroje pro správu verzí

* = inteligentní sdílení a zálohování
* Usnadňují souběžný vývoj softwaru více lidmi
* Sledují historii změn zdrojových textů i dokumentů
* Středem celého systému je datové skladiště (repository)
* Metody ukládání: changeset vs. Snapshot
* Dostupné nástroje:
  + Revision Control Sysytem (RCS), 1982
  + Concurrent Versions Systém (CVS), 1986
  + Apache Subversion (SVN), 2000
  + Git, 2005

### Typický pracovní cyklus Subversion

* Aktualizace lokální (pracovní) kopie
* Práce (provádění změn)
* Kontrola vlastních změn
* Připojení změn od ostatních
* Zapsání vlastních změn

## Dokumentace programu

### Interní dokumentace

* Současně se čtením programu
* Slouží pro opravu chyb, k údržbě

### Externí dokumentace

* Pro ty, kdo se nemusí zabývat vlastním programem
* Např. pro návrháře pro potřeby modifikace návrhu apod.
* Popis problému, algoritmů, údajů,..

### Hlavičky souborů

* Co soubor obsahuje
* Kdo a kdy ho vytvořil/upravil
* Závislosti na dalších souborech

### Komentáře

* Komentovat účelně (hlavně nestandardní a neočekávané obraty)
* Stručný popis algoritmů

### Co by měla dokumentace obsahovat

* Název, autoři, datum, kam je daný modul (soubor) zařazen, účel, předpoklady (jaké se očekávají vstupy apod.)

### Dokumentaci lze automaticky generovat ze zdrojového kódu

* Struktura programu, komentáře, kontrakty
* Doxygen, Javadoc, Sandcastle

# Validace a verifikace programu

* Zjišťujeme, zda software odpovídá specifikaci a splňuje potřeby uživatele

### Verifikace

* Vytváříme výrobek správně? (podle požadavků, specifikace)

### Validace

* Vytváříme správný výsledek? (Jsou splněny potřeby uživatele? Odpovídá tomu specifikace?)

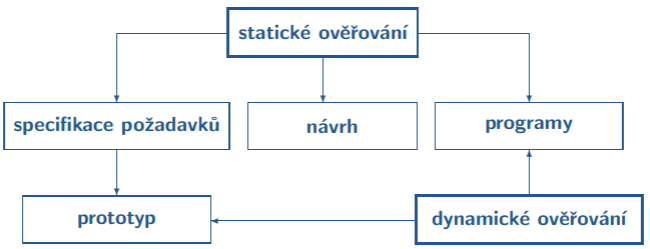
### Sledované vlastnosti

* Správnost
* Spolehlivost
* Efektivnost
* Bezpečnost

## Cíl verifikace a validace

* Odhalit chyby během vývoje
  + Test, který neodhalí nesprávné chování systému, je neúspěšný.
* Prokázat požadované vlastnosti

## Typy ověřování

* **Statické** – nevyžaduje běh programu, lze v libovolné etapě vývoje SW
* **Dynamické** – proces odvození vlastností výrobku na základě výsledků použití (běhu) programu s vybranými vstupy

# Statické ověrování

## Statické ověřování – prohlídka dokumentu

* Prohlídka dokumentu je založena na statické prohlídce vytvořených dokumentů (včetně zdrojových textů programů).

### Existují různé přístupy:

* Formální (Inspection)
* Neformální (Walkthrough)
* Koukání přes rameno (Over-The-Shoulder)
* Párové programování (Pair Programming)
* Koupací kačenka (Rubber Duck Debugging)

## Code Review

#### Doporučení

* 200 až 400 LOC pro prohlídku
* Obvyklá rychlost procházení 300 LOC/h
* Délka nejlépe do 60 minut (max. 90 minut)
* Intenzita odhalování 15 chyb za hodinu

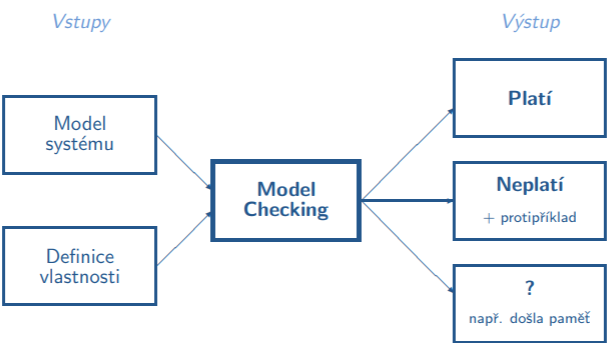
#### Výhody

* Nejefektivnější způsob odhalování chyb v kódu
* Zvyšuje i čitelnost a udržovatelnost kódu
* Zlepšuje schopnosti méně zkušených programátorů
* Úspora nákladů později

#### Nevýhody

* Nemusí to být moc příjemné

## Statické ověřování – formální verifikace

* formální matematický důkaz
* Ověřovaný dokument musí být formálně reprezentovaný (přesná definice sémantiky)

### Modelování systému

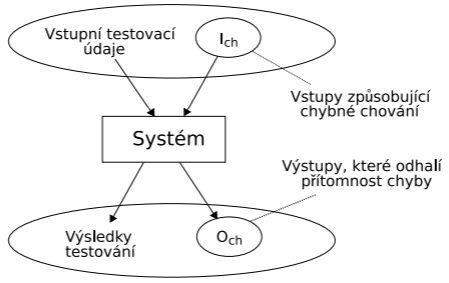
* Automaty
* Petriho sítě
* Procesní algebry
* Programovací jazyky

### Specifikace vlastností

* Obecné vlastnosti (např. bez uváznutí)
* Tvrzení (assertions)
* Zakázané stavy (bad states)
* Temporální logiky

# Dynamické ověřování – testování

### Cíl

* Vybrat takové testovací vstupy, pro které je pravděpodobnost příslušnosti do množiny **Ich** vysoká

### Proces testování

## Množina testovacích vstupů

* Velikost množiny testovacích vstupů musí být přijatelná
* Množina testovacích vstupů se vybírá na základě testovacího kritéria
* Testovací kritérium určuje podmínky, které musí splňovat množina testovacích vstupů, např. pokrytí všech příkazů v programu
* Testovací kritérium může splňovat více množin testovacích vstupů

## Vlastnosti testovacího kritéria

* **Spolehlivost**: kritérium K je spolehlivé, když všechny množiny testovacích vstupů splňující kritérium K odhalí ty samé chyby => nezáleží na tom, která množina testovacích vstupů se vybere, vždy odhalíme ty samé chyby
* **Platnost**: kritérium K je platné, když pro každou chybu v programu existuje množina testovacích vstupů, která splňuje kritérium K a která odhalí chybu

Když je testovací kritérium spolehlivé a platné a množina testovacích vstupů, která splňuje kritérium, neodhalí žádné chyby, tak program neobsahuje chyby.

Ale bylo dokázané, že neexistuje algoritmus, který určí platné kritérium pro libovolný program.

## Techniky testování

### Náhodné testování

* Množina testovacích vstupů se vybere náhodně

### Funkcionální testování

* Na základě specifikace programu (vstupy, výstupy)
* Metoda černé skříňky
* Black box, data driven, functional, input/output driven, closed box

### Strukturální testování

* Na základě vnitřní struktury programu
* Metoda bílé skříňky
* White box, glass box, logic driven, path oriented, open box

### Testování rozhraní

* Na základě znalostí rozhraní mezi moduly a specifikace programu

## Funkcionální testování

* Zjištění zda vstupně-výstupní chování vyhovuje specifikaci (např. matematická funkce se specifikuje vstupy a výstupy)
* Testovací vstupy se odvozují přímo ze specifikace
* Neuvažuje se vnitřní struktura, logika modulu => velká množina testovacích vstupů (problém)
* Úplné funkcionální testování je v praxi nemožné

### Třídy ekvivalence vstupů/výstupů

* Každý možný vstup/výstup patří do jedné z tříd ekvivalence, pro které je chování systému identické (vstup-výstup)
* Žádný vstup/výstup nepatří do více tříd ekvivalence
* Pokud se při daném vstupu/výstupu zjistí chyba, pak stejnou chybu je možné odhalit použitím jiné vstupu/výstupu z dané třídy ekvivalence

### Granularita třídy ekvivalence

* Rozsah
* Hodnota
* Podmnožinah

### Výběr testovacích údajů z třídy ekvivalence

* Průměr, medián třídy ekvivalence
* Hranice třídy ekvivalence (příp. s okolními hodnotami)
* Náhodně (doplnění množiny testovacích vstupů)

## Strukturální testování

* Vychází se z vnitřní struktury programu
* Testuje se implementace programu
* Snaha o pokrytí různých struktur programu – řízení, data
* Kritéria
  + Založená na tocích řízení (pokrytí cest, pokrytí rozhodovacích bloků nebo podmínek a pokrytí příkazů)
  + Založená na tocích dat
* Mutační testování
  + Do programu se úmyslně zavedou chyby
  + Kontrolujeme, zda navržené testy tyto chyby odhalí (kvalita testu)

## Testování vícevláknových aplikací

* Je obtížné najít podobné problémy klasickým (sekvenčním) testováním

### Způsoby testování vícevláknových aplikací

* Systematické testování (řízení plánovače)
* Vkládání šumu
* Saturační testování

## Strategie testování

### Testování zdola-nahoru (bottom-up testing)

* Testují se komponenty na nižší úrovni, poté se integrují do komponenty vyšší úrovně a znovu otestují
* Vhodné, pokud většina modulů stejné úrovně je připravena

### Testování shora-dolů (top-down testing)

* Testují se integrované moduly nejvyšší úrovně, poté se testují submoduly
* Problém s připraveností všech modulů (simulace modulů na nižších úrovních)

### Sendvičové testování (sandwich testing)

* Kombinace strategií bottom-up a top-down testování
* Moduly se rozdělí do dvou skupin
  + Logické: řízení a rozhodování, top-down
  + Funkční: vykonávání požadovaných funkcí, bottom-up

### Jednofázové testování (big-bang testing)

* Moduly se otestují samostatně a poté se naráz integrují
* Náročná identifikace místa chyby při integraci
* Náročné rozlišení chyb v rozhraní modulů od ostatních chyb

### Testování porovnáváním (comparison testing, back-to-back testing)

* Více verzí systému na testování
  + Prototyp
  + Technika programování N-verzí => vývoj vysoce spolehlivých systémů
  + Vývoj více verzí produktu pro různé platformy
* Stejné výsledky značí, že verze pravděpodobně pracují správně
* Problémy:
  + Stejné chyby ve verzích
  + Nevyhovující specifikace

## Testování produktu

### Různé způsoby (účely) testování

* Testování funkčnosti celého systému
* Testování robustnosti celého systému
* Kritické testování (stress testing) – testování hraničních podmínek
* Testování objemu dat (volume testing)
* Regresní testování (ověří, že fungují dříve vytvořené funkce systému)
* Testování splnění mezí – doba odezvy, paměťové nároky, bezpečnost (jak je drahé proniknutí do systému)
* Testování zotavení (chybný vstup, odpojení napájení)
* Testování dokumentace (dodržení standardů, aktuálnost, použitelnost)

### Způsob testování

* Automatické
* Ruční

### Testovací scénář

* Postup testování vlastnosti komponenty
* Skript/tester postupuje podle scénáře
* Během vývoje se scénáře mohou modifikovat (rozšiřovat)

### Zpráva o chybách

* Shrnuje, jaké testy (scénáře) byly provedeny, s jakými daty a s jakým výsledkem
* Je nutné přesně specifikovat vyvolanou chybu (nějak to špatně počítá…)
* Je nutné zaznamenat postup vyvolání chyby

## Podpora testování

### Statická analýza

* Analýza programu bez spuštění
* Snaží se najít časté programátorské chyby
  + Syntaktické chyby
  + Nedosažitelné části programu
  + Neinicializované proměnné
  + Nevyužití hodnoty po jejím přiřazení do proměnné
  + Odkaz přes NULL ukazatel
  + Použití paměti po jejím uvolnění
  + Opakované uzavření souboru
  + Dělení nulou
  + Uváznutí (deadlock)
  + Časově závislé chyby (race condition)
* Obvykle hlásí řadu falešných chyb

### Dynamická analýza

* Analýza při běhu testovaného programu
* Může detekovat některé chyby:
  + Nesprávná práce s dynamickou pamětí (valgrind)
  + Uváznutí (deadlock)
  + Časově závislé chyby (race condition)
  + Obvykle hlásí méně falešných chyb než statická analýza
* Profiling – zjišťuje např. využití paměti nebo počet vyvolání a čas strávený v jednotlivých funkcích (užitečné pro optimalizace)
* Analýza pokrytí

## Akceptační testování

* Testuje se na reálných datech
* Testuje se u uživatele
* Uživatel určí, zda produkt splňuje zadání
* Další změny po akceptaci systému již představují údržbu systému
* Vztahuje se na zakázkový software

## Alfa a Beta testování

* Pro generické softwarové výrobky, kde není možné provést akceptační testy u každého zákazníka (operační systémy, kompilátory)

### Alfa testování

* Tam, kde se vyvíjí software
* Testuje uživatel, vývojáři sledují a evidují chyby
* Známé prostředí

### Beta testování

* Testují uživatelé u sebe
* Neznámé prostředí
* Výsledkem je zpráva uživatele => modifikace softwaru => předání softwaru k používání

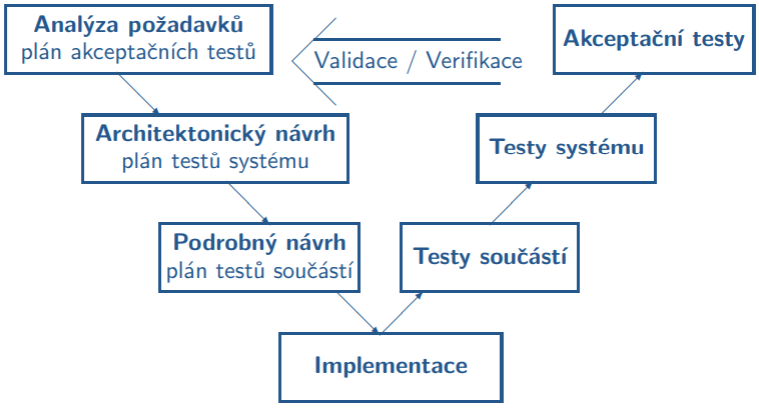
# Modely životního cyklu

## Lineární (sekvenční) modely

* Životní cyklus jde postupně od první etapy až do poslední
* Typický představitel je vodopádový model

## V-model

### Vlastnosti

* Vychází z vodopádového modelu
* Písmeno V symbolizuje grafické uspořádání etap, zdůrazňuje vazby mezi návrhovou a testovací částí
* Písmeno V je také synonymum pro validaci a verifikaci
* Zachovává si jednoduchost a srozumitelnost vodopádového modelu

### Levá část

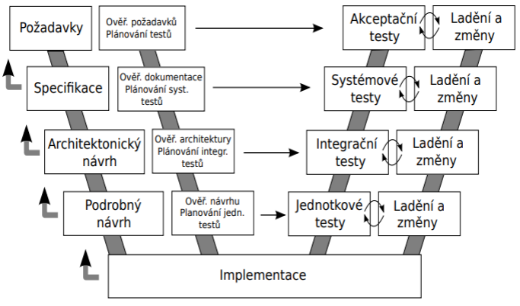
* Vývojové aktivity
* Plánování testů

### Pravá část

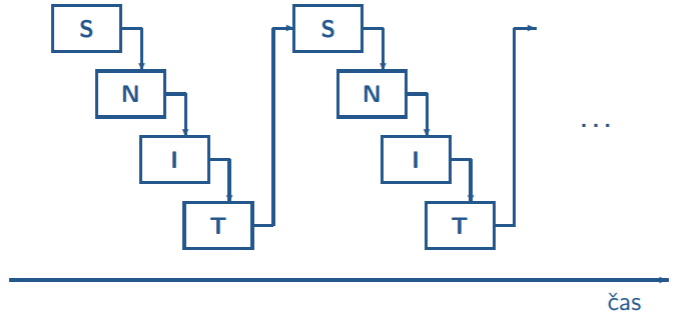
* Testovací aktivity
* Provádění testů podle plánů

## W-model

### Vlastnosti

* Vychází z V-modelu
* Aktivity spojené s ověřováním a testováním jsou na stejné úrovni jako návrhové aktivity => druhé souběžné V

## Iterativní modely životního cyklu

* Sekvence etap se v životním cyklu opakuje

### Vlastnosti

* Systém se vyvíjí v iteracích
* V každé iteraci se vytvoří reálný výsledek
* Zákazník se účastní vývoje (předpoklad

### Výhody

* V každé iteraci se vytvoří reálný výsledek => zákazník má možnost validovat výsledek se svými požadavky, rychlejší odhalení chyb ve specifikaci

### Nevýhody

* Náročnější na řízení
* Potenciálně horší výsledná struktura
* Existují techniky, jak tento nedostatek zmírnit (např. refaktorizace)

## Inkrementální model

### Vlastnosti

* Zjednodušení zavedení změn během vývoje, omezení projektových rizik
* Vývoj po částech může vést ke ztrátě vnímaní logiky a technických požadavků celého systému
* Zvýšená pozornost musí být věnována rozhraním modulů (částí)
* Na základě specifikace celého systému se stanoví ucelené části systému
* Možnosti

1. Série vodopádů, každý pro jednu část systému

Vodopád je dokončen před dalším přírůstkem

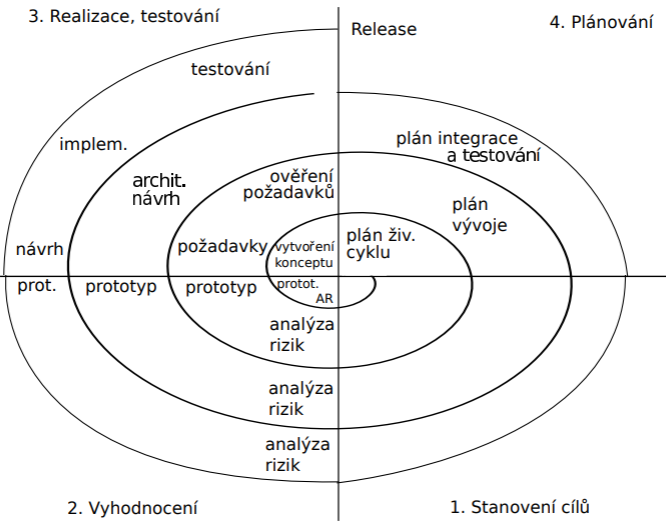
Je možné předávat uživateli po částech

1. Počáteční analýza, specifikace a návrh jsou provedeny vodopádem

Následuje iterativní přístup kombinovaný s prototypováním

Systém se vyvíjí postupně, v každé další verzi je systém rozšířen

## Spirálový model

* Kombinace prototypování a analýzy rizik
* Jednotlivé kroky se ve vývoji opakují vždy na vyšším stupni
* Vyžaduje stálou spolupráci se zákazníkem
* Přístupy řízené riziky (risk-driven approach)

### Význam cyklů

* První – globální rizika, základní koncept vývoje, volba metod a nástrojů
* Druhý – vytváření a ověřování specifikace požadavků
* Třetí – vytvoření a ověření návrhu
* Čtvrtý – implementace, testování a integrace

### Úvodní fáze každého cyklu identifikuje

* Cíle cyklu – např. výkonnostní požadavky, funkcionální požadavky apod.
* Alternativy – různé způsoby řešení cílů
* Omezující podmínky – např. cena, plán projektu
* Následně se vyhodnocuje (analýza rizik, prototypování, simulace)

### Analýza rizik

#### Jaké jsou cíle

* Zjistit možná ohrožení průběhu projektu
* Připravit reakce na tato rizika
* Rizika se identifikují a analyzují v každé fázi vývoje
* Včasné vyloučení nevhodných řešení

#### Jaká mohou být rizika

* Projektová – odchod lidí, snížení rozpočtu
* Technická – neznámé technologie, selhání hardwaru
* Obchodní – špatný odhad zájmu

### Mezníky (Milestones)

#### Life Cycle Objectives (LCO): po 2. cyklu

* Vyhodnocení záměrů a cílů projektu, rozhodnutí o pokračování projektu
* Všechny požadavky podchyceny, stejné chápání požadavků
* Cena, plán, priority apod. odpovídají záměrům
* Jsou identifikována rizika a procesy pro jejich odstranění/zmírnění

#### Life Cycle Architecture (LCA): po 3. cyklu

* Vyhodnocení výběru architektury, řešení závažných rizik
* Požadavky a architektura jsou stabilní
* Osvědčené postupy testování a vyhodnocování

#### Initial Operation Capability (IOC): po 4. cyklu

* Systém je připraven na distribuci pro uživatelské testování
* Stabilní verze schopná testového nasazení u zákazníka/uživatele
* Srovnání plánovaných a skutečných výdajů a použitých zdrojů

### Výhody

* Komplexní model vhodný pro složité projekty
* Chyby a nevyhovující postupy jsou odhaleny dříve (analýza rizik)
* Nezávislost na metodice či strategii

### Nevýhody

* Závislý na analýze rizik – musí být prováděna na vysoké odborné úrovni
* Vyžaduje precizní kontroly výstupů, zkušené členy týmu
* Software je k dispozici až po posledním cyklu (lže vyřešit použitím většího počtu implementačních cyklů)
* Problematické je přesné plánování termínů a cen

## RUP metodika – Rational Unified Process

* Výsledek výzkumu zkušeností řady velkých firem koordinovaný firmou Rational Software, 1997
* Spíše než konkrétní metodika je chápán jako rozšířitelný Framework, který by měl být uzpůsoben organizaci či projektu
* Komerční produkt, dodávaný společně s nástroji

### Základní vlastnosti

* Objektové orientovaná metodika
* Iterativní vývoj
* Přístupy řízené případy užití (use-case-driven approach)
* Věnuje se všem otázkám procesu tvorby softwaru (kdo, co, kdy a jak)

### Šest základních praktik

* **Využívání existujících komponent**
* **Vývoj softwarového produktu iteračním způsobem** – po každé iteraci spustitelný kód
* **Model softwarového systému je vizualizován** – UML
* **Průběžná kontrola kvality produktu** – objektivní měření, metriky
* **Správa požadavků na softwarový systém** – umění získávání požadavků od zákazníka
* **Řízení změn systému** – každá změna je přijatelná, všechny změny jsou sledovatelné

### Základní elementy

#### Pracovní a role (kdo)

* Chování je popsáno pomocí činností
* Důležitá je role: analytik, návrhář..

#### Činnosti – Activities (jak)

* Jasně definovaný účel s definovaným výsledkem (meziprodukt)

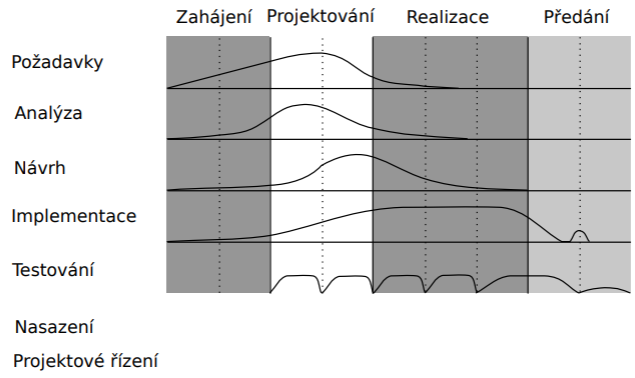
#### Meziprodukty – Artifacts (co)

* Výsledky projektu (činností)
* Model, dokument, zdrojový kód..

#### Pracovní proces – Workflows (kdy)

* Definuje posloupnost činností a interakce mezi pracovníky
* RUP definuje 9 klíčových procesů (Specifikace požadavků, Implementace..)

### Vývojový cyklus

* **Initial Development Cycle** – výsledkem je funkční softwarový produkt
* **Evolution Cycles** – další vývoj, verze..

### Základní cyklus

* Je rozdělen na čtyři fáze
  + Zahájení (inception) 10 %
  + Projektování (elaboration) 30 %
  + Realizace (construction) 50 %
  + Předání (transition) 10 %
* Každá fáze je rozdělena na iterace
  + Délka jedné iterace 2 až 6 týdnů

### Model životního cyklu

* Iterativní model jednoho vývojového cyklu
* Etapy (pracovní procesy) se překrývají (souběžné provádění)

### Fáze cyklu

#### Zahájení (inception)

* Rozsah projektu, náklady, základní rizika, základní UC
* 1-2 iterace

#### Projektování (elaboration)

* Plánování, specifikace požadavků, architektura, analýza rizik
* Zpravidla 2 iterace, může až 4 iterace

#### Realizace (construction)

* Kompletace analýzy a návrhu, implementace, hodnocení výstupů
* 2-4 iterace

#### Předání (transition)

* Dodání, školení, podpora při zavádění
* Alespoň 2 iterace (betaverze, plná verze)

### Mezníky (milestones)

* Převzaté ze Spirálového modelu
  + Life Cycle Objectives – vyhodnocení zámeřů/cílů projektu, rozhodnutí o pokračování projektu
  + Life Cycle Architecture – vyhodnocení výběru architektury, řešení závažných rizik
  + Initial Operation Capability – systém je připraven na distribuci pro uživatelské testování
* + Product Release - rozhodnutí, zda byly záměry projektu splněny a zda pokračovat v dalším vývojovém cyklu (Je uživatel spokojen, odpovídají náklady plánu..)

### Výhody

* Robustní, vhodný pro velkou škálu projektů
* Iterativní přístup, včasné odhalení rizik, správa změn
* Detailní propracovanost
* Vazba na UML

### Nevýhody

* Detailní propracovanost: u menších projektů značná zátěž na zkoumání metodiky, vývoj může postrádat efektivitu
* Komerční produkt (obsahuje hodně podpůrných nástrojů)

## RAD metodika – Rapid Application Development

### Vlastnosti

* Rychlý iterativní vývoj prototypů
* Funkční verze jsou k dispozici dříve než u předchozích přístupů
* Intenzivní zapojení zákazníka/uživatele do vývojového procesu
* Zaměřuje se na splnění business potřeb (potřeby a požadavky zákazníka), technologické a inženýrské kvality mají menší důležitost
* Určen pro menší až středně velké projekty

### Fáze (přehled)

* Plánování – rozsah projektu, omezení, systémové požadavky
* Návrh – modelování, prototypování, využívání CASE nástrojů
* Provedení – pokračování návrhu, kódování, integrace, testování
* Uzavření a nasazení – příprava dat, finální tetování, přechod zákazníka na nový systém, zaškolení uživatelů

### Výhody

* Flexibilita, schopnost rychlé změny návrhu podle požadavků zákazníka
* Více projektů splňuje termíny a ceny (úspora času, peněz a lidských zdrojů)
* Vyšší kvalita zpracování business potřeb (prototypování)

### Nevýhody

* Nižší kvalita návrhu, problém s udržovatelností
* Flexibilita vede k menší míře kontroly nad změnami
* Projekt může skončit s více požadavky, než je nutné (problém s udržovatelností)

## Další přístupy k procesu vývoje softwaru

### Unified Software Development Process (zjednošeně UP)

* Stejné principy a myšlenky jako RUP, není komerční, nenabízí nástroje
* Není tak detailně rozpracována, např. pouze 5 pracovních procesů

### Modifikované verze vodopodádu

* Možnost prolínání etap
* Vodopád s podprojekty

### Agilní přístupy (metodiky)

* Skupina metodik s odlišným přístupem k procesu tvorby softwarového produktu

## Heavyweight a agilní metodiky

### Heavyweight methods

* Častá kritika byrokratizace metodik – příliš mnoho aktivit, které jsou předepisovány, způsobuje snížení efektivity celého procesu vývoje
* Člen vývojového týmu sleduje přesně postup, krok po kroku

### Lightweight methods

* Nová skupina metodik, dnes nazývána agile methods (agilní metodiky)
* Kompromis mezi chaotickým přístupem bez procesů (žádná metodika) a přístupem s mnoha procesy (heavyweight metodiky)
* Definují základní rámec vývoj, termín (milestones), předpokládané výstupy, techniky
* Agilní = čilý, aktivní => člen vývojového týmu používá procesy aktivně, tj. sám přizpůsobuje procesy a techniku potřebám projektu a týmu

### Srovnání vlastností



## Prediktivní přístupy

* Plánují velké části softwarových procesů velmi detailně pro dlouhý časový úsek
* Projekty vyžadují mnoho procedur, času, velké týmy a stabilní požadavky

### Dobrá predikovatelnost

* Projekty s jasnými a stabilními požadavky (projekty NASA)

### Špatná predikovatelnost

* Projekty s požadavky, které se v čase mění (business projekty)

## Adaptivní přístupy

* Plánují s přiměřenou mírou detailu
* Plány se v průběhu procesu vývoje revidují
* Jak řídit adaptivní procesy? => iterativní přístup

### Iterativní přístup a plánování procesů

* Jedna iterace většinou zahrnuje základní etapy, může se měnit podle zvolené metodiky
* V první iteraci se provádí plánování procesů, tento plán se v dalších iteracích upravuje podle reálného stavu
* Otázka délky iterace (týdny, měsíce), určení milestones

## Process-oriented přístupy

### Předpoklady

* Lidé jsou zdroje, které jsou dostupné v několika rolích: analytik, programátor, tester, manažer..
* Procesy by měly fungovat za všech okolností (změna týmu..)

### Důsledky

* Podstatná je role, nikoliv individualita lidí
  + Není důležité jaké analytiky máte, ale kolik jich máte
  + Člověk je predikovatelná ( a tedy jednoduše nahraditelná) komponenta vývojového procesu
* Procesy by měly fungovat za všech okolností => velký objem procesů, detailní specifikace procesů, velká míra režie
* Za standardní prostředek komunikace se považuje dokumentace => zvýšení režie

## People-oriented přístupy

### Předpoklady

* Lidé nepracují konzistentně v průběhu času
  + Pokud by člověk dostal každý den stejný úkol, vytvoří podobné výsledky, ale nikdy ne stejné
  + Schopnost pracovního nasazení/soustředění se mění
* Lidé jsou komunikující bytosti
  + Fyzická blízkost – gestikulace, hlasový projev, intonace
  + Otázky a odpovědi v reálném čase
* Žádný proces nikdy nevytváří dovednost (znalosti) vývojového týmu

### Důsledky

* Podstatná je individualita lidí
  + Důležitá je kvalita a osobní rozvoj členů týmu
  + Role člena týmu se může měnit
  + Kvalitní člen týmu je hůře nahraditelný
* Úloha procesů
  + Podpora práce vývojového týmu
  + Menší objem procesů, proces definuje především pracovní rámec a termíny
* Komunikace
  + Osobní komunikace
  + Dokumentace slouží především k dokumentačním účelům (pro potřeby revizí návrhu, údržby)

# Agilní metodiky

## Základní teze

#### Minimum formálních a byrokratických artefaktů

* Důležitou součástí dokumentace i zdrojový kód

#### Člen týmu je schopen rozhodovat technické otázky své práce

* Důraz na složení týmu a komunikaci uvnitř týmu
* Komunikace jako jedna z forem vývoje
* Techniky vyžadující komunikaci

#### Ověření správnosti navrženého systému zpětnou vazbou

* Iterativní inkrementální vývoj, časté uvolňování průběžných verzí
* Předložit zákazníkovi a na základě zpětné vazby upravovat
* Zákazník je členem vývojového týmu

#### Důraz na rigorózní, průběžné a automatizované testování

* Zejména kvůli neustálým změnám v kódu i návrhu

#### Princip jednoduchosti

* Návrh odráží aktuální potřeby uživatele
* Do systému vložíme to, co potřebujeme, když to potřebujeme

## Příklady agilních metodik

* Extreme programming (XP)
* Crystal
* Scrum
* Feature Driven Development (FDD)
* Test Driven Development (TDD)
* Dynamic Systém Development Method (DSDM)

# Extrémní programování (XP)

* Kent Beck, Ward Cunningham, 80. léta – Smalltalk, 90. léta

Charakteristické vlastnosti:

## Komunikace

Teze: **programátoři, zákazníci a manažeři musí spolu komunikovat**

* XP využívá takové techniky, které komunikaci vyžadují (testování, párové programování, odhady úkolů)
* Člen týmu (kouč), který udržuje komunikační toky, pomáhá programátorům s technickými dovednostmi, komunikuje s manažery na vyšších úrovních
* Člen týmu (velký šéf), který provádí zásadní rozhodnutí

## Zpětná vazba

Teze: **stav a kvalita vývoje se nejlépe zjistí od zákazníka a testováním**

* Snaha mít co nejdříve implementované nejdůležitější části systému, nejlépe nasazené přímo v provozu
* Odpovědí na otázku „Funguje to?“ je testování
* Zpětná vazba musí být rychlá
* Člen týmu (zákazník), který vyhodnocuje dosaženou funkcionalitu

## Jednoduchost

Teze: **Jednoduché věci se realizují a upravují rychleji s menším počtem chyb**

* je dobré udržovat si přehled o tom, co bude
* ale je nutné soustředit se na to, co je potřeba právě teď
* jednoduché věci je třeba vytvářet jednoduše => úspora času na opravdu složité věci
* v případě potřeby není problém jednoduché věci rozšířit

## Odvaha

Teze: **pokud je to potřeba, nebát se provést zásadní změny, a to i za cenu dočasného snížení úspěšnosti testů a tedy zvýšeného úsilí**

* nebát se zahodit naprogramovaný kód
* nebát se zkusit neznámé (když nevíš, že to nejde, může se to podařit)

## Základní techniky

### Přírůstkové (malé) změny

* návrh a implementace se mění v čase jen pozvolna
* uvolňování malých verzí systému (nejpodstatnější požadavky, postupně vylepšované a doplňované)

### Testování

* Co nelze otestovat, to neexistuje
* Ke každé funkci píšeme testy, někdy i před tím, než začneme programovat
* Zautomatizovaný systém testů
* Jednotkové a integrační testování

### Párové programování

* Jednu věc programují vždy 2 programátoři (ale pouze 1 skutečně píše)
* Ten, kdo píše, se soustřeďuje na nejlepší způsob implementace problému
* Druhý se soustřeďuje na problém z globálnějšího pohledu (bude to fungovat, jaké další testy, možnost zjednodušení)
* Páry jsou dynamické

### Refaktorizace

* Úprava stávajícího programu – zjednodušení, zefektivnění návrhu
* Odstranění (úprava) nepotřebných částí
* Změna architektury (pravidlo přírůstkové změny)
* Při refaktorizaci se nemění funkcionalita!

### Metriky

* Důležitá součást určení kvality softwarových procesů
* Např. poměr plánovaného času a skutečného času
* Přiměřený počet metrik (3-4)
* Pokud přestane metrika plnit svůj účel => nahradit jinou (např. metrika testů funkcionality se blíží 100 % => nahradit jinou s menší úspěšností)
* Existují pravidla udávající kdy a jak často by se měly jednotlivé techniky používat

### Motivace vývojářů

* Lidé lépe pracují, pokud je práce baví
* Jídlo, hračky, vybavení pracoviště..

## Proces vývoje

1. Zkoumání (exploration)

* Tvorba vysokoúrovňových požadavků
* Tvorba základního návrhu prostřednictvím prototypů

1. Plánování (planning)

* Odhad času, výběr minimální možné množiny požadavků
* Plánování iterací

1. Iterace (iterations to first release)

* Iterativní vývoj s využitím specifických pravidel a technik XP

1. Produkce (productionizing)

* Verifikace a validace, nasazení systému

1. Údržba (maintenance)

* Implementace zbývajících požadavků a nových potřeb do běžícího systému (iterování fází 2 až 4)

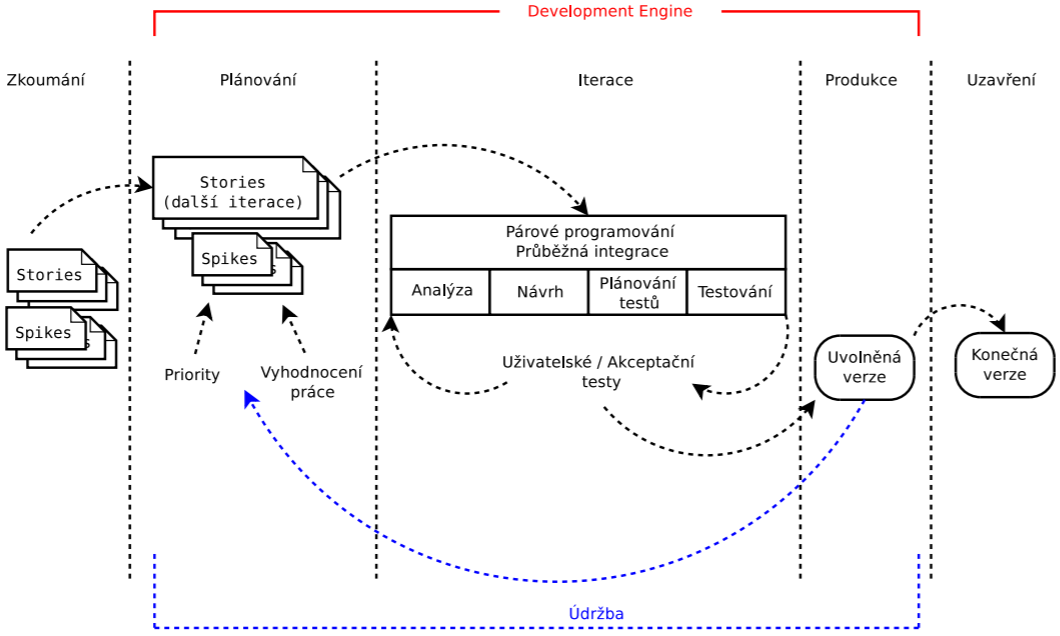
1. Uzavření (Death)

* Uzavření a zhodnocení projektu

## Zkoumání (exploration)

* Utváření týmu
* Návrh počáteční množiny User Stories.
* User story
  + Definuje vlastnost systému z pohledu zákazníka/uživatele
  + Je psána uživatelem terminologií problémové domény
* Tvorba systémových metafor (Metaphor) a prototypů (Spikes)
  + Prototypy pomáhají definovat metafory
  + Metafora je jednoduchý popis, jak má systém pracovat, srozumitelný pro všechny členy týmu => pochopení požadavků a architektury

## Development Engine

* Aktivity spojené s fázemi Plánování, Iterace a Produkce
* Každý jeden běh těchto fází produkuje novou verzi
* V jednom běhu enginu probíhá vývoj v iteracích (fáze Iterace)
* Po prvním nasazení (fáze Produkce) se kroky fáze Údržba provádějí iteracemi v enginu

## Collective-Code-Ownership

* Aplikováno na procesy analýzy, návrhu, programování, testování a integrace
* Sdílení kódu, kdokoliv má možnost měnit kód (nová funkcionalita, odstranění chyb, refaktorizace)
* Nutnost zavést principy test-driven development
  + Ke každému kódu musí existovat jednotkové testy (unit tests)
  + Testy se sdružují do sady (test suite)
  + Při každé změně kódu musí být provedena (automatizovaně) sada testů
  + Vkládání nového kódu (integrace) je chráněno sadami testů (nový kód musí testy projít)

### Základní techniky

* Využívání CRC karet – nalezení nejjednoduššího návrhu
* Jednotný styl programování (standard) – zlepšení komunikace
* Vývojáři se postupně účastní všech prací – získání znalostí o všech částech systému
* Vývojáři mají pracovat udržitelným tempem, nikdo nesmí pracovat přes čas příliš dlouho
* Párové programování
* Průběžné provádění refaktorizace
* Průběžná integrace (continuous integration)

## Průběžná integrace

* Automatizované a reprodukovatelné sestavování (build)
* Obsahuje automatizované testování, které probíhá mnohokrát za den
* Umožňuje průběžně integrovat změny a tím redukovat problémy s integrací

### Základní procesy

* Integrace zdrojového kódu – sdílené repozitáře
* Automatizovaná správa sestavování (build management)
  + Sestavování se provádí často, několikrát za den
  + Sestavení se provádí při změně kódu, v naplánovaném čase
  + Vývojář musí být informován o výsledku
* Automatizované ověřování (testování)
  + Po sestavení je nutno ověřit, že nová verze splňuje všechny testy

## Zhodnocení XP

### Silné stránky

* Iterativní inkrementální proces
* Proces se ladí na základě zpětné vazby
* Požadavky se ladí během celého vývoje
* Průběžná integrace
* Zapojení uživatelů
* Vývoj založená na testování

### Slabé stránky

* Definuje rámec, principy a praktik, ale nedefinuje přesný postup
* Nepředepisuje modely pro návrh, často se od User Stories a Metaphor přechází na implementaci
* Hůře akceptovatelný pro vývojáře – vyžaduje striktní dodržování základních principů a procesů

# Scrum

* První varianta představena v roce 1995
* Název odkazuje důležitost týmové práce (odvozeno z hry rugby)
* Dá se kombinovat s programovacími praktikami XP
* Tři základní fáze: pre-game, development (game), post-game

## Scrum Proces: Pre-Game

### Plánování

* Počáteční seznam požadavků **Product Backlog**, seřazený podle priorit (backlog – nedodělávky, nevyřízené objednávky)
* Analýza rizik
* Odhad časů, zdrojů
* Formování týmu (**scrum teams**)
  + Jeden tým má 5 až 10 členů
  + Každý člen má jinou specializaci
  + **Scrum Master** – vedoucí, zajišťuje správné používání Scrum praktik

### Architektonický návrh

* Analýza problémové domény na základě backlog
* Tvorba doménových modelů, prototypů
* Definice architektury systému
* Úprava požadavků (backlog) podle navržené architektury

## Scrum Proces: Development

### Popis fáze

* Probíhá v iteracích, iterace se nazývá **Sprint**
* Typická délka iterace je 30 dnů
* Výsledkem je funkční část (inkrement) odpovídající **Sprint Backlog** (podmnožina Product Backlog)
* Každá iterace obsahuje
  + Plánování (planning)
  + Vývoj (development)
  + Posouzení (review)

### Sprint Planning – plánování

#### Setkání

* Na začátku každé Sprintu
* Účastníci: vývojový tým, uživatelé, zákazníci, management, Scrum Masters,..
* Definuje se cíl Sprintu

#### Sprint Backlog

* Vývojový tým definuje Sprint backlog
* Seznám úloh nutných pro dosažení cíle
* Je implementačně orientovaná, rozšířená podmnožina Product backlog
* Jednotlivé položky jsou rozděleny mezi týmy

### Sprint Development – vývoj

* Analýza, návrh a implementace požadavků plynoucích z cíle Sprintu a úloh definovaných v Sprint backlog
* Pro efektivní řízení aktivit se konají setkání týmů

#### Scrum Meeting

* Každodenní, 15 minutová setkání týmu
* Účastní se členové týmu, Scrum Master, management
* Základní otázky
  + Co bylo uděláno od posledního setkání
  + Jaké překážky se objevily
  + Co bude uděláno do příští schůzky

### Sprint Review – posouzení

* Na konci každé Sprintu
* Demonstruje se výsledný produkt (inkrement)
* Vyhodnocení dosažených výsledků ve srovnání s cílem Sprintu
* Úprava Product backlog
  + Plně implementované požadavky jsou označeny
  + Nutné úpravy (opravy chyb nebo vylepšení) jsou přidány
  + Změny či nové požadavky jsou začleněny
* Vyhodnocení úsilí, splnění cíle, možné změny architektury systém

## Scum Prces: Post-Game

* Integrace výsledků jednotlivých Sprintů (inkrementů)
* Testování celého systému
* Příprava dokumentace
* Zaškolování uživatelů
* Akceptační testování

## Zhodnocení

### Silné stránky

* Iterativní inkrementální proces
* Časté uvolňování verzí (inkrementů)
* Architektura systému je navržena před procesem vývoje
* Požadavky se ladí během celého vývoje
* Zapojení uživatelů
* Jednoduchý proces

### Slabé stránky

* Nedefinuje přesný postup úloh
* Integrace až po vytvoření všech inkrementů
* Předpoklad, že přímá komunikace je vhodná pro všechny typy projektů
* Nepředepisuje modely pro návrh, často se od Project backlog přechází na implementaci

# Crystal

* Rodina metodologií – různé projekty potřebují různé přístupy
* Základní přístupy a techniky sdílí s XP – ale na rozdíl od XP není tak přísně disciplinovaný proces
* Obecně je méně produktivní než XP – ale více lidí je schopno tento proces lépe akceptovat

## Kategorie projektů

* Projekty jsou rozděleny do kategorií podle kritičnosti, důležitosti a velikosti
* Kritičnost (důležitost) – jaké jsou ztráty pokud systém selže
  + Comfort (C)
  + Discretionary Money (D)
  + Essential Money (E)
  + Life (L)
* Velikost – maximální počet lidí zapojených do vývoje

## Složitost projektů

* Větší projekt potřebuje komplexnější metodiku a lepší koordinaci
* Kritičtější projekt (criticality) potřebuje rigoróznější (preciznější) postup

## Kategorie metodik

* Metodiky patřící do rodiny Crystal jsou přiřazeny do kategorií podle své velikosti
* Kategorie metodik se označují barvou (seřazeno od nejméně náročné)
  + Clear – C6, D6
  + Yellow – C20, D20, E20
  + Orange – C40, D40, E40
  + Red – C80, D80, E80
  + Maroon, Blue, Violet
  + Další mohou být přidány

## Zhodnocení

### Silné stránky

* Iterativní inkrementální proces
* Časté uvolňování verzí
* Proces se ladí na základě zpětné vazby
* Požadavky se ladí během celého vývoje
* Zapojení uživatelů
* Průběžná integrace

### Slabé stránky

* Nedefinuje jasný společný proces
* Není vhodný pro vysoce kritické projekty
* Příliš velká závislost na přímé komunikaci, nepodporuje distribuované týmy

# Je RUP agilní metodikou?

* Základní vyjadřovací prostředek je UML
* Pracuje v iteracích
* Definuje obsah každé iterace
* Definuje pracovní rámec (Framework)

### Použití RUP

* Klasický heavyweight proces
* Agilní proces

### Agile Unified Process

* Zjednodušená verze RUP (blíží se UP)
* Hlavní modelovací jazyk je UML, ale není omezeno
* Modelování bez limitů může ohrozit agilnost metodiky

### dX proces

* Minimální RUP proces
* Považuje UML za jeden z možných pomocných prostředků

# Metodiky

## Srovnání přístupů z pohledu požadavků

* Jednotlivé typy přístupů k procesu vývoje
  + Se příliš neliší v celkovém úsilí věnovanému tvorbě požadavků
  + Se liší v rozložení tohoto úsilí v čase

## Prediktivní či agilní metodika?

### Kdy agilní?

* Neurčité nebo měnící se požadavky
* Menší nebo neurčitý rozpočet
* Odpovědní a dobře motivovaní vývojáři
* Menší až středně velký vývojový tým (kolem 80 lidí)
* Zákazník, který je ochoten zapojit se do vývoje

### Kdy prediktivní?

* Známé a stabilní požadavky
* Dostatečný rozpočet
* Velký vývojový tým (víc jak 100 lidí)
* Pevný rozsah projektu

## Metodiky v praxi

* Většina metodik může být vytvořena (použita) tak, aby pracovala v nějakém projektu
* Libovolná metodika může vést nějaký projekt k neúspěchu
* Úspěšné týmy používají inkrementální vývoj
* Heavy procesy bývají úspěšné
* Light procesy jsou častěji úspěšné

## SW inženýr a metodiky

* Neexistuje 100 % správná metodika

#### Co musí umět dobrý SW inženýř?

* Vybrat vhodnou metodiku nebo na základě metodik vytvořit scénář vývoje softwaru tak, aby projekt úspěšně dosáhl stanoveného cíle
* Stanovit cíle splnitelné v daném prostředí (cena ,termín dokončení, rozsah, kvalita) a to s ohledem na vývojový tým, který má k dispozici

# Management

* Proces koordinace činností skupiny lidí, který realizuje jednotlivec nebo skupina lidí za účelem dosažení stanovených cílů
* Tyto cíle se nedají dosáhnout jenom prací jednotlivce
* Management se uskutečňuje v rámci projektů
* Proto se zde soustředíme na management softwarových projektů

### Projekt

* Je časově ohraničené úsilí, které se vyvíjí s cílem vytvořit jedinečný výsledek (výrobek nebo službu)

#### Časově ohraničené (úsilí)

* Každý projekt má jednoznačný začátek a konec. Konec projektu je dosažen tehdy, když jsou dosaženy stanovené cíle projektu nebo když se ukáže, že těchto cílů dosáhnout nelze.

#### Jedinečný (výsledek)

* Výsledek projektu se nějak liší od výsledků podobných projektů

## Demingův manažerský cyklus (PDCA)

* Manažerské procesy by měly probíhat v této nekonečné smyčce
* Plánování (plan) – naplánování zamýšleného zlepšení
* Zavádění (do) – realizace plánu
* Ověření (check) – zhodnocení dosažených výsledků
* Jednání (act) – rozhodnutí, jaké další změny provést pro další zlepšení procesu řízení

# Procesy managementu projektu

## Inicializace

* Rozpoznání, že projekt může začít a získání všech relevantních informací potřebných proplánování projektu (např. časový a cenový horizont, základní koncepce projektu, potenciální rizika)
* Trvá několik dní až měsíců
* V některých organizacích je projekt formálně inicializovaný až po ukončení studie vhodnosti, předběžného plánu nebo jiné formy analýzy
* Zdroj nebo stimulace inicializace projektu může být:
  + poptávka na trhu
  + požadavek zákazníka
  + z důvodu prestiže
  + výhody technologie
  + požadavky legislativy

## Plánování

* Vytvoření a udržování plánu pro zabezpečení chodu projektu.
* Definují se požadavky na zdroje, požadavky na práci a definuje se kvalita a kvantita práce.
* Plánování by mělo být tak podrobné, jak je to nezbytné a ne tak, jak je to možné.
* Plánování je intenzivní hlavně v počátečních etapách projektu, v průběhu provádění a řízení se plány upravují podle potřeby.
* Vytvořený plán musí schválit všechny skupiny zapojené do projektu.
* Nedostatky v plánování představují pro projekt značné riziko.

### Důvody pro plánování

* Snížit neurčitost (výsledku projektu)
* Dosáhnout cenovou efektivitu
* Zajistit lepší pochopení cílů projektu
* Vytvořit základnu pro sledování a řízení práce

## Projektový plán

Projektový plán obsahuje:

* Definici cílů, úloh a odpovědností **co** je potřeba udělat, **pro koho** je to potřeba udělat
* Definice požadavků na zdroje, **kdo** to má udělat, **kolik** to bude stát
* Techniky, prostředky, zdroje pro vykonávání plánu, **jak** to udělat
* Kontrolní body, **kdy** je to potřeba udělat
* Definici kvality, základ pro měření postupu projektu
* Stanovení rizik projektu, **co kdy** nastane určitá situace

Plánování staví mosty mezi tím, kde jsme, a kam chceme jít.

## Řízení

* Kontrola a řízení na základě naměřených výkonů (na základě výsledků práce a požadavků na změny)
* Preventivní činnosti s cílem předcházet problémům
* Shromažďují a rozšiřují se informace
  + o stavu projektu (kde se projekt momentálně nachází v porovnání s plánem)
  + o postupu projektu (co se dosud udělalo)
  + o budoucím stavu a postupu projektu (předpověď vývoje projektu)
* Sleduje se stav projektu, porovnává a posuzuje se
  + potup dosažený v posledním období (týden, měsíc) a v obdobích předcházejících
  + dosažené výsledky s tím, co je ještě potřeba udělat
  + odhady a skutečné hodnoty, předpověď budoucích hodnot
  + přiřazení zdrojů (lidé, počítače)
  + poměr dosažených výsledků a času, který uplynul, rozhoduje se, či je dosažení cílů projektu reálné
* Zajišťuje se řízení změn

## Provádění

* Spotřebuje nejvíce času (úsilí) a peněz (u SW projektů jenom při špatném plánu)
* Realizace plánu projektu
* Manažer projektu koordinuje a usměrňuje provádění činností z plánu
  + přidělování úkolů
  + stanovení priorit
  + rozdělování pravomocí
  + sledování postupu prací na projektu
  + rozhodování o umístění důležitých zdrojů
* Provádění projektu nejvíce ovlivňuje problémová doména (oblast), technické parametry řešení a model vývoje
* Vytváří se výsledek (výrobek, služba)

## Ukončení

* Zaznamenají se nové poznatky, zkušenosti a poučení pro budoucí projekty
* Ukončí se kontrakty s dodavateli a dodávky potřebných výrobků a služeb v rámci projektu, vyřeší se také všechny otevřené problémy (závazky a pohledávky) spojené s dodávkami

# Řízení kvality SW projektů

Obvyklý postup při tvorbě softwaru spočívá v

* Co nejrychlejší **implementaci** programu
* Rychlém **testování** a s cílem najít a
* **Odstranit chyby** a nedostatky

V žádném jiném technickém oboru se nevytvářejí výrobky nekontrolovatelné kvality, přičemž vy se spoléhalo na testování.

## Kvalita

* Souhrn vlastností nebo charakteristik produktu či služby, které souvisí s jeho či její schopností splnit explicitně uvedené či implicitně předpokládané potřeby.
* Kvalita není definovaná jako absolutní míra, ale jako stupeň splnění požadavků či potřeb.
* Kvalita je:
  + míra stupně dokonalosti
  + splnění požadavků
  + vhodnost k danému účelu
  + schopnost produktu nebo služby plnit dané potřeby

## Různé pohledy na kvalitu

* Z pohledu **uživatele** může kvalita odpovídat jednoduchosti obsluhy sytému, spolehlivému a efektivnímu provádění jednotlivých funkcí systému
* Z pohledu **provozu** systému kvalita obvykle znamená dobrou provozní dokumentaci a efektivní využití výpočetních prostředků
* Z pohledu **tvůrce a údržbáře** systému kvalita odpovídá čitelným a modifikovatelným programům a srozumitelné a přesné dokumentaci
* Z pohledu **manažera** kvalita obvykle odpovídá dodání výrobku včas, v rámci rozpočtu a dohodnutých požadavků

## Parametry softwarového projektu

V každém projektu existují čtyři základní parametry:

#### Cena

* Nízké náklady: klesá kvalita i rozsah
* Příliš nízké náklady: zadání nelze splnit
* Příliš vysoké náklady: kvalita se nezvýší, čas se nesníží

#### Čas

* Málo času: snižuje se kvalita a rozsah zadání, roste cena
* Příliš mnoho času: oddaluje se zpětná vazba od systému v provozu

#### Kvalita

* Vyšší kvalita: zvyšuje počáteční náklady, do budoucna je snižuje
* Nízká kvalita: nízké počáteční náklady, do budoucna zvyšuje enormně náklady (finanční i lidské)

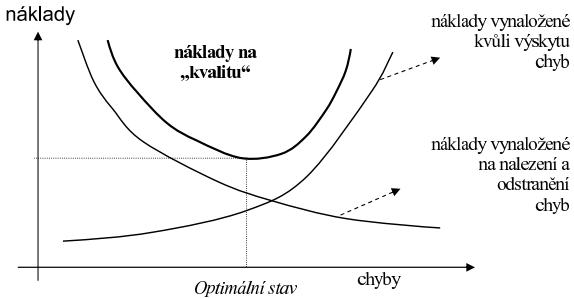
#### Rozsah

* Menší rozsah zadání: možnost vyvíjet rychle, levně a s lepší kvalitou
* Žádný účastník nemůže definovat všechny parametry
* Pro heavyweight metodiky platí, že zákazník často volí rozsah a případně kvalitu, vývojový tým pak určí cenu a čas
* Pro agilní metodiky platí, že zákazník často volí cenu, čas a kvalitu, vývojový tým pak určí rozsah

### Alternativní pohled

* Funkcionalita = rozsah + kvalita
* Čas, cena a funkcionalita (rozsah + kvalita)

## Funkcionalita, chyby a kvalita

* Nepleťte si kvalitu s funkcionalitou!
* Systém s řadou funkcí může mít nízkou kvalitu (např. příliš chyb) a naopak.
* Software bez chyb != kvalitní software

## Normy pro systém zajištění kvality

* Pro softwarové produkty se vychází z následujícího předpokladu:
* Pokud má organizace kvalitní proces tvorby výrobku (softwaru), budou i její výrobky kvalitní.
* Tento přístup se používá hlavně proto, že v softwarovém inženýrství není jednoduché měřit kvalitu programů pomocí nějaké výstupní kontroly.
* Kvalitní proces => kvalitní výrobek

## Capability Maturity Model – CMM

### Účel

* Vyhodnocení schopnosti (U.S.) vládních dodavatelů splnit softwarové projekty
* Zaměřuje se na procesy

### Maturity Model

* Nástroje popisující, jak dobře jsou nastaveny praktiky, procesy a chování organizace, jak kvalitně mohou dosáhnout požadovaných výstupů
* Lže použít jako prostředek pro srovnání procesů různých organizací a pro porozumění těmto procesům

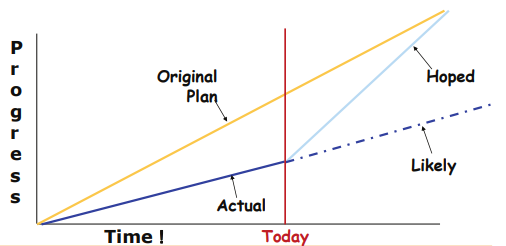
### Struktura modelu

* Úrovně zralosti (Maturity Levels) – 5 úrovní, nejvyšší stupeň reprezentuje ideální stav
* Klíčové oblasti (Key Process Areas) – soubory souvisejících aktivit pro dosažení stanovených cílů
* Cíle (Goals) – cíle definují rozsah, omezení a záměry klíčových oblastí
* Vlastnosti (Common Features) – praktiky pro začlenění klíčových oblastí do procesů organizace
* Klíčové praktiky (Key Practices) – popisují praktiky a elementy infrastruktury, které přispívají k efektivnímu začlenění oblastí

### Úrovně

* 0 – Neexistující řízení. Procesy a jejich řízení je zcela chaotické.
* 1 – Počáteční (Initial). Procesy jsou realizovaný ad hoc, organizace js schopna použít nové či nedokumentované procesy.
* 2 – Opakovatelné (Repeatable). Procesy jsou dostatečně dokumentované a umožňují opakování stejných kroků.
* 3 – Definované (Defined). Procesy jsou definovány a potvrzeny jako standardní procesy.
* 4 – Řízené (Managed), Procesy jsou vyhodnocovány na základě předem stanovených metrik.
* 5 – Optimalizující (Optimizing). Řízení procesu zahrnuje i inovační cyklus, lze optimalizovat a zlepšovat procesy.

# Měření v softwarovém inženýrství

* Měření je proces přiřazování hodnot k vlastnostem entit reálného světa
* Měření zvyšuje pravděpodobnost, že i přes nejistotu uděláme dobré rozhodnutí
* Každé měření musí mít svůj účel (cíl)

## Typy měření

### Přímé měření

* Přímé získaní hodnoty sledovaného atributu (např. počet řádků programu)

### Nepřímé měření

* Odvození z jiných atributů, které lze měřit přímo (např. udržovatelnost lze určit jako čas potřebný pro odstranění chyby)

Jsou možné různé interpretace naměřených hodnot (např. počet chyb nalezených za jednotku času reprezentuje kvalitu testování nebo (ne)spolehlivost programu)

* Pro úspěch projektu je důležitá dohoda na kritériích přijetí projektu. Musejí být měřitelná.

## Metriky pro výrobek

### Velikost, rozsah (pro odhad času a nákladů a měření produktivity)

* počet řádků zdrojového textu programu (LOC)
* počet modulů
* průměrný počet LOC na modul
* rozsah dokumentace (počet stran)

### Modularita

* svázanost modulů (počet toků údajů a řízení mezi moduly a počet globálních datových struktur)
* soudržnost modulů

### Spolehlivost

* MTTF – střední doba do následujícího výpadku
* MTTR – střední doba opravy

### Dostupnost

* Pravděpodobnost, že v daném čase program pracuje správně
* Dostupnost = 100 \* MTTF / MTBF [%]

### Složitost

* počet souborů
* velikost programu (počet příkazů, řádku..)
* počet větvení
* hloubka zanoření řídících struktur
* počet cyklů
* průměrná délka věty v dokumentaci

### Chyby

* počet chyb a nedostatků
* chybovost = počet chyb / kLOC
* klasifikace chyb a nedostatků a frekvence jejich výskytu

### Udržovatelnost

* střední doba potřebná na opravu chyby
* střední doba potřebná na pochopení logiky modulu
* střední doba na nalezení příslušné informace v dokumentaci

## Určení spolehlivosti a dostupnosti

MTTF = (3 + 3 + 5 + 7) / 4 = 4,5 dne

MTTR = (2 + 3 + 1 + 1) / 4 = 1,75 dne

MTBF = MTTF + MTTR = 25 / 4 = 6,25 dne

dostupnost = MTTF / MTBF = 18 / 25 = 72 %

* Poznámka: plánované odstávky z důvodu údržby se do výpadků nepočítají.

## Další používané metriky

### Metriky pro proces

* Úsilí – čas vynaložený na vývoj systému (člověko-měsíce)
* Změny požadavků (odráží kvalitu specifikace požadavků)
  + počet změn požadavků
  + střední doba od dokončení specifikace do požadavku na změnu
* Náklady a čas
  + začátky a konce činností
  + délka trvání činností
  + náklady na provedení jednotlivých činností

### Metriky pro zdroje

* Charakteristiky personálu (produktivita, velikost týmu, rozsah a způsob komunikace, zkušenosti), zatížení sítě

# Softwarový tým

### Role v softwarovém týmu

* Analytik, návrhář, programátor, odborník na testování, odborník na údržbu, auditor, management, podpůrný personál

## Analytik vs. Programátor

### Analytická profese

* Analytici vytvářejí cíle projektu, zpracovávají specifikační dokumenty a zajišťují jejich kontrolu a odsouhlasení zákazníkem
* Vyžaduje diplomatické vlohy při jednání s lidmi (např. zákazníky)
* Mezilidské vztahy jsou většinou komplikované
* Analytická fáze nemá jasné ohraničení

### Programátorská profese

* Programátoři navrhují technické řešení systému, implementují, ladí a testují komponenty
* Práce má předem daný cíl
* Mezilidské vztahy nejsou většinou komplikované
* Výsledky práce jsou okamžitě zřejmé

## Jak poskládat tým

* Správně poskládaný tým připomíná družstvo v kolektivním sportu
* I při vývoji softwaru je týmový duch a schopnost komunikace důležitější než kvalita nespolupracujících individualistů
* Problematické typy v týmu:

### Beran

* Prosazuje (příliš důrazně) variantu, kterou považuje za nejlepší. Nejnebezpečnější je, když se v týmu sejdou dva berani.
* Řešení: rozdělit odpovědnost

### Slabý článek

* Jeden člen týmu může znehodnotit úsilí všech
* Řešení: nepřijmout do týmu nebo zařadit na méně náročnou pozici

### Dělnická mentalita

* Mají sice dostatečné zkušenosti, ale snaží se přežít pracovní dobu s vynaložením co nejmenší námahy.
* Řešení: s lehčími případy se dá smířit, těžší se snažit propustit, pečlivě jim přidělovat a kontrolovat dobře měřitelnou práci

### Snaživec

* Odvádí na své pozici dobrou práci, ale chce na prestižnější místo, na které nemusí mít předpoklady
* Řešení: kompromis – najít pozici, která by snaživce (alespoň částečně) uspokojila, ale kdy by nemohl ohrozit projekt

## Vhodná velikost týmu

* Menší tým pracuje efektivněji
* Větší tým spotřebuje více času na vnitřní komunikaci
* Větší tým může projekt dokončit dříve
* Větší tým díky většímu objemu znalostí může najít lepší řešení

### Pravidlo jedné místnosti

* Tým pracující v jedné místnosti lze řídit neformálně, protože si lidé mezi sebou všechno mohou říct.
* Porušení tohoto pravidla přidáním jednoho nebo dvou pracovníků nemusí vyvážit úsilí spojené s nutným formálnějším řízením.
* Pro větší týmy je nutné zavést písemnou komunikaci přiměřenou velikosti a struktuře týmu.

## Jak tráví čas programátoři?

* Psaní programů - 13 %
* Čtení programů a příruček - 6 %
* Komunikace týkající se práce (konzultace) - 42 %
* Ostatní (včetně osobních věcí) - 39 %

## Efektivnost komunikace

## Vedoucí vývojového týmu

### Manažeři a obchodníci

* Pro svoji práci technické znalosti příliš nepotřebují. Naopak je pro ně životně důležitá schopnost jednat s lidmi.

### Vedoucí vývojového týmu

* Má na starosti technickou realizaci projektu. Ke své práci kromě komunikačních schopností proto potřebuje také značné technické znalosti.

## Přístupy lidí v praxi

* Vývoj podle předpisů – můžeme říci, že neúspěch není naše chyba
* Lidé preferují konzervativní přístup a neúspěch než riskovat úspěch s odlišnou metodou
* Libovolná technika návrhu komplikovanější než CRC cards vypadá příliš složitě na použití
* Každá nová technika návrhu vypadá příliš složitě na použití
* Návrhové týmy ignorují nástroje a techniky, které nemají rádi
* Lidé pracují (učí se) dobře podle příkladů

# Motivace lidí

## Lidský operační systém

**Motivace 1.0** - biologický hnací pohon (hlad, žízeň, sex, přežití)

**Motivace 2.0** – vnější motivace (odměny a tresty, cukr a bič)

**Motivace 3.0** – vnitřní motivace (nabídnout alternativu k programům typu MS Word)

### Vnější motivace (TYP X)

Nefunguje, protože:

* ničí vnitřní motivaci
* snižuje výkonnost
* potlačuje kreativitu
* podněcuje neetické chování
* podporuje krátkodobé zaměření

Ale funguje pro – rutinní, nudné úkoly, kde se kreativita stejně nevyužije

### Vnitřní motivace (TYP I)

Tři základní prvky

* **Autonomie** – možnost rozhodovat, co, kdy a s kým dělat
* **Mistrovství** – zaujetí stát se lepším v něčem, na čem záleží, stav proudění
* **Smysl** – maximalizace smyslu jednání, poslání přesahující člověka

# Ochrana intelektuální vlastnictví

Vlastnictví představuje neomezené a výlučné právo k věci, právo disponovat věcí jakýmkoli právem povoleným způsobem, totiž držet ji, užívat ji a vyloučit každého, kdo by v tom oprávněného rušil.

* **Rajská zahrada** – všeho je dostatek => vlastnictví nedává smysl
* **Robinson na pustém ostrově** – jediný uživatel => není s kým se přít
* **Robinson s Pátkem** – spory o vzácné zdroje
  + násilné řešení – silnější bere vše
  + mírové řešení – stanoví a dodržují se rozumná pravidla, například:

1. Vlastnictví sebe sama – člověk je svobodný
2. Prvotní přivlastnění – kdo najde volný zdroj, stává se jeho vlastníkem
3. Výroba – vlastník může svůj zdroj přeměnit prací na výrobek, jehož je také vlastníkem

Vlastnictví je tedy nenásilné řešení sporů o vzácné zdroje. Vede k rozvoji směny, dělby práce a je předpokladem nárůstu blahobytu.

## Intelektuální vlastnictví

**Vlastnictví hmotných zdrojů** je dobře pochopitelné, jeho zavedení je přirozené a je pro společnost prospěšné.

**Intelektuální vlastnictví** je sporné, protože nehmotné zdroje (myšlenky) postrádají atribut vzácnosti. Spory se také vedou o to, jestli je prospěšné pro celou společnost.

## Znalost ochrany intelektuálního vlastnictví

Nezávisle na tom, jestli s ochranou intelektuálního vlastnictví souhlasíte nebo ne, měli byste znát její principy, protože:

* Budete **vytvářet** díla, na která se ochrana intelektuálního vlastnictví vztahuje. Její znalost potřebuje k ochraně svého díla před zneužitím.
* Budete **používat** intelektuální vlastnictví druhých. Musíte vědět, co použít smíte a co ne, abyste nebyli potrestáni.
* Jestliže se vám něco nelíbí a chcete to **změnit**, musíte tomu rozumět.

## Ochranná známka (Trademark)

* Označení grafického znázornění tvořené zejména slovy, písmeny, číslicemi, barvou, kresbou nebo tvarem výrobku či jeho obalu určené k rozlišení výrobků nebo služeb
* Základní funkce:
  + **Rozlišovací** – odlišuje výrobce nebo poskytovatele služeb
  + **Ochranná** – brání třetím osobám používat stejné označení
* V ČR se přihlašuje u Úřadu průmyslového vlastnictví
* Platnost 10 let (lze prodlužovat o 10 let)
* Např. Microsoft Windows, Coca-Cola

## Obchodní tajemství

* Veškeré skutečnosti obchodní, výrobní či technické povahy související s podnikem, které mají skutečnou nebo alespoň potenciální materiální či nemateriální hodnotu, nejsou v příslušných obchodních kruzích běžně dostupné, mají být podle vůle podnikatele utajeny a podnikatel odpovídajícím způsobem jejich utajení zajišťuje.
* Např. recept na výrobu Coca-Coly, složení Becherovky

## Patent

* Historie patentů sahá do 15. století, kdy byly první patenty udělovány formou monopolu na určitý výrobek nebo službu
* Každý stát má svůj patentový úřad, který může udělovat patenty s platností pro území daného státu
* Funkci patentového úřadu v ČR plní Úřad průmyslového vlastnictví
* Pro udělení patentu je nutné podat přihlášku vynálezu na patentovém úřadě, který žádost prozkoumá. Rozhodující je čas podání přihlášky. Aby technické řešení bylo uznáno vynálezem, musí být:
  + nové ve srovnání se světovým stavem techniky a nezveřejněné
  + výsledkem vynálezecké činnosti
  + průmyslově využitelné
* Patent udělený v ČR platí 20 let
* Souhlas k využití patentu se uděluje licenční smlouvou
* Patent lze prodat

## Další typy ochrany

* Užitný vzor
  + Jednodušší, rychlejší a levnější alternativa k patentům
  + Zkoumá se jen splnění základních podmínek pro ochranu
  + Nezkoumá se novost a tvůrčí úroveň předmětu přihlášky
  + Maximální doba ochrany je 10 let
* Topografie polovodičových výrobků
* Průmyslové vzory – ochrana vzhledu výrobku (design)
* Označení původu / zeměpisná označení