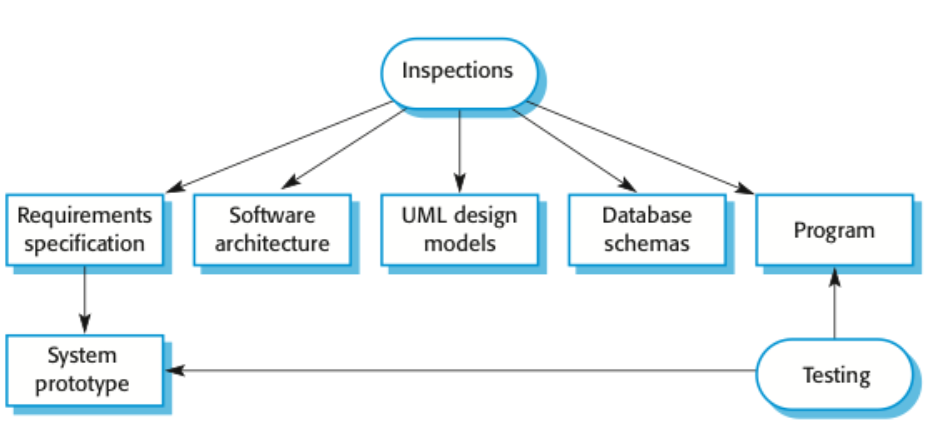
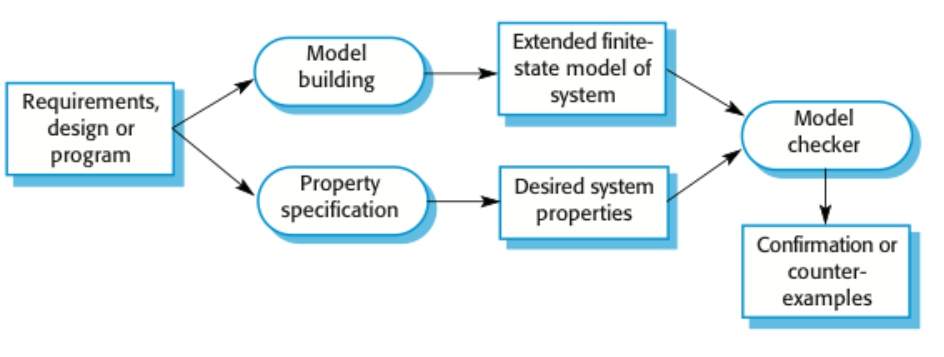
Softwarové inženýrství II

Testování, verifikace a validace. Provoz, údržba a další vývoj systému. Role jazyka UML v podpoře analýzy a návrhu SW.

# Testování

* Má ukázat co program má dělat a objevit chyby před releasem programu
* Dokáže odhalit chyby, ale nezaručuje jejich absenci
* Část verifikačního a validačního procesu
* Cíle
  + Validační testování
    - Otestovat, že software splňuje požadavky
    - (takže pro každý požadavek by měl existovat alespoň jeden test)
    - Test projde => systém operuje jak má
  + Chybové testování (Defect testing)
    - Má za úkol odhalit chyby v programu (program se nechová správně / dle požadavků)
    - Test projde => systém obsahuje chybu
* Validace vs. Verifikace
  + Validace = Ověření, že program splňuje požadavky (high level user expectations / user requirements)
  + Verifikace = Ověření, že program neobsahuje chyby, že splňuje low-level programátorské požadavky
* Statická vs. dynamická verifikace
  + Statická verifikace – zabývá se analýzou statické části programu (= kód)
    - Např. inspekce v IDE
  + Dynamická verifikace – zabývá se chováním programu
    - Software testing
  + 

# Statická analýza

* Nezahrnuje spuštění programu
* Zahrnuje např.
  + Inspekce
  + Automatickou analýzu programu
  + Formální verifikaci
* „Static analysis has its value whenever i tis cheaper to find and remove faults than to pay for systém failure“
* Inspekce
  + Podle definice předmětu jsou prováděné manuálně
    - These involve people examining the source representation with aim of discovering anomalies and defects
  + Inspekce můžou existovat pro prakticky cokoli, od zdrojového kódu až po uživatelské rozhraní
  + Dají se aplikovat na vyvíjený systém
  + Nejsou ovlivněné chováním programu (jedna runtime chyba může ovlivňovat druhou)
  + Kromě vyložených chyb můžou odhalit nedodržování standardů
* Automatická statická analýza
  + Automaticky kontroluje zdrojový kód programu a snaží se objevit chyby
  + Úrovně
    - Characteristic error checking
      * Rozpoznávání charakteristických konstrukcí, které vedou k chybám
    - User-defined error checking
      * Kontrola uživatelsky zavedených pravidel
    - Assertion checking
      * Program obsahuje „formal assertions“, které musí platit
      * The static analyzer symbolically executes the code and highlights potential problems
  + Statická analýza dokáže odhalit velké množství chyb, zvlášť pokud jde o slabě typovaný jazyk
  + Může odhalit bezpečnostní chyby (nesanitizované vstupy apod.)
* Verifikace a formální metody
  + Katka, oprav mě prosím až budu plácat hovadiny
  + Může se použít, pokud existuje matematická specifikace
  + Může se použít v různých stádiích vývoje
  + Samotný vývoj matematického modelu může objevit chyby
  + Zvládá analyzovat paralelní systémy, kde může objevit race conditions / deadlocky (testování paralelních systémů je obecně složité)
  + Zvládá odhalit chyby ještě před testováním
  + Nicméně je to časově a finančně náročné. Matematický model / důkaz může obsahovat chybu
* Model checking
  + Vytvoří se „extended finite state model of a system“, který se otestuje pomocí model checkeru
  + 
  + Model checker prozkoumá všechny možné cesty modelem a zkontroluje, že požadavky platí
  + Opět velmi užitečné pro paralelní systémy, které se obecně těžko testují
  + Výpočetně náročné, ale prakticky použitelné pro malé-střední systémy s kritickou funkčností

# Testování v různých fázích

## Development testing

* Testuje ten samý tým, co program vyvíjí
* Unit testing
  + Zaměřuje se na testování (malých) jednotek kódu
  + Odhaluje chyby (defect testing proces)
  + Může testovat
    - Individuální funkce
    - Objekty
    - Komponenty
  + Dají se automatizovat a měly by se automatizovat
  + Standardní testovací frameworky ála JUnit
  + Struktura testu
    - Nastavení prostředí
    - Zavolání testované jednotky se zvolenými parametry
    - Ověření výstupu (návratové hodnoty apod.) s očekávaným výsledkem
  + Testovací strategie
    - Partition testing
      * Rozdělí možné vstupy do kategorií, pro které se program chová stejně
        + Pro násobení:
        + Násobení dvou kladných čísel
        + Násobení dvou záporných čísel
        + Násobení záporného a kladného čísla
        + Násobní nulou
      * Otestovat by se měla každá taková kategorie
    - Guideline-based testing
      * Testování chyb, kterých se programátoři běžně dopouští
      * Otestovat všechny chybové vstupy
      * Otestovat vstupy, které způsobí overflow
      * Opakovat jeden vstup několikrát za sebou
      * Donutit systém vygenerovat neplatný výstup
      * Donutit systém pracovat s moc malými/velkými výpočty
* Component testing
  + Komponenta = několik interagujících objektů
  + K fukncionalitě propojených objektů se přistupuje přes rozhraní komponenty
  + A takové rozhraní se musí chovat podle očekávání
    - Musí správně pracovat s ostatními komponentami
    - Musí si poradit s různě rychlými komponentami
    - Musí volat ostatní komponenty ve správném pořadí
  + Guidelines
    - Testovat edge cases
    - Testovat předávání NULL (fuj, tohle je tak založené na Javě)
    - Pokud komponenta předává zprávy, snažit se ji zahltit
    - In shared memory systems, vary the order in which components are activated
* System testing
  + Testování celku zaměřené na interakce mezi jednotlivými komponentami
  + Prakticky většinou nelze otestovat veškeré možné chování a vyvíjí se „testing policies
    - Musí být otestované funkce přístupné z menu
    - Funkce, které si berou vstup od uživatele musí být otestované s validními i nevalidními vstupy…
* Test-driven development (TDD)
  + Nejdřív se napíšou testy, až potom se píše kód
  + Kód musí postupně splnit všechny testy
  + Výhody
    - Všechen kód má pak alespoň jeden test
    - Existující testy dokáží odhalit chyby ve změnách
    - Zjednodušuje hledání chyb
    - Slouží jako forma dokumentace

## Release testing

* Testuje se verze programu určená pro vydání
* Primární cíl je přesvědčit dodvatele, že je systém použitelný
  + => ukazuje, že systém dovává specifikovanou funkcionalitu, výkon a že nepadá
* Forma System testing, ale s rozdíly
  + Testuje jiný tým
  + Validation testing – testuje, že program jako celek splňuje požadavky
    - Program sice může běžet úplně perfektně a bez chyb, ale to neznamená, že poskytuje všechnu funkcionalitu
* Requirements based testing
  + Prozkoumají se požadavky a pro každý se napíše test

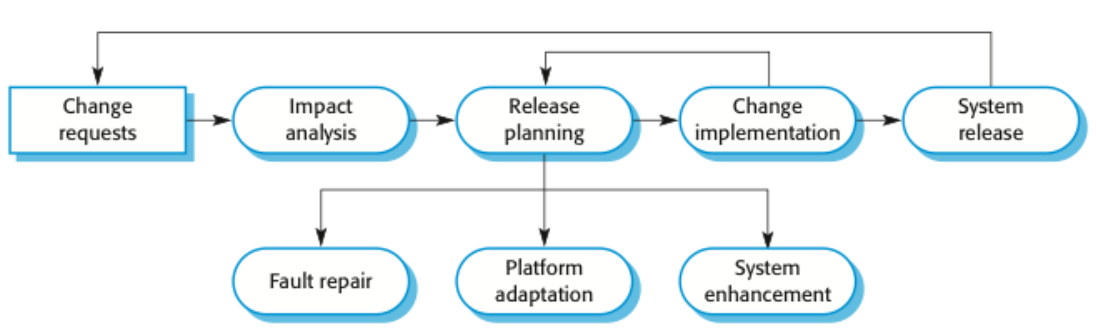
## User testing

* Koncoví uživatelé systému testují a dodávají zpětnou vazbu
* Důležité, i pokud je systém testován během vývoje a release (pokud mám skvělou službu, kterou nikdo nedokáže používat, je mi k ničemu)
* Typy
  + Alpha testing
    - Uživatelé spolupracují s vývojovým týmem a testují produkt ve vývojovém prostředí (eg. na počítačích vývojáře)
  + Beta testing
    - Udělá se relase uživatelům, kteří ho otestují v domácích podmínkách
  + Acceptance testing
    - Zákazník otestuje systém, rozhodne se, jestli je spokojený a jestli se má systém spustit
    - Systematická aktivita s definovanými kritérii, plánem

# Testování nefunkčních požadavků

* Performance testing
  + Release testing může zároveň testovat výkon
  + Testy by měly simulovat reálné používání systému
  + Většinou se napíše testovací scénář a pak se postupně zvyšuje zátež
  + Stress testing = záměrné přetížení systému, ať se otestuje jeho chování
* Reliability testing
  + Testuje spolehlivost
  + Zaměřuje se na validní vstupy
  + Problémy
    - Test může být špatně navržený a systém se v reálu používá jinak
    - Statistická nepřesnost
      * Selhání musí být hodně, aby začala být statisticky významná
      * Jenže dobře postavený systém jich má jen velmi málo
    - Testovací data nemusí být jednoduché připravit
* Security testing
  + Testuje, jestli systém přežije útoky
  + Pomáhá s ním statická analýza
  + Nicméně
    - Testování nemusí být kompletní, nemusí odhalit všechny chyby

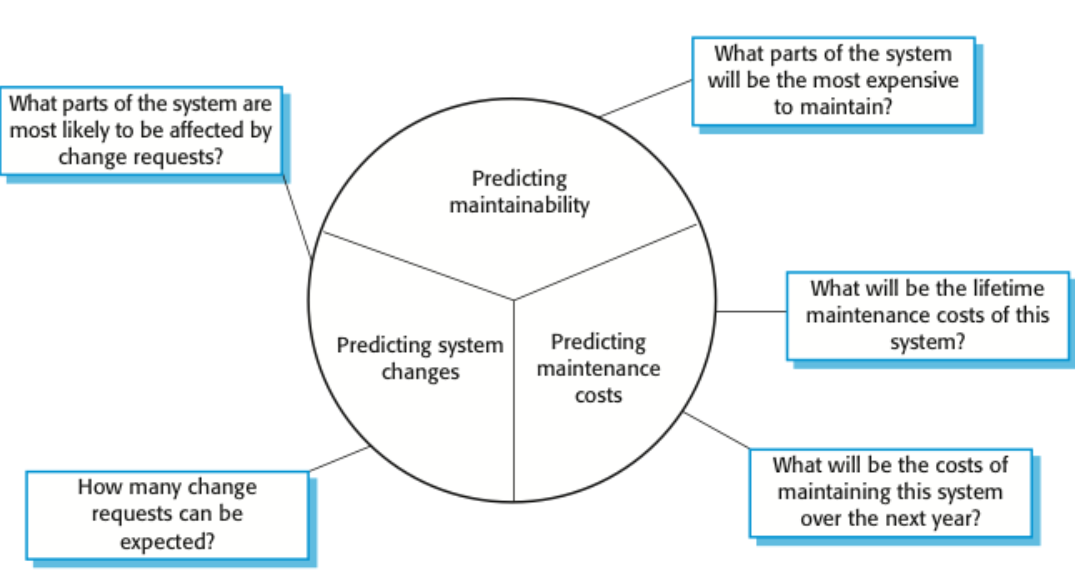
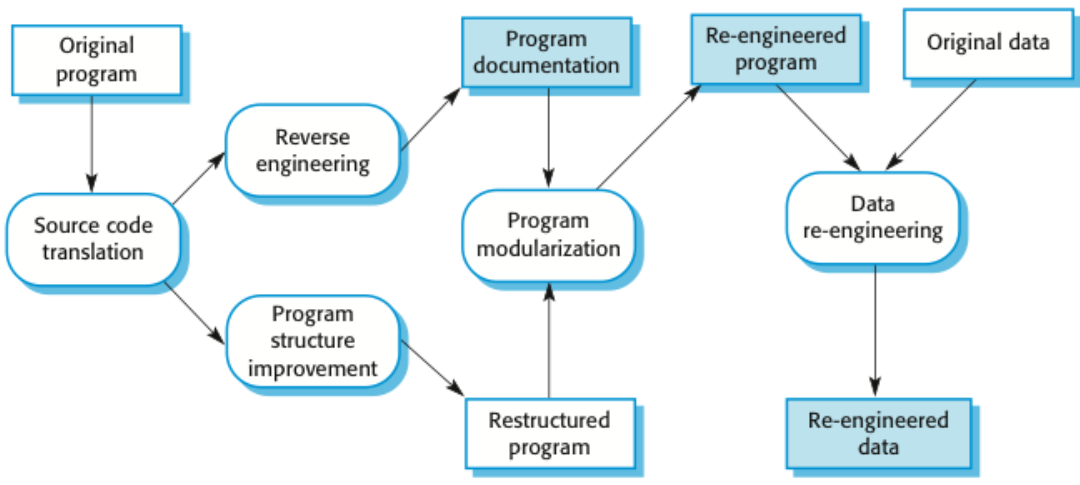
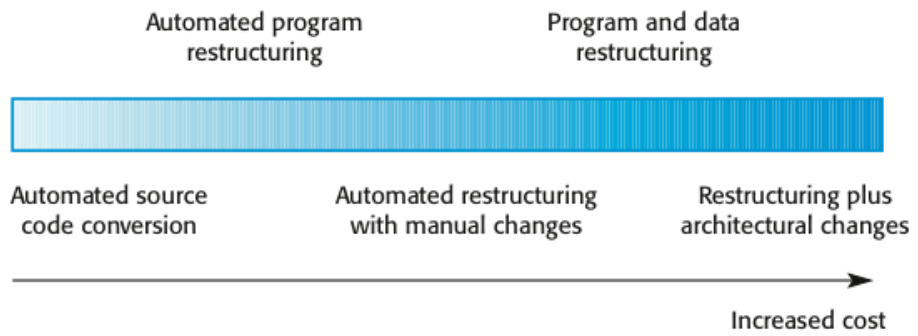
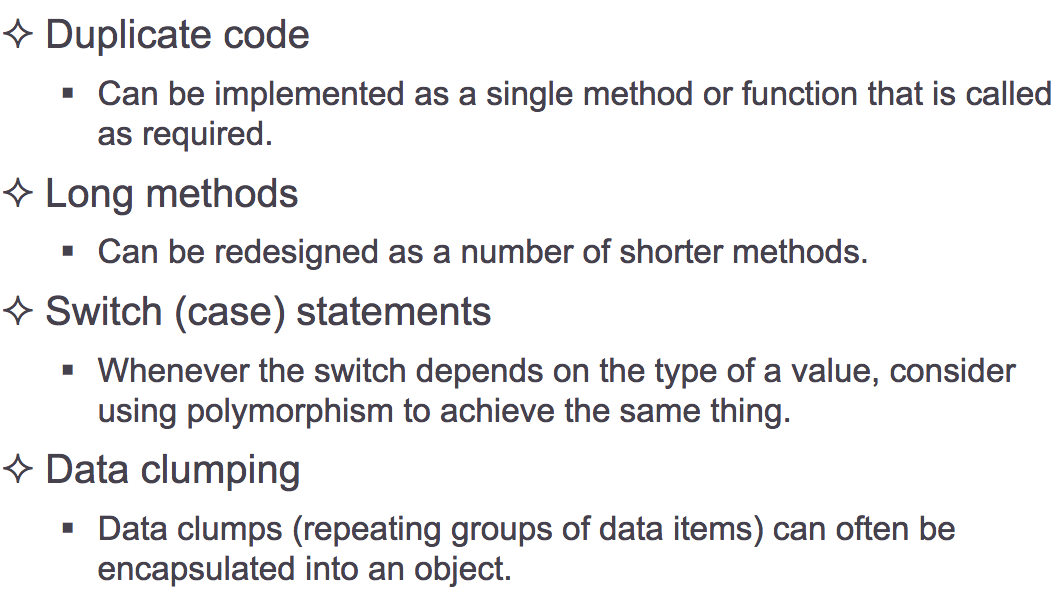
# Další vývoj systému

* Změna požadavků je nevyhnutelná
  + Z používání programu vyplnynou nové požadavky
  + Změní se prostředí
  + Je potřeba opravit chyby
  + …
* => potřeba dalšího vývoje
* Ideálně  
  
* Implementace změny = další vývojová iterace
  + Nemusí ji dělat originální tým
  + Je potřeba dobře porozumět, jak změna ovlivní zbylé části systému
* Urgentně potřebné změny se často implementují ve zrychleném řízení (hotfixy)
  + Může dojít k vytvoření technického dluhu
* Agilní metodiky vývoje si se změnou poradí jednoduše, jde pouze o další iteraci
* Automatické testy jsou při implementaci změn zvlášť užitečné (pokud nová funkčnost ovlivní jiné, neměnné funkčnosti, testy by to měly zachytit)

## Lehman’s laws

* „zákony“ založené na pozorování
* Aplikovatelné na velké „tailored“ systémy vyvíjené velkými organizacemi
* Zákony
  + Continuing change
    - program se musí měnit, jinak se postupně stává méně užitečným
  + Increasing complexity
    - přidáváním změn se systém komplikuje
    - extra zdroje musí být vynaložené na zjednodušení struktury systému
  + Self regulation
    - evoluce je samo regulující proces
    - Systémové atributy jako čas mezi releasy a počet objevených chyb se moc nemění
  + Organizational stability
    - Rychlost vývoje je přibližně konstantní a nezávislá na zdrojích vyhrazených pro vývoj
  + Conservation of familiarity
    - Změny v jednotlivých releasech by měly být přibližně stejně velké, aby je uživatel zvládnul vstřebat
  + Continuing growth
    - Funkcionalita musí uspokojit uživatelovy potřeby
  + Declining quality
    - Kvalita systému se snižuje, pokud nejsou změněny tak aby reflektovaly jejich aktuální prostředí
  + Feedback system
    - Vývoj obsahuje několik forem feedbacku, na který se musí hledět, pokud se má program zlepšovat

# Údržba

* = modifikace programu po vydání
* Termín se používá pro změny vlastního software, pro generické produkty se používá „evolution“ a vydávají nové releasy
* Údržba většinou nezahrnuje velké změny architektury
* = změny existujících komponent a přidávání nových
* Typy
  + Corrective = bug fixing = oprava bugů
  + Adaptive = adaptace programu, aby zvládal běžet v nových prostředích (eg. na nové verzi windows)
  + Evolutionary = změna/přidání funkcionality
* Cena údržby bývá větší než cena vývoje
  + A s časem a úpravami roste
  + Ovlivněna
    - Stabilitou týmu
    - Schopnostmi týmu
    - Věkem software
      * S postupem času se jejich struktura většinou stává méně srozumitelnou (používají se jiné techniky apod.) a špatně se pak upravují
* Provádí se odhady potřebné údržby a její ceny
  + Které části sytému budou dělat problémy a kolik to bude stát
  + Doba implementace potřebných změn záleží na udržovatelnosti systému
  + Implementace změny většinou degraduje systém a snižuje jeho udržovatelnost
* Odhad údržby (maitenance prediction)  
  
  + Odhad změn (change prediction)
    - Odhad počtu změn (vyžaduje hlubokou znalost vazeb v systému)
    - Úzce svázané systémy se musí hodně měnit při každé změně
    - Faktory ovlivňující složitost změn
      * Počet a komplexita rozhraní
      * Počet měnících se požadavků systému (volatile system requirements)
      * Jak se software používá
  + Na základě komplexity
    - Odhad potřebné údržby lze provést na základě zhodnocení komplexity systémových komponent
    - Většinu údržby potřebuje jenom malé množství komponent
    - Komplexita závisí na
      * Komplexitě kódu
      * Komplexitě použitých datových struktur
      * Velikosti jednotek kódu (funkcí, objektů…)
  + Na základě údržby samotné (proces metrics)
    - Faktory
      * Počet požadavků na bug fixy
      * Průměrný čas strávený na analýze dopadu změny (impact analysis)
      * Průměrný čas strávený implementací požadované změny
      * Počet vybočujících požadavků na změnu (outstanding change requests)
    - Pokud některý z faktorů roste, může to znamenat zmenšující se udržovatelnost
* System reengineering
  + Restrukturalizace nebo přepsání části systému bez změny jeho funkcionality
  + Možnost pokud se některé části systému často mění / často musí probíhat údržba
  + Snaží se takové části zjednodušit pro údržbu, odstraňuje technický dluh
  + Bývá výhodnější než vyvíjení nového software
    - Je rychlejší a většinou levnější
    - Psaní nového software = problémy se sháněním kvalifikovaných programátorů, problémy se specifikacemi, …
  + Co může obnášet
    - Přepsání do jiného jazyka
    - Reverse engineering existujícího programu
    - Restrukturalizaci kódu
    - Modularizaci částí programu
    - Restrukturalizaci a pročištění systémových dat
  + 
  + Faktory ovlivňující cenu
    - Kvalita upravovaného software
    - Absence / přítomnost nástrojů ulehčujících reengineering
    - Rozsah potřebných (datových) změn
    - Cena za lidi  
      
* Preventivní údržba = refactoring
  + Změny v programu, mají za úkol zpomalit degradaci kvality
  + Redukuje budoucí problémy
  + Má za úkol zlepšit strukturu, redukovat komplexitu a dělat kód čitelnějším (ideálně všechno dohromady)
  + Refactoring by neměl přidávat funkcionalitu
  + „preventivní údržba“
* Reengineneering vs refactoring
  + Reengineering
    - Dochází k němu až potom, co systém nějakou dobu běží a náklady na jeho údržbu se zvyšují
    - Používá automatizované nástroje za účelem vytvoření udržovatelnějšího programu
  + Refactoring
    - Neustálý proces zlepšování pomocí vývoje / evoluce
    - Snaží se vyhnout degradaci kódu a struktury
* Pozor na  
  

# Údržba legacy systémů

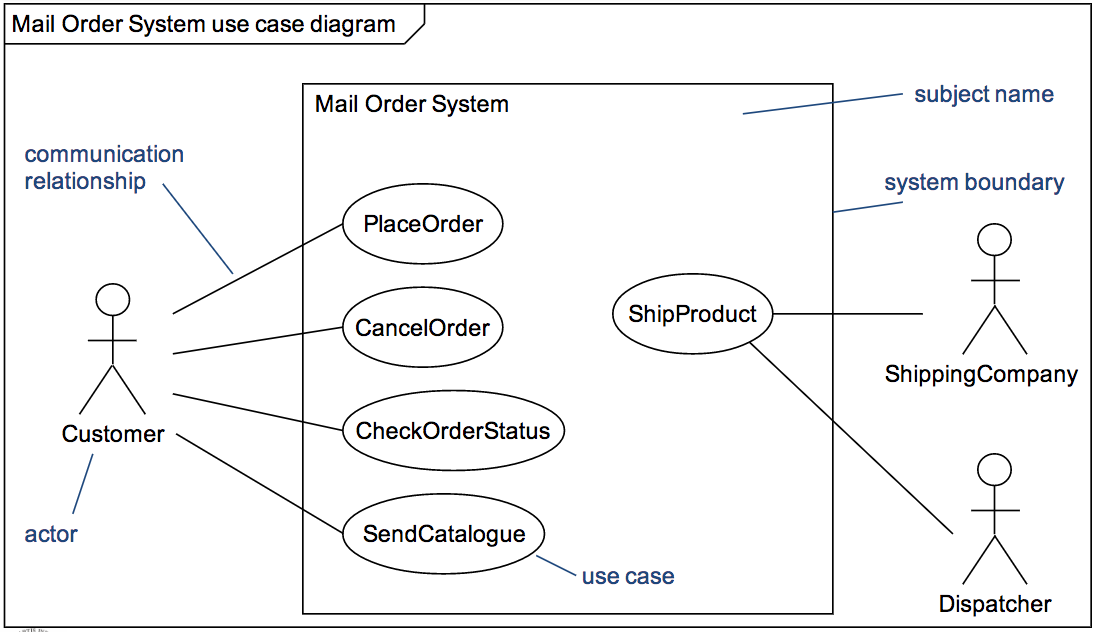
* Organizace používající staré systémy si musí zvolit strategii rozvoje
  + Zahoď systém a změň procesy tak, aby software nebyl potřeba
  + Udržuj systém
  + Reengineer
  + Nahraď
* Strategie by měla být založena na kvalitě software a jeho hodnotě pro business
  + Nízká kvalita, nízká hodnota => zahodit
  + Nízká kvalita, vysoká hodnota
    - Drahé na údržbu, ale přináší hodnotu
    - => Reingineer / replace
  + Vysoká kvalita, nízká hodnota => zahodit, anebo zvýšit hodnotu
  + Vysoká kvalita, vysoká hodnota => udržuj!
* Zhodnocení přínosu software
  + Jaké business procesy podporuje, pokud vynucuje použití neefektivních procesů, jeho hodnota se snižuje
  + Stabilita
  + Měly by se vzít v úvahu názory všech skupin lidí, kteří se software pracují
    - Systém je nepoužitelný pro 990 lidí, ale 10 lidem hodnotu přináší
    - Systém je nepoužitelný pro 10 lidí, ale 990 lidem hodnotu přináší
  + Jak moc se business spoléhá na výsledky daného systému
* Zhodnocení kvality
  + Zhodnocení kvality programu a ceny jeho údržby
  + Zhodnocení prostředí, ve kterém program běží (server etc.) a ceny jeho údržby
  + Jak moc se systém musí měnit
  + Datová náročnost
  + Další faktory
    - Srozumitelnost
    - Dokumentace
    - Data
      * Má systém explicitně definovaný a popsaný datový model?
      * Jak efektivně a konzistentně s daty pracuje
    - Výkon
    - Programovací jazyk
      * Je to stále používaný jazyk?
    - Složitost konfigurace
    - Existence testů
    - Schopnosti vývojového týmu
    - Stabilita dodavatele
    - Stabilita
    - Věk
    - Cena údržby
    - Interoperabilita

# UML při analýze a návrhu

* Unified modeling language (Univerzálně mučící langusta)
* Modelování = vývoj abstraktního modelu systému
  + Každý model zobrazuje jiný pohled na systém
  + Používá nějakou grafickou notaci
  + Aktuálně UML
  + Pomáhá porozumět funkcionalitě systému
* Perspektivy
  + Externí
    - Modeluje hranice systému a jeho prostředí
    - Case diagram
  + Strukturální
    - Organizace systému
    - Organizace dat zpracovávaných systémem
    - Class, Object, Component, Package, Deployment, Composite Structure diagram
  + Interakční
    - Interakce systému a prostředí
    - Interakce komponent
    - Sequence, Communication, Interaction overview, Timing diagram
  + Behaviorální
    - Dynamické chování systému
    - Jak reaguje na různé události
    - Activity Diagram, State Diagram

## Use case diagram

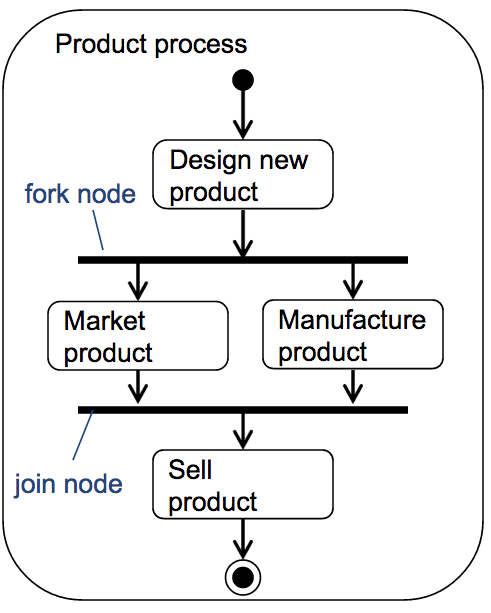
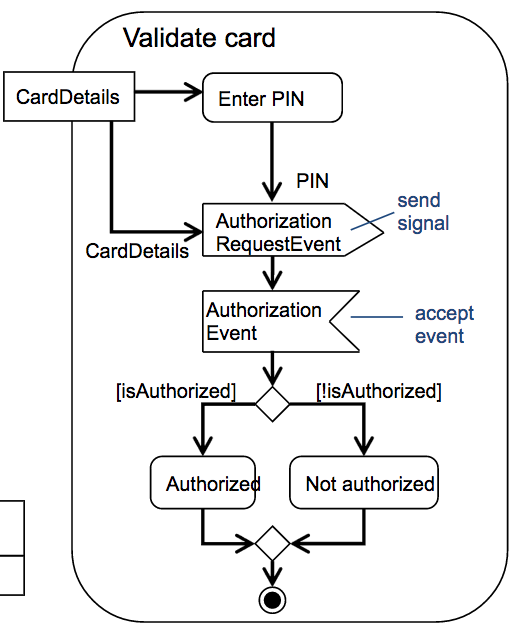
* Slouží ke specifikaci software
* Proces
  + Find the systém boundary
  + Find the actors
  + Find use cases
  + Specify use cases (using either text or UML aktivity diagrams)



* Use cases popisují nějakou akci, většinou jsou popisovány pomocí sloves
  + Můžou mít nějaké vstupní a výchozí podmínky
  + Mají svoje popsatelné flow (může se větvit a opakovat)
    - If, while, flow
* Umožňuje generalizaci actors i use cases (pomocí extend, include)

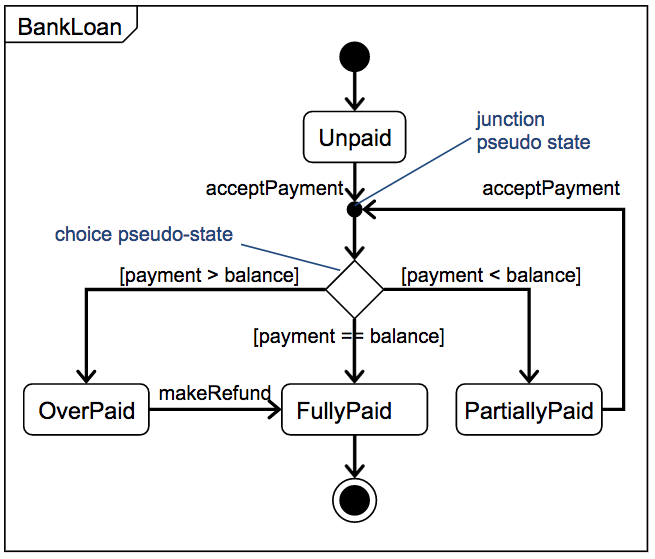
## Activity diagram

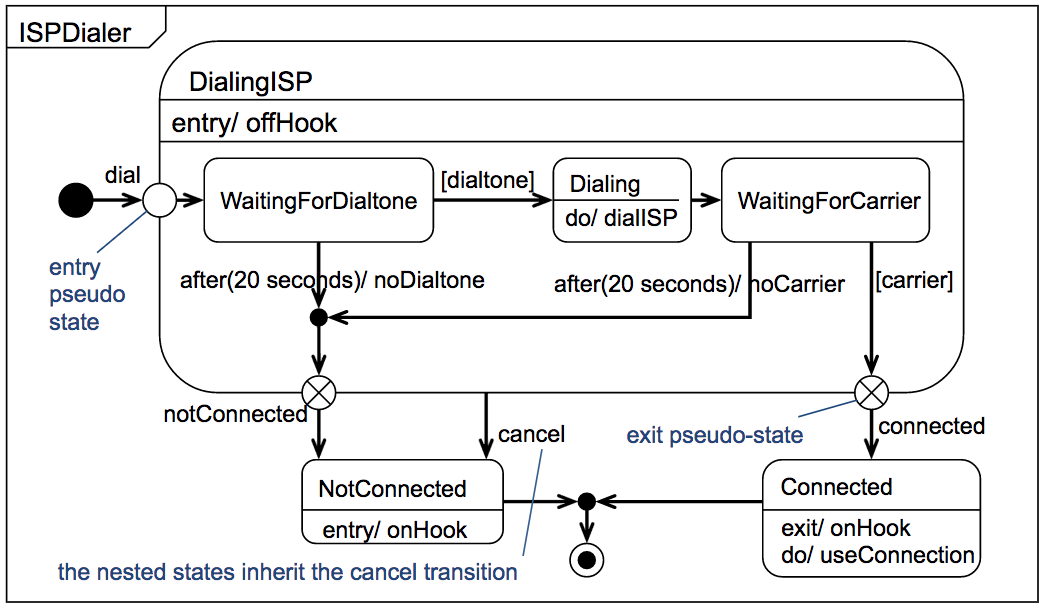
* Modeluje určitý use case / aktivitu
* Dá se členit to swimlanes
* Action nodes
  + Volají akce
  + Vysílají / přijímají signály
* Control nodes
  + Decision & merge
  + Fork & join
* Object nodes



## State diagram

* Popisuje stav nějakého objektu





## Class diagram

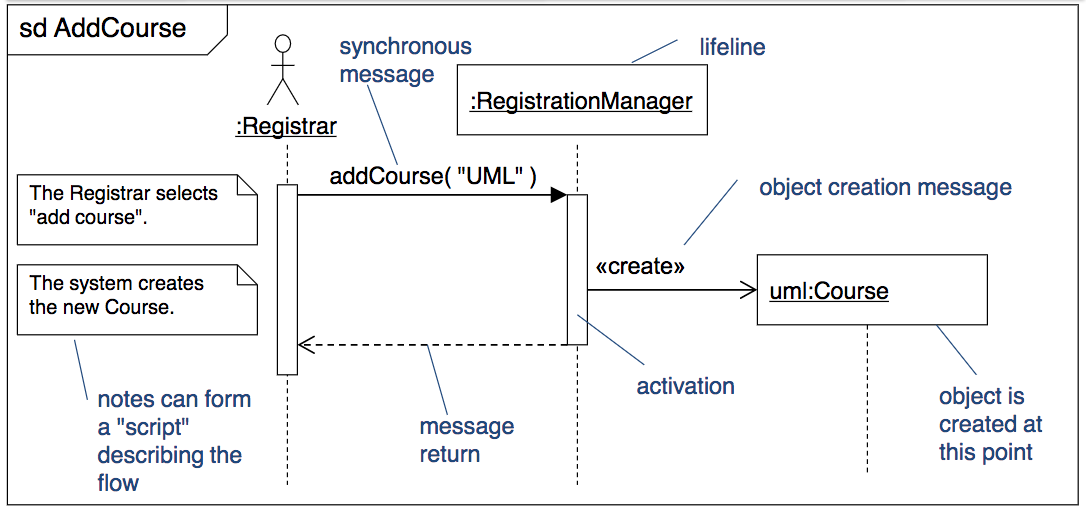
* Modeluje strukturu a chování systému

## ERD Diagram

* Modeluje pouze strukturu dat

## Sekvenční diagram

* Interakční diagram
* Modeluje interakci, dává důraz na časovou posloupnost
* Možnost větvení a cyklů
  + Alt
  + Loop
  + Opt



## Komunikační diagram

* Interakční diagram
* Modeluje interakci
* Důraz na strukturu komunikace

