Co je to asociacia?:
- Je mozne zobrazit ju ako atribut
- Vyjadrenie vztahu medzi dvoma triedami.

Ktore diagramy z UML mozu byt pouzite na znazorneni interakcie?
- Communication diagram
- Sequence diagram

Preco su vyssie programovacie jazyky (VJ) riesenim podstaty softverovej krizy?
- VJ su pretoze zatienuju pracu s nizkou urovnovymi zdrojmi (pamat, disk a pod) cim ulahcuju riesenie podstay problemu
- VJ su riesenim pretoze poskytuju vyssiu mieru abstrakcie cim zjednodusuju vyvoj

Ktore tvrdenie je pravdive o implementacii?
- Rozhrania implementovane triedou predstavuju jej poskytovanie rozhrania
- Trieda implementujuca rozhranie musi splnat podmienky rozhrania

Ziskavanie poziadaviek mozy byt vo forme
- Bezneho jazky
- Graficky notacii

Ktory z diagramov nepatri UML diagramy chovania
- Dataflow diagram

Co nie je sucastou validacie poziadaviek
- Formalna spravnost UML diagramov
- Cena poziadavaiek

Ake su zakladne charakteristiky scenara v use case diagramoch?
- Alternativy scenar sa moze zapisat do hlavneho scenara
- Alternativne scenare mozu byt jednoduche alebo zlozite
- Je zapisany v textovej forme, pricom tato forma nie je definovana v UML
- Zlozity alternativny scenar nemal mat ziadne dalsie alternativne scenare

Abstrakcia
- Je filtrovanie vlastosti a operacii objektu, pokial nezostanu len tie, ktore potrebujeme

Kto alebo co je actor – pouzivatel
- Jeho lepsim vyjadrenim je pojem „rola“
- Moze to byt clovek alebo system

Naco sluzi model zivotneho cyklu softveroveho procesu?
- Rozdeluje proces na etapy, etapam prideluje cinnosti
- Robi system vyvoja programovych systemov kontrolovatelnejsim

Aka je charakteristika vnutornych dovodov softverovej krizy
- Tieto dovody su neodstranitelne, nie je predpoklad, ze by ich bolo mozne obist

Analyticky model
- Obsahuje napriklad analyticke diagramy tried a use case
- Definuje co ma programovy system robit, nedefinuje ako to ma robit

K zmenam v poziadavkach
- Dochadza k nim celkom prirodzene napriklad z dovodu zmeny prostredia

Softverovy proces je
- Prostupnost aktivit specifikacia, vyvoj, validacia a evolucia
- Mnozina aktivit a prislusnych vysledkov, ktorych vysledkom je programovy system

Nefunkcne poziadavky
- Mozu ovplyvnovat celu architekturu pretoze moze ovplyvnit viacero casti systemu naraz

Ciastocnym riesenim, ktora nema kvality striebornej gulky je inkrementalny vyvoj. Co sa pod nim chape?
- Programovy system sa pestuje, nie stavia
- Budovanie programoveho systemu zacina malym plne funkcnym systemom a postupne nabera pozadovane funkcionality

Co je pouzivatelsky scenar
- Je textova forma zapisu postupnosti krokov
- Sada postupnosti krokov spojena spolocnym zamerom pouzivatela

Na co sluzia stereotypy
- Na oznacenie tried, ktore maju spolocne charakteristiky
- Stereotyp oznacujem nazvom uvedem v zatvorkach <<nazov>>

Ake su zakladne elementy sekvencneho diagramu
- Trieda
- Posielanie sprav

Na obrazku vztaz medzi osobou (person) a autom (car)

- Jedna osoba moze mat ziadne alebo viac aut
- Definova ako obojsmerny vztah

Abstrakcia a generalizacia su s ostatnymi pojmami vo vztahu
- Specializacie je opakom generalizacie
- Konkretizacia je opakom abstrakcie

Co patri medzi charakteristiky RUP
- Je objektovo orientovana
- Je prepojena s UML

Co je specifikovane pomocou use case
- Pozadovane pouzitie systemu

Modelovanie udajov
- Modeluje datove entity a ich atributy
- Je typicke pre strukturovane modelovanie

Na obrazku, co v oboch diagramoch predstavuje „size“
- Size je viazanost, ktora je znazornena ako atribut aj ako asociacia

Co je to softverove inzinierstvo
- Aplikovanie systematickeho, disciplinovaneho, kvantifikovatelneho pristupu k tvorbe, prevadzke a udrzbe softveru, tj aplikovanie inzinierstva na softver
- Systematicky pristup k analyze, dizajnu, odhadovaniu, implementacii, testovaniu, udrzbe a reinzinieringu sotveru aplikovanim inzinierskych postupov

Co je to nasobnost
- Rozsah dovolenych kardinalit

Akym sposobom mozu byt reprezentovane vlastnosti triedy
- Ako asociacie
- Ako atributy triedy

Aka je to konceptualna perspektiva UML
- Vytvorenie slovnej zasoby pre domenu

Strukturalny model systemu
- Pri objektovom modelovani vyuziva hlavne diagram tried

- Definuje zakladne stavebne casti systemu (napr. Triedy)

Co dostavate kupou metodiky RUP
- Sadu dokumentov a softverovych nastrojov dokumentujucich a podporujucich metodiku

Co je najvhodnejsim opisom aktora v UML
- Pouzivatelia alebo externe systemy mozu byt aktorom

Medzi hlavne vyhody spiraloveho modelu patri
- Komplexnost
- Vytvaranie znovupouzitelnych komponentov

Na obrazku, co je notaciou poskytovaneho rozhrania
- A

Modelom riadeny navrh a vyvoj
- Pri tomto type modelovania je zakladnym prvkom model, na zaklade ktoreho sa robi implementacia. Pri zmenach sa zmeny prenasaju naprv do modelu a potom do implementacie

Aky je rozdiel medzi agregaciou a kompoziciou
- Pri kompozicii spolu s objektom zanika aj komponovany objekt
- Obe su detailnejsim vyjadrenim asociacie

Co je trieda
- Trieda je zakladnym stavebnym prvkom v objektovo-orientovanom programovani. Logicky spaja udaje a operacie
- Trieda je kategoria alebo skupina veci, ktore maju podobne vlastnosti a rovnake alebo podobne chovanie

Model drivem architecture (MDA) vyuziva UML ako
- Programovaci jazyk

Co urcuje metodika RUP
- Urcuje kto, co, ako a kedy

Co patri medzi druhotne dovody softverovej krizy
- Praca v time
- Programatorska produktivita. Niektori programatori su az 10krat produktivnejsi

Co nepatri medzi dosledky softverovej krizy

- Zlozitost softveru
- Vysoka cena hardveru oproti softveru

Vztah <<include>> use case diagram
- Je sucastou hlavneho scenara a nejedna sa o alternativny scenar

Dokument specifikacia poziadaviek
- Sa po skonceni etapy odovzda zakaznikovi a doplnenie zmien do dokumentu je vitane
- Urcuje len „co“ ma programovy system robit, nie ako

Generalizacia a klasifikacia
- Generalizacia je tranzitivna, klasifikacia nie je tranzitivna.
- Generalizaciu je vhodne vytvorit ako dedenie

Ake vlastnosti ma objekt oznacovany ako value objekt
- Tento objekt sluzi na prenos dat, spravidla nema ziadne chovanie

Vztah Border kolia je pes a pes je druh su
- Pes je druh – je formou generalizacie
- Oba vztahy mozu byt vyjadrene ako dedenie, co nie je spravne riesenie
- Border kolia je pes – je formou generalizacie

Co zachytava use case
- Poziadavky na system
- Chovanie systemu

Co patri medzi vhodne nefunkcne poziadavky
- Aby programovy system mal odozvu do 2 sekund
- Aby programovy system vyuzival k autentifikacii ISIC kartu

Medzi zakladne modely softveroveho procesu nepatri
- BPM (business proces medeling)
- RUP (rational unified proces)

Medzi prvotne problemy softveru nepratri
- Prekrocenie rozpoctu
- Nizka kvalita

V akych formach moze byt uvedena nasobnost

- 1 nasobnost vyjadrujuca prave jednu vazbu
- 0..1 nasobnost vyjadrujuca jednu vazbu alebo ziadnu vazbu

Ciastocnymi riesenia, ktore riesia dovody softverovej krizy su
- Pristup kupovat programove systemy namiesto vytvarat
- Prototypovanie programovych systemov

Co je UML
- Graficka technika
- Jazyk na vizualizacie, specifikaciu, navrhovanie a dokumentaciu programovych systemov

Kedy bol prvy krat vo svete oficialne pouzity termin Softveroveho inzinierstva
- Na konferencii NATO

Sekvancny diagram patri medzi
- Diagramy interakcii
- Diagramy chovania

Ktory diagram nie je sucastou UML
- Entity diagram

Co je pouzivatelsky scenar
- Sada postupnosti krokov spojena spolocnym zamerom pouzivatela
- Je textova forma zapisu postupnosti krokov

Na obrazku, co znazornuje biely diamant (Boat = lod, Engine = motor)
- Lod obsahuje motor
- Motor umiestneny v lodi moze byt sucasne sucastou niecoho ineho v rovnakom case

Aka je to softverova perspektiva UML
- Element diagramy UML su prenasane do softveru ako softverove elemetny

Nefunkcne poziadavky
- Mozu ovplyvnovat celu architekturu, pretoze moze ovplyvnit viacero casti systemu naraz

Ake vlastnosti ma objekt oznacovany ako repository
- Tento objekt je ulozeny v databaze (napr. Relacnej)

Ciastocnymi rieseniami, ktore riesia dovody softverovej krizy su

- Prototypovanie programovych systemov
- Pristup kupovat programove systemy namiesto vytvarat

Co je specifikovane pomocou use case
- Pozadovane pouzitie systemu

Co je informacny system
- Je system na zber, udrziavanie, spracovanie a poskytovanie informacii
- Je subor ludi, technologickych prostriedkov a metod, ktora zabezpecuje prenos, spracovanie a uchovanie dat za ucelom tvorby informacii

Medzi diagramy struktur nepatri
- Aktivity diagram
- Komunikacny diagram

Co je charakteristicke pre neviditelnost ako dovod softverovej krizy
- Neexistuje sposob vizualizacie softveru tak aby pokryl vsetky jeho aspekty

Desat zamestnancov oddelenia moze pomocou formualra zaslat svoje cestovne vydavky aby im boli preplatene. Manazer musi tento formular (formulare) schvalit, kolko je v tomto pripade aktorov.
- 2 aktori

Ktora etapa zivotneho cyklu softveroveho procesu trva vo vseobecnosti najdlhsie
- Evolucia

Moznymi kandidatmi na striebornu gulku su
- Neexistuju ziadny kandidati na striebornu gulku pretoze by musela vyriesit naraz vsetky hlavne dovody softverovej krizy

Co musi obsahovat atribut triedy v uml diagrame
- Nazov

Ktory diagram nie je sucastou UML
- Entity diagram

Co je to atribut
- Atributom je napriklad datumobjednania v triede objednavka
- Atribut moze vzniknut ako vysledok asociacie
- Atribut je vlastnost triedy

Na co sluzi model systemu

- Model je abstrakciou toho, comu chceme porozumiet este predtym, ako to vybudujeme
- Model neobsahuje nepodstane detaily

Strukturalny model systemu
- Pri obkejtovom modelovani vyuziva hlavne diagram tried
- Definuje zakladna stavebnice casti systemu (napri. Triedy)

Co je to asociacia?:
- Je mozne zobrazit ju ako atribut
- Vyjadrenie vztahu medzi dvoma triedami.

Model interakcii systemu
- Definuje vztah systemu s pouzivatelom pripadne inymi systemami
- Pri objektovom modelovani vyuziva hlavne use case diagram

Medzi hlavne vyhody vodopadoveho modelu patria
- Vytvara dostatok dokumentacia
- Jednoducha kontrola pri riadeni

Co je to metoda a operiacia UML
- Operacia je v konceptualnom modely popisom chovania triedy
- Metoda je implementaciou operacie

Co nepatri medzi charakteristiky vodopadoveho modelu
- Prace na dalsej etape zacinaju az po skonceni predchadzajucej
- Iterativny vyvoj

Nefunkcne poziadavky
- mozu generovat funkcne poziadavky

Dedenie
- Porusuje zakladny princip OO programovania . zapuzdrenie
- Je silny nastroj OO programovania, ktory sa vyuziva casto nespravne

Co znamena pouzitie UML ako nacrtku
- Ak je UML nakreslene strucne (elektronicky alebo na papieri) a nesplna vsetky detailne nalezitosti diagramov

Kto alebo co je actor – pouzivatel
- Jeho lepsim vyjadrenim je pojem „rola“
- Moze to byt clovek alebo system

Udajovy slovnik
- Obsahuje zoznam datovych objektov a popis ich atributov

Co je informatika
- Subor vednych disciplin a postupov pricspracovani informacii
- Informatika nie je viac o pocitacoch ako astronomia o teleskopoch
- Veda o informacii a jej spracovani

Co je softverovy prokejt
- Vykonanie softveroveho procesu pre specificke poziadavky pouziatela, konkretizuje cinnosti a poradie definovane procesom

Co reprezentuje biely diamant v aktivity diagrame
- Rozhodovanie
- Spojenie

Co patri medzi vhodne nefunkcne poziadavky
- Aby programovy system dokazal obsluzit 1000 pouzivatelov

Ziskavanie poziadaviek moze byt vo forme
- Grafickych notacii
- Bezneho jazyka

Na obrazku, co popisuje vztah medzi use case A a use case B
- Usa case B rozsiruje use case A

Ktora kategoria nepatri do klasifikacie diagramov UML
- Use diagramy
- Diagramy balikov

SW projekt je docasny znamena, ze
- Ma svoj zaciatok a stanoveny koniec

Preco je lepsim pomenovanim pre actora rola ako pouzivatel
- Pretoze jeden a ten isty pouzivatel moze mat v systeme viac roli co bude znazornene viacerymi aktormi

Ako je ovplyvneny softverovy proces typom ucastnikov, ktori v nom vystupuju
- Ovplyvnuju akym sposobom sa kombinuju casti softveroveho procesu tj typ vhodnosti motediky

Aky je rozdiel medzi scenarom a diagramom pripadov use case diagramov

- Pripad pouzitia je slovesne pomenovanie funkcie programoveho systemu. Diagram znazornuje interakciu medzi tymito pripadmi pouzitia
- Pouzitavelsky scenar je v textovej podobe a nie je sucastou UML, diagram je graficke znazornenie tychto scenarov, pricom pouziva iba nazvy scenarov

Model systemu
- Musi byt spravny
- Nemusi byt kompletny, moze zachytavat iba niektore perspektivy

Medzi prvotne problemy softveru nepatri
- Prekrocenie rozpoctu
- Nizka kvalita

Model drivem architecture (MDA) vyuziva UML ako
- Programovaci jazyk

Co patri medzi charakteristiky RUP
- Je objektovo orientovana
- Je prepojena s UML
- Je znama a casto pouzivana, existuje o nej siroke povedomie
- Je iterativna

Na co sa pouziva spiralovy model prototyp
- Na odstranenie moznych rizik
- Na overenie funkcnych, technologickych rieseni predtym ako je vybrana konkretna

Co je to poziadavka
- Abstraktna formualcia sluzby, ktoru ma system poskytovat
- Detailne specifikovana poziadavka funkcie systemu

V akom diagrame sa objavuje element actor
- Use case diagram

Co je to softverove inzinierstvo
- Aplikovanie systematickeho, disciplinovaneho, kvantifikovatelneho pristupu k tvorbe, prevadzke a udrzbe softveru, tj aplikovanie inzinierstva na softver
- Systematicky pristup k analyze, dizajnu, odhadovaniu, implementacii, testovaniu, udrzbe a reinzinieringu sotveru aplikovanim inzinierskych postupov
-

Co specifikujeme funkcnymi poziadavkami
- Co system v definovanych situaciach robit nesmie
- Ako bude system reagovat na vstupy a ako sa bude chovat v roznych situaciach

Na obrazku, co je notaciou vyzadovaneho rozhrania
- C

Chyba nazov otazky
->b je typu c2

Abstrakcia a generalizacia su s ostatnymi pojmami vo vztahu
- Specializacie je opakom generalizacie
- Konkretizacia je opakom abstrakcie

Jazyk UML oznacujeme ako deskriptivny pretoze
- Pretoze jeho pouzitie je zalozene na konvencii a odchylky alebo vlastne znacky neznehodnocuju jeho vypovedaciu schopnost

Co modeluje model softveroveho procesu
- Modeluje poradie aktivit a ich vzajomne prepojenie

Co znamena, ze funkcne poziadavky maju byt konzistentne
- Poziadavky si navzajom neodporuju

Ktory model nepatri medzi modely MDA
- Model nezavisly na UML
- Model zavisly na pocitaci

Preco je jednotne programovacie prostredie (IDE) riesenim podstaty softverovej krizy
- IDE je riesenim ale iba druhotnych dovodov softverovej krizy
- IDE je riesenim pretoze poskytuje jednotny pristup k programovacim nastrojom, kniziciam ai

Medzi hlavne nevyhody vodopadoveho modelu patria
- Prilisna jednoduchost
- Dodanie programoveho systemu zakaznikovi az na konci

Aky je to analyticky diagram tried
- Elementy z diagramu sa nemusia preniest do programu (trieda v diagrame nemusi byt triedou v programe)
- Obsahuje menej detailov (chybaju typy atributov, typy parametrov metod a pod). Nie su uvedene vsetky operacie

Co znamena, ze funkcne poziadavky maju byt kompletne
- V dalsich fazach vyvoja uz nemaju vznikat nove poziadavky

Co je zavislost triedy
- Predstavuje vyjadrenie vseobecneho vztahu tried, ktory hovori, ze ak sa zmeni jedna strana moze to sposobit zmenu druhej strany

Naco sluzi model zivotneho cyklu softveroveho procesu?
- Rozdeluje proces na etapy, etapam prideluje cinnosti
- Robi system vyvoja programovych systemov kontrolovatelnejsim

Preco su vyssie programovacie jazyky (VJ) riesenim podstaty softverovej krizy?
- VJ su pretoze zatianuju pracu s nizkou urovnovymi zdrojmi (pamat, disk a pod) cim ulahcuju riesenie podstaty problemu
- VJ su riesenim pretoze poskytuju vyssiu mieru abstrakcie cim zjednodusuju vyvoj

Na obrazku, kolki dalsi aktory mozu byt zapojeni do scenara ak aktor X pouzije use case A
- 2

Medzi diagramy struktur nepatri
- Aktivity diagram
- Komunikacny diagram

Jan pracuje ako knihovnik. Moze si tiez poziciat knihu ako obycajny student. Kolko aktorov je potrebnych pri modelovani use case
- Dvaja neprepojeni aktori

Auto ma jedneho alebo viacerych majitelov sa da vyjadrit nasobnostou
- d..*

Co je to domenova trieda
- trieda, ktora je klucova danu oblast (napriklad ucet v bankovom systeme)

Co sa v softverovom inzinierstve chape pod pojmom Silver bullet (strieborna gulka)
- strieborna gulka je mytus, ktory sa zatial nepodarilo dosiahnut
- je to nieco co vyriesi naraz hlavne dovody softverovej krizy

Kompletne a konzistentne poziadavky
- v praxi je to tazko dosiahnutelny stav

Ktory z diagramov nepatri UML diagramy chovania
- dataflow diagram

Co predstsavuje cierny kruh, na obrazku, v aktivity diagrame
- zaciatok

Abstrakcia
- je filtrovanie vlastnosti a operacii objektu, pokial nezostanu len tie, ktore potrebujeme

Specifikacia poziadaviek
- je zhmotnena v podobe dokumentu „Dokument poziadavaiek“
- je procesom ziskavania poziadaviek na programovy system

Datami riadene modelovanie
- pri strukturovanom modelovani vyuziva DFD (data flow diagram)
- je typicke pre strukturovane modelovanie analyzy

Co nepatri medzi dosledky softverovej krizy
- Zlozitost softveru
- Vysoka cena hardveru oproti softveru

V akom poradi prebieha kombinacia na obrazku
- !p, !q, ?q, ?q

Co nepatri medzi popis operacie a jej parametrov
- Nasobnost
- Datum

Co je UML
- Graficka technika
- Jazyk na vizualizacie, specifikaciu, navrhovanie a dokumentaciu programovych systemov

Na obrazku, co znazornuje cierny diamant

Na obrazku, ktory use case nemusi byt dostupny a aj tak sa vykona scenar reprezentovany usa case D
- Use case C

Ktore tvrdenie je pravdive o implementacii
- Rozhrania implementovanie triedou predstavuju jej poskytovanie rozhrania
- Trieda implementujuca rozhranie musi splnat podmienky rozhrania

Je programovy system sofftverom
- Ano, softverom moze byt aj programovy system

Ktory symbol, na obrazku, reprezentuje asynchronnu spravu
- B

Co zachytava use case
- Poziadavky na system
- Chovanie systemu

Triedy do diagramu tried
- Analyzou slovies a podstatnych mien
- Je mozne najst vyuzitie CRC stitkov

Aky je analyticky diagram tried
- Obsahuje menej detailov (chybaju typy atributov, typy parametrov metod a pod) nie su uvedene vsetky operacie
- Elementy z diagramu sa nemuia preniest do programu (trieda v diagrame nemusi byt triedou v programe)

Co je vhodnym prikazom abstrakcie
- Namiesto pojmu BMW a opel pouzivat pojem „auto“

Pouzivatel pouziva system na objednanie knih a system pouziva: system kreditnych kariet na validaciu pouzivatelovej kreditnej karty, system na validaciu adresy a internu databazu pouzivatelov. Predpokladame, ze chceme namodelovat objednanie knih ako jeden scenar v use case. Aky aktory budu pouzity
- Zakaznik, system kreditnych kariet, system kontroly adries

Modelom riadeny navrh a vyvoj
- Pri tomto type modelovania je zakladnym prvkom model, na zaklade ktoreho sa robi implementacia. Pri zmenach sa zmeny prenasaju najprv do modelu a potom do implementacie

Co je pouzivatelsky scenar
- Je textova forma zapisu postupnosti krokov
- Sada postupnosti krokoj spojena spolocnym zamerom pouzivatela

Aka je to konceptualna perspektiva UML
- Vytvorenie slovnej zasoby pre domenu

Na obrazku, kolki dalsi aktory mozu byt zapojeni do scenara ak aktor X pouzije use case A
- 2

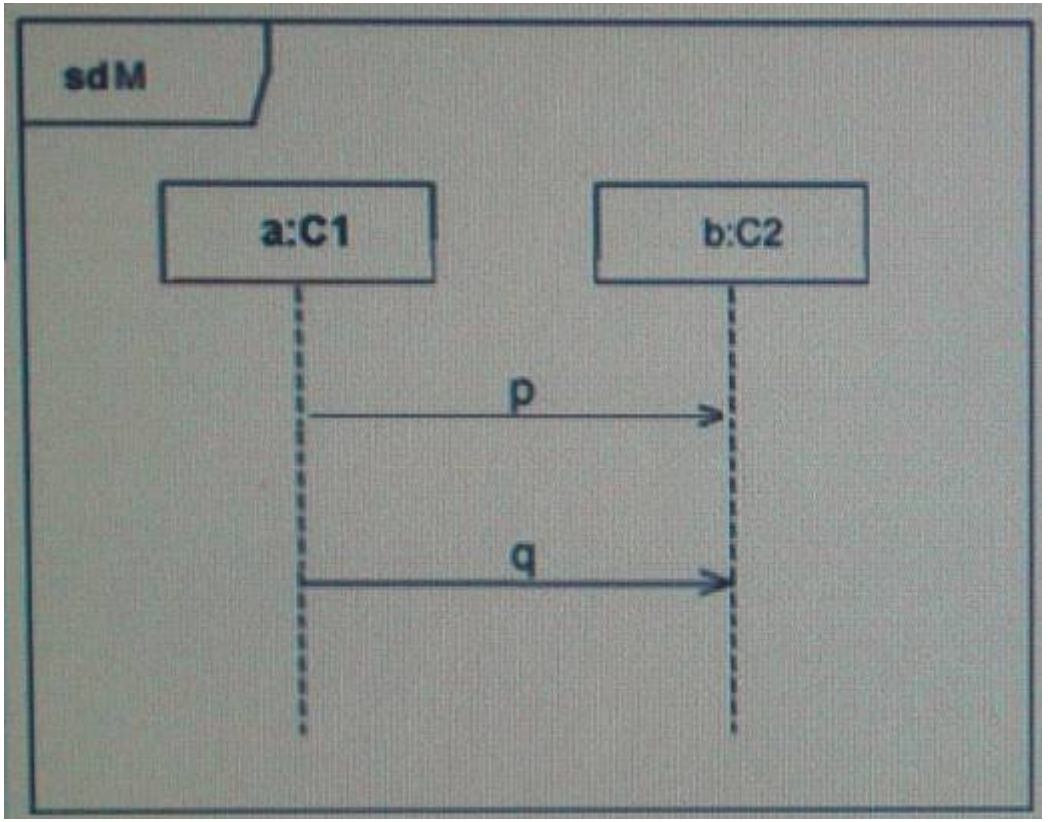
Domenou riadeny navrh
- Vyuziva abstrakciu na vytvorenie domenovych objektov

Aky je primarny vyznam pouzitia balikov (package)

- Vytvorit menne priestory
- Zoskupenie modelovane elementy do logickych casti

Co znamena skratka UML

- Unified modeling language



Vyberte aspoň jednu odpoveď (v):

a.) !p,?q,!q,?p

b.) !p,!q,?p,?q

c.) ?p,!p,?q,!q

d.) !p,?p,!q,?q

2. Sekvenčné diagramy sa používajú v prvom rade na :

a.) modelovanie procesných tokov

b.) sekvencia jednotlivých etáp pri vývoji softvéru

c.) modelovanie komunikácie medzi jednotlivými objektmi

d.) modelovanie sekvencie príkazov rámci konkrétnej metódy

3. Stavové diagramy sa používajú na (v):

a.) modelovanie workflow jednotlivých procesov

b.) modelovanie možných stavov objektu a prechodov medzi nimi

c.) modelovanie algoritmov jednotlivých metód

d.) modelovanie možných stavov systému a prechodov medzi nimi

4. Prechod medzi stavmi je charakterizovaný (v) :

a.) spúšťou (trigger-signature)

b.) aktivitou (activity) Pamatam si takuotazku ale neviem ci by tam nemala byt tato moznost...ale nie som si ista:)

c.) podmienkovým výrazom (guard)

d.) odoslanou správou (message)

5. Prečo sú vyššie programovacie jazyky (VJ) riešením podstaty softvérovej krízy (v):

a.) VJ sú pretože zatieňujú prácu s nízko úrovňovými zdrojmi (pamäť, disk a pod.), čím uľahčujú riešenie podstaty problému

b.) VJ nie sú riešením pretože VJ nie sú multiplatformové a ich výkon je nižší v porovnaní s jazykmi nižšej úrovne

c.) VJ je riešením ale iba „vnútorným“ (neodstrániteľných) dôvodov softvérovej krízy

d.) VJ sú riešením pretože poskytujú vyššiu mieru abstrakcie, čím zjednodušujú vývoj

e.) VJ nie sú riešením pretože neriešia podstatu ale iba dôsledky softvérovej krízy

6. Abstraktné triedy v diagramoch tried (v):

a.) zakresľujeme rovnako ako ostatné triedy, označíme ju však kľúčovým slovom {abstract}

b.) zakresľujeme prerušovaným obdĺžnikom

c.) nezakresľujeme

d.) zakresľujeme rovnako ako ostatné triedy, názov zapíšeme kurzívou

7. Medzi typy vzťahov medzi triedami nepatrí (v): (tu moze byt len 1 odpoved)

a.) agregácia

b.) generalizácia

c.) asociácia

d.) abstrakcia

8. Vzťahy *Border kóliaje pes* a *Pes je druh* sú (v):

a.) Border kólia je pes – je formou generalizácie

b.) Pes je druh – je formou generalizácietato nie

c.) Border kólia je plemeno

d.) Pes je drhu – je formou klasifikácie tato ano

e.) Oba vzťahy môžu byť vyjadrené ako dedenie, čo je nie je správne riešenie

f.) Border kólia je pes – je formou klasifikácie

g.) Oba vzťahy môžu byť vyjadrené ako dedenie, čo je aj správne riešenie

(tu mas aj nejake odpovede, na ktore nie som si ista, ale pozriem to, ale teraz nie som na intaku)

9. Počiatočný stav :

a.) sa môže nachádzať v jednom stavovom diagrame aj viackrát

b.) sa v nemusí nachádzať v každom stavovom diagrame

c.) sa musí nachádzať v každom stavovom diagrame práve 1 raz

10. Ornament diagramu aktivít join slúži na :

a.) nepoužíva sa v diagramoch aktivít

b.) označenie konca paralelného spracovávania

c.) označenie konca podmieneného spracovávania

11. Čo je informačný systém (v):

a.) Jeho súčasťou „musia“ byť počítače resp. počítačové programy, ktoré IS prevádzkujú

b.) Je systém na zber, udržiavanie, spracovanie a poskytovanie informácií

c.) Synonymum k „programovému systému“

d.) Je to všeobecné pomenovanie pre účtovnícky a ekonomický program v podnikoch (ekonomických objektoch)

e.) Jeho súčasťou nemusia byť počítačové programy

f.) Je súbor ľudí, technologických prostriedkov a metód, ktoré zabezpečujú prenos, spracovanie a uchovávanie dát za účelom tvorby informácií

12. Väzba extend v diagramoch USE CASE slúži na (v):

a.) na vyčlenenie opakujúcich sa časti scenárov do samostatného prípadu použitia

b.) na znázornenie doplnkového scenára k hlavnému scenáru

c.) dekompozíciu hlavného prípadu použitia na sub-prípady použitia

13. Funkčný polymorfizmus, ako princíp orientového modelovania (v) :
    a.) znamená, že rodič môže plne nahradiť funkcionalitu potomka
    b.) znamená, že v závislosti od zoznamu formálnych parametrov môže byť volaná iná verzia volanej metódy
    c.) znamená, že potomok môže plne nahradiť funkcionalitu rodiča
    d.) znamená, že inštancia tried sa navzájom môžu líšiť zoznamom atribútov

14. Scenár slúži v prvom rade na (v) :
    a.) modelovanie komunikácie aktéra so systémom
    b.) modelovanie sekvencie jednotlivých príkazov v rámci konkrétnej metódy
    c.) modelovanie sekvencie akcií v rámci konkrétneho prípadu použitia
    d.) modelovanie komunikácie jednotlivých inštancií tried

15. Plný kruh v sekvenčných diagramoch predstavuje (v):
    a.) rozhranie
    b.) databázu
    c.) odkaz na iný sekvenčný diagram
    d.) sondu typu extend

16. Zapuzdrenie (enkapsulácia) ako princíp objektového modelovania (v) :
    a.) je možnosť vkladať do tela triedy definície iných trie
    b.) je mechanizmus zabezpečenia prístupu k niektorým členom triedy
    c.) je možnosť deklarovať atribúty jednej triedy, ktorých dátovými typmi sú iné triedy
    d.) hovorí, že súkromne členy triedy sú viditeľné len z tela danej triedy

17. Prípad použitia (USE CASE) predstavuje (v) :
    a.) spôsob použitia systému
    b.) vybranú časť funkcionality systému
    c.) hrubý návrh konkrétnej metódy

18. Abstraktná trieda je (v):

a.) triedy, ktorá obsahuje len dátové položky
b.) trieda, ktorá sa používa len na dedenie
c.) triedy, ktorej metódy používať aj bez vytvorenia inštancie
d.) inštančná metóda

19. Šípka s plnou čiarou v sekvenčných diagramoch znázorňuje :
a.) metódu
b.) návratovú hodnotu
c.) poslanú správu skor by som dala toto
d.) akciu

20. Prečo je vhodné vytvárať a poznať modely životného cyklu :
a.) Aby sme si vedeli lepšie vizuálne predstaviť vytváraný programový systém
b.) Aby sme mohli programový systém zakresliť pomocou UML
c.) Aby sme vedeli proces vývoja programového systému lepšie riadiť na základe určených etáp a činnosti
d.) Aby sme vedeli ľahšie odhadnúť cenu programového systému

21. Ak je medzi dvoma triedami asociačný vzťah (v) :
a.) tak ide o rovnoprávny typ vzťahu
b.) hovoríme, že jedna trieda obsahuje atribút, ktorého dátový typ je druhá trieda
c.) tak jedna trieda je časťou druhej triedy
d.) tak jedna trieda vlastní druhú

22. Čo je to softvérový projekt :
a.) Vykonanie softvérového procesu pre špecifické požiadavky používateľa, konkretizuje činnosti a poradie definované procesom
b.) Dokumentácia k programovému systému vytvorená v UML
c.) Opakujúca sa činnosť pri tvorbe programových systémov
d.) Dokument vytvorený vo vhodnom softvérovom nástroji (napr. MS Project)

23. Čo sa v softvérovom inžinierstve chápe pod pojmom **Silverbullet**(Strieborná guľka) (v):
a.) Strieborná guľka je mýtus, ktorý sa zatiaľ nepodarilo dosiahnuť
b.) Je spôsob modelovania programového systému, ktorý je odolný voči chybám

c.) Je to ocenenie, ktoré je každoročné udeľované v rámci organizácie ACM
d.) Spôsob vývoja programového systému, tak aby sa podobal na strieborný kov (tj. dizajn programového systému)
e.) Je tradičný a overený model životného cyklu vývoja softvéru
f.) Je to „niečo“ čo vyrieši naraz hlavné dôvody softvérovej krízy

24. Nefunkčne požiadavky :
a.) Sa vymedzujú iba na určitý modul programového systému
b.) Ovplyvňujú grafický a dizajnové prevedenie programového systému
c.) Neovplyvňujú architektúru systému, pretože majú dopad iba platformu, ktorú systém bude používať
d.) Môžu ovplyvňovať celú architektúru, pretože môže ovplyvniť viacero časti systému naraz
e.) Neovplyvňujú architektúru systému, pretože sú zadávané hneď na začiatku
f.) Môže ovplyvňovať celú architektúru, pretože sú zadávané hneď na začiatku

25. Ornament diagramu aktivít fork označuje :
a.) začiatok paralelného spracovávanie
b.) nepoužíva sa v diagramoch aktivít
c.) rozhodovací bod

26. Aký je rozdiel medzi programom a programovým systémom – výrobkom (v) :
a.) Program je súhrn programových systémov
b.) Programový systém používa autor sám v podmienkach, pre ktoré ho vyvinul
c.) Program používa autor sám v podmienkach, pre ktoré ho vyvinul
d.) Programový systém je otestovaný a jeho použitie spĺňa dohodnuté obmedzenia
e.) Programový systém je to isté ako informačný systém
f.) Program je synonymum pre operačný systém počítača
g.) Programový systém môže byť 9-krát drahší ako program

27. Čo patrí medzi druhotné dôvody softvérovej krízy (v):
a.) prekročenie rozpočtu
b.) Práca v tíme
c.) nízka kvalita
d.) Programátorská produktivita. Niektorí programátori sú až 10 krát produktívnejší.

28. Ak je medzi triedami kompozičný vzťah, hovoríme že (v):
    a.) medzi triedami je vzťah typu 1:1
    b.) nadradená trieda dedí atribúty a metódy podradenej
    c.) nadradená trieda je zodpovedná za životný cyklus podradenej
    d.) ide tzv. silnú agregáciu

29. Objektovo – orientovaný prístup sa od štruktúrovaného odlišuje (v) :
    a.) zoskupenie dát a aplikačnej logiky, ktorá s dátami manipuluje
    b.) znovopoužiteľnosťou určitých častí kódu
    c.) oddelením dátovej, aplikačnej a prezentačnej vrstvytoto by som asi nedala
    d.) stavbou programu podlamna aj toto

30. Obdĺžnikom v sekvenčných diagramoch zakresľujeme :
    a.) sekvenciu
    b.) objekt
    c.) triedu
    d.) správu

31. Čo nepatrí medzi popis operácie a jej parametrov (v) : operacieakozemetody v triede alebo v niektorom z diagramov ?
    a.) Viditeľnosť
    b.) Násobnosť
    c.) Názov
    d.) Návratová hodnota
    e.) Smer parametrov
    f.) Dátum

32. Čo je to doménová trieda :
    a.) Trieda, ktorú označíme stereotypom <<domain>>
    b.) Trieda, ktorá v sebe obsahuje názov webovej domény zákazníka (napr. sk)
    c.) Trieda, ktorá je najväčšia a dominuje nad ostatnými (môže ich byť viac)
    d.) Trieda, ktorá je kľúčová danú oblasť (napríklad účet v bankovom systéme)

33. Počiatočný stav :
    a.) sa musí nachádzať v každom stavovom diagrame práve 1 raz
    b.) sa v nemusí nachádzať v každom stavovom diagrame
    c.) sa môže nachádzať v jednom stavovom diagrame aj viackrát

34. Sekvenčné diagramy sa používajú v prvom rade na :
    a.) modelovanie komunikácie medzi jednotlivými objektmi
    b.) modelovanie procesných tokov
    c.) sekvencie jednotlivých etáp pri vývoji softvéru
    d.) modelovanie sekvencie príkazov rámci konkrétnej metódy

35. Čo znamená, že funkčné požiadavky majú byť kompletné :
    a.) V požiadavkách musia byť aj funkčné aj nefunkčné požiadavky
    b.) Požiadavkám nesmú chýbať formálne náležitosti ak je napr. dátum a čas zadania požiadavky a meno toho kto požiadavku zadal
    c.) Dokument požiadaviek musí obsahovať aj textovú formu požiadaviek aj diagramovú (usecase)
    d.) V ďalších fázach vývoja už nemajú vznikať nové požiadavky

36. Generalizácia a klasifikácia (v):
    a.) Generalizácia je tranzitívna, klasifikácia nie je tranzitívna
    b.) Generalizácia nie je tranzitívna, klasifikácia je tranzitívna
    c.) Klasifikácia je opakom generalizácie
    d.) Klasifikáciu je vhodné vytvoriť ako dedenie
    e.) Generalizáciu je vhodné vytvoriť ako dedenie
    f.) Generalizácia je opakom agregácie
    g.) Generalizácia je opakom klasifikácie

37. Čiastočnými riešeniami, ktoré riešia dôvody softvérovej krízy sú (v):
    a.) Vodopádový model
    b.) Prístup „Kupovať programové systémy namiesto vytvárať“
    c.) Prototypovanie programových systémov
    d.) Yourdonové diagramy

e.) Aplikovanie počítačovej vedy do softvérového inžinierstva

39. Čo je trieda (v):

a.) Trieda je skupina prvkov v rámci systému

b.) Trieda je kategória, alebo skupina veci, ktoré majú podobné vlastnosti a rovnaké, alebo podobné chovanie.

c.) Trieda je inštanciou objektu

d.) Trieda je základným stavebným prvkom v objektovo – orientovanom programovaní. Logický spája údaje a operácie

e.) Trieda je kategória v hierarchickom pohľade na model programového systému

40. Diagram objektov predstavuje množinu inštancií tried a vzťahov medzi nimi v určitom okamihu :

ÁNO

NIE

41. Medzi hlavné výhody vodopadového modelu patria (v) :

a.) Vytvára dostatok dokumentácia

b.) Pravidelná validácia zo strany zákazníka

c.) Jednoduchá kontrola pri riadení

d.) Je pružný pri zmene situácie, ktorá môže pri realizácií nastať

e.) Umožňuje zákazníkovi modifikovať požiadavky podľa aktuálnych okolnosti

42. Aké vlastnosti má objekt označovaný ako repository :

a.) Tento objekt slúži na prenos dát, spravidla nemá žiadne chovanie

b.) Tento objekt existuje v programe ako jediná inštancia

c.) Tento objekt je uložený databáze (napr. relačnej)

d.) Tento objekt nie je možné inštanciovať

43. Sondy v sekvenčných diagramoch predstavujú (v) :

a.) odkazy na iné sekvenčné diagramy

b.) kontrolné body

c.) odkazy na externé prvky systému

d.) statické dátové štruktúry

44. Ktorý diagram nie je súčasťou UML : bud a alebo c, nie som si ista

a.) compositestructure diagramtoto nie

b.) entity diagram urcite len toto

c.) component diagram

d.) class diagram

e.) package diagram

42. Získavanie požiadaviek môže byť vo forme (v) :

a.) Bežného jazyka

b.) Faktúry za programový systém

c.) V klasickom modely životného cyklu slovným dohovorom

d.) Grafických notácií

43. Čo je atribút (v):

a.) Atribútom je napríklad „dátumObjednania“ v triede „Objednávka“

b.) Atribút nie je možné stotožniť s asociáciou

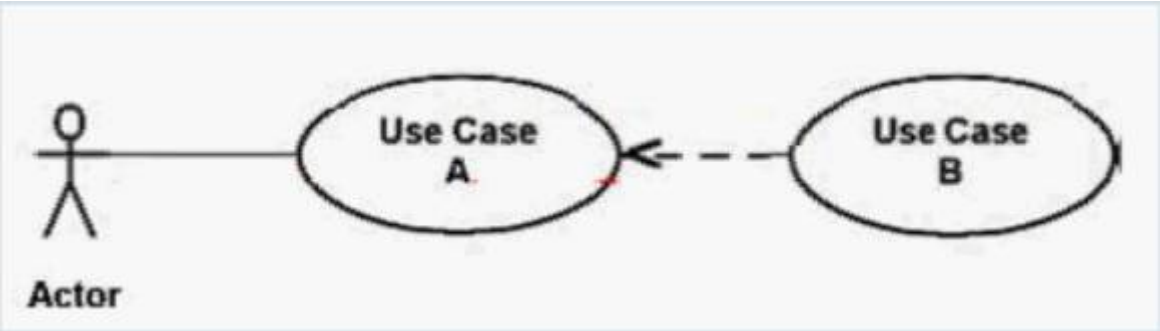
c.) Atribút deklaruje chovanie triedy

d.) Atribútom je napríklad „vyratajDan()“ v triede „Objednávka“

e.) Atribút je vlastnosť triedy

f.) Atribút môže vzniknúť ako výsledok asociácie

44. Na obrázku, čo najlepšie popisuje vzťah medzi usecase A a usecase B :



a.) usecase B rozširuje usecase A

b.) usecase A rozširuje usecase B

c.) usecase B generalizuje usecase A

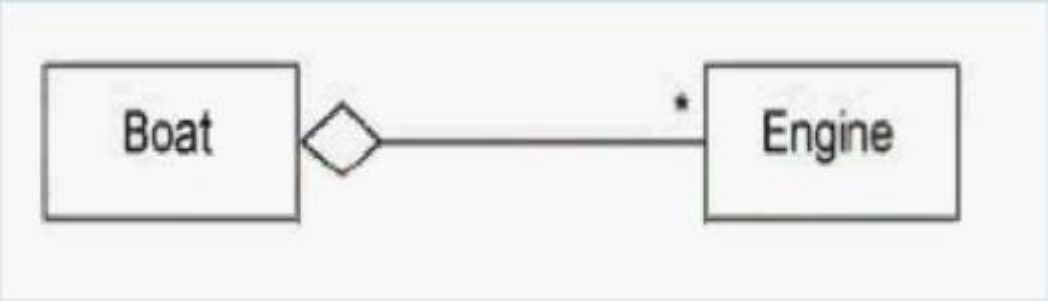
d.) usecase A zahŕňa (include) usecase B
e.) usecase A generalizuje usecase B
f.) usecase B zahŕňa (include) usecase A

45. Čo je UML (v)?
a.) Metodika vytvorená firmou Rational– dala by som aj to ale nie som si ista slovom Metodikaasi nie
b.) Metóda na tvorbu špecifikácie programového systému
c.) Diagram architektúry programového systému
d.) Špeciálny programovací jazyk na vytváranie ekonomických aplikácií
e.) Grafická technika
f.) Jazyk na vizualizáciu, špecifikáciu, navrhovanie a dokumentáciu programových systémov

46. SW projekt je postupne vypracovávaný znamená, že :
a.) počas priebehu projektu sa objavujú nové informácie, na základe ktorých prijať správne rozhodnutia
b.) programový systém sa vytvára postupne vo fázach podľa vodopádového modelu
c.) projektová dokumentácia sa postupne vypracováva pokým dôjde k implementácií a realizácií
d.) projekt sa pestuje nie buduje

47. Čo špecifikujeme funkčnými požiadavkami (v):
a.) Aký dizajn/grafiku (v prípade webovej) aplikácie má systém používať
b.) Aká metodika má byť pri vývoji použitá
c.) Čo systém v definovaných situáciách robiť nesmie
d.) Ako bude systém reagovať na vstupy a ako sa bude chovať v rôznych situáciách
e.) Akými platnými právnymi predpismi sa programový systém ma riadiť

48. Na obrázku, čo znázorňuje biely diamant (Boat = loď, Engine = motor) (v) :

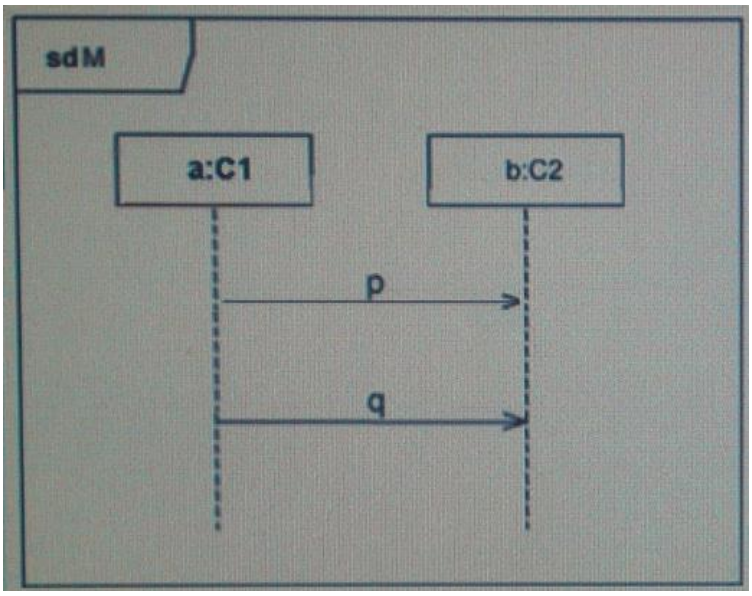


a.) Loď obsahuje motor

b.) Motor umiestnený v lodi môže byť súčasne súčasťou niečoho iného v rovnakom čase

c.) Ak je zmazaná loď, inštancie jej motora sú zmazané tiež

d.) Motor nemôže byť odstránený z lode



Vyberte aspon jednu odpoved (v):

a. !p,?q,!q,?p
b. !p,!q,?p,?q
c. ?p,!p,?q,!q
d. !p,?p,!q,?q

2. Sekvencne diagramy sa pozivaju v prvom rade na:

a. modelovanie procesnych tokov
b. sekvencia jednotlivych etap pri vyvoji softveru
c. modelovanie komunikacie medzi jednotlivymi objektmi
d. modelovanie sekvencie prikazov ramci konkretnej metody

3. Stavove diagramy sa pouzivaju na:

a. modelovanie workflow jednotlivych procesov
b. modelovanie moznych stavov objektu a prechodov medzi nimi
c. modelovanie algoritmov jednotlivych metod
d. modelovanie moznych stavov systemu a prechodov medzi nimi

4. Prechod medzi stavmi je charakterizovany (v) :

a. spustou (trigger-signature)
b. aktivitou (activity)
c. podmienkovym vyrazom (guard)
d. odoslanou spravou (message)

5. Preco su vyssie programovacie jazyky (VJ) riesenim podstaty softverovej krizy

VJ su pretoze zatienuju pracu s nizko urovnovymi zdrojmi (pamät, disk a pod.), cim ulahcuju riesenie podstaty problemu
a. VJ nie su riesenim pretoze VJ nie su multiplatformove a ich vykon je nizsi v porovnani s jazykmi nizsej urovne
b. VJ je riesenim ale iba „vnutornym“ (neodstranitelnych) dovodov softverovej krizy
c. VJ su riesenim pretoze poskytuju vyssiu mieru abstrakcie, cim zjednodusuju vyvoj
d. VJ nie su riesenim pretoze neriesia podstatu ale iba dosledky softverovej krizy

6. Abstratkne triedy v diagramoch tried (v):

a. zakreslujeme rovnako ako ostatne triedy, oznacime ju vsak klucovym slovom {abstract}
b. zakreslujeme prerusovanym obdlznikom
c. nezakreslujeme
d. zakreslujeme rovnako ako ostatne triedy, nazov zapiseme kurzivou

7. Medzi typy vztahov medzi triedami nepatri (v):

agregacia
a. generalizacia
b. asociacia
c. abstrakcia

8. Vztahy *Border kolia je pes* a *Pes je druh* su (v):

a. Border kolia je pes – je formou generalizacie
b. Pes je druh – je formou generalizacie
c. Border kolia je plemeno
d. Pes je druh – je formou klasifikacie
e. Oba vztahy mozu byt vyjadrene ako dedenie, co je nie je spravne riesenie
f. Border kolia je pes – je formou klasifikacie

g. Oba vztahy mozu byt vyjadrene ako dedenie, co je aj spravne riesenie

9. Pociatocny stav :
a. sa moze nachadzat v jednom stavovom diagrame aj viackrat
b. sa v nemusi nachadzat v kazdom stavovom diagrame
c. sa musi nachadzat v kazdom stavovom diagrame práve 1 raz

10. Ornament diagramu aktivit join sluzi na :
a. nepouziva sa v diagramoch aktivit
b. oznacenie konca paralelneho spracovavania
c. oznacenie konca podmieneneho spracovavania

11. Co je informacny system (v):
a. Jeho sucastou „musia“ byt pocitace resp. pocitacove programy, ktore IS prevadzkuju
b. Je system na zber, udrziavanie, spracovanie a poskytovanie informacii
c. Synonymum k „programovemu systemu“
d. Je to vseobecne pomenovanie pre uctovnicky a ekonomicky program v podnikoch (ekonomickych objektoch)
e. Jeho sucastou nemusia byt pocitacove programy
f. Je subor ludi, technologickych prostriedkov a metod, ktore zabezpecuju prenos, spracovanie a uchovavanie dat za ucelom tvorby informacii

12. Vazba extend v diagramoch USE CASE sluzi na (v):
a. na vyclenenie opakujucich sa casti scenarov do samostatneho pripadu pouzitia
b. na znazornenie doplnkoveho scenara k hlavnemu scenaru
c. dekompoziciu hlavneho pripadu pouzitia na sub-pripady pouzitia

13. Funkcny polymorfizmus, ako princip orientoveho modelovania (v) :
a. znamena, ze rodic moze plne nahradit funkcionalitu potomka
b. znamena, ze v zavislosti od zoznamu formalnych parametrov moze byt volana ina verzia volanej metody
c. znamena, ze potomok moze plne nahradit funkcionalitu rodica
d. znamena, ze instancia tried sa navzajom mozu lisit zoznamom atributov

14. Scenar sluzi v prvom rade na (v) :
a. modelovanie komunikacie aktera so systemom
b. modelovanie sekvencie jednotlivych prikazov v ramci konkretnej metody
c. modelovanie sekvencie akcii v ramci konkretneho pripadu pouzitia
d. modelovanie komunikacie jednotlivych instancii tried

15. Plny kruh v sekvencnych diagramoch predstavuje (v):
a. rozhranie
b. databazu
c. odkaz na iny sekvencny diagram
d. sondu typu extend

16. Zapuzdrenie (enkapsulacia) ako princip objektoveho modelovania (v) :
a. je moznost vkladat do tela triedy definicie inych tried
b. je mechanizmus zabezpecenia pristupu k niektorym clenom triedy
c. je moznost deklarovat atributy jednej triedy, ktorych datovymi typmi su ine triedy
d. hovori, ze sukromne cleny triedy su viditelne len z tela danej triedy

17. Pripad pouzitia (USE CASE) predstavuje (v) :
a. sposob pouzitia systemu
b. vybranu cast funkcionality systemu
c. hruby navrh konkretnej metody

18. Abstraktna trieda je (v):
a. triedy, ktora obsahuje len datove polozky
b. trieda, ktora sa pouziva len na dedenie
c. triedy, ktorej metody pouzivat aj bez vytvorenia instancie
d. instancna metoda

19. Sipka s plnou ciarou v sekvencnych diagramoch znazornuje :
a. metodu
b. navratovu hodnotu
c. poslanu spravu
d. akciu

20. Preco je vhodne vytvarat a poznat modely zivotneho cyklu :
a. Aby sme si vedeli lepsie vizualne predstavit vytvarany programovy system

b. Aby sme mohli programovy system zakreslit pomocou UML
c. Aby sme vedeli proces vyvoja programoveho systemu lepsie riadit na zaklade urcenych etap a cinnosti
d. Aby sme vedeli lahsie odhadnut cenu programoveho systemu

21. Ak je medzi dvoma triedami asociacny vztah (v) :
a. tak ide o rovnopravny typ vztahu
b. hovorime, ze jedna trieda obsahuje atribut, ktoreho datovy typ je druha trieda
c. tak jedna trieda je castou druhej triedy
d. tak jedna trieda vlastni druhu

22. Co je to softverovy projekt :
a. Vykonanie softveroveho procesu pre specificke poziadavky pouzivatela, konkretizuje cinnosti a poradie definovane procesom
b. Dokumentacia k programovemu systemu vytvorena v UML
c. Opakujuca sa cinnost pri tvorbe programovych systemov
d. Dokument vytvoreny vo vhodnom softverovom nastroji (napr. MS Project)

23. Co sa v softverovom inzinierstve chape pod pojmom Silver bullet (Strieborna gulka) (v):
a. Strieborna gulka je mytus, ktory sa zatial nepodarilo dosiahnut
b. Je sposob modelovania programoveho systemu, ktory je odolny voci chybam
c. Je to ocenenie, ktore je kazdorocne udelovane v ramci organizacie ACM
d. Sposob vyvoja programoveho systemu, tak aby sa podobal na strieborny kov (tj. dizajn programoveho systemu)
e. Je tradicny a overeny model zivotneho cyklu vyvoja softveru
f. Je to „nieco“ co vyriesi naraz hlavne dovody softverovej krizy

24. Nefunkcne poziadavky :
a. Sa vymedzuju iba na urcity modul programoveho systemu
b. Ovplyvnuju graficky a dizajnove prevedenie programoveho systemu
c. Neovplyvnuju architekturu systemu, pretoze maju dopad iba platformu, ktoru system bude pouzivat
d. Mozu ovplyvnovat celu architekturu, pretoze moze ovplyvnit viacero casti systemu naraz
e. Neovplyvnuju architekturu systemu, pretoze su zadavane hned na zaciatku
f. Moze ovplyvnovat celu architekturu, pretoze su zadavane hned na zaciatku

Nefunkcne poziadavky
- mozu generovat funkcne poziadavky

25. Ornament diagramu aktivit fork oznacuje :
a. zaciatok paralelneho spracovavanie
b. nepouziva sa v diagramoch aktivit
c. rozhodovaci bod

26. Aky je rozdiel medzi programom a programovym systemom – vyrobkom (v) :
a. Program je suhrn programovych systemov
b. Programovy system pouziva autor sam v podmienkach, pre ktore ho vyvinul
c. Program pouziva autor sam v podmienkach, pre ktore ho vyvinul
d. Programovy system je otestovany a jeho pouzitie splna dohodnute obmedzenia
e. Programovy system je to iste ako informacny system
f. Program je synonymum pre operacny system pocitaca
g. Programovy system moze byt 9-krat drahsi ako program

27. Co patri medzi druhotne dovody softverovej krizy (v):
a. prekrocenie rozpoctu
b. Praca v time
c. nizka kvalita
d. Programatorska produktivita. Niektori programatori su az 10 krat produktivnejsi.

29. Objektovo – orientovany pristup sa od strukturovaneho odlisuje (v) :
a. zoskupenie dat a aplikacnej logiky, ktora s datami manipuluje
b. znovopouzitelnostou urcitych casti kodu
c. oddelenim datovej, aplikacnej a prezentacnej vrstvy
d. stavbou programu

30. Obdlznikom v sekvencnych diagramoch zakreslujeme :
a. sekvenciu
b. objekt
c. triedu
d. spravu

31. Co nepatri medzi popis operacie a jej parametrov (v) :

a. Viditelnost
a. Nasobnost
b. Nazov
c. Navratova hodnota
d. Smer parametrov
e. Datum

32. Co je to domenova trieda :

a. Trieda, ktoru oznacime stereotypom << domain >>
b. Trieda, ktora v sebe obsahuje nazov webovej domeny zakaznika (napr. sk)
c. Trieda, ktora je najvăcsia a dominuje nad ostatnymi (moze ich byt viac)
d. Trieda, ktora je klucova pre danu oblast (napriklad ucet v bankovom systeme)

34. Sekvencne diagramy sa pouzivaju v prvom rade na :

a. modelovanie komunikacie medzi jednotlivymi objektmi
b. modelovanie procesnych tokov
c. sekvencie jednotlivych etap pri vyvoji softveru
d. modelovanie sekvencie prikazov ramci konkretnej metody

35. Co znamena, ze funkcne poziadavky maju byt kompletne :

a. V poziadavkach musia byt aj funkcne aj nefunkcne poziadavky
b. Poziadavkam nesmu chybat formalne nalezitosti ak je napr. datum a cas zadania poziadavky a meno toho kto poziadavku zadal
c. Dokument poziadaviek musi obsahovat aj textovu formu poziadaviek aj diagramovu (use case)
d. V dalsich fazach vyvoja uz nemaju vznikat nove poziadavky

36. Generalizacia a klasifikacia (v):

a. Generalizacia je tranzitivna, klasifikacia nie je tranzitivna
b. Generalizacia nie je tranzitivna, klasifikacia je tranzitivna
c. Klasifikacia je opakom generalizacie
d. Klasifikaciu je vhodne vytvorit ako dedenie
e. Generalizaciu je vhodne vytvorit ako dedenie
f. Generalizacia je opakom agregacie
g. Generalizacia je opakom klasifikacie

37. Ciastocnymi rieseniami, ktore riesia dovody softverovej krizy su (v):

a. Vodopadovy model
b. Pristup „Kupovat programove systemy namiesto vytvarat“
c. Prototypovanie programovych systemov – asi aj toto
d. Yourdonove diagramy
e. Aplikovanie pocitacovej vedy do softveroveho inzinierstva

39. Co je trieda (v):

a. Trieda je skupina prvkov v ramci systemu
b. Trieda je kategoria, alebo skupina veci, ktore maju podobne vlastnosti a rovnake, alebo podobne chovanie.
c. Trieda je instanciou objektu
d. Trieda je zakladnym stavebnym prvkom v objektovo – orientovanom programovani. Logicky spaja udaje a operacie
e. Trieda je kategoria v hierarchickom pohlade na model programoveho systemu

40. Diagram objektov predstavuje mnozinu instancii tried a vztahov medzi nimi v urcitom okamihu :

ANO
NIE

41. Medzi hlavne vyhody vodopadoveho modelu patria (v) :

a. Vytvara dostatok dokumentacie
b. Pravidelna validacia zo strany zakaznika
c. Jednoducha kontrola pri riadeni
d. Je pruzny pri zmene situacie, ktora moze pri realizacii nastat
e. Umoznuje zakaznikovi modifikovat poziadavky podla aktualnych okolnosti

Medzi hlavne nevyhody vodopadoveho modelu patria

- Prilisna jednoduchost
- Dodanie programoveho systemu zakaznikovi az na konci

42. Ake vlastnosti ma objekt oznacovany ako repository :

a. Tento objekt sluzi na prenos dat, spravidla nema ziadne chovanie
b. Tento objekt existuje v programe ako jedina instancia

c. Tento objekt je ulozeny databaze (napr. relacnej)
d. Tento objekt nie je mozne instanciovat

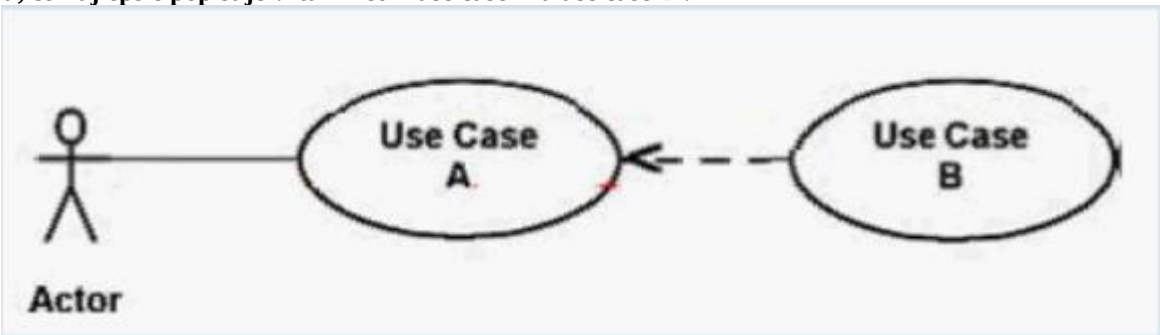
43. Sondy v sekvencnych diagramoch predstavuju (v) :
a. odkazy na ine sekvencne diagramy
b. kontrolne body
c. odkazy na externe prvky systemu
d. staticke datove struktury

44. Ktory diagram nie je sucastou UML
composite structure diagram
a. entity diagram
b. component diagram
c. class diagram
d. package diagram

42. Ziskavanie poziadaviek moze byt vo forme (v) :
a. Bezneho jazyka
b. Faktury za programovy system
c. V klasickom modely zivotneho cyklu slovnym dohovorom
d. Grafickych notacii

43. Co je atribut (v):
a. Atributom je napriklad „datumObjednania“ v triede „Objednavka“
b. Atribut nie je mozne stotoznit s asociaciou
c. Atribut deklaruje chovanie triedy
d. Atributom je napriklad „vyratajDan()“ v triede „Objednavka“
e. Atribut je vlastnost triedy
f. Atribut moze vzniknut ako vysledok asociacie

44. Na obrazku, co najlepsie popisuje vztah medzi use case A a use case B :



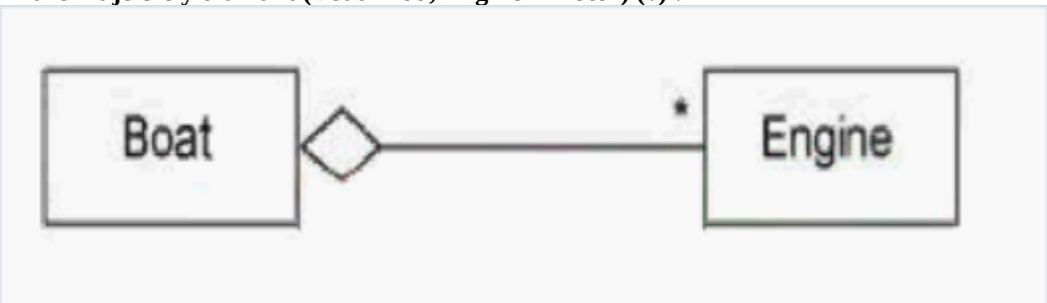
a. use case B rozsiruje use case A
b. use case A rozsiruje use case B
c. use case B generalizuje use case A
d. use case A zahŕna (include) use case B
e. use case A generalizuje use case B
f. use case B zahŕna (include) use case A

45. Co je UML (v)?
a. Metodika vytvorena firmou Rational
b. Metoda na tvorbu specifikacie programoveho systemu
c. Diagram architektury programoveho systemu
d. Specialny programovaci jazyk na vytvaranie ekonomickych aplikacii
e. Graficka technika
f. Jazyk na vizualizaciu, specifikaciu, navrhovanie a dokumentaciu programovych systemov

46. SW projekt je postupne vypracovavany znamena, ze :
a. pocas priebehu projektu sa objavuju nove informacie, na zaklade ktorych prijat spravne rozhodnutia
b. programovy system sa vytvara postupne vo fazach podla vodopadoveho modelu
c. projektova dokumentacia sa postupne vypracovava pokym dojde k implementacii a realizacii
d. projekt sa pestuje nie buduje

47. Co specifikujeme funkcnymi poziadavkami (v):
a. Aky dizajn/grafiku (v pripade webovej) aplikacie ma system pouzivat
b. Aka metodika ma byt pri vyvoji pouzita
c. Co system v definovanych situaciach robit nesmie
d. Ako bude system reagovat na vstupy a ako sa bude chovat v roznych situaciach
e. Akymi platnymi pravnymi predpismi sa programovy system ma riadit

48. Na obrazku, co znazornuje biely diamant (Boat = lod, Engine = motor) (v) :



a. Lod obsahuje motor
b. Motor umiestneny v lodi moze byt sucasne sucastou niecoho ineho v rovnakom case
c. Ak je zmazana lod, instancie jej motora su zmazane tiez
d. Motor nemoze byt odstraneny z lode

1) Co je charakteristicke pre analyticky model v UML
a) Definuje ako to ma system robit a nie co ma robit
b) Jeho vystupom je fyzicky narvh databazy
c) Definuje co ma system robit a nie ako to ma urobit
d) Definuje co a ako ma programovy system urobit
e) Jeho vystupom su „ Analyticke triedy „
f) Jeho vystupom je Model zavisly na platforme
g) Pouziva stavove diagramy

2) Priradte jednotlivym fazam vyzrievania softveru v case typy udrziavania softveru
- a) Faza zrelosti **- doraz na rozsierenie funkcnosti**
- b) Faza rastu **- doraz na opravu chyb**
- c) Faza upadku **- doraz na rast know-how**
- d) Uvodna faza - , **dotaz na podporu uzivatelov**

moznosti : doraz na rast know-how, dotaz na rozsierenie funkcnosti, dotaz na podporu uzivatelov, doraz na opravu chyb,

3) Pri pouziti riadenia projektu v realnom case (jedna) ???
a) sa spusti napravne opatrenie, ak extrapolacia trendu dosiahne limitne hodnoty
b) sa spusti napravne opatrenie, ak prognozy buduciach rozdielov su vacsie ako limitne hodnoty (asi)
c) sa spusti napravne opatrenie, ak je odchylka vacsia, ako vopred stanovena hranicna hodnota

4) Jazyk UML oznacujeme ako deskriptivny pretoze (jedna)
a) pouziva anglicke slova pri popise diagramov
b) pretoze jeho pouzitie je zalozene na konvencii a odchylky alebo vlastne znacky neznehodnucuju jeho vypodevaciu schopnost
c) pretoze opisuje chovanie a strukturu programoveho systemu
a) pretoze pouziva elementy, ktore vychadzaju z deskriptivnej geometrie (tvary elementov)

5) **Model interakcii systemu**
a) pri objektovom modelovani vyuziva hlavne Use case diagram
b) Definuje vztah systemu s pouzivatelom pripadne inymi systemami
c) Pri objektovom modelovani vyuziva hlavne diagram tried
d) Definuje zakladne stavebne casti systemu (napr. triedy)

6 **) Ktore tvrdenie je pravdive o implementacii rozhrani**
a) Rozhrania implementovane triedou predstavuju jej vyzadovane rozhrania
b) Rozhranie moze implementovat ine triedy (viac ako jednu)
c) Rozhrania implementovane triedou predstavuju jej poskytovane rozhrania
d) Trieda implementujuca rozhranie musi splnat podmienky rozhrania
e) Trieda moze implementovat iba jedno rozhranie

8) Model systemu
a) Nemusi byt kompletny, moze zachytavat iba niektore perspektivy
b) Musi byt spravny
c) Je vytvoreny vylucne pouzitim UML diagramov
d) Nemusi byt spravny, staci, ze mu rozumie zakaznik
e) Musi byt kompletny, podobne ako poziadavky

10) Analyticky model
a) Je urceny hlavne pre komunikaciu s programatormi
b) Je urceny pre komunikaciu so zakaznikom
c) Ma obsahovat co najviac tried buduceho programoveho systemu
d) Obsahuje implemenacne detaily dolezitych domenovych tried

e) Obmedzuje sa hlavne na domenove triedy

Analyticky model

- Obsahuje napriklad analyticke diagramy tried a use case
- Definuje co ma programovy system robit, nedefinuje ako to ma robit

11) Dedenie

a) Porusuje zakladny princip OO programovania – polymorfizmus
b) Je silny nastroj OO programovania, ktory sa malo vyuziva
c) Je silny nastroj OO programovania,ktory sa vyuziva casto nepsravne
d) Vyuziva princip Liskovej substitucie
e) Porusuje zakladny pricip OO programovania – zapuzdrenie

12) Vyvoj softveru je ukonceny (jedna) S

a) Po podpisani preberacieho protokolu
b) Po skusobnej prevadzke
c) Nie je ukonceny nikdy, vzhladom na dynamicke prostredie pouzivania softveru
d) Po implementacii

14) Reverzne inzinierstvo

a) Proces analyzy softveru z dovodu identifikacie softverovych komponentov a ich vazieb a vytvorenia reprezentacie softveru v inej forme, alebo na vyssej urovni abstrakcie
b) Meni funkcnost softveru a vysledkom je novy softver
c) Proces analyzy softveru z dovodu identifikacie softverovych komponentov a ich vazieb a vytvorenia reprezentacie softveru v inej forme, alebo na nizsej urovni abstrakcie
d) Nemeni funkcnost softveru a vysledkom nie je novy softver

Reverzne inzinierstvo

e) • Proces analyzy softveru z dovodu identifikacie softverovych komponentov a ich väzieb a vytvorenia reprezentacie softveru v inej forme, alebo na vyssej urovni abstrakcie
f) • Je to pasivna technika – nemeni funkcnost a vysledkom nie je novy softver
g) • Redokumentacia
h) • Design recovery
i) • Data RE

15) Analyza vplyvu sa vykonava

a) Pre potrebu odhadnut mieru rizika v dosledku modifikacie softveru
b) Z dovodu urcenia vhodnej technologie modifikacie softveru
c) Z dovodu potreby odhadnut zdroje pre modifikaciu sofveru
d) S cielom identifikovat vsetky komponenty, ktore budu ovplyvnene poziadavkou na modifikaciu

16) Ktora kategoria nepatri do klasifikacie diagramov UML

a) Diagramy balikov
b) Diaramy chovania
c) Diagramy interakcii
d) Diagramy struktur
e) Use Diagramy

18) Adaptivna udrzba

a) Je proaktivna
b) Je reaktivna
c) Rozsiruje funkcnost softveru
d) Opravuje funkcnost softveru

1**. Rozpocet projektu reprezentuje ??? (Jedna alebo viac)**

a) Kratkodoby plan
b) Strednodoby plan
c) Dlhodoby plan

2. **Rozvrh projektu – dala by som to takto**

a) Je jedna z troch hlavnych sucasti planu projektu
b) Zachytava planovane vykonavanie projektovych uloh a casovych intervalov, v ramci ktorych su potrebne rozne zdroje (ludia, vybavenie) na realizaciu aktivit potrebnych k uspesnej realizacii projektu
c) Rozvrh je nevyhnutny pre zhotovenie projektoveho planu ale nie pre riadenie zmien projektoveho planu
d) Rozvrh projektu nemoze byt prezentovany Ganttovym diagramom

3**. Modelovanie udajov**

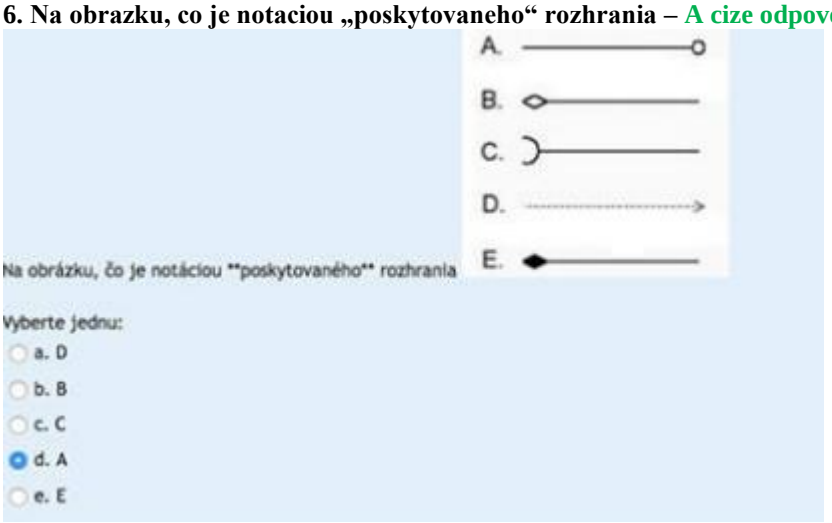
a) Modeluje triedy a chovanie triedy t.j. metody
b) Je typicke pre objektovo orientovane modelovanie
c) Modeluje datove entity a ich atributy

d) Je typicke pre strukturovane modelovanie
e) Modeluje vystupny format dat z programoveho systemu

4. Reinziniering
a) Sa nepouziva na zlepsovanie udrziavatelnosti, ale na nahradenie zostarnuteho softveru
b) Je analyza a alternacia softveru s cielom jeho reorganizacie do novej formy a jej implementacia
c) Je reorganizacia softveru bez zmeny funkcnosti – zlepsenie struktury a udrziavatelonsti

5**. Najvyssi podiel na udrziavanie softveru ma (jedna) S**
a) Zdokonalujuca udrzba
b) Preventivna udrzba
c) Korekcna udrzba
d) Adaptivna udrzba

6. Na obrazku, co je notaciou „poskytovaneho“ rozhrania – A cize odpoved D



7. **Adaptivna udrzba zahrna (jedna) S**
a) Zabezpecenie funkcnosti softveru do odstranenia poruchy
b) Modifikaciu softveru na zlepsenie vykonnosti
c) Modifikaciu softveru reagujucu na zmeny prostredia

8. Medzi typy vztahov medzi triedami nepatri (jedna a viac) asi S
a) Generalizacia
b) Agregacia
c) Eskalacia
d) Asociacia
e) Abstrakcia

9. **Podiel jednotlivych kategorii udrziavania softveru S**
a) Preventivna - 4%
b) Korekcna – 21%
c) Adaptivna- 25%
d) Zdokonalujuca – 50 %

10**. Objektovo orientovany pristup sa od strukturovaneho odlisuje** ?? (jedna a viac)
a) **Oddelenim datovej, aplikacnej a prezencnej vrstvy** – mozno aj toto
b) Zoskupenim dat a aplikacnej logiky, ktora s datami manipuluje
c) Konceptom zasielanie sprav
d) Znovupouzitelnostou urcitych casti kodu – toto urcite

12. Oznac viacero moznosti udalosti v stavovom diagrame – ASI
a) Signal
b) Sprava
c) Volanie
d) Necinnost
e) Zmena
f) Chyba
g) Cas

13. Vztah clovek ma krvnu skupinu A* by ste vyjadrili ? (jedna)
a) Dedenim medzi triedou Clovek a triedou Skupina (Skupina je rodicovska trieda)
b) Dedenim medzi triedou Clovek a triedou Skupina (Skupina je rodicovska trieda)

c) Kompoziciou triedy Clovek a triedou Skupina (Skupina je sucastou cloveka)
d) Volanim operacie skupina () na triede Clovek
e) Agregaciou triedy Clovek a triedy Skupina (Skupina je sucastou cloveka)

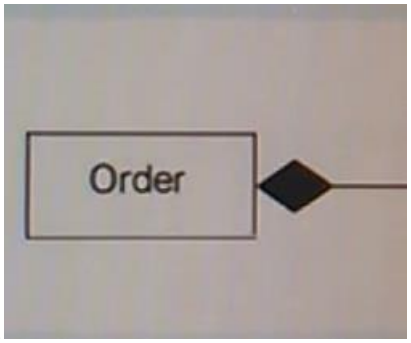
1**. Co je metoda a operacia v UML S**
a) Metoda ma navratny typ,operacia je typu void
b) Operacia je v konceptualnom modely popisom chovania triedy
c) Operacia ma navratovy typ, metoda je typu void
d) Metoda je implementaciou operacie
e) Operacia je implementaciou metody

2. **Medzi ulohy analyzy vplyvu patri S**
a) Analyza poziadavky na modifikaciu
b) Ziskanie suhlasu na organizaciu nestrandartneho rezimu prace IS
c) Ziskanie suhlasu na zvolenu modifikaciu
d) Domumentacia poziadavky a vysledkov modifikacie
e) Replikacia a verifikacia problemu

milnik ?
a) Nemoze obsahovat testovanie
b) Reprezentuje produkt alebo udalost
c) Je dolezita udalost v zivotnom cykle projektu. Reprezentuje kontrolny bod pre aktivity - dala by som toto

Activity diagramy : S
Jedna alebo viac
A, **nie su viazane iba na jeden objekt**
B, umoznuju paralelne spracovanie
C, su viazane iba na jeden objekt
D, **umoznuju iba sekvencne spracovanie**
E, umoznuju znazornit aloritmus
F, **umoznuju znazornit algoritmus, ale iba sekvencne**

Liskovej princip substitucie
Jedna alebo viac
A, formalne vyjadruje za akych okolnosti je vhodne medzi dvoma triedami
B, hovori ze aby bol dodrzany princip chovania nadtriedy je podriadeny ... (nwm dalej)
C, princip vymenovava metody nadenia tried vyuzitim roznych metod ... (nwm dalej)
D, hovori ze pravouholnik nieje stvorec
E, hovori, ze trieda moze byt typom inej tiredy ale instancie podtried.. (nwm dalej)



Na obrazku co znazornuje cierny diamant
Jedna laebo viac asi ??
A, ak je zmazany order instancie jeho line item zostavaju
B, ak je zmazany order jeho line item su zmazane tiez
C, line item moze byt sucastou len jednej instancie order
D, line item nemoze byt odstraneny z order v ktorom je
E, line item musi byt sucastou jednej instancie order

Co je vhodnym prikladom generalizacie
Jedna
A, **namiesto kniha, casopis, pouzivat pojem literatura**
B, namiesto auto pouzivat pojem osobne auto , nakladne auto
C, namiesto podtriedy pouzivat v programe vzdy jej nadtriedu ...(nwm dalej)

Medzi vyhody specializovanych oddeleni udrzby patri S
Jedna alebo viac
A**, vytvorenie komunikacnych kanalov**
B, znizenie zavislosti na jednotlivcoch
C, specializacia

D, integracia z oddelenim vyvoja

Na obrazku co v obcoh diagramoch predstavuje size (nemam obrazok) ☹ S
Jednu
A, size su 2 vlastnosti, ktore sa neprekryvaju pretoze su private
B, size je vlastnost ktora je znazornena ako atribut aj ako asociacia
C, size nemoze byt pouzity ako nazov atributu aj ako nazov asociacie
D, size ako atribut je sucastou triedy window /diagram vlavo/ size ako ..(nwm dalej)

Co znamena skratka UML
Jednu
A, unified method language
B, unified modeling language
C, UML modeling language
D, unified methodology language
E, universal modeling language

Medzi diagramy struktur nepatri S
Jedna alebo viac
A, diagram balikov / zoskupene
B, diagram tried
C, aktivity diagram
D, k**omuikacny diagram**

Softverova metrika S
Jedna alebo viac
A, pocet obrazoviek a tlacovych vystupov
B, pocet riadkov kodu
C, jednotka merania sw
D, **pocet entit, atributov a vztahov**
E, zakladna velicina vykonu sw

aktivity udrziavania sw zahrnaju
Jedna alebo viac
A, riadenie modifikacie sw
B, riadenie kazdodennej funkcionality
C, prevenciu znizovania vykonnosti
D, identifikaciu bezpecnostnych hrozieb a opravu zranitelnych miest
E, zlepsovanie existujucej funkcionality

Rozdiel medzi abstrakciou a generalizaciou
Jednu
A, abstr. Odbera detaily a vytvara domenovy objekt, pri generalizacii n....
B, general. Pouziva vytvara domenove objekty
C, prakticky ziadny – synonymum
D, gener. Odbera detaily a vytvara domenovy objekt ...
E, gener. Pridava detaily abstrakcia odobera

Reverzne testovanie sa pouziva
?nwm kolko spravne asi 1?
A, ... priciny reportovanej chyby sw
B, **.... efektivnosti modifikovanych komponentov asi** ☺
C, ... ze modifikacia nema nezamyslane dosledky

Ak je medzi triedami kompozicny vztah hovorime ze
Jedna alebo viac
A, nadriadena trieda dedi atributy a metody podriadenej
B, **nadradena trieda je zodpovedna za zivotny cyklus podriadenej**
C, medzi triedami je vztah typu 1:1
D, ide o tzv silnu agregaciu

Ktory z diagramov nepatri UML diagramy chovania
- Dataflow diagram

Co nie je sucastou validacie poziadaviek
- Formalna spravnost UML diagramov
- Cena poziadavaiek

Ake su zakladne charakteristiky scenara v use case diagramoch?

- Alternativne scenare mozu byt jednoduche alebo zlozite
- Je zapisany v textovej forme, pricom tato forma nie je definovana v UML
- Zlozity alternativny scenar nemal mat ziadne dalsie alternativne scenare

Abstrakcia
- Je filtrovanie vlastosti a operacii objektu, pokial nezostanu len tie, ktore potrebujeme

Kto alebo co je actor – pouzivatel
- Jeho lepsim vyjadrenim je pojem „rola“
- Moze to byt clovek alebo system

Naco sluzi model zivotneho cyklu softveroveho procesu?
- Rozdeluje proces na etapy, etapam prideluje cinnosti
- Robi system vyvoja programovych systemov kontrolovatelnejsim

Aka je charakteristika vnutornych dovodov softverovej krizy
- Tieto dovody su neodstranitelne, nie je predpoklad, ze by ich bolo mozne obist

K zmenam v poziadavkach
- Dochadza k nim celkom prirodzene napriklad z dovodu zmeny prostredia

Softverovy proces je
- Prostupnost aktivit specifikacia, vyvoj, validacia a evolucia
- Mnozina aktivit a prislusnych vysledkov, ktorych vysledkom je programovy system

Ciastocnym riesenim, ktora nema kvality striebornej gulky je inkrementalny vyvoj. Co sa pod nim chape?
- Programovy system sa pestuje, nie stavia
- Budovanie programoveho systemu zacina malym plne funkcnym systemom a postupne nabera pozadovane funkcionality

Co je pouzivatelsky scenar
- Je textova forma zapisu postupnosti krokov
- Sada postupnosti krokov spojena spolocnym zamerom pouzivatela

Na co sluzia stereotypy
- Na oznacenie tried, ktore maju spolocne charakteristiky
- Stereotyp oznacujem nazvom uvedem v zatvorkach <<nazov>>

Ake su zakladne elementy sekvencneho diagramu
- Trieda
- Posielanie sprav

Na obrazku vztah medzi osobou (person) a autom (car)
- Jedna osoba moze mat ziadne alebo viac aut
- Definova ako obojsmerny vztah

Abstrakcia a generalizacia su s ostatnymi pojmami vo vztahu
- Specializacia je opakom generalizacie
- Konkretizacia je opakom abstrakcie

Co patri medzi charakteristiky RUP
- Je objektovo orientovana
- Je prepojena s UML

Co je specifikovane pomocou use case
- Pozadovane pouzitie systemu

Co je to softverove inzinierstvo
- Aplikovanie systematickeho, disciplinovaneho, kvantifikovatelneho pristupu k tvorbe, prevadzke a udrzbe softveru, tj aplikovanie inzinierstva na softver
- Systematicky pristup k analyze, dizajnu, odhadovaniu, implementacii, testovaniu, udrzbe a reinzinieringu sotveru aplikovanim inzinierskych postupov

Akym sposobom mozu byt reprezentovane vlastnosti triedy
- Ako asociacie

- Ako atributy triedy

Aka je to konceptualna perspektiva UML
- Vytvorenie slovnej zasoby pre domenu

Strukturalny model systemu
- Pri objektovom modelovani vyuziva hlavne diagram tried
- Definuje zakladne stavebne casti systemu (napr. Triedy)

Co dostavate kupou metodiky RUP
- Sadu dokumentov a softverovych nastrojov dokumentujucich a podporujucich metodiku

Co je najvhodnejsim opisom aktora v UML
- Pouzivatelia alebo externe systemy mozu byt aktorom

Medzi hlavne vyhody spiraloveho modelu patri
- Komplexnost
- Vytvaranie znovupouzitelnych komponentov

Modelom riadeny navrh a vyvoj
- Pri tomto type modelovania je zakladnym prvkom model, na zaklade ktoreho sa robi implementacia. Pri zmenach sa zmeny prenasaju naprv do modelu a potom do implementacie

Aky je rozdiel medzi agregaciou a kompoziciou
- Pri kompozicii spolu s objektom zanika aj komponovany objekt
- Obe su detailnejsim vyjadrenim asociacie

Co urcuje metodika RUP
- Urcuje kto, co, ako a kedy

Co nepatri medzi dosledky softverovej krizy
- Zlozitost softveru
- Vysoka cena hardveru oproti softveru

Vztah <<include>> use case diagram
- Je sucastou hlavneho scenara a nejedna sa o alternativny scenar

Dokument specifikacia poziadaviek
- Sa po skonceni etapy odovzda zakaznikovi a doplnenie zmien do dokumentu je vitane
- Urcuje len „co“ ma programovy system robit, nie ako

Specifikacia poziadaviek
- je zhmotnena v podobe dokumentu „Dokument poziadaviek“
- je procesom ziskavania poziadaviek na programovy system

Ake vlastnosti ma objekt oznacovany ako value objekt
- Tento objekt sluzi na prenos dat, spravidla nema ziadne chovanie

Co zachytava use case
- Poziadavky na system
- Chovanie systemu

Co patri medzi vhodne nefunkcne poziadavky
- Aby programovy system mal odozvu do 2 sekund
- Aby programovy system vyuzival k autentifikacii ISIC kartu – Nie som si ista

CO patri medzi vhodne nefunkcne poziadavky
-aby programovy system dokazal obsluzit 1000 pouzivatelov

Medzi zakladne modely softveroveho procesu nepatri
- BPM (business proces medeling)
- RUP (rational unified proces)

Medzi prvotne problemy softveru nepratri
- Prekrocenie rozpoctu
- Nizka kvalita

V akych formach moze byt uvedena nasobnost

- 1 nasobnost vyjadrujuca prave jednu vazbu
- 0..1 nasobnost vyjadrujuca jednu vazbu alebo ziadnu vazbu

Kedy bol prvy krat vo svete oficialne pouzity termin Softveroveho inzinierstva

- Na konferencii NATO

Sekvencny diagram patri medzi

- Diagramy interakcii
- Diagramy chovania

Aka je to softverova perspektiva UML

- Element diagramy UML su prenasane do softveru ako softverove elemetny

Co je charakteristicke pre neviditelnost ako dovod softverovej krizy

- Neexistuje sposob vizualizacie softveru tak aby pokryl vsetky jeho aspekty

Desat zamestnancov oddelenia moze pomocou formulara zaslat svoje cestovne vydavky aby im boli preplatene. Manazer musi tento formular (formulare) schvalit, kolko je v tomto pripade aktorov.

- 2 aktori

Ktora etapa zivotneho cyklu softveroveho procesu trva vo vseobecnosti najdlhsie

- Evolucia

Moznymi kandidatmi na striebornu gulku su

- Neexistuju ziadny kandidati na striebornu gulku pretoze by musela vyriesit naraz vsetky hlavne dovody softverovej krizy

Co musi obsahovat atribut triedy v uml diagrame

- Nazov

Na co sluzi model systemu

- Model je abstrakciou toho, comu chceme porozumiet este predtym, ako to vybudujeme
- Model neobsahuje nepodstane detaily

Co nepatri medzi charakteristiky vodopadoveho modelu

- Prace na dalsej etape zacinaju az po skonceni predchadzajucej
- Iterativny vyvoj

Co znamena pouzitie UML ako nacrtku

- Ak je UML nakreslene strucne (elektronicky alebo na papieri) a nesplna vsetky detailne nalezitosti diagramov

Udajovy slovnik

- Obsahuje zoznam datovych objektov a popis ich atributov

Co je informatika

- Subor vednych disciplin a postupov pricspracovani informacii
- Informatika nie je viac o pocitacoch ako astronomia o teleskopoch
- Veda o informacii a jej spracovani

Co reprezentuje biely diamant v aktivity diagrame

- Rozhodovanie
- Spojenie

SW projekt je docasny znamena, ze

- Ma svoj zaciatok a stanoveny koniec

Preco je lepsim pomenovanim pre actora rola ako pouzivatel

- Pretoze jeden a ten isty pouzivatel moze mat v systeme viac roli co bude znazornene viacerymi aktormi

Ako je ovplyvneny softverovy proces typom ucastnikov, ktori v nom vystupuju

- Ovplyvnuju akym sposobom sa kombinuju casti softveroveho procesu tj typ vhodnosti motediky

Aky je rozdiel medzi scenarom a diagramom pripadov use case diagramov

- Pripad pouzitia je slovesne pomenovanie funkcie programoveho systemu. Diagram znazornuje interakciu medzi tymito pripadmi pouzitia
- Pouzitavelsky scenar je v textovej podobe a nie je sucastou UML, diagram je graficke znazornenie tychto scenarov, pricom pouziva iba nazvy scenarov

Co je to poziadavka
- Abstraktna formualcia sluzby, ktoru ma system poskytovat
- Detailne specifikovana poziadavka funkcie systemu

V akom diagrame sa objavuje element actor
- Use case diagram

Na obrazku, co je notaciou vyzadovaneho rozhrania
- **C _____**

Chyba nazov otazky
->b je typu c2

Co modeluje model softveroveho procesu
- Modeluje poradie aktivit a ich vzajomne prepojenie

Co znamena, ze funkcne poziadavky maju byt konzistentne
- Poziadavky si navzajom neodporuju

Ktory model nepatri medzi modely MDA
- Model nezavisly na UML
- Model zavisly na pocitaci

Co je zavislost triedy
- Predstavuje vyjadrenie vseobecneho vztahu tried, ktory hovori, ze ak sa zmeni jedna strana moze to sposobit zmenu druhej strany

Jan pracuje ako knihovnik. Moze si tiez poziciat knihu ako obycajny student. Kolko aktorov je potrebnych pri modelovani use case
- Dvaja neprepojeni aktori

Kompletne a konzistentne poziadavky
- v praxi je to tazko dosiahnutelny stav

Co predstsavuje cierny kruh, na obrazku, v aktivity diagrame
- zaciatok

Datami riadene modelovanie
- pri strukturovanom modelovani vyuziva DFD (data flow diagram)
- je typicke pre strukturovane modelovanie analyzy

Na obrazku, ktory use case nemusi byt dostupny a aj tak sa vykona scenar reprezentovany usa case D
- Use case C

Je programovy system softverom
- Ano, softverom moze byt aj programovy system

Ktory symbol, na obrazku, reprezentuje asynchronnu spravu
- B

Triedy do diagramu tried
- Analyzou slovies a podstatnych mien
- Je mozne najst vyuzitie CRC stitkov

Co je vhodnym prikazom abstrakcie
- Namiesto pojmu BMW a opel pouzivat pojem „auto“

Domenou riadeny navrh
- Vyuziva abstrakciu na vytvorenie domenovych objektov

Aky je primarny vyznam pouzitia balikov (package)

- Vytvorit menne priestory
- Zoskupenie modelovane elementy do logickych casti

6. Abstraktne triedy v diagramoch tried (v):

a.) zakreslujeme rovnako ako ostatne triedy, oznacime ju vsak klucovym slovom {abstract} – asi aj toto
b.) zakreslujeme prerusovanym obdlznikom
c.) nezakreslujeme
d.) zakreslujeme rovnako ako ostatne triedy, nazov zapiseme kurzivou

Co je specifikovanie pomocou use case

-pozadovane pouzitie systemu

Pouzivatel pouziva system na objednanie knih a system pouziva system kreditnych kariet na validaciu pouzivatelovej kreditnej karty, system na validaciu adresy a internu databazu pouzivatelov. Predpokladajme ze chceme namodelovat objednanie knih ako jeden scenar v use case. aky aktory budu pouzity??

-zakaznik , system kred kariet system kontroly adries

auto ma jedneho alebo viacerych majitelov sa da vyjadrti nasobnostou

1..*

Aky je to analyticky diagram tried

-elementy z diagramu sa nemusia preniest do programu (trieda v diagrame nemusi byt triedou v programe)
-obsahuje menej detailov (chybaju typy atributov , typy parametrov metod atd) nie su uvedene vsetkz operacie

Preco je jednotne programovacie prostredie (IDE) riesenim podstaty softv krizy

-IDE je riesenim **druhotnych **dovodov softerovej krizy
-IDE je riesenim pretoze poskytuje jednotny pristup k programovacim nastrojom, kniznicam a **i**

Naco vyuziva spiralovy model prototyp

-na odstranenie moznych rizik
-na overenie funkcnosti technologickych rieseni predtyn ako je vybrana konkretna

Model Driven Architektur (MDA) vyuziva UML ako

- programovaci jazyk

Co je asociacia

- vyjadrenie vztahu medzi dvoma tridaami
- je mozne zobrazit ako atribut

Sekvencne diagramy patria medzi

- diagramy interakcii , diagramy chovania

Kde bolo prvy krat vo svete oficialne pouzity termin SWI

- Na konferencii NATO

Co je to nasobnost

- rozsah dovolenych kardinalit

Ktore diagramy z UML mozu byt pouzite na znazornenie interakcie

- communication diagram
- sequence diagram

Co nieje vyjadrenim zavislosti triedy

- trieda je sucastou rovnakeho balika ako ina trieda

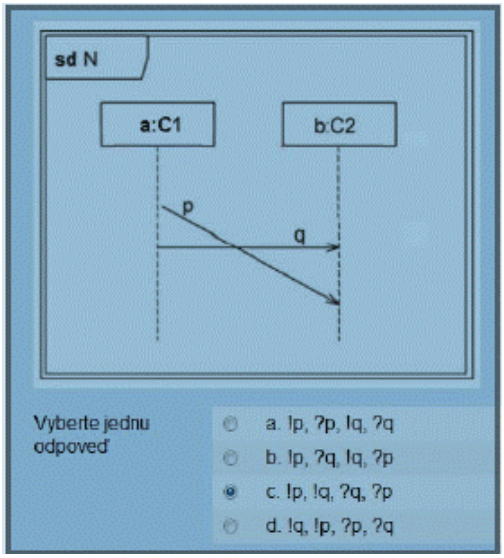
Na obrazku, kolki dalsi aktory „mozu“ byt zapojeni do scenara ak aktor X pouzije use case A

- 2

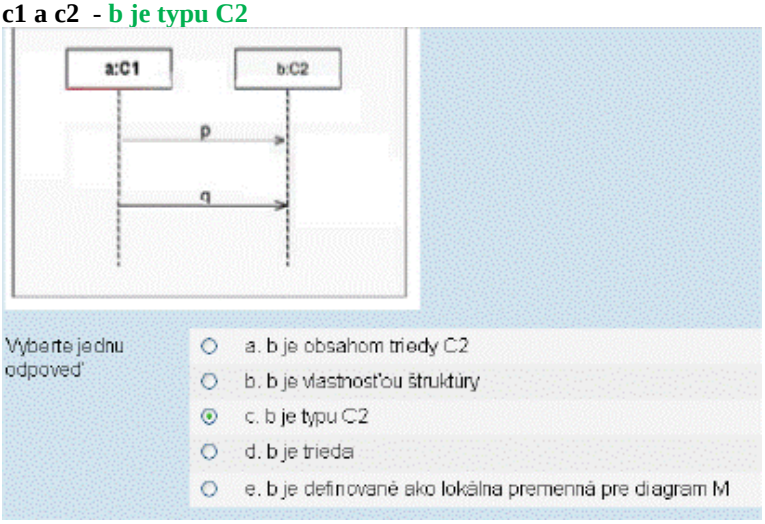
Na obrazku, co znazornuje cierny diamant

- ak sa vymaze to kde je cierny diamant, zrusi sa aj to druhe, kompozicia

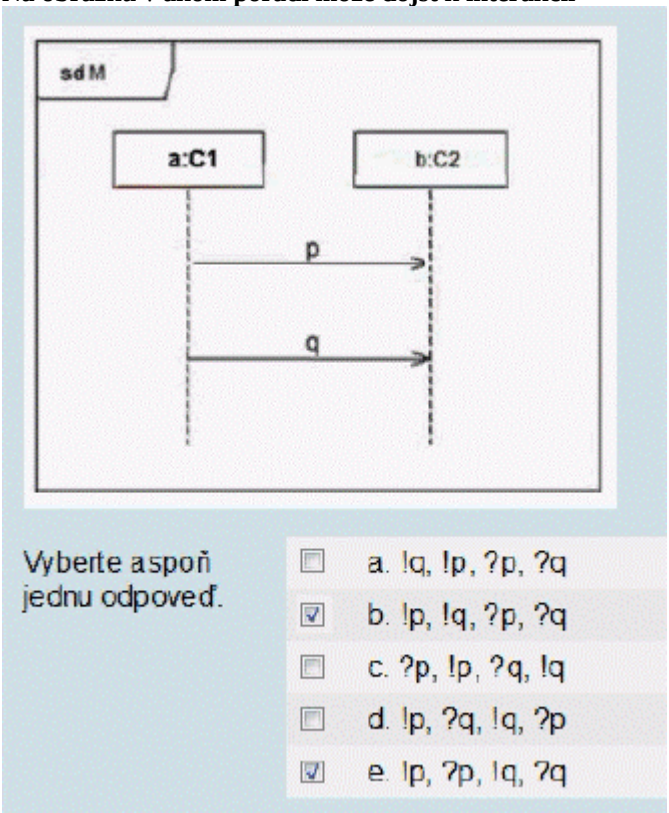
V akom poradi prebieha komunikacia na obrazku - C



c1 a c2 - b je typu C2



Na obrazku v akom poradi moze dojst k interakcii



ZAKLADY (FUNDAMENTY) UDRZIAVANIA SOFTVERU (US)

Definicie a terminologia

• Medzinarodny standard ISO/IEC/IEEE 14764

• Modifikacia softveru po dodani, za ucelom oprav chyb, zlepsenia vykonnosti alebo inych vlastnosti, alebo adaptacie na zmenene prostredie
• Ciel SU

Povaha udrziavania softveru
• Zaznamenavaju sa poziadavky na modifikacie
• Urci sa dopad pozadovanych zmien
• Modifikuje sa kod
• Otestuje sa zmena
• Nasadi sa nova verzia
• Skolenie a podpora uzivatelov

Povaha udrziavania softveru
V pouzivani je vela systemov starych aj 15 - 20 rokov
• Ich sucasna podoba je podstatne odlisna od vyvojovej verzie
• Povodni tvorcovia su casto nedosiahnutelni
• Je potrebne vychadzat z dokumentacie a kodu

Potreba udrziavania softveru
... softver je modelom realneho sveta ...
... svet okolo nas sa stale meni ...

Potreba udrziavania softveru
-US je potrebne, aby softver mohol nadalej sluzit uzivatelom. (bol operabilny)
- Nezalezi, aky model bol pouzity pri vyvoji softveru, US sa da aplikovat vzdy.

Potreba udrziavania softveru
-Softver uplne nenaplnil povodne poziadavky uzivatela
(tvorcovia softveru plne neporozumeli poziadavkam uzivatela)
- V softveri zostali chyby (vzdy)
- poziadavky uzivatela sa menia (uzivatel uz vie lepsie ako vyuzit softver)
-meni sa prostredie (zmeny legislativy, politik atd.)

Udrziavanie je potrebne
• Pre odstranenie chyb
• Zlepsenie navrhu
• Implementaciu rozsirenia funkcnosti
• Prepojenie s inym softverom
• Pre adaptaciu na zmeneny hardver, operacne prostredie
• Pre migraciu
• Pre vyradenie softveru z uzivania

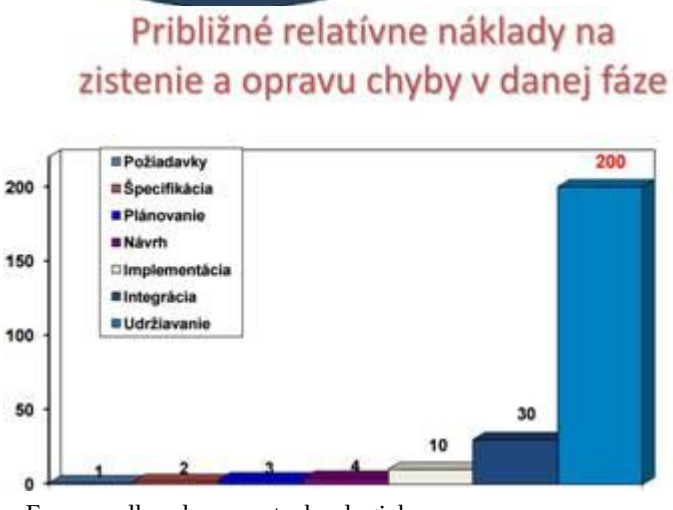
US aktivity zahŕnaju
• Riadenie kazdodennej funkcionality
• Riadenie modifikacie softveru
• Zlepsovanie existujucej funkcionality
• Identifikaciu bezpecnostnych hrozieb a opravu zranitelnych miest
• Prevenciu znizovania vykonnosti softveru

Naklady udrziavania softveru
• V sucasnosti je US najnakladnejsia cast zivotneho cyklu softveru (Foster [1993])
• Statistiky ukazuju, ze viac ako 80% tvoria ine ako korektivne aktivity (Pigosky 1997).
• Priklad
• Porozumenie faktorom vplyvajucim na udrziavatelnost softveru pomaha riadit naklady
• Operacne prostredie: – Hardver a softver
• Organizacne prostredie: – Politiky, procesy, pracovnici, konkurencia, ...

Priblizne naklady na jednotlive fazy zivotneho cyklu softveru (DOLE OBRAZOK)
Udrziavanie – 67 %
Integracia – 6%
Testovanie modulov -7%
Kodovanie modulov - 5%
Navrh - 6%
Planovanie - 2%
Specifikacia 4%

Percentualny podiel zdrojov chyb pri vyvoji softveru

Pracnost opravy chyb v zavislosti na zdroji chyb



Priblizne relativne naklady na zistenie a opravu chyby v danej faze

Evolucia softveru

- US predstavuje evolucny vyvoj softveru
- Vyvoj softveru nie je nikdy ukonceny
- Jeho komplexnost rastie

Posun v type US v priebehu casu

- Uvodna faza: doraz na podporu uzivatelov
- Faza rastu: doraz na opravu chyb
- Faza zrelosti: doraz na rozsirenie funkcnosti
- Faza upadku: doraz na technologicke zmeny

Kategorie udrziavania softveru
• Korekcna udrzba (oprava softveru) odstranovanie chyb, vratane havarijnych stavov a zabezpecenia funkcnosti do odstranenia poruchy
• Adaptivna udrzba : prisposobovacia udrzba – modifikacia softveru reagujuca na zmeny prostredia, napr. upgrade OS ...

Kategorie udrziavania softveru
• Zdokonalujuca udrzba – modifikacia softveru na zlepsenie vykonnosti alebo udrziavatelnosti softveru
• Preventivna udrzba : modifikacia softveru za ucelom zistovania a odstranenia skrytych chyb predtym, ako sa objavia

Podiel jednotlivych kategorii US

	oprava	rozsirenie
proaktivne	preventivne	zdokonalujuce
reaktivne	korekcne	adaptivne

Kategorie udrziavania softveru

KLUCOVE PRVKY UDRZIAVANIA SOFTVERU

US je vyzvou
• hladanie problemov v kode inych
• sutaz o firemne zdroje s vyvojom
• priprava nasledovnej verzie a sucasne riesenie problemov sucasnej verzie

US je vyzvou
• 1975: ~75,000 pracovnikov (17%)
• 1990: 800,000 (47%)
• 2005: 2,500,000 (76%)
• 2015: ??

Technicke otazky
• Obmedzene porozumenie
• Testovanie
• Analyza vplyvu
• Udrziavatelnost

Obmedzene porozumenie
• cca. polovica usilia v US je venovana pochopeniu problemu
• Pochopenie je narocnejsie v textovej reprezentacii – zdrojovy kod –
• tazko vysledovatelne zmeny vo verziach, ak nie su dokumentovane

Testovanie
• Potreba verifikovat reportovany problem prostrednictvom testov
• Regresne testovanie – vyberove testovanie sw. alebo komponentov na overenie, ze modifikacia nema nezamyslane dosledky
• Najst cas na testovanie je problem, najmä ak nie je mozne sw. testovat off-line
• Koordinacia testovania je narocna – rozni pracovnici pracuju na roznych problemoch sucasne
• Testovat produkcne systemy je vo väcsine pripadov nemozne

Analyza vplyvu
• Ciel – identifikovat vsetky komponenty, ktore budu ovplyvnene poziadavkou na modifikaciu
• Odhadnut potrebne zdroje
• Odhadnut mieru rizika v dosledku modifikacie
• Zavaznost problemu urcuje rozvrh riesenia

Ulohy analyzy vplyvu

• Analyza poziadavky na modifikaciu/ chybovej spravy
• Replikacia a verifikacia problemu
• Navrh moznosti pre implementaciu modifikacie
• Dokumentacia poziadavky a vysledkov modifikacie
• Ziskanie suhlasu na zvolenu modifikaciu

Udrziavatelnost

• Schopnost softveru byt modifikovany (IEEE 14764)
• Udrziavatelnost sa stava dolezitym kriteriom kvality softveru • Musi byt: – Specifikovana – Posudena – Riadena ... pocas vyvoja sw.

Kluc k US spociva vo vyvoji

• Vyssia kvalita => menej (korekcnej) udrzby
• Predvidanie zmien => menej (adaptivnej a zdokonalovacej) udrzby
• Vyssia podpora uzivatelskych poziadaviek => menej (zdokonalovacej) udrzby
• Menej kodu => menej udrzby
• Pritomnost procesov, technik a nastrojov pomaha zvysit udrziavatelnost

Otazky riadenia US

• Sulad s cielmi organizacie
• Personalne zabezpecenie
• Proces
• Organizacne aspekty udrziavania
• Outsourcing

Sulad s cielmi organizacie

• Organizacne ciele - zabezpecit navratnost investicii do udrziavania
• Vyvoj – projekt s definovanym casom a zdrojmi
• Udrziavanie – cielom je predlzit pouzivanie softveru co najdlhsie

Personalne zabezpecenie

• Udrziavanie sa casto nepovazuje za pritazlivu pracu
• Je problem personalne zabezpecit udrziavanie

Proces

• Na procesnej urovni maju aktivity US mnoho spolocneho s vyvojom (napriklad konfiguracny manazment je rozhodujuci pre oba)
• US vyzaduje aj unikatne aktivity – dolezite pre riadenie

Organizacne aspekty udrziavania

• Ktora zlozka organizacie je zodpovedna za udrziavanie softveru – nevyhnutne urcit
• Nemusi to byt vyvojovy tim
• Specializovane oddelenia udrzby maju niektore vyhody: – Vytvorenie komunikacnych kanalov – Znizenie zavislosti na jednotlivcoch – Specializacia ...

Outsourcing

• Outsorcovanie US sa skor uplatnuje pri nekritickom softveri
• Nie je mozne stratit kontrolu na softverom vyuzivanym v klucovych procesoch
• Vyzva pre outsorcera:
– Urcit rozsah sluzieb
– Podmienok poskytovania sluzby
– Podrobnosti zmluvy

Outsourcing

• Vyzaduje pociatocne investicie
• Vybudovanie infrastruktury
• Help desk
• Zamestnanie jazykovo zdatnych operatorov

Odhad nakladov udrziavania

• Odhad nakladov
• Parametricke modely
• Skusenosti
• Meranie

Odhad nakladov

• Na odhad nakladov vplyvaju rozne faktory – technicke aj netechnicke
• Pozivaju sa tri pristupy:
– Parametricke modely
– Skusenosti
– Ich kombinacia
• Specificke miery

Parametricke modely

• Su matematicke modely
• Je potrebne mat k dispozicii udaje z minulosti na kalibraciu modelov

Skusenosti

• Vo forme expertnych odhadov
• Odhad nakladov (vo forme cloveko casovych jednotiek) je najpresnejsi ak sa skombinuju skusenosti s historickymi udajmi
• udaje je mozne ziskat meranim

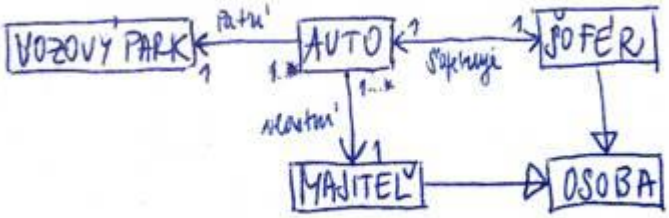
Meranie

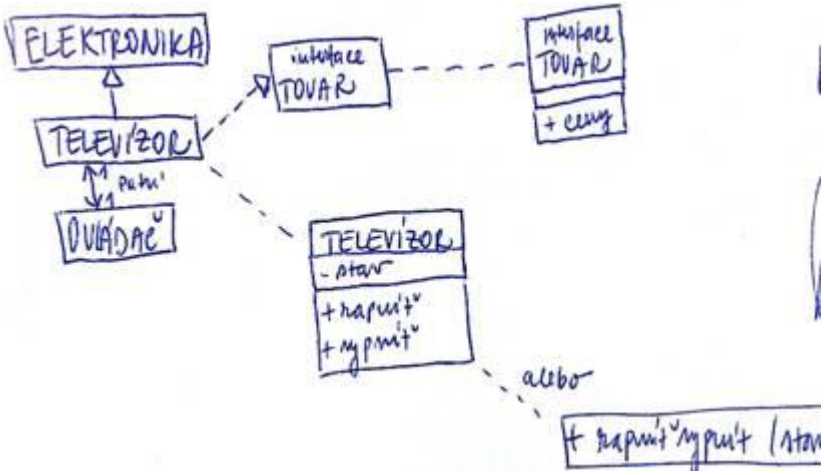
• Vlastnosti procesov, zdrojov a vysledkov je mozne merat v ramci SU

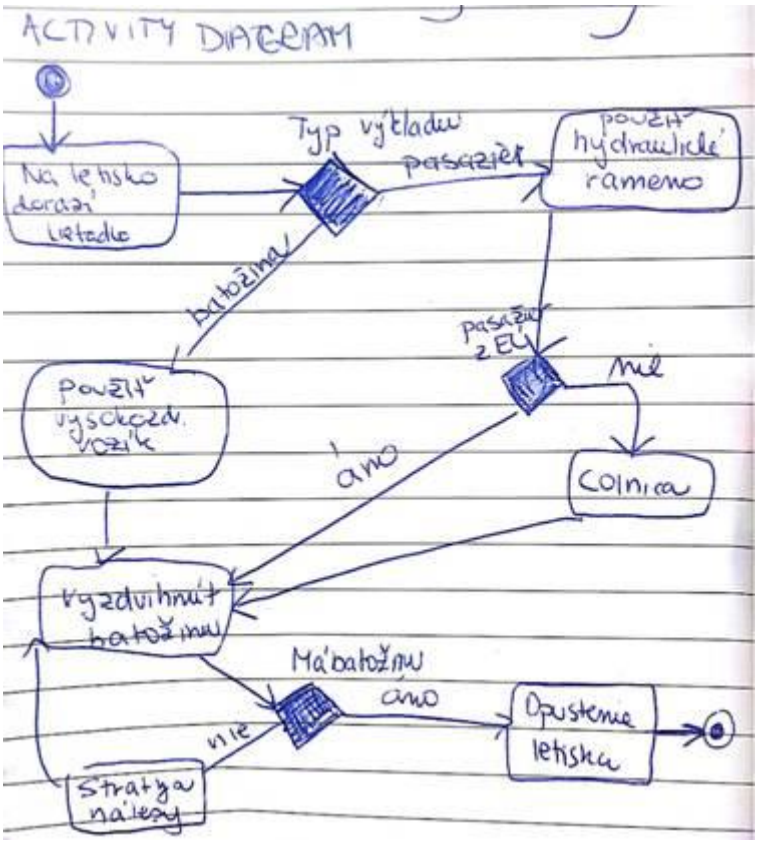
Specificke metriky
• Treba urcit vhodne metriky pre danu organizaciu
• Metriky udrziavatelnosti obsahuju:
– Analyzovatelnost – Menitelnost – Stabilita – Testovatelnost – Ine (velkost, zlozitost sw. ...)
TECHNIKY UDRZIAVANIA SOFTVERU
Pochopenie programov
• Citanie kodu s cielom porozumenia
• Prehliadace kodu – klucove nastroje
• Vyuzivane na organizaciu kodu
• Dobra dokumentacia ulahcuje pochopenie
Reinziniering
• Analyza a alternacia softveru s cielom jeho reorganizacie do novej formy a jej implementacia
• Nepouziva sa na zlepsenie udrziavatelnosti, ale na nahradenie zostarnuteho softveru
• Refaktoring je technika reinzinieringu, ktorej cielom je reorganizacia softveru bez zmeny funkcnosti – zlepsenie struktury a udrziavatelnosti

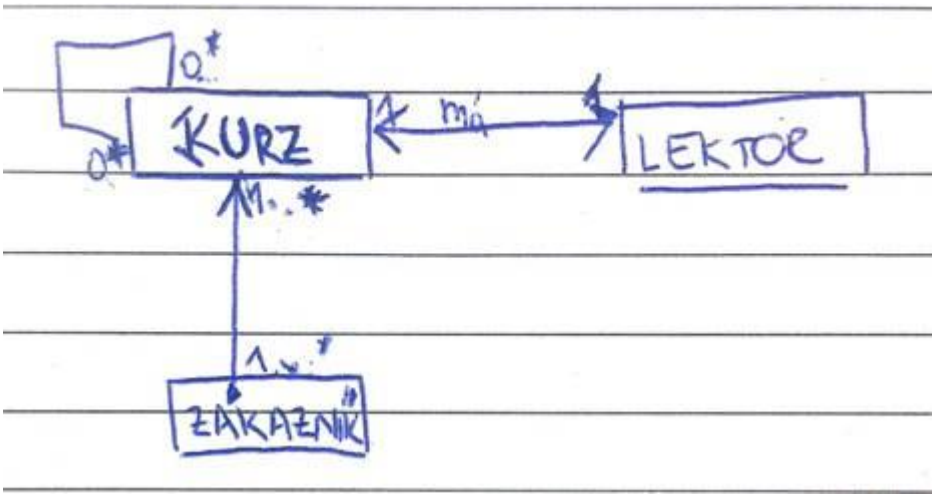
Migracia
• Modifikacia softveru s cielom jeho implementacie v odlisnom prostredi

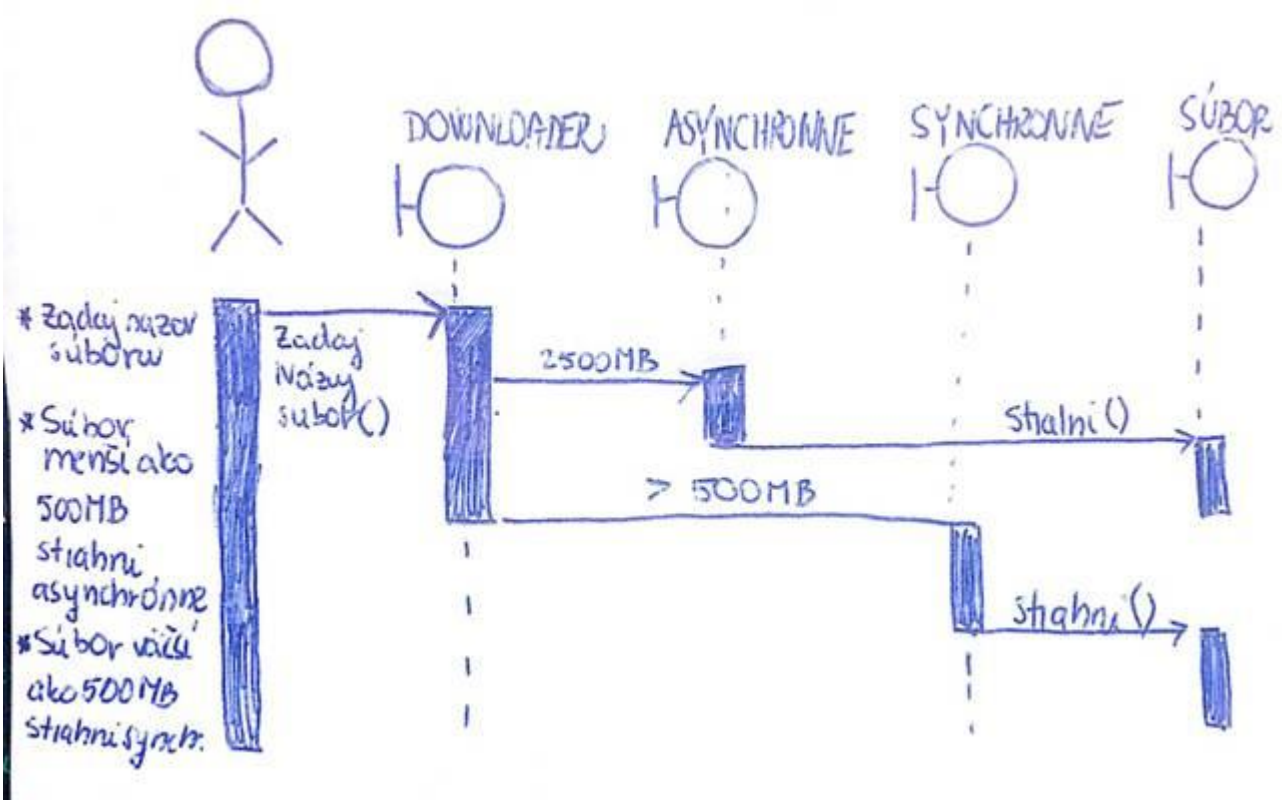
Vyradenie z pouzivania
• Je potrebne spracovat analyze
• Podklad pre plan vyradenia
– Poziadavky
– Vplyv
– Nahrada
– Rozvrh
– Pracovna sila
– Archivacia
NASTROJE UDRZIAVANIA SOFTVERU
Nastroje US
• Staticke analyzatory
• Dynamicke analyzatory
• Analyzatory toku dat
• Dokumentatory krizovych odkazov
• Analyzery zavislosti
• ...
Forrester analyst Duncan Jones: • It’s one of the unwritten rules of software that maintenance costs only go up — never down. But we used to say that about house prices too. Today, software vendors are coming under increasing pressure from customers to cut maintenance bills, but the vendors are robustly defending their lifeblood with age-old, inflexible policies. However, enterprises’ imperatives to reduce cost and the contrasting flexibility of new commercial models such as softwareas-a-service (SaaS) are making the perpetual licensors realize that they may have to change their ways if they are to be in place to win new business when budgets return.


VOZOVÝ PARK
patrí
AUTO
šofér
ŠOFÉR
vlastní
MAJITEĽ
OSOBA


ELEKTRONIKA
implementace TOVAR
interface TOVAR
+ ceny
TELEVÍZOR
patrí
DVÁDAČ
TELEVÍZOR
- stav
+ zapnúť
+ vypnúť
alebo


ACTIVITY DIAGRAM
Na letisko dorazí lietadlo
Typ výkladu
pasažier
použitie hydraulické rameno
batožina
pasažier z EU
áno
nie
použitie vysokozdv. vozík
Colnica
áno
Vyzdvihnúť batožinu
Má batožinu
áno
nie
Opustenie letiska
Strata a nálezy





domenovy model - http://sccg.sk/~paulis/oa11/domenove_modelovanie.pdf

diagram tried - http://hornad.fei.tuke.sk/~chodarev/zsi/ref/magyar-pulen.pdf

sekvencny + diagram tried - http://www2.fiit.stuba.sk/~bielik/courses/psi-slov/slajdy/vytah-dynamicke-modely.pdf

activity,tried,sekvencny - http://web.tuke.sk/fei-cit/babic/anis/ANIS_pred4%202015%20tlac.pdf

activity + sekvencny - http://is.muni.cz/th/207992/fi_b/text_prace.pdf

http://uml.czweb.org/sekvencni_diagram.htm

http://mpavus.wz.cz/uml/uml-sequence-7.php

1. Prednáška

Komplexné systémy

Vedci vo viacerých oblastiach skúmajú komplexné systémy (biologické, mechanické, vesmírne apod). Aby ich pochopili museli identifikovať ich základné vlastnosti a metódy ako tieto vlastnosti skúmať.

Medzi základné a spoločné vlastnosti komplexných systémov patrí:

1. Hierarchická štruktúra
2. Relatívne primitíva
3. Oddelenie záujmov
4. Spoločné vzory
5. Stabilné prechodné formy

Hierarchická štruktúra

Hierarchická štruktúra komplexných systémov znamená, že v systéme sa všetky prvky (časti) vyskytujú v hierarchii. Napríklad vesmírne objekty. Poznáme galaxie, ktoré sa skladajú z rôznych sústav (slnečných), tieto sú ďalej zložené z hviezd a tak ďalej. Ľudský sval je tiež zložený z viacerých snopcov tie z vláken, vlákna z myofibril a tie zo sarkomérov a tak ďalej.

Človek tiež vytváral komplexné systémy v súlade s touto vlastnosťou. Napríklad podniky. Majú riaditeľovstvo, to sa skladá z úsekov, úseky z oddelení a tak ďalej.

Relatívne primitíva

Primitívum je niečo čo nie je možné ďalej deliť. Relatívne primitívum je také, ktoré je nedeliteľné z jedného pohľadu no z iného pohľadu ho je možné ďalej deliť. Pri pohľade na ľudský sval je pre niekoho podstatné iba úroveň svalových vláken a myofibrily sú nepodstatné a dokonca môžu byť prekážkou pri analýze svalu, pretože by prinášali zbytočnú zložitosť do pohľadu na sval.

Na druhej strane, však môže existovať pohľad, ktorý potrebuje sval skúmať na nižšej úrovni a nedeliteľnou časťou je pre tento pohľad úroveň myofibrilu. A samozrejme môžu existovať pohľady, ktoré môžu ísť ešte “hlbšie”.

Komplexné systémy sa skladajú z takých prvkov, ktoré môžu byť v závislosti od uhla pohľadu chápaná ako nedeliteľné - tj. primitívne, no z iného uhla pohľadu sa môžu rozložiť do väčších detailov.

Oddelenie záujmov

V komplexných systémov je možné pozorovať, že rôzne časti systému majú rôzne záujmy (v zmysle rôzne úlohy). Tieto záujmy sa dajú relatívne dobre vymedziť a je jasné kde začína jedna úloha a končí druhá. Tieto záujmy sa navzájom nemiešajú. Napríklad najjednoduchší pohľad na rastlinu je, že má korene, stonku a listy. Úlohou koreňov je čerpať živiny s pôdy, úlohou stonky je transport týchto živín a úlohou listov je spracovanie slnečného žiarenia a látok dodaných stonkou.

Oddelenie častí systému podľa úloh, majúc pritom istotu, že rovnaká úloha sa nevykonáva inde, umožňuje analyzovať časť systému izolovane od ostatných častí, čo takúto činnosť zjednodušuje.

Spoločné vzory

Vlastnosť spoločných vzorov v komplexných systémoch znamená, že systémy sa skladajú často-krát z veľkého množstva primitív, no tieto primitíva sa opakujú, buď samostatne, alebo ako viacero primítiv spolu. Najjednoduchším príkladom je bunka v liste rastliny. Na to aby sme pochopili ako funguje bunka listu nemusíme skúmať všetky bunky, stačí nám jedna, pretože ostatné sú rovnaké - tj. sú vytvorené podľa rovnakého vzoru. Podobne aj vesmírne galaxie, alebo oddelenia v podniku. Vo všetkých komplexných systémoch sa dajú identifikovať vzory usporiadania prvkov, ktoré sa opakujú.

Identifikovanie týchto vzorom potom pomáha pri pochopení celého systému, pretože ak znovu narazíme na známy vzor v inom systéme, alebo v inej časti systému, môžem predpokladať, že jeho chovanie a vlastnosti budú zhodné so vzorom.

Stabilné prechodné formy

Žiaden komplexný systém nevznikol naraz. To znamená, že žiaden systém jeden deň nebol a druhý deň existoval v celej svojej kráse. Všetky komplexné systémy prechádzajú cez stabilné prechodné formy pokým sa dostanú do súčasného stavu (ktorý môže byť tiež iba prechodovou formou). Najväčšie podniky vznikajú v garáži s minimálným počtom zamestnancov a postupne rastú. Rastlina začína od malé klíčku, ktorý už od prvého dňa funguje až postupne vyrastie do celej rastliny.

V niektorých prípadoch sú prechodné formy badateľnejšie (podnik) v iných sú menej výrazne, pretože zotrvanie v jednej stabilnej forme je relatívne krátke (rastlina)

Objektovo orientovaný prístup

V súčastnosti sa objektovo orientovaný prístup považuje za najvhodnejší pre riešenie väčšiny projektov. Či už pri analyzovaní, návrhu alebo implementácii. Samozrejme nie je možné tvrdiť, že objektovo orientovaný prístup je univerzálny a že je vhodný na akúkoľvek úlohu. Sú typy problémov, na ktoré je vhodnejšie zvoliť iný prístup. Je tiež pravdepodobné, že v budúcnosti príde úspešný nástupca. Momentálne je však objektový prístup najpoužívanejší a najvhodnejší pre väčšinu úloh.

Objektovo orientovaný prístu je logickým pokračovaním klasického tradičného (štruktúrovaného) prístupu. Viacero princípov je rovnakých alebo veľmi podobných v oboch prístupoch. Pre objasnenie pojmov: *Za klasický a tradičný prístup budeme pokladať, pre zjednodušenie, štruktúrovaný prístup*.

Pri tradičnom prístupe sa modul buduje okolo jednotky (časti), ktorá vznikne funkčnou dekompozíciou. Pri objektovom prístupe sa modul buduje okolo objektu.

Objektový (ako aj klasický) prístup je možné uplatniť tak pri analýze a návrhu informačných systémov ako aj programových systémov. Rozdiel je v tom, že podľa všeobecne uznávanej definície, nemusí byť informačný systém podporovaných počítačom. Preto je možné využiť niektorý z prístupov na analýzu a návrh systému, ktorý nebude implementovaný (resp. nebude implementovaný *celý*) do podoby programového systému. Faktom však je, že pojem informačný systém sa častejšie vyskytuje pri klasických prístupoch. Objektovo orientovaný prístup viac uvažuje s programovým systémov a softvérom.

Objektovo orientovaný prístup k tvorbe informačných systémov nie je to isté ako objektovo **orientované programovanie**. Objektovo orientované programovanie vychádza z prístupu a jeho jeho časťou. Programovanie je zamerané na implementáciu. Naopak, prístup sa uplatňuje v analýze, návrhu a implementácii. Teoreticky je možné, ale nie veľmi výhodné, uplatňovať klasické (štruktúrované) prístupy v analýze a návrhu a implementáciu realizovať v objektovom jazyku.

Objektový prístup nahradil štruktúrovaný z viacerých dôvodov. Jedným z nich je napríklad to, že štruktúrovaný prístup citeľne zlyhával pri analýze a návrhu veľkých a komplexných programových a informačných systémov.

Napríklad najčastejšie používaná metóda funkčnej dekompozície zhora dole v klasickom prístupe vedie k tomu, že systém by mal mať iba jednu funkciu, ktorej dekomponovaním získame ďalšie "podfunkcie". V skutočnosti však žiaden komplexný systém nemá iba jednu funkciu, má ich viac. Funkčná dekompozícia v tomto prípade nie je nápomocná, pretože neumožňuje pohľad na systém tak aby zodpovedal tomuto faktu.

Dekompozícia

Každé skúmanie a analýza komplexného systému je podriadená dávno známemu princípu "*rozdeľuj a panuj*". Čiže celok sa snažíme rozdeliť na menšie časti, ktoré potom analyzujeme osobitne. Hovoríme, že probléme **dekomponujeme**.

Dekompozícia môže byť

1. Algoritmická dekompozícia
2. Objektovo orientovaná dekompozícia

V prípade **algoritmickej dekompozície** najprv rozložíme aktivitu na menšie a jednoduchšie. V ďalšom kroku nájdeme prvky (objekty), ktoré tieto menšie aktivity budú vykonávať.

Objektový dekompozícia najprv identifikuje prvky, ktorým priradí aktivity.

Oba prístupu sú dôležité, objektový prístup však verí, že identifikovať najprv objekty je vhodnejšie.

Aplikácia OO na kompexné systémy

Objektovo orientovaný prístup rieši spomenuté vlastnosti komplexných systémov pomocou základných princípov. Niekedy sa uvádzajú iné názvy princípov prípadne iný počet, ich podstata je však vždy rovnaká:

1. Znovupoužiteľnosť
2. Abstrakcia
3. Hierarchia

Všetky tieto princípy vychádzajú z klasického prístupu. Hlavný rozdiel je v tom, že pri objektovom prístupe je rozdiel medzi konceptom ako človek chápe komplexné systémy a tým ako ich analyzuje a navrhuje do podoby informačných systémov menší ako pri klasickom prístupe. Zjednodušene povedená, človek v reálnom svete najprv spoznáva objekty a až potom ich funkcie.

Znovupožiteľnosť

Znovupoužiteľnosť je analogická k *spoločným vzorom* v komplexných systémoch. Ale tiež k *relatívnym primitívam* a *oddeleniu záujmov*.

Znovupoužiteľnosť umožňuje aby sme rovnaký problém neriešili na viacerých miestach, jeho kopírovaním, ale aby sme ho mali na jednom mieste, v jednom prvku. Takýto prvkov potom môžeme používať v rámci jednej úlohy na viacerých miestach. Ale môžeme ho tiež použiť na riešenie úplne inej úlohy v inom projekte.

Napríklad vytvoríme prvok systému, pomocou ktorého si používateľ môže prepnú jazyk zobrazenia na obrazovke programu. Tento prvkom môžeme použiť znovu na iných obrazovkách toho istého programu, ale aj na obrazovke iných programov, ktoré v budúcnosti vytvoríme.

Nato aby sme takýto prvok mohli vytvoriť musíme ho vhodne určiť jeho hranice ako *primitíva* a oddeliť jeho záujmy od ostatných častí systému. Pri nevhodne navrhnutých hraniciach a záujmoch sa môže stať, že prvok nebude znovupoužiteľný, pretože bude závislý na systéme (obrazovke), na ktorej bolo pôvodne vytvorený.

Znovupoužiteľnosť vytvorených prvkov nemusí byť iba v rámci jedného tímu, jednej firmy. Môže presahovať hranice jednej spoločnosti a znovupoužiteľné prvky sa môžu prenášať medzi firmami.

Zo znovupožiteľnosťou úzko súvisia dva syndrómy:

1. *NIH* (Not Invented Hier) tj. *Nevynájdené tu* . Tento syndróm sa prejavuje tým, že ľudia nie sú ochotní znovupoužívať prvky, ktoré nevytvorili sami resp. nevytvorila ich spoločnosť, v ktorej pracujú.
2. HIN (Habit Inhibiting Novelty) tj. *Vlasnosť zakazujúca novinky* [1]. Je opačný syndróm. Jeho prejavom je, že ľudia nie sú ochotní používať nič iné iba jeden konkrétny prvok, aj keď v skutočnosti by mohli znovupoužiť aj iné.

Interakcia

Zavedením znovupoužiteľnosti sa dostávame k interakcii. Interakcia medzi prvkom a jeho okolím. Pre “jednorázových* prvkoch nie je interakcia taká podstatná. Napríklad aj preto, že ak prvok nie je znovupoužiteľný nemá jasne definované hranice a preto nevieme, v ktorom momente už hovoríme o interakcii dvoch prvkov a kedy je aktivita stále v samostatnom prvku.

Pri interakcii navzájom komunikujú **klient** prvku a znovupoužiteľný prvok. Túto interakciu budeme nazývať **poskytovaním služby**. Iné označenie v objektovom prístupe je aj *posielanie správ*.

Správne definovanie služby je veľmi podstatné. Pri volaní služby prvku nevieme (nemáme vedieť) či prvkov našu požiadavku rieši sám, alebo jej časť (prípadne celú) posiela na spracovanie inému prvku. Zo služby nám teda má byť jasné iba to **čo** prvok robí ale **nie ako** to robí.

Abstrakcia

Ľudia majú výbornú vlasnosť ako chápať zložité problémy. Jednoducho abstrahujú z problému tie časti, ktoré pre jeho pochopenie nie sú dôležité. Abstrakcia súvisí s *relatívnymi primitívami* komplexných systémov, ale tiež s *hierarchickou štruktúrou*.

Abstrakcia je procesom odoberania detailov (slovesom) ale aj výsledkom (podstatné meno) tohto procesu.

To aké detaily budú odobraté a aké služby bude prvok ponúkať závisí od uhla pohľadu. Môžeme hovoriť o uhloch pohľadu jedného pozorovateľa a uhloch pohľadu rôznych pozorovateľov.

Napríklad pohľad na pomaranč (prvok systému) má iný maliar, ktorý vidí jeho farbu a tvar ako prípadnú súčasť svojho diela. Inak ho vidí farmár, ktorý ho chce predať a inak ho vidí smädný človek, ktorý ho vidím ako spôsob zasýtenia smädu.

Pri rôznych pohľadoch toho istého pozorovateľa hovoríme o **úrovniach abstrakcie**. Smädný človek síce vidí pomaranč ako prvkom poskytujúci službu zasýtenia smädu (čiže abstrahoval od všetkých ostatných vlastností pomaranča) no pri bližšom pohľade ho začne zaujímať iba dužina. Z pôvodnej abstrakcie pomaranč - šťava sa stala na nižšej úrovni abstrakcie iba dužina - šťava. Vždy je podstatné zostávať pri analýze prvkov systému na tej istej úrovni abstrakcie a na tejto úrovni zohľadniť pohľady všetkých relevantných pozorovateľov. Je dôležité si uvedomiť, že hľadanie vhodnej abstrakcie je procesom a môže sa stať, že aj keď sa nám zdá, že sme našli

vhodnú abstrakciu problému, tak prihladnutím na pohľad iného pozorovateľa zistíme, že abstrakcia nevyhovuje.

Ešte dôležitejšie ako zostať vždy na rovnakej úrovni abstrakcie je zostať na správnej strane pohľadu. Na každú službu poskytovanú prvkom sa môžeme dívať **zvnútra** a **zvonku**. Pri pohľade zvonku vidíme **čo** má prvok skrz službu robiť, ale nevidíme **ako**. Pri pohľade zvnútra vidíme nie len čo ma služba robiť ale aj **ako**. Častým problémom nesprávnej analýzy je, že miešajú tieto dva pohľady. Pričom pre pochopenie komplexného systému je potrebné najprv pochopiť čo má robiť, ako to má robiť je až ďalším krokom.

Napríklad (viď tiež [2]) Máme študenta a zamestanca, ktorí sú súčasťou školského informačného systému, obaja majú rodné číslo. To, že majú rodné číslo je to **čo** vedia poskytnúť. Ak by sme skĺzli k tomu, že by sme hneď na začiatku povedali aj **ako**, čiže napríklad, že rodné číslo je atribútom študenta aj zamestnanca, tak by sme si situáciu skomplikovali pre ďalšiu analýzu. Totiž ak neskôr zistíme, že študent môže byť zamestnancom mali by sme informáciu o rodnom čísle dva krát. Aj v zamestnancovi aj v študentovi. Čiže rodné číslo nie je atribútom ale službou, ktorú tieto dva prvky vedia poskytnúť.

Podobne ako pri pohľade zvonku nevieme *ako* je služba vykonaná, neviem ani pri pohľade zvnútra pre koho je služba vykonaná. Pre službu platí **anonymita klienta** [2].

Literatúra

2. Prednáška

Modelovanie

Nato aby sme mohli správne vytvoriť programový systém, je potrebné aby sme čo najlepšie poznali komplexný systém, v ktorom bude programový systém automatizovať vybrané úlohy.

Vďaka znalostiam o vlastnostiach komplexných systémov môžeme lepšie navrhnúť programový systém.

Prvým krokom bude vždy analýza systému ako takého. Druhým krokom bude výber vlastností, ktoré chceme zahrnúť do programového systému (vytvorenie programového systému chápeme ako našu prioritnú úlohu).

V oboch krokoch môžeme pre lepšie pochopenie vytvoriť **model**. ako zjednodušený pohľad na realitu. Bude však vždy iba jedným z pohľadov. Napríklad, na nemocničný systém (tj. systém fungovania nemocnice všeobecne - nemyslíme softvér) môže existovať viacero pohľadov. Jeden pohľad zaujíma ekonomická podstata systému, iný zase technické fungovanie prístrojov a zariadení a ich vzájomné prepojenie apod. V reálnom svete tieto dva pohľady súvisia, ale v modely sa zohľadní iba jeden z nich.

Model môže byť zachytený:

1. V textovej forme
2. V grafickej forme
3. V 3D forme

3D forma je pre softvér najmenej používaná (ak vôbec) trojrozmerný model softvéru by nepriniesol žiaden efekt, pretože samotný softvér je nehmotný.

Najčastejšie používanými sú modely zachytené pomocou textovej a grafickej formy. Obe majú svoje výhody, nevýhody a limity.

Textová forma má nevýhodu vo svojej neprehľadnosti. Pochopiť a predstaviť si model zapísaný v textovej forme trvá dlhšie. Vyplýva to z vlastností písaného textu. Napríklad popis postupu ako si zaviazať šnúrky do topánok bude zrejme ľahšie pochopiteľnejší ak bude vo forme obrázku, nie textu.

Grafická forma má teda výhodu rýchlejšieho pochopenia a tiež v tom, že do istej miery kompenzuje to, že softvér je nehmotný a teda ťažko predstaviteľný. Vďaka grafickej forme si vieme softvér predstaviť o niečo "hmotnejšie".

V objektovo orientovanom prístupe sa prakticky štandardom grafickej formy modelovanie stalo UML. UML je výrazne orientované na analýzu a návrh programového systému. V menšej miere sa používa na analýzu komplexných systémov.

Úrovne modelov

Tak ako môžu existovať rôzne modely budúceho domu, existujú aj rôzne modely programového systému. Treba správne chápať aký je rozdiel medzi tým, že existujú rôzne modely reality a rôzne modely programového systému.

Rôzne modely reality (príklad s nemocnicou) znamenajú, že z reálneho komplexného systému si vyberieme iba tú časť, ktorá je pre nás podstatná. Podstatná je tá časť, ktorá bude zabezpečená programovým systémom.

Rôzne modely programového systému vychádzajú z predchádzajúceho kroku. Vybrali sme si jeden z modelov reality a v rámci tohto modelu vytvárame ďalšie čiastkové modely, z ktorých každý predstavuje iný pohľad. Jeden model sa môže zameriavať viac na dynamiku iný zase na statický pohľad a ďalší môže modelovať chovanie systému v čase. Vždy však pôjde o modely, ktoré modelujú jednú vybranú časť reálneho komplexného systému.

Úrovne abstrakcie modelov

Hlavné kategórie modelov programových systémov sú závislé od úrovne abstrakcie modelu. Čím je úroveň nižšia, tým je model bližšie k skutočnej implementácii (tj. programu napísaného v programovacom jazyku) a tým je abstrakcia menšia. Na najvyššej úrovni abstrakcie[1] nie je zrejmé, aký programovací jazyk bude použitý, aká platforma apod. Na najvyššej úrovni abstrakcie je modle pochopiteľný aj pre človeka, ktorý nemá žiadne znalosti z oblasti softvérového inžinierstva.

Úrovne modelov:

1. Analytické modely
2. Modely dizajnu
3. Implementačné modely

Analytické modely modelujú iba to **čo** robí systém, nikdy nie **ako**. Tieto modely sú na takej úrovni abstrakcie, ktorú by mal pochopiť aj človek bez znalostí programovania a softvérového modelovania. V analytických modeloch sa vytvárajú **doménové názvy**. Sú to pojmy, ktoré sú špecifické pre danú oblasť komplexného systému, pojmy ktoré používajú ľudia pracujúci v tejto oblasti.

Modely dizajnu modelujú návrh programového systému. Už v náznakoch ukazujú z akých častí bude programový systém zložený a ako budu tieto časti organizované do rôznych modulov. Slovo dizajn nie je použité vo význame grafického rozhrania.

Implementačné modely sú podkladom pre konkrétny programovací jazyk, konkrétnu platformu, konkrétny hardvér. Prvky v týchto modeloch je možné nájsť priamo v programe. Napríklad trieda v implementačnom modely predstavujú skutočnú triedu v programovacom jazyku.

Iným označením úrovne abstrakcie **perspektíva**. Hovoríme, že na model sa dívame z nižšej perspektívy alebo vyššej (používajú sa rôzne označenia: Vtáčia perspektíva, perspektíva na úrovni mora, rybia perspektíva apod.)

Najvyšší pohľad sa tiež označuje ako **konceptuálny** a najnižší ako **softvérový** pohľad.

UML

Jazyk UML sa stal prakticky štandardom pri modelovaní programových systémov. Jeho výhodou je, že sa nesnaží pokryť úplne všetky oblasti, ktoré súvisia s analýzou komplexných systémov. Zameriava sa primárne na softvérový aspekt. Na druhej strane však umožňuje rôzne formy rozšírenia, tak aby modely vyhovovali konkrétnej potrebe.

UML je nie je programovacím jazykom. Je jazykom modelovacím. UML nebolo navrhnuté s cieľom napísať program v UML, UML skompilovať a získať tak funkčný programový systém.

V zásade je možné UML použiť týmito spôsobmi [2]

1. UML ako náčrtok
2. UML ako návrh
3. UML ako programovací jazyk

UML ako náčrtok predstavuje najjednoduchšiu formu požitia. Diagramy v UML slúžia na pochopenie problému pri diskusii v analýze. Častokrát sú majú jendorázové použitie. Ak sa problém pochopí diagram už nemá svoje opodstatnenie. Vo veľmi malej miere slúžia náčrtky ako poklady pre ďalšie modely alebo ako oficiálne časti dokumentácie programového systému. Najčastejšou spôsobom zakreslenia je tabuľa a fixa, prípadne papier a pero.

UML ako návrh je forma, pomocou ktorej sa detailne navrhne budúcu programový systém. Dá sa prirovnať k návrhu domu. Je vypracovaný detailne. Prakticky v ňom chýbajú iba niektoré implementačné detaily. Býva súčasťou dokumentácie a diagramy sa aktualizujú pri zmene požiadaviek. Najčastejším spôsobom zakreslenie ja vhodný softvérový nástroj (CASE).

UML ako programovací jazyk. Aj keď bolo povedané, že UML nebol navrhnutý ako programovací jazyk, existujú technológie, ktoré dokážu UML diagram “skompilovať” do programovacieho jazyka. UML musí byť zakreslené pomocou vhodného softvérového nástroja, ktorý ma podporu na generovanie zdrojového kódu.

Diagramy UML

Diagramy v UML predstavujú grafické modely programového systému. UML pozná modely iba v podobe diagramov nepredpisuje a neformalizuje žiadnu textovú podobu modelu (s jednou drobnou výnimkou v prípade Požívateľských scenárov).

UML diagramy sú zatriedené do dvoch základných skupín, ktoré odzrkadľujú dve základné vlastnosti systémov dynamiku a štruktúru.

UML má

1. Diagramy štruktúr (6 diagramov)
2. Diagramy chovania (7 diagramov)

Diagramy chovania majú v sebe jednu podskupinu

1. Diagramy interakcií (4 diagramy)

S novými verziami UML pribúdajú nové diagramy a niektoré (väčšinou tie, ktoré sa ukážu ako málo používané) sa vynechávajú. Žiaden model programového systému v jazyku UML by nemal obsahovať všetky diagramy. Vždy iba tie, ktoré sa vyberú ako vhodné. Pravidlo *menej je niekedy viac* platí aj tu.

Medzi štandardné diagramy, ktoré sa používajú takmer vždy patria:

1. Diagramy štruktúr
    1. Diagram tried
    2. Komponentový diagram
    3. Diagram balíkov
2. Diagram chovania
    1. Diagram používateľských scenárov
    2. Aktivity diagram
    3. Diagramy interakcií
        1. Sekvenčný diagram
        2. Komunikačný diagram

Diagramy UML môžu byť použité na rôznych úrovniach abstrakcie modelov. Niektoré sú určené pre analytické modely iné pre implementačné. Niektoré je možné použiť vo viacerých.

Požiadavky

Ešte predtým ako dôjdeme k vytvoreniu prvého modelu z konceptuálneho pohľadu - analytickému modelu, potrebujem poznať požiadavky.

Analyzovať komplexný systém môžeme aj bez existencie požiadaviek. V zásade však nejaké požiadavky máme, aby sme sa mohli zamerať iba na to podstatné.

Pri analýze programového systému sa však už bez požiadaviek nezaobídeme. Nemôžeme vytvoriť model bez toho aby sme vedeli, čo modelovaný programový systém má robiť - aké požiadavky má spĺňať.

Požiadavky a model existujúceho komplexného systému sú vstupom k vytvoreniu analytického modelu programového systému.

UML nemá špeciálny diagram na požiadavky. Niektoré softvérové nástroje poskytujú špeciálne diagramy, v ktorých je možné požiadavky zachytiť.

Základné kategórie požiadaviek sa uvádzajú pod skratkou **FURPS** [3]

1. Funkčné (Functional)
2. Použiteľnosti (Usability)
3. Spoľahlivosť (Reliability)
4. Výkonnosť (Performance)
5. Obslužnosť (Supportability)

Diagram používateľských scenárov

Diagram používateľských scenárov (use case diagram) zachytáva **funkčné** požiadavky na programový systém z pohľadu používateľa.

Jeden používateľský scenár môže plniť jednu funkčnú požiadavku, ale môže ich plniť aj viac. Platí to aj opačne. Jedna funkčná požiadavka môže byť realizovaná viacerými prípadmi použitia. Vždy ide o slovesné pomenovanie.

Diagram scenárov je trochu netypický oproti iným diagramom. Jeho odlišnosť je v tom, že napriek tomu, že UML štandardizuje iba diagramy, pri scenároch sa vždy odporúča vytvoriť textovú formu scenára. Niektoré metodiky dokonca kladú dôraz iba na textovú formu a diagram vynechávajú. Samotné UML textovú formu neštandardizuje a textový formát si môže rôzny.

Scenár špecifikuje postupnosť interakcií medzi používateľom a systémom. V UML diagrame sa zobrazujú iba názvy scenárov a ich vzájomné interakcie. Na popis krokov scenára slúži textová forma. Jeden scenár by mal predstavovať jeden zámer používateľa.

Aktor je používateľ systému. Môže ísť o osobu, ale aj o iný systém. Niekdy môže byť aktorom aj čas. Aktor je z pohľadu systému vždy externý prvok a nie je súčasťou systému.

Hľadanie scenárov Na identifikovanie scenárov existujú rôzne postupy. Napríklad [4]

1. Aktérový prístup
2. Procesný prístup

Aktérový prístup scenára z pohľadu aktérov. V prvom kroku identifikujeme aktérov a následne sa hľadá akým spôsobom potrebujú systém používať - aké funkčné požiadavky naň majú.

Procesný prístup hľadá scenáre na základe existujúcich procesov prebiehajúcich v komplexnom systéme. Okrem toho, že umožňuje nájsť prípady použitia, dáva tiež priestor na popísanie chodu procesu, ktorého sa programový systém stane súčasťou. Hľadá sa teda miesto (a činnosť), ktorá pomôže celému procesu.

Pri oboch prístupoch je potrebná správna granularita. Ak by sme použili príliš podrobné členenie, omylom by sme za scenár mohli označiť niečo, čo už je v skutočnosti iba jedným z krokov v scenári.

Dobrou skúškou správnosti je tzv. **Boss test** [3]. Správny scenár (prípad použitia) je taký, pri ktorom by bol náš šéf spokojný ak aby sme ho robili celý deň.

Napríklad scenár *Prihlásiť sa* nie je správny prípadom použitia pretože náš šéf by nebol spokojný ak by sme povedali, že sme sa prihlasovali celý deň. Naproti tomu prípad použitia *Vložiť objednávku* je vhodným scenárom. Šéf bude spokojný ak celý deň budeme vkladať objednávky.

Správny prípad použitia musí tiež vychádzať z cieľov, ktoré má aktor. Nemalo by sa totiž stať, že nájdeme prípad použitia, o ktorom presne vieme ako má prebiehať ale neviem *“načo to komu vlastne je”*

Už sme naznačili, že Diagram používateľských scenárov nemá predpísanú žiadnu textovú formu, tak ako všetky ostatné diagramy v UML. No na rozdiel od ostatných sa text scenára vyslovene odporúča.

Text scenára predstavuje jednotlivé kroky ako scenár prípadu použitia prebieha. Kroky môžu byť:

1. Zapísané v texte. Každý krok scenára má svoje poradové číslo. Čísla umožňujú aj členenie scenára nad časti. Napríklad ak nejaká časť scenára nastane iba pri splnení podmienok.
2. Zapísané v *aktivity diagrame*.
3. Teoreticky môžeme skončiť tak, že prípadu použitia nebudeme mať žiaden scenár. Nie je to odporúčané riešenie, pretože ak nájdeme prípad použitia, ktorý nemusí mať scenár, pretože je jednododuchý je pravdepodobné, že by neprešiel *boss testom*.

Štruktúra, v ktorej majú byť zaznačené kroky scenára nie je nijako štandardizovaná. Najprepracovanejšie odporúčania, štruktúry scenára má metodika **RUP(UP)**.

Vo všeobecnosti sa odporúča aby scenár mal nasledujúce časti. Nemusí ich mať všetky.

1. Hlavný scenár.
2. Vedľajší scenár. Scenár, ktorý prebieha ak je splnená nejaká podmienka
3. Hlavný aktor. Určenie hlavného aktora pre scenár. Používateľský scenár je vždy vyvolaný aktorom.
4. Vedľajší aktori. Zapíšu sa iní aktori, ktorí ovplyvňujú scenár, alebo ktorí sú scenárom ovplivnení.
5. Podmienky, ktoré musia platiť pred spustením scenára

6. Podmienky, ktoré musia platiť po dokončení scenára
7. Body rozšírania. Sú to miesta v scenári, v ktorých môže dôjsť k použitiu iného prípadu použitia. Hlavný scenár nevie, o ktorý iný prípad použitia ide.
8. Body vloženia. Sú to miesta, v ktorých vždy dôjde k použitiu časti scenára, ktorý sa opakuje aj v iných prípadoch použitia. Napríklad *platba kartou* je časť scenára, ktorá môže byť použitá aj v iných scenároch. Hlavný scenár vie, ktorý prípad použitia bude v mieste vložený.

Text scenára by mal používať niektoré osvedčené techniky [4]

1. Nepoužívať na to isté označenie synonymá. Ak raz v jednom kroku scenára použijeme jedno pomenovanie nesmieme v iných scenároch používať jeho synonymá. Napríklad kniha - publikácia apod.
2. Používať ustálené slovné spojenia alebo vetné konštrukcie. V scenároch používať stále tie isté vety, v ktorých sa mení iba jedna časť.
3. Vyhnúť sa používaniu podstatného mena systém. V tomto diskutabilnom odporúčaní ide o to, že používanie slova systém môže viesť k zakomponovaniu ďalšieho aktora. Namiesto slova systém radšej používať zvratné zámena.

Aktor

Každý prípad použitia je vyvolaný aktorom. Jeden aktor môže vyvolať jeden alebo viac prípadov. Zároveň jeden prípad môže byť vyvolaný jedným alebo viacerými aktormi.

Aktorom môže byť používateľ (človek) alebo iný systém. Lepším vyjadrením pre aktora je **rola**. Rolu však treba chápať na vyššej úrovni abstrakcie. Každý systém môže používať rôzne úrovne oprávnenia. Napríklad manažér bude mať v systéme viac práv ako zamestnanec. Rola v tomto význame je však o niečo iná aj keď rozdiel je niekedy nevýrazný.

Každý aktor by mal vyvolaním prípadu použitia vytvárať **rozhranie systému**. Preto zadefinovanie aktorov *zamestnanec* a *manažér* nemusí byť správne, pokiaľ jediný rozdiel je v tom, že manažér môže na rozdiel od zamestnanca niečo zmeniť, tak z pohľadu systému je to ten istý aktor. Rozdelenie na dvoch aktorov by malo význam skôr v prípade zamestnanec a administrátor. Administrátor bude zrejme systém používať úplne odlišným spôsobom ako zamestnanec.

Ako pomôcka na nájdenie hlavných aktorov môžu slúžiť nasledujúce otázky [3]:

1. Kto zapína a vypína systém?
2. Kto pridáva a odoberá používateľov a ich práva?
3. Je aktorom čas, pretože systém niečo vykonáva po uplynutí času?
4. Kto je notifikovaný ak nastane chyba v systéme?
5. Je nejaký iný programový systém, ktorý volá náš?

3.Prednáška

Vzťahy v medzi scenármi

Prípady použitia môžu byť vo vzájomnom vzťahu. Tento vzťah znázorňujeme v diagrame, ale je možné ho zachytiť aj v scenároch prípadov použitia.

Základné vzťahy medzi prípadmi použitia sú:

1. Generalizácia
2. Vzťah rozšírenie «extends»
3. Vzťah obsiahnutia «include»

Treba poznamenať, že posledné dva vzťahy sa často zamieňajú a ich význam býva nesprávne interpretovaný. Ich používanie by malo patrí viac do pokročilých techník používania UML. Chybné použitie týchto vzťahov je skúseným analytikom odhalené hneď, pričom väčšou chybou ako ich nesprávne použitie je nesprávne identifikovanie a pomenovanie prípadu použitia.

Detailnejší popis rozdielov medzi vzťahom rozšírenia a obsiahnutia je možne nájsť v [1] a [2]

Rozdiely medzi oboma sú zhrnuté v nasledujúcej tabuľke [3]

Rozdiely vo vzťahoch obsiahnutia a vloženia

Rozdiel	**Vzťah obsiahnutia**	**Vzťah rozšírenia**
Je scenár vo vzťahu nepovinný?	Nie	Áno
Je základný scenár kompletný bez druhého scenára?	Nie	Áno
Je vykonanie druhého scenára niečim podmienené?	Nie	Áno
Mení druhý scenár priebeh hlavného scenára?	Nie	Áno

Vzťah **generalizácie** je obdobou dedenia medzi objektmi. Využíva sa ak chcem znázorniť, že scenár je skoro rovnaký vo viacerých prípadoch a rozdiel je natoľko malý, že nie je potrebné vytvárať iný scenár.

Chyby v diagrame prípadov

Medzi najčastejšie chyby v diagrame prípadov použitia resp. v scenároch použitia patria:

1. Scenár obsahuje dizajnové informácie (napríklad, ikonka, tlačidlo, obrazovka apod)
2. Prípady použitia sú funkčne dekomponované (štýlom podobným štruktúrovanej analýze). Dekompozícia prípadov použitia musí mať stále na pamäti, že prípad použitia je funkčná požiadavka z pohľadu používateľa. Nesprávna dekompozícia môže totiž prejsť do návrhu *Ako* bude scenár prebiehať.
3. Príliš veľa aktorov kvôli nesprávnemu pochopeniu role. Nie každá rola v systéme vyžaduje samostatného aktora. Ak má niektorá rola iné oprávnenia na niektorý z krokov v scenári, nemá význam aby bol scenár rozdelený na dva so znázornením dvoch aktorov.

Diagram tried

Diagram tried patrí zrejme medzi najčastejšie používané diagramy UML. Využíva sa na všetkých úrovniach pohľadu na programový systém:

1. Analytické diagramy tried
2. Konceptuálne diagramy tried
3. Diagramy návrhu (softvérový pohľad)

Častou otázkou býva, čo býva v analýze skôr. Use case diagram alebo diagram tried (prípadne iný diagram)? Odpoveď je jednoduchá. Vytvárajú sa obe naraz. V literatúre nemôžu byť uvedené diagramy naraz. Vždy musí jedna kapitola predchádzať inej. V praktickom živote môže analytik pracovať naraz na oboch diagramoch.

Diagram tried môže analytik použiť prakticky aj bez poznania požiadaviek na systém. Vzťahy medzi niektorými časťami systému sú natoľko významné, že ich existencia sa prenesie aj do

programového systému. Následné, definovaním používateľských scenárov, môže dôjsť spresneniu tried, prípadne doplneniu. Ak sa zistí, že niektorá časť reálneho systému nebude obsiahnutá v programovom systéme, nepotrebné triedy sa môžu z diagramov odstrániť.

Doménový model

Pri vytváraní analytických diagramov tried vytvárame zároveň **doménové** modely. Doménové modely slúžia primárne na pochopenie základných vzťahov a pojmov, ktoré sa v danej oblasti (doméne) používajú. Doménový model môže zachytávať celý systém, alebo sa môže upriamiť iba na pojmy a vzťahy, ktoré budú súčasťou programového systému. Pre lepšie pochopenie, je vhodné vytvoriť model bez ohľadu na to, čo bude súčasťou programového systému.

Analytické modely by mali byť v čo najväčšej miere pochopiteľné pre zákazníka. Ich prvoradou úlohou je aby analytik a odborník z doménovej oblasti začali používať rovnaké pojmy pre rovnaké prvky a rovnako tiež chápali vzťahom medzi prvkami.

Nájdenie tried

Nájdenie tried pre analytický model (doménový model) je na prvý pohľad jednoduché. Zjednodušene povedané, z opisu fungovania systému vyberáme podstatné mená (doménové pojmy), ktoré budú predstavovať triedy.

Tento spôsob je jednoduchý samozrejme iba na prvý pohľad a pri hlbšom pohľade dochádza ku komplikáciam.

Pri vnímaní okolitého sveta a teda aj pri opise fungovania systému, identifikujeme konkrétne inštancie objektov. Čiže okolitý svet vnímame cez inštancie, nie cez triedy. Z tohto pramení riziko, že za triedu (tj. niečo z čoho vznikajú inštancie objektov) budeme omylom považovať inštanciu.

Vyhľadávanie triedy cez podstatné mená, by teda malo prebiehať akoby dvojkolovo. Najprv identifikujeme triedy z podstatných mien, ale dívame sa na ne ako na inštancie, v druhom kole sa snažíme prísť na to, či niektoré z inštancií nie sú inštanciou tej istej triedy, len s inými hodnotami atribútov.

Na každý systém sa môžeme pozrieť ako na spojenie dvoch štruktúr:

1. Štruktúra tried, v ktorej je každý prvok nejakým typom. Je to statický pohľad.
2. Štruktúra objektov, v ktorej je každý prvok súčasťou iného prvku. Je to dynamický pohľad.

Byť typom nevyžaduje uvažovať nad inštanciou. Napríklad lekár je typom prvku v systéme. Rovnako tak je typom nemocničné oddelenie. Lekár pracuje na oddelení. Táto štruktúra je prakticky nemenná. Ak však chcem uvažovať nad tým, aký lekár pracuje na akom oddelení, potrebujem uvažovať nad konrétnymi inštanciami lekára a oddelenia. Lekár môže pracovať (byť vo vzťahu) postupne v čase na viacerých oddeleniach. Štruktúra systému ostane zachovaná aj napriek tomu, ak by všetci lekári zmenili svoje oddelenia.

Nájdenie vhodnej klasifikácia (štruktúry tried) je kľúčovou úlohou analýzy. Bude typom *Oddelenie* a Anestezilogicko-resuscitačné oddelenie (ARO) konkrétnou inštanciou, alebo bude samotné ARO typom?

Vlastnosti triedy

Medzi vlastnosti triedy patria:

1. Atribúty
2. Asociácie

3. Operácie [4]

Každý atribút triedy je zároveň asociáciou. To či vzťah znázorníme ako atribút alebo ako asociáciu závisí od granularity pohľadu a “veľkosti” tried a tiež od doménového významu. Napríklad dátum (ako triedu) budeme znázorňovať ako atribút, znázornenie, že trieda je v asociácii sa dátumu je formálne správne tiež, ale nepridáva to prehľadnosti diagramu.

Opačný vzťah - každá asociácia je atribútom neplatí. V analytickom modely tried znázorníme, že dve triedy sú vo vzájomnom vzťahu, majú medzi sebou asociáciu, akým spôsobom sa táto asociácia zmení závisí od ďalšieho modelovania. Môžeme prísť na to, že asociácia sa zmení na vzťah rodič - potomok.

Atribúty

Pri atribútoch uvádzame:

1. Viditeľnosť
2. Názov
3. Typ
4. Násobnosť
5. Prednastavenú hodnotu
6. Vlastnosť

Pri analytickom modely je podstatný iba názov, ostatné časti je možné vynechať.

Asociácia

Pri asociácii uvádzame:

1. Násobnosť
2. Smer
3. Pomenovanie koncov
4. Názov

V analytickom modely tried nie je povinný žiaden z týchto popisov, ale je dobré uviesť násobnosť asociácie.

Smer asociácie určuje, či o vzťahu “vedia” obe triedy, alebo iba jedna z nich. Pomenovanie koncov je prakticky pomenovaním atribútu, ktorý bude predstavovať vzťah v triede Názov asociácie sa uvádza predovšetkým v prípadoch keď z analytického pohľadu je potrebné vedieť o aký vzťah ide. Napríklad to nie je zrejmé z názvu tried, alebo triedy môžu byť navzájom v rôznych vzťahoch. Napríklad osoba a auto môžu byť vo vzťahu *osoba je majiteľom auta* alebo *osoba je šoférom auta*.

1. Úvod do UML

Komunikačným prostriedkom informačnej komunity sa postupom času stala angličtina. Ak chceme vystaviť nejakú informáciu tak, aby bola zrozumiteľná aj mimo hranice SR, použijeme angličtinu. Podobne prebieha vývoj aj v komunite softwarových inžinierov. Behom vývoja metodík softwarového inžinierstva bolo navrhnutých množstvo rôznych viac-menej formálnych jazykov a prezentačných techník, ktoré slúžia pre popis softwarových produktov od konceptuálneho modelu až po jeho implementáciu. Rôznorodosť zápisov prináša radosť tým, ktorí sa neradi viažu na určitý jazyk či grafickú techniku, menej radosti však robí tým, ktorí musia takéto zápisy po niekom študovať a spracovávať. Nové generácie metodík softwarového inžinierstva sa snažia zjednotiť užitočné vlastnosti rôznych metodík a integrovať ich do nejakej spoločnej sady. Jedným z najďalej prepracovaných prístupov je tzv. unifikovaný modelovací jazyk UML (Unified Modeling Language) – kandidát na akési "esperanto" moderného softwarového inžinierstva.

1.1 Čo je UML

UML značí skratku pre Unified Modeling Language, čo by sa dalo preložiť ako „Unifikovaný modelovací jazyk“. Tento názov už sám o sebe napovedá, že UML patrí medzi jazyky, teda má svoju vlastnú syntax. Inak povedané, **UML je možné chápať ako dohodu nad vyjadrovacími prostriedkami pre objektové modelovanie**.

UML sa však používa nielen pre modelovanie v oblasti tvorby softwaru. Pretože základným poňatím UML je objektovo orientovaný prístup, je možné tento modelovací jazyk použiť všade tam, kde je možné aplikovať pohľad na problematiku, čo nemusí byť iba tvorba SW. Podotknime, že takýchto oblastí možných použití UML môže byť veľa. Ako príklad môžeme uviesť použitie UML pri modelovaní procesu v podnikoch, v bankách, úradoch apod., tj. v problémovej oblasti logistiky firiem a organizácií. Pri popise a potom optimalizácii chodu podniku a inštitúcie je možné použiť objektovo orientovaný prístup a teda následne logicky je možné použiť pre tvorbu modelu týchto podnikov a inštitúcií jazyk UML. Objekty tu však už nevyjadrujú nejaké časti informačného systému, ale vyjadrujú priamo nejaké objekty v podniku resp. inštitúcii (objekty „oddelenie“, „pracovník“, „vedúci“, „správa“, „obežník“ atď.). Pomocou UML sa modelujú nielen existujúce vzťahy v podniku, ale taktiež vzťahy v podniku

zatiaľ neexistujúce, teda vzťahy možné a žiadúce, čo je synonymum pre optimalizáciu podniku pomocou syntaxe UML.

Avšak hlavným cieľom zvládnutia UML je naučiť sa modelovať informačný systém napísaný objektovo štandardnými prostriedkami, čo má za následok, že pokiaľ UML použijeme, potom našemu vyjadreniu modelu porozumie každý, kto pozná UML.

Charakteristika UML

- Grafický jazyk pre **vizualizáciu**, **špecifikáciu**, **navrhovanie** a **dokumentáciu** programových systémov. UML ponúka štandardný spôsob zápisu ako návrhu systému vrátane konceptuálnych prvkov ako sú business procesy a systémové funkcie, tak konkrétnych prvkov ako sú príkazy programovacieho jazyka, databázové schémy a znovupoužiteľné programové komponenty.

- UML podporuje **objektovo orientovaný** prístup k analýze, návrhu a popisu programových systémov. UML neobsahuje spôsob ako sa má používať, ani neobsahuje metodiku ako analyzovať, špecifikovať či navrhovať programové systémy.

- Jazyk UML sa snaží o unifikáciu rôznych domén

- Ponúka **vizuálnu syntax** pre modelovanie počas celého životného cyklu softwarového projektu

- Je navrhnutý pre **modelovanie čohokoľvek**

- Je **nezávislý** na akomkoľvek programovacom jazyku a na akejkoľvek platforme

- Môže podporovať aj iné osnovy procesov než Unifikovaný proces – UP (Unified process)

- Snaží sa mať malú množinu interných pojmov
- 1.2 Čo nie je UML
- **Čo nie je predmetom UML**
- Ako bolo povedané, UML poskytuje jeden štandardný unifikovaný konzistentný modelovací jazyk pre špecifikáciu, vizualizáciu, konštrukciu a dokumentáciu SW produktov v rôznych úrovniach abstrakcie ich vývoja. Existujú oblasti, ktoré zo zásady nie sú predmetom UML. V tejto stati je vypísaný ich zoznam aj s vysvetlením.
-
- **Programovacie jazyky a UML**
- UML nie je zavedený ako vizuálny programovací jazyk (angl. visual programming language). UML je jazykom pre vytváranie modelov, z ktorých je možné prechodom do ďalších viac implementačných úrovní abstrakcie v konečnom dôsledku programový kód

vytvárať. V žiadnom prípade nie je UML zamerané na nejaký programovací jazyk a dokonca nie je ani zamerané na vizuálne programovanie v nejakej všeobecnejšej rovine.

-
- **Nástroje**
- Je zrejmé, že zavedenie nejakého ľubovolného štandardu má svoj vplyv tiež na nástroje, ktorých sa tieto štandardy týkajú. Zavedenie UML má bezprestredný vplyv na funkčnosť CASE nástrojov pre modelovanie v objektovo orientovanom prostredí. Na druhej strane, UML si v žiadnom prípade nekladie za cieľ zamerať sa na nejaký nástroj resp. na špecifikáciu jeho vnútornej funkčnosti, jeho rozhranie, spôsob ukladania dát a pod.
- Vzťah UML ku CASE nástrojom je taký, že UML iba definuje štandardnú sémantiku modelovania, ktorú by mal daný nástroj dodržať, pokiaľ má spĺňať požiadavky UML (a nič viac).
-
- **Metodika**
- UML ako jazyk v žiadnom prípade nie je metodikou resp. návodom pre tvorbu SW. V syntaxi UML teda nie sú uvádzané postupy ako SW tvoriť, alebo zásady ako riadiť projekty, ako zavádzať metodiky vo firme. Pretože však táto oblasť (oblasť riadenia projektov) je veľmi dôležitá, tvorcovia UML doporučujú určité zásady ako doplnok k UML. Pritom sa však zdôrazňuje, že UML je iba jazykom a teda iba dohodou nad syntaxou zápisu modelov a „nič viac“.

1.3 Ciele UML

Základné ciele UML je možné zhrnúť do týchto bodov:

- Poskytuje užívateľom už **hotový, zmysluplný a štandardný modelovací jazyk**, pomocou ktorého je možné nielen modely vyvíjať, ale taktiež ich vzájomne predávať ako medzi jednotlivcami, tak medzi týmami a firmami.
- Umožňuje ďalej **rozvíjať** základné **koncepty** a **myšlienky** v modelovaní.
- Zavádza špecifikáciu modelovacieho jazyka, ktorý **nie je závislý** na žiadnom programovacom jazyku.
- Zavádza všeobecné zásady pre **pochopenie základu modelovacieho jazyka**.
- Podporuje rozvoj **najmodernejších konceptov vývoja SW**, ako sú OOP, komponentné technológie, použitie vzorov, použitie framework ai.

- **Zjednocuje** najlepšie známe **techniky modelovania**.

Je zrejmé, že tieto vlastnosti stavia modelovací jazyk UML do jedinečného postavenia medzi ostatné modelovacie jazyky.

1.4 História UML

Teraz si v krátkosti zosumarizujeme historický vývin jazyka UML. Tento prehľad je dôležitý, lebo je ťažké pochopiť, kde sa UML dnes nachádza, bez porozumenia histórie toho, ako sa vyvíjal až po súčasnosť.

UML sa v podstate stal štandardom pre modelovanie softvérových aplikácií a jeho popularita narastá aj v modelovaní iných domén. Vznikol v rámci úsilia zjednodušiť a konsolidovať veľký počet objektovo orientovaných vývojových metód, ktoré v tom čase vznikali.

Jeho korene sú v troch odlišných metódach: **Boochova Metóda**, ktorej autorom je Grady Booch, **Metóda na modelovanie objektov** (Object Modeling Technique), ktorej spoluautorom je James Rumbaugh, a **Objectory**, autorom ktorej je Ivar Jacobson. Známi ako Traja kamaráti (Three amigos), Booch, Rumbaugh a Jacobson začali v roku 1994 pracovať na niečom, čo sa neskôr stalo prvou oficiálnou verziou jazyka UML.

V roku 1996 OMG vydalo požiadavku na návrhy ohľadom štandardného prístupu k objektovo orientovanému modelovaniu. Booch, Rumbaugh a Jacobson začali spolupracovať s metodológmi a vývojármi z rôznych spoločností, aby vytvorili návrh dostatočne zaujímavý pre členov OMG, ako aj modelovací jazyk, ktorý by bol všeobecne akceptovaný tvorcami CASE modelovacích nástrojov, metodológmi a vývojármi, ktorí by sa v konečnom dôsledku stali jeho používateľmi.

Finálny návrh UML bol predložený OMG v roku 1997 a bol výsledkom spolupráce mnohých ľudí. V tom istom roku bol jazyk UML akceptovaný OMG a vydaný ako UML verzia 1.1. UML odvtedy prešiel niekoľkými zmenami a vylepšeniami až po súčasnú verziu 2.0. Každá revízia sa snažila venovať problémom a nedostatkom, ktoré boli identifikované v predchádzajúcich verziách, čo viedlo k zaujímavému rozširovaniu a zmenšovaniu jazyka.

Obr.: Vývoj UML

Popis k obrázku:

- Vývoj od roku 1994 (Grady Booch a Jim Rumbaugh) – firma Rational Software (IBM) – výsledkom návrh UML (verzia 0.9) a metodika RUP (Rational Unified Process)
- Štandardizačná organizácia OMG v roku 1997 prijala ako štandard UML verziu **1.1**
- Ďalšie verzie **1.2** (1998), **1.3** (1999), **1.4** (2001) a **1.5** (2002).
- Väčšie zmeny boli začlenené do verzie **1.3** (hlavné zmeny: vzťah «uses» bol nahradený zovšeobecnením a závislosťou «include», «extend» bol preklasifikovaný na závislosť, bol definovaný formát vzťahu «extend» a rozširovacích bodov)
- Od roku 2001 verzia **2.0** – podstatné rozšírenie
- Od apríla 2006 verzia **2.1** (pridanie objektových diagramov, vylepšenia diagramov komponentov, diagramov nasadenia, sekvenčných diagramov, diagramov aktivít a štruktúrnych diagramov)

UML 2.0 je zatiaľ najrozsiahlejšia špecifikácia vzhľadom na počet strán (len samotná špecifikácia superštruktúry má vyše 600 strán), ale reprezentuje doteraz najkompaktnejšiu verziu UML.

Medzi niektoré význačné črty UML verzie 2.0 patria:

- Zosúladenie jadra UML s konceptuálnymi modelovacími časťami Meta Object Facility (MOF)
- Existencia a dostupnosť profilov, ktoré umožňujú definovať doménovo a technologicky špecifické rozšírenia UML
- Vylepšená verzia jazyka Object Constraint Language

1.5 Zhrnutie: Úvod do UML

Jazyk UML je založený na pohľade 4+1 na architektúru systému:

Logický pohľad - zachytáva slovník oblasti problému ako množinu tried a objektov. Dôraz je kladený najmä na zobrazenie spôsobu, akým objekty a triedy, tvoriace základ systému, implementujú jeho chovanie

Pohľad procesov – modeluje spustiteľné vlákna a procesy ako aktívne triedy. Je to procesovo orientovaný variant logického pohľadu, ktorý obsahuje rovnaké artefakty.

Pohľad implementácie – modeluje súbory a komponenty, ktoré utvárajú hotový kód systému. Zachytáva montáž systému a správu konfigurácie

Pohľad nasadenia – modeluje fyzické nasadenie komponentov na množinu fyzických výpočtových uzlov (počítačov a periférnych zariadení). Umožňuje modelovanie distribúcie komponentov na príslušné uzly distribuovaného systému.

Pohľad prípadov použitia – všetky pohľady sú zjednotené v pohľade prípadov použitia, ktorý popisuje požiadavky používateľa.

2.1 Stavebné bloky UML

V jazyku UML sa rozlišujú tri základné stavebné bloky (Building Blocks):

1. Predmety (Things)

Do predmetov sú radené samotné elementy modelu:

- **podstatné mená** - štruktúrne abstrakcie (triedy, rozhranie, spolupráca, prípady použitia, aktívne triedy, komponenty)
- **slovesá** - chovanie (interakcia, stav)
- **balíčky** - zoskupovanie významovo súvisiacich predmetov
- **poznámky** - anotácia, ktorú je možné k modelu pripojiť s úmyslom zachytiť informáciu

2. Relácie (Relations)

Predmety neležia vedľa seba ako chaotické zhluky nesúvisiacich informácií, ale sú vzájomne prepojené pomocou vzťahov. Vzťahy v modeli ukazujú, ako spolu jednotlivé predmety súvisia. UML rozoznáva nasledujúce typy vzťahov:

- **Asociácia** (Association)

Všeobecná súvislosť predmetov. UML presne definuje asociáciu ako "meniacu sa populáciu vzťahov daných prepojených objektov". Špeciálne variácie asociácie sú kompozícia a agregácia.

- **Závislosť** (Dependancy)

Zmena v jednom predmete spôsobí zmenu v inom predmete alebo mu poskytne požadovanoú informáciu. Napríklad keď trieda pre odosielanie emailu v aplikácii získa šablónu emailu z triedy globálnych konfiguračných parametrov, mali by sme tento vzťah zachytiť ako závislosť. Dobre zmapované závislosti nás neustále informujú o celkovej miere previazanosti jednotlivých častí modelu. Pokiaľ závislosti v modeli začnú pripomínať prepletené križovatky, je na čase vzťahy medzi triedami refaktorovať, pretože ide o spoľahlivý symptóm blížiaceho sa polčasu rozpadu implementovaného systému pri prvom pokuse o jeho ďalšie rozšírenie.

- **Generalizácia** (Generalisation)

Jeden predmet je špecializáciou iného predmetu. Napríklad trieda `osobné auto` je špecializáciou všeobecnej triedy `vozidlo`. Vzťahy generalizácie a špecializácie sú v objektovo orientovanom modelovaní veľmi dôležité, pretože ich neadekvátne použitie predznamenáva väčšinou krízu celého návrhu i jeho implementácie.

- **Realizácia** (Realization)

Realizácia je druh vzťahu, pri ktorom jeden predmet predstavuje dohodu, za ktorej plnenie je zodpovedný iný predmet. Napríklad z jazykov JAVA, C# a ďalších určite poznáte rozhranie, ktoré reprezentujú dohodu, ktorej plnenie je požadované po triedach, ktorých rozhranie implementujú.

3. Diagramy (Diagrams)

Diagramy zachytávajú rôzne aspekty modelovaného systému. Bude im venovaná samostatná kapitola.

2.2 Modelovanie a metamodelovanie

Modelovanie

Modelovanie je prostriedkom na zachytenie ideí, vzťahov, rozhodnutí a požiadaviek v dobre definovanej notácii, ktorá môže byť použitá pre rôzne domény. Model je popis (časti) systému v dobre definovanom jazyku. Dobre definovaný jazyk je jazyk s dobre definovanou formou (syntaxou) a významom (sémantikou), ktorý je vhodný na automatizovanú interpretáciu počítačom.

Modelovanie umožňuje lepšie pochopenie systému. Častokrát je potrebných niekoľko prepojených modelov, aby sme boli schopní systému naozaj porozumieť. Pre softvérové systémy je preto dôležitý taký modelovací jazyk, ktorý poskytuje niekoľko rôznych pohľadov na systém a jeho architektúru, ako aj na jeho vývoj počas životného cyklu vývoja softvéru.

Jazyk UML nie je obmedzený len na modelovanie softvéru. V skutočnosti je dostatočne expresívny, aby dokázal modelovať aj nesoftvérové systémy.

Metamodelovanie

Model je definovaný ako popis systému (alebo jeho časti) napísaný v dobre definovanom jazyku. Mechanizmus definovania a vytvárania takého dobre definovaného modelovaciaho jazyka sa nazýva metamodelovanie.

Pri metamodelovaní rozlišujeme najmä medzi modelmi a metamodelmi. Model definuje, aké elementy môžu existovať v systéme. Metamodel je výsledkom procesu abstrakcie, klasifikácie a zovšeobecňovania problémovej domény modelovacieho jazyka. Teda metamodel je v zásade modelom modelovacieho jazyka a definuje elementy, ktoré môžu byť použité v modeli.

Modelovanie a metamodelovanie sú v skutočnosti identické aktivity, rozdiel je len v interpretácii. Modely sú abstraktné reprezentácie skutočných systémov alebo procesov. A keď proces, ktorý modelujeme, je procesom vytvárania iných modelov, vtedy sa modelovanie stáva metamodelovaním.

Každý element, ktorý môže byť použitý v modeli, je definovaný v metamodeli jazyka. V jazyku UML je možné používať triedy, atribúty, asociácie a iné, pretože v metamodeli jazyka UML sú elementy, ktoré definujú, čo je to trieda, atribút, asociácia a podobne. Keby napr. metatrieda Asociácia nebola zahrnutá v UML metamodeli, nebolo by možné definovať asociáciu ani v UML modeli.

Pri znázornení vzťahu medzi metamodelom a modelom dostaneme dve vrstvy. Tento vzťah znázorňuje obrázok 1. Vrstva metamodelu definuje napr., čo je to Trieda (Class). Hovorí o tom, že trieda môže mať atribúty a operácie, a že medzi triedami môžu existovať asociácie. Trieda v metamodeli je metatrieda, koncept, ktorý popisuje čo to trieda je a ako sa používa. Inštanciou metatriedy Trieda je konkrétna trieda, ktorú môžeme vidieť v UML diagrame. Rôzne inštancie metatriedy Trieda popisujú rôzne typy objektov. Metamodel takisto definuje aj Asociáciu, ktorá je tiež metatriedou, podobne ako Trieda. V tomto dvojvrstvovom príklade, vrstva metamodelu definuje symboly ako triedy a asociácie, ktoré môžu byť použité v modeli. Vrstva modelu popisuje informácie ako napr. osoby, autá a ich vzťahy, viď obrázok, použitím symbolov definovaných v metamodeli. UML diagramy vytvorené vývojármi existujú na vrstve modelu. Vrstva metamodelu definuje pravidlá, podľa ktorých je možné vytvárať diagramy a definovať elementy diagramu.

Obr.: Vzťah medzi modelom a metamodelom

2.3 Architektúra UML

OMG definuje štvorvrstvovú architektúru, ktorú používa pre svoje štandardy. Jednotlivé vrstvy sa nazývajú M0 (inštancie), M1 (model systému), M2 (metamodel), M3 (meta-metamodel).

Vlastnosti štyroch vrstiev pre meta-modelovanie sú vysvetlené v nasledujúcej tabuľke:

Vrstva	Popis vrstvy	Príklad
Meta-metamodel	Definuje pravidlá jazyka pre tvorbu metamodelu (viď nižšia vrstva)	MetaClass, MetaAttribute
Metamodel	Inštancia meta-metamodelu je metamodel Definuje pravidlá pre tvorbu modelu	Class, Attribute
Model	Inštancia metamodelu je model. Definuje pravidlá popisujúce informačnú doménu („typy“)	Osoba, Úver, Ručiteľ úveru, Cenný papier, Rodné číslo, atď.
Užívateľské objekty	Inštancia modelu sú užívateľské „dáta“	Ján Novák, Úver číslo 10215, 541212/2312

Vrstva M0: Inštancie

Na tejto vrstve sa nachádza bežiaci systém (run-time) s aktuálnymi inštanciami, ktoré v ňom existujú. Tieto inštancie sú napríklad zákazník "Janko Mrkvička" bývajúci na "Všeobecnej ulici 25" v "Bratislave". Vrstva M0 teda obsahuje dáta aplikácie, napríklad inštancie objektovo-orientovaného systému alebo riadky tabuliek v relačnej databáze. Táto vrstva je inštanciou modelu, teda vrstvy M1, ktorá sa nachádza o úroveň vyššie.

Všimnime si, že keď modelujeme biznis a nie software, inštancie na vrstve M0 sú konkrétne entity daného biznisu, napr. zamestnanci, faktúry a produkty. Keď modelujeme software, inštancie sú softwarovými reprezentáciami skutočných entít, napríklad počítačové verzie faktúr, objednávok a pod.

Vrstva M1: Model systému

Táto vrstva obsahuje modely, napríklad UML model softwarového systému. Vrstva M1 je model, ktorý popisuje artefakty a pravidlá danej domény. Model je inštanciou metamodelu. V M1 modeli je napríklad definovaná trieda Zákazník s atribútmi meno, ulica a mesto. Na tejto úrovni sa nachádzajú aj diagramy (diagramy tried, sekvenčné diagramy a iné..).

Medzi vrstvami M0 a M1 je určitý vzťah. Koncepty na vrstve M1 sú všetky klasifikáciami inštancií na vrstve M0. Podobne, každý element vrstvy M0 je vždy inštanciou elementu vrstvy M1. Zákazník "Janko Mrkvička" je inštanciou M1 elementu triedy Zákazník.

Vrstva M2: Metamodel

Hlavnou úlohou vrstvy metamodelu je definovať jazyk na špecifikovanie modelov. Elementy vrstvy M2 sú teda modelovacie jazyky. Vrstva M2 definuje koncepty, ktoré môžu byť použité pri modelovaní elementu vrstvy M1 a pravidlá pre model na vrstve M1. V prípade UML, vrstva M2 definuje metatriedy "Trieda (Class)", "Asociácia" a pod. Metamodel je inštancia meta-metamodelu, čo znamená, že každý element metamodelu je inštanciou niektorého elementu v meta-metamodeli. Známe príklady metamodelov sú UML a OMG Common Warehouse Metamodel (CWM).

Rovnaký vzťah ako je medzi elementami vrstiev M0 a M1 existuje aj medzi elementmi vrstiev M1 a M2. Každý element vrstvy M1 je inštanciou niektorého M2 elementu, a každý element vrstvy M2 klasifikuje M1 elementy.

Vrstva M3: Meta-metamodel

Vrstva meta-metamodelu tvorí základ metamodelovacej architektúry. Vrstva M3 definuje koncepty, ktoré môžu byť použité pri definovaní modelovacích jazykov. Táto úroveň abstrakcie podporuje vytváranie mnohých rôznych modelov pochádzajúcich z rovnakej množiny základných konceptov. Takže koncept UML Triedy, ktorý patri do M2, môže byť považovaný za inštanciu prislúchajúceho elementu z M3, ktorý presne definuje, čo je Trieda a jej vzťahy s ostatnými UML konceptmi.

Zase platí, že rovnaký vzťah, ako je medzi elementmi vrstiev M0 a M1, a elementmi vrstiev M1 a M2, existuje aj medzi elementmi z vrstiev M2 a M3. V rámci OMG, MOF je štandardný M3 jazyk. Všetky modelovacie jazyky (ako napr. UML, CWM a pod.) sú inštancie MOF.

Obr: Príklad švorvrstvovej architektúry jazyka UML

Hoci toto je konečná podoba UML architektúry, viacvrstvová architektúra môže mať v skutočnosti nekonečný počet vrstiev. Nasledujúce vyššie vrstvy vzniknú procesom abstrakcie, prirodzeného procesu spresňovania a zjemňovania pravidiel. Je to tak trochu ako s matematikou, v tom zmysle, že ako postupne robíme pokroky v našich vedomostiach, objavujeme najprv základné koncepty ako napr. sčítanie a odčítanie. Potom zistíme, že existujú pravidlá, ktoré hovoria o tom, prečo a ako sčítanie a odčítanie fungujú. Pri našom ďalšom štúdiu narážame na stále všeobecnejšie a abstraktnejšie princípy, ktoré môžu byť použité v rôznych oblastiach matematiky.

2.4 UML pohľady na systém

Aké hlavné aspekty modelovanej aplikácie dovoľuje UML zachytiť?

Model v UML sa skladá z rôznych diagramov, ktoré predstavujú náhľady na rôzne časti sémantického základu navrhovanej aplikácie. Žiadny dvojrozmerný diagram nemôže zachytiť komplexnú aplikáciu v celku, ale sústredí sa vždy práve na jeden dôležitý aspekt.

Jazyk UML rozoznáva päť základných pohľadov na systém:

- **Logický pohľad** sa zaoberá najmä pojmami z problémovej domény zadávateľa a ich vzájomnými statickými vzťahmi. Predstavuje funkciu systému a slovník (problémové oblasti = množina tried a objektov)
- **Procesný pohľad** sa sústreďuje na chovanie systému, ktoré musí spĺňať požiadavky a obmedzenia z prípadu použitia, ktoré sú kladené na priebeh procesu. Predstavuje výkon systému, škálovateľnosť a priepustnosť – modeluje spustiteľné vlákna a procesy ako aktívne triedy.
- **Implementačný pohľad** zachytáva fyzické rozdelenie aplikácie na samostatné komponenty a ich závislosti. Predstavuje montáž systému a správu konfigurácie – modeluje súbory a komponenty, ktoré utvárajú hotový kód systému
- **Pohľad nasadenia** mapuje komponenty na množinu fyzických výpočtových uzlov v cieľovom prostredí. Predstavuje topológiu systému, distribúciu, doručenie a inštaláciu – modeluje fyzické nasadenie komponentov na množinu fyzických výpočtových uzlov (počítačov a periférnych zariadení)
- **Pohľad prípadov použitia**. V prípadoch použitia sú vyjadrené základné požiadavky kladené na systém. Sú v ňom zjednotené všetky 4 predchádzajúce pohľady – zachytáva základné požiadavky kladené na príslušný systém ako množinu prípadov použitia.

Pohľady sú konkretizované v nasledujúcich typoch diagramu, ktorých detailný popis bude predmetom samostatnej kapitoly.

- Diagram prípadu použitia
- Diagram tried
- Diagram objektu
- Diagram komponentov
- Diagram nasadenia
- Sekvenčný diagram
- Kolaboračný diagram
- Stavový diagram
- Diagram aktivít

Obr.: Grafické zobrazenie pohľadov na systém

2.5 Odporúčania pri tvorbe UML modelu

Treba zdôrazniť, že nasledujúce rady tvorcov UML sa netýkajú problému „ako modelovať v UML“, t.j. samotných vývojárov, ale tieto doporučenia sú smerované vedúcim projektov v štýle „ako riadiť projekt v objektovom a komponentnom prostredí pri použití UML“.

Projekt riadený Use Case modelom

Prvá rada sa týka použitia tzv. Use Case modelu v riadení projektu. (Samotnému Use Case modelu bude venovaná celá kapitola.) Tu si iba všimnime, že Use Case model sa nepoužíva iba pre čisté „vývojové“ práce (ako súčasť celkového modelu softwaru), ale používa sa tiež

vedúcim projektu pre riadenie projektu a stáva sa tak dôležitým dokumentom používaným pre metodické riadenie v projekte.

Navrhovať a dodržovať správnu architektúru

Druhé doporučenie v znení „vedúci, zamerajte sa na architektúru“ neznamená zamerať sa na nejaký návrh tabuliek apod. Pod architektúrou majú autori UML na mysli rozvrstvenie systému a jeho rozloženie do rozdeliteľných celkov. Jedná sa vlastne o priamy dôsledok správneho použitia objektového a komponentného prístupu. Tvorcovia UML doporučujú, aby sa tiež riadenie projektu zameralo na toto rozvrstvenie systému, t.j. na objekty, na komponenty a na skupiny objektov a skupiny komponent, tzv. vrstvy. Jednoducho povedané, autori UML doporučujú vedúcim projektov jednoduchú a zrozumiteľnú zásadu: Pre vedúceho projektu je omnoho ľahšie a efektívnejšie riadiť práce nad „niekoľkými oddeliteľnými časťami systému“ než nad „jedným veľkým, komplikovaným a previazaným celkom, kde všetko súvisí so všetkým“. Zásadou objektového modelovania je pochopiteľne rozloženie na vrstvy, t.j. na objekty a komponenty a na vrstvy objektov a vrstvy komponentov. Toto rozloženie má veľký kladný vplyv na riadenie projektu, samozrejme za podmienky, že sa tento pohľad z hľadiska riadenia nezanedbáva.

3. Metodika UML

Ciele

Kľúčové slová

UML je zamýšľaný ako univerzálny štandard pre záznam, konštrukciu, vizualizáciu a dokumentáciu artefaktov systémov s prevažne softvérovou charakteristikou. Jazyk UML je definovaný niekoľkými dokumentmi publikovanými OMG. Tieto dokumenty sú dostupné na oficiálnej stránke OMG (http://www.omg.org). Definícia UML obsahuje 4 základné časti:

- **Definícia notácie UML** (Syntax / UML Superštruktúra)
- **Sémantika UML** (Metamodel / UML Infraštruktúra)

- **Jazyk OCL** (Object Constraint Language) pre popis ďalších vlastností modelu
- **Výmena diagramov** (Diagram Interchange Specification)

V nasledujúcich statiach si popíšeme jednotlivé špecifikácie.

3.1 Notácia UML

Definícia notácie UML / Syntax / UML Superštruktúra

Superštruktúra formálne definuje elementy jazyka UML. V podstate definuje kompletný jazyk UML tak, ako ho poznajú používatelia. Infraštruktúra obsahuje podmnožinu nazvanú kernel, ktorá zahŕňa do dokumentu superštruktúry všetky relevantné časti infraštruktúry. Táto špecifikácia je obvykle používaná tvorcami CASE nástrojov a autormi kníh o UML, aj keď boli už určité snahy spraviť ju čitateľnejšiu aj pre širšiu verejnosť.

Základom definície syntaxe UML je teda špecifikácia lexikálnych elementov – najmenších lexikálnych jednotiek, z ktorých sa môže dokumentácia v UML skladať. Existujú 4 základné druhy lexikálnych elementov UML:

- **reťazec** (postupnosť znakov, znaková sada nie je obmedzená, v niektorých prípadoch je doporučené používať ASCII kód – zaistí to ľahký prenos do programového prostredia),
- **ikona** (grafický symbol zastupujúci element, neobsahujúci žiadne zložky),
- **2D-symbol** (grafický symbol zastupujúci element, ktorý má obsah, prípadne rozdelený na zložky),
- **spojka** (path - postupnosť úsečiek, ktoré nadväzujú, môžu mať zvýraznené koncové body a môžu incidovať s hranicou 2D-symbolov - slúži k vyznačeniu prepojení).

Elementy modelu dokumentovaného v UML potrebujeme označovať - potrebujeme mená objektov, tried, atribútov, metód, a pod. Ako mená sa v UML používajú **identifikátory** – špeciálne lexikálne elementy typu reťazec, ktoré obsahujú iba ASCII písmená, cifry a znaky ‘_’ alebo ‘-’. Pre zápis identifikátorov, ktoré zastupujú viacslovný termín je vhodné voliť nejakú konvenciu, doporučené je používanie niektorého z nasledujúcich zápisov:

jozefNovak, jozef_novak, jozef-novak

Meno sa zvyčajne vzťahuje k nejakému kontextu (scope), potom hovoríme o tzv. **kvalifikovanom mene**.

Napr. úplné kvalifikované meno pre objekt “objednávka”, o ktorom hovoríme v rámci “účtovníctva” bude

Uctovnictvo::Objednavka

Reťazec pridružený ku grafickému symbolu nazývame **návestie**. Pokiaľ má identifikátor význam kľúčového slova, uvádzame ho vo “francúzskych” zátvorkách (**guillemets**):

<<kľúčové slovo>>

Výrazy v UML sú reťazce zostavené z lexikálnych elementov podľa istých pravidiel. Syntax výrazov nie je striktná, musí byť ale zvolený nejaký dobre definovaný jazyk. Tj. výraz predstavuje formulu, ktorú musí byť možné v danom jazyku vyhodnotiť (má svoju **sémantiku**). Definícia UML zahŕňa definíciu jazyka OCL (Object Constraint Language), ktorý je použitý pre definíciu sémantiky UML (metamodelu UML). Je doporučené používať OCL všade tam, kde je to možné.

3.2 Sémantika UML

Sémantika / Metamodel/ UML Infraštruktúra

Zápisy v UML (pohľady, diagramy) zostavené podľa pravidiel syntaxe musia mať pevne definovaný význam. Sú to všeobecné pravidlá (všeobecné mechanizmy), ktoré riadia tvorbu modelu. Tieto pravidlá sú neustále prítomné pri tvorbe ľubovolných diagramov a určujú legálne zdieľané komunikačné stratégie medzi rôznymi analytikmi a návrhármi. Namiesto živelných nápadov a kryptických hračiek vznikajú vďaka všeobecným mechanizmom jazyka UML všeobecne zrozumiteľné modely.

Infraštruktúra je vlastne metamodel (model popisujúci model), ktorý je použitý na vytvorenie zvyšku UML. Táto špecifikácia zvyčajne nie je používaná koncovým užívateľom UML, ale poskytuje základy pre UML Superštruktúru.

Týmto sme si definovali, čím sa zaoberá popis sémantiky UML. Je rozčlenený do 4 vrstiev, ktoré na rôznej úrovni abstrakcie popisujú vlastnosti diagramov UML, vrátane možných rozšírení. Na najnižšej úrovni sú špecifikované vlastnosti primitívnych údajov (dát) - typy dát, obory hodnôt atribútov. O úroveň vyššie je vyjadrená sémantika užívateľských objektov (tzv. model, alebo tiež meta-data) - napr. čo to je záznam o objekte typu "zamestnanec". Ešte vyššie stojí popis prvkov modelu (metamodel) - napr. čo to je trieda, atribút, vzťah, atď. Najvyššie je potom definícia vlastností metamodelu (meta-metamodel), napr. ako je možné korektne vytvárať nové prvky modelu.

.

Sémantika používa 4 všeobecné mechanizmy (vrstvy):

- **Špecifikácia** (Specification).
- **Podrobnosti** (Adornments, též Ozdoby).
- **Všeobecné delenie** (Common Divisions, taktiež Podskupiny).
- **Rozšírenie**

Každej vrstve je venovaná samostatná stať.

Úlohy

1. Popíšte 4 vrstvy sémantiky jazyka UML.

3.2.1 Špecifikácia

Špecifikácia predstavuje sémantický základ modelu. Sú to textové (a grafické) popisy funkcií a sémantiky jednotlivých elementov použitých v modeli (jadro modelu).

3.2.2 Podrobnosti

(Adornments, taktiež Ozdoby)

Každý element modelu môže byť v rôznych diagramoch reprezentovaný iným výsekom z množiny všetkých informácií, ktoré sú o ňom známe. Pri modelovaní zložitého systému sa často dostaneme do situácie, kedy sa jeden element vyskytne na viacerých diagramoch. Predstavme si systém, kde je na jednom diagrame tried zložitá trieda Užívateľ zachytená so všetkými

atribútmi a metódami. V inom diagrame pre realizáciu objednávok chceme iba modelovať reláciu, ktorá vyjadruje fakt, že práve jeden užívateľ je zakladateľom objednávky. Na diagrame pre realizáciu objednávok bude zo všetkých informácií o triede „Užívateľ“ zobrazený z dôvodu prehľadnosti iba jej názov, pretože jej atribúty a metódy sú tu irelevantné.

3.2.3 Všeobecné delenie*

(Common Divisions, taktiež Podskupiny)

Prvé všeobecné delenie, s názvom "**klasifikátor a inštancia**", zavádza dištinkciu medzi typom a inštanciou typu. Pre tých, ktorí programujú v ľubovolnom objektovo orientovanom jazyku, stačí povedať, že príkladom je rozlíšenie medzi triedou a inštanciou triedy (objektom). Pre návrhárov nedotknutých programovaním uvediem jednoduchý príklad – slovo počítač označuje našu abstraktnú predstavu o počítači (klasifikátor pre všetky počítače), konkrétny počítač, na ktorom tento článok píšem, je jednoznačnou inštanciou klasifikátora. Aristoteles, ktorému patrí prvenstvo v rozlišovaní medzi typom a inštanciou typu, je určite potešený, že jeho myšlienky v 21. storočí zaujímajú čelné miesto v pravidlách UML. Druhé všeobecné delenie má názov "**rozhranie a implementácia**" a týka sa odlíšenia toho, čo predmet vykonáva, od toho, ako to vykonáva. Rozhraní iba hovorí, čo sa predmet, ktorý rozhranie poskytuje, zaväzuje dodržať, ale celkom opomína spôsob, ako to urobí. Opäť pre tých z vás, ktorí programujete triedy a komponenty, určite tento koncept nie je nový. Internou implementáciou triedy môžete vždy meniť celkom nezávisle na jej rozhraní, pričom by tým boli dotknutí existujúci klienti triedy. Pre ostatných uvediem zase triviálny príklad – pri mikrovlnej trúbe ma zaujímajú iba ovládacie prvky (rozhranie), pomocou ktorých si ohrejem jedlo, ale je mi celkom jedno, aké procesy sa po spustení vo vnútri trúby dejú (implementácia).

<u>Prvé delenie</u>

- **Klasifikátor** - abstraktné vyjadrenie typu predmetu
- **Inštancia** - špecifický výskyt typu predmetu

Druhe delenie

- **Rozhranie** - dohoda špecifikujúca chovanie predmetu
- **Implementácia** - špecifikuje podrobnosti chovania

3.2.4 Rozšírenie

UML sa snaží byť univerzálnou notáciou pre všestranné použitie od obchodného modelovania po detailný návrh systémov pre prácu v reálnom čase. Notácia, ktorá by vyčerpávajúcim spôsobom pokryla tak široké spektrum potrieb by ale bola veľmi komplikovaná a zložitá. UML preto definuje iba základnú časť (UML core) a umožňuje rozšírenie notácie podľa konkrétnych potrieb. Rozšírenie UML je možné dosiahnuť použitím iného jazyka (než OCL) pre výrazy. Štandardnú sémantiku UML je možné doplniť štandardnými obmedzeniami, ktoré umožňujú zmeniť interpretáciu elementov:

- **príznaky** (tags) – umožňujú pridať k prvku diagramu ďalšie informácie (atribúty),
- **obmedzenia** (constraints) – umožňujú špecifikovať obmedzenia pre elementy (hodnoty atribútov),
- **stereotypy** (stereotypes) – umožňujú klasifikovať elementy.

Stereotypy umožňujú klasifikovať elementy diagramov a tým vyjadriť ďalšiu sémantiku. Zapisujú sa ako kľúčové slová, ale môžu sa prípadne zobraziť ako ikony, čo sa príliš nedoporučuje - porušuje to princíp jednotnosti, pretože tieto ikony nie sú štandardné. Ako príklad môžu poslúžiť stereotypy:

<< database >> (označenie komponentov zaoberajúcich sa správou dát)

<< entity >> (označenie triedy reprezentujúcej dáta)

Existujú štandardné stereotypy, napr. << include >>, << extend >>, ktorých význam je definovaný sémantikou UML. Príznaky umožňujú pridať k prvku diagramu ďalšie informácie vo forme dvojice atribút=hodnota:

{ autor="Jozef Novák", verzia=1.2 }

Názov atribútu môže byť ľubovolný. Príznaky sa zapisujú do zložených zátvoriek, pripájajú sa k názvu prvku diagramu a typicky je ich možné použiť napríklad ku špecifikácii verzie, autora, či implementačných poznámok. Obmedzenia umožňujú špecifikovať požiadavky na sémantiku prvkov – je možné pomocou nich formulovať rôzne obmedzenia v modeli. Zapisujú sa ako výraz uzavretý do zložených zátvoriek. Príklad:

{ počet_záznamov < 10000 }

Obmedzenia sa môžu týkať viacerých prvkov - potom je spojené s týmito prvkami čiarkovanou čiarou. Často sa kombinuje s poznámkami. Poznámka obsahuje ľubovolný text. Zobrazuje sa ako obdĺžnik s „prehnutým rohom“. Môže byť pridružená k elementu, môže obsahovať stereotyp (napr. << constraint >> - potom ide o špecifikáciu obmedzení).

3.3 Jazyk OCL

Nie všetky vlastnosti modelu je možné vyjadriť pomocou diagramu. Uvažujme napr. dátový model systému, ktorý sa zaoberá evidenciou zamestnancov. O každom zamestnancovi si potrebujeme pamätať rôzne atribúty, mimo iné napr. plat zamestnanca. Takisto si potrebujeme pamätať hierarchickú štruktúru medzi zamestnancami. Pomocou diagramov vyjadríme ľahko požiadavky na štruktúru dát, ťažko tu však zachytíme požiadavky ako "nadriadený musí mať vyšší plat než jeho podriadení". UML ponúka (ale nevnucuje) možnosť použiť pre vyjadrenie takýchto obmedzení špecifikačný jazyk OCL (Object Constraint Language). OCL slúži práve na vyjadrenie komplikovanejších integritných obmedzení, popis vlastností operácií a môže byť konieckoncov použitý aj pre vyjadrenie sémantiky UML.

OCL špecifikácia definuje jazyk na písanie rôznych obmedzujúcich podmienok a výrazov pre elementy modelu. OCL sa často využíva v situácii, keď prispôsobujeme UML konkrétnej doméne a potrebujeme použiť určité obmedzenia, ale takisto sa používa aj na formálne spresnenie modelov.

3.4 Výmena diagramov

Táto špecifikácia bola napísaná za účelom poskytnutia spôsobu, ktorý umožní zdieľať UML modely medzi rôznymi modelovacími CASE nástrojmi. Predchádzajúce verzie jazyka UML definovali XML (Extensible Markup Language) schému na zachytenie informácie o použitých elementoch v UML diagrame, ale táto XML schema neobsahovala informáciu o tom, ako sú tieto elementy v diagrame usporiadané. Na vyriešenie tohto problému bola táto špecifikácia vyvinutá spoločne s mapovaním z novej XML schémy do SVG (Scalable Vector Graphics) reprezentácie.

3.5 Sekvenčný diagram

(Sequence diagram)

Diagram popisuje spoluprácu objektov pomocou znázornenia sekvencie zaslaných správ. Jedna sekvencia zaslania správ sa taktiež nazýva scenár. Vyskytuje sa ako v analýze, tak v designe.

Sekvenčný diagram (tiež Scenár činností) dokumentuje spoluprácu participantov na **scenári** činnosti. Kladie pritom dôraz na **časový** aspekt komunikácie. Dokumentuje **objekty** a **správy**, ktoré si objekty posielajú pri riešení scenára. Je vhodný pre popis scenára pri komunikácii s užívateľmi.

3.6 Diagram spolupráce

(Collaboration diagram)

Podobne ako scenáre činností dokumentuje diagram spolupráce **spoluprácu** objektov pri riešení úlohy. Kladie väčší dôraz na **komunikačný** aspekt (čas je vyjadrený číslovaním). Dokumentuje **objekty** a **správy**, ktoré si objekty posielajú pri riešení problému. Je vhodný pre popis spolupráce objektov pri návrhu komunikácie. Vyskytuje sa ako v analýze tak aj v designe.

3.7 Stavový diagram

(Statechart diagram)

Stavové diagramy slúžia k popisu dynamiky systému. Stavový diagram definuje možné **stavy**, možné **prechody** medzi stavmi, **udalosti**, ktoré prechody iniciujú, **podmienky** prechodov a **akcie**, ktoré s prechodmi súvisia. Stavový diagram je možné použiť pre popis dynamiky objektu (pokiaľ má rozpoznateľné stavy), pre popis metódy (pokiaľ poznáme algoritmus), alebo pre

popis protokolu (vrátane protokolu o styku užívateľa so systémom). Prechod môže byť ohodnotený:

udalosť(parametre)[podmienka]/akcia^správa

Každý stav môže ďalej obsahovať popis akcií pre udalosti vstup, výstup a opakované prevádzanie:

entry/akcia

exit/akcia

do/akcia

Stavové diagramy môžu byť hierarchické. V najnovších verziách UML môžu obsahovať tzv. synchronizačné značky (pozri ďalej).

Vyskytuje sa ako v analýze tak aj v designe.

3.8 Diagram aktivít

(Activity Diagram)

Variant stavových diagramov, kde okrem stavov používame **aktivity**. Ak sa nachádza systém v nejakom stave, je prechod do iného stavu iniciovaný nejakou vonkajšou udalosťou. Pri aktivite je, na rozdiel od stavu, prechod iniciovaný ukončením aktivity, prechody sú vyvolané dokončením akcie (sú synchrónne). Používajú sa pre dokumentáciu tried, metód, alebo prípadov použitia (ako „workflow“). Nahrádzajú do určitej miery v UML neexistujúce diagramy dátových tokov. Môžu obsahovať symbol „rozhodnutia“.

Vyskytuje sa ako v analýze, tak aj designe.

3.9 Diagram komponentov

(Component Diagram)

Vyjadruje (fyzickú) štruktúru **komponentov** systému. Popisuje typy komponentov – inštancie komponentov sú vyjadrené v diagrame nasadenia. Komponenty môžu byť vnorené do iných komponentov. Pri vyjadrovaní **vzťahu** medzi komponentmi je možné používať „interface“.

Vyskytuje sa iba v designe.

3.10 Diagram nasadenia

(Deployment Diagrams)

Diagramy nasadenia popisujú fyzické rozmiestnenie elementov systému na uzly výpočtového systému. Uzly a elementy sú označené podobne ako objekty a triedy (môže byť uvedený iba typ, alebo konkrétna inštancia a typ - podčiarknutá). Popisujú potrebné väzby medzi uzlami (prípadne tiež použitý protokol - „interface“). Obsahujú iba **komponenty** potrebné pre beh aplikácie - komponenty potrebné pre preklad a zostavenie sú uvedené v diagrame komponentov.
Vyskytuje sa len v designe.

3.11 Hlavné a vedľajšie modely

Nezastupiteľné modely

Nie všetky modely sú si z hľadiska dôležitosti pre projekt rovnocenné. Niektoré z nich sú dokonca medzi sebou zastupiteľné. Uveďme si, ktoré modely je možné považovať za podstatné, nevyhnutné a nezastupiteľné, a to z akých dôvodov:

Use Case model

Je dôležitý, pretože poskytuje abstraktný popis systému bez implementačných podrobností. Jeho prvky a v ňom použité formulácie sú zrozumiteľné tak pre vývojárov („programátorov“), tak pre užívateľov a konzultantov neznalých vôbec nejakej teórie informačných systémov a programovania (buď štrukturálneho alebo objektového). Týmto sa stáva tento model kľúčovým článkom celého vývoja projektu. Naviac výstupy z neho sa znovupoužijú i v iných činnostiach projektu, napríklad pre riadenie projektu, pre vyhotovenie užívateľskej dokumentácie, pre obchodné oddelenie, pre testovanie, atď. Táto možnosť znovu použiť Use Case model z projektu vývoja aj pre iné činnosti, ako iba pre samotný vývoj, je daná jeho povahou – Use

Case model je totiž abstraktným a zrozumiteľným popisom systému a preto ho možno použiť aj v iných oblastiach, na prvý pohľad vzdialených modelovaniu IS.

Class model

Je nevyhnutný, pretože je ako hotový model východzím „zdrojom pre kód“. Zjednodušene povedané, výsledný zdrojový kód je obrazom Class modelu v danom programovacom jazyku, alebo obrátene povedané, hotový celý a úplný Class model je grafickou obdobou zdrojového kódu.

Vývoj v Class modeli sa deje po častiach tzv. iteratívnou a inkrementálnou metódou (bude popísané ďalej). Triedy a ich vzťahy sa vždy vyvíjajú v dvoch fázach: v prvej fáze sa vytvára model čisto analytický, tj. obsahuje iba triedy, atribúty a metódy tried iba v rámci chápania abstraktného systému v problémovej doméne, v druhej fáze sa tento model doplní o podrobnosti designu, tj. služobné triedy, metódy a atribúty príslušné danému prostrediu apod.

Component model

Je nevyhnutný, pretože ukazuje rozmiestnenie tried do jednotlivých modulov resp. komponentov.

V teórii komponentov existujú dva základné možné prístupy: Rozlišujú sa komponenty buď ako

- binárne komponenty (binary component) alebo ako tzv.
- zdrojové komponenty (source component).

V prvom prípade sa celý systém chápe ako celok zložený zo „skompilovaných prepojených balíkov“ (čo sú binárne komponenty), v druhom, prípade v tzv. modulárnom prístupe sa časti systému linkujú čisto na úrovni zdrojového kódu a systém sa skládá z modulov zdrojových knižníc pred kompiláciou (tu hraje komponent úlohu modulu, čoby oddeliteľejá časti zdrojového kódu). Komponentný model teda ukazuje jednak rozmiestenie tried v moduloch a komponentoch, jednak závislosti komponentov (modulov) medzi sebou.

Model je nevyhnutným preto, že ukazuje rozloženie systému do komponentov buď binárnych alebo zdrojových a to vo vzťahu k triedam, ktoré tieto komponenty pred kompiláciou tvoria

(skladajú). Súčasne ukazuje vzťah závislosti (dependency) medzi komponentami a modulmi, a tým vyjadruje nutné konfigurácie skladaných systémov a určuje tak pravidlá linkovania častí systému.

Deployment model

Je nevyhnutný, pretože celý už naprogramovaný systém sa musí v konečnom dôsledku „na niečom sprevádzkovať“. V tomto modely sa popisuje aké budú použité stroje, ako budú zapojené do siete, aké budú zdroje (procesory, apod.), disky, atď. Súčasťou modelu je tiež informácia, kde bude ktorý naprogramovaný SW nainštalovaný, ako bude riadený jeho chod apod.

Tento model zodpovedá návrhu HW a rozmiestneniu SW na ňom, čo je už veľmi implementačná záležitosť a nebudeme ju tu podrobne rozoberať.

Sprievodné modely

Ostatné modely UML (sekvenčný model, model aktivít, stavový model atď.) buď tvorbu týchto základných modelov podporujú, tj. navedú ku správnemu riešeniu, alebo ho pomáhajú upresniť, alebo sa pomocou týchto ďalších modelov vytvárajú všeobecné vzory apod. Tieto modely by skôr mali byť nazvané „sprievodnými modelmi“ a dokonca ich nemusíme považovať za povinné (i keď sú samozrejme pre tvorbu IS prínosom).

Dôležité je, že štyri základné modely vymenované vyššie zavádzajú v konečnom dôsledku tiež určitý systém postupu práce vývoja. I keď každý z nich sa riadi inými pravidlami pre riadenie práce, možno znázorniť ich časový vzťah tvorby nasledujúcim obrázkom.

Postupnosť práce na základných modeloch

Osvedčené je používať prakticky takúto „minimálnu“ zostavu modelov:

1. Vždy zaviesť **Use Case Model, Class model, Component** a **Deployment modely**. Tieto modely musia byť vytvorené v celom rozsahu.
2. **Sequence model** doporučujeme používať v popise určitých zložitých scenárov a v zavedení vzorov (design patterns podľa štandardného chápania spolupráce vo vzore). Pri zavedení tohoto diagramu nie je potrebné zavádzať Collaboration diagram ako jeho alternatívny diagram.
3. **Object diagram** doporučujeme zaviesť pre osvetlenie niektorých vzťahov medzi objektami, ktoré nemusia byť zreteľne viditeľné z Class diagramu (ale treba podotknúť, že každý vzťah z Object diagramu je odvoditeľný z Class diagramu). Class diagram je totiž svojim poňatím o úroveň abstrakcie vyššie, ako Object diagram (Object diagram ukazuje „konkrétny príklad vzťahu inštancií“, avšak Class diagram zovšeobecňuje vzťah mezi triedami týchto inštancií)

V prípade potreby doporučujeme zaviesť **Stavový diagram** (State Chart diagram) pre vystihnutie stavov objektov a to hlavne v analýze.

3.12 Zhrnutie: Diagramy UML

Zhrnutie

- **diagramy tried a diagramy objektov** (class diagrams, object diagrams) popisujú statickú štruktúru systému, znázorňujú dátový model systému od konceptuálnej úrovne až po implementáciu,
- **modely jednania** (diagramy prípadov použitia - use case diagrams) dokumentujú možné prípady použitia systému - udalosti, na ktoré musí systém reagovať,
- **scenáre činností** (diagramy postupností - sequence diagrams) popisujú scenár priebehu určitej činnosti v systému,
- **diagramy spolupráce** (collaboration diagrams) zachytávajú komunikáciu spolupracujúcich objektov,
- **stavové diagramy** (statechart diagrams) popisujú dynamické chovanie objektu nebo systému, možné stavy a prechody medzi nimi,
- **diagramy aktivít** (activity diagrams) popisujú priebeh aktivít procesu či činnosti,
- **diagramy komponentov** (component diagrams) popisujú rozdelenie výsledného systému na funkčné celky (komponenty) a definujú náplň jednotlivých komponent,
- **diagramy nasadenia** (deployment diagrams) popisujú umiestenie funkčných celkov (komponentov) na výpočtové uzly informačného systému.

4.1 Charakteristika

Prípady použitia alebo Use Case sú písané z pohľadu zákazníka a podávajú prvú predstavu o rozsahu projektu. V tejto fáze analýzy sa ešte nezaoberáme technologickými aspektami riešenia a používame iba pojmy prirozeného jazyka a termíny z problémovej domény, aby sme čo najvernejšie a pre zákazníka čo najzrozumitelnejšie načrtli funkčný skelet systému.

Neriešime, či systém bude postavený na platforme .Net či J2EE, ani nediskutujeme so zákazníkom nad výhodami objektových či relačných databáz, ale zisťujeme, ktoré procesy má systém podporovať a akí užívatelia ho budú používať.

Use Case model, alebo taktiež model úžitkových činností alebo model prípadov použitia, sa začína tvoriť vo fáze Špecifikácie požiadavkov a dokončuje sa vo fáze Analýzy. Tento model na rozdiel od dokumentu Špecifikácia požiadavkov už popisuje chovanie systému, a to **požadované chovanie z hľadiska užívateľa**. Aj keď sa súčasne s modelom aplikácie popisuje okolie aplikácie a tým sa taktiež vymedzuje hranica aplikácie (čo do systému patrí a čo už nie), najväčší dôraz sa kladie hlavne na nájdenie všetkých úžitkových činností systému.

Čo je to modelovanie pomocou prípadov použitia (Use Case)?

- poskytuje pohľad na systém so zameraním na **správanie**, ako sa javí vonkajším používateľom
- **rozdeľuje funkcionalitu systému** na menšie časti, transakcie („Use Cases“ – prípady použitia), ktoré sú majú význam pre používateľov („Actors“ – účinkujúci)

Model prípadu použitia tvorí:

1. **diagram prípadu použitia** (Use Case Diagram)

Diagramom prípadu použitia je venovana samostatná kapitola.

2. **popis prípadu použitia** (Use Case Description)

Popis Use Casu je umiestňovaný do Note, (Note má každý model element – viď). Pozícia dôležitosti popisu u Use Casu je však v tomto modeli oveľa vyššia, ako je pozícia popisu u iných typov elementu v iných modeloch. Zatiaľčo u iných typov model elementov má popis (tj. Note - poznámka) skôr význam vysvetľujúci, v Use Case modelu patrí k stežejním. V Use Casu je získanie popisu oproti tomu jedným z hlavných cieľov stvárnenia Use Casu. Teda je to popis Use Casu, ktorý je podstatnou vlastnosťou Use Casu a na neho je zameraná hlavná pozornosť analytika.

Hlavným poslaním Use Case modelu je získať celkový zoznam všetkých úžitkových činností systému.

V tejto súvislosti sa samozrejme naskytuje otázka: Čo to je jedna úžitková činnosť systému – jeden Use Case?

Element Use Case - úžitková činnosť systému (prípad použitia)

Prvok Use Case - úžitková činnosť v systéme, špecifikuje jeden prvok funkcionality systému. Najlepšie si uvedomíme význam úžitkovej činnosti na základe požiadavkov na jej zrod.

Na počiatku jednej úžitkovej činnosti existuje jeden prvok informačnej nerovnováhy medzi okolím a našim systémom. Táto jedna nerovnováha je charakterizovaná ako (+ / -) alebo (- / +) vo vzťahu okolie versus informačný systém. Táto nerovnováha vedie k požiadavke na určitú funkcionalitu systému, ktorá tuto nerovnováhu vyrovnáva.

Celý systém sa v takom pohľade skladá zo samých úžitkových činností vyrovnávajúcich takéto deficity, tj. celková činnost sytému je neustále takéto vyrovnávanie informačných deficitov. Každá takáto činnosť charakterizuje určité použitie systému okolím. Z hľadiska funkcionality je jedna úžitková činnosť systému chápaná ako to, čo vedie v konečnom dôsledku ku splneniu určitej požadovanej funkcionalite, tj. vyrovnaniu deficitu informácií okolie - systém.

4.2 Odlišnosť Use Case od iných modelov

Zvláštne vlastnosti Use Case modelu vzhľadom k ostatným modelom UML

Úplnosť modelu v dvoch rovinách

Use Case model patrí k najdôležitejším modelom UML so zvláštnym postavením. Jeho zvláštnosť spočíva v tom, že pokiaľ máme k dispozícii Use Case model systému, tak vlastne držíme v ruke celý a úplný abstraktný popis systému.

Vyžaduje sa, aby Use Case model bol úplný. Vlastnosť „úplnosti“ je tu chápaná v dvoch rovinách:

- model je úplný, pretože v ňom **nechýba žiadna z úžitkových činností** (inak je model chápaný ako chybný model)

- model na rozdiel od iných niektorých modelov popisuje systém tak, že ktokoľvek, kto ho vlastní, je schopný **iba na základe tohoto modelu** navrhovať ďalšie modely a nakoniec systém naprogramovať

Čo tieto dve roviny úplnosti vlastne znamenajú prakticky? Je zrejmé, že Use Case model musí obsahovať všetky Use Casy, tj. je úplný podľa prvej požiadavky. Taktiež vieme, ako sa tohto druhu úplnosti dosahuje: metódou hierarchického rozkladu a hľadaním aktéra.

Druhá vlastnosť úplnosti znamená, že Use Case model je natoľko úplným a výstižným popisom systému, že v konečnom dôsledku je z neho možné vytvoriť ďalším modelovaním a kódovaním naprogramovaný systém. V Use Case modeli je totižto celý systém uschovaný ako jeho prvá úplná abstraktná extrakcia. Systém už de facto existuje vo svojej podobe Use Case modelu a „zostáva ho iba zrealizovať a naprogramovať".

Nie každý model sa vyznačuje touto „axiomatickou" vlastnosťou, že sa od neho dá systém v konečnom dôsledku naprogramovať. Napríklad stavový model nemá túto vlastnosť: pokiaľ by ste dostali do rúk stavový model, dostali by ste síce dosť výstižnú informáciu, ale ťažko by ste podľa nej tvorili kód. Stavový diagram sa používa ako vodítko pre vysvetlenie zložitých situácií v živote objektu. Naviac je technicky obtiažne a teda nie je účelné pomocou stavového diagramu popísať všetky stavy všetkých objektov v systéme.

Zrozumiteľnosť modelu

Okrem toho, že je Use Case model úplný a okrem toho, že podľa neho je možné naprogramovať celý systém, tak naviac je ešte dobre zrozumiteľný pre všetkých účastníkov projektu. Pretože model musí byť bezpodmienečne napísaný iba v pojmoch problémovej domény, rozumejú mu nielen vývojári, ale je zrozumiteľný aj pre užívateľov, ktorí už nemusia napríklad rozumieť Class modelu.

Veta: „Obsluha vyberie zo zoznamu Druhov tovaru Druh tovaru a dosadí ho do editovaného Tovaru." bude užívateľovi veľmi dobre zrozumiteľná a môže ju buď zamietnuť alebo odsúhlasiť. Vďaka týmto vlastnostiam sa Use Case model dostáva do zvláštnej výnimočnej pozície vzhľadom k ostatným modelom.

Rýchlosť tvorby modelu

V porovnaní s inými modelmi sa Use Case tvorí až prekvapivo rýchlo. Najväčší „zádrheľ“ pri písaní Use Case modelu nie je ani tak v hľadaní formulácií, tj. v hľadaní toho „ako to napísať“, ale v hľadaní toho „čo sa má vlastne napísať“. Inak povedané, a to si sami môžete overiť v praxi, pre písanie Use Case modelu sú najväčšou brzdou nedostatočné informácie v analýze a ich dopĺňanie.

Existujú dve základné okruhy nevedomostí pri tvorbe Use Case modelu:

Ako prvé sú nevedomosti zásadné, pretože sú to **nevedomosti z problémovej domény** ako také. Jednodoucho „nevieme, čo sa má vlastne v danej situácii urobiť“. Musíme samozrejme tieto nevedomosti odstrániť, napríklad pomocou konzultanta – experta na danú problémovú doménu.

Druhý typ nevedomosti sa netýka problematiky ako takej, tá je dobre známa, ale ide skôr o otázku, ako by sa najlepšie daný informačný systém v danej situácii použil a to „s čo najmenšou námahou jeho tvorby“. Jedná sa teda skôr o **hľadanie kompromisu, čo systém bude podporovať a čo už nie** (čo užívateľ prijme ako riešenie a čo už nie). Tieto otázky súvisia s otázkou nákladu a rovnako cenou za projekt, čo by si mal aj zákazník uvedomiť.

V každom prípade pri dobrej znalosti problémovej domény a pri rýchlom riešení situácie „čo v systéme bude a čo nie“, je rýchlosť tvorby Use Case modelu prekvapivo enormná. Veľká rýchlosť tvorby je spôsobená tým, že sa v Use Case modelu nezaťažujeme tým, **ako** sa má daná vec realizovať, ale popisujeme danú vec priamo. Popisujeme priamo vec bez implementačných podrobností a nestaráme sa o to, či to „bude chodiť alebo nie, prečo to nechodí alebo či to navrhnúť takto alebo takto“. To je až predmetom ďalšej fázy vývoja nazývanej Design.

4.3 Použitie Use Case vo firme

Use Case model sa s výhodou používa v nasledujúcich oblastiach:

Použitie Use Case modelu vedúcim projektu pre riadenie projektu (evidencia požiadaviek)

Často býva pre firmu veľmi výhodné, aby si viedla evidenciu požiadaviek na svoje produkty a tvorené systémy samostatne ako špeciálne vnútrofiremnú agendu. Vždy, keď je vznesená nejaká požiadavka od kohokoľvek na funkcionalitu systému (napríklad od užívateľa, od analytika, od vedúceho projektu apod.), tak sa táto požiadavka vo firme zaeviduje, a potom sa s ňou začne „nejako pracovať“. Vzniká tak nová logistická vnútrofiremná agenda ako riadená evidencia požiadaviek na SW produkty firmy.

Napríklad sa eviduje:

- kedy požiadavka vznikla,
- kto požiadavku vzniesol,
- dôležitosť požiadavky,
- v ktorej verzii sa bude realizovať,
- vzťah k modelom (hlavne k Use Case modelu, tj. ktorého Use Case sa týka apod.),
- ai.

Existujú aplikácie, ktoré túto a podobné evidencie požiadaviek podporujú. Pomocou riadenej evidencie požiadaviek vznikajú špecifikácie požiadaviek podľa predchádzajúceho odstavca systematicky a efektívnejšie. Pre firmu to samozrejme znamená zavedenie vyššej kvality do metód riadenia projektu.

Tento prístup pomocou softwarom riadenej evidencie požiadaviek ukazuje názorne, aká je vlastne poloha požiadaviek na systém z hľadiska modelovania. Špecifikácia požiadaviek je akýmsi „informačným backgroundom“ pre modelovania a požiadavky sa evidujú zvlášť od modelu.

Použitie Use Case modelu pre tvorbu dodatku k zmluvám so zákazníkom

Use Case model sa môže stať veľmi dobrým východiskom pre popis špecifikácie produktu pri spisovaní zmluvy so zákazníkom v tej časti zmluvy, kde je treba túto špecifikáciu popísať. Predsalen je dobré podchytiť v zmluve, čo má dodávaný systém vlastne robiť. Pokiaľ tak

neurobíme alebo tak urobíme iba povrchne, spísaná zmluva stojí viac menej na vode a po určitej dobe sa môžete s týmto zákazníkom škriepiť, ako to vlastne bolo v zmluve myslené.

Pochopiteľne nie je riešením „vziať Use Case model tak ako je a prilepiť ho k zmluve“. Use Case model by mal podliehať určitým pravidlám utajenia, pretože pokiaľ ho dostane do rúk konkurencia, dostane vlastne celý váš systém ako na dlani. Stačí však vziať tento model a pomocou určitých jeho pasáží je možné špecifikáciu produktu vytvoriť veľmi ľahko. Navyše užívateľ veľmi dobre rozumie slovníku Use Case modela a teda nemusíme príliš meniť v týchto pasážach formulácie.

Použitie Use Case modelu pre zoznámenie členov týmu s problémom

Všetky práce na systéme, či už sa jedná o vytvorenie následných modelov, návrhu, kódovania atď., sa veľmi urýchli vďaka takej jednoduchej skutočnosti, akou je tá, že sa ktokoľvek z týmu môže zoznámiť s Use Case modelom a prečítať si „o čo vlastne ide“. Dobrý a kvalitný Use Case model sa tak stáva v týme veľmi žiadaným dokumentom.

Použitie Use Case modelu pre tvorbu užívateľskej dokumentácie

Je zaujímavé, že na všetkých školeniach reagovali účastníci na otázku: „Aká je najnepríjemnejšia činnosť programátora?“ rovnakou odpoveďou: „Tvorba užívateľskej dokumentácie.“ Sám som to v niektorých softwarových firmách zažil osobne: Užívateľská dokumentácia sa tvorí ako úplne posledná (dokonca sa veľakrát tlačí v noci tesne pred odovzdaním produktu) a patrí k doslova najodpornejším činnostiam programátora.

Pritom táto otázka „kto tvorí užívateľskú dokumentáciu a ako sa tvorí“ je veľmi vypovedajúca o tom, ako sa vo firme tvorí software. Pokiaľ firma používa klasickú metódu priečneho rezu, kde jeden pracovník zastáva rolu analytika, designéra a programátora, tak musí písať aj užívateľskú dokumentáciu, pretože nie je nikdo iný, kto by túto problematiku poznal. Pritom tvorba užívateľskej dokumentácie by nemala spadať pod rolu programátora! Programátor má programovať!

Pritom pokiaľ je napísaný Use Case model, môže užívateľskú dokumentáciu písať ktokoľvek, a to dokonce už v priebehu vývoja systému, iba musí dodatočne dostať obrazovky.

Použitie Use Case modelu v testovaní

Vo veľmi mnohých firmách sa na otázku „ako testujete“ odpovedá „každý sám po sebe“. To však nie je testovanie. Programátor má mať odovzdanú prácu sám po sebe preverenú, o tom by nemalo byť pochýb. Táto dôsledná kontrola vlastných výtvorov nie je v žiadnom prípade procesom testovania! Testovanie je proces už nezávislý na tvorcovi a malo by sa konať systematicky podľa tzv. testovacích plánov.

V testoch sa pochopiteľne mimo iného skúma bezchybnosť fungovania systému ako v normálnych, tak v tzv. medzných situáciách (bezchybnosť v medzných situáciách sa ľudovo nazýva „blbovzdornosť systému“). Testovacie plány sú potrebné nielen z toho dôvodu, že je treba výsledok testu niekam zapísať, ale taktiež preto, že tester sa musí nejako zoznámiť s tým, čo sa vlastne má testovať. Pre tvorbu testovacích plánov pri skúmaní funkcionality má opäť nezastúpiteľnú rolu Use Case model. Testy existujú vo dvoch rovinách: testy analytickej funkcionality, pre ktoré vytvára testovacie plány analytické oddelenie a testy technológie (záťaž, kritické body systému, krajné medze naplnenia, rýchlost apod.), ktoré navrhuje designér. Pre tvorbu testu analytickej funkcionality je možné s výhodou použiť Use Case model.

Použitie Use Case modelu v obchodnom oddelení

Podobne, ako na otázku: „Aká je najnepríjemnejšia činnosť programátora“ všetci do jedného odpovedajú: „Tvorba užívateľskej dokumentácie“, tak na otázku: „Ktoré oddelenie nespadajúce pod vývoj je najotravnejším oddelením vo vašej firme?“ odpovedajú všetci zhodne: „Obchodné oddelenie a to najmä pred Invexom.“ Sú to pracovníci obchodného oddelenia, ktorí neustále dorážajú otázkami, čo ten náš systém vlastne vie robiť, čo majú ponúkať, aké má výhody atď. Pokiaľ má firma k dispozícii Use Case model systému, tak sa táto frekvencia otravnosti obchodníkov voči vývojárom výrazne zmenší. Najprv sú obchodníci odkázaní na tento dokument a až potom môžu analytici s obchodníky diskutovať o obchodných materiáloch. Pretože obchodníci sú už znalí problematiky, tak táto diskusia sa už zameria na to, o čom má skutočne byť, tj. aké sú hlavné výhody, čo v materiáloch zdôrazniť (a čo naopak nezdôrazňovať), apod. Samozrejme obchodník nevezme Use Case model, neskopíruje ho a

nepoloží ho ako ponukový dokument na stolík v stánku na veľtrhu. Určitý stupeň utajenia Use Case modelu musí byť dodržaný.

4.4 Diagram Use Case

Prekladaný ako diagram úžitkových činností, diagram prípadov použitia alebo diagram úžitkových prípadov apod. Zobrazuje účinkujúcich, jednotlivé prípady použitia a ich vzájomné vzťahy. Vnútorné časti prípadu použitia môžu byť bližšie určené textom a/alebo diagramami interakcií (Interaction Diagrams).

Diagram prípadu použitia popisuje systém ako hierarchický zoznam prípadov použitia, resp. úžitkových činností - Use Casov. Každý prípad použitia popisuje jeden spôsob použitia systému, popisuje teda jeho jednu požadovanú funkčnosť. Zachytáva funkcionalitu systému tak, ako sa javí jednotlivým používateľom. Je vyvíjaný analytikmi a doménovými expertmi.

Dokumenty Use Case diagramu sa tvoria len v analýze a to hneď v jej počiatku.

Účel:

- približuje prostredie v ktorom je systém umiestnený
- zachytáva požiadavky systému
- validuje architektúru systému
- je sprievodcom pri implementácii
- generuje jednotlivé prípady použitia

Prípady použitia sú zachytené vo vizuálnej a textovej podobe. Diagramy prípadu použitia nám poskytujú rýchlu predstavu o jednotlivých funkciách systému, ale presné postupy, rozširujúce a alternatívne scenáre musia byť zachytené v textovej forme. V jazyku UML je kodifikovaný iba diagram prípadu použitia, štruktúru dokumentu s textom prípadu použitia si musíme navrhnúť sami.

Komponentmi Use Case diagramu sú:

- **aktér** (actor) - užívateľská rola alebo spolupracujúci systém
- **hranice systému** (systém boundary) - vymedzenie hranice systému

- **prípad použitia** (use case) – dokumentácia udalosti, na ktorú musí systém reagovať
- **komunikácia** - väzba medzi aktérom a prípadom použitia (aktér komunikuje so systémom na danom prípade)

Pri vytváraní diagramu úžitkových činností je potrebné brať do úvahy, že existujú tzv. sekundárni aktéri, tj. užívateľské role alebo spolupracujúce systémy, pre ktoré nie je systém priamo určený, ale ktoré sú pre jeho činnosť potrebné. Prípady, kde chceme vyznačiť smer komunikácie, značíme orientovanými šípkami. Medzi prípadmi použitia môžu existovať vzťahy. Pokiaľ chceme explicitne vyjadriť fakt, že takýto vzťah existuje, používame stereotypy, štandardne << include >> - pokiaľ jeden prípad zahŕňa prípad iný (napr. autentifikáciu), alebo << extend >> - pokiaľ nejaký prípad rozširuje chovanie iného (je tu možnosť voľby).

Dva pohľady Use Case:

- **Business Use Case diagram** – uvažuje sa z pohľadu organizácie
- **Use Case diagram** – uvažuje sa z pohľadu klienta organizácie
- 4.5 Elementy Use Case

Prvok	Opis	Značenie
prípad použitia	postupnosť krokov, zahŕňajúca varianty, ktoré môže systém (alebo iná entita) vykonávať vo vzájomnom pôsobení s účinkujúcimi systému	
aktér / účinkujúci	celistvá množina rolí v ktorých môžu pri interakciách so systémom účinkujúci vystupovať	
hranice systému	reprezentuje hranice medzi fyzickým systémom a účinkujúcimi, ktorí s ním interagujú	

4.5.1 Aktér

Aktér nie je prvkom systému a reprezentuje kohokoľvek (alebo čokoľvek) mimo systém, kto nejako komunikuje so systémom a interaguje s ním. Základné pravidlo identifikácie aktéra znie: **Aktér môže iba prijímať alebo dávať do systému informácie.**

V predchádzajúcej vete je vyjadrený hlavný význam pojmu aktér v modeli: Aktér reprezentuje prvok okolia systému, ktorý buď informáce dostáva, alebo ich prijíma. Pokiaľ vyjmenujeme všetkých aktérov, tak vymenujeme celé okolie systému a vymedzíme tak jeho hranice. Nájdenie všetkých aktérov je podstatné: Pre tvorbu aplikácie je dôležité poznať, čo je ich súčasťou čo už ich súčasťou nie je. Aktérom nemusí byť iba živá osoba (obsluha, manažér, riaditeľ, ai.), ale taktiež iné systémy (videné ako externé), dátové prípojky apod. V tomto prípade vymedzenie Aktéra ukazuje, čo je predmetom riešenia aplikáce a čo nie, tj. čo sa považuje za externý prvok. Veľakrát sa táto hranice práve pri systéme zanedbáva a vznikajú tak nedorozumenia. Napríklad pokiaľ je dátová prípojka na port COM vymedzená ako aktér, znamená to, že táto prípojka nie je predmetom riešenia a s aplikáciou iba komunikuje.

Ako sa vyhľadáva aktér

Väčšinou pri konzultácii s užívateľom aplikácie (resp. pri predstave, kto ju bude používať) sa zisťuje:

- Čo je to za organizáciu, ktorá systém bude používať?
- Aké role majú zamestnanci pracujúci so systémom?
- Kto má záujem na splnení nejakej funkcionality systému?
- Kto bude systém administrovať?
- Kto má prospech z používania systému?
- Kto pristupuje k systému aby do neho vložil informáciu, zmenil informáciu alebo vymazal informáciu?
- Aké existujú zdroje dát z okolia systému?
- Akí existujú spotrebitelia dát z okolia systému?

Označenie Aktéra v modeli

Aktér sa v syntaxi Unified Modeling Language označuje „panáčikom“.

Obr.: Označenie Aktéra v UML

Popis aktéra v modeli

Súčasne s označením sa do modelu taktiež uvádza stručný a výstižný popis aktéra. Napríklad obsluha v banke za prepážkou, manažér, vedúci, pokladníčka, externý systém dávok dát, dátová prípojka atď.

Správne pomenovanie a navrhnutie aktérov:

- administrátor
- obchodný zástupca
- lektor
- systém SAP Processor pre prenos založených objednávok do SAPu

Nesprávne identifikovaní a pomenovaní aktéri:

- Vilém Málek (konkrétní osoba nie je aktér)
- osoba vyhľadávajúca údaje o objednávkách (príliš všeobecný popis, aktér iba s jednou zodpovednosťou)

4.5.2 Pripad pouzitia

Prvok Use Case / úžitková činnosť / prípad použitia špecifikuje jeden prvok funkcionality systému, ktorú využíva aktér. Celý systém sa v modeli Use Case skladá z úžitkových činností. Každý prvok úžitková činnosť charakterizuje určité použitie systému užívateľom (aktérom).

Z hľadiska funkcionality je jedna úžitková činnosť systému chapaná ako množina následných transakcií v systéme, ktoré v konečnom dôsledku vedú k splneniu požadovanej funkcionality.

Ako sa určuje úžitková činnosť / Use Case

Väčšinou pri konzultácii s užívateľom aplikácie (resp. pri predstave, ako sa aplikácia bude používať) sa zjštuje:

- Aké sú úlohy aplikácie (zo špecifikácie požiadaviek)?
- Bude niektorý z aktérov pridávať, odoberať alebo meniť informáciu v systéme?
- Bude niektorý z aktérov potrebovať informácie o zmenách v systéme?
- Aké zmeny v okolí systému budú viesť k prísunu informácií do systému?
- Aké činnosti sú potrebné k administrácii systému?

Označenie úžitkovej činnosti v modeli

Jeden Use Case - úžitková činnosť sa v syntaxi Unified Modeling Language označuje „elipsou“. Je odporúčané, aby z dôvodu prehľadnosti a rýchlej orientácie bolo súčasťou názvu prípadu aj jeho jednoznačné identifikačné číslo (UC001, UC002 a tak ďalej).

Obr.: Označenie Use Case v UML

Popis úžitkovej činnosti v modeli

V popise sa jeden Use Case uvedie stručnými vetami, čo sa od systému očakává, že bude vzhľadom k užívateľovi poskytovať za funkcionalitu v tejto jednej úžitkovej činnosti. V tejto fázi sa v modeli nepoužívajú pojmy ako objekty, triedy apod. Vystupujú tu iba pojmy z danej problémovej domény (majiteľ auta, rodné číslo, adresa atď.). Pri popise funkcionality sa nehľadajú kompetencie objektu, ale popisuje sa vlastnými slovami požadovaná funkcionalita. Pojmy z problémovej domény sa stávajú iba kandidátmi na objekty pre budúci model.

Poznámka: Skladba a obsah úžitkových činností musí byť v súlade s dokumentom Špecifikácie požiadavkov.

Názvy prípadov použitia musia byť dostatočne všeobecné, pritom jednoznačné a výstižné. Príklady názvu prípadu použitia:

- prihlásenie užívateľa do systému
- založenie novej pracovnej pozície
- pridanie zbožia do košíka
- úprava údajov o trvalom bydlisku uchádzača

Pokiaľ zákazník naše prípady použitia nepochopí a nechá ich spracovanie a realizáciu len na nás a našej agilite, potom sa takmer s určitosťou z vyvíjaného systému stáva nedotiahnutý projekt.

4.5.3 Hranice systému*

Hranice systému = rozdelenie zodpovednosti medzi systémom a okolím.
Prípady použitia napomáhajú nájdeniu hraničnej čiary medzi systémom a jeho okolím a informujú, aké sú väzby systému k okoliu. Pod pojmom okolie systému budeme rozumieť užívateľa, procesy a vzťahy, ktoré síce systém ovplyvňujú, ale nie sú jeho priamou súčasťou.
Predstavme si, že analyzujeme systém bezpečnostnej agentúry, určený na evidenciu a obsadenie pracovných pozícií. Na pohovore s manažérom náboru dostaneme informáciu, že niektoré poíicie vyžadujú vyhlásenie výberového konania. Výberové konanie musí byť schválené tromi členmi vedenia spoločnosti. Po skončení pohovoru napíšeme prípad použitia, ktorý popíše priebeh procesu pre vyhlásenie výberového konania. Na ďalšej schôdzke pri debate nad obsahom nového prípadu manažér upresní, že workflow pre vyhlásenie výberového konania už podporuje iný systém, s ktorým sú celkom spokojní. V našom systéme sa bude evidovať iba dátum vyhlásenia výberového konania. Evidencia dát výberového konania sa stane novým prípadom použitia v rámci systému, workflow pre vyhlásenie výberového konania vytesníme mimo hranice systému. Ak nie sú podchytené hranice systému, nemôže existovať zhoda medzi zadávateľom a realizátorom nad rozsahom systému.

Označenie hraníc systému v modeli

Hranice systému v modeli sa označujú obdĺžnikom.

Obr.: Označenie hraníc systému v UML

Popis hraníc systému v modeli

4.6 Vzťahy Use Case

Pri analýze reálneho systému však bežne identifikujeme desiatky i stovky prípadov použitia s množstvom závislostí, ktorých priame a naivné zobrazenie by vypovedajúcu hodnotu diagramu celkom ochromilo. Ako naznačiť, že určitá sekvencia kroku sa v mnohých Use Case opakuje? Ako rozšíriť nejaký prípad použitia o nepovinné chovanie? Čo s prípadmi použitia alebo aktérmi, zdieľajúcimi veľa spoločných rysov a zvyšujúcimi komplexnosť diagramu redundantnými väzbami? Ako sa vyrovnať s nudnými a monotónnymi prípadmi použitia pre správu číselníkov? Ako vidíte, podobných otázok nie je málo. Táto stať sa preto bude venovať mimo iného upresňujúcim vzťahom **include** a **extend** medzi prípadmi použitia a zameria sa taktiež na význam **generalizácie** elementu. Taktiež si popíšeme **asociáciu** ako jediný vzťah, ktorý existuje medzi aktérom a úžitkovou činnosťou.

Každý prípad použitia špecifikuje radu nejakých akcií a taktiež interakcie s okolím. Možné typy vzťahov medzi prípadmi použitia (pri generalizácii aj medzi aktérmi) sú iba tri:

- Generalization (Generalizácie)
- Extend (Rozšírenie)

- Include (Zahrnutie)

Možný typ vzťahu medzi prípadmi použitia a aktérom je iba jeden:

- Asociation (Asociácia)

Prvok	Opis	Značenie
asociácia	znamená, že prepojené časti diagramu spolu vzájomne komunikujú	
zovšeobecnenie	taxonomický vzťah medzi „všeobecnejším“ a „špeciálnejším“ prípadom použitia	
rozšírenie «extend»	vzťah medzi prípadmi použitia, kedy jeden prípad použitia môže rozširovať funkcionalitu iných prípadov použitia; tento druh vzťahu je nepovinný	
zahrnutie «include»	vzťah medzi dvoma prípadmi použitia, kedy je jeden prípad použitia obsiahnutý v jednom alebo viacerých ďalších prípadoch použitia	

4.6.2 Zovšeobecnenie

Zovšeobecnenie (generalizácia / dedenie) účinkujúceho A na účinkujúceho B znamená, že inštancia A môže komunikovať s rovnakými druhmi inštancií prípadov použitia ako inštancia B.

Generalizácia aktérov

Riešením problému s aktérmi, u ktorých sa prekrývajú zodpovednosti, je generalizácia (zovšeobecnenie). Vytvoríme nového aktéra, na ktorom budú delegované zdieľané zodpovednosti.

Príklad: Predstavme si situáciu, kde vytvárame HelpDesk – systém, v ktorom sa evidujú závady nahlásené zákazníkmi a priebeh ich riešenia. Pre každú nahlásenú závadu sa zakladá takzvaný sprievodný lístok (incident), ktorý nesie všetky informácie o konkrétnej chybe a priebehu riešenia. (Zákazníka, sériové číslo výrobku, či ide o záručnú opravu). Pri analýze je nám oznámené, že zakladať a upravovať vlastnosti incidentu môže operátorka na call centre a rešiteľ incidentu. Rozhodnutie o uzavretí incidentu môže však vydať len operátorka na call centre a riešenie zapisuje iba riešiteľ. Ide o typický prípad – dvaja aktéri, ktorých množiny zodpovedností majú spoločný prienik a súčasne každý z nich má svoje vlastné a so žiadnym iným aktérom nezdielateľné zodpovednosti. Riešením je vytvorenie (abstraktného) predka oboch aktérov s názvom zakladateľ a správca incidentu, ktorého zodpovednosťou bude vytváranie a editácia incidentu. Aktér je abstraktný, pretože sa jedná len o hypotetický analytický konstrukt, ktorý nemá žiadnu predlohu v realite. I keď to nie je tak časté, aj zovšeobecnení aktéri môžu byť konkrétni, to znamená že ide o plne autonomne a nezávislé role majúce svoj predobraz v problémovej doméne zákazníka. pri generalizácii vždy platí, že potomkovia majú ("dedia") všetky vlastnosti predka. Pri generalizácii aktérov potomkovia od predka dedia všetky väzby k prípadom použitia.

Generalizácia sa v UML zakresľuje nevyplnenou šípkou smerujúcou od potomka k predkovi.

Obr.: Generalizácia aktéra v UML

Generalizácia prípadu použitia

Rovnako ako zovšeobecňujeme aktérov, môžeme zovšeobecniť aj podobné prípady použitia. Typickými kandidátmi na zovšeobecnenie sú prípady použitia zamerané na správu číselníkov. Prípady použitia s názvami Založenie užívateľa, Úprava užívateľa a Editácia užívateľa môžu byť zovšeobecnené v prípade použitia Správa užívateľov. V zovšeobecnení môžu pokračovať a pre všetky prípady použitia s názvami Správa vytvoriť jedného predka - prípad použitia Správa systému, ktorý bude používať aktér administrátor. Výhoda je zrejmá – diagram prípadu použitia sa nielen sprehľadní, ale taktiež sú v ňom zoskupené súvisiace prípady použitia, a preto poskytne svojemu čitateľovi viac informácií.

Príklad: Ak budeme pokračovať analýzou HelpDesku vidíme, že môžeme napríklad zovšeobecniť prípady použitia Založenie incidentu a Editácia incidentu do prípadu použitia Správa incidentu.

Obr.: Generalizácia prípadu použitia v UML

4.6.3 Zahrnutie «include»

Zahrnutie / vkladanie povinných prípadov použitia – relácia << include >>

Viac prípadov použitia zdieľa rovnakú funkčnosť. Inak povedané, pri písaní textu prípadov použitia zistíme, že kroky scenára sa vo viacerých prípadoch použitia opakujú. Spoločnú časť prípadov použitia môžeme v UML vyňať do samostatného prípadu použitia a ostatné prípady použitia sa na nej budú odkazovať pomocou relácie << include >>. **Relácia Include je špeciálnym typom závislosti** medzi elementami – prípad použitia so spoločnou logikou je nazývaný dodávateľský prípad použitia a prípady, ktoré sa na neho odkazujú (a zahŕňajú jeho funkcionalitu), sú klientské prípady použitia. Musí platiť, že scenár dodávateľského prípadu použitia je zahrnutý do klientského prípadu použitia povinne a vždy. Bez dodávateľského prípadu použitia je klientský prípad použitia neúplný a nie je možné ho realizovať.

Dodávateľský prípad použitia môže byť samostatným prípadom použitia, ktorý môžu aktéri používať i samostatne, alebo môže obsahovať iba fragment chovania, ktorý nadobúda významu jen v spojení s chovaním definovanom v klientskom prípade použitia.

Príklad: Predstavme si, že v HelpDesku pred akoukoľvek operáciou s incidentom musí prísť k zobrazeniu detailu incidentu. Aby sme nepísali stále dokola postup, ako uživateľ otvorí detail incidentu, vytvoríme samostatný prípad použitia Otvorenie detailu incidentu a ostatné prípady použitia vyjadrujúce operácie nad incidentom s ním budú prepojené relaciou << include >>.

Obr.: Otvorenie detailu incidentu

4.6.4 Rozšírenie «extend»

Vkladanie nepovinných prípadov použitia – relácia << extend >>

Reláciu Extend použijeme pre vyjadrenie faktu, že prípad použitia môže byť **nepovinne** rozšírený o chovanie z iného prípadu použitia. Slovo nepovinne zdôrazňujem, pretože narozdiel od relácie << include >> je klientský prípad použitia úplný i bez dodávateľského prípadu použitia a jazyk UML dokonca vyžaduje, aby bol jeho základný scenár celkom oslobodený od scenára prípadu použitia, s ktorým je spojený väzbou << extend >>. Klientský prípad použitia len deklaruje takzvané body rozšírenia (Extension points), ktoré si môžete predstaviť ako sloty, do ktorých je zasúvané chovanie (Vkladané segmenty - Insertion segments) dodávateľských prípadov použitia. Inými slovami, bod rozšírenia je vlastnosť prípadov použitia, ktorá dovoľuje identifikovať bod, kde môže byť správanie sa prípadu použitia rozšírené o ďalšie prvky iného (rozširujúceho) prípadu použitia.

Počet vkladaných segmentov v dodávateľskom prípade musí byť zhodný s počtom bodov rozšírenia, o ktoré má v klientskom elemente záujem. Informáciu, aké body rozšírenia dodávateľský prípad použitia rozširuje, nesie samotná relácia Extend. Ak nie sú pri relácii <<

extend >> žiadné body rozšírenia zadané, predpokladá sa, že dodávateľský prípad použitia rozširuje všetky body rozšírenia.

Príklad: Predstavme si, že v Helpdesku chceme VIP zákazníkom odoslať zdarma informatívne SMS vždy, keď nimi nahlásený incident byl uzatvorený. Pri prípadoch použitia Uzatvorenie incidentu ponúkneme bod rozšírenia "Zákazník je VIP". Nový prípad použitia Odoslanie SMS VIP zákazníkovi prepojíme reláciou << extend >> s prípadom použitia Uzatvorenie incidentu a vytvoríme v ňom vkladaný segment pre odoslanie SMS.

Obr.: Vkladanie nepovinných prípadov použitia

4.6.5 Hierarchia v Use Case

Medzi jednotlivými Use Casy – úžitkovými činnosťami sa môžu vyskytovať jednosmerné interakcie – vzťahy medzi Use Casy. Tieto vzťahy sú v UML chápané ako ďalšie model elementy a ich vizuálnym elementom je šipka.

Jednou z týchto interakcií je už zavedená hierarchia od elementu use case „hore“ k elementu „dole“. Tento vzťah slúži k prehľadnému rozkladu úžitkových činností v hierarchickom členení. V diagrame najlepšie znázorníme vzťah hierarchie šipkou bez žiadneho ďalšieho popisu.

Obr.: Rozklad hierarchie Use Casu

4.7 Dokumentácia

Pre každý prípad použitia je vytvorené množstvo dokumentov opisujúcich konkrétne udalosti. Dokumentácia obsahuje podrobnosti, čo musí systém umožniť účinkujúcemu počas vykonania tohto prípadu použitia.

Typický obsah:

- ako sa prípad použitia spustí a ukončí,
- štandardný sled udalostí,
- alternatívny sled udalostí,
- sled udalostí počas výnimiek (chybových stavov).

4.8 Typy problémov

Na počiatku všetkého je problém

V tejto stati si popíšeme problémy, ktoré pred nami vyvstanú pri vytváraní zložitých projektov.

1. **Pri analýze bolo identifikovaných množstvo aktérov, ktorých zodpovednosti sa prekrývajú.** Nepríjemným dôsledkom tejto situácie je nárast relačných priradení pri každom prípade použitia, ktoré vyjadrujú stále to isté – aktér xy používa túto funkciu. Potrebovali by sme nájsť spôsob, ako zdieľané zodpovednosti aktérov vyčleniť a naviazať na jednu spoločnú entitu ("pseudoaktéra"), ktorej zodpovednosti by boli

všetkými aktérmi implicitne prejímány bez explicitného zachytenia priradenia každého aktéra ku každému prípadu použitia.

2. Podobný problém predstavuje **nárast prípadu použitia**. Často nájdeme prípady použitia s mnohými spoločnými rysmi, ale drobné odchýlky v ich účeli či realizácii nám zabraňujú vytvoriť jediný invariantný prípad zahrňujúci všetky variácie. Jeden aktér často používa všetky podobné prípady použitia a preto v systéme opäť existuje množstvo nadbytočných väzieb priradení, ktoré sťažujú pochopenie relevancie vzťahov. Riešením by bolo vytvorenie jedného všeobecnejšieho prípadu použitia, ku ktorému by sa aktér vzťahoval a pod ktorým by boli subsumované všetky jeho varianty, takže by z diagramu bolo jasné, že priradenie aktéra k všeobecnému prípadu použitia značí aj priradenie ku každej variante všeobecného prípadu použitia.

3. V scenároch inak nezávislých prípadov použitia zistíme v každom z nich rovnakú sekvenciu kroku – **rovnaké chovanie systému**. Agresívne pravidlo číslo jedna v ľubovolnej fáze návrhu a vývoja systému teda je: "Akákoľvek duplicita musí byť ihneď odstránená za použitia akéhokoľvek analytického arzenálu".

4. Prípad použitia môže byť voliteľne rozširovaný pri splnení určitých podmienok. Chceme vizuálne zachytiť, že do scenára môžu byť vložené nepovinne ďalšie kroky, ktoré reagujú na špecifické podmienky konkrétnej inštancie prípadu použitia.

5. Snáď ani nemusíme dodávať, že pri vytváraní diagramu prípadu použitia zložitejších systémov sa stretávame so všetkými zmienenými problémami naraz.
6. 4.9 Zásady pri tvorbe Use Case modelu
7. **Dôležité zásady pri tvorbe Use Case modelu**
8. Majte vždy na pamäti, že hlavným cieľom Use Case modelu je nájsť všetky (úplne všetky) úžitkové činnosti ako listy hierarchického stromu modelu a previesť potom ich konzistentný popis. Ostatné pracovné postupy zavedené v tvorbe Use Case modelu, ako

sú vyhľadávanie a tvorba hierarchie Use Casu, určenie prvku okolia systému apod., sú **iba pomocnými metódami** (i keď niekedy nevyhnutnými) na ceste jedinému cieľu: Získať všetky Use Casy ako listy hierarchie.

9.

10. **Konzistencia Use Case modelu a „zlaté Use Casy“**
11. Pri tvorbe Use Case modelu postupujeme odhora dolu postupným logickým rozkladom. Pokiaľ pridávame nový odhalený Use Case, tak ako analytici musíme dobre sledovať **konzistenciu modelu**. Jeden nový Use Case so sebou prináša potrebu zaviesť jeden alebo viac ďalších iných Use Casov v iných častiach modelu (doslova na druhom konci systému), ktoré sú nutné k tomu, aby vôbec tento náš nový Use Case mohol fungovať. Problém je v tom, že pokiaľ v novom Use Case existuje nejaká informácia, s ktorou musí nový Use Case pracovať, tak pochopiteľne musí existovať iný Use Case, ktorý s touto informáciou nejako narába „z druhej strany“, ktorý nám túto informáciu nejako pripravuje pre novo zavádzaný Use Case.
12. Napríklad keď potrebujeme pre faktúru vybrať Odberateľa, tak musí existovať jeden a viac Use Casov, ktoré pracujú s Odberateľmi. Činnosti musia nejako tento zoznam pripraviť so všetkým nevyhnutným komfortom, ako editácia prvku apod. Pokiaľ na tieto Use Case pozabudneme, tak z pochopiteľných dôvodov nebude systém môcť správne fungovať, pretože zoznam nebude použiteľný.
13. Pri dobrom sledovaní konzistencie sa začnú objavovať ďalšie a ďalšie Use Casy potrebné pre obsluhu novo pridávaných Use Casov a model tak „utešene rastie“. Z hľadiska významu Use Case (nie z hľadiska nevyhnutnej funkčnosti) môžeme teda rozdeliť Use Casy na zlaté Use Casy a „obslužné“ Use Casy. Zlaté Use Casy sú tie, pre ktoré systém vôbec existuje a sú dôležité z obchodného hľadiska. Obslužné Use Casy musia byť zavedené len z toho dôvodu, aby zlaté Use Casy mohly fungovať. Toto rozdelenie je však iba z hľadiska pohľadu obchodného významu funkcionality, nie však z hľadiska samotnej funkcionality. Aby systém dobre fungoval, musí obsahovať všetky Use Casy, ako všetky zlaté, tak všetky obslužné.
14. Samozrejme, rovnako ako vo všetkých oblastiach ľudskej činnosti, aj tu platí obdoba akéhosi všeobecne platného Murphyho zákona, ktorý môžeme v súvislosti s týmto postupom vypozorovať:
15. **Nepodstatných činností, ktoré sú však nevyhnutné pre to, aby mohly fungovať veci podstatné, je viac než 99% zo všetkých činností dohromady.**

16. Naozaj, väčšina Use Casov má povahu „okolných podporných “tančekov“ potrebných pre to, aby niekoľko málo Use Casov, mohlo fungovať. Pri tvorbe Use Case modelu zahajujeme vždy naše úvahy pri zlatých Use Casoch. Tieto Use Casy začnú plodiť pomocné obslužné Use Casy, ktoré však taktiež musia v systéme existovať. Postup je teda taký, že vytvoríme nejaký „zlatý“ Use Case a postupne od neho „odbiehame“ pre tvorbu nových obslužných Use Casov.

4.10 Tipy pri modelovaní prípadov použitia

1. Uistite sa, že každý prípad použitia obsahuje **podstatnú časť systému**, ktorá je **ľahko pochopiteľná** doménovým expertom aj programátorom.

2. Ak definujete prípad použitia v textovej forme, používajte v ňom **priliehavé a jednoznačné podstatné mená a slovesá**, aby sa dali jednoduchšie odvodiť správy pre interakčné diagramy.

   - Ak používate systém podporujúci prípady použitia, začnite tvorbou prípadov použitia a z nich odvoďte štrukturálne modely a modely správania.
   - Ak nepoužívate systém podporujúci prípady použitia, uistite sa, že Vaše prípady použitia sú v súlade s Vašimi štrukturálnymi modelmi a modelmi správania.

3. Berte do úvahy **všeobecné funkcie**, ktoré sú využívané viacerými prípadmi použitia:

   - ak je funkcia **povinná**, použite **« include »**,
   - ak je rodičovský prípad použitia úplný a práve navrhovaná funkcia je **nepovinná**, zvážte použitie **« extend »**.

4. Diagram prípadu použitia by mal obsahovať:

   - len také prípady použitia, ktoré sú na **rovnakej úrovni abstrakcie**,
   - len takých účinkujúcich, ktorí sú **povinní**.

5. Väčšie množstvá prípadov použitia je lepšie organizovať do **balíčkov**.

4.11.1 Univerzitný elektronický register*

Univerzitný elektronický register

Univerzita chce zaviesť elektronický register:

- Tajomník pripraví plán výučby pre nastávajúci semester:
  - jeden predmet smie ponúkať viacero kurzov.
- Študenti si vyberajú 4 hlavné predmety a 2 alternatívne.
- Po registrácii študenta na semester, je notifikovaný poplatkový systém, takže študenti môžu byť v každom semestri požiadaní o platbu.
- Študenti môžu počas vopred stanovenej lehoty od registrácie pridávať alebo odoberať svoje predmety.
- Vyučujúci dostanú od systému rozpis kurzov predmetov s počtami prihlásených študentov.
- Používatelia systému dostanú heslá, ktoré musia používať pri prihlasovaní.

Účinkujúci je niekto alebo niečo, čo musí interagovať s vyvíjaným systémom.

Prípad použitia je vzor správania, ktorý bude systém ponúkať.

- Každý prípad použitia je postupnosť vzájomne súvisiacich akcií, ktoré vykonávajú účinkujúci a systém vo vzájomnom dialógu.

Požiadavky účinkujúcich je nutné vopred preskúmať.

- Tajomník – udržuje plán výučby.
- Vyučujúci – požaduje rozpis predmetov.

- Študent – spravuje (v rámci možností) plán svojho štúdia.
- Poplatkový systém – získava informácie nutné pre stanovenie výšky poplatkov.

Prípad použitia **„Pripraviť plán výučby“**:

- Tento prípad použitia začne po prihlásení sa Tajomníka do systému (zadaním hesla a prihlasovacieho mena).
- Systém overí platnosť hesla (E-1) a vyzve Tajomníka na označenie semestra alebo budúceho semestra (E-2).
- Tajomník zvolí požadovaný semester.
- Systém vyzve Tajomníka na výber požadovanej aktivity: Pridanie, Zmazanie, Kontrolu alebo Ukončenie.
- Ak je označená aktivita Pridanie, vykoná sa poddiagram S-1: Pridanie predmetu.
- Ak je označená aktivita Zmazanie, vykoná sa poddiagram S-2: Zmazanie predmetu.
- Ak je označená aktivita Kontrola, vykoná sa poddiagram S-3: Kontrola predmetov.
- Ak je označená aktivita Ukončenie, prípad použitia skončí.
- …

Diagramy prípadov použitia znázorňujú vzťahy medzi účinkujúcimi a prípadmi použitia.

Počas vypracovania dokumentácie k týmto prípadom použitia, môžu byť medzi nimi odhalené rôzne vzťahy:

- vzťah „zahŕňa“ («include») znázorňuje správanie, ktoré je spoločné pre jeden alebo viacero prípadov použitia,
- vzťah „rozširuje“ («extends») znázorňuje voliteľné správanie.

4.11.2 Systém ľudských zdrojov*

Účinkujúci:

- zamestnanec, databáza zamestnancov, systém zdravotného poistenia, systém sociálneho poistenia

Počiatočné podmienky:

- Zamestnanec sa prihlásil do systému a označil možnosť „aktualizovať odvody“.

Štandardný sled udalostí:

- Systém získa údaje o konte zamestnanca v databáze zamestnancov.
- Systém vyzve zamestnanca na výber typu programu lekárskej starostlivosti; zahŕňa Aktualizovať program lekárskej starostlivosti.
- Systém vyzve zamestnanca na výber typu programu zubnej starostlivosti; zahŕňa Aktualizovať program zubnej starostlivosti.
- …

Alternatívny sled udalostí:

- Ak nie je zvolený program zdravotnej starostlivosti v oblasti pôsobenia zamestnanca k dispozícii, zamestnanec je informovaný a vyzvaný na výber iného programu.
- ...

4.11.3 Opis prípadu použitia „Zmeniť let“*

Účinkujúci:

- cestujúci, databáza klientov, rezervačný systém aerolínií.

Počiatočné podmienky:

- cestujúci sa prihlásil do systému a zvolil možnosť „zmeniť trasu letu“.

Štandardný sled udalostí:

- Systém získa informácie o konte cestujúceho trase letu z príslušných databáz.

- Systém vyzve cestujúceho na voľbu segmentu trasy, ktorú si želá zmeniť; cestujúci označí segment.

- Systém požiada cestujúceho o zadanie nových informácií o odlete a prílete; cestujúci poskytne požadované informácie.

- Ak je linka voľná, tak…

- …

- Systém zobrazí zhrnutie o transakcii.

Alternatívny sled udalostí:

- Ak nie sú k dispozícii žiadne lety, tak…

4.11.4 Priklad* (pouzitelny?)

4.11.5 Priklad* (pouzitelny?)

Úvodní informace

Každý případ užití je identifikován svým názvem a pořadovým číslem a je umístěn v samostatném dokumentu. Název i pořadové číslo případu užití se samozřejmě shoduje s údaji v diagramu případů užití. Z důvodu přehlednosti a snadného vytváření rychlých odkazů mezi dokumenty doporučuji, aby i název souboru obsahoval název a pořadové číslo případu užití. Sám v názvech dokumentů nepoužívám diakritiku a mezery mezi slovy nahrazuji podtržítky.

Název případu užití:
UC001 Předání zásilky ke zpracování řešiteli
Název souboru s textovým popisem případu užití
UC001_ Predani_zasilky_ke_zpracovani_ resiteli

Ihned za názvem by měla být informace o stavu dokumentu. Pro případy užití jsou většinou důležité jen 2 stavy - stav “Na dokumentu se pracuje“ a stav “Schválen zákazníkem“.

Stav dokumentu:
Schválen zákazníkem

Každý dokument by měl obsahovat stručnou historii provedených úprav. Eviduje se datum změny, nová verze dokumentu, autor změny a stručný popis změny.

Datum změny	Verze dokumentu	Autor	Popis změny
1.7.2004	1.0	René Stein	Počáteční verze
30.7.2004	1.1	René Stein	Zanesen požadavek na notifikace
2.8.2004	1.2	René Stein	Dokument schválen zákazníkem

Případ užití je uvozen stručným popisem za účelem rychlého zorientování čtenáře v řešené problematice.

Stručný popis:
Po přijetí zásilky a vytvoření její elektronické podoby (karty) je vybrán řešitel zásilky. Vybrat nového řešitele může také již přiřazený řešitel v případě, že o řešení zásilky sám nemá zájem.

Abychom se při designu a vývoji systému soustředili na nejčastěji využívané funkce systému a zbytečně nealokovali finanční a lidské zdroje na technologicky precizní řešení funkcí, jež jsou ale pak využívány jen v přestupném roce, zjišťujeme a evidujeme, jak často by měl být dle zadavatele scénář případu užití realizován.

Frekvence užití
30x denně

Následuje seznam aktorů i s popisem jejich role v případu užití.

- Zakladatel karty - osoba, která mj. poprvé přiděluje kartu zásilky řešiteli.
- Řešitel - osoba, které bylo přiděleno řešení zásilky a která řešení deleguje na jinou osobu.

Tok událostí

Nejdůležitější je v případech užití tok událostí neboli podrobný postup, jakým je dosaženo hlavního funkčního záměru případu užití. V šabloně rozlišuji tři typy toku událostí.

- Základní tok událostí – v základním toku událostí je umístěn maximálně stručný a přitom kompletní postup při realizaci případu užití. V základním toku se nesmí objevit žádné větvení toku ani žádné podrobné informace, jak je daného postupu dosaženo. Separací alternativního a rozšiřujícího chování nepřetížíme hlavní tok událostí pozornost odvádějícími podrobnostmi ani rozptylujícími variacemi chování. Tok případů užití, s nimiž je spojen popisovaný případ užití relací <<Include>> nebo <<Extend>>, může být do hlavního toku vložen zapsáním slova <<Include>> nebo <<Extend>> následovaném plným názvem inkriminovaného případu užití.
- Alternativní tok událostí – v alternativním toku jsou zapsány všechny odchylky od lineárního toku událostí (podmíněné kroky postupu, opakování kroků – cykly)
- Rozšiřující tok událostí – rozšiřující tok obsahuje všechny relevantní podrobnosti k bodům postupu v hlavním toku.

Základní tok událostí

1. <<Include>> UC002 –Zobrazení seznamu karet zásilek.
2. Uživatel vybere kartu zásilky.
3. Uživatel dá příkaz k zobrazení karty zásilky.
4. Systém zobrazí požadovanou kartu zásilky.
5. Uživatel vybere nového řešitele.
6. Uživatel potvrdí přiřazení řešitele.
7. Systém uloží informaci o novém řešiteli.
8. Systém odešle novému řešiteli email s notifikací o přiřazení úkolu.

Alternativní tok událostí

Body 3,4

- Uživatel může přerušit proces změny řešitele. Žádná změna nebude uložena Konec případu užití.

Rozšiřující tok událostí

Bod 7

1. V emailu musí být zahrnuty všechny evidované informace o odesílateli zásilky.

Doplňující informace

Již v případech užití můžeme začít navrhovat aplikační práva pro jednotlivé aktory v systému.

Přístupová práva

Systémová role	**Přístupová práva**
Recepční Řešitel	Právo na zobrazení karty zásilky Právo na přidělení/změnu řešitele

Zákazník při analytických pohovorech také často vznáší předběžné požadavky na očekávané odezvy systému K evidenci zadavatelem preferované doby odezvy slouží sekce Doba odezvy.

Doba odezvy
Otevření karty zásilky – 5 sekund

Důležité operace v systému by měly být ukládány do protokolu o činnosti systému, aby bylo snadné v případě nutnosti dohledat, kdy jaká změna nastala a kdo je původcem změny.

Požadavky na logování
Systém zaeviduje do logu datum a čas změny, číslo zásilky, identifikátor uživatele, který změnu provedl a identifikátor nového i původního řešitele.

Jestliže zadavatel vznese ještě další speciální požadavky na průběh případu užití, tak je zapíšeme do sekce s názvem Ostatní požadavky.

Kritické podmínky, které musejí být splněny, aby mohl být případ užití spuštěn, jsou v sekci Podmínky před spuštěním.

Podmínky před spuštěním
Karta zásilky není označena příznakem „Nevyžaduje řešení“.

Dokončení instance scénáře případu užití znamená, že v systému proběhly všechny změny, jejichž seznam je v sekci Stav po ukončení.

Stav po ukončení

- U zásilky změněn řešitel.
- Odeslán email novému řešiteli.

Čerpá-li náš případ užití informace z jiného dokumentu, například z rámcového zadání vytvořeného zákazníkem, uvedeme v dokumentu seznam použitých zdrojů.

Požadavky uživatelů relevantní pro funkcionalitu
Návrh systému Evidence korespondence – kapitola Řešení zásilek na straně 25

Všechny nejasnosti a sporné body vyžadující konzultaci se zadavatelem, dalším analytikem nebo projektovým vedoucím si poznamenáme v poslední sekci Poznámky a problémy.

Poznámky a problémy
Bude se držet historie řešitelů?

Po schválení finálního znění textových specifikací všech případů užití zákazníkem přistoupíme většinou k vytváření diagramu tříd. Diagram tříd bude hlavním tématem následujících částí seriálu.

Register použitých pojmov*

Agent	An independent software module capable of registering its interests in an ontologically modeled environment and acting in a manner that brings the environment closer to the goal state when those interests are satisfied.
Agent Engine	A software application or module capable of hosting one or more software agents. The application must provide the means for the agents to interact with modeled environments and typically offers specialized facilities such as those for expert system rule processing or case-based reasoning.
ARES Domain Framework	A generic ontology template, developed internally by CDM Technologies, which can be customized for a particular application domain.
Artificial Intelligence	The branch of computer science concerned with developing software capable of performing activities normally thought of as requiring (human) intelligence.
Application interface	The structured internal software mechanisms by which one software component may utilize the features of another.
Attribute	An elemental property of object (object modeling) or entity (relational modeling) such as weight, height, and name.
Automated Information System	A client server software system in which multiple users and applications may access shared data.
Blackboard Interaction	A pattern of interaction between software components in which multiple software modules known as agents interact with a shared representation of an environment without direct knowledge of each other.
Build Environment	The software development environment in which a specific software product is created and maintained.
Case Base	A persistent repository for cases to be used in case-based reasoning.

Case-Based Reasoning	An artificial intelligence technique which seeks to develop courses of action by comparing the current problem 'case' with a collection of past cases to find a similar problem and adapt its previously found solution to the current problem.
Command Line Interface	A textual interface based on the underlying operating system of a computational platform, such as Unix or DOS.
Composite Score	A semantic equivalence score that combines multiple individually derived agent scores into a singe number using a weighted average.
Database Instance	A specific operational copy of a logical database that is executing within a RDBMS to serve a user community.
Data Management Component	The IMT component responsible for populating the IMT Information Server and Agent Engine with information derived from externally managed data elements.
Distributable	Software elements that can function together when executed on multiple networked processors.
Enterprise	An organizational entity.
Enterprise Data Model	The structure model of the enterprise information.
Enterprise Environment	The networked computing platform of an enterprise.
Enterprise Information	The information that must be shared across different user communities within an enterprise environment.
Enterprise Interoperability	The ability of the information systems within a domain to seamlessly exchange data without requiring human interaction.
Entity	A thing within a relational model such as a person or asset.
Event Notification	The act of a software element notifying an interfacing element that a previously registered interest has occurred.
Field	A standard relational meta-data element for representing the discrete properties of entities within a domain.
Field Name	A standard relational meta-data element that uniquely identifies a specific field.
Field Description	A standard relational meta-data element that provides a human readable description of a specific field.
Foreign Key	A standard relational meta-data element that designates a field or set of fields as uniquely identifying another domain entity.
Hyponym	A word that denotes a subcategory of a more general class.
Information Caching	The local storage of remotely obtained information for improved efficiency and support of mobile computing.
Information Server	The software component responsible for managing the objects representing the problem domain.
Instance Data	Individual records within a relational database.
Instance-Data Mapping	The mapping of equivalent or related instance data elements within a relational database.
Interface Element	A semantic chunk represented by a line item within an IRDD.
Interface Requirements Design Document	A formal document specifying the semantic data elements to be exchanged between two information systems.
Java	A popular general-purpose platform-independent software development language.

Java Database Connectivity (JDBC)	A popular standard interface for a software element to access data management functions within an RDBMS.
Knowledge Instance Model	A set of interlinked instances that together define the knowledge of domains modeled using the knowledge-operational split pattern.
Knowledge-Operational Split	An ontological modeling pattern in which instances representing operational entities such as actions (or in this case, tables) are kept fundamentally distinct from those that represent knowledge that would, for instance, define the type of action or table.
Mapping Problem	A set of relational elements that needs to be semantically mapped to another set of relational elements.
Master Model	The USTC-developed enterprise data model for the DoD
Meta-Data	Data about data. In the IMT domain, the meta-data consists of the data associated with a relational schema.
Ontology	A structured information model of a domain capable of supporting reasoning by human users and software agents.
Persistence	The capability for the virtual representations of domain entities to exist (on disk) beyond the execution of the associated software application.
Persistence Repository	The disk storage location.
Primary Key	A standard relational meta-data element that designates a field or set of fields as uniquely identifying the entity to which the fields belong.
Problem Domain	The virtual representation of a real-world problem.
Relational Schema	The meta-data elements that describe the structures and constraints of the data representing a domain.
Reference Data Management	The enterprise process of managing the creation and maintenance of data that is commonly shared throughout the enterprise and relatively static in nature.
Relational Database	A domain represented by a relational model.
Relational Database Management System	A system designed to manage executable databases for access and manipulation by client applications.
Rich Client Application	A full-featured software application that must be installed on individual user machines to be used. See *Thin Client Application*.
Segmentation	A DII COE mechanism for encapsulating software elements in a standard fashion such that they can be used by other elements on the same platform.
Semantic Equivalence Score	A number between 0 and 1 that indicates the degree to which two modeled entities can be considered the same thing.
Semantic Equivalence Technique	An algorithm or computational method to determine how close two modeled entities are to being the same thing.
Semantic Map	A data structure that correlates semantic equivalences between relational entities (in the case of this project).
Service-Oriented Interface	Software module interface that specifies the format for exchanging data in a service-oriented architecture.
Similarity Metric	A similarity score generated using a specific semantic equivalence technique.

Subscription Profile	The set of ontology described interests (e.g., vessels whose length extends 100 yards) a specific software agent has registered with domain Information Server.
Table	A standard relational meta-data element that uniquely identifies a class of entities within the domain.
Table Management and Distribution System	A USTC-developed system for managing enterprise reference data.
Table Description	A standard relational meta-data element that uniquely identifies a class of entities within the domain.
Table Field	A standard relational meta-data element that uniquely identifies a class of entities within the domain.
Table Name	A standard relational meta-data element that uniquely identifies a class of entities within the domain.
Thin Client Application	A software application housed on a single web server but accessible to a community of users, without requiring its installation on the user machines, via a web browser. See *Rich Client Application*.
Trigram	A three letter sequence of characters within a word or phrase.
Unified Modeling Language	A standard graphical language for the specification of software systems and domain ontologies.
Use Case	A description of a domain task in which users will employ a software application.
Web Service Interface	A software interface implement using the W3C web service standard that provides for loose coupling amongst software components distributed across the Internet as well as a limited means for discovery.
WordNet	A publicly available linguistic ontology used to identify occurrences of terminological variations arising from conceptual abstraction.
World-Wide Port System	A USTC system for managing the flow of military cargo into and out of military ports, both CONUS and OCONUS.
XML	eXtensible Markup Language.
XML Meta-Data Interchange	An extensible XML-based language for the specification of model data.

Abstraction	Abstrakce
Action	Akce
ActionSequence	Sekvence Akcí
ActionState	Akce-stav
ActivityGraph	Graf aktivit
Actor	Aktér
Argument	Argument
Association	Asociace

AssociationClass	Asociativní třída
AssociationEnd	Konec Asociace
AssociationEndRole	Role Konce Asociace
AssociationRole	Role Asociace
Attribute	Atribut
AttributeLink	Spojení atributu
BehavioralFeature	Rys chování
Binding	Vazba
CallAction	Akce volání
CallEvent	Událost volání
Class	Třída
Classifier	Klasifikátor
ClassifierInState	Klasifikátor ve stavu
ClassifierRole	Role Klasifikátoru
Collaboration	Spolupráce
Comment	Komentář
Component	Komponenta

ComponentInstance	Instance komponenty
CompositeState	Kompozitní stav
Constraint	Omezení
CreateAction	Akce tvorby
Data type	Datový typ
DataValue	Datová hodnota
Dependency	Závislost
DestroyAction	Akce zrušení
Element	Prvek
ElementResidence	Umístění prvku
Event	Událost
Exception	Výjimka
Extend	Rozšíření
ExtensionPoint	Bod rozšíření

Feature	Rys (ve smyslu „povaha,
FinalState	Finální stav
Flow	Tok
Generalization	Generalizace
Guard	Průvodce
ChangeEvent	Událost změny
Include	Začlenění
Instance	Instance
Interaction	Interakce
Interface	Interface
Link	Spojení Instancí
LinkEnd	Konec Spojení Instancí
LinkObject	Spojení Instancí - Objekt
Message	Zpráva
Method	Metoda
Model	Model
ModelElement	Prvek modelu
Namespace	Pojmenovaný prostor
Node	Uzel
NodeInstance	Instance Uzlu
Object	Objekt

ObjectFlowState	Objektový tok-stav
Operation	Operace
Package	Svazek
Package Activity Graph	Svazek Grafy aktivit
Package Auxiliary elements	Svazek Prvky příslušenství
Package Behavioral Elements	Svazek Prvky chování
Package Common Behavior	Svazek Společné chování
Package Core	Svazek Jádro
Package Data Types	Svazek Datové typy

Package Extension Mechanisms	Svazek Mechanismy extenze
Package Foundation	Svazek Základ
Package Managment model	Svazek Řízení modelu
Package State Machines	Svazek Stavové stroje
Package Use Cases	Svazek Případy užití
Parameter	Parametr
Partition	Oddělení
Permission	Povolení
PresentationElement	Prezentační prvek
Pseudostate	Pseudostav
Qualifier	Kvalifikátor
Reception	Příjem
Relationship	Relace
ReturnAction	Akce návratová
SendAction	Akce zaslání
Signal	Signál
SignalEvent	Událost signálu
SimpleState	Jednoduchý stav
State	Stav
State Machine	Stavový stroj
StateVertex	Stavový vrchol
Stereotype	Stereotyp
Stimulus	Stimul
StructuralFeature	Rys struktury
StubState	Stub stav
SubactivityState	Subaktivita-stav
SubMachineState	Stav sub-stroje
Subsystem	Subsystém
SynchState	Synchronizační stav
Tagged Value	Tagovaná hodnota
Template	Šablona

TerminateAction	Akce ukončení
TimeEvent	Časová událost
Transition	Přechod
UninterpretedAction	Akce nespecifikovaná
Usage	Užití
UseCase	Případ užití
UseCaseInstance	Instance Případu užití