Prednáška 9: Konfigurovateľnosť v modelovaní softvéru

Metódy a prostriedky špecifikácie 2012/13

Valentino Vranić

Ústav informatiky a softvérového inžinierstva
Fakulta informatiky a informačných technológií
Slovenská technická univerzita v Bratislave

26. november 2012

Obsah prednášky

1. Úvod
2. Rady softvérových výrobkov
3. Modelovanie vlastností
4. Aspektovo-orientovaný prístup a rady softvérových výrobkov
5. Usmernenia pre aplikáciu AOP pri vývoji radov softvérových výrobkov
6. Prípady použitia a aspektovo-orientovaný prístup

Úvod

Doménové inžinierstvo

- Dnes prevláda tvorba individuálnych systémov
  - Špecifický systém pre špecifického zákazníka a kontext
- Doménové inžinierstvo – organizovaný prístup k znovupoužitiu
- Vývoj jedného softvérového systému vs. vývoj rodiny softvérových systémov
- Rad softvérových výrobkov (software product line)
- Modelovanie domény: zachytenie variability
- Tvorba komponentov
- Implementácia jednotlivých členov rodiny

Konfigurovateľnosť

- Individuálne softvérové výrobky získavame prednostne konfiguráciou vlastnosti radu softvérových výrobkov
- Vývoj radu softvérových výrobkov (výber techník implementácie) je v najväčšej miere ovplyvnený potrebami konfigurovania
- Explicitné *modelovanie vlastností* pomáha v tomto
- Na dosiahnutie väčšej flexibility konfigurovania potrebné sú pokročilé mechanizmy kompozície softvéru – *aspektovo-orientovaný prístup*

Rady softvérových výrobkov

Doménové inžinierstvo[1]

- Doménové inžinierstvo (*domain engineering*)

  > *Aktivita zbierania, organizovania a ukladania skúseností v stavbe systémov alebo ich častí v určitej doméne v tvare znovupoužiteľných prvkov, ako aj poskytovania adekvátnych prostriedkov na znovupoužitie týchto prvkov pri stavbe nových systémov.*

- Znovupoužitie sa uskutočňuje v aplikačnom inžinierstve
- Softvérové systémy v doméne sú podobné

[1] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

Doména

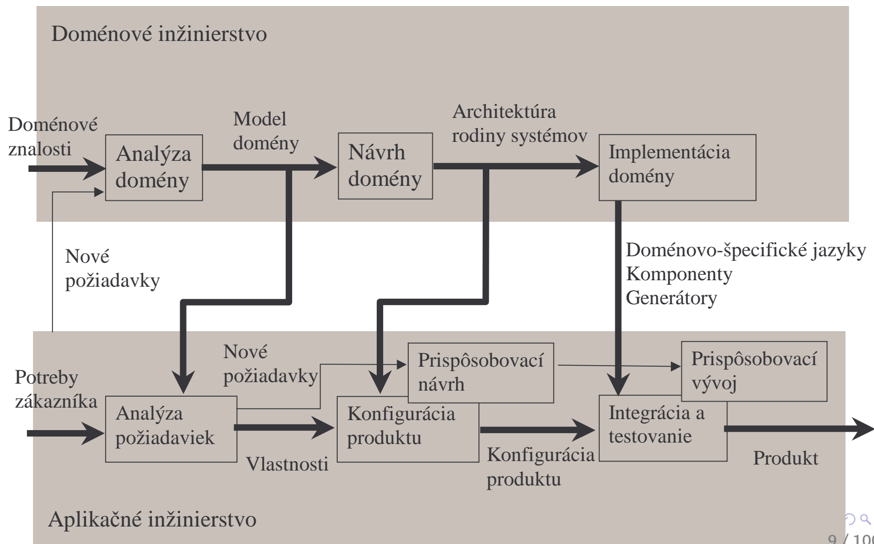
- Doména: oblasť záujmu[2]
- Dva pohľady:
  - Doména ako „reálny svet“ (napr. bankovníctvo)
  - *Doména ako množina systémov* (napr. bankovníctvo ako také + softvérové systémy na jeho podporu)
- Doména: oblasť vedomostí[3]
  - S rozsahom určeným tak aby uspokojil požiadavky zainteresovaných (stakeholders)
  - Zahŕňa množinu pojmov a terminológiu zrozumiteľnú pôsobiacim v danej oblasti
  - Zahŕňa vedomosti o tom ako tvoriť softvérové systémy (alebo ich časti) v danej oblasti

---

[2] J. O. Coplien. Multi-Paradigm Design for C++. Addison-Wesley, 1999.

[3] K. Czarnecki, U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

Proces doménového inžinierstva



Analýza domény

- Tvroba modelu domény
- Aktivity:
    - Vymedzenie domény
        - Vyplýva zo záujmov zainteresovaných
        - Ekonomický zvládnuteľné
    - Modelovanie domény
        - Definícia domény
        - Slovník domény
        - Modely konceptov
        - *Modely vlastností*
- Každá reálna doména sa dá zjemňovať donekonečna

Návrh a implementácia domény

- Návrh domény
  - Tvorba architektúry rodiny systémov
  - Tvorba plánu výroby
- Implementácia domény
  - Architektúra
  - Komponenty
  - Plán výroby

Rady softvérových výrobkov

- Software product lines
- A Framework for Software Product Line Practice[4]
- Iniciatíva SEI (Software Engineering Institute)
  - http://www.sei.cmu.edu/productlines/
- Rieši záležitosti manažmentu a organizácie ohľadom zriadenia procesu doménového a aplikačného inžinierstva

[4] http://www.sei.cmu.edu/productlines/frame_report/

Zavedenie radu softvérových výrobkov[5]

- Evolučne / revolučne
- Jestvujúci rad výrobkov / nový rad výrobkov

⇒ Štyri možnosti:

  - Postupný vývoj radu výrobkov z jestvujúcej množiny výrobkov (výberom a zovšeobecnením ich spoločných vlastností bez prerušenia tvorby výrobkov)
  - Nahradenie jestvujúcej množiny výrobkov radom výrobkov (výberom a zovšeobecnením ich spoločných vlastností, ale pri zastavení ich tvorby)
  - Postupný vývoj nového radu výrobkov (počas ktorého vznikajú aj samotné výrobky)
  - Vývoj nového radu výrobkov (pred vznikom prvého výrobku)

---

[2] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

Modelovanie vlastností

Modelovanie vlastností

- Feature modeling
- Zachytáva prepojenia medzi vlastnosťami a variabilitu
- Model vlastností: množina diagramov vlastností plus ďalšie informácie
- Založené na pojmoch *domény*, *konceptu* a *vlastnosti*[6]
  - Koncept – chápanie triedy alebo kategórie prvkov v doméne
  - Vlastnosti – dôležité charakteristiky konceptov; spoločné a premenlivé
  - Každú vlastnosť možno ďalej rozoberať ako koncept
  - Inštancie konceptov: špecializácie konceptov získané výberom vlastností

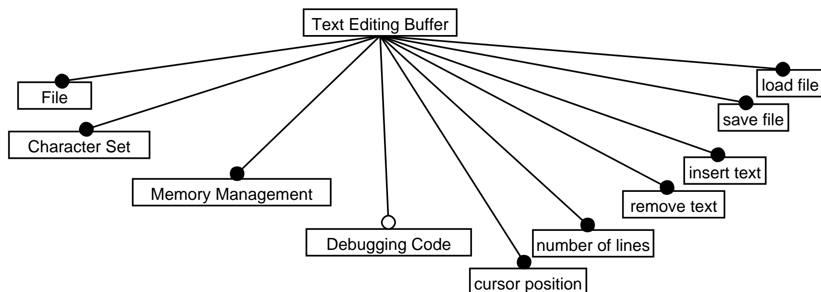
[6] V. Vranić. Reconciling Feature Modeling: A Feature Modeling Metamodel. In M. Weske and P. Liggesmeyer, Eds., *Proc. of 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays 2004)*, Erfurt, Germany, Sept. 2004. Springer.

Rôznorodosť notácií modelovania vlastností

- Veľa rôznych (aj špecializovaných) notácií
  - FODA (Feature-Oriented Domain Analysis) – prvá notácia[7]
  - ODM (Organizational Domain Modeling)
  - Základná Czarneckého-Eiseneckerova notácia
  - FORM (Feature-Oriented Reuse Method)
  - Czarneckého-Eiseneckerova notácia založená na kardinalite
  - pure::variants
  - Modelovanie vlastností pre multiparadigmový návrh
- Na tu poslednú notáciu, ktorá je založená na základnej Czarneckého-Eiseneckerovej notácii, sa pozrieme prostredníctvom príkladov

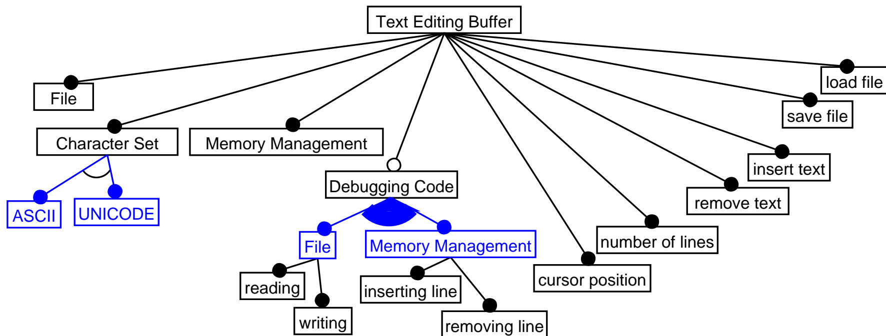
[7] K. C. Kang et al. Feature-oriented domain analysis (FODA): A feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, 1990.

Variabilita vlastností (1)



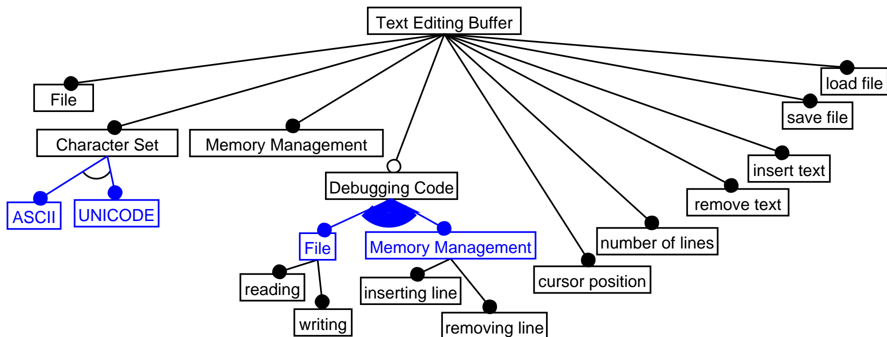
- *Povinné* vlastnosti (hrany ukončené vyplnenými krúžkami)
- *Voliteľné* vlastnosti (hrany ukončené prázdnymi krúžkami)

Variabilita vlastností (2)



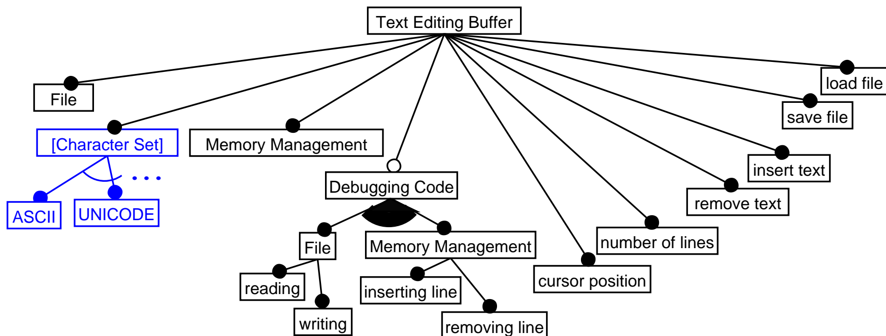
- *Alternatívne* vlastnosti (prázdny oblúk)
- *Alebo*-vlastnosti (vyplnený oblúk)

Variabilita vlastností (3)



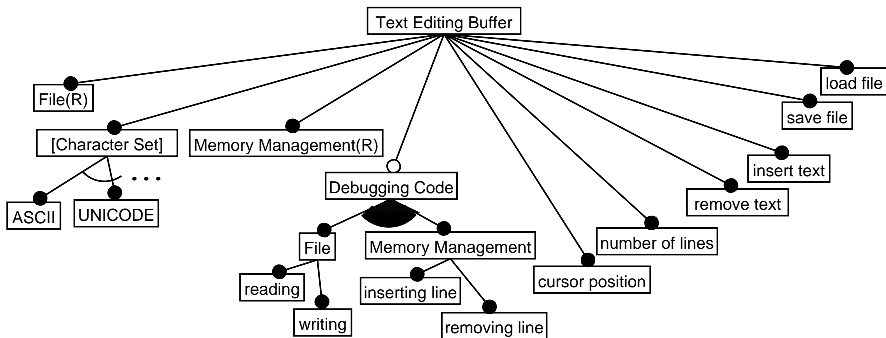
- Oblúky upravujú význam hrán
  - Povinné alternatívne / voliteľné alternatívne vlastnosti
  - Povinné alebo- / voliteľné alebo-vlastnosti

Variabilita vlastností (4)



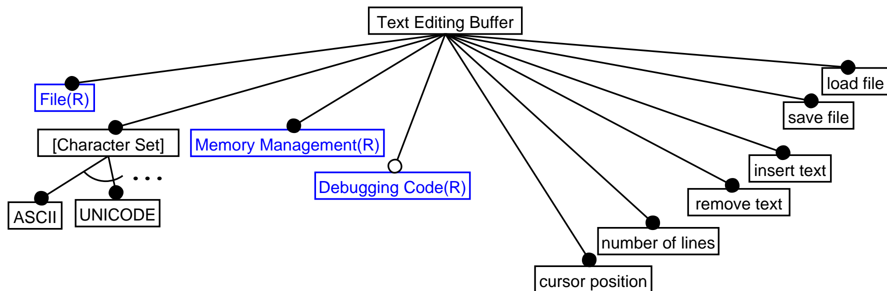
- Otvorené vlastnosti
  - Očakávajú sa ďalšie premenlivé podvlastnosti
  - Označuje sa hranatými zátvorkami; prípadne sa upresní tromi bodkami

Variabilita vlastností (5)



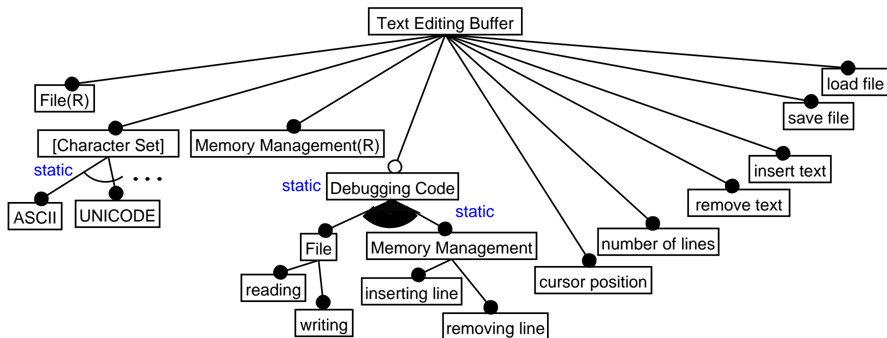
- Zahrnutie vlastnosti v inštancii konceptu je podmienené zahrnutím jej rodiča
- Vlastnosti hocijakého typu variability sa môžu vyskytovať na hocijakej úrovni

Referencie konceptov



- Označujú sa rôzne – tu su označené symbolom ® (v diagramoch ako (R) z technických dôvodov)
- Môžno ich rozvinúť v prípade potreby

Čas/mód viazania

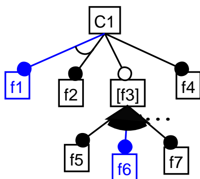


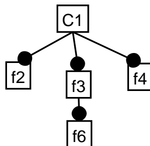
- Binding time/mode
- Kedy/ako sa vlastnosť bude viazať
- Obvykle časy viazania: tvorba zdrojového kódu, preklad, spájanie, načítavanie a vykonávanie
- Mód viazania: statický alebo dynamický

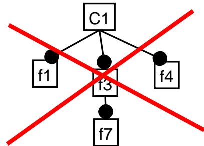
Ďalšie informácie v modeloch vlastností

- Informácie pridružené konceptom a vlastnostiam
  - Textuálne informácie: opis, dôvod prítomnosti, dôvod zahrnutia, poznámka
  - Čas/mód viazania
- Obmedzenia a pravidlá preddefinovaných závislostí
  - Príklad obmedzenia:

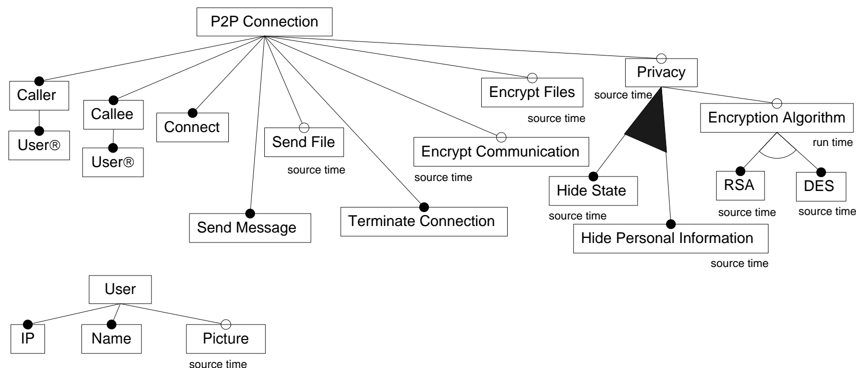
$$f1 \Rightarrow f6$$



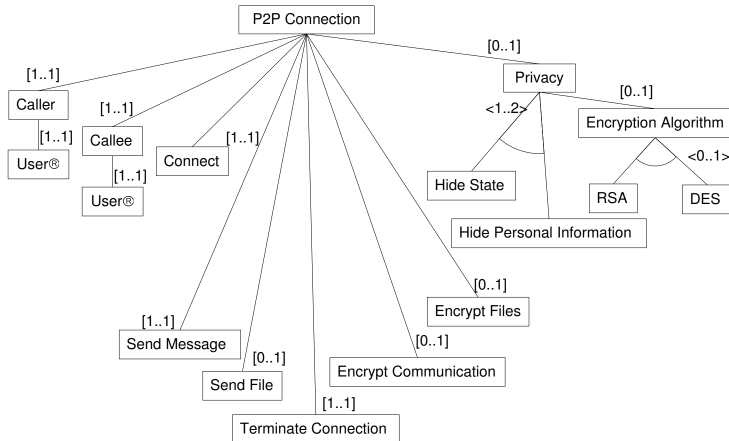




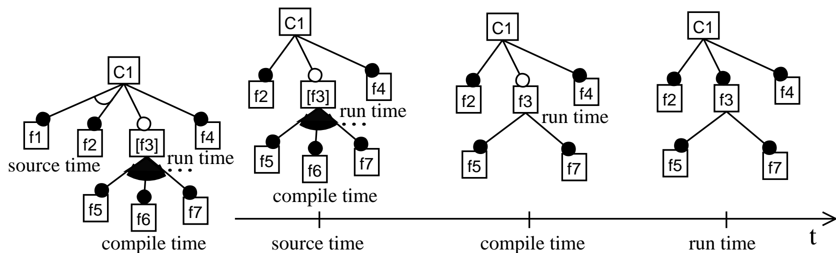
Iný príklad: P2P Connection



P2P Connection v notácii zloženej na kardinalite



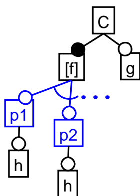
Inštanciácia konceptov



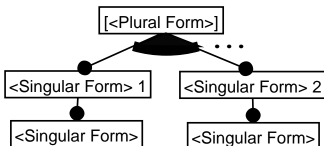
Parametrizácia v modeloch vlastností

- Parametrizované názvy vlastností a konceptov

Obmedzenie: $\forall <i> \in N \; p<i>.h \veebar g$



- Parametrizované koncepty



Konfigurovanie kódu – pure::variants

- pure::variants – komerčný nástroj na podporu vývoja radov softvérových výrobkov založeného na modelovaní vlastností[8]
- Implementácia radu softvérových výrobkov je definovaná v špeciálnom modeli rodín komponentov (component family model)
- Tento model sa konfiguruje výberom vlastností z modelu vlastností

[8] http://www.pure-systems.com/

Konfigurovanie modelu – superponované varianty (1)

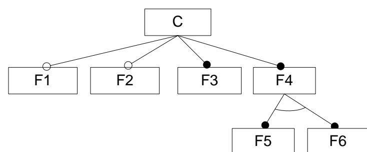
- Superimposed variants[9]
- Model (v UML) obsahuje všetky možné varianty
- Napr. diagram tried obsahuje všetky triedy a vzťahy medzi nimi
- Varianty sú *superponované* – „navrstvené“
- Výber variantov je regulovaný výberom vlastností v modeli vlastností
- Pre toto prvky v modeli UML musia byť priradené vlastnostiam, čo definuje podmienky prítomnosti (presence conditions)

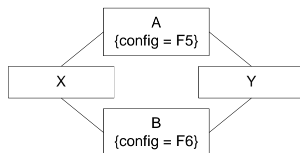
[9] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In Proc. of GPCE'05, 2005.
http://www.swen.uwaterloo.ca/~kczarnec/tr2005-05.pdf

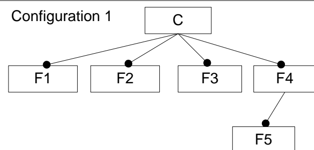
Konfigurovanie modelu – superponované varianty(2)

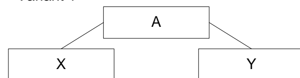
- Niektoré podmienky prítomnosti sú implicitné – napr. zrušenie triedy automaticky znamená aj zrušenie asociácie, na ktorej konci bola
- Prístup sa dá uplatniť aj na behaviorálne modely (bol implementovaný pre diagramy aktivít)

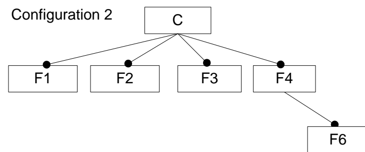
Superponované varianty – príklad

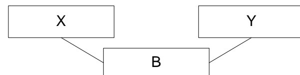












Softvérová podpora superponovaných variantov

- Podpora prototypmi (pluginy do Eclipse):
  - Konfigurovanie pomocou fmp2rsm[10]
  - Samotný model vlastností zostrojený v nastroji fmp[11] je plugin do Eclipse

[10] http://gp.uwaterloo.ca/fmp2rsm/
[11] http://gp.uwaterloo.ca/fmp/

Aspektovo-orientovaný prístup a rady softvérových výrobkov

Aspekty a variabilita (1)

- Pri tradičných prístupoch k implementácii kód pre danú vlastnosť môže byť roztrúsený (scattered) vo viacerých komponentoch
- Toto je zvlášť významné pre variabilné vlastnosti, lebo ich potrebujeme pripájať a odpájať podľa zvolenej konfigurácie
- Lee et al.[12] – základná idea: spoločné vlastnosti implementovať bežným spôsobom, a variabilné vlastnosti implementovať aspektmi

[12] Lee et al. Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development. In Proc. of 10th International Software Product Line Conference, August 2006. IEEE Computer Society.

Aspekty a variabilita (2)

- Reálne je však potrebná dôkladnejšia analýza pre danú vlastnosť, ktorá ukáže ako má byť implementovaná
- Uvažuje sa všeobecne v zmysle aspektovo-orientovaného prístupu – vlastnosti, ktoré pretínajú iné, sa implementujú aspektovo-orientovaným spôsobom bez ohľadu na to, či sú variabilné alebo nie
- Sledujú sa aj špecifické záležitosti pre rady softvérových výrobkov – predovšetkým na základe závislosti medzi vlastnosťami
- Berie sa do úvahy aj čas viazania

Vyčlenenie biznis pravidiel

- Z hľadiska konfigurovateľnosti je vhodné vyčleniť biznis pravidlá do aspektov
- Potrebnú konfiguráciu aplikácie potom vytvoríme zo základného kódu (môže tiež obsahovať aspekty) a vybraných biznis pravidiel
- Príklad z AspectJ in Action[13]
  - Správu minimálneho zostatku (biznis pravidlo) pokladáme za zvláštnu záležitosť
  - Nechceme ju miešať so základnou funkcionalitou účtu

[13] http://www.manning.com/laddad/

Príklad: účet – trieda Account (1)

```
public abstract class Account {
  private float _balance;
  private int _accountNumber;

  public Account(int accountNumber) {
    _accountNumber = accountNumber;
  }

  public void credit(float amount) {
    setBalance(getBalance() + amount);
  }
...
```

Príklad: účet – trieda Account (2)

```
...
  public void debit(float amount) throws InsufficientBalanceException {
    float balance = getBalance();
    if (balance < amount) {
      throw new InsufficientBalanceException(
        "Total balance not sufficient");
    }
    else {
      setBalance(balance - amount);
    }
  }
...
```

Príklad: účet – trieda Account (3)

```
...
  public float getBalance() {
    return _balance;
  }

  public void setBalance(float balance) {
    _balance = balance;
  }
}
```

Príklad: účet – trieda SavingsAccount

```
public class SavingsAccount extends Account {
    public SavingsAccount(int accountNumber) {
        super(accountNumber);
    }
}
```

Príklad: účet – správa minimálneho zostatku

```
public aspect MinimumBalanceRuleAspect {
  private float Account._minimumBalance;

  public float Account.getAvailableBalance() {
    return getBalance() − _minimumBalance;
  }

  after(Account account):
    execution(SavingsAccount.new(..)) && this(account) {
    account._minimumBalance = 25;
  }

  before(Account account, float amount) throws InsufficientBalanceException:
    execution(∗ Account.debit(..)) && this(account) && args(amount) {
    if (account.getAvailableBalance() < amount) {
      throw new InsufficientBalanceException(
        "Insufficient available balance");
    }
  }
}
```

Zavedenie dedenia

- Aspektom sa dá zaviesť aj vzťah **implements**, aj **extends**
- Musia sa dodržiavať pravidlá dedenia platné v Jave
- Zavedenie značkovacieho rozhrania za účelom sledovania prvkov:

```
aspect AccountTrackingAspect {
  declare parents: banking..entities.* implements Identifiable
}
```

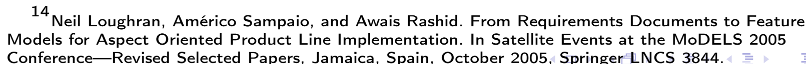
- Závadzanie vzťahov **extends** je zaujímavé predovšetkým z hľadiska konfigurovateľnosti
  - Viac tried rovnakého rozhrania, ale inej implementácie
  - Aspektom určíme ktorá z týchto tried sa využije

Problém interakcie vlastností

- Vlastnosť môže vyžadovať prítomnosť alebo neprítomnosť inej vlastnosti
- Tento vzťah môže byť jednostranný alebo obojstranný
- Prejavuje sa aj potrebou prispôsobenia bodových prierezov jednotlivým konfiguráciam
- Abstraktné aspekty – spôsob ako vyčleniť závislosti
  - Závislosti sa implementujú ako konkrétne bodové prierezy v konkrétnych aspektoch
  - Samotná funkcionalita je v abstraktnom aspekte

NAPLES a Framed Aspects (1)

- Loughran a Rashid[14] navrhujú *Natural language Aspect-based Product Line Engineering for Systems* (NAPLES)
- V NAPLES sa koncepty (záležitosti) v požiadavkách identifikujú pomocou na to vytvoreného nástroja
- Na základe tohto procesu sa vytvorí model vlastností v Czarneckého-Eiseneckerovej základnej notácii



[14] Neil Loughran, Américo Sampaio, and Awais Rashid. From Requirements Documents to Feature Models for Aspect Oriented Product Line Implementation. In Satellite Events at the MoDELS 2005 Conference—Revised Selected Papers, Jamaica, Spain, October 2005, Springer LNCS 3844.

NAPLES a Framed Aspects (2)

- Na model vlastností sa aplikuje prístup Framed Aspects[15]
- Implementácia jednej záležitosti typicky pozostáva z viacerých aspektov a tried[16]
- Framed Aspects
  - Aspekty, ktoré spoločne predstavujú jednu záležitosť, sú zoskupené do rámcov (frames)
  - Aspekty sú parametrizované
  - Parametre sa nastavujú na úrovni rámcov
- Rámce sa identifikujú v modeli vlastnosti
  - Povinná vlastnosť je vždy v spoločnom rámci so svojim rodičom
  - Každá variabilná vlastnosť vystupuje vo vlastnom rámci, ktorý sa potom dá samostatne pripájať a odpájať

[15] Neil Loughran and Awais Rashid. Framed Aspects: Supporting Variability and Configurability for AOP. In Proc. of 8th International Conference on Software Reuse, ICSR 2004, Madrid, Spain, July 2004, Springer LNCS 3107.

[16] Adrian Colyer, Awais Rashid, and Gordon Blair. On the Separation of Concerns in Program Families. Technical Report, Computing Department, Lancaster University, January 2004.

Multiparadigmový návrh s modelovaním vlastností (1)

- Pri aplikácii aspektovo-orientovaného prístupu sa musíme rozhodnúť kedy použiť aspekty
- Ide vlastne o *výber paradigmy*
- Multiparadigmový návrh s modelovaním vlastností (MPD_{FM})[17] – metóda pre výber paradigiem
- Prístup je založený na Coplienovom multiparadigmovom návrhu[18]

---

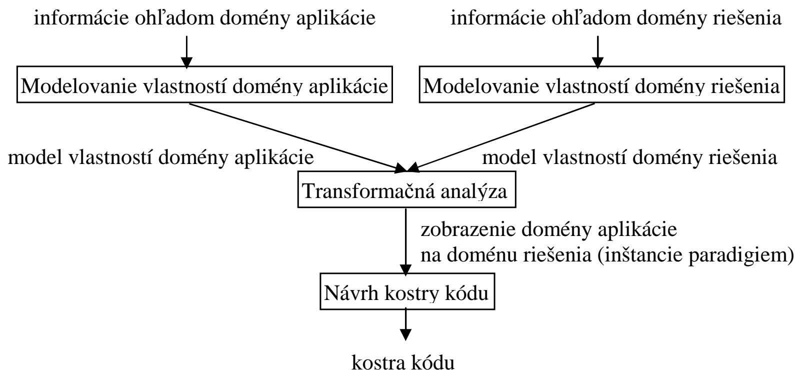
[17] V. Vranić. Multi-paradigm design with feature modeling. Computer Science and Information Systems Journal (ComSIS). 2(1): 79–102, 2005.
http://comsis.fon.bg.ac.yu/ComSIS/Vol2No1/RegularPapers/VVranic.htm

[18] J. O. Coplien. Multi-Paradigm Design for C++. Addison-Wesley, 1998.
(J. O. Coplien. Multi-Paradigm Design. PhD thesis, Vrije Universiteit Brussel, 2000.
http://users.rcn.com/jcoplien/Mpd/Thesis/Thesis.pdf

Multiparadigmový návrh s modelovaním vlastností (2)

- Proces vývoja softvéru: zobrazenie domény aplikácie (problému) na doménu riešenia
- Paradigma vývoja softvéru: ako vyjadriť koncepty domény aplikácie v zmysle konceptov domény riešenia
- Koncepty domény riešenia zodpovedajú mechanizmom programovacích jazykov
- Jednotlivé koncepty domény riešenia (v AspectJ trieda, aspekt, videnie atď.) možno považovať za paradigmy

Aktivity v MPD_{FM}

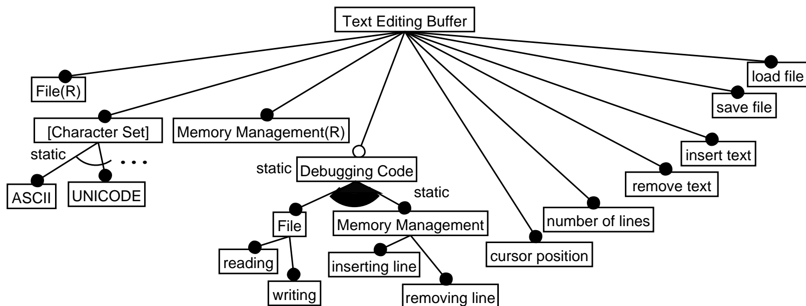


- Ak je doménou riešenia AspectJ, v MPD_{FM} sa rozhoduje aj o použití aspektov

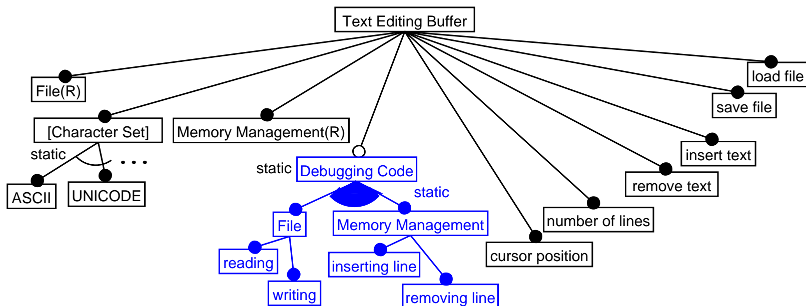
Inštanciácia paradigiem v MPD_{FM}

- Inštanciácia paradigiem v MPD_{FM} je vlastne inštanciácia konceptov
  - Chápe sa ako špecializácia konceptov
  - Inštancie konceptov sa reprezentujú pomocou diagramov vlastností
  - Berie sa do úvahy čas viazania
- Inštanciácia zdola nahor
- Zahrnutie uzlov paradigmy podmienené zobrazením uzlov konceptov domény aplikácie
  - Konceptuálny súlad
  - Súlad dôb viazania

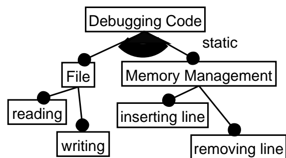
Príklad transformačnej analýzy (1)



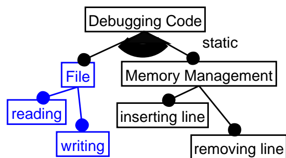
Príklad transformačnej analýzy (2)



Príklad transformačnej analýzy (3)

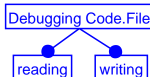


Príklad transformačnej analýzy (4)

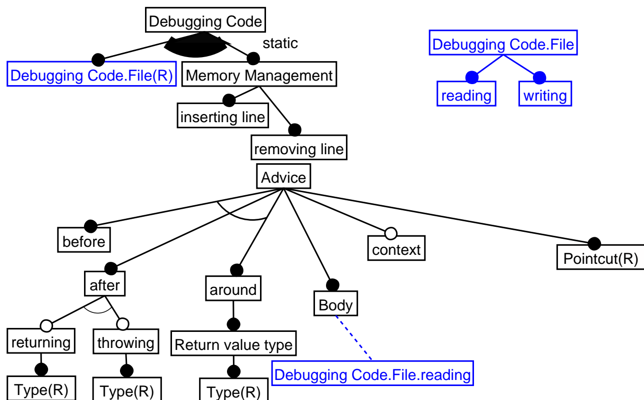


Príklad transformačnej analýzy (5)

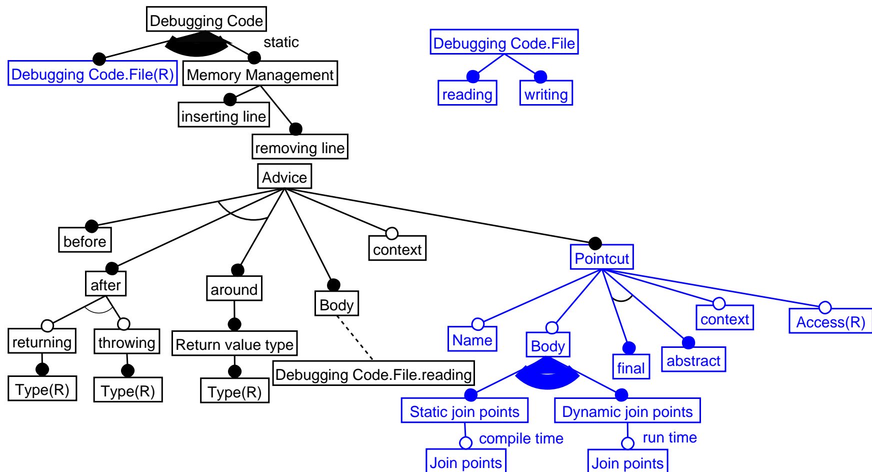
Debugging Code
static
Debugging Code.File(R)
Memory Management
inserting line
removing line
Advice
before
after
around
Body
context
Pointcut(R)
returning
throwing
Return value type
Type(R)
Type(R)
Type(R)



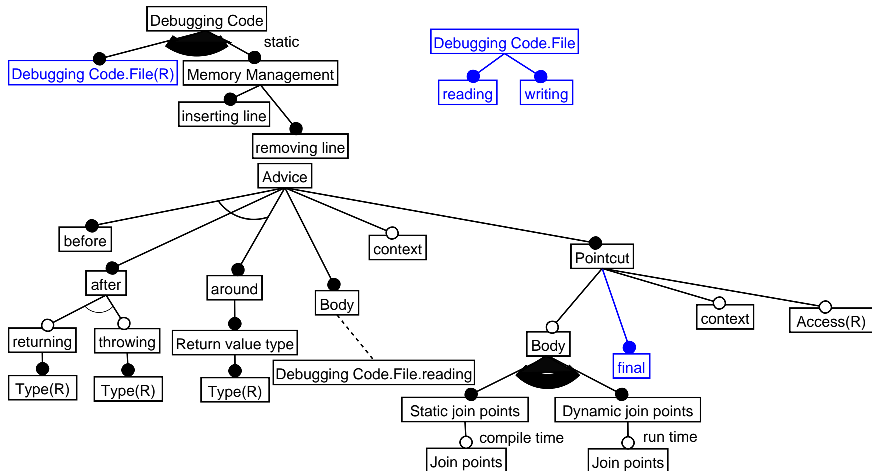
Príklad transformačnej analýzy (6)



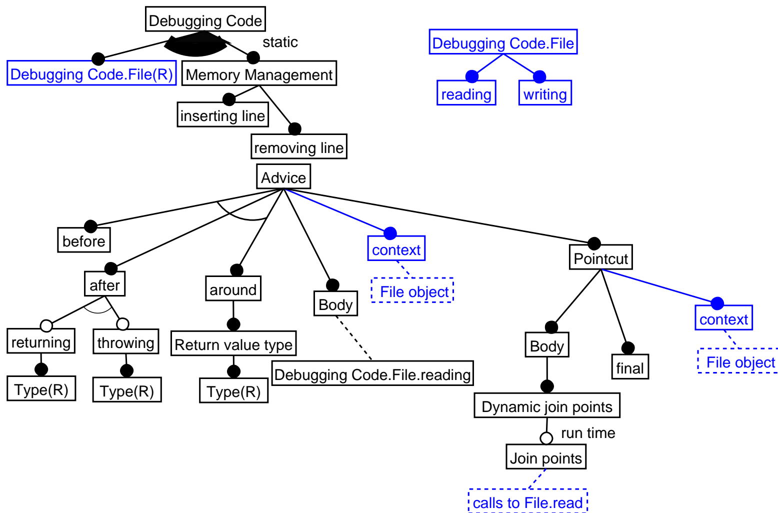
Príklad transformačnej analýzy (7)



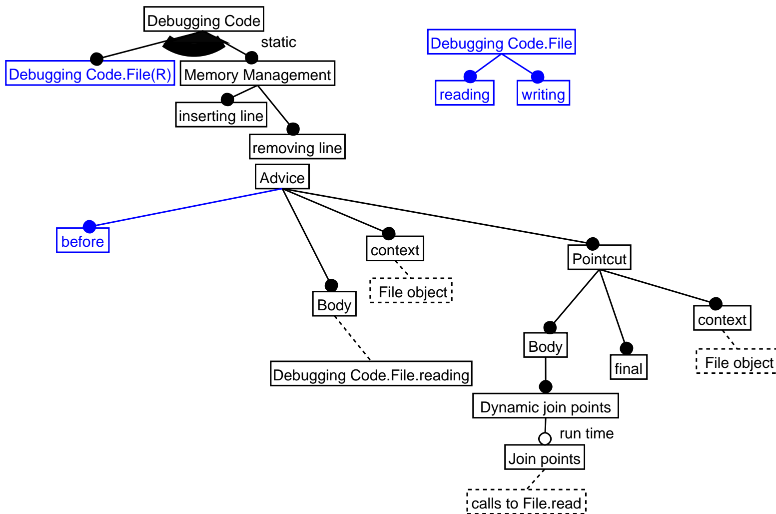
Príklad transformačnej analýzy (8)



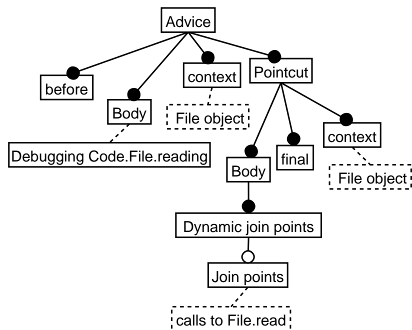
Príklad transformačnej analýzy (9)



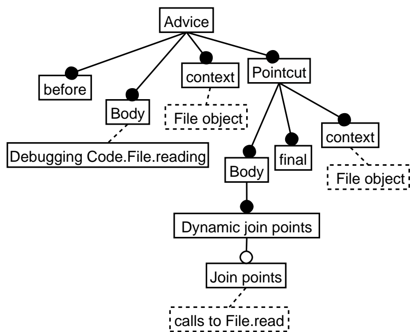
Príklad transformačnej analýzy (10)

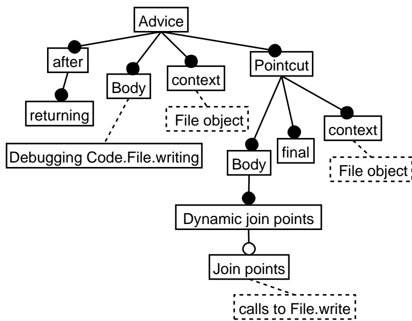


Príklad transformačnej analýzy (11)

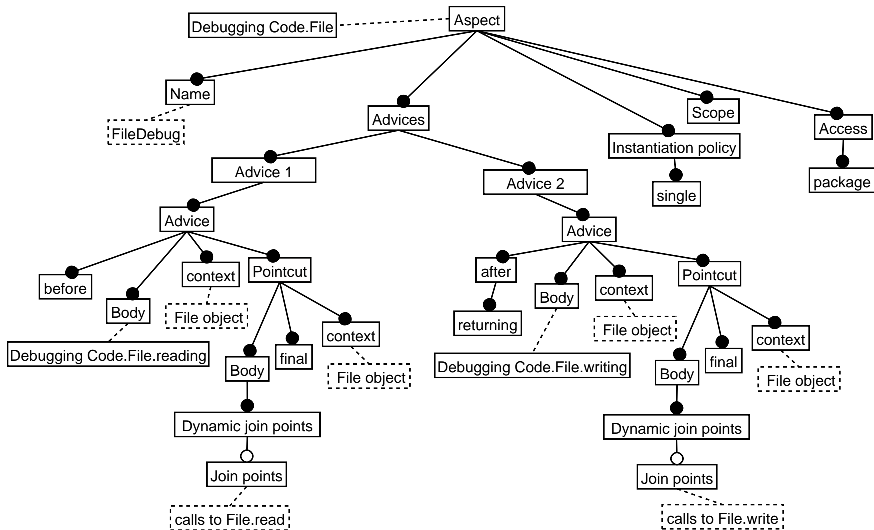


Príklad transformačnej analýzy (12)





Príklad transformačnej analýzy (13)



Návrh kostry kódu

- Uskutočňuje sa prechádzaním stromami inštancií paradigiem
- Najprv inštancie štrukturálnych paradigiem
- Príklad: aspekt ladenia kódu práce so súbormi

```
aspect FileDC {
  before(File f): target(f) && call(* File.read(..)) {
     . . .
  }

  after(File f): target(f) && call(* File.write(..)) {
     . . .
  }
}
```

Usmernenia pre aplikáciu AOP pri vývoji radov softvérových výrobkov

Usmernenia (1)

- Usmernenia (guidelines) pre aplikáciu aspektovo-orientovaného programovania pri vývoji radov softvérových výrobkov[19] [20]
- Odporúčania kedy použiť aspekty vzhľadom na danú konfiguráciu vlastností
- Vyjadrené v štruktúre podobnej návrhovým vzorom
- Môžu predstavovať sústredené vyjadrenie vedomostí získaných použitím $\mathrm{MPD_{FM}}$ overených v praxi

[19] V. Vranić and J. Kohut. Guidelines for Using Aspects in Product Lines. In Proc. of 8th International Symposium on Applied Machine Intelligence and Informatics, SAMI 2010, January 2010, Herľany, Slovakia, IEEE.

[20] J. Kohut. Usmernenia pre využitie aspektovo-orientovaného prístupu v radoch softvérových výrobkov. Diplomová práca, FIIT STU, 2009.
http://fiit.stuba.sk/~vranic/proj/dp/Kohut/diplomova_praca.pdf

Usmernenia (2)

- Implementing Features of Refactored Legacy Applications
  - The aspects are unsuitable for implementing features of refactored legacy applications
  - Feature refactoring of legacy applications is a difficult problem because such applications do not imply that their design was amenable to feature extensibility
- Implementing Mandatory Features with no Crosscutting Concerns
  - Do not use aspects in mandatory features if there are no crosscutting concerns
  - Aspects flatten inherent object-oriented structure of collaboration, obscure the intent of the programmer, and the result is a program that is difficult to read

Usmernenia (3)

- Code Reduction in Homogenous Crosscutting Concerns
  - The aspects are suitable for reduction of replicated code in homogenous crosscutting concerns
  - Aspects reduce replicated code in code with homogenous crosscutting concerns
- Transforming a Mandatory Feature into Alternative Features
  - Do not use aspects in transforming a mandatory feature into alternative features
  - By transforming a mandatory feature into two or more alternative features AspectJ adds and changes more components and lines of code compared to non-aspect-oriented approaches because all aspects rely on the join points provided by the core

Usmernenia (4)

- Implementing Features which Share no Code
  - Use aspects in implementing features which share no code and which have crosscutting concerns
  - Aspect-oriented solution provides low cost and superior stability in terms of tangling and scattering over components

Prípady použitia a aspektovo-orientovaný prístup

Zlepšenie konfigurovateľnosti

- Príležitosti pre zlepšenie konfigurovateľnosti radu softvérových výrobkov treba hľadať od začiatku
- Prvým modelom nad špecifikáciou požiadaviek často býva model prípadov použitia
- Nezameriava sa na konfigurovateľnosť ako modelovanie vlastností, ale presnejšie vyjadruje funkcionalitu
- Model prípadov použitia možno konfigurovať modelom vlastností (uplatnením superponovaných variantov)

Prípadmi použitia riadený AO vývoj softvéru

- Ivar Jacobson, OOSE – a use case driven approach, UML, RUP, agile methods. . .
- Podľa Ivara Jacobsona prípady použitia sú principiálne aspektovo-orientované
- Preto navrhuje riadiť aspektovo-orientovaný vývoj softvéru prípadmi použitia
- Toto výrazne zjednodušuje konfigurovanie modelu prípadov použitia

Požiadavky, prípady použitia a záležitosti

- Záležitosť (concern):

  > *anything that is of interest to some stakeholder, whether an end user, a project sponsor, or even a developer*[21]

- Treba rozlišovať medzi požiadavkami a záležitosťami
- Požiadavky opisujú záležitosti; zvyčajne sa viac požiadaviek vzťahuje na jednu záležitosť (spomeňte si na prístup Theme)
- Prípad použitia zachytáva záležitosť

[21] I. Jacobson and P. W. Ng. Aspect-Oriented Software Development with Use Cases. Addison-Wesley, 2004.

Dva druhy AO vzťahov medzi prípadmi použitia

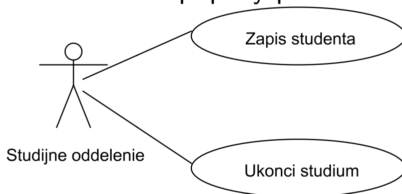
1. Prípady použitia na rovnakej úrovni (peer use cases)
2. Pretínanie vyjadrené vzťahom rozšírenia («extend»)

Prípady použitia na rovnakej úrovni

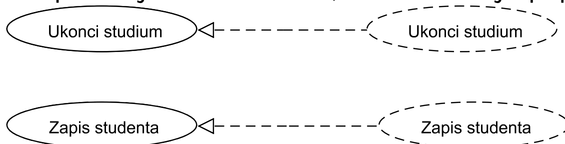
- Peer use cases – prípady použitia na rovnakej úrovni
- Nemajú vzájomné vzťahy
- V ich realizáciách sa objavujú rovnaké triedy

Príklad: zápis a ukončenie štúdia

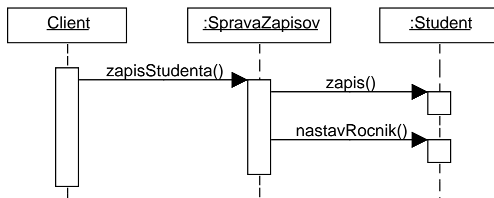
- Dva nezávislé prípady použitia:

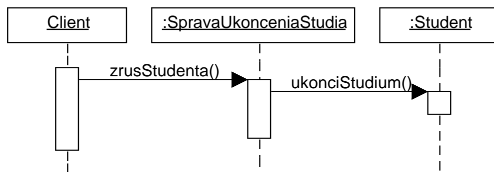


- Zodpovedajúce kolaborácie, ktoré realizujú prípady použitia:



Kolaborácia vyjadrená diagramom sekvencií



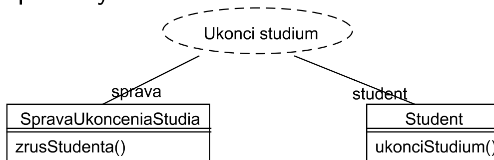


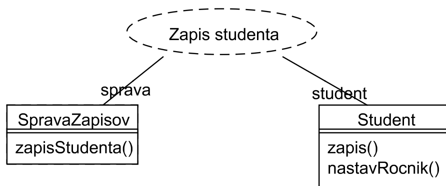
Identifikácia rozšírení tried (1)

- V diagramoch prípadov použitia identifikujeme spoločné triedy
- V uvedenom príklade je to trieda Student
- V realite však nemusia mať rovnaké meno

Identifikácia rozšírení tried (2)

- Pre každý prípad použitia identifikujeme potrebné prvky spoločných tried





Identifikácia rozšírení tried (3)

- Takéto triedy predstavujú *pohľady* na tú istú triedu z perspektívy rôznych záležitostí
- Kľúčové je potom implementovať takéto pohľady oddelene
- Presne toto umožňujú až aspektovo-orientované jazyky
- V uvedenom príklade prípady použitia Zapíš šudenta a Ukonči štúdium majú svoj pohľad na triedu `Student`

Implementácia rozšírení tried (1)

- Pre implementáciu záležitostí z prípadov použitia na rovnakej úrovni riadime symetrickým prístupom
- V jazyku AspectJ sa dá zabezpečiť medzitypovými deklaráciami
- V triede `Student` budú niektoré spoločné veci alebo úplne prazdna trieda:

```
class Student {
    String meno;
    . . .
}
```

Implementácia rozšírení tried (2)

- Prípady použitia dopĺňajú triedu Student atribútmi a metódami potrebnými z ich perspektívy:

```
aspect ZapisStudenta {
    public void Student.zapis(Student s) { . . . }
    public void Student.nastavRocnik(Student s, int rocnik) { . . . }
    . . .
}

aspect UkonciStudium {
    public void Student.ukonciStudium(Student s) { . . . }
    . . .
}
```

Implementácia rozšírení tried (3)

- Uvedený príklad je zjednodušený
- V realite budeme mať pretínanie na úrovni metód
- Vyžaduje použitie videní: metóda jednej záležitosti sa spustí pred, za alebo namiesto metódy inej záležitostí

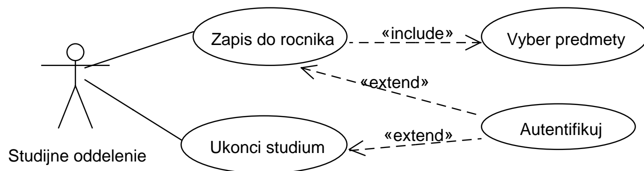
Vzťah «extend»(1)

- Vzťah «extend» znamená, že jeden prípad použitia rozširuje iný prípad použitia – v tzv. bodoch rozšírenia
- Základná funkcionalita rozšíreného prípadu použitia je zachovaná aj bez rozšírenia

Vzťah «extend»(2)

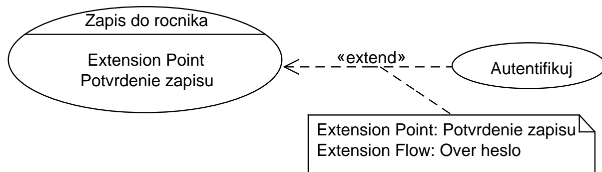
- Body rozšírenia (extension points) – identifikujeme ich v opise rozširovaného prípadu použitia
- Opis môže byť na rôznych úrovniach detailov
- Zvyčajne sa uvádza ako zoznam krokov, ale môže byť aj v tvare neštrukturovaného textu
- Toky: základný (basic), alternatívny (alt) a podtok (sub)

Príklad: rozšírenie prípadu použitia (1)



Príklad: rozšírenie prípadu použitia (2)

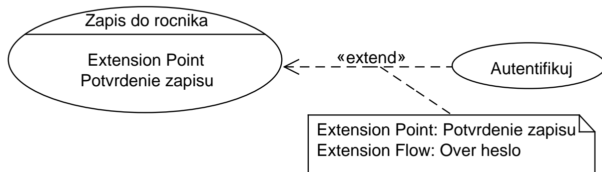
- Obvyklé vizuálne vyjadrenie bodov rozšírenia v UML



- Poznámka môže obsahovať aj Extension Condition (podmienku rozšírenia)

Príklad: rozšírenie prípadu použitia (2)

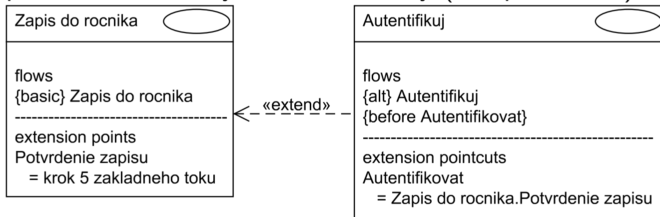
- Obvyklé vizuálne vyjadrenie bodov rozšírenia v UML



- Poznámka môže obsahovať aj Extension Condition (podmienku rozšírenia)

Príklad: rozšírenie prípadu použitia (3)

- Ivar Jacobson navrhuje použiť obdĺžnikový zápis prípadov použitia obohatený o ďalšie oddiely (compartments)



- Nové prvky:
  - sufix pre rozširujúci tok: *before* alebo *after*
  - *extension pointcut* – bodový prierez rozšírenia

Príklad: body rozšírenia v opise prípadu použitia (1)

Prípad použitia: Zapíš do ročníka

Základný tok

1. Študijné oddelenie zvolí zápis študenta do ročníka.
2. Študijné oddelenie zadá ročník.
3. Aktivuje sa prípad použitia **Vyber predmety**.
4. Ak študijné oddelenie potvrdí zápis, systém uloží zadané údaje.
5. Prípad použitia končí.

Alternatívne toky

...

Body rozšírenia

- Potvrdenie zápisu: krok 4

Príklad: body rozšírenia v opise prípadu použitia (2)

Prípad použitia: Autentifikuj

Alternatívne toky

Tok: Over heslo

Tok sa aktivuje pred nasledujúcimi bodmi rozšírenia:

- Zapíš do ročníka.Potvrdenie zápisu...
- ...

1. Systém vyžiada heslo.
2. Študijné oddelenie zadá heslo.
3. Ak je heslo správne, systém potvrdí autentifikáciu. Inak ju zamietne a ukončí základný tok.

Body rozšírenia vs. body spájania

- V AOP je kľúčové, že body spájania sa určujú mimo prvku, ktorý ich obsahuje
- Nepotrebujeme žiadne „háky“
- Je pri bodoch rozšírenia toto narušené?
- Explicitné vymenovanie bodov rozšírenia zodpovedá skôr určeniu exponovaných bodov spájania
- Keďže prípady použitia opisujeme v prirodzenom jazyku, nedá sa táto záležitosť formalizovať
- Cockburn pripúšťa deklaratívne uvedenie bodov rozšírenia[22]

> *Spúšťač:*[23] *Kedykoľvek študent uskutočňuje operáciu, ktorá vyžaduje autentifikáciu*

---

[22] Alistair Cockburn. Writing Effective Use Cases. Addison-Wesley, 2000.

[23] trigger

Implementácia

- Rozširujúci prípad použitia implementujeme aspektom
  - Rozširujúca funkcionalita bude implementovaná videním
  - Body rozšírenia predstavujú body spájania a budú zachytené bodovým prierezom

```
public aspect Authentication {
    around(): call(* Enrollment.acknowledge(..)) {
        ... // vyžiadaj a over heslo používateľa
        if (...) { // ak je OK
            proceed(); // pokračuj v zachytenom potvrdení zápisu
        }
    }

}
```

Body rozšírenia vs. body spájania

- V AOP je kľúčové, že body spájania sa určujú mimo prvku, ktorý ich obsahuje
- Nepotrebujeme žiadne „háky“
- Je pri bodoch rozšírenia toto narušené?
- Explicitné vymenovanie bodov rozšírenia zodpovedá skôr určeniu exponovaných bodov spájania
- Keďže prípady použitia opisujeme v prirodzenom jazyku, nedá sa táto záležitosť formalizovať

Symetrickosť AOP

- Pri implementácii prípadov použitia na rovnakej úrovni bolo použitý tzv. symetrické AOP, kým pri vzťahu extend asymetrické AOP
- Asymetrické aspektovo-orientované prístupy[24]
  - Rozlišuje sa medzi základnou dekompozíciou a pretínajúcimi záležitosťami (aspektmi)
  - Príkladom je AspectJ
- Symetrické aspektovo-orientované prístupy
  - Program ako celok vzniká spájaním rozličných pohľadov (teda aspektov)
  - Príkladom je Hyper/J

[24] W. H. Harrison, H. L. Ossher, and P. L. Tarr. Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition. Technical report RC22685, IBM Research, 2002.

Symetrický prístup (1)

- Monitorovanie prístupu v syntaxi podobnej jazyku Hyper/J (staršia verzia)
- Monitorovanie implementujeme ako zvláštnu triedu:

```
class PointAccessMonitoring {
    void movingPoint() {System.out.println("Moving a point."); }
    void movedPoint() {System.out.println("Moved a point."); }
}
```

Symetrický prístup (2)

- Potom *zvlášť* definujeme pravidlá spájania (composition rules):

```
namespace AccessMonitoredGraphics{
   AccessMonitoredGraphics.Point.setX :=
      Merge[Graphics.Point.setX, Monitoring.PointAccessMonitoring.movingPoint]

   AccessMonitoredGraphics.Point.setY :=
      Merge[Graphics.Point.setY, Monitoring.PointAccessMonitoring.movingPoint]
   ...
}
```

- Hyper/J používa koncept priestoru mien (namespace) – rôzne aspekty tej istej záležitosti potom môžu byť pod rovnakým názvom
- Predpokladáme, že trieda `Point` je definovaná v priestore mien `Graphics`
- Získame novú triedu `Point` s pripojeným kódom pre monitorovanie

Sumarizácia

Sumarizácia

- Konfigurovateľnosť softvéru je dôležitá predovšetkým pri vývoji radov softvérových výrobkov – vplýva na výber implementačných mechanizmov
- Na konfigurovateľnosť treba myslieť od začiatku
- Rady softvérových výrobkov a doménové inžinierstvo ako organizovaný prístup k znovupoužitiu
- Modelovanie vlastností ako technika na zachytenie variability a konfigurovanie ostatných modelov a kódu
- Uplatnenie aspektovo-orientovaného vývoja softvéru na zachovanie prípadov použitia
- Uplatnenie aspektovo-orientovanej kompozície na zlepšenie konfigurovateľnosti

Odporúčaná literatúra

- K. Czarnecki. Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. Ph.D. Thesis, Computer Science Department, Technical University of Ilmenau, Ilmanau, Germany, 1998. `http://www.issi.uned.es/doctorado/generative/Bibliografia/TesisCzarnecki.pdf`
  - Kapitola 3 – po s. 45
  - Kapitola 5 – bez časti 5.4.1.8
- V. Vranić. Reconciling Feature Modeling: A Feature Modeling Metamodel. In M. Weske and P. Liggesmeyer, Eds., *Proc. of 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays 2004)*, Erfurt, Germany, Sept. 2004. Springer.
- Kapitola 16 z knihy V. Vranić. Objektovo-orientované programovanie: Objekty, Java a aspekty. STU, 2008.