MODELOVANIE SOFTVÉRU

Prípady použitia, UML a ďalej

Valentino Vranić

SPEKTRUM STU

Modelovanie softvéru

Prípady použitia, UML a ďalej

Valentino Vranić

Modelovanie softvéru

Prípady použitia, UML a ďalej

SPEKTRUM
STU

Recenzenti: doc. Ing. Miroslav Bureš, Ph.D.
doc. Ing. William Steingartner PhD.

Vydala Slovenská technická univerzita v Bratislave vo Vydavateľstve SPEKTRUM STU, 2022.

Jazyková úprava: Mgr. Aleksandra Vranić

Titulný obrázok: Branislava Vranić

ISBN 978-80-227-5248-0

Ave a Sandre

Obsah

Zoznam obrázkov

Zoznam tabuliek

Predhovor

Táto kniha zhŕňa to, o čo sa delím so študentmi v predmete *Modelovanie softvéru*[1] už viac ako desať rokov. Tento predmet sa pôvodne volal *Metódy a prostriedky špecifikácie* a bol založený primárne na špecifikačnom jazyku Z. Keď som ho začal prednášať, rekoncipoval som ho, aby zohľadňoval realitu vývoja softvéru. Jazyk Z síce umožňuje presné, matematické vyjadrenie špecifikácie, ale trpí problémom prešpecifikácie, lebo vnucuje vyjadrenie modelovaných konceptov matematickými štruktúrami (primárne postupnosťí). Prešpecifikácia je horšia než podšpecifikácia alebo nepresná špecifikácia. Navyše, už je nejakú dobu známe, že nepresnosť špecifikácie nepredstavuje až taký problém, ako pochopenie toho, čo zákazník chce, vrátane toho, aby to pochopil aj samotný zákazník.

UML je štandardom v dnešnom modelovaní softvéru. Preto sa táto kniha venuje UML, a to vrátane OCL, ale aj niektorým menej známym možnostiam UML. Dôležitejšie než UML sú prípady použitia, ktoré – napriek tomu, že sú jednoduché už ako také – v dnešnej dobe čelia ešte väčšiemu zjednodušeniu do tzv. používateľských príbehov. Aj keď UML má byť zjednotený jazyk modelovania, chýba v ňom podpora zachytenia variantnosti, ako aj dôslednejšia podpora vyjadrenia operácií bez predurčenia štruktúry. Preto sa kniha venuje aj modelovaniu vlastností a algebraickej špecifikácii, ktorá – na rozdiel od jazyka Z – neupadá do problému prešpecifikácie.

Ďakujem recenzentom a všetkým spolupracovníkom pri realizácii predmetu *Modelovanie softvéru*. Verím, že táto kniha bude užitočná všetkým, ktorí hľadajú aktuálne poznanie v oblasti modelovania softvéru aj so zohľadnením historických ponaučení.

Bratislava, 22. 11. 2022 Valentino Vranić

[1] `http://fiit.sk/~vranic/msoft/`

1 Úvod

Vývoj softvéru je činnosť, ktorej konečný výsledok – programový kód – je len na krok od myšlienok a má formu textu. Pre zložité a rozsiahle softvérové systémy nie je jednoduché ani lacné napísať zodpovedajúci kód. Chybné rozhodnutia urobené na začiatku majú ďalekosiahle dôsledky. Ťažko sa menia, keď sa na nich postavia ďalšie rozhodnutia a milióny riadkov kódu.

Aj keď sa dá aj priamo programovať, predtým, ako sa dostaneme ku kódu, často potrebujeme urobiť zodpovedné odhady a dohody. Za týmto účelom prirodzene tvoríme rôzne zápisy a grafické schémy ako *modely* – menšie alebo čiastkové reprezentácie želanej skutočnosti. Ak stanovíme a dodržiavame určité pravidlá, vznikajú notácie. Ak ich akceptujú aj iní, zjednoduší sa komunikácia.

Modely analyzujeme, nad nimi diskutujeme, spresňujeme ich, používame ako návod na implementáciu... Nakoniec ich opúšťame a implementácia žije ďalej svojím životom. Alebo ich udržiavame, aby reflektovali aj posledné zmeny v implementácii, aby sme sa pomocou nich mohli orientovať v implementácii.

Model lode sa nikdy nestane loďou a nikdy nevypláva na oceán. Rovnako sa ani model budovy nestane budovou. Model softvérového systému sa však postupným spresňovaním môže stať samotným softvérovým systémom. Nakoniec, programový kód možno chápať ako konečný a vykonateľný model softvérového systému.

Táto kniha prináša paletu etablovaných techník modelovania softvéru, ktoré možno kombinovať, aby vznikol jasný obraz predsavzatého softvérového systému. Nevysvetľuje všetky ich detaily, ale prostredníctvom príkladov ozrejmuje ich podstatu. Nie je nutné, aby tieto techniky využívali v poradí, v akom sú uvedené.

Kniha je organizovaná nasledovne:

Kapitola 2 vysvetľuje podstatu prípadov použitia ako jednoduchej a lacnej, ale mimoriadne účinnej techniky modelovania softvéru vo forme textu.

Kapitola 3 ozrejmuje spôsob modularizácie prípadov použitia a ich vyjadrenia v Unified Modeling Language (UML).

Kapitola 4 vysvetľuje čo v architektúre softvéru vyplýva z prípadov použitia, a čo je dané hlbšími poznatkami o doméne.

Kapitola 5 ukazuje, ako modularizovať a konceptualizovať štruktúru v UML.

Kapitola 6 prezentuje stavové diagramy v UML.

Kapitola 7 vysvetľuje tvorbu detailného modelu operácie v UML a ako operáciu špecifikovať nepriamo prostredníctvom podmienok vyjadrených v Object Constraint Language (OCL).

Kapitola 8 prezentuje algebraickú špecifikáciu ako spôsob modelovania triedy bez špecifikácie štruktúry.

Kapitola 9 vysvetľuje variantnosť softvéru a jej zachytenie pomocou modelovania vlastností.

2 Podstata prípadov použitia a ich základné vyjadrenie

Modely najčastejšie vytvárame preto, aby sme bez veľkej investície zhmotnili určité aspekty veci, ktorú potrebujeme vytvoriť. Následne model môžeme študovať a vyhodnocovať, a na základe toho upravovať, aby sme nakoniec na základe modelu mohli vytvoriť to, čo je skutočne potrebné. Prípady použitia sú jednoduchou a lacnou technikou modelovania softvéru vo forme textu, ktorá je však mimoriadne účinná. Táto technika bola knižne publikovaná v roku 1992 [Jac92], ale používali sa už od roku 1967 [Jac04].

Časť 2.1 ukazuje ako prípady použitia zachytávajú typické interakcie so systémom. Časť 2.2 vysvetľuje pozíciu predpokladov a dôsledkov prípadov použitia. Časť 2.3 sa zmieňuje o účastníkoch prípadov použitia. V časti 2.4 je pojednané o vzťahu prípadu použitia a koncového používateľa. Časť 2.5 vysvetľuje, prečo prípad použitia nie je opis používateľského rozhrania.

2.1 Prípady použitia zachytávajú typické interakcie so systémom

Predstavte si, že máte vytvoriť elektronický obchod (e-obchod) na základe zoznamu požiadaviek, ktorý by mohol vyzerať takto:

1. Systém musí umožniť zákazníkovi vyhľadať výrobok.
2. Systém musí umožniť zákazníkovi nastaviť počet položiek, ktoré sa naraz zobrazujú.
3. Systém musí umožniť zákazníkovi objednať výrobok.
4. Systém musí umožniť zákazníkovi zrušiť objednávku.

⋮

99. Systém musí umožňovať expedovať objednávku.

⋮

125. Do systému musí byť možné zadať nové druhy výrobku.

⋮

Nepochybne v tomto zozname požiadaviek vidíme, *čo* sa od e-obchodu očakáva. Nevidíme však *ako* to má byť zabezpečené. Napríklad, ako prebieha objednanie výrobku, resp. aká je postupnosť krokov, ktorá k objednaniu vedie. Inak povedané, nevidíme, ako by mala vyzerať interakcia používateľa so systémom. Nevidíme ani súvis medzi požiadavkami. Napríklad, že zákazník výrobok, ktorý objednáva, môže vyhľadať na základe jeho vlastností.

Aké typické interakcie so systémom – prípady jeho použitia – by sme mohli identifikovať v e-obchode? Výber výrobku do objednávky, zadanie mena a adresy alebo uloženie údajov určite predstavujú typické interakcie so systémom, ale používateľ ich určite nevníma ako ucelené úlohy, ktoré si kladie za cieľ. Zabezpečenie životného cyklu objednávky tiež predstavuje typickú interakciu so systémom, ale na príliš vysokej úrovni abstrakcie. To, čo vystihuje ciele používateľa, leží medzi takýmito sumárnymi cieľmi a pomocné funkcie, akou je zadanie mena a adresy [Coc00]. Napríklad: *Zadaj objednávku*, *Vyhľadaj výrobok*, *Vytvor výrobok*, *Expeduj objednávku*... Toto sú *prípady použitia*, ktoré predovšetkým potrebujeme vystihnúť.

Ak by sme mali e-obchod rýchlo sprevádzkovať, ktorý z prípadov použitia by sme mali realizovať ako prvý? Je jasné, že e-obchod bez možnosti objednávania výrobku profit neprinesie. Výrobky zo začiatku môžu byť prezentované v zozname, ktorý zadáme napevno. Vyhľadávanie vynecháme, expedovanie si poznačíme do externej tabuľky... Určite to nie je ten najlepší e-obchod, ale môže začať zarábať.

Samotný prípad použitia *Zadaj objednávku* by mohol vyzerať nasledovne:

Prípad použitia: Zadaj objednávku

Zákazník vyberá výrobok do košíka. Ten sa stáva súčasťou jeho objednávky, ktorú nakoniec potvrdí, čím sa objednávka presunie na expedovanie.

1. Zákazník zvolí zadanie objednávky.
2. Systém zobrazí možnosti vyhľadávania.
3. Zákazník nastaví možnosti vyhľadávania a spustí vyhľadávanie.
4. Systém zobrazí vyhľadané položky.
5. Zákazník vyberie z vyhľadaných položiek a potvrdí výber.
6. Systém vloží zvolený výrobok do košíka.
7. Zákazník môže pokračovať vo výbere výrobkov – prípad použitia pokračuje krokom 2.

8. Zákazník objedná výrobky v košíku.

9. Systém vyžiada údaje potrebné na realizáciu objednávky vrátane spôsobu platby.

10. Zákazník zadá požadované platobné údaje.

11. Kedykoľvek počas objednávania, zákazník môže vzdať tento proces.

12. Systém uloží objednávku do zoznamu objednávok na expedovanie.

13. Pre každý výrobok v objednávke, systém zistí stav zásob.

14. Systém uloží záznam do plánu doplnenia zásob o potrebe zvýšenia stavu každého objednaného výrobku, ak by jeho stav po expedovaní poklesol pod stanovený limit.

15. Prípad použitia končí.

Predpoklady: zákazník je prihlásený

Dôsledky:

- Minimálne: výrobok, ktoré pôvodne boli súčasťou objednávky, sú v nej naďalej
- V prípade úspechu: výrobky, ktoré zákazník chce objednať sú súčasťou objednávky

Názov prípadu použitia je v imperatíve (prikazovací spôsob), aby sme nezabudli, na to, že sledujeme dosiahnutie určitého cieľa. Nech je akokoľvek výstižný, názov je dobré rozviesť v krátkom opise prípadu použitia.

Podstatou prípadu použitia je interakcia účastníka (angl. actor) so systémom, a tu je najjednoduchšie vyjadriť vo forme postupnosti krokov. Táto postupnosť sa označuje ako tok udalostí alebo jednoducho iba tok. Prvý krok vystihuje, čím prípad použitia začína. Posledný krok predstavuje formálne ukončenie prípadu použitia.

V ostatných krokoch sa striedajú akcie na strane účastníka s akciami na strane systému. Niektoré akcie na strane systému bývajú vynechané, aby opis bol kratší. Aj keď sa snažíme dodržať postupnosť krokov, krok môže čokoľvek, čo prirodzený jazyk dovoľuje. Tak napríklad, krok 7 vyjadruje skok. Krok 11 má nadčasový charakter: hovorí, že kedykoľvek počas objednávania výrobku, zákazník môže vzdať tento proces.

2.2 Predpoklady a dôsledky

Prípad použitia nemusí byť možné realizovať za každých okolností. Tie sa označujú ako *predpoklady*. Väčšinou sú vyjadrené v zmysle iných prípadov použitia, ktoré majú prebehnúť pred daným prípadom použitia.

Výsledok prípadu použitia sa označuje ako *dôsledok*. Dobré je rozlišovať medzi minimálnymi dôsledkami a dôsledkami v prípade úspechu [Coc00].

2.3 Účastníci

V prípade použitia *Zadaj objednávku* vystupuje zákazník. V prípade použitia *Vytvor výrobok* by vystupoval obchodník. Nestačí hovoriť proste „používateľ“? Prostredníctvom účastníkov prípadov použitia si ujasňujeme, kto a ako bude používať systém, ale nemodelujeme tým priamo používateľské práva. V prípadoch použitia ako účastník vystupuje aj systém alebo jeho časti.

2.4 Prípad použitia je koncept vedomia koncového používateľa

Spísanie typických interakcií s vytváraným systémom na úrovni sledovania cieľov používateľa, t. j. prípadov použitia systému, umožňuje odhaliť skutočný zámer klienta a vyjadriť ho zrozumiteľne, ale predsa blízko ku kódu. Prípad použitia formulujeme na základe informácií od klienta alebo – ešte lepšie – koncového používateľa. Inak povedané, prípad použitia je koncept vedomia koncového používateľa, resp. je súčasťou jeho mentálneho modelu [CB10].

Predstava o prípade použitia na strane vývojárov sa môže podstatne líšiť od očakávaní klienta. Napríklad, môžeme domnievať, že sa objednávky majú expedovať jednotlivo, ako aj indikuje názov prípadu použitia *Expeduj objednávku*. Možno to tak klient chce, ale mohol by chcieť aj niečo iné: expedovať viac objednávok naraz, aby mohol prioritne priradiť deficitárny výrobok k hodnotnejším objednávkam alebo významnejším klientom. V takom prípade by sa príslušný prípad použitia mal volať skôr *Expeduj objednávky*. Prostým odsledovaním krokov prípadu použitia klient zistí, či by mu plánovaná implementácia vyhovovala alebo nie. Prípady použitia sú skutočne veľmi lacnou, avšak mimoriadne účinnou technikou modelovania softvéru.

2.5 Prípad použitia nie je opis používateľského rozhrania

Predstavme si situáciu, že obchodník eviduje zákazky v Exceli s pripravenou podporu na úrovni sumárnych výpočtov a pod. Aj keď takáto „implementácia“ nie je úplne

adekvátna, čo postrehneme hlavne na úrovni používateľského rozhrania, predsa v nej možno identifikovať prípad použitie *Zaeviduj zákazku*. Všimnime si, že v opise prípadu použitia nevystupujú prvky používateľského rozhrania.

Prípady použitia ako koncepty vedomia koncových používateľov (ich mentálneho modelu) majú v softvérovom systéme a jeho používateľskom rozhraní viac alebo menej dôslednú podporu, ale nie sú jeho opisom. Jeden prípad použitia môže byť realizovaný jedným alebo viacerými formulármi používateľského rozhrania, kým ten istý formulár môže byť zahrnutý do viacerých realizácií prípadov použitia. Preto prípady použitia nemajú spomínať okná, tlačidlá, menu, označenia alebo iné prvky používateľského rozhrania založeného na oknách, ako ani akéhokoľvek iného druhu používateľského rozhrania vôbec. Spomenutie prvkov používateľského rozhrania tu a tam zvyčajne nepredstavuje problém a často sa s týmto stretávame v praxi [Coc00]. Avšak, aby sme nepadli do pasce pomýlenia prípadov použitia za špecifikáciu používateľského rozhrania, najlepšie je aktívne abstrahovať od hocijakého výskytu prvkov používateľského rozhrania. Napríklad, namiesto toho, aby sme povedali „klikne tlačidlo OK“, môžeme povedať „potvrdí akciu“.

3 Modularizácia prípadov použitia a ich vyjadrenie v UML

Pokus o ošetrenie všetkých možnosti v jednom slede krokov nevyhnutne vedie k zahmleniu podstaty prípadu použitia. Preto je potrebné prípady použitia organizovať – alebo, inak povedané, modularizovať – ako na vnútornej, tak aj na vonkajšej úrovni. Celistvý obraz o systéme vzniká na základe prepojení týchto modulov. Pri väčšom počte prípadov použitia môžeme začať strácať prehľad, a toto je moment, kedy je vhodné zobraziť ich diagramom. UML za týmto účelom poskytuje diagram prípadov použitia.

Časť 3.1 vysvetľuje pojem základného toku a podtoku prípadu použitia. Časť 3.2 vysvetľuje vzťah zahrnutia. Časť 3.3 vysvetľuje pojem alternatívneho toku. Časť 3.4 vysvetľuje vzťah rozšírenia. Časť 3.5 ukazuje, ako možno transformovať zahrnutie na rozšírenie a naopak. Časť 3.6 vysvetľuje vzťah dedenia. Časť 3.7 ukazuje, ako zoskupiť viac podobných hlavných tokov do jedného prípadu použitia. Časť 3.8 vysvetľuje, ako zosúladiť účastníkov pri vzťahu zahrnutia. Časť 3.9 ozrejmuje pozíciu sekundárneho účastníka. Časť 3.10 vymedzuje účel diagramov prípadov použitia. Časti 3.11–3.13 ukazujú, ako sa prípad použitia dá vyjadriť grafickým modelom: diagramom aktivít, diagramom sekvencií a diagramom komunikácie. Časť 3.15 vysvetľuje prípad použitia ako kolaboráciu. Časť 3.16 prináša záverečné poznámky.

3.1 Základný tok a podtok

Tak, ako je uvedený v predchádzajúcej kapitole, prípad použitia *Zadaj objednávku* zahŕňa aj vyhľadanie výrobku. Je to iba niekoľko krokov navyše, ale narúšajú plynulosť vnímania podstaty tohto prípadu použitia, ktorou je proces objednávania výrobku. Tieto kroky by sme mohli vyčleniť do určitého *podtoku*, ktorý aktivujeme na príslušnom mieste *základného toku*:[1]

[1] V prípadoch použitia v tejto kapitole je vynechaný ich krátky opis a predpoklady a dôsledky, lebo nie sú potrebné pre vysvetlenie spôsobu modularizácie. To neznamená, že ich v úplnych modeloch prípadov použitia netreba uvádzať.

Prípad použitia: Zadaj objednávku

Základný tok: Zadaj objednávku

1. Zákazník zvolí zadanie objednávky.
2. Aktivuje sa podtok *Vyhľadaj výrobok*.
3. Systém vloží zvolený výrobok do košíka.
4. Zákazník môže pokračovať vo výbere výrobkov – prípad použitia pokračuje krokom 2.
5. Zákazník objedná výrobky v košíku.
6. Systém vyžiada údaje potrebné na realizáciu objednávky vrátane spôsobu platby.
7. Zákazník zadá požadované platobné údaje.
8. Kedykoľvek počas objednávania, zákazník môže vzdať tento proces.
9. Systém uloží objednávku do zoznamu objednávok na vybavenie.
10. Ak by stav zásob hociktorého výrobku v objednávke po jej expedovaní poklesol pod stanovený limit, systém uloží záznam do plánu doplnenia zásob o potrebe zvýšenia stavu príslušného výrobku.
11. Prípad použitia končí.

Podtok: Vyhľadaj výrobok

1. Systém zobrazí možnosti vyhľadávania.
2. Zákazník nastaví možnosti vyhľadávania a spustí vyhľadávanie.
3. Systém zobrazí vyhľadané položky.

Podtok sa samostatne neaktivuje, ani neukončuje prípad použitia, a preto neobsahuje aktivačný a ukončovací krok ako základný tok. Pomenovanie základného toku možno vynechať.

3.2 Zahrnutie

Vyčlenený podtok môže byť užitočný aj pre iné prípady použitia, a preto by sme ho mohli vyjadriť ako prípad použitia:

Prípad použitia: Vyhľadaj výrobok

Podtok: Vyhľadaj výrobok

1. Systém zobrazí možnosti vyhľadávania.
2. Zákazník nastaví možnosti vyhľadávania a spustí vyhľadávanie.
3. Systém zobrazí vyhľadané položky.

Takýto prípad použitia nie je určený na samostatnú aktiváciu. Aktiváciu je potrebné upraviť, aby zohľadňovala skutočnosť, že tok, ktorý *zahŕňame* (include), je už v inom prípade použitia:

Prípad použitia: Zadaj objednávku

Základný tok: Zadaj objednávku

1. Zákazník zvolí zadanie objednávky.
2. Aktivuje sa prípad použitia *Vyhľadaj výrobok*, jeho rovnomenný podtok.
3. Systém vloží zvolený výrobok do košíka.
4. Zákazník môže pokračovať vo výbere výrobkov – prípad použitia pokračuje krokom 2.
5. Zákazník objedná výrobky v košíku.
6. Systém vyžiada údaje potrebné na realizáciu objednávky vrátane spôsobu platby.
7. Zákazník zadá požadované platobné údaje.
8. Kedykoľvek počas objednávania, zákazník môže vzdať tento proces.
9. Systém uloží objednávku do zoznamu objednávok na vybavenie.
10. Ak by stav zásob hociktorého výrobku v objednávke po jej expedovaní poklesol pod stanovený limit, systém uloží záznam do plánu doplnenia zásob o potrebe zvýšenia stavu príslušného výrobku.
11. Prípad použitia končí.

Tento vzťah medzi prípadmi použitia sa označuje ako *zahrnutie* (angl. *include*) a zodpovedá volaniu metódy, ako je naznačené v tomto kóde v Jave:

```
public class Objednavanie {
    ...
    public void objednaj(Vyrobok vyrobok, int mnozstvo) {
        ...
        new VyhladavanieVyrobkov().vyhladaj(vyrobok);
        ...
        if (zistiStavZasob(vyrobok) >= mnozstvo) {
            ...
        } else ...
```

```
    }
...
}
```

Ak by prípad použitia *Vyhľadaj výrobok* malo význam aktivovať aj samostatne, potrebné je pridať základný tok:

Prípad použitia: Vyhľadaj výrobok

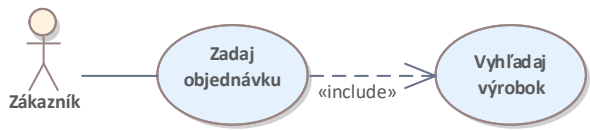
Základný tok: Vyhľadaj výrobok

1. Zákazník zvolí zadanie objednávky.
2. Aktivuje sa podtok *Vyhľadaj výrobok*.
3. Prípad použitia končí.

Podtok: Vyhľadaj výrobok

1. Systém zobrazí možnosti vyhľadávania.
2. Zákazník nastaví možnosti vyhľadávania a spustí vyhľadávanie.
3. Systém zobrazí vyhľadané položky.

Vzťah zahrnutia možno znázorniť graficky ako na obrázku 3.1. Toto je diagram prípadov použitia v UML. Všimníme si, že šípka smeruje od prípadu použitia *Zadaj objednávku* k prípadu použitia *Vyhľadaj výrobok*, čím je vyjadrené to, že prípad použitia *Zadaj objednávku* pozná prípad použitia *Vyhľadaj výrobok*. Z textu je to zjavné: prípad použitia *Zadaj objednávku* sa odkazuje na prípad použitia *Vyhľadaj výrobok*.



Obr. 3.1: Vzťah zahrnutia.

Všimnime si, že v diagrame ako účastník nevystupuje systém. Mohli by sme ho tam uviesť, ale musel by byť spojený s každým prípadom použitia, čo by značne znehľadnilo diagram. Ak namiesto jednoliateho systému uvádzame jeho časti alebo modelujeme interakciu s viacerými prepojenými systémami, môže byť potrebné ich uviesť. Treba myslieť na to, že diagramy prípadov použitia nie sú určené na modelovanie súvislosti medzi systémami alebo komponentmi. Na to sú vhodnejšie diagramy komponentov a kompozitnej štruktúry, ktorými sa zaoberá kapitola 5.

3.3 Alternatívny tok

Ku koncu základného toku prípadu použitia *Zadaj objednávku* sa uskutočňuje kontrola stavu zásob výrobkov v objednávke. Tento krok taktiež narúša plynulosť vnímania podstaty tohto prípadu použitia. Na rozdiel od vyhľadania výrobku, ktorého spomenutie v prípade použitia *Zadaj objednávku* pomáha pochopiť celý proces, kontrola stavu zásob takúto povahu nemá. Môžeme ju preto nielen uviesť mimo základného toku, ale aj podmienky jej aktivácie môžeme stanoviť mimo neho:

Prípad použitia: Zadaj objednávku

Základný tok: Zadaj objednávku

1. Zákazník zvolí zadanie objednávky.

2. Aktivuje sa prípad použitia *Vyhľadaj výrobok*, podtok *Vyhľadaj výrobok*.

3. Systém vloží zvolený výrobok do košíka.

4. Zákazník môže pokračovať vo výbere výrobkov – prípad použitia pokračuje krokom 2.

5. Zákazník objedná výrobky v košíku.

6. Systém vyžiada údaje potrebné na realizáciu objednávky vrátane spôsobu platby.

7. Zákazník zadá požadované platobné údaje.

8. Kedykoľvek počas objednávania, zákazník môže vzdať tento proces.

9. Systém uloží objednávku do zoznamu objednávok na vybavenie.

10. Prípad použitia končí.

Alternatívny tok: Modifikuj plán doplnenia zásob

V kroku 9 základného toku, ak by stav zásob hociktorého výrobku v objednávke po jej expedovaní poklesol pod stanovený limit:

1. Systém uloží záznam do plánu doplnenia zásob o potrebe zvýšenia stavu každého výrobku v objednávke, ktorého stav zásob by po jej expedovaní poklesol pod stanovený limit.

Takéto *alternatívne toky* môžu aj úplne nahrádzať kroky alebo celé rozsahy krokov.

3.4 Rozšírenie

Tak ako podtok, aj vyčlenený alternatívny tok môže byť užitočný aj pre iné prípady použitia, a preto by sme ho mohli vyjadriť ako prípad použitia:

> **Prípad použitia: Modifikuj plán doplnenia zásob**
>
> *Alternatívny tok: Modifikuj plán doplnenia zásob*
>
> V bode rozšírenia *Uloženie objednávky* základného toku prípadu použitia *Zadaj objednávku*, ak by stav zásob hociktorého výrobku v objednávke po jej expedovaní poklesol pod stanovený limit:
>
> 1. Systém uloží záznam do plánu doplnenia zásob o potrebe zvýšenia stavu každého výrobku v objednávke, ktorého stav zásob by po jej expedovaní poklesol pod stanovený limit.

Aby nevznikali problémy pri prípadných úpravách prípadu použitia *Zadaj objednávku*, pri ktorých by mohlo dôjsť k prečíslovaniu krokov, namiesto priameho odkazu na príslušný krok použijeme jeho pomenovanie:

> **Prípad použitia: Zadaj objednávku**
>
> *Základný tok: Zadaj objednávku*
>
> 1. Zákazník zvolí zadanie objednávky.
> 2. Aktivuje sa prípad použitia *Vyhľadaj výrobok*, jeho rovnomenný podtok.
> 3. Systém vloží zvolený výrobok do košíka.
> 4. Zákazník môže pokračovať vo výbere výrobkov – prípad použitia pokračuje krokom 2.
> 5. Zákazník objedná výrobky v košíku.
> 6. Systém vyžiada údaje potrebné na realizáciu objednávky vrátane spôsobu platby.
> 7. Zákazník zadá požadované platobné údaje.
> 8. Kedykoľvek počas objednávania, zákazník môže vzdať tento proces.
> 9. Systém uloží objednávku do zoznamu objednávok na vybavenie.
> 10. Prípad použitia končí.
>
> *Body rozšírenia:*
>
> - Vloženie výrobku do košíka: krok 3

- Uloženie objednávky: krok 9

Tento vzťah medzi prípadmi použitia sa označuje ako *rozšírenie* (angl. extend). Krok (alebo rozsah krokov), v ktorom dochádza k rozšíreniu, sa označuje ako *bod rozšírenia* (angl. extension point). Samotné zavedenie bodu rozšírenia, ako je to s bodom rozšírenia *Vloženie výrobku do košíka*, nemá vplyv na daný prípad použitia.

Vzťah rozšírenia už nezopdovedá volaniu metódy. V Jave ako takej by sa však musel implementovať volaním metódy:

```
public class Objednavanie {
    ...
    public void objednaj(Objednavka objednavka) {
        ...
        ulozObjednavku(objednavka);
        for (Vyrobok vyrobok : objednavka.objednanePolozky)
            if (planZasob.zistiStavZasob(vyrobok)
                < planZasob.minimalneZasoby(vyrobok) + vyrobok.mnozstvo) {
                planZasob.dopln(vyrobok);
    }
    ...
}
```

V jazyku AspectJ, ktorý podporuje aspektovo-orientované programovanie, možno triedu `Objednavanie` oslobodiť od záležitosti kontroly stavu zásob:

```
public class Objednavanie {
    ...
    public void objednaj(Vyrobok vyrobok, int mnozstvo) {
        ...
        ulozObjednavku();
    }
    ...
}
```

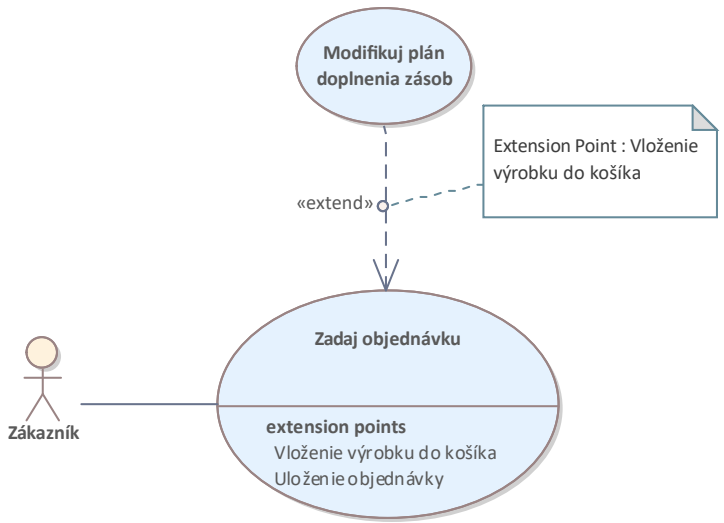
a kontrolu stavu zásob implementovať vo forme aspektu, ktorý zasiahne po uložení objednávky:

```
public aspect PlanDoplneniaZasob {
    ...
    void after(Objednavanie objednavanie, Objednavka objednavka):
        this(objednavanie)
        && call(* Objednavanie.ulozObjednavku(Objednavka))
```

```
        && args(objednavka) {
        for (Vyrobok vyrobok : objednavka.objednanePolozky)
            if (objednavanie.planZasob.zistiStavZasob(vyrobok)
                < vyrobok.minimalneZasoby() + vyrobok.mnozstvo) {
                objednavanie.planZasob.dopln(vyrobok;
    }
}
```

Volanie metódy `ulozObjednavku`() je v tomto prípade v terminológii aspektovo-orientovaného programovania *bod spájania* (angl. join point). Ako vidíme, body spájania sú analógiou bodov rozšírenia. Rozširujúci prípad použitia je potom aspektom rozšíreného prípadu použitia. Rozsiahlejšie o tomto píšu Jacobson a Ng [JN04].

Vzťah rozšírenia možno znázorniť graficky ako na obrázku 3.2. Ako aj pri vzťahu zahrnutia, šípka znovu smeruje vždy od toho prípadu použitia, ktorý pozná ten iný prípad použitia. Tentokrát to nie je prípad použitia *Zadaj objednávku*, ale prípad použitia *Modifikuj plán doplnenia zásob*. Znovu, z textu je to zjavné: prípad použitia *Modifikuj plán doplnenia zásob* sa odkazuje na prípad použitia *Zadaj objednávku*.



Obr. 3.2: Vzťah rozšírenia.

Vzťah rozšírenia by sa dal vyjadriť aj bez explicitného viazania na body rozšírenia alebo kroky:

Prípad použitia: Modifikuj plán doplnenia zásob

Alternatívny tok: Modifikuj plán doplnenia zásob

Kedykoľvek sa v systéme ukladá objednávka, ak by stav zásob hociktorého výrobku v nej po jej expedovaní poklesol pod stanovený limit:

1. Systém uloží záznam do plánu doplnenia zásob o potrebe zvýšenia stavu každého výrobku v objednávke, ktorého stav zásob by po jej expedovaní poklesol pod stanovený limit.

Nevýhodou takéhoto vyjadrenia je, že zistenie, na ktoré kroky sa vzťahuje, vyžaduje analýzu celého modelu prípadov použitia. V tomto môže pomôcť obmedzenie rozsah platnosti len na určité prípady použitia – v našom prípade na prípad použitia *Zadaj objednávku*:

Prípad použitia: Modifikuj plán doplnenia zásob

Alternatívny tok: Modifikuj plán doplnenia zásob

Kedykoľvek sa v prípade použitia Zadaj objednávku ukladá objednávka, ak by stav zásob hociktorého výrobku v nej po jej expedovaní poklesol pod stanovený limit:

1. Systém uloží záznam do plánu doplnenia zásob o potrebe zvýšenia stavu každého výrobku v objednávke, ktorého stav zásob by po jej expedovaní poklesol pod stanovený limit.

Doterajšie príklady prípadov použitia sú vyjadrené v Jacobsonovej notácii [JN04]. Takéto deklaratívne vyjadrenie rozšírenia je príznačné pre Cockburnovu notáciu [Coc00]. Konvencie a prvky notácií modelovania prípadov použitia možno kombinovať [VuZ13], a tak vytvárať vlastné notácie.

3.5 Tranasformácia zahrnutia na rozšírenie a naopak

Rozšírenie možno veľmi jednoducho transformovať na zahrnutie:

Prípad použitia: Zadaj objednávku

Základný tok: Zadaj objednávku

1. Zákazník zvolí zadanie objednávky.
2. Aktivuje sa prípad použitia *Vyhľadaj výrobok*, jeho rovnomenný podtok.
3. Systém vloží zvolený výrobok do košíka.

4. Zákazník môže pokračovať vo výbere výrobkov – prípad použitia pokračuje krokom 2.

5. Zákazník objedná výrobky v košíku.

6. Systém vyžiada údaje potrebné na realizáciu objednávky vrátane spôsobu platby.

7. Zákazník zadá požadované platobné údaje.

8. Kedykoľvek počas objednávania, zákazník môže vzdať tento proces.

9. Systém uloží objednávku do zoznamu objednávok na vybavenie.

10. Aktivuje sa prípad použitia *Modifikuj plán doplnenia zásob*, jeho rovnomenný podtok.

11. Prípad použitia končí.

Alternatívny tok v prípade použitia *Modifikuj plán doplnenia zásob* sa zmení na podtok. Časť podmienky aktivácie sa stáva súčasťou jediného kroku tohto podtoku, t. j. vraciame sa k zneniu tohto kroku kým ešte bol súčasťou prípadu použitia *Zadaj objednávku* (pozri časť 3.1):

Prípad použitia: Modifikuj plán doplnenia zásob

Podtok: Modifikuj plán doplnenia zásob

1. Ak by stav zásob hociktorého výrobku v objednávke po jej expedovaní poklesol pod stanovený limit, systém uloží záznam do plánu doplnenia zásob o potrebe zvýšenia stavu príslušného výrobku.

Samozrejme, táto transformácia sa dá spraviť aj naopak, t. j. zahrnutie sa dá transformovať na rozšírenie. Mohli by sme, teda, transformovať zahrnutie prípadu použitia *Vyhľadaj výrobok* na rozšírenie. Otázkou je, prečo sme tu uprednostnili vzťah zahrnutia pred rozšírením, a pri prípade použitia *Modifikuj plán doplnenia zásob* naopak. Kľúč je v tom, ako veľmi potrebujeme vidieť danú záležitosť v základnom toku. Vyhľadanie výrobku nám akosi bude chýbať, aby sme pochopili, aký to výrobok zákazník vkladá do košíka. Na druhej strane, proces vytvorenia objednávky pohodlne odsledujeme bez potreby, aby sme hneď vedeli o tom, že sa plán doplnenia zásob modifikuje. Samozrejme, voľba medzi zahrnutím a rozšírením nemusí byť taká jasná v všetkých situáciách.

3.6 Dedenie

Prípad použitia môže špecializovať už definovaný prípad použitia, t. j. môže od neho dediť. V opise sa uvedie, ktoré kroky prekonáva. Napríklad takto:

Zadaj rýchlu objednávku

Prípad použitia špecializuje prípad použitia *Zadaj objednávku*. Kroky sú prekonané nasledovne:

1. Zákazník zvolí urýchlené objednanie výrobku.
2. Zákazník priamo zadá kód výrobku.

Ostatné kroky zostávajú v platnosti.

Ako ukazuje obrázok 3.3, v diagramoch prípadov použitia sa tento vzťah značí ako generalizácia/špecializácia (angl. generalization/specialization), čo je UML termín pre vzťah vo vývoji softvéru známy ako dedenie (angl. inheritance).

Obr. 3.3: Dedenie medzi prípadmi použitia.

3.7 CRUD prípad použitia

V e-obchode je potrebné aj evidovať výrobok. S tým sú spojené operácie vytvorenia, zobrazenia, úpravy a vyradenia výrobku, t. j. typické operácie nad údajovou entitou známe pod skratkou CRUD: create–read–update–delete. Pre každú z nich by mohol vzniknúť samostatný prípad použitia, ale to by len zahlcovalo model prípadov použitia nie príliš zaujímavými prípadmi použitia. Jacobson a Ng navrhujú združiť tieto operácie do jedného prípadu použitia, označovaného ako CRUD prípad použitia, ktorý potom bude obsahovať základný tok pre každú z nich [JN04]. V našom systéme by to mohlo vyzerať takto s tým, že namiesto *CRUD výrobok* prípad použitia pomenujeme *Spravuj výrobok*:

Prípad použitia: Spravuj výrobok

Základný tok: Vytvor výrobok

1. Obchodník zvolí vytvorenie výrobku.
2. Systém vyžiada údaje o výrobku.
3. Obchodník zadá názov, typ a obrázok výrobku.
4. Systém prispôsobí veľkosť obrázku výrobku štandardnej veľkosti.
5. Obchodník zaradí výrobok do kategórie.

6. Ak obchodník potvrdí zadané údaje, systém ich uloží.
7. Prípad použitia končí.

Základný tok: Zobraz výrobok

1. Obchodník zvolí zobrazenie jestvujúceho výrobku.
2. Aktivuje sa prípad použitia *Vyhľadaj výrobok*, jeho rovnomenný tok.
3. Systém zobrazí vybraný výrobok.
4. Obchodník prezrie výrobok.
5. Prípad použitia končí.

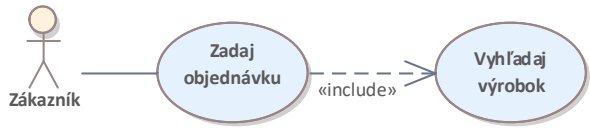
Základný tok: Uprav výrobok

1. Obchodník zvolí úpravu jestvujúceho výrobku.
2. Aktivuje sa prípad použitia Vyhľadaj výrobok.
3. Systém otvorí vybraný výrobok a umožní jeho úpravu.
4. Obchodník upraví údaje o výrobku.
5. Ak obchodník potvrdí zmeny, systém ich uloží.
6. Prípad použitia končí.

Základný tok: Vyraď výrobok

1. Obchodník zvolí vyradenie výrobku.
2. Aktivuje sa prípad použitia *Vyhľadaj výrobok*, jeho rovnomenný tok.
3. Ak obchodník potvrdí vyradenie výrobku, systém na danom výrobku nastaví príznak vyradenia.
4. Prípad použitia končí.

Ako je možné vidieť na obrázku 3.4, aj v diagrame takýto CRUD prípad použitia vystupuje ako jeden prípad použitia.



Obr. 3.4: CRUD prípad použitia.

CRUD prípad použitia nemusí zahŕňať všetky štyri operácie. Napríklad, zrušenie často nie je aplikovateľné. Navyše, tento prístup sa dá uplatniť na akýkoľvek systém príbuzných operácií.

3.8 Účastníci a zahrnutie

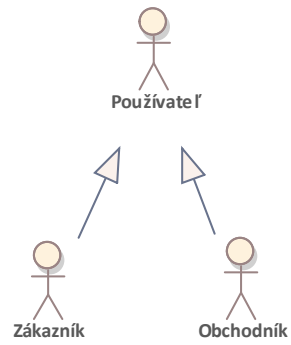
V prípade použitia *Vyhľadaj výrobok* – ako je uvedný v časti 3.2 – vystupuje zákazník, ale toky prípadu použitia *Spravuj výrobok*, v ktorých vystupuje obchodník, tiež zahŕňajú tento prípad použitia. Mohli by sme v pípade použitia *Vyhľadaj výrobok* použiť formuláciu „zákazník alebo obchodník“, ale typy účastníkov môžu pribúdať. Lepšie je zovšeobecniť účastníka:

Prípad použitia: Vyhľadaj výrobok

Podtok: Vyhľadaj výrobok

1. Systém zobrazí možnosti vyhľadávania.
2. Používateľ nastaví možnosti vyhľadávania a spustí vyhľadávanie.
3. Systém zobrazí vyhľadané položky.

K modelu prípadov použitia by sme mali poskytnúť vysvetlenie, že zákazník a obchodník sú druhmi používateľa alebo použijeme diagram ako na obrázku 3.5. Tak ako pri dedení medzi prípadmi použitia (pozri časť 3.6), v diagrame použijeme vzťah generalizácie/špecializácie.



Obr. 3.5: Zovšeobecnenie účastníkov.

3.9 Sekundárny účastník

V jednom prípade použitia môže vystupovať viac účastníkov. Predstavme si, že objednávku má potvrdiť obchodník. Či to tak má byť, závisí od klienta, t. j. prevádzkovateľa e-obchodu. Prípad použitia *Zadaj objednávku* by potom vyzeral takto:

Prípad použitia: Zadaj objednávku

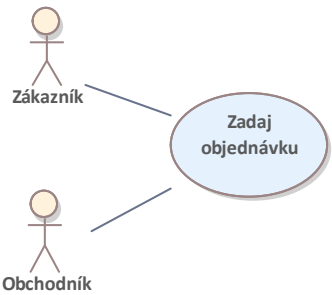
Základný tok: Zadaj objednávku

1. Zákazník zvolí zadanie objednávky.
2. Aktivuje sa prípad použitia *Vyhľadaj výrobok*, jeho rovnomenný tok.
3. Systém vloží zvolený výrobok do košíka.
4. Zákazník môže pokračovať vo výbere výrobkov – prípad použitia pokračuje krokom 2.
5. Zákazník objedná výrobky v košíku.
6. Systém vyžiada údaje potrebné na realizáciu objednávky vrátane spôsobu platby.
7. Zákazník zadá požadované platobné údaje.
8. Kedykoľvek počas objednávania, zákazník môže vzdať tento proces.
9. Systém uloží objednávku do zoznamu objednávok na vybavenie.
10. Obchodník skontroluje objednávku a potvrdí, že objednávka bude spracovaná alebo ju zamietne (ak usúdi, že sa nedá realizovať).
11. Systém uloží informáciu o začatí spracovania objednávky alebo jej zamietnutí a informuje zákazníka.
12. Prípad použitia končí.

Body rozšírenia:

- Vloženie výrobku do košíka: krok 3
- Uloženie objednávky: krok 9

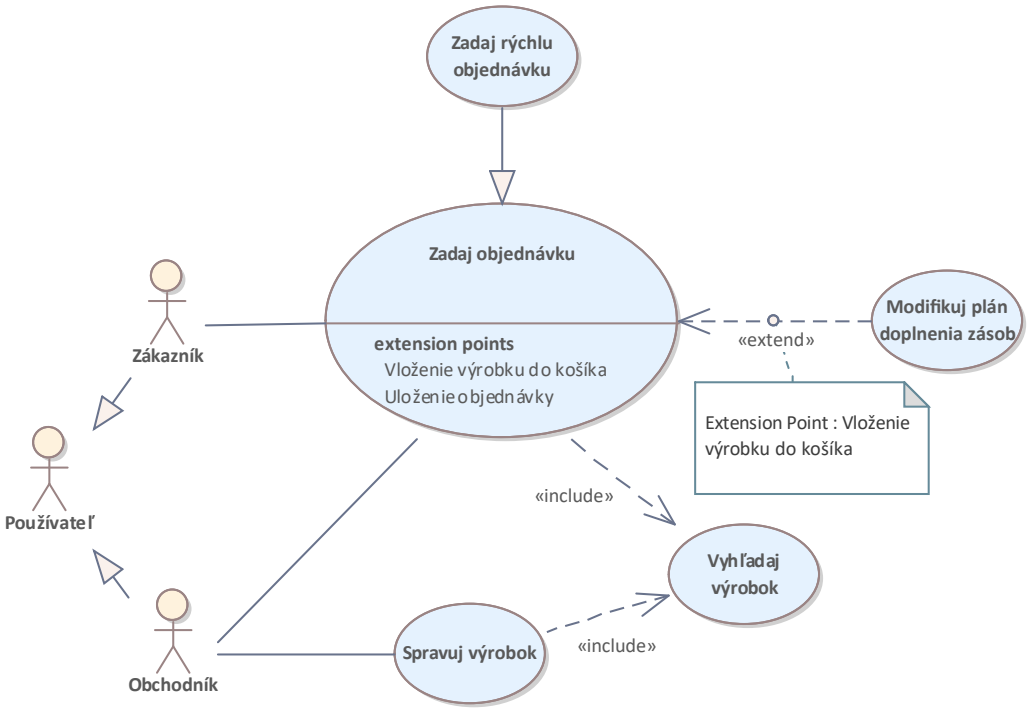
Obchodník je tu iba sekundárnym účastníkom. Zákazník zostáva primárnym účastníkom, lebo prípad použitia aktivuje. Ako možno vidieť na obrázku 3.6, diagram prípadov použitia medzi primárnym a sekundárnym účastníkom (alebo účastníkmi) nerozlišuje.



Obr. 3.6: Diagram prípadov použitia medzi primárnym a sekundárnym účastníkom nerozlišuje.

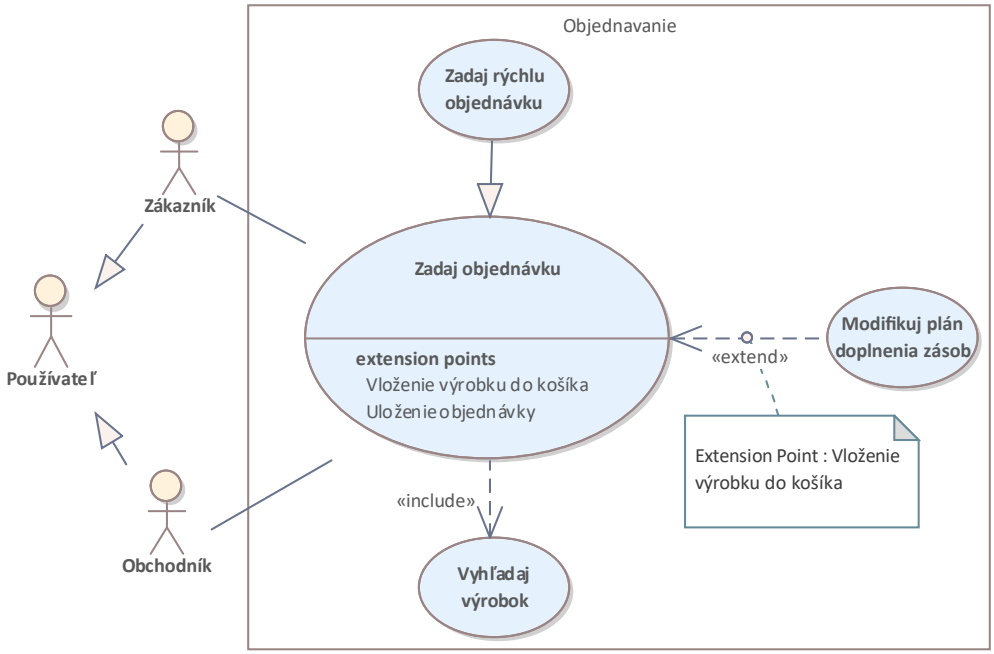
3.10 Účel diagramov prípadov použitia

Model sa skomplikoval. Diagram prípadov použitia na obrázku 3.7 zobrazuje všetky prípady použitia, ktoré sme identifikovali v e-obchode.



Obr. 3.7: Celkový diagram prípadov použitia.

Prípadov použitia môže byť viac, a preto je vhodné ich zobrazovať vo viacerých diagramoch podľa predmetu, na ktorý sa zameriavajú. Pre e-shop by to mohli byť objednávanie, expedovanie, správa výrobkov, správa zákazníkov atď. Predmet sa niekedy vyznačuje orámovaním. Príklad je na obrázku 3.8.



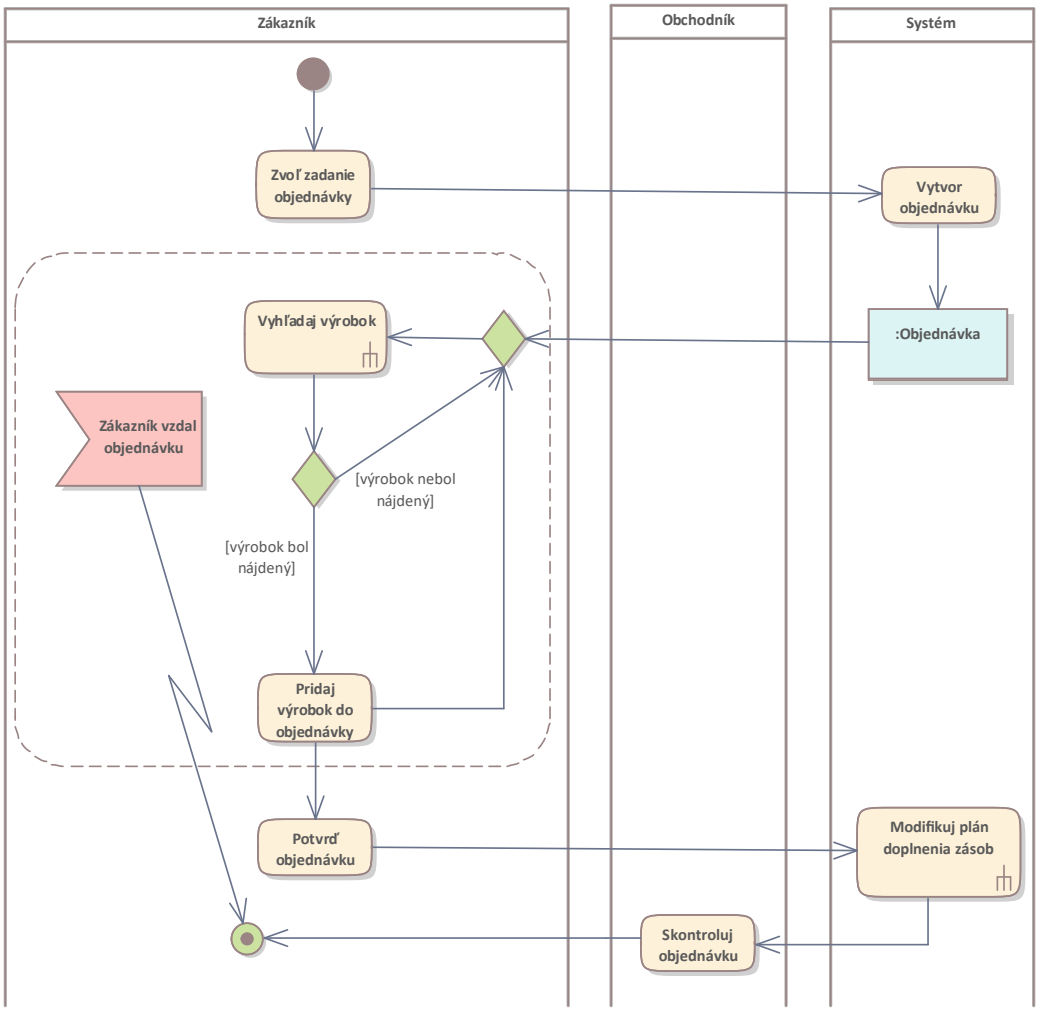
Obr. 3.8: Diagram prípadov použitia, ktoré sa vzťahujú na objednávanie.

Tie isté prípady použitia sa môžu vyskytovať v rôznych predmetoch. Samotný predmet môže indikovať určitý modul softvérového systému (balík, komponent, rámec. . .), ale nemusí mať priamy obraz v implementácii.

Diagram prípadov použitia je vhodný ako prehľad, ale nie je hlavnou časťou modelu prípadov použitia.

3.11 Vyjadrenie prípadu použitia diagramom aktivít

Grafická reprezentácia postupnosti činností alebo krokov, kde činnosti sú reprezentované uzlami, je veľmi intuitívna. UML na to poskytuje diagram aktivít. Príklad využitia diagramu aktivít na zachytenie prípadu použitia je na obrázku 3.9.



Obr. 3.9: Diagram aktivít pre prípad použitia *Zadaj objednávku*.

Akcie (činnosti) sú jednoducho spojené hranami tak, ako nasledujú. Každá akcia vysiela značku (angl. token) cez každú hranu, ktorá z nej vychádza. Akcia sa aktivuje iba vtedy, keď prijme značku cez každú hranu, ktorá do nej vstupuje.

Medzi aktivitami sa môže vyskytnúť objekt, ktorý sa medzi nimi prenáša, ako je to prípad s objektom typu *Objednávka*. Názov objektu, ktorý by bol uvedený pred dvojbodkou, je vynechaný, lebo nie je podstatný.

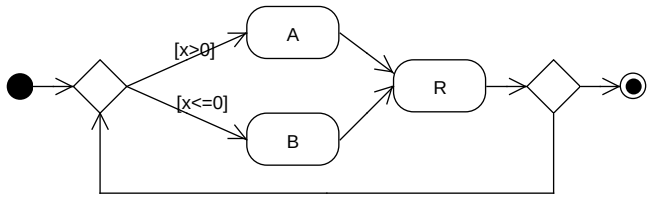
Diagram aktivít začína počiatočným uzlom aktivitou označovanou vyplneným krúžkom, a končí finálnym uzlom označovanou symbolom terča.

Niektoré akcie sú zložené z ďalších akcií, čo je indikované symbolom vidličky. Takéto

akcie sa označujú ako aktivity. Každý diagram aktivít vlastne predstavuje aktivitu, ktorú možno takto vyvolať v inom diagrame aktivít. V našom diagrame sú to aktivity *Vyhľadaj výrobok* a *Modifikuj plán doplnenia zásob*, ktoré vlastne zodpovedajú rovnomenným prípadom použitia.

Opakovanie vyhľadávania je modelované pomocou tzv. rozhodovacích uzlov (angl. decision nodes) a spájacích uzlov (merge nodes), ktoré aj jedny, aj druhé majú formu kosoštvorca. Rozhodovací uzol v skutočnosti o ničom nerozhoduje, t. j. je to iba spájací uzol. Rozhodnutia sú modelované pomocou strážcov (angl. guards), ktoré sa uvádzajú na hranách v hranatých zátvorkách. Ak je podmienka, ktorú strážca stanovuje splnená, značka hranou prejde. Inak ju strážca zastaví.

Použitie spájacích uzlov je nevyhnutné, aby nedošlo k zablokovaniu diagramu aktivít v čakaní, aby akcia dostala značku z každej hrany, ktorá do nej vstupuje, pričom je konfigurácia diagramu taká, že to nikdy nenastane. Napríklad, na obrázku 3.10 akcia R nikdy nedostane značky z obidvoch hrán, lebo na základe hodnoty x značka pôjde iba jednou vetvou. Tým nikdy nevyšle značku a diagram aktivít zablokuje v tejto akcii. Spájací uzol vysiela značku cez každú hranu, ktorá z neho vystupuje, hneď ako prijme čo len jednu značku.



Obr. 3.10: Diagram aktivít sa zablokuje v akcii R.

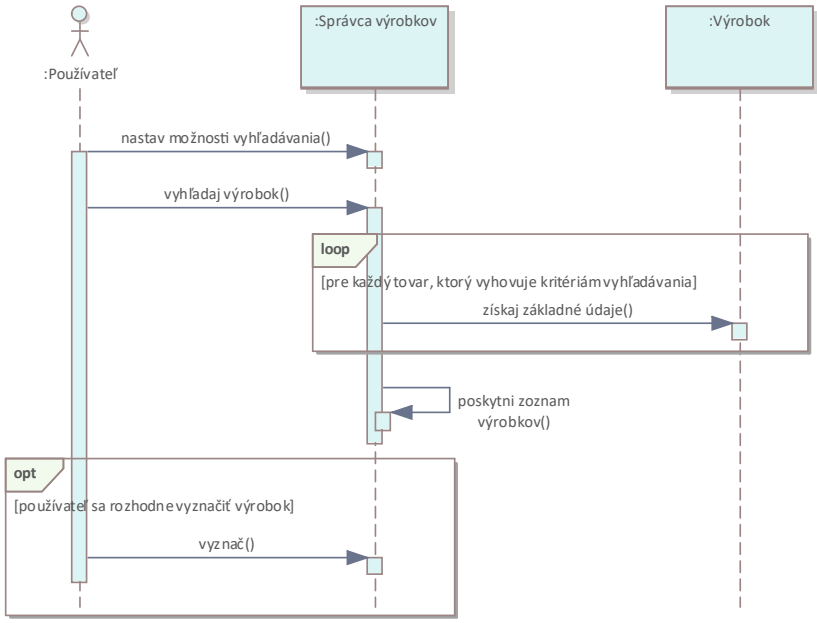
To, že kedykoľvek počas objednávania, zákazník môže vzdať tento proces je modelované regiónom prerušiteľnej aktivity (angl. region of an interruptible activity). Región je vyznačený oválom s prerušovanou čiarou. Samotná podmienka prerušenia je vyznačená prvkom so zárezom vo forme trojuholníka, čo symbolizuje miesto pre prijatie signálu. Z tohto prvku smeruje zalomená hrana k akcii, v ktorej proces pokračuje. V našom prípade je to finálna akcia, t. j. proces skončí.

Rozšírenie prípadu použitia sa modeluje rovnako, ako zahrnutie: vyvolaním príslušnej akcie. Diagramy aktivít nepodporujú aspektovo-orientovaný prístup.

Diagramy aktivít vytvorené podľa prípadov použitia priamočiaro vyjadrujú činnosti a ich postupnosti, ale potláčajú vyjadrenie štruktúry.

3.12 Vyjadrenie prípadu použitia diagramom sekvencií

Obrázok 3.11 ukazuje iný spôsob vyjadrenia prípadu použitia diagramom: diagramom sekvencií. Diagram sekvencií zobrazuje postupnosť posielania správ medzi inštanciami. Inštancie sú vyznačené vo vrchnej časti diagramu, pričom z každej vychádza prerušovaná čiara, ktorá sa označuje ako línia života (angl. lifeline). Čas plynie zhora nadol. Správy najčastejšie majú význam volania operácií objektov príslušných tried.



Obr. 3.11: Diagram sekvencií pre prípad použitia *Vyhľadaj výrobok*.

Inštancia po prijatí správy vykonáva príslušnú činnosť, čo je indikované obdĺžnikom na línii života. Ten sa volá špecifikácia vykonávania (angl. execution specification). Často sa vynecháva.

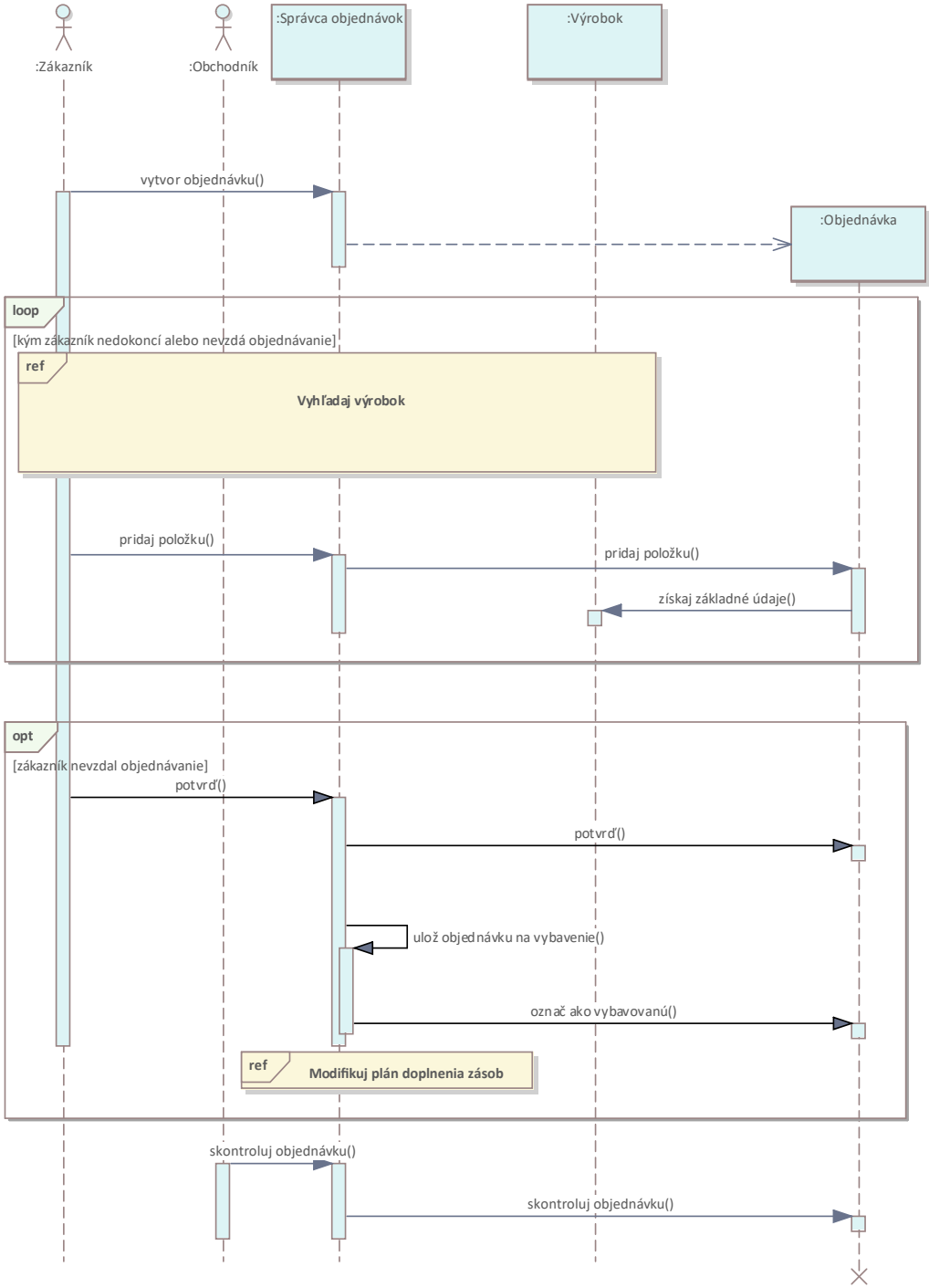
Pri vyjadrení diagramom sekvencií tendenciou je prípad použitia spresniť: analýzou správania objaviť jeho ďalšie aspekty a k tomu zodpovedajúcu štruktúru. V diagrame sekvencií pre prípad použitia *Vyhľadaj výrobok* tak vznikli triedy *Výrobok* a *Správca výrobkov*. Tieto triedy len indikujú štruktúru a v návrhu sa môžu transformovať do viacerých ďalších tried.

Účastník prípadu použitia je zobrazený tiež ako inštancia, aj keď nepredstavuje súčasť systému. Správy, ktoré posiela, súčastiam systému, sú volania ich operácií. V skutočnosti, účastník bude interagovať so systémom prostredníctvom používateľského

rozhrania. To by mohlo byť indikované ďalšími inštanciami, ktoré by predstavovali formuláre, čo sa niekedy aj robí. Avšak aj tak by sme sa nevyhli volaniu operácií, hoci v tomto prípade operácií formulárov.

Diagramy sekvencií sa pôvodne používali na vyjadrenie partikulárnych situácií interakcie inštancií. Až neskôr sa začali využívať na úplnú špecifikáciu operácií, ktorá je vysvetlená v kapitole 7. Za týmto účelom do nich boli pridané prvky, ktoré sa volajú kombinované fragmenty (angl. combined fragment), a ktoré umožňujú vyjadriť podmienené a opakované správanie. Diagram sekvencií na obrázku 3.11 zobrazuje slučku (loop) a voľbu (opt).

Diagram sekvencií pre prípad použitia *Zadaj objednávku* je na obrázku 3.12. Tento diagram sa odkazuje na diagram sekvencií pre prípad použitia *Vyhľadaj výrobok* pomocou na to určeného kombinovaného fragmentu (ref).

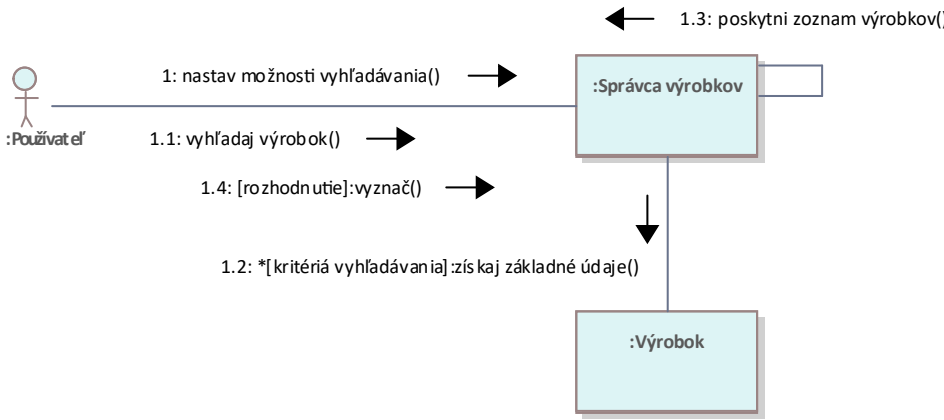


Obr. 3.12: Diagram sekvencií pre prípad použitia *Zadaj objednávku*.

Prípady použitia sa dajú pomerne priamočiaro vyjadriť prostredníctvom diagramov aktivít, ale až diagramy sekvencií pomáhajú odhaliť štruktúru systému, lebo vyjadrujú prípady použitia ako spoluprácu objektov.

3.13 Vyjadrenie prípadu použitia diagramom komunikácie

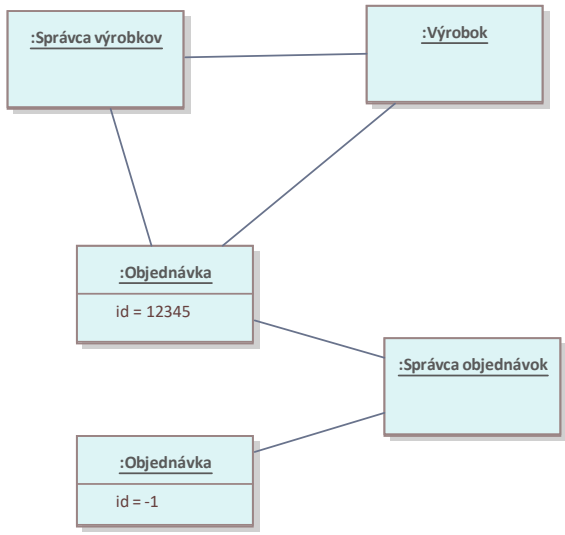
Obrázok 3.13 zobrazuje diagram komunikácie analogicky k diagramu sekvencií na obrázku 3.11. Diagram komunikácie lepšie ozrejmuje štruktúru, ale je v ňom náročnejšie sledovať postupnosť správ. Slučky (ako pri správe 1.2) a podmienené príkazy (ako pri správe 1.4) sú v ňom menej zreteľne vyjadrené (podmienky boli skrátené).



Obr. 3.13: Diagram komunikácie pre prípad použitia *Vyhľadaj výrobok*.

3.14 Diagram inštancií

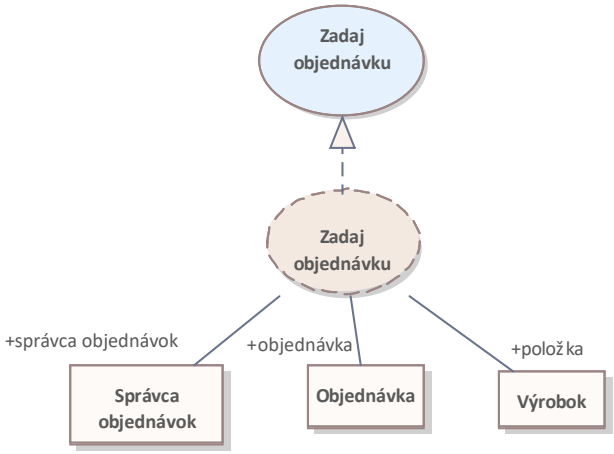
Niekedy vieme alebo chceme vyjadriť štruktúru (objekty), ktorá zabezpečuje realizáciu prípadu použitia, ale nechceme sa zaoberať detailmi ich interakcie, t. j. správami, ktoré sa medzi nimi posielajú. Diagram inštancií (predtým označovaného ako diagram objektov) v jednej možnej situácii v prípade použitia *Zadaj objednávku* je na obrázku 3.14.



Obr. 3.14: Diagram inštancií (objektov) v jednej možnej situácii v prípade použitia *Zadaj objednávku*.

3.15 Kolaborácie

Kolaborácia v UML predstavuje abstrakciu spolupráce objektov bez nevyhnutnosti identifikácie komunikačných väzieb. Kolaborácia predstavuje realizáciu prípadu použitia, čo býva aj vyjadrené príslušným vzťahom znázorneným prerušovanou hranou s uzavretou šípkou na konci. Obrázok 3.15 zobrazuje realizáciu prípadu použitia *Zadaj objednávku* príslušnou kolaboráciou.



Obr. 3.15: Realizácia prípadu použitia *Zadaj objednávku* príslušnou kolaboráciou.

S kolaboráciou sú vzťahmi asociácie (angl. association) spojené triedy, ktorých úloha v nej je vyznačená prostredníctvom asociačných rolí (angl. association roles).

3.16 Záverečné poznámky

Prípady použitia zaviedol Ivar Jacobson, ktorý pôvodnú notáciu modelovania prípadov použitia [Jac92] publikoval v roku 1992, ale idea vznikla už v roku 1967 [Jac04]. Už táto notácia umožňovala zahrnutie (aktiváciu) prípadu použitia v rámci iného prípadu použitia, ako aj rozšírenie (zásah do) prípadu použitia zo strany iného prípadu použitia.

Alistair Cockburn navrhol trochu iný štýl písania prípadov použitia, hlavne z hľadiska alternatívnych tokov [Coc00]. Čo je však dôležitejšie, Cockburn rozlíšil medzi konceptom prípadov použitia a ich notáciou, zdôrazňujúc, že najdôležitejšie prípady použitia sú tie, ktoré sa zameriavajú na ciele používateľov. Zároveň, ukázal, že tá istá notácia môže byť použitá na vyjadrenie prípadov použitia, ktoré zachytávajú sumárne ciele alebo pomocné funkcie (angl. subfunctions) [Coc00].

Kým Cockburn, na rozdiel od Jacobsona, uprednostňuje ešte koncíznejšiu formuláciu krokov, používanie alternatívnych tokov namiesto podmieneného správania (výrokov typu „if“), Overgaard and Palmkvist [ÖP04] zapisujú prípady použitia ako výrečný, na prózu sa podobajúci text a vôbec nepoužívajú číslovanie krokov. Coplien and Bjørnvig [CB10] navrhujú formát založený na dvojstĺpcovej tabuľke: jeden stĺpec predstavuje akcie používateľa, kým ten druhý akcie na strane systému. Jestvujú desiatky prístupov k tomu ako „správny“ prípad použitia má byť napísaný a navrhnutý, s viac alebo menej štruktúrovania a ohraničení v zmysle povolených kľúčových slov a –

dokonca – úplných fráz, pričom sú často protirečivé vo svojich odporúčaniach [TG15]. Vývojári softvéru zvyčajne vyberajú vlastnosti modelovania prípadov použitia, o ktorých sa dozvedeli z rôznych zdrojov, alebo aj zavádzajú vlastné. Niektoré vlastnosti z rôznych notácií možno kombinovať, kým sa iné vzájomne vylučujú [VuZ13].

Hoci prípady použitia vyzerajú ako algoritmy, dôležité je mať na pamäti, že ich budú čítať neprogramátori, a tak vnucovanie používania kľúčových slov a obmedzovanie ich sémantiky môže ľahko zabezpečiť, aby prípady použitia vyzerali ako kód, čo môže narúšať ich zrozumiteľnosť. Na druhej strane, bolo vyvinuté úsilie za účelom zachovania prípadov použitia v kóde [CB10, Cop09, JN04, BV17, BV16], aby taký kód mohol byť čítaný ako prípady použitia, ale nie za zámerom nahradiť iniciálne modelovanie prípadov použitia.

Prilákaní ich grafickou povahou, mnohí vývojári softvéru ľahko zamieňajú diagramy prípadov použitia za skutočnú formu prípadov použitia, ktorou je text. Diagramy prípadov použitia poskytujú len prehľad prípadov použitia a nemôžu vyjadriť ich toky. V dôsledku toho, najlepšie je kresliť diagramy prípadov použitia až potom, ako sú napísané. Inak ľahko vznikne veľký počet diagramov prípadov použitia – alebo jeden veľký – s bublinami a vzťahmi, ktoré nikto nerozumie. Ešte horšie, tí, ktorí budú písať text prípadov použitia, môžu pociťovať záväzok držať sa tohto nepodloženého obrazu a začať tvoriť opisy prípadov použitia, ktoré nemajú zmysel. Žiaľ, aj inak veľmi dobré knihy ohľadom modelovania prípadov použitia, akou je tá od Overgaarda a Palmkvist [ÖP04], zvádzajú vývojárov softvéru, aby modelovanie prípadov použitia začínali diagramami prípadov použitia.

Nie je zlé vedieť, že diagramy prípadov použitia vznikli dodatočne v rámci UML, čiže až v roku 1997, čo je päť rokov po publikovaní pôvodnej, textovej notácie, ktorá sa používala tridsať rokov predtým. Aj keď neskôr vznikli ďalšie podobné notácie [Coc00, DL06], ale aj veľmi odlišné [Buh98], v praxi, diagramy prípadov použitia zostávajú doménou UML.

4 Architektúra softvéru a prípady použitia

Aby sme mohli vytvoriť softvérový systém, musíme rozhodnúť o jeho fundamentálnych konceptoch alebo vlastnostiach zhmotnených v jeho prvkoch, vzťahoch a princípoch jeho vývoja a evolúcie. Toto je krátka ISO/IEC/IEEE definícia architektúry softvéru.[1] Nepochybne, dôležité rozhodnutia o štruktúre softvérového systému sú jej súčasťou. Časť štruktúry možno vyťažiť z prípadov použitia, ale významná časť vyplýva z poznania domény, a niektoré rozhodnutia o štrukturálnom usporiadaní sú dokonca generické. Na zachytenie štruktúry sa primárne používa diagram tried (angl. class diagram), ktorý je súčasťou UML.

Časť 4.1 vysvetľuje, ako zachytiť štruktúru z prípadov použitia základným diagrmaom tried. Časť 4.2 ukazuje, ako podstatná časť štruktúry vyplýva z domény a ako ju vyjadriť diagramom tried s použitím pokročilejších možností. Časť 4.3 vysvetľuje klasifikáciu tried podľa ich povahy. Časť 4.4 ukazuje prechod k návrhovému diagramu tried. Časť 4.5 ukazuje, ako zachytiť sledovateľnosť.

4.1 Štruktúra z prípadov použitia

Časť štruktúry softvérového systému možno odvodiť z prípadov použitia. Táto štruktúra odráža mentálny model používateľa (jeho predstavy o fungovaní systému). Niečo z nej vyplýva priamo z opisu prípadov použitia, ale viac sa dá získať ich rozborom prostredníctvom diagramov sekvencií, ako sme videli v časti 3.12. Stačí vymenovať typy definované líniami života. Túto štruktúru môžeme vyjadriť diagramom tried (angl. class diagram) v UML. Iniciálne, to môžu byť iba prvky bez vzťahov, ako sú uvedené v diagrame tried na obrázku 4.2.

Trieda je znázornená ako obdĺžnik. Môže obsahovať oddiely, ktoré zobrazujú jej atribúty a operácie. Viditeľnosť operácií pre ďalšie triedy sa vyjadruje podobne ako v objektovo-orientovaných programovacích jazykoch od úplného skrytia, označovaného

[1] `http://www.iso-architecture.org/ieee-1471/defining-architecture.html`

Obr. 4.1: Prvky štruktúry.

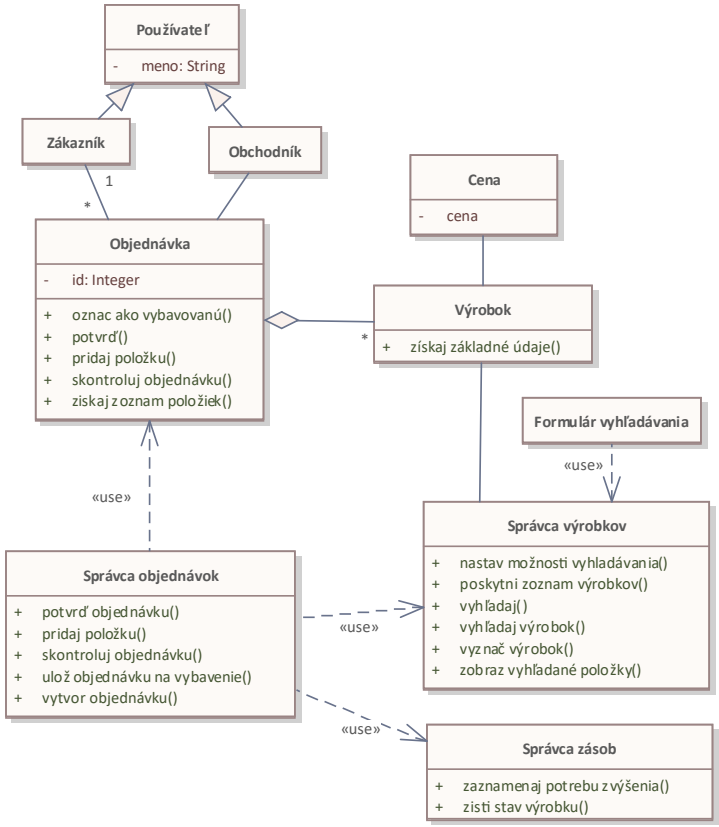
ako private a symbolom -, až po úplnú viditeľnosť public, označovanú ako public a symbolom +. Medziúrovne sú ešte protected (#) a package (~).

Pri rozbore štruktúry môže byť nadnesené hneď stanovovať triedy so všetkými ich prvkami. Môžeme začať hypotetickými interakciami ich inštancií – t. j. objektov – ktoré môžu vzniknúť v systéme počas jeho fungovania, ale bez uvedenia samotných interakcií. Príklad takéhoto diagramu inštancií sme mohli vidieť v časti 3.14 na obrázku 3.14. Tento prístup môžeme použiť aj ak nemáme diagramy sekvencií alebo ich chceme zjednodušiť. Výhodou je, že týmto vyjadrujeme – hoci nie úplne presne – vzťahy medzi štrukturálnymi prvkami.

Obrázok 4.2 zobrazuje diagram tried e-obchodu, v ktorom sú aj vzťahy a niektoré ďalšie prvky. Vzťahy môžu byť len na úrovni naznačenia súvisu, na čo sa používa asociácia, ako medzi triedami *Obchodník* a *Objednávka*. Niekedy poznáme počty výskytov jedného prvku voči druhému, čo sa v UML vyjadruje násobnosťou (ang. multiplicity), ako medzi triedami *Zákazník* a *Objednávka*: zákazník môže mať veľa objednávok, kým objednávka sa vzťahuje presne na jedného zákazníka.

Niekedy vieme, že jedna trieda používa druhú. Napríklad, trieda *Správca objednávok* používa triedu *Objednávka*. Základný vzťah zobrazený prerušovanou šípkou je tu závislosť (angl. dependency). Ten hovorí, že jeden prvok závisí od druhého. Ak chceme byť presnejší, modifikujeme tento význam tzv. stereotypom. V tomto prípade je to «use». Toto je jeden z predefinovaných stereotypov v UML, ale možné je pridávať ďalšie.

Objednávka zahŕňa výrobky. Toto je vzťah agregácie, ktorý môžeme vidieť medzi príslušnými triedami. Symbol kosoštvorca sa uvádza pri hierarchicky vyššom prvku, ktorým je agregujúci prvok. Možné je indikovať aj násobnosť. Často sa to robí iba na



Obr. 4.2: Prvky štruktúry a vzťahy medzi nimi.

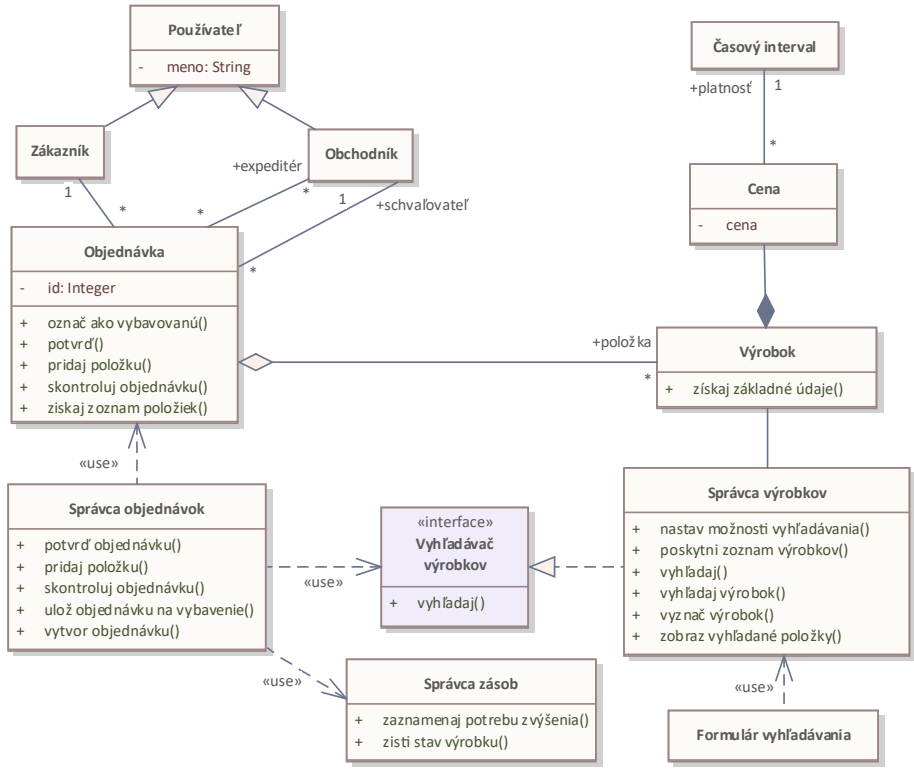
strane agregovaného prvku, ako je to aj v našom príklade, ktorý hovorí, že objednávka môže obsahovať viac výrobkov.

Vzťah dedenia alebo, v UML terminológii, generalizácie/špecializácie sme už mali príležitosť vidieť v časti 3.6 medzi prípadmi použitia a v časti 3.8 medzi zákazníkom a používateľom, ako aj medzi obchodníkom a používateľom, ale ako externými entitami, t. j. entitami, ktoré nie sú súčasťou vyvíjaného systému. Ak uvažujeme o ich evidencii, musíme zákazníka, obchodníka a používateľa uviesť ako triedy. Dedením sa do odvodených tried prenášajú prvky základnej triedy, ktorou je v tomto prípade *Používateľ*. Navyše, kdekoľvek sa očakáva inštancia základnej triedy, možno zaradiť inštanciu odvodenej triedy, čo je v objektovo-orientovanom programovaní známe ako polymorfizmus. Všimnime si, že sa symbol trojuholníka, ktorým sa indikuje dedenie, uvádza pri hierarchicky vyššom prvku – rovnako ako pri agregácii kosoštvorec.

4.2 Štruktúra z domény

Aj keď štruktúru odvádzame z prípadov použitia alebo diagramov sekvencií, ktoré im zodpovedajú, máme tendenciu pridávať aj triedy, ktoré sa v nich nevyskytujú. V našom príklade je to trieda *Cena*, ktorá súvisí s triedou *Výrobok*. Začínajú sa teda objavovať aj ďalšie vzťahy. Štruktúra sa spresňuje. Postupné spresňovanie je príznačné nie len pre modelovanie, ale aj pre celý proces vývoja softvéru. Stabilnejšia časť štruktúry softvérového systému vlastne vyplýva z domény (aplikácie). Ak poznáme doménu, zásadné prvky štruktúry nebudeme odvádzať z prípadov použitia.

Obrázok 4.3 zobrazuje ďalšiu verziu diagramu tried e-obchodu, ktorá obsahuje aj zohľadnenie vývoja ceny výrobku v čase. Toto je jedným z analytických vzorov (angl. analysis patterns), presnejšie povedané je súčasťou analytického vzoru Scenario [Fow96, s. 188]. Dokumentované sú aj mnohé ďalšie analytické vzory. Ich použiteľnosť často presahuje hranice jednej domény.



Obr. 4.3: Rozšírenie štruktúry o doménové znalosti.

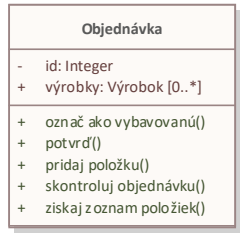
Na ujasnenie rolí, ktoré prvky štruktúry zohrávajú jeden voči druhému používame asociačné roly, s ktorými sme sa už stretli pri kolaboráciách (pozri časť 3.15). Naprí-

klad, výrobok hrá rolu položky v objednávke. Obchodník môže byť expeditér alebo schvaľovateľ. Časový interval predstavuje platnosť ceny.

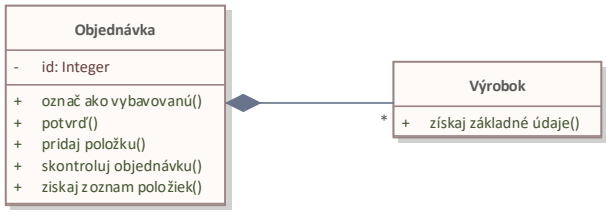
Iný spôsob vyjadrenia roly je prostredníctvom rozhrania (angl. interface). *Správca objednávok* na vyhľadanie výrobkov používa triedu *Správca výrobkov*. Tento vzťah bol vyjadrený priamo v predchádzajúcej verzii diagramu tried znázornenej na obrázku 4.2. *Správca výrobkov* len hrá rolu vyhľadávača výrobkov, čo je indikované vzťahom realizácie: *Správca výrobkov* realizuje rozhranie *Vyhľadávač výrobkov*. V diagrame tried sa rozhranie označuje streotypom «interface». Túto rolu môže zohrať aj iná trieda, t. j. jej inštancie. Rozhranie predpisuje *Vyhľadávač výrobkov* príslušnú metódu, ktorú *Správca výrobkov* implementuje.

Správca výrobkov sa stáva podtypom typu *Vyhľadávač výrobkov*: jeho inštancie môžu vystupovať kdekoľvek je očakávaná inštancia typu *Vyhľadávač výrobkov*. Pritom rozhranie ako také inštancie nemôže mať. Realizácia rozhrania zodpovedá vzťahu **implements** medzi triedou a rozhraním v Jave. Rozhrania podporuje aj C#. C++ nepodporuje rozhrania, ale možno ich emulovať pomocou tried, ktorých všetky metódy (členské funkcie) sú plne virtuálne.

Agregáciu možno vyjadriť aj vo forme atribútu a to vrátane násobnosti, ako ukazuje obrázok 4.4. Podľa špecifikácie UML, to však bude tzv. silná agregácia v UML označovaná aj ako kompozitná agregácia alebo kompozícia. V kompozitnej agregácii agregovaná inštancia môže byť súčasťou iba jednej agregujúcej inštancie. V našom prípade by sme tým stanovili, že jeden výrobok môže byť súčasťou iba jednej objednávky. Ak každá inštancia triedy *Výrobok* predstavuje jeden výrobok, toto je v poriadku. Ak však inštancia triedy *Výrobok* predstavuje typ výrobku, čo je pravdepodobne to, čo sme chceli, správnou voľbou je agregácia ako na obrázku 4.2, ktorá sa označuje ako zdieľaná agregácia (angl. shared aggregation). Pri triede *Cena* použitie kompozitnej agregácie korektne vyjadruje zámer: cena bude jedinečná pre každý výrobok.



Obr. 4.4: Agregácia vyjadrená atribútom.



Obr. 4.5: Kompozitná agregácia.

4.3 Klasifikácia tried

Unified Process, jeden z najznámejších procesných rámcov vývoja softvéru, zahŕňa výstižný spôsob klasifikácie tried na základe ich povahy:[2]

- entity – triedy prevažne údajového charakteru, v ktorých sú kľúčové atribúty
- control – triedy prevažne procesného charakteru, v ktorých sú kľúčové operácie
- boundary – triedy rozhrania voči používateľovi alebo iným systémom

Obrázok 4.6 zobrazuje diagram tried nášho e-obchodu s triedami klasifikovanými podľa prístupu Unified Process, čo je vyznačené rovnomennými stereotypmi.

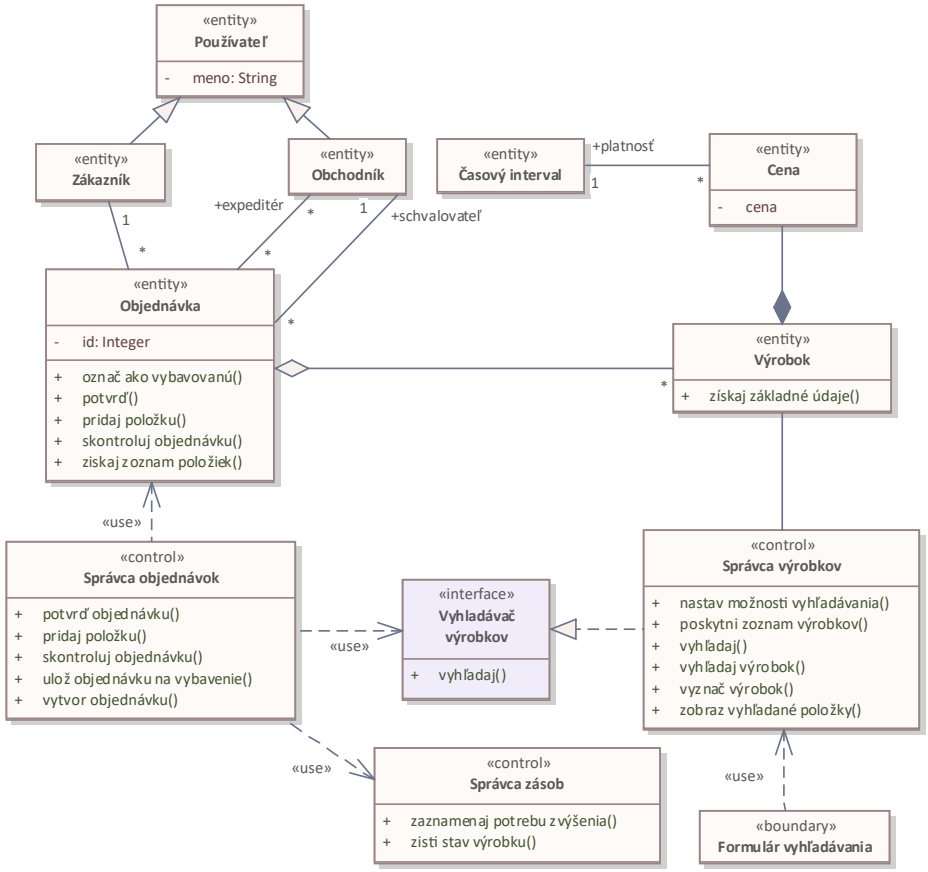
Táto klasifikácia usmerňuje k oddeleniu používateľského rozhrania od aplikačnej logiky.

4.4 Návrh

Obrázok 4.7 zobrazuje návrhový diagram tried. Pribudli ďalšie detaily. Atribúty a operácie sú spresnené smerom k implementácii. Identifikátory sú v angličtine, čo nie je podmienkou, ale často sa na úrovni návrhu robí. Zvlášť to má význam, ak je vývojový tím medzinárodný. V analýze je dobré držať sa jazyka, ktorým hovorí klientska strana, aby modelové artefakty boli zrozumiteľné aj pre ňu.

Architektonické usporiadanie známe ako Model–View–Controller (MVC), ktoré je niekedy označované ako architektonický vzor (angl. architectural pattern), a inokedy pokladané za súčasť jazyka návrhových vzorov (angl. design patterns) [BHS07], predstavuje dômyselný spôsob, ktorým sa dá dosiahnuť oddelenie používateľského rozhrania od aplikačnej logiky. MVC vlastne rieši komplexnejší problém: systém musí zodpovedať mentálnemu modelu používateľa, ale toto vyžaduje koordináciu viacerých

[2] Túto klasifikáciu navrhol Jacobson [Jac92, JGJ97].

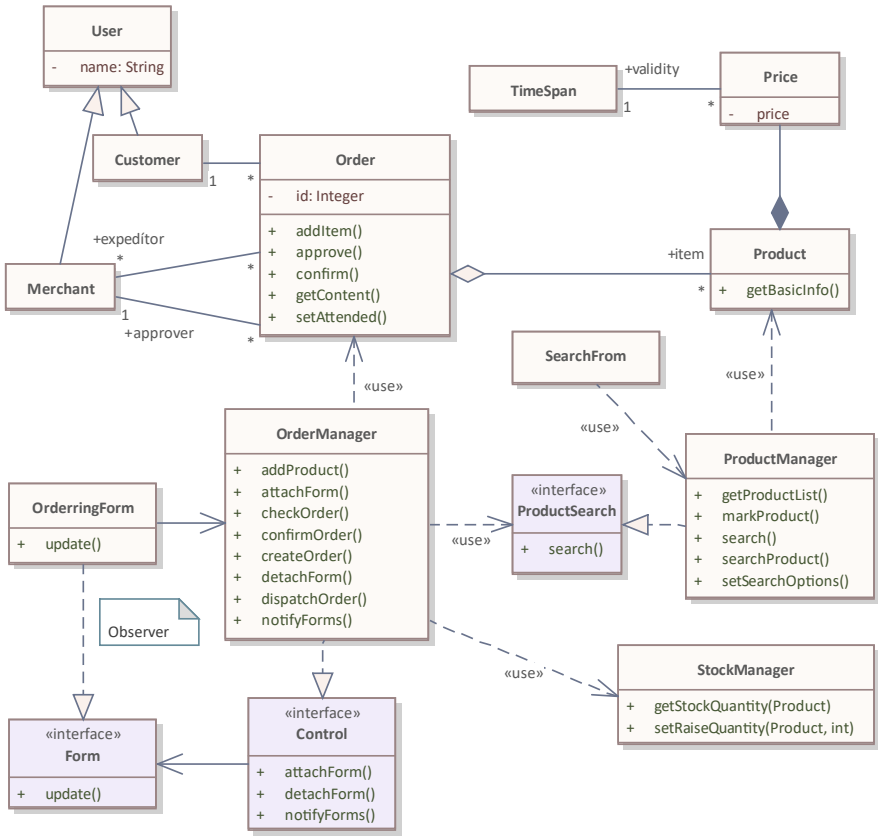


Obr. 4.6: Klasifikácia tried na základe ich povahy podľa prístupu Unified Process.

pohľadov nad zložitým modelom. Tento konflikt protichodných síl navodí dojem, že niet východiska. Toto je charakteristické pre všetky vzory. Vzor potom prináša dôvtipné riešenie, ktoré protichodné sily uvádza do rovnováhy. Riešenie, ktoré prináša MVC, spočíva v už spomenutom oddelení používateľského rozhrania, t. j. pohľadov, ktoré zodpovedajú mentálnemu modelu používateľa, od aplikačnej logiky, ktorá predstavuje (vnútorný) model danej domény, pričom koordináciu pohľadov na základe udalosti, ktoré vstupujú do systému, zabezpečuje kontroler.

Kde by sme mali hľadať MVC v diagrame tried? MVC je rozpracovaný pomocou ďalších návrhových vzorov. Aby sme ho identifikovali, museli by sme odsledovať návrhové vzory, z ktorých pozostáva. Základ MVC tvoria tri návrhové vzory [GHJV95]:

- Observer rieši vzťah Model–View, kde Model je Subject, a View je Observer
- Strategy rieši vzťah View–Controller, kde View je Context, a Controller predstavuje Concrete Strategy



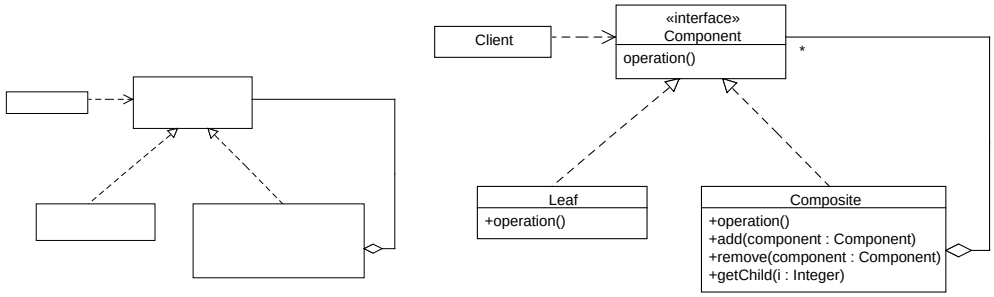
Obr. 4.7: Návrhový diagram tried e-obchodu.

- Composite rieši vhniezdené pohľady, kde View je Composite alebo Leaf

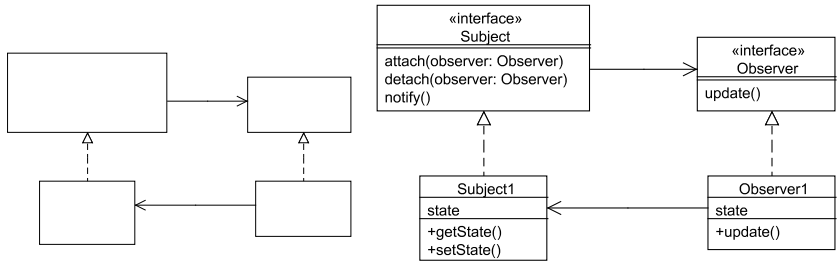
Návrhové vzory je niekedy možné rozpoznať už na základe samotného štrukturálneho usporiadania tried a rozhraní, ktoré v nich vystupujú. Obrázky 4.8–4.10 zobrazujú príklady návrhových vzorov vo forme zaslepenej štruktúry a štruktúry s obsahom. Všimnime si, napríklad, veľmi charakteristickú kombináciu agregácie a dedenia v opačnom smere vo vzore Composite.

V našom príklade bol uplatnený vzor Observer, ktorého štruktúra nie je až taká charakteristická. Na obrázku 4.2 je vyznačený poznámkou umiestnenou medzi triedami, ktoré tento vzor realizujú.

Ak MVC prirovnáme k stereotypom Unified Process, triedy označené ako entity a control spolu zodpovedajú role Model, pričom boundary zodpovedá role View. Rola Controller v tejto analytickej reprezentácii štruktúry ešte nie je vyčlenená. Dôležité je poznamenať, že napriek podobnosti v pomenovaní triedy označené ako control nezodpovedajú role Controller.

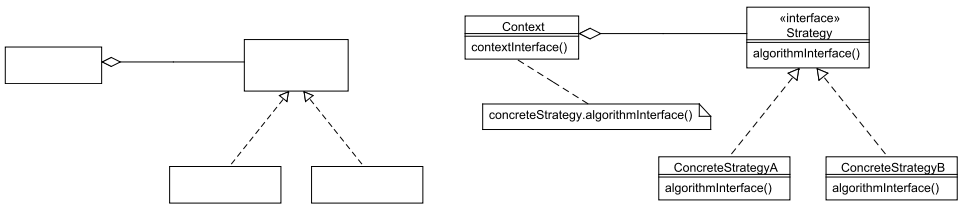


Obr. 4.8: Návrhový vzor Composite.



Obr. 4.9: Návrhový vzor Observer.

V diagrame na obrázku 4.2 je použitá orientovaná asociácia, ktorá má formu hrany so šípkou. Intuitivný, prirodzený význam je, že prvok, z ktorého hrana smeruje, akosi pozná prvok, do ktorého hrana smeruje, a takto asociácia v danom diagrame aj bola myslená. Od modelovacieho jazyka očakávame istú úroveň abstrakcie, a preto ani neočakávame striktné vymedzenie významu. Avšak, podľa platnej špecifikácie UML, šípka označuje navigovateľnosť, a to presne v tom zmysle, že sa počas vykonávania dá účinne dostať z inštancie prvku, z ktorého asociácia smeruje, k inštancii prvku, do ktorej asociácia smeruje [OMG17]. Ak potrebujeme vyjadriť, že jeden prvok obsahuje alebo, presnejšie, vlastní odkaz na iný prvok (koniec asociácie naň), podľa platnej špecifikácie UML treba použiť bodku. Od vzniku UML v roku 1997 do vzniku verzie UML 2 v roku 2005, orientovaná asociácia mala voľnú interpretáciu, a ukonče-

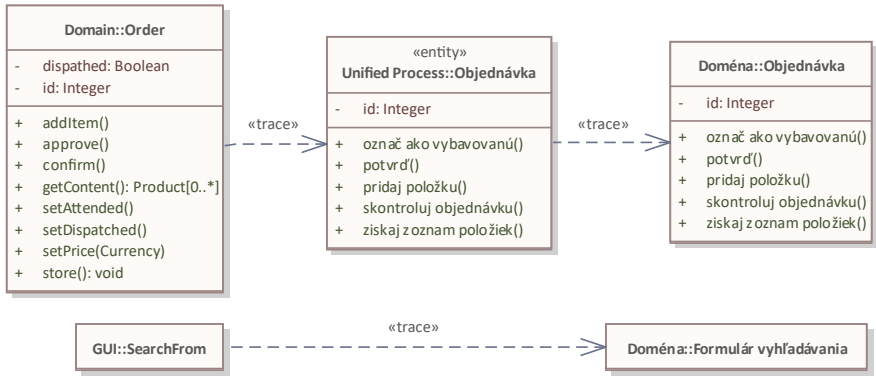


Obr. 4.10: Návrhový vzor Strategy.

nie asociácie bodkou vôbec nebolo súčasťou notácie [Sel13]. V praxi je bodka slabo používaná.

4.5 Sledovateľnosť

Ako sme už zistili v časti 4.1, modelovanie predpokladá postupné spresňovanie: stopu k pôvodným prvkom je niekedy vhodné uchovať. Príklad znázorňuje obrázok 4.11.



Obr. 4.11: Sled spresňovania niektorých tried e-obchodu vyjadrený vzťahom «trace».

5 Modularizácia a konceptualizácia štruktúry

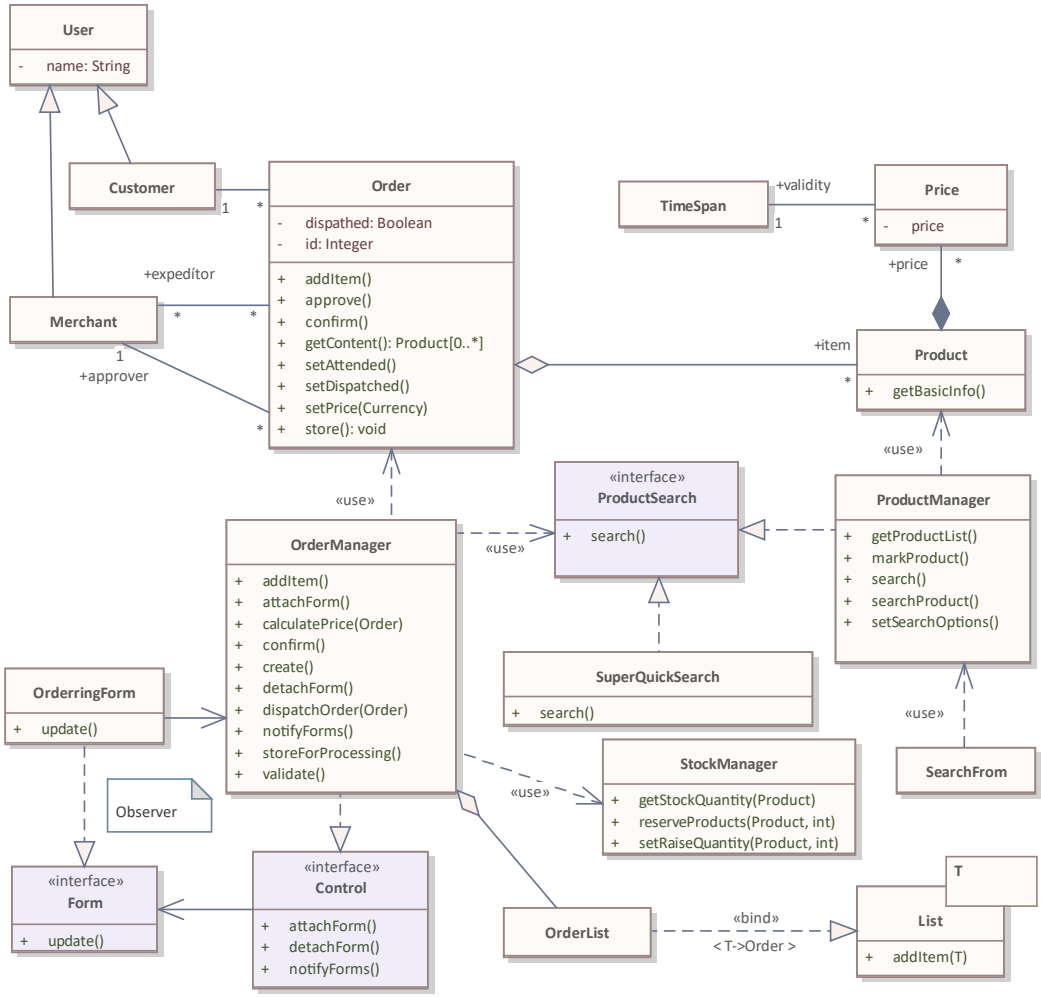
Rozsiahla štruktúra vyžaduje ďalšiu modularizáciu. UML za týmto účelom ponúka balíky, ale aj samotné diagramy ako také sú tiež prostriedkom modularizácie štruktúry. Ak je pri konceptualizácii systému potrebné vyhnúť sa viazaniu na triedy ako prvkom príliš blízkym k realizácii, možno použiť diagramy komponentov a kompozitnej štruktúry.

Časť 5.1 ozrejmuje pojem balíka a diagramu balíkov. Časť 5.2 poukazuje na význam prierezových diagramov tried Časť 5.3 poukazuje na to, že rozhranie je často umiestnené v inom balíku než trieda, ktorá ho realizuje. Časť 5.4 vysvetľuje diagramy komponentov a kompozitnej štruktúry.

5.1 Balíky

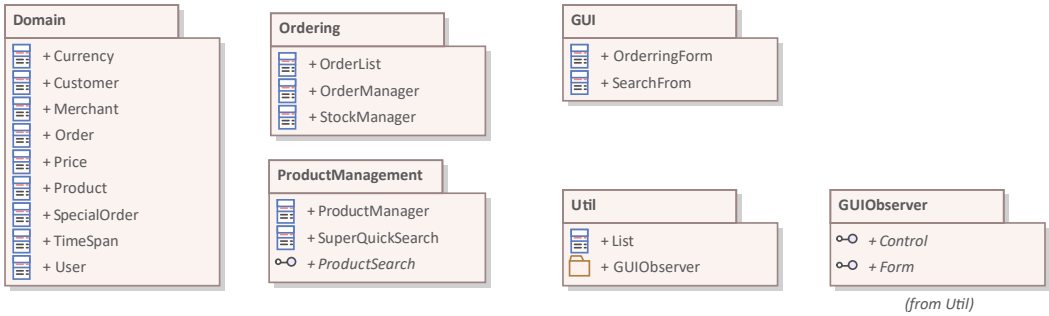
Na obrázku 5.1 sa štruktúra e-obchodu ešte o niečo rozrástla a spresnila oproti poslednej verzii z predchádzajúcej kapitoly.[1] Už začína byť náročné vynájsť sa v nej. Ako ju usporiadať, a pritom zachovať prehľad?

[1] Význam triedy *List* a vzťahu so stereotypom «bind» je vysvetlený v kapitole 9.



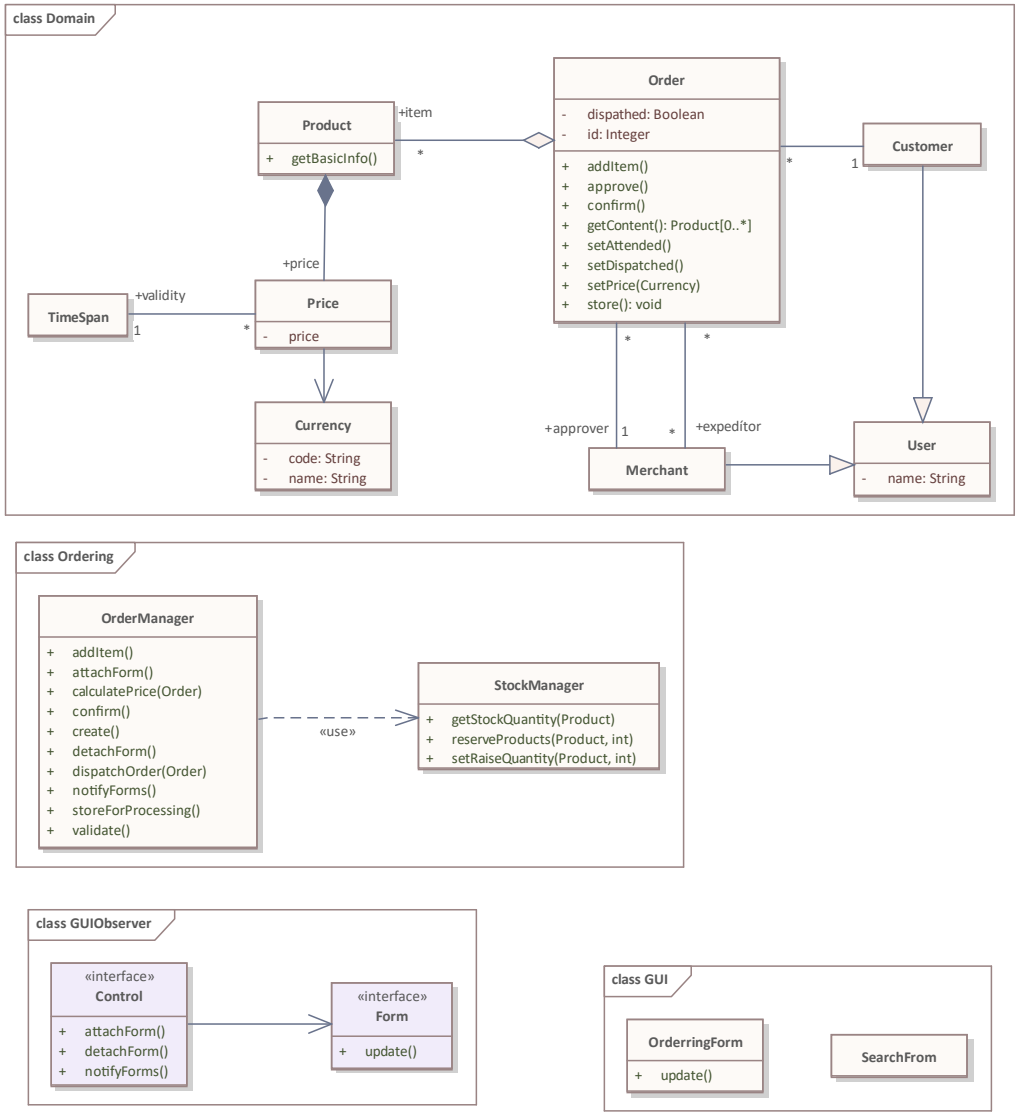
Obr. 5.1: Celkový diagram tried e-obchodu.

Aj UML podporuje koncept balíka (angl. package), ktorý je používaný v mnohých programovacích jazykoch. Obrázok 5.2 zobrazuje jednu možnú organizáciu doteraz identifikovanej štruktúry do balíkov. Prvok môže patriť len do jedného balíka.



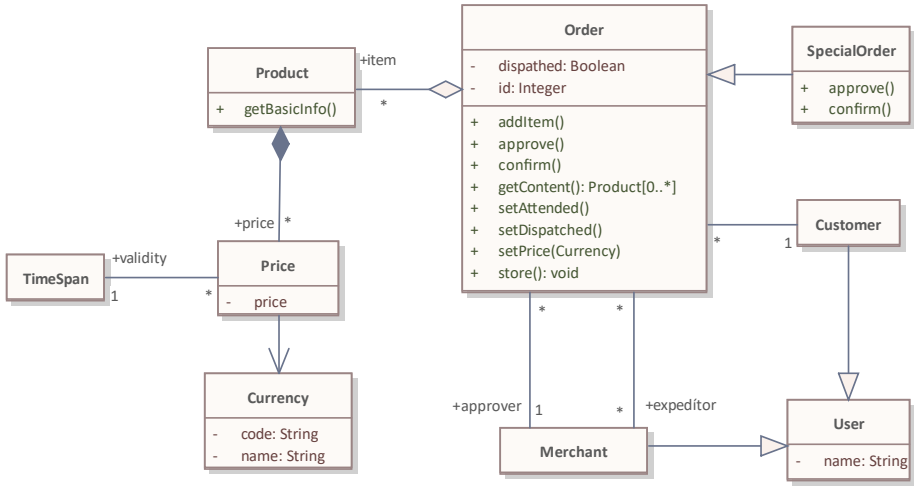
Obr. 5.2: Diagram balíkov v e-obchode.

Balíky je možné zobraziť aj s diagramami tried, ktoré predstavujú ich obsah, ako to ukazuje obrázok 5.3.

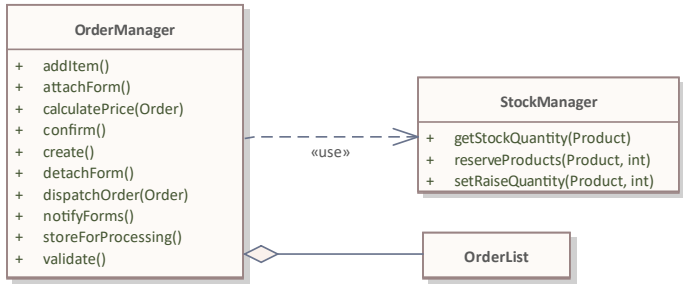


Obr. 5.3: Balíky v e-obchode s diagramami tried.

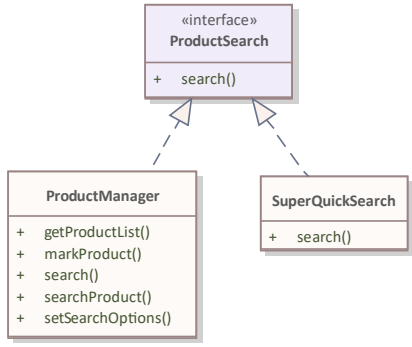
Obsah balíkov možno zobraziť v samostatných diagramoch, aby sme mohli s ním jednoduchšie pracovať. Tieto diagramy sú uvedené na obrázkoch 5.4– 5.9 Strácame však prehľad, lebo nevidíme ako balíky súvisia.



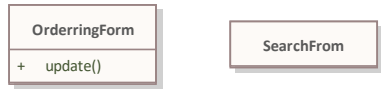
Obr. 5.4: Balík *Domain*.



Obr. 5.5: Balík *Ordering*.

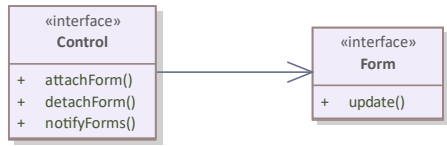


Obr. 5.6: Balík *ProductManagement*.



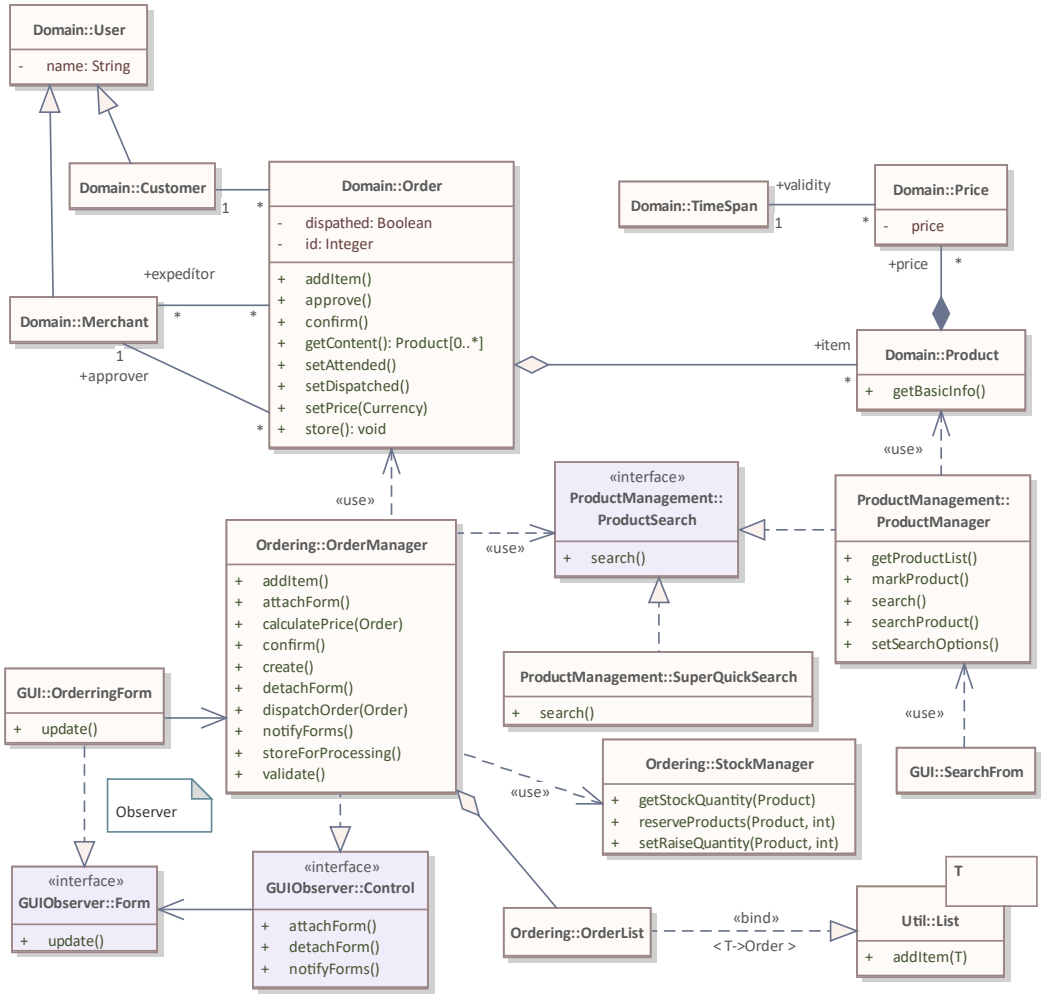
Obr. 5.7: Balík *GUI*.

Obr. 5.8: Balík *Util* (vzťah «bind» je vysvetlený v časti 5.3).

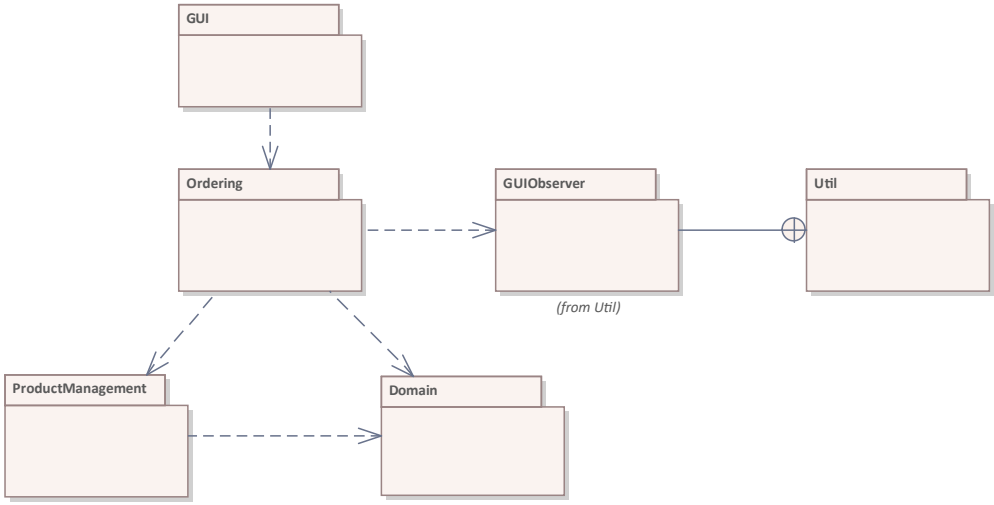


Obr. 5.9: Balík *GUIObserver*.

Na obrázku 5.10 je zobrazený diagram tried e-obchodu s plne kvalifikovanými názvami prvkov. Na základe kvalifikácie vidíme, do ktorého balíka prvok patrí. Z neho môžeme odvodiť závislosti medzi balíkmi, ktoré sú zobrazené na obrázku 5.11. Ak prvok jedného balíka závisí od prvku iného balíka alebo ho inak „pozná“, t. j. je do neho agregovaný, je od neho odvodený, vedie k nemu orientovaná asociácia a pod., medzi príslušnými balíkmi je závislosť v rovnakom smere. Typ závislosti sa dá spresniť stereotýpom: napríklad «access», ak prvky jedného balíka pristupujú k prvkom iného balíka, alebo «import», ak jeden balík importuje prvky iného balíka.

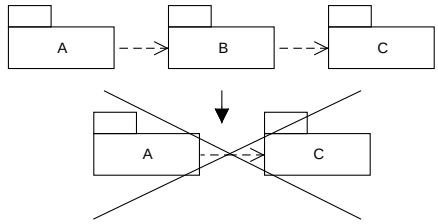


Obr. 5.10: Celkový diagram tried e-obchodu s plne kvalifikovanými názvami tried.



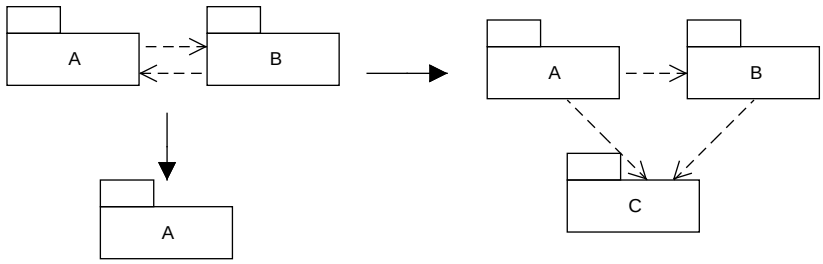
Obr. 5.11: Vzťahy medzi balíkmi v e-obchode.

Ako ukazuje obrázok 5.12, vzťah závislosti medzi balíkmi nie je tranzitívny.



Obr. 5.12: Vzťah závislosti medzi balíkmi nie je tranzitívny.

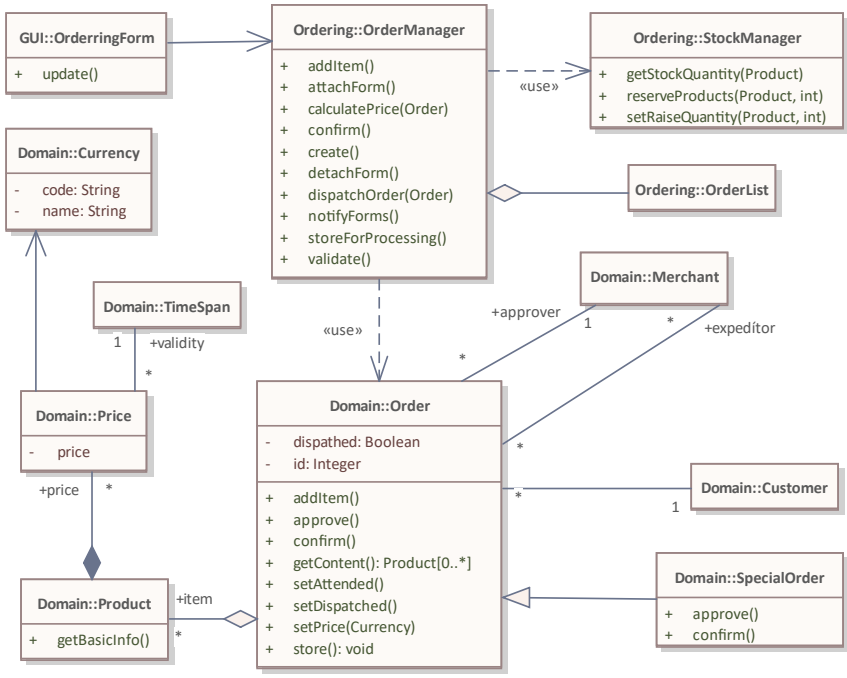
V prípade cirkulárnej závislosti medzi balíkmi ako na obrázku 5.13, balíky možno zlúčiť do jedného alebo vyčleniť prvky, od ktorých sú obidva závislé, do samostatného balíka.



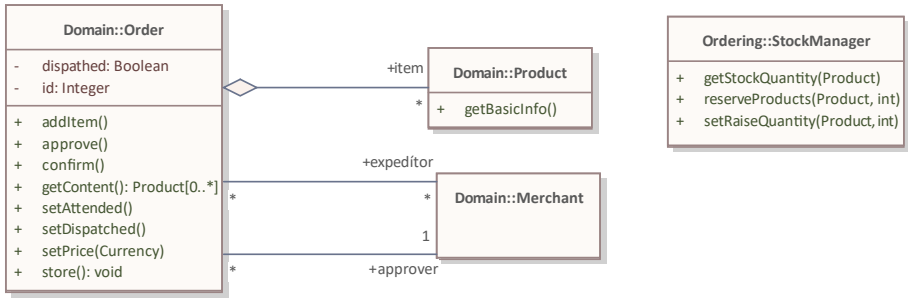
Obr. 5.13: Riešenie cirkulárnej závislosti medzi balíkmi.

5.2 Prierezové diagramy tried

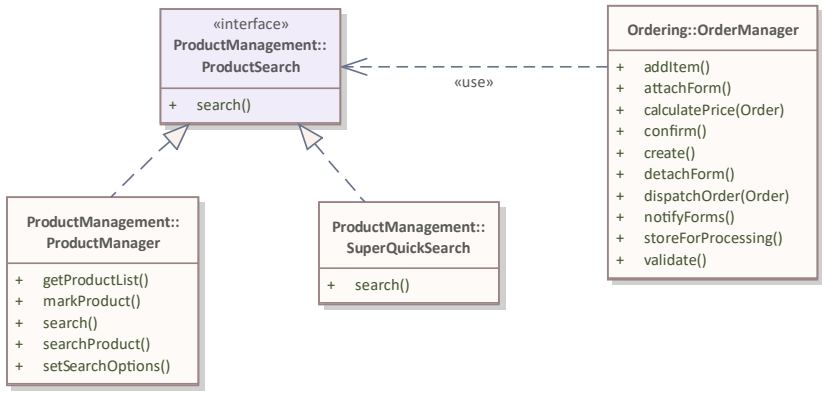
Celkový diagram tried býva príliš rozsiahly, kým diagramy tried balíkov zobrazujú iba ich vlastné prvky. Na pochopenie súvislosti medzi prvkami štruktúry potrebujeme prierezové diagramy tried: také, ktoré zobrazia charakteristické prepojenia prvkov štruktúry s určitým zámerom. V e-obchode možno identifikovať viac takýchto charakteristických prepojení prvkov štruktúry: obrázok 5.14 zobrazuje objednávanie, obrázok 5.15 zobrazuje objednávanie, obrázok 5.16 zobrazuje vyhľadávanie, a obrázok 5.17 zobrazuje aplikáciu vzoru Observer v používateľskom rozhraní.



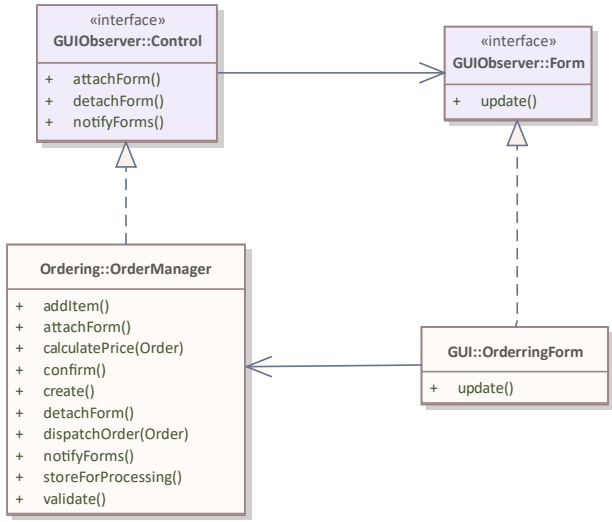
Obr. 5.14: Objednávanie.



Obr. 5.15: Zásobovanie.

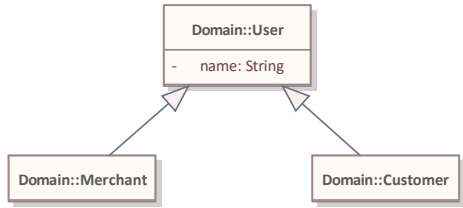


Obr. 5.16: Vyhľadávanie.



Obr. 5.17: Aplikácia vzoru Observer v používateľskom rozhraní.

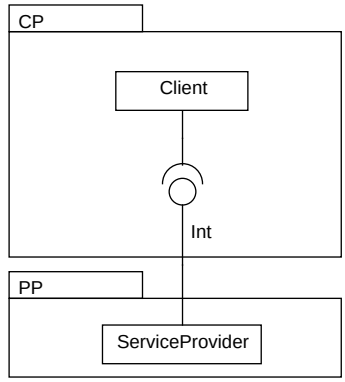
Niekedy je vhodné zobraziť iba časť jedného balíka. Obrázok 5.18 zobrazuje hierarchiu typov používateľov evidovaných v e-obchode.



Obr. 5.18: Hierarchia typov používateľov evidovaných v e-obchode.

5.3 Balíky a rozhrania

Rozhranie je často umiestnené v inom balíku než trieda, ktorá ho realizuje. Túto situáciu približuje obrázok 5.19. Rozhranie *Int* je znázornené v tzv. lízatkovej (angl. lollipop) notácii. Kruh predstavuje rozhranie. Rozhranie je spojené hranou s triedou, ktorá ho realizuje, t. j. ponúka. Konektor okolo kruhu predstavuje závislosť a je spojený s triedou, ktorá rozhranie používa.



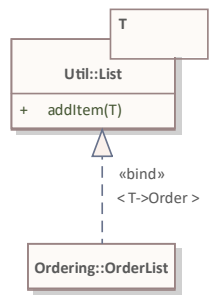
Obr. 5.19: Rozhranie je často umiestnené v inom balíku než trieda, ktorá ho realizuje.

Toto je príznačné pre rámce (angl. framework), v ktorých rozhrania zastupujú chýbajúce prvky, t. j. prvky, ktoré má doplniť klient, aby rámec pomocou nich zabezpečil potrebné činnosti. Situáciu ilustruje nasledujúci kód v Jave:

```
package CP;

\\ klient predpisuje rozhranie
interface Int {
    ...
```

```
}

class Client {
   void m(Int o) {...}
    ...
}

_____________________________


package PP;
import CP;

\\ rozhranie implementuje poskytovatel sluzby z ineho balika
class ServiceProvider implements Int {
    ...
   void op() {
      new Client().m(this);
   }
}
```

Analogickú situáciu si môžeme všimnúť pri špecializácii šablón tried (angl. template class). V e-obchode máme triedy *OrderList*, ktorá je viazanou triedou (angl. bound class) odvodenou od šablóny triedy (tiež označovanej ako parametrizovaná trieda (ang. parameterized class)) *List* prostredníctvom viazania (angl. binding). Trieda *OrderList* je umiestnená v klientskom balíku Ordering, pričom balík *Util*, v ktorom sa nachádza trieda *List*, je akýsi miniatúrny rámec. Obrázok 5.20 zobrazuje príslušný prierezový diagram tried.

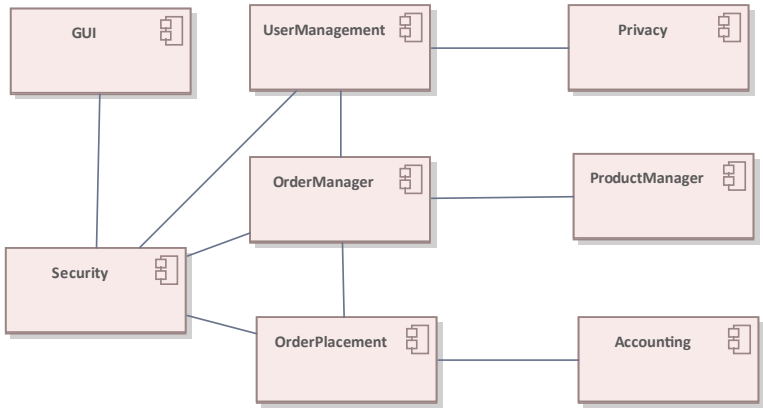


Obr. 5.20: Viazanie parametrov šablóny triedy prostredníctvom vzťahu «bind».

5.4 Komponenty a kompozitná štruktúra

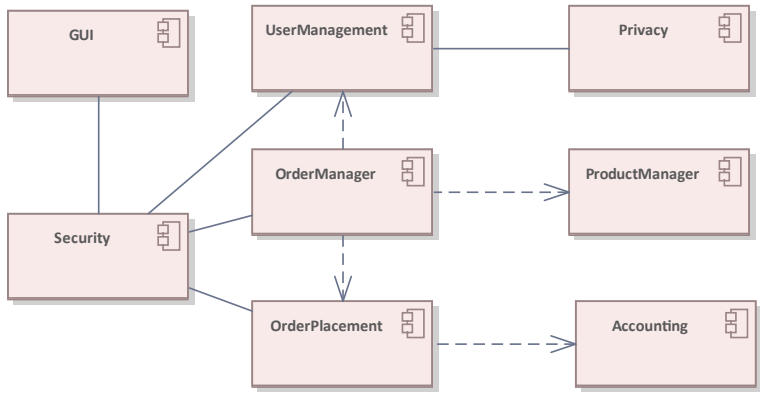
Balíky by mohli poslúžiť na vyjadrenie vysokoúrovňovej štruktúry systému bez toho, aby sa muselo ísť až na úroveň tried, a niekedy sa takto aj používajú. Problémom je, že balíky v UML predstavujú iba zoskupenia prvkov, a nie celky, ktoré možno pokladať za aktívne a vyjadrovať aj interakciu ich inštancií. Taktiež, vnútro balíkov sú vždy triedy a rozhrania, ktoré zahŕňa. Nemôžeme ho skúmať na vyššej úrovni abstrakcie. UML za týmto účelom ponúka komponenty a kompozitnú štruktúru, ktoré umožňujú konceptuálne uvažovať nad štruktúrou bez viazania sa na presný spôsob implementácie.

V prvom priblížení je možné proste vymenovať komponenty a naznačiť ich prepojenia vo forme asociácií, ako ukazuje obrázok 5.21.



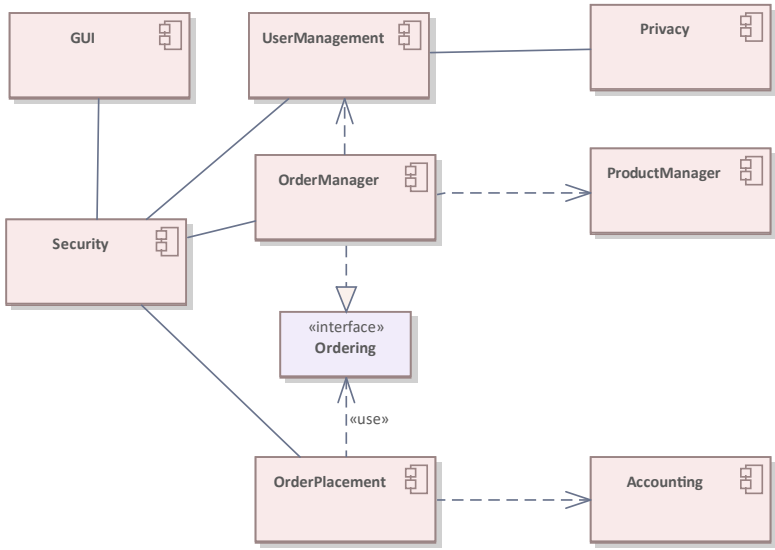
Obr. 5.21: Komponenty e-obchodu.

Pre niektoré asociácie vieme, že vlastne predstavujú závislosti. Situáciu znazorňuje obrázok 5.22.



Obr. 5.22: Závislosti medzi komponentmi e-obchodu.

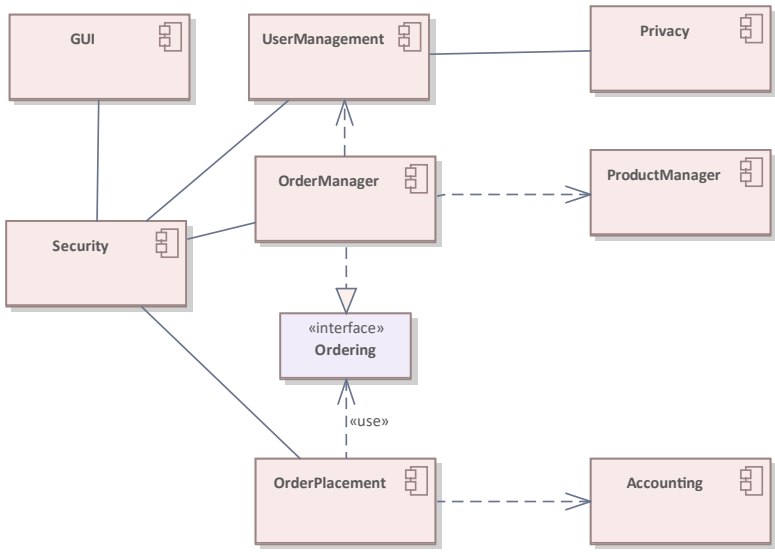
Závislosť ako taká, aj keby sme k nej pridali stereotyp, nevyjadruje doménové pozadie vzťahu komponentov. Za týmto účelom môžeme použiť rozhrania, ako ukazuje obrázok 5.23. V diagramoch komponentov rozhrania často nepredpisujú žiadne operácie – zámer je vyjadrený iba názvom. V danom prípade na základe názvu rozhrania vidíme, že sa vzťah komponentov týka objednávania. Zavedením rozhrania sa navyše stáva nepriamym.



Obr. 5.23: Vzťahy medzi komponentmi e-obchodu prostredníctvom rozhrania.

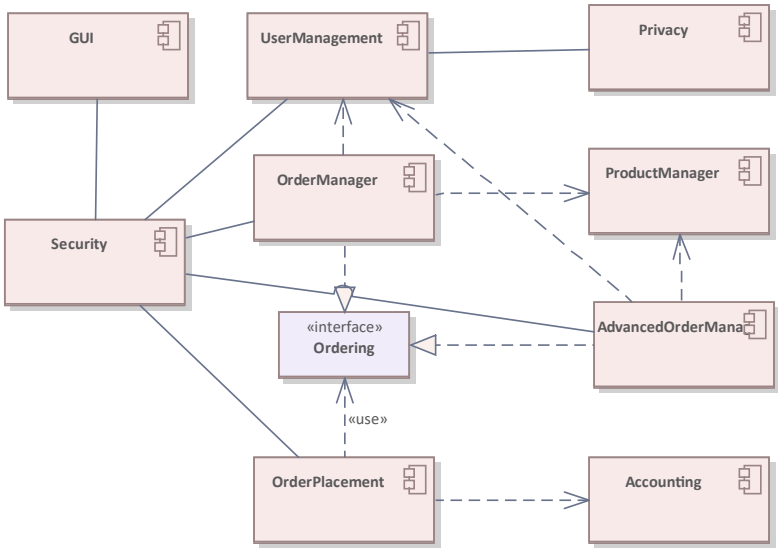
Obrázok 5.24 zobrazuje rozhranie *Ordering* v čiastočnej lízatkovej notácii, ktorú sme v jej úplnej forme videli na obrázku 5.19. V tomto prípade je druhá časť konektora ponechaná v bežnej forme závislosti. Takýto ústupok je nutné spraviť v niektorých

nástrojoch na modelovanie v UML. Na druhej strane, takto je jednoduchšie znázorniť prípad, keď viac komponentov používa to isté rozhranie.

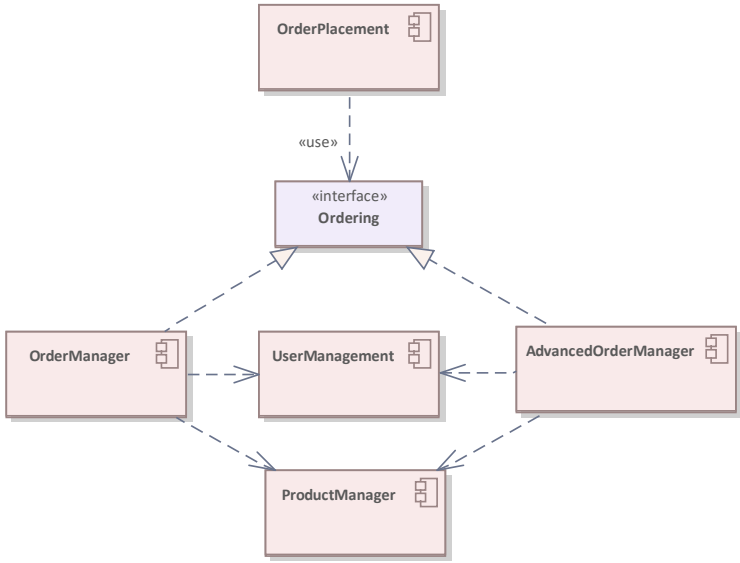


Obr. 5.24: Rozhranie v lízatkovej notácii.

Ako je vidno z obrázku 5.25, zavedením komponentu *AdvancedOrderManager*, ktorý tiež realizuje rozhranie *Ordering* diagram komponentov sa zneprehľadňuje. Situáciu rieši znázornenie čiastkového diagramu komponentov s komponentmi, ktoré realizujú rozhrnanie *Ordering*, ako ukazuje obrázok 5.26.



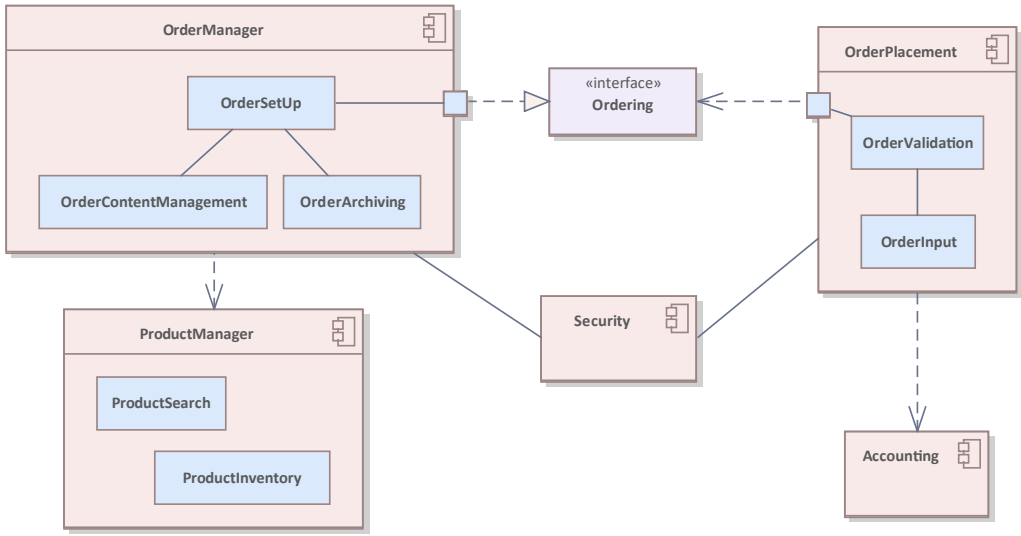
Obr. 5.25: Zavedením komponentu *AdvancedOrderManager* diagram komponentov sa zneprehľadňuje.



Obr. 5.26: Znázornenie čiastkového diagramu komponentov s komponentmi, ktoré realizujú rozhrnanie *Ordering*.

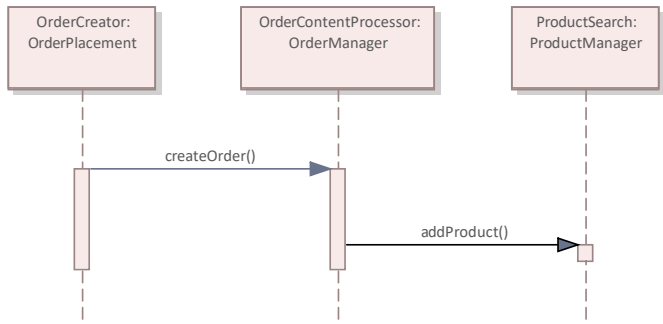
Vnútro komponentov môžeme vyjadriť vo forme, ktorá sa neviaže na triedy. Zložky, z ktorých komponent pozostáva, sa označujú ako časti (angl. parts). Evokujú skôr

roly, ktoré príslušné prvky implementačnej štruktúry zohrajú. Na okrajoch komponentov možno indikovať miesta prepojenia, ktoré sa označujú ako porty. Príklad je na obrázku 5.27.



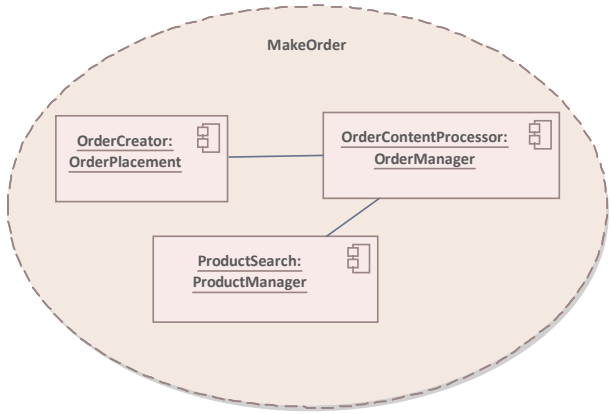
Obr. 5.27: Kompozitná štruktúra.

Tak ako pri triedach (pozri časť 3.12), interakciu inštancií komponentov – ktoré sa označujú tiež ako komponenty – možno vyjadriť diagramom sekvencií. Diagram sekvencií na obrázku 5.28 zachytáva partikulárnu situáciu interakcie komponentov, bez snahy zachytiť všetky možnosti (preto neobsahuje kombinované fragmenty). Správy nemusia nevyhnutne predstavovať volania operácií komponentov, ktoré sa aj tak v komponentoch zvyčajne neuvádzajú.



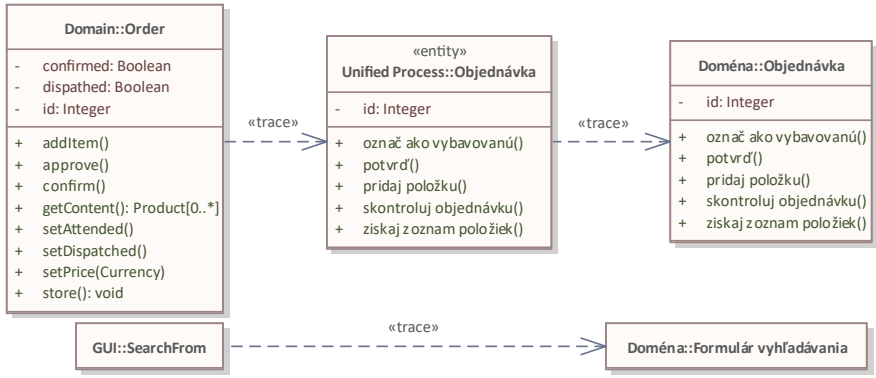
Obr. 5.28: Diagram sekvencií zachytáva partikulárnu situáciu interakcie komponentov.

Spoluprácu komponentov možno ako pri triedach vyjadriť aj kolaboráciou. Obrázok 5.29 zobrazuje trochu inú formu kolaborácie, v ktorej sú spolupracujúce komponenty zobrazené vnútri. V takom prípade sú to inštancie.



Obr. 5.29: Diagram kolaborácie.

Podobne ako pri postupne rozpracovávaných verziách tried (pozri časť 4.5), návrhové triedy ako prvky bližšie k implementácii možno spojiť vzťahmi «trace» s komponentmi, ktoré implementujú.



Obr. 5.30: Zaznamenanie pôvodu triedy *OrderManager* z rovnomenného komponentu.

Tak ako triedy, aj komponenty možno organizovať do balíkov.

6 Stavové diagramy

UML umožňuje modelovanie stavového priestoru pomocou stavových diagramov (angl. state diagrams), ktoré sú v angličtine častejšie označované ako *state machine diagram*, t. j. diagramy stavových strojov alebo iba stavové stroje. Tieto diagramy boli používané dávno pred vznikom UML, ako nakoniec aj väčšina diagramov UML. Stavové diagramy predstavujú alternatívny, ale aj komplementárny spôsob uvažovania o softvérovom systéme. Stavové diagramy môžu byť veľmi jednoduché, ale aj veľmi prepracované, s identifikovanými akciami prepojenými na konkrétne operácie a stavmi so zložitou vnútornou štruktúrou.

Časť 6.1 vysvetľuje identifikáciu stavov a prechodov. Časť 6.2 sa podrobnejšie zaoberá prechodmi. Časť 6.3 ozrejmuje pojem kompozitného stavu. Časť 6.4 ozrejmuje pojem paralelného stavu.

6.1 Identifikácia stavov a prechodov

Ak niečo nemôžeme v danom momente urobiť, hovoríme, že nie sme v *stave* to urobiť. Rovnako, ani s objednávkou sa nedá robiť hocičo kedykoľvek. Objednávka môže byť v rôznych stavoch, ako napríklad:

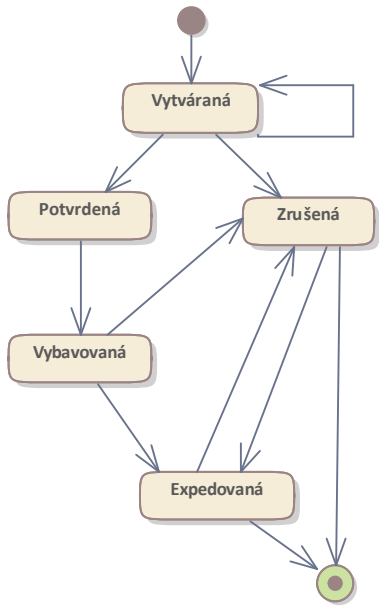
- vytváraná
- potvrdená
- vybavovaná
- expedovaná
- zrušená

Bez ohľadu na výstižnosť pomenovania stavu, vždy je potrebné vysvetlenie jeho významu. Napríklad, že objednávka je vytváraná môže znamenať, že ešte nevznikla a že do nej nie je možné pridávať položky. Ale v našom príklade je to práve naopak: objednávka je vytváraná, kým sa do nej pridávajú položky. Ak by sme potrebovali odlíšiť iniciálny stav objednávky od stavu, keď je napĺňaná položkami, mohli by sme uvažovať o stavoch *inicializovaná* a *napĺňaná*. Cieľom však nie je identifikovať všetky

možné označenia stavov, ale iba charakteristické stavy, ktoré skutočne určujú, čo sa dá s objednávkou v danom momente robiť.

Čím je dané, že daný objekt je v určitom stave? V implementácii sa to redukuje na hodnoty atribútov alebo atribútov agregovaných objektov. Tieto atribúty nemusia vyjadrovať stavy explicitne. Stav môže byť daný kombináciou hodnôt rôznych atribútov. Pri identifikácii stavov objednávky sme o žiadnych atribútoch neuvažovali. Z identifikovaných stavov môžeme odvodiť, aké atribúty sú pre objednávku potrebné, ale na samotnú identifikáciu stavov môžeme abstrahovať od ich implementácie, a to aj z hľadiska ich využitia pri vykonávaní operácií. Stav je abstrakcia pripravenosti realizovať správanie.

Po realizácii správania, stavový stroj prechádza do iného stavu. Nie je to hociktorý iný stav, ale stav, ktorý zodpovedá zmeneným okolnostiam. Takto je to aj so stavmi objednávky, ako ukazuje stavový diagram na obrázku 6.1.



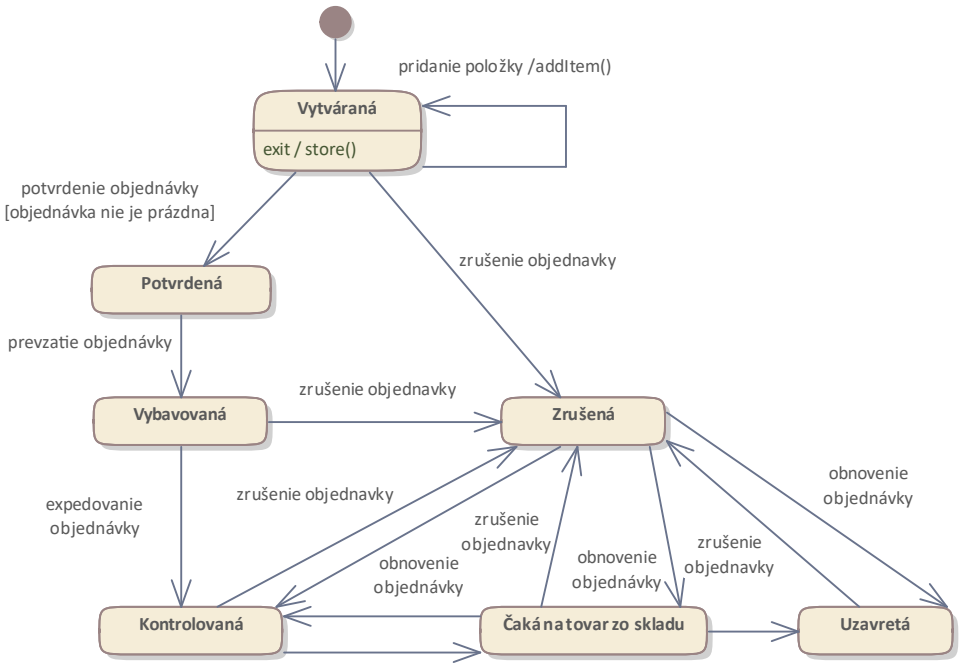
Obr. 6.1: Základná identifikácia stavov a prechodov.

Stavový stroj začína pracovať v stave, ktorý nasleduje po počiatočnom stave.[1] Ten je indikovaný výplneným krúžkom. Stavový stroj končí dosiahnutím finálneho stavu, ktorý je indikovaný symbolom terča.

[1]Počiatočný stav sa nepokladá za skutočný stav, ale za pseudostav, ktorý iba indikuje, kde stavový stroj začína.

6.2 Spresnenie prechodov

Tak ako stav predstavuje abstrakciu pripravenosti realizovať správanie, prechod (angl. transition) medzi stavmi predstavuje abstrakciu schopnosti dostať sa do určitého stavu. Prechod spôsobuje určitá *udalosť* (angl. event) alebo, v terminológii UML, *spúšťač* (trigger). Obrázok 6.2 trochu prepracovaný stavový diagram objednávky s vyznačenými udalosťami: pridanie položky, potvrdenie objednávky, prevzatie objednávky atď.[2] Hneď ako nastane udalosť, ktorá je vyznačená na prechode, ktorý vedie z daného stavu, stavový stroj prechádza do stavu, do ktorého vedie daný prechod.



Obr. 6.2: Spresnenie prechodov v stavovom diagrame.

Spúšťače v UML formálne predstavujú signály, ktoré sa znázorňujú obdĺžnikmi s trojuholníkovým ukončením, ako ukazuje obrázok 6.3. To znamená, že spúšťač zrušenia objednávky alebo obnovenia objednávky, ktorý sa v stavovom diagrame objednávky opakuje veľakrát, predstavuje ten istý signál. Spomeňme si na diagram aktivít uvedený na obrázku 3.9 v časti 3.11. V ňom sa vyskytovala špeciálna akcia prijatia signálu s trojuholníkovým výsekom, ktorý má indikovať, že táto akcia očakáva signál.

[2]Diagram nie je dokončený: nie sú v ňom vyznačené udalosti na všetkých prechodoch.

Obr. 6.3: Spúšťače ako signály.

Prechod môže byť spojený s realizáciou určitej akcie (angl. action) alebo, v terminológii UML, efektu (angl. effect). Efekt môže byť iba pomenovaný, ale aj prepojený na určitú operáciu. Stavový stroj na obrázku 6.2 sa zdržiava v stave *Vytváraná* kým sa do objednávky pridávajú položky, čo je indikované cirkulárnym prechodom. Pridanie položky je realizované operáciou *addItem()* triedy *Order*.

Všimnime si, že definovaním akcií odhaľujeme operácie, ktoré má poskytovať systém alebo jeho časť. Ak by sme v tomto boli dôslední, stavový diagram sa začne podobať na diagram aktivít.

Akcie možno prisudzovať aj momentu vstupu do stavu (angl. entry), zotrvaniu v stave (angl. do) alebo opustení stavu (angl. exit). V našom príklade sme v stave *Vytváraná* nastavili, že sa po opustení tohto stavu objednávka uloží (operácia *store() triedy Order*).

Aj keď nastane príslušná udalosť, t. j. spúšťač, aktivácia prechodu môže byť nežiaduca, ak nie je splnená určitá podmienka (angl. condition). Stavový stroj na obrázku 6.2 podmieňuje prechod zo stavu *Vytváraná* do stavu *Potvrdená* tým, že objednávka nie je prázdna. Na toto sa používajú strážcovia (angl. guards), s ktorými sme sa stretli v diagramoch aktivít (pozri časť 3.11).

Značenie prechodov v stavových diagramoch možno sumarizovať takto:

event [condition] / action

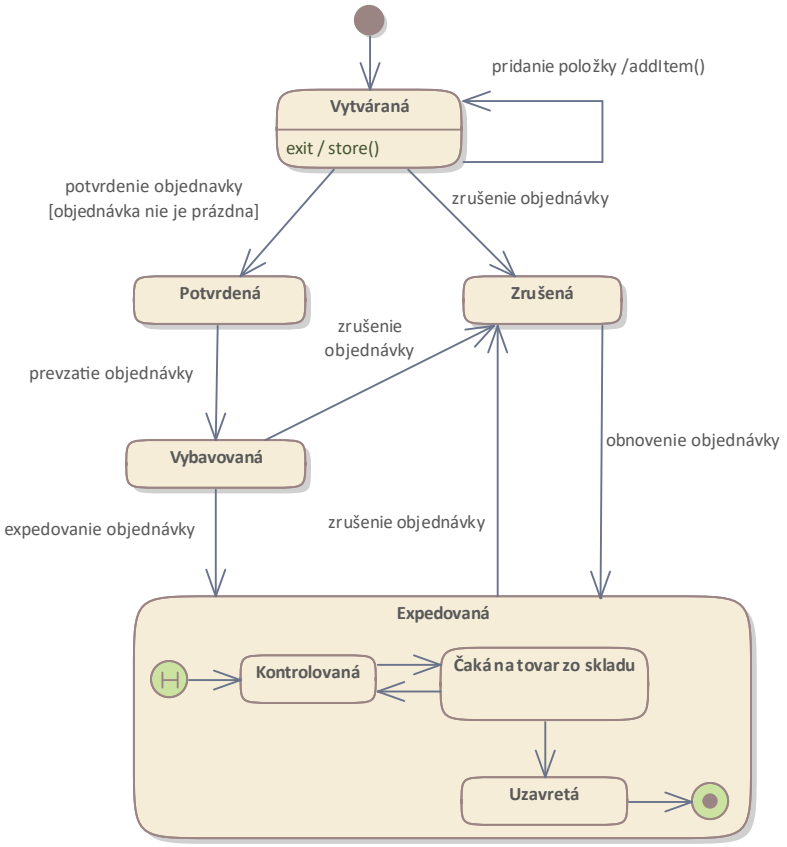
V terminológii UML je to:

trigger [guard] / effect

6.3 Kompozitné stavy

Čo ak sa z mnohých stavov možno dostať do určitého stavu alebo dokonca do viacerých stavov? Túto situáciu máme na obrázku 6.2. Riešením je zavedenie kompozitného

stavu ako na obrázku 6.4. Stav *Expedovaná* obsahuje malý stavový stroj, ktorý začína pracovať vstupom do tohto stavu.



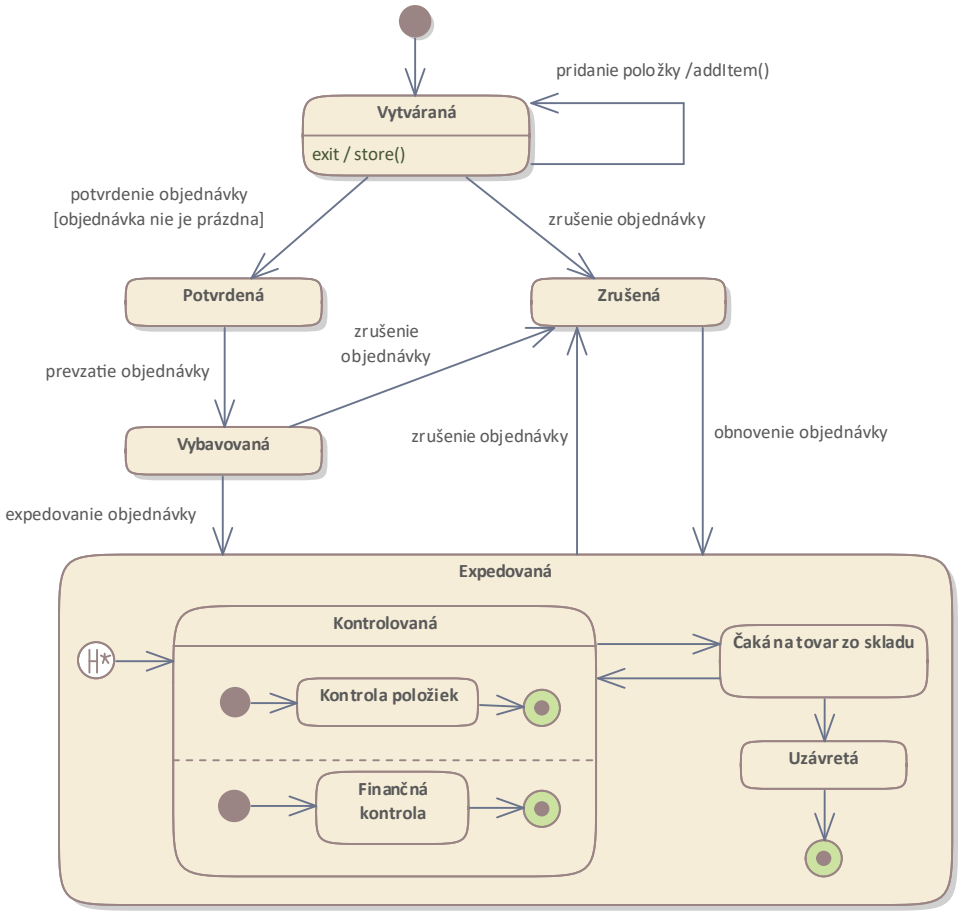
Obr. 6.4: Stavový diagram s kompozitným stavom.

Namiesto počiatočného stavu sa najčastejšie uvádza historický stav (angl. history state), ktorý je indikovaný písmenom *H* v krúžku. Ak je použitý tento stav, potom sa, po prípadnom návrate, vnútorný stavový stroj ocitne v stave, v ktorom bol, keď bol opustený.

6.4 Paralelné stavy

Všimnime si, že pri kompozitných stavoch stavový stroj môže byť naraz vo viacerých stavoch. Napríklad, kým je expedovaná, objednávka môže čakať na tovar zo skladu. Ešte výraznejšie je to, ak pripustíme, že viaceré stavové stroje týkajúce sa tej istej

entity môžu prebiehať paralelne. Obrázok 6.5 znázorňuje takúto situáciu. Kontrola objednávky prebieha v zmysle položiek a financií, pričom sú tieto procesy nezávislé. Stav *Expedovaná* je rozdelený do dvoch regiónov, v ktorých prebiehajú nezávislé stavové stroje.



Obr. 6.5: Stavový diagram s paralelným stavom.

Regiónov môže byť viac. Takýto stav sa označuje ako paralelný (nie v terminológii UML).

Ako vidíme, aj vnútorné stavy môžu byť kompozitné, hoci prílišné vnáranie môže byť kontraproduktívne. V takýchto prípadoch má význam použiť hlboký historický stav (angl. deep history state), ktorý označuje ako H^* v krúžku. Potom si vnútorný stavový stroj pamätá stav, v ktorom bol, bez ohľadu na to, ako hlboko ten stav je.

7 Detailný model operácie a ako sa mu vyhnúť pomocou OCL

Modelovaním operácií preverujeme štruktúru a identifikujeme jej ďalšie časti. Modelovaním štruktúry identifikujeme ďalšie potrebné operácie nad jej časťami. Takto sa model softvérového systému postupne rozrastá. Operáciu je v UML možné vyjadriť pomocou diagramu sekvencií, diagramu aktivít a diagramu komunikácie. V určitých prípadoch takáto grafická forma môže byť prijateľnejšia ako kód, lebo skrýva detaily syntaxe partikulárneho programovacieho jazyka. Samozrejme, potrebné je rozumieť syntaxi UML.

Modely operácií môžu byť pomerne zložité, a ich vyššia čitateľnosť oproti kódu je diskutabilná. Ani z úplného vyjadrenia operácie grafickým modelom, ani z jej kódu bezprostredne nevyčítame jej zámer. Ten je možné lepšie vystihnúť prostredníctvom formálneho vyjadrenia podmienok spojených s operáciou, na čo UML poskytuje textový jazyk, ktorý sa volá Object Constraint Language (OCL).

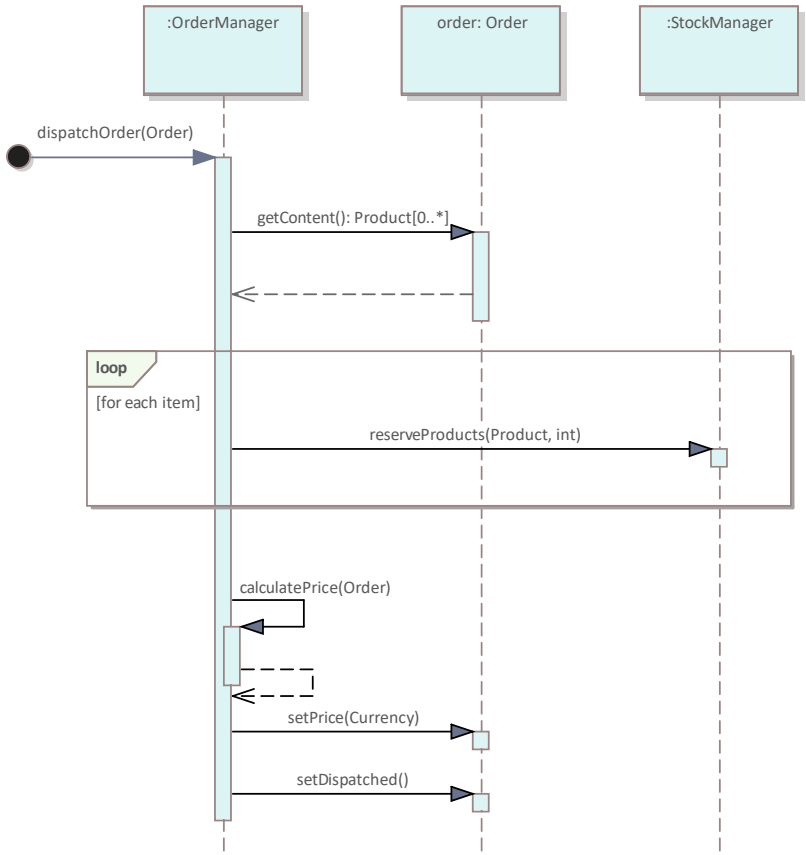
Časť 7.1 ukazuje, ako vyjadriť operáciu pomocou diagramov sekvencií, diagramov komunikácie a diagramov aktivít. Časť 7.2 vysvetľuje operáciu ako službu a ako ju vymedziť prostredníctvom predpokladov, dôsledkov a invariantov vyjadrených v OCL. Časť 7.3 vysvetľuje pozíciu predpokladov, dôsledkov a invariantov pri prekonávaní (angl. overriding). Časť 7.4 poukazuje kontext OCL výrazu.

7.1 Detailný model operácie

Ako už bolo spomenuté v časti 3.12, diagramy sekvencií sa pôvodne používali na vyjadrenie partikulárnych situácií inštancií. Takéto diagramy sekvencií neobsahujú kombinované fragmenty, ako sme už mali možnosť vidieť na obrázku 5.28 v časti 5.4.

Ak použijeme kombinované fragmenty, môžeme zachytiť celý priebeh operácie. Príklad je na obrázku 7.1. Diagram sekvencií začína prvou správou, čo v prípade vyjadrenia celej operácie musí byť volanie tejto operácie. Nevieme však, kto túto operáciu volá, a ani to nechceme uviesť. Preto táto prvá správa prichádza z neznáma, čo je

indikované symbolom vyplneného krúžku, ktorý sa v UML používa na indikáciu začiatku v rôznych diagramoch. Takáto správa sa označuje ako nájdená správa (angl. found message).



Obr. 7.1: Operácia *dispatchOrder()* vyjadrená diagramom sekvencií.

Nájdená správa iniciuje špecifikáciu vykonávania, v rámci ktorej musí byť vyjadrený celý priebeh operácie. Otázkou je, do akej hĺbky zobrazovať tok riadenia. V kóde je v tele operácie viditeľná iba prvá úroveň, a tak je to prirodzená voľba aj pre diagram sekvencií. Notácia umožňuje pokračovať do ľubovoľnej hĺbky, a tak by sme mohli mať ďalšie správy z vyvolaných operácií. Ak sú tieto operácie vyjadrené v samostatných diagramoch sekvencií, dochádza k redundancii a musíme vyjadrenia ich priebehu zosúlaďovať.

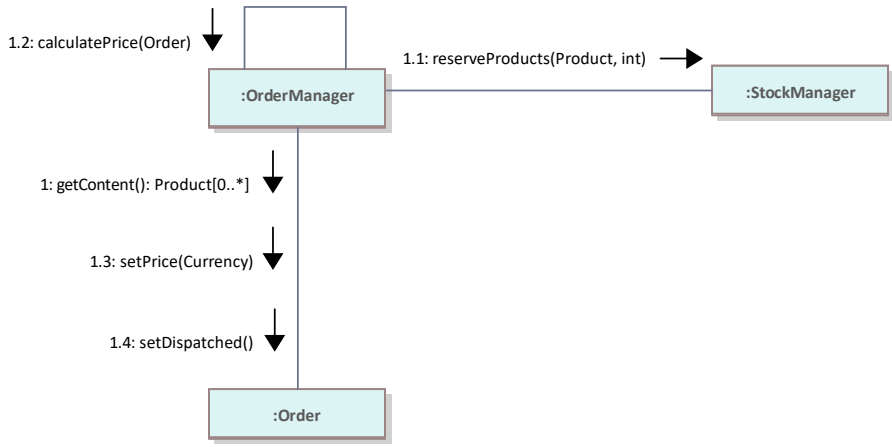
Všetky správy v diagrame sekvencií na obrázku 7.1 sú synchrónne: špecifikácia vykonávania po odoslaní správy čaká na návratovú správu, resp. na dokončenie vyvolanej operácie. Toto je najčastejšie aj potrebné, lebo pre pokračovanie volajúcej operácie zvyčajne potrebujeme výsledok volanej operácie, či vo forme návratovej hodnoty

alebo vedľajšieho účinku. Ak potrebujeme, aby volajúca operácia pokračovala bez čakania, treba to indikovať správou s bežnou, otvorenou (nevyplnenou) šípkou, ktorou sa označuje asynchrónna správa.

Dĺžka špecifikácie vykonávania nie je úmerná času. Príznačné je len poradie správ: ich sekvencia. Keby to bolo potrebné, čas sa dá indikovať poznámkou. V takom prípade môže mať význam kresliť správy aj šikmo, aby bolo vidno, že samotné odoslanie a prijatie správy trvajú určitý čas.

Návratové správy sa značia prerušovanou hranou ukončenou otvorenou šípkou. Zvyčajne sa vynechávajú. V našom príklade sú dve také správy: *getContent()* a *calculatePrice()*.

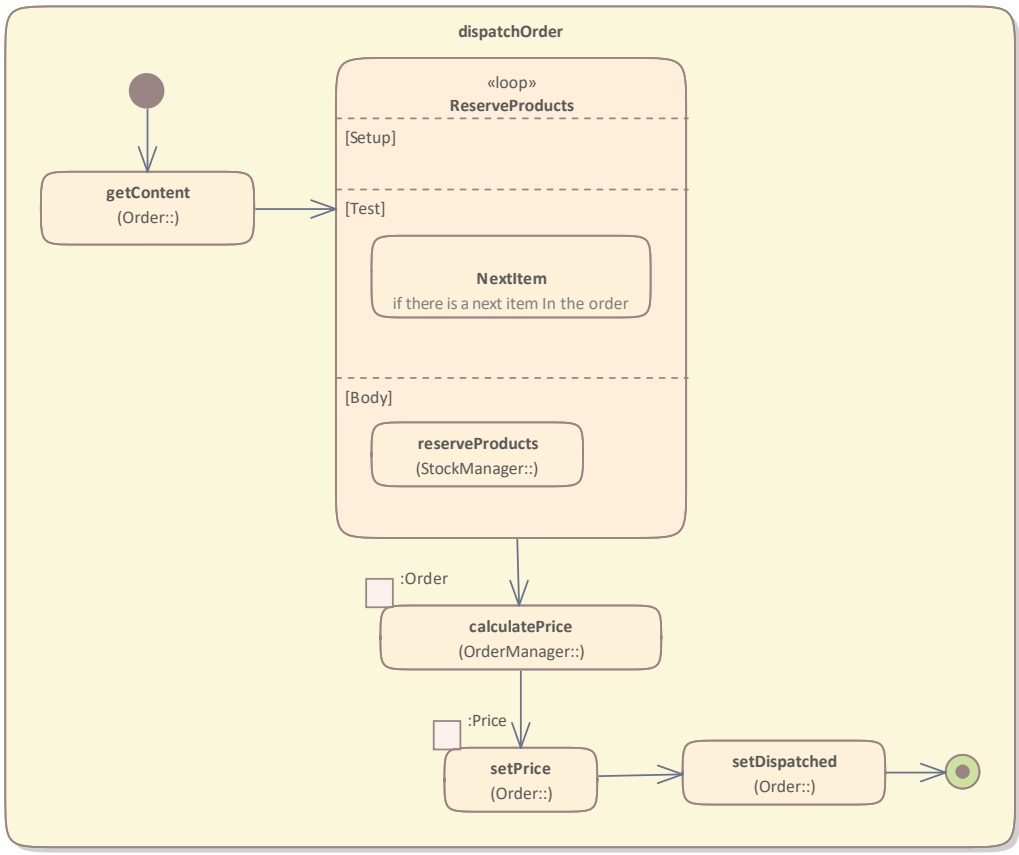
Sekvencie možno vyjadriť aj diagramom komunikácie.[1] Diagram komunikáce operácie *dispatchOrder()* je na obrázku 7.2.



Obr. 7.2: Operácia *dispatchOrder()* vyjadrená diagramom komunikácie.

Operáciu možno vyjadriť aj diagramom aktivít, ako ukazuje obrázok 7.3. V takom prípade, celá operácia predstavuje jednu aktivitu. Akcie, z ktorých pozostáva, sú naviazané na operácie, ktorých volania reprezentujú.

[1] Pôvodne sa tento diagram označoval ako diagram kolaborácie. Kolaborácia v UML teraz znamená niečo iné (pozri časti 3.15 a 5.4).



Obr. 7.3: Operácia *dispatchOrder()* vyjadrená diagramom aktivít.

Daný diagram aktivít demonštruje použitie štruktúrovanej aktivity na vyjadrenie slučky namiesto staršieho spôsobu založeného na rozhodovacích a spájacích uzloch, ktorý bol použitý v časti 3.11).

V diagrame sú použité kliny (angl. pin) na aktivitách (vo forme štvorčekov) na vyznačenie prenášaných objektov namiesto ich vloženia medzi akcie ako v časti 3.11).

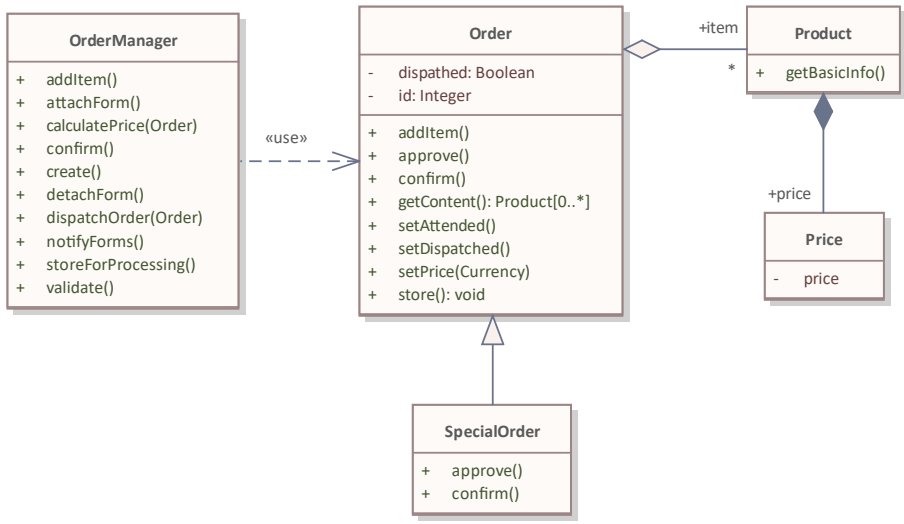
7.2 Operácia ako služba

Ani z úplného vyjadrenia operácie grafickým modelom nie je možné vyčítať priamo jej zámer. Skúsme sa nad tým zamyslieť inak. Stroj vykonáva určitú operáciu nad určitým predmetom, ale očakáva, že tento predmet bude mať určité vlastnosti (váha, rozmery, tvar...). Má stroj samotný preverovať, či výrobok spĺňa podmienky? Aj keby stroj bol takýto robustný, pravdepodobne by mal vyčlenený predstupeň, ktorý

zabezpečuje príslušnú kontrolu na vstupe.

Podobne by mohla byť zabezpečená aj kontrola na výstupe. Keby v reťazovom spracovaní každý stroj bol robustný, zbytočne by to zvyšovalo zložitosť ich umiestnenia a transparentnosť procesu, lebo by sme stroje aj zreťazili tak, aby výstup z jedného vyhovoval vstupu na nadväzujúcom stroji. Analogická situácia vzniká aj medzi operáciami (funkciami, metódami, procedúrami...) v programovaní.

Obrázok 7.4 pripomína prvky, ktoré sa týkajú objednávania v e-obchode. Pre jednoduchosť neberieme do úvahy balíky, ako ani priebeh ceny v čase. Predpokladáme, že produkt má len jednu cenu.



Obr. 7.4: Prvky, ktoré sa týkajú objednávania v e-obchode.

Operácia *OrderManager::dispatchOrder(order: Order)* nemôže expedovať už expedovanú objednávku. Či je objednávka expedovaná, je čitateľné z jej atribútu *dispatched*, a tak by sme mohli zapísať, že pred jej vykonaním má platiť nasledujúci *predpoklad* (angl. precondition):

pre `OrderMustNotBeAllreadyDispatched`: **not** `order.dispatched`

Po ukončení, operácia garantuje, že objednávka bude expedovaná, t. j. po jej vykonaní má platiť nasledujúci *dôsledok* (angl. postcondition):

post `OrderIsDispatched`: `order.dispatched`

Niektoré podmienky majú zostať zachované. Napríklad, ak je objednávka nulová, nemôže byť expedovaná, t. j. počas jej vykonávania bude dodržaný nasledujúci *invariant*:

```
inv ZeroPriceOrderCantBeDispatched:
   self.item.price.price->sum() = 0 implies not dispatched
```

Invariant platí vždy pre aktuálny objekt, ktorý sa označuje výrazom `self`. V danej situácii, keď nedochádza k prekrytiu názvov s inými premennými, by výraz `self` mohol byť vynechaný. Keďže je invariant uvedený z perspektívy triedy *Order*, a cena je až v triede *Price*, musíme sa k nej dostať prostredníctvom vzťahov.

Výraz `self.item` z dôvodu násobnosti príslušného vzťahu agregácie by vrátil množinu inštancií triedy *Product*. Každý prvok množiny je jedinečný. Pre každý produkt máme len jeden objekt ceny, ktorý je tiež jedinečný, a tak je `self.item.price` stále množina. Výraz `self.item.price.price` je však už tzv. multimnožina alebo vrece (angl. bag), lebo čísla, ktoré reprezentujú ceny, jedinečné nie sú. Operácia `sum()` sčíta prvky akejkoľvek kolekcie prvkov.

Operácia je teda ako služba. Podmienky jej uskutočnenia sú regulované zmluvou:

> Za daných predpokladov, a ak služba bude poskytnutá, budú dosiahnuté dôsledky. Invarianty budú zachované.

Toto je idea za prístupom k vývoju softvéru, ktorý sa volá Design by Contract [Mey97]:

Problém je, že zmluva, podľa ktorej operácia pracuje, obvykle nie je vyjadrená explicitne. Ako vidíme, v modeloch v UML by zachytená byť mohla. Formalizmus, ktorý na to bol použitý v uvedených príkladoch je Object Constraint Language (OCL) [OMG14], ktorý je súčasťou UML.

7.3 Predpoklady, dôsledky a invarianty pri prekonávaní

Zoberme do úvahy operáciu *Order::confirm()*. Pre túto operáciu by mohli platiť nasledujúce predpoklady (komentár sa v OCL značí ako `--`):

```
pre CantConfirmEmptyOrder: not self.item->isEmpty()
--Neda sa potvrdit prazdna objednavka

pre CantConfirmOrderWorthMoreThan1000euros:
self.item.price.price->sum() <= 100000
--Neda sa potvrdit objednavka nad 100000 eur
```

Operácia *SpecialOrder::confirm()* prekonáva (angl. override) operáciu *Order::confirm()*. Zachováva podmienku, že sa nedá potvrdiť prázdna objednávka, ale umožňuje potvrdiť objednávku bez obmedzenia hodnoty:

```
pre CantConfirmEmptyOrder: not self.item->isEmpty()
--Neda sa potvrdit prazdna objednavka

pre CantConfirmOrderWorthMoreThan1000euros: true
--Specialna objednavka sa da potvrdit bez ohladu na jej hodnotu
```

Je toto problém? Aby sme to posúdili, musíme sa zamyslieť z pohľadu klientskeho kódu. Je známe, že pri dedení musí byť dodržaný Liskovej princíp substitúcie (angl. Liskov substitution principle) [Lis87]:

> Ak pre každý objekt o_1 typu S jestvuje objekt o_2 typu T taký, že pre všetky programy P definované v zmysle T správanie P je nezmenené, keď o_1 nahradí o_2, potom je S podtypom T.

Inak povedané, ak klientsky kód funguje korektne s objektom určitého typu, musí fungovať korektne aj s objektom jeho podtypu. Korektne znamená, že nezlyhá, a ani nespôsobí to, že sa atribúty objektu zmenia tak, že daný objekt už nebude korektný.

Predstavme si slučku, v ktorej budú potvrdzované objednávky:

```
for (Order order : orders)
   if (calculatePrice(order) <= 100000)
      order.confirm();
```

Vďaka polymorfizmu, zoznam `orders` môže obsahovať aj inštancie tried odvodených od triedy *Order*. Tento kód zbehne korektne aj pre inštanciu triedy *SpecialOrder*. Ak má hodnotu vyššiu ako 100000, nepotvrdí ju, čo je v poriadku, lebo takto je koncipovaný klientsky kód. Inštancia triedy *SpecialOrder* nebude žiadnym spôsobom narušená.

Vidíme, že zoslabenie predpokladu nespôsobuje problémy, kým jeho zosilnenie áno. Ako je to s dôsledkami? Ak by pre operáciu *SpecialOrder::confirm()* bol zavedený nasledujúci dôsledok:

```
pre OrderIsConfirmed: self.confirmed
```

bolo by to zosilnenie implicitného dôsledku operácie *Order::confirm()*:

```
pre OrderIsConfirmed: true
```

ktorý platí vždy, t. j. bez ohľadu na hodnotu atribútu *confirmed*. Klientsky kód, ktorý v slučke potvrdzoval objednávky, by zbehol korektne aj pre inštanciu triedy *SpecialOrder*. Nenarušil by ju.

Keby dôsledky boli priradené opačne, došlo by k zoslabeniu dôsledku. Klientsky kód, ktorý v slučke potvrdzoval objednávky, by mohol (správne) očakávať, že všetky objednávky budú potvrdené, vrátane tých typu *SpecialOrder*. Nasledovať by mohlo spracovanie, ktoré zlyhá, ak objednávka nie je potvrdená, k čomu môže dôjsť, lebo sme zoslabili dôsledok.

Čo sa týka invariantov, tie pri dedení majú charakter dôsledkov. Spomeňme si na invariant `ZeroPriceOrderCantBeDispatched`. Žiadna operácia nesmie priviesť objednávku do takého stavu, že je expedovaná, ak má nulovú hodnotu.

Sumárne by sa dalo povedať, že pri prekonávaní predpoklady operácie nesmú byť silnejšie, kým dôsledky a invarianty nesmú byť slabšie. Inak povedané, neprekáža nám, ak nejaká služba od nás chce na vstupe menej než sme boli pôvodne informovaní, ale nie je pre nás prijateľné, ak na výstupe dostaneme slabší výsledok, než nám bolo pôvodne sľúbené.

7.4 Kontext

Pre OCL výrazy uvedené v predchádzajúcej časti sme predpokladali, že sú zadané priamo k prvkom, na ktoré sa vzťahujú, t. j. predstavujú ich *kontext*. Graficky sa uvádzajú v poznámkach spojených s príslušnými prvkami, ale v nástrojoch na modelovanie v UML môžu byť len ukladané k príslušným prvkom bez zobrazenia vo forme poznámky. Ak OCL výrazy uvádzame zvlášť, potom je potrebné uviesť aj kontext. Napríklad:

```
context OrderManager::dispatchOrder(order: Order)

    pre OrderMustNotBeAllreadyDispatched: not order.dispatched

    post OrderIsDispatched: order.dispatched
```

8 Algebraická špecifikácia

Modelovanie v UML je súhra modelovania správania a štruktúry: jedno ťaha to druhé. Softvérový systém nevznikne, ak nerozpracujeme aj jedno, aj druhé. Z používateľského hľadiska – a používateľ môže byť aj programátor, ktorý používa určitý rámec – je dôležité iba správanie. Je možné dostatočne presne vyjadriť správanie bez predurčenia štruktúry?

V časti 8.1 sa pozrieme na zásobník a možnosti jeho implementácie a vyjadrenia modelom. Časť 8.2 prezentuje použitie techniky algebraickej špecifikácie na príklade zásobníka ako generického typu. Časť 8.3 demonštruje, ako sa algebraická špecifikácia dá použiť pre vyjadrenie podstaty bežnej triedy.

8.1 Zásobník

Zásobník predstavuje známy spôsob ukladania údajov. V angličtine sa volá *stack*, čo znamená jednoducho *stoh*. Viditeľný je dokonca aj etymologický súvis. Zásobník sa presne podľa toho aj správa. Vkladá sa vždy na vrch. Prvok vložený na vrch je jediný viditeľný a jediný sa dá vybrať. Po výbere prvku, ktorý je na vrchu, sprístupni sa nasledujúci prvok. Toto sa označuje ako LIFO: last in, first out.

Zásobník má rozsiahle uplatnenie v rôznych algoritmoch. Zvlášť je užitočný pri kontroly syntaxe a pri vyhodnocovaní aritmetických výrazov. Používa sa aj na ukladanie volaní procedúr (funkcií, metód...) za účelom zabezpečenia toku riadenia v programe. Pri nekonečnej rekurzii dostávame typickú hlášku *stack overflow*, lebo sa takýto zásobník volaní (angl. call stack) preplní.

Vo všetkých týchto použitiach, zásobník sa správa rovnako: typ údajov uchovávaných v ňom nemá na to vplyv. To znamená, že pri špecifikácii podstaty zásobníka môžeme abstrahovať od typu údajov, t. j. zásobník ako typ údajov sa stáva abstraktným. Abstrahujeme vlastne aj od štruktúry, ktorá zabezpečuje potrebné správanie pri práci so zásobníkom. Zásobník aj je jedným zo známych abstraktných typov údajov (angl. abstract data type), medzi ktoré patria aj zoznam, rad (FIFO: first in, first out), postupnosť, množina, strom, graf atď.

Zásobník sa dá implementovať rôznymi spôsobmi. Programovacie jazyky bežne poskytujú polia (angl. array). Ak uchovávame index posledne vloženého prvku, vždy ho môžeme prečítať. Pomocou inkrementácie a dekrementácie indexu posledne vloženého prvku implementujeme vloženie prvku a výber prvku. V poli nemusia byť uchované iba jednoduché hodnoty. Môžu tam byť odkazy na štruktúry údajov. Najčastejšie sú to ukazovatele alebo referencie na objekty.

Veľkosť poľa sa stanovuje vždy pri jeho vytvorení. Ak by sme objekty spájali prostredníctvom referencií uchovávaných priamo v nich, dostaneme veľmi flexibilnú a voľne rastúcu štruktúru, ktorá sa volá spájaný zoznam (angl. linked list). Postačuje jednosmerné spájanie. Pre implementáciu zásobníka stačí potom obmedziť vkladanie iba na chvost (angl. tail) zoznamu a uchovať referenciu na posledne vložený prvok. Flexibilita spájaného zoznamu prichádza za cenu efektivity a to aj časovej, aj pamäťovej, ale je to možná implementácia často používaná ako cvičenie v programovaní.

Spájaný zoznam by sme ľahko mohli vystihnúť diagramom tried. K tomu by sme mohli pridať triedu, ktorá by reprezentovala samotný zásobník a jeho operácie. Následne by sme ich mohli modelovať diagramami sekvencií, diagramami komunikácie alebo diagramami aktivít, ako aj každú inú operáciu. Problém je, že by sme pracovali so štruktúrou, ktorá predstavuje len jednu z možných. Zásobníku by sme prisudzovali detaily, ktoré nemá, čím by sme dosiahli prešpecifikáciu (angl. overspecification) [Mey97]. Programátori by nasledovali nadbytočné štrukturálne detaily, a nie to, čo sme im chceli ukázať: ako sa zásobník správa.

V stavových diagramoch štruktúra nevystupuje, čo znie sľubne. Operácie nad zásobníkom by boli udalosťami. Problém je, že stavové diagramy nevystihujú vzťahy medzi prechodmi, ale stavy. Tie by mohli byť iba prázdny a neprázdny. Vystihli by sme to, že sa pridaním prvku na prázdny zásobník dostane neprázdny zásobník, ale nevedeli by sme špecifikovať, kedy sa z neprázdneho zásobníka stane prázdny, lebo zásobník nepozná počet prvkov, ktorý je na ňom. Rovnako, význam operácie pozretia prvku na vrchu zásobníka by bol nešpecifikovateľný.

Mohli by sme uvažovať o prípadoch použitia. V podstate by to bola akási analógia CRUD prípadu použitia (pozri časť 3.7). Problémom je, že ak by sme snažili o presnejší opis než slovný opis podstaty zásobníka zo začiatku tejto časti, aj tak sa začne vynárať štruktúra lebo akcie v prípadoch použitia sú opisované nad štruktúrou.

8.2 Algebraická špecifikácia zásobníka

Ak opis operácie nemôže byť založený na predpokladanej štruktúre, aby sme ju predčasne nešpecifikovali, na čom ho potom založiť? Zostávajú nám už iba operácie, a tak sa musíme spoľahnúť na ne.

To, čo robí zásobník zásobníkom, je správanie. Operácie nad zásobníkom mávajú rôzne

pomenovania, ale tradičné názvy sú tieto:

- push – vloží prvok na vrch zásobníka
- top – vráti prvok z vrchu zásobníka
- pop – odoberie prvok z vrchu zásobníka

Ak by sme označili vkladaný prvok ako `e`, potom jeho vloženie môžeme zapísať takto:

`push(e)`

Ak by operácia `push()` vracala odkaz na zásobník, potom by

`top(push(e))`

bol práve prvok `e`.

Presne takto sa postupuje pri tvorbe algebraickej špecifikácie. Operácie sú v matematickom zmysle funkcie. Celý zásobník by potom mohol byť špecifikovaný takto (vytvorené na základe Meyerovho príkladu [Mey97]):

Typy

$Stack[E]$

Funkcie

$new : Stack[E]$
$empty : Stack[E] \rightarrow Boolean$
$push : Stack[E] \times E \rightarrow Stack[E]$
$pop : Stack[E] \nrightarrow Stack[E]$
$top : Stack[E] \nrightarrow E$

Axiómy

$\forall\, e : E, s : Stack[E]$

$A1 : top(push(s, e)) = e$
$A2 : pop(push(s, e)) = s$
$A3 : empty(new)$
$A4 : \neg\ empty(push(s, e))$

Predpoklady

$pop(s : Stack[E])$ requires $\neg\ empty(s)$
$top(s : Stack[E])$ requires $\neg\ empty(s)$

Algebraická špecifikácia najprv deklaruje typy, ktoré v nej vystupujú okrem implicitne definovaných typov, akým je *Boolean*. Jedným z deklarovaných typov je vždy aj typ, ktorého sa špecifikácia týka. Špecifikácia následne deklaruje funkcie nad špecifikovaným typom. V terminológii programovania, doménou (definičný obor) funkcie sú jej parametre, a kodoménou (obor hodnôt) jej návratová hodnota, t. j. ich typy. Ak ich je viac, bývajú združené pomocou kartézskeho súčinu, čím tvoria n-tice.

Jadrom špecifikácie sú axiómy: tvrdenia, ktoré platia vždy. Na axiómu A1 sme prišli už v úvodných úvahách. Tu je len explicitne uvedený zásobník ako parameter, ktorý sme my pokladali za implicitný. Podľa A2, po funkcii *pop*() dostaneme späť zásobník bez vloženého prvku e.

Axiómy A3 a A4 sú tu pre úplnosť. Funkcia *new* nemá parametre: z ničoho vytvára zásobník. Ak funkcia nemá parametre, môžeme vynechať zátvorky. Funkcia *empty*() vracia hodnotu pravda (angl. true) alebo nepravda (angl. false) podľa toho, či je zásobník prázdny alebo nie. Axióma A3 hovorí, že nový zásobník je prázdny, kým axióma A4 hovorí, že zásobník, na ktorý je vložený prvok, prázdny nie je.

Funkcie môžu byť úplné (angl. total) a parciálne (angl. partial) podľa toho, či je pre každý prvok domény definovaný prvok kodomény, t. j. hodnota funkcie. Pri úplných funkciách sa medzi doménou a kodoménou používa označenie $\rightarrow$, a pri parciálnych $\nrightarrow$. Napríklad, funkcia *top*() nemá definovanú hodnotu pre prázdny zásobník.

V tejto súvislosti vystupujú ešte predpoklady a to presne v zmysle v akom sme sa nimi zaoberali v kapitole 7. Konkrétne, funkcie *top*() a *pop*() vyžadujú na vstupe neprázdny zásobník.

Vzniká otázka, či a kde sú uvedené dôsledky. Ak sa pozrieme bližšie, rozpoznáme ich v axiómach.

Funkcie chápeme matematicky. Netrápi nás, že niektoré vracajú celý nový zásobník, lebo algebraická špecifikácia je deklaratívna a nebude sa vykonávať na počítači.

8.3 Algebraická špecifikácia objednávky

Algebraická špecifikácia nie je obmedzená iba na generické abstraktné typy údajov. Dá sa použiť na akýkoľvek abstraktný typ údajov, ktorý možno vidieť za každou triedou. Trieda je vlastne implementácia abstraktného typu údajov [Mey97]. Tu je algebraická špecifikácia objednávky:

Typy

Objednavka, *Polozka*

Funkcie

$nova : Objednavka$
$pridajPolozku : Objednavka \times Polozka \to Objednavka$
$odoberPolozku : Objednavka \times Polozka \to Objednavka$
$expeduj : Objednavka \to Objednavka$
$zrus : Objednavka \to Empty$
$prazdna : Objednavka \to Boolean$

Axiómy

$\forall\, p : Polozka, o : Objednavka$

$A1 : odoberPolozku(pridajPolozku(o, p), p) = o$
$A2 : prazdna(nova)$
$A3 : \neg\, prazdna(pridajPolozku(o, p))$

Predpoklady

$odoberPolozku(o : Objednavka, p : Polozka)$ requires $\neg\, prazdna(o)$
$expeduj(o : Objednavka)$ requires $\neg\, prazdna(o)$

Všimnime si, že špecifikácia síce obsahuje funkciu *expeduj*(), ale táto funkcia nevystupuje v axiómach. To znamená, že jej výsledok nie je špecifikovaný. Ten by mal spočívať v zmene objednávky na expedovanú, ale výsledkom tejto operácie je proste objednávka. Na odlíšenie by sme mohli zaviesť typ *ExpedovanaObjednavka*. Ale čo, keby sme pridali operácie *uloz*(), *potvrd*() a pod.? Nastála by kombinatorická explózia typov.

Keby sme programovali, jednoducho by sme zaviedli atribút `expedovana`, čo sa v algebraickej špecifikácii spraviť nedá. Na nastavovanie atribútov sa často používajú metódy, aby sa klientsky kód nestal závislý od vnútornej štruktúry. V algebraickej špecifikácii stačí zaviesť funkciu, ktorá zisťuje, či je objednávka expedovaná. Následne môžeme sformulovať axiómu, ktorá bude hovoriť, že po uplatnení operácie *expeduj*(), objednávka je expedovaná. Tu je upravená algebraická špecifikácia objednávky s novou axiómou (A4) a predpokladmi:

Typy

Objednavka, *Polozka*

Funkcie

$nova : Objednavka$
$pridajPolozku : Objednavka \times Polozka \to Objednavka$
$odoberPolozku : Objednavka \times Polozka \to Objednavka$
$expeduj : Objednavka \to Objednavka$
$zrus : Objednavka \to Empty$
$prazdna : Objednavka \to Boolean$
$expedovana : Objednavka \to Boolean$

Axiómy

$\forall\, p : Polozka, o : Objednavka$

$A1 : odoberPolozku(pridajPolozku(o, p), p) = o$
$A2 : prazdna(nova)$
$A3 : \neg\ prazdna(pridajPolozku(o, p))$
$A4 : expedovana(expeduj(o))$

Predpoklady

$odoberPolozku(o : Objednavka, p : Polozka)$ requires
$\neg\ prazdna(o) \wedge \neg\ expedovana(o)$
$expeduj(o : Objednavka)$ requires $\neg\ prazdna(o) \wedge \neg\ expedovana(o)$
$zrus(o : Objednavka)$ requires $\neg\ expedovana(o)$
$pridajPolozku(o : Objednavka, p : Polozka)$ requires $\neg\ expedovana(o)$

Použiť algebraickú špecifikáciu na všetky triedy v softvérovom systéme by pravdepodobne nebolo veľmi účelné. Pri dôležitých triedach tento prístup však pomáha vymedziť význam operácií a identifikovať neúplnosť množiny poskytnutých operácií.

9 Modelovanie variantnosti softvéru

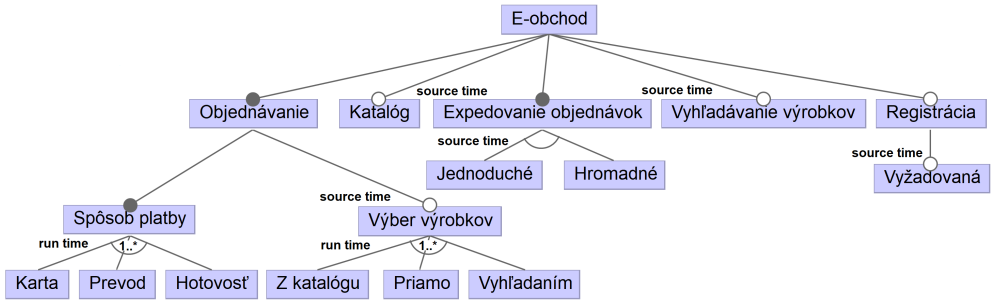
Mal by e-obchod podporovať registráciu? Mal by ju vyžadovať? Musí podporovať vyhľadávanie tovaru? Nepostačoval by katalóg? V prípade predaja iba niekoľkých výrobkov, tie by mohli byť len vymenované na hlavnej stránke staticky.

E-obchod by určite mal podporovať objednávanie. Mohli by sme si predstaviť e-obchod, ktorý by podporoval registráciu, ale nepodporoval by vyhľadávanie výrobkov, a ani by neposkytoval ich katalóg. Tiež by sme si mohli predstaviť e-obchod, ktorý by nepodporoval registráciu, a ani vyhľadávanie výrobkov, ale poskytoval by ich katalóg. Bude to stále e-obchod, ale sú to jeho *varianty*. Variantnosť je jednou zo zásadných vlastností softvéru. S daným klientom by sme mohli dohodnúť presný variant, ale čo ak vyvíjame pre viacerých klientov, a to aj viacerých potenciálnych klientov? V takom prípade musíme modelovať doménu aplikácie ako takú.

Časť 9.1 vysvetľuje základy techniky modelovania vlastností, ktorá umožňuje zachytiť variantnosť v doméne. Časť 9.2 ozrejmuje viazanie vlastností a konfigurovanie modelu vlastností. Časť 9.3 približuje prístup vývoja softvéru založený na radoch softvérových výrobkov. Časť 9.4 vysvetľuje parametrizáciu v UML ako spôsob podpory variability v UML.

9.1 Modelovanie vlastností

Objednávanie, katalóg, vyhľadávanie výrobkov, registrácia a pod. sú *vlastnosti* (angl. features) e-obchodu. Vlastnosti môže byť veľa, a aj súvislostí medzi nimi. Na ich zachytenie sa používa technika modelovania vlastností (angl. feature modeling). Súčasťou tejto techniky sú diagramy vlastností (angl. feature diagrams). Obrázok 9.1 zobrazuje diagram vlastností e-obchodu. Diagram obsahuje vlastnosti, ktoré sme identifikovali v predchádzajúcej časti a ešte niektoré ďalšie.



Obr. 9.1: Diagram vlastností e-obchodu.

Vlastnosti by mali byť výstižne pomenované, ale – tak ako prípady použitia – vyžadujú ďalší opis. Prípadne môžu byť rozpracované v ďalších modeloch. Aby neboli príliš dlhé, názvy vlastností často nadväzujú na názvy rodičovských vlastností. Ak názov vlastností nie je v danom diagrame vlastností jedinečný, možno ho kvalifikovať, ako napríklad *E-obchod.Objednávanie.Výber výrobkov.Vyhľadaním*.

Diagram vlastností je zvyčajne strom. Vrcholom je modelovaný koncept. Základná notácia modelovania vlastností [KCH+90] rozlišuje:

- povinné vlastnosti (angl. mandatory features), ktoré sú vyznačené vyplneným krúžkom (napríklad *Objednávanie*)
- voliteľné vlastnosti (angl. optional features), ktoré sú vyznačené vyplneným krúžkom (napríklad *Objednávanie*)
- alternatívne vlastnosti (angl. alternative features), ktoré sú vyznačené oblúkom (napríklad *Jednoduché* a *Hromadné*)

Vlastnosť môže byť zahrnutá v inštancii modelu vlastností, iba ak je zahrnutá jej rodičovská vlastnosť. Povinné vlastnosti pritom musia byť zahrnuté, voliteľné môžu byť zahrnuté, a z alternatívnych môže byť zahrnutá iba jedna.

Na oblúkoch pod vlastnosťami *Spôsob platby* a *Výber výrobkov* je uvedená kardinalita. Toto predstavuje rozšírenie základnej notácie modelovania vlastností [CHE05]. Pri oblúkoch s kardinalitou, počet vlastností, ktoré môžu byť zahrnuté, sa pohybuje v rozpätí danej kardinality. Kardinalita *1..** má sémantiku logického alebo (or), kým alternatívne vlastnosti majú sémantiku exkluzívneho alebo (xor). Známe sú aj ďalšie varianty modelovania vlastností [Vra04].

Vlastnosti, ktoré obsahuje každá inštancia modelu vlastností, tvoria spoločné (angl. commonality). Ostatné vlastnosti, ktoré môžu, ale nemusia byť zahrnuté v inštancii modelu vlastností, predstavujú variabilitu (angl. variability).

Ak majú zostať stromami, čo je vhodné z hľadiska prehľadnosti, v diagramoch vlastností nie je možné uviesť všetky súvislosti medzi vlastnoťami. Prídavné ohraničenia

(angl. addtional constraints) sa uvádzajú v textovej forme k diagramu vlastností. Napríklad, v našom modeli vlastností nemá význam umožňovať výber výrobkov ich vyhľadaním, t. j. nedá sa zahrnúť vlastnosť *Vyhľadaním*, ak nie je zahrnutá vlastnosť *Vyhľadávanie výrobkov*:

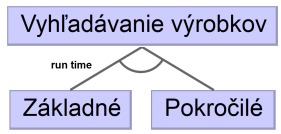
$$\textit{Vyhľadaním} \Rightarrow \textit{Vyhľadávanie výrobkov}$$

Rovnako,

$$\textit{Z katalógu} \Rightarrow \textit{Katalóg}$$

Význam týchto implikácií je taký, že vlastnosť na ľavej strane *vyžaduje* (angl. requires) vlastnosť na pravej strane. Tento výraz sa niekedy používa namiesto symbolu implikácie.

Aby diagramy vlastností neboli príliš rozsiahle, vlastnosti je možné rozpracovať v samostatných diagramoch vlastností. Príklad je na obrázku 9.2. Týmto začína byť zrejmé, že každý uzol v diagrame vlastností je vlastne koncept, a vlastnosť (byť vlastnosťou) je vlastne vzťah medzi dvomi konceptmi [Vra04].



Obr. 9.2: Diagram vlastností konceptu *Vyhľadávanie*.

9.2 Viazanie vlastností a konfigurovanie modelu vlastností

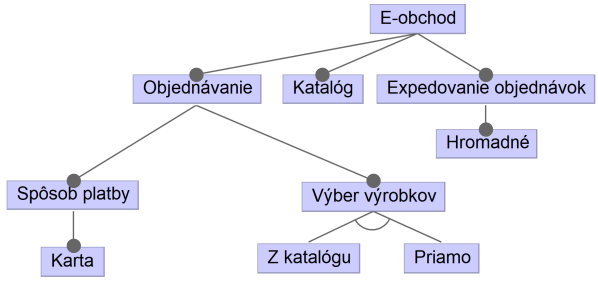
Zahrnutie alebo viazanie (angl. binding) vlastností má svoj obraz v kóde. Vlastnosť nemusí byť implementovaná na jednom mieste ako akýsi komponent, ktorý stačí pripojiť alebo odpojiť, ale dá sa určiť, ako jej zahrnutie vplýva na výsledný kód. Na prepojenie vlastností a kódu sa dajú použiť nástroje, akými sú napríklad pure::variants[1] alebo FeatureIDE[2].

Rozhodnutie o tom, že daná variabilná vlastnosť je zahrnutá, t. j. viazaná (angl. bound) môže prísť v rôznych časoch tvorby zdrojového kódu [Vp06]. V diagrame vlastností na obrázku 9.1 sú pri variabilných vlastnostiach vyznačené dva časy viazania (angl. binding time): *source time*, t. j. čas tvorby zdrojového kódu, a *run time*, t.

[1] `https://www.pure-systems.com/purevariants`
[2] `https://featureide.github.io/`

j. čas vykonávania. V závislosti od programovacieho jazyka, časy viazania môžu byť aj *compile time*, *link time*, *load time* atď. Ak nevieme, v ktorom programovacom jazyku systém bude tvorený, alebo ak sa nechceme viazať na jeden programovací jazyk, možno použiť zjednodušené rozlíšenie: staticky a dynamicky viazané vlastností.

Viazaním vlastností model vlastností konfigurujeme. Obrázok 9.3 zobrazuje jednu možnú konfiguráciu nášho e-obchodu. Vlatnosti *Z katalógu* a *Priamo* zostali variabilné, lebo sa viažu priamo počas vykonávania. To nemusí znamenať, že ich prítomnosť reguluje bežný používateľ. Množina variabilných vlastností pod vlastnosťou *Spôsob platby* viazaných v čase vykonávania bola zúžená na jednu vlastnosť.



Obr. 9.3: Inštancia e-obchodu.

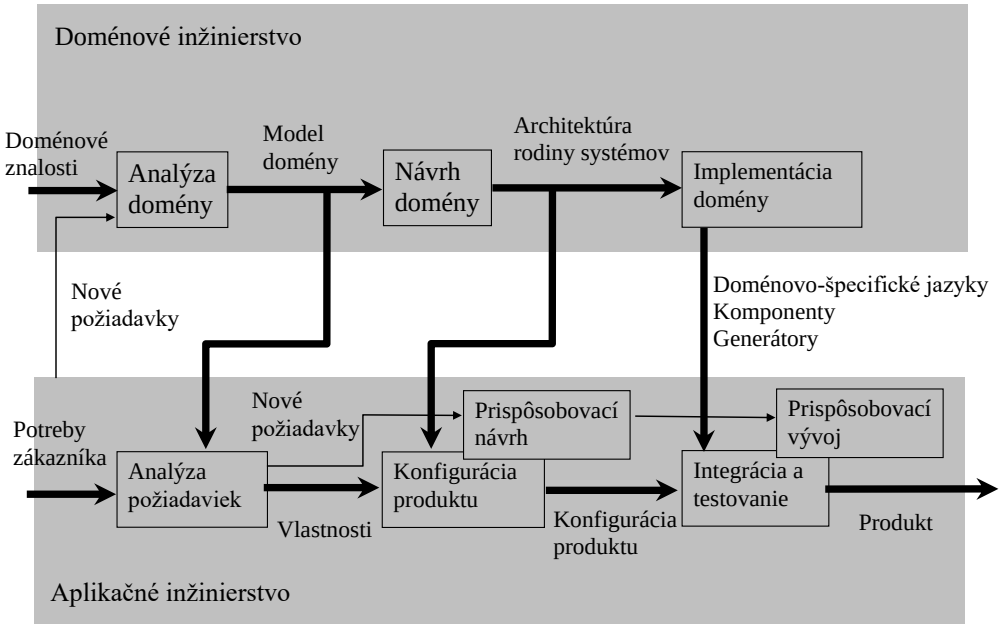
Modelovanie vlastností umožňuje abstraktné vyjadrenie variability. Neviažeme sa na implementačné mechanizmy. Tie vyberáme tak, aby zodpovedali variabilite a času viazania. Vyhľadávanie (obrázok 9.2) môžeme implementovať pomocou dedenia, a následne využitím polymorfizmu v čase vykonávania viazať základné alebo pokročilé vyhľadávanie.

9.3 Rady softvérových výrobkov

Hoci priemyselné aplikácie akademickej formy modelovania vlastností, nepočítajúc niektoré prípady uplatnenia notácie a nástroja pure::variants [BNR+14, BRN+13, BLR+15], nie sú známe [HCMH10], všeobecne sa uznáva, že spoločné a variabilita predstavujú jadro organizácie vývoja softvéru pre znovupoužitie v prístupe známom ako rady softvérových výrobkov (angl. software product lines) [VT16], ktorý priemyselné použitie má. To znamená, že spoločnosti vyvíjajúce softvér modely vlastností udržiavajú vo vlastných nástrojoch a notáciách.

Vývoj radu softvérových výrobkov, alebo proste radu výrobkov, ako sa tento prístup kratšie označuje, prebieha podľa schémy znázornenej na obrázku 9.4 (schéma prevzatá od Czarneckeho a Eiseneckera [CE00, Cza98]). Niekedy sa celý prístup vývoja softvéru založeného na radoch výrobkov označuje ako doménové inžinierstvo (angl. domain engineering). V užšom zmysle je to proces, v ktorom na základe zistenej va-

riability vznikajú artefakty, z ktorých možno tvoriť jednotlivé výrobky. Tie vznikajú v procese aplikačného inžinierstva (angl. application engineering), ktoré by za ideálnych okolností malo predstavovať iba konfigurovanie radu výrobkov, ale v skutočnosti takmer vždy vyžaduje prispôsobovací návrh a implementáciu. Tie sa zase propagujú do procesu doménového inžinierstva, aby ich rad výrobkov zohľadňoval pri konfigurovaní ďalších výrobkov.



Obr. 9.4: Procesy domenového a aplikačného inžinierstva vo vývoji radu softvérových výrobkov (schéma prevzatá od Czarneckeho a Eiseneckera [CE00, Cza98]).

Treba poznamenať, že rady softvérových výrobkov nie sú výrobné linky, ako sa niekedy mylne označujú, ale línie výrobkov, množiny príbuzných výrobkov, t. j. rodiny výrobkov (angl. product families). Známe sú rady (línie) aút, kozmetických výrobkov, oblečenia...

Zavedenie a prevádzka vývoja softvéru založeného na radoch výrobkov nie je jednoduchá ani lacná záležitosť. Organizácia musí zvoliť vhodnú stratégiu podľa svojich cieľov a možností. Tabuľka 9.1 (prevzatá od Boscha [Bos00]) uvádza možné stratégie zavedenia radov softvérových výrobkov.

V praxi tvorba radov výrobkov prebieha najčastejšie evolučne z jestvujúcej základne výrobkov. Organizácia najprv vytvorí jeden alebo viac podobných softvérových systémov, a až potom začne vyčleňovať a manažovať vlastnosti a ich implementáciu. Vývoj a dodávanie nových systémov nezastavuje, čím má zabezpečený príjem. Organizácia si často ani neuvedomuje, že vytvára a začína prevádzkovať rad softvérových výrobkov, čo je škoda, lebo jestvuje rad praktík pre to, aby tento prístup bol úspešný [NC+12].

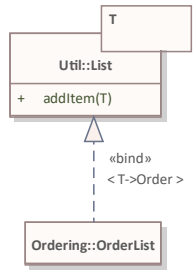
Tabuľka 9.1: Stratégie zavedenia radov softvérových výrobkov (prevzaté od Boscha [Bos00]).

	Revolučne	**Evolučne**
Bez základne výrobkov	Vývoj nového radu výrobkov pred dodaním prvého výrobku	Postupný vývoj radu výrobkov, počas ktorého sa dodávajú výrobky
Jestvujúca základňa výrobkov	Vývoj nového radu výrobkov z jestvujúcej množiny výrobkov pred dodaním prvého výrobku	Postupný vývoj radu výrobkov z jestvujúcej množiný výrobkov

9.4 Parametrizácia v UML

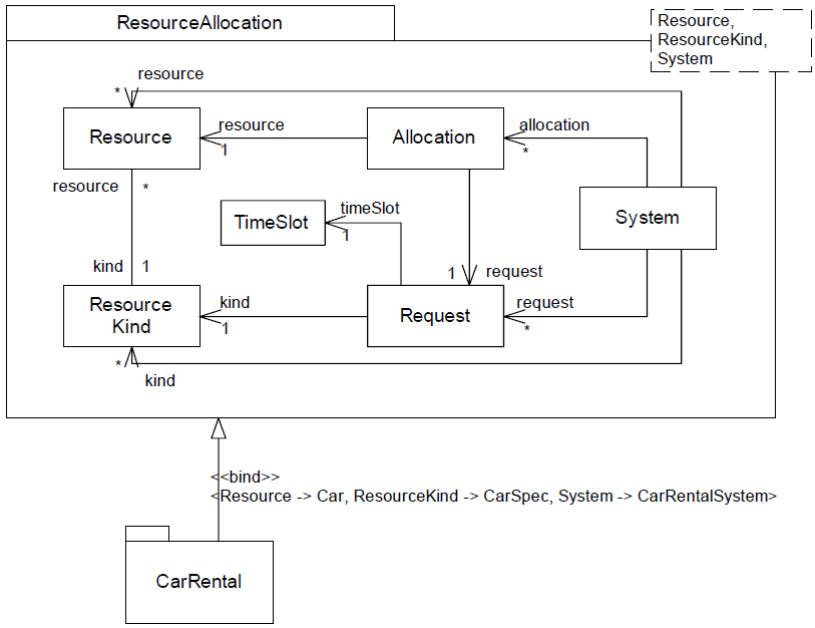
UML nepodporuje priamo modelovanie vlastností. Mohol by však byť rozšírený v tomto zmysle [Vn06]. Prípadne môžeme jestvujúcim prvkom, akými sú triedy a prípady použitia, prisúdiť význam vlastností, a tak získať komplexnú podporu modelovania vlastností bez potreby zásahu do UML [Gom04].

Variabilita je to, čo je premenlivé, a jeden zo spôsobov regulácie premenlivosti je parametrizácia. Obrázok 9.5, zopakovaný z časti 5.3, pripomína šablónu triedy a viazanie parametrov prostredníctvom vzťahu «bind».



Obr. 9.5: Viazanie parametrov šablóny triedy prostredníctvom vzťahu «bind» (zopakovaný obrázok 5.20).

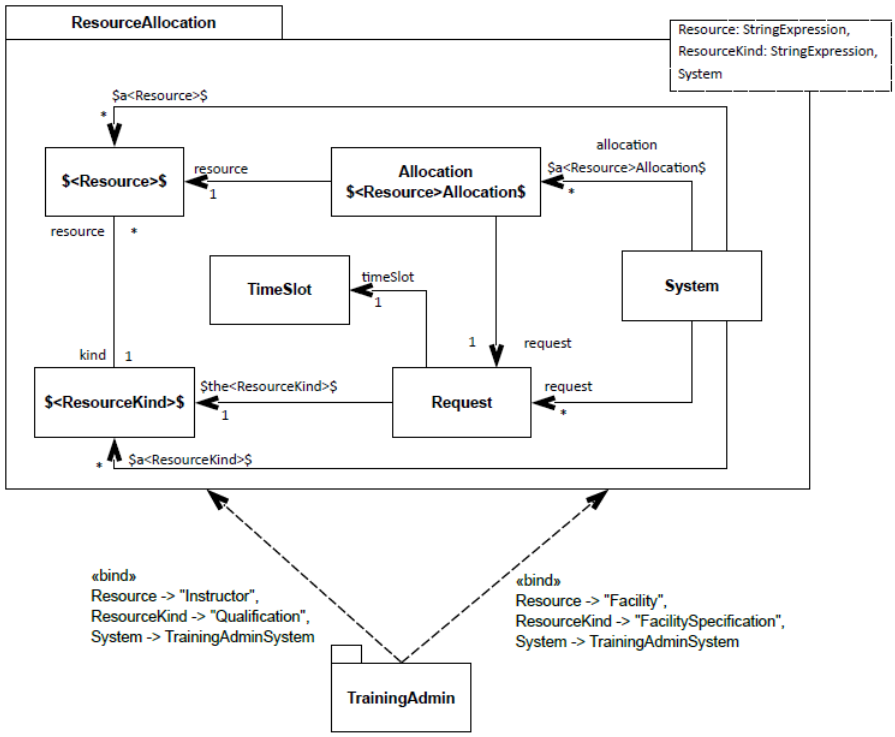
UML podporuje aj šablóny balíkov, kde je možné nahradiť zastupujúce triedy alebo rozhrania obsiahnuté v balíkoch konkrétnymi triedami alebo rozhraniami. Diagram na obrázku 9.5, prevzatý zo špecifikácie UML, verzia 2.4.1 [OMG11], ukazuje príklad takéhoto viazania. Na základe dodaných špecifických tried, vznikne viazanie generického balíka pre alokáciu zdrojov špecifické pre prenajímanie áut. V diagrame je menšia chyba: hraná «bind» by mala byť prerušovaná.



Obr. 9.6: Viazanie parametrov šablóny balíka (diagram prevzatý zo špecifikácie UML [OMG11]).

Diagram na obrázku 9.5, prevzatý z aktuálnej verzie špecifikácie UML [OMG17], ukazuje dvojité viazanie parametrov šablóny balíka, pričom vzniká systém na manažment trénovania, kde je generická alokácia zdrojov využitá aj pri inštruktoroch, aj pri zariadeniach. V diagrame je chyba: použité sú zlé ukončenia hrán a hrany «bind» nie sú prerušované, čo je zaznamenané v systéme na sledovanie chýb.[3]

[3] `https://issues.omg.org/issues/spec/UML/2.5#issue-41103`



Obr. 9.7: Dvojité viazanie parametrov šablóny balíka (diagram prevzatý zo špecifikácie UML [OMG17]).

Literatúra

[BHS07] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*, volume 4. Wiley, 2007.

[BLR+15] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. What is a feature? a qualitative study of features in industrial software product lines. In *Proceedings of 19th International Software Product Line Conference, SPLC '15*, Nashville, TN USA, 2015. ACM.

[BNR+14] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. Three cases of feature-based variability modeling in industry. In *Proceedings of ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems, MODELS 2014*, volume LNCS 8767, Valencia, Spain, 2014. Springer.

[Bos00] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[BRN+13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A survey of variability modeling in industrial practice. In *Proceedings of 7th International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS 2013*. ACM, 2013.

[Buh98] R. J. A. Buhr. Use case maps as architectural entities for complex systems. *IEEE Transactions on Software Engineering*, 24(12):1131–1155, 1998.

[BV16] Michal Bystrický and Valentino Vranić. Literal inter-language use case driven modularization. In *Proceedings of LaMOD'16: Language Modularity À La Mode, workshop, Modularity 2016*, Málaga, Spain, 2016. ACM.

[BV17] Michal Bystrický and Valentino Vranić. Preserving use case flows in source code: Approach, context, and challenges. *Computer Science and Information Systems Journal (ComSIS)*, 14(2):423–445, 2017.

[CB10] James Coplien and Gertrud Bjørnvig. *Lean Architecture for Agile Software Development*. Wiley, 2010.

[CE00] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[Coc00] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2000.

[Cop09] James Coplien. The DCI architecture: Supporting the agile agenda. Øredev Developer Conference, November 2009.

[Cza98] Krzysztof Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Technical University of Ilmenau, Germany, 1998. `https://www.researchgate.net/publication/220842465_Generative_Programming`.

[DL06] Adenekan Dedeke and Benjamin Lieberman. Qualifying use case diagram associations. *IEEE Computer*, 39(6):23–29, June 2006.

[Fow96] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1996.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gom04] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley, 2004.

[HCMH10] Arnaud Hubaux, Andreas Classen, Marcílio Mendonça, and Patrick Heymans. A preliminary review on the application of feature diagrams in practice. In *Proceedings of 4th International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS 2010*, ICB Research Report 37, 2010.

[Jac92] Ivar Jacobson. *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[Jac04] Ivar Jacobson. Use cases – yesterday, today, and tomorrow. *Software and Systems Modeling*, 3(3):210–220, 2004.

[JGJ97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture Process and Organization for Business Success*. Addison-Wesley, first edition, 1997.

[JN04] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, 2004.

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA): A feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, 1990.

[Lis87] Barbara Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, 1987.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[NC+12] Linda M. Northrop, Paul C. Clements, et al. A framework for software product line practice, version 5.0. Software Engineering Institute, Carnegie Mellon University, 2012. `https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=495357`.

[OMG11] OMG. OMG Unified Modeling Language (OMG UML), version 2.4.1. OMG, 2011. `https://www.omg.org/spec/UML/2.4.1/`.

[OMG14] OMG. Object Constraint Language. OMG, 2014. `https://www.omg.org/spec/OCL/`.

[OMG17] OMG. OMG Unified Modeling Language (OMG UML), superstructure, version 2.5.1. OMG, 2017. `https://www.omg.org/spec/UML/2.5.1/`.

[ÖP04] Gunnar Övergaard and Karen Palmkvist. *Use Cases: Patterns and Blueprints*. Addison-Wesley, 2004.

[Sel13] Bran Selić. Getting it right on the dot. OMG, 2013. `https://www.omg.org/ocup-2/documents/getting_it_right_on_the_dot.pdf`.

[TG15] Saurabh Tiwari and Atul Gupta. A systematic literature review of use case specifications research. *Information and Software Technology*, 67:128–158, 2015.

[Vn06] Valentino Vranić and Ján Šnirc. Integrating feature modeling into uml. In *Proceedings of Net.ObjectDays 2006, NODe 2006*, LNI P-88, Erfurt, Germany, 2006. GI.

[Vp06] Valentino Vranić and Miloslav Šípka. Binding time based concept instantiation in feature modeling. In *Proceedings of 9th International Conference on Software Reuse, ICSR 2006*, LNCS 4039, Turin, Italy, 2006. Springer.

[Vra04] Valentino Vranić. Reconciling feature modeling: A feature modeling metamodel. In *Proceedings of 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World, Net.ObjectDays 2004*, LNCS 3263, Erfurt, Germany, 2004. Springer.

[VT16] Valentino Vranić and Roman Táborský. Features as transformations: A generative approach to software development. *Computer Science and Information Systems Journal (ComSIS)*, 13(3):759–778, 2016.

[VuZ13] Valentino Vranić and Ľuboš Zelinka. A configurable use case modeling metamodel with superimposed variants. *Innovations in Systems and Software Engineering: A NASA Journal*, 9(3), 2013.

Register

Autor:	doc. Ing. Valentino Vranić, PhD.
Názov:	Modelovanie softvéru: prípady použitia, UML a ďalej
Vydanie:	prvé
Náklad:	elektronické vydanie
Rozsah:	117 strán, 72 obrázkov
Edičné číslo:	
Vydala:	Slovenská technická univerzita v Bratislave vo Vydavateľstve SPEKTRUM STU
Rok:	2022

ISBN 978-80-227-5248-0