IMPLEMENTACE USE CASE POMOCÍ NÁVRHOVÉHO VZORU CONTROLLER

Miloš Kudělka, Vladimír Sklenář

KMI PřF UP, Tomkova 40, 779 00 Olomouc, ČR, milos.kudelka@upol.cz, vladimir.sklenar@upol.cz

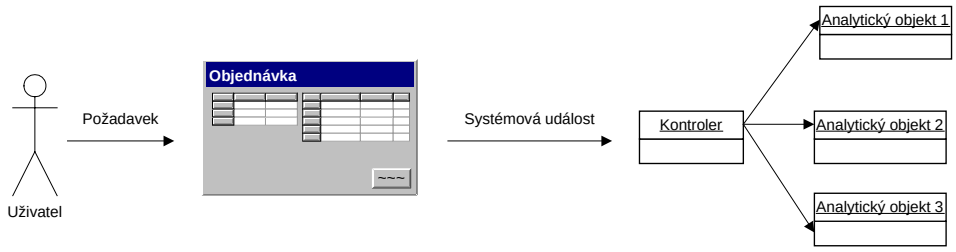
Abstrakt

V prostředí aplikace s prezentační vrstvou jsou USE CASE vyvolávány a řízeny uživatelem prostřednictvím prezentačních objektů (textová pole, výběry, nabídky, tlačítka). V článku je popsán návrh implementace USE CASE založený na kombinaci vzoru *Controller* se vzorem *Bridge*, který umožňuje práci s abstraktní prezentační vrstvou. Takto implementovaný USE CASE je snadno znovupoužitelný v různých prezentačních prostředích (Windows formuláře, ASP stránky, apod.). Další výhodou je snadná modifikovatelnost existujících a implementace nových USE CASE formou skládání existujících.

Kontrolér a jeho funkce

Funkční požadavky na softwarový produkt lze zaznamenat formou USE CASE. V nich je textovou formou popsán způsob komunikace mezi uživatelem a systémem. Realizace USE CASE popisuje, jak je daný USE CASE realizován v konkrétním návrhu skupinou spolupracujících objektů. V prostředí aplikace s prezentační vrstvou jsou USE CASE vyvolávány a řízeny uživatelem prostřednictvím prezentačních objektů (textová pole, výběry, nabídky, tlačítka). S jejich pomocí uživatel formuluje, *CO* se musí udělat včetně zadání vstupních dat. To *JAK* se má činnost udělat, tedy vlastní logiku USE CASE, realizují třídy z analytické vrstvy (tj. třídy reprezentující pojmy z problémové domény). Jejich instance mají většinou pasivní povahu. To znamená, že zajistí vykonání požadavku, ale samy o sobě žádnou aktivitu nevykonávají a nijak se nezabývají svým okolím, tj. dalšími objekty, které se podílejí na realizaci USE CASE. Jedním z úkolů při návrhu realizace USE CASE tedy je stanovit, kdo je zodpovědný za průběh USE CASE jako celku. To znamená přidělení zodpovědnosti za korektní zpracování vstupních systémových událostí. Pojmem vstupní systémová událost budeme označovat logickou událost generovanou externím aktérem (např. vznesení požadavku na zaznamenání nové objednávky). K jednotlivým systémovým událostem jsou asociovány systémové operace, to je operace, které reagují na systémové události a zajišťují jejich korektní ošetření. Aplikace tedy přijímá externí vstupní systémové události, které typicky generuje uživatel prostřednictvím GUI. Je-li použit objektový návrh, pak musí být některému objektu přiřazena odpovědnost za jejich zpracování. Je vhodné, aby systémové operace byly zpracovávány v aplikační vrstvě, ne v prezentační vrstvě. Možným řešením je vytvoření speciální třídy označované jako kontrolér (*Controller*). Tato třída je součástí aplikační vrstvy a poskytuje rozhraní pro zpracování všech systémových událostí v daném USE CASE. Kontrolér by měl přenést činnosti, které je nutné provést na instance analytických tříd a sám pouze řídit a koordinovat jejich aktivity. Je žádoucí, aby jeden kontrolér obsluhoval všechny systémové události spojené s jedním USE CASE. V tomto případě je totiž možné udržovat informace o stavu realizace v kontroléru a zajistit např. kontrolu správné návaznosti jednotlivých systémových událostí. V jednoduchém scénáři objekty prezentační vrstvy zachytí GUI operace, transformují je na systémovou událost a

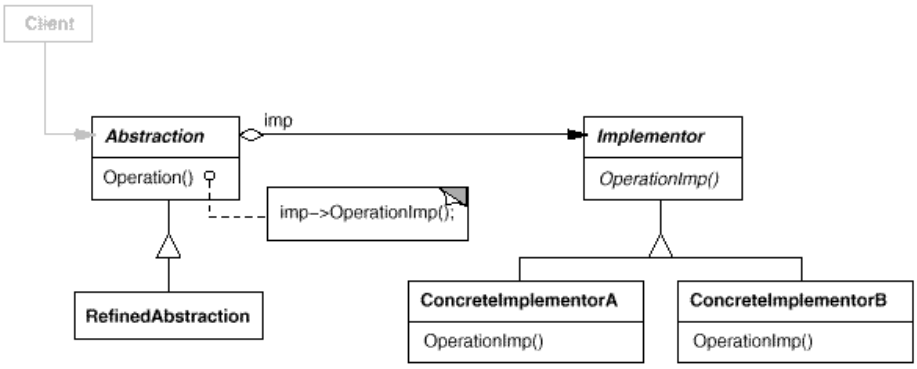
vyvolají odpovídající systémovou operaci příslušného kontroléru. Ten pak koordinuje plnění požadavků a deleguje jednotlivé logické operace na objekty aplikační vrstvy.



Problém znovupoužitelnosti

Přiřazení odpovědnosti za systémové operace objektům aplikační vrstvy zvyšuje znovupoužitelnost a modifikovatelnost realizace USE CASE. Jsou-li naopak systémové operace realizovány v prezentační vrstvě, je logika aplikace obsažena v kódu tříd prezentační vrstvy (např. formuláře nebo dialogu) a je tedy spojena s konkrétním způsobem prezentace, což výrazně snižuje možnost znovupoužití v jiném prezentačním prostředí. Umístění zodpovědnosti za realizaci systémových operací v kontroléru tedy znovupoužití aplikační logiky výrazně usnadňuje. Je také snazší změnit způsob prezentace, popřípadě použít jiné technologie pro implementaci prezentační vrstvy.

V předchozích scénářích jsme předpokládali, že realizace USE CASE nezahrnuje žádné změny zobrazovaných informací v závislosti na svém stavu. Například, že se zpřístupní, popřípadě znepřístupní některé prvky GUI, nebo že se aktualizuje obsah některých prvků v závislosti na stavu prvků jiných. Jednoduchým příkladem může být například požadavek, aby se na formuláři po zadání ID výrobku zobrazil jeho název a cena. Takovéto požadavky mohou být chápány jako součást zadání USE CASE. Jedním z použitelných (a také často používaných) řešení je umístit příslušný kód do metod ošetřujících události v GUI. Toto řešení ale výrazně snižuje možnost znovupoužití realizace USE CASE jako celku (včetně implementace požadavků na způsob prezentace jeho průběhu). Chceme-li totiž implementovat prezentační vrstvu v jiném prostředí, je nutné její kód znovu vytvořit, a to včetně implementace požadovaného chování. Nezanedbatelné není ani to, že činnosti požadované v USE CASE jsou implementovány na více místech (v kódu kontroléru a kódu prezentační vrstvy). Snadno znovupoužitelný je pouze kód umístěný v kontroléru. Pokud chceme zajistit znovupoužitelnost USE CASE jako celku, je zřejmě nutné, aby veškerý (nebo alespoň naprostá většina) jeho kódu byla umístěna na jediném místě, a to ve třídě reprezentující USE CASE kontrolér. V podstatě tedy jde o to, aby průběh (hlavní i alternativní) USE CASE popsaný v jeho textovém zápisu, byl implementován v jediné třídě, která by byla umístěna v aplikační vrstvě. Tento popis ale obecně zahrnuje jak logiku aplikace (aplikační vrstva), tak chování prezentační vrstvy. Je jasné, že není možné, aby se kontrolér obracel přímo na prezentační vrstvu, protože pak by se stal kontextově závislý. Chceme-li dosáhnout toho, aby kód kontroléru byl beze změny použitelný pro různé platformy prezentační vrstvy, musíme zajistit, aby pracoval s abstraktním uživatelským rozhraním. Pro tyto situace je vhodné použít návrhový vzor *Bridge*.



V našem případě jeho použití znamená vytvoření abstraktního rozhraní, které bude poskytovat obecnou funkčnost prezentační vrstvy a oddělenou implementaci tohoto abstraktního rozhraní pro různé platformy. Instance třídy, která reprezentuje abstraktní rozhraní, obsahuje odkaz na instanci třídy reprezentující implementaci a jeho prostřednictvím předává této implementační třídě všechny výkonné operace k realizaci. Kód kontroléru pak implementuje logiku chování prezentační vrstvy voláním metod obsažených v abstraktním rozhraní. Takto realizovaný kontrolér pak je jednotný pro všechny typy platforem prezentační vrstvy a je tedy znovupoužitelný.

Vlastnosti abstraktní prezentační vrstvy

Rozhraní abstraktní vrstvy musí obsahovat metody, které umožní, aby kontrolér mohl určovat typ a umístění jednotlivých prvků GUI a stanovovat datové hodnoty spojené s jednotlivými prvky (např. Položky umístěné v Listboxu, text zobrazovaný v Editboxu, implicitní volba v check boxech …). Pro jednoduchost se omezíme pouze na několik málo prvků grafického rozhraní. Jedná se o tlačítko (Buton), textové pole (EditBox), seznam textových řetězců (ListBox) a prvek umožňující volbu (CheckBox). Tyto prvky je možné seskupovat do větších celků, které budeme označovat jako kontejnery. Všechny prvky mají některé společné vlastnosti. Patří sem:

- Jejich umístění a rozměry.
- Aktivita a viditelnost (zda je prvek přístupný uživateli).
- Schopnost přijímat události. Mezi základní typy událostí patří změna fokusu, kliknutí, dvojité kliknutí a změna obsahu(hodnoty).

Kromě těchto společných vlastností mají jednotlivé prvky své specifické vlastnosti. Ty jsou především spojeny s jejich informačním obsahem. Proto je pro jednotlivé běžně používané prvky GUI v této abstraktní vrstvě vytvořen jejich abstraktní reprezentant. Kontrolér při své inicializaci požádá abstraktní vrstvu o prázdný základní formulář. Do tohoto formuláře umístí jednotlivé grafické prvky a poskytne jim odkaz na data, která mají zobrazovat.

Abychom mohli zajistit odpovídající reakci kontroléru na akce prováděné uživatelem, je nutné zajistit, aby abstraktní vrstva byla schopna přijímat události generované uživatelem prostřednictvím GUI a předávat je kontroléru. Tyto události jsou samozřejmě platformově závislé. Je proto nutné je převést na abstraktní událost prezentační vrstvy. V podstatě lze říci, že úkolem abstraktní vrstvy je pouze převedení konkrétní události z konkrétního prostředí do jednotného tvaru a její předání kontroléru. Abstraktní vrstva se tedy nezabývá logickým vyhodnocením dané události, převod na systémovou událost a její zpracování zajistí až kontrolér.

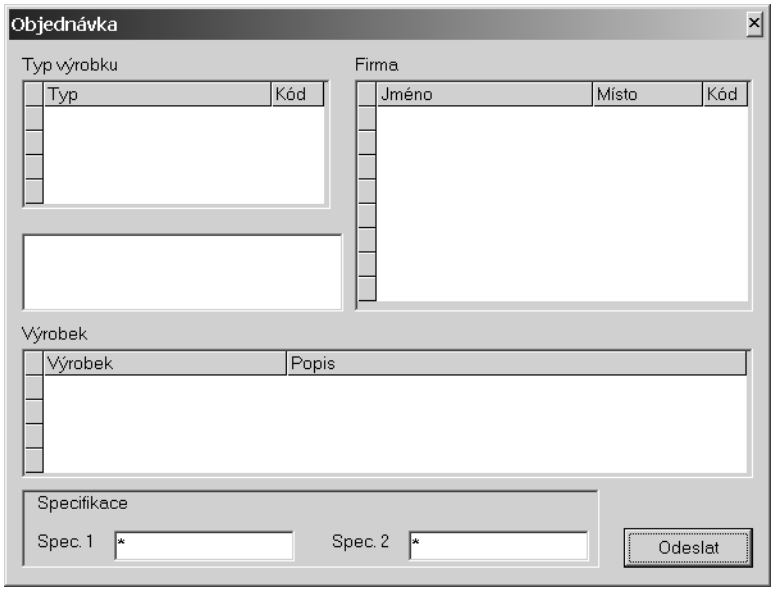
Vytváření USE CASE skládáním

V duchu uvedených myšlenek se jako nejdůležitější princip jeví znovupoužití již vytvořených jednoduchých USE CASE při vytváření složitějších, což v důsledku vede k vyřešení problému kompozice existujících USE CASE do nových (v budoucnu stejným způsobem použitelných) USE CASE a k vyřešení způsobu jejich koordinace. Uveďme příklad, při jehož řešení ukážeme, jak postupujeme v případě, když některé USE CASE byly již dříve implementovány.

Příklad

Aplikace poskytuje rozhraní, ve kterém uživatel objednává konkrétní výrobek, přičemž postupuje tak, že vybírá typ výrobku a dodavatelskou firmu ze seznamů, poté je mu nabídnut seznam výrobků daného typu, po výběru z tohoto seznamu je uživateli zobrazen formulář pro dodatečné upřesnění výrobku. Pokud jsou vyplněna povinná pole, systém nabídne tlačítko pro odeslání objednávky.

Vyjádření v grafické podobě



Zápis formou dialogu uživatele se systémem

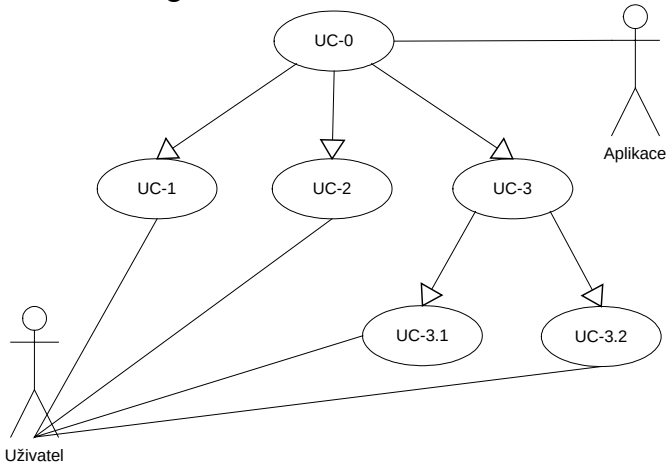
Uživatel vyvolá z nabídky formulář pro vytvoření objednávky.	Systém poskytne formulář se dvěma seznamy vedle sebe. V levém seznamu budou zobrazeny typy výrobků ve dvou sloupcích (název a kódové označení). V pravém seznamu bude seznam firem ve třech sloupcích (název, místo a kódové označení). Oba seznamy budou obsahovat prázdný řádek, na kterém budou nastaveny.
Uživatel vybere ze seznamu typů výrobků jeden řádek s konkrétním typem.	Pod seznamem s typy výrobků se zobrazí textové pole, v němž se zobrazí popis typu. Seznam firem se vyplní pouze záznamy o firmách poskytujících vybraný typ výrobku. V seznamu firem zůstane vybraný stejný řádek, který byl vybrán před akcí.
Uživatel vybere v seznamu typů výrobků prázdný řádek.	Skryje se textové pole pod seznamem s typy výrobků. Seznam firem se vyplní záznamy o všech firmách. V seznamu firem zůstane vybraný stejný řádek, který byl vybrán před akcí.

Uživatel vybere ze seznamu firem jednu konkrétní firmu.	Seznam typů výrobků se vyplní pouze záznamy o typech, které firma poskytuje. V seznamu typů výrobků zůstane vybraný stejný řádek, který byl vybrán před akcí.
Uživatel vybere v seznamu firem prázdný řádek.	Seznam typů výrobků se vyplní záznamy o všech typech. V seznamu typů výrobků zůstane vybraný stejný řádek, který byl vybrán před akcí.
Uživatel má vybrán typ výrobku i firmu.	Systém zobrazí pod seznamy s typy výrobků a firmami seznam konkrétních výrobků vybraného typu ve dvou sloupcích (název a stručná charakteristika). Seznam bude obsahovat prázdný řádek, na kterém bude nastaven.
Uživatel vybere ze seznamu výrobků jeden řádek s konkrétním výrobkem.	V dolní části formuláře pod seznamem s konkrétními výrobky se zobrazí popis výrobku a pole a seznamy pro upřesnění objednávky. Povinné položky jsou označeny. Tato část formuláře může být pro každý výrobek jiná. Dole na formuláři bude zablokované tlačítko pro odeslání objednávky.
Uživatel vybere ze seznamu výrobků prázdný řádek.	Skryje se dolní část formuláře s popisem výrobku.
Uživatel vyplní povinné položky.	Systém zpřístupní tlačítko pro odeslání objednávky.
Uživatel stiskne tlačítko pro odeslání objednávky.	Systém zajistí odeslání objednávky a ukončí formulář.

Při bližším pohledu na slovní popis je zřejmé, že se v této části aplikace vyskytuje celkem šest USE CASE.

1. UC-0: Zajišťuje celkový průběh zadání až po odeslání objednávky.
2. UC-1: Zajišťuje nabídku a výběr typu výrobků ze seznamu.
3. UC-2: Zajišťuje nabídku a výběr firmy.
4. UC-3: Zajišťuje výběr a upřesnění konkrétního výrobku.
5. UC-3.1: Zajišťuje nabídku a výběr konkrétního výrobku.
6. UC-3.2: Zajišťuje výpis popisu výrobku a kontrolu vyplnění povinných položek ke konkrétnímu výrobku. Tento USE CASE může existovat v různých variantách (specializacích).

Vyjádření formou USE CASE diagramu



Předpokládejme že UC-1, UC-2, UC-3_1 už byly dříve z nějakého důvodu izolovaně implementovány. Pro plnou funkčnost uvedené úlohy je potřeba

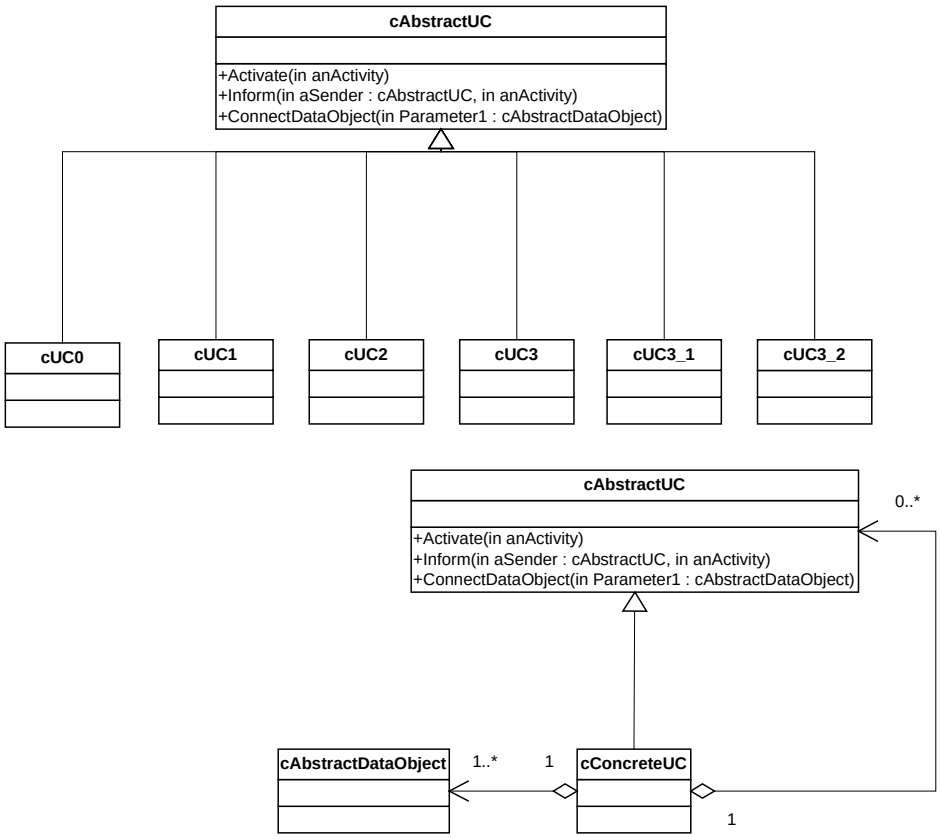
- implementovat UC-0, UC-3 a UC-3.2,
- zapojit nové a existující USE CASE do nové logiky bez zásahu do jejich implementace.

Oba úkoly v důsledku znamenají (kromě vlastní implementace chybějícího kódu) formulovat obecné rozhraní kontrolérů tak, aby jejich propojení bylo co nejjednodušší a nejtenčí. Celý problém je celkem jednoduchý, pokud si uvědomíme, že v systému kontrolérů, které implementují USE CASE, dochází k určitým aktivitám, které vedou k vyvolání událostí, při jejichž zpracování se pouze musí zajistit informace datových objektů kontrolérů, kterých se tato událost týká. Vše ostatní musí zůstat důsledně skryto.

- Každý kontrolér pracuje se skupinou datových objektů, které jsou vyjádřením analytické logiky. Kontroléry pak mohou pracovat nad různými částmi těchto objektů, musí si je umět předat pro použití v jednotlivých USE CASE. Pro tuto část pohledu na funkčnost musí mít každý kontrolér implementovány metody pro připojení datových objektů. Předpokládáme tedy, že v hierarchii kontrolérů existuje způsob, jak připojit datové objekty do místa (USE CASE), které aktuálně s objektem pracuje. Každý kontrolér tedy pro svou plnou funkčnost potřebuje zvenčí připojit analytickou část aplikace.
- Pokud předpokládáme, že kontrolér realizuje více souvisejících činností se společnou logikou (zobrazení, editaci, apod.), lze je chápat jako určitý typ aktivity kontroléru. Tato aktivita může být vyvolána jednak nadřízeným kontrolérem, jednak přímo uživatelem přímo z prezentační části kontroléru. Každá aktivita je pak v systému jednoznačně identifikovatelná tím, KDO ji provádí a o JAKOU aktivitu se jedná. Každá aktivita je tedy identifikovatelná těmito dvěma atributy.Musí se dodržet jednoduché schéma, ve kterém má nadřízený kontrolér ve své režii podřízené kontroléry, a to prostřednictvím zprávy *Activate* podřízenému kontroléru (parametrem zprávy může být např. řetězech vyjadřující jméno aktivity). Podřízený kontrolér zareaguje buď akceptováním a svou další činností (jednou z nich je nutné informování okolí) nebo požadavek z nějakého definovaného důvodu ignoruje, pak nenásleduje žádná další činnost. Pokud se změní aktivita kontroléru bez zásahu nadřízeného kontroléru, musí mít možnost upozornit na svou aktivitu nadřízený kontrolér. Udělá to prostřednictvím události *Inform*, k jejímuž odebírání je přihlášen nadřízený kontrolér. Zpráva obsahuje odesílatele (*aSender*) a popis aktivity (*anActivity*). Nadřízený kontrolér tak rozpozná, kdo a jakou zprávu posílá, podle toho může reagovat dál (jak směrem k nadřízeným, tak k podřízeným kontrolérům).

Diagramy tříd

Vyjádříme-li uvedené popisy diagramy tříd, je zřejmé, že pro kontroléry můžeme použít dědičnost a definovat společného předka, který popisuje povinnou implementaci rozhraní.



Řešení příkladu

Jsou-li USE CASE implementovány výše uvedeným způsobem jako kontroléry, lze je využít a pouze zajistit vzájemnou informovanost o změnách stavu a předání dat, se kterými se aktuálně pracuje. Pro zjednodušení označme instance odpovídajících kontrolérů obdobně jako USE CASE.

1. Aplikace vytvoří UC0 a dodá mu objekty potřebné pro zadání objednávky. Aplikace aktivuje UC0.
2. UC0 vytvoří UC1, UC2 a UC3. UC0 připojí k UC1 (UC2) kolekci typů výrobků (firem) a oba aktivuje pro dialog s uživatelem.
3. UC1 (UC2) vyvolají událost zvolen resp. nezvolen typ výrobku (firma). UC0 podle toho zašle zprávu kolekcím (ty zareagují odpovídajícím způsobem) a reaktivuje UC2 (UC1).
4. Pokud je zvolena kombinace typ výrobku a firma, UC0 připojí k UC3 kolekci výrobků odpovídající výběru. UC0 aktivuje UC3.
5. UC3 vytvoří UC3_1 a dodá mu kolekci výrobků. UC3 aktivuje UC3_1 pro dialog s uživatelem.
6. Pokud není zvolena kombinace typ výrobku a firma, UC0 deaktivuje UC3.
7. Pokud je zvolen výrobek, UC3 vytvoří UC3_2 podle identifikace výrobku a připojí k němu výrobek. UC3 aktivuje UC3_2.
8. Pokud není zvolen výrobek, UC3 odstraní UC3_2.
9. Pokud jsou vyplněné povinné položky v rámci UC3_2, je zpřístupněno tlačítko odeslat.
10. Pokud nejsou vyplněné povinné položky v rámci UC3_2, je zablokováno tlačítko odeslat.
11. Pokud je stisknuto tlačítko odeslat, UC3_2 pošle výrobek modulu pro objednávky.
12. Pokud odeslání proběhne v pořádku, je UC3_2 vyvolána událost, která je v důsledku zpracována aplikací, která ukončí UC0.

Závěr

Důsledná aplikace uvedeného řešení umožňuje velmi efektivně zvládnout měnící se požadavky uživatele na existující řešení i na použití existujících řešení v jiném kontextu. Příkladem může být úloha, ve které byla aplikace skládající se z postupného nabízení jednotlivých formulářů (určených k vyřešení podúloh) převedena do prostředí, kde se vše řeší v jednom formuláři se vzájemnou koordinací jednotlivých složek. Na základě několikaměsíčních zkušeností s využíváním pracovního rámce založeného na tomto modelu můžeme konstatovat výrazné zvýšení efektivity práce programátora a snížení nároků na zařazení nového programátora do existujícího projektu. Je to tím, že se programátor může více věnovat samotné logice úlohy a méně technickým podrobnostem implementačního prostředí.

Literatura:

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994. ISBN 0-201-63361-2
2. Larman, C. Applying UML and Patterns. Second Edition. Prentice Hall, 2002. ISBN 0-13-092569-1
3. Sundblad, S., Sundblad, P. Designing for Scalability with Microsoft Windows DNA. Microsoft Press, 2000, ISBN 0-7356-0968-3
4. Shalloway, A., Trott, J. Design Patterns Explained, Addison-Wesley, 2002, ISBN 0-201-71594-5
5. Sklenář, V. Snášel, V. Aplikování návrhových vzorů. In sborník z konference OBJEKTY '1999, Praha, ČZU, str.87
6. Kudělka, M, Sklenář, V. Výuka, návrhové vzory a tvorba komponent. In sborník z konference OBJEKTY '2000, Praha, ČZU, str.197
7. Booch, G., Rumbaugh, J., Jacobson, I. The Unified Modeling Language, User Guide. Addison-Wesley, 1999. ISBN 0-201-57168-4
8. Jacobson, I., Rumbaugh, J., Booch, G. The Unified Software Development Process. Addison-Wesley, 1999. ISBN 0-201-57169-2