Žilinská univerzita v Žiline
Fakulta riadenia a informatiky

Diplomová práca

Študijný program: Hospodárska informatika

Pavol Odlevák

Návrh a tvorba podsystému „Undo/Redo" do nástroja UML .FRI

Vedúci: Ing. Ján Janech

Reg. č.: 236/2008 jún 2009

Žilina

Anotačný list

Anotačný list diplomovej práce študenta v štúdijnom programe Hospodárska informatika na Žilinskej univerzite v Žiline v akademickom roku 2008/09.

1.	Meno a priezvisko	: *Pavol Odlevák*
2.	Fakulta	: *Fakulta Riadenia a Informatiky*
3.	Názov práce	: *Návrh a tvorba podsystému „Undo-/Redo" do nástroja UML .FRI*
4.	Počet strán	: *68*
5.	Počet obrázkov	: *22*
6.	Počet príloh	: *1*
7.	Počet titulov použitej literatúry	: *27*
8.	Kľúčové slová	: *UML .FRI, Undo/Redo subsystém, Python, Modely undo systémov*

Resumé

V diplomovej práci sú rozobraté teoretické základy spolu s modelmi a spôsobmi implementácie undo/redo systémov. Následne je navrhnutý najvhodnejší model a je popísaný postup implementácie tohto modelu do CASE nástroja UML .FRI.

Summary

In this diploma thesis is discussed the theory, models and implementation techniques of undo/redo systems. Subsequently an optimal model is chosen and its implementation into the CASE tool UML .FRI is described.

Poďakovanie

Týmto by som sa rád poďakoval vedúcemu diplomovej práce Ing. Jánovi Janechovi za jeho trpezlivosť a množstvo času, ktoré venoval pri vysvetľovaní princípov a spôsobu fungovania UML .FRI aplikácie, ako aj za jeho užitočné rady a cenné pripomienky pri písaní diplomovej práce.

Prehlásenie

Prehlasujem, že som diplomovú prácu spracoval samostatne a že som uviedol všetky použité pramene a literatúru, z ktorých som čerpal.

V Žiline dňa 15. mája 2009 Podpis.....

Obsah

Zoznam obrázkov

Zoznam výpisov

Zoznam použitých označení

ACS	*Archer, Conway and Schneider* – model undo systému pomenovaný podľa jeho tvorcov
API	*Application Programming Interface* – rozhranie pre programovanie aplikácií
ATK	*Accessibility Toolkit* – toolkit umožňujúci aplikácií používať nástroje ako alternatívne vstupné zariadenia.
BSD	*Berkeley Software Distribution* – operačný systém, derivát pôvodného systému Unix. V súčastnosti sa pojem používa na súhrnné označenie vetvy, resp. rodiny nástupcov tohto systému, ako napríklad FreeBSD, NetBSD alebo OpenBSD[WIK09]
CAD	*Computer-aided design* – použitie počítačovej technológie na návrh objektov, reálnych alebo virtuálnych
CASE	*Computer Aided Software Engineering* – využitie počítača pri softvérovom inžinierstve, teda za účelom návrhu, tvorby a údržby počítačových programov
Emacs	*Editor MACroS* – pokročilý textový editor s množstvom príkazov a schopnosťou spájania príkazov do makier
ENIAC	*Electronic Numerical Integrator And Computer* – historicky prvý Turing-kompletný elektrónkový počítač [WIK09]
FOSS	*Free and open source software* – Slobodný softvér a softvér s otvoreným kódom
FSF	*Free Software Foundation* – nadácia pre slobodný softvér, bola založená s cieľom podporovať práva používateľov počítačov: používať, študovať, kopírovať, modifikovať a redistribuovať počítačové programy[WIK09]
GIMP	*GNU Image Manipulation Program* – slobodná multiplatformová aplikácia slúžiaca na tvorbu a úpravu rastrovej grafiky
Glade	*Glade Interface Designer* – nástroj na tvorbu grafických používateľských rozhraní pre GTK+

GNU	*GNU's Not Unix* – slobodný operačný systém
GPL	*GNU General Public License* – licencia pre slobodný softvér
GTK+	*The GIMP Toolkit* – slobodný multiplatformový toolkit slúžiaci na tvorbu grafických požívateľských rozhraní, pôvodne vznikol pre potreby aplikácie GIMP
HCI	*Human-Computer Interaction* – štúdium interakcií medzi ľuďmi a počítačmi
Qt	*„Cute"* – multiplatformový framework pre tvorbu aplikácií, používaný predovšetkým na tvorbu GUI programov, ale taktiež na konzolové nástroje a serverové aplikácie [WIK09]
RAD nástroj	*Rapid application development tool* – nástroj na rýchly návrh, alebo vývoj aplikácie
UML	*Unified Modeling Language* – vizuálny jazyk na modelovanie a komunikáciu o systéme pomocou diagramov a podporného textu [RUZ08]
US&R	*Undo, Skip and Rotate* – rozšírený model undo systému
svn	*Subversion* – slobodný systém pre správu softvéru a jeho verzií
Vi	*rodina textových editorov* – zdieľajúca spoločné charakteristiky, ako napríklad spôsob ovládania a používateľský interface.
widget	*„window gadget"* – element grafického používateľského rozhrania
XML	*eXtensible Markup Language* – rozšíriteľný značkovací jazyk, umožňujúci jednoduché vytváranie konkrétnych značkovacích jazykov

Kapitola 1

Úvod

„To see and to be seen, in heaps they run; Some to undo, and some to be undone."

(John Dryden)

V súčasnosti je undo[1] nedielnou súčasťou takmer všetkých interaktívnych aplikácii, avšak ako uvádza [BRO03] nejde o nový koncept. Primitívne prostriedky na undo existovali už v roku 1948 v podobe mechanizmu kontrolných bodov a obnovy v počítačoch *ENIAC*[2]. Postupne sa undo modely a jeho výskum rozložili do rôznych oblastí počítačovej vedy, najmarkantnejšia časť používateľsky orientovaného undo systému je však z oblasti *HCI*[3], ktorej cieľom boli interaktívne aplikácie ako textové a grafické editory, *CAD*[4] systémy, tabuľkové a textové procesory. Avšak undo má svoju históriu aj v iných oblastiach, ako sú databázy, programovacie jazyky alebo debuggery - týmito oblasťami sa však ďalej nebudeme zaoberať, keďže v nich ide skôr o vnútorný obnovovací nástroj, než o prostriedok opravy ľudskej chyby. Ľudské chyby v interakcii s rôznymi systémami sú však pomerne bežné, napríklad [OGP03] vo svojej štúdii ohľadom spoľahlivosti Internetových služieb poukazuje na fakt, že operátor, resp. ľudský faktor je príčinou najväčšieho počtu výpadkov služieb, ako zobrazuje obrázok 1.1 a zmiernenie takto spôsobených výpadkov trvá jednoznačne najdlhšiu dobu, viac ako všetkých ostatných prí-

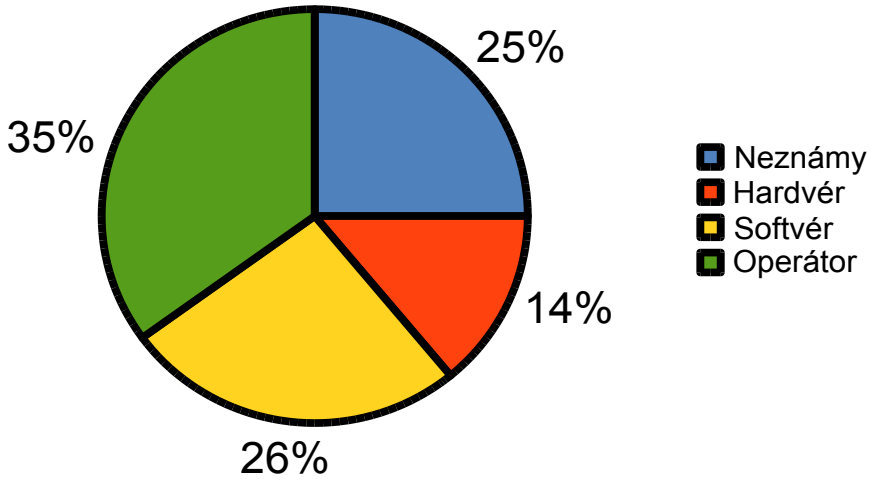
[1] V texte sa uvádzajú nepreložené anglické označenie pre krok späť – undo, resp. krok vpred – redo, predovšetkým z dôvodu ich väčšej výstižnosti, ako aj pomernej zaužívanosti výrazov aj v slovenskom jazyku.

[2] *Electronic Numerical Integrator And Computer* – historicky prvý Turing-kompletný elektrónkový počítač [WIK09]

[3] *Human-Computer Interaction* – štúdium interakcií medzi ľuďmi a počítačmi

[4] *Computer-aided design* – použitie počítačovej technológie na návrh objektov, reálnych alebo virtuálnych

čin dohromady (obrázok 1.2).



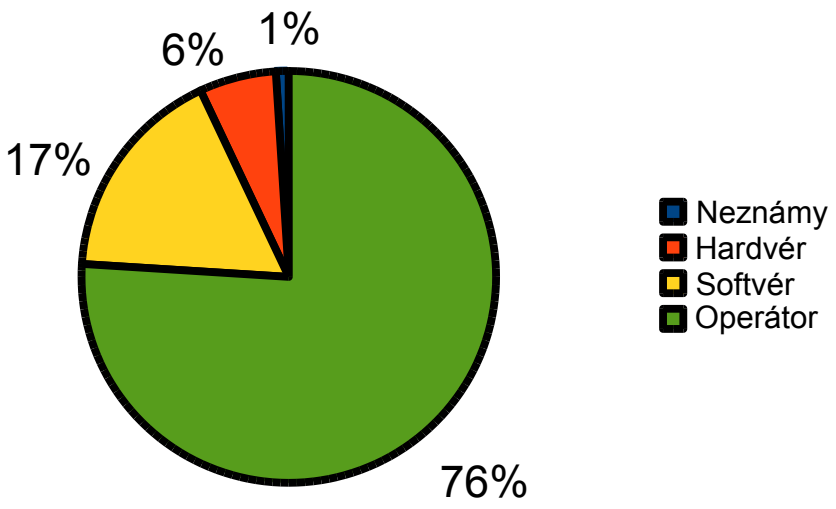
Obrázok 1.1: *Príčiny výpadkov služieb*

Undo je určitou formou zabezpečenia, alebo očakávania, že používatelia spravia chybu a undo ponúka obnovenie z tejto chyby umožnením vrátenia späť jej dôsledkov. Ide vlastne o metódu *„designing for error"*, resp. „navrhovanie pre chybu". Teda je aj určitou podporou prirodzeného ľudského učiaceho procesu, metódou uvažovania „pokus a omyl" a skúmanie jednotlivých akcií bez obáv z následkov. Tieto vlastnosti sú označované ako kritické pre efektívnu interakciu človeka so systémom. Podľa [BRO03] undo zodpovedá spôsobu akým ľudia objavujú svoje chyby – človek dokáže sám odhaliť medzi 70% až 83% svojich vlastných chýb, ale sám opraviť dokáže len 25%, 50%, alebo 70% pri korekcii na základe vedomostí, pravidiel, resp. zručnosti. Napriek tomu zostáva spôsob fungovania undo systému v niektorých aplikáciách bežnému používateľovi nejasný. V čase skorých formulácií undo funkcionality, bola väčšina používateľov interaktívnych systémov experti, alebo prinajmenšom počítačovo gramotný. Aj keď nechápali fungovanie komplexných undo mechanizmov, ako napríklad *GNU*[5] *Emacs*[6], prinajmenšom neboli zaskočení jeho nevyspytateľným správaním [PK94]. Preto pri undo systéme by malo byť jeho správanie dobre predvídateľné používateľom, aby sa používal s istotou a bez obáv z nepredvídateľných následkov.

[5] *GNU's Not Unix* – slobodný operačný systém

[6] *Editor MACroS* – pokročilý textový editor s množstvom príkazov a schopnosťou spájania príkazov do makier

Undo ako také, opravuje odhalené chyby, takže dobre zapadá s ľudskou schopnosťou samodetekcie chýb a upevňuje ľudskú schopnosť opravy týchto chýb. Treba však brať v úvahu, že undo nepomáha v procese odhaľovania chyby, ale slúži ako proces obnovy (časť tohto procesu) po odhalení chyby a vykonaní akcie na jej odstránenie.



Obrázok 1.2: *Celkový čas potrebný na opravu výpadkov rozdelený podľa príčiny ktorá výpadok spôsobila*

Kapitola 2

Cieľ diplomovej práce

V súčasnej dobe existuje značné množstvo prístupov k práci undo systému v rámci aplikácie alebo systému. Cieľom práce je na základe analýzy princípov fungovania, predpokladov, ako aj rôznych modelov a spôsobov implementácie undo systémov navrhnúť a implementovať undo a redo systém v slobodnom[1] *CASE*[2] nástroji UML .FRI. Práca by mala objasniť nejasnosti v oblasti undo systémov, navrhnúť možnosti riešenia. Z predložených riešení vybrať najvhodnejší systém a tento potom implementovať. Jednotlivé ciele by bolo možné zhrnúť do nasledujúcich bodov:

- objasnenie definície undo systému
- definovanie teoretických predpokladov undo systému
- analyzovanie prístupov používaných pri modelovaní správania sa takéhoto systému
- prezentovanie jednotlivých modelov undo systému a objasnenie spôsobov ich fungovania
- prezentovanie a analýza prístupov používaných pri implementácii undo systému
- analýza undo systémov reálnych aplikácii
- na základe prejednaných záverov výber najvhodnejšieho undo systému pre nástroj UML .FRI

[1] Pojmom slobodný softvér je v práci myslená definícia slobodného softvéru od FSF.

[2] *Computer Aided Software Engineering* – využitie počítača pri softvérovom inžinierstve, teda za účelom návrhu, tvorby a údržby počítačových programov

- implementácia tohto systému do nástroja
- popis spôsobu implementácie a fungovania undo systému v nástroji UML .FRI

Undo systém vybraný pre UML .FRI by mal mať určité všeobecné vlastnosti, bez ohľadu na vybratý model, prípadne spôsob implementácie. Teda cieľom implementácie undo/redo systému do aplikácie UML .FRI je dostať systém ktorý by bol:

- stabilný
- s rastom aplikácie ľahko rozšíriteľný
- umožňoval vrátenie, prípadne znovu vykonanie deštruktívnych operácií
- ľahko udržiavateľný, kód implementovaného systému by mal byť pokiaľ možno čo najjednoduchší
- schopný vrátiť späť, resp. vykonať znova samostatné operácie, ale aj operácie zlučovať do skupín
- mal prehľadnú vizualizáciu
- intuitívny na ovládanie pre bežného používateľa
- jeho správanie by malo byť používateľom ľahko predvídateľné

Kapitola 3

Podstata undo systému

V úvode boli pomerne všeobecne popísané základne predpoklady undo systému ako nástroja na opravu (ľudských) chýb. Pri hľadaní podstaty pojmu undo sa v odbornej literatúre [AD92, DML96], stretávame s pomerne striktným rozdelením tohto pojmu z uhlu pohľadu používateľa a systému. Zo systémového pohľadu je undo *funkcia* a ako taká niečo robí. Z pohľadu používateľa je undo *úmysel*. Jeho úmyslom je obnoviť predošlú situáciu a táto situácia môže byť obnovená využitím akejkoľvek systémovej funkcie, nie iba funkciou undo. Napriek tomu je pomerne užitočné, aby mal používateľ k dispozícii nástroj na realizovanie tohto úmyslu. Týmto nástrojom je undo funkcia, či už je k nej pristupované klávesovou skratkou, cez menu, alebo tlačidlo.

Ak uvedené predpoklady spojíme, je undo teda úmysel používateľa zabezpečený systémovou funkciou. Pri pohľade na to, čo vlastne undo funkcia robí, zistíme, že dovoľuje používateľovi dosiahnuť ihneď predošlý stav, takže undo funkcia by sa dala popísať ako funkcia, ktorá dovoľuje používateľovi odstrániť, alebo vymazať následky predošlej akcie. Vyskytujú sa však aj systémy povoľujúce „undo undo funkcie". Taktiež existujú systémy, ktoré umožňujú dosiahnuť nie len predošlý stav, ale umožňujú vybrať si, resp. obnoviť ľubovoľný vykonaný stav.

Pri zohľadnení vyššie uvedených predpokladov, voľnejšia definícia pre undo by podľa [DML96] bola: *undo je systémová funkcia umožňujúca dosiahnuť určitý stav z minulosti*. Toto podľa Dixa znamená, že undo môže byť považované za špeciálny prípad dostupnosti. Systém má vlastnosť dostupnosti, alebo v pôvodnom názve *reachability property*, ak počnúc od hociktorého stavu systému, je pre používateľa možné dosiahnuť ľubovoľný iný stav za použitia dostupných príkazov. Používateľ teda môže dosiahnuť každý stav, aj tie stavy ktoré boli dosiahnuté predtým (v minulosti), alebo tie ktoré ešte dosiahnuté neboli (možné budúce akcie). Pre samotné undo platí

samozrejme iba možnosť pohybu iba v jednom smere – do minulosti. Dix ďalej poznamenáva, že zo všetkých funkcií, ktoré môže používateľ vykonať pri interakcii s počítačom patrí undo medzi najkomplexnejšie a jeho správanie sa odlišuje od všetkých ostatných systémových funkcií. Predovšetkým po vykonaní undo funkcie, môže používateľ zistiť, že sa systém nenachádza v tom stave ako predpokladal. Táto nekonzistentnosť môže byť spôsobená viacerými dôvodmi, niektoré môžu súvisieť s danou funkciou pre undo, iné s rôznymi pohľadmi používateľa na undo bližšie popísanými v kapitole 3.2. Dôvod nejasnosti zo strany používateľov je aj pomerne veľké množstvo rôznych interpretácií undo funkcionality v jednotlivých aplikáciách, kde často správanie na ktoré je používateľ zvyknutý z jednej aplikácie sa v druhej (čiastočne) odlišuje.

3.1 Všeobecné vlastnosti

Pre undo systém je potrebné špecifikovať jednotlivé vlastnosti ktoré by mal spĺňať. Požadované vlastnosti príkazov undo systémov rozoberá napríklad [PEH00]. Niektoré vlastnosti si navzájom protirečia.

Vlastnosti reversibility a inversibility, teda schopnosť návratnosti – reverzibility, resp. inverzie sú navrhované ako postačujúce podmienky pre systém, ktorý má byť schopný stať sa systémom podporujúcim undo. Pre definovanie vlastností je potrebné definovať nasledujúce symboly:

`C` sa používa na označenie množiny úlohovo orientovaných príkazov, ktoré patria do undo domény

`U` predstavuje množinu undo príkazov

`S` reprezentuje množinu stavov nad daným objektom, kde s_0 predstavuje pôvodný stav, alebo stav eqvivalentný tomuto stavu.

Výraz $f_n(f_{n-1}(\ldots(f_1(s))))$ je skrátený na formu $f_n \ldots f_1(s)$. Následne sme schopní podľa Yanga stanoviť vlastnosti:

1. *reverzibilita (reversibility)*
   $c_1 \ldots c_n$ na $s \in S$ je reverzibilná ak $\exists u_1 \ldots u_m$ taká že

$$u_m \ldots u_1 c_n \ldots c_1(s) = s$$

Podľa [PEH00] umožňuje vlastnosť reverzibility vrátiť stav objektu do predošlého stavu, po poradí úlohovo orientovaných príkazov vykonaných nad

objektom. Takže, ak systém má vlastnosť reverzibility, efekt každého úlohovo orientovaného príkazu v rámci undo domény je možné vrátiť späť. Ak m môže byť iba 1, návrat predstavuje iba jeden krok, inak je potrebné krokov niekoľko. Schopnosť reverzibility – vrátenia späť, jedného kroku je základná vlastnosť systému podporujúceho undo. Ďalšie prípady pre rôzne hodnoty m a n sú rozobraté v [YAN88].

2. *inverznosť (inversibility)*
   Postupnosť undo príkazov $v_1 \dots v_n$ na $s \in S$ je inverzibililná ak $\exists u_1 \dots u_m$ taká že

$$u_m \dots u_1 v_n \dots v_1(s) = s$$

Vlastnosť inverzibility uvádza, že pre každý undo príkaz môžu byť jeho následky vrátené späť pomocou jedného, alebo viac undo príkazov. Teda ak undo systém má túto vlastnosť, tak efekt každého undo príkazu je možné vrátiť späť.

I keď vlastnosti reverzibility a inreverzibility sa zaoberajú návratom systému do určitého stavu ktorý predtým existoval, majú odlišnú orientáciu – reverzibilita operuje smerom dozadu a inverzibilita operuje smerom vpred.

Keďže undo systém môže mať viac než jeden undo príkaz a rôzne undo príkazy majú rôzne príkazy, navrhuje Yang ďalšie dve vlastnosti pre takýto systém – samo-aplikovateľnosť a unstacking[1].

3. *samo-aplikovateľnosť*

$$u \text{ je samo-aplikovateľné ak } \forall s \neq s_0, uu(s) = s$$

Ak má undo príkaz u vlastnosť samo-aplikovateľnosti, u dokáže vrátiť späť svoj vlastný efekt.

4. *unstacking*
   u je unstacking ak $\forall s \in S, \forall c_1 \dots c_n$,

$$u \dots u c_n \dots c_1(s) = s(\text{ n kópií } u)'$$

Ak má príkaz u vlastnosť unstacking, u nie je samo-aplikovateľné. Tieto dve vlastnosti platia pre daný undo príkaz. Sú nezlučiteľné, teda undo príkaz s vlastnosťou samo-aplikovateľnosti nemôže mať vlastnosť unstacking a naopak. Táto nezlučiteľnosť delí undo modely na dva druhy – *primitívne undo*

[1] Je použitý pôvodný anglický názov, keďže neexistuje vhodný slovenský ekvivalent. Voľnejší preklad pre unstaking by mohol byť „zrušenie naukladania na seba"

modely a *meta undo modely*. Teda môže poslúžiť ako kritérium pre delenie jednotlivých undo modelov do kategórii podľa undo príkazov na undo model s primitívnym undo príkazom, alebo meta príkazom. Toto delenie je ďalej spomenuté v kapitole 4, pri podrobnejšom popise jednotlivých undo modelov a spôsobov ich delenia. Takže, ak ide o primitívny undo model je príkaz undo samo-aplikovateľný, pri meta undo modely príkaz má vlastnosť unstacking.

Všetky doteraz spomínané vlastnosti popisovali funkčné charakteristiky undo systému. Nasledujúca vlastnosť – úplnosť (thoroughness) určuje výkon. Yang navrhuje danú vlastnosť ako podmienku pre výkon systému. Predstavuje koncept vykonaného stavu, ako stav v ktorom má používateľ znemožnené vrátenie späť akéhokoľvek príkazu, ktorý bol vykonaný pred týmto stavom.

5. *úplnosť*
   $\forall u \in U, \forall c \in C$ ak stav $u_c(s_k)$ je vykonaný stav,

$$\forall s_k \in S, \forall c_1 \dots c_n, c_n \dots c_1 uc(s_k) = c_n \dots c_1(s_k)$$

Vlastnosť úplnosti hovorí, že situácia keď sú efekty príkazu vrátené späť je ekvivalentná situácii keď ešte nebol daný príkaz vykonaný. Daná vlastnosť garantuje, že podpora undo systému je určitým obohatením použiteľnosti systému a schopnosti používateľa učiť sa z práce s ním. Zaoberá sa však iba efektami príkazov na systém, nie spôsobom ako tieto efekty dosiahnuť, to je už vec konkrétnej implementácie. [PEH00]

3.2 Pohľad používateľa

Jednou z logických potrieb pri konštrukcii možnosti undo funkcionality je pochopenie, čo musí používateľ o undo systéme vedieť v závislosti na tom aby ho mohol efektívne používať. Podľa [AD92], resp. analýzy publikovanej v [YW90] vystupujú štyri otázky, na ktoré by mal používateľ vedieť odpovedať:

1. *Aký sled aktivity je relevantný?* Pre jedno-používateľský systém, to obyčajne predstavuje postupnosť používateľových akcií, pri viac-používateľskom systéme treba brať ohľad na to, či ide o sled používateľových vlastných akcií, alebo zmiešaný sled akcií všetkých používateľov.

2. *Ako je sled akcií delený do jednotlivých častí?* Môžeme uvažovať o relevantnom slede akcií na lexikálnej úrovni – kliky myšou, stlačenie kláves; syntaktickej úrovni – príkazy; alebo sémantickej úrovni – operácie na dátovej štruktúre

3. *Ktorá časť je undo operáciou ovplyvnená?* Používateľ musí vedieť ktorá časť postupnosti akcií je relevantná pri každom vyvolaní undo operácie. Napríklad undo môže operovať na úrovni jednotlivých používateľských akcií, ale môže byť schopné vrátiť späť len poslednú takúto akciu.

4. *Aká je definícia undo funkcionality?* Napríklad ak používateľ zmaže označený blok textu klávesou Delete, vráti undo funkcia blok textu, ale bude tento text aj označený ?

Odpovede používateľov na otázky môžu byť vhodným nástrojom pri výbere, resp. doladení undo systému. Používateľ musí byť oboznámený s undo funkcionalitou, aby dokázal undo efektívne používať, teda pre undo systém by sa mali stanoviť také predpoklady, po naplnení ktorých bude systém pre cieľovú skupinu používateľov čo najintuitívnejší.

Kapitola 4

Modely undo systémov

Modely undo systémov sa odlišujú vo svojich vlastnostiach, prípadne akciách, ktoré je možné vrátiť späť. Vo všeobecnosti je možné rozdeliť undo modely podľa kritérií:

- lineárne a nelineárne
- jedno-používateľské a viac-používateľské

Trocha odlišný pohľad na rozdelenie ponúka [BRO03], ktorý navrhuje porovnávať jednotlivé undo modely na základe vlastností:

- spôsob reprezentácie samotnej undo operácie
- schopnosti vykonať selektívne undo
- linearitu histórie

Prvá vlastnosť určuje reprezentáciu samotnej undo operácie, ako jednak *primitívneho príkazu* alebo *meta príkazu*.

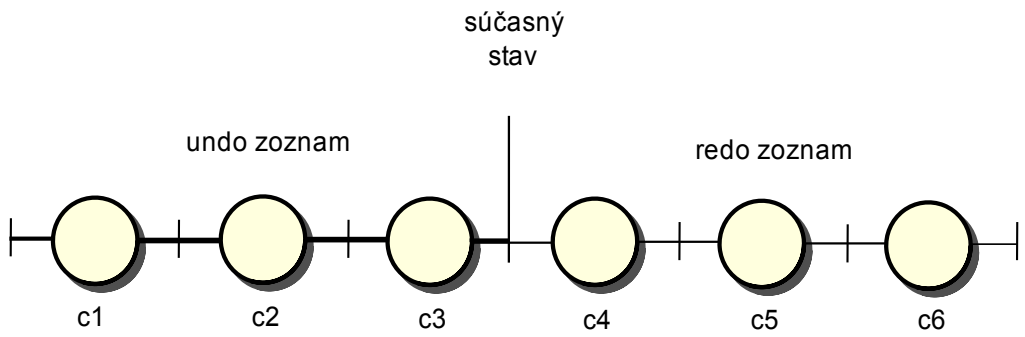
V systéme s undo modelom s primitívnym príkazom, vyvolanie príkazu undo spôsobí pridanie undo príkazu do histórie spoločne s operáciou, ktorú vracia späť. Následná undo operácia bude vlastne „undo operácie undo" a obnoví pôvodný príkaz. Undo operácia je teda samo-aplikovateľná. V takomto modely nie je výslovne určená redo operácia.

Naproti undo modelu s primitívnym príkazom je undo model založený na meta príkaze. V takomto modely je undo operácia, ktorá ovplyvňuje systémovú reprezentáciu histórie, ale nikdy sa v nej nevyskytuje. Model s undo meta príkazom majú taktiež explicitne definovaný aj redo meta príkaz.

Schopnosť selektívnej undo operácie aj s ukážkou práce modelu a obdobne linearita histórie je popísaná v kapitole 4.1.

4.1 Kritérium lineárnosti histórie modelu

Pri *lineárnom* modele undo systému musí byť vždy vrátená späť, resp. je aplikované undo na predošlú akciu, predtým ako je možné pokračovať na ďalšiu položku v histórií a obdobne akcia naposledy vrátená späť musí byť vykonaná znova – použitím redo príkazu, aby bolo možné pokračovať na ďalšiu položku redo zoznamu[1]. Teda napríklad pri postupnosti akcií $A_1, \ldots, A_i, \ldots, A_n$, vykonanie príkazu undo, vráti späť len akciu A_n, a akcia A_i nemôže byť vrátená späť bez toho, aby boli predtým postupne vrátené akcie A_{i+1}, $\ldots$,A_n. Princíp linearity histórie v undo modely demonštruje obrázok 4.1.



Obrázok 4.1: *Undo model s lineárnou históriou*

Iný prístup ponúkajú *nelinárne* modely undo systému. Jeden z týchto modelov – *selektívne undo* povoľuje ľubovoľné vybratie akcií v zozname vykonaných akcií, resp. v histórii, ktoré budú vrátené späť bez toho, aby bolo nutné vrátiť späť – použiť undo na nasledujúce akcie. Väčšina implementácii selektívneho undo systému využíva paradigma „scenárov", anglicky „script", v ktorom výsledok vrátenia späť samotnej akcie A_i je ekvivalentné výsledku dosiahnutom pri vykonaní používateľských akcií $A_1, ..., A_{i-1}, A_{i+1}, ..., A_n$ v tomto poradí. Teda ak je na zoznam používateľských akcií hľadené ako na určitý scenár, vrátenie späť jednej akcie je rovné odobratiu tejto akcie zo scenára, bez ďalších zmien. Takže napríklad pri vykonaní nasledovných akcií v textovom procesore:

[1] Za predpokladu že daný model podporuje viacnásobné undo/redo

1. `napíš`(ahoj)
2. `aplikuj_kurzívu`(ahoj)
3. `napíš`(svet)
4. `skopíruj`(ahoj)
5. `prilep na pozíciu x`

Použitie selektívneho undo príkazu v súčasnom stave (stav po piatej operácii) na druhú akciu, aplikovanie kurzívy na text, bude mať za následok odstránenie kurzívy z napísaného(prvá akcia), ale aj prilepeného (piata akcia) textu. Na druhej strane, podľa iných autorov ([DML96], [AD92]), by vrátenie späť druhej akcie vyvolalo odstránenie kurzívy iba z originálneho textu. Táto významová nejednoznačnosť poukazuje na nepomer medzi spôsobom ako je undo vnímané jednotlivými programátormi a používateľmi.[CF05] Nepomer súvisí predovšetkým z dôvodu rôznych prístupov pri sledovaní závislostí jednotlivých akcií.

Nelinearitu reprezentácie histórie demonštruje napríklad *US&R*[2] model (kapitola 4.3.7) a naopak linearitu napríklad Reštriktívneho Lineárne undo popísané v kapitole 4.3.2. Modely s nelineárnou reprezentáciou histórie udržujú viac možných verzií histórie v stromovej štruktúre s viacerými možnými vetvami v každom bode histórie. Naopak model pre Reštriktívneho Lineárne undo je lineárny undo model, pretože udržuje len jeden, lineárny pohľad na históriu systému v každom časovom bode.

4.2 Kolaboratívne kritérium

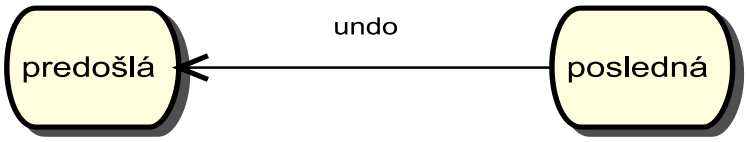
Kolaboratívne kritérium, teda či sa jedná o model pre *jedno-používateľské*, alebo *viac-používateľské* aplikácie. V aplikáciách kde je umožnené viacerým používateľom editovať dokument naraz, je potrebné samozrejme viac-používateľské undo. Tu sa však môže líšiť prístup na pohľad na históriu – teda, či používateľom vyvolaná undo operácia sa správa *lokálne*, vrátiac späť poslednú akciu vykonanú používateľom, ktorý undo operáciu spustil, alebo *globálne*, teda vráti späť poslednú používateľskú akciu bez ohľadu na to, ktorý používateľ ju vykonal. Samozrejme model lokálneho undo systému sa javí ako lepší, ale na druhej strane je potrebné, aby používateľ mohol vrátiť späť len svoju akciu a pritom mali všetci ostatní simultánne editujúci používatelia k dispozícií rozumný (aktuálny) náhľad na editovaný dokument.

[2] *Undo, Skip and Rotate* – rozšírený model undo systému

4.3 Druhy undo modelov

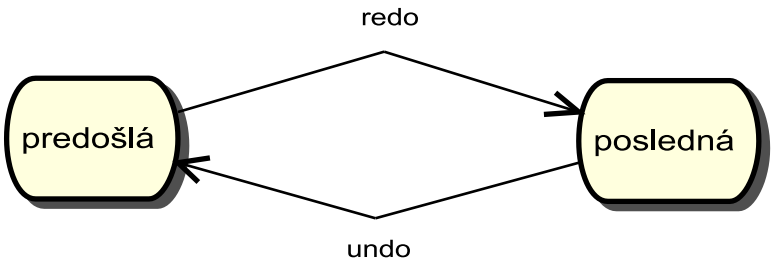
4.3.1 Single-step undo

Medzi najjednoduchší model undo systému patrí tzv. *Single-step undo*, tento model dovoľuje vrátiť späť iba poslednú akciu, teda najväčšia veľkosť histórie je jedna – tento model zobrazuje obrázok 4.2. Ide o model s primitívnym undo príkazom. Na ukážku fungovania tohto modelu, predpokladajme, že máme štyri príkazy, resp. akcie: C_1, C_2, C_3, C_4 Tieto sa zobrazia v zozname histórie. Používateľ môže vrátiť späť iba príkaz C_4, ostatné príkazy nie. Po vykonaní príkazu undo U sa systém vráti späť do predošlého stavu C_1, C_2, C_3 a zoznam histórie vypadá: C_1, C_2, C_3, C_4, U. Je zrejmé, že tento základný undo model s primitívnym príkazom je možné ďalej rozšíriť, aby podporoval aj viacnásobné vrátenie príkazov.



Obrázok 4.2: *Najjednoduchší model undo systému*

Podobne, ak model obsahuje aj možnosť redo, nazýva sa *flip undo*, keďže redo je interpretované ako opak undo operácie, potom vrátenie späť a zasa dopredu, teda undo/redo, je vlastne prepínanie sa medzi dvoma stavmi: predposledným a terajším (obrázok 4.3).



Obrázok 4.3: *Flip undo systém*

4.3.2 Reštriktívneho Lineárne undo

Pre demonštráciu reštriktívneho lineárneho undo modelu je možné použiť príklad z grafického editora: predpokladajme, že používateľ nakreslil na obrazovku kruh a následne ho premiestnil na inú pozíciu. Vytvorenie kruhu predstavuje jeden príkaz, jeho premiestnenie druhý. Následne používateľ použije undo operáciu na premiestnenie kruhu a ďalej vykoná nový príkaz – jeho zmazanie. V zozname redo príkazov sa však nachádza príkaz na premiestnenie kruhu, tento však už neexistuje, teda je jasné že redo operácia sa nepodarí.

Tento problém rieši reštriktívna verzia lineárneho undo modelu, spĺňajúc vlastnosť ktorú Berlage nazýva *vlastnosť stabilného vykonania*:

- Príkaz je vždy vykonaný znova (je použité redo) v rovnakom stave, v ktorom bol vykonaný pôvodne a je vrátený späť (je použité undo) v stave, ktorý bol dosiahnutý po pôvodnom spustení príkazu. Stav je v tomto kontexte usporiadaný zoznam príkazov, ktoré pôsobia, resp. boli vykonané a sú aktívne.

Pojem stav definovaný v popísanej vlastnosti je odlišný od definície používateľom zaznamenaného stavu aplikácie. Napríklad stav aplikácie je zmenený, keď používateľ „zoskroluje" v textovom editore svoj náhľad dole, ale

táto akcia skrolovania nie je obyčajne príkaz, ktorý je možné vrátiť späť pomocou undo operácie. Jediný prípad keď lineárny undo model porušuje túto vlastnosť je v prípade, že je vykonaný nový príkaz, keď zoznam redo príkazov – redo list nie je prázdny.

Je možné použiť aj ďalšie reštrikcie. Napríklad obmedzením veľkosti histórie. Ak je pridaný nový príkaz do zoznamu histórie a je dosiahnutá maximálna veľkosť, najstarší príkaz je zo zoznamu vymazaný. Toto nijako neporušuje definovanú vlastnosť stabilného vykonania, pretože iba obmedzuje množinu stavov, ktoré môžu byť dosiahnuté. Zmazaný príkaz zastáva naďalej súčasťou stavu, pretože ostáva vykonaný.[BER94]

Graficky je model zhodný s modelom na popísaným na obrázku 4.1, kde sa pre pohyb v histórií musia postupne prechádzať jej položky.

4.3.3 Strom histórie

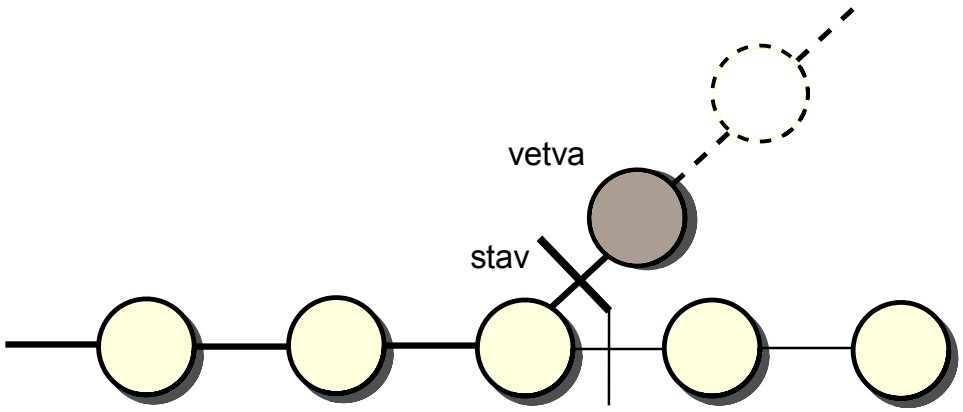
Model je vlastne rozšírením reštriktívneho lineárneho undo modelu o strom histórie bez toho, aby bola porušená vlastnosť stabilného vykonania. Redo operácia teraz potrebuje dodatočný parameter na rozhodnutie ,ktorou cestou pokračovať, ak existuje viac ako jedna vetva. V strome histórie každý stav od začiatku histórie môže byť znova reprodukovaný pomocou vrátenia späť – undo operácie, po začiatok vetvy a potom znovu vykonania – redo príkazov vybranej vetvy. Žiadne príkazy nie sú vymazané.

Graficky povolenie vetiev pri lineárnom undo modely demonštruje obrázok 4.4. Treba si uvedomiť, že stále ide o lineárny model. Nie je však veľmi rozšírený, keďže kladie na bežného používateľa väčšie nároky a môže pôsobiť mätúco.

4.3.4 ACS model

Medzi modelmi fungovania undo systému je jeden najjednoduchších a asi najnázornejší *scenarovy*(skriptový) model autorov Archer, Conway a Schneider popísaný v [ACS84]. Mnohí autori ([AD92], [DML96]) model uvádzajú ako vzorovú ukážku práce undo systému. Model sa v literatúre uvádza ako *ACS*, podľa priezvisk jeho autorov. Založený je na troch druhoch akcií – Používateľskej Histórií, Aktívnom Scenári a Čakajúcom Scenári.

- *User History* – Používateľská História – predstavuje zoznam uchovávajúci všetky používateľské akcie
- *Active Script* – Aktívny Scenár – zoznam používateľových príkazov, brané v úvahu v súčasnom stave



Obrázok 4.4: *Rozšírenie lineárneho undo modelu o vetvu*

- *Pending Script* – Čakajúci Scenár – zoznam príkazov zmazaných pomocou undo operácie v Aktívnom Scenári, ktoré sú teraz k dispozícií pre redo

Podľa [DML96] je možné tento model použiť na prejednanie skoro všetkých undo systémov a ponúka taktiež nasledovnú ukážku práce ACS modelu na vzorovom textovom editore, ktorý má funkciu undo aj redo. Môžeme sledovať rôzne scenáre, akonáhle používateľ vykoná akcie písania, undo, alebo redo. Ukážka začína z prázdneho stavu, následne ako používateľ napíše slovo, toto sa stane súčasťou Používateľskej histórie a aj Aktívneho Scenára:

Používateľská História napíš(ahoj)

Aktívny Scenár `<napíš(ahoj)>`

State ahoj

Čakajúci Scenár `< >`

Dopísanie ďalšieho slova, ktoré sa taktiež stáva súčasťou Používateľskej histórie a Aktívneho Scenára:

Používateľská História napíš(ahoj) napíš(všetci)

Aktívny Scenár `<napíš(ahoj), napíš(všetci)>`

State ahoj všetci

Čakajúci Scenár < >

Keď používateľ uskutoční undo operáciu, stane sa akcia súčasťou Používateľskej histórie, ale predošlý príkaz je z Aktívneho Scenára odstránený – na tento príkaz je „použité" undo a stav vyzerá nasledovne:

Používateľská História napíš(ahoj) napíš(všetci) undo

Aktívny Scenár <napíš(ahoj)>

State ahoj

Čakajúci Scenár <napíš(všetci)>

Aj keď bol príkaz *napíš(všetci)* odstránený z Aktívneho Scenáru je stále zaznamenaný v Čakajúcom Scenári, z dôvodu aby bolo známe, čo by sa malo vykonať po redo operácii:

Používateľská História napíš(ahoj) napíš(všetci) undo redo

Aktívny Scenár <napíš(ahoj), napíš(všetci)>

State ahoj všetci

Čakajúci Scenár < >

Autori poznamenávajú, že pre tento model si treba uvedomiť niektoré jeho predpoklady a charakteristické vlastnosti:

1. Musíme brať do úvahy dva druhy histórie, jeden zaznamenávajúci všetky používateľove akcie, druhý zaznamenávajúci scenár, resp. „script" príkazov.

2. Do úvahy pri tomto modely je potrebné vziať aj prítomnosť dvoch druhov príkazov, normálne príkazy, ako boli v ukážke písanie a špeciálne, umožňujúce undo a redo.

3. Scenáre použité v modely nemusia byť nevyhnutne zaznamenávané v reálnych undo systémoch, slúžia na popis správania sa systému, nie jeho implementáciu.

4. Stav v modely nepredstavuje celkový stav systému, zodpovedá stavu systému keď sa ignoruje undo. Vždy musia byť ukladané dodatočné informácie pre históriu.

4.3.5 Trojitý model

Pomerne zaujímavý je Yangov *Trojitý model*, ktorý definuje ďalší meta príkaz - otoč (rotate), k príkazom undo a redo. Systém v Trojitom modeli udržuje redo časť scenára v otočnom kruhovom buffri, dovoľujúc tak používateľovi vybrať si ďalší redo príkaz pomocou jeho otáčania, umožňujúci tak napríklad preskočiť existujúci príkaz alebo pozmeniť poradie ich spúšťania, ďalej vkladať nové príkazy medzi undo a redo existujúcich príkazov.

Dátová štruktúra je pomerne jednoduchá, obsahuje zoznam histórie a redo zoznam. Napríklad príkazy, ktoré boli vrátené späť sú pridané na začiatok redo zoznamu. Príkaz otočenia umiestni posledný príkaz redo zoznamu na začiatok redo zoznamu. Otáčanie sa teda využíva na vybratie príkazu, ktorý bude vykonaný znova pomocou jeho umiestnenia na začiatok redo zoznamu, alebo na vybratie miesta v redo zozname, kde nasledujúca undo operácia umiestni svoj príkaz. Redo zoznam sa ale stáva neusporiadaný a nie je spôsob ako zistiť, ktoré podmnožiny boli sekvenčne spojené.

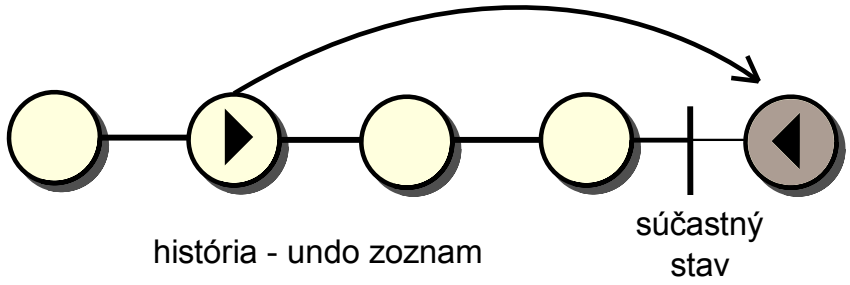
4.3.6 Priame Selektívne undo

Priame Selektívne undo, alebo Direct Selective Undo, je rozšírenie reštriktívneho lineárneho undo modelu so stromom histórie. Selektívna undo operácia vytvorí kópiu vybraného príkazu, vykoná tento skopírovaný príkaz a pridá ho na začiatok zoznamu histórie (undo zoznamu). Operácia zvyšok zoznamu nijako nemení.[BER94] Tento postup je možné vidieť na obrázku 4.5, kde bola vybraná druhá akcia, resp. príkaz. Pri selektívnom undo modely musí mať nový (vybraný) príkaz opačný – reverzný efekt originálu.

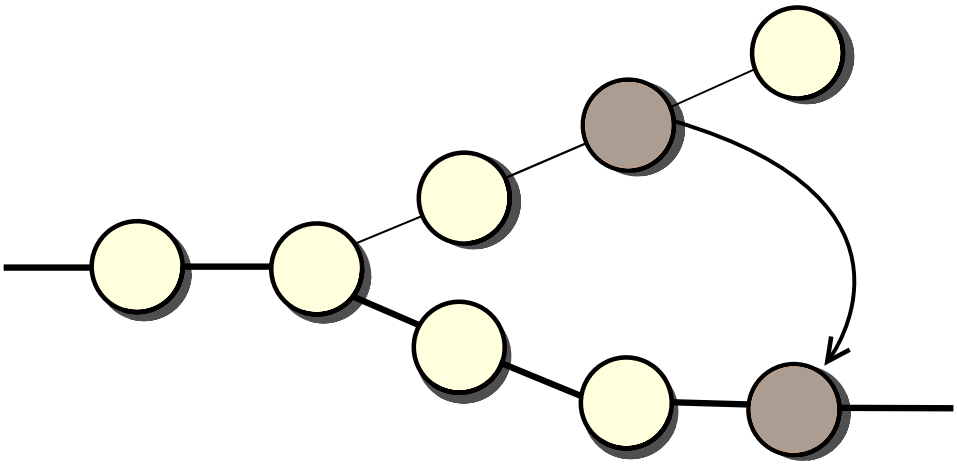
Podobne existuje selektívny redo mechanizmus ktorý povoľuje používateľovi redo príkazov zo starých vetví histórie, ktoré nie sú momentálne aktívne. Príkaz pre redo operáciu je skopírovaný, ale nemusí byť nevyhnutne reverzný(obrázok 4.6). Staré, resp. neaktívne vetvy boli vytvorené počas predošlej undo operácie a všetky príkazy na nich už boli vrátené späť. Na zistenie použiteľnosti selektívnej undo a redo operácie na príkaz sa model pozerá na samotný príkaz, namiesto na históriu ako takú. To je umožnené vďaka dvom rozdielnym metódam, ktoré overujú možnosť undo a redo operácie ľubovoľného príkazu z histórie v terajšom (danom) stave.

Berlage v [BER94] ďalej poukazuje na možnosť znovu využitia príkazov aj v jednotlivých vetvách histórie, ak boli príkazy vrátené späť, ale používateľ neskôr dôjde k záveru, že dané akcie neboli až tak zlé. Selektívne redo je pomerne dôležité, ak sú vetvy využívané na reprezentáciu rôznych verzií dokumentu.

Pre ukážku práce je v [PK94] ponúknutý príklad z grafického editora.



Obrázok 4.5: *Znázornenie modelu pre Priame Selektívne undo*



Obrázok 4.6: *Selektívne redo vykonané skopírovaním príkazu z vetvy.*

Predpokladajme, že používateľ podnikol v editore nasledujúce akcie:

C1 vytvorenie kruhu s východzou farbou výplne

C2 zväčšenie kruhu

C3 zmena farby výplne kruhu

C4 zmena pozície kruhu

C5 zduplikovanie kruhu

Následne ak sa chce vrátiť k pôvodnej farbe kruhu, bude zoznam histórie vypadať nasledovne : $C_1, C_2, C_3, C_4, C_5, C'^{-1}_3$ kde C'^{-1}_3 zmení farbu iba pôvodnému objektu, bez zmeny farby duplikovaného objektu. Vlastne inverzná kópia C'^{-1}_3 vytvorená priamou selektívnou undo operáciou je inštancia príkazu C_3, so zmenenými parametrami. V tomto príklade, ak by sme v súčasnom stave uvažovali o vrátení späť akcie C_1 z pohľadu selektívnej undo operácie by to nedávalo zmysel, kvôli závislosti C_2 na C_1. Ak by ale bol príkaz C_2 vrátený pred C_1, teda veľkosť objektu by bola rovnaká, ako jeho veľkosť pri vytvorení C_1, mohol by byť príkaz C_1 vrátený späť pomocou undo operácie.

4.3.7 US&R model

Taktiež je známy aj Vitterov *US&R model* popísaný v [VIT84], ktorý pridáva k undo a redo meta príkazom, príkaz preskoč, resp. skip v Anglickom jazyku. Skok umožňuje používateľovi preskočiť cez zaznamenané príkazy, zároveň však použiť redo na postupnosť predtým vrátených späť (undo) príkazov. Umožňuje taktiež vkladať nové príkazy medzi undo a redo existujúcich príkazov. Takže

undo vráti späť poslednú akciu súčasného stavu systému a táto akcia je zo stavu systému odstránená

skip spôsobí že akcia vrátená späť pomocou undo operácie bude preskočená a tak sa vlastne vyhneme jej znovuvykonaniu – redo akcie. Akcia preskočenia je pridaná k súčasnému stavu systému.

redo umožňuje znovu vykonanie akcie, ktorá bola predtým vrátená späť pomocou undo operácie a jej pridanie k súčasnému stavu.

US&R model taktiež demonštruje vlastnosť nelinearity histórie, keďže udržuje niekoľko možných verzií, resp. predstavuje selektívny undo systém.

4.3.8 Undo model History

History undo model taktiež ako US&R dovoľuje vrátiť späť sekvenciu príkazov, ale na rozdiel od US&R, pridaním inverzného príkazu na koniec zoznamu histórie, tak ako samotné príkazy. Počas opakovanej undo operácie príkazov, každý príkaz iný ako príkaz undo, preruší sekvenciu undo príkazov a vráti undo ukazovateľ späť na koniec zoznamu histórie. Nový undo príkaz začne proces vracania späť príkazov od konca zoznamu. Keď takáto situácia nastane sú predošlé inverzné operácie považované za bežné príkazy, ktoré môžu sami seba vrátiť späť. Takže ak máme príkazy C_1, C_2, C_3 a C_4 a príkaz C_4 bol vrátený späť za použitia undo operácie, bude zoznam histórie vypadať („→" indikuje pozíciu ukazovateľa histórie):

$$C_1, C_2, \rightarrow C_3, C_4, C_4^{-1}$$

V tomto bode pridanie nového príkazu C_5 preruší undo mód a ukazovateľ bude indikovať pozíciu príkazu C_5, ktorý bol pridaný na koniec zoznamu histórie. Ak budú vykonané dva ďalšie undo príkazy zoznam histórie bude vyzerať nasledovne:

$$C_1, C_2, C_3, \rightarrow C_4, C_4^{-1}, C_5, C_5^{-1}, C_4^{-1^{-1}}$$

Takýmto spôsobom je možné ísť späť do každého predošlého stavu v histórií bez možnosti konfliktu, keďže príkazy sa zaznamenávajú v poradí v akom boli spustené. V modeli je však aj možnosť znovuvykonania príkazu, ktorý už bol vrátený späť pomocou undo operácie v predošlom kroku, dokonca i znovavrátenie späť – re-undoing. Napríklad dva undo príkazy sú vykonané na C_4, a toto je označené ako $C_4^{-1^{-1}}$, čo predstavuje vlastne C_4 príkaz. [PEH00]

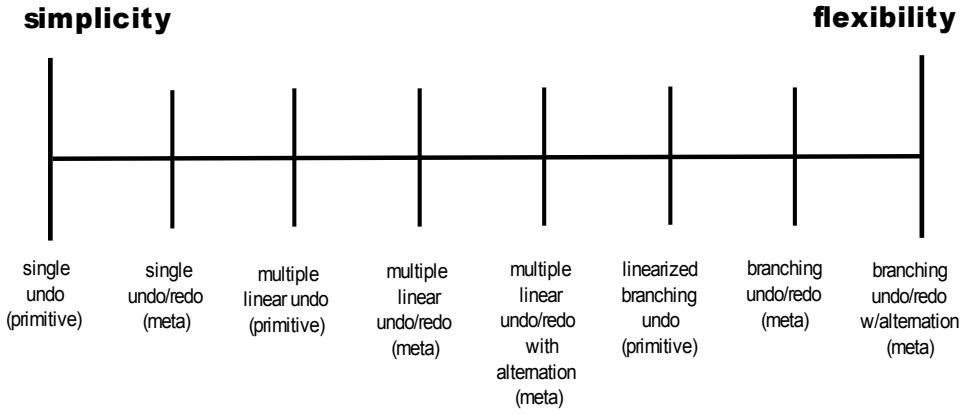
4.3.9 Timewarp

Cieľom práce síce nie je sledovať vývoj a modely viac-používateľských undo modelov, resp. systémov, ale pre úplnosť je vhodné popísať aspoň jeden z najznámejších – systém *Timewarp* (ide aj o názov toolkitu na jeho implementáciu) popísaný v [EM97]. Timewarp sa snaží riešiť problém pri nezávislej – autonómnej spolupráci, kde používatelia neoperujú konkurentne, ale nezávisle na spoločnom – zdielanom stave. Pridáva do undo/redo modelu viacnásobné histórie. Príkazy každého používateľa tvoria jeho vlastnú, osobnú históriu, história dokumentu sa potom stane akousi kompiláciou týchto histórií. Keď používateľ chce jednostranne zmeniť svoju históriu, systém musí pozlučovať novú zdielanú históriu dokumentu, ktorá berie do úvahy používateľove zmeny, zachováva históriu ostatných používateľov a identifikuje a kompenzuje sémantické konflikty, ktoré môžu nastať z danej zmeny histórie.[BRO03]

4.4 Výber undo systému

Pre výber undo systému je potrebné si určitým spôsobom porovnať, alebo usporiadať jednotlivé modely. Brown v [BRO03] ponúka umiestnenie modelov undo systému na osi, podľa flexibility modelu histórie prezentovanej používateľovi. Toto rozdelenie demonštruje obrázok 4.7. Obrázok zobrazuje či je daný model založený na primitívnom, alebo meta undo príkaze, či je história lineárna alebo s vetvami a či je povolené selektívne undo – názvy jednotlivých modelov sú teda zostavené z ich charakteristických vlastností.

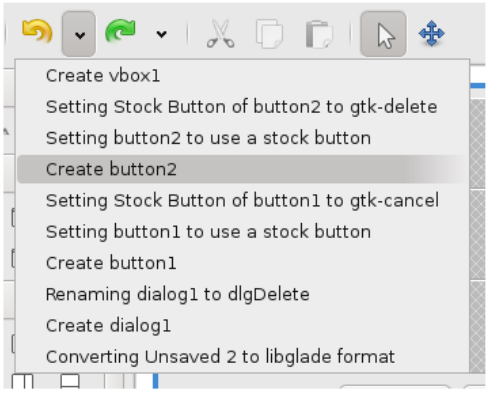
Na krajoch sú extrémy v undo modeloch – na jednej strane najjednoduchší možný model, ktorý dokáže obnoviť pravé jeden, predošlý stav a neponúka žiadny spôsob návratu do súčasného stavu, ak bol tento stav obnovený. Na druhej strane systém ponúkajúci selektívne undo a nelineárny model histórie, ako napríklad popísaný US&R model. V reálnom svete najpoužívanejšie modely sa nachádzajú v strede tejto osi, teda ide o akýsi kompromis medzi jednoduchosťou a flexibilitou.



Obrázok 4.7: *Zobrazenie jednotlivých undo modelov na osi, podľa ich jednoduchosti, resp. flexibility*

Pri vyberaní undo systému je potrebné zohľadniť niekoľko faktorov. Tieto sa môžu pri každom systéme odlišovať. Jedným z faktorov je druh aplikácie – jedná sa textový editor, tabuľkový procesor, atď. Taktiež naväzujúci faktor sú používatelia, ktorý budú daný systém používať: Aká je ich počítačová gramotnosť? Zvládnu zložitejší systém? Naučia sa ho intuitívne

používať? Z hľadiska implementácie treba zohľadniť väčší čas potrebný na implementáciu zložitejších undo modelov do systému, jeho udržiavanie, resp. rozširovanie. Taktiež spôsob vizualizácie, teda prezentovanie undo systému používateľovi – ako mu bude prezentovaný systém napríklad s kruhovým buffrom, aby s ním mohol efektívne pracovať, prípadne aká náročná bude realizácia vizualizácie tohto modelu.



Obrázok 4.8: *Slovný popis jednotlivých položiek histórie v nástroji Glade*

Koncept implementovaného lineárneho reštriktívneho undo modelu zobrazený na obrázku 4.8 je v súčasnej dobe najrozšírenejší koncept v reálnych aplikáciách. Nejedná sa však pre človeka o najintuitívnejšie prostredie pri oprave chýb, ako dokázali napríklad Cass a Fernandes v [CF05] vo svojom prieskume, v ktorom mali jednotliví pokusní používatelia vybrať najprirodzenejší model správania sa pri kroku späť. Avšak jeden z dôvodov obľúbenosti tohto prístupu je jednoduchá predvídateľnosť kroku späť, alebo vpred. Keďže je tento model prítomný v prevažnej väčšine aplikácií, mnohí používatelia sú na takéto správanie undo systému zvyknutý a preto ho aj očakávajú. Z hľadiska programátorov je takýto systém pomerne ľahko implementovateľný, ľahšie sa udržiava a vizualizácia je taktiež pomerne jednoduchá.

Kapitola 5

Implementácia systémov pre undo/redo

Spôsoby implementácie undo systému sú rôzne, môže byť napríklad implementovaný za použitia knižnice, alebo frameworku. I keď toto riešenie ponúka pomerne hotovú množinu nástrojov, resp. metód a funkcií, môže priniesť zbytočnú záťaž pri vývoji a údržbe systému. Autori sa taktiež nezhodujú na preferovanom prístupe k implementácii undo systému – môžeme sa stretnúť s návrhmi na automatickú generáciu kódu, použitím frameworku, a samozrejme aj samotným naprogramovaním systému.

Medzi problémy frameworkov sa radia predovšetkým ich zameranie na používateľské rozhranie, pravidlá pre stanovenie kedy zavolať undo mechanizmus a pre ktoré akcie môže byť zavolaný, ale detaily implementácie undo mechanizmu už ostávajú ponechané na programátorovi aplikácie.

Pred popisom samotných spôsobov implementácie, je vhodné si stanoviť rozdelenie bežných inverzných mechanizmov, ako sú popísané v [BER94]:

- *Full checkpoint* medzi používateľskými akciami sa ukladá celý stav doménového modelu. V praxi sa však používa len zriedka.

- *Complete rerun* uloží sa počiatočný stav doménového modelu a každý stav v histórií je možné dosiahnuť opakovaním príkazov v zozname histórie. Ako predchádzajúci mechanizmus, aj Complete rerun je pomerne ne-efektívny.

- *Partial checkpoint* patrí medzi najpoužívanejšie stratégie. V tomto mechanizme sa zaznamená iba tá časť stavu doménového modelu, ktorá bola používateľom zmenená, undo sa potom dosiahne ako obnovenie tohto stavu. Metóda je však zložitá na implementáciu, keďže progra-

mátor musí zaistiť, aby sa všetky relevantné zmeny zaznamenali, inak undo nebude fungovať správne.

- *Inverse function* využíva fakt, že niektoré príkazy majú inverzné funkcie, ktoré nepotrebujú dodatočné informácie. To môže byť napríklad pohnutie daného objektu na kresliacej ploche o určitú vzdialenosť – krok späť potom bude pohnutie objektu späť – v opačnom smere, o rovnakú vzdialenosť. Problémom môže byť nemožnosť použitia na nelinárne undo, ďalším problémom s inverznými funkciami je možná postupná akumulácia chýb.

Pri použití hociktorého spomenutého mechanizmu je prirodzené, že dodatočné informácie potrebné pre undo systém budú klásť väčšie požiadavky na pamäť. Najčastejšie riešenie tohto problému je v obmedzení veľkosti histórie, resp. odstránením veľmi starých položiek.

Pri implementácii undo a redo systému sa v súčastnosti prakticky všade stretávame s použitím návrhových vzorov. V nasledujúcej časti sú popísané najčastejšie vzory a prístupy použité pre realizáciu undo a redo systému.

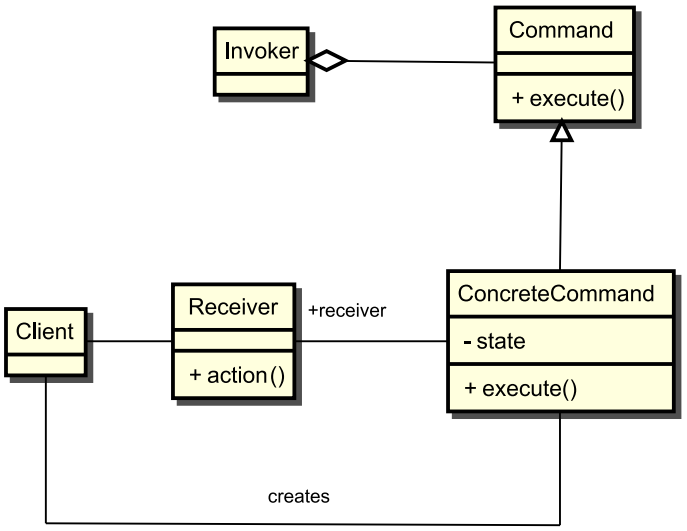
5.1 Návrhový vzor Command

V prevažnej väčšine literatúry sa pri implementácii undo/redo systému stretneme s použitím Command návrhového vzoru. V objektovo orientovanom programovaní je Command návrhový vzor, vzor v ktorom objekt reprezentuje a zapúzdruje všetky informácie potrebné pre volanie metódy v neskorší čas. Tieto Informácie zahŕňajú meno metódy, objekt ktorému daná metóda patrí a hodnoty pre parametre metódy. S návrhovým vzorom command sú spojené tri pojmy – client, invoker a receiver. *Client* vytvára inštancie jednotlivých príkazov a poskytuje informácie potrebné pre zavolanie metódy neskôr. *Invoker* rozhoduje kedy má byť metóda zavolaná. *Receiver* je inštancia triedy ktorá obsahuje kód danej metódy.[WIK09]

UML[1] diagram Command návrhového vzoru je znázornený na obrázku 5.1. Tento návrhový vzor umožňuje Complete rerun, Partial checkpoint a Inverse function prístupy popísané v úvode kapitoly.

Pre podporu undo a redo operácií stačí pridať do príkazu metódy `undo()` a `redo()`. Redo funkcia bude obvykle rovnaká, alebo veľmi podobná `execute()` funkcii, ale pri špeciálnych prípadoch môžu existovať aj výnimky. Použitie tohto návrhového vzoru v aplikácii s jednotlivými vrstvami by mohlo vyzerať podľa [BJO05] nasledovne:

[1] *Unified Modeling Language* – vizuálny jazyk na modelovanie a komunikáciu o systéme pomocou diagramov a podporného textu [RUZ08]



Obrázok 5.1: *UML diagram Command návrhového vzoru*

1. Vrstva používateľského rozhrania vytvorí nový ConcreteCommand objekt na zmenu Receiver objektu.

2. Používateľské rozhranie predá ConcreteCommand Invoker objektu, ktorý daný príkaz ConcreteCommand spustí. Invoker nevie že ide o ConcreteCommand príkaz, pretože používa Command interface.

3. Neskôr, keď si používateľ želá vrátiť príkaz späť, Invoker zavolá `undo()` metódu posledného príkazu, ktorá vráti späť jednotlivé dôsledky `execute()` metódy.

Ukážkový príklad príkazu na zmenu názvu elementu kresliacej plochy demonštruje výpis 5.1. Potrebné informácie sa predávajú pomocou konštruktora z dôvodu, aby metódy `execute()` a `undo()` boli bez parametrov.

5.2 Návrhový vzor Memento

Memento návrhový vzor predstavuje prístup ako zaistiť uchovanie stavu objektu, bez toho, aby sa tento stav stal verejne prístupným a mohol byť ovplyvnený inou častou systému. Stav objektu je sada premenných, ktoré je potrebné znova obnoviť. Ak chceme uchovať stav určitej premennej,

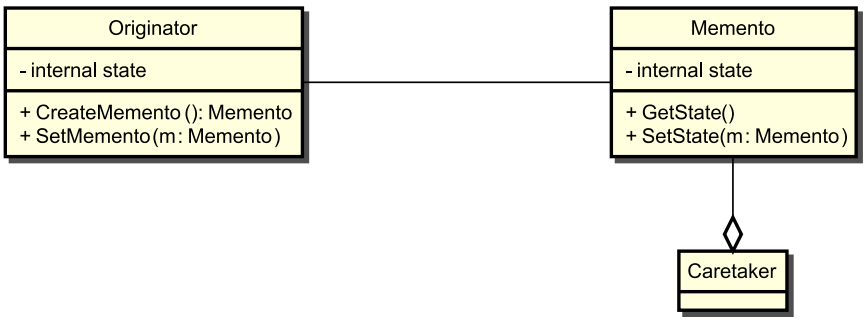
```
class RenameElementCommand:
    def __init__(self, element, name)
        self.name = name
        self.element = element

    def execute(self):
        self.old_name = self.element.get_name()
        self.element.set_name(self.name)

    def undo(self):
        self.element.set_name(self.old_name)
```

Výpis 5.1: *Definícia triedy – príkazu slúžiaceho na zmenu názvu elementu*

je jedna možnosť zápis do externého dátového zdroja. Keď ale uskutočníme toto uloženie, riešime problém, že sa týmto tento stav stal verejne prístupný a môže sa stať, že bude zmenený inou časťou systému. Tento postup odporuje zásade zapúzdrenosti dát v objekte a umožneniu prístupu k ním iba pomocou verejných metód daného objektu. Memento tento problém rieši vytvorením špeciálneho objektu, ktorý zabezpečuje práva prístupu a logiku uchovania stavu.



Obrázok 5.2: *UML diagram Memento návrhového vzoru*

Originator predstavuje zdrojový objekt, ktorý požaduje uloženie svojho

stavu a jeho znovuobnovenie v prípade potreby. Jeho stav je definovaný sadou premenných. *Originator* vytvára memento objekt a ako vstupné parametre mu predáva svoje stavové premenné. *Memento* je objekt zodpovedný za uchovanie týchto dát. Jeho základnou funkčnosťou je zaistenie bezpečnosti prístupu k týmto premenným, aby bola zaistená zapúzdrenosť dát zdrojového kódu. Väčšinou implementuje dve rozhrania. Prvé, ktoré býva zložitejšie, je využívané objektom Originator a umožňuje vytvorenie objektu, vloženie stavových premenných a získanie ich hodnôt späť. Druhé rozhranie je využívané na manipuláciu s Mementom a je určené pre objekt *Caretaker*, ktorý uchováva objekty typu Memento. Caretaker nikdy nemá prístup k obsahu Mementa, ale môže iba tieto objekty uchovávať a predávať ich iným objektom – Originator. [DVO03] Zobrazenie Memento návrhového vzoru demonštruje *UML* diagram na obrázku 5.2.

Memento návrhový vzor je možné použiť aj na undo redo funkcionalitu. Napríklad predpokladajme, že máme element kresliacej plochy a používateľ sa rozhodol zmeniť jeho farbu na modrú. Invoker povie príkazu na farbenie elementov aby sa vykonal. Príkaz pri vykonávaní najprv uloží memento elementu ktorého farbu ide meniť a potom zmení farbu elementu pomocou príslušnej metódy elementu na modrú. Neskôr, keď používateľ bude chcieť vrátiť farbu späť Invoker zavolá undo metódu príkazu na farbenie elementov, ktorý z uloženého Mementa elementu obnoví pôvodný stav objektu.

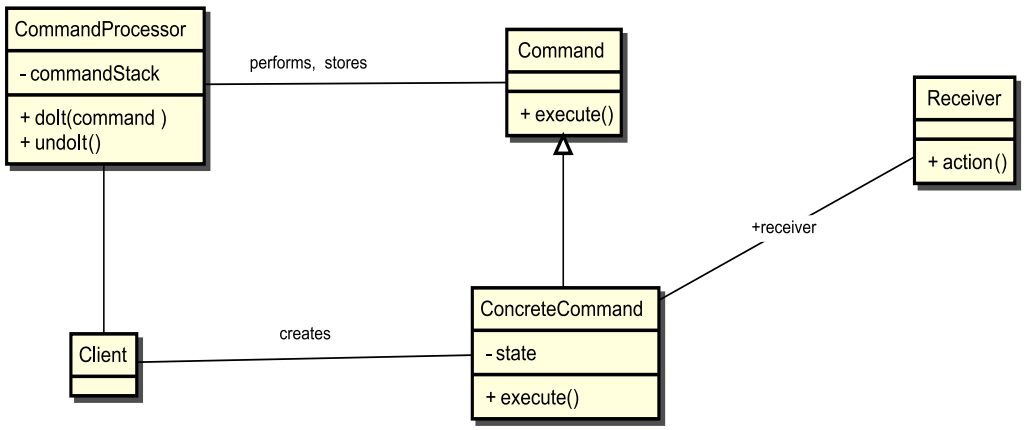
5.3 Návrhový vzor Command Processor

Command Processor návrhový vzor je určitou nadstavbou Command návrhového vzoru. Obidva vzory zapúzdrujú požiadavky do objektu. Vždy keď používateľ zavolá špecifickú funkciu aplikácie, požiadavka je premenená na Command objekt. Command Processor návrhový vzor sa zaoberá bližšie spôsobom akým sú jednotlivé Command objekty spravované. *UML* diagram Command Processor návrhového vzoru demonštruje obrázok 5.3.

Command Processor rozvrhuje spúšťanie príkazov, môže ich ukladať pre neskoršie undo a môže poskytovať aj dodatočné služby ako záznam sekvencie príkazov pre testovanie.

5.4 Návrhový vzor Composite

Zložený objekt v terminológií návrhového vzoru Composite znamená, že objekt obsahuje kolekciu iných objektov z ktorých každý môže byť buď jednoduchý alebo znova zložený objekt. Jednoduchý objekt neobsahuje referencie na



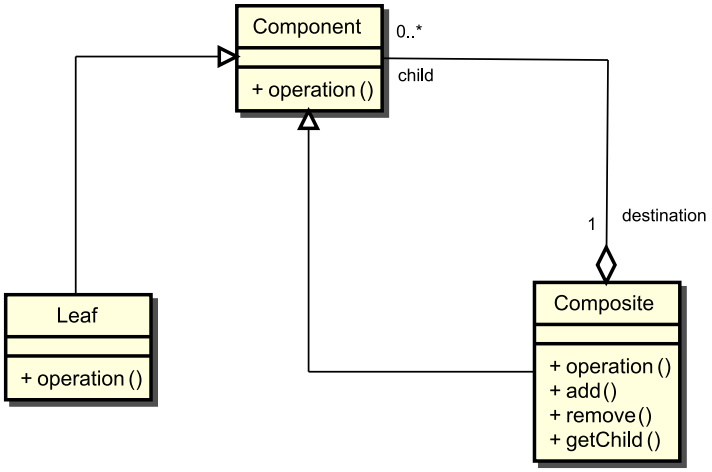
Obrázok 5.3: *UML diagram Command Processor návrhového vzoru*

iné objekty. Použitím návrhového vzoru Composite je možné usporiadať jednoduché objekty a kompozitné objekty do stromovej hierarchickej štruktúry. Táto má jeden východzí uzol (koreň), ktorý obsahuje referencie na potomkov, teda jednoduché objekty (listy), alebo zložené objekty (vetve). Každá vetva sa môže ďalej vetviť, alebo obsahuje už jednoduché koncové objekty. Composite návrhový vzor rieši situáciu, ako usporiadať túto štruktúru, aby bolo možné k vetvám i listom pristupovať jednotným spôsobom.

Všeobecne tento návrhový vzor predpokladá, že existuje jednotné rozhranie pre jednoduché objekty i kompozitné objekty, ktoré je v diagrame predstavované triedou Component.

Základnou výhodou tohto vzoru je, že sa klient nemusí starať, či získaný objekt predstavuje vetvu (*Composite*), alebo list (*Leaf*), pretože sú dedené z rovnakej triedy. Ak klientský objekt zavolá metódu `operation()` na listovom objekte, je priamo vykonaná. Pri vetve je vo väčšine prípadov táto metóda vyvolaná u všetkých potomkov a môžu byť vykonané ďalšie operácie, ako napríklad sčítanie získaných výsledkov a použitie získaného súčtu ako návratovej hodnoty. [DVO03]

Tento návrhový vzor pochopiteľne neslúži na implementáciu samotného undo systému, ale je možné jeho využitie na implementáciu jednej želanej vlastnosti a to zloženej undo operácie. Predpokladajme že máme na kresliacej ploche šesť elementov a tieto označíme a naraz zmažeme. Mazanie všetkých elementov je možné zlúčiť do kompozitného objektu. Undo príkaz by potom mal vrátiť všetky tieto objekty na jedno spustenie `undo()` operácie.



Obrázok 5.4: *UML diagram Composite návrhového vzoru*

5.5 Qt undo framework

Undo framework v *Qt*[2] je dobrým príkladom niektorých princípov a využitia popísaných vzorov. Keďže je *Qt* toolkit aj vďaka svojej licencii pomerne rozšírený, túto funkčnosť využívajú tisícky aplikácií. *Qt* undo framework využíva Command návrhový vzor. Framework pritom implementuje štyri triedy[QRD09]:

- `QUndoCommand` – základná trieda pre všetky príkazy, commands. Jednotlivé príkazy sú ukladané v undo zásobníku, undo stack. Na príkaz je možné použiť metódu undo alebo redo, vrátiac späť, resp. vpred jednu akciu vykonanú v dokumente.

- `QUndoStack` – predstavuje zoznam inštancií QUndoCommand tried. Obsahuje všetky príkazy spustené, resp. vykonané na danom dokumente a môže vrátiť stav dokumentu späť alebo dopredu, pomocou využitia undo a redo metód príkazov.

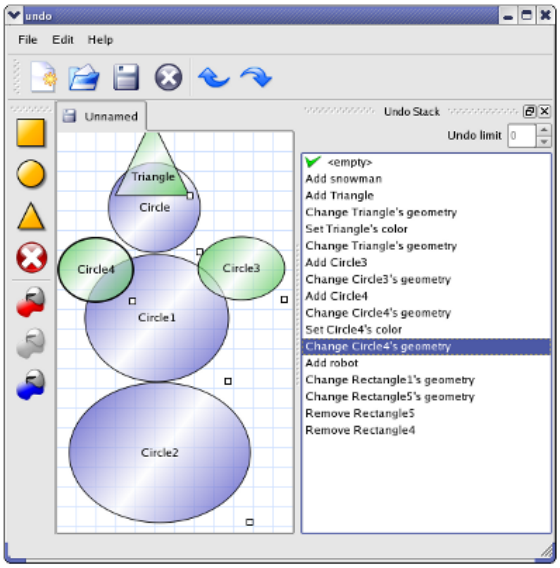
- `QUndoGroup` – je skupinou undo zásobníkov – QUndoStack. Je potrebná, ak má napríklad aplikácia otvorených viac dokumentov. Predáva

[2] *„Cute"* – multiplatformový framework pre tvorbu aplikácií, používaný predovšetkým na tvorbu GUI programov, ale taktiež na konzolové nástroje a serverové aplikácie [WIK09]

požiadavky na undo a redo aktívnemu undo zásobníku, ktorý patrí dokumentu práve editovaným používateľom.

- `QUndoView` – ide o *widget*[3] ktorý zobrazuje obsah undo zásobníka

Framework ďalej podporuje aj koncepty ako kompresiu príkazov – *command compression*, teda „stlačenie" postupnosti príkazov do jedného príkazu.



Obrázok 5.5: *Qt undo framework a jeho implementácia*

Napríklad napísanie jednotlivých písmen v textovom editore bude stlačené do príkazu pre napísanie slova. Koncept príkazových makier – *command macros* umožňuje spojiť väčší počet príkazov a následne aplikovať undo/redo operácie na všetky príkazy naraz. Napríklad príkaz pre pohnutie označených objektov sa vytvorí skombinovaním príkazov pre pohnutie jednotlivých objektov.

Qt undo framework ponúka na používanie jednoduchý a pomerne prepracovaný undo systém. Jeho implementáciu na ukážkovej aplikácii od jeho tvorcov je možné vidieť na obrázku 5.5.

[3] *„window gadget"* – element grafického používateľského rozhrania

5.6 Undo v používateľských aplikáciách

Emacs má pomerne pokročilý systém pre undo. Podľa [STA07] všetky zmeny v textovom buffri môžu byť vrátené späť (do určitého počtu zmien, nejde o „nekonečné" undo). Spustenie undo príkazu vráti späť efekt posledného príkazu. Viacnásobné vyvolanie undo operácie vráti text až do poslednej zaznamenanej pozície. Každý príkaz iný ako undo, preruší sekvenciu undo príkazov. Od tohto momentu, predošlé undo príkazy sa stanú obyčajnými zmenami, ktoré môžu byť vrátené späť pomocou ďalšieho undo príkazu. Toto je ekvivalentné s aplikovaním redo operácie na pôvodné príkazy. Bežné undo platí pre všetky zmeny na aktuálnom buffri. *Emacs* dovoľuje aj vykonanie selektívnej undo operácie limitovanej na určitý región.

Vi[4] editor obsahuje dva oddelené undo príkazy na vrátenie späť poslednej zmeny a všetkých zmien vykonaných na danom riadku. Pri celkovej undo operácii predstavuje dva typy správania: v prvom je možné vrátiť sa späť za použitia undo príkazu, zasa dopredu s použitím redo príkazu, ale keď sa spraví zmena po undo príkaze redo už nie je možné. Druhý, nazvaný vi kompatibilný spôsob, kde undo príkaz vráti späť efekt predošlého príkazu a tiež predošlého undo príkazu. Redo príkaz potom zopakuje predošlý undo príkaz.[MOO08]

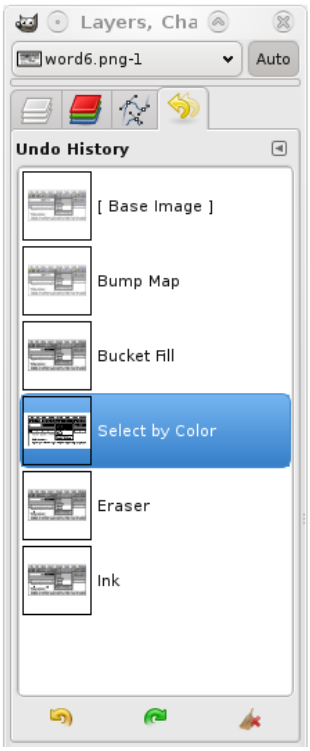
GIMP[5] používa jednoduchý, ale efektívny spôsob lineárnej histórie. Keď používateľ použije undo príkaz, môže vrátiť predošlé stavy, alebo pomocou redo príkazu stavy zrušené undo operáciou. Pri vykonaní príkazu iného ako undo, alebo redo, sa všetky príkazy vrátené späť pomocou undo operácie stratia. *GIMP* ako grafický editor má pomerne zaujímavý spôsob zobrazovania zoznamu undo a redo zmien, ktorý je vidieť na obrázku 5.6. Ako je možné vidieť *GIMP* na vizuálne zobrazenie undo a redo zoznamu používa jeden spoločný zoznam, v ktorom je možné zmeny zistiť vďaka popisu a markantnejšie zmeny na editovanom obrázku aj na jeho miniatúre. Aktuálna pozícia je určená modrým označením.

Glade[6] aplikácia, používaná aj na návrh GUI *CASE* nástroja UML .FRI, zobrazená na obrázku 5.7, taktiež využíva lineárne reštriktívne undo. Jednotlivé akcie na ktoré je možné undo a redo použiť popisuje, používateľ má tak lepší prehľad a orientáciu v zozname histórie a redo zozname.

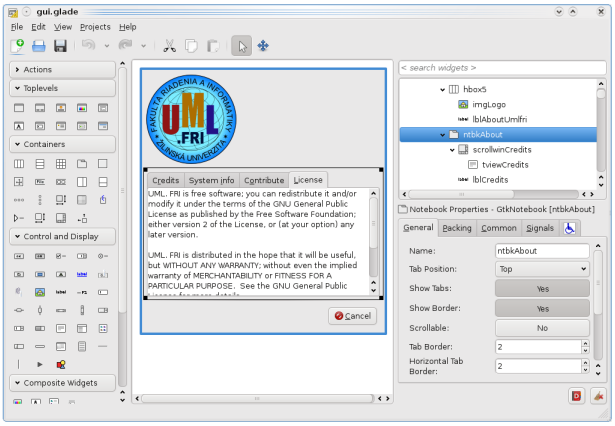
[4] *rodina textových editorov* – zdieľajúca spoločné charakteristiky, ako napríklad spôsob ovládania a používateľský interface.

[5] *GNU Image Manipulation Program* – slobodná multiplatformová aplikácia slúžiaca na tvorbu a úpravu rastrovej grafiky

[6] *Glade Interface Designer* – nástroj na tvorbu grafických používateľských rozhraní pre GTK+



Obrázok 5.6: *GIMP a jeho grafická reprezentácia undo a redo zoznamu*



Obrázok 5.7: *Návrh GUI UML .FRI pomocou Glade aplikácie*

Kapitola 6

Undo a redo subsystém v nástroji UML .FRI

UML .FRI je slobodný *CASE* nástroj, ktorého vývoj začal na Fakulte Riadenia a Informatiky, Žilinskej Univerzity. Cieľom bolo jednak vytvorenie nástroja na používanie, výučbu a prácu s *UML*, za ktorý by škola nemusela platiť licenčné poplatky, ale aj vyplnenie určitej medzery na trhu s podobnými nástrojmi.

Medzi charakteristické vlastnosti UML .FRI patrí jeho multiplatformovosť, v súčastnosti medzi podporované platformy patrí Linux, Windows, MacOS, *BSD*[1] a Solaris. Ďalej pomerne jednoduchý spôsob tvorby nových modelov, používateľská prívetivosť, a v neposlednom rade fakt, že je vyvíjaný pod slobodnou *GPL*[2] licenciou, garantujúc tak každému práva používať, študovať, kopírovať, modifikovať a redistribuovať tento nástroj.

UML .FRI aplikácia je v čase písania tejto práce ešte vo verzii beta. Je však už aktívne využívaná či jednotlivcami na tvorbu diagramov, tak aj Fakultou Riadenia a Informatiky, kde je ponúkaná ako alternatíva pri výučbe viacerých predmetov. Aplikácia sa radí medzi slobodný softvér a pri procese vývoja sa využíva tzv. *Bazaar model* alebo slovensky „trhový model". Ide o označenie z [RAY99], kde sú predstavené dva modely vývoja *FOSS*[3] softvéru, bežne používané v praxi. Spomínaný trhový model znamená, že aplikácia je vyvíjaná prostredníctvom Internetu, kde má široká verejnosť prístup k zdrojovému kódu aj medzi verziami, možnosť posielať opravy, prípadne sa inak aktívne zapojiť do vývoja. Za vynálezcu tohto modelu autor považuje

[1] *Berkeley Software Distribution* – operačný systém, derivát pôvodného systému Unix. V súčastnosti sa pojem používa na súhrnné označenie vetvy, resp. rodiny nástupcov tohto systému, ako napríklad FreeBSD, NetBSD alebo OpenBSD[WIK09]

[2] *GNU General Public License* – licencia pre slobodný softvér

[3] *Free and open source software* – Slobodný softvér a softvér s otvoreným kódom

Linusa Torvaldsa, autora Linuxového jadra. Opakom je *Cathedral model* – katedrálový model, pri ktorom je síce zdrojový kód k jednotlivým verziám k dispozícií, ale zmeny v kóde medzi verziami sú dostupné len úzkej skupine vývojárov, ako príklad môže slúžiť *GNU Emacs*. Zdrojové kódy hlavnej, ako aj vývojových vetiev UML .FRI sú teda k dispozícii online[4] vo verejne prístupnom *svn*[5] repozitári, ďalej je umožnené nahlasovanie nových bugov aj pre anonymných používateľov jednak cez projektový bugtracking systém, alebo pohodlne z okna aplikácie.

Keďže aplikácia UML .FRI je intenzívne vyvíjaná tímom ľudí už niekoľko rokov, narástla do pomerne veľkých rozmerov z hľadiska funkcionality, ale aj komplexnosti. Pred samotným popisom postupu implementácie systému undo a redo do tejto aplikácie, je pre úplnosť a pochopenie postupov vhodné popísať jednak architektúru tohto systému, ako aj technológie použité pri jeho návrhu a programovaní, ako sú popísané v [JBO09].

6.1 Technológie

Snaha spraviť UML .FRI ľahko používateľnou, stabilnou, multiplatformovou a slobodnou aplikáciou ovplyvnila aj výber technológií. Použité technológie patria medzi overené, masovo používané a vysoko oceňované.

Jazyk Python[6] je všeobecne využiteľný, vysoko úrovňový programovací jazyk. Navrhnutý bol tak, aby pri práci s ním programátor maximalizoval produktivitu a čitateľnosť zdrojových kódov. Umožňuje používať pri programovaní viaceré paradigmy – objektovo orientovaný, imperatívny, štruktúrovaný a funkcionálny prístup. V jazyku je implementovaná automatická správa pamäte. Výhodou jazyka Python je multiplatformovosť. To znamená, že jeden zdrojový kód možno používať bez zásahov na väčšine platforiem. Podmienkou je, že sa programátor vyhne použitiu vlastností špecifických iba pre jednu platformu. (Napríklad napevno použité spätné lomítka pri určení cesty k súboru.)

GTK+[7] je na platforme nezávislý súbor nástrojov pre tvorbu grafických používateľských rozhraní. Knižnica bola pôvodne napísaná v C jazyku. Vydaná je pod slobodnou *GNU* Less General Public License, ktorá umožňuje vývoj slobodného, ale aj proprietárneho softvéru. V súčastnosti je založená predovšetkým na štyroch knižniciach, taktiež vyvíjaných GTK projektom –

[4] Zdrojové kódy je možné získať na adrese `https://umlfri.org/svn/`

[5] *Subversion* – slobodný systém pre správu softvéru a jeho verzií

[6] V texte je pod pojmom Python myslená implementácia v C jazyku, tzv. CPython.

[7] *The GIMP Toolkit* – slobodný multiplatformový toolkit slúžiaci na tvorbu grafických požívateľských rozhraní, pôvodne vznikol pre potreby aplikácie GIMP

Glib(objektová podpora pre C), Pango (text a jeho jazyková lokalizácia), Cairo(grafická knižnica) a *ATK*[8]. UML. FRI využíva PyGTK, teda set Python obaľovacích tried (wrapper) pre *GTK+* knižnicu.

Glade uľahčuje návrh a implementáciu používateľských rozhraní pre UML .FRI. GUI navrhnuté v *Glade* je uložené ako *XML*[9] súbor a za použitia libglade knižnice je možné toto používateľské rozhranie podľa potreby dynamicky načítať aplikáciou. Návrh UML .FRI používateľského rozhrania demonštruje obrázok 5.7.

6.2 Architektúra

Z hľadiska architektúry je aplikácia UML .FRI rozdelená do vrstiev štruktúrovane – na vrchnej úrovni sa systém rozdeľuje na dve vrstvy, *prezentačnú* a *logickú*. Logická vrstva je ďalej rozdelená na *logickú* a *vizuálnu* časť. Toto delenie vyplynulo z logiky nástroja.

Celková architektúra systému UML .FRI je zobrazená na obrázku 6.1. Naznačené je aj prepojenie systému s niektorými knižnicami, ako je knižnica Cairo, alebo knižnica *GTK+*.

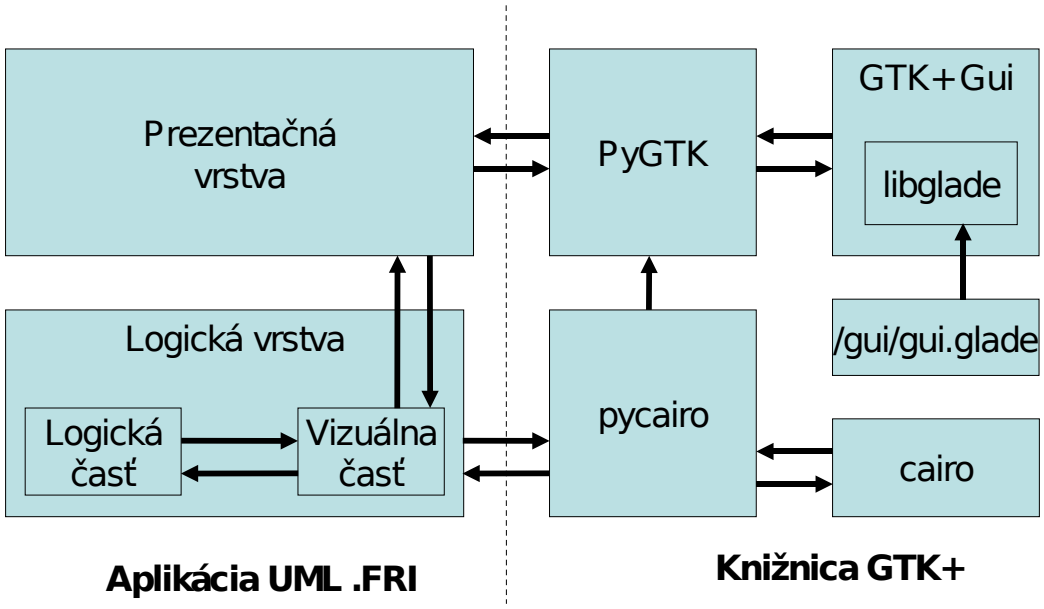
Prezentačná vrstva slúži, ako u väčšiny softvéru, na komunikáciu s používateľom. Predstavuje grafické rozhranie a jeho logiku. Všetky triedy prezentačnej vrstvy sú uložené v module `lib.Gui`. Sú tam dva typy objektov: *okná* a ich *komponenty*. Komponent vždy predstavuje určitú časť okna, ktorá má definovanú určitú vnútornú logiku. Rozdelenie na komponenty bolo vytvorené kvôli prehľadnosti zdrojových kódov, nakoľko väčšina logiky prezentačnej vrstvy sa nachádza v hlavnom okne programu.

Delenie logickej vrstvy na logickú a vizuálnu časť sa môže zdať mätúce, ale skvele sa hodí pre potreby aplikácie. Pre pochopenie je možné uviesť príklad: máme element, ktorý sme pridali na plochu – teda do určitého diagramu. Projekt ale obsahuje viac diagramov a ten istý element sa môže vyskytovať v každom z nich[10]. Element v projekte predstavuje jedna inštancia `CElementObject` z logickej časti logickej vrstvy a jeho zobrazenie na jednotlivých diagramoch `CElement`(na každom diagrame, kde je element zobrazený) z vizuálnej časti logickej vrstvy.

[8] *Accessibility Toolkit* – toolkit umožňujúci aplikácií používať nástroje ako alternatívne vstupné zariadenia.

[9] *eXtensible Markup Language* – rozšíriteľný značkovací jazyk, umožňujúci jednoduché vytváranie konkrétnych značkovacích jazykov

[10] Ak ide o diagramy rovnakého typu, resp. daný element logicky do daného typu diagramu patrí



Obrázok 6.1: *Architektúra systému UML. FRI*

6.3 Výber undo modelu

Pri výbere undo modelu bol braný ohľad na jednoduchosť pri používaní, ako aj na prehľadnosť pri implementácii a následnom udržiavaní kódu. Ako najlepšia voľba bol preto vybraný model reštriktívneho lineárneho undo systému, ktorý je bližšie popísaný pri pojednaní o modeloch undo systémov v kapitole 4.3.2. Tento systém je najrozšírenejší a pre bežne počítačovo gramotného používateľa, ale aj experta, ide o prirodzený spôsob práce s históriou aplikácie.

Ak sa pozrieme na podobne orientované aplikácie, teda určené pre podporu vývoja softvéru, s podobnou cieľovou skupinou používateľov, ako napríklad *RAD nástroj*[11] *Glade*, zistíme, že používajú spomínaný model undo systému. Porovnanie s *Glade* nástrojom má význam aj z dôvodu, že ide o celosvetovo rozšírený nástroj s množstvom používateľov, taktiež využívajúci *GTK+* s podobnou funkcionalitou resp. grafickým rozhraním. Podobnosť grafického rozhrania aplikácie UML .FRI s *Glade* nástrojom demonštruje obrázok 6.3. Táto podobnosť je však náhodná, vyplynula z potrieb aplikácie UML .FRI umožniť efektívnu prácu v logicky a intuitívne rozdelenom grafickom rozhraní.

[11] *Rapid application development tool* – nástroj na rýchly návrh, alebo vývoj aplikácie

Pridané reštrikcie do modelu sú v prípade UML .FRI obmedzenie veľkosti histórie. Taktiež pre dodržanie vlastnosti stabilného vykonania popísanej v kapitole 4.3.2 – k jej porušeniu by došlo ak by redo zoznam nebol prázdny v momente vykonania ďalšieho príkazu, resp. akcie, bude v UML .FRI v takomto prípade redo zoznam vyprázdnený.

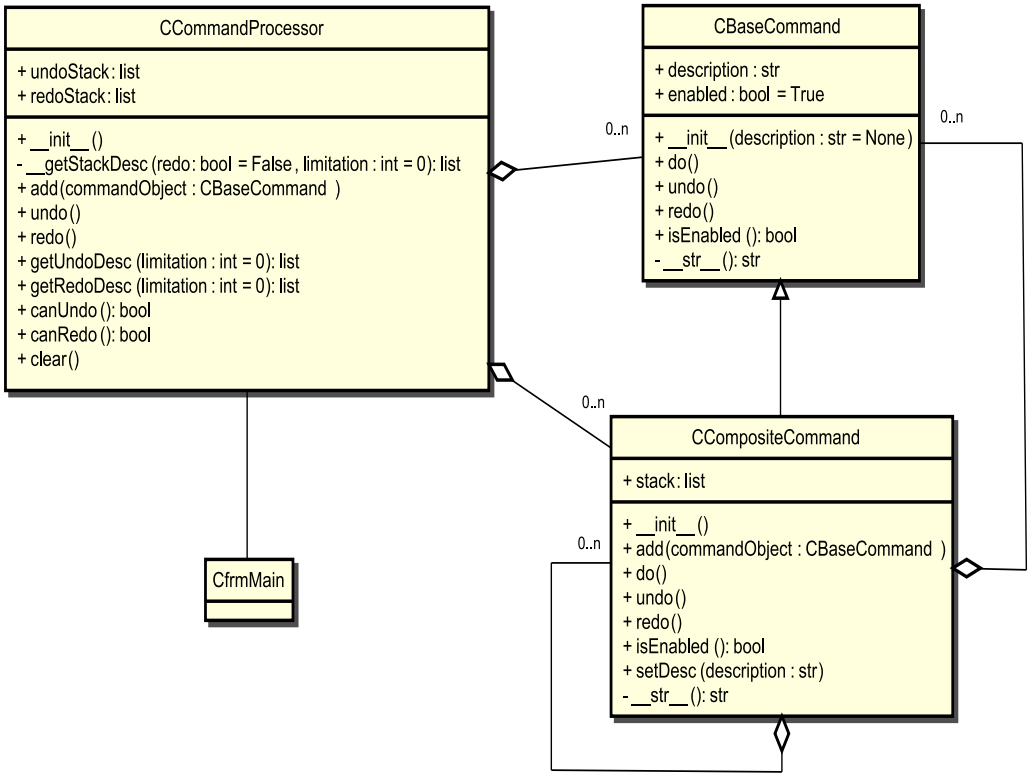
6.4 Implementácia

Undo a redo systém je v UML .FRI implementovaný za využitia návrhových vzorov, predovšetkým inšpirovaný návrhovým vzorom CommandProcessor (kapitola 5.3). Jazykom implementácie je podobne ako pri samotnej aplikácii jazyk Python. V súborovom strome aplikácie sa triedy realizujúce undo/redo funkcionalitu nachádzajú v `lib/Commands/` adresári, resp. podadresároch. Základom undo/redo systému sú tri triedy `CCommandProcessor`, `CBaseCommand` a `CCompositeCommand`. *UML* diagram týchto tried je zobrazený na obrázku 6.2. Na diagrame je možné vidieť aj prepojenie návrhových vzorov, ktoré logicky vyplynulo pre potreby danej funkcionality undo systému. Ide o spomínaný návrhový vzor Command, resp. jeho nadstavbu CommandProcessor s návrhovým vzorom Composite, ktorý je potrebný pre tvorbu zložených príkazov, resp. zoskupovanie používateľových akcií, ktoré spolu logicky súvisia. Vytvorené prepojenie návrhového vzoru CommandProcessor so vzorom Composite rozširuje možnosti undo/redo systému a je typický práve pre aplikáciu UML .FRI.

Z hľadiska popísanej architektúry UML .FRI tvorí undo/redo systém medzivrstvu nachádzajúcu sa medzi prezentačnou a logickou vrstvou. Používateľov úmysel je zachytený prezentačnou vrstvou, spracovaný undo/redo systémom, ktorý následne prevedie potrebné operácie v logickej vrstve aplikácie. Z dôvodu nutnosti jednotlivých príkazov uchovávať dodatočné informácie na obnovu predošlého stavu, resp. opätovného návratu z tohto stavu, teda undo/redo funkcionalitu, zasahuje undo/redo systém do oboch častí logickej vrstvy – vizuálnej aj logickej. Pri implementácii undo/redo systému došlo teda k úplnému oddeleniu prezentačnej a logickej vrstvy.

Pri implementovaní undo/redo systému vznikol problém obnovovania používateľského rozhrania. Ako je aj z obrázku UML .FRI aplikácie zrejmé, z hľadiska užívateľského rozhrania je možné aplikáciu rozdeliť na kresliacu plochu, strom projektu a vlastnosti prvkov projektu. V aplikácii sú spomínané komponenty reprezentované triedami `CpicDrawingArea`[12], `CtwProjectView` a `CnbProperties`. Bolo potrebné implementovať obnovovanie odlišným

[12]Jednotlivé triedy aplikácie UML .FRI relevantné k práci undo systému sú popísané v dodatku C



Obrázok 6.2: *UML model základných tried zabezpečujúcich undo/redo funkcionalitu*

spôsobom ako bolo v UML .FRI používané doposiaľ, keď často dochádzalo iba k čiastočnej obnove, napríklad len o pridaný diagram v strome projektu. V stave s implementovaným undo/redo systémom je potrebné robiť obnovu nie len v čase (prvotného) vykonania príkazu, ale aj po vykonaní undo/redo operácie. Do množiny obnovovaných častí GUI je potrebné pridať aj záložky s otvorenými diagramami reprezentované triedou `CTabs`. Bližšie to demonštruje príklad: Používateľ pridá nový diagram do projektu, ktorý sa automaticky otvorí v novej záložke. Je evidentné, že je potrebná obnova spomínaných časti používateľského rozhrania. Potom sa ale rozhodne, že chce danú akciu vrátiť späť – vykoná operáciu undo, funkcia obnovy používateľského rozhrania sa postará o prekreslenie stromu projektu, ako aj vlastností, ďalej je potrebné zistiť či je daný diagram otvorený v záložke, ak áno, bude záložka zatvorená a dôjde k prekresleniu kresliacej plochy. Táto časť nespadá priamo do undo/redo systému, pretože undo/redo systém nemá k používateľskému rozhraniu prístup, ale rozšírenie schopnosti obnovovania GUI bolo nutné pre správne fungovanie samotného undo/redo systému.

6.4.1 CommandProcessor

Na spúšťanie používateľom vykonaných príkazov, ale aj zaznamenávanie samotnej histórie príkazov, teda undo zoznamu, ako aj akcií vrátených späť – redo zoznamu, slúži v aplikácií trieda `CCommandProcessor`. Ako atribúty trieda obsahuje dva zoznamy `self.undoStack` a `self.redoStack` slúžiace na ukladanie používateľových príkazov.

Nové príkazy sa do zoznamu histórie `self.undoStack` pridávajú pomocou metódy `add()`. Metóda overí, či pridávaná inštancia je skutočne používateľský príkaz, ak áno, tento príkaz vykoná a v prípade, že vykonanie prebehlo v poriadku zaradí ho do undo zoznamu. Pred zaradením vyprázdni redo zoznam. Ďalej sa metóda stará aj o dodržanie reštrikcie veľkosti zoznamu histórie. Ak zoznam prekročí stanovenú veľkosť sú najstaršie položky zahodené. Veľkosť undo zoznamu je definovaná v `lib/consts.py` premennou `STACK_MAX_SIZE`.

Trieda ďalej definuje metódy `undo()` a `redo()`. Tieto operácie sú prakticky identické, odlišujú sa len v zozname s ktorým pracujú. Ukážka kódu metódy `undo()` je vo výpise 6.1. Metóda `canUndo()`, resp. `canRedo()` vráti hodnotu True, ak nie je undo resp. redo zoznam prázdny. Vypísaná `undo()` metóda potom pomocou metódy `pop()`, odstráni z undo zoznamu poslednú položku a pridá ju na koniec redo zoznamu.

`CCommandProcessor` trieda ďalej definuje metódy na získanie opisu jednotlivých príkazov, uchovávaných v undo a redo zozname, slúžiacu pri vizualizácii systému a metódu `clear()` na vyčistenie týchto zoznamov.

```
class CCommandProcessor:

[...]

    def undo(self):
        if self.canUndo():
            undoStackItem = self.undoStack.pop()
            undoStackItem.undo()
            self.redoStack.append(undoStackItem)

[...]
```

Výpis 6.1: *Metóda undo() triedy CCommandProcessor*

6.4.2 BaseCommand

Trieda `CBaseCommand` predstavuje predka všetkých príkazov použitých v aplikácií a definuje základné metódy potrebné pre každý príkaz. Samotná trieda je pomerne jednoduchá a je zobrazená vo výpise 6.2

Atribútmi `CBaseCommand` sú `self.description` – popis príkazu, ktorý sa využíva pri vizualizácii systému, aby bolo používateľovi jasné, čo daný príkaz vykonal a `self.enabled`. Inštancia triedy `CCommandProcessor` v popísanej metóde `add()` overuje či je hodnota `self.enabled` rovná True, pomocou metódy príkazu `isEnabled()`. Potreba zistenia či je daný príkaz enabled, teda povolený je zrejmá: počas vykonávania príkazu mohlo dôjsť k situácií, že daný príkaz nezmenil stav aplikácie a príkaz sa preto nemôže stať súčasťou histórie. Napríklad, zmenu mena elementu realizuje príkaz, teda ak máme názov elementu „Class1" a používateľ premenuje element na rovnaký názov – „Class1", príkaz to pri vykonávaní zistí, nastaví `self.enabled = False`, CommandProcessor ho nepridá do histórie.

`CBaseCommand` ďalej definuje metódy `do()` a `undo()`, metóda `redo()` je vo východzom nastavení vlastne opätovné volanie `do()` metódy, ale v mnohých reálnych príkazoch tomu tak nie je. Tieto metódy budú u potomkov triedy vykonávať operácie nutné pre vykonanie daného príkazu, jeho vrátenie späť a prípadné znovuvykonanie. Tieto metódy nesmú mať parametre, ani návratové hodnoty. Všetky dáta potrebné pre vykonanie príkazu, resp. jeho vrátenie späť sa predávajú cez konštruktor konkrétneho príkazu.

```
class CBaseCommand:

    def __init__(self, description = None):
        self.description = description
        self.enabled = True

    def do(self):
        pass

    def undo(self):
        pass

    def redo(self):
        self.do()

    def isEnabled(self):
        return self.enabled


    def __str__(self):
        if self.description == None:
            return _("History Operation")
        else:
            return self.description
```

Výpis 6.2: *Definícia spoločného predka jednotlivých príkazov*

6.4.3 CompositeCommand

Trieda `CCompositeCommand` je potomok triedy `CBaseCommand`. Slúži ako trieda na zoskupovanie ostatných príkazov, ktoré spolu logicky súvisia, resp. majú byť vykonané v jednej používateľovej undo/redo operácii. Ide o inšpirovanie sa návrhovým vzorom Composite (kapitola 5.4). Do inštancie `CCompositeCommand` môžu byť pridané jednak objekty typu `CBaseCommand`, ale aj `CCompositeCommand`. Používa atribút `self.stack` na uchovávanie zoznamu príkazov. Príkazy sa pridávajú do zoznamu pomocou metódy `add()` a jednotlivé metódy `do()`, `undo()` a `redo()` vykonajú iteráciu po pridaných príkazoch a pre každý spustia spomínanú do, undo alebo redo operáciu. Zložený príkaz je pridaný do `CCommandProcessor` inštancie kde sa k nemu pristupuje ako k iným príkazom.

6.4.4 Implementácia príkazov

V aplikácií sú implementované asi dve desiatky príkazov, potomkov triedy `CBaseCommand`. Jednotlivé rozmiestnenie príkazov, teda ich vytváranie v komponentoch GUI demonštrujú *UML* diagramy v prílohe A. Podrobnejší výpis názvov tried, ich umiestnenie v súborovom strome aplikácie, ako aj ich funkcia je rozpísaná v dodatku B.

6.5 Spôsob práce undo/redo systému

Všeobecný popis fungovania systému undo/redo v aplikácii UML .FRI by bol: Používateľov úmysel vykonať požadovanú zmenu zaregistruje prezentačná vrstva aplikácie. Ak sa jedná o akciu, ktorej zmenu chceme zaznamenať v histórii aplikácie, zadefinujeme pre túto akciu triedu – potomka `CBaseCommand`, ktorá bude zapúzdrovať všetky informácie potrebné pre jej opätovné vykonanie. Vykonanie príslušnej akcie používateľom bude mať za následok vytvorenie inštancie tejto triedy a jej poslanie do objektu typu `CCommandProcessor`, v ktorom sa daný príkaz spustí a pridá sa do zoznamu histórie. Používateľ môže efekt príkazu vrátiť späť, prípadne vykonať znova príslušnou `undo()`, resp. `redo()` metódou.

Inštancia `CCommandProcessor` sa v prípade UML .FRI volá `history` a je definovaná v triede `CfrmMain`. Tvorba inštancií jednotlivých príkazov sa v zásade uskutočňuje v jednotlivých častiach prezentačnej vrstvy aplikácie ako `CpicDrawingArea`, `CtwProjectView`, `ClwProperties`, atď. – čo je podrobnejšie zobrazené v prílohe A.

Napríklad, ak používateľ klikne v menu na položku Cut, teda vystrihnúť, vytvorí sa inštancia príkazu `CCutCmd` (výpis 6.3), `CpicDrawingArea` následne

```
from lib.Commands.ClipboardCommands import *
[...]

    @event("mnuCtxCut", "activate")
    def ActionCut(self, widget = None):
        cutCmd =  CCutCmd(self.Diagram, self.
           application.GetClipboard())
        self.emit('history-entry', cutCmd)

[...]
```

Výpis 6.3: *Ukážka použitia CCutCmd príkazu*

```
from lib.Commands import CCompositeCommand
from lib.Commands.AreaCommands import *
[...]
    groupCmd = CCompositeCommand()
    for sel in self.Diagram.GetSelected():
        deleteItem = CDeleteItemCmd(self.Diagram,
           sel)
        groupCmd.add(deleteItem)
    self.emit('history-entry',groupCmd)
[...]
```

Výpis 6.4: *Zoskupenie príkazov pri mazaní elementov z diagramu*

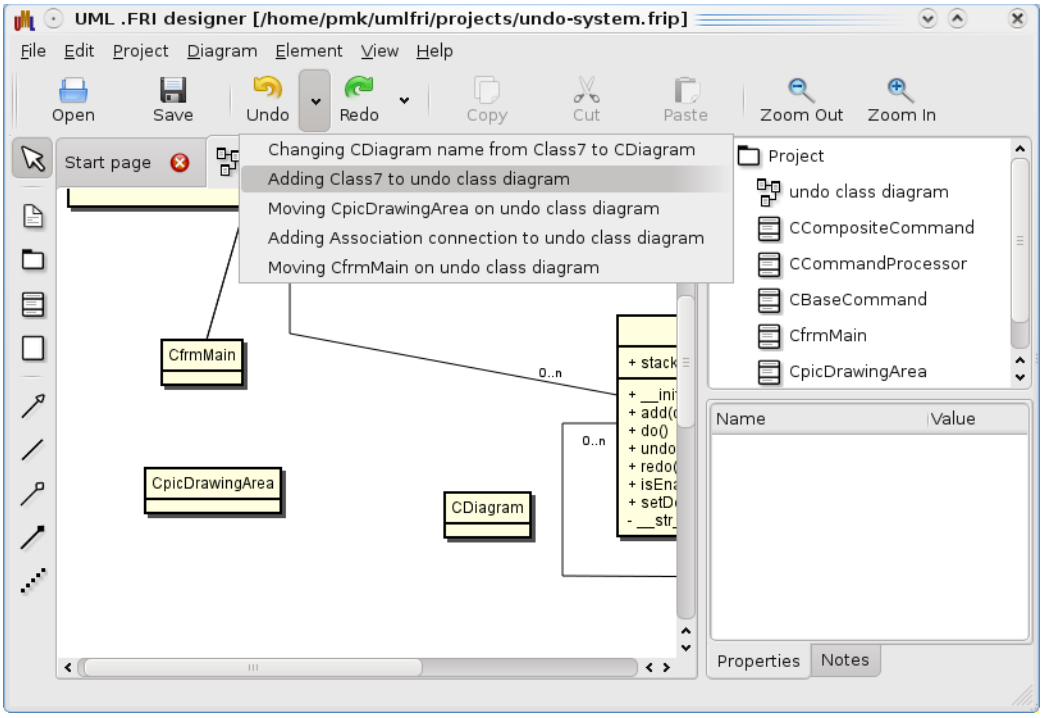
emituje history-entry signál pomocou `self.emit` [13]. Signál aj s objektom je odchytený v `CfrmMain` a pridaný do `history` objektu, kde sa príkaz vykoná pomocou jeho metódy `do()`. Ak vykonanie prebehne v poriadku, zaradí sa do undo zoznamu `undoStack`.

Spôsob vytvorenia príkazu na zmazanie elementu a využitie zloženého príkazu demonštruje výpis 6.4. Používateľ označil elementy, prípadne spojenia na kresliacej ploche a stlačením klávesy Delete ich chce zmazať – pre každý označený element sa vytvorí inštancia triedy `CDeleteItemCmd` – príkaz, ktorý daný element zmaže z diagramu. Príkazu sú predané informácie z ktorého diagramu má byť element, alebo spojenie zmazané, teda v prípade ukážky `self.Diagram` a aj ktorý objekt má byť z diagramu odstránený – `sel`. Jednotlivé inštancie sa zoskupia v zloženom príkaze groupCmd a pošlú sa do `history` objektu.

[13] GTK+ emit je vlastne implementácia Observer návrhového vzoru.

6.6 Vizualizácia

Vizualizácia predstavuje ovládacie prvky používateľského rozhrania na ovládanie funkcionality undo/redo systém v aplikácii a spôsob jeho prezentovania používateľovi. V zásade existujú dve časti GUI umožňujúce ovládanie systému: hlavné menu a lišta s nástrojmi.



Obrázok 6.3: *Vizualizácia undo systému v nástroji UML .FRI*

Pri ovládaní cez menu sa môže používateľ pohybovať v histórii dopredu a dozadu po jednej položke. Lišta s nástrojmi ponúka okrem rovnakej funkcionality, aj možnosť zobrazenia dodatočného menu s popismi jednotlivých akcií, ktoré je možné vrátiť späť, resp. dopredu. Po vybratí položky z tohto menu, bude iteračne vykonaná undo, resp. redo operácia až k tejto položke (vrátené vybratej položky). Popis jednotlivých príkazov vykonaných používateľom v menu undo a redo tlačidla na lište s nástrojmi má umožniť lepšiu prehľadnosť a vyššiu efektivitu práce s undo systémom a stala sa aj v ostatných aplikáciách štandardom. Zobrazenie menu s popismi ako aj celú aplikáciu prezentuje obrázok 6.3. UML .FRI nezobrazuje v týchto menu všetky položky histórie resp. redo zoznamu, ale vždy, z dôvodu prehľadnosti,

len počet stanovený v `lib/consts.py` premennou `STACK_SIZE_TO_SHOW`.

6.7 Budúcnosť undo systému

Cesta undo systému v aplikácií UML .FRI sa ešte neskončila. Implementácia undo systému neprebiehala v hlavnom repozitári(trunk), ale v undo vetve(branch). Do budúcnosti je plánované spojenie jednotlivých vetiev do hlavnej (merge). Taktiež napojenie undo systému na využívanie a spolupracovanie s pluginovacím systémom[14], ako aj ďalšia optimalizácia kódu jednotlivých príkazov, ako aj prepracovanie a využitie komplexnejšieho a jednotného spôsobu na obnovovanie používateľského rozhrania. Po umiestnení do hlavnej vývojovej vetvy bude systém vystavený väčšej používateľskej záťaži a bude intenzívnejšie testovaný.

[14] Ďalšia vývojová vetva aplikácie.

Kapitola 7

Záver

Ako sa v počítačovom svete začali postupne objavovať jednotlivé interaktívne používateľské nástroje, rástla aj potreba undo/redo systému. Dôvod bol jednoduchý: počítače mali uľahčovať prácu a možnosť opraviť predchádzajúcu chybu je veľmi pohodlná. Undo vyplnilo medzeru pri interakcii človeka s počítačom, pri ktorej sú chyby bežné a človek ich dokáže skoro odhaliť, ale odstránenie ich následkov bolo problematické. Keďže undo opravuje odhalené chyby, dobre zapadá s ľudskou schopnosťou samodetekcie chýb a upevňuje schopnosť človeka na ich opravu.

Postupne boli v práci navrhnuté rôzne prístupy a modely na implementovanie undo systému, odlišujúce sa jednak svojím prístupom k samotnému undo príkazu, deliace tak modely na primitívne a meta undo modely, ako aj v interpretácii histórie – lineárne, alebo sekvenčne. Mnohé z modelov sú však prevažne len teoretickou formuláciou a ich implementácia sa obmedzovala na pár aplikácií.

Pri analýze spôsobov implementácie undo systému som prezentoval v súčasnej dobe najpoužívanejšie a najosvedčenejšie postupy reprezentované jednotlivými návrhovými vzormi, jednak na implementáciu celého undo systému, prípadne jeho častí.

Ukázal som, že pri aplikáciách pre jedného používateľa je v prevažnej miere využívaný reštriktívny lineárny undo model, keďže ponúka dobre predvídateľné správanie a používatelia tento systém už poznajú. Pri výbere undo modelu pre aplikáciu UML .FRI som sa preto rozhodol pre túto variantu.

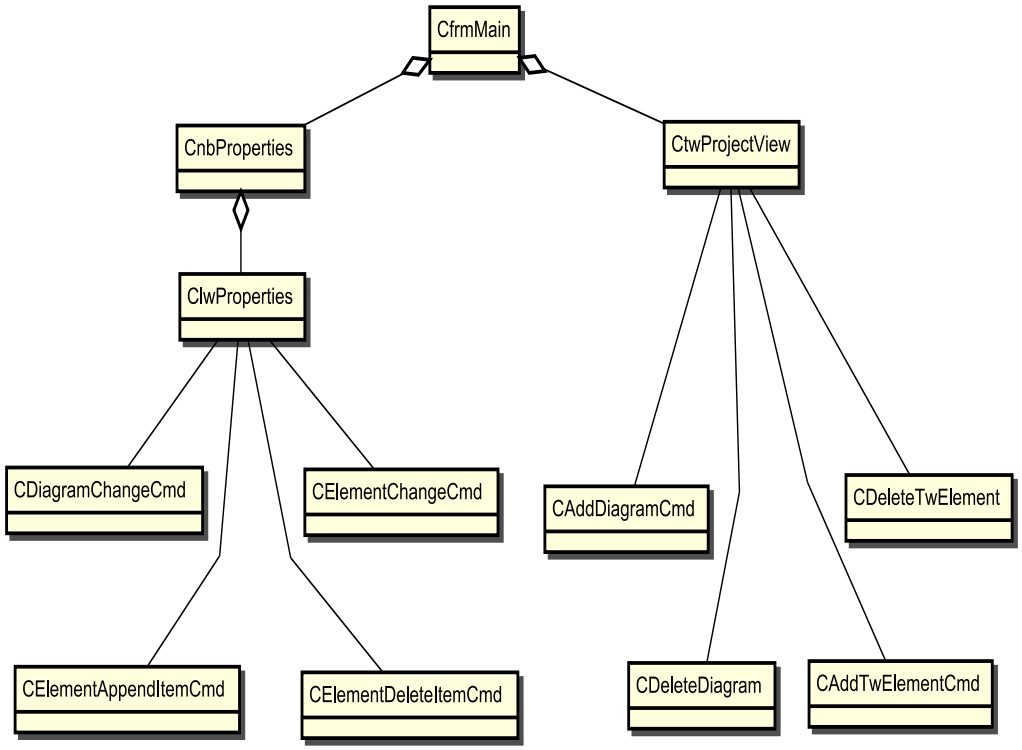
Implementácia prebehla za využitia návrhových vzorov v programovacom jazyku Python. Vo výsledku sú jednotlivé deštruktívne operácie používateľa zapúzdrené do inštancie príkazu, ktorý okrem informácií potrebných pre vykonanie operácie, má na používateľov podnet, schopnosť efekt tejto operácie vrátiť späť, prípadne znova obnoviť. Pre undo systém som pridal reštrikciu vo forme obmedzenia veľkosti histórie a vyprázdnenie redo zoznamu v prípade

vyvolania deštruktívnej operácie používateľom.

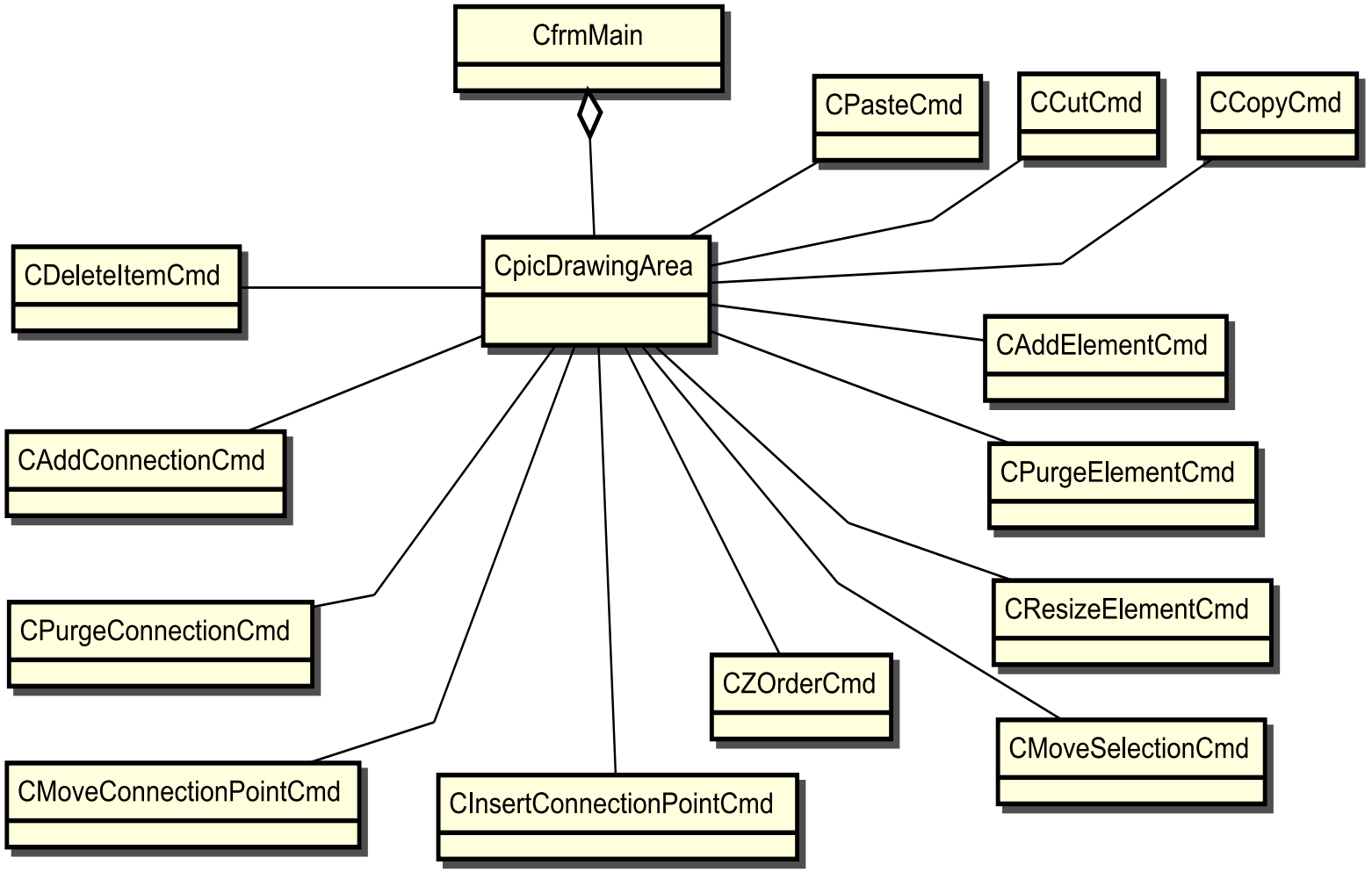
Výsledný undo/redo systém v aplikácií UML .FRI je pomerne jednoduchý a pre priemerne počítačovo gramotného používateľa by jeho efektívne používanie malo byť intuitívne a bezproblémové.

Dodatok A

UML diagramy



Obrázok A.1: *Zjednodušený UML model zobrazujúci vzťah komponentov GUI a príkazov (časť 1)*



Obrázok A.2: *Zjednodušený UML model zobrazujúci vzťah komponentov GUI a príkazov (časť 2)*

Dodatok B

Zoznam implementovaných príkazov

B.1 AreaCommands

Príkazy vytvárané pri práci s grafickou plochou. Cesta v súborovom strome aplikácie je `lib/Commands/AreaCommands/`

Trieda	Funkcia
`CAddConnectionCmd`	pridanie spojenia do projektu a na zobrazený diagram
`CAddElementCmd`	pridanie elementu do projektu, prípadne iba do zobrazeného diagramu
`CDeleteItemCmd`	zmazanie elementu alebo spojenia zo zobrazeného diagramu
`CInsertConnectionPointCmd`	pridanie bodu spojeniu
`CMoveConnectionPointCmd`	presun bodu spojenia na iné súradnice
`CMoveSelectionCmd`	presun označených elementov na inú pozíciu v zobrazenom diagrame
`CPurgeConnectionCmd`	zmazanie spojenia z projektu (aj z diagramov)
`CPurgeElementCmd`	zmazanie elementu z projektu (aj z diagramov)
`CResizeElemntCmd`	zmena veľkosti elementu
`CZOrderCmd`	zmena zoskupenia elementu na z-ose

B.2 ClipboardCommands

Príkazy pre prácu so schránkou UML .FRI aplikácie. Cesta v súborovom strome aplikácie je `lib/Commands/ClipboardCommands/`

Trieda	Funkcia
`CCopyCmd`	kopírovanie označených elementov na grafickej ploche
`CCutCmd`	vystrihnutie označených elementov na grafickej ploche
`CPasteCmd`	prilepenie skopírovaných alebo vystrihnutých elementov na iný diagram (diagram musí tieto elementy podporovať)

B.3 ProjectViewCommands

Príkazy pre prácu so stromom projektu. Cesta v súborovom strome aplikácie je `lib/Commands/ProjectViewCommands/`

Trieda	Funkcia
`CAddDiagramCmd`	pridanie nového diagramu do projektu
`CAddTwElementCmd`	pridanie nového elementu do projektu
`CDeleteDiagramCmd`	zmazanie diagramu z projektu
`CDeleteTwElementCmd`	zmazanie elementu z projektu

B.4 PropertiesCommands

Príkazy na zmenu vlastností jednotlivých označených elementov, spojení, diagramov. Cesta v súborovom strome aplikácie je `lib/Commands/Properties-Commands/`

Trieda	Funkcia
`CDiagramChangeCmd`	zmena vlastnosti označeného diagramu
`CElementAppendItemCmd`	pridanie prvku označenému elementu, napríklad pridanie atribútu, alebo metódy
`CElementChangeCmd`	zmena vlastnosti označeného elementu, spojenia; napríklad zmena mena
`CElementDeleteItemCmd`	odstránenie prvku označenému elementu, napríklad odstránenie atribútu, alebo metódy

Dodatok C

Zoznam použitých UML .FRI tried

Príloha obsahuje názvy a popis tried ako sú definované v [JBO09] (so zmenami berúcimi v úvahu undo/redo systém) použitých v texte, alebo inak relevantných k popisovanej problematike undo systému[1].

C.1 Prezentačná vrstva

Prezentačná vrstva, resp. jednotlivé okná a ich komponenty relevantné k undo/redo systému.

Trieda	Funkcia
`CfrmMain`	reprezentuje hlavné okno. Vzhľadom na to, že je rozdelené na komponenty, v tejto triede je vykonávaných minimum akcií. Slúži teda skôr ako ich kontainer a zabezpečuje komunikáciu medzi nimi.
`CmnuItems`	reprezentuje položky hlavného menu *Project/Add diagram* a *Project/Add element*.
`CTabs`	obsluhuje záložky. Stará sa o ich otváranie, zatváranie, prepínanie, presúvanie a podobne.

[1] Ide len o výber tried, pre kompletný zoznam je potrebné pozrieť API dokumentáciu nástroja UML .FRI, dostupnú napríklad cez internetovú stránku projektu.

Trieda	Funkcia
`CpicDrawingArea`	stará sa o interakciu používateľa s diagramom. Teda obsluhuje udalosti od používateľa, spracováva ich a predáva ďalej. Následne sa postará o zobrazenie prekresleného diagramu. Iným slovom je to kresliaca plocha.
`CtbToolBox`	nástrojová lišta na kreslenie diagramu. Obsahuje nástroj „označovanie" a zoznam elementov a spojení, ktoré je možné pridávať na aktuálny diagram. Pri zmene tohoto prepínača komponent sám nič nevykonáva, iba zmenu oznámi hlavnému oknu pomocou signálu `toggled`.
`CtwProjectView`	je strom projektu. Na požiadanie sa dokáže zosynchronizovať so stromom projektu z logickej vrstvy. Komponent obsluhuje aj kontextové menu pre jednotlivé položky stromu. Sám vykonáva všetky akcie, ktoré dokáže spracovať. Ostatné, ako otvorenie diagramu, vytvorenie diagramu a podobne posiela pomocou signálov (`selected-diagram`). hlavnému oknu.
`CnbProperties`	reprezentuje panel na nastavovanie vlastností so záložkami *Properties* a *Notes*. Každá táto záložka je reprezentovaná samostatným komponentom.
`ClwProperties`	zoznam vlastností elementu (v žargóne UML .FRI atribútov) na prvej záložke je reprezentovaný ovládacím prvkom *strom* (*gtk.TreeView*). Komponent `ClwProperties` sa stará o jeho obsah a editáciu hodnôt.
`CfrmProperties`	zobrazí okno so zoznamom spojení daného elementu a ponúka možnosť ich zobrazenia, alebo odstránenia z diagramu .

C.2 Logická vrstva

Logická vrstva sa v aplikácií UML .FRI delí na logickú a vizuálnu časť. Pri popise triedy je uvedené do ktorej časti daná trieda patrí.

Trieda	Funkcia
`CDiagram`	reprezentuje diagram projektu, okrem iného obsahuje zoznam elementov a spojení, ktoré sa v danom diagrame nachádzajú, tak ako aj metódy na ich pridávanie, odstránenie z diagramu, alebo zmenu ich pozície v rámci diagramu. Patrí do vizuálnej časti logickej vrstvy
`CConnection`	grafická reprezentácia spojenia medzi dvoma elementami na danom diagrame
`CConnectionObject`	logická reprezentácia spojenia: logická reprezentácia spojenia je v projekte len jedna, ale môže mať viac grafických reprezentácií `CConnection`.
`CElement`	grafická reprezentácia elementu na diagrame
`CElementObject`	logická reprezentácia elementu
`CProject`	logická reprezentácia projektu, obsahuje metamodel, metódy na uloženie, načítanie projektu a prácu s uzlami v strome projektu `CProjectNode`

Referencie

[AD92] Abowd, Gregory D. – Dix, Alan J.: *Giving undo attention.* [online]. 1992.

[ACS84] Archer, James E. – Conway, Richard – Schneider, Fred B.: *User Recovery and Reversal in Interactive Systems.* [ACM Transactions on Programming Languages and Systems]. 1984.

[BER94] Berlage, Thomas: *A Selective Undo Mechanism for Graphical User Interfaces Based On Command Objects.* 1994.

[BJO05] Bjorklund, Kaj: *A Serialization Library with Undo Support.* [online]. 2005.
Dostupné online: <http://www.iki.fi/kbjorklu/mthesis/>.

[BRO03] Brown, Aaron: *A Recovery-Oriented Approach to Dependable Services: Repairing Past Errors with System-Wide Undo.* [online]. 2003.
Dostupné online: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2004/CSD-04-1304.pdf>.

[CF05] Cass, Aaron G. – Fernandes, Chris S. T.: *Modeling Dependencies for Cascading Selective Undo.* [online]. 2005.
Dostupné online: <http://cs.union.edu/ fernandc/pub/ifip05.pdf>.

[DML96] Dix, Alan J. – Mancini, Roberta – Levialdi, Stefano: *Reflections on Undo.* [online]. 1996.

[DVO03] Dvořák, M.: *Composite Pattern.* [online]. 2003.
Dostupné online: <http://objekty.vse.cz/Objekty/Vzory-Composite>.

[DVO03] —: *Memento Pattern.* [online]. 2003.
Dostupné online: <http://objekty.vse.cz/Objekty/Vzory-Memento>.

[EM97] Edwards, Keith W. – Mynatt, Elizabeth D.: *Timewarp: Techniques for Autonomous Collaboration.* [online]. 1997. Dostupné online: <http://www2.parc.com/csl/members/kedwards/pubs/autonomous.pdf>.

[JBO09] Janech, Ján – Bača, Tomáš – Odlevák, Pavol a i.: *Programová podpora modelovania IS.* [Interná projektová dokumentácia]. 2009.

[MOO08] Moolenaar, Bram: *VIM Reference Manual.* 2008.

[OGP03] Oppenheimer, David – Ganapathi, Archana – Patterson, David A.: *Why do Internet services fail, and what can be done about it.* [Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems(USITS' 03). Seattle, WA]. 2003. Dostupné online: <http://www.usenix.org/event/usits03/tech/full_papers/oppenheimer/oppenheimer_html/index.html>.

[PEH00] Pehlivan, Huseyin: *A Sophisticated Shell Environment.* 2000.

[PK94] Prakash, Atul – Knister, Michael J.: *A Framework for Undoing Actions in Collaborative Systems.* 1994.

[QRD09] *Qt Reference Documentation.* [online]. 2009. Dostupné online: <http://doc.trolltech.com/4.5/qundo.html>.

[RAY99] Raymond, Eric S.: *The Cathedral and the Bazaar.* 1999.

[RUZ08] Ružbarský, Ján: *Základy UML.* [online]. 2008. Dostupné online: <http://kst.uniza.sk/predmety/uml/>.

[STA07] Stallman, Richard: *GNU Emacs manual.* [online]. 2007. prístup 7-máj-2009. Dostupné online: <http://www.gnu.org/software/emacs/manual/emacs.html>.

[VIT84] Vitter, J.S.: *USR A new framework for redoing.* 1984.

[WIK09] Wikipédia: *Berkeley Software Distribution.* [online]. 2009. prístup 10-máj-2009. Dostupné online: <http://en.wikipedia.org/wiki/Bsd>.

[WIK09] —: *Command pattern.* [online]. 2009. prístup 5-máj-2009. Dostupné online: <http://en.wikipedia.org/wiki/Command_pattern>.

[WIK09] —: *ENIAC*. [online]. 2009. prístup 1-máj-2009.
Dostupné online: <http://en.wikipedia.org/wiki/Eniac>.

[WIK09] —: *Free Software Foundation*. [online]. 2009. prístup 10-máj-2009.
Dostupné online: <http://en.wikipedia.org/wiki/Free_Software_Foundation>.

[WIK09] —: *Qt*. [online]. 2009. prístup 3-máj-2009.
Dostupné online: <http://en.wikipedia.org/wiki/Qt_toolkit)>.

[YAN88] Yang, Y.: *Undo support models. International Journal of Man Machine Studies*. 1988.

[YW90] Young, R. – Whittington, J.: *Using a knowledge analysis to predict conceptual errors in text-editor usage.*. 1990.