PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

DIPLOMOVÁ PRÁCE

Hranice prototypových programovacích jazyků

2011 Jan Laštovička

Anotace

Pojem třídy bývá prohlašován jako základní kámen objektového programování. Co se stane, když třídy odebereme? Jedná se i pak o objektové programování? Nejsou možnosti jazyků bez tříd menší? Práce hledá odpověď na tyto otázky.

Děkuji především vedoucímu diplomové práce
Doc. RNDr. Michalu Krupkovi, Ph.D.
Dále děkuji
Heleně a Ladislavu Laštovičkovi,
Heleně, Vladimírovi a Monice Karlů,
Lise Ahner,
Lucii Vaverové,
Veronice Valáškové,
Janu Studničkovi
Vandě Laštovičkové.

Obsah

Seznam obrázků

1. Úvod

Práce navazuje na autorovu bakalářskou práci, ve které byl vytvořen prototypový objektový jazyk Dolly. Jazyk Eggs představený v této práci je nástupcem jazyka Dolly.

Jazyky založené na třídách tvoří velkou část objektově orientovaných jazyků. Pojem třídy v nich hraje zásadní roli. Každý objekt v nich musí být instancí třídy. Kromě jazyků založených na třídách do objektových jazyků patří malá skupina jazyků, kde pojem třídy neexistuje. Jejich objekty nejsou instancemi tříd. Tyto jazyky nazýváme prototypové.

Budeme předpokládat, že čtenář zná objektové programování založené na třídách. Na tento předpoklad se budeme odvolávat při dalším vymezování prototypového přístupu.

Následuje shrnutí cílů práce a její obsah.

Účel práce Hlavním účelem práce bylo prozkoumat hranice prototypových jazyků. Za tímto účelem byly stanoveny následující cíle:

- Vytvořit úvod do problematiky prototypových jazyků.
- Srovnat prototypové jazyky s jazyky založenými na třídách.
- Popsat hlavní rysy vybraných prototypových jazyků.
- Za použití zkušeností z prostudovaných jazyků navrhnout vlastní prototypový jazyk.
- Vytvořit kompilátor navrhnutého jazyka a vývojové prostředí pro něj.
- Pokusit se vytvořit netriviální aplikaci ve vývojovém prostředí navrhnutého prototypového jazyka.
- Použít zkušeností při tvorbě netriviální aplikace a poznatků z prostudovaných prototypových jazyků v diskusi o možnostech prototypového přístupu.

Obsah práce Tato diplomová práce se skládá z programové a teoretické části. Do programové části patří kompilátor prototypového objektového jazyka Eggs, vývojové prostředí pro tento jazyk a malá textová hra napsaná v jazyce Eggs za použití vývojového prostředí. Programová část se nachází v adresáři `src` na přiloženém CD.

Teoretická část je rozdělena do kapitol. Po úvodní kapitole 1. následuje kapitola 2. popisující základní myšlenky prototypových objektových jazyků a jejich rozdíly oproti jazykům založeným na třídách. Kapitola 3. srovnává prototypové jazyky a jazyky založené na třídách. Jsou zde uvedeny výhody a nevýhody obou

jazykových rodin. Příklady několika významných prototypových jazyků čekají na prostudování v kapitole 4. Popis vlastního prototypového jazyka Eggs spolu s ukázkami jeho použití se nalézá v kapitole 5. Tento jazyk je možné si vyzkoušet ve vývojovém prostředí. Základy práce s vývojovým prostředím objasňuje kapitola 6. Součástí programové části je i malá textová hra odehrávající se v daleké budoucnosti. Pravidla hry vysvětluje kapitola 7. Tato kapitola také popisuje vytvoření hry v jazyce Eggs. Možnosti použití prototypových jazyků a kde se nachází jejich hranice stanovuje kapitola 8. Obsah přiloženého CD se nalézá v příloze A.

2. Prototypové objektové jazyky

Kapitola popisuje základní myšlenky prototypových jazyků a poukazuje na rozdíly oproti jazykům založeným na třídách.

Nejprve se kapitola zabývá podstatou objektů v prototypových jazycích. Dále načrtne mechanizmus posílání zpráv. Poté se zabývá odlišností vývojových prostředí pro prototypové jazyky. Následuje část o dědičnosti dosažené pomocí delegování a využití delegace k reprezentaci konceptů. Za ní se nachází krátký popis psychologické teorie, která souvisí s prototypovými jazyky. Poslední slova kapitoly patří obecně objektovým jazykům.

V této kapitole se vyskytují ukázky z prototypového jazyka Self. Jejich pochopení necháváme na intuici čtenáře. Tímto jazykem se podrobněji zabývá podkapitola 4.1.

Čistě objektový jazyk musí splňovat dvě následující podmínky. Všechna data, která se mohou v jazyce vyskytnout, jsou objekty ve smyslu objektového programování. Posílání zpráv je jedinou výpočetní operací.

2.1. Objekty bez tříd

V jazycích založených na třídách jsou objekty instancemi tříd. Třída definuje strukturu a chování svých instancí. Objekt v prototypovém jazyku má také určenou strukturu a chování. Protože objekt v prototypových jazycích není instancí třídy, je jeho struktura a chování definována přímo pro něj.

Bod v rovině modelujeme objektem, který obsahuje informace o jeho souřadnicích. To je struktura objektu. Bodu lze přikázat, aby se posunul o zadaný vektor. To patří k jeho chování. V prototypových jazycích lze takový bod vytvořit z ničeho. Stačí jen popsat jeho strukturu a chování. Na formu, jakou se bodu posunutí přikáže, se podíváme později v části o posílání zpráv.

Objekty prototypových jazyků často bývají tvořeny sloty. *Slot* se skládá z jména slotu a hodnoty slotu. Jméno slotu je řetězec. Hodnotou slotu může být jakýkoli objekt.

Příklad kódu, který vytvoří objekt o dvou slotech:

```
( | x = 4 . y = 17 | )
```

Metoda je objekt obsahující kód jazyka. Objekty, které nejsou metodami, nazýváme *datové objekty*. Například číslo, nebo grafické okno jsou datové objekty.

Je-li ve slotu uložen datový objekt, mluvíme o *atributu objektu*.

Například objekt

```
( | name = "Quark" | )
```

má atribut `name`.

Ukázka metody:

```
( :n | deposit: n |)
```

Tvoření nových objektů Nové objekty se v jazycích založených na třídách tvoří instanciací třídy. Z důvodu absence třídy není tento způsob tvoření nových objektů v prototypových jazycích možný. V prototypových jazycích můžeme nový objekt vytvořit z ničeho. Další možnosti jsou rozšíření a klonování existujícího objektu.

Následuje popis klonování a tvoření objektů z ničeho. Na tvoření objektů rozšířením se podíváme později v části zabývající se delegací.

Kopírování objektů vzhledem k biologické analogii nazýváme *klonováním*. Při potřebě nového objektu vybereme existující objekt, který je podobný naší představě. Tento objekt naklonujeme (zkopírujeme). Díky kopírování bude mít nově vytvořený objekt (*klon*) stejnou strukturu i chování jako jeho vzor. Klon objektu můžeme dále měnit tak, aby vyhovoval naší představě.

Představme si, že chceme modelovat barevný bod. Podívejme se jak postupujeme v případě prototypových jazyků. Vezmeme existující bod. Vyrobíme jeho klon. Tím dostaneme nový objekt se strukturou i chováním bodu. Přidáme do klonu vlastnost barva, kterou nastavíme na hodnotu červená. Dále můžeme přidat schopnost na požádání měnit barvu na zadanou hodnotu. Změna struktury i chování klonu vedla k vytvoření barevného bodu.

Může se stát, že nenacházíme vhodný objekt pro klonování. Například proto, že naše představa o novém objektu je velmi vzdálená od existujících objektů. Proto umožňují prototypové jazyky tvořit *objekt z ničeho*. Stačí popsat strukturu a chování nového objektu.

Například pracovníka můžeme modelovat objektem, který bude uchovávat jeho jméno a zdraví. Ukázka kódu jazyka Self, který takový objekt vytvoří:

```
(| name = 'Karel' . health = 0.5 |)
```

2.2. Posílání zpráv

Společným rysem prototypových jazyků a jazyků založených na třídách je zásadní postavení výpočetního mechanizmu posílání zpráv.

Mechanizmus posílání zpráv má v prototypových jazycích významnou roli. Často je jediným primitivním výpočetním mechanizmem jazyka.

Podkapitola vymezuje základní pojmy použité při posílání zpráv a také popisuje jeho základní průběh. Jednotlivé prototypové jazyky průběh posílání zpráv často mění.

Základní pojmy Na objekty pohlížíme jako na uzavřené prvky jazyka. *Posílání zpráv* objektu je jedinou možností jak s ním komunikovat. Objektu, kterému zprávu posíláme, se říká *příjemce zprávy*. Poslaná zpráva má své jméno, například `factorial`.

Posílání zpráv lze využít k zadávání příkazů objektům. Například pro zavření grafického okna mu pošleme zprávu `destroy`.

Poslání zprávy *vrací hodnotu*. Vrácená hodnota je objekt. Tím je možno pokládat objektům otázky. Odpověď obdržíme formou vrácené hodnoty. Například poslání zprávy `factorial` číslu 3 vrátí hodnotu 6. Takto také lze získávat části složeného objektu. Poslání zprávy `first` seznamu vrátí jako hodnotu jeho první prvek.

Zpráva může být poslána společně s dalšími objekty, které nazýváme *argumenty zprávy*. Například součet čísel 3 a 5 dostaneme posláním zprávy `+` číslu 3 s argumentem 5. Vrácenou hodnotou bude číslo 8.

Objektu mohou být poslány jen některé zprávy. Říkáme, že těmto zprávám objekt *rozumí*. Všechny zprávy, kterým objekt rozumí, tvoří *rozhraní* objektu.

Základní průběh Posílání zpráv se skládá ze dvou částí: *hledání obsluhy zprávy* a *vyhodnocení obsluhy*. Nejprve se hledá obsluha zprávy. Obsluha je objektem. Pokud je hledání úspěšné, je nalezená obsluha vyhodnocena.

Může dojít k nalezení více obsluh poslané zprávy. V takovém případě říkáme, že byla objektu poslána *nejednoznačná zpráva* a poslání zprávy skončí chybou. V případě, že nebyla nalezena žádná obsluha, říkáme, že příjemce poslané zprávě *nerozumí* a poslání zprávy skončí chybou.

Má-li příjemce zprávy slot stejného jména jako je jméno zprávy, pak je hodnota slotu hledanou obsluhou zprávy. Toto platí pouze v případě, je-li objekt reprezentován sloty.

Vyhodnocení obsluhy se liší podle toho, zda je obsluha metodou nebo datovým objektem. Datový objekt se vyhodnocuje sám na sebe. Pokud má bod své souřadnice uloženy ve slotech `x` a `y`, lze tyto souřadnice získat posláním zpráv stejného jména. Například poslání zprávy `x` bodu o souřadnicích $[1, 2]$ vede k nalezení obsluhy 1. Protože číslo je datovým objektem, bude jednička i vrácenou hodnotou.

Metoda je *vyhodnocena* tak, že se *vyhodnotí* její kód a vrátí se hodnota kódu. Metoda má při vyhodnocování k dispozici příjemce i argumenty zprávy. Při imperativním charakteru kódu metody budeme mluvit o *vykonání kódu metody*.

Někdy také místo vyhodnocení metody budeme používat termínu *zavolání metody*. Pošleme například zprávu `rotate` s argumentem 30 bodu v rovině. Obsluha této zprávy bude metoda. Vykonání jejího kódu otočí tímto bodem.

Posílání zprávy zajišťuje jak čtení obsahu slotů, tak i volání metod. Vrácená hodnota poslané zprávy může být počítána metodou nebo pouze obsahem příslušného slotu. Například poslání zprávy `x` objektu vrátí číslo 5. Bez nahlížení do vnitřní struktury objektu nemůžeme poznat, zda číslo 5 bylo uloženo ve slotu `x` nebo zda ve slotu `x` byla metoda, která se vyhodnotila na číslo 5.

2.3. Vývojová prostředí

Prototypové jazyky často bývají dynamické a perzistentní. Podkapitola se vztahuje pouze k těmto jazykům.

Dynamičnost jazyka umožňuje za běhu programu prohlížet jeho strukturu a měnit ji. V případě dynamických prototypových jazyků to znamená možnost nahlížet do vnitřní struktury objektů a neomezeně ji měnit. Změnou struktury lze změnit i chování objektu.

Například lze nejen libovolně měnit hodnoty slotů, ale i přidávat a odebírat sloty z objektu nebo měnit kód metod.

Upozorněme, že dynamický jazyk nemusí být dynamicky typovaný. Například jazyk Omega popsaný v podkapitole 4.4. je dynamický jazyk se statickým typovaným systémem.

Jazyky, které umožňují objektům pokračovat v existenci i po ukončení programu, se nazývají *perzistentní*. Díky perzistenci jazyka může vývojové prostředí uložit změny v programu způsobené programátorem. Všechny existující objekty a jejich vzájemné vztahy tvoří *svět*. Svět má jeden kořenový objekt. V jazyce Self je tento objekt nazýván Lobby.

Často je vývojové prostředí vytvořené v tomtéž jazyce, pro který je určeno. Poté vývojové prostředí bývá propojené s tvořenou aplikací. To má za důsledek, že se ve světě mísí vývojové prostředí s tvořenou aplikací.

Vývojové prostředí dokáže zviditelňovat strukturu objektů tvořících aplikaci a nabízí nástroje jak ji měnit. Tím programátor tvoří.

Vývojové prostředí umožňuje perzistenci světa. Například uložením světa na disk. Uloženému světu se říká *image* nebo *snapshot* (česky momentka). To zavádí nový styl programování. Tradičně je programování spjato s psaním zdrojových kódů, které jsou následně přeloženy a spuštěny. Tradiční styl je označován jako *source-based*. Nový styl spočívá ve změnách běžícího programu a možnosti jeho ukládání. Novému stylu se říká *image-based*.

V podkapitole 4.1. můžeme najít část o vývojovém prostředí jazyka Self. Nachází se zde i obrázek 3., který jej zachycuje.

Podotkněme, že tento styl programování se nevyskytuje pouze u prototypových jazyků. Lze jej najít i u jazyka Smalltalk, který je založen na třídách. Zvláštní pozornost si zasluhuje jeho implementace Squeak.

2.4. Delegování

Pro podporu sdílení mezi objekty zavádí prototypové jazyky *delegaci*. Delegace je silnější prostředek než dědičnost používaná v jazycích založených na třídách. Dědičnost lze zavádět pouze mezi třídami, tedy abstraktními pojmy. Oproti tomu delegace funguje mezi jakýmikoli objekty, a to jak objekty zastupujícími konkrétní prvky, tak objekty reprezentujícími abstrakce.

Budeme-li mluvit o dědičnosti v souvislosti s prototypovými jazyky, budeme mít na mysli delegaci.

Některé jazyky, jako například Kevo a Omega, využívají jiné způsoby sdílení než delegaci. Více o těchto jazycích se lze dovědět v kapitole 4.

Mezi objekty se zavádí vztah *delegování*. Pokud první objekt deleguje na druhý objekt, pak říkáme, že první je *přímým potomkem* druhého, nebo že druhý objekt je *rodičem* prvního. Přímý potomek objektu se také nazývá *dítě*.U některých jazyků může mít objekt více rodičů.

Zavedeme vztah potomka objektu: Dítě objektu je jeho *potomkem*. Dítě potomka objektu je jeho *potomkem*. Obdobně zavedeme i vztah předka objektu: Rodič objektu je jeho *předek*. Rodič předka objektu je jeho *předkem*.

Uvažujeme-li, že tlačítko deleguje na obdélník a obdélník na grafický objekt, pak obdélník je rodičem tlačítka a potomkem grafického objektu. Dále grafický objekt je předkem tlačítka. Neboli tlačítko je potomkem grafického objektu.

Posílání zpráv využívá delegaci. Delegace se uplatňuje ve fázi hledání obsluhy zprávy následujícím způsobem. Pokud v objektu nebyla nalezena obsluha zprávy, pak hledání pokračuje v jeho rodičích. Říkáme, že objekt *deleguje zprávy* na své rodiče.

Objekty spolu se vztahem delegace tvoří orientovaný graf, kterému se říká *graf delegování*. U některých jazyků může graf delegování obsahovat cykly.

Cyklů v grafu delegování se využívá například mezi obecným objektem a kořenovým objektem světa. Kořenový objekt světa je potomkem obecného objektu. Kořenový objekt světa uchovává důležité objekty, jakými jsou prototypy. Z důvodu přístupu k těmto prototypům je nezbytné, aby obecný objekt delegoval zprávy na kořenový objekt.

Rozšíření objektu Jazyky podporující delegaci umožňují další způsob vytváření nových objektů, nazývaný *rozšíření*. Při rozšíření objektu se vytvoří nový objekt, který bude mít za rodiče objekt, jenž byl rozšířen. Takto vytvořenému objektu se mohou přidat další vlastnosti.

Například barevný bod může vzniknout rozšířením bodu a jeho obohacením o vlastnost barva.

2.5. Koncepty

V této podkapitole budeme chápat „koncept“ jako synonymum pro „pojem“. Dalším jeho možným označením je „kategorie“.

Jazyky založené na třídách používají třídy k zachycení konceptů. Prototypové jazyky místo tříd používají k zachycení konceptů prototypy a sdílená úložiště.

Sdílená úložiště použijeme v případě, že koncept nemá žádného typického představitele. Například koncept čísla. Naopak pokud koncept má typického představitele, pak jej lze tímto představitelem reprezentovat. Jinými slovy jej můžeme ustanovit prototypem konceptu. Zbytek podkapitoly se zabývá prototypy a sdílenými úložišti.

Prototypy Prototyp je objekt, který reprezentuje koncept. Například prototypem konceptu grafické okno je konkrétní prázdné grafické okno.

Prototypy umožňují sdílení atributů a metod mezi objekty konceptu, který reprezentují.

Ostatní objekty konceptu jsou potomky prototypu. Prototyp se kromě této vlastnosti nijak neliší od ostatních objektů konceptu.

Koncept tlačítko můžeme reprezentovat konkrétním tlačítkem s nápisem „Text“. To je prototypem konceptu. Tlačítko s nápisem „Stiskni mě!“ vytvoříme rozšířením tohoto prototypu a nastavením nápisu. Ostatní vlastnosti nové tlačítko zdědí od svého prototypu.

Objekt konceptu může být prototypem jeho podkonceptu. Takto mohou prototypy tvořit hierarchii, která se kryje s hierarchií konceptů, které zastupují. Objekty, které nejsou prototypy, tvoří v této hierarchii nejnižší patro.

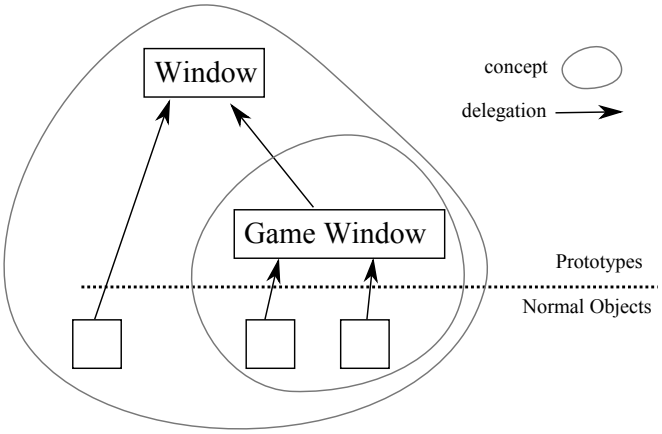
Představme si malý příklad. Grafické okno (Window) je prototypem všech grafických oken. Mezi nimi je i grafické okno Hra (Game Window). To je potomkem grafického okna a současně prototypem dvou oken. Nakonec tu je grafické okno, které není potomkem okna Hra. Tuto situaci shrnuje obrázek 1. Objekty patřící do konceptů grafické okno a herní okno jsou vymezeny na obrázku šedou čárou.

Změníme-li prototyp konceptu, může tato změna ovlivnit díky delegaci ostatní objekty konceptu.

Sdílená úložiště Existují koncepty, které nemají žádného typického představitele. Například čísla.

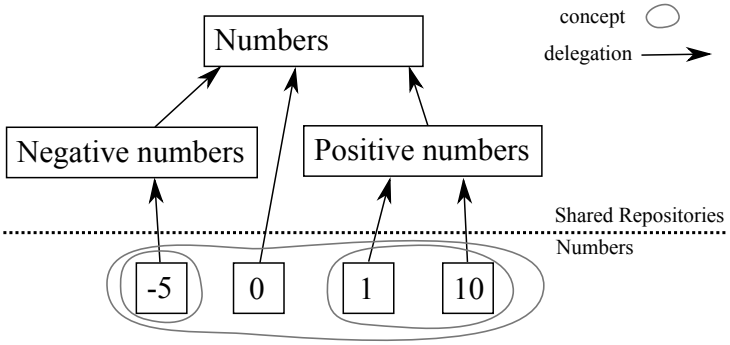
Sdílená úložiště reprezentují koncepty, ale sama do tohoto konceptu nepatří. Sdílená úložiště uchovávají atributy a metody společné všem objektům konceptu, který reprezentují. Objekty patřící do konceptu reprezentovaného sdíleným úložištěm jsou potomky tohoto sdíleného úložiště. Například sdílené úložiště pro čísla uchovává metodu sčítající čísla. Čísla jsou potomky tohoto sdíleného úložiště.

Potomkem sdíleného úložiště může být sdílené úložiště, které zastupuje jeho podkoncept. Sdílená úložiště tak mohou tvořit hierarchii, která se kryje s hierarchií konceptů, které reprezentují.



Obrázek 1. Prototypy

Kladná a záporná čísla mohou mít svá sdílená úložiště. Tato úložiště budou přímými potomky úložiště pro čísla. Toto řešení je ukázané na obrázku 2.



Obrázek 2. Sdílená úložiště

Objekt může patřit do více konceptů, které nejsou vzájemně podkoncepty. Proto objekt může být přímým potomkem několika sdílených úložišť. Objekt také může být přímým potomkem jak sdílených úložišť, tak i prototypů.

2.6. Prototypová teorie

Podkapitola shrnuje hlavní myšlenky prototypové teorie, která je filosofickým pozadím prototypových jazyků. Myšlenky byly čerpány z [8].

V polovině sedmdesátých let vznikla v kognitivní psychologii *prototypová teorie*. Některé její poznatky jsou důležité pro prototypové jazyky. Postupně si je představíme.

- Některé prvky konceptu jsou jeho lepšími příklady než jiné. Dobrým příkladem konceptu nábytek je židle. Už méně dobrým příkladem je komoda.

- Neexistuje ideální objektivní systém konceptů. Vytváření systému konceptů je subjektivní činnost. Například lidé žijící blízko rovníku nejsou schopni rozlišit mezi ledem a sněhem. Oproti nim Eskymáci mají mnoho slov pro popis různých druhů sněhu a ledu.

- Koncepty nejsou uspořádané čistě hierarchicky. Některé koncepty leží ve více nadkonceptech. Například koncept rozbité auto leží v nadkonceptech auto a vrak.

- Lidé často klasifikují věci jen pro nějaký okamžitý účel. Například koncept věcí, které jsou vhodné jako narozeninový dárek.

Prototypová teorie kritizuje některé důsledky klasické klasifikace, jak ji zformuloval Platón. Podle Platóna jsou koncepty vymezeny pouze vlastnostmi sdílenými všemi prvky konceptu. Podotkněme, že na této klasické teorii stojí objektové jazyky používající třídy. Kritika se týká zejména těchto dvou důsledků klasické teorie:

- Koncept podle klasické teorie je vymezen pouze vlastnostmi sdílenými všemi objekty konceptu. Z toho vyplývá, že všechny prvky tohoto konceptu jsou jeho stejně dobrými představiteli. V prototypové teorii existují koncepty, které mají nejlepšího představitele.

- Je-li koncept vymezen pouze sdílenými vlastnostmi, pak je nezávislý na subjektu, který klasifikaci provádí, neboli je objektivní. Naproti tomu v prototypové teorii je klasifikace závislá zejména na schopnostech a vědomostech jedince, který ji provádí.

2.7. Objektové jazyky

Při návrhu programu programátor vytváří konkrétní objekty a abstrakce. Podle toho, jaká z těchto dvou aktivit v jeho činnosti převažuje, mluvíme o programování zaměřeném na objekty, respektive na abstrakce.

Prototypové programování bez delegace neumožňuje vytvářet abstraktní objekty, proto se jedná o programování extrémně zaměřené na objekty.

V programování založeném na třídách musí každý objekt být instancí třídy. Protože instance je konkrétní objekt a třída abstrakce, dostáváme, že před vytvořením konkrétního objektu musí být vytvořena jeho abstrakce. Proto tento styl řadíme mezi programování založené na abstrakcích.

Prototypové programování s delegací leží mezi těmito dvěma extrémy.

3. Výhody a nevýhody prototypových jazyků

Kapitola porovnává prototypové jazyky s jazyky, které jsou založené na třídách. Nejprve jsou uvedeny výhody prototypových jazyků. Za nimi následují jejich nevýhody. Zajímavou kritiku prototypových jazyků nabízí doslov knihy [12].

3.1. Výhody prototypových jazyků

Jednoduchost návrhu Díky nepřítomnosti tříd je návrh prototypových jazyků jednodušší. Důsledkem toho bývají prototypové jazyky lehčí na pochopení a jejich implementace bývá snadnější.

Objekty konceptu s odlišnou strukturou nebo chováním Všechny instance určité třídy musí mít stejnou strukturu a chování. Připomeňme, že třídy zastupují koncepty (pojmy). V konceptu se mohou vyskytovat objekty s nepatrně odlišnou strukturou nebo chováním. V jazycích založených na třídách se pro tyto objekty musí zakládat speciální podtřídy.

V prototypových jazycích lze tyto objekty s nepatrnou odlišností jednoduše modelovat. Uvědomme si, že prototyp konceptu nemusí mít stejnou strukturu a chování jako ostatní objekty konceptu. Odlišnost objektu konceptu můžeme definovat přímo pro něj.

Nepřítomnost metaobjektů Každý objekt v objektových jazycích založených na třídách musí být instancí nějaké třídy. V některých jazycích jsou třídy také objekty a musí tedy být instancemi určité třídy. Těmto třídám se říká *metatřídy*. Protože metatřída je třídou a třídy jsou objekty, musí být metatřída opět instancí některé třídy. Takto vznikající řetězec metatříd lze ukončit tak, že nejobecnější metatřídu prohlásíme za instanci sama sebe.

Z nepřítomnosti tříd v prototypových jazycích plyne i nepřítomnost metatříd. Díky tomu se prototypové jazyky vyhýbají složitostem, které s nimi souvisejí.

Singletony Singleton je koncept obsahující jediný objekt. V objektových jazycích založených na třídách musí mít singleton třídu. U této třídy by mělo být zabezpečeno, aby bylo možno vytvořit pouze jednu její instanci. V prototypových jazycích lze singleton vytvořit přímo popsáním jeho vlastností.

Abstrakce v pozdějších fázích V prototypových jazycích může programátor nejprve pracovat s konkrétními objekty. Vytváření abstraktních objektů (konceptů a sdílených úložišť) může být odsunuto na pozdější fázi vývoje aplikace.

Respektuje se tak přirozený proces poznávání. Z počátku prozkoumávání nového světa máme pouze informace o konkrétních objektech, ze kterých se svět

skládá. Zobecňování je možné až tehdy, když poznáme dostatek konkrétních příkladů. V třídových jazycích musíme nejprve vytvořit třídu (abstrakci), teprve poté můžeme vytvářet její instance (konkrétní objekty).

3.2. Nevýhody prototypových jazyků

Nevýhody prototypů Protože prototyp je sám objektem konceptu, který reprezentuje, může s ním být zacházeno jako s každým objektem konceptu. Například je možné nechat zobrazit prototyp grafického okna. To může vést k chybám, protože změna prototypu ovlivňuje jeho potomky. Například změna poloměru prototypu kruhu může změnit poloměr dalších kruhů.

Nevýhody sdílených úložišť Připomeňme, že sdílené úložiště reprezentuje koncept, ale samo do něj nepatří. To má následující nepříjemné důsledky. Sdílená úložiště definují obsluhy zpráv, které poté, co jim jsou poslány, skončí chybou.

Například sdílené úložiště pro čísla zavádí obsluhu zprávy +. Poslání zprávy + úložišti skončí chybou, protože úložiště není číslem. Teprve jeho potomkům (číslům) může být zpráva + bezchybně poslána.

Další nevýhoda sdílených úložišť vyplývá z jejich dvojí povahy. Na jedné straně sdílené úložiště reprezentuje koncept, na straně druhé je plnohodnotným objektem.

Definovat operace, které bychom chtěli provést se sdílenými úložišti, ale už ne s jejich potomky, není možné, protože každá operace úložiště je platná i pro jeho potomky. Například dává smysl rozšířit sdílené úložiště pro čísla, ale rozšířit číslo je nesmysl.

Nevýhody delegace Připomeňme, že změny objektu díky delegaci ovlivní jeho potomky. Změny prototypu nevedou vždy k požadovanému efektu. Například odebrání atributu nebo metody z prototypu nevyvolá odebrání atributů a metod vlastněných potomky prototypu.Zde je dědičnost jazyků se třídami silnější. Odebereme-li například atribut z třídy, odebere se i všem instancím této třídy.

Dvě formy abstraktních objektů U některých konceptů je problém rozhodnout, zda mají být reprezentovány prototypem, nebo sdíleným úložištěm. Příkladem takového konceptu je grafický objekt. Co je typický představitel grafického objektu? Kruh? Tlačítko?

Náročnost optimalizace kompilátoru Díky dynamičnosti prototypových jazyků a síle delegace jsou kompilátory jejich jazyků náročnější na optimalizaci. Neznamená to, že by jejich výkon byl nižší. Například prototypový jazyk Self má srovnatelný výkon s jazykem Smalltalk, který je založený na třídách.

4. Příklady prototypových jazyků

Tato kapitola představuje několik prototypových jazyků a jejich zajímavé rysy.

První podkapitola se věnuje jazyku Self, který je nejvýznamnějším zástupcem prototypových jazyků. Jazyk Kevo nepoužívá pro mnoho jazyků základní mechanizmus delegování. O tomto jazyce pojednává další podkapitola. O komerčně úspěšném jazyce NewtonScript, který byl používaný pro osobní výpočetní zařízení, pojednává další podkapitola. Poté bude představen staticky typovaný jazyk Omega. Podotkněme, že statické typování je u prototypových jazyků výjimečné. U následující dvojice návrhem podobných objektových systémů Amulet a Garnet bude ukázáno, jak umožňují deklarativně vyjadřovat vztahy mezi objekty pomocí omezení (constraint). Jazyk JavaScript používaný ve webových prohlížečích je v dnešní době nejrozšířenějším prototypovým jazykem, a bude mu věnována další podkapitola. Dále následuje popis jazyka Agora, který umožňuje definovat vlastní vyhodnocování výrazů. Poslední podkapitola je věnována mladému jazyku Ambient. Zprávy jsou v něm netradičně posílány seznamu objektů. Každý z těchto objektů ovlivňuje hledání obsluhy zprávy.

4.1. Self

Programovací jazyk Self je nejpropracovanějším a nejznámějším prototypovým objektovým jazykem. Je odvozený od jazyka Smalltalk, který je založený na třídách. Tato podkapitola čerpá z [1]. Základní myšlenky, které stály při zrodu jazyka, a ukázky práce s vývojovým prostředím, jsou prezentovány na videu [4].

Self i s vývojovým prostředím lze stáhnout z domovské stránky projektu [2].

David Ungar a Randall Smith vytvořili v roce 1986 jazyk Self během práce v Xerox PARC. Několik realizací a grafických vývojových prostředí jazyka Self vzniklo na Stanfordské Univerzitě, kam se tým zanedlouho přesunul. Po uveřejnění jazyka v roce 1990 projekt pokračoval pod Sun Microsystems, kde byl oficiálně zrušen v roce 1995. Část původního týmu spolu s nezávislými programátory pokračuje dál ve vývoji. V roce 2010 vydali novou verzi.

Motivace Programátor je lidská bytost žijící ve světě smyslových vjemů, která se řídí nejen čistou logikou. Programátor při tvorbě potřebuje logiku, ale také spolehlivost, pohodlí a spokojenost. Self se snaží skloubit tuto intelektuální a prožitkovou stránku programování.

V Selfu programátor žije v soudržném a tvárném světě, který je postaven z přímo manipulovatelných objektů.

Objekty Self je čistě objektovým jazykem. Důsledkem toho je, že všechna data, se kterými jazyk pracuje, jsou objekty. Objekty jsou v zásadě tvořeny sloty. O slotech hovoří odstavec v podkapitole 2.1.

Nové objekty mohou být tvořeny klonováním (kopírováním) existujících objektů, nebo vytvořením nového objektu z ničeho. Objekty se tvoří z ničeho výčtem slotů napsaných mezi `(|` a `|)`.

Bod v rovině o souřadnicích $[3, 4]$ může být reprezentován objektem se dvěma sloty: slot jménem `x` s hodnotou 3 a slot jménem `y` s hodnotou 4. Čísla 3 a 4 jsou objekty. Tento objekt může být vytvořen z ničeho vyhodnocením výrazu:

```
(| x = 3 . y = 4|)
```

Ukázka vytvoření objektu s metodou `+`:

```
( |
    + = ( | :arg | (clone x: x + arg x) y: y + arg y ).
| )
```

Dvojtečka před názvem slotu označuje argumentový slot.

Posílání zpráv Další důsledek čistě objektového návrhu je, že posílání zpráv je jediným výpočetním prostředkem. Průběh posílání zpráv je představený v podkapitole 2.2.

Zprávy, jejichž jméno začíná podtržítkem, nazýváme *primitivní*. Primitivní zprávy označují primitivní operaci. Poslání primitivní zprávy vede přímo k vykonání příslušné primitivní operace, aniž by se hledala její obsluha. Například primitivní zpráva `_Clone` poslána objektu vykoná primitivní operaci klonování. Výsledkem bude, že obdržíme klon příjemce formou vrácené hodnoty.

Delegace Delegace v jazyku Self hraje silnou roli. Sloty mohou být označeny jako rodičovské. Objekt uložený v rodičovském slotu nazýváme rodičem objektu. Objekt může mít více rodičovských slotů a v důsledku toho i více rodičů. Cyklická dědičnost je podporovaná.

Pro určení rodičovského slotu stačí za jeho jméno uvést hvězdičku. Například v objektu

```
(|parent* = traits point . x = 3 . y = 4|)
```

je slot `parent` rodičovský. Uchovává tedy rodiče objektu.

Obsluhy zpráv jsou hledané nezávisle ve všech rodičích objektu. Může se tak stát, že se nalezne více obsluh zprávy. V takovém případě byla poslána nejednoznačná zpráva a poslání zprávy skončí chybou.

Nastavovací primitiva Protože je Self čistě objektový, nemůže zde být žádná speciální operace pro změnu obsahu slotu. Místo toho se nastavení slotu provádí posláním zprávy, která vykoná nastavovací primitivum. Pokud lze hodnotu slotu měnit, pak v témže objektu existuje slot obsahující nastavovací primitivum. Jméno tohoto slotu vznikne připojením dvojtečky za jméno nastavovaného slotu.

Pokud například je slot `x` objektu nastavitelný, pak má objekt ve slotu `x:` nastavovací primitivum. Nastavení slotu `x` na hodnotu 2 provedeme posláním zprávy `x:` objektu společně s argumentem 2. Díky této technice je Self schopen nastavovat sloty jen za použití posílání zpráv.

Pokud místo rovnítka (`=`) v popisu slotu uvedeme šipku (`<-`), dojde navíc k vytvoření slotu s nastavovacím primitivem. Tedy objekt

```
(|age <- 0|)
```

má dva sloty. První `age` obsahující nulu a druhý `age:` s nastavovacím primitivem. Pro změnu věku na pět let stačí objektu poslat zprávu `age: 5`.

Zrcadla *Zrcadla* slouží k přímé manipulaci a zjišťování informací o objektech. Zrcadlo je spjato s objektem, kterému se říká *zrcadlený*.

Zrcadlo je objektem. Posíláním zpráv zrcadlu lze například přidávat, odebírat sloty zrcadleného objektu, zjistit jména všech slotů zrcadleného objektu, či získat metodu zrcadleného objektu.

Zrcadlo pro objekt `x` obdržíme posláním

```
reflect: x
```

objektu se standardním chováním. Všechna jména slotů objektu `x` obdržíme posláním zprávy `names` jeho zrcadlu.

Traits a Mixins Jednou z forem sdílených úložišť jsou zde Traits. Obecně o sdílených úložištích pojednává odstavec v podkapitole 2.5.

Další formou sdílených úložišť jsou Mixins. To jsou objekty zajišťující dodatečnou funkčnost, která může být za použití dědičnosti přidána do objektů.

Zprávy Forma posílání zpráv ve výrazech vychází z jazyka Smalltalk. Detailní popis gramatiky jazyka Self včetně příkladů se nachází v [3].

Rozeznáváme tři typy zpráv: unární, binární a klíčové.

Unární zprávy nemají žádný argument. Tvoří je identifikátor napsaný za příjemce zprávy.

Například poslání zprávy `factorial` číslu 5:

```
5 factorial
```

Unární zprávy jsou asociativní zleva doprava.

Proto výraz

```
4 factorial print
```

je ekvivalentní výrazu

```
(4 factorial) print
```

Binární zprávy mají jeden argument. Jsou tvořeny operátorem napsaným mezi příjemce a argument.

Příklad binární zprávy, která pošle zprávu `+` číslu 5 společně s argumentem 10:

```
5 + 10
```

Binární zprávy nejsou asociativní. Výraz

```
3 + 4 * 7
```

je nepřípustný. Výraz se musí uzávorkovat buď

```
(3 + 4) * 7
```

nebo

```
3 + (4 * 7)
```

Výjimkou jsou binární zprávy se stejným operátorem. Ty jsou asociativní zleva doprava. Tedy výraz

```
3 + 4 + 7
```

je interpretován jako

```
(3 + 4) + 7
```

Binární zprávy mají nižší prioritu, než unární zprávy. Proto výraz

```
pi sine + 10 factorial
```

je interpretován jako

```
(pi sine) + (10 factorial)
```

Klíčové zprávy mohou mít jeden nebo více argumentů. Jsou tvořeny příjemcem následovaným dvojicemi klíč a argument. Klíče končí dvojtečkou. První klíč musí začínat malým písmenem, ostatní klíče musí začínat velkým písmenem. Klíčové zprávy mají tolik argumentů, kolik klíčů je tvoří. Jméno poslané zprávy pak obdržíme spojením všech klíčů.

Například

```
5 min: 4 Max: 7
```

pošle zprávu `min:Max:` číslu 5 společně s argumenty 4 a 7.

Klíčové zprávy jsou asociativní zprava doleva. Výraz:

```
5 min: 4 max: 7
```

je interpretován jako

```
5 min: (4 max: 7)
```

Klíčové zprávy mají nejnižší prioritu. Proto výraz

```
i: 5 factorial + pi sine
```

je interpretován jako

```
i: ((5 factorial) + (pi sine))
```

Pokud neuvedeme příjemce zprávy, bude zpráva poslána implicitnímu příjemci (objektu `self`). Příklady zpráv bez uvedeného příjemce:

```
factorial
+ 3
max: 5
```

Napodobování tříd Self umožňuje jednoduché simulování tříd pomocí objektů. A to tak dokonale, že lze do systému nahrávat kód Smalltalku. Smalltalk je objektový jazyk postavený na třídách. Více se dovíte v podkapitole 8.2.

4.1.1. Vývojové prostředí

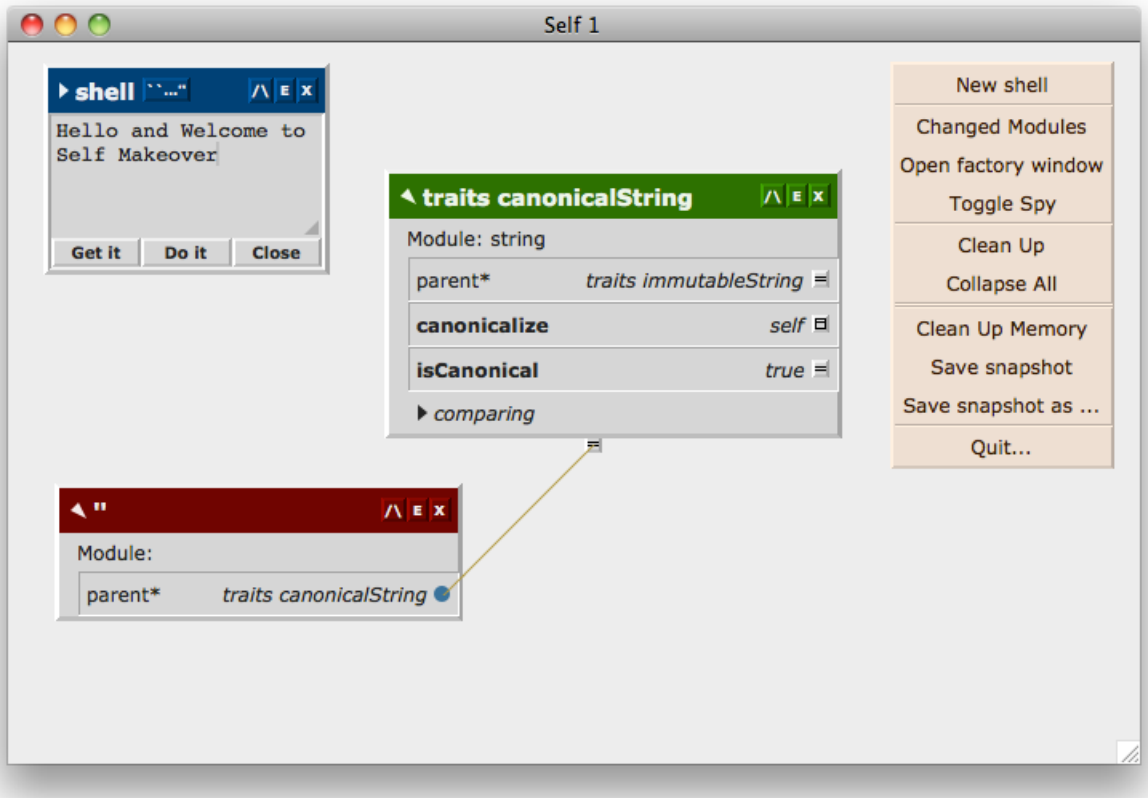
Obecně o vývojových prostředí prototypových jazyků pojednává podkapitola 2.3.

Vývojové prostředí je postavené na myšlence přímé manipulace. Programátor může upravovat běžící aplikaci bez nutnosti jejího znovuspouštění. Uživatelské rozhraní a vývojové prostředí se zde prolínají. Přímé manipulace grafických objektů je dosaženo zavedením dvou principů nazývaných zhmotnění struktury a úpravy zaživa. V [1] lze nalézt podrobný příklad práce s vývojovým prostředím. Vývojové prostředí zachycuje obrázek 3.

Zhmotnění struktury Nejprve se podíváme, jak se zhmotňuje struktura uživatelského rozhraní. Uživatelské rozhraní je postaveno z grafických objektů. Ty jsou vzhledem k dědičnosti uspořádány do hierarchické struktury. Například obdélník, tlačítko a aplikace jsou grafické objekty. Rodičem tlačítka je obdélník.

Každý grafický objekt může mít grafické podobjekty. Například podobjektem textového tlačítka je text. Vztah podobjektu zavádí další hierarchii mezi grafickými objekty. Tu lze pomocí nástrojů vývojového prostředí prohlížet. Jinými slovy tyto nástroje zhmotňují strukturu podobjektů grafických objektů.

Pokud například učiníme kolečko podobjektem obdélníku, dojde intuitivně k slepení těchto dvou objektů. Posun obdélníku vyvolá i posun kolečka.



Obrázek 3. Vývojové prostředí jazyka Self

K lepší organizaci grafických objektů slouží sloupcové a řádkové grafické objekty. Ty uspořádávají své grafické podobjekty do sloupce, respektive do řádku. Například lištu tlačítek vytvoříme tak, že tlačítka učiníme podobjekty řádkového grafického objektu.

Úprava zaživa Úprava zaživa umožňuje při běhu programu libovolně měnit grafické objekty tvořící jeho uživatelské rozhraní. Metanabídka a outliner jsou dva nástroje, které umožňují úpravu zaživa.

Metanabídka je místní (kontextová) nabídka vyvolaná pro grafický objekt. Obsahuje položky pro manipulaci a prohlížení grafického objektu. Například přebarvení červeného kruhu na zelený uskutečníme vybráním položky měnící barvu metanabídky červeného kruhu.

Outliner je grafický objekt, který zobrazuje vnitřní strukturu určitého objektu. Outliner například zobrazuje sloty objektu. Outliner umožňuje přímo měnit objekt.

Například outliner pro bod zobrazuje obsahy slotů `x` a `y` a umožňuje měnit jejich hodnoty. Outliner umožňuje přidat slot obsahující metodu pro výpočet vzdálenosti bodu od počátku.

Skrze metanabídku lze zobrazit ouliner pro kterýkoli grafický objekt.

Nabídky jsou také grafické objekty. Ty však zmizí poté, co uživatel pustí tlačítko myši, které nabídku vyvolalo. Nabídky lze proto pomocí speciální položky učinit trvalými grafickými objekty. Tomuto jevu se říká přišpendlení nabídky. Přišpendlené nabídky lze použít k vytváření ovládacích prvků.

Například vytvoření tlačítka pro změnu barvy kruhu se provede následovně. Vyvoláme metanabídku pro kruh. Přišpendlíme ji. Vybereme položku pro změnu barvy, která je grafickým podobjektem přišpendlené nabídky, a přes metanabídku položky ji vyjmeme. Vyjmutou položku umístíme jako tlačítko vedle kruhu. Kliknutí na toto tlačítko otevře dialog měnící barvu kruhu.

4.2. Kevo

Delegace hraje silnou roli ve většině prototypových jazyků. S delegací souvisí problémy uvedené v kapitole 3. Jazyka bez delegace by se tyto problémy netýkaly. To je případ jazyka Kevo, který je prototypovým jazykem bez delegace. Podkapitola čerpá z [7].

Kevo vytvořil Antero Taivalsaari v roce 1992. Vyšlo několik vědeckých článků publikovaných autorem. Po roce však vývoj jazyka Kevo ustává.

Objekt je v Kevu tvořen sloty. Objekt ve slotech uchovává obsluhy všech zpráv, kterým rozumí. Objekty se tak stávají nezávislými. Změna jednoho objektu neovlivní ostatní objekty. Přestože jsou objekty na logické úrovni nezávislé, ve vnitřní reprezentaci se uplatňuje sdílení společných částí objektů.

Objekty s podobnou strukturou a chováním tvoří *rodinu klonů*.

Obrázek 4. MessagePad

Kevo zavádí skupinové operace, které pracují s rodinami klonů.

Například existuje skupinová operace pro přidání slotu všem objektům patřícím do rodiny klonů určenou daným objektem. Použitím takové operace na barevný bod můžeme přidat slot všem barevným bodům. Všechny barevné body mají podobnou strukturu a chování, a proto patří do stejné rodiny klonů.

4.3. NewtonScript

Jazyk NewtonScript vytvořil Walter Smith pod firmou Apple. NewtonScript je hlavním jazykem platformy Newton, používanou pro levná osobní výpočetní zařízení. Byl uveřejněn spolu s prvním zařízením MessagePad v roce 1993. Je to jeden z mála komerčně úspěšných prototypových jazyků.

MessagePad se nachází na obrázku 4.

Tato podkapitola čerpá z článku [13].

Nejzajímavějším rysem jazyka je, že zavádí dva typy delegace. První vyjadřuje vztah potomek-rodič a druhý vztah prvek-kontejner.

Obsluha zaslané zprávy se nejdříve hledá v předcích příjemce. Pokud se zde obsluha nenajde, pokračuje hledání v kontejneru příjemce. Zde se postupuje obdobně. Toto řešení delegace je kompromisem mezi obecností a výpočetní složitostí. Jazyk Eggs představený v kapitole 5. mechanizmus delegace zobecňuje.

Návrh NewtonScriptu byl více řízen praktickými potřebami, než touhou po jednoduchosti návrhu. Proto není čistě objektový.

Objekty se tvoří pomocí rámů (frame). Rám je soubor slotů. O slotech více v odstavci podkapitoly 2.1.

Například

```
{ a: 1, b: "a", c: fun () 5 + 3 }
```

je rám o třech slotech se jmény `a`, `b` a `c`. Slot `a` odkazuje na číslo `1`, slot `b` na řetězec `"a"` a slot `c` na funkci s kódem `5 + 3`.

Delegace mezi rámy používá dva speciální sloty `_parent` a `_proto`. Slot `_parent` odkazuje na kontejner rámu a slot `_proto` na rodičovský rám. Rám spolu s jeho předky tvoří *entitu*.

Pro nastavení hodnoty atributu platí speciální pravidlo: Nastavení atributu se provede v první entitě v linii prvek-kontejner, který proměnnou zavádí. Nastavení atributu objektu je objasněno v ukázce.

Ukázka Vytvoříme skupinu dvou přepínačů. K tomu budeme potřebovat pět rámů `protoRadioButton`, `protoRadioCluster`, `cluster`, `radio1`, `radio2`. Rodičem rámů `radio1` a `radio2` bude rám `protoRadioButton`. Rám `protoRadioCluster` bude rodičem rámu `cluster`. Rám `cluster` bude kontejnerem rámů `radio1` a `radio2`.

Následující kód vytvoří požadovanou strukturu rámů.

```
protoRadioButton :=
     {Click: func ()
              self:ChooseButton(self)};

protoRadioCluster :=
     {chosenButton: nil,
       ChooseButton: func (whichBtn)
                          chosenButton := whichBtn};

cluster :=
     {_proto: protoRadioCluster};

radio1 :=
     {_proto: protoRadioButton,
      _parent: cluster};

radio2 :=
     {_proto: protoRadioButton,
      _parent: cluster};
```

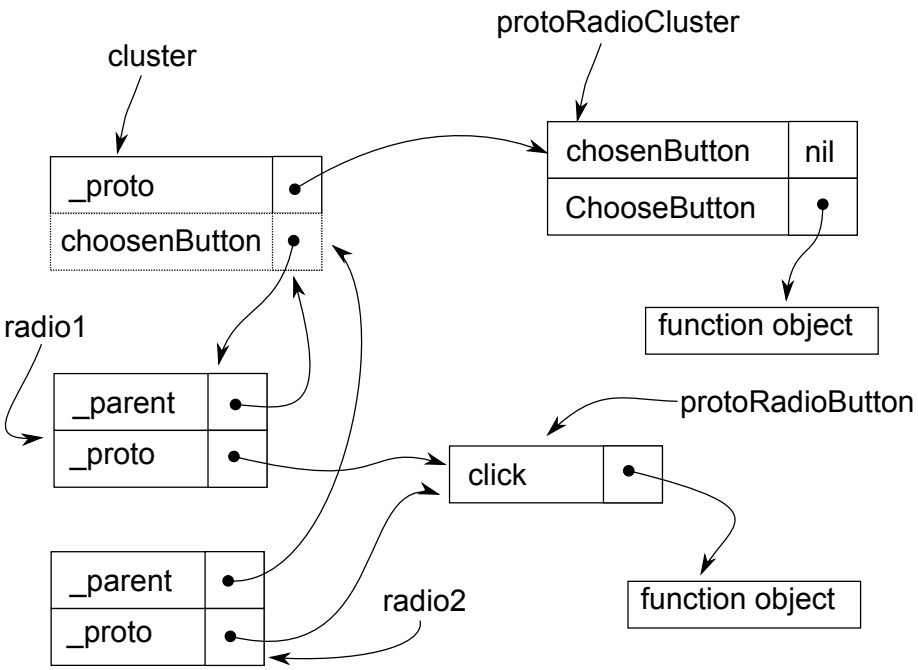
Vztahy rámů vyjadřuje obrázek 5.

Na začátku není vybrán žádný přepínač. Atribut `chosenButton` objektu `cluster` je navázán na hodnotu `nil`.

Následuje průběh akce vyvolaný kliknutím myší na první přepínač.

Rámu `radio1` se pošle zpráva `click`.

Protože rám `radio1` neobsahuje slot s názvem `click`, prohledávání pokračuje u jeho rodiče. To je rám, na který odkazuje slot `_proto`. Tedy rám `protoRadioButton`.



Obrázek 5. NewtonScript: Ukázka

V tomto rámu existuje slot `click` odkazující na funkci. Zde prohledávání skončí a dojde k zavolání funkce. Na proměnnou `self` se naváže objekt `radio1` a vyhodnotí se kód:

```
self:ChooseButton(self)
```

To způsobí, že se objektu `radio1` pošle zpráva `ChooseButton` s argumentem `radio1`.

Hledání obsluhy zprávy `ChooseButton` probíhá přes rámy `radio1`, `protoRadioButton`. Protože ani jeden z těchto rámů nemá slot `ChooseButton`, pokračuje prohledávání v rámu `cluster`. To je rám, na který odkazuje slot `_parent` rámu `radio1`.

Ani ten neobsahuje slot požadovaného jména. Hledání obsluhy dojde k jeho rodiči `protoRadioCluster`.

Slot `ChooseButton` rámu `protoRadioCluster` odkazuje na funkci, která je hledanou obsluhou zprávy `ChooseButton` poslané objektu `radio1`.

Tato funkce se zavolá, čímž se vyhodnotí kód:

```
chosenButton := whichBtn
```

Proměnná `whichBtn` je navázána na argument zprávy, tedy na `radio1`.

Nastavení atributu pomocí operátoru `:=` nastaví slot `chosenButton` rámu `cluster` na hodnotu `radio1`.

Rám `cluster` byl vybrán proto, že je první entitou v linii prvek-kontejner rámu `radio1`, který proměnnou `chosenButton` zavádí.

Opravdu rám `protoRadioCluster` patří do entity určenou rámem `cluster` a obsahuje slot `chosenButton`. Tento slot odkazuje na hodnotu `nil`.

4.4. Omega

Jazyk Omega je na rozdíl od většiny prototypových jazyků staticky typovaný.

Jazyk Omega vytvořil Günther Blaschek v roce 1990. Poslední vědecký článek o jazyku vychází v roce 1996.

Statické typování tohoto jazyka je jeho nejpozoruhodnějším rysem. Ten přináší na jedné straně omezení a na straně druhé výhody staticky typovaných jazyků: silnější kontrolu chyb při kompilaci a větší optimalizaci kódu. Statický typovaný systém byl převzat od jazyku Eiffel.

Tato podkapitola vychází z článku [14].

Prototypy a typy Pojem prototypu a typu v tomto jazyku splývá.

Prototypy jsou pojmenované objekty. Každý prototyp určuje typ stejného jména. Typ je možné použít v deklaracích metod a proměnných.

Omega nevyužívá delegace, ale klasické dědičnosti známé z jazyků založených na třídách.

Dědičnost se týká pouze prototypů. To znamená, že pouze prototyp může mít rodiče. Například rodičem prototypu `Bird` je prototyp `Object`. To vytváří i hierarchii typů, která se kryje s hierarchií prototypů. Například typ `Bird` je podtypem typu `Object`.

Nejsilnějším omezením jazyka je, že klony musí mít stejnou strukturu i chování jako jejich prototyp. Nemohou tedy měnit svoji strukturu ani chování. Toto omezení je vynucené právě statickým typováním. Změna struktury nebo chování prototypu se proto promítne i do všech jeho klonů. Mezi prototypem a jeho klony je stálá vazba, která toto chování umožňuje.

Následující kód ukazuje deklaraci proměnné `statusWindow` na typ `Window` a přiřazení klonu prototypu `Window` k této proměnné.

```
statusWindow: Window;
statusWindow := Window copy;
```

K proměnné deklarované pro typ lze přiřadit objekt specifičtějšího typu. V níže uvedeném kódu je proměnná `toolWindow` deklarována na typ `Window`. Může jí být přiřazen klon prototypu `FloatingWindow`, protože ten je potomkem prototypu `Window`.

```
toolWindow: Window := FloatingWindow copy;
```

Jazyk dokáže určit typ proměnné podle typu hodnoty výrazu na pravé straně. Například v následujícím kódu bude proměnná `statusWindow` deklarována na typ `Window`.

```
statusWindow ::= Window copy;
```

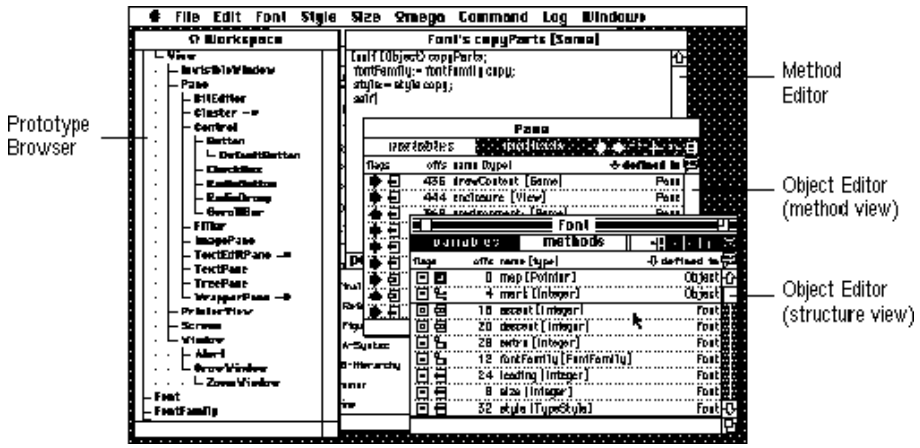
V dalším kódu se tohoto využije u proměnné `phoneNumber`. Nemusíme vědět, jakého typu tato proměnná je. Její typ bude stejný jako typ návratové hodnoty metody `lookup:`. Tento typ se určí v době kompilace. Nejedná se tedy o dynamické typování.

```
phoneNumber ::= phoneDirectory lookup: "Blaschek";
phoneNumber == Nil ifFalse: [ phoneNumber dial ];
```

Výhodou tohoto řešení je, že pokud změníme typ návratové hodnoty metody `lookup:`, kód nebude třeba měnit. Kód bude nutné znovu zkompilovat. Kompilace se provede automaticky.

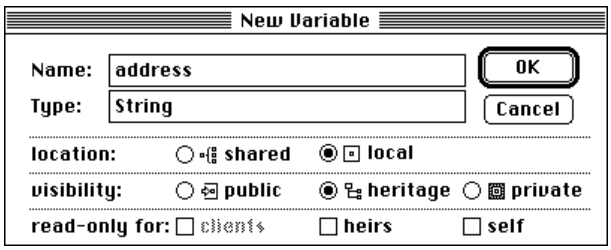
Vývojové prostředí K jazyku Omega existuje silné vývojové prostředí.

Některé akce, jako například vytvoření nového prototypu nebo přidání atributu do prototypu, se odehrávají prostřednictvím dialogů ve vývojovém prostředí. Standardní situace ve vývojovém prostředí je zachycena na obrázku 6. Na obrázku 7. se nachází dialog přidání atributu do prototypu.



Obrázek 6. Omega: Vývojové prostředí

Kovariance Zpráva může při poslání objektům různých typů vracet hodnotu různých typů. Například zpráva `copy`. Nastává problém, na jaký typ deklarovat její návratovou hodnotu. Nemůže být deklarována na nejobecnější typ `Object`, protože pak by návratová hodnota šla přiřadit jedině proměnné typu `Object`.



Obrázek 7. Omega: Přidání atributu do prototypu

To by znemožňovalo volání metod zavedených u konkrétnějších typů než je typ `Object`.

Za tímto účelem zavádí Omega pseudotyp `Same`. Pokud je návratová hodnota zprávy deklarována na pseudotyp `Same`, znamená to, že její typ je stejný jako typ příjemce zprávy. Právě na typ `Same` je deklarována návratová hodnota zprávy `copy`. Proto výraz `Window copy` se vyhodnotí na objekt typu `Window`.

Podmínkové přiřazení Proměnné obecnějšího typu může být přiřazen objekt typu konkrétnějšího. Můžeme také říci podtypu. To však znemožňuje volat metody objektu, které obecný typ nezavádí. Například typ `bird` deklaruje metodu `sing`, ta však není deklarována pro typ `Object`. *Podmínkové přiřazení* je jediným prostředkem jazyka, který umožňuje přiřadit objekt obecnějšího typu do proměnné konkrétnějšího typu.

V následujícím příkladu deklarujeme dvě proměnné. Proměnnou `anything` typu `Object` a proměnnou `tweety` typu `Bird`. Déle v kódu budeme chtít zkontrolovat, zda proměnné `anything` je přiřazen objekt, který může být přiřazen proměnné `tweety`, neboli zda je tento objekt typu `Bird`. Pokud ano, pak tomuto objektu pošleme zprávu `sing`.

```
anything: Object; tweety: Bird;
...
(tweety :? anything) ifTrue: [ tweety sing ];
```

Podmínkové přiřazení (operátor `:?`) funguje tak, že se zjistí, zda může proměnné na levé straně být přiřazen objekt na straně pravé. Přesněji se porovná statický typ proměnné na levé straně s dynamickým typem hodnoty na straně pravé. Pokud je odpověď kladná, pak dojde k přiřazení objektu do proměnné a podmínkové přiřazení vrátí pravdu. V opačném případě podmínkové přiřazení vrátí nepravdu.

Další ukázka představuje zkrácenou formu spojující deklaraci proměnné a podmínkového příkazu. Otestujeme v ní proměnnou `anything` na typy `Bird` a `Cat`.

```
(tweety: Bird :?= anything) ifTrue: [ tweety sing ];
(sylvester: Cat :?= anything) ifTrue: [ sylvester meow ];
```

Generické prototypy Pro vytváření kolekce objektů se používají *generické prototypy*. Ty povolují používat pseudotyp `Parameter` v deklaracích metod a atributů. Když je vytvořený nový generický prototyp, musí být určen jeho základní typ. Jazyk pak vytvoří prototyp, kde všechny výskyty pseudotypu `Parameter` budou nahrazeny základním typem. Ve většině případů je základním typem nejobecnější typ, neboli typ `Object`.

Z generických prototypů lze tvořit *generované prototypy* pomocí parametrizace. Parametrem je typ. Jako parametr pro generický prototyp lze použít pouze typ, který je konkrétnější než základní typ.

Uvažujme například generický prototyp `Array` se základním typem `Object`. Ten slouží k ukládání libovolných objektů do pole. Zadáním parametru `Bird` z něj vytvoříme generovaný prototyp. Parametr se píše do složených závorek. V našem příkladě se vytvoří generovaný prototyp následovně: `Array{Bird}`.

V generovaném prototypu budou všechny výskyty pseudotypu `Parameter` v deklaracích generického prototypu nahrazeny zadaným parametrem.

V příkladu je pseudotyp `Parameter` generického typu `Array` nahrazen typem `Bird`. Takto vytvořený prototyp slouží k ukládání objektů typu `Bird` do pole. Mohou pro něj být definovány metody, které pracují s polem obsahujícím výhradně objekty typu `Bird`.

Za pozornost stojí následující vlastnost generických prototypů. Uvažujme libovolný generický prototyp. Dále uvažujme dva typy, které mohou být použity jako parametr pro tento generický prototyp. Jsou to tedy typy, které jsou konkrétnější než základní typ generického prototypu. Dále požadujeme, aby první typ byl obecnější než typ druhý. Pak platí, že generovaný prototyp vytvořený parametrizací generického prototypu podle prvního typu není obecnější než typ generovaný podle druhého typu. Tuto vlastnost si ukážeme na příkladě.

Vezměme generický prototyp `Array` se základním typem `Object`. Jako dva typy vezměme typy `Bird` a `Vulture` (kondor). Tyto typy jsou konkrétnější než typ `Object` a typ `Bird` je obecnější než typ `Vulture`. Výše zmíněné pravidlo nám říká, že typ `Array{Bird}` není obecnější než typ `Array{Vulture}`.

Předpokládáme, že typ `Array{Bird}` je obecnější než typ `Array{Vulture}`. Uvažujme proměnnou `parkPopulation` deklarovanou na typ `Array{Bird}`. Přiřaďme této proměnné objekt typu `Array{Vulture}`, což nám dovoluje předpoklad. Uvažujme dále objekt typu `Pinguin`. Protože typ `Bird` je obecnější než typ `Pinguin`, mělo by být možné jej vložit do pole přiřazenému proměnné `parkPopulation`. To však nelze, protože toto pole může uchovávat pouze objekty typu `Vulture` a typ `Vulture` není obecnější než typ `Pinguin`. Sporem jsme dokázali, že typ `Array{Bird}` není obecnější než typ `Array{Vulture}`.

Monomorfické typy Všechna data vyskytující se v jazyce jsou objekty, dokonce i čísla. Základní objekty, jako například zmíněná čísla, jsou *monomorfického typu*.

Základním omezením monomorfického typu je, že nemůže mít podtypy. To znamená, že proměnné deklarované pro monomorfický typ bude vždy přiřazen objekt právě tohoto typu.

Uvedené omezení přináší řadu výhod. Běžné proměnné jsou vnitřně reprezentovány jako odkaz na objekt. Monomorfické proměnné místo toho ve vnitřní reprezentaci přímo obsahují přiřazený objekt. Dále se pro monomorfické objekty používá rychlejší statická vazba mezi zprávou a metodou. A nakonec, kde je to možné, tam jsou kódy krátkých metod přímo vkládány do míst, kde jsou volány. Tím se předchází výpočetně náročnějšímu posílání zpráv.

4.5. Amulet a Garnet

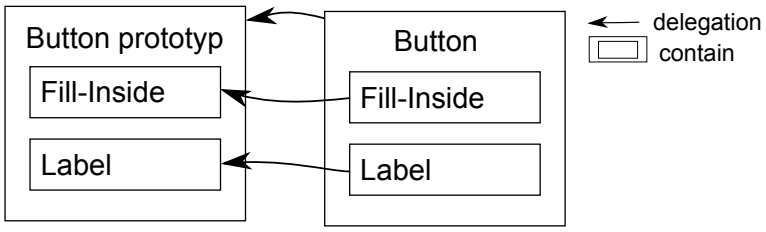
Objektové systémy Amulet a Garnet umožňují deklarativně vyjadřovat vztahy mezi objekty.

Objektový systém Garnet byl vytvořen pro Common Lisp. Byl vyvíjen mezi roky 1987 a 1994 skupinou User Interface Software Group v the Human Computer Interaction Institute pod School of Computer Science v Carnegie Mellon University. Po ukončení jeho vývoje skupina vytvořila systém Amulet. Objektový systém Amulet je učený pro jazyk C++. Skupina jej vyvíjela od roku 1994 do roku 1997.

Nejprve se podíváme, jak tyto jazyky podporují hierarchii prvek-kontejner. Poté následuje popis nejsilnějšího prostředku těchto systémů - omezení (anglicky constraints). Omezením jsou věnovány všechny ostatní odstavce.

Hierarchie prvek-kontejner Tato hierarchie je tvořená *kontejnery*. To jsou objekty, které mohou obsahovat další objekty, které nazýváme prvky. Prvky kontejneru jsou uloženy ve slotech. Při vytváření rozšíření kontejneru se vytvoří rozšíření i jeho prvků.

Obrázek 9. znázorňuje kontejner tlačítko (`Button`) o prvcích pozadí (`Fill-Inside`) a text (`Label`). Níže je uvedený kód Garnetu, který tento kontejner vytvoří. Obrázek 8. vyjadřuje vztahy prototypu tlačítka a jeho rozšíření.



Obrázek 8. Vztahy prototypu tlačítka a jeho rozšíření

Omezení Kromě metod a obyčejných objektů existují v systémech Garnet a Amulet omezení. *Omezení* vyjadřuje vztahy mezi objekty.

Tyto systémy podporují styl programování, který se anglicky nazývá Constraint programming.

Například pokud chceme, aby dva obdélníky byly těsně vedle sebe, stačí formulovat omezení, které vyjadřuje tuto vlastnost.

Jednou zavedené omezení je udržováno systémem. Změna objektu vede ke změně omezením závislého objektu. Systém po změně grafického objektu zajistí jeho překreslení.

V příkladu s obdélníky, vede posun prvního obdélníku k posunu i druhého tak, aby zůstalo zachováno omezení těsného sousedství.

Zavedení omezení do systému přináší jiný styl programování. Metody v něm hrají menší roli. Programování se stává více deklarativní a více založené na datech.

Nejjednodušším typem omezení jsou omezení používající formule připojené ke slotu. Tato omezení jsou jednosměrná a mají jediný výstup. Mocnějším typem omezení jsou web omezení, která jsou vícesměnná a více výstupová. O obou druzích pojednávají následující odstavce. Poslední odstavec objasňuje, jak omezení fungují.

Omezení formulí Ke slotu objektu může být připojena formule, která počítá jeho hodnotu. Hodnota slotu může být počítána na základě hodnot jiných slotů v libovolných objektech. Pokud jsou hodnoty těchto slotů změněny, formule automaticky vypočítá hodnotu a uloží ji do příslušného slotu.

Na omezení může být závislý pouze slot, ke kterému je formule přiřazena. Proto říkáme, že tento druh omezení má jediný výstup. Slot může být závislý na více slotech různých objektů. Změny se promítají v jediném směru: od nezávislých slotů k závislému slotu. Proto říkáme, že omezení formulí je jednosměrné omezení.

Formule omezení může obsahovat libovolný kód.

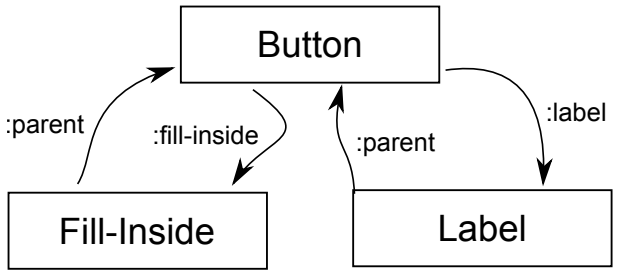
Například textový objekt má omezení pro sloty `width` a `height`. Ty mění jeho rozměry v závislosti na změně textu a použitého fontu.

Odkaz na slot jiného objektu se vyjadřuje za použití cesty. *Cesta* je posloupnost jmen slotů, které vedou do cílového objektu.

Například v tlačítku, jehož struktura je vyjádřena obrázkem 9., může objekt `Fill-Inside` odkazovat na slot `width` objektu `Label` pomocí cesty `(gv :Self :parent :label :width)`. Tato cesta začíná u objektu `Fill-Inside`, jde ke kontejneru `Button`, odtud k prvku `Label`, kde se nakonec získá jeho šířka.

Následující kód vytvoří strukturu tlačítka zachycenou na obrázku 9.

```
(create-instance 'button aggregate
 (:left 20)
 (:top 20)
 (:string "label")
```



Obrázek 9. Tlačítko

```
'(:parts (
  (:fill-inside ,rectangle
   (:left ,(formula (+ 2 (gv :Self :parent :left))))
   (:top ,(formula (+ 2 (gv :Self :parent :top))))
   (:width ,(formula (+ 4  (gv :Self :parent :label :width))))
   (:height ,(formula (+ 4 (gv :Self :parent :label :height))))
   (:color ,gray))
  (:label ,text
    (:left ,(formula (center-x (gv :Self :parent :fill-inside))))
    (:top ,(formula (center-y (gv :Self :parent :fill-inside))))
    (:string ,(formula (gv :Self :parent :string)))))))
```

Pro slot `width` objektu `Fill-Inside` je vytvořeno omezení

```
(formula (+ 4  (gv :Self :parent :label :width)))
```

Slot vyjadřuje šířku pozadí tlačítka. Formule vezme šířku objektu `Label` a přičte k ní číslo 4. Formule vyjadřuje šířku textu tlačítka zvětšenou o čtyři. Celý efekt omezení spočívá v tom, že šířka obdélníkového pozadí tlačítka se bude rovnat šířce textu tlačítka zvětšeného o čtyři. Při změně textového fontu tlačítka nebo jeho textu dojde k přepočítání uvažovaného omezení a změně šířky tlačítka.

Jiné použití omezení kopíruje text z objektu `Button` do objektu `Label`. Toto omezení je uložené ve slotu `string` objektu `Label` a má formuli

```
(formula (gv :Self :parent :string))
```

Změna textu tlačítka vede ke změně textu objektu `Label`.

Ve výše zmíněném příkladu je tlačítko rozšířením prototypu kontejneru (`aggregate`).

Při vytváření rozšíření mohou být zadány parametry. U výše popsaného tlačítka to mohou být parametry `left` a `top` určující pozici tlačítka, nebo parametr `string` určující text tlačítka. Tyto parametry jsou v rozšíření vloženy do příslušných slotů.

Při změně jsou ovlivněná omezení vždy vyhodnocena pouze jednou. Systém hlídá cykly mezi omezeními. Pokud vyhodnocování omezení narazí na cyklus, projde jej pouze jednou.

Nepřímá omezení Systém umožňuje vytvořit omezení, které dynamicky určuje objekty a sloty, na kterých závisí.

Pomocí nepřímých omezení lze například vyjádřit, že šířka kontejneru je rovna maximální šířce jeho prvků. Toto omezení bude přepočítáno, kdykoli se do kontejneru přidá, nebo odebere prvek, anebo když se změní šířka libovolného prvku kontejneru.

Ve výše uvedeném příkladu tlačítka jsou všechna omezení nepřímá. Objekt, na kterém závisí hodnota slotu, je dynamicky určen podle příslušné cesty.

Celá myšlenka cest a omezení připojených ke slotům umožňuje vytvořit rozšíření objektu s omezeními bez nutnosti změny nového objektu.

Například všechna omezení uvedená ve výše zmíněném tlačítku budou fungovat i v jeho rozšířeních. To je způsobeno tím, že při vytvoření rozšíření jsou vytvořena také rozšíření všech jeho omezení. Tedy omezení v novém objektu budou závislá na nových objektech, ne na objektech, které byly rozšířeny.

Omezení s vedlejším efektem Garnet používá líné vyhodnocování omezení. Omezení je proto vyhodnoceno, až když je dotázáno na hodnotu slotu, který je na něm závislý. Toto chování znemožňuje vkládat vedlejší efekt do omezení.

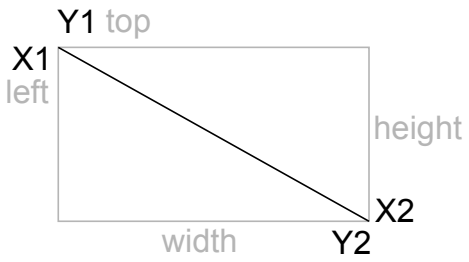
Omezení Amuletu jsou vyhodnocena okamžitě po změně závislé hodnoty. Díky tomu může mít omezení vedlejší efekt. Následující část se týká pouze omezení systému Amulet.

Uvažujme úsečku vedoucí z levého horního rohu do pravého spodního rohu obdélníku. Obrázek 10. znázorňuje úsečku i obdélník a jejich sloty. Obdélník je vymezen hodnotami slotů `left`, `top`, `width` a `height`. Sloty `X1` a `Y1` uchovávají souřadnice počátečního bodu úsečky. Sloty `X2` a `Y2` slouží k uložení souřadnic koncového bodu úsečky.

Souřadnice koncového bodu úsečky jsou závislé na rozměrech a pozici obdélníku. Tuto závislost lze zachytit omezením uloženém ve slotu `X2`, které jako svůj vedlejší efekt nastaví hodnotu slotu `Y2`. Bez dalšího vysvětlování pouze uveďme kód omezení napsaný v jazyce C++ pro systém Amulet.

```
Am_Define_Formula(int, line_x2y2) {
    Am_Object source_obj = self.Get(INPUT_1);
    int x2 = (int)source_obj.Get(Am_WIDTH) +
            (int)source_obj.Get(Am_LEFT);
    int y2 = (int)source_obj.Get(Am_HEIGHT) +
            (int)source_obj.Get(Am_TOP);
    self.Set(Am_Y2, y2);
    return x2; }
```

Pokud programátor používá omezení s vedlejšími efekty, musí si dát pozor, aby vyhodnocování omezení nespadlo do nekonečné smyčky. Například tak, že jedno omezení ve vedlejším efektu učiní neaktuální druhé omezení a druhé omezení vedlejším efektem učiní neaktuální první omezení.



Obrázek 10. Sloty úsečky

Více omezení v jediném slotu Systém Amulet umožňuje připojit více omezení k jedinému slotu. Toho lze využít v situaci, kdy hodnota slotu může být měněna více způsoby.

Například výřez grafického okna může být posunut posuvníkem nebo změnou proměnné aplikace. Obě tyto závislosti lze vyjádřit omezeními připojenými ke slotu textového okna. Výřez okna se posune při manipulaci posuvníku uživatelem nebo při změně proměnné aplikace.

Více výstupová a vícesměnná omezení Systém Amulet podporuje více výstupové a vícesměnné omezení nazývané *web omezení*.

Web omezení může mít libovolný počet vstupních a výstupních slotů. Stejně jako omezení formulí, web omezení může dynamicky počítat závislosti a závislosti mezi web omezeními jsou hlídané systémem.

Webová omezení dokáží zachytit složitější závislosti, na které omezení formulí nestačí.

Lze je například použít pro zachování konzistence slotů úsečky. Úsečka má dva typy vstupních slotů. Jedna skupina odvozená od koncových bodů úsečky má vstupní sloty `X1`, `Y1`, `X2` a `Y2`. Druhá, vycházející z obdélníku, kterým je úsečka ohraničena, má sloty `left`, `top`, `width` a `height`. První skupinu slotů použijeme pro posun koncových bodů úsečky. Pro posun celé úsečky bude výhodnější použít druhou skupinu slotů. Obě skupiny slotů jsou znázorněny na obrázku 10.

Představme si, že úsečce změníme sloty `X1`, `top` a `width`. Řešení jednosměrnými omezeními nemusí vést ke správnému pořadí vyhodnocování omezení, a tedy nemusí skončit očekávaným výsledkem. Řešení web omezeními udržuje původní pořadí změn a vede ke správnému výsledku.

Animace Amulet zavádí *animační omezení*. Ty slouží k automatické postupné změně hodnoty slotů od jedné hodnoty k druhé.

Uvažujme, že došlo ke změně slotu s animačním omezením. Nejprve se získá stará hodnota slotu. Poté se postupně hodnota slotu mění od staré hodnoty k nové.

Pro animační omezení lze nastavit rychlost změny i průběh změny.

Animovat lze různé typy hodnot: skaláry, souřadnice, fonty, barvy, seznam vrcholů (pro polygony), pravdivostní hodnoty (pro viditelnost).

Jak omezení fungují Ke každému slotu existují dva seznamy omezení. První je seznam omezení, která jsou závislá na hodnotě slotu. Ve druhém seznamu jsou omezení, na kterých závisí hodnota slotu.

Omezení posílá zprávy slotu, v jehož seznamech se nachází. Nejdůležitější takové zprávy jsou:

- `Set` (nastavení hodnoty slotu),
- `Invalidate` (učiní hodnotu slotu neplatnou) a
- `Get` (vrátí hodnotu slotu).

Zprávu `Invalidate` posílá omezení na něm závislým slotům.

Slot může posílat zprávy omezením ve svých seznamech. Některé důležité zprávy:

- `Change` (ohlásí změnu hodnoty slotu),
- `Invalidated` (ohlásí neplatnost své hodnoty),
- `Get` (požádá omezení pro výpočet nové hodnoty).

Zprávu `Invalidated` posílá slot poté, co se stal neplatným, všem na něm závislým omezením.

Následuje popis akcí, které nastanou po dotázání na hodnotu slotu.

Pokud je hodnota slotu platná, dojde k vrácení hodnoty slotu.

Při neplatné hodnotě slotu se posílá zpráva `Get` omezením, na nichž závisí hodnota slotu. Omezení se dotazují na hodnotu ve stejném pořadí, ve kterém se stávala neplatná.

Omezení nemusí na zprávu `Get` vrátit hodnotu. Pokud omezení vrátí hodnotu, pak se slot stává platným, hodnota je uložena do slotu a vrácena. Jestliže omezení nevrátí hodnotu, pak se pošle zpráva `Get` dalšímu omezení v pořadí.

Když žádné omezení nevrátí hodnotu, pak si slot ponechá starou hodnotu, učiní se platným a vrátí starou hodnotu.

Nakonec systém překreslí všechny omezeními změněné grafické objekty.

4.6. JavaScript

Tato podkapitola popisuje JavaScript, jazyk s prototypovým objektovým systémem.

JavaScript vytvořil Brendan Eich v roce 1995 pro webový prohlížeč Netscape Navigator. JavaScript je implementací standardu ECMAScript. JavaScript

je hlavně používán ve webových prohlížečích. Kód JavaScriptu bývá vkládán do webových stránek a interpretován webovými stránkami. Tím umožňuje vytvářet dynamické webové stránky.

Další použití JavaScriptu se nachází v PDF dokumentech, prohlížečích, malých aplikacích umístěných na ploše. Poslední dobou se stává populární použití JavaScriptu na webové aplikace na straně serveru.

Kód JavaScriptu je ovlivněn jazykem C. Základní principy JavaScript přebírá z jazyků Self a Scheme.

Podkapitola vychází z [18] a standardu ECMAScript [19].

Objekty a vlastnosti Návrh jazyka není čistě objektový.

Objekt obsahuje sloty, které se zde nazývají vlastnosti. Hodnotou vlastnosti může být objekt nebo primitivum.

K hodnotám vlastností se přistupuje přes tečkovou notaci:

objectName.*slotName*

Například zjištění modelu auta: `myCar.model`

Vlastnosti se přidávají a mění přiřazovacím příkazem.

Následující příkaz změní vlastnost jménem `model` objektu `myCar`:

```
myCar.model = "Mustang";
```

Zprávy se posílání podobně jako volání funkcí.

Příklad poslání zprávy `ageAfter` objektu `person` s argumentem 5:

```
person.ageAfter(5)
```

Metody jsou vlastnosti, jejichž hodnota je funkce. Oproti ostatním jazykům používajícím sloty (vlastnosti) musí být obsluha zprávy funkce.

Proto například následující poslání zprávy skončí chybou:

```
myCar.model()
```

Metody se definují přiřazením funkce k vlastnosti.

Například definování metody `show`:

```
myCar.show = function () alert(this.model);
```

Poslání zprávy, které tuto metodu vyhodnotí:

```
myCar.show()
```

Při volání metody je proměnná `this` nastavena na příjemce zprávy. V předchozím příkladě jím je objekt `myCar`.

Vytváření nových objektů Nové objekty můžeme tvořit z ničeho nebo za použití vytvářecí funkce. Podíváme se zblízka na oba způsoby.

Z ničeho objekt vytvoříme tak, že ve složených závorkách popíšeme jeho vlastnosti.

Ukázka tvoření objektu z ničeho:

```
var myHonda = {color: "red",
               wheels: 4,
               engine: {cylinders: 4, size: 2.2}};
```

Vytvářecí funkce je funkce, která ve svém těle nastavuje vlastnosti nově vytvořeného objektu.

Příklad vytvářecí funkce:

```
function Person(age)
{this.age = age};
```

Volání vytvářecí funkce s klíčovým slovem `new` vykoná kód funkce, kde proměnná `this` bude navázána na nově vytvořený objekt.

Příklad vytvoření objektu:

```
var alex = new Person(20);
```

Objekt může mít jednoho rodiče. Funkce jsou objekty. Proto mohou mít vlastnosti. Vytvářecí funkce má vlastnost `prototype`. Funkcí vytvořený objekt vznikne rozšířením hodnoty této vlastnosti. Jinými slovy hodnota vlastnosti `prototype` bude rodičem nově vzniklého objektu.

Definovaní metody prototypu:

```
Person.prototype.ageAfter = function (time)
                     {return this.age + time};
```

Výraz `alex.ageAfter(5)` vyhodnotí metodu prototypu objektu a vrátí 25.

Jak ukazuje další příklad, můžeme vytvářet hierarchii objektů.

```
function Woman(age,numberOfChildren)
{
  this.age = age;
  this.numberOfChildren = numberOfChildren
};

Woman.prototype = new Person(40);

Woman.prototype.sex = "F";
Woman.prototype.hasChildren = function ()
    {return this.numberOfChildren !== 0};
```

Vytvoříme nový objekt:

```
var elisabeth  = new Woman(35,2);
```

Tento objekt má vlastnosti `age` a `numberOfChildren`. Rozumí zprávám `hasChildren` a `ageAfter`.

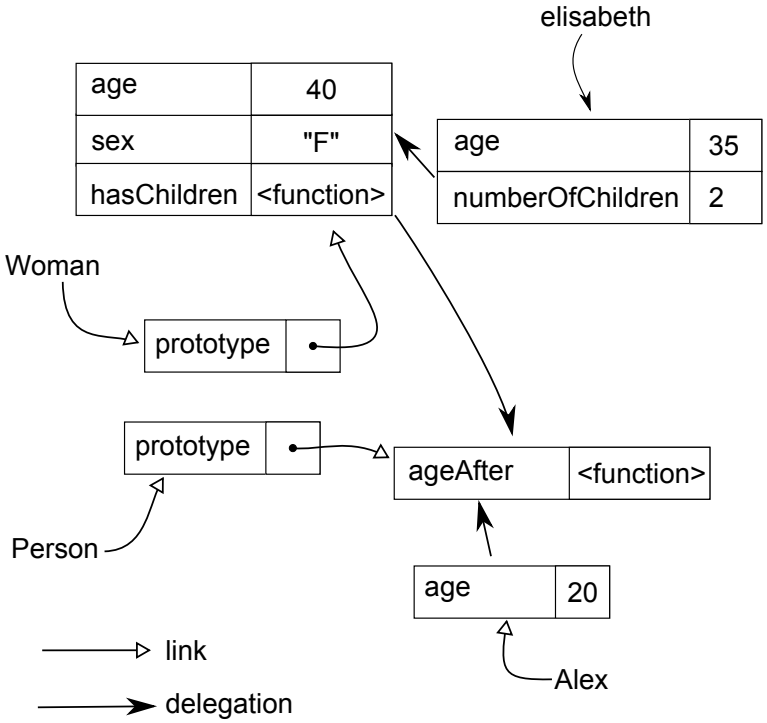
Při pokusu o získání hodnoty vlastnosti se hodnota hledá stejně jako obsluha poslané zprávy.

Proto výraz

```
elisabeth.sex
```

vrátí `"F"`.

Vytvořené objekty včetně vztahů zachycuje obrázek 11.



Obrázek 11. JavaScript: Ukázka

4.7. Agora

Agora je čistě objektový programovací jazyk s velkým důrazem na posílání zpráv. Jazyk je podobný jazyku Self, který je popsán v podkapitole 4.1.

Agora byla vyvíjena v devadesátých letech minulého století v Software Languages Lab pod Vrije Universiteit Brussel.

Následující text čerpá hlavně z [11] dále z [9] a [10].

První část popisuje objekty Agory rozdělené na veřejnou a soukromou část. Následuje odstavec zabývající se výrazy jazyka se speciálním způsobem vyhodnocování, které se nazývají Reifiery. Dále budou prozkoumány způsoby vytváření objektů pomocí mixinmetod a klonovacích metod a klonovacích atributů. Následuje odstavec o líném vyhodnocování. Závěrečný odstavec nahlíží do vnitřní struktury jazyka, která se skládá z metaobjektů.

Objekty Objekty Agory jsou tvořeny veřejnou a soukromou částí. Každá z těchto částí se skládá ze slotů. Veřejná část je přístupná okolním objektům a tvoří rozhraní objektu. Přístup do soukromé části objektu mají jen jeho metody. Toho je dosaženo následujícím způsobem hledání obsluhy zprávy.

Obsluha zpráv poslaných implicitnímu příjemci se hledá v soukromé části objektu. Obsluha zprávy s explicitním příjemcem se hledá ve veřejné části.

Příklad kódu metody hledající obsluhu zprávy `x` v soukromé části objektu:

```
x + 2
```

Oproti tomu tento kód hledá obsluhu zprávy `x` ve veřejné části objektu:

```
point x + 2
```

Reifier zprávy Reifier zprávy zavádějí speciální způsob vyhodnocování.

Reifier zprávy se ze syntaktického hlediska nijak neliší od posílání zpráv. Reifier zprávy budeme v kódu zvýrazňovat tučným písmem.

Následující výraz ukazuje použití reifier zprávy **variable:**. Ta slouží k přidání atributu do soukromé části objektu.

```
amount variable: 5000
```

Tento výraz přidá slot `amount` s hodnotou 5000 do soukromé části objektu, ve kterém byl vyhodnocen. Dále je do soukromé části přidána metoda `amount:` sloužící k nastavení tohoto atributu.

Reifier zprávy mají nižší prioritu vyhodnocování, než obyčejné zprávy.

Proto výraz

p **variable:** Point x: 10 y: 20

je vyhodnocen

(p) **variable:** (Point x: 10 y: 20)

Vytváření objektů z ničeho Reifier zprávy nahrazují syntaktické prvky jazyka Self pro vytváření nových objektů z ničeho.

Objekt lze jednoduše vytvořit napsáním výrazů jazyka oddělených středníkem mezi hranaté závorky. Výrazy se vyhodnotí v kontextu nově vytvořeného objektu.

Příklad z ničeho vytvořeného objektu:

```
[ element variable: null ; next variable: null]
```

Pro přidání metody do veřejné části objektu se používá reifier zpráva **method:**

Výraz

```
increment: amount method: counter: counter + amount
```

přidá metodu `increment:` s argumentem `amount` a kódem `counter: counter + amount`.

Pro změnu části objektu, do které se má atribut nebo metoda přidat, slouží reifiery **public** (veřejná část) a **local** (soukromá část).

Příklad definice veřejného atributu:

```
x public variable: 3
```

a soukromé metody:

```
m: n local method: self n
```

Použití reifieru **functional** v definici atributu potlačí vytváření metody pro jeho nastavení. Vznikne tak atribut pouze pro čtení.

Příklad vytvoření atributu pouze pro čtení:

```
x functional variable: 10
```

Mixinmetody Jednou z možností, jak vytvářet rozšíření objektů, je použít mixinmetody. Mixinmetody vytvářejí potomka příjemce. V jeho kontextu se vyhodnotí kód mixinmetody. Mixinmetody se vytvářejí umístěním reifieru **mixin** před reifier **method:**

Příklad mixinmetody:

```
colorMixin mixin method: [color public variable: "red"]
```

Vyhodnocení kódu

```
coloredObject variable: self colorMixin
```

způsobí zavolání mixinmetody. Ta vytvoří potomka kořenového objektu a přidá do něj nový slot `color`.

Klonovací metody Stejně jako rozšíření, tak i klonování je uskutečněno posláním zprávy. Objekt se tak sám může rozhodnout, jakým způsobem bude klonován. Tento způsob je vyjádřen v klonovací metodě. Kód klonovací metody se vyhodnotí v klonu příjemce.

Klonovací metody se přidávají jako obyčejné metody, kde nahradíme reifier `method:` za reifier **cloning:**

Předvedeme si klonování na příkladu jednoduchého účtu:

```
account variable:
[
amount variable: 5000;
deposit: val method: [amount: amount + val];
newAccount: initval cloning: [amount: initval];
]
```

Na účtu je částka 5000. Přidáme 100 na účet:

```
account deposit: 100
```

Vytvoříme nový účet s počáteční částkou 1000:

```
account2 variable: account newAccount: 1000
```

Klonovací metoda `newAccount` naklonuje objekt `account`. V klonu se nastaví atribut `amount` na hodnotu 1000.

Klonovací atributy Pokus o změnu klonovacího atributu vyvolá naklonování vlastníka atributu. Změna se provede v tomto klonu. Vlastník klonovacího atributu nebude změněn.

Klonovací atribut se vytvoří přidáním reifieru **cloning** v definici atributu.

Klonovací atributy si ukážeme v příkladu modelování bodu.

```
point mixin method:
[
x public local cloning variable: 0 ;
y public local cloning variable: 0 ;
x:xx y:yy method:(x:xx) y:yy ;
]
```

Použití kombinace reifierů `public local` způsobí, že atribut se bude nacházet jak v soukromé, tak ve veřejné části objektu.

Vytvoříme prototyp bodu zavoláním mixinmetody:

```
Point functional variable: self point
```

Mixinmetoda rozšíří kořenový objekt. Rozšíření se stane prototypem bodu.

Vytvoříme bod `p` klonováním prototypu:

```
p variable: Point x: 10 y: 20
```

Klonování se vyvolá změnou obou klonovacích atributů.

Vytvoříme bod `q` změnou klonovacího atributu:

```
q variable: p x: 100
```

Souřadnice bodu `q` jsou $[100, 20]$

Líné vyhodnocování Při přidání atributu do objektu je někdy potřeba pozdržet vyhodnocení výrazu určujícího hodnotu atributu do momentu, kdy je na ní dotázáno.

Pro dosažení tohoto efektu stačí před reifier **variable:** umístit reifier **lazy**.

Výsledkem vyhodnocení kódu

```
a lazy variable: self b * 2;
b variable: 10;
a ;
```

je číslo 20. Bez použití reifieru **lazy** by vyhodnocení kódu skončilo chybou, neboť používáme atribut `b` předtím, než jej definujeme.

Metaobjekty Uživatel jazyka Agora může ovlivnit jeho implementaci. Například je možné zavádět vlastní reifiery.

Implementace Agory je objektová. Je umožněno zaměňovat objekty Agory a objekty její implementace. Díky tomu si může uživatel strukturu překladače přizpůsobit nebo nahradit strukturami napsanými v Agoře.

Vnitřní objektová reprezentace objektu se nazývá *metaobjekt*. Jedinou zprávou, které metaobjekt rozumí, je zpráva `send`.

Dědičnost a klonování jsou uskutečněny pomocí metaobjektů.

4.8. Ambient

Ambient je prototypový čistě objektový systém napsaný pro jazyk Common Lisp.

Ambient byl vytvořen v roce 2008 na Université catholique de Louvain v Belgii. Jeho vývoj stále pokračuje.

Jeho nejvýraznějším rysem je, že zde není příjemce zprávy. Zpráva se posílá seznamu objektů. Každý z těchto objektů ovlivňuje hledání obsluhy zprávy. Tento přístup se nazývá *multiple-dispatch*. Následující odstavce objasňují jeho principy.

Kapitola čerpá z článku [16].

Objekty Pozorovatelné vlastnosti objektu jsou identita, známosti a chování. Identita objektu je neměnná. Známosti a chování se mohou měnit v čase. Tyto dvě vlastnosti tvoří stav objektu.

Podle konvence jména prototypů začínají symbolem `@`.

Ukázka vytvoření nového prototypu `@phone` a přidání dvou slotů:

```
(defproto @phone (clone @object))
(add-slot @phone 'calls (clone @call-manager))
(add-slot @phone 'speaker (clone @phone-speaker))
```

Klonování Objekty se klonují zasláním zprávy `clone`. Příklad vytvoření klonu prototypu `@phone`:

```
(clone @phone)
```

Zprávy Poslání zprávy je požadavek na interakci mezi objekty, které se nazývají argumenty zprávy. Není zde žádný příjemce poslané zprávy.

Ukázka poslání zprávy `receive` s argumenty `alices-call` a `bobs-phone`:

```
(receive alices-call bobs-phone)
```

Delegace Vícenásobná a cyklická delegace je podporovaná. Objekty spolu s delegačními odkazy tvoří orientovaný graf, který nazýváme *delegační*.

Pro vytvoření rozšíření objektu mu stačí poslat zprávu `extend`.

Vytvoření prototypu `@mobile-phone`, který bude delegovat na prototyp `@phone`:

```
(defproto @mobile-phone (extend @phone))
```

Metody Metody popisují interakci mezi objekty. Metoda má své jméno. Metoda může mít argumenty. Pro každý argument metody je určený objekt (většinou prototyp), na který je argument specializovaný. Nazýváme jej *specializátorem metody*.

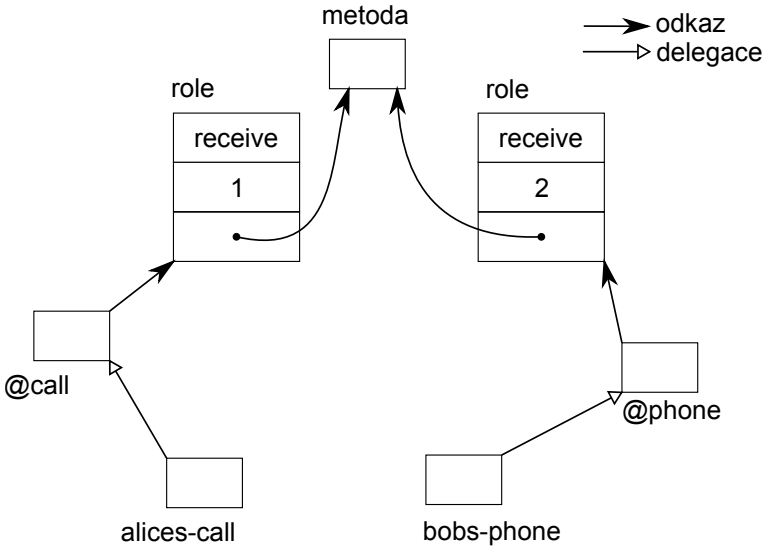
V následující ukázce definujeme metodu jménem `receive` s dvěma argumenty, které budou specializované na prototypy `@call` a `@phone`.

```
(defmethod receive ((call @call) (phone @phone))
  (adverise call phone)
  (enqueue call (incoming (calls phone))))
```

Použitelnost metod Pro obsluhu poslané zprávy můžeme použít danou metodu, jestliže se jméno zprávy shoduje se jménem metody a argumenty zprávy se hodí k specializátorům metody. Argument se hodí k specializátoru metody, jestliže je s ním totožný nebo je jeho potomkem. Jinými slovy v delegačním grafu existuje cesta z argumentu zprávy k specializátoru metody.

Role Odkaz mezi metodou a jejími specializátory se utváří prostřednictvím *rolí*. Role pro specializátor uchovává tři informace: název metody, pozici specializátoru a metodu.

Obrázek 12. zachycuje role pro metodu `receive` specializovanou na prototypy `@call` a `@phone`. Obrázek také ukazuje, že je tato metoda použitelná na argumenty `alices-call` a `bobs-phone`.



Obrázek 12. Role

Hledání obsluhy zprávy Může nastat situace, že pro poslanou zprávu je použitelných více metod. Pak se jako obsluha zprávy vybere metoda, která se nejvíce hodí k argumentům zprávy. Způsob výběru je popsán níže.

Vhodnost použitelných metod je určená *hodnotícími vektory*. Každý prvek hodnotícího vektoru je vzdáleností v delegačním grafu mezi argumentem zprávy a příslušným specializátorem metody. Například hodnotící vektor metody na obrázku 12. pro argumenty `alices-call` a `bobs-phone` je $(1, 1)$.

Pro výpočet vzdáleností v delegačním grafu se používá linearizační algoritmus. Jeho popis lze nalézt v [17] .

Hodnotící vektory se porovnávají lexikograficky. Tedy například vektor $(1, 2)$ předchází vektor $(2, 1)$. Jako obsluha zprávy se vybere metoda s nejmenším hodnotícím vektorem.

Důsledkem toho budou metody, které se více shodují v dřívějších argumentech, pokládány za více vhodné. Toto chování je ospravedlněno pozorováním, že více důležité argumenty jsou v metodách uváděny dříve.

5. Návrh vlastního jazyka

Tato kapitola popisuje návrh vlastního jazyka, který nese název Eggs.

Cílem bylo navrhnout prototypový jazyk tak, aby se v něm dala vytvořit jednoduchá aplikace. Z důvodu snadné konstrukce jazyka je cílem, aby byl jeho návrh jednoduchý. Některé volby návrhu jazyka byly rozhodnuty s přihlédnutím k jeho praktické potřebě.

Hlavní inspirací Eggs je jazyk Self, který je představen v podkapitole 4.1. Stejně jako Self je i Eggs čistě objektový. To vede k jednoduššímu návrhu.

V jazyce NewtonScript, který je prozkoumaný v podkapitole 4.3., je představen dvojí druh delegace. Jeden pro vyjádření vztahu přímý potomek-rodič a druhý pro vztah prvek-kontejner. Eggs přináší zobecnění tohoto přístupu. Delegace zde může vyjadřovat rozmanité vztahy mezi objekty. Řešení delegace bylo vymyšleno.

Se zobecněným systémem delegace souvisí i následující významná vymyšlená vlastnost Eggs. Zprávy mohou rozhodovat o tom, jakým způsobem budou posílány. Například jsou zde zprávy, které při nenalezené obsluze zprávy neskončí chybou.

Z jazyka Agora, který je představený v kapitole 4.7., Eggs přebírá způsob, kterým lze měnit vyhodnocování některých výrazů.

Nejprve jsou představeny hlavní pilíře jazyka. Poté následuje část o technikách programování. V závěrečné části se nalézá několik poznámek o realizaci jazyka.

5.1. Objekty

Jazyk je čistě objektový. To znamená, že všechna data jsou objekty ve smyslu objektového programování.

Objekt se, stejně jako v jazyce Self, skládá ze *slotů*. O slotech pojednává část podkapitoly 2.1. Na rozdíl od Agory a stejně jako v Selfu jsou všechny sloty veřejné. Důvodem je jednoduchost řešení.

Ze Selfu byla přejata i základní struktura světa. Proto se kořenový objekt světa nazývá `lobby`.

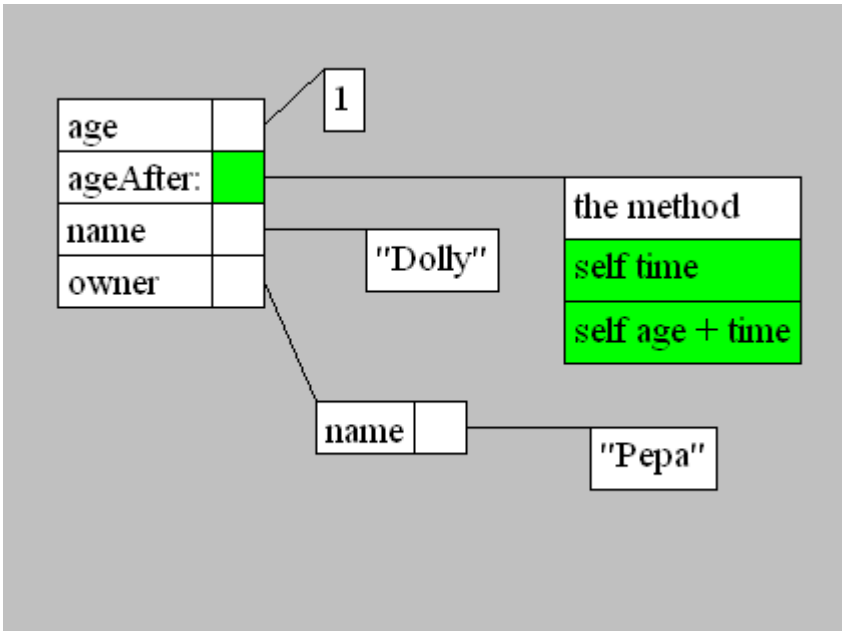
Nově vymyšlenou technikou je způsob odebírání slotů z objektu. Slot z objektu odebereme tak, že do něj vložíme speciální objekt, který se jmenuje *inherit*. Tím jediná operace nastavení hodnoty slotu zajišťuje přidávání slotů, změny obsahu slotů i odebírání slotů.

Metody mají kromě kódu i seznam jmen argumentových slotů. Jejich účel bude objasněn níže v části o vyhodnocování metod.

Objekty zobrazujeme na obrázcích jako obdélníky. Sloty objektu se zobrazují jako řádky v tomto obdélníku. V levé části řádku se nalézá jméno slotu, v pravé čtvereček zastupující hodnotu slotu.

Nemusí se zobrazovat všechny sloty objektu. Budeme zobrazovat jen ty sloty, které jsou pro momentální účel důležité.

Na obrázku 13. se nalézá několik objektů zobrazených jako obdélníky. Obdélník nacházející se nejvíce vlevo zobrazuje čtyři sloty objektu.



Obrázek 13. Eggs: Zobrazení objektů a metod

Pro vyjádření skutečnosti, že objekt má slot s určitou hodnotou, vede linka od čtverečku zastupující hodnotu slotu k obdélníku zobrazujícího objekt, který je ve slotu uložený.

Objekt, který je na obrázku nejvíce vlevo, má ve slotu `age` uloženo číslo jedna.

Čtverečky zastupující hodnotu slotů, které obsahují metodu, jsou zbarvené zeleně. Obdélník metody může v dolní části obsahovat dva zelené řádky. V prvním z nich se nalézají jména argumentových slotů metody a v druhém kód metody.

Objekt, který je nejvíce vlevo, má ve slotu `ageAfter:` metodu (the method). Jména argumentových slotů metody jsou `self` a `time`. Kód metody je

```
self age + time
```

V prvním řádku obdélníku se může nalézat textový popis objektu.

Například metoda má popis „the method“. Řetězec ve slotu `name` objektu, který se nalézá nejvíce vlevo, má textový popis „Dolly“.

5.2. Dědičnost

Delegace prototypových jazyků, jak o ní mluví podkapitola 2.4., se v jazyce Eggs nazývá dědičnost.

Rodiče objektu jsou uspořádáni podle důležitosti.

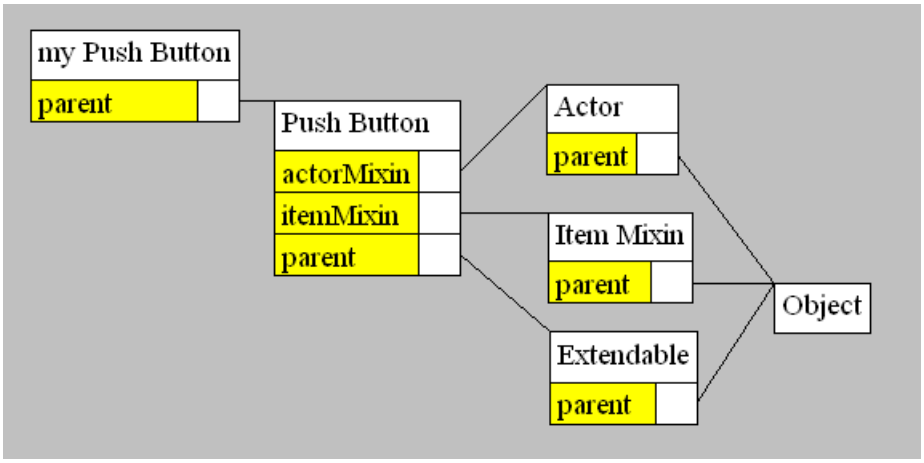
Rodiče objektu i jejich důležitost lze za běhu programu libovolně měnit.

Rodiče objektu jsou uloženi ve speciálních slotech, kterým říkáme *rodičovské*. Rodičovské sloty mají v obrázcích žluté pozadí. Dále jsou v obrázcích dříve zobrazené rodičovské sloty s důležitějšími rodiči.

Pro dosažení většího sdílení mezi objekty bývá modelovaná věc rozdělena do více objektů provázaných dědičností. Intuitivně si můžeme představit, že tyto objekty složíme v jediný objekt. Složený objekt bude plně zastupovat modelovanou věc.

Podobně jako v NewtonScriptu objekt a všechny jeho předkové tvoří *entitu*. Říkáme, že entita je určena tímto objektem.

Entita moje tlačítko (my Push Button) je znázorněná na obrázku 14. Rodiče tlačítka (Push Button) jsou uspořádáni podle důležitosti v tomto pořadí: herec (Actor), prvek mixin (Item Mixin) a rozšiřitelný objekt (Extendable).



Obrázek 14. Moje tlačítko a jeho předkové

Seznam jmen rodičovských slotů objektu je uložen v jeho skrytém slotu `parentSlots`.

5.3. Delegování

Delegace má v Eggs trochu jiný význam než v ostatních prototypových jazycích. Tam delegace vyjadřuje vztah přímý potomek-rodič. V Eggs delegací zachycujeme různé vztahy objektů, například vztah prvek-kontejner.

Zprávy jsou objekty. Před jméno zprávy se uvádí symbol „$“. Předci zpráv jsou uspořádáni do hierarchie, jejíž část je zachycena obrázkem 22. Říkáme, že *zpráva je* určitého *typu*, jestliže je jeho potomkem. Například zpráva `$buttonPress` je typu `eventMessage`.

Objekt může mít *delegáty*. Každý delegát je určitého typu. Typ delegáta je objekt.

Delegát objektu je *platný pro zprávu*, je-li zpráva stejného typu jako je typ delegáta. Například delegát typu `containerMessage` je platný pouze pro zprávy typu `containerMessage`. To jsou zprávy, jejichž předek je objekt `containerMessage`.

Uvažujme například tlačítko vložené do grafického okna. Grafické okno je delegátem typu kontejner (`containerMessage`) tlačítka. Tento delegát je platný pro zprávu `$update`, ale ne pro zprávu `$+`.

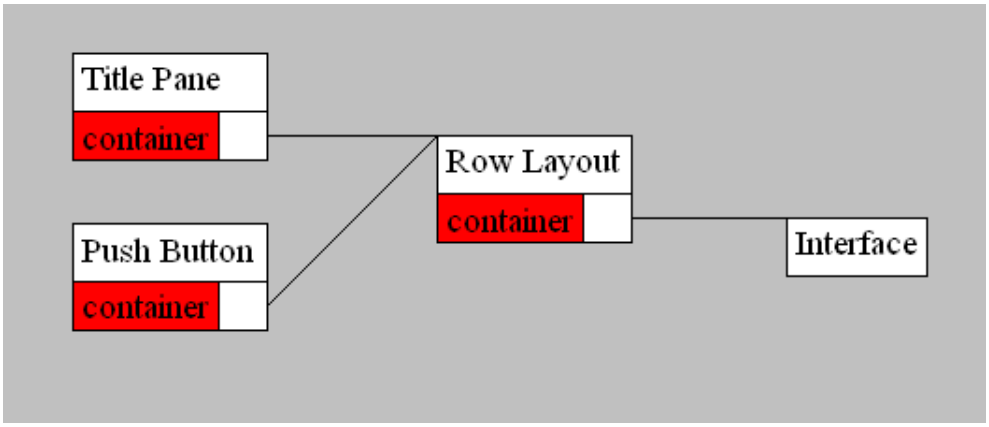
Delegáti objektu nejsou nijak uspořádáni.

Za běhu programu je možno libovolně měnit delegáty i jejich typy.

Delegáti objektu jsou uloženi ve speciálních slotech, které nazýváme *delegující sloty*. Delegující slot má typ. Objekt uložený v delegujícím slotu určitého typu je delegátem tohoto typu.

Delegující sloty různého typu mají v obrázcích různé barvy pozadí. Například červené pozadí značí delegující sloty typu kontejner (`containerMessage`) a modré typu obsah (`contentMessage`).

Obrázky 15. a 16. nahlížejí na stejnou situaci mezi objekty z různých pohledů. V této situaci jsou textový grafický prvek (Title Pane) a tlačítko (Push Button) uspořádány do řádku (Row Layout) a tento řádek je vložen do grafického okna (`Interface`). Obrázek 15. zachycuje delegáty typu kontejner (`containerMessage`) a obrázek 16. delegáty typu obsah (`contentMessage`).



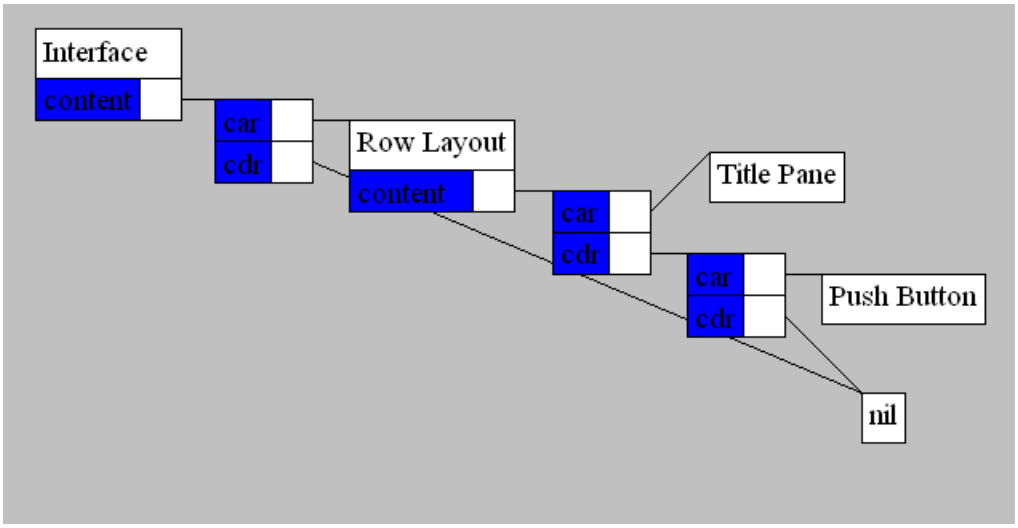
Obrázek 15. Eggs: Delegáti typu kontejner

Seznam jmen delegujících slotů objektu společně s jejich typy je uložen v jeho skrytém slotu `delgateSlots`.

5.4. Posílání zpráv

Mechanizmus posílání zpráv je jediným výpočetním primitivem. Posílání zpráv v jazyce Eggs vychází ze základů posílání zpráv uvedených v podkapitole 2.2.

Vyhodnocení metody Jak bylo řečeno, vyhodnocení metody má k dispozici příjemce a argumenty poslané zprávy. *Aktivace metody* je objekt, který slouží k jejich uložení. Podrobněji bude vysvětleno níže.



Obrázek 16. Eggs: Delegáti typu obsah

Vyhodnocení metody spočívá ve vyhodnocení jejího kódu v kontextu nově vytvořené aktivace metody. Hodnota vyhodnoceného kódu bude i vrácenou hodnotou vyhodnocení metody.

Metoda má určený *lexikálně nadřazený objekt*. Často jím bývá kořenový objekt `lobby`.

Připomeňme, že metoda má seznam jmen argumentových slotů.

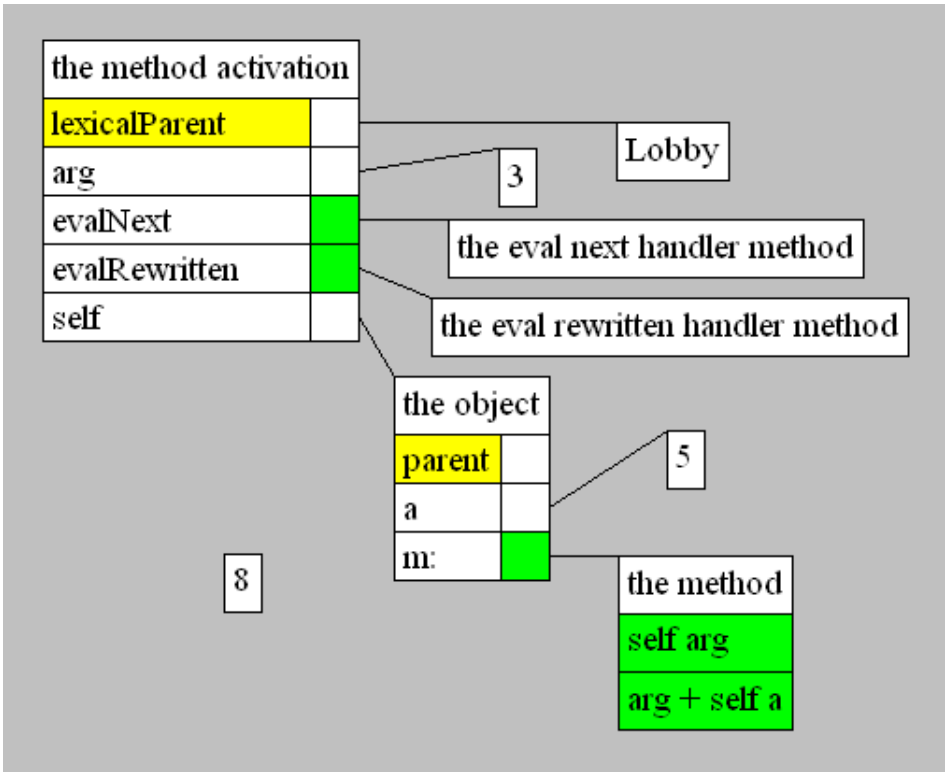
Aktivace metody vznikne rozšířením lexikálně nadřazeného objektu a vložením příjemce a argumentů poslané zprávy do jeho slotů. Jména těchto slotů budou postupně brána ze seznamu jmen argumentových slotů. Tedy příjemce zprávy bude uložen ve slotu, jehož jméno je prvním prvkem seznamu jmen argumentových slotů. Dále například jméno slotu obsahujícího první argument poslané zprávy se bude shodovat s druhým prvkem seznamu jmen argumentových slotů.

Dále aktivaci metody mohou být přidány dvě metody s názvy `evalNext` a `evalRewitten`. Jejich účel je objasněn níže, v části o vyhodnocování dalších a přepsaných obsluh.

Obrázek 17. zachycuje metodu (the method). Ta je uložena ve slotu `m:` objektu (the objekt). Dále je zde zobrazena aktivace metody (the method activation) vytvořená vyhodnocením této metody. Metoda byla vyhodnocena v důsledku poslání zprávy `m:` objektu (the object) s argumentem 3. Kód metody se v kontextu aktivace metody vyhodnotil na číslo 8.

Seznam jmen argumentových slotů metody je uložen ve skrytém slotu `argumentSlots` metody. Kód metody se nalézá ve skrytém slotu `code` metody. Lexikálně nadřazený objekt metody může být uložen v jejím slotu `lexicalParent`. Není-li tomu tak, pak lexikálně nadřazený objekt metody je objekt `lobby`.

Eggs na rozdíl od Selfu nezávádí primitivní zprávy. Místo nich zde existují *primitivní metody*. Primitivní metody jsou spojeny s primitivní operací. Vyhod-



Obrázek 17. Eggs: Vyhodnocení metody

nocení primitivních metod spočívá ve vykonání příslušné primitivní operace.

Například obecný objekt má slot `setSlot:Value:` obsahující primitivní metodu, která je spojena s primitivní operací nastavení hodnoty slotu.

Hledání obsluhy v entitě Chceme najít obsluhu zprávy v entitě. Hledání začíná u objektu, který entitu určuje.

Pokud prohledávaný objekt obsahuje slot stejného jména jako je jméno zprávy, pak je obsah tohoto slotu hledanou obsluhou. Jinak hledání pokračuje v rodičích objektu, má-li objekt nějaké. Připomeňme, že rodiče objektu jsou seřazeni podle důležitosti. Hledání pokračuje od nejdůležitějšího rodiče k méně důležitým. Celý postup hledání se u rodičů objektu opakuje.

Tímto způsobem se prohledá celá entita určená příjemcem zprávy. Každý objekt tvořící entitu je prohledán nejvýše jednou.

V Selfu nejsou rodiče objektu uspořádáni podle důležitosti. Při hledání obsluhy jsou v Selfu prohledáni všichni rodičové objektu. Může se stát, že se nalezne více obsluh v různých rodičích objektu. V takové situaci poslání zprávy skončí chybou.

Uspořádáním rodičů objektu podle důležitosti umožňuje Eggs používat některé techniky, které by v Selfu nebyly možné. Například obsluha zprávy `extend`

zajišťující rozšíření objektu, který má více rodičů. Každý rodič může zavádět obsluhu zprávy `extend` a tím ovlivňovat způsob rozšíření.

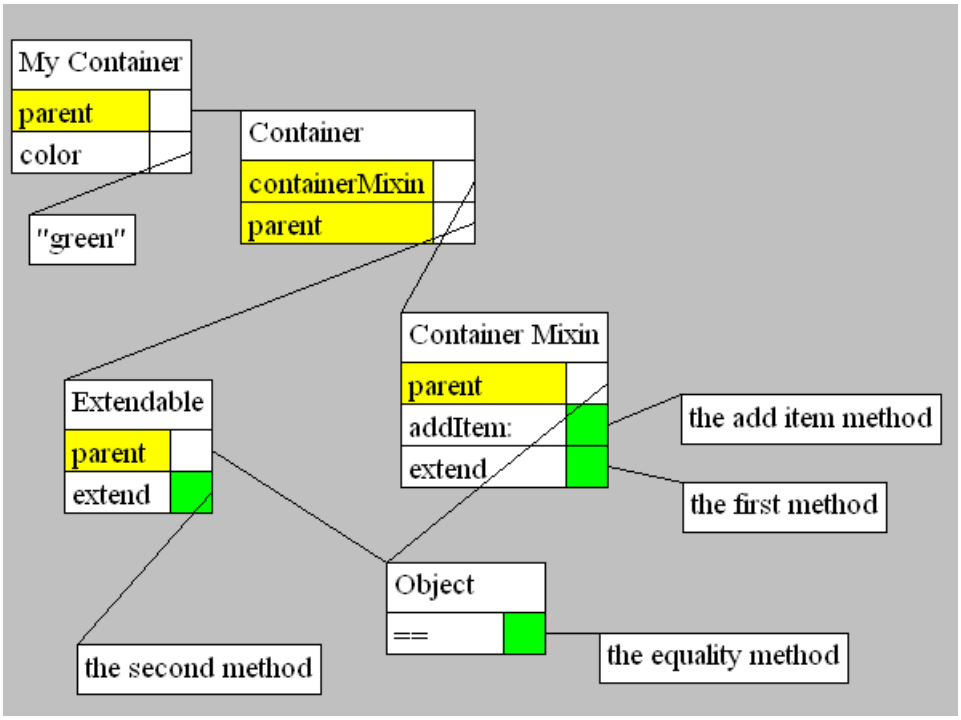
Uvažujme entitu můj kontejner (My Container) zachycenou na obrázku 18. Budeme hledat obsluhy zpráv v této entitě.

Obsluha zprávy `color` je řetězec `"green"`, protože tento řetězec je obsažen ve slotu stejného jména jako je jméno zprávy.

Metoda přidávající prvek (the add item method) je obsluhou zprávy `addItem:`, protože kontejner mixin (Container Mixin) je předkem mého kontejneru (My Container) a obsahuje vhodný slot.

Obsluhou zprávy `extend` je první metoda (the first method) a ne druhá metoda (the second method). Důvodem toho je, že kontejner mixin (Container Mixin) je důležitější rodič kontejneru (Container) než rodič rozšiřitelný objekt (Extendable). Připomeňme, že pořadí rodičovských slotů zobrazeného objektu respektuje jejich důležitost.

Metoda rozhodující rovnost objektů (the equality method) je obsluhou zprávy `==`.



Obrázek 18. Eggs: Hledání obsluhy v entitě

Hledání obsluhy Nejprve se prohledá entita určená příjemcem zprávy.

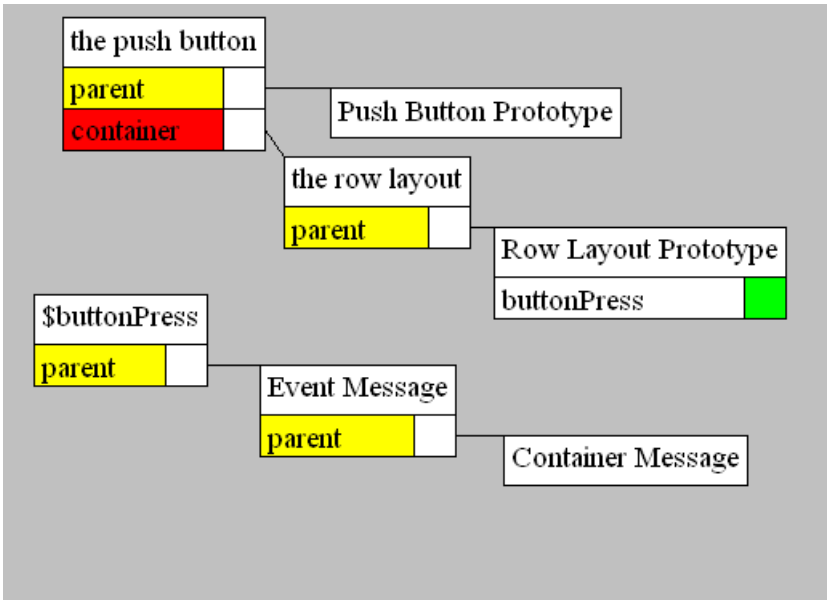
Může nastat situace, že v prohledávané entitě nebyla nalezena obsluha zprávy. Objekt určující prohledávanou entitu může mít pro poslanou zprávu platné delegáty. V takové situaci hledání pokračuje v entitách určených jeho delegáty.

Delegáti nejsou seřazeni podle důležitosti. Proto se prohledávají všichni. V důsledku toho se může stát, že se nalezne obsluha ve více delegátech. Pak jsou výsledkem hledání všechny tyto obsluhy.

Může se stát, že žádná obsluha nebyla nalezena.

Vlastník obsluhy je objekt určující entitu, ve které byla obsluha nalezena.

Uvažujme situaci popsanou obrázkem 19. Tlačítku (the push button) byla poslána zpráva `buttonPress`. Nejprve se prohledá entita určená tlačítkem. To znamená tlačítko a jeho prototyp (Push Button Prototype). Zde se nenalezne žádná obsluha. Zpráva `buttonPress` je potomkem kontejnerové zprávy (Container Message). Z tohoto důvodu má tlačítko pro zprávu `buttonPress` jednoho delegáta. Tím je řádkové uspořádání (the row layout). Hledání pokračuje u entity řádkové uspořádání. Prototyp řádkového uspořádání (Row Layout Prototype) má ve slotu `buttonPress` uloženou metodu. Tato metoda je hledanou obsluhou zprávy. Řádkové uspořádání je vlastníkem obsluhy.



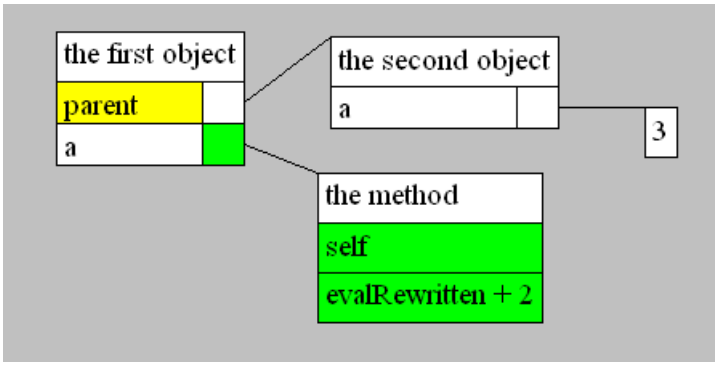
Obrázek 19. Eggs: Hledání obsluhy

Vyhodnocení přepsaných a dalších obsluh Metoda se ve svém kódu může rozhodnout zavolat obsluhu zprávy, kterou přepsala.

Poslání zprávy `evalRewritten` bez uvedení příjemce v kódu metody vyvolá pokračování hledání obsluhy v entitě. Nalezená obsluha je vyhodnocena. Nenalezení obsluhy vede k chybě.

Představme si situaci zachycenou obrázkem 20. Poslání zprávy „`a`“ prvnímu objektu (the first object) vede k vyhodnocení kódu metody (the method). V tomto kódu je poslána zpráva `evalRewritten`. To vyvolá pokračování hledání

obsluhy zprávy „a“. Obsluha se nalezne v druhém objektu (the second object). Touto obsluhou je číslo 3. To je i návratová hodnota zprávy `evalRewritten`. Kód metody (the method) se vyhodnotí na číslo 5.



Obrázek 20. Eggs: Vyhodnocení přepsané obsluhy

Podobnou technikou je možné vyhodnocovat další obsluhy uložené v delegátech. Za tímto účelem se zavádí zpráva `evalNext`. Po jejím zaslání pokračuje hledání obsluhy v delegátech příjemce. Přepsané obsluhy jsou touto technikou přeskočeny.

Zabývejme se situací zachycenou na obrázku 21. Poslání zprávy `needUpdate` prvnímu objektu (the first object) vede k vyhodnocení kódu `evalNext + 3`. Posláním zprávy `evalNext` pokračuje hledání obsluhy zprávy v delegátech pro zprávu `needUpdate`. První objekt má jediného delegáta pro zprávu `needUpdate` a tím je druhý objekt (the second object). Zde se nalezne obsluha číslo 2. Kód se proto vyhodnotí na číslo 5. Třetí objekt (the third object) obsahující přepsanou obsluhu zprávy `needUpdate`, je při prohledávání přeskočen.

Zprávy Zprávy mohou ovlivnit způsob, jakým jsou posílány.

Zprávy, které jsou následujících typů, mění standardní chování posílání zpráv.

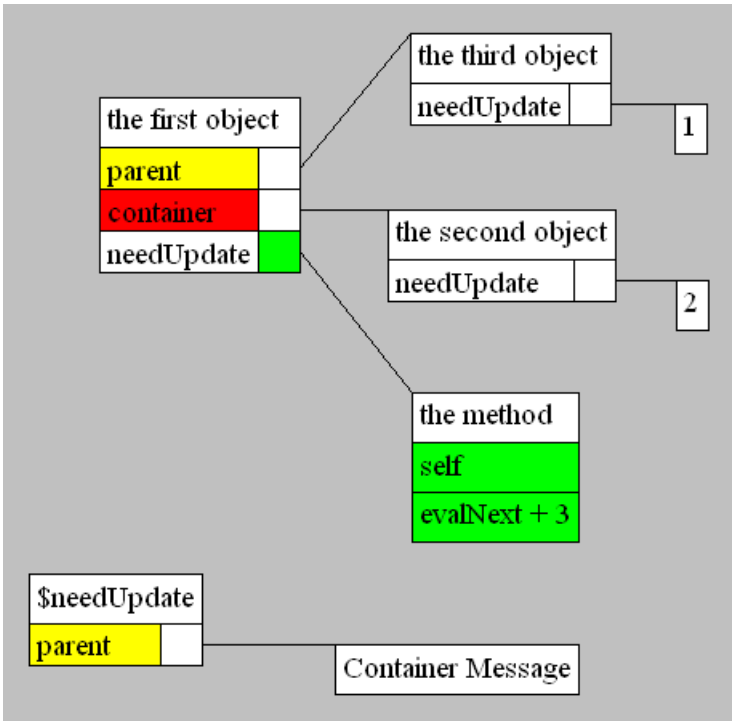
Zprávy typu `containerMessage` a typu `contentMessage`, nevyvolají chybu v případě, že nebyla nalezena jejich obsluha. Místo toho vrátí objekt `nil`. Tyto zprávy také do aktivace metod vkládají místo příjemce zprávy vlastníka obsluhy.

Zprávy typu `contentMessage`, v situaci, kdy je nalezeno více obsluh této zprávy, místo vyvolání chyby vyhodnotí všechny obsluhy a vrátí seznam všech vrácených hodnot.

Zprávy typu `eventMessage`, vkládají do slotu `sender` aktivace metody příjemce zprávy. Tyto zprávy zastávají úlohu událostí známých z objektových jazyků.

5.5. Kód jazyka

Kód se skládá z výrazů oddělených tečkou.



Obrázek 21. Eggs: Vyhodnocení další obsluhy

Příklad kódu o dvou výrazech:

```
evalNext . self isValid: inherit
```

Výrazy jazyka mohou obsahovat atomy.

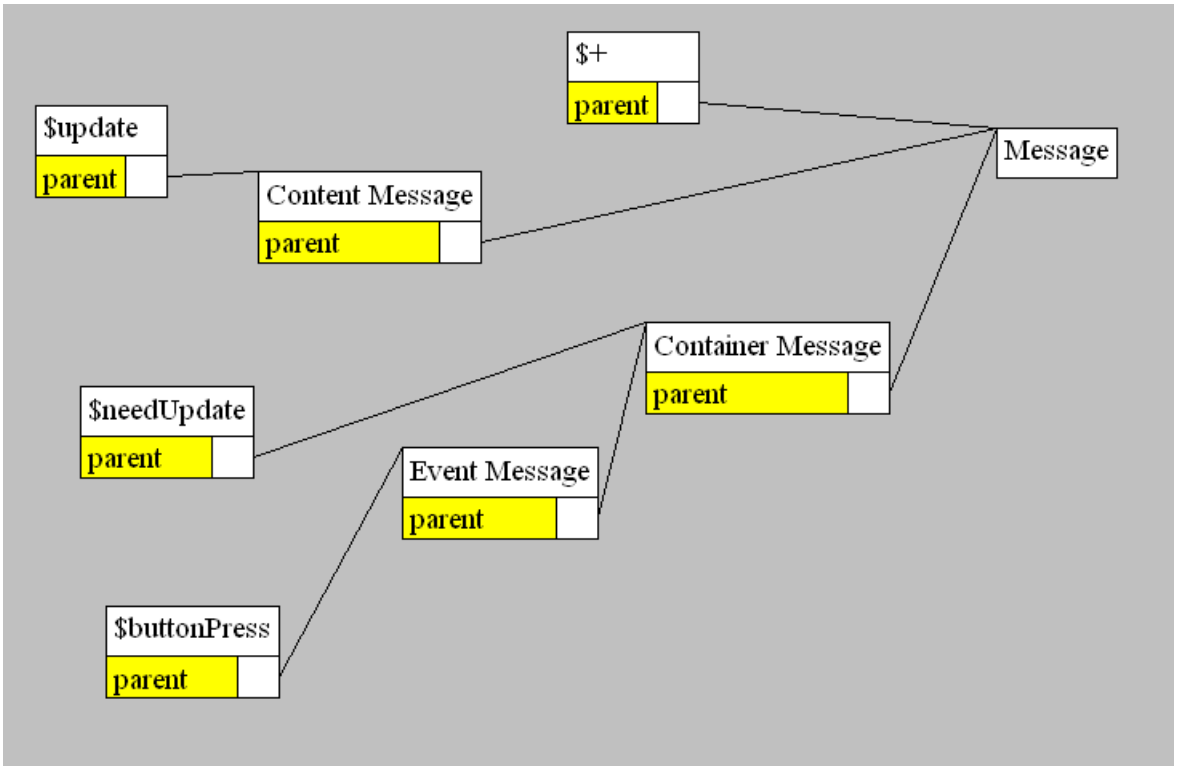
Atomy Mezi *atomy* patří čísla, řetězce a zprávy.
Příklady čísel:

```
4
3.5
0
```

Příklady řetězců:

```
""
"Máma mele maso"
"Řetězec
na
více
řádků"
```

Před jméno zprávy uvedeme symbol „$“. Příklady zpráv:



Obrázek 22. Eggs: Část hierarchie zpráv

```
$stage
$==
$insertInto:
```

Posílání zpráv Způsob posílání zpráv vychází z jazyka Self a zabývá se jím část podkapitoly 4.1. Popíšeme pouze rozdíly oproti Selfu.

Identifikátor unární zprávy končí písmenem nebo číslovkou.

Operátor binární zprávy nekončí číslovkou, písmenem ani dvojtečkou. Binární zprávy jsou asociativní zleva doprava.

Výraz

```
3 + 4 * 5
```

je ekvivalentní výrazu

```
(3 + 4) * 5
```

Klíče klíčových zpráv mohou začínat libovolnými písmeny. Klíčové zprávy nejsou asociativní.

Výraz

```
1 min: 4 min: 5
```

pošle zprávu `min:min:` objektu 1 společně s argumenty 4 a 5.

Vyhodnocení výrazu Výraz se *vyhodnocuje* v kontextu objektu. Tento objekt je implicitním příjemcem ve výrazu poslaných zpráv. Vyhodnocení výrazu vrací hodnotu, která je objektem. Jestliže je objekt vrácenou hodnotou vyhodnocení výrazu, může říci, že se výraz na něj vyhodnotil. Při imperativním charakteru výrazu můžeme místo vyhodnocení výrazu mluvit o *vykonání příkazu*. Vyhodnocení výrazu je závislé na kontextu, ve kterém se vyhodnocuje.

Například vyhodnocení výrazu

```
+ 3
```

v kontextu čísla 4 vede k poslání zprávy `+` číslu 4 s argumentem číslo 3. Výraz se vyhodnotí na číslo 7.

Vyhodnocení kódu Podobně jako výraz se i kód vyhodnocuje v kontextu objektu. Vyhodnocení spočívá v postupném vyhodnocení jeho výrazů v kontextu tohoto objektu. Kód se vyhodnocuje na hodnotu, která je objektem. Hodnota posledního výrazu kódu je hodnotou celého kódu.

Výrazy kódu lze místo tečkou také oddělovat závorkami.

Kód

```
evalNext . self isValid: inherit
```

lze ekvivalentně přepsat na

```
(evalNext) (self isValid: inherit)
```

Reifiery Reifiery jsou zprávy, které zavádějí svůj vlastní způsob vyhodnocování. Reifiery jsou v kódu vyznačeny tučně.

Například reifier **method:** slouží k přidání metody do objektu.

Výraz

```
(inc) method: (self + 1)
```

při vyhodnocení přidá do objektu, který je jeho kontextem, metodu `inc` s kódem `self + 1`. Kdyby zpráva `method:` nebyla reifier zprávou, pak by vyhodnocení výrazu skončilo chybou: „Objekt `lobby` nerozumí zprávě `inc`.“

Reifier zprávy se často vyhodnocují tak, že expandují na jiný výraz, který je poté vyhodnocen.

Například výše zmíněná reifier zpráva expanduje na:

```
setSlot: $inc Value: ((self) makeMethod: (self + 1))
```

Myšlenka reifierů byla převzata z Agory. Rozdíl reifierů v Eggs oproti reifierům Agory je v tom, že mají stejnou prioritu jako obyčejné zprávy. Díky tomu lze v Eggs zaměňovat reifiery a obyčejné zprávy. Toho se využívá například v optimalizaci.

5.6. Techniky programování

5.6.1. Přímá manipulace s objekty

V Selfu slouží k přímé manipulaci s objekty zrcadla. O zrcadlech pojednává odstavec v podkapitole 4.1. Na rozdíl od toho v jazyce Eggs je možné s objekty manipulovat přímo posíláním zpráv. Existují zprávy, které mění vnitřní stav objektu nebo zjišťují informace o jeho vnitřním stavu.

Například zpráva `setSlot:Value:` slouží ke vkládání objektu do slotu.

Ukázka vložení čísla 5 do slotu „`a`“ příjemce zprávy:

```
setSlot: $a Value: 5
```

Příklad odebrání slotu z příjemce zprávy:

```
setSlot: $a Value: inherit
```

Zpráva `setSlot:Value:` zajišťuje přidávání slotů, změnu obsahu slotů i odebírání slotů.

Přímá manipulace objektů souvisí s tvořením objektů pomocí reifierů.

Například objektu o dvou slotech vytvoříme výrazem:

```
makeObject inEval: (x = 1 . y = 2)
```

Ten expanduje na výraz:

```
(makeObject setSlot: $x Value: 1) setSlot: $y Value: 2
```

Reifier **inEval:** vyhodnotí argument v kontextu vyhodnoceného příjemce.

5.6.2. Vytváření objektů

Nové objekty lze tvořit pouze rozšířením existujících objektů.

Pro rozšíření objektu mu stačí poslat zprávu `extend`. Například rozšíření grafického okna se provede následovně: `interface extend`. Tímto způsobem lze rozšířit pouze objekt `extendable` a jeho potomky. Některé objekty nelze rozšířit. Například čísla.

Rozšíření obecného objektu `object` se provede zasláním zprávy `makeObject` libovolnému objektu. Tímto způsobem se vytvoří nový obecný objekt připravený pro další úpravu.

5.6.3. Bloky

K potlačení vyhodnocování kódu se používají *bloky*.Bloky jsou objekty. Bloky se tvoří pomocí reifieru # a mají následující tvar:

```
( args # ( body ))
```

Kde *args* jsou argumenty bloku a *body* je kód.

Blokovaný kód se vyhodnotí posláním speciální zprávy bloku. Tato zpráva je závislá na počtu argumentů bloku.

Příklad vytvoření bloku bez argumentů a jeho vyhodnocení:

```
(#( 1 + 3)) value
```

Blok si pamatuje prostředí, ve kterém byl vytvořen. Proto ho lze použít jako lexikální uzávěr.

Vyhodnocení výrazu

```
((a # ( b # ( a + b))) value: 1) value: 2
```

vrací číslo tři.

Příklad použití bloků pro větvení programu:

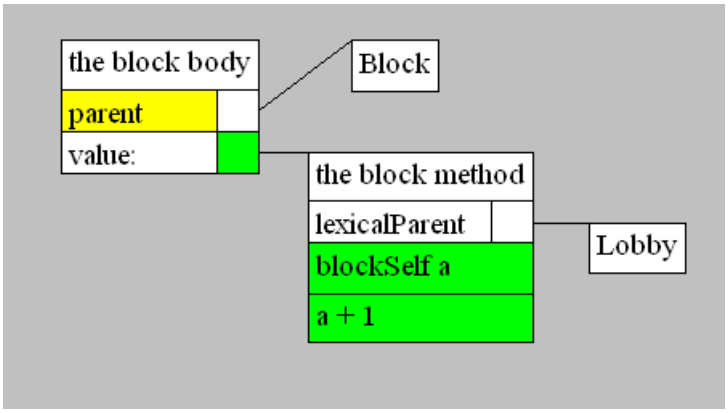
```
(item == self first) ifTrue: (# (self rest remove: item))
                     False: (# (self first cons: (self rest remove: item)))
```

Další příklad znázorňuje použití bloku s argumentem:

```
stage content map: (item # (item display))
```

Tento kód zobrazí všechny prvky jeviště (Stage).

Na obrázku 23. je znázorněn blok vytvořený výrazem `(a # (a + 1))`. Blok se skládá ze dvou částí: těla bloku (the block body) a metody bloku (the block method). Metoda bloku je uložena ve slotu těla bloku. Jméno tohoto slotu je současně zprávou, která vyhodnotí blokovaný kód.



Obrázek 23. Eggs: Blok

5.6.4. Základní operace

Aritmetika Pro sčítání, odčítání, násobení a dělení čísel se používají binární zprávy `+`, `-`, `*`, `/` poslané číslu. Argument těchto zpráv musí být číslo. Binární zprávy nedodržují klasická pravidla priority aritmetických výrazů.

Proto například výraz `5 + 3 * 2` je ekvivalentní výrazu `(5 + 3 ) * 2` a vyhodnotí se na číslo 16.

Logika Objekt `nil` má pravdivostní hodnotu nepravda. Ostatní objekty mají pravdivostní hodnotu pravda. Pro negaci, konjunkci a disjunkci se používají zprávy `not`, `and:` a `or:`

Příklad logického výrazu jenž vrací nepravdu:

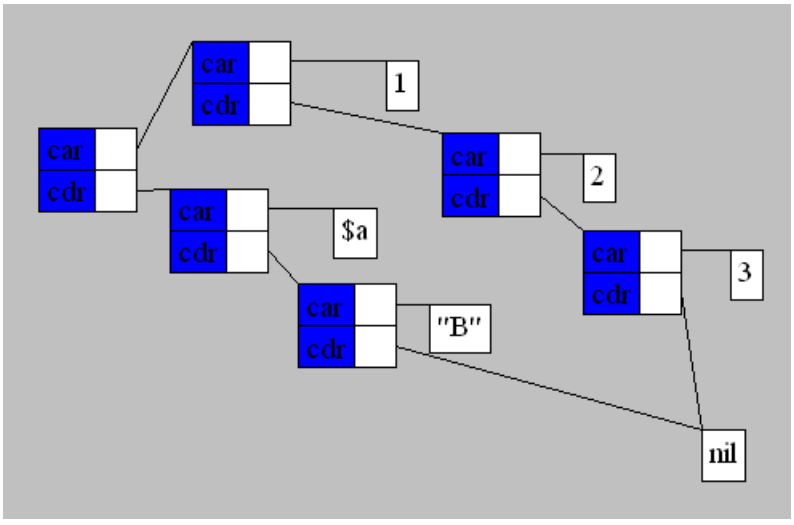
```
((true or: nil) not ) and: 4
```

Seznamy Objekt `nil` je prázdným seznamem. Neprázdné seznamy jsou objekty tvořené dvojicemi (`cons`). Dvojice je objekt se sloty `car` a `cdr`. Ve slotu `car` je uložen první prvek seznamu. Slot `cdr` obsahuje zbylé prvky seznamu, kromě prvního. Tyto sloty jsou delegující sloty typu obsah (`contentMessage`). To má následující důsledek. Je-li seznamu poslána zpráva typu obsah, pak je delegována na všechny prvky seznamu.

Do kódu lze vložit seznam tak, že se před závorky vloží symbol „`$`“. Prvky mezi závorkami budou prvky seznamu. Příklad seznamu o třech prvcích, z nichž první je opět seznamem:

```
$((1 2 3) a "B")
```

Tento seznam si lze prohlédnout na obrázku 24.



Obrázek 24. Eggs: Seznam

Pro přidání prvku do seznamu se používá zpráva `cons:`. Příklad vytvoření tříprvkového seznamu:

```
1 cons: (2 cons: (3 cons: nil))
```

Pro vykonání operace s každým prvkem seznamu se používá zpráva `map:`. Například přičtení dvojky ke každému prvku seznamu:

```
$(5 6 7) map: (n # (n + 2))
```

Výslednou hodnotou je seznam `$(7 8 9)`.

Seznamy lze filtrovat pomocí zprávy `filter:`. Například ponechání pouze sudých čísel v seznamu:

```
$(1 2 3 4 5 6) filter: (n # ( n isEven))
```

Výslednou hodnotou je seznam `$(2 4 6)`.

Zprávy `map:` a `filter:` vždy vytvoří nový seznam.

5.6.5. Hierarchie prvek-kontejner

Eggs umožňuje jednoduše podchytit vztah prvek-kontejner mezi objekty. K tomuto účelu se používají dva objekty `item` a `container`. Oba tyto objekty jsou rozšiřitelné. To znamená, že jejich potomky získáme jednoduše zasláním zprávy `extend`.

Mezi prvkem a jeho kontejnerem jsou oboustranně navázány vztahy delegování. Od prvku ke kontejneru je to vztah delegování typu kontejner (`containerMessage`). Od kontejneru k jeho prvkům vedou vztahy delegování typu obsah (`contentMessage`).

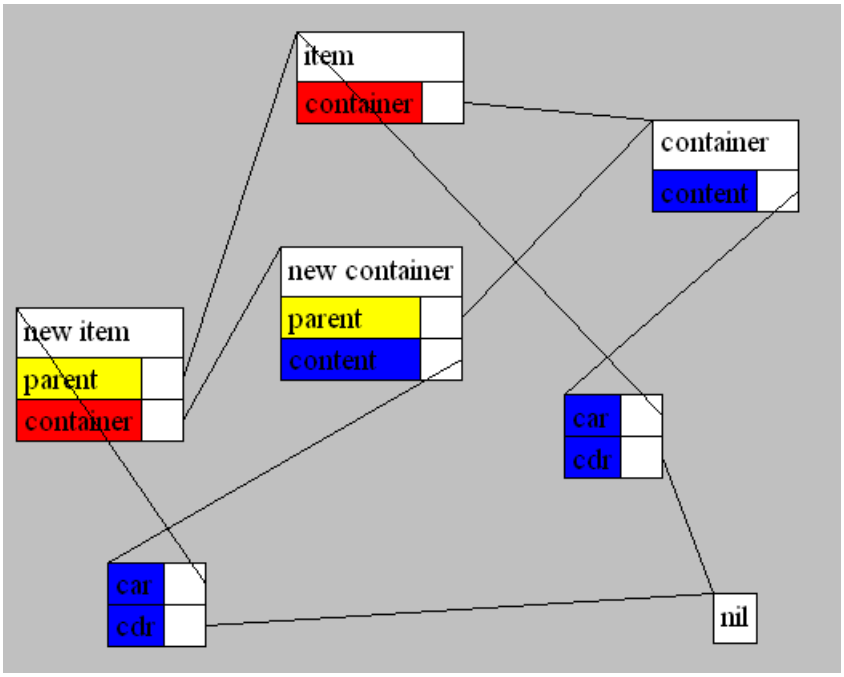
Rozšíření kontejneru vyvolá rozšíření jeho prvků. Tím bude provázán dědičností nejen nově vytvořený kontejner se starým, ale i sobě si odpovídající prvky.

Obrázek 25. vyobrazuje kontejner (container) s jedním prvkem (item) a jeho rozšíření (new container). Lze zde také najít rozšíření prvku (new item), které bylo vyvolané rozšířením kontejneru. Na obrázku jsou ukázány všechny vazby dědičnosti i delegace mezi těmito objekty.

Obousměrná delegace zajišťuje komunikaci od kontejneru k jeho prvkům (delegace typu obsah) a od prvku ke kontejneru (delegace typu kontejner).

Často bývá prvek sám kontejnerem. Takto lze tvořit stromovou hierarchickou strukturu. Při rozšíření se rozšíří celá tato struktura.

Pokud chceme, aby se objekt stal kontejnerem, stačí mu poslat zprávu `addContainerMixin`. Podobně uděláme z libovolného objektu prvek posláním zprávy `addItemMixin`.



Obrázek 25. Eggs: Rozšíření kontejneru

5.7. Realizace jazyka

Podkapitola předpokládá znalost jazyka Common Lisp.

Praktická část práce obsahuje kompilátor jazyka Eggs napsaný v jazyce Common Lisp. Ten byl pro tento účel vybrán především pro jeho velkou dynamičnost. Ta umožňuje za běhu programu kompilovat nově vytvořené lispové funkce. Toto je klíčová vlastnost, protože volání lispových funkcí realizuje kód metod jazyka Eggs.

Metody Ve skrytém slotu `function` metody se nachází zkompilovaný kód metody ve formě funkce jazyka Common Lisp.

Skrytý slot **lispSrc** obsahuje zdrojový kód této funkce.

Zdrojový kód funkce metody je zkompilován při prvním vyhodnocení metody.

Zdrojový kód funkce metody je sestaven při změně kódu metody.

Zdrojový kód funkce metody se převážně skládá z volání lispové funkce `send`. Ta zajišťuje posílání zpráv.

Například kód metody `a: 1 + 2` vede k sestavení zdrojového kódu lispové funkce:

```
(lambda (o) (send o '|a:| (send 1 + 2)))
```

Tato funkce je zkompilována při prvním vyhodnocení metody. Vyhodnocení kódu metody probíhá tak, že se zavolá lispová funkce kódu metody na aktivaci metody.

Optimalizace Reifier zprávy se vyhodnocují při sestavování zdrojového kódu funkce metody. Díky tomu je lze použít pro optimalizaci kódu.

Například výraz

```
0 setSlot: \$factorial Value: 1
```

vede bez optimalizace k sestavení lispového kódu:

```
(send 0 '|setSlot:Value:| '|factorial| 1)
```

Uvedením optimalizačního reifieru `setSlot:Value:` se sestaví rychlejší kód:

```
(set-slot-value 0 '|factorial| 1)
```

Perzistence Ukládání světa na disk využívá knihovnu cl-store. Ta neumožňuje uložit lispovou funkci. Z tohoto důvodu jsou všechny zkompilované kódy metod při uložení zahozeny. Uloženy jsou jen zdrojové kódy lispových funkcí realizující kódy metod.

6. Vývojové prostředí

Tato kapitola je průvodcem ve vývojovém prostředí jazyka Eggs. Po krátkém úvodu následuje popis jeho ovládání, které probíhá většinou myší. Další části průvodce dokumentují místní (kontextové) nabídky, hlavní nabídku a dialogy vývojového prostředí. Poslední část obsahuje ukázku jeho použití.

Instrukce pro spuštění vývojového prostředí se nachází v souboru `readme.txt`, který se nalézá v kořenovém adresáři přiloženého CD.

6.1. Úvod

Vývojové prostředí po spuštění zachycuje obrázek 26.

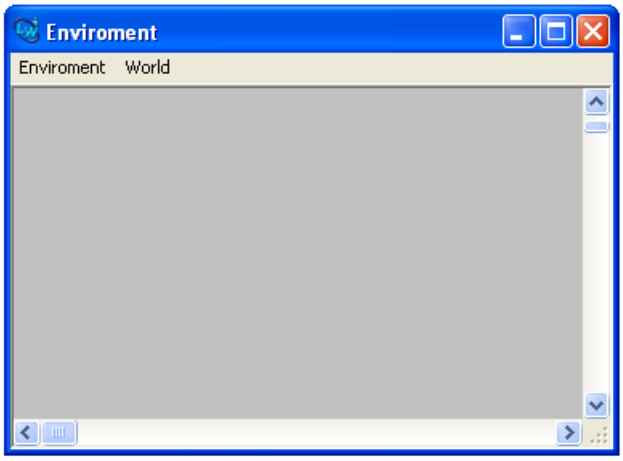
Velká šedivá plocha uvnitř vývojového prostředí slouží k zobrazování objektů. Ty se zobrazují stejně, jako popisuje kapitola 5. Uživatelské prostředí umožňuje skrývat nedůležité a pomocné sloty. Pokud jsou skryté sloty zobrazeny, jejich názvy jsou psány šedou barvou.

Na ploše může být vybrán jeden objekt. Vybraný objekt je zvýrazněn silnějším orámováním.

Obrázek 27. ukazuje vývojové prostředí při běžném sezení.

6.2. Ovládání

Vývojové prostředí je ovládáno hlavně myší. Nejprve představíme akce, které lze vyvolat použitím levého a pravého tlačítka myši. Akce, které lze vyvolat klávesnicí, se nachází na konci podkapitoly.



Obrázek 26. Vývojové prostředí po spuštění

Levé tlačítko myši Dvojí kliknutí na plochu otevře dialog vyhodnocení kódu pro objekt `lobby`.

Chceme-li například na ploše nechat zobrazit objekt `lobby`, provedeme tyto kroky:

1. Dojím kliknutím na plochu vývojového prostředí otevřeme dialog vyhodnocení kódu.

2. Do otevřeného dialogu zadáme `lobby`.

3. Dialog potvrdíme. Na ploše se v místě, kde bylo provedeno dvojí kliknutí, zobrazí objekt `lobby`.

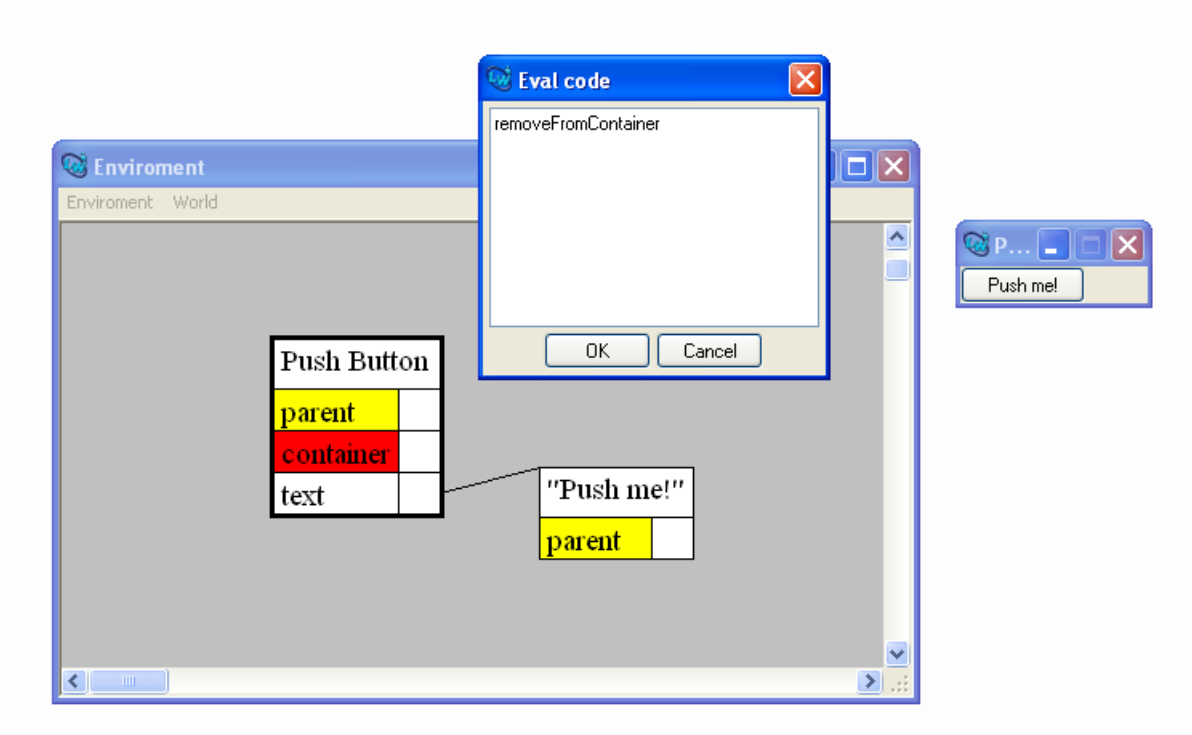
Objekt vybereme kliknutím na něj. Zobrazené objekty se po šedivé ploše posouvají uchopením a tažením levým tlačítkem myši.

Pro zobrazení hodnoty slotu stačí kliknout levým tlačítkem myši na čtvereček umístěný vpravo od názvu slotu. Opětovným kliknutím na tento čtvereček zmizí linka mezi ním a hodnotou slotu. Uchopením, přetažením a puštěním linky na objekt měníme obsah slotu, ze kterého linka vychází.

Dvojím kliknutím na objekt se otevře dialog vyhodnocení kódu pro tento objekt. Dialog si lze prohlédnout na obrázku 28. výjimkou je dvojí kliknutí na metodu. U ní se otevře dialog změna metody. Dialog je na obrázku 29.

Pravé tlačítko myši Kliknutím pravým tlačítkem na objekt, plochu či linku se otevře místní (kontextová) nabídka. Místní nabídky popisuje podkapitola 6.3.

Klávesnice Stisknutím tlačítka `Delete` dojde k odstranění vybraného objektu z plochy.



Obrázek 27. Běžné sezení ve vývojovém prostředí

6.3. Místní nabídky

Místní neboli kontextové nabídky se zobrazí po kliknutí pravého tlačítka na prvek vývojového prostředí. Následuje popis těchto nabídek pro různé prvky.

6.3.1. Objekt

Místní nabídka pro objekt je tvořena dvěma podnabídkami: Outliner a Object.
Outliner je obdélník zobrazující objekt.
Podnabídka Outliner obsahuje tyto položky:

Setting Otevře dialog nastavení zobrazení objektu.

Dismiss Odebere objekt z plochy.

Dismiss group Odebere skupinu objektů určenou tímto objektem z plochy.

Objekt patří do skupiny určené objektem, vede-li na něj linka z nějakého objektu patřícího do této skupiny. Určený objekt patří do této skupiny.
V podnabídce Object lze nalézt položky:

Eval code Otevře dialog vyhodnocení kódu.

Add slot Otevře dialog přidání slotu.

Add method Otevře dialog přidání metody.

Metody mají navíc položku **Edit method** pro otevření dialogu změna metody.

Pokud byla místní nabídka vyvolána nad slotem objektu, objeví se v ní navíc podnabídka Slot, ve které se mohou vyskytnout položky:

Delete Odebere slot z objektu.

Hide Schová slot.

Make visible Zviditelní schovaný slot.

Become delegate Učiní slot delegátovým a dialogem se zeptá na jeho typ.

Become parent Slot se stane rodičovským.

Become non parent Slot přestane být rodičovský.

Become non delegate Slot přestane být delegátový.

Make high priority parent Rodičovský slot se stane nejdůležitější.

6.3.2. Linka

Místní nabídka pro linku je tvořená jedinou položkou **Dismiss**. Ta odstraňuje linku z plochy.

6.3.3. Plocha

V místní nabídce pro plochu lze nalézt položku **Eval code**. Ta otevře dialog vyhodnocení kódu pro objekt `lobby`. Dále se zde nalézá podnabídka **Make new**. V této podnabídce jsou položky:

Object Vytvoří rozšíření obecného objektu a zobrazí jej na plochu.

Enviroment Nejprve se zeptá na počet proměnných. Vytvoří rozšíření objektu `lobby` s tolika sloty, kolik bylo zadáno proměnných. Sloty mají jména písmen z počátku abecedy(a, b, c . . .). Ve všech těchto slotech je uložen objekt `nil`.

6.4. Hlavní nabídka

Hlavní nabídka je tvořena dvěma podnabídkami Enviroment a World. Podnabídka Enviroment obsahuje tyto položky:

Update Aktualizuje zobrazené objekty a linky na ploše.

Clean Odebere z plochy všechny objekty.

Set font Nastaví font textu v objektech.

Exit Ukončí vývojové prostředí bez uložení změn.

V podnabídce World se nacházejí položky:

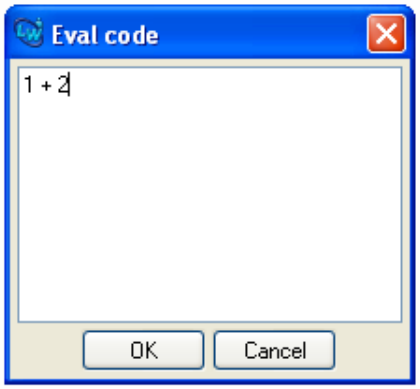
Open... Otevře svět uložený ve vybraném souboru. Vyčistí plochu. Starý svět bude ztracen.

Save As... Uloží svět včetně rozložení zobrazených objektů na ploše do vybraného souboru.

6.5. Dialogy

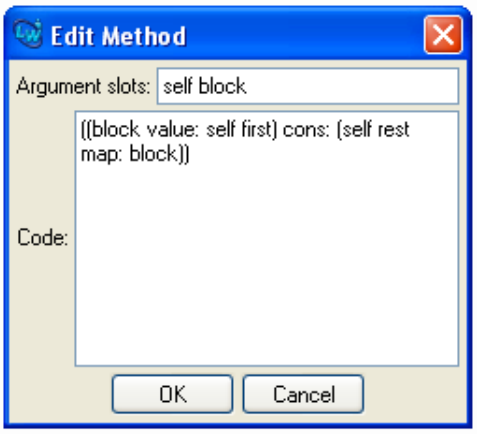
Ve vývojovém prostředí se lze setkat s dialogy, které jsou vysvětleny v této podkapitole. Dialogy jsou většinou vyvolané skrz místní nabídku prvku vývojového prostředí.

Dialog vyhodnocení kódu Tento dialog je zachycen na obrázku 28. Dialog je vždy otevřen pro určitý objekt. Do textového pole se vkládá kód jazyka. Po potvrzení dialogu dojde k vyhodnocení zadaného kódu v určeném objektu. Objekt vrácená hodnota se zobrazí a stane se vybranou.



Obrázek 28. Vývojové prostředí: Dialog vyhodnocení kódu

Dialog změna metody Tento dialog lze spatřit na obrázku 29. Lze v něm měnit argumentové sloty a kód metody.



Obrázek 29. Vývojové prostředí: Dialog měnící metodu

První argumentový slot (většinou `self`) slouží pro uložení příjemce zprávy.

Dialog přidání slotu Tento dialog je zachycen na obrázku 30.

Obrázek 30. Vývojové prostředí: Dialog přidání slotu

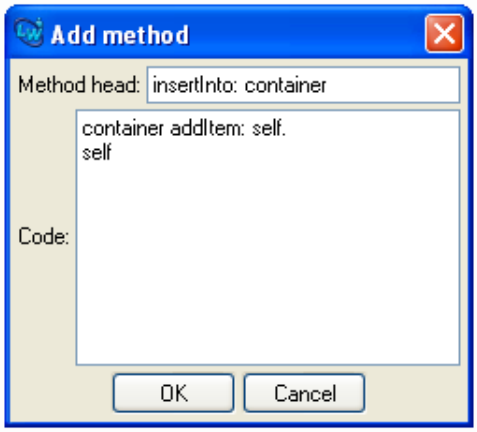
Kromě jména slotu (slot name) lze v dialogu zadat i obsah slotu (slot value). Obsah slotu je kód, který se po potvrzení dialogu vyhodnotí v objektu `lobby`. Vrácená hodnota se vloží do přidaného slotu. Dialog obsahuje zaškrtávací políčko určující, zda má být slot pouze pro čtení (read only slot).

Pokud není přidán slot pouze pro čtení, pak je navíc do objektu přidána metoda sloužící k nastavení slotu. Její jméno vznikne spojením jména slotu a dvojtečky.

Tedy například pro slot `a` bude přidána nastavovací metoda uložená ve slotu `a:`.

Tento dialog lze také použít pro změnu obsahu slotu.

Dialog přidání metody Obrázek 29. zachycuje tento dialog.



Obrázek 31. Vývojové prostředí: Dialog přidání metody

Dialog se skládá ze dvou textových polí. První slouží k zadání hlavičky metody (method head) a druhé ke kódu metody (code).

Hlavička metody určuje jak jméno slotu, kde má být metoda uložena, tak i argumentové sloty metody.

Následují příklady hlaviček metod. Hlavička

```
inc
```

značí, že metoda o jednom argumentovém slotu `self` bude uložena do slotu `inc`.

Hlavička

```
+ arg
```

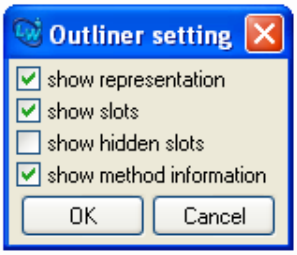
určuje dva argumentové sloty `self` a `arg` a slot `+` pro uložení metody.

Konečně hlavička

```
to: limit By: step
```

určuje tři argumentové sloty (`self`, `limit` a `setp`) a slot `to:By:`, kde bude metoda uložena.

Dialog přepisuje původní obsah slotu.



Obrázek 32. Vývojové prostředí: Dialog nastavení zobrazení objektu

Dialog nastavení zobrazení objektu Tento dialog si můžete prohlédnout na obrázku 32.

Dialog obsahuje následující zaškrtávací tlačítka měnící zobrazení objektu:

- show representation (zobrazení textové reprezentace objektu)
- show slots (zobrazení slotů objektu)
- show hidden slots (zobrazení skrytých slotů objektu)
- show method information (zobrazení argumentových slotů a kódu metody)

6.6. Ukázky použití vývojového prostředí

6.6.1. Práce s anonymními objekty

Na ploše jsou zobrazené tlačítko (Push Button) a řádkové uspořádání (Row Layout).

Můžeme je například vytvořit a nechat zobrazit následovně:

1. Dvojím klikem na plochu vývojového prostředí otevřeme dialog vyhodnocení kódu.
2. Zadáme kód `pushButton extend` a dialog potvrdíme. Na ploše se zobrazí nové tlačítko (Push Button).
3. Podobně vyhodnotíme kód `rowLayout extend`. Dojde k zobrazení nově vytvořeného řádkového uspořádání (Row Layout).

Tyto objekty nejsou nikde uložené. Chceme vložit tlačítko do řádkového uspořádání.

1. Klikem pravého tlačítka na plochu vyvoláme místní nabídku. Vybereme položku **Make new -> Enviroment**.

2. V zobrazeném dialogu zadáme číslo dva. Vytvoří se nový objekt se sloty `a`, `b`.

3. Kliknutím levého tlačítka na čtverečky k těmto slotům zobrazíme jejich hodnoty. Zobrazí se objekt `nil` a ze slotů k němu vedoucí linky.

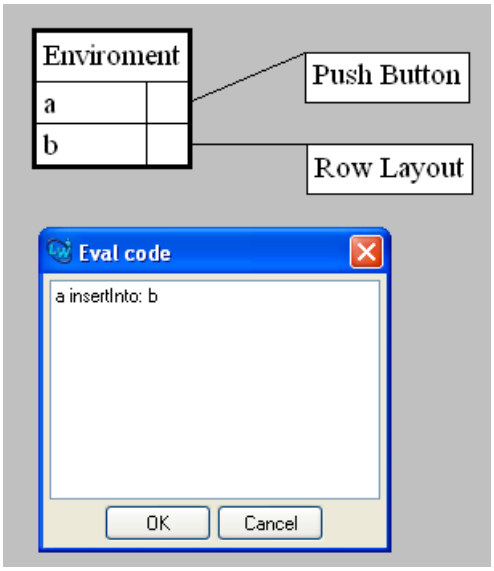
4. Přetáhneme linku vedoucí ze slotu `a` na tlačítko. Linku ze slotu `b` přetáhneme na řádkové uspořádání.

5. Dvojí kliknutí na prostředí (Enviroment) otevřeme dialog vyhodnocení kódu. Zadáme a potvrdíme kód

   ```
   a insertInto: b
   ```

   Dojde k přidání tlačítka do řádkového uspořádání.

Situace je zachycená na obrázku 33..



Obrázek 33. Použití vývojového prostředí k manipulaci anonymních objektů

6.7. Realizace vývojového prostředí

Vývojové prostředí je vytvořené v jazyce Common Lisp za použití knihoven CAPI, CLMG3. Autorem knihovny CLMG3 je Michal Krupka.

7. Prakticky použitelná aplikace

V jazyce Eggs byla za použití jeho vývojového prostředí napsána jednoduchá textová hra.

Hra využívá grafické rozhraní napsané v jazyce Eggs. Tomuto rozhraní je věnována první podkapitola. Druhá podkapitola se zabývá hrou. Její realizace v jazyce Eggs je objasněna v poslední podkapitole.

7.1. Grafické uživatelské rozhraní

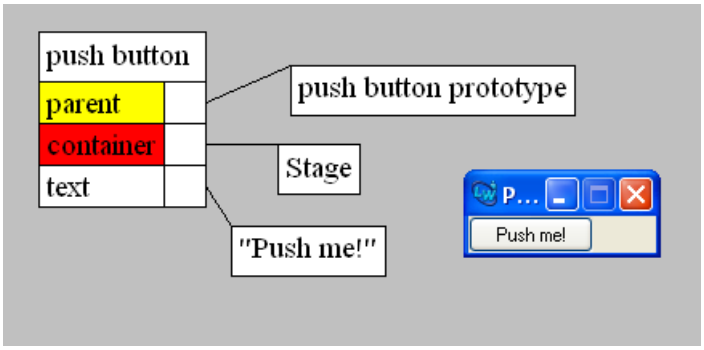
Grafické uživatelské rozhraní Eggs je inspirované multimediálním prostředím SK8 ([20]) vytvořené firmou Apple.

Grafické prvky se nazývají *herci* (actors). Monitor je reprezentován *jevištěm* (stage). Je-li herec na jevišti, znamená to, že je zobrazen.

Například tlačítko je hercem. Následující kód vytvoří rozšíření prototypu tlačítka, nastaví mu text a vloží jej na jeviště.

```
(pushButton extend text: "Push me!") insertInto: stage
```

V důsledku toho se na obrazovce objeví tlačítko s nápisem „Push me!“. Situaci zachycuje obrázek 34.



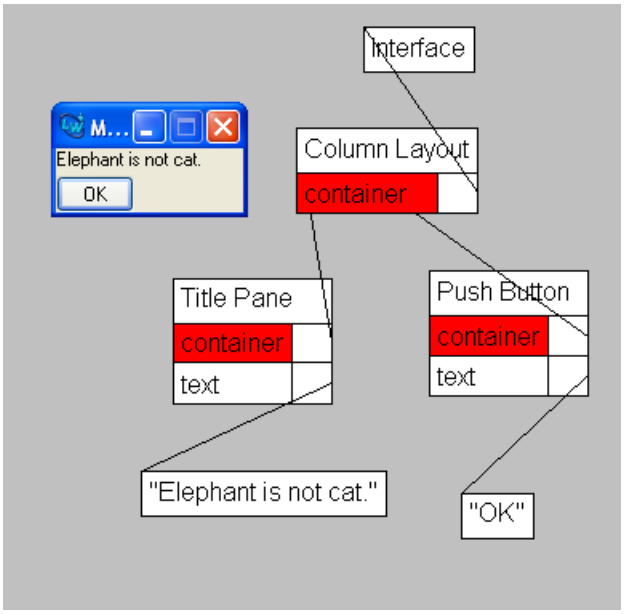
Obrázek 34. Herec tlačítko na jevišti

V grafickém uživatelském rozhraní se používá hierarchie prvek-kontejner popsaná v podkapitole 5.6.5.

Jeviště je kontejner. Herci jsou prvky. Někteří herci jsou navíc také kontejnery. Například herec grafické okno. Ten je prvkem, proto může být vložen do kontejneru jeviště. Současně je také kontejnerem a může do něj být vložen prvek. Jako například herec řádkové uspořádání (`rowLayout`).

Složitější hierarchie grafických prvků se nalézá na obrázku 35.

U některých grafických prvků (herců) lze zobrazit místní nabídku kliknutím pravého tlačítka myši. Ta obsahuje položku `Display outliner`. Po jejím vybrání



Obrázek 35. Ukázka hierarchie grafických prvků

se tento grafický prvek objeví na ploše vývojového prostředí. Objekt na ploše vývojového prostředí lze jednoduše prohlížet a upravovat.

Například pro úpravu textu tlačítka na něj klikneme pravým tlačítkem. V zobrazené místní nabídce vybereme položku `Display outliner`. Tlačítko se objeví na ploše vývojového prostředí. Dvojím klikem na něj otevřeme dialog Vyhodnocení kódu. Zde zadáme `text: "New name"` a potvrdíme. Text tlačítka se změní.

Více v kapitole 6. o vývojovém prostředí.

Aktualizace Každého herce lze rozšířit. Z tohoto důvodu může změna herce ovlivnit jeho potomky. Ti mohou být také na jevišti. Proto je potřeba po každé změně herce, který má potomky, jeviště aktualizovat.

Po změně herce se nejprve zkontroluje, má-li potomky. Herec s potomky na zprávu `hasChildren` odpoví (vrátí) `true`. Nemá-li změněný herec potomky, aktualizuje se jenom on. Aktualizace se provede posláním zprávy `doUpdate`.

Aktualizace herce s potomky je náročnější a probíhá následovně.

Nejprve se u změněného herce nastaví slot `isValid` na hodnotu `nil`. Obecný herec má ve slotu `isValid` uložen objekt `true`. Tím se docílí toho, že všichni potomci změněného objektu odpovídají na zprávu `isValid` nepravdou, zatímco nezměněné objekty pravdou. Poté se provede aktualizace jeviště. Nakonec se odebere slot `isValid` ze změněného objektu. Tím se změněný objekt i jeho potomci stanou platní. Budou tedy na zprávu `isValid` odpovídat pravdou.

Aktualizace jeviště se spustí posláním zprávy `update` jevišti. Tato zpráva je typu obsah (content message). Proto je automaticky delegována prvkům jeviště.

Nebo-li hercům na jevišti.

Metoda `update` herce má kód:

```
evalNext . self isValid ifFalse: (# (self doUpdate))
```

Nejprve je zpráva delegována na případné prvky herce. O to se postará zpráva `evalNext`. Tím se nejdřív aktualizují prvky herce (podherci). Poté se zkontroluje, je-li herec aktuální. To se provede posláním zprávy `isValid` sobě samému.

Pokud herec není aktuální, provede se jeho aktualizace. Ta se spustí zprávou `doUpdate`.

Pokud je na jeviště přidán nový herec, nemůžeme vědět, je-li aktuální. Proto je aktualizován on i jeho podherci. To se provede posláním zprávy `forceUpdate`.

Metoda `forceUpdate` herce má následující kód:

```
evalNext . self doUpdate
```

Stejná situace nastane, když je herci přidán nový podherec. Proto i nový podherec se aktualizuje zprávou `forceUpdate`.

Knihovna herců K stavbě uživatelského rozhraní lze použít tyto herce: okno (`interface`), tlačítko (`pushButton`), popisek (`titlePane`), řádkové uspořádání (`rowLayout`), sloupcové uspořádání (`columnLayout`) a textové pole (`displayPane`).

Herec okno musí mít jednoho podherce, který je uspořádáním.

Podherci řádkového a sloupcového uspořádání se zobrazí v řádku, respektive v sloupci.

Tlačítko má atribut `text` obsahující popisek tlačítka. Atribut `enabled` určuje, zda je tlačítko aktivní. Tlačítko poté, co je stisknuto, pošle sobě zprávu `buttonPress`. Tato zpráva je typu událost (event message).

Atribut `text` popisku a textového pole určuje zobrazený text.

Známé problémy Pod operačním systémem Windows nemusí vždy dojít k požadované změně. Například při změně popisku tlačítka.

Řešením je vynutit aktualizaci celého jeviště výrazem `stage forceUpdate`. Pokud to nepomůže, lze problémového herce z jeviště vyjmout a opět jej na něj vrátit.

7.2. Jednoduchá textová hra

Podkapitola obsahuje instrukce pro spuštění hry a její pravidla.

Spuštění Pro spuštění hry stačí ve vývojovém prostředí jazyka Eggs načíst svět uložený v souboru `/src/worlds/game.wld`

To se provede následovně:

- V hlavním nabídce vývojového prostředí (Enviroment) vyberte položku `World -> Open...`
- V otevřeném dialogu vyberte soubor `/src/worlds/game.wld` Po chvíli se objeví okno hry (Game).

Pravidla V této hře se hráč stává postavou v herním světě. Herní svět je tvořen místy. Postava se vždy nachází na jednom místě. Místo umožňuje provést určité akce, jako je například přesun postavy na jiné místo, změna místa nebo změna postavy.

Hra probíhá v následující smyčce:

Z možných akcí pro místo, na kterém se nachází postava hráče, se třikrát provede náhodný výběr. Akce může být vybrána vícekrát. Hráč si musí jednu akci zvolit. Zvolená akce se provede. Hráči se zobrazí textový popis průběhu akce a popis místa, kde se postava nachází.

7.3. Uskutečnění hry

Podkapitola popisuje objekty, které byly použity k tvorbě hry v jazyce Eggs.

Hra se nachází v objektu `game`. K prohlížení objektů, z kterých se hra skládá, můžete použít vývojové prostředí jazyka Eggs. Pro zobrazení objektu `game` stačí ve vývojovém prostředí vyhodnotit kód `game` v objektu `lobby`.

7.3.1. Herní objekty

Základní objekty potřebné k uskutečnění hry jsou: herní svět (`world`), místo (`location`), postava (`character`) a akce (`action`).

Herní svět Herní svět tvoří místa, kde se postava může nacházet. Svět je objektem. Místa ve světě jsou uložená v jeho slotech. Ve slotu `locationSlotList` si svět uchovává seznam jmen slotů, ve kterých jsou místa uložená.

Pro novou hru je nutné výchozí herní svět rozšířit. To proto, aby se změny, které hráč ve hře provede, nepromítly do výchozího světa. Protože hráč může změnit jednotlivá místa světa, je potřeba rozšířit i tato místa. Navíc je nutné, aby místa odkazovala zpátky na rozšířený svět namísto výchozího.

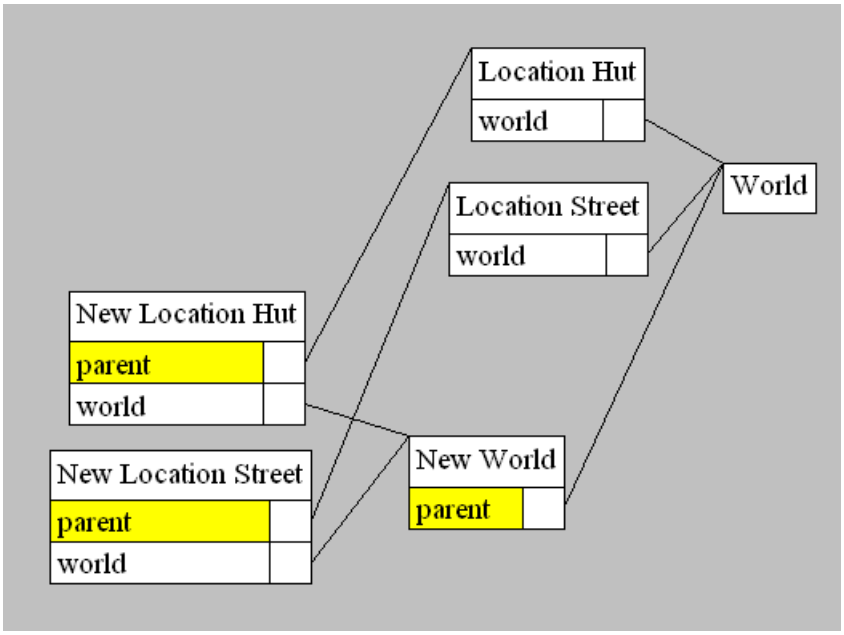
Metoda `extend` pro svět má kód:

```
newWorld = evalRewritten.
self locationSlotList map:
       (locationSlot #
```

```
        (newWorld setSlot: locationSlot Value:
              ((self send: locationSlot)
                                  extend
                                    world: newWorld))).
newWorld
```

V druhém výrazu kódu se nejprve zjistí seznam slotů s místy výchozího světa (`self locationSlotList`). Pro každý tento slot (`map:`) se provedou následující operace. Vezme se místo výchozího světa (`self send: locationSlot`). To se rozšíří (`extend`). Nastaví se svět nového místa (`world: newWorld`). Nové místo se uloží do slotu nového světa (`newWorld setSlot: locationSlot Value: (` ... `)`).

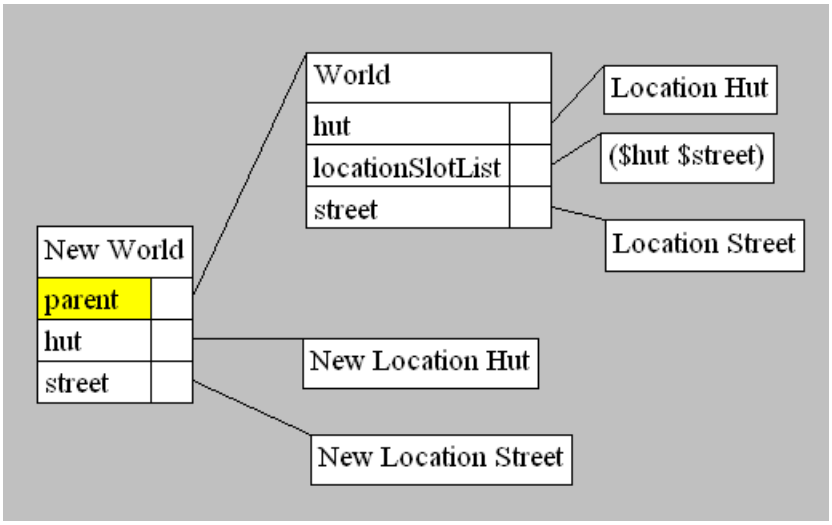
Uvažujme například svět o dvou místech: chatrč (Hut Location) a ulice (Street Location). Chatrč je uložená ve slotu `hut` a ulice ve slotu `street`. Rozšíření tohoto světa si můžete prohlédnout na obrázcích 36. a 37. První obrázek zachycuje rodičovské vztahy a příslušnost místa ke světu. V druhém obrázku je patrné, z jakých míst se světy skládají.



Obrázek 36. Hra: Rozšíření herního světa (první část)

Akce Akce se spustí posláním zprávy „`do`“. Pro zjištění, zda je akce platná, se používá zpráva `isValid`. Tato zpráva může být poslána pouze akci, která je přizpůsobená pro postavu a místo.

Textový popis průběhu akce obdržíme posláním zprávy `description`. Popis akce, který se zobrazí hráči při výběru akce, získáme posláním zprávy `choiceText`.



Obrázek 37. Hra: Rozšíření herního světa (druhá část)

Například na obrázku 38. si můžete prohlédnout akci pro rozsvícení světla (Action Switch on the Light). Vidíme, že akce je platná pouze, když je světlo zhasnuté. O to se stará metoda `isValid` s kódem:

```
self location isLightOn not
```

Dále si povšimněte, že spuštění akce zprávou `do` způsobí zapnutí světla. To se provede vyhodnocením kódu:

```
self location isLightOn: true
```

Pro vykonání akce musí být akce přizpůsobená postavě a místu. To se provede následovně.

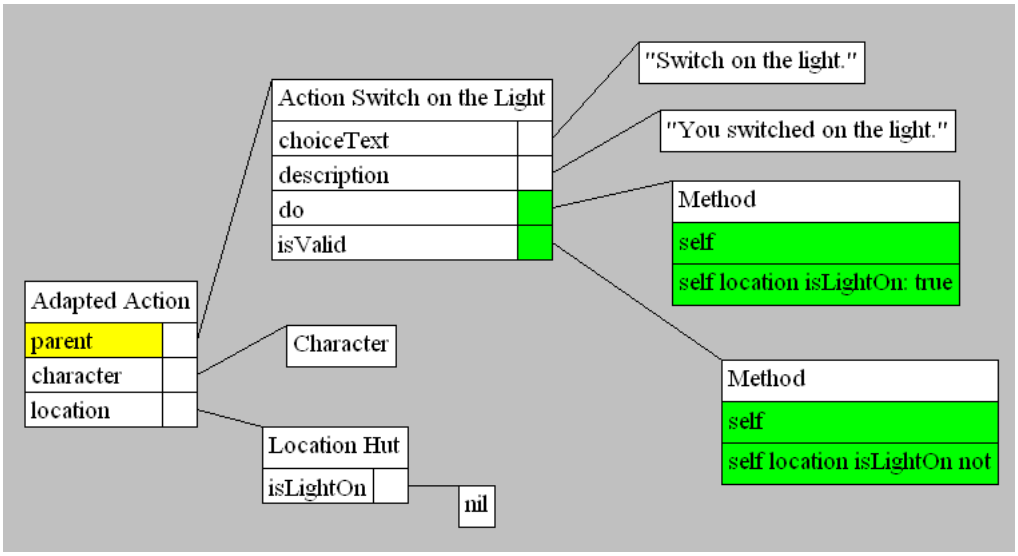
Pro přizpůsobení akce se nejprve akce rozšíří. Poté je do slotu `character` rozšíření vložena postava a do slotu `location` vloženo místo.

Tím může vykonání akce změnit postavu a místo, pro které je akce určena. Obrázek 38. také zobrazuje akci (Adapted Action) přizpůsobenou postavě (Character) a chatrči (Location Hut).

Kód metody `do` akce, která přemístí postavu na ulici, vypadá takto:

```
self character location: self location world street
```

Podvýraz `self location world street` nejprve odkazuje na aktuální akci (`self`). Poté se zjistí místo, pro kterou je akce určena (`location`). Následuje přesun do světa, ve kterém se místo nachází (`world`). Nakonec se v tomto světě najde místo ulice (`street`).



Obrázek 38. Hra: Akce rozsvit světlo

Místo Místo ve slotu `actions` uchovává seznam všech možných akcí.

Místo na zprávu `generateActions: character` vrací seznam platných akcí přizpůsobených pro danou postavu (character). Kód obsluhy zprávy:

```
(self actions map:
     (action #
        ((action extend character: character) location: self)))
     filter: (action # (action isValid))
```

V prvním kroku se získají všechny akce (`self actions`). Druhý krok přizpůsobí akce postavě a místu (`(action extend character: character) location: self`). Ve třetím a posledním kroku se z akcí vyberou pouze ty platné (`filter: (action # (action isValid))`).

Místo na zprávu `description` vrací popis místa. Tento popis se zobrazí hráči. Popis místa může být statický nebo sestavený podle stavu místa. Zde je s výhodou použita možnost jazyka zaměňovat atributy a metody objektu. Proto může být ve slotu `description` řetězec nebo metoda vracející řetězec.

Postava Postava má ve slotu `location` uložené místo, kde se nachází.

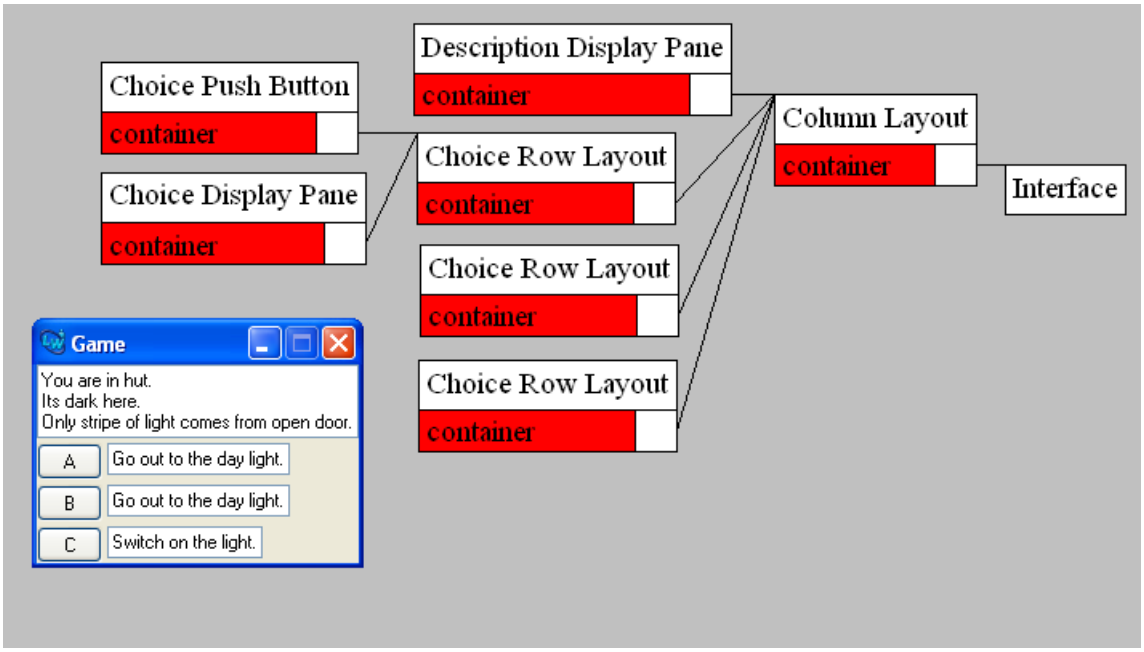
Charakteristika postavy může být uložena ve slotech postavy. Například nálada postavy může být uložena ve slotu `mood`.

Posláním zprávy `generateAction` postavě obdržíme seznam všech akcí, které postava může na místě, kde se nachází, dělat. Metoda `generateAction` postavy má kód:

```
self location generateActions: self
```

7.3.2. Uživatelské rozhraní

Grafické okno hry (Interface) je rozděleno na čtyři vodorovné částí. V horní části se nalézá textové pole (Description Display Pane) pro popis hry. Zbylé tři části jsou tvořeny volbami (Choice Row Layout). Každá volba je řádkou obsahující tlačítko (Choice Push Button) a textové pole (Choice Display Pane). Grafické okno hry i strukturu herců, kteří jej tvoří, si lze prohlédnout na obrázku 39.



Obrázek 39. Grafické okno hry a jeho struktura

Volba uchovává akci, kterou zastupuje, ve slotu `action`.

Výběr volby Hráč si vybere volbu stiskem jejího tlačítka. Stisk tlačítka vyvolá poslání zprávy `buttonPress` sobě samému. Tato zpráva je typu událost (`eventMessage`). Proto je delegována kontejneru tlačítka. To je volba (Choice Row Layout). Volba má definovanou obsluhu této zprávy následovně:

```
self doAction: self action
```

Tento kód pošle vybrané volbě kontejnerovou zprávu (`containerMessage`) `doAction:` s argumentem vybrané akce. Tato zpráva bude postupně delegována až na grafické okno hry (Interface). Okno tuto zprávu obslouží vykonáním kódu:

```
action do .
self updateGame: action description
```

První výraz způsobí vykonání akce. Druhý výraz aktualizuje hru. Výraz `action description` vrátí jako hodnotu popis akce.

Poslání zprávy `updateGame:` vede k vyhodnocení kódu:

```
(self updateDescription: actionDescription) updateChoices
```

Ten nejprve aktualizuje popis herního stavu zprávou `updateDescription:`. Poté se aktualizují herní volby zprávou `updateChoices`

Herní volby se aktualizují následujícím kódem:

```
actions = self character generateActions.
self choices map: (choice #( choice action: actions randomChoice)).
self
```

První výraz kódu vytvoří seznam všech možných akcí pro postavu ve hře a uloží jej do slotu `actions` prostředí. Tato technika zastupuje roli proměnných známých z jiných programovacích jazyků. V druhém výrazu se nastaví akce všem třem volbám díky zprávě `map:` poslané volbám `self choices`. Nové akce budou vybrány náhodně ze seznamu všech možných akcí výrazem `actions randomChoice`.

Konečně nastavení akce volby se provede kódem:

```
self setSlot: $action Value: action .
self text: action choiceText
```

První výraz kódu nastaví slot `action`. Druhý výraz se postará o zobrazení popisu akce.

8. Možnosti prototypových jazyků

První podkapitola vytyčuje hranici použitelnosti prototypového přístupu. Vychází z 3. kapitoly o výhodách a nevýhodách prototypových jazyků. Další podkapitoly se zabývají způsoby, jak v prototypových jazycích modelovat třídy pouze za použití jejich prostředků. Závěrečná podkapitola nabízí řešení některých problémů prototypových jazyků.

8.1. Hranice prototypového přístupu

V počátcích vývoje aplikace nezná programátor abstrakce, které bude potřebovat. Pracuje s konkrétními objekty. V této etapě vývoje je lepší použít přístup prototypový než třídový. Ten umožňuje přímo tvořit konkrétní objekty a nabízí lepší modelovací schopnosti.

Jak vývoj aplikace postupuje, objevují se potřebné abstrakce. Zde se prototypový přístup stává slabším. Jeho prostředky na zachycení abstrakcí (konceptů) jsou horší.

Hranice prototypového přístupu je neostrá a nachází se tam, kde programátor požaduje dobře uspořádaný systém abstrakcí.

8.2. Napodobování tříd

Tato podkapitola čerpá z článku [6]. V tomto článku je ukázáno, jak v jazyku Self simulovat třídy Smalltalku. Simulace je tak věrná, že lze vykonávat kód Smalltalku. Efektivnost vykonávání kódu je srovnatelná s nativní implementací Smalltalku. Realizaci Smalltalku v Selfu si lze stáhnout z domovské stránky jazyka Self ([2]).

Následuje ukázka modelování třídy `Date` jazyka Smalltalk.

Třída `Date` dědí z třídy `Object`. Instance třídy `Date` mají atribut`days` a metodu `addDays:`. Třída `Date` má atribut `mounthNames` a metodu `initialize`.

Obrázek 40. zachycuje schéma napodobení této třídy v jazyce Self.

V rodičovském slotu `my_behaviour` uchovávají objekty zastupující instance objekt určující jejich chování. Například instance třídy `Date` má v tomto slotu objekt uchovávající metody instance. Díky tomu tato instance rozumí zprávě `addDays:` Instance může zdědit chování z nadtřídy své třídy. To je zajištěno rodičovským slotem `inherited_meths` v objektu Metody instance a Metody třídy `Date`.

Šipka (`<-`) ve slotu označuje nastavující primitivum pro příslušný slot. Slot `days` instance lze nastavit na 30 posláním zprávy `days: 30`.

Protože každý objekt musí být instancí třídy a třídy jsou objektem, musí mít třída `Date` svoji třídu (metatřídu). Třídy ve slotu `inst_meths` uchovávají objekt obsahující metody pro jejich instance.

Metatřída pro třídu `Date` definuje metodu `initialize` pro třídu `Date`. Tato metoda tvoří nové instance třídy `Date`. To probíhá tak, že se nejprve naklonuje prototypická instance uložená ve slotu `proto_inst` objektu Metody třídy `Date`. Poté se nastaví slot `days` nově vzniklé instanci.

Z nepodstatných důvodů musí být za jména proměnných tříd přidána koncovka `_g`.

8.3. Prototypový jazyk se třídami

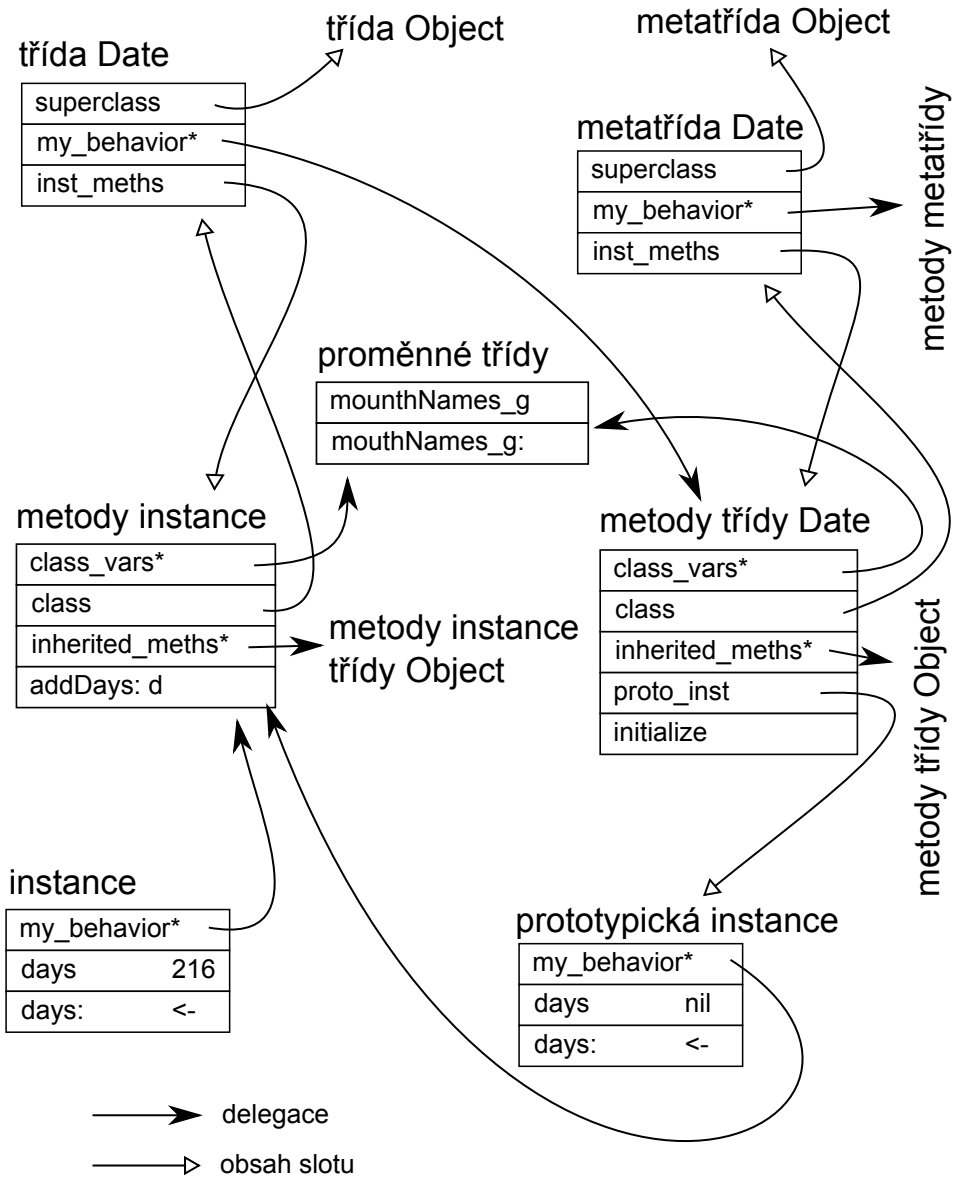
Odeberme tyto požadavky z pojmu třída:

- Každý objekt musí být instancí nějaké třídy.
- Všechny instance třídy musí mít stejnou strukturu i chování.

Takto upravené třídy lze jednoduše přidat do prototypových jazyků.

Díky tomu, že objekty nemusí být instancemi tříd, odpadá i potřeba metatříd.

Vzniká tak objektový model, který kombinuje jek prototypový, tak i třídový přístup. Tento model spojuje výhody obou modelů. V tomto modelu lze z počátku tvorby používat prototypový přístup a později zavádět třídy. Tento model nebyl v této práci prozkoumán. Ukazuje směr, kterým by se mohl vývoj prototypových jazyků ubírat.



Obrázek 40. Napodobení třídy `Date` v jazyce Self

8.4. Abstraktní objekty

Několik nevýhod prototypových jazyků spočívá v horší podpoře abstraktních objektů (prototypů a sdílených úložišť). O nich mluví kapitola 3.

Jako řešení se nabízí udělit abstraktním objektům speciální statut. Abstraktní objekty by zaváděly abstraktní obsluhy zpráv. Tyto obsluhy by se daly použít pouze k obsloužení zpráv poslaných jejich neabstraktním potomkům. Pokus o vyhodnocení abstraktní obsluhy, kde příjemce je abstraktní objekt, by skončil chybou.

Například sdílené úložiště pro čísla by bylo abstraktním objektem. Zavádělo by abstraktní obsluhu zprávy +. Čísla jsou neabstraktní potomky úložiště, proto by rozuměly zprávě +. Poslání zprávy + sdílenému úložišti by skončilo chybou, protože sdílené úložiště je abstraktní objekt s abstraktní obsluhou zprávy +.

Prototyp, poté co by se odladil, by se stal abstraktním objektem. To by znemožnilo jej použít v aplikaci, protože poslání zprávy prototypu by končilo chybou.

8.5. Dědičnost dosažená použitím omezení

Další možností, jak se zbavit nevýhod souvisejících s delegací, je použít místo delegace omezení (constraint).

Mezi objektem a jeho rodičem by nebyl vztah delegace. Místo toho by se uvedlo omezení, které by dědičnost zajišťovalo. Omezení by bylo udržováno systémem. Při změně rodiče by se omezení aktivovalo a změnilo by potomky.

Závěr

Prototypové jazyky odebírají z objektového programování třídy a nahrazují dědičnost delegací. Tím dosáhnou zvýšení modelovacích schopností jazyka.

Navíc jak ukazuje kapitola 8.2., lze v prototypovém jazyce modelovat třídy pouze za použití objektů a delegace.

Prototypové jazyky mají problémy, o kterých mluví kapitola 3. Jejich vyřešení by vedlo k vytvoření silného objektového přístupu s výhodami jak prototypových tak třídových jazyků.

Náčrt možného řešení některých problémů prototypových jazyků nabízí podkapitola 8.4.

Conclusions

Object model without classes and with delegation has better modeling ability than class-based object model.

It is possible to model classes using only objects and delegation. Subsection 8.2. shows this.

Prototype-based languages have problems. They are discussed in section 3. If we are able to solve their problems, then we receive object system with advantages of class-based and prototype-based object systems.

Subsection 8.4. try to solve some problems of prototype languages.

Reference

[1] Randall B. Smith, David Ungar. *Programming as an Experience: The Inspiration for Self.* Původně publikováno v EXOOP95, W. Olthoff (ed.), Lecture Notes in Computer Science, Vol. 952, Springer-Verlag pages 303-330. 1995

[2] *Domovská stránka jazyku Self.* Elektronická publikace, 2010.

[3] *The Self Handbook.*Elektronická publikace, 2010.

[4] David Unger. *Self and Self: Whys and Wherefores.* Elektronická publikace, IBM Research 2009.

[5] Christphe Dony , Jacques Malenfant, Daniel Bardou. *Classifying Prototype-based Programming Languages.* Prototype-based Programming: Concepts, Languages and Applications. 1998

[6] Mario Wolczko. *Self:* `includes:` *Smalltalk.*

[7] Randall B. Smith. *Prototype-based languages (panel): object lessons from class-free programming.* In Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications (OOPSLA '94). 1994

[8] Antero Taivalsaari. *Classes vs. Prototypes. Some Philosophical and Historical Observations.* Journal of Object-Oriented Programming 10(7), Nov/Dec 1997, pages 44-50. SIGS Publications. 1997

[9] Wire Codenie, Koen De Hondt, Thou D'Hondt, Patrick Steyaert. *Agora: Message Passing as a Foundation for Exploring OO Language Concepts.* ACM SIGPLAN Notices, Volume 29, No. 12, December 1994

[10] Wolfgang De Meuter. *Agora. The Scheme of Object-Orientation, or, the Simplest MOP in the World*

[11] Wolfgang De Meuter, Tom Mens, Patrick Steyaert. *Agora: Reintroducing Safety in Prototype-based Languages*

[12] Dave Thomas. *Doslov.* Prototype-Based Programming: Concepts, Languages and Applications. 1998

[13] Walter R. Smith. *NewtonScript: Prototypes on Palm.* Prototype-based Programming: Concepts, Languages and Applications.1998

[14] Günther Blaschek. *Omega: Statically Typed Prototypes.* Prototype-based Programming: Concepts, Languages and Applications.1998

[15] Brad A. Myers, Rich McDaniel, Rob Miller, Brad Vander Zanden, Dario Giuse, David Kosbie, Andrew Mickish. *The Prototype-Instance Object Systems in Amulet and Garnet*. Prototype-based Programming: Concepts, Languages and Applications. 1998

[16] Sebastián González, Kim Mens, Alfredo Cádiz. *Context-Oriented Programming with the Ambient Object System*. Journal of Universal Computer Science, vol. 14, no. 20. 2008

[17] Sebastián González Montesinos. *Programming in Ambience: Gearing up for dynamic adaptation to context*. Dizertační práce. 2008

[18] Dmitry Soshnikov. *JavaScript. The core.*. Elektronická publikace .2010

[19] *ECMA-262 ECMAScript Language Specification 5.1 edition.* June 2011

[20] Apple Computer. *SK8 User Guide Version 0.9* 1995

A. Obsah přiloženého CD

`bin/`
: Tato složka je prázdná. Instrukce, jak spustit vývojové prostředí jazyka Eggs a hru v něm napsanou, naleznete v souboru `readme.txt`, který se nalézá v kořenovém adresáři CD.

`doc/`
: Teoretická část diplomové práce se nalézá v souboru `diplomka.pdf` Soubor `diplomka.zip` obsahuje zdrojový text teoretické části a vložené obrázky.

`src/`
: Složka obsahuje zdrojové texty kompilátoru jazyka Eggs, vývojového prostředí a malé hry.

`readme.txt`
: Instrukce pro instalaci a spuštění vývojového prostředí jazyka Eggs a malé v něm napsané textové hry.

`install/`
: Instalátory LispWorks Personal Edition.