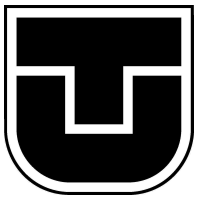


TECHNICKÁ UNIVERZITA V KOŠICIACH

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

emuStudio

Peter Jakubčo

Príloha C
Diplomová práca
Košice, 2009

Vade-mecum zásuvných modulov

Vedúci práce a konzultant: *ing. Slavomír Šimoňák, PhD.*
Pre viac informácií si prečítajte licenčné podmienky.

Obsah

Úvod

Dokument popisuje spôsob, ako vyvíjať zásuvné moduly pre emulačnú platformu ***emuStudio***.

Predpokladá sa, že čitateľ vie programovať v jazyku Java SE, vie čo je to dokumentácia typu *javadoc*, pozná pojmy ako sú napr. *rozhranie*, *balík*, *trieda*, *objekt*, *GUI*, atď. Tento dokument nie je programátorskou príručkou a preto v prípade potreby by sa čitateľ mal obrátiť na príslušné zdroje.

Predpokladá sa tiež, že čitateľ má k dispozícii dokumentáciu knižnice `emu_ifaces.jar` vo formáte *javadoc* (resp. Systémovú príručku), pretože tá podrobne popisuje rozhrania a ich metódy, ktoré zásuvné moduly implementujú a tento dokument sa bude na ňu často odvolávať.

Ďalším predpokladom je, že čitateľ má akú-takú predstavu o tom, čo je to emulácia, a ako funguje. Dokument popíše princíp práce emulačnej platformy ***emuStudio***, no nebude popisovať existujúce algoritmy emulácie. Algoritmy sú v skratke popísané v [5], podrobnejšie ich možno nájsť v [6].

Celá platforma (hlavný modul, aj zásuvné moduly) je napísaná v jazyku Java. Pre vývoj knižnice `emu_ifaces.jar` bolo použité prostredie JDK verzie `1.4`; no hlavný modul, ako aj zásuvné moduly už využívajú JDK verzie `1.6`. Tieto prostredia je možné stiahnuť zo stránky [1].

Kapitola 1

Komunikačný model

1.1 História modelu

Komunikačný model môžme chápať ako súhrn princípov, pravidiel a metód, ktoré sa aplikujú pri realizácii komunikácie. V tomto duchu je chápaný aj komunikačný model platformy ***emuStudio***, ktorého vývoj začína zvláštne od verzie 2. Komunikujúcimi stranami sú hlavný modul a zásuvné moduly, pričom na jednom komunikačnom konci môže stáť hlavný modul a na druhom zásuvný modul, alebo môžu zásuvné moduly komunikovať navzájom medzi sebou.

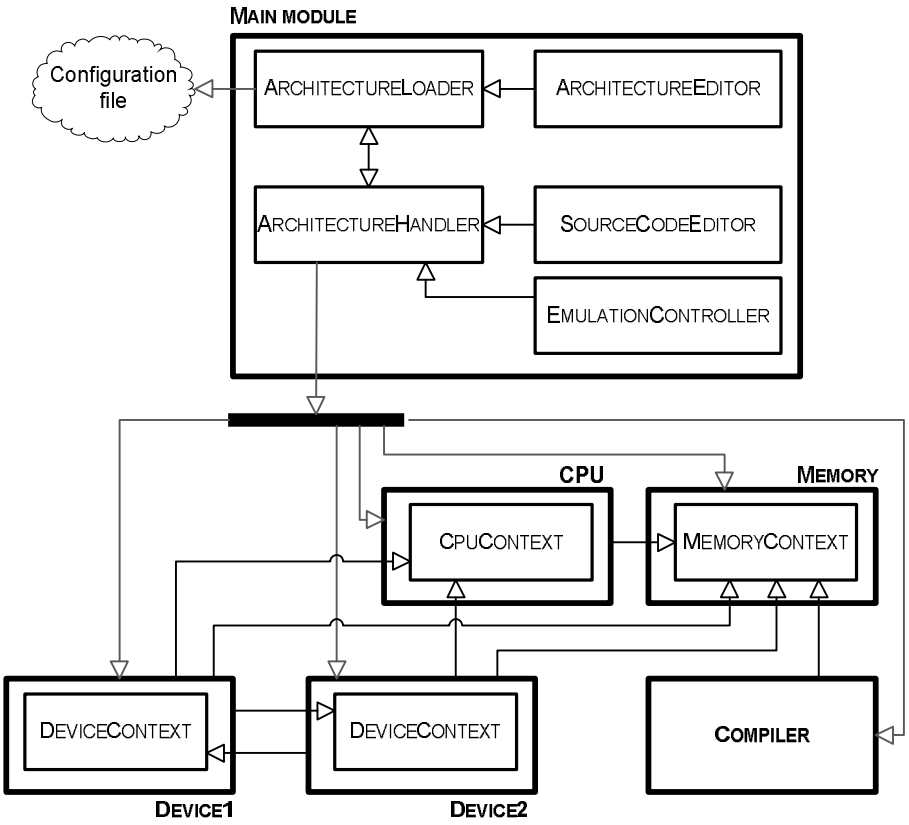
Komunikačný model zásuvných modulov verzie 2 nepodporoval hierarchické zapájanie zariadení (čiže zariadenia sa nemohli navzájom prepájať vôbec), striktne vyžadoval prepojenie CPU s operačnou pamäťou, a neumožňoval rozšírenie zásuvných modulov o ďalšiu funkcionalitu, čo ho nečinilo univerzálnym.

Od verzie verzie 3 sa tieto obmedzenia odstraňujú. Model umožňuje hierarchické prepájanie zariadení do ľubovoľnej hĺbky vnorenia a nevyžaduje prepojenie operačnej pamäte s CPU (čo sa však trochu vymyká Von-Neumannovskej predstave, no je to z dôvodu poskytnutia čo možno najväčšieho pohodlia pre tvorbu vlastných zásuvných modulov). Komunikačné rozhrania ohraničujú iba základné vlastnosti zásuvných modulov a existujú mechanizmy, ktorými môžu zásuvné moduly poskytovať aj neštandardnú funkcionalitu. Ďalším rysom je poskytnutie zásuvným modulom možnosť uložiť/obnoviť si vlastné, vnútorné (skryté) nastavenia pre každú architektúru zvlášť, čím je možné prispôsobiť si architektúru podľa vlastných požiadaviek.

Vývoj komunikačného modelu však stále nie je ukončený, vyvíja sa neustále. Dalo by sa povedať, že jeho vývoj je riadený požiadavkami na emulovanie. Jeho vývoj však už nespôsobuje rapídne zmeny v návrhu, pretože väčšinu bežných emulačných požiadaviek je schopný splniť už teraz.

1.2 Štruktúra platformy

Štruktúru platformy znázorňuje Obr.1.1. Smer šípok je chápaný nasledovne: majme objekt O_1 a objekt O_2 zo schémy. Nech O_1 ukazuje na O_2 ($O_1 \rightarrow O_2$). Potom len O_1 môže volať operácie O_2, a nie naopak (O_2 nemá prístup k O_1, ale výsledok operácie vrátiť môže).



Obr. 1.1: *Štruktúra platformy emuStudio*

Srdcom platformy ***emuStudio*** je hlavný modul. Z hľadiska implementácie ide o jednu aplikáciu napísanú v jazyku Java. Tento program zabezpečuje načítanie zásuvných modulov, ich prepojenie a tiež riadenie emulácie. Skladá sa z niekoľkých komponentov, ktoré toto všetko zabezpečujú:

ArchitectureLoader - správca konfigurácie architektúry. Spravuje konfiguračný súbor (ukladá a načítava konfigurácie), a vytvára inštanciu virtuálnej architektúry (cez komponent `ArchitectureHandler`).

ArchitectureEditor - editor abstraktných schém. Umožňuje vytváranie, editáciu a mazanie abstraktných schém, spolupracuje s komponentom `ArchitectureLoader`.

ArchitectureHandler - správca inštancie virtuálnej architektúry. Poskytuje ostatným komponentom inštancie zásuvných modulov a implementuje rozhranie pre ukladanie/načítavanie

nastavení zásuvných modulov.

SourceCodeEditor - editor zdrojového kódu. Umožňuje vytvárať a editovať zdrojové kódy pre zvolený prekladač, podporuje zvýrazňovanie syntaxe a označovanie riadkov a komunikuje priamo s prekladačom (cez komponent `ArchitectureHandler`).

EmulationController - správca emulácie. Riadi celú emuláciu, a je prostredníkom interakcie medzi virtuálnou architektúrou a používateľom.

Konkrétnu virtuálnu architektúru emulovaného počítača tvorí množina vybratých zásuvných modulov, ktoré sú navzájom prepojené podľa konfigurácie (schémy) danej virtuálnej architektúry. Zásuvné moduly sú prepojené priamo, bez použitia zberníc. Platforma emuStudio pozná štyri typy zásuvných modulov: kompilátor, CPU, operačná pamäť a periférne zariadenia. Jednotlivé zásuvné moduly budú popísané v samostatných častiach.

1.2.1 Kontext

Ako je možné vidieť aj na Obr. 1.1, zásuvné moduly s výnimkou kompilátora zahŕňajú špeciálny komponent nazývaný *kontext*. Prepájacie čiary medzi týmito zásuvnými modulmi vychádzajú z okraja znázorneného zásuvného modulu (napr. **DEVICE2**), ale ukazujú na kontext druhého zásuvného modulu (**CPUCONTEXT**). Znamená to, že zásuvný modul, ktorý je prepojený, resp. má požiadavky na druhý zásuvný modul, nemá k nemu úplný prístup. Môže pristupovať len ku kontextu požadovaného zásuvného modulu.

Úlohou kontextu je preto poskytnúť zásuvným modulom len funkcionalitu, ktorú tieto naozaj potrebujú počas behu emulácie, z hľadiska jej správneho fungovania. Dôležité je, aby neposkytoval žiadnu funkcionalitu navyše, aby ju zásuvné moduly nemohli zneužiť.

Dôsledkom je, že kontext berie na seba ďalšiu úlohu, a tou je ochrániť zásuvné moduly pred zneužitím a volaním citlivých funkcií (napr. na ovládanie samotného behu emulácie alebo na prepísanie používateľom nastavených parametrov zásuvného modulu). Jediný, kto má prístup ku všetkým zásuvným modulom ako takým, je hlavný modul. Zásuvné moduly ho preto (nutne) považujú za dôveryhodný.

1.2.2 Proces tvorby inštancie virtuálnej architektúry

Slovné spojenie „vytvorenie architektúry“ v skratke hovorí o počiatočnom načítaní a prepojení jednotlivých zásuvných modulov. Výsledkom je objekt, ktorý bude držať načítané zásuvné moduly, ich nastavenia a prepojenia ako jeden celok. Tento objekt nazývam virtuálna architektúra. V tejto časti popíšem spôsob, akým hlavný modul vytvára virtuálnu architektúru zo zvolenej konfigurácie.

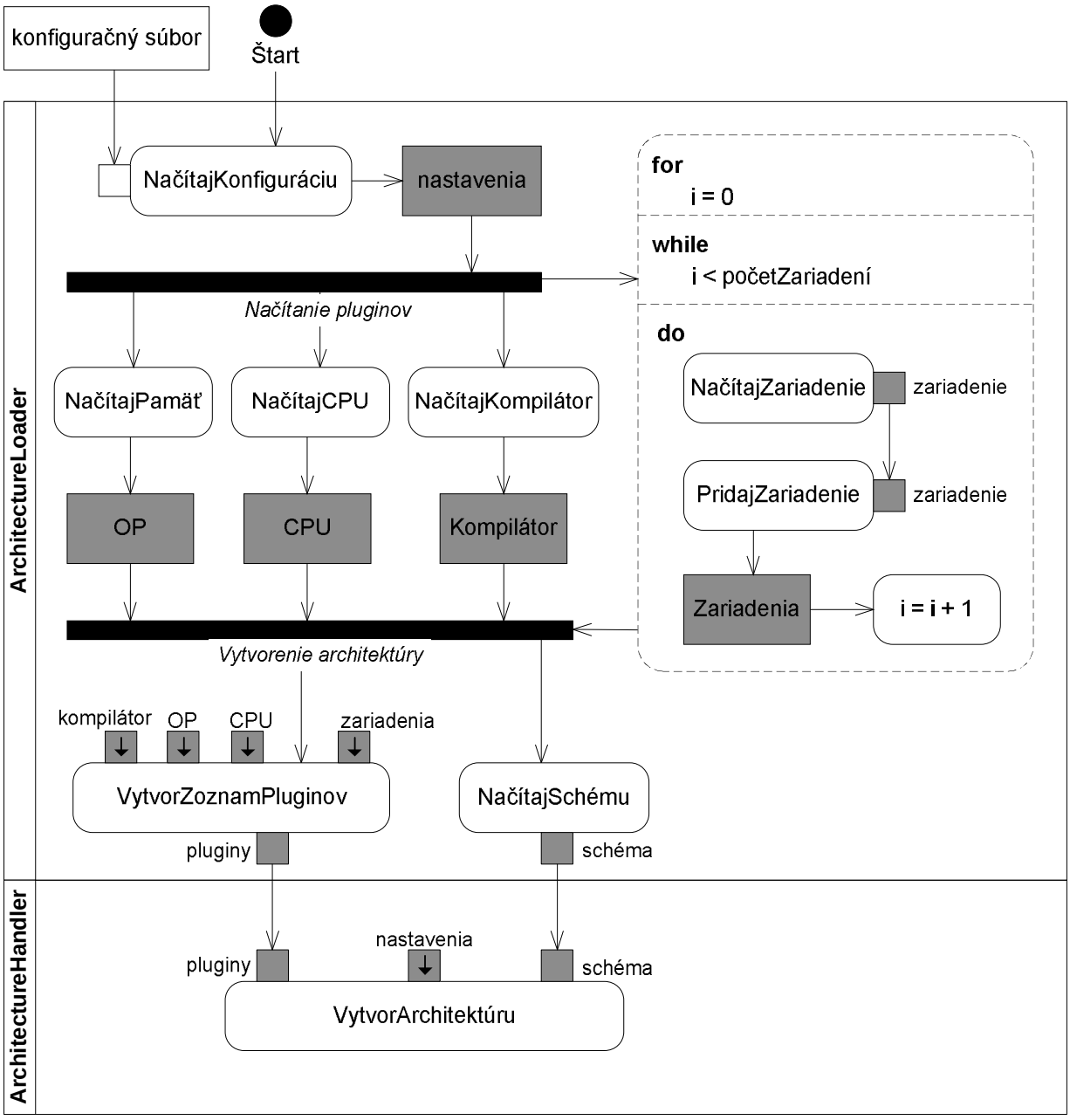
V prvom rade je potrebné načítať zvolenú konfiguráciu do pamäte - tj. všetky zásuvné moduly a ich nastavenia. Túto činnosť vykonáva komponent `ArchitectureLoader` (Obr. 1.1) v hlavnom module. Nasleduje ich inicializácia a prepojenie. Operácia je implementovaná v komponente `ArchitectureHandler`. Spoluprácou týchto dvoch komponentov vznikne virtuálna

architektúra, ktorá je uchovaná v komponente `ArchitectureHandler`. Detailnejší postup činností pri vytváraní virtuálnej architektúry je uvedený tu:

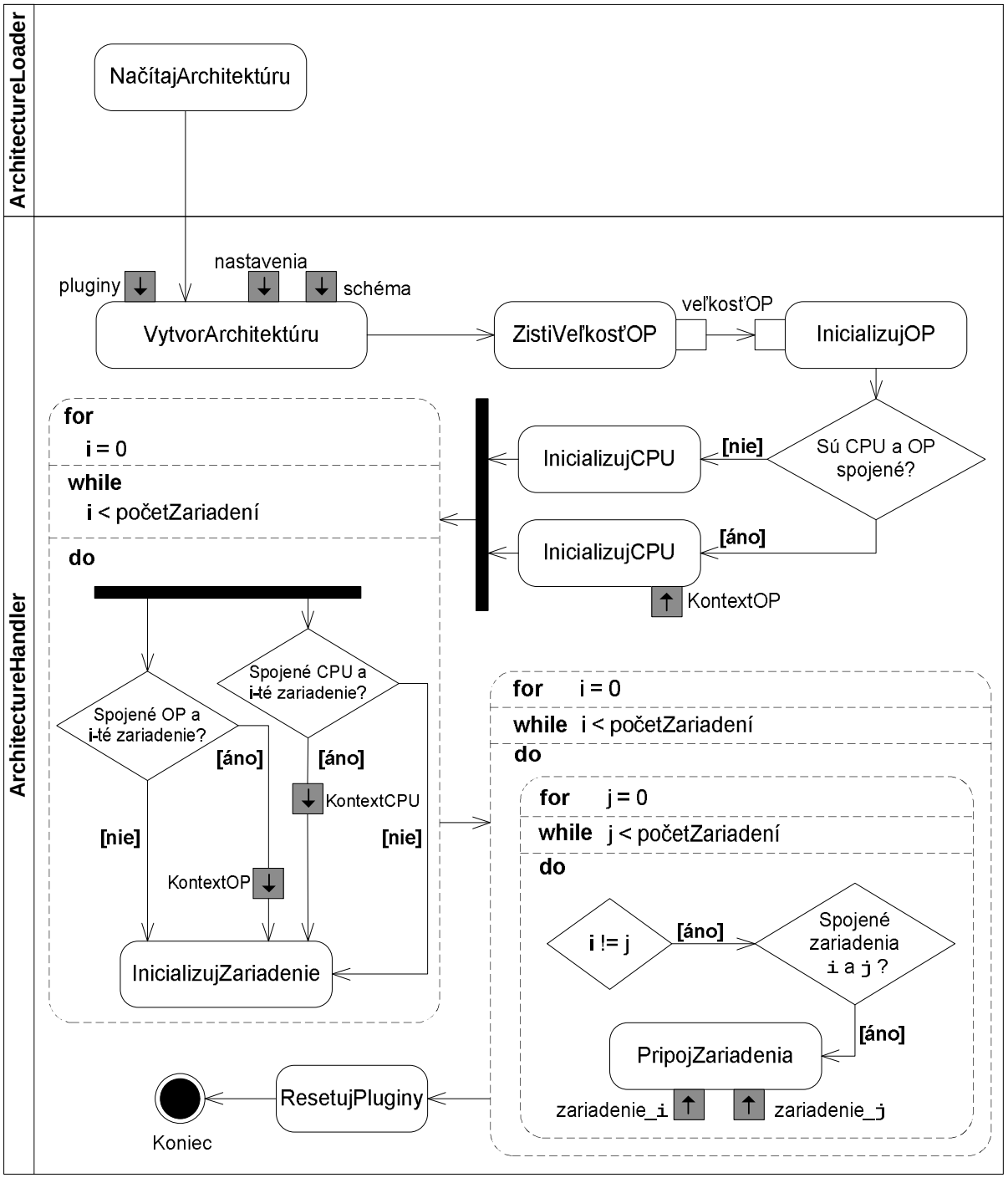
1. Komponent `ArchitectureLoader` načíta konfiguračný súbor zvolenej architektúry
2. Z načítaných nastavení tento komponent zistí typ virtuálnej CPU, OP a zariadení
3. Každý zistený zásuvný modul tento komponent načíta zo súboru, načítané dáta premení na inštanciu zásuvného modulu daného typu a nakoniec odovzdá všetky inštancie zásuvných modulov komponentu `ArchitectureHandler`
4. komponent `ArchitectureHandler` nájde zásuvné moduly, ktoré sa majú medzi sebou prepojiť, a to podľa nastavení v načítanom konfiguračnom súbore
5. Pre každú dvojicu zásuvných modulov, ktoré majú byť prepojené ich začne tento komponent prepájať nasledovne:
   (a) Prepájanie CPU s OP je realizované formou odovzdania parametra pri inicializácii CPU, pričom je realizované len jednosmerné zapojenie OP do CPU (CPU má požiadavky na OP). Prípadné prepojenie v druhom smere môže realizovať CPU, ak to kontext operačnej pamäte umožňuje (zmysel by to malo ak má operačná pamäť informovať CPU o svojej zmene a teda potrebuje zavolať CPU).
   (b) Prepájanie CPU so zariadeniami je realizované formou odovzdania parametra pri inicializácii zariadenia, pričom je realizované len jednosmerné zapojenie CPU do zariadení (zariadenia majú prístup k CPU). Zapojenie v druhom smere realizuje samotné zariadene, nie hlavný modul, pretože spôsob prepojenia sa môže líšiť v závislosti od kontextu daného CPU.
   (c) Prepájanie OP so zariadeniami je realizované formou odovzdania parametra pri inicializácii zariadenia, pričom je realizované len jednosmerné zapojenie OP do zariadenia (zariadenia majú požiadavky na OP).
   (d) Prepájanie zariadení medzi sebou je realizované obojsmerne, pričom je vykonaná kontrola, či zariadenie zapojenie umožňuje (v ľubovoľnom smere).

Diagramy aktivít vytvárania virtuálnej architektúry môžte vidieť na Obr. 1.2 a Obr. 1.3. Pre programátorov zásuvných modulov je dôležité poznať, ako sa budú jeho zásuvné moduly načítavať a inicializovať. Poznanie postupnosti volaní jednotlivých inicializačných metód pomôže programátorom lepšie pochopiť sémantiku týchto metód, ktoré musia implementovať.

Červené obdĺžniky reprezentujú objekty, obdĺžniky s hladkými hranami bez výplne sú vykonávané aktivity alebo činnosti. Diagram funguje na podobnom princípe ako klasický vývojový diagram. Bližšie informácie môže čitateľ nájsť napr. v [7].



Obr. 1.2: *Načítavací proces virtuálnej architektúry*



Obr. 1.3: *Inicializačný proces virtuálnej architektúry*

1.3 Kompilátor

Zásuvný modul *Compiler* reprezentuje kompilátor, teda prekladač zdrojového kódu do strojového kódu konkrétneho procesora. Logická je voľba takého kompilátora, ktorý kompiluje zdrojový kód pre procesor zvolený vo virtuálnej architektúre. Je však možné zvoliť aj iný kompilátor.

Výstupom kompilátora je súbor so strojovým kódom (napr. vo formáte Intel HEX [4]) a voliteľne je výstup presmerovaný aj do operačnej pamäte.

1.3.1 Prístup a bezpečnosť

Kompilátor pre svoju prácu potrebuje mať prístup k samotnému zdrojovému textu a z vyššie popísanej vlastnosti môže mať prístup aj k operačnej pamäti, ale to len vtedy, ak používateľ vyslovene požiada o kompilovanie aj priamo do operačnej pamäte.

Inicializácia kompilátora na predchádzajúcich diagramoch aktivít nie je zobrazená, pretože táto je realizovaná až po týchto aktivitách, keď je vytvárané grafické rozhranie (GUI) hlavného modulu. Pri inicializácii kompilátor ešte kontext pamäte nedostane, ten sa odovzdáva ako parameter pri volaní metódy kompilovania. Vtedy by mal kompilátor skontrolovať, či kontext pamäte podporuje. Ak kompilátor používa len štandardné operácie kontextu, nemusí ho kontrolovať až tak dôsledne. Kontrola sa dá realizovať veľmi jednoduchým spôsobom, popísaným v kapitole 2 - Implementácia.

1.3.2 Knižnica emu-ifaces.jar

Tomuto zásuvnému modulu patrí balíček `plugins.compiler`, a všetky rozhrania v ňom.

1.4 CPU

Zásuvný modul *CPU* reprezentuje virtuálny procesor. Je základom pre celú emuláciu, pretože riadenie behu emulácie v hlavnom module predstavuje vlastne riadenie behu procesora. Podľa predstavy Von-Neumanna je CPU zostava riadiacej a aritmetikej-logickej jednotky, ktorej hlavná činnosť je vykonávanie inštrukcií. Tieto inštrukcie sa fyzicky nachádzajú v operačnej pamäti spolu s dátami. Tieto inštrukcie sú emuluované umelo, ľubovoľnou emulačnou technikou.

V komunikačnom modeli verzie 2 bola CPU vždy s operačnou pamäťou nutne prepojená. Nový komunikačný model toto obmedzenie odstraňuje (čo sa vymyká Von-Neumannovskej predstave), a to z dôvodu, aby mal programátor-vývojár zásuvných modulov čo možno najväčšie pohodlie a možnosti pre tvorbu vlastných zásuvných modulov.

Programátor si teda bude môcť vytvoriť zásuvný modul zariadenia, ktoré sa bude chovať ako CPU (bude implementovať jeho funkcie) bez komunikácie s operačou pamäťou. Môže si teda vymyslieť aj špeciálne CPU alebo čipy, ktoré vykonávajú len činnosti závislé na vstupe pripojených zariadení a teda nie sú riadené programom z operačnej pamäte.

Operačná pamäť však vybratá byť musí (už len z dôvodu tu neopisovaného návrhu hlavného modulu, ktorý implementuje zobrazenie jej obsahu a grafického rozhrania).

Zásuvný modul obsahuje špeciálny komponent nazvaný *kontext procesora*, ktorého operácie sú špecifické pre konkrétnu CPU (okrem štandardných operácií môže ďalšie zvoliť programátor). Zariadenia, ktoré potrebujú mať prístup k CPU, dostanú k dispozícii len jej kontext. Preto kontext pre zariadenia predstavuje akési "pieskovisko", ktoré zariadeniam znemožňuje zasahovať do citlivých nastavení a riadení behu procesora. Ak má byť zariadenie prepojené s CPU, kontext by mal obsahovať operácie, ktoré pripojenie zariadení umožňujú.

Medzi základné operácie zásuvného modulu *CPU* patria: *step*, *stop*, *pause*, *execute*. Tieto operácie hovoria o tom, ako sa bude dať ovplyvniť chovanie CPU. Samozrejme všetky tieto funkcie v reálnom svete CPU nepodporujú a naopak určite existujú funkcie riadenia CPU, ktoré do modelu nie sú zahrnuté. Ide však väčšinou o funkcie, ktoré nie sú spoločné pre každé CPU, a preto ich podpora je voliteľná v rámci kontextu CPU.

Step - krok emulácie. CPU vykoná jeden inštrukčný cyklus (závisí od implementácie konkrétneho CPU, všeobecne ide o fázy *Fetch*, *Decode*, *Execute* a *Store*)

Stop - Zastaví bežiacu, alebo pozastavenú emuláciu. CPU sa uvedie do stavu, v ktorom už ďalej nie je schopný vykonávať inštrukcie, až do svojho resetu.

Pause - Pozastaví bežiacu emuláciu. CPU sa uvedie do stavu, v ktorom prestane vykonávať inštrukcie, ale jeho kontext (hodnoty jeho vnútorných registrov, signálov a nastavení) je nezmenený po posledne vykonanej inštrukcii. Z tohto stavu sa CPU môže znovu uviesť do behu.

Execute - Spustí pozastavenú emuláciu. CPU sa uvedie do stavu, v ktorom neustále vykonáva inštrukcie (spôsobom svojej konkrétnej implementácie). Podmienka jeho zastavenia alebo pozastavenia je daná jedine konkrétnou implementáciou zásuvného modulu.

1.4.1 Prístup a bezpečnosť

Prístup CPU (už z koncepcie Von-Neumanna) k okoliu je priamočiary. CPU pri svojej hlavnej činnosti (ktorou je vykonávanie inštrukcií) číta inštrukcie z OP, teda musí mať prístup k nej (pritom je potrebné uvedomiť si, že požiadavky na zápis/čítanie z OP vychádzajú z CPU, teda prepojenie stačí, ak bude jednosmerné[1]). Všimnite si aktivitu *InicializujCPU* na Obr. 1.3. Ak je CPU spojené s OP, rozhodovací blok "*Spojenie OP a CPU?*" spustí túto aktivitu s parametrom kontextu OP (teda CPU bude mať prístup k OP), v opačnom prípade sa táto aktivita spustí bez parametra (CPU nebude mať prístup k OP).

Zariadenia, ktoré s CPU komunikujú sú pomocou hlavného modulu pri načítavaní virtuálnej architektúry prepojené len jednosmerne. Ak aj CPU potrebuje mať prístup k zariadeniam, prepojenie v druhom smere musí vyriešiť programátor - použitím „neštandardizovaného" kontextu CPU, ktorý bude zariadeniam prístupný a schopný prijímať/odmietať požiadavky týchto zariadení o ich zapojenie do CPU.

[1] tu a aj všade, kde to nebude explicitne inak uvedené, sa termín „prepojenie" týka softvérovej implementácie tejto vlastnosti

Kontext CPU môže obsahovať ľubovoľné operácie, nekompatibilné s inými procesormi. Zariadenia podporujúce daný procesor by mali jeho kontext poznať. Ak však zariadenia pracujú len so štandardnými operáciami kontextu, nie je potrebné kontrolu vykonať, jedine ak by zariadenia striktne vyžadovali štandardný kontext. Z dôvodu bezpečnosti (aby zariadenia nemohli riadiť emuláciu), zariadenia prepojené s procesorom dostanú k dispozícii iba jeho kontext. Všimnite si aktivitu *InicializujZariadenie* na Obr. 1.3. V prípade, že je zariadenie s CPU prepojené, rozhodovací blok "*Spojene CPU+device[i]?*" posunie ďalej kontext procesora, inak nie. Potom aktivita *InicializujZariadenie* (reprezentujúca metódu implementovanú v zásuvnom module zariadenia) dostane ako jeden z parametrov buď kontext procesora (ak je zariadenie s CPU prepojené), alebo nič miesto neho (ak nie je).

Tým, že zariadenia dostanú v tejto metóde kontext procesora, si vedia sami overiť, či je pre nich procesor vhodný alebo nie - či prepojenie je možné realizovať. Ak nie, zásuvný modul by mal vypísať chybové hlásenie a skončiť inicializáciu s vrátením chyby. To isté platí aj pre CPU, keď dostane kontext operačnej pamäte. CPU by mala vykonať kontrolu, či je kontext pamäte podporovaný. Kontrola sa realizuje veľmi jednoducho a je popísaná v kapitole 2 - Implementácia.

1.4.2 Knižnica emu-ifaces.jar

Tomuto zásuvnému modulu patrí balíček `plugins.cpu`, a všetky rozhrania v ňom.

1.5 Operačná pamäť

Operačná pamäť (ďalej OP) reprezentuje virtuálnu operačnú pamäť (niečo ako odkladací priestor pre dáta a inštrukcie). Všeobecne sa OP skladá z buniek operačnej pamäti, pričom tvar, typ, rozmer a hodnota buniek nie sú bližšie špecifikované. Bunky sú umiestnené sekvenčne a teda sa dá určiť jednoznačná pozícia bunky v rámci pamäte (nazývaná adresa).

Základnými operáciami zásuvného modulu sú *read*, ktorej funkciou je prečítať a vrátiť hodnotu konkrétnej bunky OP, a *write*, ktorej funkciou je naopak do konkrétnej bunky pamäte zapísať špecifikovanú hodnotu. Tieto operácie sú základnými operáciami každej OP, no ich implementácia (teda ako sa napr. adresa reprezentuje významovo) môže byť rôzna. Rovnako tak typ prenášanej hodnoty nie je bližšie špecifikovaný.

OP obsahuje komponent nazvaný *kontext pamäte*, ktorý okrem štandardných operácii (čítanie a zápis buniek pamäte) môže obsahovať aj špecifické operácie konkrétneho zásuvného modulu. Zariadenia (a kompilátor), ktoré potrebujú mať prístup k OP (napr. zariadenia využívajúce DMA), dostanú k dispozícii len tento kontext (pri volaní metódy kompilovania, nie pri inicializácii kompilátora, ako je to napr. v prípade zariadení a kontextu OP). Preto operácie v kontexte musia byť bezpečné z hľadiska použitia pre ostatné zásuvné moduly.

1.5.1 Prístup a bezpečnosť

OP nie je typom zariadenia[2], ktoré priamo ovplyvňuje beh iných zariadení. Preto všetky prepojenia s OP sú len jednosmerné, pričom sa OP vždy zapojí do zariadenia, a nie naopak. Z toho vyplýva, že zariadenie môže využívať služby OP, ale tá nemôže využívať služby zariadenia. Z toho vyplýva, že OP nepotrebuje mať prístup k žiadnemu zásuvnému modulu.

Pod inicializáciou OP si predstavujem vytvorenie poľa buniek operačnej pamäti, ktoré budú pripravené na zápis alebo čítanie. Ak je OP implementovaná ako dynamické pole pevnej veľkosti, je potrebné poznať jeho veľkosť už pri inicializácii OP (všimnite si aktivity *ZistiVelkostPamate* a *InicializujPamat* na Obr. 1.3). Existujú však aj triedy jazyka Java (napr. `java.util.ArrayList`) zachovávajúce vlastnosti dynamického poľa s premenlivou veľkosťou. Programátor by si však mal byť vedomý ich oveľa nižšiemu výkonu oproti dynamickým poliam a zvážiť ich použitie, pretože rýchlosť komunikácie OP s CPU je kritická.

Pokročilé techniky OP ako bankovanie, stránkovanie a segmentácia, ktoré významovo ovplyvňujú pojem „adresa“ (použitím týchto techník môže mať adresa napr. rôzny tvar, niekedy rovnaká adresa môže ukazovať na rôzne skutočné pozície buniek alebo niekedy rozsah adresy nemusí obsiahnuť rozsah celej OP), nie sú základnými technikami, ale využívajú sa pomerne často. Riešením je teda opäť použitie „neštandardizovaného“ kontextu OP, ktorý bude zariadeniam prístupný. Zariadenia by teda mali poznať kontexty podporovaných OP, a tak sa môžu samotné zariadenia rozhodnúť, či podporujú prepojenie s ponúkanou OP. Ak zariadenia používajú len štandardné operácie kontextu, nie je potrebné kontext kotrolovať, len ak by zariadenia vyžadovali výhradne štandardný kontext. Kontext OP môže tiež obsahovať metódy na realizovanie prepojenia v druhom smere, ktoré štandardne nie je podporované. Môže sa to hodiť, ak potrebuje byť nejaké zariadenie informované o zmene stavu OP.

Všimnite si aktivitu *InicializujZariadenie* na Obr. 1.3. V prípade, že je zariadenie s OP prepojené, rozhodovací blok "*Spojene OP+device[i]?*" posunie ďalej kontext pamäte, inak nie. Potom aktivita *InicializujZariadenie* (reprezentujúca metódu implementovanú v zásuvnom module zariadenia) dostane ako jeden z parametrov buď kontext pamäte (ak je zariadenie s OP prepojené), alebo nič miesto neho (ak nie je).

1.5.2 Knižnica emu-ifaces.jar

Tomuto zásuvnému modulu patrí balíček `plugins.memory`, a všetky rozhrania v ňom.

1.6 Periférne zariadenia

Periférne zariadenia sú virtuálne zariadenia, emulujúce funkcionalitu skutočných zariadení. Vo všeobecnosti nie je účel alebo typ zariadení bližšie špecifikovaný, ani štandardizovaný, teda zásuvné moduly ani nemusia reprezentovať reálne zariadenia.

Do určitej miery (táto miera tiež nie je obmedzená) zariadenia pracujú samostatne (a reagujú na výzvy pripojených zásuvných modulov), prípadne interagujú s používateľom. Zariadenia

[2] v tomto oddieli je pojem *zariadenie* chápaný aj ako CPU, OP alebo periférne zariadenie

môžu komunikovať s zásuvným modulom procesora, a/alebo OP, a/alebo s inými zariadeniami.

Komunikačný model podporuje hierarchické zapojenie zariadení (ktoré tak môžu medzi sebou komunikovať bez podpory CPU). Zbernica kvôli jednoduchosti nie je implementovaná vôbec (v modeli nie sú žiadne zbernice - jedine ak by bola zbernica priamo naprogramovaná ako zariadenie). Je vhodné vyhnúť sa prepojeniam zasahujúcich do viac realizačného charakteru skutočného sveta (ktorým zbernica nepochybne je), naopak je vhodné, aby mal komunikačný model viac logický charakter (výhodou je okrem ľahšej implementácie aj prehľadnosť a jednoduchosť).

Každé zariadenie obsahuje komponent *kontext zariadenia*, ktorého štandardné operácie sú vstup a výstup zariadenia (*Input* a *Output*). Hlavnou myšlienkou práce každého zariadenia je, že všetky informácie v rámci komunikácie prechádzajú do zariadenia cez jeho vstup a vychádzajú zo zariadenia ako výstup. Zariadenia potom môžu byť vstupné, výsupné, alebo vstupno-výstupné. Kontext zariadenia môže byť konkrétnym zásuvným modulom rozšírený o neštandardné operácie. Iné zariadenia, ktoré potrebujú mať prístup k tomuto zariadeniu dostanú k dispozícii len jeho kontext a preto operácie v kontexte musia byť bezpečné z hľadiska použitia pre ostatné zásuvné moduly.

1.6.1 Prístup a bezpečnosť

Ako už bolo popísané skôr, zariadenie môže mať prístup k CPU, OP a k iným zariadeniam. Zariadenia medzi sebou môžu byť prepojené jednosmerne, alebo aj obojsmerne, pričom hlavný modul sa vždy snaží o prepojenie obojsmerné, a ak to nejde, skúša prepojenie jednosmerné najprv v jednom smere a ak sa ani to nepodarí, potom v druhom smere.

Z dôvodov podobných ako aj pri popise zásuvných modulov *CPU* a *OP*, aj zásuvný modul zariadenia obsahuje jednoznačný a jedinečný kontext, ktorý môže implementovať okrem štandardných operácií vstupu/výstupu zariadenia aj neštandardné operácie, ktoré robia zariadenie jednoznačným.

Zariadenia, ktoré akceptujú pripojenia iných zásuvných modulov by mali poznať ich kontexty, aby pri procese prepájania mohli overiť, či pripájané zariadenie je vôbec možné pripojiť. Kontrola sa dá realizovať veľmi jednoduchým spôsobom, popísaným v kapitole 2 - Implementácia. V prípade, že zariadenia pracujú len so štandardnými operáciami kontextu, nie je nutné kontrolu vykonať, jedine ak by zariadenia vyžadovali výhradne štandardný kontext.

1.6.2 Knižnica emu-ifaces.jar

Tomuto zásuvnému modulu patrí balíček `plugins.device`, a všetky rozhrania v ňom.

Kapitola 2

Implementácia

2.1 Inštalácia nového zásuvného modulu

Každý zásuvný modul musí byť reprezentovaný jediným súborom typu JAR (Java ARchive). Zásuvné moduly rovnakého typu sa ukladajú do určených podadresárov, štruktúra adresárov nainštalovanej platformy ***emuStudio*** je nasledovná:

`\compilers` → Zásuvné moduly kompilátorov
`\config` → Súbory konfigurácií
`\cpu` → Zásuvné moduly všetkých CPU
`\devices` → Zásuvné moduly zariadení
`\lib` → Potrebné knižnice
`\mem` → Zásuvné moduly operačných pamätí

Inštalácia zásuvného modulu teda spočíva iba v nakopírovaní súboru zásuvného modulu do príslušného adresára, hlavný modul zásuvný modul automaticky nájde.

2.2 Štruktúra zásuvného modulu

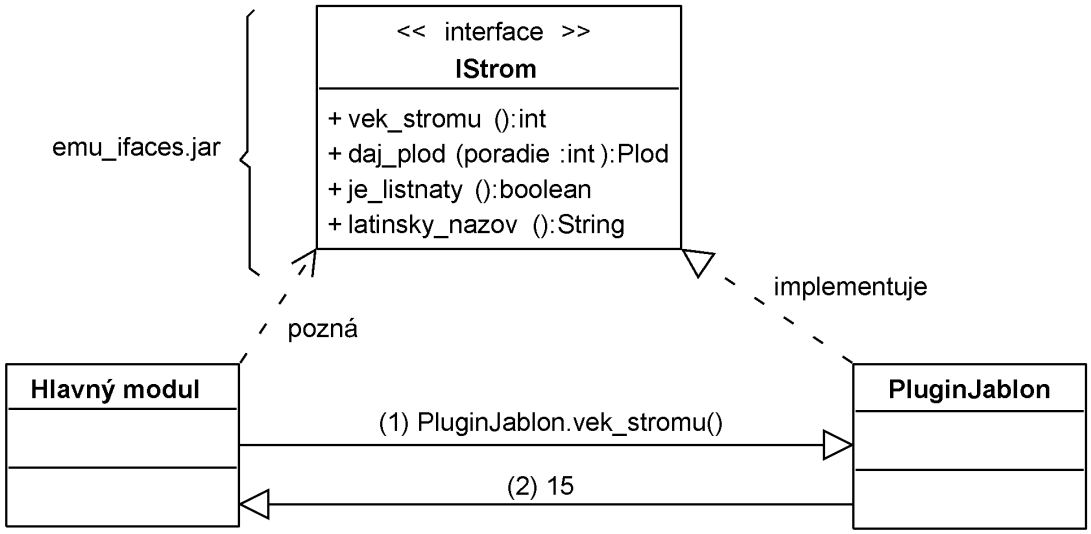
V reči programovania je zásuvný modul obyčajnou knižnicou - archívom typu JAR, ktorý zahŕňa adresáre (balíčky) a súbory (skompilované triedy a resources - prostriedky). Tvorbou takého archívu sa nebudem zaoberať, čitateľ - programátor v Jave by to mal vedieť. Existujú však konvencie, ktorých by sa mal programátor zásuvných modulov držať, tie uvediem v nasledujúcich častiach.

Určitá trieda (alebo triedy) v tejto knižnici musí implementovať jedno alebo viac *rozhraní* (interfaces)[1], definovaných v externej knižnici platformy. Konkrétna skupina (alebo sada) rozhraní, ktoré musí zásuvný modul implementovať, nazývam **štruktúrou zásuvného modulu**.

Každý zásuvný modul má rozdielnu štruktúru, ale spoločnú pre ten istý typ. To znamená, že kompilátory implementujú inú sadu rozhraní, ako napr. procesory, no každý kompilátor implementuje tú istú sadu rozhraní. Hlavný modul komunikuje so zásuvným modulom práve

[1] Jedná sa o pojem z jazyka Java

prostredníctvom volaní metód objektov (inštancií tried), implementujúcich známe rozhrania. Rozhrania sú teda známe hlavnému modulu, čiže ak zistí, že nejaká trieda zásuvného modulu implementuje známe rozhranie, vie s jej objektom komunikovať (volaním príslušných metód). Príklad je na Obr. 2.1.



Obr. 2.1: *Príklad komunikácie hlavného modulu a zásuvného modulu*

Takže existujú rozhrania určené len pre kompilátory, ďalšie len pre procesory, atď. Existujú však aj rozhrania, ktorých implementácia - trieda, je v hlavnom module a zásuvné moduly (v rámci niektorých metód implementovaných rozhraní) dostanú jej objekt ako parameter. Takýmto spôsobom je možné, aby zásuvné moduly využívali funkcie hlavného modulu, a hlavný modul na druhej strane komunikuje so zásuvnými modulmi cez metódy danej sady rozhraní.

Ako už bolo spomenuté, všetky potrebné rozhrania sa nachádzajú v externej knižnici - súbor `lib\emu_ifaces.jar`. Pri programovaní zásuvného modulu preto treba použiť túto knižnicu. Popis všetkých dostupných rozhraní tejto knižnice je v jej dokumentácii typu *javadoc*.

2.2.1 Inicializačný proces

Po spustení emulátora, a po voľbe konfigurácie hlavný modul zistí, ktoré zásuvné moduly má načítať. Tieto postupne načíta do pamäte z disku, identifikuje v ňom všetky triedy, balíky a prostriedky. V rámci načítaných tried nájde takú triedu, ktorá implementuje hlavné rozhranie pre daný typ zásuvného modulu (hlavné rozhranie je len jedno) a vytvorí *inštanciu* tejto triedy. Pri vytváraní sa použije konštruktor s jedným parametrom typu `java.lang.Long`, ktorý táto trieda musí mať. V prípade, že ho nemá, emulátor skončí s chybovým hlásením.

Tvar konštruktora triedy "`PluginImplementation`" implementujúcej vymyslené hlavné rozhranie "`IPluginInterface`" je takýto:

```
public class PluginImplementation implements IPluginInterface {
    private long hash;
    ...

    public PluginImplementation(java.lang.Long hash) {
        this.hash = hash;
        ...
    }
    ...
}
```

Parameter konštruktora typu `java.lang.Long` s názvom `hash` predstavuje jednoznačné číslo zásuvného modulu (identifikátor). Tento identifikátor vytvára hlavný modul nejakým algoritmom a pridelí ho zásuvného modulu pri vytvorení jeho inštancie. Každý zásuvný modul dostane iný identifikátor. Týmto identifikátorom sa zásuvný modul prezentuje hlavnému modulu pri požiadavkách o citlivé informácie alebo úkony, ku ktorým iné zásuvné moduly nemajú mať prístup (napríklad načítavanie a ukladanie nastavení). Po každom novom spustení emulátora sa vytvárajú rôzne identifikátory a majú platnosť počas celej doby jeho behu.

2.2.2 Čo zásuvný modul môže

- môže sa skladať z ľubovoľného počtu tried
- môže využívať ľubovoľný počet balíčkov a ľubovoľné úrovne ich vnorenia
- môže využívať aj iné *resources* (prostriedky - súbory), ktoré zásuvný modul využije v rámci svojej implementácie - ide hlavne o obrázky, ikony, fonty, . . .

2.2.3 Čo zásuvný modul musí

- každý skompilovaný zásuvný modul musí byť zbalený do jedného výsledného `jar` súboru
- aby zásuvný modul mohol fungovať a bol správne identifikovaný, musí byť umiestnený do príslušného podadresára (pozri časť Úvod)
- niektorá trieda (alebo triedy) zásuvného modulu musí implementovať sadu rozhraní, ktorá prislúcha danému typu zásuvného modulu

  Pritom rozhrania nesmú byť definované v zásuvnom module, ale zásuvný modul na nich musí odkazovať prostredníctvom častokrát spomínanej externej knižnice `emu_ifaces.jar`. Popis toho, ako to urobiť už nespadá do kontextu tohto dokumentu, ale programátor v Jave by to mal ovládať. V prostredí Netbeans aj Eclipse to je možné spraviť veľmi jednoducho.
- trieda implementujúca hlavné rozhranie musí obsahovať verejný konštruktor s jedným parametrom typu `java.lang.Long`. Tento konštruktor sa bude volať pri vytváraní inštancie tejto triedy

2.2.4 Čo zásuvný modul nesmie (alebo nemôže)

- Ak to nebude explicitne uvedené inak, nesmú existovať viaceré implementácie (triedy) rovnakého rozhrania. V prípade ich existencie nie je jednoznačné, ktoré z nich hlavný modul uzná za to „hlavné“.
- Nemôže používať externé knižnice nedodávané s emulátorom, iba ak by sám zásuvný modul zabezpečil ich načítanie.
- Nesmie používať viacero baličkov na prvej úrovni vnorenia, tj. všetky balíčky musia byť umiestnené do hlavného balíčka a jeho názov musí byť jednoznačný v rámci všetkých zásuvných modulov, najlepšie sa hodí názov súboru zásuvného modulu bez prípony.

  Je to z dôvodu, aby sa predišlo prepísaniu tried s rovnakým názvom implementovaných v rôznych zásuvných moduloch. Virtuálny stroj Javy (JVM) ich nedokáže odlíšiťa preto hrozí, že druhý výskyt triedy s rovnakým názvom prepíše predošlú triedu. Napríklad, ak sa trieda grafického okna konfigurácie v zásuvnom module ”P1” nazýva `gui.ConfigDialog`, a v zásuvnom module ”P2” tiež `gui.ConfigDialog`, tak aj keď ide o odlišné triedy, niektorá z nich bude tou druhou prepísaná (podľa poradia načítavania zásuvných modulov). Aby sa tomu predišlo, tak sa všetky balíky umiestnia do hlavného balíčka s názvom zásuvného modulu, teda naše triedy budú mať názvy: `P1.gui.ConfigDialog` a `P2.gui.ConfigDialog`, a už ich aj JVM rozlíši.

2.3 Štruktúra knižnice emu_ifaces.jar

Obrázok č. 2.2 zobrazuje štruktúru knižnice `emu_ifaces.jar`, ktorá je kľúčovou knižnicou pri programovaní zásuvných modulov.

Okrem balíčkov a rozhraní, ktorých implementácia má byť v jednotlivých zásuvných moduloch, obsahuje triedu s názvom `runtime.StaticDialogs`. Táto obsahuje statické metódy, ktoré majú odbremeniť zásuvné moduly od implementácie fundamentálnych a veľmi často používaných metód. Jedná sa o metódy slúžiace na zobrazovanie chybových (`showErrorMessage`) a informačných (`showMessage`) hlásení; a na zistenie verzie komunikačného modelu (`getModelVersion`, `getModelMinor`). Zásuvné moduly si tak môžu overiť, či daný komunikačný model podporujú.

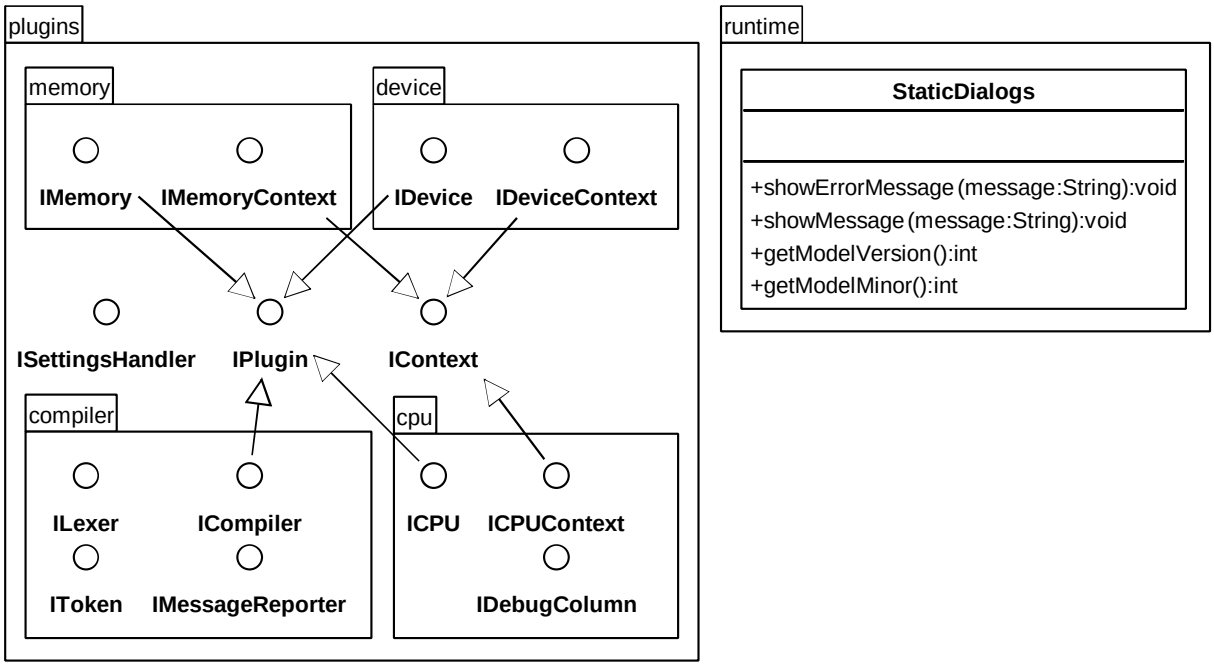
Napríklad výpis chybového hlásenia zásuvný modul môže na ľubovoľnom mieste vyvolať takto:

```
StaticDialogs.showErrorMessage(”Strašná chyba!”);
```

Chybové hlásenie sa zobrazí ako grafický dialóg, volaním metódy

```
javax.swing.JOptionPane.showMessageDialog()
```

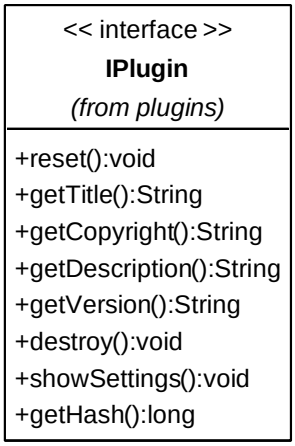
Podrobný popis všetkých balíčkov, tried, rozhraní a ich metód je v dokumentácii knižnice `emu_ifaces.jar` vo formáte *javadoc*.



Obr. 2.2: *Štruktúra knižnice emu_ifaces.jar*

2.3.1 Rozhranie *IPlugin*

Rozhranie `plugins.IPlugin` (Obr. 2.3) je rozhraním, od ktorého dedí každé hlavné rozhranie každého zásuvného modulu. Obsahuje fundamentálne metódy, ktoré by mal implementovať každý zásuvný modul, bez ohľadu na svoj typ.



Obr. 2.3: *Rozhranie IPlugin*

V rozhraní sú definované štyri obzvlášť zaujímavé metódy:

- Metódu `reset()` by mal zásuvný modul implementovať tak, že po jej volaní zásuvný modul zmení svoj vnútorný stav (bližšie nešpecifikovaný) na stav, aký by mal hneď po spustení (teda po inicializácii). Dalo by sa povedať, že ide aj o umelé vyvolanie re-inicializácie zásuvného modulu. Môžme si to predstaviť, ako keď na reálnom počítači stlačíme tlačidlo RESET.

  Túto metódu volá hlavný modul pre všetky zásuvné moduly vždy na konci inicializačného procesu (aktivita *ResetujPluginy* na Obr. 1.3), a pre CPU môže byť volaná aj kedykoľvek neskôr (viď ďaleký Obr. 2.11).

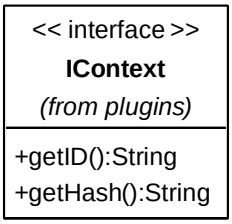
- Metóda `destroy()` je naopak volaná pri ukončovacom procese, ktorý však nie je zobrazený na žiadnom obrázku. Zásuvný modul volaním tejto metódy môže predpokladať, že sa emulácia *natrvalo zastavila*, a teda môže (a mal by) uvoľniť všetky prostriedky, ktoré používa, ako napr. objekty, GUI, sieťové spojenia, otvorené súbory, atď.

- Každý zásuvný modul má právo zobraziť vlastné GUI okno s nastaveniami. A práve na tento účel slúži metóda `showSettings()`. Je volaná z hlavného modulu, keď používateľ vyvolá nastavenia. V súčasnosti sa z hlavného modulu volajú touto metódou iba nastavenia periférnych zariadení, ale nie je na škodu, ak ostatné zásuvné moduly túto metódu implementujú a zo svojho iného GUI (napr. status okno CPU) budú mať napr. tlačidlo vyvolávajúce túto metódu.

- Metóda `getHash()` má vrátiť hash, pridelený v konštruktore hlavného rozhrania zásuvného modulu (viď časť 2.2.1).

2.3.2 Rozhranie *IContext*

Toto rozhranie definuje metódy, ktoré sú dedené každým kontextom (kontextom pamäte, procesora a aj kontextom zariadenia). Je zobrazené na Obr. 2.4.



Obr. 2.4: *Rozhranie IContext*

Rozhranie definuje len dve metódy, ktoré zohrávajú veľmi dôležitú úlohu pri rozpoznávaní kontextu inými zásuvnými modulmi.

- Metóda `getID()` má vrátiť jednoznačný identifikačný reťazec, obvykle zložený z nejakých slov. Tento reťazec môže byť vytvorený spojením druhu kontextu a názvu zásuvného modulu (napr. "brainduck-mem-context"). Týmto reťazcom zásuvný modul môže do určitej miery skontrolovať, či daný kontext podporuje.
- Metóda `getHash()` má vrátiť špeciálny hash reťazec, ktorý slúži na sofistikovanejší spôsob kontroly správneho kontextu. Vytvorenie tohto hashu popisujem v nasledujúcej časti.

2.3.2.1 Tvorba reťazca "hash" pre kontext

Hash reťazec (nezáleží na tom, aký je dlhý), je odvodený z úplných mien všetkých metód kontextu, ktoré sú abecedne zoradené a pospájané do jedného reťazca, oddelené bodkočiarkou (`;`). Výsledný reťazec musí byť následne prevedený hash funkciou MD5.
Dôvody použitia takého zložitého hashu sú tieto:

- názvy tried kontextov môžu byť rôzne, teda názov triedy nemôže byť použité ako kritérium pre rozpoznávanie kontextu
- ostáva teda jediná vec, ako kontext identifikovať - použiť metódu(y) na to určenú(é).
- Ak by sme použili len metódu `getID()`, identifikácia by neodrážala ďalšie zmeny kontextu v priebehu jeho vývoja. Vymýšľanie stále nového názvu pri nepatrnej zmene vedie k zavedeniu verzie kontextu. Avšak metódy na udržovanie verzie kontextu sa mi zdajú na tento účel nevhodné a aj zložité, keďže už máme verziu zásuvného modulu a v rámci neho by mali ešte existovať ďalšie pod-verzie...

Posledný dôvod zavážil najviac, pretože identifikácia kontextu by mala odrážať jeho zmenu. Metódou `getHash()` vrátim obyčajný reťazec, nijak zvlášť dlhý a zložitosť sa javí len pri výpočte tohto hashu, ktorý je realizovaný mimo zásuvného modulu.

Nasleduje algoritmus výpočtu hashu štandardného kontextu pamäte *IMemoryContext*, uvedený ako trieda s názvom `impl.MainClass`:

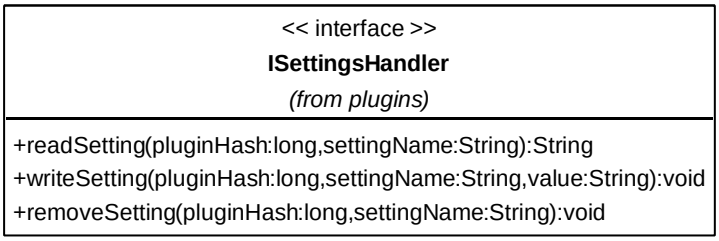
```
package impl;

import java.lang.reflect.Method;
import java.security.MessageDigest;
import plugins.memory.IMemoryContext;

public class MainClass {

    private static String convertToHex(byte[] data) {
        StringBuffer buf = new StringBuffer();
        for (int i = 0; i < data.length; i++) {
            int halfbyte = (data[i] >>> 4) & 0x0F;
            int two_halfs = 0;
```

```
            do {
                if ((0 <= halfbyte) && (halfbyte <= 9))
                    buf.append((char) ('0' + halfbyte));
                else
                    buf.append((char) ('a' + (halfbyte - 10)));
                halfbyte = data[i] & 0x0F;
            } while(two_halfs++ < 1);
        }
        return buf.toString();
    }

    public static String MD5(String text) {
        try {
            MessageDigest md;
            md = MessageDigest.getInstance("MD5");
            byte[] md5hash = new byte[32];
            md.update(text.getBytes("iso-8859-1"), 0, text.length());
            md5hash = md.digest();
            return convertToHex(md5hash);
        } catch(Exception e) {}
        return null;
    }

    public static void main(String[] args) {
        int i;
        Method[] methods;
        String hash = "";

        // získame všetky metódy rozhrania
        methods = IMemoryContext.class.getMethods();

        // teraz odvodíme reťazec pospájaných mien
        for (i = 0; i < methods.length; i++) {
            hash += methods[i].toGenericString() + ";";
        }
        System.out.println(MD5(hash));
    }
}
```

2.3.3 Rozhranie *ISettingsHandler*

Toto rozhranie (Obr. 2.5) slúži zásuvným modulom na manipuláciu ich interných nastavení, v rámci jednej konfigurácie. Všetky nastavenia sú čítané a ukladané do konfiguračného súboru

aktuálne zvolenej konfigurácie, teda pre rôzne konfigurácie môžu mať zásuvné moduly rôzne nastavenia. Rozhranie implementuje hlavný modul a jeho objekt odovzdáva zásuvným modulom pri ich inicializácii.



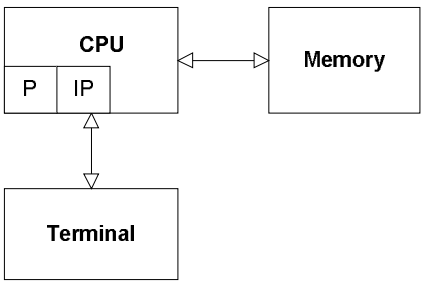
Obr. 2.5: *Rozhranie ISettingsHandler*

Použitie týchto metód je viacmenej jednoznačné a priamočiare. Doplním len poznámku, že parameter `pluginHash` je hash, ktorý zásuvný modul obdržal v konštruktore triedy implementujúcej hlavné rozhranie zásuvného modulu (viď časť 2.2.1). Keďže zásuvné moduly nepoznajú hashe ostatných zásuvných modulov (ten je/mal by byť prideľovaný po každom štarte iný), nemôžu zasahovať do nastavení ostatných zásuvných modulov. Tento parameter teda predstavuje akýsi "preukaz", ktorým sa zásuvný modul identifikuje.

2.4 BrainDuck

V rámci popisu implementácie jednotlivých zásuvných modulov budem postupne implementovať vymyslenú architektúru primitívneho počítača, ktorú som odvodil z ezoterického jazyka *brainfuck* [9]. Táto architektúra je Turingovsky úplná, teda je na nej možné vyjadriť ľubovoľný program, ktorý pracuje podľa nejakého algoritmu vypočítateľnom na Turingovom stroji. Jazyk a architektúru som veľmi mierne pozmenil, a nazval som ju "BrainDuck".

Na Obr. 2.6 je možné vidieť štruktúru tejto architektúry.



Obr. 2.6: *Štruktúra architektúry BrainDuck*

Procesor má iba dva registre:

- `P` - reprezentuje 16-bitový ukazovateľ (pointer) na adresu operačnej pamäte. Je inicializovaný na adresu, ktorá sa nachádza bezprostredne za programom.

- `IP` - programové počítadlo - adresa aktuálnej inštrukcie (Instruction Pointer)

Operačná pamäť má bunky veľkosti 1 byte, ktoré sú lineárne usporiadané. Každá bunka má svoju jednoznačnú adresu (poradové číslo). Program je uložený v tejto operačnej pamäti. Zariadenie *Terminal* má za úlohu zobrazovať výstupné znaky na obrazovku a čítať vstupné znaky, keď dostane pokyn od procesora. Inštrukcie tohto procesora sú uvedené v Tab. 2.1.

Mnemonika	**Op.kód**	**Popis**	**Príkaz v C**
`inc`	1	Inkrementuj register `P` (bude ukazovať na nasledujúcu bunku pamäte)	++P;
`dec`	2	Dekrementuj register `P` (bude ukazovať na predchádzajúcu bunku pamäte)	--P;
`incv`	3	Inkrementuj hodnotu pamäte, na ktorú ukazuje register `P`	++*P;
`decv`	4	Dekrementuj hodnotu pamäte, na ktorú ukazuje register `P`	--*P;
`print`	5	Vypíš na obrazovku hodnotu bunky, na ktorú ukazuje register `P`	putchar(*P);
`load`	6	Načítaj jeden byte zo vstupu a ulož ho do pamäte na adresu, na ktorú ukazuje register `P`	*P=getchar();
`loop`	7	Ak hodnota bunky pamäte, na ktorú ukazuje register `P` **je** 0, potom miesto posunu registra `IP` na ďalšiu inštrukciu *skoč* na inštrukciu, ktorá sa nachádza hneď za inštrukciou `endl`	while(*P) {
`endl`	8	Ak hodnota bunky pamäte, na ktorú ukazuje register `P` **nie je** 0, potom miesto posunu registra `IP` na ďalšiu inštrukciu *skoč* späť na inštrukciu, ktorá sa nachádza hneď za odpovedajúcou inštrukciou `loop`	}

Tabuľka 2.1: *Inštrukcie procesora architektúry BrainDuck*

2.5 Kompilátory

Tvorba kompilátora ide ruka v ruke s tvorbou procesora, pretože aby mohol programátor vyvíjať programy pre nový procesor, musí ich vedieť skompilovať. Teória prekladačov je veľmi rozsiahla a náročná, preto sa tu nebudem tejto oblasti venovať. Čitateľov však môžem odkázať na skvelú literatúru, ako implementovať programovací jazyk [8], v ktorej čitateľa prevedie naozaj veľmi zaujímavými úskaliami. Navyše popisuje využitie generátorov prekladačov JFlex [2] a Cup [3], ktoré sú odporúčanými nástrojmi pre vytváranie kompilátorov pre platformu ***emuStudio***.

Nezáleží na tom, aký jazyk programátori zásuvných modulov navrhnú a implementujú, najčastejšie však iste pôjde o kompilátory assemblerov. Assebler je symbolický jazyk procesora, teda jazyk najnižšej úrovne, ktorý používa označenie inštrukcií miesto ich strojového tvaru. Urobiť prekladač assemblera nie je až také náročné, v porovnaní s tvorbou prekladačov iných (vyšších) jazykov. Inštrukcie sú prekladané priamo, bez použitia medzijazyka (strojovo nezávislého), a tiež bez strojovo nezávislej a strojovo závislej optimalizácie. Generovanie kódu je potom viacmenej priamočiare. Postup pri tvorbe assemblera pre platformu ***emuStudio*** sa dá rozdeliť do troch základných bodov:

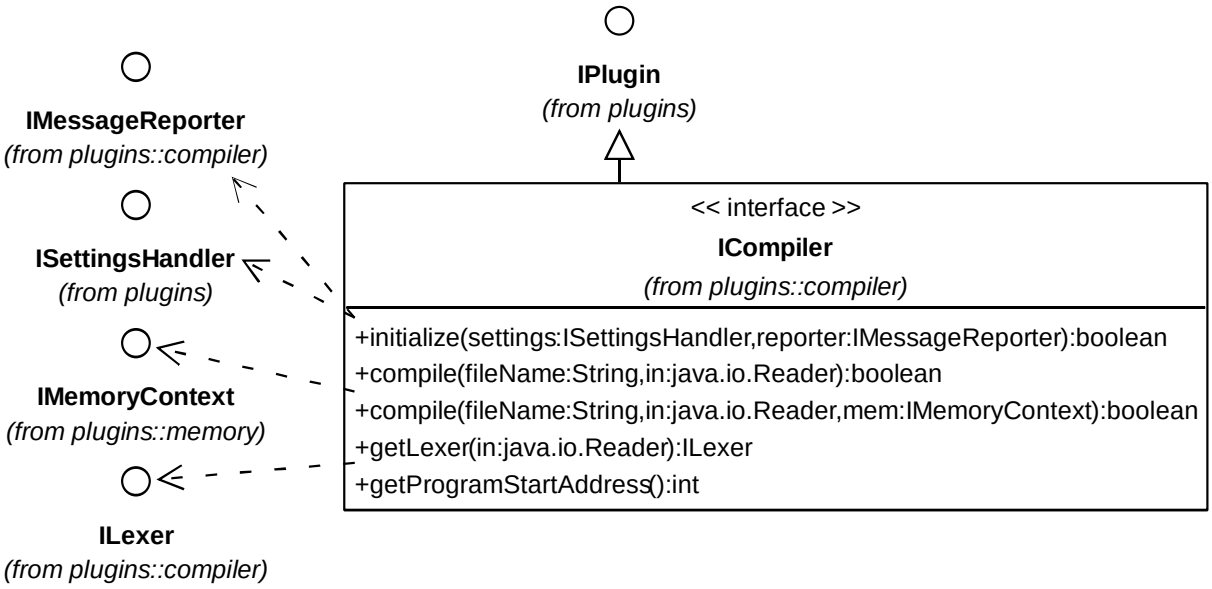
- Tvorba lexikálneho analyzátora
- Tvorba syntaktického analyzátora (parsera), ktorý vybuduje abstraktný syntaktický strom
- Tvorba generátora kódu, ktorý generuje kód podľa abstraktného syntaktického stromu

Pri tvorbe prekladačov sa používajú hlavne dva druhy nástrojov - generátory lexikálneho analyzátora, a generátory syntaktického analyzátora (parsera). Pri tvorbe mojich zásuvných modulov kompilátorov pre procesory 8080 a Z80 využívam ako generátor lexikálneho analyzátora nástroj s názvom *JFlex* [2] a generátor LALR(1) parsera nástroj s názvom *Cup* [3]. Oba tieto nástroje generujú kód napísaný v jazyku Java, čo je nevyhnutnosťou. Štýlom práce a vlastnosťami sa podobajú klasickým generátorom *flex* a *yacc*, generujúcim kód v jazyku C. Nástroj *Cup* však potrebuje pri behu mať spustenú run-time knižnicu. Táto knižnica je dodávaná s emulátorom, čiže programátori môžu tento nástroj využiť. Verzia dodávanej run-time knižnice je `11a beta 20060608`, súbor s názvom `lib\java-cup-11a-runtime.jar`.

2.5.1 Rozhrania

Sada rozhraní z knižnice `lib\emu_ifaces.jar` pre kompilátor sa nachádza v balíčku `plugins.compiler`. **Hlavné rozhranie** má názov *ICompiler* (Obr. 2.7).

Programátor nie je obmedzovaný v použití nástrojov (generátorov prekladačov), ak pravda nevyžadujú beh kompilátora s externou knižnicou nedodávanou s emulátorom (externé knižnice zásuvné moduly nemôžu používať). Ale je potrebné povedať, že rozhrania sú prispôsobené tak, aby ich bolo možné ľahko zosúladiť s možnosťami nástrojov Cup a JFlex.



Obr. 2.7: *Hlavné rozhranie kompilátora - ICompiler*

Zoznam rozhraní, ktoré musí programátor implementovať je tu:

Názov	Počet ks.	Popis
ICompiler	1	hlavné rozhranie (Obr. 2.7)
ILexer	1	lexikálny analyzátor
IToken	1	token (lexikálna jednotka)

Ako však bude ukázané neskôr, nepôjde o jediné 3 triedy, ktoré bude kompilátor obsahovať. Podrobnejší popis tried v balíku `plugins.compiler` čitateľ nájde v dokumentácii typu javadoc knižnice `emu_ifaces.jar`.

2.5.2 Výstup kompilátora

Výstupom kompilovania (vygenerovaný kód) by mal byť súbor v nejakom formáte. Odporúčaný je klasický binárny formát, alebo špeciálny formát Intel HEX [4]. Tento súborový formát slúži na vyjadrenie binárneho kódu v hexadecimálnom tvare. Ide o jeden z najstarších dostupných súborových formátov určených na tento účel, ktorý sa začal používať už od 70-tych rokov 20-teho storočia. Dôvodov, prečo odporúčam tento formát, je niekoľko:

- je jednoduchý a s kontrolou integrity dát
- uchováva v sebe začiatočnú adresu programu
- využíva ho veľa kompilátorov, hlavne starších architektúr
- používajú ho programátory (zariadenia na programovanie) "jednočipov" (napr. Intel 8051), EPROM-iek, atď., ktoré sa využívajú hlavne pre vložené (embedded) zariadenia

Kompilátor by mal tiež vedieť vygenerovaný kód zapísať do operačnej pamäte. Na tento účel dostane k dispozícii jej kontext.

2.5.3 Postup pri tvorbe kompilátora

V tejto časti uvediem príklad jednuduchého assemblera pre vymyslenú a primitívnu počítačovú architektúru BrainDuck, ktorá je uvedená v časti 2.4. Jej procesor pozná iba 8 inštrukcií. Urobiť kompilátor týchto inštrukcií nie je vôbec ťažké a nie je potrebné ani využitie generátorov prekladačov (vieme napísať lexikálny analyzátor a LL(1) parser jednoduchými metódami). My však budeme postupovať tak, ako keby sme robili veľmi zložitý a rozsiahly kompilátor assembleru - aj s generovaním kódu do výstupného súboru formátu Intel HEX.

Pri vytváraní nového projektu nezabudnite pridať ako externé referencie dve knižnice - `emu_ifaces.jar` a run-time knižnicu nástroja Cup s názvom `java-cup-11a-runtime.jar`.

2.5.3.1 Lexikálny analyzátor

Pre tvorbu lexikálneho analyzátora použijeme nástroj JFlex [2]. Tento program zo špeciálneho vstupného súboru vygeneruje triedu lexikálneho analyzátora v Jave. Popis nástroja JFlex však spadá mimo kontextu tohto dokumentu.

Náš lexikálny analyzátor nebude rozlišovať veľkosti znakov, bude akceptovať znaky UNICODE, trieda lexikálneho analyzátora sa bude volať `brainduck.impl.BDLexer` a bude implementovať rozhranie lexikálneho analyzátora `plugins.compiler.ILexer`.
Zoznam lexikálnych jednotiek jazyka je tu:

Text	**Symbol**	**Typ**
`inc`	`INC`	`IToken.RESERVED`
`dec`	`DEC`	`IToken.RESERVED`
`incv`	`INCV`	`IToken.RESERVED`
`decv`	`DECV`	`IToken.RESERVED`
`print`	`PRINT`	`IToken.RESERVED`
`load`	`LOAD`	`IToken.RESERVED`
`loop`	`LOOP`	`IToken.RESERVED`
`endl`	`ENDL`	`IToken.RESERVED`
`;`	`TCOMMENT`	`IToken.COMMENT`
koniec riadku	`EOL`	`IToken.SEPARATOR`
koniec súboru	`EOF`	`IToken.TEOF`
chyba	`error`	`IToken.ERROR`

Tabuľka 2.2: *Lexikálne jednotky jazyka BrainDuck*

Názov symbolu pre token môže byť zvolený ľubovoľne[2]. Tieto názvy použijeme v generátore parsera Cup v gramatike, ktorý ich prevedie na celočíselné konštanty. O tom však budem

[2]teda skoro ľubovoľne - názov symbola nesmie byť ani jeden z konštánt v rozhraní *plugins.compiler.IToken*

hovoriť v časti "Parser". Typ tokenu však už musí byť zvolený ako konštanta z rozhrania `plugins.compiler.IToken`, kde sú zadefinované tieto typy:

- `RESERVED` (rezervované slovo - inštrukcie, príkazy)
- `PREPROCESSOR` (rezervované slovo preprocesora)
- `REGISTER` (register procesora)
- `SEPARATOR` (oddeľovač - medzera, tabulátor, atď.)
- `OPERATOR` (operátor - +,-,*,/, atď.)
- `COMMENT` (komentár)
- `LITERAL` (literál - číslo, reťazec, znak, atď.)
- `IDENTIFIER` (identifikátor - meno premennej, makra, atď.)
- `LABEL` (návestie)
- `ERROR` (chyba - neznámy token)
- `TEOF` (token reprezentuje koniec súboru)

Určenie typu tokenu má len jeden účel - podľa nich hlavný modul vie, ako má zvýrazniť tento token v textovom editore. Teda pre každý kompilátor s rôznymi tokenmi sa tokeny rovnakého typu zafarbia, resp. zvýraznia v textovom editore rovnako.

Nasleduje vytvorenie triedy, ktorá bude reprezentovať token. Bude obsahovať metódy a atribúty príznačné všetkým tokenom (ako jeho pozíciu, hodnotu, atď.). Samozrejme musí implementovať rozhranie `plugins.compiler.IToken`. Umiestnime ju do balíčka `brainduck.impl` a bude mať názov `tokenBD`.

```
package brainduck.impl;

import java_cup.runtime.Symbol;
import plugins.compiler.IToken;

public class tokenBD extends Symbol implements IToken,sym8080 {
    public final static int ERROR_UNKNOWN_TOKEN = 0xA05;

    private String text; // hodnota tokenu
    private int row;     // číslo riadka
    private int col;     // číslo stĺpca
    private int offset;  // pozícia tokenu
    private int length;  // dĺžka tokenu
    private int type;    // typ tokenu
```

```
    public tokenBD(int ID, int type, String text,
            int line, int column, int offset) {
        super(ID);
        this.type = type;
        this.text = text;
        this.row = line;
        this.col = column;
        this.offset = offset;
        this.length = (text==null)?0:text.length();
    }

    public int getID() { return super.sym; }
    public int getType() { return type; }

    public String getText() { return text; }
    public String getErrorString() {
        switch (super.sym) {
            case ERROR_UNKNOWN_TOKEN: return "Unknown token";
        }
        return "";
    }
    public int getLine() { return row; }
    public int getColumn() { return col; }
    public int getOffset() { return offset; }
    public int getLength() { return length; }
}
```

Môžte si všimnúť, že okrem rozhrania `IToken` trieda implementuje ďalšie rozhranie s názvom `symBD`. Toto rozhranie zatiaľ neexistuje, ale vytvorí ho nástroj Cup automaticky. Bude obsahovať celočíselné konštanty všetkých terminálnych symbolov gramatiky (čiže lexikálnych jednotiek) tak, ako sme si ich zvolili v tabuľke 2.2. Tým, že táto trieda implementuje toto rozhranie, preberá jeho konštanty.

Teraz uvediem vstupný súbor nášho lexikálneho analyzátora `lex.jflex`:

```
package brainduck.impl;

import plugins.compiler.ILexer;
import plugins.compiler.IToken;
import java.io.Reader;
import java.io.IOException;

%%
```

```
%class BDLexer
%cup
%public
%implements ILexer
%line
%column
%char
%caseless
%unicode
%type tokenBD

%{
    @Override
    public tokenBD getSymbol() throws IOException {
        return next_token();
    }

    @Override
    public void reset(Reader in, int yyline, int yychar,
            int yycolumn) {
        yyreset(in);
        this.yyline = yyline;
        this.yychar = yychar;
        this.yycolumn = yycolumn;
    }

    @Override
    public void reset() {
        yyline = yychar = yycolumn = 0;
    }

    private tokenBD token(int id, int type) {
        return new tokenBD(id,type,yytext(),yyline,yycolumn,yychar);
    }

%}

/* Koniec súboru */
%eofval{
  return token(tokenBD.EOF, IToken.TEOF);
%eofval}

/* Pomocné regulárne výrazy */
Comment    = ";"[^\r\n]*
```

```
Eol        = \n|\r|\r\n
WhiteSpace = ([ ]|[\t]|[\f])

%%
/* Inštrukcie */
"inc"   { return token(tokenBD.INC,  IToken.RESERVED); }
"dec"   { return token(tokenBD.DEC,  IToken.RESERVED); }
"incv"  { return token(tokenBD.INCV, IToken.RESERVED); }
"decv"  { return token(tokenBD.DECV, IToken.RESERVED); }
"print" { return token(tokenBD.PRINT,IToken.RESERVED); }
"load"  { return token(tokenBD.LOAD, IToken.RESERVED); }
"loop"  { return token(tokenBD.LOOP, IToken.RESERVED); }
"endl"  { return token(tokenBD.ENDL, IToken.RESERVED); }

{Eol}   { return token(tokenBD.EOL, IToken.SEPARATOR); }

/* Komentár potrebujeme poznať kvôli zvýrazňovaniu syntaxe */
{Comment}     { return token(tokenBD.TCOMMENT, IToken.COMMENT); }
{WhiteSpace}+ { /* ignorujeme medzery  */ }

/* Ostatné znaky a slová sú chybové */
.+            { return token(tokenBD.error, tokenBD.ERROR); }
```

Výsledný súbor `BDLexer.java` vygenerujete spustením príkazu

`jflex-1.4.3/bin/jflex lex.jflex` (unix,linux), resp.
`jflex-1.4.3\bin\jflex.bat lex.jflex` (windows).

2.5.3.2 Gramatika

Nasleduje tvorba gramatiky nášho jazyka pre parser. Nástroj Cup generuje parser typu LALR(1), teda môžeme použiť LR(k) alebo LALR(k) bezkontextovú gramatiku. V týchto typoch gramatík je použitie ľavej rekurzie výhodné, pravej nie veľmi dobré - na rozdiel od LL(k) gramatík, kde je to naopak.

Prečo je to tak, sa dá vysvetliť veľmi jednoducho. Parsery typu LR(k), resp. LALR(k) pracujú zdola nahor, teda najprv sa do zásobníka posúvajú symboly zo vstupu až dovtedy, pokiaľ ich parser identifikuje ako pravú stranu nejakého pravidla gramatiky. Potom ich redukuje na symbol na ľavej strane tohto pravidla. Zásobník pri priberaní symbolov zo vstupu rastie smerom zľava doprava. Pri pravej rekurzii, ako je napr. pri tomto pravidle:

```
L -> id , L
   | id
```

pri parsovaní rastie zásobník dovtedy, až kým sa nedosiahne posledné `id`, po ktorom nasledujú redukcie:

Poradie	Zásobník	Zostávajúci vstup	Akcia
1.	`$`	id,id,id	SHIFT
2.	`$id`	,id,id	SHIFT
3.	`$id,`	id,id	SHIFT
4.	`$id,id`	,id	SHIFT
5.	`$id,id,`	id	SHIFT
6.	`$id,id,id`		REDUCE
7.	`$id,id,L`		REDUCE
8.	`$id,L`		REDUCE
9.	`$L`		ACCEPT

Pri veľkom množstve týchto identifikátorov sa pamäť zásobníka zapĺňa veľmi rýchlo. Použitie pravej rekurzie môže okrem toho spôsobiť kolízie typu SHIFT/REDUCE, ak by množina FOLLOW nášho pravidla `L` obsahovala čiarku (`,`). Ľavá rekurzia, naopak, šetrí pamäť. Pozmeňme trochu naše pravidlo na ľavú rekurziu:

```
L -> L , id
   | id
```

Majme ten istý vstup. Parsovanie bude prebiehať takto:

Poradie	Zásobník	Zostávajúci vstup	Akcia
1.	`$`	id,id,id	SHIFT
2.	`$id`	,id,id	REDUCE
3.	`$L`	,id,id	SHIFT
4.	`$L,`	id,id	SHIFT
5.	`$L,id`	,id	REDUCE
6.	`$L`	,id	SHIFT
7.	`$L,`	id	SHIFT
8.	`$L,id`		REDUCE
9.	`$L`		ACCEPT

Ako vidíme, zásobník sa "drží" v čo možno najmenšej veľkosti.
A teraz poďme ku gramatike nášho jazyka BrainDuck:

```
Program   -> Row
           | Program Row EOL

Row       -> Statement Comment
           | Comment

Comment   -> TCOMMENT
           | epsilon

Statement -> INC   | DEC  | INCV | DECV
           | PRINT | LOAD | LOOP | ENDL
```

Štartovací symbol gramatiky je `Program`. Ako je možné vidieť, v tomto pravidle na vyjadrenie programu (ako zoznamu riadkov ukončených znakom `EOL`) som použil som ľavú rekurziu. Je dôležité poznamenať, že riadky musia byť *oddelené* novým riadkom, nie ním *ukončené*. To znamená, že token prázdneho riadku je separátorom, nie terminátorom. Riadok (pravidlo `Row`) je buď prázdny ("epsilon" predstavuje prázdny symbol), alebo obsahuje len komentár, alebo sa skladá z príkazu a s alebo bez komentára (pravidlo `Comment` vyberá buď terminál komentára (`TCOMMENT`), alebo nič). Príkazov (pravidlo `Statement`) máme 8. Ide o veľmi jednoduchú gramatiku, ktorú teraz prepíšem do kódu pre nástroj Cup.

2.5.3.3 Parser

Vstupný súbor pre nástroj Cup som pomenoval `parser.cup`. Trieda parsera, ktorá bude výstupom nástroja Cup sa bude volať `brainduck.impl.BDParser`.

```
import java_cup.runtime.Symbol;
import plugins.compiler.IMessageReporter;
import plugins.compiler.IToken;
//import brainduck.tree.*;

init with {: errorCount = 0; :}
parser code {:
    private IMessageReporter reporter = null;
    public IToken lastToken;
    public int errorCount = 0;

    public BDParser(java_cup.runtime.Scanner s,
            IMessageReporter reporter) {
        this(s);
        this.reporter = reporter;
    }

    public void syntax_error(Symbol current) {
        errorCount++;
        report_error("Syntax error: ",current);
    }

    public void unrecovered_syntax_error(Symbol current) {
        errorCount++;
        report_error("Fatal syntax error: ", current);
        done_parsing();
    }

    public void report_error(String message, Symbol current) {
        String mes;
```

```
        IToken t = (IToken)current;
        mes = message + t.getErrorString()
            + " ('"+t.getText()+"')";

        reporter.report(t.getLine()+1, t.getColumn(),mes,
            IMessageReporter.TYPE_ERROR);
    }
:}

terminal INC,DEC,INCV,DECV,PRINT,LOAD,LOOP,ENDL;
terminal EOL;
terminal TCOMMENT;

non terminal Program;
non terminal Row;
non terminal Statement;
non terminal Comment;

start with Program;

Program ::= Row
         |  Program EOL Row;

Row ::= Statement Comment | Comment ;

Comment ::= TCOMMENT | ;

Statement ::= INC | DEC | INCV | DECV | PRINT | LOAD | LOOP | ENDL;
```

Kód parsera ešte nie je úplný. Je ešte potrebné vytvoriť akcie - elementy kódu v jazyku Java, ktoré sa vykonajú po úspešnom parsovaní niektorého pravidla. Tieto elementy kódu sa zapisujú medzi symboly `{:` a `:}` hneď za pravidlo, ako to uvidíme neskôr. Pri vzniku syntaktickej chyby parser automaticky (zabezpečené nástrojom Cup) zavolá funkciu `sytax_error()`, resp. `unrecovered_syntax_error()`.

Import všetkých tried z balíčka `brainduck.tree` zatiaľ necháme zakomentovaný, v tomto balíčku sa budú nachádzať uzly abstraktného stromu. Neskôr, po dokončení parsera nezabudnite tento riadok odkomentovať.

2.5.3.4 Uzly abstraktného stromu

Elementy kódu pri príslušných pravidlách budú postupne vytvárať abstraktný syntaktický strom programu. Dalo by sa povedať, že (skoro) každá ľavá strana ľubovoľného pravidla gramatiky predstavuje uzol syntaktického stromu. No každý program napísaný v našom jazyku

má iný abstraktný syntaktický strom, teda nie každý uzol sa v ňom nachádza.

Každý uzol má svoje atribúty a operácie, ktoré sa volajú v ďalších fázach kompilácie. Uzol v jazyku Java je reprezentovaný jednou *triedou*. Element kódu sa vykoná po parsovaní pravej strany pravidla. Tento kód by mal v konečnom dôsledku vytvoriť a vrátiť inštanciu triedy - uzla - z ľavej strany tohto pravidla.

Avšak sa bežne stáva, že jedno pravidlo má viac pravých strán, ktoré sú oddelené znakom "|". Je to tak aj v našej gramatike. To však zatiaľ nič nemení. Ale niekedy môže jedna pravá strana požadovať trochu inú "starostlivosť", alebo má prirodzene iné vlastnosti ako iná pravá strana - a potom je ťažké alebo možno neprehľadné vytvoriť jedinú triedu ľavej strany, ktorá by "uspokojovala" potreby všetkých pravých strán.

Napríklad, majme pravidlo pre inštrukcie `Instruction ->`, ktorého pravé strany popisujú inštrukcie s operandom, ale aj bez operanda. Vtedy inštrukcie bez operanda by mohli chcieť vytvoriť inštanciu triedy, ktorá reprezentuje inštrukcie bez operanda. Ostatné inštrukcie s operandom zas naopak môžu chcieť vytvoriť inštanciu triedy reprezntujúcej inštrukcie s operandom. Ako uvidíme neskôr, sú to dosť opodstatnené požiadavky, pretože uzol - trieda v ďalších fázach kompilácie inak pristupuje ku prekladu takých inštrukcií, inak k prekladu onakých.

Riešením v takomto prípade bude vytvorenie 3 tried - ľavá strana bude reprezentovaná abstraktnou triedou inštrukcie, a pravé strany budú reprezentované dvoma triedami, implementujúcimi abstraktnú triedu na ľavej strane pravidla - každá riešiaca svoj druh inštrukcií.

V našom prípade nemáme také pravé strany žiadneho pravidla, ktoré by vyžadovali taký prístup. Máme 4 ľavé strany pravidiel - `Program`, `Row`, `Comment` a `Statement`. Z toho komentár nebudeme do nášho stromu zahŕňať, pretože pre preklad nie je nijak dôležitý. Ostatné uzly však áno. Triedy uzlov stromu budem ukladať do balíčka `brainduck.tree`.

Prvou triedou je trieda `brainduck.tree.Program`. Táto trieda bude zatiaľ len uchovávať všetky riadky programu.

```
package brainduck.tree;

import java.util.Vector;

public class Program {
    private Vector<Row> list; // zoznam všetkých inštrukcií

    public Program() {
        list = new Vector<Row>();
    }

    public void addRow(Row node) {
        list.addElement(node);
    }

}
```

Ďalšou triedou je trieda `brainduck.tree.Row`. Táto trieda drží riadok programu. Ak môže mať riadok návestia, tak návestie bude uchované v tejto triede. V našom prekladači návestia nie sú, čiže riadok bude uchovávať len inštrukcie programu. Teda dalo by sa vynechať túto triedu a vytvárať rovno triedu s inštrukciami. Ale nechávam ju naschvál, aby bolo vidieť prechod medzi riadkom a inštrukciou.

```
package brainduck.tree;

public class Row {
    private Statement stat;

    public Row(Statement stat) {
        this.stat = stat;
    }

}
```

Poslednou triedou je trieda `brainduck.tree.Statement`. Táto trieda drží už konkrétnu inštrukciu programu. Sú v nej verejné statické konštanty, ktoré reprezentujú danú inštrukciu - a parser ich použije pri vytváraní inštancie tejto triedy. Je možné všimnúť si, že hodnoty týchto konštánt už reprezentujú operačné kódy jednotlivých inštrukcií (porov. tabuľku 2.1). Je dobré konštanty inštrukcií označiť operačnými kódmi (pokiaľ to je možné), čím sa výrazne zefektívni kompilovanie.

```
package brainduck.tree;

public class Statement {
    public final static int INC   = 1;
    public final static int DEC   = 2;
    public final static int INCV  = 3;
    public final static int DECV  = 4;
    public final static int PRINT = 5;
    public final static int LOAD  = 6;
    public final static int LOOP  = 7;
    public final static int ENDL  = 8;

    private int instr;

    public Statement(int instr) {
        this.instr = instr;
    }
}
```

Teraz, keď máme napísané prvé verzie tried uzlov abstraktného stromu, je potrebné doplniť jeho vytvorenie do parsera. Takže si otvoríme súbor `parser.cup` a doplníme definíciu neterminálnych symbolov a pravidiel takto:

```
...
non terminal Program Program;
non terminal Row Row;
non terminal Statement Statement;
non terminal Comment;

start with Program;

Program ::= Row:row
            {:
                Program program = new Program();
                if (row != null) program.addRow(row);
                RESULT = program;
            :}
          | Program:program EOL Row:row
            {:
                if (row != null) program.addRow(row);
                RESULT = program;
            :};

Row ::= Statement:stmt Comment
        {: RESULT = new Row(stmt); :}
      | Comment
        {: RESULT = null; :};

Comment ::= TCOMMENT | ;

Statement ::= INC   {: RESULT = new Statement(Statement.INC); :}
            | DEC   {: RESULT = new Statement(Statement.DEC); :}
            | INCV  {: RESULT = new Statement(Statement.INCV); :}
            | DECV  {: RESULT = new Statement(Statement.DECV); :}
            | PRINT {: RESULT = new Statement(Statement.PRINT); :}
            | LOAD  {: RESULT = new Statement(Statement.LOAD); :}
            | LOOP  {: RESULT = new Statement(Statement.LOOP); :}
            | ENDL  {: RESULT = new Statement(Statement.ENDL); :}
            ;
```

Parser je týmto úplne dokončený. Triedy parsera a symbolov vygenerujeme pomocou nástroja Cup takto (jeden riadok som rozdelil na dva, aby sa tu zmestil):

```
java -jar java-cup/java-cup-11a.jar -package "brainduck.impl"
     -parser "BDParser" -symbols "symBD" -interface "parser.cup"
```

2.5.3.5 Generátor kódu

Teraz je potrebné vytvoriť ďalšiu fázu kompilovania - generovanie kódu. Je potrebné doplniť uzly abstraktného stromu o ďalšie metódy, ktoré preložia svoju pravú stranu príslušného pravidla do strojového tvaru.

Ak sa v programe nachádzajú návestia a dopredné referencie, generovanie kódu sa rozdelí na minimálne dve fázy. V prvej z týchto fáz sa budú vyhodnocovať adresy jednotlivých inštrukcií (relatívne), ich veľkosť a tiež adresy všetkých návestí (relatívne). Zoznam všetkých návestí, premenných, funkcií, atď. sa musí zozbierať v tejto fáze a uložiť do špeciálnej tabuľky, resp. triedy, ktorá sa všeobecne označuje prostredie kompilovania[3]. Dôvod použitia takéhoto prostredia je ten, že ak sa nejaký príkaz odvoláva na nejaké návestie, premennú, atď., máme ich hneď po ruke bez nutnosti prehľadávania celého abstraktného syntaktického stromu. V našom prekladači však nemusíme toto prostredie použiť, pretože nemáme možnosť vytvárať premenné, ani návestia, makrá, funkcie, atď.

Táto fáza sa bude opakovať dovtedy, kým nebudú vyhodnotené všetky adresy (už ako absolútne) a veľkosti inštrukcií.

V druhej fáze sa podľa vyhodnotených údajov kód preloží a vygeneruje pomocou nejakého generátora. Pre tento účel som vytvoril generátor vytvárajúci súbor vo formáte Intel HEX.
Takže upresnime naše fázy:

- v prvej fáze každý uzol vráti svoju absolútnu adresu
- v druhej fáze každý uzol vygeneruje kód pomocou generátora

Predtým, ako doplníme kód jednotlivých fáz do uzlov, uvediem generátor kódu pre súbor Intel HEX. Ide o jednu triedu, ktorá sa nemusí meniť v rámci rôznych zásuvných modulov. Okrem generovania kódu do súboru poskytuje aj generovanie do operačnej pamäte. Triedu som nazval `HEXFileHandler` a umiestnil som ju do balíčka `brainduck.impl`.

```
package brainduck.impl;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.util.Collections;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;
import plugins.memory.IMemoryContext;
import runtime.StaticDialogs;

public class HEXFileHandler {
    private Hashtable<Integer, String> program;
    private int nextAddress;
```

[3] v prípade interpretera (nie prekladača) by táto tabuľka mala názov prostredie behu (runtime environment)

```
public HEXFileHandler() {
    this.program = new Hashtable<Integer, String>();
    nextAddress = 0;
}

/**
 * Put code on next address
 * if element exist on the address, then is rewritten
 * @code Code in hex format. Every hex item must be
 *       of even size.
 */
public void putCode(String code) {
    program.put(nextAddress,code);
    nextAddress += (code.length()/2);
}

private String getCode(int address) {
    return (String)program.get(address);
}

public void setNextAddress(int address) {
    nextAddress = address;
}

private String checksum(String lin) {
    int sum = 0, chsum = 0;
    for (int i =0; i < lin.length()-1; i += 2)
        sum += Integer.parseInt(lin.substring(i,i+2),16);
    sum %= 0x100;
    chsum = 0x100 - sum;
    return String.format("%1$02X",chsum);
}

/**
 * Keys of the hashtable have to represent adresses
 * and values have to represent compiled code.
 * Method copies all elements from param hashtable
 * to internal data member.
 */
public void addTable(Hashtable<Integer,String> ha) {
    Vector<Integer> adrs = new Vector<Integer>(ha.keySet());
    int largestAdr = nextAddress;
    for (Enumeration<Integer> e = adrs.elements();
```

```
            e.hasMoreElements();) {
        nextAddress = (Integer)e.nextElement();
        String cd = (String)ha.get(nextAddress);
        program.put(nextAddress,cd);
        nextAddress += (cd.length()/2);
        if (nextAddress > largestAdr)
            largestAdr = nextAddress;
    }
    nextAddress = largestAdr;
}

public Hashtable<Integer, String> getTable() {
    return this.program;
}

// generate hex file
private String generateHEX() {
    String lines = "";       // all lines
    String lineAddress = ""; // starting line address
    String line = "";        // line data
    int address = 0;         // current address in hex file
    int bytesCount = 0;      // current count of data bytes
                             // on single line

    Vector<Integer> adrs = new Vector<Integer>(program.keySet());
    Collections.sort(adrs);

    // for all code elements (they won't be separated)
    for (Enumeration<Integer> e = adrs.elements();
            e.hasMoreElements();) {
        int adr = (Integer)e.nextElement();

        // is line at very beginning ?
        if (lineAddress.equals("")) {
            address = adr;
            lineAddress = String.format("%1$04X",address);
        }

        // if element's address do not equal suggested
        // (natural computed) address or line is full
        if ((address != adr) || (bytesCount >= 16)) {
            String lin = String.format("%1$02X", bytesCount)
                    + lineAddress + "00" + line;
            lines += ":"+ lin + checksum(lin) + "\n";
```

```
                bytesCount = 0;
                line = "";
                address = adr;
                lineAddress = String.format("%1$04X",address);
            }

            // code have to be stored as number of separate
            // pairs of hex digits
            String cd = (String)program.get(adr);

            // cd hasn't to be longer than 16-bytesCount
            while ((cd.length()+line.length()) > 32) {
                int len = 32 - line.length();
                line += cd.substring(0,len);
                cd = cd.substring(len,cd.length());

                address += (len / 2); // compute next addr
                bytesCount += (len / 2);

                // save line
                String lin = String.format("%1$02X",bytesCount)
                        + lineAddress + "00" + line;
                lines += ":"+ lin + checksum(lin) + "\n";
                bytesCount = 0;
                line = "";
                lineAddress = String.format("%1$04X",address);
            }
            if (cd.length() > 0) {
                line += cd;
                address += (cd.length() / 2); // compute next addr
                bytesCount += (cd.length() / 2);
            }
        }
        if (line.equals("") == false) {
            String lin = String.format("%1$02X",bytesCount)
                    + lineAddress + "00" + line;
            lines += ":"+ lin + checksum(lin) + "\n";
        }
        lines += ":00000001FF\n";
        return lines;
    }
```

```
    /**
     * Method is similar to generateHex() method in that way,
     * that compiled program is also transformed into chunk
     * of bytes, but not to hex file but to the operating
     * memory.
     * @param mem context of operating memory
     */
    public boolean loadIntoMemory(IMemoryContext mem) {
        if (mem.getDataType() != Short.class) {
            StaticDialogs.showErrorMessage("Incompatible operating"
                    + " memory type!\n\nThis compiler can't load"
                    + " file into this memory.");
            return false;
        }
        Vector<Integer> adrs = new Vector<Integer>(program.keySet());
        Collections.sort(adrs);
        for (Enumeration<Integer> e = adrs.elements();
                e.hasMoreElements();) {
            int adr = (Integer)e.nextElement();
            String code = this.getCode(adr);
            for (int i = 0, j = 0; i < code.length()-1; i+=2, j++) {
                String hexCode = code.substring(i, i+2);
                short num = (short)((Short.decode("0x" + hexCode))
                    & 0xFF);
                mem.write(adr+j, num);
            }
        }
        return true;
    }
    public void generateFile(String filename) throws java.io.IOException{
        String fileData = generateHEX();

        BufferedWriter out = new BufferedWriter(new FileWriter(filename));
        out.write(fileData);
        out.close();
    }
    public int getProgramStart() {
        Vector<Integer> adrs = new Vector<Integer>(program.keySet());
        Collections.sort(adrs);
        if (adrs.isEmpty() == false)
            return (Integer)adrs.firstElement();
        else return 0;
    }
}
```

Preložený kód programu postupne pridávame do objektu triedy `HEXFileHandler` metódami:

- `putCode(String code)`, kde parameter `code` musí byť preložený súvislý reťazec kódu v hexadecimálnom tvare párnej dĺžky, napr. inštrukciu `INC` preložíme do kódu "`01`". Po pridaní kódu do generátora tento automaticky inkrementuje nasledujúcu absolútnu adresu v kóde
- `addTable(Hashtable<Integer,String> ha)`, ktorá do aktuálneho programu vloží podprogram. Tento podprogram je reprezentovaný tabuľkou typu `java.util.Hashtable`, kde kľúče reprezentujú absolútne adresy a hodnoty preložené súvislé reťazce kódu v hexadecimálnom tvare párnej dĺžky.

Môžme tiež nastaviť nasledujúcu absolútnu adresu kódu pomocou metódy `setNextAddress(int address)`.

Výsledný súbor sa vygeneruje volaním metódy `generateFile(String filename)`, a načíta sa do pamäte volaním metódy `loadIntoMemory(IMemoryContext mem)`. Začiatočnú adresu programu môžme zistiť volaním metódy `getProgramStart()`.

Teraz je potrebné doplniť kód uzlov, aby vedel prekladať kód. Čiže doplníme prvú a druhú fázu generovania kódu.
Do triedy `brainduck.impl.Program` doplníme kód:

```
// v prvej fáze zistíme absulútnu adresu poslednej inštrukcie
public int pass1(int addr_start) throws Exception {
    int curr_addr = addr_start;

    // prejdeme celým programom a takto postupne
    // zistíme absulútnu adresu poslednej inštrukcie
    for (int i = 0; i < list.size(); i++)
        curr_addr = list.get(i).pass1(curr_addr);
    return curr_addr;
}

// druhou fázou je samotné generovanie kódu - pridávanie
// preloženého kódu do generátora
public void pass2(HEXFileHandler hex) throws Exception {
    for (int i = 0; i < list.size(); i++)
        list.get(i).pass2(hex);
}
```

Do triedy `brainduck.impl.Row` doplníme kód:

```
public int pass1(int addr_start) throws Exception {
    if (stat != null)
```

```
            addr_start = stat.pass1(addr_start);
        return addr_start;
    }

    public void pass2(HEXFileHandler hex) throws Exception {
        if (stat != null)
            stat.pass2(hex);
    }
```

A konečne do triedy `brainduck.impl.Statement` doplníme kód:

```
    // prvá fáza vracia nasledujúcu absulútnu adresu
    // inštrukcie od adresy addr_start
    public int pass1(int addr_start) throws Exception {
        // každá inštrukcia zaberá len 1 byte
        return addr_start +1;
    }

    public void pass2(HEXFileHandler hex) {
        hex.putCode(String.format(”%1$02X”,instr));
    }
```

2.5.3.6 Ostatné

Teraz nám ostáva už len implementovať hlavné rozhranie zásuvného modulu kompilátora - *ICompiler*.

```
package brainduck.impl;

import brainduck.impl.HEXFileHandler;
import brainduck.impl.BDLexer;
import brainduck.impl.BDParser;

import java.io.Reader;

import plugins.ISettingsHandler;
import plugins.compiler.ICompiler;
import plugins.compiler.ILexer;
import plugins.compiler.IMessageReporter;
import plugins.memory.IMemoryContext;
import brainduck.tree.Program;

public class BrainDuck implements ICompiler {
    private long hash;
```

```
private BDLexer lex = null;
private BDParser par;
private IMessageReporter reporter;
@SuppressWarnings("unused")
private ISettingsHandler settings;

// aktualizuje sa až po kompilácii
private int programStart = 0;

public BrainDuck(Long hash) {
    this.hash = hash;
    // POZOR! lex sa musí resetovať
    // pred kompiláciou s objektom typu
    // java.io.Reader
    lex = new BDLexer((Reader)null);
}

private void print_text(String mes, int type) {
    if (reporter != null) reporter.report(mes, type);
    else System.out.println(mes);
}

@Override
public String getTitle() { return "BrainDuck Assembler"; }
@Override
public String getVersion() { return "1.0b1"; }
@Override
public String getCopyright() {
    return "\u00A9 Copyright 2009, P. Jakubčo";
}
@Override
public String getDescription() {
    return "Assembler for esoteric language"
        + " BrainDuck derived from brainfuck";
}

@Override
public long getHash() { return hash; }

@Override
public boolean initialize(ISettingsHandler sHandler,
        IMessageReporter reporter) {
    this.settings = sHandler;
    this.reporter = reporter;
```

```
        par = new BDParser(lex, reporter);
        return true;
    }

    @Override
    public void destroy() {}

    @Override
    public void reset() {}

    @Override
    public int getProgramStartAddress() {
        return programStart;
    }

    /**
     * Skompiluje zdrojový kód do generátora HEXFileHadler
     *
     * @param in  Objekt typu java.io.Reader - zdrojový kód
     * @return  Objekt HEXFileHandler
     */
    private HEXFileHandler compile(Reader in) throws Exception {
        if (par == null) return null;
        if (in == null) return null;

        Object s = null;
        HEXFileHandler hex = new HEXFileHandler();

        print_text(getTitle()+", version "+getVersion(),
            IMessageReporter.TYPE_INFO);
        lex.reset(in,0,0,0);
        s = par.parse().value;

        if (s == null) {
            print_text("Unexpected end of file",
                IMessageReporter.TYPE_ERROR);
            return null;
        }
        if (par.errorCount != 0)
            return null;

        // tu sa vykonajú obe fázy kompilovania
        Program program = (Program)s;
```

```
        program.pass1(0);
        program.pass2(hex);
        return hex;
    }

    @Override
    public boolean compile(String fileName, Reader in) {
        try {
            HEXFileHandler hex = compile(in);
            if (hex == null) return false;
            hex.generateFile(fileName);
            print_text("Compile was sucessfull. Output: "
                    + fileName, IMessageReporter.TYPE_INFO);
            programStart = hex.getProgramStart();
            return true;
        } catch (Exception e) {
            print_text(e.getMessage(),
                IMessageReporter.TYPE_ERROR);
            return false;
        }
    }
    @Override
    public boolean compile(String fileName, Reader in,
            IMemoryContext mem) {
        try {
            HEXFileHandler hex = compile(in);
            hex.generateFile(fileName);
            print_text("Compile was sucessfull. Output: "
                    + fileName, IMessageReporter.TYPE_INFO);
            programStart = hex.getProgramStart();
            boolean r = hex.loadIntoMemory(mem);
            if (r)
                print_text("Compiled file was loaded into"
                    + " operating memory.",
                    IMessageReporter.TYPE_INFO);
            else
                print_text("Compiled file couldn't be loaded"
                    + " into operating memory due to an error.",
                    IMessageReporter.TYPE_ERROR);
            return true;
        } catch (Exception e) {
            print_text(e.getMessage(), IMessageReporter.TYPE_ERROR);
            return false;
        }
```

```
    }
    @Override
    public ILexer getLexer(Reader in) {
        return new BDLexer(in);
    }

    @Override
    public void showSettings() {
        // TODO: Kompilátor môže používať vlastné
        //       GUI, ktoré aktivuje hlavný modul
        //       volaním tejto metódy
    }
}
```

Týmto sme tvorbu kompilátora ukončili. Výsledný kompilátor treba zabalit' do súboru JAR a umiestnit' medzi zásuvné moduly kompilátorov.

2.6 Operačné pamäte

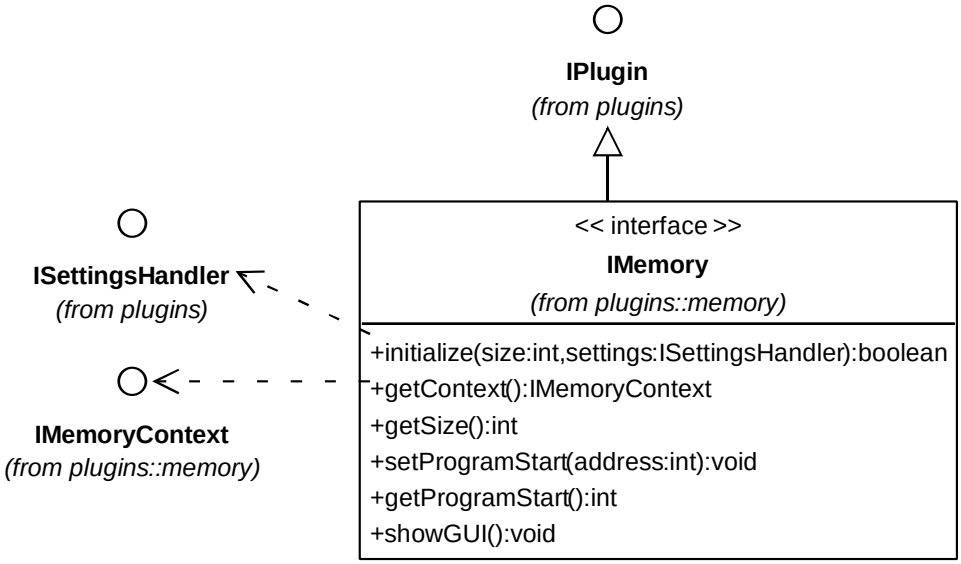
Predtým, ako sa pustíme do vývoja procesora, potrebujeme spravit' operačnú pamäť. Je to z dôvodu, aby sme vedeli, ako bude vyzerat' jej kontext, aby sme ho potom mohli v procesore využit'. Tvorba operačnej pamäte nie je nijak zložitá, len je potrebné okrem implementácie samotnej pamäte implementovat' aj jej GUI - grafické rozhranie. Môj zásuvný modul operačnej pamäte `Standard.jar` má dost' prepracované GUI a väčšinou aj všeobecne kód GUI je rozsiahly. Preto som sa rozhodol, že tvorbu GUI ponechám už na čitateľa a tu uvediem len základnú implementáciu operačnej pamäte pre architektúru BrainDuck. Svojou štruktúrou a vlastnosťami bude vlastne rovnaká, ako je aj môj spomenutý zásuvný modul, čiže bude možné spustit' emulátor s architektúrou BrainDuck s mojim zásuvným modulom operačnej pamäte.

Tvorba operačnej pamäte zahŕňa kroky:

- Tvorbu kontextu pamäte
- Tvorbu implementácie hlavného rozhrania
- Tvorbu GUI

2.6.1 Rozhrania

Sada rozhraní z knižnice `lib\emu_ifaces.jar` pre operačnú pamäť sa nachádza v balíčku `plugins.memory`. **Hlavné rozhranie** má názov *IMemory* (Obr. 2.8).



Obr. 2.8: *Hlavné rozhranie operačnej pamäte - IMemory*

Zoznam rozhraní, ktoré musí programátor implementovať je tu:

Názov	Počet ks.	Popis
IMemory	1	hlavné rozhranie (Obr. 2.8)
IMemoryContext	1	kontext operačnej pamäte

Metóda `initialize()` je volaná pri inicializačnom procese, viď Obr. 1.3, teda ako celkom prvá a je zaručené, že bude vždy volaná. Jej parametre sú veľkosť pamäte (jednotky nie sú podstatné, zásuvný modul si môže túto veľkosť interpretovať po svojom) a objekt typu `ISettingsHandler`, implementovaný v hlavnom module, pomocou ktorého môže zásuvný modul pristupovať k svojim nastaveniam, uloženým v konfiguračnom súbore. Metóda `initialize()` by mala okrem iného (napr. načítavania nastavení) vytvoriť objekt kontextu operačnej pamäte.

Podrobnejší popis metód týchto rozhraní čitateľ nájde v dokumentácii typu javadoc knižnice `emu_ifaces.jar`.

2.6.2 Postup pri tvorbe OP

V tejto časti uvediem príklad jednuduchej operačnej pamäte pre vymyslenú počítačovú architektúru BrainDuck, ktorá je uvedená v časti 2.4. Samotná operačná pamäť bude implementovaná ako **pole** pevnej veľkosti hodnôt typu `short`. Ide o 16-bitový znamienkový typ, ktorý poskytuje rozsah od -32768 do 32767.

POZNÁMKA:

My nepotrebujeme znamienka, ani taký veľký rozsah. Teoreticky by nám mohol stačiť aj typ `byte` *(8-bitový typ). Vzniká tu však problém, že tento typ je povinne znamienkový a teda rozsah má od* -128 *do* 127*. Neviem ako by sa správal tento typ, ak by som mu priradil hodnotu napr.* 200*.*

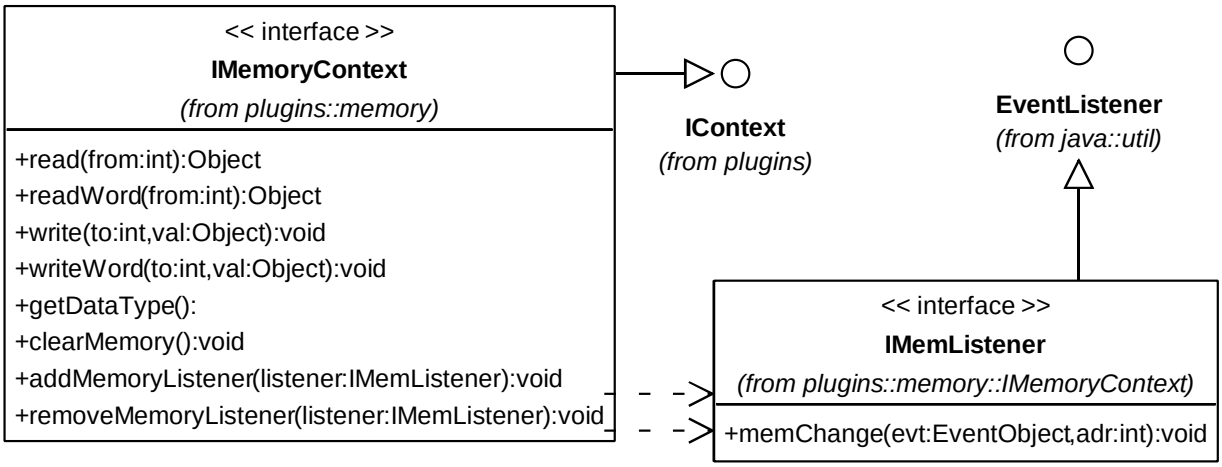
V prípade, že by prekladač túto hodnotu zobral ako dvojkový doplnok, tak hodnota by bola -56*. Ak by sme potom pracovali s touto hodnotou, museli by sme ju správne (nie jednoduchým pretypovaním) prekonvertovať na minimálne typ* `short`*. Jednoduchou konverziou by sme totiž neodstránili znamienko, museli by sme vymyslieť lepší spôsob. A vykonávanie takejto konverzie pri každom prístupe k OP (počas behu CPU môže ísť aj o 10 a viac tisíc prístupov za sekunu) je naozaj časovo náročné, čím by sa podstatne spomalila emulácia.*

V horšom prípade by mohol prekladač hodnotu orezať tak, aby sa zmestila do rozsahu -128 *až* 127*, v tomto prípade by bola hodnota* 72 *(odstránením najvyššieho bitu) a teda by sme pôvodnú hodnotu úplne stratili.*

Pri vytváraní nového projektu nezabudnite pridať ako externé referencie dve knižnice - `emu_ifaces.jar` a run-time knižnicu nástroja Cup s názvom `java-cup-11a-runtime.jar`.

2.6.2.1 Kontext pamäte

Štruktúru štandardného kontextu je možné vidieť na Obr. 2.9. Tento kontext obsahuje operácie spojené s prácou s operačnou pamäťou:



Obr. 2.9: *Štandardný kontext operačnej pamäte*

- Metódy `read()`, resp. `readWord()` vracajú hodnotu bunky OP. Typ vrátenej hodnoty nie je bližšie špecifikovaný (väčšinou býva `java.lang.Short`, resp. `java.lag.Integer`).
- Metódy `write()`, resp. `writeWord()` zapisujú na adresu OP novú hodnotu. Typ tejto hodnoty tiež nie je bližšie špecifikovaný.

- Metóda `clearMemory()` má zmazať celú operačnú pamäť
- Metóda `getDataType()` vracia typ buniek operačnej pamäte. Takýmto spôsobom sa vlaste bližšie špecifikuje typ vrátenej hodnoty metód `read()` a `readWord()`, resp. typ parametra novej hodnoty v metódach `write()`, resp. `writeWord()`.
- Metódou `addMemoryListener()` pridáme do zoznamu listenerov objekt typu `IMemoryContext.IMemListener`, ktorý je implementovaný mimo operačnej pamäte (kľudne aj v inom zásuvnom module). Jediná metóda tohto objektu (`memChange()`) je zavolaná vždy, keď nastane akákoľvek zmena v operačnej pamäti. Funkčná implementácia tejto metódy nie je až taká podstatná, ak v architektúrach využívajúcich túto operačnú pamäť sa to nebude využivať. My túto a nasledujúcu metódu pre zreteľnosť implementujeme.
- Metódou `removeMemoryListener()` odoberieme zo zoznamu listenerov konkrétny objekt typu `IMemoryContext.IMemListener`.

Ako je možné vidieť, štandardný kontext poskytuje dosť rozsiahlu "databázu" metód, ktoré postačujú našim účelom. Teda v našom prípade nebudeme nijak kontext rozširovať, a pre implementáciu použijeme štandardné rozhranie.
Triedu kontextu som nazval `BrainMemContext` a dal som ju do balíčka `brainduckmem.impl`. Určite si všimnete, že trieda obsahuje viac metód, ako je uvedené na Obr. 2.9. Nedajte sa však zmiasť, ostatné metódy (s klauzulou `@Override`) sú "zdedené" z rozhrania `plugins.IContext`.

```
package brainduckmem.impl;

import java.util.EventObject;
import javax.swing.event.EventListenerList;

import plugins.memory.IMemoryContext;

public class BrainMemContext implements IMemoryContext {
    private short[] mem; // toto je operačná pamäť

    // zoznam listenerov, ktorí budú informovaní
    // o zmenách OP
    private EventListenerList deviceList;
    private EventObject changeEvent;

    // Konštruktor
    public BrainMemContext() {
        changeEvent = new EventObject(this);
        deviceList = new EventListenerList();
    }
```

```
/**
 * Inicializuje kontext pamäte. Metóda je
 * volaná z implementácie hlavného rozhrania.
 *
 * @param size  Veľkosť pamäte
 * @return      vráti true ak inicializácia bola OK
 */
public boolean init(int size) {
    mem = new short[size];
    return true;
}

@Override
public void clearMemory() {
    for (int i = 0; i < mem.length; i++)
        mem[i] = 0;
    fireChange(-1); // informuj o zmene
}

@Override
public Class<?> getDataType() { return Short.class; }

@Override
public Object read(int from) { return mem[from]; }

@Override
public Object readWord(int from) {
    if (from == mem.length-1) return mem[from];
    int low = mem[from] & 0xFF;
    int high = mem[from+1];
    return (int)((high << 8)| low);
}

@Override
public void write(int to, Object val) {
    if (val instanceof Integer)
        mem[to] = (short)((Integer)val & 0xFF);
    else
        mem[to] = (short)((Short)val & 0xFF);
    fireChange(to);
}
```

```
@Override
public void writeWord(int to, Object val) {
    short low = (short)((Integer)val & 0xFF);
    mem[to] = low;
    fireChange(to);
    if (to < mem.length-1) {
        short high = (short)(((Integer)val >>> 8) & 0xFF);
        mem[to+1] = high;
        fireChange(to+1);
    }
}

@Override
public void addMemoryListener(IMemListener listener) {
    deviceList.add(IMemListener.class, listener);
}

@Override
public void removeMemoryListener(IMemListener listener) {
    deviceList.remove(IMemListener.class, listener);
}

@Override
public String getID() {
    return "brainduck_memory";
}

/**
 * Metóda vráti jednoznačný hash tohto kontextu.
 * Hash je vypočítaný podľa špecifického algoritmu.
 *
 * @return hash kontextu
 */
@Override
public String getHash() {
    return "949fe1a163b65ae72a06aeb09976cb47";
}
```

```
    /***
     * Táto metóda notifikuje všetkých listenerov,
     * že nastala zmena v bunke operačnej pamäti.
     * @param adr Adresa, na ktorej nastala zmena
     */
    private void fireChange(int adr) {
        Object[] listeners = deviceList.getListenerList();
        for (int i = listeners.length-2; i>=0; i-=2) {
            if (listeners[i]==IMemListener.class) {
                ((IMemListener)listeners[i+1]).
                    memChange(changeEvent, adr);
            }
        }
    }
}
```

2.6.2.2 Hlavné rozhranie

Triedu hlavného rozhrania som nazval `BrainDuckMem` a dal som ju tiež do balíčka `brainduckmem.impl`. Určite si všimnete, že trieda obsahuje viac metód, ako je uvedené na Obr. 2.8. Nedajte sa však zmiasť, ostatné metódy (s klauzulou `@Override`) sú ”zdedené” z rozhrania `plugins.IPlugin`.

```
package brainduckmem.impl;

import plugins.ISettingsHandler;
import plugins.memory.IMemory;
import plugins.memory.IMemoryContext;

public class BrainDuckMem implements IMemory {
    private BrainMemContext memContext;
    private long hash;
    @SuppressWarnings(”unused”)
    private ISettingsHandler settings;
    private int programStart;
    private int size;

    public BrainDuckMem(Long hash) {
        this.hash = hash;
        memContext = new BrainMemContext();
    }

    @Override
```

```
public String getTitle() { return ”BrainDuck OM”; }
@Override
public String getVersion() { return ”1.0b1”; }

@Override
public String getCopyright() {
    return ”\u00A9 Copyright 2009, P. Jakubčo”;
}
@Override
public String getDescription() {
    return ”BrainDuck operating memory. Don’t even have a GUI.”;
}

@Override
public long getHash() { return hash; }

@Override
public boolean initialize(int size, ISettingsHandler sHandler) {
    this.settings = sHandler;
    this.size = size;
    memContext.init(size);
    return true;
}

@Override
public void showGUI() {
    // my nemáme GUI
}

@Override
public void destroy() { }

@Override
public IMemoryContext getContext() {
    return memContext;
}

@Override
public int getProgramStart() {
    return programStart;
}

@Override
public int getSize() {    return size; }
```

```
    @Override
    public void reset() {}

    @Override
    public void setProgramStart(int address) {
        programStart = address;
    }

    @Override
    public void showSettings() {
        // nemáme ani GUI pre nastavenia
    }
}
```

Týmto je základná implementácia operačnej pamäte hotová. Bolo by dobré doplniť GUI, aby bolo možné vidieť (a prípadne aj meniť) jednotlivé bunky operačnej pamäte. Skompilujte všetky triedy a vytvorte z nich súbor `BrainMem.jar`. Tento zásuvný modul umiestnite do príslušného adresára.

2.7 Procesory

Nasleduje tvorba jadra celej architektúry - tvorba procesora. Procesor realizuje beh celej emulácie. Jeho základná činnosť je vykonávanie inštrukcií. Tiež interaguje so zariadeniami a s operačnou pamäťou. Implementácia procesora neobmedzuje programátora vo výbere použitia emulačnej techniky, ktorých je viac. Najjednoduchšou technikou je však interpretácia, a pre emuláciu starších 8-bitových procesorov bohato postačuje aj so svojim slabým výkonom.

Zásuvný modul CPU v platforme ***emuStudio*** okrem samotnej emulácie, musí spolupracovať pri riadení a zobrazovaní okna debuggera[4]. To zahŕňa aj tvorbu disassemblera, pretože všetky hodnoty v tomto okne (teda napr. mnemonický tvar inštrukcie, adresu, či je na nej breakpoint a operačný kód inštrukcie v peknom tvare) sú žiadané hlavným modulom **z CPU**. Teda tieto veci *neimplementuje* hlavný modul.

Okrem toho zásuvný modul musí implementovať stavové okno CPU[5], ktoré má zobrazovať vnútorný stav CPU - hodnoty jeho registrov, aktuálny stav behu CPU a možno aj nejaké nastavenia (napr. nastavenie run-time frekvencie a testovacej periódy, ako je to v mojich zásuvných moduloch procesorov Intel 8080 a Zilog Z80).

Tvorba CPU sa potom skladá z týchto častí:

- Tvorba hlavného rozhrania - samotného procesora realizujúceho emuláciu
- Tvorba kontextu procesora, doplňujúceho funkcionalitu procesora

[4] viď manuál k platforme ***emuStudio***, časť 4.2 Okno debuggera
[5] manuál, časť 4.3 Stavové okno

- Tvorba disassemblera a stĺpcov pre zoznam inštrukcií
- Tvorba GUI stavového okna

Tvorba CPU má možno podobnú, ak nie väčšiu zložitosť ako tvorba kompilátora. Napriek tomu rozhrania a postup jeho tvorby sú navrhnuté tak, aby táto tvorba bola čo najviac priamočiara, aby sa programátor mohol sústrediť na samotnú problematiku CPU, a mohol sa spoľahnúť na infraštruktúru poskytovanú platformou ***emuStudio***. Preto tvorba CPU si vyžaduje, aby programátor poznal hardvér, ktorý ide implementovať a "správne odpovedal" na "otázky" - metódy rozhraní.

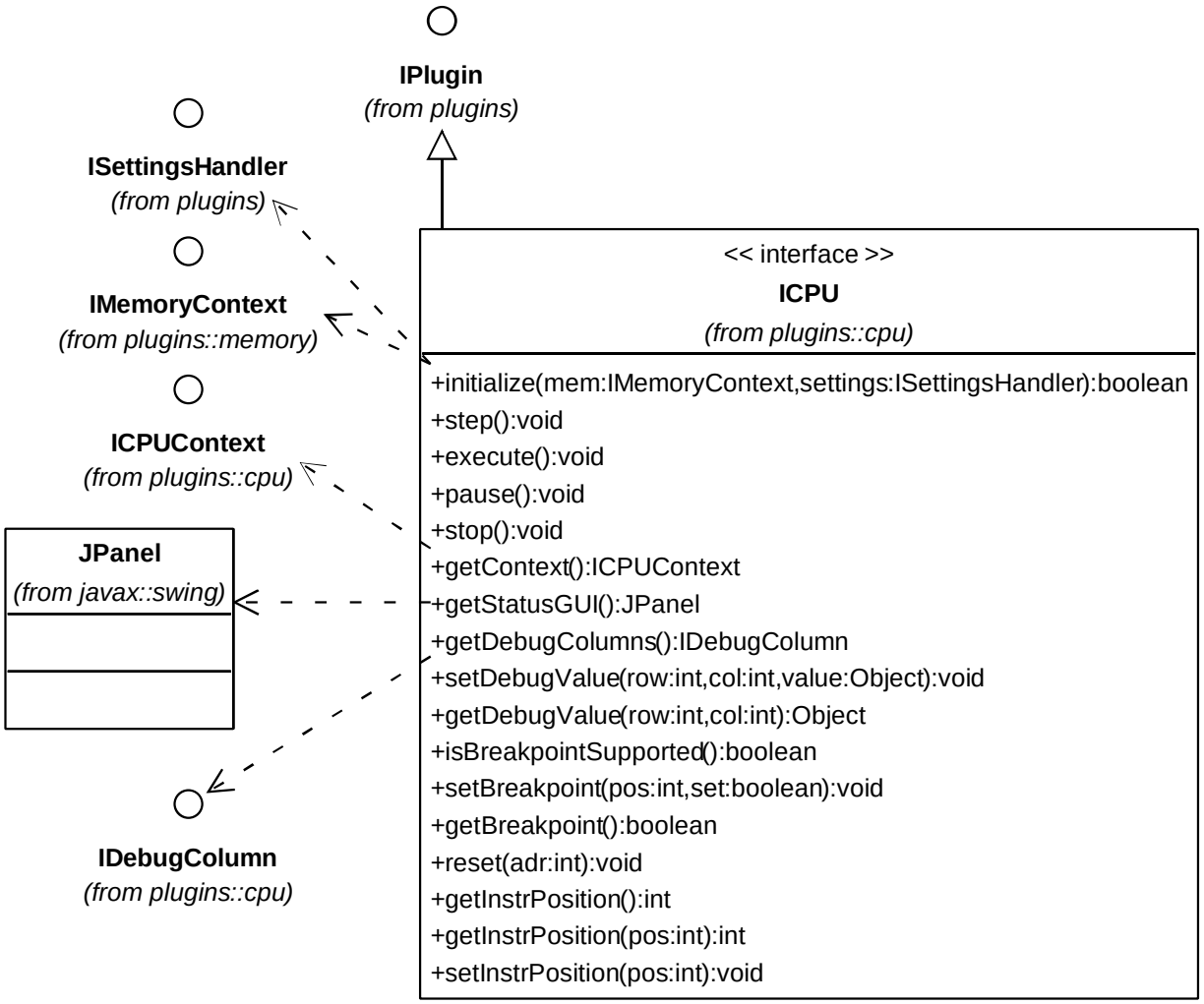
2.7.1 Rozhrania

Sada rozhraní z knižnice `lib\emu_ifaces.jar` pre procesor sa nachádza v balíčku `plugins.cpu`. **Hlavné rozhranie** má názov *ICPU* (Obr. 2.10).
Zoznam rozhraní, ktoré musí programátor implementovať je tu (počet tu hovorí o počte objektov, nie implementácií):

Názov	**Počet ks.**	**Popis**
ICPU	1	hlavné rozhranie (Obr. 2.10)
ICPUContext	1	kontext procesora
IDebugColumn	n	stĺpec v debug okne

Možno čitateľa prekvapí, ako veľa metód treba implementovať v hlavnom rozhraní. Ak sa však prizrieme bližšie na tieto metódy, zistíme, že väčšinou ide o fundamentálne a logické veci, a teda že už názov metódy programátorovi hovorí, ako má vyzerať implementácia. Implementácia hlavného rozhrania však bude zahŕňať určite aj viac metód, ktoré budú realizovať emuláciu inštrukcií.

- Metódu `initialize()` volá hlavný modul (komponent `ArchitectureHandler`) pri inicializačnom procese (Obr. 1.3). Táto metóda je volaná vždy a tiež vždy ako prvá, čo umožňuje inicializovať zásuvný modul. To okrem iného zahŕňa načítanie nastavení, a kontrolu, či CPU podporuje kontext pamäte.
- Metóda `step()` má vykonať krok emulácie, teda vykonanie jedinej inštrukcie. Po jej vykonaní sa nesmie zabudnúť na aktualizovanie GUI stavového okna.
- Metódou `execute()` sa má spustiť emulácia CPU bez umelého pozastavovania. Teda keď sa raz takto emulácia spustí a používateľ ju nezastaví, mala by sa zastaviť buď spontánne (napríklad vykonaním inštrukcie typu HALT), alebo vôbec. Najlepšie je implementovať tento permanentný beh emulácie do nového vlákna (`java.lang.Thread`), čím sa nezablokuje ostatná činnosť emulátora.
- Metóda `pause()` má *pozastaviť* emuláciu, teda uviesť ju do stavu, keď sa už inštrukcie prestanú vykonávať, ale CPU si zachová svoj vnútorný stav. Dalo by sa povedať, že CPU "zaspí". Napríklad, procesor zaspí po každom volaní metódy `step()` (ak by emulácia ešte spontánne mohla pokračovať).



Obr. 2.10: *Hlavné rozhranie procesora - ICPU*

- Metóda `stop()` má zastaviť činnosť procesora úplne. Dalo by sa povedať, že ho má "vypnúť" a teda jeho opätovné spustenie je možné len keď ho najprv resetujeme (volaním metódy `reset()` a tak re-inicializujeme jeho vnútorný stav.
- Metóda `getContext()` má vrátiť kontext procesora.
- Metóda `getStatusGUI()` má vrátiť GUI stavového okna CPU, ktoré je povinne implementované ako trieda rozširujúca triedu `javax.swing.JPanel`. GUI je možné vytvoriť ručne, ale v prostrediach napr. NetBeans, ale aj Eclipse existujú mechanizmy umožňujúce "kreslenie" GUI.
- Metóda `getDebugColumns()` má vrátiť **pole** objektov typu `IDebugColumn`. Tieto objekty predstavujú stĺpce v zozname inštrukcií okna debuggera v hlavnom module. K tomu sa ešte vrátim.

- Metódu `setDebugValue()` volá hlavný modul, keď používateľ zmení hodnotu v zozname inštukcií okna debuggera. Aby ju mohol zmeniť, musí to byť povolené (všetko je v implementácii stĺpcov). Parametrami sú číslo stĺpca, číslo riadka a nová hodnota.

- Metódu `getDebugValue()` volá hlavný modul pri aktualizovaní hodnôt tabuľky - zoznamu inštrukcií v okne debuggera. Metóda má vrátiť hodnotu zvoleného stĺpca a riadka (ktoré sú parametrami).

- Metóda `isBreakpointSupported()` vracia `true`, ak CPU podporuje body pozastavenia (breakpointy); `false`, ak nie. Ak CPU permanentne beží a počas čítania inštrukcií operačnej pamäte narazí na bod pozastavenia, tak pozastaví svoju činnosť, akoby bola zavolaná metóda `pause()`.

- Metóda `setBreakpoint()` má nastaviť breakpoint (bod pozastavenia)

- Metóda `getBreakpoint()` vráti `true`, ak na danej adrese operačnej pamäte sa nachádza breakpoint.

- V zásuvnom module CPU existujú dve implementácie metódy `reset()`. Obe metódy majú uviesť CPU do stavu pozastavenia, s re-inicializovaným vnútorným stavom (takým, aký má mať CPU pri svojom štarte).

  Rozdiel v týchto metódach je v dodatočnom parametri. Prvá metóda, zdedená z rozhrania `plugins.IPlugin` a bez parametra, pri re-inicializačnom procese má nastaviť programové počítadlo[6] na 0 (resp. počiatočnú adresu, odkiaľ CPU začína vždy po štarte vykonávať inštrukcie). Preťažená metóda v rozhraní `ICPU` má dodatočný parameter špecifikujúci, akú hodnotu má mať programové počítadlo po vykonaní re-inicializačného procesu.

- Rovnako tak metóda `getInstrPosition()` má dve preťaženia - jedno bez parametra, a síce táto metóda vracia adresu nasledujúcej inštrukcie, ktorá sa nachádza v pamäti hneď za aktuálne vykonávanou inštrukciou (na ktorú ukazuje programové počítadlo).

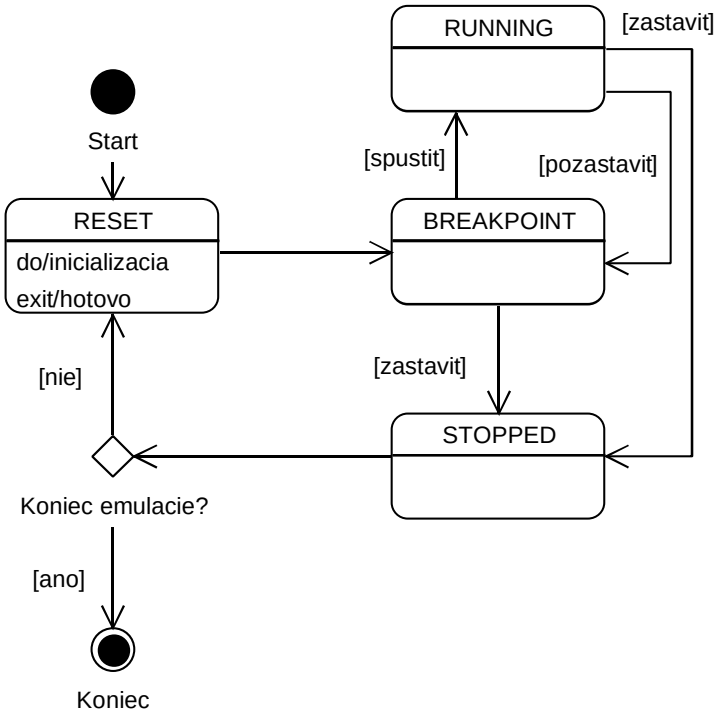
  Druhé preťaženie metódy je s dodatočným parametrom určujúcim adresu, od ktorej sa má nájsť adresa nasledujúcej inštrukcie.

  Tieto metódy slúžia na to, aby hlavný modul vedel efektívne organizovať zoznam inštrukcií zobrazený v okne debuggera.

- Nakoniec metóda `setInstrPosition()` nastaví programové počítadlo na novú hodnotu.

Pre jasnejšie objasnenie stavov behu procesora (tieto stavy sú pevne dané pre všetky zásuvné moduly procesorov), na Obr. 2.11 môžme vidieť "pracovný cyklus" CPU - jeho všetky stavy behu, do ktorých sa môže dostať a tiež z akého stavu.

[6] register smerníka na aktuálnu inštrukciu - tento register (s rôznymi názvami) má väčšina CPU, ak nie úplne všetky



Obr. 2.11: *"Pracovný cyklus" procesora na platforme emuStudio*

2.7.2 Postup pri tvorbe CPU

Teraz ako ukážku vytvorím procesor vymyslenej architektúry BrainDuck. Len pripomeniem, že procesor má dva registre (`IP` a `P`) a 8 inštrukcií (`inc`,`dec`,`incv`,`decv`,`print`, `load`,`loop`,`endl`). Vytvorenie takéhoto procesora je viac-menej veľmi jednoduché. Procesor bude podporovať body pozastavenia, implementujeme aj jednoduché GUI stavového okna a disassembler. Procesor však kvôli jednoduchosti nebude mať implementované nastavenie frekvencie, ani jej meranie (ktoré sa však bežne zíde).

2.7.2.1 Implementácia hlavného rozhrania

Teraz uvediem triedu implementujúcu hlavné rozhranie (`ICPU`), ktorej implementáciu budeme postupne dopĺňať. Okrem toho implementuje tiež rozhranie `java.lang.Runnable`, ktoré umožňuje vytvoriť nové vlákno z našej triedy. Využijeme to, keď budeme spúšťať emuláciu permanentne. Odporúčam čitateľovi pozrieť si prácu s vláknami v Jave.

Názov tejto triedy je `BrainCPU`, názov triedy kontextu CPU bude `BrainCPUContext` a obe triedy som vložil do balíčka `braincpu.impl`. Trieda `BrainCPU` nateraz implementuje len tie najjednoduchšie metódy a tiež breakpointy. Ostatné veci doplníme neskôr:

```
package braincpu.impl;

import java.util.HashSet;
import javax.swing.JPanel;

import plugins.ISettingsHandler;
import plugins.cpu.ICPU;
import plugins.cpu.ICPUContext;
import plugins.cpu.IDebugColumn;
import plugins.memory.IMemoryContext;
import runtime.StaticDialogs;

public class BrainCPU implements ICPU, Runnable {
    private final static String BRAIN_MEMCONTEXT =
        "949fe1a163b65ae72a06aeb09976cb47"; // hash kontextu
                                            // známej pamäte
    private long hash;
    private ISettingsHandler settings;

    private IMemoryContext mem;       // kontext pamäte
    private BrainCPUContext cpu;      // kontext procesora
    private int run_state;            // uchovanie stavu behu
    private HashSet<Integer> breaks; // zoznam breakpointov

    private int IP, P;                // registre procesora
    private Thread cpuThread;         // vlákno pre permanentný
                                      // beh procesora
    // konštruktor
    public BrainCPU(Long hash) {
        this.hash = hash;
        cpuThread = null;
        cpu = new BrainCPUContext();
        run_state = ICPU.STATE_STOPPED_NORMAL;
        breaks = new HashSet<Integer>();
    }

    @Override
    public String getCopyright() {
        return "\u00A9 Copyright 2009, P. Jakubčo";
    }
    @Override
    public String getDescription() {
        return "CPU for BrainDuck architecture";
    }
```

```
@Override
public String getTitle() { return ”BrainCPU”; }

@Override
public String getVersion() { return ”1.0b1”; }

@Override
public long getHash() { return hash; }

@Override
public ICPUContext getContext() {
    return cpu;
}

@Override
public boolean initialize(IMemoryContext mem, ISettingsHandler settings) {
    if (mem == null)
        throw new java.lang.NullPointerException(
            ”CPU must have access to memory”);
    if (!mem.getID().equals(”brainduck_memory”)
            || !mem.getHash().equals(BRAIN_MEMCONTEXT)
            || (mem.getDataType() != Short.class)) {
        StaticDialogs.showErrorMessage(”Operating memory”
                + ” type is not supported for this kind of CPU.”);
        return false;
    }
    this.mem = mem;
    this.settings = settings;
    return true;
}

/**** BREAKPOINTY *************************/

@Override
public boolean isBreakpointSupported() {
    return true;
}

@Override
public void setBreakpoint(int pos, boolean set) {
    if (set) breaks.add(pos);
    else breaks.remove(pos);
}
```

```
@Override
public boolean getBreakpoint(int pos) {
    return breaks.contains(pos);
}

/**** OKNO DEBUGGERA *************************/

@Override
public IDebugColumn[] getDebugColumns() {
    return null; // implementujeme..
}

@Override
public Object getDebugValue(int row, int col) {
    return null; // implementujeme..
}

@Override
public void setDebugValue(int row, int col, Object val) { }

/**** POZÍCIA INŠTRUKCIE A REGISTRA IP ***************/

@Override
public int getInstrPosition() {
    return IP;
}

/**
 * Vráti adresu nasledujúcej inštruckie od adresy
 * pos. Využívame fakt, že každá inštrukcia v
 * architektúre BrainDuck má veľkosť len 1 byte.
 * @param pos adresa inštrukcie I1
 * @return adresa nasledujúcej inštrukcie (po I1)
 */
@Override
public int getInstrPosition(int pos) {
    return pos+1;
}
@Override
public boolean setInstrPosition(int pos) {
    if (pos < 0) return false;
    IP = pos;
    return true;
}
```

```
    /**** GUI *************************/

    @Override
    public JPanel getStatusGUI() {
        return null; // implementujeme..
    }

    @Override
    public void showSettings() {
        // Nemáme žiadne GUI nastavení
    }

    /**** EMULÁCIA *************************/

    @Override
    public void reset() { reset(0); }

    @Override
    public void reset(int adr) { }

    @Override
    public void execute() { }

    @Override
    public void run() { }

    @Override
    public void pause() { }

    @Override
    public void step() { }

    @Override
    public void stop() { }

    @Override
    public void destroy() {
        run_state = ICPU.STATE_STOPPED_NORMAL;
    }
}
```

2.7.2.2 Okno debuggera

Okno debuggera je v hlavnom module implementované ako tabuľka `javax.swing.JTable`. Každá bunka môže mať rôzne vlastnosti, teda je možné ju zobraziť a pracovať s ňou úplne rôznym spôsobom. Odporúčam si pozrieť, ako sa v Jave pracuje s tabuľkami pre lepšie pochopenie, prečo je spôsob interakcie CPU s oknom debuggera riešený tak, ako je.

V okne debuggera sa budú nachádzať 4 stĺpce - *breakpoint*, *address*, *mnemo*, *opcode*. Pod každým stĺpcom budú riadky hodnôt, ktoré budú mať rovnaké vlastnosti pre celý stĺpec. Medzi tieto vlastnosti patrí:

- typ bunky (napr. `java.lang.String`, alebo `java.lang.Integer`, alebo prečo nie napríklad `javax.swing.ImageIcon`, atď.)
- či je bunka editovateľná

Keďže budú mať všetky bunky v danom stĺpci rovnaké vlastnosti, zostáva už len pridať meno stĺpca. Všetky stĺpce implementujeme ako jednu triedu, implementujúcu rozhranie `plugins.cpu.IDebugColumn`. Názov triedy som zvolil `ColumnInfo`, a umiestnil som ju do balíčka `braincpu.gui`.

```
package braincpu.gui;

import plugins.cpu.IDebugColumn;

public class ColumnInfo implements IDebugColumn {
    private String name;
    private Class<?> type;
    private boolean editable;

    public ColumnInfo(String name, Class<?> cl, boolean editable) {
        this.name = name;
        this.type = cl;
        this.editable = editable;
    }

    @Override
    public Class<?> getType() { return this.type; }
    @Override
    public String getName() { return this.name; }
    @Override
    public boolean isEditable() { return this.editable; }
}
```

Teraz je potrebné vytvoriť objekty týchto 4 spomínaných stĺpcov a vrátiť ich ako pole v metóde `getDebugColumns()` v triede `BrainCPU`.

Predtým však je potrebné povedať, že vyhodnotenie buniek tejto "tabuľky" (ako napr. zistenie mnemonického tvaru inštrukcie na danej adrese) vyžaduje použitie disassemblera. Inštrukcie sú v pamäti zakódované binárne (preložené kompilátorom) a preto ich potrebujeme znova dekódovať do čitateľného tvaru (čiže disassemblovať). Disassembler sa preto bude volať pri aktualizácii každého riadka v tabuľke okna debuggera, teda pri každom volaní metódy `getDebugValue()`. Disassembler a okno debuggera teda spolu súvisia.

Je dobré, keď všetok súvisiaci kód dáme do jednej triedy. Okrem toho, disassembler býva často rozsiahly (síce vôbec nie až tak veľký ako kompilátor, ale často zaberie možno 200 a viac riadkov kódu). Na tento účel som vytvoril triedu s názvom `BrainDisassembler`, ktorú som umiestnil do balíčka `braincpu.gui`.

```
package braincpu.gui;

import braincpu.impl.BrainCPU;
import plugins.cpu.IDebugColumn;
import plugins.memory.IMemoryContext;

public class BrainDisassembler {
    private IMemoryContext mem;     // pamäť bude potrebná pre
                                    // čítanie bytov pre dekódovanie
                                    // inštrukcií
    private BrainCPU cpu;           // procesor je potrebný pre
                                    // zistenie breakpointu na danej
                                    // adrese
    private IDebugColumn[] columns; // stĺpce okna debuggera

    /**
     * V konštruktore vytvorím stĺpce ako objekty
     * triedy ColumnInfo.
     *
     * @param mem  kontext operačnej pamäte, ktorý bude
     *             potrebný pre dekódovanie inštrukcií
     */
    public BrainDisassembler(IMemoryContext mem, BrainCPU cpu) {
        this.mem = mem;
        this.cpu = cpu;
        columns = new IDebugColumn[4];
        IDebugColumn c1 = new ColumnInfo("breakpoint",
            Boolean.class,true);
        IDebugColumn c2 = new ColumnInfo("address",
            String.class,false);
        IDebugColumn c3 = new ColumnInfo("mnemonics",
            String.class,false);
        IDebugColumn c4 = new ColumnInfo("opcode",
```

```
            String.class,false);

        columns[0] = c1;columns[1] = c2;columns[2] = c3;
        columns[3] = c4;
    }

    /**
     * Metóda vráti stĺpce pre okno debuggera ako pole.
     *
     * @return pole stĺpcov
     */
    public IDebugColumn[] getDebugColumns() { return columns; }

    /**
     * Metóda vráti hodnotu pre okno debuggera, pre daný
     * riadok a stĺpec. Riadok sa berie ako
     * <strong>adresa</strong>. O odovzdanie správneho parametra
     * adresy sa stará hlavný modul.
     *
     * @param row  Riadok okna debuggera (adresa v OP)
     * @param col  Stĺpec v okne debuggera
     * @return hodnota pre daný riadok a stĺpec
     */
    public Object getDebugColVal(int row, int col) {
        try {
            // najprv musíme disassemblovať inštrukciu
            // na danej adrese (row)
            CPUInstruction instr = cpuDecode(row);
            switch (col) {
                case 0: /* stĺpec č.0 je breakpoint */
                    return cpu.getBreakpoint(row);
                case 1: /* stĺpec č.1 je adresa */
                    return String.format("%04Xh", row);
                case 2: /* stĺpec č.2 je mnemonický tvar inštr. */
                    return instr.getMnemo();
                case 3: /* stĺpec č.3 je operačný kód */
                    return instr.getOperCode(); // operacny kod
                default: return "";
            }
        } catch(IndexOutOfBoundsException e) {
            // Tu sa dostaneme iba v prípade, ak používateľ manuálne
            // zmenil hodnotu operačnej pamäte tak, že vyjadruje
            // inštrukciu s viacerými bytami, ktoré sa už nezmestili
            // do operačnej pamäte a teda vznikla výnimka pri
```

```
            // disassemblovaní. Pre architektúru BrainDuck sa tu
            // nedostaneme nikdy, ale uvádzam tento kód len pre
            // ilustráciu.
            switch (col) {
                case 0: return cpu.getBreakpoint(row);
                case 1: return String.format("%04Xh", row);
                case 2: return "incomplete instruction";
                case 3: return String.format("%X",
                            (Short)mem.read(row));
                default: return "";
            }
        }
    }

    /**
     * Metóda je volaná, ak používateľ v okne debuggera zmenil
     * hodnotu na pozícii riadka row a stĺpca col. Pre stĺpce,
     * ktorých riadky nie sú editovateľé, sa táto metóda
     * nezavolá. Stĺpec č. 0 predstavuje breakpoint, a ten
     * používateľ môže zmeniť aj z okna debuggera.
     *
     * @param row   Riadok v okne debuggera
     * @param col   Stĺpec v okne debuggera
     * @param value Nová hodnota
     */
    public void setDebugColVal(int row, int col, Object value) {
        if (col != 0) return;
        if (value.getClass() != Boolean.class) return;

        boolean v = Boolean.valueOf(value.toString());
        cpu.setBreakpoint(row,v);
    }

    /**
     * Disassembler. Vykoná dekódovanie jednej inštrukcie
     * začínajúcej na adrese memPos.
     *
     * @param memPos  adresa, kde začína inštrukcia
     * @return dekódovaná inštrukcia
     */
    private CPUInstruction cpuDecode(int memPos) {
        short val;
        CPUInstruction instr;
        String mnemo, oper;
```

```
        val = ((Short)mem.read(memPos++)).shortValue();
        oper = String.format("%02X",val);

        switch (val) {
            case 1: mnemo = "inc"; break;
            case 2: mnemo = "dec"; break;
            case 3: mnemo = "incv"; break;
            case 4: mnemo = "decv"; break;
            case 5: mnemo = "print"; break;
            case 6: mnemo = "load"; break;
            case 7: mnemo = "loop"; break;
            case 8: mnemo = "endl"; break;
            default: mnemo = "unknown";
        }
        instr = new CPUInstruction(mnemo,oper);
        return instr;
    }
}
```

Určite si čitateľ všimol zatiaľ neexistujúcu triedu `CPUInstruction`. Táto trieda uchováva jednu disassemblovanú inštrukciu:

```
package braincpu.gui;

public class CPUInstruction {
    private String mnemo;
    private String operCode;

    public CPUInstruction(String mnemo, String opCode) {
        this.mnemo = mnemo;
        this.operCode = opCode;
    }

    public String getMnemo() { return this.mnemo; }
    public String getOperCode() { return this.operCode; }
}
```

Nakoniec je potrebné v hlavnej triede `BrainCPU` doplniť vytvorenie objektu triedy `BrainDisassembler` a doplniť kód pre metódy `getDebugColumns()`, `getDebugValue()` a `setDebugValue()`:

```
...
public class BrainCPU implements ICPU, Runnable {
    ...
    private BrainDisassembler dis;   // disassembler
    ...
    public boolean initialize(IMemoryContext mem,
            ISettingsHandler settings) {
        ...
        dis = new BrainDisassembler(mem,this);
        return true;
    }
    ...
    /**** OKNO DEBUGGERA *************************/
    @Override
    public IDebugColumn[] getDebugColumns() {
        return dis.getDebugColumns();
    }
    @Override
    public Object getDebugValue(int row, int col) {
        return dis.getDebugColVal(row, col);
    }
    @Override
    public void setDebugValue(int row, int col, Object val) {
        dis.setDebugColVal(row, col, val);
    }
    ...
}
```

2.7.2.3 GUI stavového okna

Nasleduje tvorba stavového okna. Okno implementujeme ako triedu odvodenú od triedy `javax.swing.JPanel` (ako to predpisuje hlavné rozhranie `ICPU`). Kód, ktorý uvediem, je napísaný ručne bez použitia akýchkoľvek vizuálnych nástrojov na vytváranie GUI. Čitateľ však nemusí nasledovať môj príklad, a môže si okno nakresliť. V okne sa budú zobrazovať registre `P` a `IP` a stav behu CPU (Obr. 2.12).

Obr. 2.12: *GUI stavového okna CPU*

Triedu som nazval `BrainStatusPanel` a umiestnil som ju do balíčka `braincpu.gui`.

```
package braincpu.gui;

import java.awt.Color;
import java.awt.Font;
import java.util.EventObject;

import javax.swing.BorderFactory;
import javax.swing.GroupLayout;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.LayoutStyle;

import plugins.cpu.ICPU;
import plugins.cpu.ICPUContext.ICPUListener;

import braincpu.impl.BrainCPU;

@SuppressWarnings("serial")
public class BrainStatusPanel extends JPanel {
    public BrainStatusPanel(final BrainCPU cpu) {
        initComponents();

        cpu.getContext().addCPUListener(new ICPUListener() {
            @Override
            public void runChanged(EventObject evt, int state) {
                switch (state) {
                    case ICPU.STATE_STOPPED_NORMAL:
                        lblStatus.setText("stopped (normal)");
                        break;
                    case ICPU.STATE_STOPPED_BREAK:
                        lblStatus.setText("breakpoint");
                        break;
                    case ICPU.STATE_STOPPED_ADDR_FALLOUT:
                        lblStatus.setText("stopped "
                            + "(address fallout)");
                        break;
                    case ICPU.STATE_STOPPED_BAD_INSTR:
                        lblStatus.setText("stopped "
                            + "(instruction fallout)");
                        break;
                }
```

```
        }
        @Override
        public void stateUpdated(EventObject evt) {
            txtP.setText(String.format("%04X", cpu.getP()));
            txtIP.setText(String.format("%04X", cpu.getIP()));
        }

    });
}

/**
 * Inicializácia GUI komponentov
 */
private void initComponents() {
    JLabel lblP = new JLabel("P");
    JLabel lblIP = new JLabel("IP");
    txtP = new JTextField("0000");
    txtIP = new JTextField("0000");
    lblStatus = new JLabel("breakpoint");

    lblStatus.setFont(lblStatus.getFont().deriveFont(Font.BOLD));
    txtP.setEditable(false);
    txtP.setBorder(BorderFactory.createMatteBorder(1, 5, 1, 1,
        Color.lightGray));
    txtIP.setEditable(false);
    txtIP.setBorder(BorderFactory.createMatteBorder(1, 5, 1, 1,
        Color.lightGray));

    GroupLayout layout = new GroupLayout(this);
    this.setLayout(layout);
    layout.setHorizontalGroup(
        layout.createSequentialGroup()
        .addContainerGap()
        .addGroup(layout.createParallelGroup(
                GroupLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()
                .addGroup(layout.createParallelGroup(
                        GroupLayout.Alignment.LEADING)
                    .addComponent(lblP)
                    .addComponent(lblIP))
                .addPreferredGap(
                    LayoutStyle.ComponentPlacement.RELATED)
                .addGroup(layout.createParallelGroup(
```

```
                                GroupLayout.Alignment.LEADING)
                            .addComponent(txtP)
                            .addComponent(txtIP)))
                    .addComponent(lblStatus))
                    .addContainerGap(20,Short.MAX_VALUE));
            layout.setVerticalGroup(
                layout.createSequentialGroup()
                .addContainerGap()
                .addGroup(layout.createParallelGroup(
                        GroupLayout.Alignment.BASELINE)
                    .addComponent(lblP)
                    .addComponent(txtP))
                .addGroup(layout.createParallelGroup(
                        GroupLayout.Alignment.BASELINE)
                    .addComponent(lblIP)
                    .addComponent(txtIP))
                .addPreferredGap(
                    LayoutStyle.ComponentPlacement.UNRELATED)
                .addComponent(lblStatus)
                .addContainerGap());
    }
    private JTextField txtP;
    private JTextField txtIP;
    private JLabel lblStatus;
}
```

Možno si čitateľ všimol, že pre aktualizáciu GUI som do kontextu procesora pridal nový objekt `ICPUListener`, ktorého príslušné metódy sú volané pri zmene stavu CPU. Samozrejme, to ešte potrebujeme implementovať. Predtým však treba doplniť implementáciu metódy `getStatusGUI()` v hlavnej triede `BrainCPU`:

```
    public JPanel getStatusGUI() {
        return new BrainStatusPanel(this);
    }
```

2.7.2.4 Kontext procesora

Tieto veci implementujeme v kontexte procesora. Tento kontext rozšírime od štandardného pridaním metód na pripojenie a odpojenie jedného zariadenia (v budúcnosti do procesora pripojíme zariadenie *Terminal*, popísané v časti 2.8.2).

Rozšírené rozhranie som nazval `IBrainCPUContext` a umiestnil som ho do balíčka `braincpu.interfaces`:

```
package braincpu.interfaces;

import plugins.cpu.ICPUContext;
import plugins.device.IDeviceContext;

public interface IBrainCPUContext extends ICPUContext {
    public boolean attachDevice(IDeviceContext device);
    public void detachDevice();
}
```

Z tohto rozhrania je potrebné vypočítať hash, podľa algoritmu popísaného v časti 2.3.2.1. Teno hash je: `d32e73041bf765d98eea1b8664ef6b5e`.

Kontext som umiestnil do balíčka `braincpu.impl` a názov som mu dal `BrainCPUContext`.

```
package braincpu.impl;

import java.util.EventObject;
import javax.swing.event.EventListenerList;
import braincpu.interfaces.IBrainCPUContext;
import plugins.device.IDeviceContext;

public class BrainCPUContext implements IBrainCPUContext {
    private EventListenerList listenerList;
    private EventObject cpuEvt;
    private IDeviceContext device;

    public BrainCPUContext() {
        device = null;
        listenerList = new EventListenerList();
        cpuEvt = new EventObject(this);
    }

    @Override
    public String getID() { return "brain-cpu-context"; }

    @Override
    public String getHash() {
        return "d32e73041bf765d98eea1b8664ef6b5e";
    }

    @Override
    public void addCPUListener(ICPUListener listener) {
        listenerList.add(ICPUListener.class, listener);
    }
```

```
@Override
public void removeCPUListener(ICPUListener listener) {
    listenerList.remove(ICPUListener.class, listener);
}

/**
 * Pripojenie zariadenia do CPU. Procesor BainCPU môže mať
 * pripojené len jedno zariadenie, a to terminál
 *
 * @param listener
 * @param port
 * @return
 */
@Override
public boolean attachDevice(IDeviceContext device) {
    if (this.device != null) return false;
    if (device.getDataType() != Short.class) return false;
    this.device = device;
    return true;
}

@Override
public void detachDevice() {
    device = null;
}

/**
 * Túto metódu zavolá BrainCPU, keď sa zmení
 * stav behu procesora. Táto metóda potom zavolá
 * metódu runChanged všetkých listenerov.
 *
 * @param run_state  nový stav behu procesora
 */
public void fireCpuRun(int run_state) {
    Object[] listeners = listenerList.getListenerList();
    for (int i=0; i<listeners.length; i+=2) {
        if (listeners[i] == ICPUListener.class)
            ((ICPUListener)listeners[i+1])
                .runChanged(cpuEvt, run_state);
    }
}
```

```
    /**
     * Túto metódu zavolá BrainCPU, keď sa zmení
     * vnútorný stav procesora (zmenia sa registre).
     * Táto metóda potom zavolá metódu stateUpdated
     * všetkých listenerov.
     */
    public void fireCpuState() {
        Object[] listeners = listenerList.getListenerList();
        for (int i=0; i<listeners.length; i+=2) {
            if (listeners[i] == ICPUListener.class)
                ((ICPUListener)listeners[i+1]).stateUpdated(cpuEvt);
        }
    }

    /**
     * Metóda zapíše do zariadenia hodnotu val. Táto
     * metóda teda priamo komunikuje so zariadením.
     * @param val hodnota, ktorá sa má zapísať do zariadenia
     */
    public void writeToDevice(short val) {
        if (device == null) return;
        device.out(cpuEvt, val);
    }

    /**
     * Metóda prečíta hodnotu zo zariadenia. Ak zariadenie
     * nemá nič čo poslať, nech pošle 0.. Táto metóda
     * teda priamo komunikuje so zariadením.
     * @return hodnotu zo zariadenia
     */
    public short readFromDevice() {
        if (device == null) return 0;
        return (Short)device.in(cpuEvt);
    }
}
```

2.7.2.5 Emulácia

Týmto sme sa dostali až do poslednej a najdôležitejšej fázy vývoja procesora - jeho emulácie. Dalo by sa povedať, že predchádzajúce časti tvoria akýsi podklad, alebo "backend" pre samotné jadro procesora, ktoré budeme vytvárať v tejto časti. Ako som už uviedol skôr, na emuláciu procesora využijeme najjednoduchšiu emulačnú techniku - interpretáciu inštrukcií. Interpretácia "napodobňuje" činnosť skutočného procesora a rozdeľuje vykonávanie inštrukcií do niekoľkých fáz:

1. `FETCH` - načíta inštrukciu (alebo jej časť) z OP
2. `DECODE` - dekóduje inštrukciu (spolu s načítaním ostatných častí inštrukcie z OP)
3. `EXECUTE` - vykoná inštrukciu (podľa jej sémantiky)
4. `STORE` - uloží výsledok vykonania, zmení sa tým vnútorný stav procesora

Takže teraz implementujeme zvyšok metód v hlavnom rozhraní: `reset(adr)`,`step()`,`stop()`, `pause()`,`execute()` a `run()`:

```
...
public class BrainCPU implements ICPU, Runnable {
    ...
    /**** EMULÁCIA *************************/
    ...
    @Override
    public void reset(int adr) {
        IP = adr; // programové počítadlo na adr

        // nájdeme najbližiu "voľnú" adresu,
        // kde sa už nenachádza program
        try {
            while((Short)mem.read(adr++) != 0)
                ;
        } catch(IndexOutOfBoundsException e) {
            // tu sa dostaneme, ak "adr"
            // už bude ukazovať na neexistujúcu
            // pamäť, čiže keď prejdeme celou pamäťou
            // bez výsledku
            adr = 0;
        }
        P = adr; // a priradíme register P
                 // adresu za programom

        // zmeníme stav behu na "breakpoint"
        run_state = ICPU.STATE_STOPPED_BREAK;
        cpuThread = null;

        // oznámime zmenu stavu listenerom
        cpu.fireCpuRun(run_state);
        cpu.fireCpuState();
    }
```

```
@Override
public void execute() {
    // na permanentný beh vytvoríme
    // samostatné vlákno
    cpuThread = new Thread(this, ”BrainCPU”);
    // týmto ho ”odštartujeme” a spôsobí
    // to spustenie nasledujúcej metódy ”run()”
    cpuThread.start();
}

// metóda z rozhrania Runnable
@Override
public void run() {
    // zmeníme stav na ”running”
    run_state = ICPU.STATE_RUNNING;
    // oznámime zmenu stavu listenerom
    cpu.fireCpuRun(run_state);
    // nasleduje v podstate nekonečný cyklus,
    // pokiaľ emuláciu niečo nezastaví
    // externe: používateľ,
    // interne: chybná inštrukcia, neexistujúca pamäť
    while(run_state == ICPU.STATE_RUNNING) {
        try {
            // ak je na adrese IP nastavený breakpoint,
            // vyhodenie výnimky typu Error
            if (getBreakpoint(IP) == true)
                throw new Error();
            emulateInstruction();
        } catch (IndexOutOfBoundsException e) {
            // tu sa dostaneme, ak IP
            // ukazuje na neexistujúcu pamäť
            run_state = ICPU.STATE_STOPPED_ADDR_FALLOUT;
            break;
        }
        catch (Error er) {
            // odchytenie výnimky Error - zmena
            // stavu na ”breakpoint”
            run_state = ICPU.STATE_STOPPED_BREAK;
            break;
        }
    }
    cpu.fireCpuState();
    cpu.fireCpuRun(run_state);
}
```

```
@Override
public void pause() {
    // zmeníme stav behu na ”breakpoint”
    run_state = ICPU.STATE_STOPPED_BREAK;
    // oznámime zmenu stavu listenerom
    cpu.fireCpuRun(run_state);
}

@Override
public void step() {
    // ak je stav ”breakpoint”
    if (run_state == ICPU.STATE_STOPPED_BREAK) {
        try {
            // zmeníme stav na ”running”
            run_state = ICPU.STATE_RUNNING;
            emulateInstruction();
            // ak by emulácia sponntánne
            // pokračovala (ak ju externe nič
            // nezastavilo)
            if (run_state == ICPU.STATE_RUNNING)
                // tak zmeníme stav späť na ”breakpoint”
                run_state = ICPU.STATE_STOPPED_BREAK;
        }
        catch (IndexOutOfBoundsException e) {
            // tu sa dostaneme, ak IP
            // ukazuje na neexistujúcu pamäť
            run_state = ICPU.STATE_STOPPED_ADDR_FALLOUT;
        }
        // oznámime stav procesora listenerom
        cpu.fireCpuRun(run_state);
        cpu.fireCpuState();
    }
}

@Override
public void stop() {
    // zmeníme stav behu na ”stopped”
    run_state = ICPU.STATE_STOPPED_NORMAL;
    // oznámime zmenu stavu listenerom
    cpu.fireCpuRun(run_state);
}
...
```

```
    /**
     * Metóda emuluje jednu inštrukciu.
     */
    private void emulateInstruction() {
        // implementujeme..
    }
}
```

Všimnite si na konci metódu `emulateInstruction()`. Táto metóda je poslednou metódou, ktorú ideme implementovať, a zároveň tou najdôležitejšou. Práve v tejto metóde sa totiž bude nachádzať kód pre emuláciu jednej inštrukcie. Ako si čitateľ môže všimnúť z predchádzajúceho kódu, permanentný beh je realizovaný "skoro" nekonečným cyklom, v ktorom sa volá práve metóda. Implementácia metódy zahŕňa všetky 4 fázy, ktoré som spomínal na začiatku časti "Emulácia".

Nasleduje implementácia metódy `emulateInstruction()`. Je dobré umiestniť ju celkom na koniec triedy, pretože obyčajne pôjde o najdlhšiu metódu zo všetkých:

```
/**
 * Metóda emuluje jednu inštrukciu.
 */
private void emulateInstruction() {
    short OP; // operačný kód

    // FETCH
    OP = ((Short)mem.read(IP++)).shortValue();

    // DECODE
    switch(OP) {
        case 1: /* INC */
            P++; return;
        case 2: /* DEC */
            P--; return;
        case 3: /* INCV */
            mem.write(P, (Short)mem.read(P) + 1);
            return;
        case 4: /* DECV */
            mem.write(P, (Short)mem.read(P) - 1);
            return;
        case 5: /* PRINT */
            cpu.writeToDevice((Short)mem.read(P));
            return;
        case 6: /* LOAD */
            mem.write(P, cpu.readFromDevice());
            return;
```

```
case 7: { /* LOOP */
    if ((Short)mem.read(P) != 0)
        return;
    byte loop_count = 0; // počítadlo vnorenia
                         // v cykle (čiže počítadlo
                         // cyklov). Nerátam s viac ako
                         // 127 vnoreniami
    // inak hľadáme inštrukciu "endl" na
    // aktuálnej úrovni vnorenia (podľa loop_count)
       // IP je nastavený na nasledujúcu inštrukciu
    while ((OP = (Short)mem.read(IP++)) != 0) {
        if (OP == 7) loop_count++;
        if (OP == 8) {
            if (loop_count == 0)
                return;
            else
                loop_count--;
        }
    }
    // tu sme už na konci programu, čiže
    // niekde je chyba
    break;
}
case 8: /* ENDL */
    if ((Short)mem.read(P) == 0)
        return;
    byte loop_count = 0; // počítadlo vnorenia
                         // v cykle (čiže počítadlo
                         // cyklov). Nerátam s viac ako
                         // 127 vnoreniami
    // inak hľadáme inštrukciu "loop" hore na
    // aktuálnej úrovni vnorenia (podľa loop_count)
    IP--; // IP späť na túto inštrukciu
    while ((OP = (Short)mem.read(--IP)) != 0) {
        if (OP == 8) loop_count++;
        if (OP == 7) {
            if (loop_count == 0)
                return;
            else
                loop_count--;
        }
    }
    // tu sme už na konci programu, čiže
    // niekde je chyba
```

```
            break;
        default: /* chybná inštrukcia*/
            break;
    }
    run_state = ICPU.STATE_STOPPED_BAD_INSTR;
}
```

Týmto sme ukončili tvorbu procesora. Skompilujte všetky triedy, z ktorých vytvorte archív `BrainCPU.jar` a umiestnite ho medzi ostatné zásuvné moduly procesorov.

Možno by to chcelo pridať ďalšiu inštrukciu `halt` na zastavenie procesora. Len na to bude potrebné doplniť aj kompilátor. Teraz už máme hotovú funkčnu architektúru, ktorú si môžte vyskúšať. Zatiaľ však nebude fungovať čítanie údajov z klávesnice, ani výpis na obrazovku. Tieto veci implementujeme v nasledujúcej časti.

2.8 Zariadenia

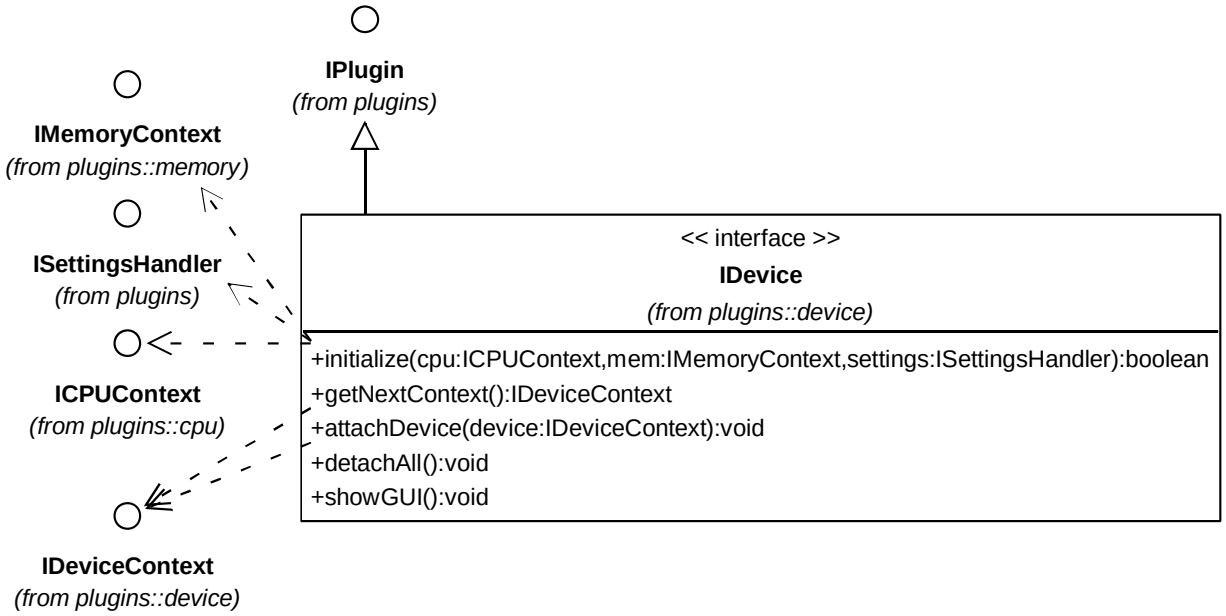
Periférne zariadenia sa implementujú až na koniec, keď už je jasná celá ostatná architektúra - sú už známe kontexty procesora a pamäte. Zložitosť tvorby zariadení závisí od toho, aké zariadenie treba implementovať. Je možné implementovať virtuálne zariadenia, ktoré komunikujú so skutočnými zariadeniami. Pri programovaní zariadení treba mať na zreteli tieto skutočnosti:

- V prípade, že sa zariadenia zapájajú do procesora, treba to urobiť pri inicializácii zariadenia. Hlavný modul toto automaticky neurobí.
- Kontexty zariadení predstavujú akési "zásuvky", či "zástrčky", ktoré môžu umožňovať pripojenie ďalších zariadení a sú tiež použité pre zapojenie do iných zariadení. Hlavný modul zariadenia medzi sebou prepojí automaticky.
- Zariadenie tak môže mať niekoľko kontextov. Napríklad sériová karta môže mať niekoľko fyzických portov, do ktorých je možné zapojiť rôzne zariadenia (do každého portu jedno).
- V prípade, že zariadenie sústavne vykonáva nejakú činnosť (bez ohľadu na výzvy z CPU), treba túto činnosť umiestniť do samostatného vlákna

2.8.1 Rozhrania

Sada rozhraní z knižnice `lib\emu_ifaces.jar` pre zariadenia sa nachádza v balíčku `plugins.device`. **Hlavné rozhranie** má názov *IDevice* (Obr. 2.13).
Zoznam rozhraní, ktoré musí programátor implementovať je tu:

Názov	**Počet ks.**	**Popis**
IDevice	1	hlavné rozhranie (Obr. 2.13)
IDeviceContext	n	kontext zariadenia



Obr. 2.13: *Hlavné rozhranie zariadenia - IDevice*

Zmienim sa len o jednej metóde hlavného rozhrania, a tou je `getNextContext()`. V prípade, že má zariadenie viac kontextov, opakované volanie tejto metódy by malo vracať nasledujúci kontext v zozname všetkých kontextov zariadenia. Ak sa už vyčerpali všetky kontexty, zariadenie by malo vracať buď niektorý z už odovzdaných kontextov (napr. posledný), alebo `null`.

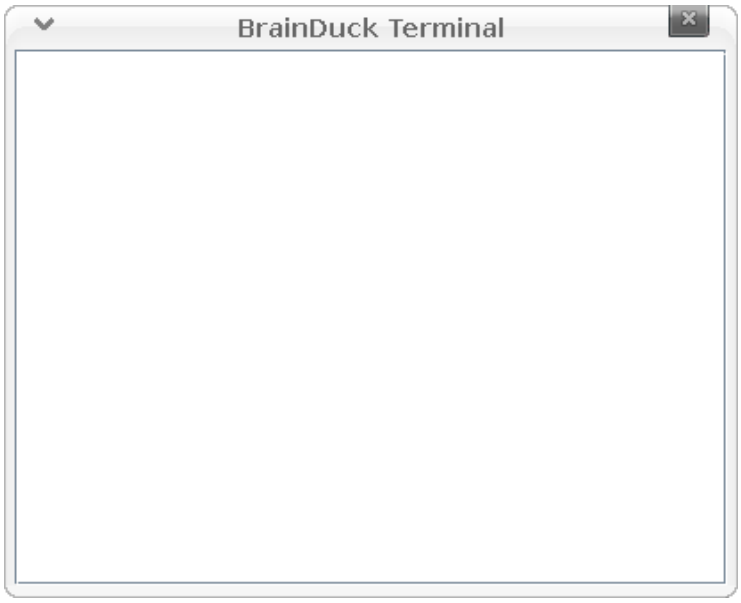
2.8.2 Postup pri tvorbe zariadenia

V tejto časti uvediem implementáciu zariadenia "Terminal", pre vymyslenú architektúru BrainDuck (časť 2.4). Pre projekt nezabudnite pridať externú knižnicu `emu_ifaces.jar` a, keďže budeme využívať kontext procesora BrainCPU, pridajte tiež externú knižnicu `BrainCPU.jar`, čím budeme mať prístup ku žiadanému kontextu.

2.8.2.1 GUI

Začneme netradične, a to tvorbou GUI - grafického rozhrania terminálu. Pôjde o veľmi jednoduché okno typu `javax.swing.JDialog`, ktoré bude obsahovať len jeden komponent `javax.swing.JTextArea`. Tento komponent bude predstavovať okno terminálu (Obr. 2.14). Pre jednoduchosť nebude editovateľné, ale pri požiadavke CPU o vstup z terminálu vyskočí okno, kde uživateľ zadá vstup. Okno bude mať vlastnosť *always-on-top* (vždy navrchu).

Vstup od používateľa si budeme uchovávať do buffra, ktorý sa bude správať ako fronta FIFO. Ak CPU požiada o vstup, tak v prípade, že tento buffer nie je prázdny, vráti sa prvý znak z buffra (a ten z buffra vyberie). V opačnom prípade vyskočí okno žiadajúce používateľa, aby zadal vstupný znak (alebo reťazec), ktorý sa doplní na koniec buffra.



Obr. 2.14: *GUI terminálu*

Pre implementáciu GUI som vytvoril triedu s názvom `BrainTerminalDialog`, ktorú som umiestnil do balíčka `brainterminal.gui`:

```
package brainterminal.gui;

import javax.swing.GroupLayout;
import javax.swing.JDialog;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.WindowConstants;

@SuppressWarnings(”serial”)
public class BrainTerminalDialog extends JDialog {
    private String inputBuffer;

    public BrainTerminalDialog() {
        super();
        inputBuffer = ””;
        initComponents();
    }
```

```
/**
 * Metóda ”zmaže” obrazovku
 */
public void clearScreen() {
    txtTerminal.setText(””);
}

/**
 * Metóda vypíše znak na obrazovku
 *
 * @param c znak, ktorý sa má vypísať
 */
public void putChar(char c) {
    txtTerminal.append(String.valueOf(c));
}

/**
 * Metóda vráti jeden znak zo vstupného
 * buffra. Ak je prázdny, naplní ho.
 *
 * @return znak z buffra
 */
public char getChar() {
    if (inputBuffer.equals(””)) {
        inputBuffer += JOptionPane
            .showInputDialog(”Zadaj vstupný znak”
                + ” (alebo reťazec):”);
    }
    try {
        char c = inputBuffer.charAt(0);
        inputBuffer = inputBuffer.substring(1);
        return c;
    } catch(Exception e) {
        // ak náhodou používateľ zadal prázdny
        // reťazec
        return 0;
    }
}

private void initComponents() {
    JScrollPane scrollTerminal = new JScrollPane();
    txtTerminal = new JTextArea();

    setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
```

```
        setTitle("BrainDuck Terminal");
        setAlwaysOnTop(true);

        txtTerminal.setColumns(20);
        txtTerminal.setEditable(false);
        txtTerminal.setRows(5);
        scrollTerminal.setViewportView(txtTerminal);

        GroupLayout layout = new GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(
            layout.createParallelGroup(GroupLayout.Alignment.LEADING)
            .addComponent(scrollTerminal,
                GroupLayout.DEFAULT_SIZE, 400, Short.MAX_VALUE)
        );
        layout.setVerticalGroup(
            layout.createParallelGroup(GroupLayout.Alignment.LEADING)
            .addComponent(scrollTerminal,
                GroupLayout.DEFAULT_SIZE, 300, Short.MAX_VALUE)
        );
        pack();
    }
    private JTextArea txtTerminal;
}
```

Metódy `getChar()` a `putChar()` budú volané z kontextu zariadenia.

2.8.2.2 Kontext

Kontext nebudeme nijak rozširovať, a hash štandardného kontextu zariadení je `4a0411686e1560c765c1d6ea903a9c5f`. Názov triedy som dal `BrainTerminalContext` a umiestnil som ju do balíčka `brainterminal.impl`:

```
package brainterminal.impl;

import java.util.EventObject;
import brainterminal.gui.BrainTerminalDialog;
import plugins.device.IDeviceContext;

public class BrainTerminalContext implements IDeviceContext {
    private BrainTerminalDialog gui;
    public BrainTerminalContext(BrainTerminalDialog gui) {
        this.gui = gui;
    }
```

```
    @Override
    public Class<?> getDataType() {
        return Short.class;
    }

    @Override
    public Object in(EventObject evt) {
        return (short)gui.getChar();
    }

    @Override
    public void out(EventObject evt, Object val) {
        short s = (Short)val;
        char c = (char)s;
        gui.putChar(c);
    }

    @Override
    public String getHash() {
        return "4a0411686e1560c765c1d6ea903a9c5f";
    }

    @Override
    public String getID() {
        return "brain-terminal-context";
    }

}
```

2.8.2.3 Implementácia hlavného rozhrania

Zostáva už len implementovať hlavné rozhranie. Triedu som nazval `BrainTerminal` a umiestnil som ju do balíčka `brainterminal.impl`:

```
package brainterminal.impl;

import braincpu.interfaces.IBrainCPUContext;
import brainterminal.gui.BrainTerminalDialog;
import plugins.ISettingsHandler;
import plugins.cpu.ICPUContext;
import plugins.device.IDevice;
import plugins.device.IDeviceContext;
import plugins.memory.IMemoryContext;
import runtime.StaticDialogs;
```

```
public class BrainTerminal implements IDevice {
    private final static String BRAIN_CPU_CONTEXT =
        ”d32e73041bf765d98eea1b8664ef6b5e”;
    private long hash;
    private IBrainCPUContext cpu;
    private BrainTerminalContext terminal;
    private BrainTerminalDialog gui;

    public BrainTerminal(Long hash) {
        this.hash = hash;
        gui = new BrainTerminalDialog();
        terminal = new BrainTerminalContext(gui);
    }

    @Override
    public String getTitle() {
        return ”BrainDuck terminal”;
    }

    @Override
    public String getVersion() {
        return ”1.0b1”;
    }

    @Override
    public String getCopyright() {
        return ”\u00A9 Copyright 2009, P. Jakubčo”;
    }

    @Override
    public String getDescription() {
        return ”Simple terminal device for BrainDuck architecture”;
    }

    @Override
    public boolean initialize(ICPUContext cpu, IMemoryContext mem,
            ISettingsHandler settings) {
        if (!(cpu instanceof IBrainCPUContext)
                || !cpu.getHash().equals(BRAIN_CPU_CONTEXT) ) {
            StaticDialogs.showErrorMessage(”BrainTerminal doesn’t”
                + ” support this CPU”);
            return false;
        }
```

```
        this.cpu = (IBrainCPUContext)cpu;
        this.cpu.attachDevice(terminal);
        return true;
    }

    @Override
    public void reset() {
        // zmažeme obrazovku
        gui.clearScreen();
    }

    @Override
    public void destroy() {
        gui.dispose();
        gui = null;
    }

    @Override
    public long getHash() { return hash; }

    @Override
    public IDeviceContext getNextContext() {
        return terminal;
    }

    @Override
    public boolean attachDevice(IDeviceContext device) {
        return true;
    }
    @Override
    public void detachAll() { }

    @Override
    public void showGUI() {
        gui.setVisible(true);
    }
    @Override
    public void showSettings() {
        // nemáme GUI s nastaveniami
    }
}
```

Teraz už len stačí vytvoriť JAR archív zo skompilovaných tried, napr. s názvom `BrainTerminal.jar` a umiestniť ho medzi ostatné zariadenia.

2.9 Spustenie architektúry

Teraz môže čitateľ spustiť platformu ***emuStudio***, vytvoriť novú konfiguráciu a nakresliť abstraktnú schému. Netreba zabudnúť zvoliť náš kompilátor. Veľkosť OP je možné ponechať navrhovaných 65536 bytov. Keď je konfigurácia vytvorená, je už možné si ju odskúšať.

2.9.1 Ukážka "Hello,World!"

Ako ukážku ponúkam program, ktorý vypíše hlášku "Hello,World!" na obrazovku terminálu. Z tejto ukážky je celkom jasný hlavný nedostatok jazyka - keďže inštrukcie nemajú parameter, koľko krát sa majú vykonať, je potrebné ich volať toľko krát manuálne, čo značne zvýši počet riadkov programu. Odporúčam čitateľovi "hrať" sa s touto architektúrou a opraviť jej nedostatky, čo prispeje k lepšiemu pochopeniu princípu tvorby zásuvných modulov.

```
incv
incv
incv
incv
incv
incv
incv
incv
incv
incv ; inicializuje aktuálnu bunku na 10
loop
   inc
   incv
   incv
   incv
   incv
   incv
   incv
   incv
   inc
   incv
   incv
   incv
   incv
   incv
   incv
   incv
   incv
   incv
   incv
```

```
   inc
   incv
   incv
   incv
   inc
   incv
   dec
   dec
   dec
   dec
   decv
endl ; tento cyklus nastaví nasledujúce 4 bunky na
     ; hodnoty 70/100/30/10
inc
incv
incv
print ; 'H'
inc
incv
print ; 'e'
incv
incv
incv
incv
incv
incv
incv
print ; 'l'
print ; 'l'
incv
incv
incv
print ; 'o'
inc
incv
incv
print ; medzera
dec
dec
incv
incv
incv
incv
incv
```

```
incv
incv
incv
incv
incv
incv
incv
incv
incv
incv
print ; 'W'
inc
print ; 'o'
incv
incv
incv
print ; 'r'
decv
decv
decv
decv
decv
decv
print ; 'l'
decv
decv
decv
decv
decv
decv
decv
decv
print ; 'd'
inc
incv
print ; '!'
inc
print ; nový riadok
```

2.9.2 Užitočné fragmenty

V tejto poslednej časti ponúkam pár užitočných fragmentov kódu, ktoré môže využit' pri programovaní v architektúre BrainDuck.

2.9.2.1 Zmazanie bunky

```
loop
  decv
endl
```

Tento jednoduchý fragment nastaví aktuálnu bunku na 0, a to jej iteratívnym dekrementovaním až kým sa nerovná 0.

2.9.2.2 Jednoduchý cyklus

```
load
loop
  print
  load
endl
```

Ide o spojitý cyklus, ktorý číta text zo vstupu z klávesnice a hneď ho aj vypisuje na obrazovku. Predpokladá sa, že na konci vstupu (keď používateľ už nič nezadá), aktuálna bunka (na ktorú ukazuje `P`) je nastavená na 0.

2.9.2.3 Posúvanie smerníka P

```
inc
load
loop
  print
  inc
  load
endl
```

Je to vylepšená verzia predchádzajúceho príkladu, kde sa vstup už ukladá do poľa pre budúce použitie - inkrementovaním registra `P` po každom vstupe.

2.9.2.4 Sčítanie

```
loop
  decv
  inc
  incv
  dec
endl
```

Tento fragment pripočíta hodnotu aktuálnej bunky k hodnote nasledujúcej bunky, ale deštruktívnym spôsobom (prvá bunka je vynulovaná).

```
loop
  decv
  inc
  incv
  inc
  incv
  dec
  dec
endl
inc
inc
loop
  decv
  dec
  dec
  incv
  inc
  inc
endl
```

Táto verzia nezničí pôvodnú bunku, ale potrebuje 1.5 krát viac pamäte (3 byty).

2.9.2.5 Podmienený cyklus

```
load
decv
decv
decv
decv
decv
decv
decv
decv
decv
decv
loop
  decv
  decv
  decv
  decv
  decv
  decv
  decv
  decv
```

```
  decv
  decv
  decv
  decv
  decv
  decv
  decv
  decv
  decv
  decv
  decv
  decv
  decv
  decv
  print
  load
  decv
  decv
  decv
  decv
  decv
  decv
  decv
  decv
  decv
  decv
endl
```

Tento program zo vstupu načíta malé písmená a prevedie ich na veľké písmená, pričom sa ukončí, ak používateľ stlačí "ENTER".

Literatúra

[1] Java Downloads for All Operating System
http://www.java.com/en/download/manual.jsp

[2] JFlex - The Fast Scanner Generator for Java
http://jflex.de/

[3] Cup - LALR Parser Generator for Java
http://www2.cs.tum.edu/projects/cup/

[4] Intel Hexadecimal Object File Format Specification, Revision A, Intel corp., 1988
http://pages.interlog.com/~speff/usefulinfo/Hexfrmt.pdf

[5] S. Šimoňák, P. Jakubčo: *Software based CPU emulation*, Acta Electrotechnica et Informatica, Vol. 8, No. 4, 2008, p50-59

[6] BARRIO, V.M.: Study of the techniques for emulation programming, 2001
http://personals.ac.upc.edu/vmoya/docs/emuprog.pdf

[7] UML ® Resource Page
http://www.uml.org/#UML2.0

[8] Bruce Hutton: Computer Language Implementation Lecture Notes, 2006
http://www.cs.auckland.ac.nz/~bruce-h/lectures/

[9] Brainfuck
http://en.wikipedia.org/wiki/Brainfuck