VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

POKROČILÝ NÁSTROJ PRO MONITOROVÁNÍ ORACLE DATABÁZE

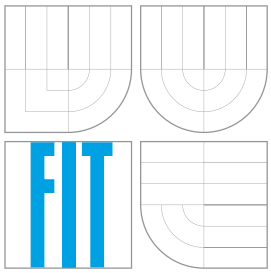
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. DAVID MIKULKA

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

POKROČILÝ NÁSTROJ PRO MONITOROVÁNÍ ORACLE DATABÁZE

ADVANCED ORACLE DATABASE MONITORING TOOL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. DAVID MIKULKA
AUTHOR

VEDOUCÍ PRÁCE — Doc. Dr. Ing. DUŠAN KOLÁŘ
SUPERVISOR

BRNO 2010

Zadání práce

1. Seznamte se s možnostmi monitoringu Oracle Databáze 10g a 11g, zejména AWR a STATSPACK.
2. Analyzujte interní katalogy Oracle DB, které uchovávají statistiky a informace o běžící instanci, včetně historie (AWR snapshots).
3. Na základě analýzy udělejte návrh programu, kterým bude možné, v člověku čitelné podobě (výpisy, příp. grafy), zobrazit aktuální stav databázové instance. Návrh konzultujte s vedoucím.
4. Implementujte navržené řešení v jazyce C++, C# nebo Java.
5. Po konzultaci s vedoucím otestujte implementovaný nástroj na produkční instanci a navrhněte, co by se mělo zlepšit, dle výsledků získaných z nástroje.
6. Zhodnoťte přínos, výhody a nevýhody vámi implementovaného řešení oproti jiným nástrojům komerční sféry.

Abstrakt

Tato diplomová práce popisuje možnosti monitorování Oracle Databáze 10g a 11g. Seznamuje s nástroji použitelnými pro monitorování a popisuje interní katalogy databáze, které uchovávají statistiky a informace o běžících databázových instancích, včetně jejich historie. Následně popisuje návrh aplikace pro monitorování Oracle databáze, popis její implementace a její zhodnocení a porovnání s ostatními podobnými aplikacemi.

Abstract

This master's thesis describes possibilities of Oracle Database 10g and 11g monitoring. It let the reader know about practical tools for monitoring and describes the database's internal catalogs preserving statistics and the information about running database instances within history. Next, it describes design of an Oracle database monitoring tool, describtion of its implementation and at the end its evaluation and comparison with other similar applications.

Klíčová slova

Oracle Databáze, 10g, 11g, STATSPACK, AWR, ADDM, ASH, snapshot, pohled, tabulka, Wait class, ladění, výkonnost, UML, diagramy, diagram tříd, diagram balíků, diagramy interakce, Java SE, Swing, Modul, Modularita, databáze, Java reflexe, JAR archiv, XML, API, rozhraní

Keywords

Oracle Database, 10g, 11g, STATSPACK, AWR, ADDM, ASH, snapshot, view, table, Wait class, tuning, performance, UML, diagrams, class diagram, package diagram, diagrams of interaction, Java SE, Swing, Module, Modularity, database, Java reflection, JAR archive, XML, API, interface

Citace

David Mikulka: Pokročilý nástroj pro monitorování Oracle Databáze, diplomová práce, Brno, FIT VUT v Brně, 2010

Pokročilý nástroj pro monitorování Oracle Databáze

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Doc. Dr. Ing. Dušana Koláře. Další informace mi poskytl Bc. Stanislav Studený. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
David Mikulka
19. května 2010

Poděkování

Děkuji Bc. Stanislavu Studenému za jeho čas, obětavost, připomínky, cenné rady a informace, které mi při vypracování této diplomové práce poskytl.
Děkuji Doc. Dr. Ing. Dušanu Kolářovi za vedení této diplomové práce a za vstřícnost a cenné rady.

Obsah

Kapitola 1

Úvod

V roce 1977 založili tři inženýři informačních technologií Larry Ellison, Bob Miner a Ed Oates společnost Software Development Laboratories (SDL). Prvním jejich projektem byl speciální programový kód s názvem „Oracle“ pro ústřední zpravodajskou službu USA (CIA). Návrh databáze byl inspirován relačním databázovým systémem, který umožňuje uživateli spravovat data ve více přístupech. Po dokončení tohoto projektu byl vyvinut první relační systém řízení báze dat (SŘBD) pro komerční využití.

Společnost SDL mění svůj název na Relational Software Inc. (RSI). Také sází na strukturovaný dotazovací jazyk (SQL) od firmy IBM a věří, že se stane standardem jazyka pro SŘBD. Dále je vyvíjeno programové vybavení jak pro sálové počítače od společnosti IBM, tak pro minipočítače běžící na 16 nebo 32 bitových procesorech s podporou transakčního zpracování a s novým uživatelsky přívětivým rozhraním pro ovládání databáze nejenom programátory. Společnost znovu mění název, tentokráte na Oracle Systems Corporation. Je vydána databáze Oracle verze 3, napsaná v jazyce C, tudíž je použitelná na jakékoliv platformě s kompilátorem jazyka C [26].

V dalších verzích Oracle databáze jsou implementovány nové vlastnosti jako je konzistence, architektura klient/server, podpora distribuovaného zpracování a další. Následně společnost vydává produkt Oracle7. Tato databáze má rozsáhlou škálu nových výkonných vlastností, administrátorských vylepšení, nové nástroje pro vývoj aplikací a nové bezpečnostní metody. Také obsahuje nové nástroje jako jsou vložené procedury, spouštěče (angl. *trigger*), deklarativní referenční integrita, atd. Oracle databáze se stává nejvýkonnějším a nejvíce rozšířeným databázovým systémem na trhu.

S nástupem internetu a multimédií v oblasti počítačů přichází Oracle s podporou rozsáhlých databází, s novými datovými typy pro ukládání rozsáhlých obrazových a multimediálních dat (BLOB, CLOB, BFILE) a nově také s podporou objektově orientovaných technologií. Také rozšiřuje podporu Java aplikací, HTML[1] pro publikování na internetu, OLTP[2] pro internetové obchodování. Momentálně je na světe Oracle Database 11g, poskytující uživateli ty nejlepší služby v oblasti databázových systémů, čerpající ze třicetileté historie a rozsáhlých zkušeností s tvorbou a provozem databázových systémů.

Postupně se sama databáze stávala složitější a složitější. Aby ovšem zůstal tento databázový systém stále uživatelsky přívětivý, samozřejmě s neustále se zlepšujícími základními funkcemi, a poskytoval co nejlepší nástroje pro ovládání databáze, vyvinul Oracle také nejrůznější podpůrné aplikace. Celkově se systém stal velmi rozsáhlým, což s sebou nese také

[1] HTML – HyperText Markup Language; značkovací jazyk pro prezentaci na webu.

[2] OLTP – OnLine Transaction Processing. Technologie uložení dat v databázi pro jednoduché operace nad nimi v mnohauživatelském prostředí.

riziko snižování rychlosti databáze. Proto musely vzniknout také prostředky k ladění databáze, které pomáhají zajistit spolehlivý a bezchybný chod databázového systému Oracle.

Mezi takové prostředky patří například ty, které provádějí diagnostiku činnosti samotné databáze. Např. kolik sezení bylo za určitou dobu inicializováno, jaké byly v jednotlivých sezeních prováděny SQL dotazy, jak dlouho trvaly, kam všude přistupovaly, a další informace o jednotlivých sezeních, které jsou velmi užitečné pro zjištění nesprávně fungující databáze.

Cílem této diplomové práce je aplikace, která bude vhodným způsobem sbírat a zobrazovat zmíněné statistiky. Abychom mohli takovou aplikaci vytvořit, je třeba znát detailně nástroje, které se pro zaznamenávání a diagnostiku interních katalogů databáze používají. Stejně tak je ale důležité znát schéma interních katalogů samotných, abychom věděli, kde co máme hledat.

V kapitole 2 nejprve provedeme analýzu interních katalogů Oracle databáze spolu s prověřením možností monitorování databáze. Dále si v kapitole 2.2 popíšeme mechanismus *Wait Interface* , který je nejvíce používaným prostředkem pro získávání informací o databázové instanci. V dalších kapitolách potom také nástroje pro získání informací z tohoto mechanismu, konkrétně v kapitole 2.3 nástroj STATSPACK a AWR v kapitole 2.5.

Následně, v kapitole 3 vytvoříme návrh samotné aplikace. Ten bude obsahovat všechny důležité kroky návrhu od neformální specifikace, plánování projektu až po diagram tříd a diagramy interakce. Poté provedeme implementaci navržené aplikace (kapitola 4). Otestujeme implementovanou aplikaci na produkční instanci databáze Oracle (kapitola 6) a na závěr zhodnotíme její přínos.

Kapitola 2

Analýza Oracle databáze

Analýzou Oracle databáze rozumíme zjištění používaných a dostupných nástrojů, které slouží pro monitorování databáze. Dále nastudování struktury interních katalogů, jež databáze používá pro ukládání statistických informací o sobě samé, o svém provozu.

2.1 Smysl a možnosti ladění Oracle databáze

Oracle databáze jsou velmi rozsáhlé, komplikované a vyžadující obrovskou péči a údržbu. Zvlášť když malá databáze za dobu svého používání vyroste v obrovskou a složitou databázi, když proces zpracování konkrétního dotazu jindy prováděný v adekvátním čase trvá později o moc déle z důvodu dlouhé doby, po kterou je databáze využívána. Tyto situace nastávají díky zvyšujícím se požadavkům na provádění akcí databází stejně jako zvyšující se množství dat, které musí systém prohledat pro úspěšné provedení požadované operace.

2.1.1 Proaktivní a reaktivní ladění databází

Ladění databází můžeme rozdělit na dva typy. Na proaktivní a reaktivní.

Proaktivní ladění zahrnuje návrh a vývoj struktury výkonnosti pro databázový systém již během raného stádia implementace databáze. To zahrnuje výběr technického vybavení, plánování a výkonnosti a kapacity databáze, výběr systému uložení dat, konfigurace V/V podsystému, atd. Reaktivní ladění naopak zahrnuje hodnocení výkonnosti databáze, odstraňování poruch, ladění a dolaďování Oracle prostředí. Je to ladění, které se provádí až v době, kdy vznikne nějaký problém.

Cílem ladění databázového systému je tedy provedení takových akcí, které zajistí lepší běh databáze než tomu bylo doposud. Můžeme to například srovnat s prováděním údržby a vylepšením automobilu, aby tento jezdil efektivněji s nižší spotřebou a dosahoval větší rychlosti. K tomu nám slouží daná pravidla, jako je kupř. pravidelná výměna motorového oleje, vzduchového filtru, brzdové kapaliny a destiček... V databázi rutinní údržba znamená operace jako je analýza nejrůznějších statistik, obnovení indexů, pokud je to třeba, atd.

Pro ladění databází je používáno mnoho různých postupů, které byly vyvinuty na základě testování a analýzy výsledků. Jeden z prostředků pro ladění databází jsou tzv. ukazatele (angl. *ratios*). Tyto vstupují do hry s porovnáváním rozdílu v hledání dat typicky uložených v paměti a dat, která jsou uložena na disku a jejichž zobrazení vyžaduje V/V operaci. Kupř. SQL dotaz může procházet 10 000 datových bloků. Pokud 9 000 z nich může být nalezeno ve vyrovnávací paměti, potom 1 000 bloků musí být přečteno z disku.

Daný dotaz má tedy ukazatel čtení z vyrovnávací paměti (angl. „*buffer cache hit ratio*“) roven 90 % [45].

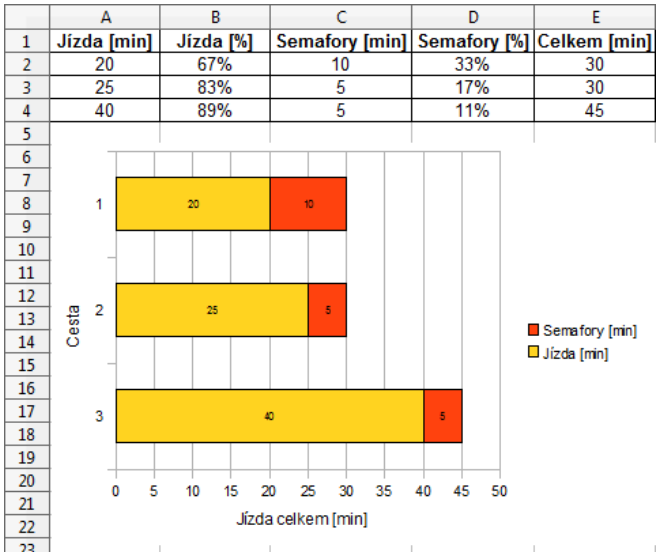
Dalším prostředkem je mechanismus zvaný *Wait Interface*. Základní příručkou, která uvozuje tuto techniku je, že cokoliv, na čem databáze tráví nejvíce času čekáním, je úzké místo databáze (místo, které způsobuje zpomalení). Opravením těchto zúžení zvýšíme výkon databáze, dokud nenarazíme na další, která jsou, podle definice, větší než ta stávající, takže výkonnost musí být vzrůstající.

2.1.2 Ukazatele

Ladění databází bylo vždy velmi důležitou částí náplně práce administrátora databáze, na stejné úrovni jako třeba zálohování nebo obnovení databáze. Často na něj bylo nahlíženo jako na tzv. černou skříňku. Tato představa pochází z myšlenek mnoha expertů a knih o ladění databází.

Princip používající ukazatele je minimálně z části založen na racionálním uvažování. Například je zřejmé, že je rychlejší číst data z paměti RAM než z disku. Proto jestliže mnoho dat je čteno právě z disku, může být problémem způsobujícím pokles výkonnosti databáze malá velikost vyrovnávací paměti, která pak nemůže pojmout dostatečný počet dat. Řešením tedy může být zvětšení kapacity vyrovnávací paměti.

Pokusíme se pochopit ukazatele na příkladu z oblasti nedatabázové, a sice na ilustraci cesty do práce osobním automobilem. Je patrné, že se rychleji dostaneme do práce, pokud nemusíme stát na semaforech. Jedna z možností je zkrácení doby projetím semaforů na červenou, což je ale velmi nebezpečné, nebo zvolit jinou trasu, kde se semaforům co nejvíce vyhneme.



Obrázek 2.1: Příklad ukazatele.

Předpokládejme, že čekání na semaforu charakterizuje ukazatel čekání na semaforu (UCS), přičemž naším cílem je, aby hodnota tohoto ukazatele byla co nejmenší. Dále předpokládáme, že čekání na semaforech prodlužuje dobu cesty do práce, čili čím menší ukazatel bude, tím dříve budeme v práci.

Z výše uvedeného obrázku 2.1 je patrné, že pokud jedeme do práce cestou 1, strávíme 67 % času jízdou a 33 % čekáním na semaforech. Pokud zvolíme cestu 2, strávíme čekáním pouze 17 % času místo původních třiatřiceti. Můžeme zpozorovat, že UCS se zlepšil, nicméně doba cesty do práce se nezměnila a zároveň více minut stráveného na cestě znamená větší spotřebu paliva a delší urazenou vzdálenost. Čili stále jsme nenalezli objektivně výhodnější cestu do práce. Zvolíme tedy cestu 3, na které pozorujeme další zlepšení UCS na 11 %, což je nejlepší výsledek ze všech tří uvedených. Cesta touto trasou ale trvá o 50 % delší dobu a je výrazně delší než předchozí dvě.

Tento příklad nám jasně ukázal nevýhody ukazatelů. Ačkoliv jsme UCS neustále zlepšovali, neznamenalo to automaticky také zrychlení cesty do práce. Vylepšení konkrétní metriky tedy nemusí nutně znamenat zlepšení celého problému.

Další nevýhodou ukazatelů je, že jejich věrohodnost je závislá na tom, jak dlouho je databáze v provozu. Pokud je databáze nově zprovozněna, žádná data nejsou ve vyrovnávací paměti uložena dokud nejsou poprvé čtena. Proto dokud není databáze naplněna po nějakou dobu, ukazatele typu „*buffer cache hit ratio*“ a další jsou nerelevantní.

2.1.3 Mechanismus *Wait Interface*

Wait Interface je mechanismus používaný v posledních letech. Díky němu administrátor zjišťuje kde se spotřebovává kolik času, zda je tento čas využit pro čekání nebo pro práci.

Konferenci uspořádanou pro uživatele databází s tématem *Wait Interface* ovládla skepse nad tímto prostředkem, nicméně pochopení praktických ukázek aplikací bylo klíčem k porozumění tomuto přístupu. *Wait Interface* je naprosto odlišný princip vzhledem k ukazatelům. Například proč bychom měli zvýšit velikost vyrovnávací paměti, když dlouhá časová odezva databáze je způsobena přetíženým řadičem disku nebo neefektivním SQL dotazem?

Mechanismus *Wait Interface* byl uveden ve verzi Oracle7 a s každou další verzí byl spolehlivější a robustnější. Časem si získal čím dál větší pozornost a s tím také lepší porozumění jak jej použít pro ladění databází. Sestává z dynamicky generovaných pohledů.

Naše aplikace bude pracovat na základě získávání informací právě z *Wait Interface*, proto se dále v textu budeme zabývat právě jím a nástroji, které z něj umí dolovat požadované informace.

2.2 Mechanismus *Wait Interface*

Wait Interface je název mechanismu v Oracle databázi, který využívají administrátoři ke zkoumání vnitřního chování databáze. Umožňuje jim zjistit co které komponenty dělají, když čekají. I když nejrůznější další informace lze získat z jiných databázových pohledů (angl. *view* – pro zjednodušení budeme dále používat zkráceného označení „pohled“ přičemž bude myšleno databázový), tři stěžejní pohledy jsou [46]:

- *v$system_event* — Zobrazuje události systému (V/V, síť), které způsobují čekání.
- *v$session_event* — Kumulativně zaznamenává čas, který každé sezení strávilo čekáním na konkrétní událost.
- *v$session_wait* — Zobrazuje informace o aktivních sezeních, které jsou ve stavu čekání.

Ještě než si zde budeme ukazovat některé SQL dotazy nebo jejich výsledky, je potřeba předeslat, že musí být mechanismus *Wait Interface* v databázi spuštěn, jinak nebudou žádné statistiky zaznamenávány. Spuštění se provádí nastavením parametru `timed-statistics` na hodnotu `TRUE` v souboru *init.ora* nebo *spfile* nebo přímo dotazem na databázi:

```
ALTER SYSTEM SET TIMED_STATISTICS = TRUE;
```

Zapnutí tohoto mechanismu má sice určitý dopad na databázi, nicméně ten je mnohonásobně přebit nespornou výhodou a využitím mechanismu *Wait Interface* při ladění databáze. Do verze Oracle9i bylo výchozí nastavení inicializováno na hodnotu `FALSE`, od této verze včetně je již mechanismus *Wait Interface* ve výchozím nastavení spuštěn.

2.2.1 Pohled v$system_event

```
SQL > desc v$system_event
Name                 Null?    Type
-------------------- -------- --------------
 EVENT                          VARCHAR2(64)
 TOTAL_WAITS                    NUMBER
 TOTAL_TIMEOUTS                 NUMBER
 TIME_WAITED                    NUMBER
 AVERAGE_WAIT                   NUMBER
TIME_WAITED_MICRO             NUMBER
```

Obrázek 2.2: Výpis nejdůležitějších polí pohledu *v$system_event*.

Detailní popis struktury jednoho z nejdůležitějších pohledů mechanismu *Wait Interface* podle obrázku 2.2:

- EVENT — Název události.
- TOTAL_WAITS — Celkový počet čekání, která zapříčinila daná událost.

- TIME_WAITED — Celkový čas, který byl stráven čekáním na tuto událost v *centisekundách.*
- AVERAGE_WAIT — Průměrná doba čekání na danou událost – TIME_WAITED / TOTAL_WAITS.
- TIME_WAITED_MICRO — Čas TIME_WAITED v mikrosekundách, pro lepší zpracování.

Ne všechny události, které způsobují čekání databáze, shromážděné v tomto pohledu, jsou hodny naší pozornosti. Některé z nich, které se v databázi vyskytují, mohou být ignorovány, pokud se díváme na celý systém globálně. Tyto obyčejně odkazují na tzv. nicnedělající (klidové) události, které se vyskytují pouze pokud databáze je ve stavu nečinnosti nebo čekání. Teoreticky tyto události nijak nezabraňují v dokončení úloh a nemohou tedy být potenciálními potížemi výkonnosti databáze. Podrobněji se těmito událostmi budeme zabývat v kapitole 2.2.5.

Nicméně je třeba si také zapamatovat, že ačkoliv jsou někdy klidové události ošetřeny jako čekající události nebo jako nedůležité, je výhodné zpřesnit jejich měření, aby bylo možné rozhodnout, zda jsou opravdu čekajícími nebo mohou ukazovat na výkonnostní riziko.

Musíme mít také na paměti, že pohled *v$system_event* je kumulativní! Jednoduše řečeno, shromažďuje údaje o počtu a časech jednotlivých událostí jak nastaly od doby, kdy byla daná instance spuštěna. To znamená, že když běží databáze už např. jeden rok, tento pohled obsahuje statistiky nastřádané za celý rok. Tato skutečnost rapidně zmenšuje šanci nalézt problémy databáze, které měly strašlivé důsledky na její chod po dobu pouze několika dnů, v kontextu celých 365 dní jejího provozu.

Události způsobující určitý problém trvají většinou krátký časový úsek a pak už se dále nemusejí znovu vyskytnout. Proto by se mělo na tento pohled nahlížet jako na krátké časové úseky nebo rozdíly mezi dvěma krátkými snímky. Tento pohled obsahuje hodnotné informace o tom, co celá databáze dělá a kde tráví kolik času, to je ale jenom počáteční místo při ladění databáze. Je třeba zabřednout hlouběji a zkoumat přesné příčiny problémů.

2.2.2 Pohled v$session_event

```
SQL > desc v$session_event
Name                  Null?    Type
--------------------- -------- --------------
 SID                             NUMBER
 EVENT                           VARCHAR2(64)
 TOTAL_WAITS                     NUMBER
 TOTAL_TIMEOUTS                  NUMBER
 TIME_WAITED                     NUMBER
 AVERAGE_WAIT                    NUMBER
 MAX_WAIT                        NUMBER
 TIME_WAITED_MICRO               NUMBER
```

Obrázek 2.3: Výpis nejdůležitějších polí pohledu *v$session_event.*

Struktura tohoto pohledu je velmi podobná jako u pohledu *v$system_event*. Jsou zde akorát dvě nová pole, SID a MAX_WAIT – obrázek 2.3.

SID je identifikátor sezení, který umožňuje spojení mezi aktuálním sezení a daným uživatelským jménem ve spojení s pohledem *v$session*. Díky tomuto spojení je jednodušší vystopovat uživatele, kteří se nyní přihlašují pod svým uživatelským jménem a obecně neznají svoje SID.

MAX_WAIT je maximální čas ve stovkách sekund, který daná událost v tomto sezení spotřebovala, což může být vhodné např. při situaci, kdy pozorný uživatel oznámí, že jeho sezení občas „tuhne“. Potom je administrátor schopen porovnat časy MAX_WAIT a AVERAGE_WAIT aby zjistil, zda je zde nějaký problém, což lze zjistit jednoduše tehdy, kdy čas MAX_WAIT bude mnohem větší než AVERAGE_WAIT. Pohled *v$session_event* lze výhodně použít i pro porovnání několika sezení, zda se zkoumané statistiky odlišují od ostatních, atd.

2.2.3 Pohled v$session_wait

Často potřebujeme mít k dispozici také aktuální informace o běhu sezení v reálném čase. Ty nám poskytuje tento pohled. Je obnovován záznamovým procesem každé 3 sekundy.

```
SQL > desc v$session_wait
Name                 Null?    Type
-------------------- -------- --------------
 SID                           NUMBER
 SEQ#                          NUMBER
 EVENT                         VARCHAR2(64)
 P1TEXT                        VARCHAR2(64)
 P1                            NUMBER
 P1RAW                         RAW(4)
 P2TEXT                        VARCHAR2(64)
 P2                            NUMBER
 P2RAW                         RAW(4)
 P3TEXT                        VARCHAR2(64)
 P3                            NUMBER
 P3RAW                         RAW(4)
 WAIT_TIME                     NUMBER
 SECONDS_IN_WAIT               NUMBER
 STATE                         VARCHAR2(64)
```

Obrázek 2.4: Výpis nejdůležitějších polí pohledu *v$session_wait*.

Podrobný popis nejdůležitějších parametrů tohoto pohledu:

- P1TEXT — Popis prvního parametru události.
- P1 — Dekadická hodnota prvního parametru události.
- P2TEXT — Popis druhého parametru události.
- P2 — Dekadická hodnota druhého parametru události.

- P3TEXT — Popis třetího parametru události.
- P3 — Dekadická hodnota třetího parametru události.
- WAIT_TIME — Jestliže je tato hodnota `0`, sezení právě čeká. Když je hodnota `-1`, doba posledního čekání byla menší než jedna setina sekundy (jedna centisekunda).
- SECONDS_IN_WAIT — Počet sekund mezi aktuálním časem a časem, od kterého sezení čeká (WAIT_TIME=0).
- STATE — Reprezentuje současný stav čekání nebo stav, ve kterém se sezení nachází.
- WAITING — Značí, že sezení právě čeká na nějaký zdroj.
    - WAITED UNKNOWN TIME — Doba posledního čekání je neznámá, ale o čekání šlo.
    - WAITED SHORT TIME — Doba čekání byla menší než setina sekundy.
    - WAITED KNOWN TIME — Doba čekání odpovídá hodnotě parametru WAIT_TIME.

Pole mající v názvu přídavek RAW značí hexadecimální hodnoty ekvivalentní hodnotám dekadickým (parametry P1, P2, P3).

Např. na základě zobrazených informací, které ukazují, že čekání nepřetržitě vyvolává zprávu „*SQL*Net*“ s parametrem `WAIT_TIME=0`, můžeme usoudit, že na vině jsou problémy se sítí.

2.2.4 Další důležité pohledy

Mechanismus *Wait Interface* lze symbolicky rozdělit na úrovně podle důležitosti jednotlivých pohledů. Tři výše popsané pohledy jsou základní. Lze je označit za úroveň 0. Do dalších úrovní patří ostatní pohledy, jež dále specifikují informace o běhu databáze.

Pohled v$event_name

Tento pohled je tzv. číselník, který můžeme použít kdykoliv při prohlížení pohledu *v$session_wait*. Obsahuje informace o událostech samotných a co pro každou z nich znamenají parametry P1, P2 a P3.

Pohled v$filestat

Jedna oblast, ze které může vzejít problém ve výkonnosti databáze jsou V/V operace. Tento pohled stopuje čas, jak dlouho trvá dokončení V/V operace v milisekundách (ms). Zaznamenává poslední operaci, nejhorší, nejlepší a průměrnou dobu trvání dané operace. Také zaznamenává počet čtení, zapisů a počet bloků souboru.

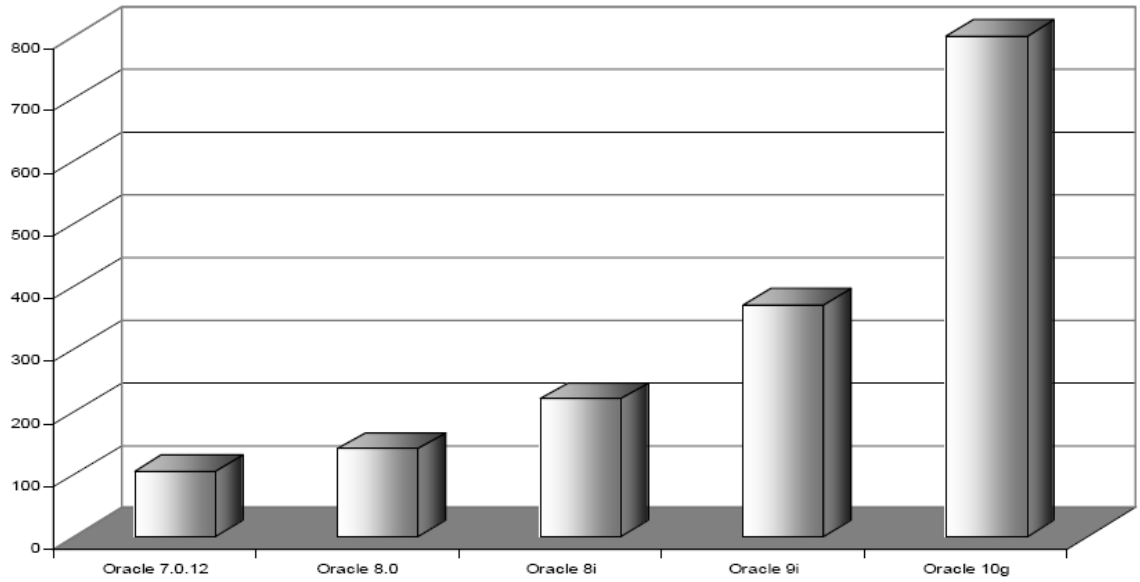
Pohled v$tempstat

Struktura tohoto pohledu je totožná se strukturou pohledu *v$filestat*. Rozdíl je však ten, že pohled *v$tempstat* obsahuje informace o dočasných souborech. Dočasné soubory jsou používány pro určité operace, např. řazení, které pracují s velkým množstvím dat na to, aby se všechna data vlezla do paměti.

Pohled v$waitstat

Pohled *v$waitstat* soustřeďuje statistiky soupeření o bloky dat pro každou třídu datových bloků. Díky tomu je možné vidět, jestli pohyb bloků z a do vyrovnávací paměti způsobuje zpomalení databáze. Když jsou problémy s vyrovnávací pamětí signalizovány jinými komponentami mechanismu *Wait Interface*, poskytuje tento pohled, na úrovni celé databáze, informaci o tom, na co se vlastně čeká.

2.2.5 Třídy čekání



Obrázek 2.5: Počet čakacích událostí v různých verzích Oracle databáze [13].

Pokročilí administrátoři seskupili specifické události, ve kterých databáze čeká, do tříd. Mechanismus *Wait Interface* ve verzi Oracle10g tyto třídy obsahuje. V dřívějších verzích nebylo třeba tyto třídy stanovit, protože událostí nebylo takové množství, jako je právě ve verzi Oracle10g a výše, viz. obrázek 2.5. Deset tříd, které jsou v Oracle10g definovány [47]:

- Cluster
- Commit
- Concurrency
- Configuration
- Idle
- Network
- Other
- Sheduler
- System I/O

- User I/O

Každá událost je zařazena do některé z těchto tříd. Každá třída tedy sdružuje čekací události podle oblasti, ve které k danému čekání dochází. Díky tomuto rozčlenění událostí do tříd čekání je administrátor při zjištění dlouhého čekání některé události schopen rychle zjistit, v které oblasti se problém nachází a zaměřit se přímo na tuto oblast. Tím výrazně klesá doba nalezení a zjištění problému databáze, jelikož správně klasifikujeme zdroj čekající události.

Nebezpečí nesprávně klasifikovaných událostí však stále trvá. Např. „*SQL*Net*“ události byly obyčejně klasifikovány jako klidové události, ale v mnoha případech reprezentují velký problém, který potřebuje být správně zařazen, aby se problém výkonnosti databáze zmírnil. Navzdory tomuto nebezpečí, třídy čekání vzbuzují velkou naději, že ladění výkonnosti databází bude pro nezkušené administrátory jednodušší.

Pohled v$system_wait_class

Tento pohled umožňuje administrátorovi zobrazit souhrnné statistiky pro každou registrovanou třídu čekání v celé databázové instanci. S ním související pohled *v$session_wait_class* zaznamenává stejné statistiky vázané na uživatele pomocí SID a sériového čísla. Pro pochopení může být vhodné si myslet o těchto pohledech, že se jedná o souhrnné pohledy z pohledů *v$system_event* a *v$session_wait*, které shrnují dostupné hodnoty a seskupují je podle předdefinovaných tříd.

Pohled v$event_histogram

Tento pohled sděluje informace o rozdělení tříd čekání. Povoluje porovnání na základě frekvence a „hutnosti“ čekacích událostí. Toto administrátorovi umožňuje vidět, zda normální čekací doba je vysoká nebo jestli je zde pár „bludných“ událostí, kde čekací doba byla závažná. Tento typ analýzy je vhodný pro případ, že je potřeba zjistit rozsah a velikost daného problému.

Pohled v$eventmetric

Pohled *v$eventmetric* zobrazuje hodnoty metrik čekacích událostí pro určitý, velmi krátký, časový interval. Typicky ukládá informace z několika posledních minut. Můžeme tedy usoudit, že tento pohled je shodný s pohledem *v$session_wait* v tom smyslu, že uchovává informace o čekání z nedávné minulosti. Nicméně jsou rozdílné v tom, že tento pohled ukládá statistiky napříč celým systémem a místo aby zaznamenával aktuální čekání, ukládá přibližně jednu minutu starou historii.

Přirozenost tohoto prostředku dělá analýzu nedostatků databáze preciznější oproti ukazatelům. Analýza zobrazuje konkrétní oblast zájmu, což dovoluje přesnější výsledek. Zatímco ukazatele zobrazují detaily aktivity databáze a shrnují je do jednotlivých metrik, mechanismus *Wait Interface* používá detaily, aby administrátorovi ukázal, co konkrétně databázi zpomaluje. Měří, kde databáze tráví čas čekáním, což dovoluje zaměřit se na oblasti, které se momentálně chovají jako překážky. Právě tohle dělá ladění databází efektivnějším.

Naopak omezujícím faktorem může být obrovské množství dat, které mechanismus *Wait Interface* sbírá. Administrátor se může při hledání chyb a jejich opravení zaměřit nesprávným směrem. Stejně tak může být problém zvolit vhodný časový úsek pro správné odchycení daného problému. V neposlední řadě je také potřeba znát prostředky, které mohou

být použity pro odstranění chyb. Jedná se o konzolové skripty, např. Perl nebo Korn, dále také PL/SQL procedury nebo samotné SQL dotazy. Toto může být bráno svým způsobem také jako omezení, jelikož tohle může dělat pouze programátor, nikoliv běžný uživatel.

2.3 Nástroj STATSPACK

STATSPACK je efektivní nástroj využívající pohledy mechanismu *Wait Interface*. Mezi výhody tohoto nástroje patří vestavěné pořizování snímků chodu databáze a také např. hotový dotaz ve formě zprávy nazvaný *spreport.sql*.

2.3.1 Shromažďování dat a jeho frekvence

Získávání dat pomocí nástroje STATSPACK zahrnuje tzv. pořizování snímků. Jedná se o druh snímků, reprezentující informace uložené v systémových pohledech a tabulkách v okamžiku, kdy byl snímek pořízen.

Pro pořízení snímků se stačí k databázi přihlásit jako uživatel `PERFSTAT` a spustit příslušnou proceduru. Tato procedura má název *snap* a je umístěna v balíčku *statspack*. Výsledkem jsou data sebraná ze systémových pohledů a tabulek, uložená do již vytvořeného úložiště nástroje STATSPACK. Výstup při zadání příkazu pro vytvoření snímku do příkazové řádky databáze je následující:

```
SQL > execute statspack.snap;

PL/SQL procedure succesfully completed.
```

Po pořízení prvního snímku necháme uplynout nějaký čas a můžeme pořídit další. Pokud již máme aspoň dva snímkmy, mezi kterými nebyla databáze ve stavu nečinnosti, STATSPACK připraví zprávu o aktivitě databáze během daného intervalu. Časový interval samotný však není vůbec lehké určit a bývá předmětem diskuzí mezi administrátory. Ideálně by měl být tento časový úsek dostatečně dlouhý na to, aby byly statistiky, které je potřeba pro odhalení daného problému, kompletně sesbírány, ale na druhou stranu nesmí být ani příliš dlouhý, aby nastřádaná data daný problém nezatemnila. Cílem je, aby hledaná statistika nezapadla do průměru.

Frekvence pořizování snímků je možné nastavit od intervalu jedné minuty až po měsíční periodu. Vtip je ve zvolení co nejlepšího intervalu. Běžně se pro reaktivní ladění databází používají intervaly od 5 do 15 minut.

Ikdyž mohou být snímky pořizovány častěji než je vhodné, není třeba používat pouze statistiky mezi těmito intervaly, ale lze použít i větší časový rámec. Např. pokud pořizujeme snímky každých pět minut po dobu více než jedné hodiny a když nelze vyčíst ze dvou snímků s časovým rozdílem pět minut požadováné informace, je možné se podívat na dva, mezi nimiž je rozdíl třeba třicet minut. Z toho je patrné, že je výhodnější snímky databáze pořizovat spíše častěji než méně často.

2.3.2 Vytváření zpráv

Pro vytváření zpráv na základě snímků vygenerovaných nástrojem STATSPACK existuje v Oracle databázi skript s názvem *spreport.sql*. Pokud je použit interaktivně, vyžádá si, aby uživatel zadal identifikátor startovního a koncového snímku a potom se vytvoří zpráva zobrazující rozdíly v hodnotách pro mnohé pohledy, které jsou součástí mechanismu *Wait Interface*. Díky používání skriptu *spreport.sql* je možné rychle zjistit, co se s databází děje.

Navíc je možné využít analyzátoru zpráv. Ten vytvořil Anjo Kolk, vědec z Oracle Corp. a zpřístupnil jej na webové stránce http://www.oraperf.com/. Jakmile si uživatel nechá

vygenerovat zprávu o stavu databáze, nahraje ji na uvedenou internetovou stránku a ta z ní vygeneruje analýzu. Ta umožňuje rychlé vyhodnocení možných příčin problémů výkonnosti databáze. Další možností je provést ruční analýzu pomocí nejrůznějších skriptů.

Jsou také oblasti, ze kterých neumí STATSPACK zachytávat data. Statistiky jako je volná paměť či využití CPU jsou dva body, které mohou být důležitými ukazateli na potenciální problém. S použitím nástrojů jako je *vmstat* může administrátor nahlížet na tyto statistiky, ale nemůže se posunout zpět v čase a zjistit, o co se jednalo. Právě zde jsou možnosti pro rozšíření nástroje STATSPACK, tedy když bude vytvořen snímek databáze, bude obsahovat informace z operačního systému a ukládat je do jiné tabulky. Získávání dodatkových informací tohoto zaměření je správná cesta k obohacení nástroje STATSPACK k odstranění některé jeho slabosti.

2.3.3 Úrovně sběru dat

STATSPACK rozeznává dva typy nastavení sběru dat. Jeden typ nastavení je *level* a druhý *threshold*. Parametr *level* určuje úroveň sběru dat, zatímco parametr *threshold* se chová jako filtr pro ukládání SQL dotazů do tabulky *stat$sql_summary*.

Parametr *Level*

Snímky, které jsou nástrojem STATSPACK pořizovány, mohou být shromažďovány v různých úrovních[24]. Každá vyšší úroveň sbírá vyšší počet informací o databázi. S každým vyšším označením úrovně, tato zahrnuje všechny údaje z předchozích úrovní a přidává něco dalšího. Seznam úrovní je v následující tabulce.

Úroveň	Sbírané statistiky
0	Obecné výkonnostní statistiky
5	Přídavek: SQL dotazy
6	Přídavek: SQL plány a jejich použití
7	Přídavek: Statistiky na úrovni segmentů
10	Přídavek: Otcovské a synovské zámky

Tabulka 2.1: Seznam úrovní sběru dat nástroje STATSPACK.

Výchozí úrovní sběru dat je úroveň 5. Je ale možné použít balíček pro změnu výchozího nastavení nebo pro speciální případ. Toto je možné provést funkcí *statspack.snap*. Příklady změny nastavení úrovně je možné nalézt také ve zdroji [27]. Podrobnější popis úrovní nalezneme na internetu [25].

Parametr *Threshold*

Parametr „*práh*“ a jeho úrovně se týkají pouze SQL dotazů, které jsou uloženy v tabulce *stat$sql_summary*. Tato tabulka se může rychle stát nejrozsáhlejší, protože každý snímek v ní může shromáždit několik tisíc řádků, jeden pro každý SQL dotaz, který byl v tu dobu uložen ve vyrovnávací paměti.

Úrovně prahů jsou uloženy v tabulce *stat$statspack_parameter*. Základních prahy jsou:

- **executions_th** — Počet vykonání SQL dotazu (výchozí hodnota je 100)

- **disk_reads_th** — Počet čtení z disku při vykonání SQL dotazu (výchozí hodnota je 1 000)
- **parse_calls_th** — Počet volání syntaktického analyzátoru při vykonání SQL dotazu (výchozí hodnota je 1 000)
- **buffer_gets_th** — Počet získání dat z vyrovnávací paměti při vykonání SQL dotazu (výchozí hodnota je 10 000)

Každý SQL dotaz bude hodnocen vůči všem těmto prahům a bude do tabulky *stat$sql_summary* vložen, pokud kterýkoliv z těchto prahů bude překročen. Bude tam vložen, pokud bude splňovat kterýkoliv z těchto prahů, nikoliv každý – mezi nimi není relace logický součin, nýbrž logický součet. Toto opatření je zavedeno z důvodu kontroly extrémního zvýšení počtu řádků v tabulce *stat$sql_summary*, který velmi aktivní databáze s mnoha sty SQL dotazy ve vyrovnávací paměti provádí.

Výchozí nastavení prahů je možné změnit voláním funkce *statspack.modify_statspack_parameter*. Příkazem

```
SQL > execute statspack.modify_statspack_parameter -
        (i_buffer_gets_th=>100000, i_disk_reads_th=>100000);
```

změníme výchozí nastavení prahů „*disk_reads*“ a „*buffer_gets*“ na hodnotu 100 000. Ve všech následujících snímcích budou ukládány pouze SQL dotazy, které překročí tyto dva prahy.

Nástroj STATSPACK je vhodný jak pro reaktivní, tak i proaktivní ladění databáze. Jeho použití je jednoduché díky existujících knihovním funkcím. Lze jej také vhodně rozšířit na snímání pokročilých a rozšířených statistik, které zde doposud zahrnuty nejsou. Nevýhodou, hlavně pro reaktivní ladění je, že se musí instalovat. Data zaznamenává až od okamžiku, kdy byl nainstalován. To je velmi omezující zejména při použití reaktivního ladění databáze, jelikož musíme odstranit problém, který se vyskytl v minulosti. Pokud ale STATSPACK nemáme nainstalovaný, těžko můžeme po jeho instalaci nějaké statistiky z minulosti získat.

2.4 Pokročilé diagnostické nástroje

Od verze Oracle7, kdy byl mechanismus *Wait Interface* do databáze nově zabudován, byl neustále vylepšován, aby informace jím zachytávané byly co nejefektivněji využívány. S příchodem Oracle 10g se vysledované statistiky zlepšily co do kvality, hodnotnosti a použitelnosti. Navíc byla uvedena další vylepšení výkonnosti a nástroje. Nástroje jako je *Active Session History*, *Automatic Database Diagnostic Monitor* a další, dále zvýšily efektivitu ladění Oracle databáze.

Některá z nových rozšíření jsou samostatné produkty oddělené od základní verze Oracle databáze. Nástroje AWR a ADDM jsou licencované – Oracle vyžaduje speciální licenci pro přístup k nim.

Nástroj *Active Session History* (ASH) sestává z dat jednotlivých databázových relací, která jsou snímána každou sekundu a ukládána do kruhové vyrovnávací paměti, která je umístěna v části paměti RAM, která je využívána všemi procesy Oracle databáze.

Každá databázová relace je označena jako aktivní tak dlouho, dokud nečeká na událost, která přísluší do třídy čekaní označené jako *Idle*. Ladění databáze pomocí ASH zahrnuje použití pohledu *v$active_session_history* a tabulek tzv. pracovní schránky, jako je například *wrh$active_session_history*. ASH ukládá nedávnou historii aktivit sezení, včetně událostí čekání, a umožňuje tak provádět analýzu výkonnosti databáze jak nyní, tak i v její nedávné minulosti. Protože je nástroj ASH navržen jako kruhová paměť, dřívěji nashromážděné informace jsou přepsány novými, pokud je třeba.

Dalším pokročilým nástrojem je *Automatic Database Diagnostic Monitor* (ADDM). Používá se pro automatické monitorování a diagnostiku problémů databáze. ADDM monitoruje data uložená v pracovní schránce a generuje doporučení pro vyladění databáze v oblastech jako je využití CPU, správa připojení k databázi, V/V operace, atd.

2.5 Nástroj AWR – Automatic Workload Repository

Pokud jsme chtěli v dřívějších verzích Oracle databáze zaznamenávat data starší než jen pár sekund nebo minut, bylo potřeba použít osobně vytvořené procedury, STATSPACK nebo aplikace třetích stran. S verzí 10g včleňuje do databáze samotné novou funkcionalitu nástroj *Automatic Workload Repository* (AWR), který využívá mnoho funkcí nástroje STATSPACK.

2.5.1 Struktura AWR

Nástroj AWR je součástí samotné databáze Oracle10g a vyšší. To je jedna z jeho výhod oproti nástroji STATSPACK, že se nemusí zvlášť instalovat. AWR automaticky sbírá data hned jak je databáze nainstalovaná a běžící. Narozdíl od nástroje STATSPACK, kde bylo potřeba nastavit frekvenci pořizování snímků a potom nastavit plánovač jejich pořizování, AWR snímá statistiky každých 30 minut. Tento interval může být ale uživatelem změněn. Získaná data jsou v AWR uložena po dobu sedmi dnů a poté jsou automaticky smazána.

Data jsou do AWR ukládána pomocí nových procesů s názvem *MMON* a *MMNL* (*MMON Lite*). Proces *MMON* je, mimo jiné, zodpovědný za zobrazení hlášení, která jsou zobrazena jestliže jsou nalezeny metriky, které překročily své prahové hodnoty.

K přístupu k těmto datům může být také využito nástroje *Oracle Enterpise Manager* (OEM), který má pohodlné uživatelské rozhraní.

AWR sestává z několika tabulek:

- *dba_hist_active_sess_history*
- *dba_hist_baseline*
- *dba_hist_database_instance*
- *dba_hist_snapshot*
- *dba_hist_sql_plan*
- *dba_hist_wr_control*

Struktura tabulky *dba_hist_active_sess_history* je identická se strukturou tabulky *v$active_session_history*, pouze s výjimkou několika nových polí na začátku s detailními informacemi o tom, kdy a kde byla data sesbírána. Tato tabulka reprezentuje data, která jsou

periodicky sbírána z pohledu *v$active_session_history* stejně jako pomocí nástroje STATSPACK.

2.5.2 Vytváření zpráv

Tak jak bylo v kapitole 2.3.2 uvedeno, že STATSPACK generuje zprávy pomocí dotazu *spreport.sql*, tak i AWR poskytuje obdobnou funkcionalitu ve formě skriptů *awrrpt.sql* a *awrrpti.sql*. Skript *awrrpt.sql* je velmi podobný skriptu *spreport.sql*, skript *awrrpti.sql* navíc obsahuje vytváření zpráv pro specifické databáze, když data z více databází jsou ukládána do jednoho úložiště.

Skript *awrrpt.sql* poskytuje informace o časovém rozmezí, které zpráva pokrývá, o hostiteli, instanci, jménu databáze, množině metrik, ukazatelů pro letmé prověření funkčnosti databáze, atd.

2.5.3 Skupina snímků a snímky samotné

Informace o databázi ve formě skupiny snímků mohou být porovnány vzhledem k nějaké jiné skupině snímků, pro zjištění, zda databáze funguje lépe, hůře nebo se nic nezměnilo. Takovéto skupiny snímků se vytvářejí pomocí procedury *dbms_workload_repository.create_baseline*.

Snímky samotné pomáhají administrátorovi vzpomenout si na problémy, které se v databázi přihodily. Snímek poskytuje možnosti, jak si uchovat záznam problémů, které způsobují problémy výkonnosti databáze. Standardně jsou snímky pořizovány každou hodinu a jsou uchovávány po dobu sedmi dnů. Tyto hodnoty však lze změnit pomocí procedury *dbms_workload_repository.modify_snapshot_settings*. Plánování frekvence pořizování snímků je analogické jako u nástroje STATSPACK.

```
DBMS_WORKLOAD_REPOSITORY.MODIFY_SNAPSHOT_SETTINGS(
  retention    IN NUMBER DEFAULT NULL,
  interval     IN NUMBER DEFAULT NULL,
  dbid         IN NUMBER DEFAULT NULL);
```

Číselné hodnoty pro uchování snímku a interval jejich pořizování mají být uváděny v minutách. Pro oba je výchozí hodnota NULL. Pokud je hodnota NULL zadána pro některou z hodnot definující nastavení frekvence pořizování snímků, ponechává se původní hodnota beze změny. Pokud je parametr *dbid* opomenut nebo ponechán na NULL, procedura použije aktuální a lokální parametr *dbid*.

Nástroj AWR je spolu s nástrojem ASH nejdůležitějším rozšířením databáze Oracle10g. ASH udržuje vzorky dat o sezeních, které jsou právě aktivní a poskytuje tyto informace právě nástroji AWR. Ten můžeme považovat za vestavěný systém pro ukládání statistik o výkonnosti databáze a zobrazování zpráv o nich, podobně jako to umí STATSPACK. Proto je AWR základem pro mnoho tzv. samoladicích vlastností Oracle10g databáze.

Kapitola 3

Návrh aplikace

Pro návrh aplikace z části využijeme možností jazyka UML. Tento jazyk je v současné době hojně využíván v oboru zvaném *softwarové inženýrství*. Jedná se o jazyk s grafickou interpretací, sloužící pro vizualizaci, specifikaci, návrh a dokumentaci projektů, jejichž náplní je vývoj programového vybavení [41]. UML podporuje objektově orientovaný přístup k návrhovým prostředkům, což je velmi výhodné z důvodu, že v dnešní době se již používá převážně objektově orientovaných programovacích jazyků. Díky tomu můžeme rychle a intuitivně přecházet z návrhu do implementace aplikace samotné.

3.1 Prvotní analýza a plán návrhu

Prvním krokem při návrhu jakéhokoliv programového vybavení je specifikace neformálních požadavků. Jedná se o zadání požadavků na program sepsané zadavatelem. Jde o čistě textovou formu popisu. Na základě těchto požadavků bude vytvořena prvotní analýza aplikace, na jejímž základě bude dále vytvořen plán projektu.

3.1.1 Neformální specifikace

Program, jenž chceme vytvořit, má shromažďovat, vyhodnocovat a v člověku čitelné podobě zobrazovat informace o chodu Oracle databáze. Má se jednat o klasickou aplikaci s grafickým uživatelským rozhraním (GUI), která se připojuje k různým databázovým instancím a monitoruje jejich stav. Monitorování databází se provádí prostřednictvím modulů aplikace, kde každý modul zjišťuje a zobrazuje určitou charakteristiku databáze, ke které je připojen.

Grafické uživatelské rozhraní

Z hlediska GUI se jedná o klasickou podobu aplikace, v jejímž hlavním okně nalezneme v horní části lištu s menu, na níž budou jednotlivá menu sloužící pro administraci aplikace samotné, seznamu databázových instancí a také modulů, které v ní budou zaregistrovány.

Ve spodní části bude tlačítková lišta, v jejíž levé části bude tlačítko, které bude sloužit k zobrazení menu s kategoriemi modulů a s moduly uvnitř těchto kategorií. Poklepáním na položku modulu bude vybraný modul spuštěn. Po spuštění modulu se ve spodní liště objeví tlačítko právě pro ovládání okna spuštěného modulu.

Okno modulu se zobrazí v hlavní části aplikace, tedy ve zbytku plochy okna mezi horní a spodní lištou. Toto okno lze maximalizovat, minimalizovat, obnovit, podobně jako to lze

provádět s aplikacemi běžícími v operačních systémech s grafickou nádstavbou.

Databázové instance

Aplikace v sobě bude mít zabudovaný seznam databázových instancí, ke kterým se bude připojovat a které bude zkoumat. Bude možné registrovat nové instance, stejně tak editovat a odstraňovat stávající, za předpokladu, že žádná databáze, jejíž nastavení chceme měnit nebo smazat, nebude v daný okamžik aktivní.

Připojit se k databázi bude možné za běhu aplikace, přičemž bude možné se připojit k více databázím najednou. Bude také možné se od databáze odpojit, ale pouze za předpokladu, že nebude používána žádnými aktuálně běžícími moduly.

Seznam databází, ke kterým je aplikace připojena, bude aplikace zprostředkovávat spuštěným modulům. Moduly se budou moci připojit pouze k databázím, které naleznou v tomto seznamu. Aplikace obhospodařuje pro každý modul všechna jeho připojení k dané databázi (relace), tudíž, bude-li chtít modul vytvořit novovou relaci k vybrané databázi, musí o to aplikaci požádat. Ta příslušné připojení inicializuje a předá jej modulu. Aplikace v sobě pro každý modul uchovává jemu poskytnuté databázové relace a při ukončení činnosti modulu všechna tato připojení odpojí, neučiní-li tak modul sám .

Moduly

Moduly budou tvořit výkonnou část aplikace. Každý modul bude samostatná aplikace, která bude mít svoji aplikační logiku a svoje GUI uzpůsobené tak, aby vhodně zobrazovalo výsledky monitorování dané databáze. Jelikož modul bude mít svoje vlastní GUI, bude tedy plně záviset na jeho naprogramování, jakým způsobem bude s uživatelem komunikovat a jaké akce bude uživateli poskytovat nad sebou sama. Ačkoliv moduly mají být plně autonomní, musejí spolupracovat s aplikací, která jim pro jejich chod bude poskytovat zázemí, hlavně co se týká databázových připojení k vybrané databázi.

3.1.2 Prvotní analýza požadavků

Na základě neformální specifikace aplikace lze odvodit, že se jedná o klasickou aplikaci s grafickým uživatelským rozhraním. Aplikace jako taková slouží jako podpůrný prostředek pro běh jednotlivých modulů. Jejím hlavním úkolem je administrovat seznam databázových instancí a modulů.

Pro administraci modulů bude nutné implementovat manažera kategorií a modulů, který bude provádět veškeré operace nad seznamem modulů a také nad seznamem kategorií, které bude, stejně jako moduly, možné vytvářet, editovat a mazat. Stejného manažera budeme muset vytvořit pro administraci databází.

Je zbytečné uvažovat o nějakém systému řízení báze dat (SŘDB), do kterého budeme ukládat informace o zaregistrovaných kategoriích, modulech i databázích. Informací o jednom modulu, kategorii či databázi bude minimum a celkový počet modulů, kategorií i databází nebude nikterak obsáhlý, bude se jednat o řádově desítky. Pro tyto údaje si plně vystačíme s možnostmi různých formátů souborů, jako je například formát XML. Tyto soubory uložíme jednoduše do souborového systému, ve kterém bude aplikace nainstalována.

Každý spuštěný modul bude mít svoje vlastní okno, zasazené uvnitř pracovní plochy aplikace, a tlačítko na spodní tlačítkové liště, pomocí kterého bude možné ovládat zmíněné okno modulu. Inspiraci lze vzít např. z operačního systému Windows.

Přestože moduly budou autonomní aplikace, budou úzce spolupracovat s aplikací samotnou a budou její součástí. Proto je nutné vytvořit jednoduché aplikační programovací rozhraní (API), které musí každý modul implementovat a využívat.

Vzhledem k tomu, že musí být možné moduly do aplikace přidávat a spouštět dynamicky za běhu aplikace, je třeba navrhnout a implementovat vhodné programové vybavení. Jako nejvhodnější se zdá využití funkcionality zvané *Java reflexe*, kterou poskytuje vývojová sada společnosti *Sun Microsystems*, zvaná *Java Development Kit* (JDK). Java reflexe bude použita pro dynamické vytváření instancí tříd, v kombinaci s JAR archívem jako výstupní formou vytvořeného modulu. Musejí však být dána určitá pravidla, jež musejí být při implementaci modulu dodržena, aby tato kombinace byla funkční. Především bude třeba, aby vytvořený modul implementoval případná rozhraní či definoval třídy odvozené z abstraktních tříd, které bude poskytovat již zmiňované API pro tvorbu modulů, protože pouze takto lze dosáhnout cílené funkčnosti.

Jelikož každý modul bude běžet nezávisle na ostatních, je potřeba, aby měl svůj vlastní adresový prostor a měl pro svůj výpočet přidělen určitý čas procesoru. Proto bude vhodné, aby každý modul běžel ve svém vlákně, které mu bude dáno již při spuštění na základě API, které bude implementovat. Bude-li chtít využívat více vláken, bude záležet již na implementaci každého modulu zvlášť.

3.1.3 Naplánování projektu

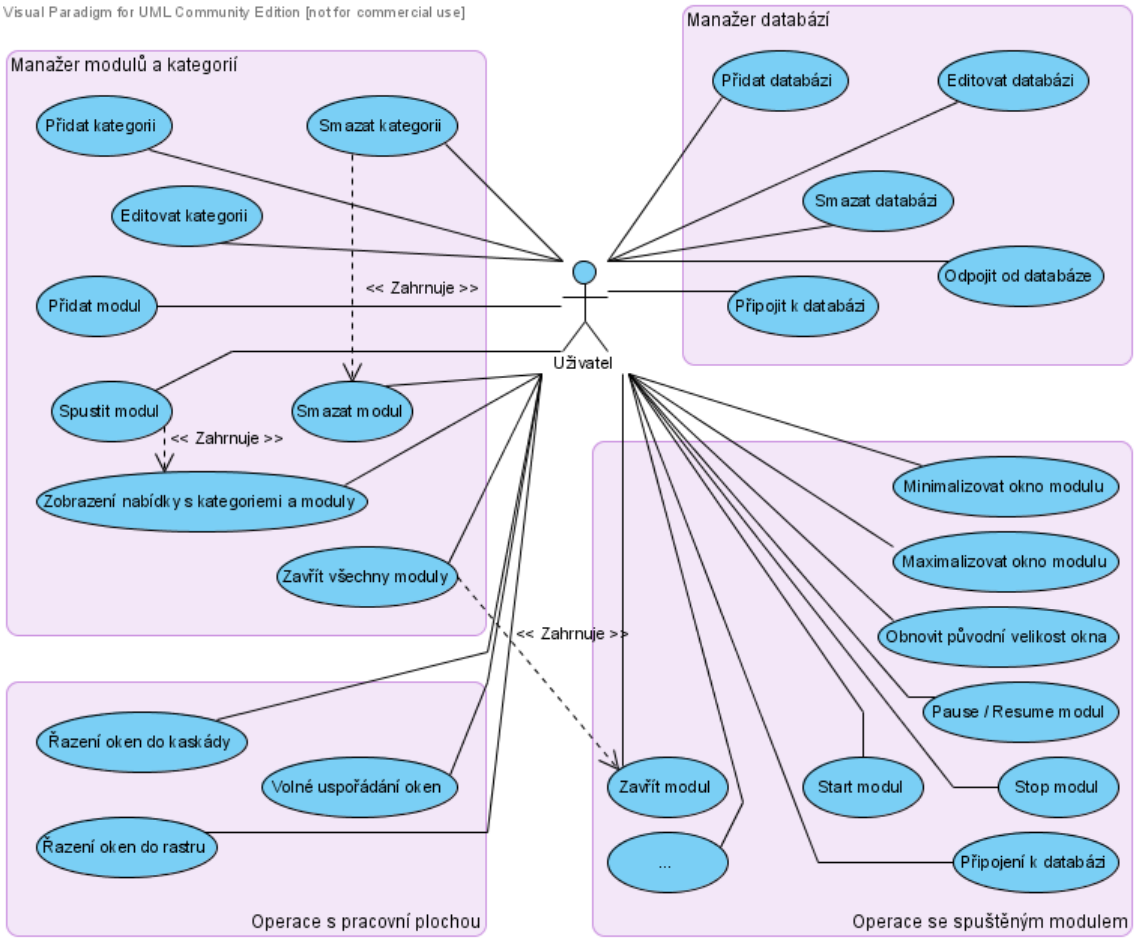
Vývoj aplikace bude prováděn v několika etapách. Nejdříve bude vytvořena aplikační logika a GUI aplikace, aby bylo vytvořeno zázemí pro běh modulů. Jedná se především o rozmístění grafických komponent a implementaci reakcí na jejich použití. Konkrétně máme na mysli okna modulů a tlačítka odpovídající těmto oknům, umístěná na spodní tlačítkové liště, aby byly tyto prvky korektně propojeny.

Dále je třeba implementovat manažery modulů, kategorií a databází. Je nutné zvolit a naprogramovat vhodné úložiště dat pro tyto manažery a implementovat operace nad těmito seznamy.

Další fází projektu je nastudování, vyzkoušení a implementace načítání a spouštění modulů dynamicky za běhu aplikace. S tím souvisí také implementace vláken a na závěr bude třeba vytvořit API pro tvorbu modulů, jehož funkcionality musejí moduly využívat pro spojení s aplikací.

V poslední fázi již zůstává implementace modulů. Hlavní náplní modulů je samozřejmě monitorování určitých statistik a vlastností Oracle databáze. Statistiky budou sbírány většinou z pohledů mechanismu *Wait Interface*. Bude nutné prostudovat všechny souvislosti, které se zobrazovanými statistikami souvisejí.

S tím také souvisí problematika zobrazení získaných dat v člověku srozumitelné podobě, především pomocí grafů. Je potřeba nastudovat různé knihovny, které se vykreslováním grafů v jazyce Java zabývají, jejich možnosti, omezení, licenční politiku, jednoduchost a především vhodnost nasazení v naší aplikaci.



Obrázek 3.1: Diagram případů užití aplikace.

3.2 Diagram případů užití

Diagram případů užití, viz. obrázek 3.1, je jakýsi přehled všech úkonů, které můžou aktéři s aplikací provádět. V kontextu celého procesu návrhu programu je tento diagram posledním krokem, kdy se na navrhovanou aplikaci díváme z pohledu koncového uživatele, nebo-li z hlediska toho, co všechno by měla aplikace umět.

Všechny případy užití jsou vztaženy na jednoho aktéra. Nemáme definovanou žádnou hierarchii uživatelských práv, jelikož aplikace jako taková je, díky svému charakteru, sama o sobě zaměřená na úzkou skupinu uživatelů. Předpokládá se tedy, že uživatel je znalý dané problematiky a záleží na něm, jak bude celou aplikaci využívat.

Roli aktéra pojmenovanou intuitivně *Uživatel* můžeme označit jako roli superuživatele či administrátora se všemi atributy, kterými je tato role charakteristická.

Diagram je kategoricky rozdělen na čtyři oblasti. Podle přirovnání z kapitoly 3.1.2, kde jsme uvedli, že aplikace sama o sobě slouží pouze jako nástroj pro administraci prostředků, se kterými operuje výkonné jádro v podobě modulů, je diagram případů užití rozdělen na oblasti, které spolu v rámci administrace prostředků úzce souvisejí.

Vpravo nahoře je kategorie obsahující výčet operací souvisejících s registrací modulů a kategorií. Vycházíme ze zadání, které říká, že jednotlivé moduly musejí být organizovány v kategoriích. Proto aplikace musí uživateli umožňovat vytváření, editaci a mazání kategorií modulů.

Teprve s možnostmi správy kategorií je možné dále pracovat přímo s moduly samotnými. Editace modulů není možná, jelikož modul je samostatný program, jehož konfigurace a funkcionalita je daná programem, který nelze upravit. Případ užití *Smazat kategorii* v sobě zahrnuje případ užití *Smazat modul*. To je zde zobrazeno proto, že při mazání kategorie je provedeno smazání veškerých modulů, které jsou evidovány uvnitř kategorie, kterou chceme smazat. Analogicky k výše uvedenému případu užití, přeje-li si uživatel spustit modul, musí si nejdříve zobrazit nabídku s kategoriemi a moduly, aby z této nabídky mohl zvolený modul spustit. Posledním případem užití z této oblasti je zavření všech modulů, které využívá implementace funkcionality obstarávající zavření modulu, tentokrát však pro všechny spuštěné moduly najednou.

Ostatní tři oblasti diagramu případů užití jsou výmluvné samy o sobě. Za zvýraznění stojí už pouze případ užití v oblasti nazvané *Operace se spuštěným modulem*, se strohým názvem „…“ , jež má za úkol informovat o tom, že se spuštěnými moduly lze provádět ještě další operace, které ale nejsme schopni v současné době přesně definovat. Jedná se totiž o operace, které jsou dány implementací každého vytvořeného modulu zvlášť. Nezávisejí na našem návrhu nebo na požadavcích zadavatele na funkcionalitu aplikace, nýbrž na požadavcích kladených na každý modul zvlášť.

3.2.1 Specifikace případů užití

Jednotlivý případ užití charakterizuje podrobně konkrétní postup, který je krok za krokem prováděn za účelem provedení požadované akce uživatelem. Každý takový specifický případ užití má svůj název, jednoznačný identifikátor, který je zde pro dobrou orientaci ve všech případech užití, hlavně z důvodu, aby bylo vidět, které alternativní či chybové toky patří ke kterému případu užití. Dále jsou vyjmenovaní aktéři, kteří se dané akce účastní. Ještě před začátkem provádění musejí být specifikovány předpoklady, což jsou, dá se říct, podmínky, které musejí být splněny, aby daná akce bezchybně proběhla. Následuje posloupnost operací vedoucí k řádnému a úspěšnému dokončení požadované akce a definice následných podmínek.

V těchto specifikacích je popsáno [44]:

- co systém dělá, ale ne jak to dělá
- jak a kdy činnost začíná a končí
- kdy má systém interakce s aktérem
- které údaje jsou měněny
- jaké kontroly vstupních údajů jsou prováděny
- základní, alternativní a chybové průběhy

Alternativními toky označujeme postupy, které vedou také ke správnému dokončení požadované akce, ale nepřímým způsobem. Typicky se jedná o nestandardní situaci vzniklou

uvnitř hlavního toku případu užití. V alternativním toku potom provádíme kroky, které na vzniklou nestandardní situaci reagují a směřují k úspěšnému dokončení operace.

Oproti tomu chybové toky zobrazují postupy při vzniku chyby během hlavního toku, která není slučitelná s korektním dokončením dané akce a popisuje co v tomto případě aplikace provede.

Specifikace případů užití pro naši aplikaci odpovídající diagramu případů užití z obrázku 3.1 jsou uvedeny v příloze A.

3.3 Návrh architektury

Při výběru modelu architektury je možné zvolit prakticky mezi dvěmi architekturami:

- **Klasický model** – Odděluje datový model od uživatelského rozhraní s řídící logikou
- **Model – View – Controller (MVC)** – Odděluje datový model, uživatelské rozhraní a řídící logiku

Klasický model architektury, nazývaný *Model 1*, odděluje uživatelské rozhraní s aplikační logikou od datového modelu. Druhá architektura vychází z předchozí a její odlišnost spočívá v oddělení aplikační logiky od uživatelského rozhraní při zachování oddělení datové části.

MVC je vhodná především a hlavně pro použití pro webové aplikace. Komponenta *Model* zahrnuje datový model, tedy perzistentní data aplikace (většinou databáze) spolu s aplikační logikou nutnou k provádění operací nad modelem dat. Komponenta *View* se stará o zobrazení určených dat koncovému uživateli, čili má na starosti správu uživatelského rozhraní. Poslední část zvaná *Controller* je mezičlánek mezi prvními dvěma částmi architektury. Registruje požadavky uživatelského rozhraní a předává je ke zpracování aplikační logikou a naopak určuje, jak má uživatelské rozhraní naložit s daty předanými datovou částí.

Struktura modelu MVC:

- **Model** – Reprezentuje informace, s nimiž aplikace pracuje
- **View** – Převádí data do podoby vhodné k prezentaci
- **Controller** – Reaguje na události, zajišťuje změny v pohledu (view) a modelu

V naší aplikaci použijeme architekturu MVC. Princip oddělení aplika-ční logiky, GUI a datové části je velmi užitečný, je flexibilnější, přehlednější a hlavně čitelnější. Díky své struktuře je jednodušší definovat, které činnosti patří do které vrstvy architektury.

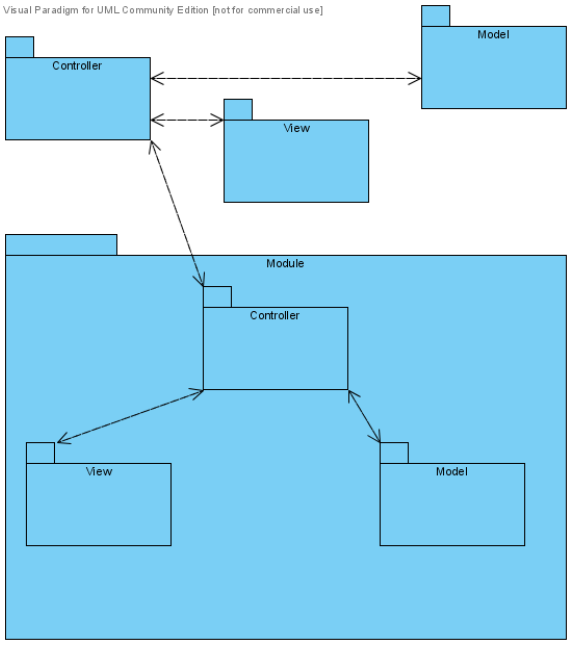
Za část architektury nazvané *Model* budeme považovat třídy a objekty, které se starají o správu seznamu databází a modulů či kategorií. Jejich úkolem je poskytnout takové rozhraní, s jehož využitím budeme schopni provádět veškeré potřebné operace. Zároveň, v souladu s pojmem zapouzdření z terminologie OOP, zastíníme detaily práce s těmito seznamy, jako je způsob jejich uložení (SŘDB, uložení do souboru, např. ve formátu XML, ...), s čímž jde ruku v ruce také způsob získávání, editace a vkládání dat, apod.

Komponenta *View* obsahuje veškeré grafické komponenty aplikace. Sem patří hlavní okno aplikace. Dále grafická prostředí všech manažerů aplikace, tedy okna manažerů a všech dialogových oken, které jsou skrze manažery vyvolávána.

Vrstvu *Controller* chápeme jako prostředníka mezi předchozími dvěmi komponentami. Vrstva *Controller* zpracovává signály, které jsou inicializovány uživatelem aplikace skrze prvky grafického uživatelského rozhraní komponenty *View*. Na základě těchto zpráv provádí operace vedoucí k žádanému cíli, přičemž, je-li to nutné, využívá možností komponenty *Model*, resp. jejích složek. V mnoha případech je třeba výsledek provedené akce opět zobrazit, čili komunikace mezi komponentami *View* a *Controller* je obousměrná.

Dá se říct, že komponenty *View* a *Model* jsou striktně odděleny a jedna o druhé nemá žádné informace, ani o jejich samotné existenci, ačkoliv jde především o grafické zobrazení dat. Všechno je prováděno prostřednictvím vrstvy *Controller*.

3.4 Diagram balíků



Obrázek 3.2: Diagram balíků

Vyvíjená aplikace však není stavěná tak, že se skládá z jedné třídy *View*, *Controller* a *Model*. Každá vrstva architektury se skládá z většího počtu tříd, které definují a implementují operace pro několik logických celků, které dohromady tvoří celou aplikaci.

Základní členění těchto tříd provádíme podle toho, který prvek architektury implementují. Třídy rozčleňujeme do tzv. balíků. Propojení těchto balíků ukazuje, jak spolu balíky spolupracují.

Kromě základních tří balíků, které rozdělují aplikaci podle architektury aplikace, je zde navíc ještě jeden balík s názvem *Module*, který abstrahuje modul, který uvnitř aplikace běží. Těchto modulů může být samozřejmě víc, ale kvantita nás momentálně nezajímá, jde tady pouze o jakési rozčlenění, abychom věděli, které operace se týkají modulů a které samotné aplikace.

Architektura modulů zůstává stejná, tedy MVC, což naznačují balíky vnořené balíku *Module*. Nelze však přehlédnout znázornění, že samotný balík *Module* není aktérem žádné

komunikace v rámci aplikace. Jediným, kdo je v rámci modulu s aplikací schopen komunikovat s aplikací, je vrstva *Controller* modulu. Tuto skutečnost můžeme nazvat protokolem komunikace mezi běžícím modulem a aplikací samotnou.

Díky tomuto rozložení aplikace definujeme komunikační kanály mezi jednotlivými třídami, resp. mezi vrstvami architektury. Čím užší je komunikační linka mezi jednotlivými balíky, tím je čitelnější a přehlednější. Komunikační linky jsou tvořeny tak, aby jich mezi balíky bylo co nejméně. V rámci balíku mohou třídy komunikovat téměř libovolným způsobem, přičemž pokud chce některá třída z jednoho balíku komunikovat s třídou z jiného balíku, musí vědět, že jí k tomu slouží ta a ta komunikační linka. Rozčleněním aplikace do balíků se snažíme vyhnout situaci, kdy každá třída komunikuje s jinou třídou nahodile, což způsobuje zcela nepřehlednou strukturu programu.

3.5 Diagram tříd

Oproti diagramu balíků aplikace nám diagram tříd poskytuje detailnější pohled na strukturu navrhované aplikace. Kromě balíků, jejichž strukturu zachovává, zobrazuje konkrétní třídy, které jsou součástí výsledné aplikace, a vztahy mezi nimi.

Jedná se o konceptuální diagram, je tedy nutné se na něj dívat s nadhledem. Nelze jej brát jako závazný předpis pro implementaci aplikace z pohledu definice tříd a jejich atributů a operací. Tyto jsou spíše návrhem z hlediska koncepce aplikace. Jejich úkolem je přiblížit, které funkce a vlastnosti jsou pro konkrétní třídu charakteristické a které jsou z hlediska jejího významu stěžejní a díky nimž je patrné, co se od dané třídy očekává a k čemu má sloužit.

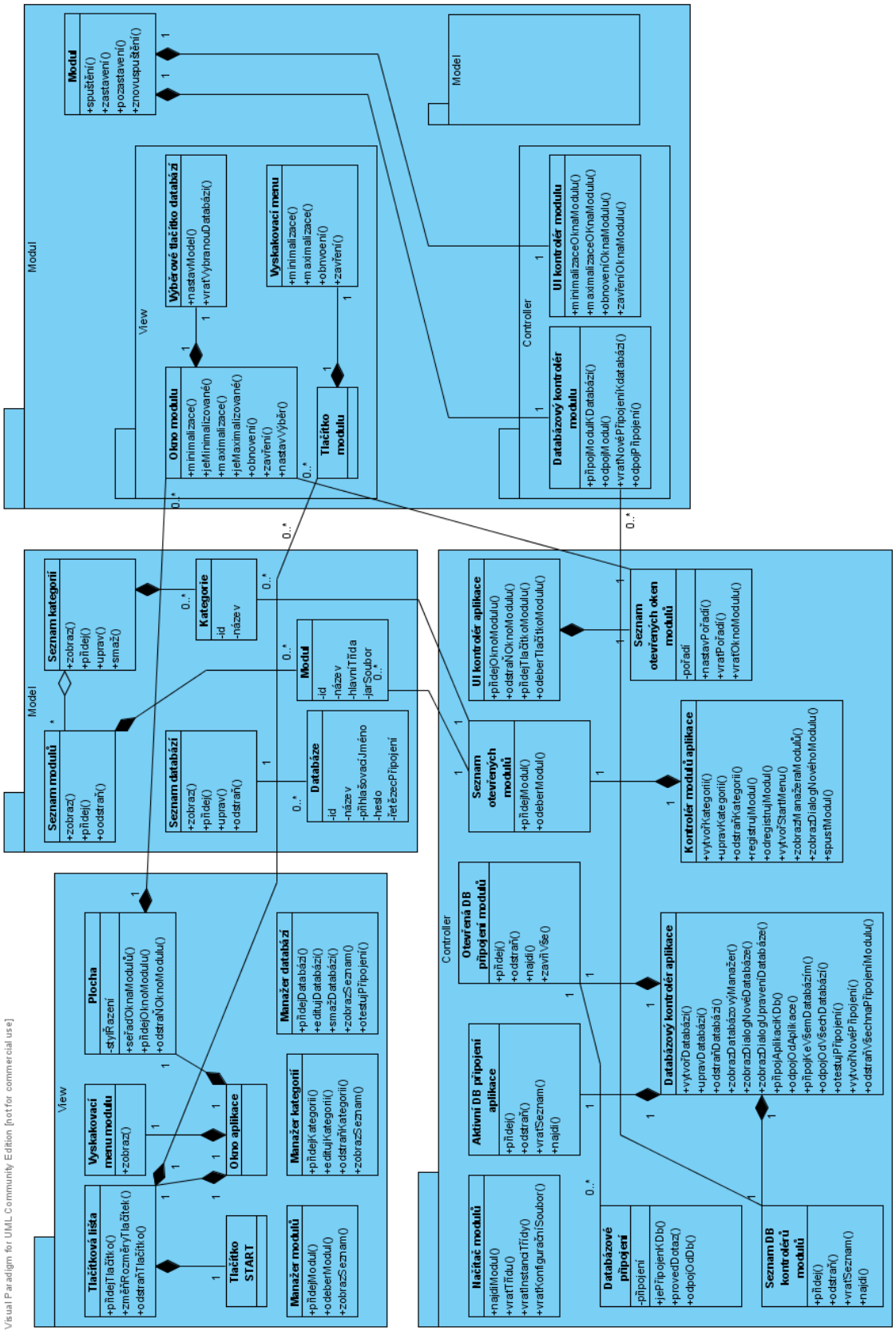
Např. třída `Databázový kontrolér aplikace` by měla poskytovat operace pro práci s databázemi aplikace. Ve spolupráci s modelem by měla být schopna registrovat nové databázové instance, editovat vlastnosti stávajících a také je mazat (operace `vytvořDatabázi()`, `upravDatabázi()` a `odstraňDatabázi()`).

V souvislosti s těmito operacemi musí ovládat zobrazení grafického rozhraní, které uživateli poskytuje prostředky pro operace (`zobrazDatabázovýManažer()`, `zobrazDialogNovéDatabáze()` a `zobrazDialogUpraveníDatabáze()`), tedy musí spolupracovat s komponentou *View*.

Posledním typem operací je vytváření instancí databázových připojení pro moduly a jejich správa. Funguje jako nadřazený správce, který je modulem požádán o získání instance databázového připojení k některé databázi. Tato třída jej vytvoří a předá mu jej, zároveň si však ale uchová informaci o jeho vytvoření a je-li třeba, všechna tato připojení deaktivuje, např. při ukončení modulu, pokud to neimplementuje modul sám. Tuto funkcionalitu popisují operace `vytvořNovéPřipojení()` a `odstraňVšechnaPřipojeníModulu()`.

Vztahy, které jsou v diagramu mezi třídami vyznačeny, definují strukturu či složení těchto tříd. Udávají, že jedna třída je, mimo jiné, složena ze třídy druhé, případně z několika tříd z druhého konce vztahu. Pro pochopení významu udejme příklad, přičemž využijeme, stejně jako v předchozím příkladu, třídy `Databázový kontrolér aplikace`.

Tato třída má s třídami `Otevřená DB připojení modulů`, `Aktivní DB připojení aplikace` a `Seznam DB kontrolérů modulů` definován vztah kom-



Obrázek 3.3: Diagram tříd

pozice[1] (speciální typ agregace[2]). To znamená, že třída `Otevřená DB připojení modulů` obsahuje instance objektů těchto tříd. Kardinalita tohoto vztahu je 1, což znamená, že třída `Otevřená DB připojení modulů` dané instance obsahuje právě jednou.

Analogicky můžeme sledovat vztahy také v balíku *View* mezi třídami `Okno aplikace`, `Tlačítková lišta`, `Plocha`, `Tlačítko START` a `Vyskakovací menu modulu`. Na základě toho odvodíme, že `Okno aplikace` obsahuje právě jednu instanci tříd `Plocha`, `Tlačítková lišta` a `Vyskakovací menu modulu` a protože se jedná o třídy definované v balíku *View*, takže se nejspíš jedná o grafické prvky, tak se lze dovtípit, že `Okno aplikace` by mohlo být okno aplikace obsahující právě jednu tlačítkovu lištu, plochu, na níž se zobrazují okna modulů, atd. Dále můžeme odvodit, že tlačítková lišta obsahuje právě jedno tlačítko START - třída `Tlačítko START`.

Vrátíme-li se zpět ke třídě `Databázový kontrolér aplikace`, resp. ke třídě `Otevřená DB připojení modulů`, postřehneme, že definuje ještě jeden vztah a sice ke třídě `Databázové připojení`. Nyní se však jedná o vztah typu asociace[3] s kardinalitou `0..*`, což znamená, že třída `Otevřená DB připojení modulů` může obsahovat `N` instancí objektu třídy `Databázové připojení`. Můžeme si to představit např. tak, že `Otevřená DB připojení modulů` je z hlediska programovacího jazyka proměnná, do které lze vkládat více hodnot, tedy pole, seznam, vektor, apod. a hodnoty, které do ní vkládáme, jsou právě instance objektu třídy `Databázové připojení`.

Na předchozích příkladech jsme ukázali, jak se na diagram tříd dívat a jak jej chápat. Definovali jsme tedy jakýsi koncept, od kterého by se měl dále návrh aplikace odvíjet. Zároveň jsme každou třídu zařadili do některého z balíků. Struktura balíků i jejich názvy odpovídají komponentám architektury MVC, čímž jsme chtěli ukázat, které třídy patří do které vrstvy architektury aplikace.

3.6 Diagramy interakce

Diagramem tříd jsme definovali a popsali statickou strukturu aplikace. To ale samo o sobě nestačí. Ačkoliv jsme schopni na základě diagramu tříd rámcově pochopit strukturu, nejsme schopni si představit jak spolu objekty (instance tříd) komunikují. Tato komunikace utváří řízení toku aplikace – určuje posloupnost provádění příkazů procesorem v čase, čímž jsme schopni zobrazit dynamickou část aplikace.

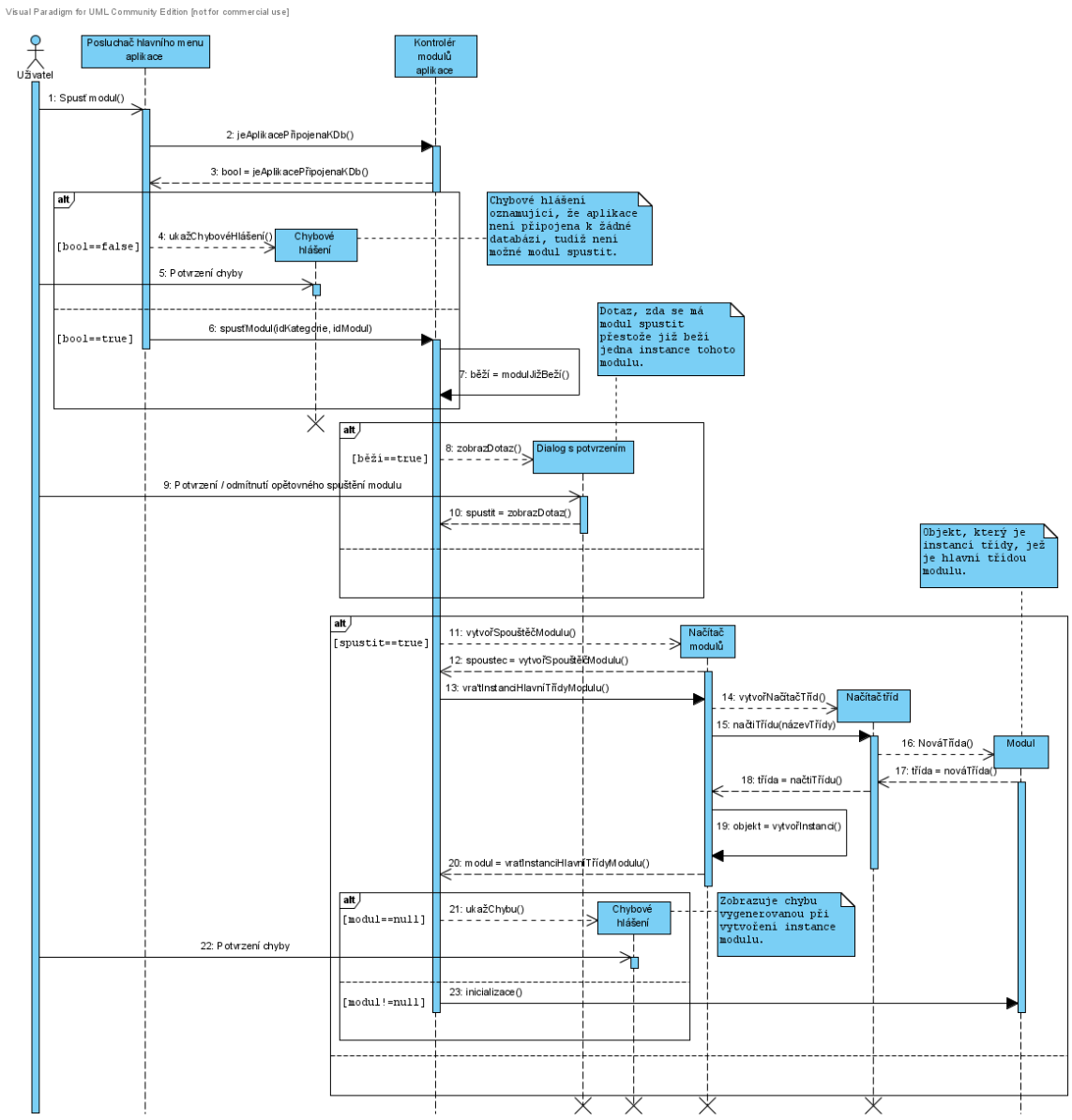
Ke znázornění dynamického chování aplikace slouží tzv. diagramy interakce, konkrétně diagramy sekvenční. Pomocí nich můžeme přehledně ukázat, jak spolu objekty spolupracují a díky existenci časové osy jsme schopni rozpoznat i návaznost jednotlivých kroků výpočtu.

První sekvenční diagram na obrázku 3.4 znázorňuje proces spuštění modulu. Ukazuje jak jsou provázány jednotlivé vrstvy architektury MVC a jak vzájemně spolupracují. Uživatel spolupracuje pouze s vrstvou *View*, kterou zde reprezentují objekty zobrazující chybová a tázací dialogová okna a také posluchač, který zaregistroval akci uživatele, že zadal akci spuštění modulu. Tento posluchač pak dále komunikuje s vrstvou *Controller*, jíž reprezentuje

[1] kompozice — Speciální typ agregace, ve které daný objekt může existovat pouze v rámci definovaného celku, nikoliv samostatně a může být součástí maximálně jednoho celku

[2] agregace — Vztah mezi třídami, resp. objekty. Speciální forma spojení objektů, která specifikuje vztah mezi celkem a jeho částí. Objekt ve vztahu definovaná jako část může existovat i bez svého celku. Část může být součástí různých celků.

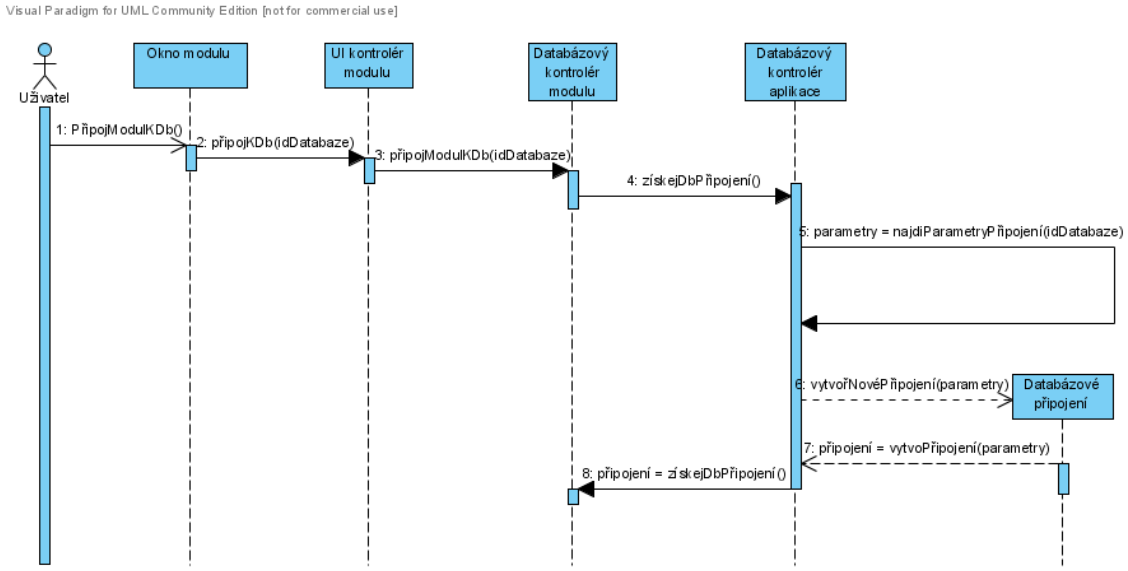
[3] asociace — Obecná souvislost (vztah) objektů.



Obrázek 3.4: Sekvenční diagram znázorňující proces spuštění modulu.

`Kontrolér modulů aplikace`. Ten přebírá kontrolu nad prováděním dalších akcí spojených se spuštěním modulu a stává se řídícím prvkem tohoto procesu.

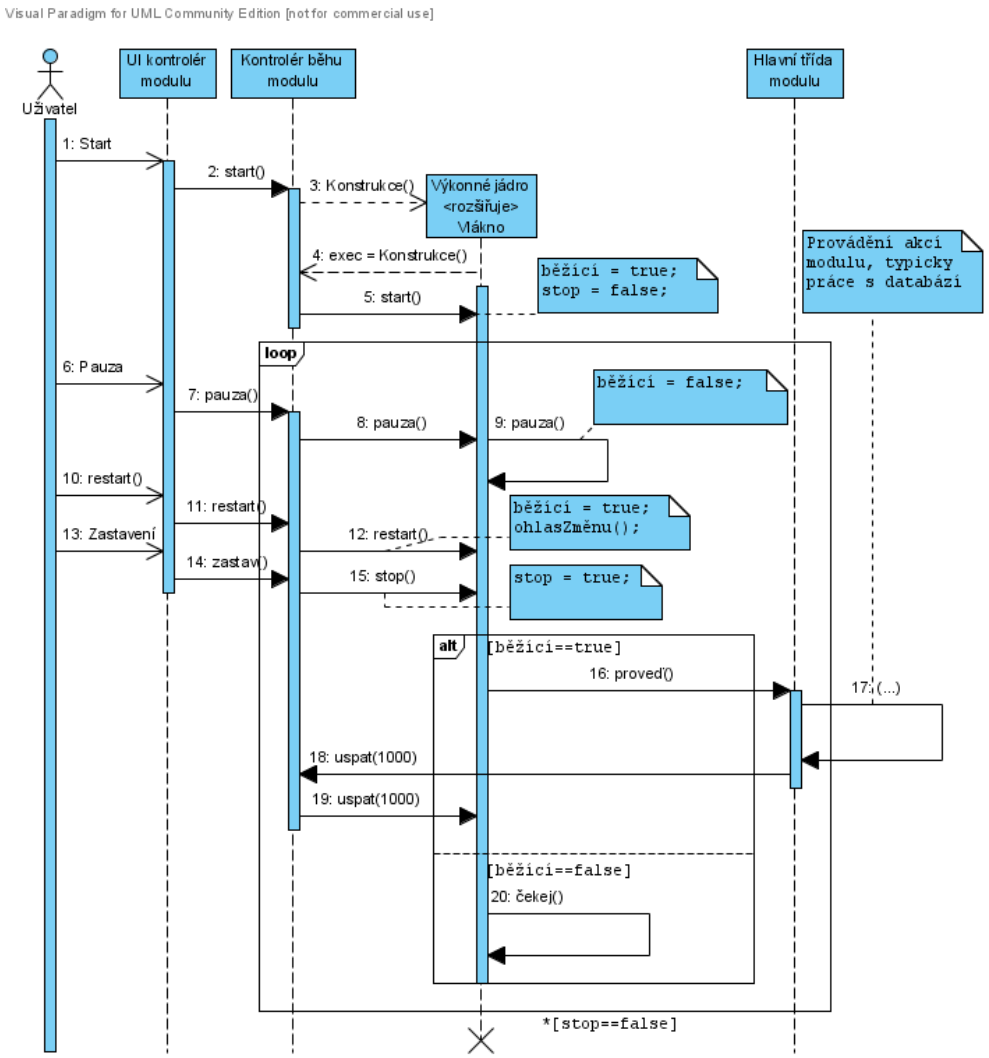
Na počátku je akce uživatele, kterou zaregistroval příslušný posluchač. Ten si nejdříve od kontroléru zjistí, zda je aplikace připojena k alespoň jedné databázi. Pokud ne, zobrazí se chybové hlášení, které musí uživatel potvrdit, čímž dá najevo, že s tím byl srozuměn a podnikne příslušné další kroky. Jestliže aplikace k nějaké databázi připojená je, pošle posluchač zprávu `spust'Modul(idKategorie, idModul)` kontroléru, pro nějž to znamená, aby přebral kontrolu nad dalšími činnostmi. Ten nejprve zjistí, zda již daný modul neběží. Pokud běží, oznámí to uživateli a zeptá se jej, zda chce spustit další instanci tohoto modulu či ne. V případě kladné odpovědi následuje vytvoření objektu třídy, která poskytuje rozhraní pro spouštění modulů. Tu následně požádá o vytvoření instance hlavní třídy modulu a čeká na vyřízení žádosti. Mezitím je, v případě bezchybného průběhu, vytvořen objekt hlavní třídy modulu, který je předán kontroléru. Ten poté celý nově spuštěný modul inicializuje posláním zprávy `inicializace()`. Tím je proces spuštění modulu ukončen a výsledkem je modul, který je připraven k práci.



Obrázek 3.5: Sekvenční diagram znázorňující proces vytvoření databázového připojení modulu.

Druhý diagram (obrázek 3.5) je velmi významný z hlediska chápání aplikace navrženou na základech architektury MVC. Nutno zdůraznit, že na architektuře MVC není postavena jenom aplikace samotná, ale i moduly!

Celá operace začíná opět u uživatele, který pomocí grafických prvků v okně modulu zvolí operaci připojení modulu k některé z nabízených databází. Následně je poslána zpráva kontroléru `UI Kontrolér modulu`, který je v rámci modulu navržen jako výhradní komunikační kanál mezi vrstvou *View* a *Controller* uvnitř modulu. Tento kontrolér tedy zjistí, že se objevil požadavek na připojení modulu k databázi a ví, že pro operace nad databázovými připojeními modulu slouží objekt nazvaný `Databázový kontrolér modulu`, jemuž zašle zprávu o tom, že je požadováno, aby se modul připojil k vybrané databázi. Jakmile `Databázový kontrolér modulu` obdrží tuto správu, zažádá o nové připojení jemu nadřazený objekt, jímž je `Databázový kontrolér aplikace`, který již není součástí mo-



Obrázek 3.6: Sekvenční diagram znázorňující žitovní cyklus běhu modulu.

dulu, nýbrž se jedná o tzv. „jedináčka“[4] jež je součástí samotné aplikace jako nadřazené instance všech spuštěných modulů. Teprve `Databázový kontrolér aplikace` na základě žádosti o vytvoření databázového připojení toto inicializuje a předá zpátky databázovému kontroléru modulu, který už s ním dále nakládá.

Nyní už by mělo být snadnější pochopit architekturu aplikace. Především, ve smyslu toho, že jak samotná aplikace, tak i jednotlivé moduly jsou postaveny na architektuře MVC, která striktně vzájemně odděluje aplikační logiku od prezentační vrstvy a modelu. Díky tomuto řešení je struktura aplikace přehledná a jsou přesně definovaná pravidla k čemu každá vrstva a také komponenta v rámci vrstvy slouží.

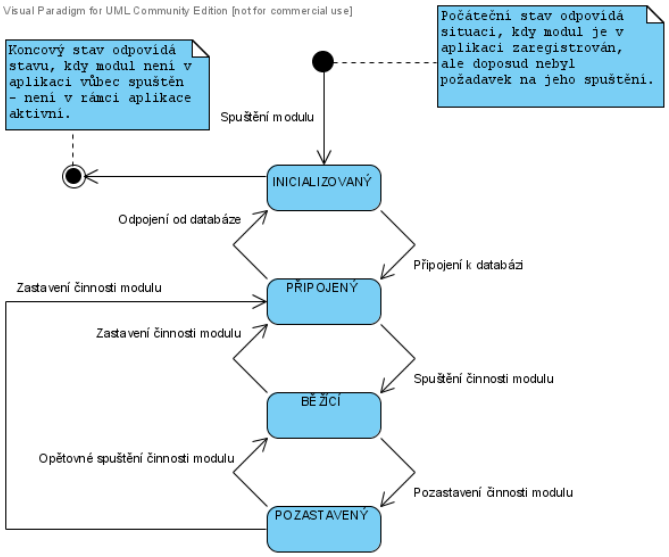
Také jsou jasně nastaveny komunikační kanály mezi kontroléry aplikace a kontroléry modulů, které již z diagramu nevyplývají, ale jejich spolupráce může být logická. Např. `Databázový kontrolér modulu` uvnitř modulu zná a používá pro komunikaci s aplikací jenom a pouze `Databázový kontrolér aplikace`, analogicky `UI kontrolér modulu`, který

[4]jedináček — Třída, jež má v programu právě jednu instanci.

se stará o komunikaci s prezentační vrstvou modulu, komunikuje v rámci aplikace jenom a pouze s `UI kontrolér aplikace`. Takto přesným definováním pravidel komunikace tvoříme také jakási bezpečnostní pravidla, jelikož zúžením rozhraní poskytovaného kontroléry aplikace při vytváření modulů částečně zajistíme, že budou pro určité operace, jež slouží pro komunikaci mezi aplikací a modulem, použity správné metody a konstrukce.

Poslední sekvenční diagram je opět specifický. Můžeme na něm vidět celý životní cyklus běhu modulu. Od jeho počátku, přes pozastavení, opětovné spuštění až po ukončení běhu modulu. Svým způsobem se jedná o čtyři nezávislé akce a interakce, ale každá interakce sama o sobě by nebyla příliš vypovídající a všechny by byly téměř identické. Díky společnému znázornění lze lépe popsat souvislosti, viz. obrázek 3.6.

V čase 0 je, z pohledu diagramu, modul ve stavu inicializace. Je připraven ke spuštění, se všemi náležitostmi, které s tím souvisejí. Uživatel tedy vybere akci spuštění běhu modulu. Tu, jak již známe z předchozích diagramů, převezme `UI kontrolér modulu`, který ji předá kontroléru `Kontrolér běhu modulu`, který je určený právě pro správu běhu modulů. Ten následně vytvoří a inicializuje objekt `Výkonné jádro`, jenž je potomkem třídy `Vlákno`. Okamžitě s vytvořením vlákna toto začíná běžet a sice tak, že v nekonečné smyčce kontroluje hodnotu proměnné `stop` a dokud je tato proměnná `TRUE`, tak se celá smyčka opakuje. Uvnitř této smyčky je podmíněný příkaz navázaný na proměnnou `běžící`, jež signalizuje, zda modul běží nebo zda je běh pozastaven. V případě, že modul běží, zašle objekt `Výkonné jádro` zprávu hlavní třídě modulu s názvem `proved'()`, která v sobě obsahuje implementaci toho, co se má za běhu vlákna s modulem dít. Pokud je běh modulu pozastaven, vlákno změní svůj stav na „*čekající*“.



Obrázek 3.7: Stavový diagram znázorňující jednotlivé stavy modulu, do kterých se může v průběhu svého životního cyklu dostat.

Výše popsaná smyčka se po spuštění běhu modulu pořád opakuje, dokud do toho nevstoupí uživatel. Ten do toho může vstoupit s požadavkem na pozastavení činnosti modulu (pouze pokud modul běží), může běh modulu opětovně spustit (možné pouze po předchozím pozastavení běhu) nebo běh modulu úplně ukončit. Toho lze dosáhnout nastavením příslušných proměnných, které ovlivňují rozhodovací proces uvnitř smyčky objektu

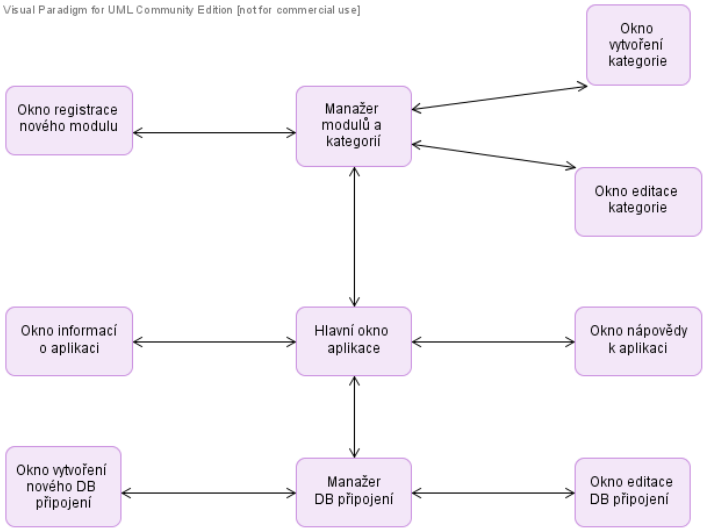
`Výkonné jádro`. V diagramu je požadavek na pozastavení běhu modulu popsán akcemi s pořadovými čísly 6 – 9, opětovné spuštění s čísly 10 – 12 a ukončení běhu modulu pod čísly 13 – 15. Ačkoliv na obrázku jsou tyto akce zobrazeny, z pohledu časové osy, za sebou, ve skutečnosti je třeba je chápat jako asynchronní zprávy, které jsou na sobě nezávislé a přicházejí jednotlivě. Jsou ale umístěny v rámci nekonečné smyčky, čímž chceme ukázat, že smyčka se opakuje neustále, ale díky akcím uživatele dochází k nastavení proměnnných, které jsou pro chování výkonné jednotky rozhodující.

Po ukončení běhu modulu (jakmile proměnná `stop == TRUE`) končí hlavní smyčka výkonné jednotky. Ihned po ukončení tohoto cyklu je objekt `Výkonné jádro` zrušen, což koresponduje s tím, že po ukončení běhu modulu je možné modul opět spustit, čímž se vytvoří tento objekt nový. Odstranění výkonné jednotky po ukončení běhu modulu je tedy nutné pro čistotu aplikace, aby se opakovaným spuštěním a ukončováním běhu modulu zbytečně nezvyšovala její paměťová náročnost.

Nyní máme konkrétní představu o tom, jakým způsobem probíhá komunikace mezi jednotlivými částmi aplikace pro zvládnutí nějakého úkolu. Kromě definování pravidel komunikace v rámci vrstev aplikace (MVC) jsme také objasnili, jakým způsobem mají být implementovány některé činnosti aplikace, které můžeme považovat za stěžejní a můžeme si z nich vzít příklad pro ostatní úkony.

3.7 Diagram návaznosti obrazovek

Chceme-li programátorovi aplikace přesně zadat, jakým způsobem na sebe mají navazovat různá okna a dialogy aplikace, můžeme tak učinit pomocí diagramu na obrázku 3.8. Pro každé okno přesně určíme, které je jeho předkem (ze kterého je voláno) a také, která další okna mohou být z daného okna vyvolána (následníci).



Obrázek 3.8: Diagram návaznosti obrazovek.

V tomto okamžiku jsme ve stádiu, kdy máme konkrétní zadání a také máme zpracovaný návrh, jak by měla aplikace vypadat. Počáteční specifikace požadavků nám poskytuje velké množství informací o tom, co všechno by měl program umět. Podrobně popsaný návrh architektury a diagram tříd nám dodal konkrétní představu o tom, jakou strukturu by měla implementace aplikace odrážet a na čem by měla být stavěna. Diagramy interakce doplňují další pohled, který ukazuje, jakým způsobem spolu mají objekty tvořící aplikaci spolupracovat. Diagram návaznosti obrazovek částečně definuje, jak by se měla aplikace chovat a jak by měla vypadat z pohledu uživatele. Prezentační vrstva je ale jinak ponechána na vývojáři aplikace samotném. Předpoládá se ale, že aplikace bude jevit známky kvalitního programu, který je lehce ovladatelný, přehledný a intuitivní, se všemi usnadněními, které lze nabídnout.

Kapitola 4

Implementace aplikace

Na počátku implementace jsme stáli před rozhodnutím, jaké technologie a programovací jazyky zvolit. Volba padla na programovací jazyk Java a platformu *Java Standard Edition*(Java SE).

Programovací jazyk Java je objektově orientovaný a díky tomu, že vychází z jazyků C a C++, jejichž dobré vlastnosti se snaží zachovat a špatné odstranit či vylepšit, tak je jednodušší a je velmi robustní, dynamický a výkonný [36]. Jedná o jazyk interpretovaný, čili zdrojový kód se nekompiluje do strojového kódu, nýbrž do tzv. mezikódu, který je potom interpretován virtuálním strojem, který se nazývá *Java Virtual Machine* (JVM). Protože programovat v jazyce Java musí být možné na všech existujících platformách, musí být i JVM multiplatformní. Díky tomu jsou multiplatformní i všechny programy, které jsou v jazyce Java implementované. Jeden kód je možné použít na libovolné platformě, což je obrovská výhoda.

V pozdějších verzích jazyka Java již není mezikód interpretován. Využívá se tzv. kompilace v aktuálním čase (JIT – angl. *Just In Time Compilation*). Před prvním spuštěním programu je mezikód dynamicky zkompilován do strojového kódu daného počítače. V současné době se převážně používají technologie zvané *HotSpot compiler*. Ty mezikód zpočátku interpretují a na základě statistik získaných z této interpretace později provedou překlad do strojového kódu.

Platforma *Java SE* je kompletní prostředí pro vývoj aplikací. Do rodiny platformy *Java SE* patří dva produkty. Jedním je *Java SE Runtime Environment* (JRE), které obsahuje knihovny, JVM a další komponenty, které jsou nutné pro běh apletů a aplikací naprogramovaných v jazyce Java. Všechno tohle a k tomu ještě další nástroje pro vývoj aplikací, jako je např. překladač či ladicí program a další, které jsou nezbytné nebo vhodné pro vývoj apletů či aplikací, obsahuje produkt s názvem *Java SE Development Kit* (JDK) [1]. *Java SE* obsahuje velké množství knihoven, od základních, definující základní datové typy a operace nad nimi, přes matematické operace, V/V operace až po knihovny pro tvorbu uživatelských rozhraní a mechanismus zvaný *Java reflexe*.

Pro programovací jazyk Java existuje také několik vývojových prostředí (Netbeans, Eclipse, JBuilder), která ještě více programování aplikací v jazyce Java usnadňují. I my pro vývoj použijeme prostředí Netbeans, jelikož je možné jej získat jako součást balíčku spolu s JDK, čili je volně dostupné.

Programování aplikace bylo prováděno v několika fázích. Jednotlivé fáze jsou analogické s naplánováním projektu, které bylo popsáno v kapitole 3.1.3.

4.1 Forma uložení parametrů databází

Záhy po začátku implementace aplikace bylo potřeba definovat způsob uložení parametrů databází, jejichž seznam aplikace spravuje. Jak bylo uvedeno v kapitole 3.1.2, počet informací, které je nutné o každé databázi evidovat, není velký. Prakticky se jedná čistě o údaje, které jsou nezbytné pro vytvoření spojení s danou databází – název hostitele, číslo portu, identifikátor sezení (SID), přihlašovací jméno a heslo. Případně nějaká rozšířená nastavení.

Ačkoliv registrovaných databází může být teoreticky velké množství, je možné předpokládat, že jejich počet se bude pohybovat v řádu desítek. Také četnost operací nad seznamem bude v průběhu běhu aplikace minimální. Ani rychlost prováděných operací není rozhodujícím faktorem.

Vzhledem k uvedeným předpokladům a požadavkům na správu seznamu registrovaných databází, jsme se rozhodli použít jako paměťový modul klasický soubor namísto nějakého SŘBD. Toto rozhodnutí bylo učiněno také v korespondenci s formou uložení modulů aplikace (viz. kapitola 4.2). Výhodou uložení dat v souboru je možnost přímého nahlížení do něj a také přímé editace. Ikdyž to nelze označit za zcela správný přístup, jelikož ke správě databází i modulů slouží příslušní manažeři aplikace. Nicméně přímá editace nastavení nezpůsobí při chodu aplikace žádné problémy a aplikace na nastalé změny pružně reaguje.

Nejvhodnějším formátem souboru, do kterého budeme nastavení ukládat, je formát XML [3]. Díky jeho struktuře jsou data uložena strukturovaně a přehledně, a díky vhodně zvoleným názvům popisných značek lze okamžitě pochopit i sémantiku uložených dat.

Informace, které musíme o každé databázi znát, abychom se k ní mohli připojit, jsou:

- Název hostitele
- Číslo portu
- Identifikátor sezení (SID)
- Přihlašovací jméno
- Heslo

Dále musíme počítat se situací, kdy výše uvedené údaje nejsou postačující a je třeba definovat nějaká rozšířená nastavení.

Výslednou strukturu XML souboru lze spatřit na obrázku 4.1. Každou registrovanou databázi je potřeba v rámci aplikace jednoznačně identifikovat. Proto má každá databáze jednoznačný identifikátor (značka `idConn`) a každou databázi si může uživatel podle libosti pojmenovat (`connectionName`), přičemž i název musí být unikátní.

Aplikace umožňuje definovat dva způsoby definice databází. Oba dva jsou zobrazeny v ukázce. V pořadí první databáze (s názvem *LOCALHOST*) je definována základním způsobem. Má definovány všechny základní položky, které jsou pro ustavení připojení k ní potřeba. Bez povšimnutí jsme doteď přešli značky s názvy `isExpert` a `expert`. První z nich slouží jako signalizace, zda jsou pro danou databázi definována rozšířená nastavení či nikoliv (0 = NE, 1 = ANO). V případě, že jsou použita rozšířená (expertní) nastavení, tak jsou základní parametry, kromě přihlašovacího jména a hesla, ignorovány a je použit řetězec, který je uložen v elementu `expert`. Tento způsob definice parametrů je použit v ukázce u databáze s názvem *CISB*.

Seznam databází je uložen v souboru `APP_ROOT/config/connections.xml`.

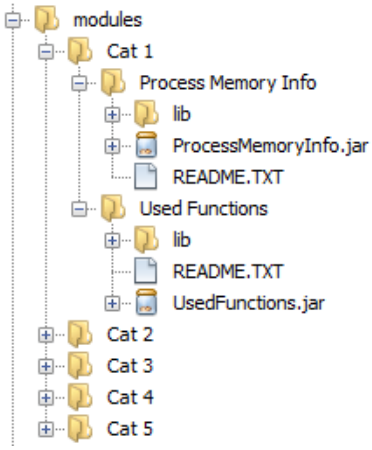
```
<?xml version="1.0" encoding="UTF-8"?>
<connections>
  <conn>
    <idConn>1</idConn>
    <connectionName>LOCALHOST</connectionName>
    <hostname>localhost</hostname>
    <port>1521</port>
    <SID>1</SID>
    <user>USER</user>
    <password>******</password>
    <isExpert>0</isExpert>
    <expert/>
  </conn>
  <conn>
    <idConn>2</idConn>
    <connectionName>CISB</connectionName>
    <hostname/>
    <port/>
    <SID/>
    <user>MIKULKA</user>
    <password>******</password>
    <isExpert>1</isExpert>
    <expert>
      (DESCRIPTION =
        (ADDRESS_LIST = (ADDRESS =  (PROTOCOL = TCP)
                                    (HOST = cisb.ro.vutbr.cz)
                                    (PORT = 1522)
                        )
        )
        (CONNECT_DATA = (SERVICE_NAME = cisb.ro.vutbr.cz)
                        (SERVER = DEDICATED)
        )
       )
    </expert>
  </conn>
</connections>
```

Obrázek 4.1: Ukázka uložení parametrů databází v XML souboru.

Z pohledu aplikace má plnou kontrolu nad seznamem databází databázový manažer. Ten se stará o to, aby byl tento seznam platný, generuje tzv. připojovací řetězce, díky nimž je vytvářeno připojení k dané databázi. Inicializuje všechna připojení, která jsou v rámci celé aplikace i jednotlivých modulů užívána a stará se i o jejich zničení.

4.2 Forma uložení modulů aplikace a jejich parametrů

Moduly jsou výkonnými jednotkami aplikace jako takové. Každý modul provádí monitorování určitých výkonnostních charakteristik databáze, ke které je připojený. Moduly musejí být, stejně jako databáze, které se mají sledovat, v aplikaci registrovány. Oproti seznamu databází je u modulů situace komplikovanější. Nestačí pouze znát pár informací o modulu, např. jak se jmenuje a jaké je číslo jeho verze, ale je třeba také uchovávat zdrojové soubory modulu, z nichž je modul spouštěn. Proto dělíme seznam modulů a kategorií na dvě části. Jedna část je seznam ve formě XML soubor, ve kterém se uchovávají základní informace. Soubor se nachází v `APP_ROOT/config/modules.xml`. Druhá část je místo na disku počítače, na němž je aplikace nainstalovaná. Zde se ukládají spustitelné soubory modulů. Kořenový adresář, ve kterém jsou moduly uloženy, je `APP_ROOT/modules`.



Obrázek 4.2: Ukázka adresářové struktury uložených modulů.

Moduly jsou organizovány v kategoriích. Máme tedy seznam kategorií a pro každou kategorii seznam modulů, které pod danou kategorii spadají. Kategorie slouží jako jakýsi pořadač, díky němuž lze moduly třídit, např. podle toho, jaké charakteristiky databáze zkoumají, apod. Z hlediska položek parametrů, které je potřeba o každé kategorii schraňovat, se jedná pouze o její název a identifikátor, obojí jsou unikátní. Tyto základní informace jsou uloženy v XML souboru. V korespondenci s tím existuje pro danou kategorii, v adresáři `APP_ROOT/modules`, podadresář, jehož název je identický s registrovaným unikátním názvem kategorie.

Pro každý modul musíme mít uloženy jeho spustitelné soubory. Každý modul je samostatná aplikace. Je uložena v podobě JAR archívu a patří k ní další adresáře a soubory, bez kterých by modul nebyl schopen fungovat (např. adresář `lib`, ve kterém jsou uloženy knihovny, které modul využívá). Celou strukturu adresářů a souborů modulu je třeba zapouzdřit do nadřazeného adresáře, který můžeme označit jako kořenový adresář modulu. Jeho název musí být stejný s názvem modulu, který je definován v XML souboru. Kořenový adresář modulu je umístěn v adresáři kategorie, ve které je modul registrovaný, viz. obrázek 4.2.

V XML souboru je o každém modulu uloženo jenom pár informací. Všechny jsou ale důležité. Název modulu je důležitý pro to, abychom našli adresář modulu se zdrojovými soubory, název hlavního JAR archívu a cestu k hlavní třídě abychom modul korektně spustili. Ukázka XML souboru, ve kterém se ukládají informace o modulech a kategoriích je na obrázku 4.3. Struktura je analogická s organizační strukturou modulů a kategorií. Synovské uzly kořenového prvku s názvem `modules` definují kategorie, uvnitř kategorie jsou definovány moduly.

Veškeré operace nad seznamem modulů a kategorií obstarává manažer modulů aplikace. Ten je také zodpovědný za to, že adresářová struktura modulů odpovídá seznamu kategorií a modulů uloženém v XML souboru. V případě porušení konzistence, dojde k odebrání modulu či celé kategorie ze seznamu. Konzistence může být porušena dvěma způsoby. Buď je odebrán modul či kategorie ze seznamu v XML souboru, avšak adresář modulu (kategorie) zůstane zachován. V tom případě je adresář modulu smázan, včetně obsahu. Nebo je modul v XML zachován, ale chybí adresář se zdrojovými soubory. Potom je daný modul odstraněn

```
<?xml version="1.0" encoding="UTF-8"?>
<modules>
  <category name="Cat 1" id="1">
    <module id="1">
      <jarfile>ProcessMemoryInfo.jar</jarfile>
      <name>Process Memory Info</name>
      <version>1.0</version>
      <mainclass>pmi/Main.class</mainclass>
    </module>
    <module id="2">
      <jarfile>UsedFunctions.jar</jarfile>
      <name>Used Functions</name>
      <version>1.0</version>
      <mainclass>uf/Main.class</mainclass>
    </module>
  </category>
  <category name="Cat 2" id="2"/>
  <category name="Cat 3" id="3"/>
  <category name="Cat 4" id="4"/>
  <category name="Cat 5" id="5"/>
</modules>
```

Obrázek 4.3: Ukázka uložení seznamu kategorií a modulů v XML souboru.

ze seznamu v XML souboru.

4.3 Modulární struktura aplikace

První fází implementace modulů a jejich zakomponování do aplikace, je nastudování možností vytvoření modulární aplikace a struktury modulárních aplikací v jazyce Java obecně. V této fázi bylo studováno několik možností.

Dokonce existují různé připravené sestavy (angl. *framework*), které poskytují obšírnou funkcionalitu pro tvorbu modulárních aplikací. Jednou z mnoha, na které jsme narazili, a o které se zmiňnuje většina internetových fór a stránek řešících danou problematiku, je soustava s názvem *Java Plug-in Framework* (JPF)[19]. Tato soustava je však, stejně jako ostatní, pro naši potřebu zbytečně robustní a komplikovaná. Proto jsme záhy od podobných soustav upustili a vydali se na dráhu zkoumání jednoduchých řešení, která budou pro naši modulární strukturu plně postačující a přitom nenáročná. A aby se výsledek co nejvíce blížil vizi uvedené v analýze požadavků v kapitole 3.1.1, kde se uvádí, že ideálním řešením je kombinace mechanismu Java reflexe a JAR archívu, coby spustitelného kódu modulu.

Po prostudování několika dokumentů (např. [2]) jsme se postupně propracovali ke konkrétnějším představám jak by mohla být modulární struktura řešena. Nalezli jsme a otestovali řešení, které počítá s využitím mechanismu Java reflexe v kombinaci s JAR archívem, jímž je modul definován, což plně odpovídá návrhu, jak by mohlo řešení vypadat. K tomu nám, do jisté míry, dopomohl článek na internetové stránce *Onyxbit*[28] a také vlákno známého fóra *StackOverflow*[35].

4.4 Spuštění a ukončení modulu

Proces spuštění a poté také ukončení modulu, byl nejsložitějším problémem, který byl v rámci implementace aplikace řešen. Aby byl modul korektně ukončen, je potřeba při

jeho spuštění dodržet určitá pravidla a postupy. Proto je problematiku spuštění a ukončení modulu nutno řešit společně.

Modul je samostatný JAR archív – samostatná aplikace. Z pohledu aplikace má modul neznámou strukturu a také není známa ani jeho funkčnost. Jediné, co o něm aplikace ví, je skutečnost, že je postaven na základech API, které je pro tvorbu modulů vytvořeno (viz. následující kapitola 4.5). Díky němu jsou stanovena určitá pravidla, díky nimž je aplikace schopna modul spustit.

V tomto ohledu nejvýznamnější pravidlo je to, že modul musí mít jednu třídu, kterou nazýváme hlavní třídou modulu. Ta je zděděna z abstraktní hlavní třídy modulu, jíž poskytuje zmíněná API. Hlavní třída má definované mechanismy, které jsou vyvolány při její inicializaci, čímž dojde ke spuštění modulu. Naším úkolem je vytvořit instanci hlavní třídy modulu. Cesta k hlavní třídě modulu je jednou z informací, které jsou uchovány v XML souboru coby seznamu modulů.

Kromě této informace ještě známe cestu k JAR archívu modulu, z nějž musíme hlavní třídu modulu načíst a vytvořit její instanci. Všechno musí být možné provést dynamicky za běhu aplikace. Jazyk Java má pro tyto potřeby mechanismus zvaný *Java reflexe*.

Po zvládnutí základních technik načítání a instanciace tříd bylo potřeba implementovat způsob jak hlavní třídu modulu získat z JAR archívu. Způsob načtení třídy a vytvoření její instance byl převzat z internetu [15][16][18]. Toto řešení vkládá cesty k třídám, které se mají načíst do tzv. *CLASSPATH* virtuálního stroje JVM. Při načtení třídy pak JVM prochází obsah adresářů v *CLASSPATH* a načte požadovanou třídu pouze v případě, že ji v některé ze zadaných cest nalezne.

V počátcích vývoje modulů, kdy ještě nebylo třeba žádných knihoven, které modul využíval, použité řešení stačilo. Problém nastal v momentě, kdy už byly pro činnost modulu použity různé knihovny. Tyto knihovny se shromažďují do adresáře `lib`, který se nachází na úrovni JAR archívu modulu. Modul od té doby nebylo možné načíst. Díky výpisům ladicího programu jsme sice věděli, kde máme problém hledat. V kontextu s předchozím odstavcem jsme se nejprve pokusili přidat do *CLASSPATH* i cestu k adresáři `lib` daného modulu, ale bez úspěchu. Při hledání informací na internetu jsme se v na fóru *VelocityReviews*[9] dozvěděli, že problém může být způsoben nesprávným obsahem souboru `MANIFEST.MF`, který se vytváří při generování JAR archívu v době kompilace modulu. A opravdu to tak bylo. Kámen úrazu byl v tom, že v uvedeném souboru nebyl uveden parametr `Class-Path`, za kterým následoval výčet cest ke knihovnám modulu.

Další problém vyplul na povrch, dá se říct, náhodou, když při testování aplikace bylo, po spuštění a následném ukončení jakéhokoliv modulu, nemožné z pevného disku smazat jeho spustitelné soubory (JAR archív, atd.), což je součástí operace vedoucí k odebrání modulu ze seznamu modulů.

Bylo zjištěno, že za to může načítání souborů do *CLASSPATH*. Řešením je použití tzv. zavaděče tříd[1], kterému se zadá cesta k souboru modulu a poté název hlavní třídy modulu. On provede její načtení a vytvoření instance. Nejvhodnější zavaděč tříd v jazyce Java je třída s názvem `java.net.URLClassLoader`, kterou používáme. Podle diskusí je však k úspěšnému uvolnění souborů z *CLASSPATH* nutné nejprve odstranit a uklidit veškeré instance tříd, které jsou v daných souborech (v našem případě se jedná o třídy obsažené

[1] angl. Class Loader – Používá se při potřebě načtení třídy za běhu programu.

v JAR archívu modulu nebo jemu přidružených knihoven) a všech jejich referencí. Potom musí být také uklizen samotný zavaděč.

A v uklizení zavaděče byl další problém. Jak informují různá fóra (např. [42]), ve svém blogu John Mazz [30] a konečně i sama společnost *Sun Microsystems* skrze článek Michaela McMahona [31] s odkazem na záznam do databáze chyb[43] – jakmile dojde k uvolnění všech zdrojů zavaděče, není možné předpovědět, kdy dojde k tzv. finalizaci[2] a tím k opravdovému odstranění objektu pomocí tzv. sběrače smetí[3]. Proto se může stát, že pokud je potřeba smazat soubor, jenž byl zavaděčem načten, nemusí být tento doposud uvolněn z JVM a nebude možné jej smazat.

My však musíme být bezpodmínečně schopni umožnit uživateli po ukončení modulu odstranit jej ze seznamu registrovaných modulů. Uvedený problém se týká především operačních systémů Windows, které neumožňují smazat nebo přepsat otevřené soubory. Řešení ale naštěstí nabízí opět sama společnost *Sun Microsystems*, která dává, na zmíněných internetových stránkách, k dispozici kód metody `close()`[4], jež má být definována ve třídách zděděných z třídy `java.lang.ClassLoader` a jež se má, při jejím zavolání, postarat o okamžité uvolnění svých zdrojů.

Tím byl největší problém, který se týká jak spuštění, tak i ukončení modulů, vyřešen. Pro jistotu jsme se však rozhodli při spouštění modulů kopírovat jejich spustitelné soubory do dočasného adresáře a teprve z něj požadované třídy načítat. Díky této kombinaci se nemůže stát, že nebude možné při odstranění modulu ze seznamu smazat jeho soubory z disku počítače.

Při řešení tohoto problému došlo také k nezištnému vyřešení dalšího. Díky tomu, že jsme donuceni při ukončení každého modulu zničit instance všech jeho tříd i jejich reference, dochází tím k uvolnění operační paměti, kterou modul spotřeboval. Dokud nebyly moduly uvolňovány, se stoupajícím počtem spouštěných modulů rostla spotřeba paměti, která se po jejich ukončení nezmenšovala. Což by časem vedlo k zaplnění veškeré operační paměti a ke zpomalení chodu aplikace či dokonce jejímu pádu, ale i ke zmenšení výkonu celého počítače. Díky vyřešení problému spuštění a ukončení modulu, které v sobě obsahuje i zničení již nepotřebných tříd, je z pamětí počítače zacházeno hospodárně.

4.5 API pro implementaci modulů

Každý modul je relativně samostatný program. Z pohledu toho, jak se implementuje, tak tomu tak je. Nicméně z hlediska samostatnosti fungování samostatný není. Je naprogramován tak, aby nezávisle na svém okolí poskytoval požadovanou funkcionalitu. Ikdyž v sobě aplikace moduly registruje, tak vůbec netuší, jak modul pracuje a pro jaký účel je vytvořen.

Ačkoliv je modul nezávislá výpočetní jednotka, nemůže být schopen fungovat mimo prostředí aplikace. Ta mu totiž poskytuje velké množství funkcí, bez kterých se modul při běhu neobejde. Každý modul tedy musí být schopen s aplikací komunikovat.

Aby to bylo možné, musí být na obou stranách komunikace dopředu definován a znám jazyk, kterým mezi sebou mohou aplikace a modul komunikovat. Proto je pro moduly potřeba vytvořit API. Aby byl modul schopen v rámci aplikace fungovat, musí být postaven

[2] finalizace – skutečné uvolnění paměti daného objektu pomocí sběrače smetí.
[3] viz. http://cs.wikipedia.org/wiki/Garbage_collector
[4] Metoda `close()` je již součástí JDK7, build 48.

na základech tohoto API. Pokud se tak stane, má programátor modulu jistotu, že modul lze do aplikace zabudovat a že spolupráce aplikace a modulu bude korektní.

API můžeme označit jako jádro modulu. Implementuje rozhraní pro práci s databázemi – umí pro modul inicializovat připojení k vybrané databázi, poskytuje rozhraní pro ovládání grafických prvků modulu a pro ovládání vlákna, které modul využívá pro svůj běh.

Základní součástí API je abstraktní hlavní třída modulu. Z této třídy musí být vždy zděděna hlavní třída modulu! Ta obsahuje abstraktní metodu `public void proved'()`, která definuje co má vlákno modulu provádět.

Dále API obsahuje grafickou komponentu, z níž má být odvozeno okno modulu. GUI musí být nedílnou součástí každého modulu. Okno modulu je po jeho spuštění umístěno na plochu aplikace a spolu s dalšími prvky slouží pro ovládání modulu.

API obsahuje také kontroléry. Jeden slouží pro operace nad databázovými připojeními modulu a je napojen na kontrolér aplikace. Druhý kontrolér slouží jako prostředník mezi GUI modulu a aplikace. Vždy když v modulu dojde k nějaké akci ze strany uživatele, tento kontrolér danou zprávu, pokud je to nutné, předá aplikaci. Pokud aplikace potřebuje provést akci uvnitř GUI modulu, použije také tohoto kontroléru.

Poslední kontrolér poskytuje operace pro ovládání výkonné jednotky modulu – vlákna. Umožňuje jeho start, pozastavení, opětovný start i úplné zastavení. Ačkoliv API modulu dává každému modulu do vínku jedno vlákno, které může pro svůj běh využívat, záleží čistě na programátorovi, jestli jej využije. Modul jeho vlastní výkonnou jednotku nemusí potřebovat vůbec. Může mu stačit tady tahle, ale také si může vytvořit své vlastní. Ty už jsou ale plně v jeho režii.

Podoba API se utvářela v průběhu celého vývoje aplikace. Jednotlivé části byly postupně doplňovány o nové a nové funkcionality, aby API poskytovalo vše co modul pro správnou funkci může potřebovat. V současné době je API zapozdřeno v JAR archívu, který lze jednoduše přidat jako knihovnu do projektu, ve kterém se modul vyvíjí.

4.6 Další řešení

Stav databáze a její vlastnosti, jež se moduly snaží zobrazit, jsou čistá data, ve většině případů číselných hodnot. Proto je vhodné, pokud to jde, zobrazit je v grafické podobě, která je člověku čitelnější. Pro tyto účely bylo v modulech použito pro některá data znázornění pomocí grafů. Bylo tedy třeba zvolit některou z dostupných knihoven, kterou by bylo možné pro tyto účely využít. Snažili jsme se vybrat co nejjednodušší knihovnu, která má jednoduché použití a není třeba dlouhého nastudování. Také bylo potřeba vzít v potaz licenční podmínky jednotlivých knihoven. Po zmapování internetu byly do užšího výběru vybrány čtyři knihovny – *JFreeChart*[21], *Ptolemy II Ptplot*[4], *jCharts*[6] a *JCharts2D*[20]. Po zvážení všech pro a proti jsme se nakonec rozhodli pro knihovnu *JCharts*, která se zdá nejjednodušší, splňuje všechny požadavky na podobu grafů.

Méně důležitou, avšak mnohdy velmi diskutovanou stránkou aplikace je její vzhled, který je součástí tzv. uživatelsky přívětivého prostředí. Jeho úkolem je co nejvíce uživateli zjednodušit a zatraktivnit práci s programem. V tomto duchu bylo použito značné množství prvků na implementační úrovni. Mimo to však bylo potřeba prvky GUI aplikace i modulů vhodně vybavit doplňky – ikonami.

Na internetu je dnes nepřeberné množství portálů, na kterých lze stáhnout nejrůznější motivy. My jsme se snažili držet konzervativního stylu, jelikož je potřeba upřednostnit funkcionalitu a jednoduchost ovládání. K tomu přispívají i vhodně zvolené ikony, když na první pohled dávají uživateli tušit, k čemu dané tlačítko slouží. Proto byly pro různé účely zvoleny dva druhy ikon – *Crystal Office Icon Collection*[33] a *1000 Free Farm-Fresh Web Icons*[29]. Oba soubory ikon splňují jak požadavek na grafický vzhled, tak také licenční politiku, aby bylo možné je použít.

Licenční politika obecně je při vývoji aplikace velmi důležitá a setkáváme se s ní prakticky každodenně, chceme-li využít dostupných knihoven, které nám vývoj aplikace usnadní. Je nutné brát v potaz účel aplikace, pro který bude používána a teprve na jeho základě vybírat knihovny povolující použití pro takové účely. V opačném případě je porušen zákon!

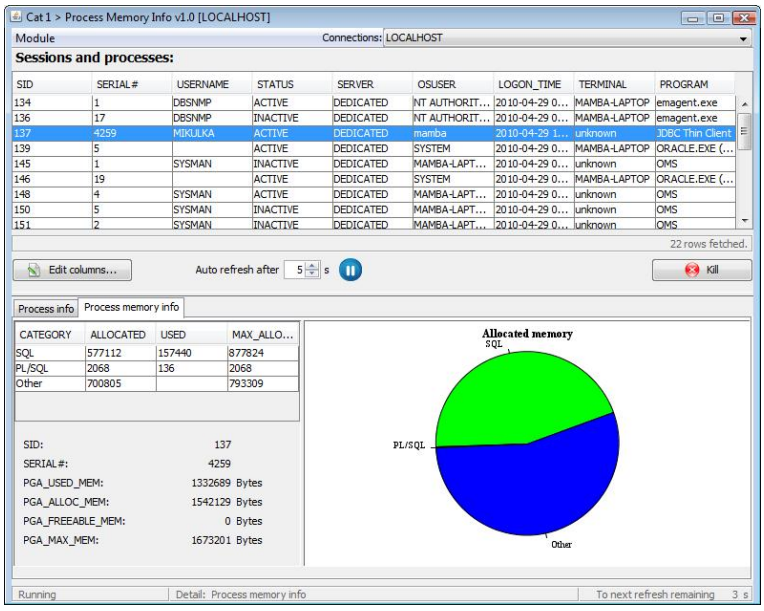
Kapitola 5

Moduly

Moduly jsou výkonným jádrem aplikace, které chod databází zkoumají. Činnost modulu je závislá na požadavcích toho, kdo aplikaci používá a jaké statistiky a charakteristiky databáze potřebuje analyzovat. Modulů může být tedy nekonečně mnoho a pro všelijaké účely.

Po konzultaci s vedoucím a konzultantem této diplomové práce jsme vytvořili čtyři moduly, o nichž můžeme říct, že slouží pro monitorování základních veličin, které mají na chod databáze vliv. Všechny statistiky jsou získávány z pohledů mechanismu *Wait Interface*.

5.1 Modul „Process Memory Info“



Obrázek 5.1: Snímek běžícího modulu „Process Memory Info“.

První modul, jenž má pracovní název *Process Memory Info*, zobrazuje otevřené relace databáze, ke které je připojen. O každé relaci pak dává k dispozici detailní informace o procesu a o jeho spotřebě paměti.

Na obrázku 5.1 je modul zobrazen v době běhu. Nahoře v liště s menu je výběrové tlačítko, ve kterém je seznam databází, ke kterým je aplikace připojena a ke kterým lze připojit i tento modul. Toto výběrové tlačítko i jeho umístění je stejné pro všechny moduly.

V horní části okna je tabulka, která zobrazuje akutálně běžící sezení databáze. Data plnící tuto tabulku jsou získávána z pohledu *v$session*. Dotaz, který tato data si generuje automaticky sloupce, které se mají zobrazit podle toho, které jsou uživatelem vybrány. Po spuštění jsou nastaveny pro zobrazení vybrané sloupce. Výchozí dotaz je:

```
SELECT
  sid, serial#, username, status,
  server, osuser, logon_time,
  terminal, program
FROM
  v$session;
```

Pod tabulkou je ovládací panel, v jehož střední části se nachází mechanismus pro snadnou úpravu časového intervalu, po jehož naplnění dochází k automatické aktualizaci již zmíněné hlavní tabulky.

V levé částí ovládacího panelu se nachází tlačítko, které otevírá okno editoru sloupců, které se zobrazují v hlavní tabulce. Zde je udržován seznam všech sloupců, které je možné v tabulce zobrazit, přičemž uživatel sám si může zvolit, které sloupce se mají zobrazit. Seznam sloupců pohledu *v$session* se získá dotazem:

```
SELECT
  column_name
FROM
  dba_tab_columns
WHERE
  owner = 'SYS'
  AND table_name = 'V_$SESSION';
```

Na pravé straně panelu se nachází tlačítko *Kill*, které umožňuje ukončit sezení, které je zvoleno v horní tabulce. Tlačítko je po vybrání sezení aktivováno pouze pokud má uživatel, který se k databázi přihlásil, má práva sezení ukončit. To se zjistí po přihlášení daného uživatele dotazem:

```
SELECT
  COUNT(1) AS can_kill
FROM
  user_sys_privs
WHERE
  privilege = 'ALTER SESSION';
```

Po stisku tlačítka *Kill* se ukončení sezení provede dotazem:

```
ALTER SYSTEM KILL SESSION ':sid, :serial#';
```

V dolní polovině modulu jsou zobrazeny detailní informace o konkrétním sezení. Tyto se zobrazují pouze v případě, že je označeno některé sezení v horní tabulce. Detail je rozdělen na dvě záložky. První obsahuje několik informací o procesu sezení – PID, PID v rámci Oracle, zda proces běží na pozadí, apod. Dotaz, kterým data získáváme je:

```
SELECT
  pid AS oracle_pid,
  spid AS os_ppid,
  username AS os_username,
  background
FROM
  v$process
WHERE
  addr = ':addr';
```

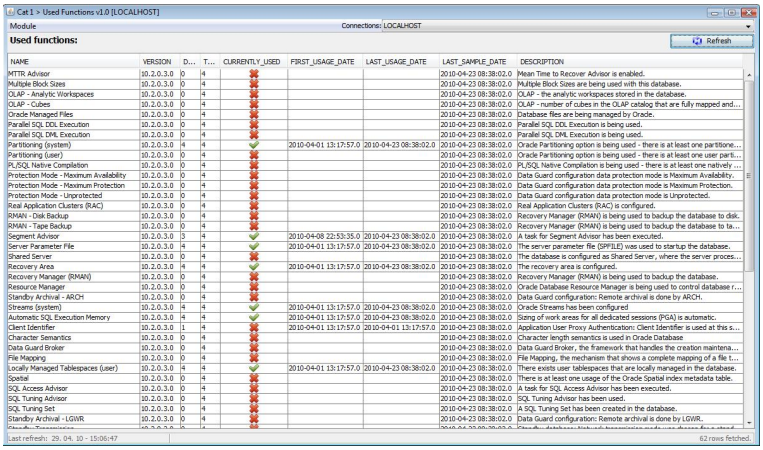
Druhá část zobrazující informace o využití paměti procesem je členitější. V levé části zobrazuje v číselné formě kolik paměti má proces alokováno, kolik jí využívá, jaké je použité maximum. V pravé části je v grafické podobě zobrazeno, která kategorie dotazů používá kolik paměti. Data do tabulky tohoto detailu jsou získávána pomocí dotazu:

```
SELECT
  pm.category, pm.allocated, pm.used, pm.max_alocated
FROM
  v$process AS p, v$process_memory AS pm
WHERE
  p.pid = pm.pid
  AND p.serial# = pm.serial#
  AND p.addr = ':addr';
```

Vpravo v grafu je koláčový graf zobrazující sloupec `pm.allocated` podle kategorií. Data zobrazená pod tabulkou jsou získána dotazem:

```
SELECT
  s.sid, s.serial#,
  p.pga_used_mem, p.pga_alloc_mem, p.pga_freeable_mem, p.pga_max_mem
FROM
  v$process AS p, v$session AS s
WHERE
  s.paddr = p.addr
  AND s.paddr = ':addr';
```

V příloze C je k dispozici zvětšený snímek modulu a také ukázka editoru sloupců hlavní tabulky modulu, stejně jako další ukázky ostatních modulů.



Obrázek 5.2: Snímek běžícího modulu „Used Functions“.

5.2 Modul „Used Functions“

Další modul zobrazuje seznam funkcí, které jsou aktuálně databází používány.

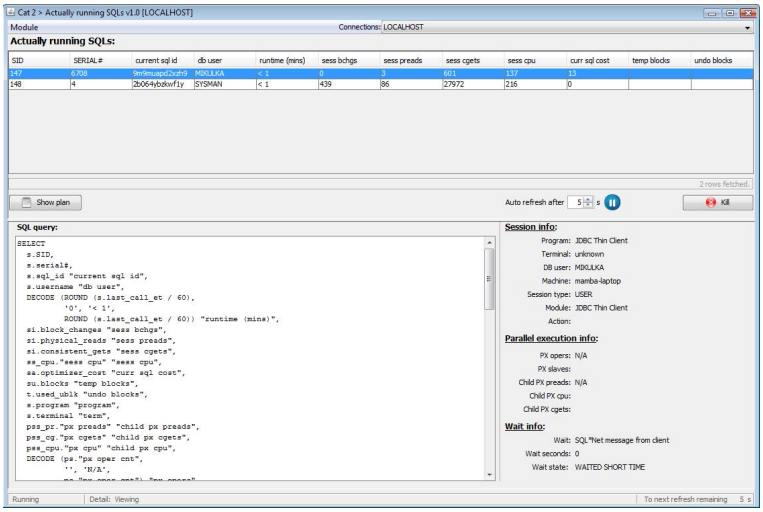
Tento modul je oproti předchozímu strožejší. Pro obnovení obsahu zde slouží tlačítko, což je v případě tohoto modulu postačující.

Modul uživateli poskytuje přehled využití funkcí, které Oracle databáze nabízí. Modul je vhodný pro kontrolu funkcí, které jsou licencované a používají se, a naopak těch, které licencované nejsou a také se používají.

Dotaz, který získává seznam používaných funkcí je:

```
SELECT
  fu.name, fu.detected_usages,
  samp.version, samp.total_samples
  DECODE(TO_CHAR(fu.last_usage_date, 'MM/DD/YYYY, HH:MI:SS'),
         NULL, 'FALSE',
         TO_CHAR(fu.last_sample_date, 'MM/DD/YYYY, HH:MI:SS'),
         'TRUE', 'FALSE') AS currently_used,
  fu.first_usage_date,
  fu.last_usage_date,
  samp.last_sample_date,
  mt.description
FROM
  sys.wri$_dbu_usage_sample AS samp,
  sys.wri$_dbu_feature_usage AS fu,
  sys.wri$_dbu_feature_metadata AS mt
WHERE
  samp.dbid = fu.dbid
  AND samp.version = fu.version
  AND fu.name = mt.name
  AND fu.name NOT LIKE '_DBFUS_TEST%'
  AND BITAND(mt.usg_det_method, 4) != 4
```

5.3 Modul „Actually Running SQLs“



Obrázek 5.3: Snímek běžícího modulu „Actually Running SQLs“.

Další modul, který jsme v rámci této práce implementovali, zobrazuje seznam SQL dotazů a jejich statistik, které jsou momentálně nad databází prováděny. Statistiky jsou automaticky aktualizovány za použití stejného principu jako u modulu *Process Memory Info*. Celková struktura modulu je velmi podobná, včetně rozlože-ní grafických prvků. V detailu se pro vybraný řádek hlavní tabulky zobrazují v textovém poli SQL dotaz, jež je formátován tak, jak byl databází přijat. V pravé části se zobrazují informace o sezení, ve kterém byl dotaz prováděn, o stavu čekání na vyřízení tohoto dotazu, apod.

Všechna tato data včetně detailních informací se získávají dotazem pro svou obsáhlost uvedeným v příloze B.

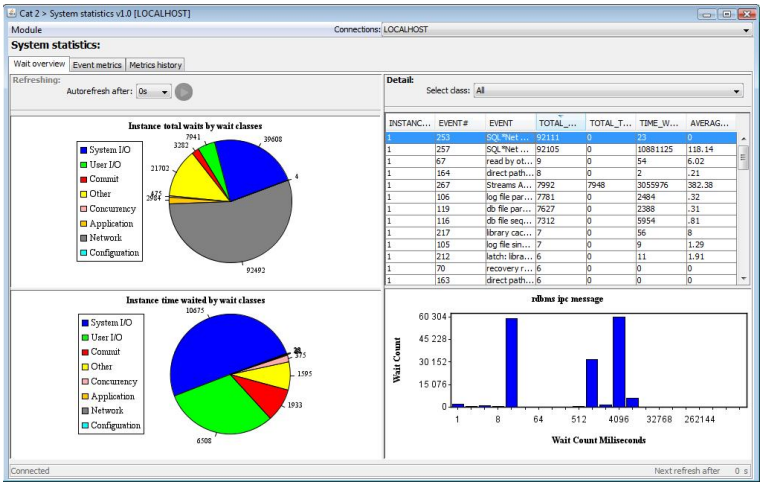
Tlačítko na levé straně ovládacího panelu slouží pro zobrazení plánu SQL dotazu, který databáze před jeho provedením zpracovala, v dialogovém okně – viz. příloha C, obrázek C.5. Plán SQL dotazu se z databáze získá SQL dotazem:

```
SELECT
  *
FROM
  TABLE(dbms_xplan.display_cursor(':sql_id', :sql_child_number));
```

Modul zobrazuje seznam aktuuálně běžících SQL dotazů, informace o databázové relaci, která SQL dotaz spustila, v detailu zobrazuje text SQL dotazu a v dialogovém okně jeho plán vykonání SQL dotazu, který se používá pro případné ladění dotazu.

5.4 Modul „System Statistics“

Poslední modul je nejkomplexnější a nejrobustnější implementovaný modul. Je rozdělen na tři tématické celky, které jsou graficky odděleny záložkami.



Obrázek 5.4: Snímek běžícího modulu „System Statistics“.

První část zobrazuje základní přehled statistik mechanismu *Wait Interface*, popsaného v kapitole 2.2. Pomocí grafů znázorňuje dvě základní statistiky. První graf zobrazuje počet čekání procesů na výpočet podle tříd čekání. Dotaz, podle kterého se data z databáze získávají je:

```
SELECT
  wait_class,
  total_waits,
  ROUND(100 * (time_waited / sum_time), 2) AS pct_time,
FROM
  (SELECT
     wait_class,
     wait_class#,
     total_waits,
     time_waited
   FROM
     v$system_wait_class
   WHERE
     wait_class != 'Idle'),
  (SELECT
     SUM(total_waits) AS sum_waits,
     SUM(time_waited) AS sum_time
   FROM
     v$system_wait_class
   WHERE
     wait_class != 'Idle')
ORDER BY 3 DESC;
```

Druhý graf ukazuje celkový čas strávený čekáním, opět podle tříd čekání – dotaz:

```
SELECT
  wait_class,
  time_waited,
  ROUND(100 * (time_waited / sum_time), 2) AS pct_time,
FROM
  (SELECT
     wait_class,
     wait_class#,
     total_waits,
     time_waited
   FROM
     v$system_wait_class
   WHERE
     wait_class != 'Idle'),
  (SELECT
     SUM(total_waits) AS sum_waits,
     SUM(time_waited) AS sum_time
   FROM
     v$system_wait_class
   WHERE
     wait_class != 'Idle')
ORDER BY 3 DESC;
```

Pro konkrétní třídu čekání lze také zobrazit seznam čekacích událostí dotazem:

```
SELECT
  se.INST_ID as INSTANCE_NUMBER,
  en.EVENT#,
  se.EVENT,
  se.TOTAL_WAITS,
  se.TOTAL_TIMEOUTS,
  se.TIME_WAITED,
  se.AVERAGE_WAIT,
  se.TIME_WAITED_MICRO
FROM
  gv$system_event AS se, gv$event_name AS en
WHERE
  se.INST_ID = 1
  AND se.INST_ID = en.INST_ID
  AND en.EVENT_ID = se.EVENT_ID
  AND se.wait_class# = :classID
ORDER BY time_waited_micro DESC;
```

Při výběru konkrétní čekací události se zobrazí její histogram znázorňující počet čekání a čas strávený čekáním. Vytažení příslušných dat provedeme dotazem:

```
SELECT
  eh.event,
  eh.wait_time_milli,
  eh.wait_count
FROM
  gv$event_histogram AS eh
WHERE
  eh.EVENT# = :eventID
  AND eh.INST_ID = 1
ORDER BY 2,3;
```

Ve druhé části se zobrazuje seznam metrik čekacích událostí, který se může automaticky obnovovat. Seznam sbírá data podle dotazu:

```
SELECT
  mh.INST_ID AS instance_number,
  TO_CHAR(mh.begin_time,'dd.mm.yyyy hh24:mi') AS begin_time,
  TO_CHAR(mh.end_time,'dd.mm.yyyy hh24:mi') AS end_time,
  mh.ENTITY_ID,
  en.NAME,
  en.WAIT_CLASS,
  en.WAIT_CLASS#,
  mh.metric_id,
  mh.metric_name,
  mh.value || ' ' || mh.metric_unit,
  en.EVENT#
FROM
  gv$metric_history AS mh,
  gv$metricgroup AS mg,
  v$event_name AS en
WHERE
  mh.inst_id = mg.inst_id
  AND mh.group_id = mg.group_id
  AND mg.group_id = 0
  AND mh.ENTITY_ID = en.EVENT#
  AND en.WAIT_CLASS# != 6
  AND en.wait_class# = :classID
  AND en.event# = :eventID
ORDER BY begin_time, metric_id;
```

Po vybrání zvolené události se v detailu zobrazí histogram vybrané metriky podle dotazu:

```
SELECT
  eh.inst_id as instance_number,
  eh.EVENT#,
  eh.event,
```

```
  eh.wait_time_milli,
  eh.wait_count
FROM
  gv$event_histogram AS eh
WHERE
  eh.event# = " + eventID + " " +
  AND eh.event IN (SELECT
                     e.name
                   FROM
                     v$event_name AS e
                   WHERE
                     e.wait_class# != 6)
ORDER BY 1,3,4;
```

U tohoto modulu je speciální to, že každý ze dvou autoobnovovacích mechanismů v tomto modulu je nezávislý. Pod panelem ovládajícím autoobnovu se nachází panel umožňující uživateli filtrovat data zobrazená v tabulce podle dvou kritérií. Jedním může být filtrace dat podle třídy čekání a (nebo) podle čekací události. Uživatel má možnost vybírat do filtru čekací události podle libosti. Provádí se ve speciálním editoru, viz. obrázek C.8.

Poslední část zobrazuje historická data metrik. SQL dotaz, který čerpá data má podobu:

```
SELECT
  dhss.instance_number,
  TO_CHAR(dhss.begin_time,'dd.mm.yyyy hh24') as begin_time,
  TO_CHAR(dhss.end_time,'dd.mm.yyyy hh24') as end_time,
  dhss.metric_id,
  dhss.metric_name,
  dhss.metric_unit,
  TRUNC(dhss.minval,4) AS minval,
  TRUNC(dhss.maxval,4) AS maxval,
  TRUNC(dhss.average,4) AS average,
  TRUNC(dhss.standard_deviation,4) AS stddeviation
FROM
  dba_hist_snapshot AS dhs,
  dba_hist_sysmetric_summary AS dhss,
  dba_hist_metric_name AS dhmn
WHERE
  dhs.snap_id = dhss.snap_id
  AND dhs.dbid = dhss.dbid
  AND dhs.instance_number = dhss.instance_number
  AND dhss.group_id = dhmn.group_id
  AND dhss.dbid = dhmn.dbid
  AND dhss.metric_id = dhmn.metric_id
  AND dhss.instance_number = ':instanceNumber'
  AND dhss.metric_id = :metricID
  AND TO_CHAR(dhss.begin_time,'dd.mm.yyyy hh24') >= ':begin_time'
  AND TO_CHAR(dhss.end_time,'dd.mm.yyyy hh24') <= ':end_time'
ORDER BY begin_time, metric_id;
```

Při vybrání metriky v hlavní tabulce se v detailu zobrazí histogram s počtem čekání v závislosti na čase čekání pro danou metriku. Viz. následující dotaz:

```
SELECT
  dhss.instance_number,
  TO_CHAR(dhss.begin_time, 'dd.mm.yyyy hh24') AS begin_time,
  TRUNC(dhss.average,4) AS average,
  TRUNC(dhss.standard_deviation,4) AS stddeviation,
  TRUNC(dhss.minval,4) AS minval,
  TRUNC(dhss.maxval,4) AS maxval,
  dhss.metric_name,
  dhss.metric_unit,
  dhss.metric_id
FROM
  dba_hist_snapshot AS dhs,
  dba_hist_sysmetric_summary AS dhss,
  dba_hist_metric_name AS dhmn
WHERE
  dhs.snap_id = dhss.snap_id
  AND dhs.dbid = dhss.dbid
  AND dhs.instance_number = dhss.instance_number
  AND dhss.group_id = dhmn.group_id
  AND dhss.dbid = dhmn.dbid
  AND dhss.metric_id = dhmn.metric_id
  AND dhss.metric_id = :metricID
  AND TO_CHAR(dhss.begin_time,'dd.mm.yyyy hh24') >= ':begin_time'
  AND TO_CHAR(dhss.end_time,'dd.mm.yyyy hh24') <= ':end_time'
ORDER BY begin_time;
```

Tato část již neobsahuje mechanismus pro automatické obnovování dat. Data lze filtrovat, konkrétně podle datového intervalu či instance či metrik samotných, jejichž seznam pro filtr se dá definovat analogicky jako filtr čekacích událostí v prostřední části (obrázek C.10). Zajímavostí této části je grafické znázornění celkové historie pro zvolenou metriku. Ta si pamatuje ve kterém časovém okamžiku je zvolený řádek metriky v grafu zobrazen a na této pozici vykreslí žlutou svislou čáru. Tato drobnost je velmi užitečná v případě, kdy se snažíme zjistit, která statistika odpovídá kterému okamžiku zobrazenému v grafu.

Modul zobrazuje systémové statistily z mechanismu *Wait Interface*. Zobrazuje jednotlivé čekací události a jejich histogram podle jednotlivých tříd čekání, což umožňuje zjistit, ve kterých situacích databáze nejvíce čeká. Histogram ukazuje rozložení, resp. počet čekání a jak dlouho daná skupina čekala.

Kapitola 6

Nasazení aplikace a testování

Aplikaci jsme nasadili na produkční databázi, abychom zjistili, zda je má ambice stát se používáným nástrojem pro sledování chování databáze v ostrém provozu. Jedná se o centrální databázi VUT v Brně – databázový shluk, jenž se skládá ze tří serverů. Po nasazení aplikace byla zjištěna následující úzká místa sledované databáze.

6.1 Problém „*cursor: pin S wait on X*“

Při monitorování databáze bylo zjištěno, že velkou část čekání databáze tvořily události *„cursor: pin S wait on X“* (viz. obrázek 6.1) a *„library cache pin“*. Tyto dvě události spolu úzce souvisejí a znamenají v podstatě totéž (V Oracle 10gR2 byla událost *„library cache pin“* nahrazena algoritmy vzájemného vyloučení[1], které jsou podrobnější).

Událost *„cursor: pin S wait on X“* je v převážné většině spojena se vzájemným vyloučením a tzv. hrubou analýzou.

Hrubá analýza znamená vykonávání SQL dotazů, které nejsou uloženy ve speciální vyrovnávací paměti Oracle databáze zvané *shared pool*. Paměť *shared pool* je část paměti *System Global Area* (SGA) Oracle databáze, ve které jsou, mimo jiné, uloženy:

- optimalizované plány SQL dotazů
- bezpečnostní kontroly
- analyzované SQL dotazy
- balíčky
- informace o objektech

Paměť *shared pool* je rychlá vyrovnávací paměť, díky níž jsou SQL dotazy prováděny rychleji.

[1] angl. mutual exclusion, mutex – Algoritmy, které se snaží zabránit vícenásobnému přístupu do paměti více procesy najednou. Viz. http://en.wikipedia.org/wiki/Mutual_exclusion

Obrázek 6.1: Úzké místo produkční databáze, které bylo zjištěno vytvořenou aplikací.

Při použití hrubé analýzy musí databáze vykonat určité operace navíc. Především musí

- alokovat paměť pro vykonávaný SQL dotaz v paměti *shared pool*
- zkontrolovat syntaktickou správnost SQL dotazu
- provést kontrolu, zda uživatel vykonávající SQL dotaz má oprávnění nezbytná pro jeho vykonání

Databáze také musí pro tento SQL dotaz získat exkluzivní přístup do paměti *Library cache pin*. To znamená, že proces, který dotaz vykonává, musí získat exkluzivní přístup do kritické sekce. Zároveň jiný proces vykonávající stejný SQL dotaz požaduje také exkluzivní přístup do stejné kritické sekce, ale ten nemůže získat, protože je používána jimým procesem.

Hrubá analýza SQL dotazu je velmi drahá operace. Je náročná jak z hlediska využití CPU, tak i z hlediska počtu přístupů do paměti, které jsou nezbytné pro vykonání dotazu. Je proto doporučeno a nanejvýše vhodné se tomuto způsobu vykonání SQL dotazu vyhnout kdykoliv je to možné.

Redukce tohoto čekání je značně závislá na tom, jaký scénář blokací je prováděn. Běžným problematickým scénářem je používání dynamických SQL dotazů[2] uvnitř PL-SQL procedur, kde PLSQL kód je překompilován a dynamický SQL dotaz volá něco, co je závislé na volající proceduře. Pokud je toto čekání rozšířené paušálně, může pomoct vyladění paměti *shared pool*.

6.2 Problém „*db file sequential read*“

Dále bylo zjištěno, že databáze také čeká na událost *„db file sequential read“* (viz. obrázek 6.2). Tato událost říká, že databáze je nucena číst data z datového souboru databáze, namísto vyrovnávací paměti. Přestože téměř všechna tato čekání skončila v čase do jedné milisekundy, jedná se o nechtěný jev.

Čtení dat přímo z datových souborů je nevyhnutelné. Cílem, který problém odstraní nebo alespoň zmírní počet těchto čekání, je minimalizace V/V operací, které není nutné provádět. Toho lze nejlépe dosáhnout co nejlepším návrhem databázové aplikace a vytvářením efektivních plánů vykonávání SQL dotazů. Drobné změny na systémové úrovni mohou vylepšit výkonnost o pár procent, narozíl od toho úpravy plánů vykonávání SQL dotazů může výkonnost zlepšit značně.

Pro odstranění či zmírnění dopadů této čekací události může pomoci:

- Kontrola SQL dotazů využívajících neselektivní skenování indexů.
- Zvýšení velikosti vyrovnávací paměti – Pro zvýšení velikosti se má použít parametr DB_BLOCK_BUFFERS, nikoliv DB_BLOCK_LRU_EXTENDED_STATISTICS. Nikdy však nezvyšujme velikost paměti zvané *System Global Area* (SGA), pokud to může vyvolat dodatečně stránkování systému.

[2] angl. Dynamic SQL – jsou to SQL dotazy, jejichž znění je známo až za běhu programu, viz. http://www.cs.umbc.edu/portal/help/oracle8/server.815/a68022/dynsql.htm#588

Obrázek 6.2: Úzké místo produkční databáze, které bylo zjištěno vytvořenou aplikací.

- Uložení dat ve fyzických datových blocích. Předpokládejme, že často získáváme data z nějaké tabulky podle jednoho sloupce, kde jeho hodnoty jsou v určitém rozsahu, pomocí snímání na základě indexu. Za předpokladu, že jeden blok indexu obsahuje 100 záznamů, mohou nastat dva extrémní případy:
  - Každý řádek tabulky je fyzicky uložen v jiném datovém bloku (Pro každý blok indexu musí být přečteno 100 datových bloků).
  - Všechny řádky tabulky jsou uloženy v několika po sobě jdoucích blocích (Pouze hrstka datových bloků je čtena pro každý blok indexu).
- Reorganizace dat nebo jejich předběžné seřazení.
- Umístění datových souborů na disky, které se načítají do systémové vyrovnávací paměti. Potom budou při SQL dotazu data čtena z vyrovnávací paměti operačního systému, což je lepší než přímé čtení z disku.

Popsali jsme dvě chyby, které byly zjištěny díky nasazení implementované aplikace na produkční centrální databázi VUT v Brně. Zároveň jsme tyto chyby popsali a také jsme se snažili navrhnout, jak tyto chyby odstranit.

Nejdůležitější však je, že aplikace obstála v testech a můžeme říct, že je schopna fungovat v reálném provozu a je schopna zobrazit administrátorovi databáze výsledky, na jejichž základě je schopen odhalit neduhy sledované databáze.

Kapitola 7

Závěr

Cílem této diplomové práce bylo seznámení s nástroji sloužícími pro monitorování Oracle databáze, analyzovat interní katalogy Oracle databáze a navrhnout a implementovat aplikaci, která bude v člověku čitelné podobě zobrazovat aktuální stav databázové instance.

Seznámili jsme se tedy s nástroji STATSPACK, *Active Session History* (ASH), *Automatic Database Diagnostic Monitor* (ADDM) a *Automatic Workload Repository* (AWR). Dále jsme prozkoumali mechanismus zvaný *Wait Interface*.

Poté jsme navrhli a implementovali aplikaci, která slouží pro sledování databáze. Tato aplikace je založena na modulech, jež jsou výkonnými jednotkami aplikace. Moduly mohou, ale nemusejí, pro zobrazení statistik, používat výše uvedených nástrojů. To je velkou devízou aplikace. Jednotlivé moduly lze naprogramovat podle konkrétních požadavků tak, aby zobrazovaly přesně to, co chceme a potřebujeme. Pro tvorbu modulů bylo vytvořeno API, aby byla implementace modulů jednodušší a aby byla zaručena struktura modulu, díky níž je možné modul do aplikace zakomponovat. Modulovatelnost aplikace, je, spolu s možností vytvořit moduly na míru, značná odlišnost oproti ostatním komerčním aplikacím tohoto druhu.

Vyvinutá aplikace může kvalitativně dále růst s dalšími rozšířeními a vylepšeními. Týká se to implementace prvků, které uživateli ještě více zpříjemní a ulehčí práci. Konkrétně jmenujme zachytávací mód pro okna modulů, který pomáhá uživateli při přesunu okna jej zarovnat k jinému oknu. Dále by bylo možné při ukončení aplikace ukládat konfiguraci aplikace. Např. moduly, které byly při ukončení spuštěné, ke kterým databázovým instancím byla aplikace připojena, umístění oken spuštěných modulů. Při následném spuštění aplikace by se pak ona sama inicializovala do stavu, ve kterém se před ukončením nacházela.

Ikdyž možných vylepšení by se již nyní našla celá řada, aplikace je momentálně na takové úrovni, která umožňuje jejímu uživateli s ní zacházet a pracovat velmi jednoduše, účelně a intuitivně.

Nasazením aplikace na produkční databázi jsme dokázali, že aplikace je schopna fungovat a splňuje požadavky, které byly na její vývoj kladeny. Životní cyklus aplikace nyní nekončí. Je třeba vytvořit další a další moduly, které budou nadále aplikaci rozšiřovat a díky nim celkově hodnota celé aplikace dále poroste.

Literatura

[1] Java(TM) Platform, Standard Edition 6 Overview. 1994-2010, [Online; navštíveno 14. 1. 2010].
URL
http://java.sun.com/javase/6/docs/technotes/guides/index.html#jre-jdk

[2] Mechanisms for Secure Modular Programming in Java. 1999, [Online; navštíveno 26.10. 2009].
URL http://www.ece.cmu.edu/~lbauer/papers/jmstr.pdf

[3] XML - eXtensible Markup Language. 1999, [Online; navštíveno 27. 4. 2010].
URL http://www.kosek.cz/clanky/xml/index.html

[4] Ptplot 5.1p1 - Java Plotter. 2001, [Online; navštíveno 18.12. 2009].
URL http:
//ptolemy.eecs.berkeley.edu/java/ptplot5.1p1/ptolemy/plot/doc/index.htm

[5] Exploring the Oracle Database 10g Release 1 Wait Interface. 2004, [Online; navštíveno 3. 1. 2009].
URL
http://www.oracle.com/technology/pub/articles/schumacher_10gwait.html

[6] jCharts. 2004, [Online; navštíveno 18.12. 2009].
URL http://jcharts.sourceforge.net/index.html

[7] Oracle database connection problem (from oracle.com). 2004, [Online; navštíveno 15. 4. 2009].
URL http://websina.com/bugzero/kb/oracle-connection.html

[8] Proactive vs. Reactive Tuning Explained. 2004, [Online; navštíveno 6. 1. 2009].
URL http://www.dba-oracle.com/oracle_news/2004_10_25_proactive_vs_
reactive_tuning.htm

[9] Java - dynamically change the classpath). 2005, [Online; navštíveno 2.11. 2009].
URL http://www.velocityreviews.com/forums/
t148021-dynamically-change-the-classpath.html

[10] Návrh aplikací v jazyce UML - diagram tříd. 2005, [Online; navštíveno 5. 3. 2009].
URL
http://interval.cz/clanky/navrh-aplikaci-v-jazyce-uml-diagram-trid/

[11] Wait Event Enhancements in Oracle 10g. 2005, [Online; navštíveno 3. 1. 2009].
URL
http://www.dbspecialists.com/files/presentations/wait_events_10g.ppt

[12] Java Programming [Archive] - Multithreading: pausing a thread. 2006, [Online; navštíveno 10.1. 2010].
URL http://forums.sun.com/thread.jspa?threadID=724209

[13] Oracle Wait Interface: What, Why and How. 2006, [Online; navštíveno 6. 1. 2009].
URL http://www.nyoug.org/Presentations/2006/06/Deshpande_Oracle%20Wait%20Interface_Keynote.pdf

[14] Swing Chapter 16. (Advanced topics) Desktops and Internal Frames. Easy for reading, Click here! 2006, [Online; navštíveno 5.10. 2009].
URL http://www.javafaq.nu/java-bookpage-27-5.html

[15] Develop a Java Plugin Framework. 2007, [Online; navštíveno 18.10. 2009].
URL http://twit88.com/blog/2007/10/07/develop-a-java-plugin-framework/

[16] Develop a Java Plugin Framework – Search for Plugin Dynamically. 2007, [Online; navštíveno 18.10. 2009].
URL http://twit88.com/blog/2007/10/08/develop-a-java-plugin-framework-search-for-plugin-dynamically/

[17] Diagram tříd. 2007, [Online; navštíveno 5. 3. 2009].
URL http://web.sks.cz/users/ku/PRI/tridy.htm

[18] Java - Dynamic loading of class and jar file. 2007, [Online; navštíveno 18.10. 2009].
URL http://twit88.com/blog/2007/10/04/java-dynamic-loading-of-class-and-jar-file/

[19] Java Plug-in Framework (JPF) Project. 2007, [Online; navštíveno 25.10. 2009].
URL http://jpf.sourceforge.net/

[20] JChart2D. 2007, [Online; navštíveno 18.12. 2009].
URL http://jchart2d.sourceforge.net/index.shtml

[21] JFreeChart. 2007, [Online; navštíveno 18.12. 2009].
URL http://www.jfree.org/jfreechart/

[22] How to use an INI file. 2008, [Online; navštíveno 11.11. 2009].
URL http://www.java-tips.org/java-se-tips/java.util/how-to-use-an-ini-file.html

[23] Installing Oracle STATSPACK. 2008, [Online; navštíveno 30. 12. 2008].
URL http://www.dba-oracle.com/tips_install_oracle_statspack.htm

[24] Oracle STATSPACK. 2008, [Online; navštíveno 24. 12. 2008].
URL http://download.oracle.com/docs/cd/E13160_01/wli/docs10gr3/dbtuning/statsApdx.html

[25] Oracle Statspack Survival Guide. 2008, [Online; navštíveno 29. 12. 2008].
URL http://www.akadia.com/services/ora_statspack_survival_guide.html

[26] The Oracle Timeline. 2008, [Online; navštíveno 16. 12. 2008].
URL http://www.oracle.com/timeline/index.html

[27] STATSPACK Collection Options. 2008, [Online; navštíveno 29. 12. 2008].
URL http://www.remote-dba.net/t_tuning_statspack_collection.htm

[28] Writing modular Java applications (a suggestion for a simple, but versatile plugin architecture). 2008, [Online; navštíveno 30.10. 2009].
URL http://www.onyxbits.de/content/blog/patrick/writing-modular-java-applications-suggestion-simple-versatile-plugin-architectu

[29] 1000 Free Farm-Fresh Web Icons. 2009, [Online; navštíveno 5.3. 2010].
URL http://www.fatcow.com/free-icons/

[30] Classloaders Keeping Jar Files Open. 2009, [Online; navštíveno 20.3. 2010].
URL http://management-platform.blogspot.com/2009/01/classloaders-keeping-jar-files-open.html

[31] Closing a URLClassLoader. 2009, [Online; navštíveno 20.3. 2010].
URL http://blogs.sun.com/CoreJavaTechTips/entry/closing_a_urlclassloader

[32] Connect to an Oracle database with JDBC. 2009, [Online; navštíveno 15. 4. 2009].
URL http://www.rgagnon.com/javadetails/java-0112.html

[33] Crystal Office Icon Collection. 2009, [Online; navštíveno 4.3. 2010].
URL http://www.freeiconsweb.com/crystal-office-icon-collection.html

[34] FAQ – Oracle JDBC. 2009, [Online; navštíveno 15. 4. 2009].
URL http://www.oracle.com/technology/tech/java/sqlj_jdbc/htdocs/jdbc_faq.html

[35] How to create a pluginable Java program? 2009, [Online; navštíveno 30.10. 2009].
URL http://stackoverflow.com/questions/25449/how-to-create-a-pluginable-java-program/25607

[36] Java (programovací jazyk). 2009, [Online; navštíveno 14. 1. 2010].
URL http://cs.wikipedia.org/wiki/Java_%28programovac%C3%AD_jazyk%29

[37] Lesson: JDBC Basics. 2009, [Online; navštíveno 15. 4. 2009].
URL http://java.sun.com/docs/books/tutorial/jdbc/basics/

[38] Oracle Database 11g Release 1 JDBC Drivers. 2009, [Online; navštíveno 15. 4. 2009].
URL http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/htdocs/jdbc_111060.html

[39] QueryTableModel: A basic implementation of the TableModel interface that fills o Java code example. 2009, [Online; navštíveno 18.4. 2010].
URL http://www.javafaq.nu/java-example-code-870.html

[40] Table Cell Listener. 2009, [Online; navštíveno 16.4. 2010].
URL http://tips4java.wordpress.com/2009/06/07/table-cell-listener/

[41] Unified Modeling language. 2009, [Online; navštíveno 25. 2. 2009].
URL http://cs.wikipedia.org/wiki/UML

[42] URLClassLoader won't free JAR files. 2009, [Online; navštíveno 20.3. 2010]. URL `http://www.coderanch.com/t/456847/Java-General/java/URLClassLoader-won-free-JAR-files`

[43] URLClassLoader won't free JAR files. 2009, [Online; navštíveno 20.3. 2010]. URL `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=5041014`

[44] Use Case Diagram - Diagram případů užití. 2009, [Online; navštíveno 25. 2. 2009]. URL `http://mpavus.wz.cz/uml/uml-b-use-case-3-2-1.php`

[45] Andert, S.: *Oracle Wait Event Tuning*. Kittrell, North Carolina, USA: Rampant Techpress, 2004, ISBN 0-9745993-7-9.

[46] Powell, G.: *Oracle High Performance Tuning for 9i and 10g*. Digital Press, 2004, ISBN 1555583059.

[47] Richmond Shee, K. G., Kirtikumar Deshpande: *Oracle Wait Interface: A Practical Guide to Performance Diagnostics & Tuning*. McGraw-Hill/Osborne, 2004, ISBN 007222729x.

Seznam použitých zkratek a symbolů

UML – Unified Modeling Language

GUI – Grafické uživatelské rozhraní (Graphical User Interface)

UI – Uživatelské rozhraní (User Interface)

API – Application Programming Interface

JDK – Java Development Kit

JVM – Java Virtual Machine

JRE – Java Runtime Environment

SŘDB – Systém Řízení Datové Báze

XML – Extensible Markup Language

OOP – Objektově Orientované Programování

MVC – Model-View-Controller architektura

CPU – Central Processing Unit

Seznam příloh

Příloha A

Případy užití aplikace dle diagramu případů užití

V této příloze jsou uvedeny některé případy užití vyvíjené aplikace.

Název	Přidat databázi
Aktéři	Uživatel
Popis	Uživatel má v úmyslu zaregistrovat v aplikaci novou databázi.
Počáteční podmínky	Uživatel má zobrazeno okno manažera databází. Okno manažera databází obsahuje tlačítko, po jehož klinutí se spustí mechanismus pro vytvoření nové databáze. Existuje okno pro zadávání údajů o nové databázi.
Koncové podmínky	Nová databáze zaregistrována.
Hlavní tok	**Registrace databáze** 1. Zobrazí se okno pro zadání údajů o databázi. 2. Uživatel vloží do příslušných polí údaje o databázi. 3. Uživatel potvrdí vytvoření nové databáze s těmito údaji. 4. Nová databáze je v aplikaci zaregistrována. 5. Nová databáze je zobrazena v seznamu databází.
Vedlejší toky	**Uživatel nevyplnil všechny povinné položky** (po kroku 4 hlavního toku) 1. Je zobrazena chybová hláška upozorňující na nevyplněné povinné údaje. 2. Uživatel potvrdí chybové hlášení a vrátí se k zadání údajů. 3. Pokračujeme krokem č. 4 hlavního toku.
Výjimky	**Databáze se zadaným názvem je již v aplikaci registrovaná** (po kroku 4 hlavního toku) 1. Zobrazí se chybové hlášení. 2. Uživatel potvrdí chybovou hlášku. 3. Vrátíme se zpět do okna pro zadání parametrů databáze.

Tabulka A.1: Případ užití „Přidat databázi“.

Název	Připojit k databázi
Aktéři	Uživatel
Popis	Aplikace se připojí k vybrané databázi.
Počáteční podmínky	Uživatel má vybranou databázi, ke které se chce připojit.
Koncové podmínky	Aplikace je připojena k vybrané databázi. Ve všech spuštěných modulech je aktualizován seznam databází, ke kterým se lze připojit.
Hlavní tok	**Připojení aplikace k databázi** 1. Uživatel si zobrazí seznam registrovaných databází. 2. Uživatel vybere databázi, ke které se chce připojit. 3. Uživatel potvrdí volbu. 4. Aplikace se připojí k vybrané databázi. V titulku okna aplikace je přidán název databáze, ke které se aplikace nově připojila.
Vedlejší toky	
Výjimky	**K databázi se nelze připojit** (po kroku 3 hlavního toku) 1. Zobrazí se chybové hlášení. 2. Uživatel potvrdí chybovou hlášku. 3. Aplikace zůstane ve výchozím stavu. **Aplikace je již k databázi připojena** (po kroku 3 hlavního toku) 1. Aplikace zůstane ve výchozím stavu.

Tabulka A.2: Případ užití „Připojit k databázi“.

Název	Odpojit od databáze
Aktéři	Uživatel
Popis	Aplikace ukončí spojení s vybranou databází.
Počáteční podmínky	Aplikace je připojena k databázi, s níž chceme spojení ukončit.
Koncové podmínky	Aplikace je od zvolené databáze odpojena. Ve všech spuštěných modulech je aktualizován seznam databází, ke kterým se lze připojit.
Hlavní tok	**Odpojení od databáze** 1. Uživatel vybere databázi, od které se má aplikace odpojit. 2. Uživatel potvrdí volbu. 3. Aplikace se odpojí od vybrané databáze. Z titulku okna aplikace je odebrán název databáze, od které se aplikace odpojila.
Vedlejší toky	
Výjimky	**Od databáze se nelze odpojit. Je používána některým ze spuštěných modulů** (po kroku 2 hlavního toku) 1. Uživateli je zobrazeno chybové hlášení o tom, že se nelze od databáze odpojit. 2. Uživatel potvrdí chybovou hlášku. 3. Aplikace zůstane ve výchozím stavu.

Tabulka A.3: Případ užití „Odpojit od databáze“.

Název	Přidat modul
Aktéři	Uživatel
Popis	Uživatel přidá nový modul do seznamu modulů aplikace.
Počáteční podmínky	
Koncové podmínky	V aplikaci je registrován nový modul.
Hlavní tok	**Přidat modul** 1. Uživatel si zobrazí okno manažera modulů. 2. Vybere volbu pro přidání nového modulu. 3. Zobrazí se dialogové okno, ve kterém se zadává kategorie, do které se má modul zařadit, a kde se také zadává cesta k JAR archívu modulu. 4. Uživatel potvrdí přidání vybraného modulu. 5. Okno pro přidání modulu se uzavře. 6. V manažeru modulů se aktualizuje seznam modulů.
Vedlejší toky	**Uživatel chce modul zařadit do nové kategorie, která není dosud definovaná.** (po kroku 2 hlavního toku) 1. Uživatel nemůže v seznamu kategorií najít tu, do které by rád modul zařadil. 2. Klikne na odkaz směřující na seznam kategorií. 3. Zvolí přidání nové kategorie. 4. Zadá název nové kategorie. 5. Potvrdí vytvoření nové kategorie. 6. Aktualizuje se seznam kategorií. 7. Vrátí se zpátky do okna pro přidání nového modulu. 8. Pokračujeme krokem č. 3 hlavního toku.
Výjimky	**Uživatel nevybral kategorii, do které se má modul zařadit.** (po kroku 4 hlavního toku) 1. Zobrazí se chybové hlášení. 2. Uživatel potvrdí chybovou hlášku. 3. Vrátíme se zpět do okna pro zadání nového modulu. **Uživatel nezadal cestu k JAR archívu modulu.** (po kroku 4 hlavního toku) 1. Zobrazí se chybové hlášení. 2. Uživatel potvrdí chybovou hlášku. 3. Vrátíme se zpět do okna pro zadání nového modulu. **Zvolený modul již v dané kategorii existuje.** (po kroku 4 hlavního toku) 1. Zobrazí se chybové hlášení. 2. Uživatel potvrdí chybovou hlášku. 3. Vrátíme se zpět do okna pro zadání nového modulu.

Tabulka A.4: Případ užití „Přidat modul“.

Název	Spustit modul
Aktéři	Uživatel
Popis	Uživatel přidá do seznamu modulů nový modul.
Počáteční podmínky	Vybraný modul je registrovaný v seznamu modulů aplikace.
Koncové podmínky	Modul je spuštěn.
Hlavní tok	**Spustit modul** 1. Uživatel stiskne tlačítko Start na tlačítkové liště aplikace. 2. V hlavní menu nabídce aplikace zvolí kategorii a posléze modul, který chce spustit. 3. Modul je spuštěn.
Vedlejší toky	
Výjimky	**Vybraný modul je již spuštěn.** (po kroku 2 hlavního toku) 1. Zobrazí se hlášení o tom, že je již modul spuštěn s otázkou, zda se má i přesto daný modul spustit opětovně. 2. Pokud uživatel potvrdí opětovné spuštění, modul bude spuštěn. 3. Uživatel odmítne opětovné spuštění modulu, aplikace zůstane ve výchozím stavu. **nelze nalézt JAR archív modulu.** (po kroku 2 hlavního toku) 1. Zobrazí se chybové hlášení. 2. Uživatel potvrdí chybovou hlášku. 3. Aplikace zůstane ve výchozím stavu. **Zadaný modul není komapatibilní.** (po kroku 2 hlavního toku) 1. Zobrazí se chybové hlášení. 2. Uživatel potvrdí chybovou hlášku. 3. Aplikace zůstane ve výchozím stavu.

Tabulka A.5: Případ užití „Spustit modul“.

Název	Smazat modul
Aktéři	Uživatel
Popis	Uživatel smaže vybraný modul ze seznamu modulů.
Počáteční podmínky	V aplikaci je registrovaný modul, jež chce uživatel odstranit.
Koncové podmínky	Daný modul se již nevyskytuje v seznamu modulů aplikace.
Hlavní tok	**Smazat modul** 1. Uživatel si zobrazí okno manažera modulů. 2. V seznamu modulů označí ten, který chce smazat. 3. Stiskne tlačítko pro odebrání modulu. 4. Modul je odstraněn. 5. Aktualizuje se seznam modulů aplikace.
Vedlejší toky	
Výjimky	**Vybraný modul je momentálně spuštěn.** (po kroku 3 hlavního toku) 1. Zobrazí se hlášení o tom, že modul je momentálně spuštěn a tudíž jej není možné právě odebrat. 2. Pokud uživatel potvrdí hlášení. 3. Aplikace zůstane ve výchozím stavu.

Tabulka A.6: Případ užití „Smazat modul“.

Příloha B

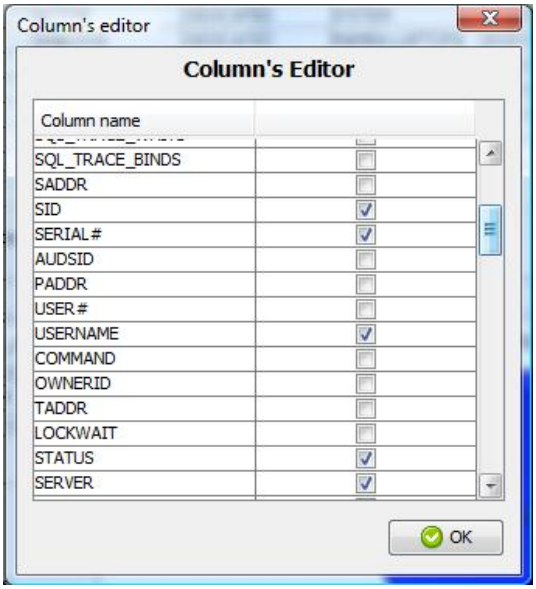
Dotaz plnící hlavní tabulku v modulu „Actually Used SQLs“

```
SELECT
  s.SID,
  s.serial#,
  s.sql_id AS "current sql id",
  s.username AS "db user",
  DECODE(ROUND(s.last_call_et / 60),
         '0', '< 1',
         ROUND(s.last_call_et / 60)) AS "runtime (mins)",
  si.block_changes AS "sess bchgs",
  si.physical_reads AS "sess preads",
  si.consistent_gets AS "sess cgets",
  ss_cpu."sess cpu" AS "sess cpu",
  sa.optimizer_cost AS "curr sql cost",
  su.blocks AS "temp blocks",
  t.used_ublk AS "undo blocks",
  s.program AS "program",
  s.terminal AS "term",
  pss_pr."px preads" "child px preads",
  pss_cg."px cgets" "child px cgets",
  pss_cpu."px cpu" "child px cpu",
  DECODE(ps."px oper cnt",
         '', 'N/A', ps."px oper cnt") AS "px opers",
  DECODE(ps."px count",
         '', 'N/A', ps."px count") AS "px slaves",
  s.event AS "wait",
  s.seconds_in_wait AS "wait secs",
  s.state AS "wait state",
  s.machine,
  s.type AS "session type",
  s.module,
  s.action,
  sa.sql_fulltext AS "sql text",
```

```
  s.sql_child_number
FROM
  v$session AS s, v$sort_usage AS su, v$transaction AS t,
  v$sess_io AS si, v$sql AS sa,
  (SELECT qcsid, COUNT(DISTINCT server_set) AS "px oper cnt",
          COUNT(*) AS "px count"
   FROM v$px_session
   WHERE NOT server_set IS NULL
   GROUP BY qcsid, DEGREE) AS ps,
  (SELECT qcsid, SUM(VALUE) AS "px preads"
   FROM v$px_sesstat
   WHERE statistic# = 54 AND sid != qcsid
   GROUP BY qcsid) AS pss_pr,
  (SELECT qcsid, SUM(VALUE) AS "px cgets"
   FROM v$px_sesstat AS pss
   WHERE statistic# = 50 AND sid != qcsid
   GROUP BY qcsid) AS pss_cg,
  (SELECT qcsid, SUM (VALUE) AS "px cpu"
   FROM v$px_sesstat AS pss
   WHERE statistic# = 12 AND sid != qcsid
   GROUP BY qcsid) AS pss_cpu,
  (SELECT ss.sid, SUM(ss.value) AS "sess cpu"
   FROM v$sesstat AS ss
   WHERE statistic# = 12
   GROUP BY ss.sid) AS ss_cpu
WHERE
  s.sql_address = sa.address AND s.sql_hash_value = sa.hash_value
  AND s.saddr = su.session_addr(+) AND s.SID = ps.qcsid(+)
  AND s.SID = si.SID(+) AND s.saddr = t.ses_addr(+)
  AND s.SID = pss_pr.qcsid(+)  AND s.SID = pss_cg.qcsid(+)
  AND s.SID = pss_cpu.qcsid(+) AND s.SID = ss_cpu.sid(+)
  AND s.TYPE != 'BACKGROUND' AND s.status = 'ACTIVE'
  AND program NOT LIKE ('%(C%') AND program NOT LIKE ('%(A%')
  AND program NOT LIKE ('%(P%')
ORDER BY sa.optimizer_cost DESC;
```

Příloha C

Ukázky modulů



Obrázek C.1: Editor sloupců hlavní tabulky modulu „Process Memory Info“.

Obrázek C.2: Běžící modul „Process Memory Info“.

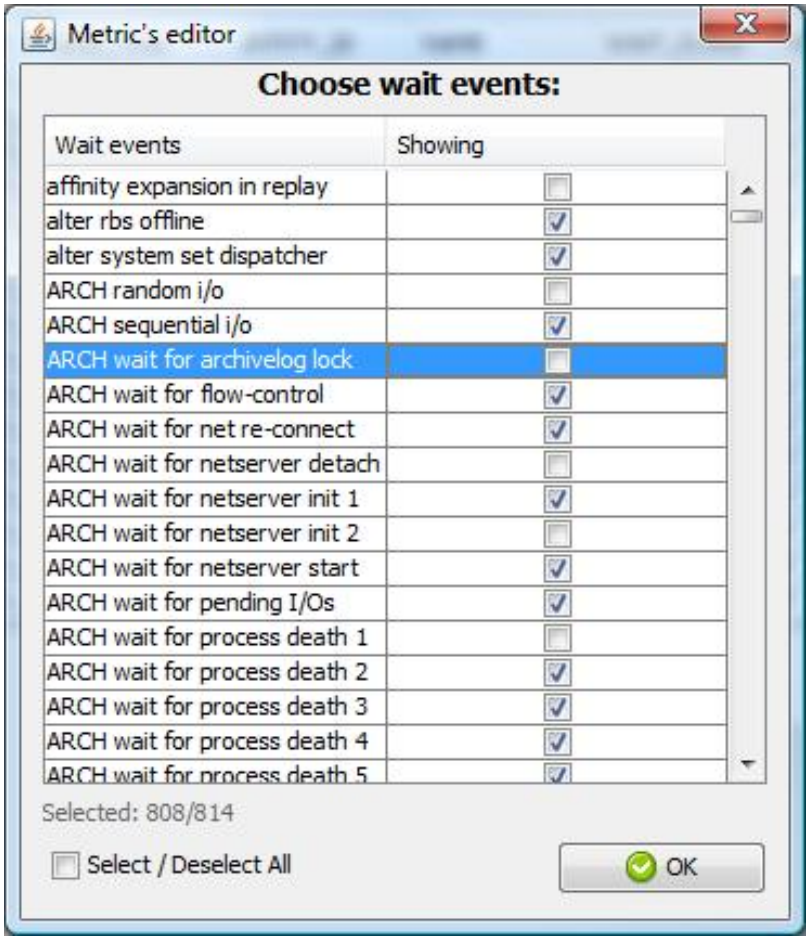
Obrázek C.3: Běžící modul „Used Functions“.

Obrázek C.4: Běžící modul „Actually running SQLs“.

Obrázek C.5: Okno zobrazující plán zvoleného dotazu modulu „Actually running SQLs“.

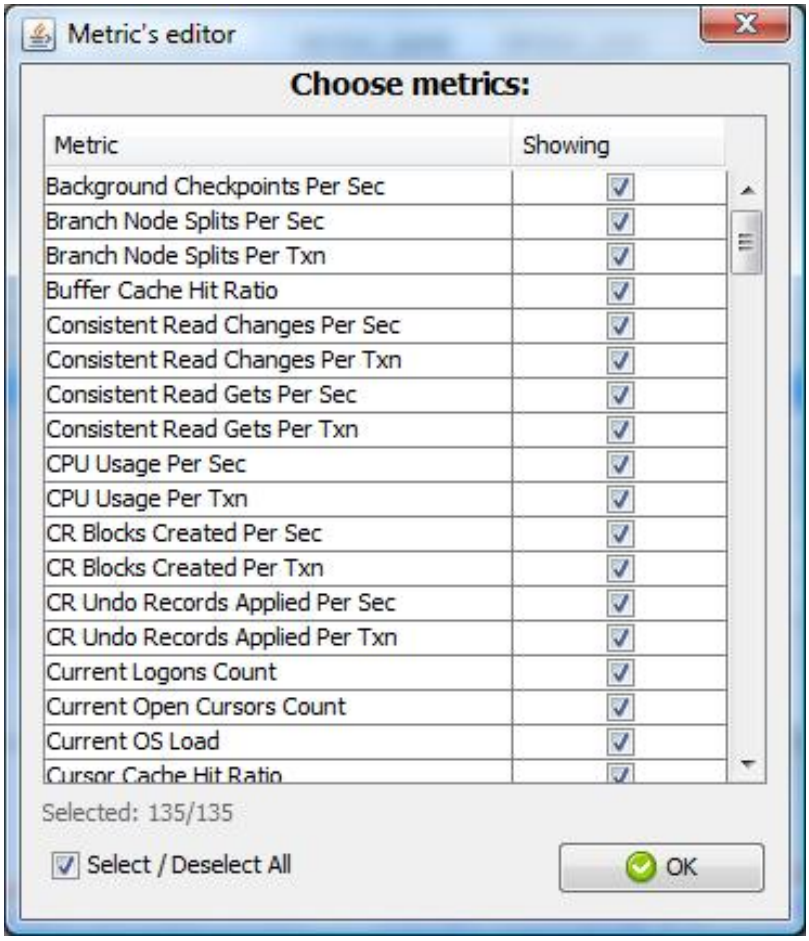
Obrázek C.6: Běžící modul „System Statistics“ – jeho část zobrazující přehled *Wait Interface*.

Obrázek C.7: Běžící modul „System Statistics“ – jeho část zobrazující metriky událostí.



Obrázek C.8: Editor čekacích událostí modulu „System Statistics“, podle kterých se mohou události filtrovat.

Obrázek C.9: Běžící modul „System Statistics“ – jeho část zobrazující historii metrik.



Obrázek C.10: Editor metrik modulu „System Statistics“, podle kterých se může filtrovat historie metrik.