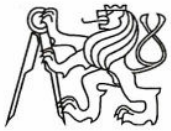
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Implementace nástroje pro licencování aplikací v jazyce Java
Student:	Jan Vybíral
Vedoucí:	Ing. Jan Trávníček
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství (magisterský)
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2012/13

Pokyny pro vypracování

1. Proveďte rešerši dostupných nástrojů pro licencování desktopových Java aplikací.

2. Navrhněte způsob licencování desktopových aplikací v jazyce Java na platformě Eclipse RCP.

3. Naimplementujte server pro distribuci licencí a klientskou část licencované desktopové aplikace.

4. Otestujte implementované řešení s ohledem na využití systémových prostředků.

5. Zhodnoťte použitelnost implementovaného řešení s důrazem na snadnost integrace a úroveň zabezpečení.

Seznam odborné literatury

Dodá vedoucí práce

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 13. dubna 2012

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Magisterská práce

Implementace nástroje pro licencování aplikací v jazyce Java

Bc. Jan Vybíral

Vedoucí práce: Ing. Jan Trávníček

11. května 2012

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Janu Trávníčkovi za pomoc s napsáním práce. Dále také Ing. Jiřímu Jandovi za připomínky a také své rodině za podporu během psaní.

Prohlášení

Prohlašuji, že jsem tuto práci vytvořil samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některý zákonů (autorský zákon), nemám závažný důvod proti užití tohoto školního díla a k užití uděluji svolení.

Praha dne 11. května 2012 .....

Odkaz na tuto práci

Jan Vybíral. *Implementace nástroje pro licencování aplikací v jazyce Java: Magisterská práce.* Praha: ČVUT v Praze, Fakulta informačních technologií, 2012.

Abstract

Licencing of applications written in Java and based on Eclipse RCP platform is investigated in this thesis. Thesis contains research of of available open-source and commercial licensing tools and evaluates their usability. Also deals with the issue of protecting applications written in Java against unauthorized modification. The result of this theses is a set of libraries adding support for licensing and license distribution into existing applications. Part of the implemented system is also license server including web user interface.

Keywords Java, Licensing, Eclipse, RCP, OSGi, App Engine

Abstrakt

Tato práce se zabývá licencováním aplikací napsaných v jazyce Java se zaměřením na platformu Eclipse RCP. Práce obsahuje rešerši dostupných open-source a komerčních nástrojů pro licencování a zhodnocuje jejich použitelnost. Dále se zabývá problematikou ochrany aplikací napsaných v Javě proti neautorizované upravě. Výsledkem práce je sada knihoven, která umožňuje do existujících aplikací přidat podporu pro licencování a distribuci licencí. Součástí je také implementovaný licenční server včetně webového uživatelského rozhraní.

Klíčová slova Java, Licencování, Eclipse, RCP, OSGi, App Engine

Obsah

1 Úvod **17**
1.1 Motivace 17
1.2 Cíle práce 17

2 Rešerše dostupných řešení **19**
2.1 License4j 19
2.2 JLicense 20
2.3 Rampart 21
2.4 Protection! 4 21
2.5 LM-X License Manager 22
2.6 Vyhodnocení 23

3 Analýza **25**
3.1 Požadavky 25
3.2 Architektura 27
3.3 Doménový model 32
3.4 Komunikace 34

4 Ochrana aplikace **39**
4.1 Bezpečnostní rizika 39
4.2 Možné způsoby ochrany 41
4.3 Shrnutí 44

5 Implementace **45**
5.1 Klientská část 45
5.2 Serverová část 47
5.3 Rozšiřitelnost 48
5.4 Webové rozhraní 48
5.5 Nástroj pro šifrování tříd 50

6 Testování **53**
6.1 Jednotkové testování 53
6.2 Výkonnostní testování 54
6.3 Ukázková aplikace 59

Závěr **65**

Literatura **67**

A Slovník **69**

B Seznam použitých zkratek **71**

C Obsah CD **73**

D Uživatelská příručka **75**
D.1 Použití klientské části 75
D.2 Použití serverové části 76
D.3 Použití serverové aplikace 77

E Obrázky **79**

F Zdrojové kódy **83**

G Přehled projektů **87**

Seznam obrázků

3.1 Diagram nasazení jednotlivých komponent 28
3.2 Diagram komponent serverové části 29
3.3 Diagram komponent klientské části 30
3.4 Doménový model aplikace 32

6.1 Ukázka průběhu měření propustnosti serverové části při lokálním nasazení 57
6.2 Ukázka průběhu měření propustnosti serverové části při nasazení na Google App Engine 58
6.3 Ukázka uživatelského rozhraní testovací aplikace 60
6.4 Ukázka dialogu pro zadání licenčního čísla 63

E.1 License4j UI pro tvorbu licencí 80
E.2 JLicense UI pro tvorbu licencí 80
E.3 Rampart UI pro tvorbu licencí 81
E.4 Protection developer control center 81
E.5 Ukázka webového uživatelského rozhraní - seznam vystavených licencí 82
E.6 Ukázka webového uživatelského rozhraní - seznam vlastností . . . 82
E.7 Ukázka webového uživatelského rozhraní - seznam šifrovacích klíčů 82

Seznam tabulek

2.1 Porovnání hodnocených produktů 23

3.1 Results 36
3.2 Results 37

6.1 Výsledky unit testů 54
6.2 Hardwarová konfigurace počítače pro testování klientské části . . . 54
6.3 Vlastnosti vybraného vzorku .jar archívů pro testování šifrování . . 54
6.4 Výsledky měření klientské části 55
6.5 Výsledky měření klientské části 56
6.6 Výsledky testů propustnosti klientské části 56
6.7 Využití systémových prostředků při načtení informace o licenci . . 58
6.8 Využití systémových prostředků při aktivaci nové licence 58
6.9 Výsledek testu na počet operací do vyčerpání kvót poskytnutých zdarma 59

KAPITOLA 1

Úvod

1.1 Motivace

Tato práce vznikla na základě požadavku přidat do existujících aplikací napsaných v jazyce Java na platformě Eclipse RCP podporu pro licencování.

Výrazem licence se v kontextu vývoje software myslí oprávnění, kterým autor software umožňuje jeho použití dalším osobám. Při komerční distribuci software se často vyskytuje potřeba upravit funkčnost na základě zakoupené licence. Takovéto úpravy mohou zahrnovat například zpřístupnění nebo zakázání některých vlastností produktu, omezení možností opakované instalace a kopírování případně časové omezení určité funkcionality.

K licencování aplikací patří také ochrana proti modifikacím. Pokud chceme pomocí licenčního nástroje omezit funkčnost nebo možnost aplikaci volně distribuovat, je potřeba také zajistit, že omezení vynucené licenčním mechanismem nepůjde snadno deaktivovat nebo odstranit. V Javě je ochrana aplikací problematická – struktura .class souborů produkovaných překladačem Javy je dobře zdokumentovaná a snadno čitelná[12]. Tento problém se obvykle částečně řeší obfuskací výsledného kódu včetně názvů tříd a metod. Pokud je aplikace postavená na Eclipse RCP/Open Services Gateway initiative framework (OSGi) platformě, jsou tyto problémy ještě závažnější. Protože názvy tříd se používají jako identifikátory, není možné je snadno obfuskovat. Díky modulární struktuře OSGi je navíc možné ochranu distribuovanou jako plugin snadno odstranit případně nahradit vlastní prázdnou implementací.

1.2 Cíle práce

Cílem práce je prozkoumat dostupná řešení, která umožňují přidat do aplikací podporu pro licencování. Dále také prozkoumat možnosti ochrany aplikací

postavených na platformě OSGi a navrhnout a naimplementovat nástroj pro licencování, který bude splňovat následující podmínky:

- Implementace v jazyce Java
- Podpora pro aplikace napsané na platformě Eclipse RCP
- Funkčnost na operačních systémech Windows a Linux
- Snadná integrovatelnost do již existujících aplikací
- Centrální správa licencí
- Webové Graphic User Interface (GUI) pro snadnější správu licencí

KAPITOLA 2

Rešerše dostupných řešení

V této kapitole je proveden průzkum existujících nástrojů pro licencování aplikací s cílem zmapování poskytované funkcionality. Hlavní důraz při hodnocení je kladen na následující vlastnosti:

- Integrovatelnost s aplikacemi napsanými v jazyce Java
- Podpora pro aplikace napsané na platformě Eclipse RCP/OSGi
- Funkčnost v operačních systémech Linux a Windows
- Možnost centrální správy licencí
- Možnost vázat licenci na konkrétní hardware
- Cena řešení

2.1 License4j

License4j[15] je jednoduchá knihovna, která umožňuje vytvářet licence pro aplikace napsané v Javě. Obsahuje GUI nástroj pro generování licenčních souborů (ukázka na obrázku E.1), které se digitálně podepíší a přibalí se k výsledné aplikaci. Platnost souboru se kontroluje v aplikaci pomocí přidaného veřejného klíče, čímž je znemožněno modifikovat licenční soubor.

Vlastnosti

Aplikace je dostupná volně ke stažení, verze zdarma umožňuje vytvářet licence s platností 10 dní, licenci pro plnou funkčnost je možné zakoupit za částku $35. K produktu je navíc možné dokoupit roční rozšířenou podporu za $15.

Distribuovaná verze obsahuje knihovny a GUI bez zdrojových kódů. Mezi vlastnosti produktu patří:

- Ruční výroba licenčního souboru pomocí dodané aplikace s GUI, předdefinované položky v licenci
- Možnost zabudování Media Access Control (MAC) adresy nebo hostname do licenčního souboru
- Kontrola platnosti licence za běhu pomocí veřejného klíče

Zhodnocení

License4j je jednoduchá knihovna s velmi omezeným použitím, čemuž odpovídá i poměrně nízká pořizovací cena.

Výhodou může být snadná integrovatelnost s existujícími aplikacemi a snadná výroba licenčních klíčů pomocí dodaného GUI.

Hlavní nevýhodou tohoto produktu je potřeba ručně generovat licenční soubory pro každého zákazníka, což může být při distribuci většího množství licencí problematické. Licence je sice možné vázat na MAC adresu nebo hostname, to ale není pro reálné použití příliš vhodné.

2.2 JLicense

Podobně jako u předchozího produktu se v případě JLicense[16] jedná o jednoduchou knihovnu, která umožňuje pomocí GUI (na obrázku E.2) generovat licenční klíče, jejichž platnost se poté v aplikaci kontroluje pomocí digitálního podpisu.

Vlastnosti

Knihovnu je možné stáhnout a používat zdarma. V případě zájmu je možné koupit zdrojové kódy za cenu $50. Vlastnosti produktu jsou:

- Ruční výroba licenčního souboru pomocí dodané aplikace s GUI, předdefinované položky v licenci
- Možnost zabudování IP adresy do licence

Zhodnocení

Jedná se o produkt funkčně shodný s knihovnou License4j. Výhodou může být možnost neomezeného použití zdarma a také dostupnost zdrojových kódů za poplatek.

2.3 Rampart

Posledním ze zástupců kategorie malých knihoven je knihovna Rampart[10]. Stejně jako v předchozích případech se produkt skládá z GUI pro generování licenčních souborů (obrázek E.3) a runtime knihovny pro ověřování vygenerovaných licencí.

Vlastnosti

Trial verzi s omezenou platností na 30 dní je možné zdarma stáhnout z webových stránek výrobce, licenci na plnou verzi je možné zakoupit za $29. Mezi vlastnosti produktu patří:

- Výroba a správa licencí pomocí GUI
- Podpora pro trial verze

Zhodnocení

Výhodou tohoto produktu oproti předchozím je mnohem propracovanější nástroj pro správu licencí. Runtime knihovna také umožňuje zobrazovat dialogy usnadňující získání licence.

Nevýhodou může být fakt, že vydané licence není možné jakkoliv vázat na hardware.

2.4 Protection! 4

Produkt Protection! 4[13] je první ze zkoumaných produktů přímo určený pro nasazení v enterprise prostředí. Obsahuje jak velmi pokročilý nástroj umožňující vytváření, správu a integraci licencí, tak i server pro centrální správu licencí. Ukázka uživatelského rozhraní je na obrázku E.4.

Vlastnosti

Jedná se o ryze komerční řešení, ze stránek výrobce se dají stáhnout všechny potřebné nástroje, je ale potřeba kontaktovat výrobce ohledně vystavení trial licence. Cena je $997 za licenci pro desktopovou aplikaci pro jednoho vývojáře, serverový backend stojí $5000 na rok na jedno procesorové jádro. Je možné také dokoupit dodatečnou podporu.

- Výroba a správa licencí pomocí pokročilého grafického nástroje
- Grafický nástroj usnadňující integraci ochrany do aplikace vytvářející vlastní spouštěcí soubor
- Podpora pro trial verze

- Webová služba umožňující integraci s existujícími systémy
- Možnost vázat licenci na hardware počítače
- Licenční server včetně webového rozhraní pro správu licencí
- Možnost vydávat licence vázané na hardware i licence omezené počtem souběžně běžících kopií
- Kontrola integrity aplikace pro znesnadnění odstranění ochrany

Zhodnocení

Jedná se o pokročilé řešení určené pro větší podniky s velkým množstvím vlastností. Nevýhodou může být vysoká pořizovací cena pro menší projekty.

2.5 LM-X License Manager

LM-X License Manager[17] je další z komerčních licenčních nástrojů orientovaný na velké společnosti. Produkt je distribuován v podobě Software Development Kit (SDK) napsaného v C++, pro integraci s aplikacemi napsanými v Javě výrobce poskytuje obalovací třídy.

Vlastnosti

Výrobce umožňuje stažení testovací verze s omezenou dobou platnosti zdarma po vyplnění kontaktního formuláře. Platnost testovací verze je možné prodloužit za poplatek 200 € na 3 měsíce. Cena licence pro komerční použití závisí na obratu společnosti a počtu platforem. Nejlevnější licence pro společnost s obratem pod 1M € vyjde na 1500 € za rok a 550 € za každou další platformu.

Vlastnosti produktu:

- Integrace s aplikacemi pomocí výrobcem dodaného SDK
- Možnost vystavovat licence vázané na hardware i plovoucí licence
- Licenční server pro centrální správu licencí

Zhodnocení

Opět se jedná o profesionální licenční řešení, takže jeho hlavní výhodou je velké množství funkcí a podpora od výrobce.

Jako nevýhoda se kromě ceny může také jevit skutečnost, že dodávaná knihovna je napsaná v C++ a pro použití s aplikacemi napsanými v Javě je potřeba používat obalovací třídy. K aplikaci je tedy nutné přibalit i zkompilované knihovny pro každou podporovanou platformu.

Název	Platformy	Podpora OSGi	Cena
License4j	Java	Ne	$35
JLicense	Java	Ne	Zdarma, $50 za zdrojové kódy
Rampart	Java	Ne	$29
Protection! 4	Java	Ne	$997 za vývojáře, $5000 za serverovou část na rok
LM-X License Manager	Windows, Linux, MacOS, AIX, Solarix,...	Ne	Dle obratu firmy, minimálně 1500 € na rok + 550 € za platformu

Tabulka 2.1: Porovnání hodnocených produktů

2.6 Vyhodnocení

Výsledky zkoumaných řešení jsou shrnuty v tabulce 2.6.

Průzkum ukazuje, že dostupná řešení se dělí ve své podstatě na 2 kategorie:

- Malé knihovny – poskytují pouze základní funkcionalitu v podobě vytváření licencí pomocí dodaného nástroje a knihovnu pro ověřování platnosti licence za běhu licencované aplikace. V případě, že nehledáme příliš sofistikované řešení může být jejich použití vhodné, obzvláště pak s přihlédnutím k ceně. Ta se pohybuje v rozmezí několika desítek dolarů.

- Komerční řešení – produkty určené k použití v enterprise prostředí. Vynikají především množstvím poskytované funkcionality – od vystavování trial licencí, přes licence vázané na hardware až po serverová řešení pro distribuci licencí. Tomu ovšem také odpovídá cena, která se pohybuje v řádu tisíců dolarů, často s nutností připlácet za další vývojáře, platformy a také za každoroční obnovení licence.

Ukazuje se také, že žádný z dostupných nástrojů neobsahuje podporu pro aplikace psané na platformě Eclipse RCP, případně OSGi.

KAPITOLA 3

Analýza

V této kapitole je provedena analýza požadavků a je navržena architektura výsledné aplikace.

3.1 Požadavky

3.1.1 Nefunkční požadavky

Výsledný navrhovaný nástroj musí splňovat následující podmínky

Implementace v jazyce Java

Vzhledem k tomu, že důvodem ke vzniku práce bylo dodání licencovacího mechanismu do existujících aplikací napsaných v jazyce Java, jedním z hlavních požadavků je celý navržený systém implementovat v Javě. Ačkoliv Java umožňuje využívat knihovny napsané například v C/C++ pomocí Java Native Interface (JNI) nebo Java Native Access (JNA), je použití nativních knihoven mnohem snadnější.

Podpora pro aplikace napsané na platformě Eclipse RCP

Důvodem tohoto požadavku je fakt, že existující aplikace jsou napsány právě na platformě. Pro snadnější integraci by měla být klientská část navrhovaného systému distribuovaná v podobě pluginu.

Důsledkem tohoto požadavku je kromě podoby výsledné distribuce klientské části také omezení týkající se knihoven třetích stran. Využívány by měli být především takové knihovny, které jsou obvykle obsaženy ve většině distribucí aplikací napsaných na platformě Eclipse RCP. Například v případě použití GUI by měla být použita knihovna Standard Widget Toolkit (SWT).

Funkčnost na operačních systémech Windows a Linux

Tento požadavek opět souvisí s tím, že existující aplikace podporují právě tyto operační systémy. Navíc jsou systémy Windows a Linux nejsnáze dostupné pro testovací účely.

Ačkoliv je Java multiplatformní jazyk, možnosti týkající se získávání informací specifických pro konkrétní operační systém jsou značně omezené. Pokud chceme vázat licenci na konkrétní hardware, je zapotřebí mnohem těsnější vazby s operačním systémem, než jakou Java umožňuje. Z tohoto důvodu bude nutné část funkcionality přizpůsobit konkrétnímu operačnímu systému a tuto část implementovat pro všechny podporované platformy. Jako cílové byly zvoleny operační systémy Windows a Linux, mělo by ale být možné v případě potřeby snadno rozšířit podporu i pro další operační systémy.

Snadná integrovatelnost do již existujících aplikací

K rozhodnutí o přidání funkčnosti pro licencování do aplikace může dojít až v pozdější fázi vývoje, případně může být požadavek licencování přidat do již hotového produktu. Z tohoto důvodu by měla být integrace co nejjednodušší. Modulární architektura platformy Eclipse RCP toto v případě integrace do desktopové aplikace usnadní. Stejný požadavek ale můžeme mít také na serverové straně. Pokud již máme existující databázi uživatelů, mohli bychom ji chtít využít pro vystavení licencí.

Centrální správa licencí

Aby bylo možné mít přehled o všech vydaných licencích, je nutné licence spravovat centrálně.

Webové GUI pro snadnější správu licencí

Protože o vydávání licencí se běžně stará obchodní oddělení, je potřeba vytvořit grafické uživatelské rozhraní, které umožní uživatelům spravovat vydané licence. Z hlediska snadnosti použití a platformní nezávislosti se jako nejvhodnější jeví webové rozhraní.

3.1.2 Funkční požadavky

Vlastnosti licence

Licence vydávané a ověřované implementovaným licenčním nástrojem musí mít následující vlastnosti.

- Časové omezení – Počátek nebo konec platnosti licence je možné časově omezit.

- Vazba na hardware – Platnost licence je možné vázat na konkrétní hardware.

- Licencované vlastnosti – Do vystavené licence je možné zanést informace o povolených vlastnostech licencovaného produktu. Licencované vlastnosti je možné přidávat a odebírat.

Vlastnosti klientské části

Tyto vlastnosti bude mít klientská část licenčního mechanismu, která se bude zabudovávat do aplikace:

- Online kontrola platnosti – Platnost licence bude možné ověřit proti licenčnímu serveru.

- Offline kontrola platnosti – Platnost údajů uvedených v licenci je možné ověřit bez nutnosti připojení k centrálnímu serveru. Jedná se především o kontrolu dat počátku a konce platnosti a kontroly vazby na hardware.

Vlastnosti serverové části

- Zobrazení přehledu vystavených licencí.

- Vytvoření nové licence.

- Zneplatnění licence.

- Ověření platnosti vydané licence.

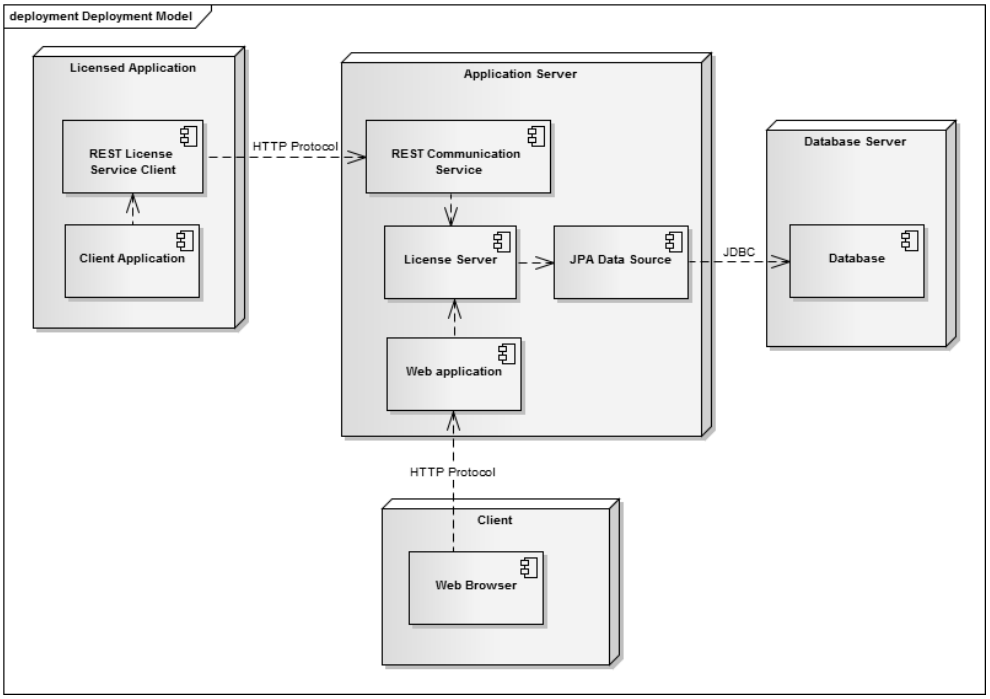
3.2 Architektura

Jako nejvhodnější architektura pro navrhovaný produkt je typ klient-server. Rozvržení jednotlivých komponent je zobrazeno v diagramu na obrázku 3.1.

3.2.1 Serverová část

Účelem serverové části je poskytovat informace o vydaných licencích a ověřovat jejich platnost. Serverová část může být integrována do existující serverové aplikace, nebo nasazena jako samostatná aplikace. Komponenty serverové aplikace jsou na obrázku 3.2.

Pro zajištění rozšiřitelnosti, snadné integrace do existujících aplikací a dobré testovatelnosti je část funkcionality serverové části rozdělena do několika komponent s definovaným rozhraním.



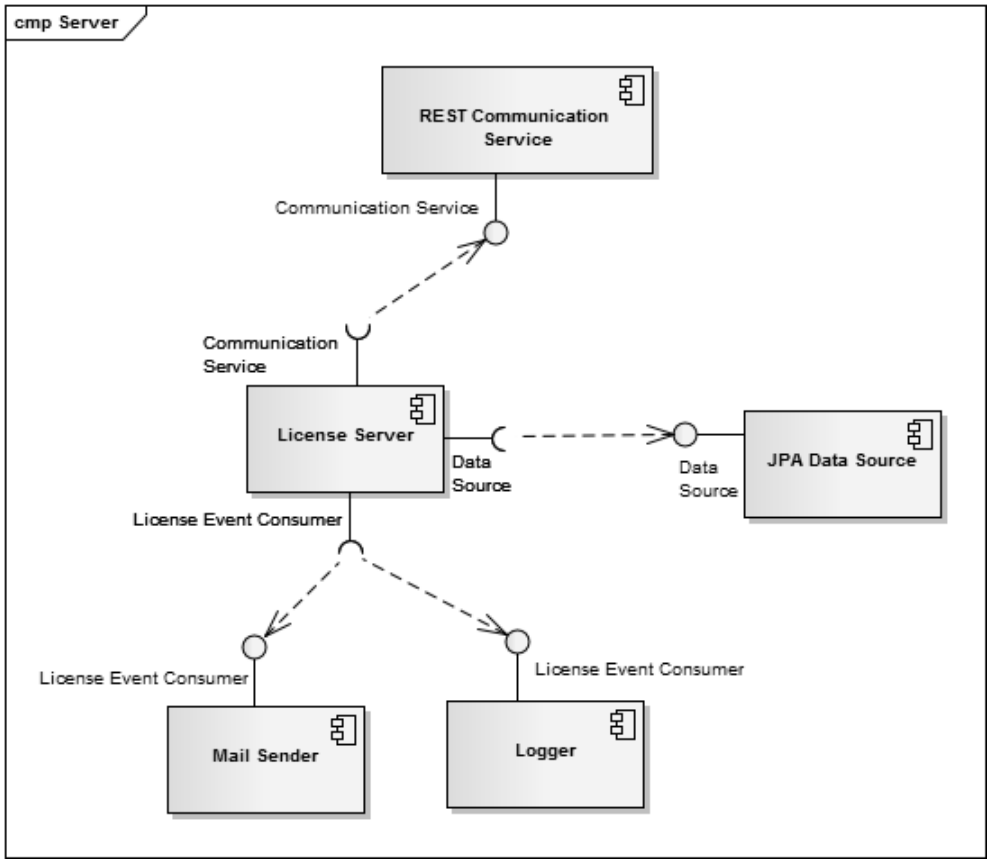
Obrázek 3.1: Diagram nasazení jednotlivých komponent

License Server

`License Server` je hlavní část serverové aplikace obsahující logiku spojenou s ověřováním licencí. Jejím úkolem je odpovídat na požadavky od klientských aplikací na ověření platnosti licence uložené v databázi. Jednotlivé části zajišťující komunikaci s klienty a databází jsou dekomponovány do samostatných částí a komunikace s nimi probíhá pomocí pevně daného rozhraní. To umožňuje udržet komponentu nezávislou na použitém komunikačním protokolu a databázové vrstvě.

Communication Service

`Communication Service` je komponenta odpovědná za komunikaci s klientskými aplikacemi. Jejím vyčleněním do samostatné části je dosaženo odstranění závislosti na použitém protokolu pro komunikaci s klientskými aplikacemi. Komponenta `Communication Service` překládá požadavky na ověření platnosti licence z podoby specifické pro použitý komunikační protokol (Extensible Markup Language (XML), JavaScript Object Notation (JSON), Serializované objekty, ...) do univerzální podoby akceptované částí `License Server`.



Obrázek 3.2: Diagram komponent serverové části

Data Source

Komponenta `Data Source` slouží k načítání a ukládání informací o vydaných licencích do persistentního úložiště. Poskytuje serverové aplikaci základní operace jako nalezení licence podle identifikátoru a zaznamenání aktivace licence. Při integraci serverové části do již existující aplikace je komponenta `Data Source` použita pro napojení na existující datový model. V případě samostatného nasazení může být komponenta naimplementována jako Data Access Object (DAO).

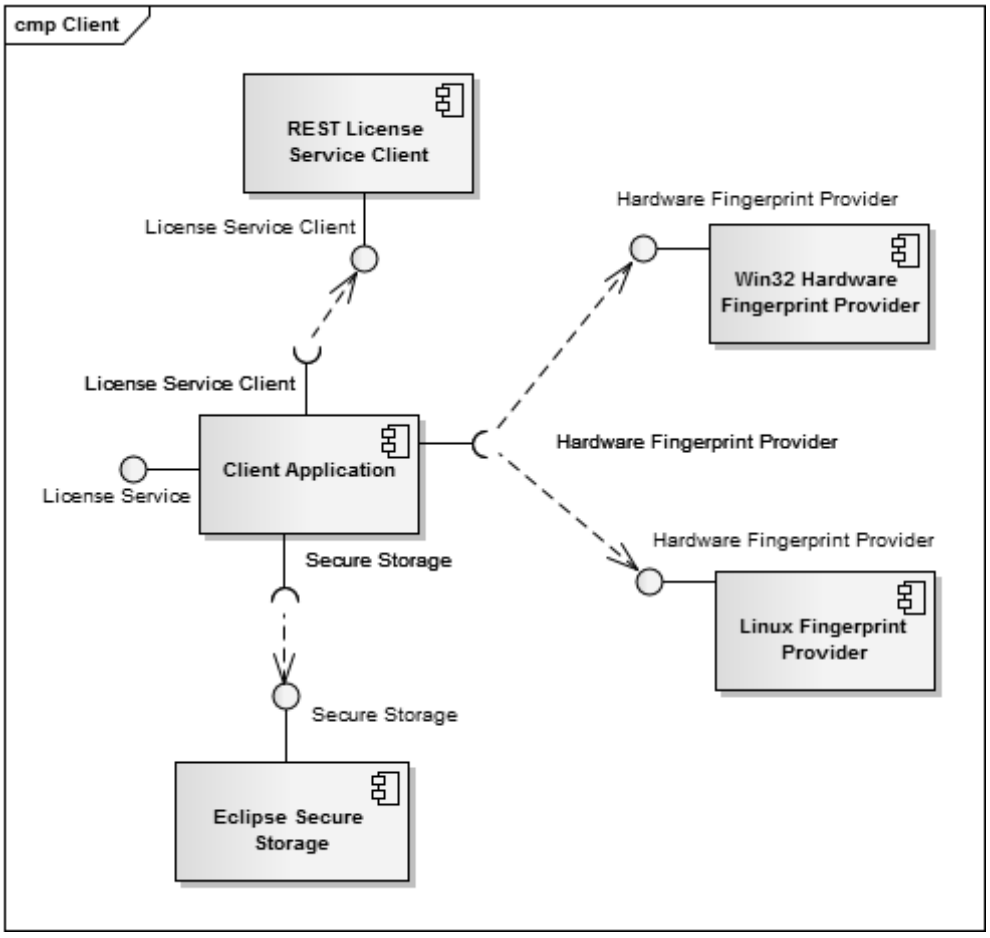
License Event Consumer

Poslední z navržených komponent je komponenta s názvem `License Event Consumer`. Účelem této komponenty je umožnit snadnější rozšíření serverové aplikace o dodatečnou funkcionalitu. Jedná se o aplikaci návrhového vzoru

observer – komponenta `License Server` umožní registrovat komponenty implementující rozhraní `License Event Consumer`. Registrované komponenty budou notifikovány o událostech v serverové aplikaci, například při povolení nebo zamítnutí ověření licence. Tímto způsobem je možné doplnit aplikaci o další funkcionalitu, například logování událostí nebo zasílání notifikací pomocí emailu.

3.2.2 Klientská část

Klientskou částí se rozumí část vyvíjeného nástroje, která se zabuduje do licencované aplikace. Účelem této části je poskytnout licencované aplikaci programové rozhraní pro vyžádání a ověření licence. Rozdělení klientské části na jednotlivé komponenty je zobrazena na obrázku 3.3.



Obrázek 3.3: Diagram komponent klientské části

Klientská část je podobně jako serverová část rozdělená do na jednotlivé komponenty. Jedná se ale především o logické dělení – při integraci do cílové aplikace mohou být jednotlivé komponenty mnohem těsněji provázány za účelem znesnadnění odstranění jednotlivých částí.

Client Application

`Client Application` je hlavní komponenta klientské aplikace obsahující všechnu potřebnou logiku. Účelem této komponenty je poskytovat aplikaci informace o aktuální licenci a případně umožnit aplikaci si vyžádat aktivaci nové licence.

License Service Client

Komponenta `License Service Client` slouží ke komunikaci se serverovou aplikací. Jedná se o protějšek serverové komponenty `Communication Service`. Vyčleněním komunikace se serverem do samostatné komponenty získáme možnost změny komunikačního protokolu bez nutnosti zásahu do ostatních částí klientské aplikace.

Hardware Fingerprint Provider

Účelem komponenty `Hardware Fingerprint Provider` je poskytnout klientské části přístup k informacím o hardware počítače, na kterém je licencovaná aplikace nainstalována. Informace o hardware počítače jsou použity pro vytvoření unikátního otisku hardware. Protože že vyvíjený licencovací nástroj musí být funkční na operačních systémech Windows a Linux, je nutné tuto funkcionalitu vyčlenit do samostatné komponenty.

Secure Storage

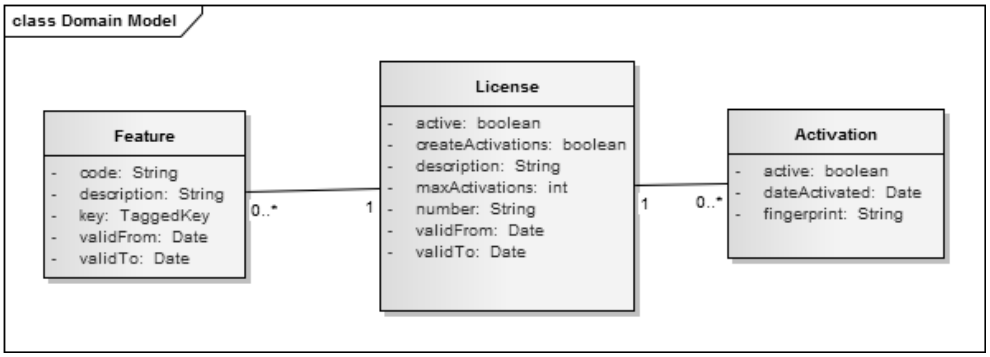
Jedním z požadavků na klientskou část je možnost ověřit platnost licence bez aktivního připojení na internet. Z toho důvodu je nutné mít informace o aktivované licenci uložené, aby je bylo možné při opětovném spuštění aplikace načíst a ověřit. Za tímto účelem je navržena komponenta `Secure Storage`. Cílem této komponenty je poskytnout klientské aplikaci persistentní úložiště pro informace o licenci. Persistentní podoba uložené licence musí být chráněna proti modifikaci z venčí.

License Service

Rozhraní `License Service` je jediná část licencovacího nástroje viditelná z klientské aplikace. Umožňuje klientské aplikaci provést ověření aktuální licence a v případě potřeby vyžádání nové licence.

3.3 Doménový model

Doménový model aplikace odpovídající požadavkům nastíněných v předchozích sekcích je zobrazen na obrázku 3.4.



Obrázek 3.4: Doménový model aplikace

Model představuje minimální základ potřebný pro serverovou aplikaci k vydávání licencí. V této podobě umožňuje vystavení licence s definovanými vlastnostmi, omezení platnosti a kontrolu nad počtem aktivací licencí. Model je navržen s ohledem na další rozšíření, možnosti jak dodat například podporu pro správu uživatelů jsou diskutovány v následující části.

Feature

Třída `Feature` představuje určitou vlastnost, kterou potřebujeme identifikovat na úrovni licencované aplikace. Z pohledu licencované aplikace může vlastnost představovat například modul aplikace. Pokud je v licenci vlastnost s odpovídajícím kódem nalezena, uživateli se zpřístupní funkcionalita poskytovaná daným modulem. Pokud není licencovaná aplikace členěna na jednotlivé moduly, je možné vlastnost použít pro zpřístupnění celé aplikace.

Třída `Feature` obsahuje následující atributy:

- **code** – Unikátní identifikátor vlastnosti, který je kontrolován v licencované aplikaci.
- **description** – Nepovinný atribut obsahující textový popis k dané vlastnosti určený pro zobrazení v uživatelském rozhraní.
- **key** – Šifrovací klíč pro dešifrování tříd patřících k dané vlastnosti.
- **validFrom, validTo** - Data označující časové ohraničení platnosti dané vlastnosti.

Při ukládání do relační databáze je vhodné třídu Feature rozdělit na 2 samostatné tabulky. První, sloužící jako číselník vlastností, by v tomto případě obsahovala pouze kód a popis. Druhá, reprezentující vazbu mezi licencí a vlastnostmi, by obsahovala data platnosti, odkaz na vlastnost z číselníku a odkaz na příslušnou licenci.

License

Třída `License` představuje licenci umožňující uživateli využívat vlastnosti v ní uvedené po dobu její platnosti. Licenci je možno chápat jako unikátní sadu omezení pro používání aplikace.

Atributy licence:

- **active** – Booleovská hodnota určující, zda je licence aktivní. Pomocí tohoto atributu je možné vydané licence zneplatnit.
- **createActivations** – Booleovská hodnota určující, zda je pro danou licenci možné vytvářet nové aktivace. Pokud je nastavena na false, požadavky na vytvoření nové aktivace pro danou licenci selžou.
- **description** – Volitelný textový popis licence, určený pro zobrazení v GUI.
- **maxActivations** – Maximální počet aktivací dané licence. Atribut umožní omezit počet aktivací licence.
- **number** - unikátní licenční číslo sloužící pro identifikaci licence. Pomocí tohoto čísla je možné z klientské aplikace vyžádat aktivování licence.
- **validFrom, validTo** – Data omezující časovou platnost licence.

`License` obsahuje podobně jako `Feature` data počátku a konce platnosti. V klientské aplikaci je důležitý pouze seznam vlastností, výsledné datum platnosti každé vlastnosti odeslané na klientskou aplikaci vznikne jako průnik dat platnosti na třídě `License` a `Feature`.

Na licenci můžou být vázána další data. K licenci můžeme například přiřadit informace o osobě, pro kterou je licence vytvořena, případně informace o provedených platbách.

Activation

Třída `Activation` slouží pro spravování počtu aktivovaných licencí. K tomu, aby bylo možné licenci na klientské aplikaci používat, je nejprve nutné licenci aktivovat na serveru. Při aktivaci klient odešle na server číslo aktivované licence a otisk počítače, na základě těchto údajů se vyrobí nová aktivace.

Atributy třídy `Activation`:

- **acitve** – Booleovská hodnota určující, zda je aktivace platná. Nastavením na `false` je možné zneplatnit aktivaci a klientská aplikace si bude muset vyžádat novou aktivaci. Zneplatněné aktivace se nezapočítávají do maximálního počtu aktivací.
- **dateActivated** – Datum, kdy byla aktivace vytvořena.
- **fingerprint** – Hardwarový otisk počítače, z něhož byla aktivace provedena.

Pomocí aktivací je možné omezit počet současně aktivovaných licencí. V případě, že chceme mít licenci vázanou na konkrétní hardware s určitým otiskem, na licenci se nastaví atribut `createActivations` na `false` a licenci se přiřadí předem připravená aktivace s požadovaným aktivačním hardwarovým otiskem. Pokud přijde požadavek na aktivaci licence se stejným hardwarovým otiskem jako je ten, definovaný v připravené aktivaci, server vrátí připravenou aktivaci. Při požadavku s jiným otiskem vytvoření aktivace selže.

3.4 Komunikace

V návrhu architektury licencovacího nástroje je v sekci 3.2.1 navržena serverová komponenta pro komunikaci s klientskými aplikacemi, v sekci 3.2.2 je popsán její protějšek umístěný v klientské aplikaci. Komponenty umožňují použití libovolného komunikačního protokolu mezi klientskou a serverovou částí, v navrhovaném licencovacím nástroji bude tato část implementována v podobě RESTful webové služby.

Architektura Representational State Transfer (REST) byla vybrána z následujících důvodů:

- Komunikace probíhá pomocí Hypertext Transfer Protocol (HTTP). To je výhodné například při použití v podnikovém prostředí, kde kromě HTTP protokolu bývají ostatní komunikační protokoly omezené nebo zakázané.
- Jazyk Java obsahuje nativní podporu pro komunikaci pomocí protokolu HTTP, na klientské straně nejsou potřeba žádné další knihovny.
- Protokol HTTP je bezestavový, což zjednodušuje implementaci.
- Protokol HTTP není vázán pouze na jazyk Java, serverovou část je možné volat i z klientů napsaných v jiných jazycích.

3.4.1 REST

REST je koncept pro návrh distribuované architektury, u které je komunikace realizována pomocí protokolu HTTP. RESTful webové služby jsou orientovány

na práci s datovými zdroji. Každý zdroj má svůj unikátní identifikátor (Unified Resource Locator (URL)), pomocí něhož se s daným zdrojem manipuluje. Rozlišují se čtyři základní operace:

- Create – Vytvoření zdroje, na úrovni protokolu HTTP se pro tuto operaci používá metoda PUT.
- Read – Čtení informací z datového zdroje, pro tuto operaci se používá metoda GET.
- Update – Upravení informací zdroje, nejčastěji realizováno metodou POST
- Delete – Smazání datového zdroje, pro tuto operaci se používá metoda DELETE.

Výsledek provedené operace je klientovi sdělen pomocí návratového kódu HTTP. Nejčastěji se používají následující návratové kódy:

- 200 OK – Kód označuje, že požadavek byl proveden úspěšně.
- 400 Bad Request – Kód vrácen v případě, že požadavek od klienta je špatně zfromulován, nebo obsahuje neplatné údaje.
- 404 Not Found – Kód označující, že požadovaný zdroj nebyl nalezen.
- 500 Internal Server Error – Kód vrácen v případě, že při zpracování došlo k chybě na serveru.

Kompletní přehled stavových kódů HTTP lze nalézt ve specifikaci [11].

Pokud chce klient například získat informace o licenci 12345, komunikace bude vypadat následovně:

1. Klient pošle HTTP požadavek na URL `http://server/activations`, v těle požadavku bude číslo licence, informace o aplikaci, ze které byl požadavek poslán a otisk počítače.
2. Server přijme požadavek a vyhledá licenci s identifikátorem 12345.
3. Server provede kontrolu platnosti licence a vyrobí záznam o aktivaci
4. Server odešle klientovi odpověď s návratovým kódem 200. Informace o licenci budou obsaženy v těle odpovědi ve formátu XML.

3.4.2 Serverové rozhraní

Serverová část poskytne klientům pomocí REST webové služby metody pro práci s licencemi.

Kód	Popis	Obsah těla odpovědi	Položky v hlavičce odpovědi
201	Aktivace licence úspěšně vytvořena	Textová reprezentace nově vytvořené aktivace	parametr Location s URL nově vytvořené aktivace
200	Aktivace licence pro počítač s daným otiskem již existuje	Textová reprezentace existující aktivace	-
403	Požadovaná licence nenalezena	Informace o typu chyby	-
403	Platnost licence vypršela	Informace o typu chyby a platnosti licence	-
403	Licence zneplatněna	Informace o typu chyby	-
403	Vyčerpán počet aktivací	Informace o typu chyby, maximální počet aktivací	-

Tabulka 3.1: Seznam možných odpovědí serveru na požadavek na aktivaci licence

Zažádání o aktivaci licence

Serverová část umožní klientským aplikacím zažádat o aktivaci licence. Aktivace se provede pomocí HTTP dotazu typu `POST` na URL `/activations`. V požadavku klient uvede:

- Identifikátor licence, kterou chce klientská aplikace aktivovat.
- Hardwarový otisk počítače, ze kterého je prováděna aktivace.

Server zpracuje požadavek, možné odpovědi jsou uvedeny v tabulce 3.4.2

Získání informace o aktivaci licence

Serverová část umožní klientským aplikacím získat informace o aktivaci licence provedením HTTP dotazu typu `GET` bez těla na URL `/activations/{licenční číslo}/{otisk počítače}` s parametrem `appid` představující identifikátor aplikace.

Možné odpovědi serveru jsou v tabulce 3.2

Kód	Popis	Obsah těla odpovědi	Položky v hlavičce odpovědi
200	Aktivace nalzena	Textová reprezentace údajů o aktivaci licence	-
404	Požadovaná aktivace licence nenalezena	-	-

Tabulka 3.2: Seznam možných odpovědí serveru na požadavek na informace o aktivaci licence

3.4.3 Zabezpečení komunikace

Informace jsou u HTTP protokolu přenášeny v čitelné textové podobě. Tím vznikají následující rizika:

- Útočník může odposlechnout přenášená data. Při komunikaci s licenčním serverem se nepřenášejí žádné důvěrné informace (například údaje o klientovi), útočník ale může získat identifikátor licence a pomocí něj si aktivovat vlastní kopii licencovaného software.
- Útočník může pozměnit obsah zprávy, případně komunikaci se serverem podvrhnout. Tím by bylo možné si například změnit typ licence nebo do licence přidat vlastnosti, ačkoliv je uživatel nemá v licenci zahrnuty.

Pokud chceme zamezit odposlechu komunikace, klientská aplikace umožní použít ke komunikaci namísto nešifrovaného protokolu HTTP bezpečnější šifrovanou verzi Hypertext Transfer Protocol Secure (HTTPS). Nevýhodou tohoto řešení je nutnost mít v klientské aplikaci i na serveru nainstalovaný důvěryhodný certifikát, pomocí něhož se bude komunikace šifrovat.

Možností, jak zamezit modifikaci požadavku mezi klientskou a serverovou aplikací, je použití digitálního podpisu. Do hlavičky každého požadavku se připojí kontrolní součet těla požadavku zašifrovaný klíčem klienta nebo serveru. Tím bude možné zkontrolovat, že zpráva byla přenesená bez modifikace. Zároveň se tím potvrdí identita odesílatele zprávy.

KAPITOLA 4

Ochrana aplikace

Cílem této kapitoly je prozkoumat problematiku zabezpečení aplikací napsaných v jazyce Java proti modifikacím a navrhnout možné způsoby, jak neautorizovaným úpravám zabránit.

4.1 Bezpečnostní rizika

Pokud chceme v aplikaci použít licenční mechanismus, který omezí funkčnost aplikace na základě vystavené licence, je nutné aplikaci zabezpečit proti neautorizovaným úpravám. Aby byl licenční mechanismus účinný, nemělo by být možné ho jednoduše obejít nebo úplně odstranit z aplikace.

V této části jsou popsány hlavní problémy se zabezpečením aplikací napsaných v Javě proti modifikacím.

4.1.1 Čitelnost bytecode

Aplikace napsané v Javě jsou, na rozdíl od aplikací napsaných například v jazycích C/C++, multiplatformní. To znamená, že aplikaci napsanou a zkompilovanou na Windows je možné spustit bez dalšího kompilování i na jiných operačních systémech. Toho je dosaženo použitím platformě nezávilsé binární reprezentace zdrojového kódu. Aplikace napsané v Javě nejsou kompilovány přímo do strojového kódu pro danou platformu, ale do tzv. bytecode, který je interpretován virtuálním strojem Javy. Při kompilaci je pro každou třídu vytvořen samostatný soubor s příponou `.class`, který kromě instrukcí pro virtuální stroj javy obsahuje i popis všech metod a atributů dané třídy.

Soubory obsahující definice tříd mají přesně specifikovanou a dobře zdokumentovanou strukturu a jsou uloženy v čitelné podobě. Navíc bytecode obsahuje v porovnání se strojovým kódem instrukce na mnohem vyšší úrovni abstrakce. Bytecode obsahuje například i instrukce pro přístup k atributům

tříd nebo volání metod. Díky tomu je možné dekompilací z `.class` souborů poměrně snadno získat zpět zdrojový kód, který je navíc velmi podobný původnímu zdrojovému kódu, jehož kompilací byl `.class` soubor vytvořen.

Ukázku toho, jak může vypadat dekompilovaný kód, je možné vidět na ukázce G.2. Původní zdrojový kód třídy `Decompilation` je na ukázce F.3.

Díky možnosti dekompilace a úpravy již zkompilovaných souborů může útočník snadno z aplikace odstranit volání licenční knihovny a tím docílit neomezeného využívání licencované aplikace.

4.1.2 Linkování knihoven

Při kompilaci aplikace je nutné k výslednému programu připojit také používané knihovny. Tomuto kroku se říká linkování a rozlišujeme dva základní způsoby – statické a dynamické linkování.

V případě statického linkování jsou všechny závislosti na knihovy vyřešeny během kompilace a jsou zabaleny společně s kódem aplikace do výsledného spustitelného souboru. Tento způsob je často používán u jazyků kompilovaných do strojového kódu jako je například C nebo C++.

U dynamického linkování jsou knihovny umístěny mimo výsledný zkompilovaný spustitelný soubor a můžou být k programu připojeny až ve chvíli, kdy je volána některá jejich metoda.

Dynamické linkování je použito v Javě pro načítání tříd. Načtení třídy se provede za běhu programu až ve chvíli, kdy je třída poprvé použita. Pro identifikaci tříd se používá celé jméno třídy včetně balíčku, ve kterém je třída umístěna. Ve zkompilovaném programu je tedy uložen pouze název odkazované třídy, případně názvy volaných metod. To může představovat bezpečnostní riziko pro aplikaci.

Pokud bychom měli například třídu `com.test.LicenseService` a aplikace by pomocí této třídy prováděla ověřování licence, útočníkovi stačí tuto třídu nahradit vlastní implementací se stejným jménem a signaturou. Aplikace by při prvním použití načetla útočníkovu třídu a použila ji pro ověření licence.

U aplikací postavených na platformě OSGi je tato vlastnost Javy ještě problematičtější. Pokud by byla část starající se o licencování přidána do aplikace jako plugin, dá se snadno odstranit a nahradit upravenou verzí. Pluginy navíc obsahují popis všech sdílených balíků a poskytovaných služeb, takže jejich nahrazení je ještě o něco snazší. Pokud by byla třída navíc publikována jako OSGi služba, stačilo by útočníkovi vytvořit vlastní plugin, který by zaregistroval stejnou službu s větší prioritou.

4.2 Možné způsoby ochrany

4.2.1 Obfuskace

Obfuskace je úprava kódu aplikace takovým způsobem, aby se tím zakryl jeho původní účel. Obfuskovaný kód aplikace vykonává stejnou činnost jako kód před obfuskací, je ovšem mnohem obtížnější z něj odhalit, co přesně má kód vykonávat. Pro Javu existuje celá řada různých nástrojů, které umožňují obfuskovat aplikace, používá se buď obfuskace zdrojových kódu nebo obfuskace zkompilovaných tříd. Obfuskace na úrovni zdrojového kódu je vhodná spíše v případě, že zdrojový kód je dostupný cizím osobám a chceme zamezit tomu, aby mohli kód použít k dalšímu vývoji. Obfuskace bytecode znesnadňuje dekompilaci obfuskovaných tříd. Výhodou obfuskace na úrovni bytecode je fakt, že pravidla pro bytecode jsou méně restriktivní než které uplatňuje překladač Javy. Tím můžeme zkompilovaný program upravit tak, že z něj nepůjde zpětně zrekonstruovat platný zdrojový kód v Javě.

Při obfuskování programů se používají různé techniky. Nejčastěji se přejmenovávají názvy balíčků, tříd, metod a proměnných tak, aby z nich nešlo jednoduše poznat, jakou funkci zastávají. Často se také do kódu přidávají složité řídící struktury nebo zbytečná volání metod.

Ačkoliv obfuskací je možné zvýšit obtížnost dekompilace aplikace a následného odstranění licenčního mechanismu, existuje několik důvodů, které mohou bránit jejímu použití:

- Při změně názvů tříd a metod je nutné tyto změny propagovat i do dalších částí programu, které s obfuskovanou komponentou pracují. Pokud například obfuskujeme OSGi plugin, je nutné buď veřejné Application programming interface (API) nechat nedotčené nebo provádět obfuskaci nad všemi pluginy najednou. Obfuskování veřejného API pluginů ovšem značně znesnadňuje další rozšíření aplikace.

- Obfuskováním může docházet ke snižování výkonu aplikace. Obfuskací se mohou přidávat zbytečná volání metod nebo složité programové konstrukce, které jsou složité na vyhodnocení.

- Protože obfuskace změní kód aplikace tak, že neodpovídá zdrojovému kódu, může být hledání chyb v obfuskovaném programu komplikované.

- Obfuskace nepřináší žádnou bezpečnost navíc. Pomocí obfuskace se sice získání původního kódu případně nalezení míst se zabezpečením znesnadní, aplikaci ale bude stále možné libovolně dekompilovat a upravit.

4.2.2 Kontrola integrity aplikace

Jako další z možností ochrany aplikace proti modifikacím se nabízí použití zabudované kontroly integrity. Princip spočívá v tom, že aplikace má v sobě

uložené informace o podobě jednotlivých částí (například v podobě kontrolních součtů nebo otisků). Při startu aplikace nebo za běhu se poté provede kontrola proti těmto informacím. V případě, že je zjištěn rozdíl, se aplikace ukončí.

Účinnost této techniky je možné ještě zvýšit tím, že se kontrola integrity aplikace zduplikuje na větší počet míst v kódu. Pokud by například licencovací funkcionalita byla aplikaci poskytnuta v podobě OSGi služby, před volání této služby by se umístil kód, který by ověřil, že služba byla poskytnuta ze zprávného pluginu a že jeho otisk se nezměnil. Opakované ruční vkládání stejného kódu na velké množství míst ale ztěžuje údržbu aplikace a případné změny by bylo nutné aplikovat opakovaně, čímž se zvyšuje riziko zanesení chyb do aplikace. Proto je vhodné pro tento účel použít specializované nástroje pro manipulaci s bytecode, například ASM[2].

Nevýhodou zabudování kontroly integrity do aplikace je hned několik:

- Je nutné udržovat otisky kontrolovaných částí aktuální. Pokud se upraví část aplikace, je nutné příslušný otisk přepočítat a nahradit ho na všech místech, kde je použit.
- Použití této techniky nijak nezabezpečí samotný kód. Ten lze stále dekompilovat a kontrolu otisků lze odstranit. Případně je možné jen upravit uložené otisky aby odpovídali upravenému kódu.
- Přepočítávání otisků může být náročné na výkon.

4.2.3 Šifrování bytecode

Šifrování bytecode je účinná metoda, která zamezuje dekompilaci a modifikaci aplikace. Zkompilované `.class` soubory jsou zašifrovány pomocí symetrické šifry. Jejich rozšifrování se provádí až za běhu před načtením zašifrované třídy pomocí uloženého klíče. Díky tomu není možné aplikaci jednoduše dekompilovat. Protože jsou `.class` soubory šifrované, nelze je přečíst pomocí dekompilátoru.

Dešifrování tříd při načtení je možné provést různými způsoby. U běžných aplikací napsaných v Javě je možné pro načítání šifrovaných tříd použít vlastní classloader. U aplikací postavených na platformě OSGi je situace ještě jednodušší. Platforma OSGi umožňuje aplikacím definovat tzv. Adaptor Hooks. Jedná se o třídy, které rozšiřují funkcionalitu platformy a je možné je do aplikace přidat pomocí jednoduché konfigurace. Pro účely dešifrování bytecode se nejlépe hodí `ClassLoadingHook`. Ten umožňuje upravit bytecode třídy ještě před tím, než se načte pomocí classloaderu.

Největším problémem této metody je potřeba mít uložený šifrovací klíč v aplikaci v čitelné podobě, aby bylo možné provést dešifrování tříd. Z toho vyplývá, že i když jsou třídy šifrované a nelze je dekompilovat, je možné z aplikace získat klíč a dešifrovat je pomocí něj. Tento nedostatek je možné částečně řešit zapojením licencovacího mechanismu. Šifrovací klíč může být do aplikace

dodán až společně s platnou licencí. Navíc je možné šifrovat různými klíči různé části aplikace a v licenci poslat jen klíče k částem programu, které jsou v licenci zahrnuty. Tím zajistíme, že bez platné licence nepůjde aplikace používat a ani nebude možné aplikaci dekompilovat a ochranu z ní odstranit.

Ani šifrování tříd neposkytuje neproniknutelnou ochranu. Bytecode tříd musí být za běhu dostupný v dešifrované podobě, aby jej bylo možné spouštět. Útočník tak může do aplikace přidat kód, který si pomocí dešifrovacího classloaderu načte bytecode libovolné třídy a poté si je může v nezašifrované podobě uložit na disk. Navíc kód aplikace sloužící k dešifrování tříd musí být dostupný v nezašifrované podobě, aby jej bylo možné načíst a použít pro dešifrování zbylých tříd. Dekompilací této části může útočník získat dostatek informací o použitém šifrovacím algoritmu a případně i šifrovací klíče potřebné pro dešifrování zbytku aplikace.

4.2.4 Kompilace do nativního kódu

Kompilací aplikace do nativního kódu lze dosáhnout dobrého zabezpečení aplikace. Při této metodě je aplikace překládána místo do bytecode přímo do strojového kódu pro danou platformu. Během překladu se provádí řada úprav a optimalizací a díky tomu je získání původního zdrojového kódu téměř nemožné. Pro kompilaci programů napsaných v Javě existuje několik nástrojů:

- GNU Compiler for Java[4] - Jedná se o open-source kompilátor, který umožňuje aplikace napsané v Javě kompilovat jak do bytecode, tak i do strojového kódu pro různé platformy, podporuje i překlad bytecode do strojového kódu. Nevýhodou GCJ je chybějící podpora pro GUI a některé další součásti standardní knihovny Javy, podporovaná je pouze Java ve verzi 1.4 a částečně 1.5.

- Excelsior JET[3] - JET je sada nástrojů a běhové prostředí pro aplikace napsané v Javě. Umožňuje kompilovat Javovské aplikace do strojového kódu a spouštět je ve vlastním běhovém prostředí, které je kompatibilní s Javou verze 1.6. Protože se jedná o komerční produkt, jako největší nevýhoda může být cena, která se pohybuje řádově v tisících euro za platformu pro komerční použití. Pro nekomerční použití je možné získat licenci zdarma.

Nevýhodou kompilování Javovských aplikací do strojového kódu je především ztráta přenositelnosti mezi platformami. Pro každou podporovanou platformu je nutné zkompilovat aplikaci samostatně. Výhodou může být kromě zmíněné ochrany proti dekompilaci také rychlejší start aplikace, rychlost aplikace za běhu se ale při použití moderních virtuálních strojů Javy příliš neliší.

4.2.5 Hardwarová ochrana

Kromě softwarových řešení existují i čistě hardwarová řešení pro ochranu Javovských aplikací. Příkladem může být produkt Validy SoftNaos[9] v podobě přídavného USB zařízení. Ochrana poskytovaná tímto zařízením spočívá v tom, že se část kódu aplikace po přeložení zašifruje. Takto zašifrovaný kód je poté možné spustit pouze pomocí připojeného USB zařízení. Takto chráněný kód není možné dekompilovat a upravit a to ani za běhu aplikace, díky čemuž může být tento způsob ochrany jako jediný ze zmíněných považován za bezpečný.

Nevýhodou při použití hardwarové ochrany je především nutnost společně s aplikací distribuovat i hardwarová zařízení pro spouštění šifrovaného kódu, jejichž cena může často i přesáhnout cenu samotné aplikace. Problematický může být také výkon takto chráněného kódu, vykonávání šifrovaného kódu může být i 40000× pomalejší než u nešifrovaného kódu[14].

4.3 Shrnutí

Zabezpečení aplikací napsaných v jazyce Java je díky snadné dekompilaci bytecode problematické. Kromě použití speciálního hardwarového zařízení pro dešifrování bytecode neexistuje spolehlivá metoda, pomocí níž by šlo zabránit dekompilaci a následné modifikaci kódu aplikace.

Pro implementaci v rámci této práce byla zvolena metoda šifrování kompilovaných tříd s distribucí šifrovacího klíče pomocí licenčního mechanismu. Pro zvýšení úrovně bezpečnosti je vhodné tuto metodu kombinovat například s obfuskováním kódu aplikace nebo s kompilováním části starající se o dešifrování tříd do nativního kódu.

KAPITOLA 5

Implementace

V této kapitole je popsán způsob implementace jednotlivých částí navrženého systému a jsou popsané použité technologie.

5.1 Klientská část

Klientskou část systému tvoří plugin pro aplikace využívající framework OSGi. Plugin poskytuje aplikaci, do které je přidán, rozhraní pro získávání informací o vystavené licenci z licenčního serveru. Umožňuje také dešifrovat zašifrované `.class` soubory aplikace pomocí klíče získaného společně s licencí.

5.1.1 Použité technologie

Při implementaci klientské části byly použity následující technologie:

- **Java** – Klientská část je napsaná v jazyce Java.
- **Eclipse RCP/OSGi** - Framework OSGi slouží ke správě závislostí na ostatní pluginy.
- **C++** – V jazyce C++ je napsána knihovna pro získávání hardwarového otisku počítače na operačních systémech Windows.
- **WMI** – Windows Management Instrumentation (WMI) rozhraní je použito pro čtení informací o počítači na operačním systému Windows.
- **JNI** – Pomocí JNI je řešena komunikace s knihovnou pro čtení otisků počítače.
- **SWT** – Pomocné třídy zobrazující dialogy pro získání licenčního čísla jsou implementovány pomocí knihovny SWT.

5.1.2 Čtení hardwarového otisku počítače

Jedním z problémů při implementaci klientské části bylo nalezení způsobu, jak získat unikátní otisk počítače. Otisk počítače je používán pro vystavování licencí s omezenou možností opakované instalace licencované aplikace. Aby bylo možné otisk efektivně použít, musí splňovat tyto vlastnosti:

- **Unikátnost** – v ideálním případě by měl mít každý počítač svůj unikátní otisk, pomocí nějž by ho bylo možné jednoznačně identifikovat. Bez dostatečně unikátního otisku není možné mít kontrolu nad počtem aktivovaných licencí.
- **Stálost** – otisk počítače by se neměl v průběhu času měnit. Při opakovaném spuštění nebo instalaci programu na stejném počítači by měl být načtený otisk stejný. Pokud by se otisk příliš často měnil, bylo by nutné neustále vytvářet nové aktivace licencí a omezení na počet aktivací by nebylo možné použít.

Aplikace často pro identifikaci počítače využívají MAC adresu síťových karet. MAC adresa je snadno dostupná (číst MAC adresu je možné i z Javy) a navíc je i unikátní pro většinu počítačů. Použití MAC adresy jako unikátní otisk počítače není vhodné – kromě toho, že adresu lze snadno změnit, má ale i další nevýhody. Tou hlavní je způsob, jakým se operační systémy chovají při vypínání síťových adaptérů. Často se stává, že pokud je síťová karta vypnutá (například vypnutá Wi-Fi u přenosných počítačů, vytažený síťový kabel), systém přestane přítomnost síťové karty hlásit. To může způsobit problémy při ověřování licence vázané právě na hodnotu MAC adresy vypnuté síťové karty. Na operačním systému Windows navíc aplikace často vytvářejí velké množství virtuálních síťových adaptérů (řádově to mohou být i stovky), mezi nimiž je nalezení adresy síťové karty velmi obtížné.

Jako nejvhodnější se pro účely pořízení otisku počítače nabízí využít sériová čísla hardwarových komponent počítače. Protože jazyk Java poskytuje pouze velmi základní přístup k informacím o hardware, je potřeba získání otisků řešit individuálně pro každý podporovaný operační systém.

Operační systém Windows umožňuje získávání informací o hardware pomocí rozhraní WMI. Rozhraní WMI umožňuje číst informace o operačním systému i o různých hardwarových komponentách, přičemž nevyžaduje žádná speciální oprávnění. Díky tomu je možné přečíst například sériové číslo základní desky, pevného disku nebo identifikátor kopie operačního systému Windows. Přístup k těmto údajům je ovšem možný pouze pomocí API v jazyce C++. Proto byla pro přístup k nim vytvořena knihovna v jazyce C++, která je pomocí JNI volána z klientské části napsané v Javě.

U operačního systému Linux je situace mnohem komplikovanější. Linux oproti Windows neposkytuje žádné vhodné rozhraní pro čtení hardwarových informací. Navíc většina metod pro přístup k sérovým číslům hardwarových

komponent vyžaduje oprávnění uživatele root, což je pro použití v běžných aplikacích nemyslitelné. Pro vytváření otisku byla zvolena kombinace hostid a informací o hardwarové konfiguraci počítače (bez sériových čísel). Kombinace těchto dvou údajů je dostatečná pro identifikaci konkrétního počítače.

5.1.3 Šifrování tříd

Klientská aplikace poskytuje oproti ostatním dostupným nástrojům pro licencování aplikací jednu vlastnost navíc, a to možnost mít `.class` soubory uložené v zašifrované podobě a dešifrovat až za běhu aplikace. Samotné šifrování `.class` souborů se obecně nepovažuje za vlastnost zvyšující zabezpečení. Třídy je nutné při načtení dešifrovat a při běhu aplikace jsou v paměti načteny v nešifrované podobě, navíc je nutné mít v aplikaci uložený šifrovací klíč. Výhodou navrženého licencovacího nástroje je možnost distribuovat šifrovací klíče společně s licencemi. Aplikaci je možné rozdělit na jednotlivé části a ty samostatně licencovat, každou část poté zašifrovat vlastním klíčem. Dešifrovat je tedy možné pouze ty části aplikace, které jsou zahrnuty ve vystavené licenci. Ostatní části aplikace zůstávají zašifrované a není je možné dekompilovat ani jinak obejít ochranu.

Pro dešifrování `.class` souborů je využito mechanismu Adaptor Hooks dostupného v implementaci OSGi frameworku Equinox. Při startu aplikace je zaregistrovaná třída implementující rozhraní `ClassLoadingHook`. Toto rozhraní obsahuje metodu `processClass`, která se volá při načítání každé třídy ještě před tím, než třídu načte classloader. Díky tomu je možné bytecode třídy podle potřeby dešifrovat.

Šifrované soubory s třídami obsahují kromě zašifrovaného obsahu původních souborů také identifikátor šifrovacího klíče, kterým se provedlo šifrování, a inicializační vektor použitý pro inicializaci šifry. Aby bylo možné při načítání rychle odlišit šifrovaný bytecode od nezašifrovaného, je na začátek bytecode šifrovaných tříd umístěn byte 0xEC (`.class` soubory Javy začínají konstantou 0xCA 0xFE 0xBA 0xBE). Pro šifrování tříd je v základním nastavení použita bloková šifra Advanced Encryption Standard (AES) v operačním módu Cipher Block Chaining (CBC).

5.2 Serverová část

Serverovou část tvoří několik tříd pro vystavování a správu licencí sloučených do jedné knihovny. Kromě třídy `LicenseManager`, obsahující logiku pro vystavování a ověřování licencí, jsou součástí knihovny také rozhraní pro napojení na datová úložiště a pro komunikaci s klientskými aplikacemi. Pro komunikaci mezi klientskými aplikacemi a serverem je připravena implementace pomocí RESTful webové služby.

5.2.1 Použité technologie

- **Java** – serverová část je napsaná v jazyce Java.
- **Apache Maven** – závislosti na cizích knihovnách jsou u serverové části řešeny pomocí nástroje Apache Maven.
- **JAXB** – pro serializaci tříd při komunikaci mezi klientem a serverem je použito rozhraní Java Architecture for XML Binding (JAX-B).
- **JAXRS** – základní implementace komponenty pro komunikaci mezi klientem a serverem je implementována pomocí rozhraní Java API for RESTful Web Services (JAX-RS).

5.3 Rozšiřitelnost

Serverová část je implementována s důrazem na možnosti rozšíření funkcionality a integrace s existujícími serverovými aplikacemi. Hlavním úkolem serverové části je ověřovat licence a na základě údajů z databáze tyto licence aktivovat. Důležité je zde napojení na existující datový model a využití příslušné persistentní vrstvy.

Z tohoto důvodu je datový model serverové vrstvy definován pomocí rozhraní. Každé entitě je definováno rozhraní obsahující gettery a případně settery pro všechny požadované atributy. Díky tomu je možné při integraci použít existující model pouze doplněný o chybějící položky. Pokud již například máme entitu podobnou licenci a chceme ji použít i pro serverový správce licencí, stačí, když tato entita bude implementovat rozhraní `License`. Podobně je možné postupovat i u ostatních entit.

Pro načítání a ukládání dat je připraveno rozhraní `DataSource`. Toto rozhraní obsahuje metody potřebné serverovou částí pro vyhledání existujících licencí a vytvoření nových aktivací. Pomocí tohoto rozhraní jsou také řešeny relace mezi entitami. Rozhraní obsahuje metody pro získání aktivací a vlastností příslušející k licenci.

5.4 Webové rozhraní

Součástí implementovaného systému je také webová aplikace, na které je demonstrováno použití serverové části (viz sekce 5.2). Webová aplikace využívá serverovou část pro komunikaci s klientskými aplikacemi pomocí aplikačního rozhraní a přidává webové GUI pro možnost ruční správy vystavených licencí.

5.4.1 Použité technologie

- **Java** – webová aplikace je napsaná v jazyce Java

- **JSP** – webové uživatelské rozhraní je vytvořeno pomocí technologie Java Server Pages (JSP).
- **Twitter Bootstrap** – webové GUI bylo vytvořeno s pomocí kolekce nástrojů pro vytváření webových aplikací Twitter Bootstrap.
- **Jersey** – jako implementace rozhraní JAX-RS je použita referenční implementace Jersey[6] od firmy Oracle. Jersey je také použito jako kontroler pro webové rozhraní.
- **Google Guice** – dependency injection framework Guice[5] je použit pro správu závislotí mezi jednotlivými částmi aplikace.
- **Google App Engine** - webová aplikace je postavena na platformě Google App Engine.
- **App Engine DataStore** – pro persistenci dat je použito úložiště App Engine DataStore, ke kterému je přistupováno pomocí rozhraní Java Data Objects (JDO).

5.4.2 Použití Google App Engine

Google App Engine je Platform as a Service (PaaS) cloudová platforma pro vývoj a hostování webových aplikací. Použití Google App Engine jako platformy pro webovou aplikaci umožňuje využít cloudového hostingu od firmy Google, který je zdarma do vyčerpání určitého množství výpočetního výkonu a přenesených dat. Hostování aplikace v cloudu umožňuje velmi dobré škálování výkonu. Při velkém nárůstu počtu požadavků se automaticky spustí více instancí aplikace a zátěž se rovnoměrně rozloží.

Použití platformy Google App Engine sebou nese jistá omezení, které bylo nutné zohlednit při návrhu a implementaci webového rozhraní. Mezi nejdůležitější patří:

- Aplikace nemůže vytvářet a spouštět nová vlákna. Spouštět lze pouze kód volaný pomocí HTTP požadavku.
- Použití souborového systému je omezené na používání virtuálního souborového systému GaeVFS. Ten například umožňuje otevřít pouze jeden soubor pro zápis v rámci virtuálního stroje.
- Podporována je pouze část tříd ze standardní knihovny Javy.
- Aplikace může být kdykoliv vypnuta (typicky po určité době nečinnosti), nastartuje se až při dalším požadavku.

Protože navržená webová aplikace nepotřebuje spouštět vlákna, přistupovat k souborovému systému nebo využívat některé z nepodporovaných tříd, při

implementaci bylo nutné se zaměřit hlavně na problém s častým startováním aplikace. Proto byly místo běžně používaných frameworků pro webové aplikace, jako je například Spring nebo Hibernate, použity frameworky s menšími nároky na systémové prostředky (jmenovitě Google Guice[5] pro dependency injection a JDO pro persistenci dat).

Jako persistentní vrstva je v aplikaci použit High Replication Datastore, ke kterému se přistupuje pomocí rozhraní JDO. Jedná se o bezeschémovou NoSQL databázi, jejíž obsah je replikován mezi několika datacentry. Díky tomu poskytuje databáze vysokou úroveň dostupnosti. Oproti relačním databázím má ale několik omezení. Nepodporuje operaci join, nad daty nelze provádět agregace (ani zjistit počet entit) a data jsou hierarchicky organizována. Z toho důvodu je v modelu entita `Feature` rozdělena na `FeatureJDO` a `AssignedFeatureJDO`. `FeatureJDO` představuje jednu vlastnost licence, přičemž více licencí může sdílet jednu vlastnost. Vlastnost přiřazená k licenci má navíc ještě data platnosti. U relační databáze by se to řešilo vytvořením párovací tabulky mezi tabulkou s vlastnostmi a tabulkou s licencemi. Při použití datastore se při přiřazení vlastnosti k licenci vytvoří její kopie v podobě entity `AssignedFeatureJDO`, která obsahuje navíc data platnosti. Ta se poté přiřadí k požadované licenci.

5.5 Nástroj pro šifrování tříd

Pro usnadnění použití šifrování tříd v klientské aplikaci byla vytvořená konzolová aplikace umožňující šifrovat celé jar soubory se zkompilovanými třídami. Využívá se při tom stejný algoritmus a formát šifrovaných tříd jako v klientské aplikaci (popsáno v sekci 5.1.3).

Šifrování `.jar` souborů probíhá v následujících krocích:

1. Jar soubor je načten do paměti.

2. Ze souboru MANIFEST.MF jsou přečteny informace o třídách, které se mají zpracovat (viz dále).

3. Jednotlivé položky z archivu jsou postupně překopírovány do nového archívu v paměti, `.class` soubory jsou zašifrovány.

4. Na disk je uložena zašifrovaná verze archívu.

Při zpracování je nejprve celý `.jar` soubor načten do paměti, výsledný soubor je také nejprve ukládán do paměti a po dokončení zpracování uložen na disk. Toto řešení je navrženo z výkonostních důvodů. Při čtení a ukládání přímo na disk byla rychlost zpracování řádově 10× menší než při načítání do paměti.

Pokud nechceme, aby byly některé třídy v `.jar` souboru při zpracování zašifrovány (protože se například načítají před tím, než je k dispozici dešifrovací klíč), je možné je ze šifrování vyloučit. Vyloučené třídy jsou při zpracování pouze zkopírovány. Vyloučení třídy je možné provést přidáním atributu `Encryption-Exclude` do souboru `MANIFEST.MF`. Jako hodnota atributu je seznam vzorů použitých pro vyloučení tříd oddělené čárkou. Podporovány jsou tyto tři typy vzorů:

- Název balíčku, například `com.test.mypackage`. Všechny třídy v balíčku `com.test.mypackage` budou při zpracování pouze zkopírovány.
- Plně kvalifikované jméno třídy, například `com.test.mypackage.MyClass`. Třída `com.test.mypackage.MyClass` bude při zpracování pouze zkopírována.
- Název balíčku se znakem *, například `com.test.*`. Všechny třídy v balíčku `com.test` a ve všech podbalíčcích budou při zpracování pouze zkopírovány.

Při zašifrování a opětovném dešifrování `.jar` souboru dochází ke změně metadat v souboru. Dešifrovaný `.jar` soubor může mít jiný kontrolní součet než originál. Samotná data jsou ovšem nedotčena, digitálně podepsaný archív je možné úspěšně ověřit i po zašifrování a dešifrování.

Aplikace čte všechny příkazy z parametrů specifikovaných na příkazové řádce při spuštění. To umožňuje například zařadit volání šifrovacího nástroje jako další krok při automatickém sestavování výsledné licencované aplikace.

KAPITOLA 6

Testování

Cílem této kapitoly je provést testování navrženého systému.

6.1 Jednotkové testování

Ověření základní funkčnosti jednotlivých navržených komponent bylo provedeno pomocí jednotkových testů. Cílem jednotkového testování je provést ověření správné funkcionality nejmenších programových jednotek. V rámci jazyka Java jsou tyto jednotky třídy a metody.

Testy byly napsány s použitím frameworku JUnit4[7]. Tento framework byl zvolen hlavně pro snadnost použití a dobrou integrovanost s použitými vývojovými nástroji. Integrace s vývojovým prostředí Eclipse umožňuje hromadně spouštět testy a získávat podrobné vyhodnocení výsledků. Při použití s nástrojem Maven jsou testy spouštěny v rámci sestavování výsledných `.jar` souborů.

Navržený systém (obzvláště serverová část) obsahuje množství vzájemně komunikujících součástí, přičemž část z nich tvoří pouze rozhraní, jejichž implementace musí být dodána až při integraci do existující aplikace. Pro testování je potřeba pro tyto části vytvořit náhradní implementaci poskytující minimální funkčnost potřebnou pro konkrétní test (tzv. mock). K tomuto účelu byla zvolená knihovna Mockito[8]. Kromě vytváření mock objektů s definovaným chováním umožňuje také testovat splnění různých podmínek – například ověřit pořadí volaných metod na mock objektu.

Testována byla klientská i serverová část. U částí sestavovaných nástrojem Maven jsou testy spouštěny automaticky při sestavení. U klientské části jsou testy umístěny v samostatném fragmentu a musejí být pouštěny v běžící instanci RCP aplikace. Celkové počty a výsledky testů jsou uvedeny v tabulce 6.1.

Projekt	Počet testů	Počet úspěšných testů
mp-clientside	8	8
mp-common	12	12
mp-serverside	13	13

Tabulka 6.1: Výsledky unit testů

6.2 Výkonnostní testování

6.2.1 Klientská část

Hardwarová konfigurace

Testování klientské části navrženého systému bylo provedeno na počítači s konfigurací uvedenou v tabulce 6.2.

Typ Procesoru	Intel Core 2 Duo P8400
Frekvence Procesoru	2,26 GHz
Počet jader procesoru	2
Velikost operační paměti	3 GB
Pevný disk	WD3200BEVT, 320 GB, 5200 rpm, 8 MB cache

Tabulka 6.2: Hardwarová konfigurace počítače pro testování klientské části

Testování probíhalo na operačním systému Windows 7. Pro měření času byla použita funkce `System.nanoTime()`. Naměřené hodnoty byly získány jako průměr z několika opakovaných pokusů.

Testovací data

Pro testování šifrování byl vybrán vzorek různě velkých `.jar` archívů tak, aby se co nejvěrněji simulovalo použití ve skutečné aplikaci. Vlastnosti vybraného vzorku souborů jsou shrnuty v tabulce 6.3.

Počet souborů	584
Celková velikost souborů	311,65 MB
Celkový počet tříd	91873
Celkový počet ostatních položek archívů	140214

Tabulka 6.3: Vlastnosti vybraného vzorku .jar archívů pro testování šifrování

Testování nástroje pro šifrování .jar souborů

Testování šifrování bylo provedeno na kompletních testovacích datech. Všechny soubory byly nejprve šifrovány a poté dešifrovány. Aby bylo možné výkon

aplikace lépe posoudit v kontextu použitého hardwaru, byly provedeny ještě dvě srovnávací měření. Jedno pro porovnání s rychlostí disku, kde se soubory pouze zkopírují. Druhé slouží k porovnání rychlosti šifrování, soubory jsou při něm zašifrovány jako celek (nezkoumá se obsah archivu). Výsledky měření jsou v tabulce 6.4.

Operace	Průměrná rychlost	Doba zpracování
Šifrování tříd	3,3 MB/s	94 s
Dešifrování tříd	5,1 MB/s	61 s
Šifrování souborů	12,7 MB/s	23,5 s
Dešifrování souborů	12,4 MB/s	24 s
Kopírování souborů	20 MB/s	14,7 s

Tabulka 6.4: Výsledky měření klientské části

Naměřené hodnoty odpovídají očekávání. Malá rychlost zpracování souborů při šifrování jednotlivých tříd je způsobena hlavně tím, že pro zašifrování pouze `.class` souborů je potřeba archív rozbalit a znovu zabalit. Část výkonu je tak spotřebována na operace spojené se zpracováním archívů jako komprese, dekomprese a výpočet kontrolních součtů.

I přes relativně nízký výkon je program schopen zpracovat vstupy o velikosti stovek megabytů v řádech minut. Při reálném nasazení by pravděpodobně byly šifrovány jen některé knihovny (velkou část aplikací postavených na platformě Eclipse RCP obvykle tvoří volně dostupné knihovny, které nemá smysl šifrovat), takže reálná velikost šifrovaných souborů by se pohybovala spíše v řádech desítek megabytů. Tento proces je navíc potřeba provádět pouze jednou před samotným distribuováním aplikace, takže i v případě delší doby zpracování nedojde ke zpomalení při používání aplikace.

Testování klientské části

Na klientské části systému jsou z hlediska výkonu 2 důležité části – část starající se o dešifrování tříd a část získávající hardwarové otisky počítače.

Testování dešifrovací části bylo provedeno na instalaci aplikace postavené na platformě Eclipse RCP. Pluginy tvořící aplikaci byly zašifrovány pomocí nástroje pro šifrování tříd souborů (viz 5.5). Při spuštění byly třídy z těchto pluginů před načtením dešifrovány, měřena byla rychlost dešifrování tříd.

Při testování části získávající hardwarové otisky počítače bylo testováno, jak dlouhá je odezva při volání nativních metod získávající jednotlivé atributy.

Všechny naměřené hodnoty jsou v tabulce 6.5.

Velká rychlost dešifrování tříd (v porovnání s rychlostmi dosaženými při použití samostatného nástroje, viz tabulka 6.4) je způsobena tím, že třídy jsou k dešifrování předkládány již v dekomprimované podobě. Dosažená rychlost je závislá především na rychlosti procesoru. Dosažená rychlost na testovacím

Operace	Rychlost zpracování
Dešifrování tříd	19 MB/s
Inicializace nativní knihovny	10 ms
Získání jednoho údaje o počítači	<10 ms

Tabulka 6.5: Výsledky měření klientské části

stroji 19 MB/s znamená, že každých cca. 20 MB/s šifrovaných tříd přidá 1 s k době potřebné pro start aplikace.

6.2.2 Serverová část

Serverová část je postavena na platformě Google App Engine, z toho důvodu byly testy serverové části prováděny při nasazení v cloudovém hostingu. Testovaná aplikace byla nasazena na instancích serveru s výkonem odpovídajícím počítači s procesorem o frekvenci 600 MHz a 128 MB operační paměti. Pro zlepšení výkonu byla použita cache a odstraněny indexy z atributů, které se nepoužívají k vyhledávání.

Testování propustnosti

Cílem tohoto testu je změřit, jak velkou propustnost má serverová čáast navrženého systému. Propustností se rozumí počet požadavků za časový úsek, který je schopna serverová část zpracovat.

Testování probíhalo na aplikaci nasazené v prostředí Google App Engine. Pro porovnání byla aplikace testována také lokálně (konfigurace počítače je v tabulce 6.2) na webovém serveru Jetty.

Pro testování byl použit nástroj Apache JMeter[1]. JMeter je desktopová aplikace napsaná v Javě určená pro provádění zátěžových testů a měření výkonu. Pro testování bylo použito 50 vláken, každé provádělo opakovaně požadavky na informace o různých vystavených licencích stejně, jako by byl odeslány z klientské aplikace.

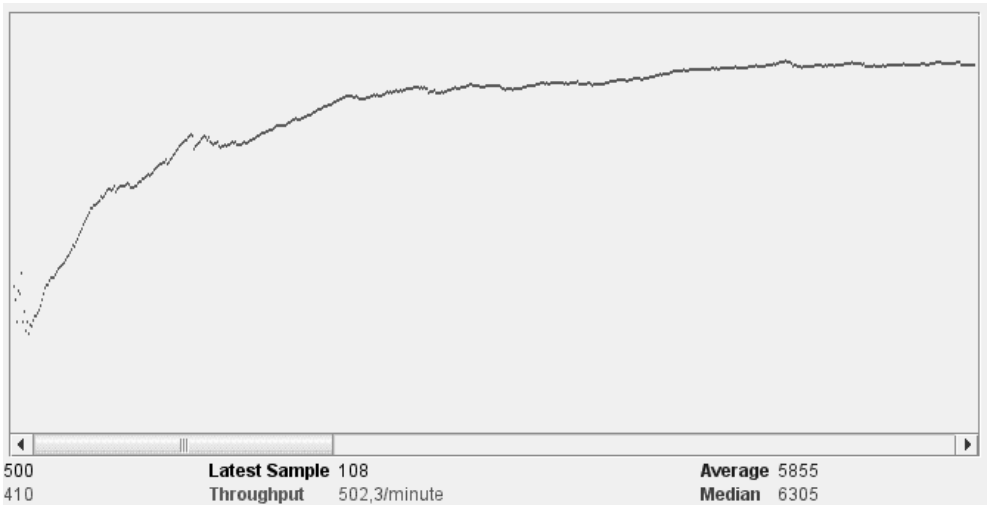
Protože aplikace nasazené na Google App Engine zdrama mají nastavené limity na počet operací nad databází, bylo možné provádět testy pouze po omezenou dobu. Ukázalo se, že při provádění zátěžových testů dochází poměrně rychle k vyčerpání limitů pro čtení z datastore.

Naměřené hodnoty jsou uvedeny v tabulce 6.6.

Propustnost lokálně nasazené aplikace	≈ 500 req/min
Propustnost aplikace na App Engine	1700+ req/min

Tabulka 6.6: Výsledky testů propustnosti klientské části

Zatímco lokálně nasazená aplikace je schopná obsloužit maximálně kolem 500 požadavků za minutu, u aplikace nasazené na Google App Engine byla naměřena propustnost více než 1700 požadavků za minutu. Toto číslo navíc není konečné, v průběhu testování docházelo po celou dobu měření k růstu naměřené propustnosti. Pokud by byly testy prováděny déle, výsledná naměřená propustnost by byla pravděpodobně mnohem větší. To je způsobeno tím, že aplikace nasazené na Google App Engine mohou běžet ve více instancích. V případě, že aplikace nestíhá obsloužit všechny požadavky, jsou spuštěny další instance. Průběh měření je možné vidět na obrázcích 6.1 a 6.2. Aplikace spuštěná na lokálním stroji rychle dosáhne maxima požadavků, které zvládne obsloužit, počet požadavků zpracovaných aplikaci nasazenou na Google App Engine po celou dobu měření roste.

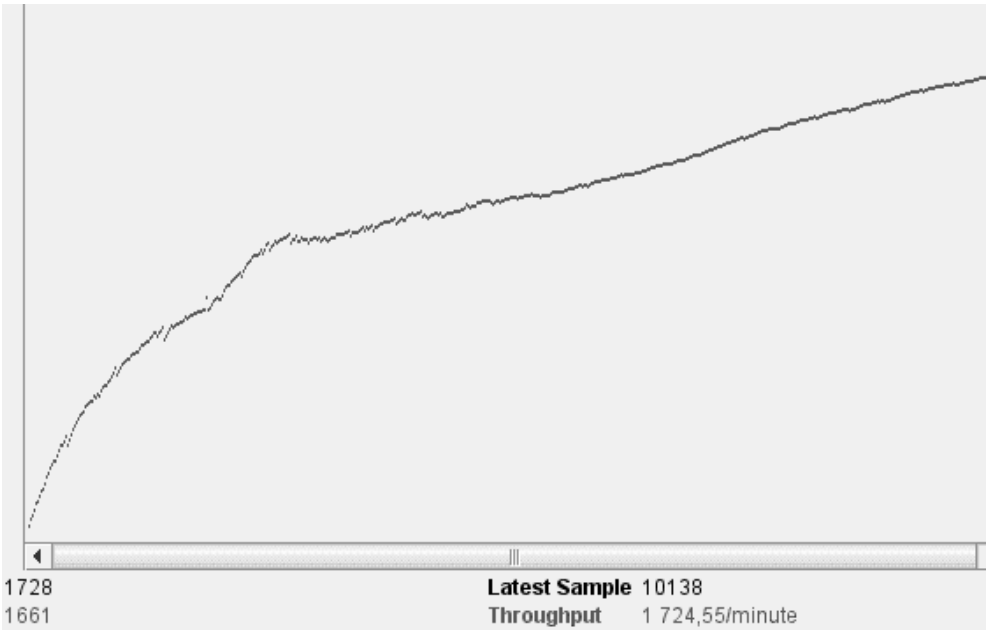


Obrázek 6.1: Ukázka průběhu měření propustnosti serverové části při lokálním nasazení

Testování vytížení serveru

Tento test má za úkol zjistit množství systémových prostředků, které serverová aplikace využívá. To je důležité zejména z toho důvodu, aplikace nasazené na Google app Engine mají na systémové prostředky nastaveny různé kvóty. Po překročení kvót je aplikace zastavena, dokud nedojde k jejich obnovení (kvóty se obnovují každých 24 hodin).

Pro měření hodnot bylo použito administrační rozhraní, které poskytuje přehled o čerpání jednotlivých kvót, a také webový filtr `AppstatsFilter`, který umožňuje získat detailní přehled o využívání zdrojů každým požadavkem na webovou aplikaci.



Obrázek 6.2: Ukázka průběhu měření propustnosti serverové části při nasazení na Google App Engine

V tabulce 6.7 je ukázka provedených operací při čtení informace o licenci, v tabulce 6.8 pak operace provedené při aktivaci licence.

Typ operace	Počet volání	Doba provádění
Čtení dat z cache	6	17 ms
Spuštění dotazu nad databází	5	50 ms

Tabulka 6.7: Využití systémových prostředků při načtení informace o licenci

Typ operace	Počet volání	Doba provádění
Čtení dat z cache	8	21 ms
Spuštění dotazu nad databází	5	57 ms
Zápis do databáze	4	143 ms
Mazání dat z cache	4	11 ms

Tabulka 6.8: Využití systémových prostředků při aktivaci nové licence

Aby bylo možné získat lepší představu o náročnosti serverové aplikace na systémové prostředky, bylo provedeno měření, jehož cílem bylo zjistit počet

požadavků, které serverová aplikace zpracuje než dojde k vyčerpání zdarma poskytnutých kvót. V době testování aplikace byly limity nastaveny následovně:

- 28 hodin procesorového času
- 50 000 zapisovacích operací do Datastore
- 50 000 čtecích operací z Datastore

Test byl prováděn odesíláním požadavků na informace o aktivacích různých licencí. Výsledné naměřené hodnoty jsou uvedeny v tabulce 6.9.

Počet dotazů na informace o aktivaci licence	$\approx$ 5800
Počet využitých procesorových hodin	1,28/28
Počet čtení z Datastore	50 000/50 000
Počet zápisů do Datastore	<500
Úspěšnost nalezení dat v cache	98%

Tabulka 6.9: Výsledek testu na počet operací do vyčerpání kvót poskytnutých zdarma

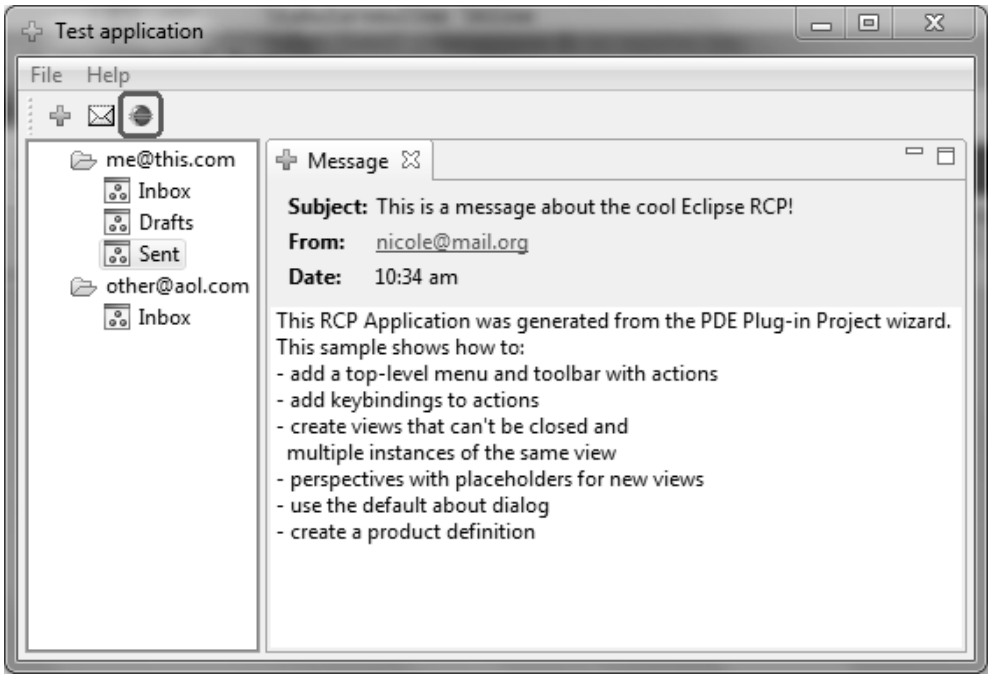
Naměřená data ukazují, že zdarma poskytnuté kvóty na Google App Engine vystačí aplikaci na zpracování přibližně 5800 požadavků (to odpovídá přibližně 4 požadavkům za minutu). Aplikace i přes použití cache nejvíce využívá operace pro čtení dat z Datastore. Ukazuje se, že Datastore není příliš vhodný pro modely obsahující více relací.

6.3 Ukázková aplikace

Pro otestování celkové funkčnosti a použitelnosti navrženého systému byla vytvořena ukázková aplikace.

6.3.1 Popis aplikace

Jedná se o aplikaci postavenou na platformě Eclipse RCP. GUI. Grafické rozhraní aplikace představuje jednoduchého klienta pro elektornickou poštu. K aplikaci je ještě přidán plugin s dodatečnou funkcionalitou. Tento plugin přidává do nástrojové lišty vlastní tlačítko, jehož stisknutí otevírá dialogové okno. Vzhled testované aplikace je zachycen na obrázku 6.3, červeně je označeno tlačítko dodané pomocí pluginu.



Obrázek 6.3: Ukázka uživatelského rozhraní testovací aplikace

6.3.2 Integrace licencovacího nástroje

Před integrací licencovacího nástroje je nutné nejprve určit, na které jednotlivé vlastnosti bude aplikace rozdělena a kterou funkcionalitu budou tyto vlastnosti zahrnovat. Ukázková aplikace je rozdělena následovně:

- **TEST-APP** – Vlastnost potřebná pro samotné spuštění aplikace. Pokud licence nebude obsahovat tuto vlastnost, aplikace nepůjde spustit.
- **EXTENDED-FEATURE** – Vlastnost potřebná pro použití funkcionality dodané externím pluginem. Bez této vlastnosti aplikace půjde spustit, ale funkcionalitu poskytovanou pluginem nebude možné využít.

Protože bez vlastnosti `TEST-APP` není možné aplikaci spustit, je potřeba získat údaje o licenci od uživatele ještě před tím, než se aplikace spustí. Čím dříve se při startu získá platná licence, tím více tříd je možné zašifrovat pomocí šifrovacího nástroje (šifrovat lze jen třídy, které se nenačítají před získáním platné licence). Jako vhodné místo pro umístění kódu pro získání licence byla zvolena třída `InteractiveSplashHandler`. Pomocí této třídy lze upravit chování úvodního obrázku s logem, který se zobrazuje při startu aplikace (tzv. splash screen). Začlenění se provede přidáním následujícího kódu:

```
@Override
public void init(Shell splash) {
  super.init(splash);

  LicenseActivationDialog dlg = new
      LicenseActivationDialog(splash);
  dlg.setTitle("Provide license number");
  dlg.setMessage("Please insert your license number");
  dlg.setShellTitle("License activation");

  LicenseInformation info = null;
  ExecutorService exec = Executors.
      newSingleThreadExecutor();
  try {
    info = LicenseHelper.getValidLicenseFromUI(dlg, exec
        , "TEST-APP");
  } finally {
    exec.shutdown();
    if (info == null) {
      System.exit(1);
    }
  }
}
```

Listing 6.1: Kód přidaný do třídy InteractiveSplashHandler

Při startu aplikace je uživatel vyzván k zadání licenčního čísla (pro získání je použit dialog, který je součástí klientské části navrženého systému), pokud zadané číslo není platné nebo neobsahuje vlastnost `TEST-APP`, aplikace je ukončena.

Je vhodné přidat kontrolu platnosti aplikace na více míst. Samotný splash handler jde totiž poměrně snadno deaktivovat a to pouze upravením XML konfiguračního souboru. Z tohoto důvodu je podobný kód vložen ještě na další místa, v případě že není nalezena platná licence je aplikace ukončena. Kód použitý dále v aplikaci pro ověřování licence je na následující ukázce

```
LicenseInformation info = LicenseService.getInstance().
    getCurrent();

if (info == null || info.containsFeature("TEST-APP") ==
    null) {
  MessageBox box = new MessageBox(PlatformUI.
      getWorkbench().getActiveWorkbenchWindow().getShell
      (), SWT.ICON_ERROR);
  box.setText("Error");
  box.setMessage("Failed to load license information");
  box.open();
  PlatformUI.getWorkbench().close();
```

```
}
```

Listing 6.2: Kód přidaný na další místa v aplikaci pro ověření licence

Přidání omezení funkcionality pro plugin poskytující tlačítko v nástrojové liště aplikace je provedeno podobným způsobem. Do třídy `Command`, která má na starosti spouštění akce po stisknutí tlačítka, je přidán následující kód:

```
@Override
public boolean isEnabled() {
  LicenseInformation info = LicenseService.getInstance()
      .getCurrent();
  return info != null && info.containsFeature("EXTENDED-
      FEATURE") != null;
}
```

Listing 6.3: Kód přidaný do třídy Command

Tento kód umožní, že tlačítko lze použít jen v případě, že aktuálně platná licence obsahuje vlastnost `EXTENDED-FEATURE`.

6.3.3 Použití šifrovacího nástroje

Jakmile je do aplikace integrován licenční mechanismus, je možné `.jar` soubory tvořící aplikaci zašifrovat pomocí šifrovacího nástroje. Licenční mechanismus umožňuje ke každé vlastnosti přidat navíc šifrovací klíč, proto budou `.jar` soubory tvořící aplikaci šifrovány samostatně. Z tohoto důvodu je vhodné mít každou vlastnost šifrovanou vlastním klíčem vyčleněnou do samostatného pluginu.

Aby nedošlo k zašifrování tříd potřebných pro start aplikace, které jsou načteny ještě před získáním licence, je potřeba tyto třídy vyloučit přidáním nastavení do souboru `MANIFEST.MF`. Konfigurace pro `.jar` soubor s aplikací vypadá následovně:

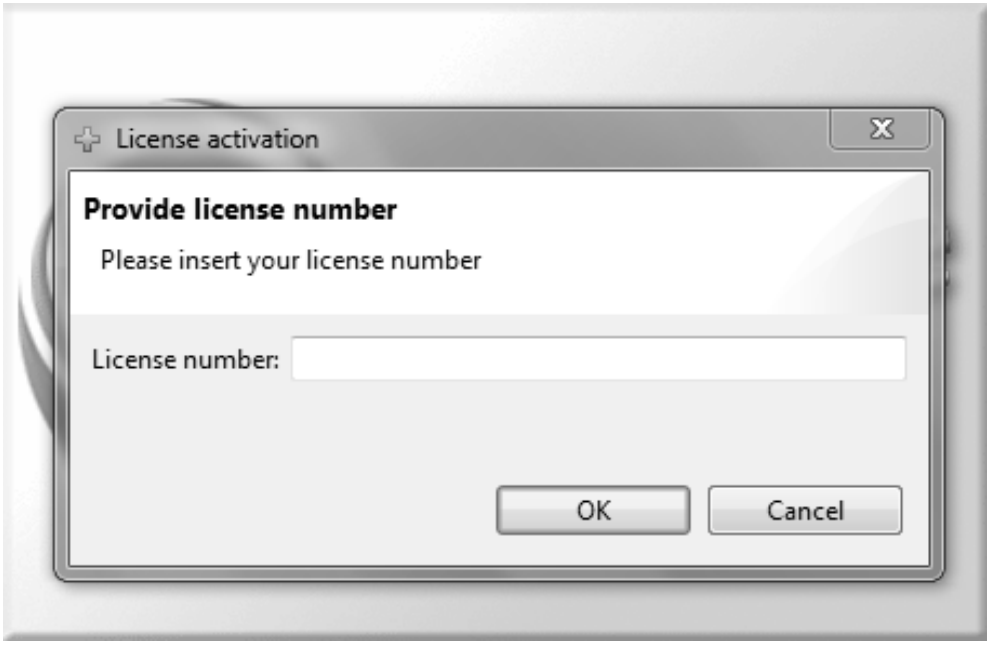
```
Encryption-Exclude: cz.cvut.fit.vybirjan.mp.testapp,
 cz.cvut.fit.vybirjan.mp.testapp.splashHandlers
```

Listing 6.4: Konfigurace přidaná do souboru MANIFEST.MF

6.3.4 Výsledek

Při spuštění testovací aplikace je nutné zadat platné licenční číslo pro licenci obsahující vlastnost `TEST-APP`, jinak dojde k vypnutí aplikace. Ukázka dialogu zobrazeného při startu aplikace je na obrázku 6.4.

Bez platné licence není možné aplikaci spustit, nelze ale ani ochranu z aplikace odstranit. Při odstranění kódu, který provádí kontrolu platnosti licence ve třídě `InteractiveSplashHandler` sice nedodje k zobrazení výzvy pro zadání licenčního čísla, spuštění aplikace ale poté selže neboť nebude možné načíst zašifrované třídy tvořící aplikaci.



Obrázek 6.4: Ukázka dialogu pro zadání licenčního čísla

Zkompilovaná testovací aplikace je přiložena na CD (viz příloha C). Testovací data pro aplikaci je možné nahrát do spuštěného licenčního serveru zavoláním URL `/init`.

Závěr

Cílem práce bylo prozkoumat existující licencovací nástroje, navrhnout a implemnetovat vlastní systém podporující aplikace postavené na platformě Eclipse RCP.

Rešerše dostupných nástrojů ukázala, že existující řešení jsou buď velmi jednoduché, nebo naopak složité s pořizovací cenou pohybující se v řádu tisíců dolarů. Žádné z dostupných řešení se navíc nezabývalo problematikou neautorizované modifikace kódu aplikace nebo integrace s desktopovými aplikacemi využívajícími platformu Eclipse RCP/OSGi.

Z tohoto důvodu byla navržen, implementován a otestován vlastní systém pro licencování aplikací. Navržený systém využíva celou řadu technologií – od nativního kódu v C++, přes desktopovou platformu Eclipse RCP až po webové technologie jako JAX-RS nebo Google App Engine. Oproti existujícím řešením má navržený systém jednu unikátní vlastnost, a to možnost šifrovat zkompilovaný kód aplikace a provést dešifrování až za běhu pomocí klíče získaného z licenčního serveru. Společně s knihovnami pro integraci do desktopové a serverové aplikace bylo také implementováno webové uživatelské rozhraní, nástroj pro šifrování `.jar` souborů a ukázková aplikace, na které bylo demonstrováno použití celého systému.

Jako další pokračování by mohlo být přidání podpory pro plovoucí licence, zlepšit podporu pro sběr hardwarových otisků na operačním systému Linux a přidání integrace šifrovacího nástroje s nástroji pro automatické sestavování apliakcí.

Literatura

[1] Apache JMeter. Available at WWW: <http://jmeter.apache.org/>

[2] ASM - Java bytecode manipulation and analysis framework. [Cited 2012-03-30]. Available at WWW: <http://asm.ow2.org/>

[3] Excelsior JET. [Cited 2012-03-31]. Available at WWW: <http://www.excelsior-usa.com/jet.html>

[4] GNU Compiler for Java. [Cited 2012-03-31]. Available at WWW: <http://gcc.gnu.org/java/>

[5] Google Guice. Available at WWW: <http://code.google.com/p/google-guice/>

[6] Jersey. Available at WWW: <http://jersey.java.net/>

[7] JUnit4. Available at WWW: <http://www.junit.org/>

[8] Mockito. Available at WWW: <http://code.google.com/p/mockito/>

[9] Validy SoftNaos for Java. [Cited 2012-04-01]. Available at WWW: <http://www.validy.com/en/products/softnaos/>

[10] Extranet Software Solutions, I.: Rampart. [Cited 2012-02-18]. Available at WWW: <http://rampartlicensing.com>

[11] Fielding, R.; Gettys, J.; Mogul, J.; etc.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999, updated by RFCs 2817, 5785, 6266. Available at WWW: <http://www.ietf.org/rfc/rfc2616.txt>

[12] Inc., S. M.: The Java Virtual Machine Specification. 1999, [Cited 2012-02-19]. Available at WWW: <http://java.sun.com/docs/books/jvms/second_edition/html/ClassFile.doc.html>

[13] jProductivity, L.: Protection! 4. [Cited 2012-02-18]. Available at WWW: <http://www.jproductivity.com/products/protection/protection.htm>

[14] Leskov, D.: Java Bytecode Encryption Revisited. 2008, [Cited 2012-04-01]. Available at WWW: <http://www.excelsior-usa.com/blog/excelsior-jet/java-bytecode-encryption-revisited/>

[15] License4j: License4j. [Cited 2012-02-18]. Available at WWW: <http://www.license4j.com/>

[16] WEBsina, I.: JLicense. [Cited 2012-02-18]. Available at WWW: <http://www.websina.com/products/jlicense.html>

[17] X-Formation: LM-X License Manager. [Cited 2012-02-19]. Available at WWW: <http://www.x-formation.com/lm-x_license_manager/index.html>

PŘÍLOHA A

Slovník

Glossary

Advanced Encryption Standard je symetrická bloková šifra využívající bloky o velikosti 128 bitů. Šifra AES je využívána například u bezdrátových Wi-Fi sítí v rámci zabezpečení WPA2..

Cipher Block Chaining je operační mód blokové šifry, u kterého je před zašifrováním bloku proveden XOR s předchozím blokem..

Eclipse Rich Clilent Platform je součást projektu Eclipse zaměřující se na tvorbu bohatých desktopových aplikací.

Extensible Markup Language je značkovací jazyk, který definuje pravidla pro zápis dokumentů v textové podobě vodoné pro strojové čtení.

Graphic User Interface je uživatelské rozhraní, které umožňuje ovládat počítač pomocí interaktivních grafických ovládacích prvků..

Hypertext Transfer Protocol je jeden ze základních protokolů používaných ke komunikaci na internetu..

Java Native Access je knihovna, která usnadňuje propojení kódu běžícího ve virtuálním stroji Javy s nativními programy a knihovnami. Narozdíl od JNI není při propojení aplikací pomocí JNA potřeba speciálních hlavičkových souborů v jazyce C. .

Java Native Interface je rozhraní umožňující propojit kód běžící ve virtuálním stroji Javy s nativními programy a knihovnami napsanými v jiných jazycích..

JavaScript Object Notation je textový formát pro přenos dat odvozený od jazyja JavaScript umožňující jendoduchý zápis datových struktur.

MAC adresa je unikátní identifikátor síťového zařízení, který je využíván různými komunikačními protokoly druhé síťové vrstvy..

NoSQL je typ databázového systému, který se liší od běžně používaných relačních databází například nepoužíváním relačního schématu..

Open Services Gateway initiative framework je specifikace dynamického modulárního systému pro jazyk Java.

Software Development Kit je sada softwarových nástrojů umožňující tvorbu aplikací pro určitý softwarový balík, platformu, operační systém nebo hardware.

Standard Widget Toolkit je knihovna pro vytváření grafického uživatelského rozhraní pro jazyk Java. Narozdíl od knihovny Swing, které je součástí standardní distribuce Javy, knihovna SWT využívá pro vykreslování grafických prvků volání nativních metod operačního systému. Díky tomu má grafické uživatelské rozhraní vytvořené pomocí knihovny SWT stejný vzhled jako nativní aplikace..

Windows Management Instrumentation je rozhraní poskytované operačním systémem Windows, pomocí něhož je možné číst informace o různých komponentách systému..

PŘÍLOHA B

Seznam použitých zkratek

Acronyms

AES Advanced Encryption Standard.

API Application programming interface.

CBC Cipher Block Chaining.

DAO Data Access Object.

GUI Graphic User Interface.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

JAX-B Java Architecture for XML Binding.

JAX-RS Java API for RESTful Web Services.

JDO Java Data Objects.

JNA Java Native Access.

JNI Java Native Interface.

JSON JavaScript Object Notation.

JSP Java Server Pages.

MAC Media Access Control.

OSGi Open Services Gateway initiative framework.

PaaS Platform as a Service.

REST Representational State Transfer.

SDK Software Development Kit.

SWT Standard Widget Toolkit.

URL Unified Resource Locator.

WMI Windows Management Instrumentation.

XML Extensible Markup Language.

PŘÍLOHA C

Obsah CD

- `readme.txt` ..... Informace o obsahu CD
- `src`
  - `impl` ..... Složka se zdrojovými kódy programů
    - `fplib` ..... Nativní knihovna pro sběr otisků počítače
    - `mp-clientside` ..... Klientská část
    - `mp-clientside-test` ..... Testy klientské části
    - `mp-common` ..... Společná knihovna
    - `mp-encryptor` ..... Apliace pro šifrování tříd
    - `mp-serverside` ..... Serverová část
    - `mp-testapp` ..... Testovací aplikace
    - `mp-testapp-extension` ..... Plugin do testovací aplikace
    - `mp-web` ..... Serverová aplikace s webovým rozhraním
  - `latex` ..... Složka se zdrojovými kódy práce v LaTeX
- `bin` ..... Složka se zkompilovanými aplikacemi
  - `cz.cvut.fit.vybirjan.mp.client_1.0.0.jar` ..... Zkompilovaná klientská část
  - `mp-common-1.0.0.jar` ..... Zkompilovaná společná knihovna
  - `mp-encryptor-1.0.0.jar` ..... Zkompilovaný šifrovancí nástroj
  - `mp-serverside-1.0.0.jar` ..... Zkompilovaná serverová knihovna
  - `mp-testapp-linux_x86.zip` ..... Testovací aplikace pro Linux 32b
  - `mp-testapp-linux_x64.zip` ..... Testovací aplikace pro Linux 64b
  - `mp-testapp-win32.zip` ..... Testovací aplikace pro Windows 32b
  - `mp-testapp-win32_x64.zip` ..... Testovací aplikace pro Windows 64b
- `text` ..... Složka s textem práce
  - `DP_Vybiral_Jan_2012.pdf` ..... Text práce ve formátu PDF

PŘÍLOHA D

Uživatelská příručka

D.1 Použití klientské části

Před zahájením používání klientksé části je potřeba nakonfigurovat licenční službu. To se provede voláním metody `LicenseService.configure()`. Při konfiguraci je pořeba nastavit:

- Veřejný klíč pro ověřování platnosti licence vystavené serverem.
- URL na licenční server.
- Identifikátor aplikace, podle kterého je na serveru vybrán klíč pro podepsání licenční informace.
- Přepínač, který určuje, zda se má použít šifrované spojení s licenčním serverem.

Tuto konfiguraci je vhodné provést co nejdříve při startu apliakce. Nabízí se například kód umístit do aktivátoru některého z pluginů aplikace. Inicializace může vypadat následovně:

```
Properties props = new Properties();
props.load(Activator.class.getResourceAsStream(
    CONFIG_FILE));

LicenseService.configure(new LicenseServiceConfig(props.
    getProperty(CONFIG_APPID),
     Utils.deserialize(Utils.decode(props.getProperty(
         CONFIG_KEY)), Key.class),
     props.getProperty(CONFIG_HOST), Boolean.parseBoolean
         (props.getProperty(CONFIG_HTTPS))));
```

Listing D.1: Inicializace licenční služby v klientské aplikaci

Konfiguraci je možné provést právě jednou. Opakované volání metody `LicenseService.configur` vyvolá vyjímku.
Jakmile je licenční služba nakonfigurovaná, je možné volat metody pro získání informací o licenci. Při volání licenční služby je nutné nejprve získat instanci voláním metody `LicenseService.getInstance()`. Licenční služba poskytuje následující metody:

- `getCurrent()` – Vrátí informace o aktuální licenci.
- `clearCurrentLicense()` – Vymaže informace o aktuální licenci, licene je odstraněna i z persistentního úložiště.
- `checkOffline()` – Provede kontrolu aktuální licence bez připojení k serveru.
- `checkOnline()` – Provede kontrolu aktuální licence ověřením na serveru.
- `activateLicense()` – Aktivuje licenci s daným licenčním číslem na serveru a nastaví ji jako aktuální.

Pokud uživatel potřebuje například aktualizovat GUI podle aktuálně platné licence, je možné využít metodu `addLicenseChangedListener()`. Listener předaný této metodě bude notifikovaný vždy, když dojde ke změně licence.

D.2 Použití serverové části

Serverovou část je možné integrovat do libovolné aplikace napsané v jazyce Java. Při integraci je potřeba vytvořit instanci třídy `LicenseManager` a předávat jí požadavky od klientských aplikací pomocí metod `activateLicense()` a `getLicense()`. Pro vytvoření instance třídy `LicenseManager` je potřeba předat jí implementace následujících rozhraní, které umožňují integraci se zbytkem aplikace:

- `DataSource` – Rozhraní datasource slouží pro přístup do persistentního úložiště.
- `EntityFactory` – Rozhraní pro vytváření instancí entit. Slouží pro oddělení třídy `LicenseManager` od konkrétní implementace doménového modelu.
- `ResponseKeyProvider` – Rozhraní pro získávání šifrovacích klíčů k podepsání odpovědi posílaných klientským aplikacím. Klíče je možné pomocí tohoto rozhraní číst například z databáze.

Pro zpracování požadavků od klienta je v serverové části připravena implementace RESTful webové služby pomocí JAX-RS. Pro její použití je potřeba mít v serverové aplikaci zprovozněnou některou z implementací JAX-RS.

Při použití Jersey je nejprve potřeba nastavit, aby se při konfiguraci Jersey prohledali balíčky obsahující implementaci webové služby pro komunikaci s klientskými aplikacemi.
Třída implementující webovou službu potřebuje získat referenci na nakonfigurovanou instanci třídy `LicenseService`. Správa závislostí se obvykle řeší použitím některého z dependency injection frameworků (ve webové aplikaci implementováné v rámci této práce je použit Google Guice). Aby zůstala serverová část nezávislá na použitém dependency injection frameworku, využívá se injektování závislostí dostupných ve standardu JAX-RS. V serverové aplikaci je nutné vytvořit třídu implementující rozhraní `ContextResolver`, která v metodě `getContext` vrátí nakonfigurovanou instanci třídy `LicenseManager`. Takto vytvořenou třídu je potřeba ještě označit anotací `@Provider` a umístit do balíčku, který je zahrnut do prohledávání konfigurace v Jersey. Možná implementace je na ukázce F.1.
Při použití Jersey v podobě servletu může výsledná konfigurace v souboru `web.xml` vypadat jako v ukázce F.2. URL webové služby, kterou je potřeba zadat do klientské aplikace bude `adresa serveru/namapovaná cesta Jersey servletu/licenses`.

D.3 Použití serverové aplikace

Implementovaná serverová aplikace poskytuje webovou službu pro vystavování licencí klientským aplikacím a webové uživatelské rozhraní pro manuální správu licencí. Aplikace je určená pro nasazení do Google App Engine. Pro sestavení je potřebné vývojové prostředí Eclipse a instalovaný Google Plugin for Eclipse.
Tetovací verze aplikace je nasazena na Google App Engine, dostupná je na URL `vybirjan-mp.appspot.com`.
Dostupné adresy:

- `/init` – Přístupem na tuto adresu se naplní databáze testovacími daty.
- `/activations` – URL pro klientské aplikace pro aktivaci licencí.
- `/web/index` – Hlavní stránka webového uživatelského rozhraní.

Webová aplikace obsahuje 3 hlavní sekce:

Issued Licenses

Stránka obsahující seznam všech vystavených licencí. Pomocí formuláře umožňuje přidávat, editovat a mazat vystavené licence. Ke každé licenci umožňuje zobrazit seznam provedených aktivací včetně hardwarových otisků počítačů, ze kterých byly aktivace provedeny. Ukázka je na obrázku E.5.

Features

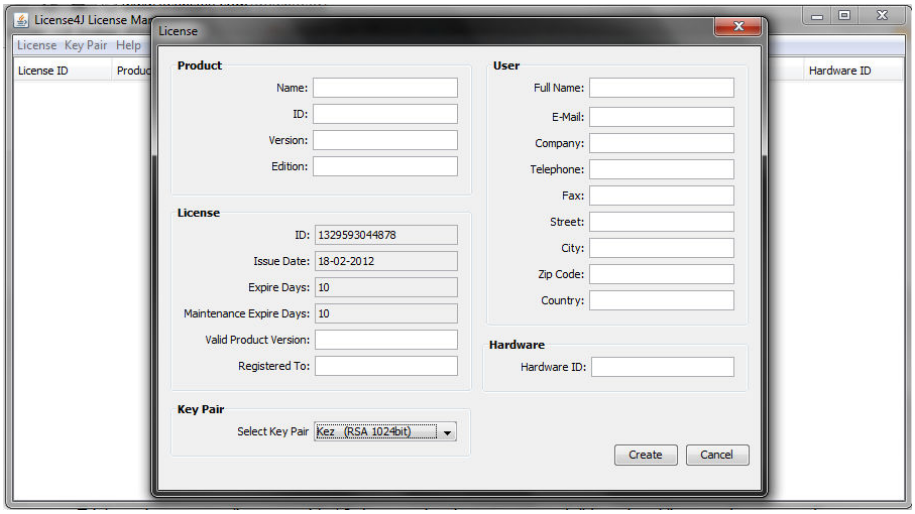
Seznam vlastností, které je možné přiřadit k licencím. Stránka umožňuje vytvářet a mazat vlastnosti. Ke každé vlastnosti je možné vygenerovat nebo vložit šifrovací klíč, který může být poté použit pro zašifrování částí klientské aplikace. Každou vlastnost vytvořenou na této stránce je možné přiřadit k licenci ve formuláři pro editaci licencí. Ukázka je na obrázku E.6.

Encryption keys

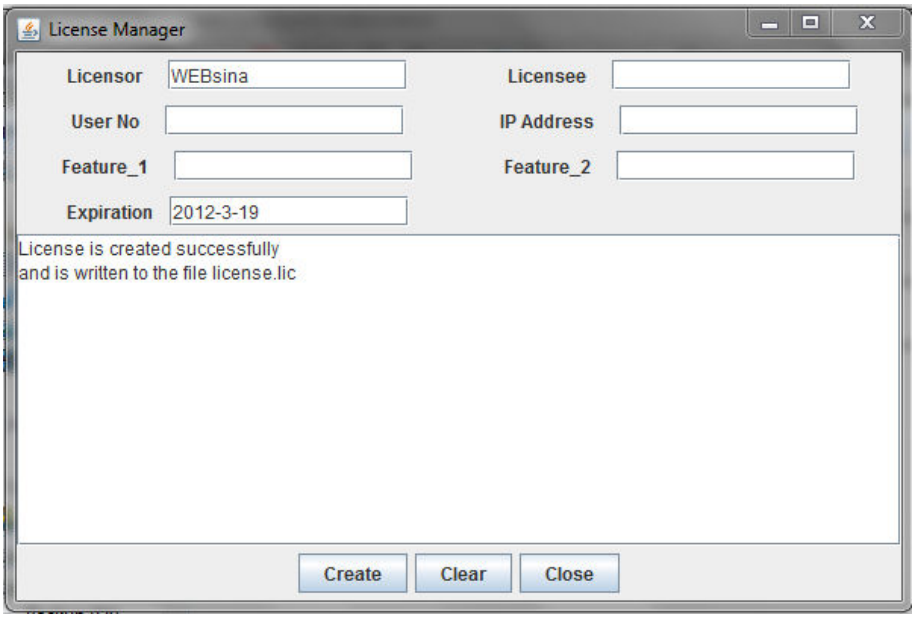
Stránka pro správu šifrovacích klíčů. Pomocí těchto klíčů je podepisována licence odesílaná na klientskou aplikaci. Na stránce je možné vygenerovat novou dvojici šifrovacích klíčů (soukromý a veřejný), přičemž veřejný klíč je možné následně zabudovat do aplikace. Ukázka je na obrázku E.7.

PŘÍLOHA E

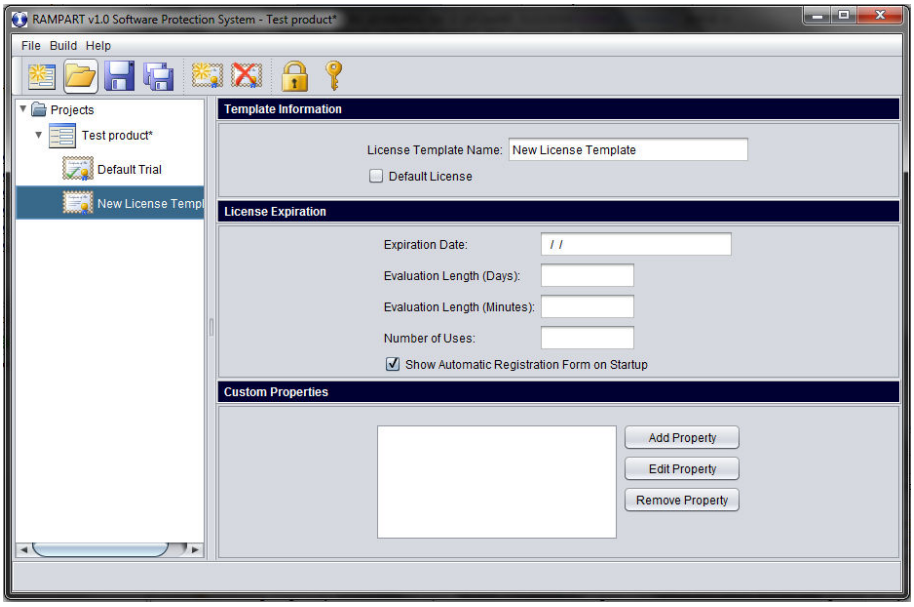
Obrázky



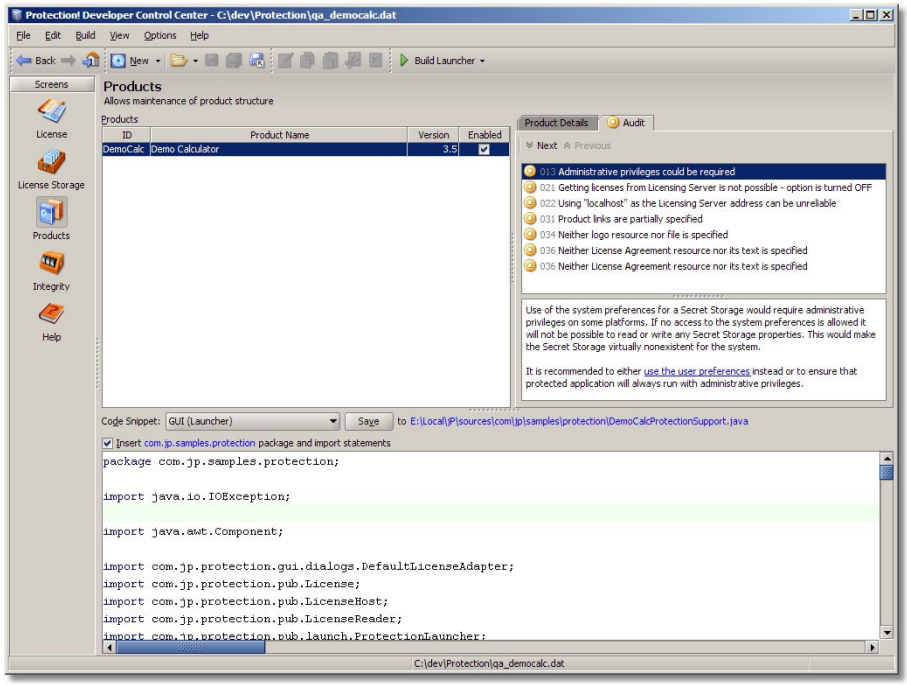
Obrázek E.1: License4j UI pro tvorbu licencí



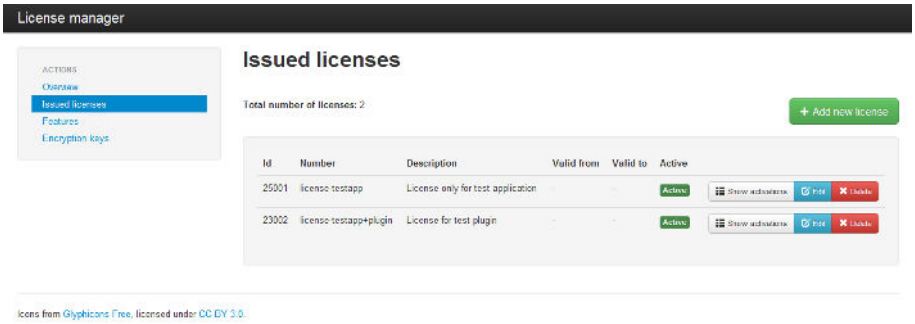
Obrázek E.2: JLicense UI pro tvorbu licencí



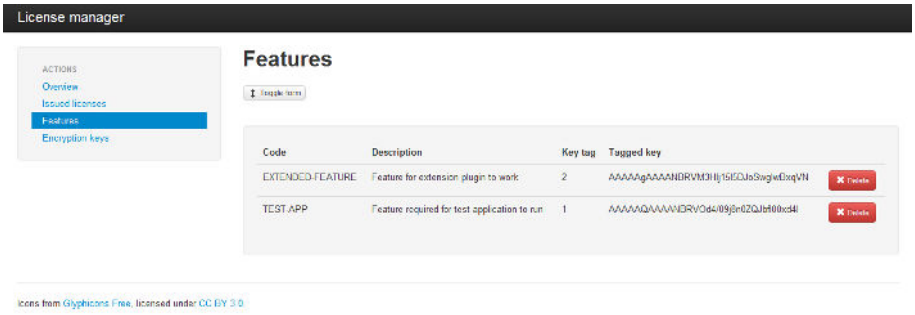
Obrázek E.3: Rampart UI pro tvorbu licencí



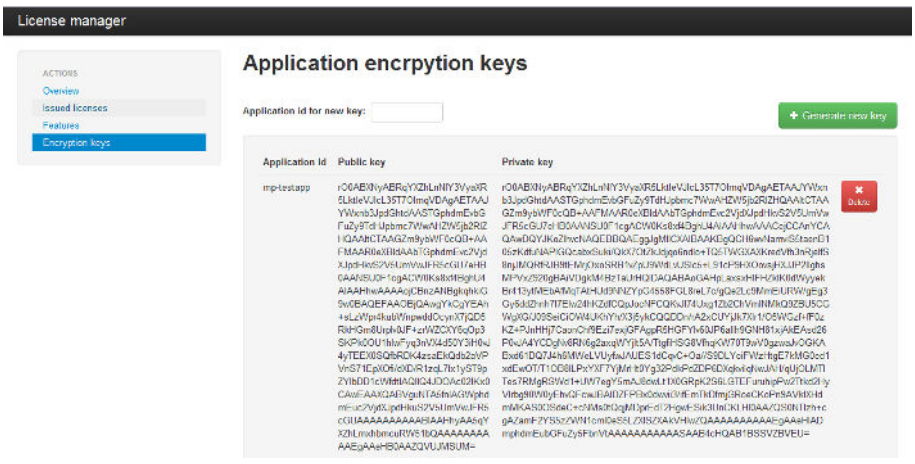
Obrázek E.4: Protection developer control center



Obrázek E.5: Ukázka webového uživatelského rozhraní - seznam vystavených licencí



Obrázek E.6: Ukázka webového uživatelského rozhraní - seznam vlastností



Obrázek E.7: Ukázka webového uživatelského rozhraní - seznam šifrovacích klíčů

PŘÍLOHA F

Zdrojové kódy

```
@Provider
public class LicenseManagerFactory implements
    ContextResolver<LicenseManager> {

  private static LicenseManager createLicenseManager() {
    //create instance of LicenseManager here
  }

  @Override
  public LicenseManager getContext(Class<?> arg0) {
    if (arg0.equals(LicenseManager.class)) {
      return createLicenseManager();
    } else {
      return null;
    }
  }

}
```

Listing F.1: Implementace třídy pro vytváření instance LicenseManager

```
<servlet>
    <servlet-name>Jersey Web Application</servlet-name>
    <servlet-class>com.sun.jersey.spi.container.servlet.
        ServletContainer</servlet-class>
    <init-param>
        <param-name>com.sun.jersey.config.property.
            packages</param-name>
        <param-value>cz.cvut.fit.vybirjan.mp.serverside.
            impl.jaxrs,com.test.otherpackage</param-value>
    </init-param>
</servlet>
```

Listing F.2: Ukázková konfigurace Jersey v souboru web.xml

```
public class Decompilation {

  public enum TestEnum {
    A, B, C
  }

  private static int integerVariable = 0;
  static {
    integerVariable = 5 * (22 + 16);
  }

  private boolean flag = false;

  public TestEnum getValue() {
    return TestEnum.values()[(int) (System.
        currentTimeMillis() % 3)];
  }

  public String testMethod() {
    String localunusedVar = "test" + " integerVariable= "
        " + integerVariable + ".";
    float variable = Float.MAX_VALUE;
    while (flag) {
      variable -= Float.MAX_VALUE / 10;
      flag = variable > 0;
    }
    switch (getValue()) {
      case A:
        System.out.print("A");
        break;
    }
    for (int var = 0;; var++) {
      if (var > 100) {
        break;
      }
      System.out.println(var);
    }
    return testObj.toString();
  }

  private final Object testObj = new Object() {
    @Override
    public String toString() {
      return String.valueOf(integerVariable);
    };
  };
}
}
```

Listing F.3: Originální zdrojový kód třídy

```
public class Decompilation {
  private static int integerVariable = 0;
  private boolean flag = false;
  private final Object testObj = new Object() {
    public String toString() {
      return String.valueOf(Decompilation.
          integerVariable);
    }
  };
  static {
    integerVariable = 190;
  }
  public TestEnum getValue() {
    return TestEnum.values()[(int) (System.
        currentTimeMillis() % 3L)];
  }
  public String testMethod() {
    new StringBuilder("test integerVariable=").append(
        integerVariable).append(".").toString();
    float f = 3.4028235E+38F;
    while (this.flag) {
      f -= 3.402824E+037F;
      this.flag = (f > 0.0F);
    }
    switch ($SWITCH_TABLE$test$Decompilation$TestEnum()[
        getValue().ordinal()]) {
      case 1:
        System.out.print("A");
        break;
    }
    for (int i = 0; i <= 100; i++)
      System.out.println(i);
    return this.testObj.toString();
  }

  public static void main(String[] paramArrayOfString) {
    Decompilation localDecompilation = new Decompilation
        ();
    localDecompilation.testMethod();
  }
  public static enum TestEnum {
    A, B, C;
  }
}
```

Listing F.4: language=Java, Dekompilovaný zdrojový kód třídy

PŘÍLOHA G

Přehled projektů

mp-common

Maven projekt obsahující společné třídy pro komunikaci mezi klientskou a serverovou částí a také další pomocné třídy.

Nastavení pro nástroj Maven

```
<dependency>
  <groupId>cz.cvut.fit.vybirjan.mp</groupId>
  <artifactId>mp-common</artifactId>
  <version>1.0.0</version>
</dependency>
```

Listing G.1: Konfigurace pro nástroj Maven projektu mp-common

Přehled obsahu balíčků projektu

```
cz.fit.cvut.vybirjan.mp
├── common ... Pomocné třídy pro klientskou
│              a serverovou část.
│   ├── comm ... Třídy pro komunikaci mezi
│   │   │        klientskou a serverovou
│   │   │        aplikací.
│   │   └── xml ... Pomocné třídy pro serializaci
│   │       │       pomocí JAX-B.
│   │       └── marshallable ... Pomocné třídy pro serializaci
│   │                            pomocí JAX-B.
│   └── crypto ... Třídy pro šiforvání a
│                  podepisování.
```

mp-clientside

Plug-in projekt obsahující všechny potřebné třídy pro integraci licencovacího systému do desktopové aplikace.

Přehled obsahu balíčků projektu

```
cz.fit.cvut.vybirjan.mp
└── clientside ... Veřejné třídy pro práci
    │              s licencemi v klientksé
    │              aplikaci.
    ├── internal ... Prefix označující privátní
    │   │            třídy.
    │   ├── client ... Třídy pro komunikaci se
    │   │              serverovou aplikací.
    │   ├── core ... Implementace tříd pro práci
    │   │            s licencemi.
    │   ├── fingerprints ... Obecná implementace třídy pro
    │   │                    sběr hardwarových otisků.
    │   │   ├── linux ... Třídy pro sběr hardwarových
    │   │   │             otisků na OS Linux.
    │   │   └── win32 ... Třídy pro sběr hardwarových
    │   │                 otisků na OS Windows.
    │   ├── hook ... Implementace AdaptorHook
    │   │           pro dešiforvání tříd za běhu
    │   │           aplikace.
    │   └── storage ... Implementace bezpečného
    │                   úložiště pro informace
    │                   o licenci.
    └── ui ... Třídy obsahující pomocné UI
               prvky.
```

mp-clientside-test

Fragment projekt obsahující testy pro projekt mp-clientside. Testy je nutné spouštět v běžící instanci RCP aplikace.

mp-encryptor

Projekt obsahující konzolovou aplikaci pro šiforvání a dešifrování `.jar` souborů.

mp-serverside

Maven projekt obsahující všechny potřebné třídy pro integraci licencovacího serveru do webové aplikace.

```
<dependency>
  <groupId>cz.cvut.fit.vybirjan.mp</groupId>
  <artifactId>mp-serverside</artifactId>
  <version>1.0.0</version>
</dependency>
```

Listing G.2: Konfigurace pro nástroj Maven projektu mp-serverside

mp-testapp

Projekt obsahující desktopovou aplikaci se vzorovým použitím licencovacího nástroje.

mp-testapp-extension

Projekt obsahující plugin do testovací aplikace. Demonstruje použití licencovacího nástroje pro omezení funkčnosti jednotlivých komponent aplikace a použití šifrovacího nástroje.

mp-web

Webová aplikace obsahující implementaci serveru pro distribuci licencí. Aplikace také obsahuje grafické uživatelské rozhraní pro vydávání licencí.

Přehled obsahu balíčků projektu

```
cz.fit.cvut.vybirjan.mp.web
└── controllers ... Kontrolery pro generování
    │               uživatelského rozhraní.
    ├── dao ... Rozhraní pro objekty
    │   │       přistupující do databáze.
    │   └── impl ... Implementace objektů
    │       │        přistupujících do databáze.
    │       └── licensemanager ... Implementace objektů
    │                              přistupujících do databáze
    │                              pro licenční server.
    ├── dto ... Objekty pro přenos dat pro
    │           uživatelské rozhraní.
    └── guice ... Konfigurační třídy pro Google
                  Guice.
```

```
├── license ... Implementace tříd pro
│              licenční server.
└── model ... Implementace doménového
              modelu webové aplikace.
```

fplib

Projekt pro Microsoft Visual Studio 2010 obsahující zdrojové kódy pro nativní knihovnu umožňující získat hardwarové otisky v operačním systému Windows. Pro správné sestavení je nutné mít ve Windows definovanou systémovou proměnnou `JAVA_HOME` ukazující do složky s instalovaným JDK.