ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

ДИПЛОМНА РАБОТА

Тема: Система за управление на пакети

Дипломант:
Ясен Ефремов

Научен ръководител:
Валентин Върбанов

СОФИЯ

2023 г.

Използвани съкращения

- REST
- HTTP
- OOP
- OS
- RAII - Resource Acquisition Is Initialization or RAII

Увод

В днешно време хората използват по-голямо количество софтуер от когато и да било. Като започнем от ежедневно употребяваните програми, а именно уеб браузърът, редакторите за документи, софтуерът за обработка на изображения, видео и аудио, и стигнем чак до видеоигрите. Средностатистическият потребител, е свикнал да си набавя всякакъв вид софтуер като просто потърси за него в интернет пространството, намери достатъчно достоверно изглеждаща уеб страница, от която да го свали, и изтегли приложението под формата на автоматизиран инсталатор (файлове с разширение .msi, .deb, .dmg) или директно изпълним файл (файлове с разширение .exe). Този подход обаче има недостатъци, най-големият от които е сигурността. Освен с полезен софтуер, интернет пространството е пълно и със зловреден софтуер. По-неопитните потребители лесно биха се заблудили от рекламите, фалшивите бутони за инсталация и предупредителните съобщения в някои уеб страници, в резултат на което биха си инсталирали вирус, спайуер (spyware) или рансъмуер (ransomware), който криптира файловете на компютъра и иска паричен откуп, за да ги възстанови. Има обаче начини да се предпазим от всички тези опасности, един от които е системата за управление на пакети (package manager). Тя ни позволява да намираме и инсталираме софтуер безопасно под формата на пакети, валидирани от истински хора и съхранявани в сигурни хранилища. Такива системи са например winget за Windows, apt за Ubuntu и homebrew за MacOS.

Системите за управление на пакети обаче могат да бъдат полезни и за самите разработчици на софтуер. Една от основните им функции е да знаят от какви други програми (библиотеки) е зависима дадена програма и съответно да управляват тези зависимости. Големите софтуерни проекти са съставени от множество модули, които зависят едни от други. Освен това всеки модул има различни версии, всяка от които може да предоставя различни функционалности. Някои модули зависят от точно определена версия на други модули. Така се създава една доста заплетена мрежа от зависимости. Ако трябваше програмистът ръчно да се грижи за всичко това, работата му щеше да се удвои. За щастие почти всеки език за програмиране в днешно време има своя система за управление на пакети. Езикът Python има системата pip, езикът JavaScript има системата npm, езикът Java има две такива системи - maven и gradle, и дори Rust, език за системно програмиране, сравним със C и C++, има своя система, наречена cargo. За жалост едни от езиците, които нямат такава общоприета система са именно C и C++. Има опити да се направи, но въпреки това голяма част от програмистите предпочитат да се грижат за зависимостите на проектите си ръчно. Една от причините за това е факта, че и двата езика са създадени преди повече от 30 години и съответно следват по-различни практики от съвременните езици за програмиране. И все пак C и C++ продължават да бъдат стандартизирани и до ден днешен, като и двата езика постепенно се сдобиват с нови по-модерни функционалности.

Текущият проект цели реализирането на система за управление на пакети за програмните езици C и C++. Самата система е написана на C и C++, като предоставя конзолен потребителски интерфейс, посредством който могат да се инсталират, премахват и обновяват пакети под формата на хранилища, взети от онлайн платформата GitHub. Системата също така позволява управление на зависимостите на даден пакет, както и създаването на изцяло нови пакети, описани чрез специален файлов формат.

Глава 1

Методи за управление на софтуерни пакети

1.1 Основни концепции

1.1.1 Видове системи за управление на пакети

Съществуват основно два вида системи за управление на пакети:

- Системни - управляват програмите на компютъра. Този вид системи са важен компонент от операционната система GNU/Linux и нейните дистрибуции, като освен популярен софтуер управляват и важни системни библиотеки на самата операционната система (glibc, linux-util, и др.). Една от най-старите такива е dpkg за дистрибуцията Debian, на която са базирани десетки други дистрибуции. Други платформи също предоставят такива системи, като най-популярните за Windows и MacOS са съответно winget и homebrew. Те обаче не са интегрирани по същия начин както при GNU/Linux и затова не са много популярни сред повечето потребители.

- Езикови - управляват библиотеките на определен програмен език. Този вид системи са доста популярни сред модерните програмни езици. Те не зависят от операционната система. Главната им цел е лесното пакетиране и публикуване на библиотеки за конкретния език.

И двата вида системи решават съществуващите проблеми по подобни начини. Текущата дипломна работа се фокусира върху втория вид системи, а именно тези, управляващи библиотеките на определен програмен език.

1.1.2 Видове софтуерни пакети

При платформата GNU/Linux софтуерните пакети се разпространяват чрез вече интегрираната система за управление на пакети. Една програма или библиотека обаче може да бъде пакетирана по два различни начина:

- Двоичен пакет - този вид пакети са предварително компилирани. Те съдържат или изпълнимия двоичен код на програмата, или двоичният код на библиотеката (както и нейните header файлове, ако е написана на C/C++). Името им понякога завършва с наставката "-bin".

- Кодов пакет - този вид пакети се компилират при инсталация. Те съдържат самия код на програмата или библиотеката, като името им завършва с наставката "-git".

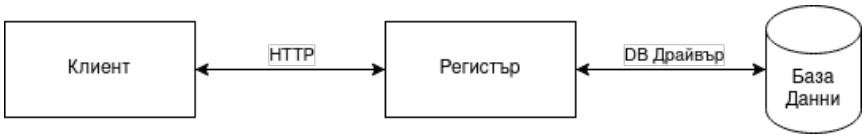
1.1.3 Моделът клиент - регистър

Всяка система за управление на пакети се състои от две основни части (фиг. 1.1):

- Клиент - (също наричан front-end) представлява програмата, с която потребителя взаимодейства и която инсталира, обновява и премахва посочените пакети. Тя може да предоставя графичен или конзолен потребителски интерфейс (сред системите за управление на библиотеки са по-популярни конзолните интерфейси).

- Регистър - (също наричан back-end) най-често представлява уеб приложение, което управлява базата данни, съхраняваща самите пакети и техните метаданни (метаданните са разгледани по-подробно в следващата точка).

Комуникацията между регистъра и клиентското приложение се осъществява посредством приложно-програмен интерфейс (API), предоставен от самия регистър. Най-популярните решения за такъв интерфейс са RESTful и GraphQL архитектурите.



Фигура 1.1: Моделът клиент - регистър

1.1.4 Описание на пакетите (метаданни)

Освен самия код на библиотеката, всеки пакет носи със себе си файл, описващ различните му характеристики. Това са неговите така наричани метаданни. Най-важните от тях са:

- Име - всеки пакет има уникално име, което го идентифицира.

- Версия [1] - даден пакет може да има различни версии. Версията е под формата на три числа, разделени с точки (например

2.11.3). Първото число обозначава главната версия, второто - второстепенната версия и третото - кръпката.

- Зависимости - даден пакет може да зависи от други пакети, които се наричат негови зависимости. Всеки пакет изброява своите зависимости и техните версии под формата на масив.

Често тези метаданни се съхраняват във файлови формати, които позволяват лесно сериализиране, извличане и обработка на данните. Такива формати са например JSON и YAML.

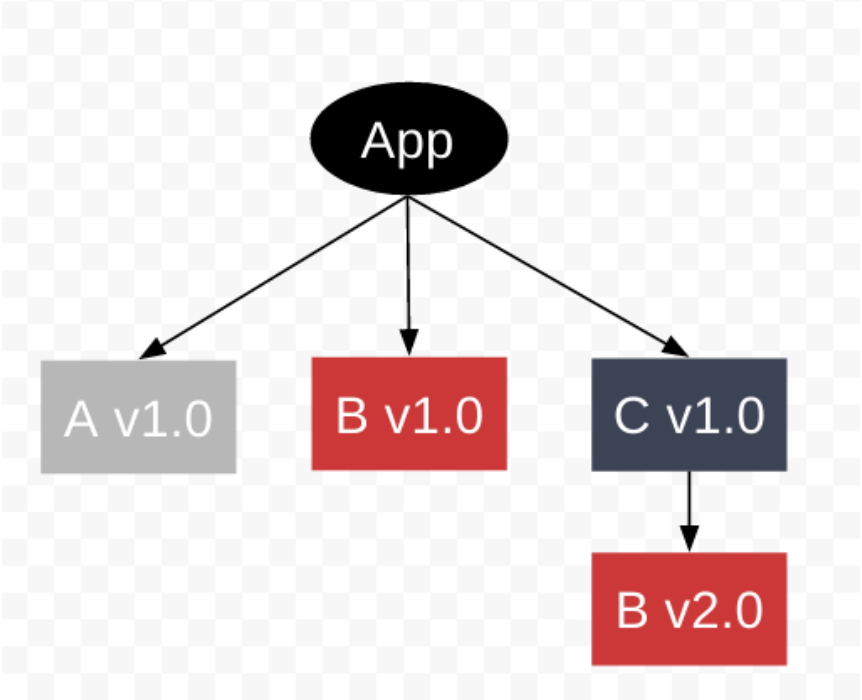
1.1.5 Управление на зависимостите

Едно от най-важните неща, за които се грижи системата за управление на пакети, е управлението на зависимостите на даден пакет. Именно това ни позволява да използваме вече съществуващи външни пакети в нашия проект, без да трябва да се грижим за версиите на техните зависимости.

Зависимостите на даден пакет могат да бъдат представени като ориентиран граф. Всеки връх в този граф представлява даден пакет, а дъгите между върховете ни показват кой пакет от кой зависи. Задачата на една системата за управление на пакети е да разбере кои версии на кои пакети е нужно да бъдат инсталирани, така че да няма повтарящи се, липсващи или конфликтни пакети. Това се постига чрез алгоритъм за разрешаване на зависимостите.

Можем да разделим зависимостите на два вида:

- Директни - пакет A зависи пряко от пакет B
- Преходни - пакет A не зависи пряко от пакет B. Вместо това пакет A зависи от пакет C, който от своя страна зависи от пакет B



Фигура 1.2: Примерен граф на зависимостите

В даден момент графът на зависимостите може да стане толкова голям и сложен, че самата операционна система да не може да го поддържа. Това се изразява в прекалено дълги файлови пътища, получени в резултат на дълбоко вложени папки. Начинът да се предотврати този проблем е така нареченото изравняване на зависимостите. При него всички пакети се съхраняват на едно ниво в една папка, като зависимостите между тях се изразяват чрез символични връзки (symlinks).

От метаданните на пакета се генерира така наречения lockfile, който описва всички зависимости на текущия пакет и техните конкретни версии. Този файл също се публикува заедно с кода на пакета и позволява точното пресъздаване на графа на зависимостите му в различни среди.

1.2 Съществуващи системи

Най-популярните и разпространени системи за управление на пакети са тези за езиците JavaScript и Python.

- npm - системата на JavaScript (акроним за Node Package Manager). При нея всеки пакет се описва чрез файл на име package.json, в който са записани името на пакета, версията, лицензa, зависимостите и т.н.

- pip - системата на Python (рекурсивен акроним за PIP Installs Packages). За разлика от npm, при pip всеки пакет се описва с помощта на няколко файла - requirements.txt, setup.py и .toml.

1.3 Съществуващи системи за C и C++

Въпреки че няма общоприета система за управление на пакети за C и C++, съществуват няколко не много популярни варианта, които си струва да бъдат взети предвид.

- Conan - система, написана на езика Python. Освен конзолен потребителски интерфейс предоставя също и собствен регистър наречен conan center, където са качени повечето известни библиотеки за C и C++.

- vcpkg - система, разработвана от Microsoft, написана на C++. Предоставя конзолен потребителски интерфейс, като за разлика от conan, идеята и е пакетите да могат да се създават лесно от GitHub хранилища и да се интегрират с build системата CMake.

Глава 2

Проектиране на система за управление на пакети

2.1 Функционални изисквания

Текущата дипломна работа цели реализирането на система за управление на пакети за програмните езици C и C++. Като основна функционалност системата трябва да предоставя възможност за:

- Основни действия с пакети:
  - Инсталация от GitHub хранилище
  - Премахване
  - Изброяване на инсталираните пакети
  - Създаване и др.
- Управление на зависимостите на даден пакет
  - Рекурсивна инсталация на всички зависимости на даден пакет

–

- Съхранение на списък с инсталираните пакети
  - Локално инсталираните пакети се съхраняват в JSON файл в текущата директория
  - Глобално инсталираните пакети се съхраняват в sqlite база данни в главната директория на потребителя
- Подаване на обратна връзка на потребителя
  - Изходни съобщения, показващи прогреса на инсталация-/премахване на пакетите
  - Диагностични съобщения, съхранявани в текстов файл в главната директория на потребителя
- Описание на характеристиките на пакетите чрез JSON файлове (метаданни)

2.2 Подбор на средства за разработка

2.2.1 Програмен език

Текущата дипломна работа е реализирана на програмния език C++.

Като версия на езика е избран стандарта ISO/IEC 9899:2017 (също познат като C++17)

- Бърз
- не заема много памет
- много функционалности

- поддържан от главните платформи

CMake, C++17 заради filesystem

- Стар
- труден за писане и разбиране
- трудно управление на зависимостите

2.2.2 Build система

Много програмни езици в днешно време предоставят така наречената система за изграждане (build system). Функциите на една такава система се различават в зависимост от конкретния език, неговото предназначение и възможности, но главната ѝ цел е да направи полесно компилирането/интерпретирането, тестването и пакетирането на софтуер.

Най-разпространената build система за C++ е CMake (съкратено от Cross Platform Make). Тя е поддържана на всички главни операционни системи (Windows, Linux, MacOS). Начинът ѝ на работа е следния:

2.2.3 Система за управление на версиите

Всеки един софтуерен проект в днешно време използва някакъв вид система за управление на версиите. Най-разпространената такава система е git. Тя позволява проследяването на всички промени по кода, като по този начин подпомага сътрудничеството на големи екипи от хора.

2.2.4 Среда за разработка

Текущата дипломна работа бе разработена на операционната система Arch Linux, като за редакция на кода и документацията бе използван текстовият редактор Visual Studio Code и неговото разширение, позволяващо използването на Vim клавиши. Кодът на проекта бе публикуван онлайн в платформата GitHub. Настоящата документация бе написана с помощта на LaTeX.

2.2.5 Постоянна интеграция (CI)

Постоянната интеграция (Continuous Integration) е практика при разработката на софтуер при която работата на много разработчици се обединява на едно място, след което се задейства автоматизирана компилация/интерпретация и тестване на софтуерния продукт.

Текущият проект използва предоставените от платформата GitHub средства за автоматизирана компилация и тестване на кода (GitHub Actions).

2.2.6 Използвани библиотеки

За реализирането на някои от функционалностите на проекта бяха използвани следните популярни библиотеки:

- cpr (C++ Requests) - обвивка (wrapper) на библиотеката curl, позволяваща изпълнение на HTTP заявки през мрежата
- zip - библиотека за компресиране и архивиране
- sqlite - малка релационна база данни, съхранявана в един файл
- spdlog - библиотека, подаваща обратна връзка на потребителя и позволяваща пазенето на диагностични съобщения

- json - библиотека за работа с JSON файлове
- argparse - библиотека, позволяваща лесно подаване на параметри към програмата и тяхното използване

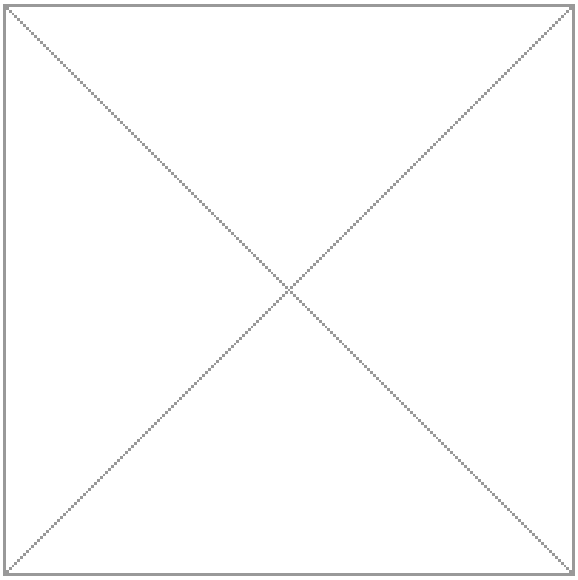
2.2.7 Алгоритъм за управление на зависимостите

NP-трудност
обхождане на граф/дърво?

Глава 3

Реализация на проекта

3.1 Структура на програмата



Фигура 3.1: UML Диаграма на структурата на програмата

- Command pattern

- GitHub API
- XDG dir spec
- Поддръжка за множество платформи

```
#ifndef PACKAGE_H
#define PACKAGE_H

#include <string>

#include <cstddef>


namespace cpm {

    /**
     * @brief A struct representing a cpm package
     */
    struct Package {

        std::string name;
        // Version version;

        bool operator==(const Package &other) const;

        struct PackageHash {
            std::size_t operator()(const Package &package) const noexcept;
        };
    };
}


#endif  // PACKAGE_H
```

Listing 3.1: Структура за пакет

```
argparse::ArgumentParser parser("cpm", CPM_VERSION);


static InstallCommand install_command("install");
```

```
install_command.add_description("Install the specified package/s");
install_command.add_argument("packages")
    .help("Packages to install")
    .required()
    .nargs(argparse::nargs_pattern::at_least_one);
install_command.add_argument("-g", "--global")
    .help("installs package/s globally")
    .default_value(false)
    .implicit_value(true);


static RemoveCommand remove_command("remove");
remove_command.add_description("Remove the specified package/s");
remove_command.add_argument("packages")
    .help("Packages to remove")
    .required()
    .nargs(argparse::nargs_pattern::at_least_one);
remove_command.add_argument("-g", "--global")
    .help("removes package/s globally")
    .default_value(false)
    .implicit_value(true);


static ListCommand list_command("list");
list_command.add_description("List all installed packages");
list_command.add_argument("-g", "--global")
    .help("lists globally installed package/s")
    .default_value(false)
    .implicit_value(true);


static CreateCommand create_command("create");
create_command.add_description("Create a new package");


static SyncCommand sync_command("sync");
sync_command.add_description(
    "Install the package/s specified in the current cpm_pack.json"
);


// argparse::ArgumentParser update_command("update");
// update_command.add_description("Update the specified package/s");
// update_command.add_argument("packages")
```

```
//     .help("updates the specified package/s")
//     .required()
//     .nargs(argparse::nargs_pattern::at_least_one);

parser.add_subparser(install_command);
parser.add_subparser(remove_command);
parser.add_subparser(list_command);
parser.add_subparser(create_command);
parser.add_subparser(sync_command);
// parser.add_subparser(update_command);
commands.insert({"install", &install_command});
commands.insert({"remove", &remove_command});
commands.insert({"list", &list_command});
commands.insert({"create", &create_command});
commands.insert({"sync", &sync_command});
// commands.insert({"update", &update_command});
```

Listing 3.2: Създаване и регистриране на командите

```
#ifndef COMMAND_H
#define COMMAND_H

#include "argparse/argparse.hpp"

#include <string>


namespace cpm {

    /**
     * @brief A class representing a cpm command
     */
    class Command : public argparse::ArgumentParser {

        public:

        Command(const std::string &name);
        virtual ~Command() = default;

        /**
         * @brief Run the command
         */
        virtual void run() = 0;
    };
```

```
26 }
27
28
29 #endif  // COMMAND_H
```

Listing 3.3: Абстрактен клас за команда

```
1  #ifndef REPOSITORY_H
2  #define REPOSITORY_H
3
4  #include <filesystem>
5  #include <unordered_set>
6
7  namespace fs = std::filesystem;
8
9
10 namespace cpm {
11
12     /**
13         * @brief A class representing an object repository
14         *
15         * @tparam T the type of objects to store
16         */
17     template<typename T, typename P>
18     class Repository {
19
20         public:
21
22         /**
23             * @brief Constructor for repository
24             *
25             * @param filename the name of the file to store the repository
26             */
27         Repository(const fs::path &filename) : filename(filename) {}
28         virtual ~Repository() = default;
29
30         /**
31             * @brief Getter for filename
32             *
33             * @return The name of the repository file
34             */
35         const fs::path &get_filename() const { return this->filename; }
36
37         /**
```

```
38              * @brief Add the specified objects/s to the repository
39              *
40              * @param object the object to add
41              *
42              * @return The number of records modified in the repository
43              */
44          virtual int add(const T &object) = 0;
45
46          /**
47              * @brief Remove the specified object/s from the repository
48              *
49              * @param object the object to remove
50              *
51              * @return The number of records modified in the repository
52              */
53          virtual int remove(const T &object) = 0;
54
55          /**
56              * @brief List all of the objects in the repository
57              *
58              * @return A list of all the objects in the repository
59              */
60          virtual std::unordered_set<T, P> list() = 0;
61
62
63          protected:
64
65          fs::path filename;
66      };
67  }
68
69
70  #endif  // REPOSITORY_H
```

Listing 3.4: Абстрактен клас за хранилище

Глава 4

Ръководство за потребителя

4.1 Инсталация

Линк към GitHub хранилището: https://github.com/YassenEfremov/cpm

Изисквания:

- git
- cmake (ninja)
- C++ компилатор (g++, MSVC)

```
git clone https://github.com/YassenEfremov/cpm
cd cpm
git submodule init
mkdir build
cd build
cmake .. -G Ninja
ninja
```

Listing 4.1: Инсталация

4.2 Синтаксис

```
$ cpm <install|remove|list|update> [options]
```

Listing 4.2: Конзолен интерфейс

Заключение

:/

Библиография

[1] Tom Preston-Werner. *Semantic Versioning 2.0.0*. URL: https://semver.org/.

Съдържание