



ТЕХНОЛОГИЧНО УЧИЛИЩЕ „ЕЛЕКТРОННИ СИСТЕМИ“
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

ДИПЛОМНА РАБОТА

по професия код 481020 „Системен програмист“
специалност код 4810201 „Системно програмиране“

Тема: Система за управление на пакети

Дипломант:
Ясен Ефремов

Научен ръководител:
Валентин Върбанов

СОФИЯ
2023 г.



**ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

Дата на заданието: 22.11.2022 г.
Дата на предаване: 22.02.2023 г.

Утвърждавам:
/проф. д-р инж. П. Якимов/

ЗАДАНИЕ **за дипломна работа**

ДЪРЖАВЕН ИЗПИТ ЗА ПРИДОБИВАНЕ НА ТРЕТА СТЕПЕН НА ПРОФЕСИОНАЛНА КВАЛИФИКАЦИЯ
по професия код 481020 „Системен програмист“
специалност код 4810201 „Системно програмиране“

на ученика Ясен Ивайлов Ефремов от 12 А клас

1. Тема: Система за управление на пакети
2. Изисквания:
 - 2.1. Инсталация, премахване и изброяване на пакети
 - 2.2. Управление на зависимостите на даден пакет
 - 2.3. Локално съхранение на списък с инсталираните пакети
 - 2.4. Подаване на обратна връзка на потребителя чрез текстови съобщения
 - 2.5. Описание на пакетите чрез файлове
3. Съдържание
 - 3.1 Теоретична част
 - 3.2 Практическа част
 - 3.3 Приложение

Дипломант:

/ Ясен Ефремов /

Ръководител:

/ Валентин Върбанов /

Директор:

/ доц. д-р инж. Ст. Стефанова /

mnenie

Използвани съкращения

ANSI American National Standards Institute. [39](#)

API Application Programming Interface. [7](#), [35](#)

CI Continuous Integration. [15](#), [68](#)

CRUD Create, Read, Update, Delete. [47](#)

GNU GNU's not Unix!. [6](#), [7](#)

GraphQL Graph Query Language. [7](#)

HTTP Hyper Text Transfer Protocol. [16](#), [24](#), [25](#), [34](#)

JSON JavaScript Object Notation. [5](#), [8](#), [12](#), [13](#), [15](#), [16](#), [65](#)

REST Representational State Transfer. [7](#)

YAML YAML Ain't Markup Language. [8](#)

ООП Обектно-Ориентирано Програмиране. [13](#)

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-големите от които са удобството и сигурността. На всеки се е случвало при инсталирането на даден софтуер да се изгуби из множеството уеб страници, опитващи се да те заблудят като рекламират различни продукти или дори поставят фалшиви бутони за инсталация, водещи към непознати подозрителни сайтове. Освен че по този начин си губи времето, потребителят също така е изложен и на риск от заразяване със зловреден софтуер. Един от начините да се избегне всичко това е използването на система за управление на пакети (package manager). Тя позволява намирането, инсталирането и актуализирането на софтуер безопасно под формата на пакети, валидирани от истински хора и съхранявани в сигурни хранилища. Това има множество предимства - унифицира се начина на работа със софтуерни пакети, улеснява се работата на потребителя и се намалява риска от злонамерени действия срещу него. Популярни примери за системи за управление на пакети са winget за Windows, apt за Ubuntu Linux 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. Голем

мите 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, която изисква постоянна поддръжка. Ако трябваше програмистът ръчно да се грижи за всичко това, работата му щеше да се увеличи значително. 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 общоприета система са именно C и C++. Има опити да се направи такава, 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. Въпреки това C и C++ продължават да бъдат развивани 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 и премахват 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 файловия формат [JSON](#).

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 са по-слабо 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 пакетизирана по два различни начина:

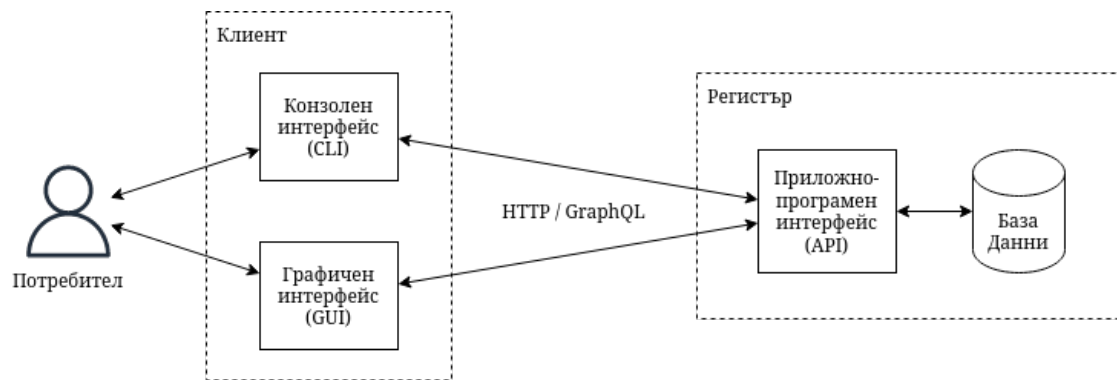
- Двоичен пакет - този вид пакети са предварително компилирани. Те съдържат или изпълнимия двоичен код на програмата, или двоичният код на библиотеката (както и нейните header файлове, ако е написана на C/C++). Името им понякога завършва с наставката “-bin”.
- Кодов пакет - този вид пакети се компилират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

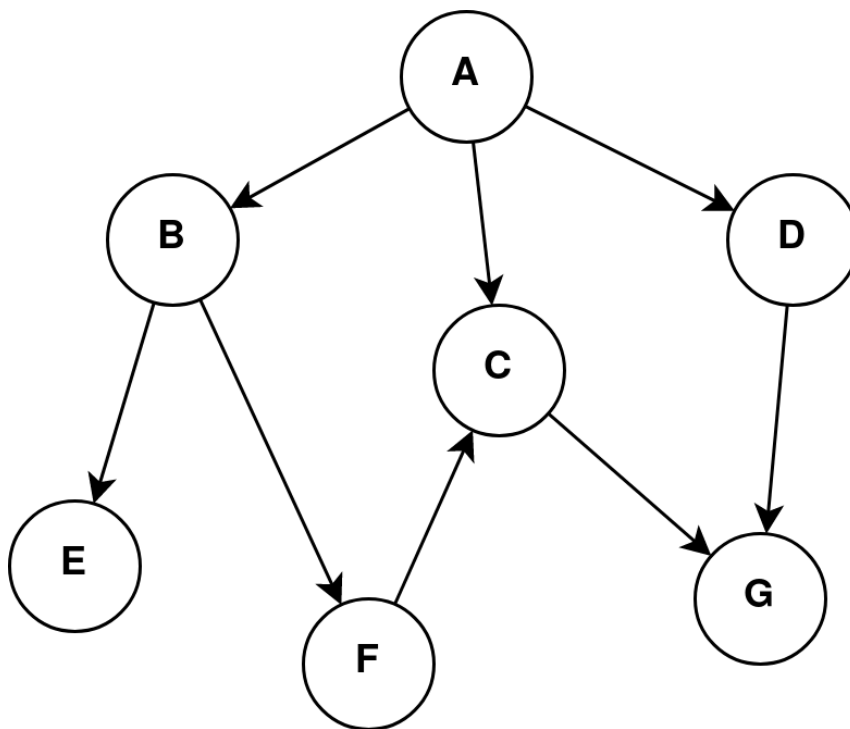
Често тези метаданни се съхраняват във файлови формати, които позволяват лесно сериализиране, извличане и обработка на данните. Такива формати са например [JSON](#) и [YAML](#).

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 (фиг. 1.2). Всеки връх в този граф представлява

даден пакет, а дъгите между върховете ни показват зависимостите им. 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.



Фигура 1.2: Примерен граф на зависимостите

На фигурата по-горе можем да видим два вида зависимости:

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

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 (dependency flattening). 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, лиценза, зависимости 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, предоставяща конзолен потребителски интерфейс и собствен регистър за теглене на пакети, наречен conan center, където са качени повечето известни библиотеки за C и C++. Системата зависи от езика Python и съответно може да бъде инсталирана чрез pip.
- vcpkg - система, разработена от Microsoft, написана на C++. vcpkg е интегриран в средите за разработка Visual Studio и Visual Studio Code. Предоставя конзолен потребителски интерфейс и регистър за теглене на пакети. Също позволява лесно пакетиране на вече съществуващи софтуерни библиотеки и интегрирането им с build системата CMake. За платформи различни от Windows инсталационния процес е малко нестандарт-

тен като изисква клониране на GitHub хранилището на проекта и изпълнението на няколко скрипта, които директно теглят от интернет изпълнимия файл на програмата.

За разлика от тези два съществуващи проекта, текущата дипломна работа се стреми да разработи система за управление на пакети, която няма никакви външни зависимости и е лесно достъпна както за Windows, така и за Linux.

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 на пакети от GitHub хранилище
 - Премахване на пакети
 - Изброяване на инсталираните пакети
 - Създаване на пакети и др.
- Управление на зависимостите на даден пакет
 - Рекурсивна инсталация на всички зависимости на даден пакет
 - Проверка на съвместимостта на версиите на пакетите
- Съхранение на списък с инсталираните пакети
 - Списък с локално инсталираните пакети се съхраняват в **JSON** файл в текущата директория
 - Списък с 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++.

C++ е създаден през 80-те години на миналия век от датския програмист Бярне Струоструп. Първоначалната цел на езика била да наподобява вече популярния език C, като добави поддръжка за обектно-ориентирано програмиране (ООП). От там идва и ранното му наименование - C с класове. Впоследствие обаче езикът еволюира и започва да поддържа и други програмни парадигми, една от които е тази на функционалното програмиране.

Основно езикът се използва за разработване на системи, при които е нужна висока производителност и които са ограничени от страна на памет поради хардуера, върху който се изпълняват. Въпреки това C++ разполага с богата стандартна библиотека, предоставяща абстракции от високо ниво, с помощта на които могат лесно да се разработват всякакъв вид програми. От тук обаче идва и един от недостатъците на езика, а именно неговата сложност. С годините към C++ биват добавени все повече и повече функционалности, които постепенно го правят все по-комплексен и труднодостъпен за по-неопитните програмисти. Понеже се счита, че езикът е с общо предназначение, има много спорове за какво и как трябва да бъде използван. И все пак C++ остава един от най-популярните езици до ден днешен, като стои в основата на много от критично важните системи, използвани ежедневно.

За разработката на текущия проект бе избран стандарта C++17 (формално ISO/IEC 14882:2017) поради няколко причини:

- C++17 е последната версия на езика, която е изцяло поддържана от главните компилатори - GCC и MSVC.
- от C++17 нататък стандартната библиотека предоставя удобни абстракции, някои от които са:
 - Интерфейс за работа с файловата система (заглавния файл `<filesystem>`)
 - Поддръжка за типове с незадължителна стойност (заглавния файл `<optional>`)
 - Поддръжка за олекотени низове, достъпни само за четене (заглавния файл `<string_view>`)
 - Поддръжка за структурирани свързвания (structured bindings)
- Съществуват множество свободно достъпни библиотеки, използващи модерните подобрения на езика

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 по-лесно компилирането/интерпретирането, тестването и пакетирането на софтуер.

В случая на C++, различните операционни системи предоставят и собствени build системи. Windows работи с Visual Studio проекти, Linux работи с Makefile и MacOS работи с Xcode проекти. За да не е нужно да поддържаме три различни системи за компилирането на един и същ проект, можем да използваме така нареченият генератор на build системи (build system generator). Най-разпространеният такъв е CMake (акроним за Cross Platform Make). Той е поддържан от всички главни операционни системи, като автоматично генерира съответните build файлове, които трябва да бъдат изпълнени, за да се компилира проекта. CMake също предоставя допълнителни инструменти за автоматизирано тестване и пакетиране на софтуера (CTest и CPack).

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.

По време на разработката на текущия проект бе използвана стратегия на разклоненията (`branching strategy`), при която всяко изискване по проекта има свое отделно разклонение:

- `cli` - разработка на конзолния потребителски интерфейс
- `commands` - разработка на командите
- `logger` - разработка на диагностиката
- `db` - работа с `sqlite` базата данни, съхраняваща списъка с глобално инсталираните пакети
- `script` - работа с `JSON` файла, съхраняващ списъка с локално инсталираните пакети
- `dep-man` - разработка на алгоритъма за управление на зависимостите
- `doc` - разработка на документацията

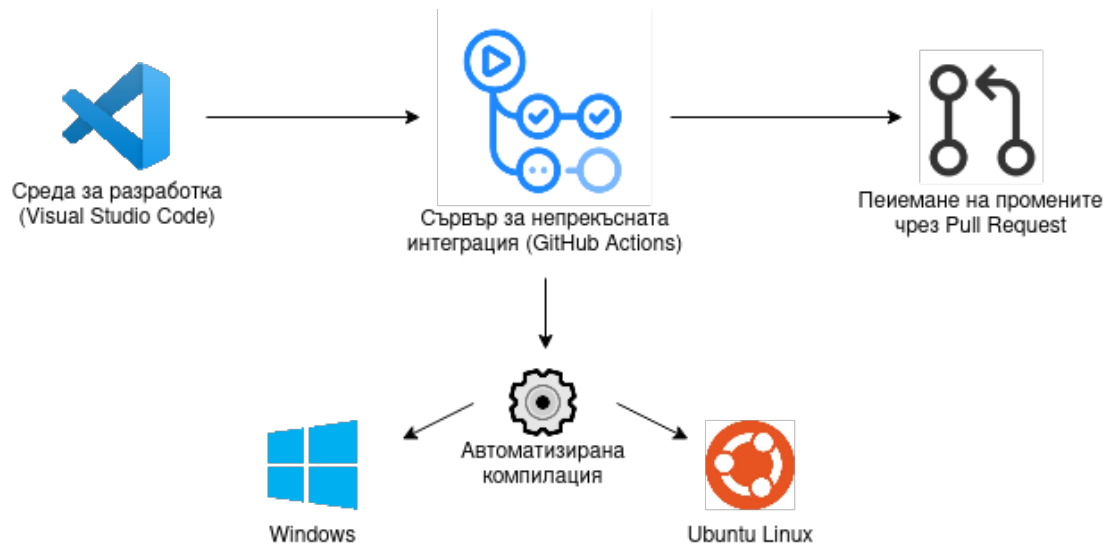
2.2.4 Среда за разработка

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 Непрекъснатата интеграция (CI)

Непрекъснатата интеграция (Continuous Integration) (фиг. 2.1) е практика при разработката на софтуер при която работата на много разработчици се обединява на едно място, след което се задейства ав-

томатизирана компилация/интерпретация и тестване на софтуерния продукт.



Фигура 2.1: Непрекъсната интеграция

По време на разработката на текущия проект бяха използвани предоставените от платформата GitHub средства за автоматизирана компилация на кода (GitHub Actions).

2.2.6 Използвани библиотеки

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

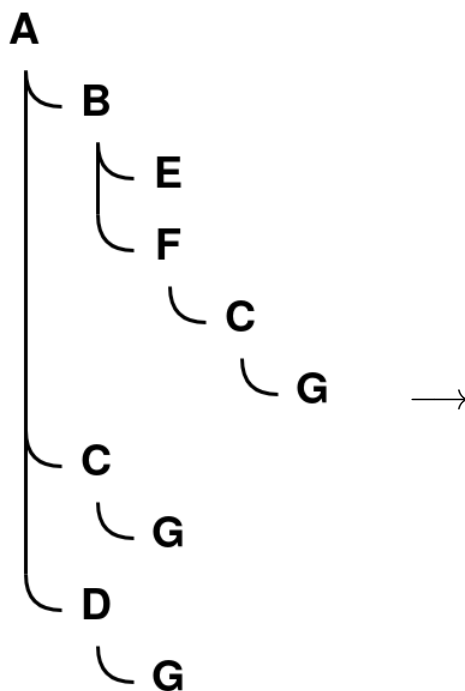
- libcpr/cpr (C++ Requests) - обвивка (wrapper) на библиотеката curl, позволяваща изпълнение на [HTTP](#) заявки [2]
- kuba-/zip - библиотека за компресиране и архивиране
- sqlite - малка релационна база данни, съх [3]
- gabime/spdlog - библиотека, подаваща обратна връзка на потребителя и позволяваща пазенето на диагностични съобщения
- nlohmann/json - библиотека за работа с [JSON](#) файлове [4]
- p-ranav/argparse - библиотека, позволяваща лесна обработка на подадените към програмата параметри и тяхното използване

2.2.7 Алгоритъм за управление на зависимостите

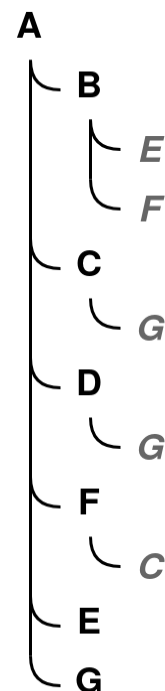
За да се постигне ефективно и бързо инсталиране на даден пакет и неговите зависимости, се разчита, че всеки пакет съдържа в хранилището си така наречения lockfile, който представлява графа на неговите зависимости. По този начин се избягва изпълнението на множество заявки, които иначе биха били нужни за построяването на този граф. Инсталирането на даден пакет и неговите зависимости се състои от следните стъпки:

1. Сдобиване с lockfile на пакета и неговите зависимости
2. Инсталиране на пакета и неговите зависимости паралелно (в отделни нишки)
3. Конструирание на графа на зависимостите във файловата система, използвайки символични връзки (symlinks).

Последната стъпка от изброените по-горе е така нареченият метод за изравняване на зависимостите (dependency flattening [5]). Методът е илюстриран по-долу чрез двете дървовидни структури, които са еквивалентни на графа, показан по-рано (фиг. 1.2).



Фигура 2.2:
Дълбоко вложени зависимости



Фигура 2.3:
Изравнени зависимости

От лявата страна (фиг. 2.2) е показано дърво на зависимостите, при което всеки пакет съдържа в себе си всички свои зависимости. Това е най-простият начин, по който могат да бъдат съхранени пакетите. Този подход обаче има недостатъка, че с добавянето на нови зависимости към пакетите, които все още нямат такива, се увеличава и нивото на влагане на пакетите.

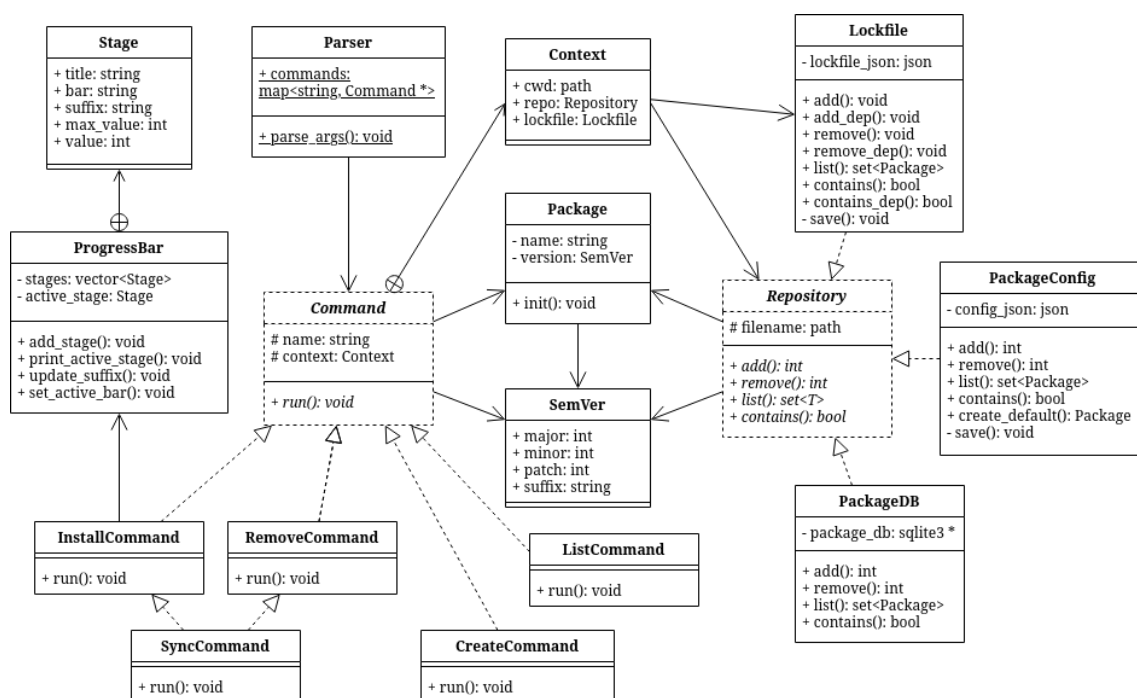
От дясната страна (фиг. 2.3) е показано дърво, при което всички зависимости са съхранени на едно ниво. Така вместо всеки пакет да съдържа в себе си своите зависимости, той съдържа символични връзки (обозначени със сиво на фигурата), сочещи към предишната директория, в която зависимостите са инсталирани. По този начин се решава проблема с влагането на пакети като нивата на влагане се намаляват до максимум две.

Глава 3

Реализация на проекта

3.1 Структура на проекта

Всички части на програмата се намират в именното пространство (namespace) `срт`. В главната директория на проекта се намира и основният файл `main.cpp`, от който започва изпълнението на програмата.



Фигура 3.1: UML Диаграма на структурата на програмата

3.2 Файлове с общо предназначение

Файловете `paths.cpp` и `util.cpp` са с общо предназначение.

Файлът `paths.cpp` предоставя именното пространство `cpm::paths`, в което са дефинирани константни имена, файлови пътища и GitHub URL пътища.

Файлът `util.cpp` предоставя именното пространство `cpm::util`, в което са дефинирани функциите `base64_decode()` и `split_string()`. Първата функция декодира base64 кодиран текст, а втората разделя символен низ в зависимост от подадения разделител.

3.3 Диагностика (Logging)

Програмата използва библиотеката `spdlog` за подаване на обратна връзка към потребителя и водене на диагностични записки. Във файла `logger.hpp` (списък 3.1) са декларирани обвиващите класове `CLILogger` и `FileLogger`. Те обаче не са предназначени за директно ползване. Вместо това заглавният файл предоставя удобни и кратки директиви, предназначени за извеждане на съобщения в конзолата и в диагностичния файл. Директивите поддържат няколко нива - грешка, предупреждение и информативно съобщение.

Максималният размер на диагностичния файл е 5MB. Ако бъде превишен се създава нов файл, като към името на стария се добавя поредна цифра (например `cpm.1log`). Това се повтаря докато файловете не станат три на брой, след което текущият файл се изчиства и записването продължава. Това са така наречените въртящи се логове (rotating logs).

```
1  #define CPM_LOG_INFO(...) cpm::FileLogger::get_file_logger()->info(__VA_ARGS__)
2  #define CPM_LOG_ERR(...)  cpm::FileLogger::get_file_logger()->error(__VA_ARGS__)
3
4  #define CPM_INFO(...)     cpm::CLILogger::get_stdout_logger()->info(__VA_ARGS__)
5  #define CPM_ERR(...)      cpm::CLILogger::get_stderr_logger()->error(__VA_ARGS__)
6  #define CPM_WARN(...)     cpm::CLILogger::get_stderr_logger()->warn(__VA_ARGS__)
7
8
9  namespace cpm {
10
11  /**
12   * @brief The cpm console logger
13   */
14  class CLILogger {
15
16  public:
```

```

17
18     /**
19      * @brief Initialize the logger
20      */
21     static void init();
22
23     static std::shared_ptr<spdlog::logger> get_stdout_logger();
24     static std::shared_ptr<spdlog::logger> get_stderr_logger();
25 };
26
27 /**
28  * @brief The cpm rotating file logger
29  */
30 class FileLogger {
31
32     public:
33
34     /**
35      * @brief Initialize the logger
36      */
37     static void init();
38
39     static std::shared_ptr<spdlog::logger> get_file_logger();
40 };
41
42 } // namespace cpm

```

Списък 3.1: Класове и директиви за диагностика и обратна връзка

3.4 Пакети

За описание на пакетите се използва класът **Package** (списък 3.2). Той съдържа всички основни характеристики на пакета - име, версия, URL, описание, автор и лиценз. Класът предоставя метода `init()`, който инициализира пакета като автоматично намира от къде да го изтегли, проверява неговата версия или намира последната такава от GitHub.

При съхранението на списъци от пакети в паметта бе използвана стандартната структура `std::unordered_set`, имплементираща математическо множество от обекти. Структурата рефлектира свойството на пакетите да са уникални като не позволява повторението на два еднакви пакета. За да могат да бъдат съхранени в нея, пакетите предоставят специална хешираща функция и оператор за сравнение.


```

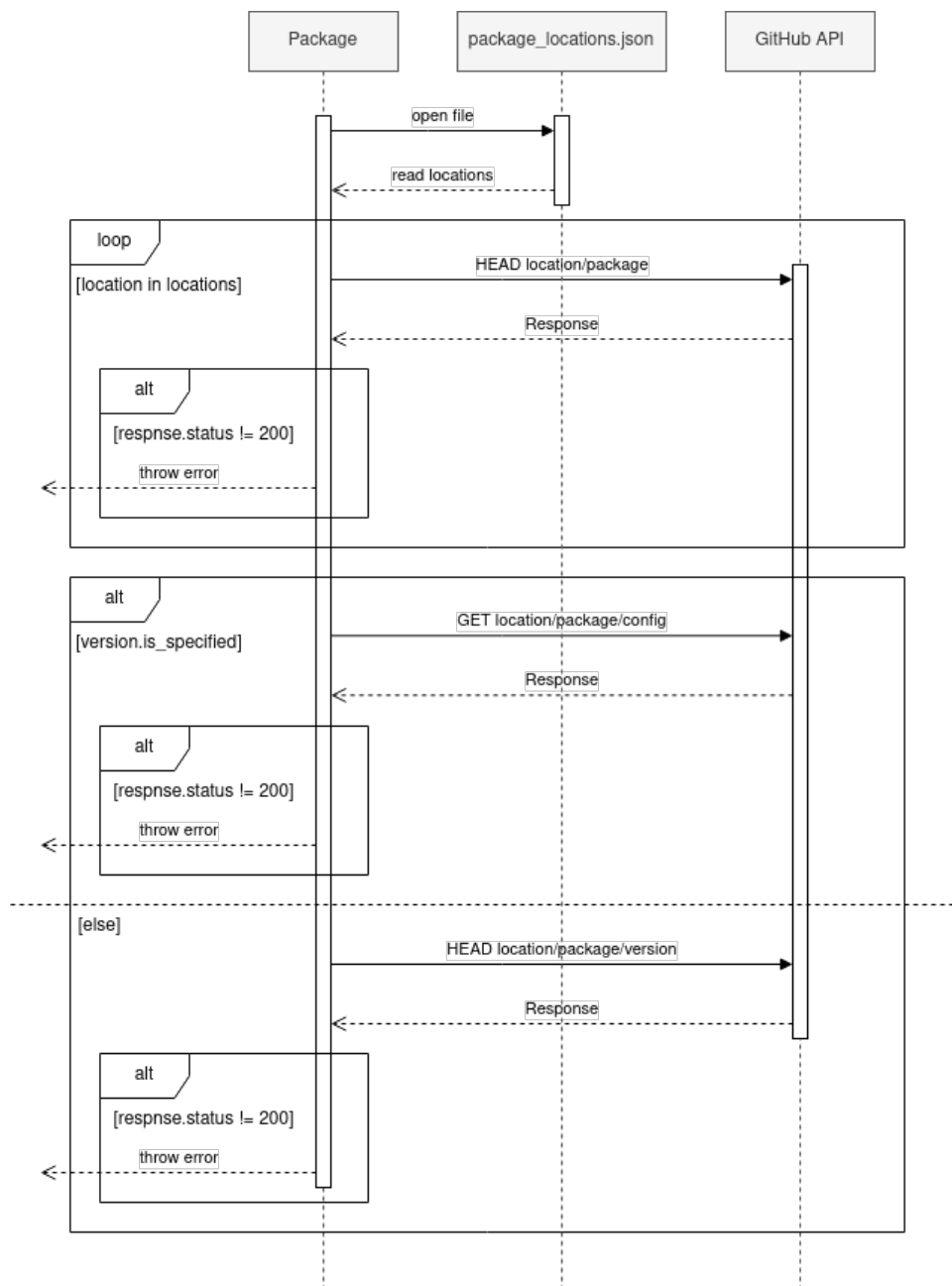
1  /**
2   * @brief A class representing a cpm package
3   */
4  class Package {
5
6      public:
7
8      struct Hash {
9          std::size_t operator()(const Package &package) const
10             noexcept;
11     };
12
13     /**
14      * @brief Constructor for package
15      *
16      * @param name the name of the package
17      * @param version the package version
18      */
19     Package(const std::string &name, const SemVer &version = SemVer());
20
21     ... // getters omitted
22
23     /**
24      * @brief Initialize the package location and resolve it's
25      *         version
26      */
27     void init();
28
29     friend bool operator==(const Package &lhs, const Package &rhs);
30
31
32     private:
33
34     std::string name;
35     SemVer version;
36     std::string location;
37     std::string url;
38     std::string description;
39     std::string author;
40     std::string license;
41 };

```

Списък 3.2: Клас за пакет

3.4.1 Инициализация на пакет

Някои команди изискват пакетите, с които работят, да бъдат инициализирани преди да бъдат използвани. Тази инициализация се изпълнява от функцията `init()` (фиг. 3.2) и се състои от следните стъпки:



Фигура 3.2: Диаграма на функцията `init()`

Намиране на източник за теглене на пакета

За да не е нужно при теглене на пакет потребителят да трябва да подава и източника му (GitHub профил или организация), функцията `init()` автоматично намира от къде да изтегли съответния пакет (списък [3.3](#)).

Това се постига чрез файла `package_locations.json`. Той се намира в глобалната конфигурационна директория на програмата и съдържа масив от имена на източници. Функцията `init()` мина-

ва през всеки записан във файла източник и прави **HTTP HEAD** заявка към него с цел да провери дали съществува. При първата успешна такава заявка източникът се запазва в полето `location` на пакета и функцията продължава с проверки на версията.

```
1 CPM_LOG_INFO("Looking for an available GitHub profile ...");
2 if (!fs::exists(paths::global_dir / paths::package_locations)) {
3     throw std::runtime_error(fmt::format(
4         "package locations file not found!\n"
5         "Make sure you've installed cpm correctly!"
6     ));
7 }
8 std::ifstream package_locations_file(paths::global_dir /
9     paths::package_locations);
10 auto package_locations_json = json::parse(package_locations_file);
11 auto package_locations = package_locations_json["package_locations"]
12     .get<std::vector<std::string>>();
13 this->location = "";
14 for (const auto &package_location : package_locations) {
15     CPM_LOG_INFO("Trying {}", package_location);
16     cpr::Response response = cpr::Head(
17         cpr::Url{(paths::api_url / package_location / this->name).string()}
18     );
19     if (response.status_code == cpr::status::HTTP_OK) {
20         CPM_LOG_INFO("Success (status {}) using {}",
21             response.status_code, package_location);
22         this->location = package_location;
23         break;
24     } else {
25         CPM_LOG_INFO("Failed (status {}) {}",
26             response.status_code, package_location);
27     }
28 }
29 }
30 if (this->location.empty()) {
31     throw std::invalid_argument(fmt::format(
32         "{}: package not found!", this->name
33     ));
34 }
```

Списък 3.3: Намиране на източник за теглене на пакета

Сдобиване с последна версия на пакета

Ако при инсталиране на даден пакет не е подадена неговата версия, функцията `init()` автоматично се сдобива с последната такава (списък 3.4). Това става чрез **HTTP GET** заявка към `cpm_pack.json` файла на пакета. От този файл се взима стойността на полето, задаващо последната версия.

```

1  if (!this->version.is_specified()) {
2      CPM_LOG_INFO("Getting latest version for package {} ...", this->name);
3      cpr::Response response = cpr::Get(
4          cpr::Url{(paths::api_url / location / this->name /
5                  paths::gh_content / paths::package_config).string()}
6      );
7      json package_config_json;
8      if (response.status_code == cpr::status::HTTP_OK) {
9          json response_json = json::parse(response.text);
10         std::string package_config_str = util::base64_decode(
11             response_json["content"].get<std::string>()
12         );
13         package_config_json = json::parse(package_config_str);
14         this->version = SemVer(package_config_json["version"]
15                                .get<std::string>());
16     }
17     else {
18         throw std::runtime_error(fmt::format(
19             "{}: package doesn't contain a {} file!",
20             this->name, paths::package_config.string()
21         ));
22     }
23 }

```

Списък 3.4: Сдобиване с последната версия на пакета

Валидиране на подадената версия на пакета

Ако при инсталиране на даден пакет е изрично подадена неговата версия чрез синтаксиса <име на пакет>@<версия>, единственото което функцията `init()` прави е да провери, че тази версия наистина съществува за дадения пакет (списък 3.5). Това става чрез **HTTP HEAD** заявка към конкретната версия на пакета.

```

1  CPM_LOG_INFO("Checking version {} for package {}",
2              this->version.string(), this->name);
3  cpr::Response response = cpr::Head(
4      cpr::Url{(paths::api_url / this->location / this->name /
5                paths::gh_zip / this->version.string()).string()}
6  );
7  if (response.status_code != cpr::status::HTTP_OK) {
8      throw std::invalid_argument(fmt::format(
9          "{}: version {} not found!", this->name, this->version.string()
10      ));
11 }

```

Списък 3.5: Валидиране на подадената версия на пакета

3.5 Версии

За описание на версиите на пакетите бе използван класът **SemVer** (съкратено от Semantic Version)(списък 3.6). Този клас имплементира частично стандарта за семантични версии [1], като включва основните му характеристики, а именно трите числени части на версията - главна, второстепенна и кръпка. Класът не предоставя поддръжка за идентификатори за предварително издание (alpha, beta, rc, и др.) поради факта, че build системата CMake, с която се интегрира текущият проект, не следва стандарта за семантични версии стриктно, а поддържа единствено първите три числени части на версията¹.

За по-удобно използване, класът предоставя имплементация на всички оператори за сравнение.

```
1  /**
2   * @brief A class representing a generic semantic version
3   */
4  class SemVer {
5
6      public:
7
8      /**
9       * @brief Default constructor for generic semantic version
10      */
11     SemVer();
12
13     /**
14      * @brief Constructor for generic semantic version
15      *
16      * @param major the major version
17      * @param minor the minor version
18      * @param patch the patch version
19      */
20     SemVer(int major, int minor, int patch);
21
22     /**
23      * @brief Constructor for generic semantic version
24      *
25      * @param version_str a semantic version as a string
26      */
27     SemVer(const std::string &version_str);
28
29     ... // getters omitted
30
31     /**
32      * @brief Check if the version is different from the default one
```

¹Има отворена дискусия в официалното хранилище на CMake за добавянето на поддръжка за идентификатори за предварително издание. Виж [6]

```

33     *
34     * @return true if the version is not the default one, false otherwise
35     */
36     bool is_specified();
37
38     friend bool operator<(const SemVer &lhs, const SemVer &rhs);
39     friend bool operator>(const SemVer &lhs, const SemVer &rhs);
40     friend bool operator<=(const SemVer &lhs, const SemVer &rhs);
41     friend bool operator>=(const SemVer &lhs, const SemVer &rhs);
42
43     friend bool operator==(const SemVer &lhs, const SemVer &rhs);
44     friend bool operator!=(const SemVer &lhs, const SemVer &rhs);
45
46
47     private:
48
49     int major;
50     int minor;
51     int patch;
52 };

```

Списък 3.6: Клас за версия

3.6 Команди

За реализиране на командите бе използван така нареченият Команден Шаблон (Command Pattern [7]). При него има базов абстрактен клас, предоставящ функция за изпълнение на командата, като всеки следващ клас, наследяващ базовия, предоставя своя имплементация на командата за изпълнение.

В основата на всяка команда стои абстрактният клас **Command** (списък 3.7). Той предоставя абстрактния метод **run**, който всяка команда трябва да имплементира. Този метод се извиква в основния файл на програмата за съответната подадена команда.

Класът **Command** наследява класът **ArgumentParser**, предоставен от библиотеката **argparse**. Този клас има функции, позволяващи задаването на много различни опции на командата - описание, параметри, задължителност, стойност по подразбиране, незадължителни опции и др.

Някои команди могат да се изпълняват в различен контекст. Такива са например командите за инсталиране, премахване и изброяване на пакети. Те могат да работят както на локално, така и на глобално ниво. Вътрешната структура **Context** се използва за описание на контекста, в който се изпълнява дадена команда. Парамет-

рите на контекста са текущата работна директория, структурата за съхранение на списък с инсталираните пакети и текущият lockfile. За да не е нужно всяка команда да проверява в какъв контекст се изпълнява и спрямо това да сменя съответното хранилище, контекста съдържа умни указатели (smart pointers) към базовите класове на хранилищата. Това позволява използването на всички техни методи, без значение кое конкретно хранилище се използва.

Структурата съдържа указател към абстрактния клас `Repository`, което позволява извикването на всички негови виртуални методи, без значение от това коя тяхна имплементация точно се използва.

```
1  /**
2   * @brief A class representing a cpm command. All other cpm commands should
3   *       derive from this class
4   */
5  class Command : public argparse::ArgumentParser {
6
7      public:
8
9          /**
10           * @brief Constructor for command
11           *
12           * @param name the name of the command
13           */
14          Command(const std::string &name);
15          virtual ~Command() = default;
16
17          /**
18           * @brief Execute the command
19           */
20          virtual void run() = 0;
21
22          friend class Parser;
23
24
25      protected:
26
27          /**
28           * @brief A struct representing the context a cpm command runs in
29           */
30          struct Context {
31              fs::path cwd;
32              std::shared_ptr<Repository<Package, Package::Hash>> repo;
33              std::shared_ptr<Lockfile> lockfile;
34          };
35
36          Command();
37          Context context;
38  };
```

Списък 3.7: Абстрактен клас за команда

3.6.1 Регистриране на команди

Командите се регистрират, използвайки функциите, предоставени от библиотеката `argparse` (списък 3.8). Всички команди се съхраняват в структура `std::map`, използваща за ключ името на командата, а за стойност указател към основния абстрактен клас `Command`. Това позволява извикването на виртуалния метод `run`, без значение от това коя негова имплементация точно се използва. Всяка команда също се регистрира в главната команда `cpm`.

```
1  argparse::ArgumentParser parser("cpm", CPM_VERSION);
2
3
4  static InstallCommand install_command("install");
5  install_command.add_description("Install the specified package/s");
6  install_command.add_argument("packages")
7      .help("Packages to install")
8      .required()
9      .nargs(argparse::nargs_pattern::at_least_one);
10 install_command.add_argument("-g", "--global")
11     .help("install package/s globally")
12     .default_value(false)
13     .implicit_value(true);
14
15 ... // code omitted
16
17 parser.add_subparser(install_command);
18 ... // code omitted
19 commands.insert({"install", &install_command});
20 ... // code omitted
```

Списък 3.8: Създаване и регистриране на команда

3.6.2 Обработка на подадените параметри

След като всички команди са регистрирани се преминава към обработката на подадените параметри към програмата (списък 3.9). Променливите `argc` и `argv` съдържат съответно броя на подадените аргументи и техните стойности под формата на символни низове.

Първо се проверява дали към програмата са подадени достатъчно параметри. Минимумът в този случай е два параметъра - името на програмата (подаден по подразбиране) и името на командата, която трябва да се изпълни. Ако броят на параметрите е по-малък от два се хвърля грешка, която се обработва от `main()` функцията и води до извеждането на съобщение, инструктиращо потребителя как да използва програмата.

Следващата стъпка е същинската обработка на параметрите. Това става лесно с помощта на функцията `parse_args()`, предоставена от библиотеката `argparse`. На тази функция също се подават променливите `argc` и `argv`. Ако има проблем в синтаксиса при извикване на програмата отново се хвърля грешка, която се обработва от `main()` функцията и води до извеждането на съобщение, информиращо потребителя какъв е проблема.

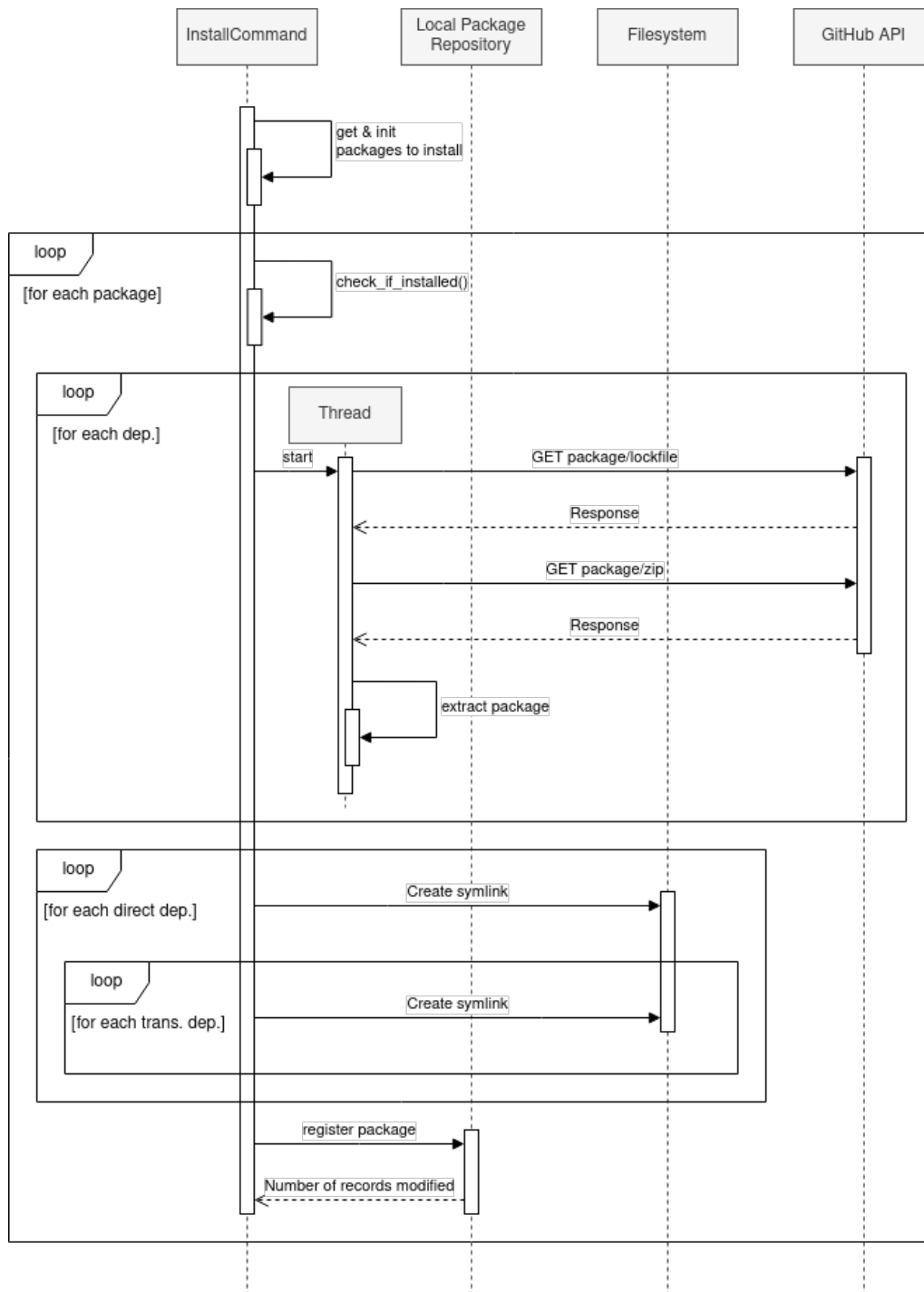
Последната стъпка е определянето на контекста, в който се изпълнява програмата. По подразбиране програмата се изпълнява на локално ниво. Това може да се промени с подаването на флага `--global` (или по-краткото `-g`). В зависимост от това се променят стойностите на вътрешната структура `context`.

```
1  if (argc < 2) {
2      throw std::runtime_error(parser.help().str());
3  }
4  parser.parse_args(argc, argv);
5
6
7  try {
8      if (commands[argv[1]]->is_used("--global")) {
9          commands[argv[1]]->context = Command::Context{
10              paths::global_dir,
11              std::make_shared<PackageDB>(
12                  paths::global_dir / paths::package_db
13              ),
14              std::make_shared<Lockfile>(
15                  paths::global_dir / paths::lockfile
16              )
17          };
18      } else {
19          throw std::runtime_error("");
20      }
21  }
22
23  } catch(const std::exception &e) {
24      commands[argv[1]]->context = Command::Context{
25          fs::current_path(),
26          std::make_shared<PackageConfig>(
27              fs::current_path() / paths::package_config
28          ),
29          std::make_shared<Lockfile>(
30              fs::current_path() / paths::lockfile
31          )
32      };
33  }
```

Списък 3.9: Обработка на подадените параметри

3.6.3 Команда за инсталиране

Класът `InstallCommand` отговаря за инсталирането на пакети. Начинът му на работа е илюстриран на по-долната диаграма (фиг. 3.3). Той се състои от следните стъпки:



Фигура 3.3: Диаграма на процеса на инсталиране на пакет

Сдобиване с пакетите за инсталиране

Поради факта, че всяка команда наследява от класа `ArgumentParser`, параметрите, подадени на програмата, могат да бъдат достъпени директно от класа `InstallCommand`. Те обаче са под формата на списък от символни низове, представляващи имената и версиите на пакетите. Този списък трябва да бъде превърнат в множество от пакети. Процесът за това е описан по-долу (списък [3.10](#)).

```
1  CPM_LOG_INFO("==== Starting install command ====");
2
3  auto packages_str = this->get<std::vector<std::string>>("packages");
4  CPM_LOG_INFO("args: {}", [&]() {
5      std::string args = "[";
6      for (const auto &arg : packages_str) {
7          args += arg + ", ";
8      }
9      return args + "]";
10 }());
11 std::unordered_set<Package, Package::Hash> packages;
12 for (const auto &package_str : packages_str) {
13     auto tokens = util::split_string(package_str, "@");
14     if (tokens.size() == 2) {
15         CPM_LOG_INFO("Checking package {}@{} ...", tokens[0], tokens[1]);
16         CPM_INFO("Checking package {}@{} ...", tokens[0], tokens[1]);
17         Package new_package(tokens[0], SemVer(tokens[1]));
18         try {
19             new_package.init();
20         } catch(const std::exception &e) {
21             CPM_INFO(" failed!\n");
22             throw std::invalid_argument(e.what());
23         }
24         CPM_LOG_INFO(
25             "version {} for package {} is valid",
26             new_package.get_version().string(), new_package.get_name()
27         );
28         CPM_INFO(" found.\n");
29         packages.insert(new_package);
30
31     } else if (tokens.size() == 1) {
32         CPM_LOG_INFO("Resolving version for package {} ...", tokens[0]);
33         CPM_INFO("Resolving version for package {} ...", tokens[0]);
34         Package new_package(tokens[0]);
35         try {
36             new_package.init();
37         } catch(const std::exception &e) {
38             CPM_INFO(" failed!\n");
39             throw std::invalid_argument(e.what());
40         }
41         CPM_LOG_INFO(
42             "found version {} (latest) for package {}",
43             new_package.get_version().string(), new_package.get_name()
44         );
45     }
```

```

45     CPM_INFO(" found latest: " YELLOW_FG("{}") "\n",
46             new_package.get_version().string());
47     packages.insert(new_package);
48
49 } else {
50     throw std::invalid_argument(fmt::format(
51         "{}: invalid package format!", package_str
52     ));
53 }
54 }

```

Списък 3.10: Сдобиване с пакетите за инсталиране

Всеки низ от списъка може да съдържа или само името на пакета, или да бъде от вида **<име на пакет>@<версия>**. Тези два случая изискват различен начин на обработка. Това е отразено на по-горния фрагмент от код, където се забелязват два по-големи **if** блока. Първият от тях създава пакет, за който са подадени както името, така и версията, след което пакетът се инициализира. Ако версията не е подадена на пакета се задава такава по подразбиране. Пакетът отново се инициализира.

По време на описания по-горе процес се водят съответните диагностични записки, които отразяват разликата в начина на създаване на пакета. Ако при въвеждане на пакета е използван различен формат от по-рано споменатите се хвърля грешка, която се обработва в **main** функцията и уведомява потребителя.

Проверка

Първото нещо, което командата за инсталиране прави, е да провери дали пакетът не е вече инсталиран (списък 3.11). Това става чрез функцията **check_if_installed()**. Ако пакет със същото име е намерен се изпраща съобщение към потребителя, предупреждаващо, че даденият пакет вече е инсталиран.

```

1  CPM_LOG_INFO("Checking if package {} is already installed ...",
2              package.get_name());
3  if (this->check_if_installed(package)) {
4      throw std::invalid_argument(fmt::format(
5          "{}: package already installed!", package.get_name()
6      ));
7  }
8
9  ... // code omitted
10
11 bool InstallCommand::check_if_installed(const Package &package) {
12     bool specified = this->context.repo->contains(package);

```

```

13     bool downloaded = fs::exists(this->context.cwd / paths::packages_dir /
14                                   package.get_name() / "");
15
16     return specified && downloaded;
17 }

```

Списък 3.11: Функция за проверка на пакет

Теглене

Функцията `download_package()` тегли подадения пакет под формата на компресиран zip архив (списък 3.12). Това става чрез HTTP GET заявка към предоставената от GitHub точка за достъп. Параметърът `download_progress` е друга функция (callback), която се извиква всеки път, когато има напредък с тегленето на пакета.

```

1  cpr::Response InstallCommand::download_package(
2      const Package &package, const std::string location,
3      std::function<bool(
4          cpr::cpr_off_t downloadTotal, cpr::cpr_off_t downloadNow,
5          cpr::cpr_off_t uploadTotal, cpr::cpr_off_t uploadNow,
6          std::intptr_t userdata
7      )> download_progress
8  ) {
9      CPM_LOG_INFO(
10         "GET {}/{}/{}/{}/{}",
11         paths::api_url.string(), location, paths::gh_zip,
12         package.get_name(), package.get_version().string()
13     );
14     cpr::Response response = cpr::Get(
15         cpr::Url{(paths::api_url / location / package.get_name() /
16                  paths::gh_zip / package.get_version().string()).string()},
17         cpr::ProgressCallback(download_progress)
18     );
19     CPM_LOG_INFO("Response status: {}", response.status_code);
20     if (response.status_code != cpr::status::HTTP_OK) {
21         throw std::runtime_error(fmt::format(
22             "{}: version {} not found!",
23             package.get_name(), package.get_version().string()
24         ));
25     }
26
27     return response;
28 }

```

Списък 3.12: Функция за теглене на пакет

Декомпресиране

Декомпресирането на пакета се извършва директно в паметта от функцията `extract_package()` (списък 3.13). Тя обхожда всички файлове в zip архива като декомпресира и сменя името на всеки от тях. Това е нужно поради ограничение на GitHub API [8], който сформира името на изтегленото хранилище като добавя наставка, отразяваща текущото разклонение. Не е желателно крайната папка да бъде запазена с такова име, защото то усложнява работата на програмата.

```
1 void InstallCommand::extract_package(  
2     const std::string &stream, const fs::path &output_dir,  
3     std::function<bool(int currentEntry, int totalEntries)> on_extract  
4 ) {  
5     struct zip_t *zip = zip_stream_open(stream.c_str(), stream.size(), 0, 'r');  
6  
7     std::size_t total_entries = zip_entries_total(zip);  
8     CPM_LOG_INFO("Total entries to extract: {}", total_entries);  
9     for (int i = 0; i < total_entries; i++) {  
10         zip_entry_openbyindex(zip, i);  
11  
12         std::string entry_name = zip_entry_name(zip);  
13         CPM_LOG_INFO("extracting {} ...", entry_name);  
14         std::size_t first_slash = entry_name.find_first_of('/');  
15         if (zip_entry_isdir(zip)) {  
16             fs::create_directories(  
17                 output_dir / entry_name.substr(first_slash + 1)  
18             );  
19         }  
20         zip_entry_fread(zip,  
21             (output_dir / entry_name.substr(first_slash + 1)).string().c_str()  
22         );  
23         on_extract(i + 1, total_entries);  
24  
25         zip_entry_close(zip);  
26     }  
27  
28     zip_stream_close(zip);  
29 }
```

Списък 3.13: Функция за декомпресиране на пакет

Създаване на символичните връзки

Процесът на свързване на пакетите и техните зависимости е илюстриран на по-долния фрагмент от код (списък 3.14).


```

1  if (packages_to_install.size() < 2) return;
2  for (const auto &[name, content] : package_lockfile_json["dependencies"]
3      .items()) {
4      this->context.lockfile->add_dep(
5          package, Package(name, content["version"].get<std::string>())
6      );
7      CPM_LOG_INFO("symlinking direct dependency: {} ...", name);
8      fs::create_directory(output_dir / package.get_name() / paths::packages_dir);
9      fs::create_directory_symlink(
10         output_dir / name,
11         output_dir / package.get_name() / paths::packages_dir / name
12     );
13     CPM_LOG_INFO("symlink created");
14
15     this->context.lockfile->add(
16         Package(name, content["version"].get<std::string>())
17     );
18     if (content.contains("dependencies")) {
19         for (const auto &[dep_name, dep_version] : content["dependencies"]
20             .items()) {
21             this->context.lockfile->add_dep(
22                 Package(name, content["version"].get<std::string>()),
23                 Package(dep_name, dep_version.get<std::string>())
24             );
25             CPM_LOG_INFO("symlinking transitive dependency: {} ...", dep_name);
26             fs::create_directory(output_dir / name / paths::packages_dir);
27             fs::create_directory_symlink(
28                 output_dir / dep_name,
29                 output_dir / name / paths::packages_dir / dep_name
30             );
31             CPM_LOG_INFO("symlink created");
32         }
33     }
34 }

```

Списък 3.14: Създаване на символичните връзки

Преди да започне свързването на пакетите се проверява дали текущият пакет има зависимости. Ако няма такива тази стъпка изцяло се пропуска.

Самото свързване се извършва с помощта на два вложени `for` цикъла. Първият от тях обхожда директните зависимости на пакета, който се инсталира в момента, добавя ги към `lockfile` на текущия пакет и създава символични връзки в поддиректорията `lib`, сочещи към директориите на зависимостите. Вторият `for` цикъл обхожда преходните зависимости на пакета, който се инсталира в момента, отново ги добавя към `lockfile` на текущия пакет и създава символични връзки в поддиректорията `lib` на текущата обхождана зависимост, сочещи към директориите на нейните зависимости.

Паралелност

За да не е нужно да се чака инсталацията на даден пакет да приключи, за да започне инсталацията на следващия, програмата тегли и декомпресира всеки пакет в отделна нишка. В този случай синхронизация между отделните нишки е нужна единствено при опресняване на индикаторите за напредък, разгледани в следващата секция.

Индикатори за напредък

По време на изпълнението на програмата се показват анимирани индикатори, отразяващи напредъка по процеса на инсталация на пакетите. Тези индикатори са съставени от заглавие, описващо операцията, която се извършва, и лента, опресняваща се всеки път, когато има напредък. Всеки индикатор има един или повече етапи. Например при инсталирането на даден пакет се минава през два етапа - теглене и декомпресиране. Също така само един етап на дадения индикатор може да бъде показан в даден момент. Това е така нареченият активен етап.

Индикаторите за напредък и техните етапи са имплементирани чрез класовете **ProgressBar** и **Stage** (фиг. 3.15).

```
1  /**
2   * @brief A class representing a cpm progress bar
3   */
4  class ProgressBar {
5
6      public:
7
8      /**
9       * @brief Constructor for progress bar. This creates the first stage of the
10       *        new progress bar and sets it as the active stage.
11       *
12       * @param title the title of the first stage
13       * @param width the width of the first stage progress bar
14       * @param symbol the symbol with which to fill the progress bar
15       * @param max_value the maximum value of the first stage progress bar. By
16       *        default it is 0, which means that the maximum value is unknown. In
17       *        that case the progress bar circles through infinitely to simulate
18       *        progress.
19       */
20     ProgressBar(const std::string &title, int width, char symbol,
21                int max_value = 0);
22
23     /**
24      * @brief Add a new stage to the current progress bar
25      *
26      * @param title the title of the new stage
```

```

27     * @param width the width of the first stage progress bar
28     * @param symbol the symbol with which to fill the progress bar
29     * @param max_value the maximum value of the new stage progress bar. By
30     *     default it is 0, which means that the maximum value is unknown. In
31     *     that case the progress bar circles through infinitely to simulate
32     *     progress.
33     */
34     void add_stage(const std::string &title, int width, char symbol,
35                   int max_value = 0);
36
37     /**
38     * @brief Print the active progress bar stage
39     */
40     void print_active_stage() const;
41
42     /**
43     * @brief Update/Add custom message after the active progress bar
44     *
45     * @param new_suffix the new message
46     */
47     void update_suffix(const std::string &new_suffix);
48
49     /**
50     * @brief Set the entire active progress bar to a new one
51     *
52     * @param new_suffix the new progress bar
53     */
54     void set_active_bar(const std::string &new_bar);
55
56     // prefix increment
57     ProgressBar& operator++();
58
59     // postfix increment
60     ProgressBar operator++(int);
61
62     // prefix decrement
63     ProgressBar& operator--();
64
65     // postfix decrement
66     ProgressBar operator--(int);
67
68
69     private:
70
71     /**
72     * @brief A class representing one of the progress bar's stages
73     */
74     class Stage {
75
76     public:
77
78         /**
79         * @brief Constructor for progress bar stage
80         *
81         * @param title the title of the new stage
82         * @param width the width of the first stage progress bar

```

```

83      * @param symbol the symbol with which to fill the progress bar
84      * @param max_value the maximum value of the new stage progress bar. By
85      *      default it is 0, which means that the maximum value is
86      *      unknown. In that case the progress bar circles through
87      *      infinitely to simulate progress.
88      */
89      Stage(const std::string &title, int width, char symbol,
90            int max_value = 0);
91
92      std::string title;
93      std::string bar;
94      std::string suffix;
95      int width;
96      char symbol;
97      int max_value;
98      int value;
99  };
100
101  std::vector<Stage> stages;
102  Stage active_stage;
103
104  /**
105   * @brief Fill the current active progress bar up to the value specified by
106   *      this->active_stage.value (or circle through to simulate progress).
107   */
108  void fill_bar();
109  };

```

Списък 3.15: Класове за индикатор за напредък и етап

Вследствие на това, че пакетите се инсталират паралелно, индикаторите за напредък също се опресняват паралелно. За това се грижи функцията `refresh_all_progress()` (списък 3.16). Тя първо връща курсора на конзолата толкова реда нагоре, колкото индикатори за прогрес са активни в момента умножено по две (един индикатор заема два реда). Това става чрез специалния ANSI код `\x1b[nF`, където `n` е броят редове за връщане. След това всички индикатори се извеждат на ново.

```

1  const auto refresh_all_progress = [&]() {
2      std::lock_guard<std::mutex> lock(printing_mutex);
3      std::cout << fmt::format("\x1b[{}F", all_progress.size() * 2);
4      for (const auto &[package, progress_bar] : all_progress) {
5          progress_bar.print_active_stage();
6      }
7  };

```

Списък 3.16: Функция за опресняване на всички индикатори

Индикаторите предоставят оператори за увеличаване и намаляване на напредъка, както и функции за свободна промяна на лентата

и добавяне на допълнителен надпис (suffix).

Отчет

Последната стъпка при инсталирането на пакета е неговото записване в локалното хранилище (списък 3.17).

```
1 int InstallCommand::register_package(const Package &package) {
2     return this->context.repo->add(package);
3 }
```

Списък 3.17: Функция за отчитане на пакет

3.6.4 Команда за премахване

Класът **RemoveCommand** отговаря за премахването на пакети. Начинът му на работа се състои от следните стъпки:

Проверка

Първото нещо, което командата за премахване прави, е да провери дали пакетът е инсталиран (списък 3.18). Ако не е се изпраща съобщение към потребителя, предупреждаващо, че даденият пакет вече е инсталиран.

```
1 bool RemoveCommand::check_if_not_installed(const Package &package) {
2     bool specified = this->context.repo->contains(package);
3     bool downloaded = fs::exists(this->context.cwd / paths::packages_dir /
4                                 package.get_name() / "");
5
6     return !specified || !downloaded;
7 }
```

Списък 3.18: Функция за проверка на пакет

Изтриване

Пакетите се премахват чрез функцията **delete_all()** (списък 3.19). Тя първо проверява дали пакета, който трябва да бъде премахнат, има зависимости. Ако има те се обхождат и преди да бъдат премахнати се проверява дали има други инсталирани пакети, които зависят от тях. При наличието на такива пакети зависимостта не се премахва.

```

1  std::uintmax_t RemoveCommand::delete_all(const Package &package) {
2      std::uintmax_t total_entries = 0;
3      this->context.lockfile->remove(package);
4      if (fs::exists(this->context.cwd / paths::packages_dir /
5                  package.get_name() / paths::packages_dir / "")) {
6          for (const auto &dep_path : fs::directory_iterator(
7              this->context.cwd / paths::packages_dir /
8              package.get_name() / paths::packages_dir / "")) {
9              Package dep(dep_path.path().filename().string());
10             CPM_LOG_INFO(
11                 "Checking transitive dependency {} ...", dep.get_name()
12             );
13             if (this->context.repo->contains(dep) ||
14                 this->context.lockfile->contains_dep(dep)) {
15                 CPM_LOG_INFO(
16                     "Transitive dependency {} is required by another package! "
17                     "Skipping ...",
18                     dep.get_name()
19                 );
20                 continue;
21             }
22             total_entries += this->delete_all(dep);
23             CPM_LOG_INFO("Removed transitive dependency {}", dep.get_name());
24         }
25     }
26     if (this->context.lockfile->contains_dep(package)) {
27         CPM_LOG_INFO(
28             "Package {} is required by another package! Skipping ...",
29             package.get_name()
30         );
31     }
32     else {
33         total_entries += fs::remove_all(
34             this->context.cwd / paths::packages_dir / package.get_name() / ""
35         );
36     }
37     return total_entries;
38 }

```

Списък 3.19: Функция за изтриване на пакет

Отчет

Последната стъпка при премахването на пакета е неговото изтриване от локалното хранилище (списък 3.20).

```

1  int RemoveCommand::unregister_package(const Package &package) {
2      return this->context.repo->remove(package);
3  }

```

Списък 3.20: Функция за отчитане на пакет

3.6.5 Команда за изброяване

Класът `ListCommand` отговаря за изброяването на пакети. Начинът му на работа се състои от следните стъпки:

Проверки и сдобиване с инсталираните пакети

Инсталираните пакети се взимат от текущото хранилище чрез метода `list()`. Ако файлът на хранилището не съществува или не съдържа никакви пакети се извежда съобщение, че няма инсталирани пакети.

```
1 CPM_LOG_INFO("==== Starting list command =====");
2
3 CPM_LOG_INFO("Checking if a {} file exists ...",
4             paths::package_config.string());
5 if (!fs::exists(this->context.cwd / this->context.repo->get_filename())) {
6     CPM_LOG_INFO("Repository doesn't exist in the current context!");
7     CPM_INFO("No packages installed!\n");
8     return;
9 }
10
11 auto installed_packages = this->context.repo->list();
12
13 CPM_LOG_INFO("Checking if there are any packages installed ...");
14 if (installed_packages.empty() &&
15     (!fs::exists(this->context.cwd / paths::packages_dir / "") ||
16      fs::is_empty(this->context.cwd / paths::packages_dir / ""))) {
17     CPM_LOG_INFO("Set of installed packages is empty!");
18     CPM_INFO("No packages installed!\n");
19     return;
20 }
```

Списък 3.21: Команда за изброяване

Изброяване на инсталираните пакети

Пакетите, които са инсталирани, могат да се изброят по два начина - съкратено (по подразбиране) и разширено. Разширеното изброяване може да се включи с флага `--all` (или по-краткото `-a`). При него се изброяват всички зависимости да инсталираните пакети.

```
1 CPM_LOG_INFO("Listing packages specified in {} ...",
2             (this->context.cwd / paths::package_config).string());
3 CPM_INFO("Packages in {}:\n",
4         (this->context.cwd / paths::packages_dir / "").string());
5 int installed = 0;
6 int not_installed = 0;
7 for (const auto &package : installed_packages) {
```

```

8     if (!fs::exists(this->context.cwd / paths::packages_dir /
9         package.get_name())) {
10         not_installed++;
11         CPM_INFO(" {} (not installed)\n", package.string());
12     }
13     else {
14         installed++;
15         if (this->is_used("--all")) {
16             this->print_deps(package, this->context.cwd / paths::packages_dir /
17                 package.get_name() / "");
18         }
19         else {
20             CPM_INFO(" {} \n", package.string());
21         }
22     }
23 }

```

Списък 3.22: Команда за изброяване

Изброяване на излишните пакети

Ако в папката с инсталирани пакети има такива, които не присъстват във конфигурационния файл на текущия пакет, те се изброяват като след името им се добавя текста “(unspecified)”.

```

1  CPM_LOG_INFO("Listing packages NOT specified in {} ...",
2              (this->context.cwd / paths::package_config).string());
3  int unspecified = 0;
4  if (fs::exists(this->context.cwd / paths::packages_dir / "")) {
5      for (const auto &dir_entry :
6          fs::directory_iterator(this->context.cwd /
7                                  paths::packages_dir / "")) {
8          Package package(dir_entry.path().filename().string());
9          if (!this->context.repo->contains(package) &&
10              !this->context.lockfile->contains_dep(package)) {
11              unspecified++;
12              CPM_INFO(" {} (unspecified)\n", package.get_name());
13          }
14      }
15  }
16
17  CPM_INFO("\nTotal: {} installed, {} not installed, {} unspecified\n",
18          installed, not_installed, unspecified);
19
20  CPM_LOG_INFO("==== Finished list command. ====");

```

Списък 3.23: Команда за изброяване

3.6.6 Команда за създаване

Класът `CreateCommand` отговаря за създаването на нови пакети (списък 3.24). Той създава нов `cpm_pack.json` файл със стойности по подразбиране (списък 3.25).

```
1 CPM_LOG_INFO("==== Starting create command =====");
2
3 CPM_LOG_INFO("Creating {} file ...", paths::package_config.string());
4 PackageConfig package_config(this->context.cwd / paths::package_config);
5 Package new_package = package_config.create_default();
6
7 CPM_INFO("Created package \"{}\"\\n", new_package.get_name());
8
9 CPM_LOG_INFO("==== Finished create. =====");
```

Списък 3.24: Команда за създаване

```
1 {
2     "name": "example-package",
3     "version": "0.1.0",
4     "url": "",
5     "description": "",
6     "author": "",
7     "license": ""
8 }
```

Списък 3.25: `cpm_pack.json` по подразбиране

Тези стойности са зададени в `create_default()` (списък 3.26) функцията на `PackageConfig` класа.

```
1 Package PackageConfig::create_default() {
2     std::string default_package_name = fs::current_path().stem().string();
3     SemVer default_package_version("0.1.0");
4     this->config_json["name"] = default_package_name;
5     this->config_json["version"] = default_package_version.string();
6     this->config_json["url"] = "";
7     this->config_json["description"] = "";
8     this->config_json["author"] = "";
9     this->config_json["license"] = "";
10    this->save();
11    return Package(default_package_name, default_package_version);
12 }
```

Списък 3.26: Стойности по подразбиране на нов пакет

3.6.7 Команда за синхронизиране

Класът `SyncCommand` модифицира дървовидната файлова структура от зависимости така че тя да съвпада с тази, описана във файла `cpm_pack.json`. Този клас наследява командите за инсталиране и премахване на пакети, поради нуждата да може както да инсталира липсващи пакети, така и да премахва ненужни такива. Това води до така нареченият проблем на диамантеното Наследяване (също наричан Deadly Diamond of Death [9]). В C++ той е решен чрез възможността за виртуално наследяване.

Инсталиране на липсващите пакети

Преди да бъдат инсталирани пакетите се инициализират по същия начин, както при командата за инсталирани.

```
1  CPM_LOG_INFO("==== Starting sync command =====");
2
3  CPM_LOG_INFO("Obtaining new packages from {} ...",
4              paths::package_config.string());
5  auto packages = this->context.repo->list();
6  CPM_LOG_INFO("New packages in {}: {}", paths::package_config.string(),
7              [&]() {
8              std::string packages_str = "[";
9              for (const auto &p : packages) {
10                 packages_str += p.get_name() + ", ";
11             }
12             return packages_str + "]";
13         })();
14 );
15
16 CPM_INFO("Synchronizing packages with {} ...\\n",
17         paths::package_config.string());
18
19 CPM_LOG_INFO("Initializing new packages ...", paths::package_config.string());
20 std::unordered_set<Package, Package::Hash> packages_to_install;
21 for (auto package : packages) {
22     CPM_LOG_INFO("Checking package {} ...", package.string());
23     CPM_INFO("Checking package {} ...", package.string());
24     try {
25         package.init();
26     } catch(const std::exception &e) {
27         CPM_INFO(" failed!\\n");
28         throw std::invalid_argument(e.what());
29     }
30     CPM_LOG_INFO(
31         "version {} for package {} is valid",
32         package.get_version().string(), package.get_name()
33     );
34     CPM_INFO(" found.\\n");
35     packages_to_install.insert(package);
36 }
```

```

36 }
37
38 CPM_LOG_INFO("Installing new packages ...", paths::package_config.string());
39 int new_packages = 0;
40 for (const auto &package : packages_to_install) {
41     try {
42         new_packages += this->install_package(
43             package, this->context.cwd / paths::packages_dir / ""
44         );
45     } catch(const std::exception &e) {
46         CPM_LOG_ERR(e.what());
47         CPM_ERR(e.what());
48     }
49 }
50 }

```

Списък 3.27: Инсталиране на липсващите пакети

Премахване на излишните пакети

Ако в папката с инсталирани пакети има такива, които се на споменати в конфигурационния файл на текущия пакет, те се премахват.

```

1 CPM_LOG_INFO("Removing unspecified packages ...",
2             paths::package_config.string());
3 int unspecified_packages = 0;
4 for (const auto &dir_entry :
5     fs::directory_iterator(this->context.cwd /
6                             paths::packages_dir / "")) {
7
8     Package package(dir_entry.path().filename().string());
9
10    if (!this->context.repo->contains(package) &&
11        !this->context.lockfile->contains_dep(package)) {
12        CPM_LOG_INFO("Found unspecified package {}, removing",
13                    package.get_name());
14        unspecified_packages += this->remove_package(package);
15    }
16 }

```

Списък 3.28: Премахване на излишните пакети

3.7 Съхранение на пакетите

За съхранение на списък с инсталираните пакети е използван абстрактният клас `Repository` (списък 3.29). Този клас е шаблонен, което позволява съхранението на произволни типове данни. Той има

референция към файл, използван за съхранение списъка с инсталираните пакети. Класът също предоставя методи за почти всички **CRUD** операции, без тази за обновяване.

```
1  /**
2   * @brief A class representing an object repository
3   *
4   * @tparam T the type of objects to store
5   */
6  template<typename T, typename P>
7  class Repository {
8
9      public:
10
11      /**
12       * @brief Constructor for repository
13       *
14       * @param filename the name of the file to store the repository
15       */
16      Repository(const fs::path &filename) : filename(filename) {}
17      virtual ~Repository() = default;
18
19      /**
20       * @brief Getter for filename
21       *
22       * @return The name of the repository file
23       */
24      const fs::path &get_filename() const { return this->filename; }
25
26      /**
27       * @brief Add the specified objects/s to the repository
28       *
29       * @param object the object to add
30       *
31       * @return The number of records modified in the repository
32       */
33      virtual int add(const T &object) = 0;
34
35      /**
36       * @brief Remove the specified object/s from the repository
37       *
38       * @param object the object to remove
39       *
40       * @return The number of records modified in the repository
41       */
42      virtual int remove(const T &object) = 0;
43
44      /**
45       * @brief List all of the objects in the repository
46       *
47       * @return A list of all the objects in the repository
48       */
49      virtual std::unordered_set<T, P> list() const = 0;
50
51      /**
```

```

52     * @brief Check if the specified object is in the repository
53     *
54     * @param object the object
55     *
56     * @return true if the object is found, false otherwise
57     */
58     virtual bool contains(const T &object) const = 0;
59
60
61     protected:
62
63     fs::path filename;
64 };

```

Списък 3.29: Абстрактен клас за хранилище

Този абстрактен клас има две имплементации в програмата - тази съхраняваща локално инсталирани пакети и тази, съхраняваща глобално инсталирани пакети.

3.7.1 Локално инсталирани пакети

Списъкът с локално инсталирани пакети се съхранява във файла `src_pack.json`, намиращ се в директорията на текущия пакет. Файлът съдържа масива `dependencies`, в който са описани имената на инсталираните пакети и техните конкретни версии. Самите пакети се инсталират в поддиректорията `lib` (съкратено от `libraries`).

```

1  PackageConfig::PackageConfig(const fs::path &filename) : Repository(filename) {
2      if (fs::exists(this->filename) && !fs::is_empty(this->filename)) {
3          std::ifstream package_config_in(this->filename);
4          this->config_json = nlohmann::ordered_json::parse(package_config_in);
5      }
6  }
7
8  int PackageConfig::add(const Package &package) {
9      if (this->contains(package)) {
10         return 0;
11     }
12
13     this->config_json
14         ["dependencies"]
15         [package.get_name()] = package.get_version().string();
16     this->save();
17     return 1;
18 }
19
20 int PackageConfig::remove(const Package &package) {
21     if (!this->contains(package)) {
22         return 0;
23     }

```

```

24
25     this->config_json["dependencies"].erase(package.get_name());
26     this->save();
27     return 1;
28 }
29
30 std::unordered_set<Package, Package::Hash> PackageConfig::list() const {
31     if (!this->config_json.contains("dependencies")) {
32         return std::unordered_set<Package, Package::Hash>();
33     }
34
35     std::unordered_set<Package, Package::Hash> packages;
36     for (const auto &[name, content] : this->config_json["dependencies"]
37         .items()) {
38         packages.insert(Package(name, SemVer(content.get<std::string>())));
39     }
40     return packages;
41 }
42
43 bool PackageConfig::contains(const Package &package) const {
44     if (this->list().find(package) != this->list().end()) {
45         return true;
46     } else {
47         return false;
48     }
49 }
50
51 Package PackageConfig::create_default() {
52     std::string default_package_name = fs::current_path().stem().string();
53     SemVer default_package_version("0.1.0");
54     this->config_json["name"] = default_package_name;
55     this->config_json["version"] = default_package_version.string();
56     this->config_json["url"] = "";
57     this->config_json["description"] = "";
58     this->config_json["author"] = "";
59     this->config_json["license"] = "";
60     this->save();
61     return Package(default_package_name, default_package_version);
62 }
63
64 void PackageConfig::save() {
65     std::ofstream package_config_out(this->filename);
66     package_config_out << std::setw(4) << this->config_json;
67 }

```

Списък 3.30: Хранилище за локално инсталирани пакети

3.7.2 Съхранение на графа на зависимостите (Lockfile)

Този файл описва графа на зависимостите на текущия пакет. Той предоставя две допълнителни функции, които добавят и премахват

преходни зависимости към и от директните зависимости на текущия пакет.

```
1  Lockfile::Lockfile(const fs::path &filename) : Repository(filename) {
2      if (fs::exists(this->filename) && !fs::is_empty(this->filename)) {
3          std::ifstream lockfile_in(this->filename);
4          this->lockfile_json = json::parse(lockfile_in);
5      }
6  }
7
8  int Lockfile::add(const Package &package) {
9      if (this->contains(package)) {
10         return 0;
11     }
12
13     this->lockfile_json
14         ["dependencies"]
15         [package.get_name()]
16         ["version"] = package.get_version().string();
17     this->save();
18     return 1;
19 }
20
21 int Lockfile::add_dep(const Package &existing_package, const Package &dep) {
22     if (!this->contains(existing_package) ||
23         this->lockfile_json
24             ["dependencies"]
25             [existing_package.get_name()]
26             ["dependencies"].contains(dep.get_name())) {
27         return 0;
28     }
29
30     this->lockfile_json
31         ["dependencies"]
32         [existing_package.get_name()]
33         ["dependencies"]
34         [dep.get_name()] = dep.get_version().string();
35     this->save();
36     return 1;
37 }
38
39 int Lockfile::remove(const Package &package) {
40     if (!this->contains(package)) {
41         return 0;
42     }
43
44     this->lockfile_json
45         ["dependencies"].erase(package.get_name());
46     this->save();
47     return 1;
48 }
49
50 int Lockfile::remove_dep(const Package &existing_package, const Package &dep) {
51     if (!this->contains(existing_package) ||
52         !this->lockfile_json
```

```

53         ["dependencies"]
54         [existing_package.get_name()]
55         ["dependencies"].contains(dep.get_name())) {
56     return 0;
57 }
58
59 this->lockfile_json
60     ["dependencies"]
61     [existing_package.get_name()]
62     ["dependencies"].erase(dep.get_name());
63 this->save();
64 return 1;
65 }
66
67 std::unordered_set<Package, Package::Hash> Lockfile::list() const {
68     if (!this->lockfile_json.contains("dependencies")) {
69         return std::unordered_set<Package, Package::Hash>();
70     }
71
72     std::unordered_set<Package, Package::Hash> packages;
73     for (const auto &[name, content] : this->lockfile_json["dependencies"]
74         .items()) {
75         packages.insert(
76             Package(name, SemVer(content["version"].get<std::string>()))
77         );
78     }
79     return packages;
80 }
81
82 bool Lockfile::contains(const Package &package) const {
83     if (this->list().find(package) != this->list().end()) {
84         return true;
85     } else {
86         return false;
87     }
88 }
89
90 bool Lockfile::contains_dep(const Package &package) {
91     for (const auto &dep : this->lockfile_json["dependencies"]) {
92         if (dep.contains("dependencies")) {
93             for (const auto &[name, version] : dep["dependencies"].items()) {
94                 if(name == package.get_name()) {
95                     return true;
96                 }
97             }
98         }
99     }
100     return false;
101 }
102
103 void Lockfile::save() {
104     std::ofstream lockfile_out(this->filename);
105     lockfile_out << std::setw(4) << this->lockfile_json;
106 }

```


3.7.3 Глобално инсталирани пакети

Мястото за съхранение на глобално инсталираните пакети и техните метаданни се определя от спецификацията за основни директории [10] (XDG Base Directory Specification). Според нея основната директория за данни се определя по следния начин:

- За Windows - %APPDATA%\...
- За Linux - \${HOME}/.local/share/cpm

Списъкът с глобално инсталирани пакети се съхранява във файла `packages.bd3`, намиращ се в директорията на потребителя. Файлът представлява sqlite база данни, в която има една таблица на име `installed_packages`. В нея се съхранява информацията за инсталираните пакети. Самите пакети се инсталират в поддиректорията `lib` (съкратено от `libraries`).

```
1 PackageDB::PackageDB(const fs::path &filename) : Repository(filename) {
2     fs::create_directories(filename.parent_path());
3     int err = sqlite3_open(filename.string().c_str(), &this->package_db);
4     if (err) {
5         sqlite3_close(this->package_db);
6         throw std::runtime_error("Can't open package DB!");
7     }
8
9     char *err_msg;
10    err = sqlite3_exec(this->package_db,
11        "CREATE TABLE IF NOT EXISTS installed_packages("
12        "name VARCHAR(100) NOT NULL,"
13        "version VARCHAR(100) NOT NULL,"
14        "PRIMARY KEY (name, version)"
15        ");",
16        nullptr, 0, &err_msg);
17    if (err != SQLITE_OK) {
18        sqlite3_free(err_msg);
19        throw std::runtime_error("Can't create packages table!");
20    }
21
22    sqlite3_free(err_msg);
23 }
24
25 PackageDB::~PackageDB() {
26     sqlite3_close(this->package_db);
27 }
```

```

28
29 int PackageDB::add(const cpm::Package &package) {
30     sqlite3_stmt *stmt;
31     sqlite3_prepare(this->package_db,
32         "INSERT INTO installed_packages VALUES (?, ?);",
33         -1, &stmt, nullptr);
34
35     sqlite3_bind_text(stmt, 1, package.get_name().c_str(),
36         package.get_name().length(), SQLITE_TRANSIENT);
37     sqlite3_bind_text(stmt, 2, package.get_version().string().c_str(),
38         package.get_version().string().length(), SQLITE_TRANSIENT);
39     sqlite3_step(stmt);
40     int rows_modified = sqlite3_total_changes(this->package_db);
41
42     sqlite3_finalize(stmt);
43     return rows_modified;
44 }
45
46 int PackageDB::remove(const cpm::Package &package) {
47     sqlite3_stmt *stmt;
48     sqlite3_prepare(this->package_db,
49         "DELETE FROM installed_packages WHERE name = ?;",
50         -1, &stmt, nullptr);
51
52     sqlite3_bind_text(stmt, 1, package.get_name().c_str(),
53         package.get_name().length(), SQLITE_TRANSIENT);
54     sqlite3_step(stmt);
55     int rows_modified = sqlite3_total_changes(this->package_db);
56
57     sqlite3_finalize(stmt);
58     return rows_modified;
59 }
60
61 std::unordered_set<Package, Package::Hash> PackageDB::list() const {
62     std::unordered_set<Package, Package::Hash> packages;
63     char *err_msg;
64     int err = sqlite3_exec(
65         this->package_db, "SELECT name, version FROM installed_packages;",
66         [](void *packages, int cols, char **col_vals, char **col_names) {
67             Package package(col_vals[0], SemVer(col_vals[1]));
68             static_cast<std::unordered_set<Package, Package::Hash> *>
69                 (packages)->insert(package);
70             return EXIT_SUCCESS;
71         }, &packages, &err_msg);
72     if (err != SQLITE_OK) {
73         sqlite3_free(err_msg);
74         throw std::runtime_error("Can't get installed packages!");
75     }
76
77     sqlite3_free(err_msg);
78     return packages;
79 }
80
81 bool PackageDB::contains(const Package &package) const {
82     sqlite3_stmt *stmt;
83     sqlite3_prepare(this->package_db,

```

```

84     "SELECT * FROM installed_packages WHERE name = ?;"
85     , -1, &stmt, nullptr);
86
87     sqlite3_bind_text(stmt, 1, package.get_name().c_str(),
88                       package.get_name().length(), SQLITE_TRANSIENT);
89     int row_count = 0;
90
91     while (sqlite3_step(stmt) != SQLITE_DONE) {
92         row_count++;
93     }
94
95     sqlite3_finalize(stmt);
96
97     if (row_count > 0) {
98         return true;
99     } else {
100         return false;
101     }
102 }

```

Списък 3.32: Хранилище за глобално инсталирани пакети

Глава 4

РЪКОВОДСТВО за потребителя

4.1 Инсталиране на програмата

Програмата е достъпна както за Linux, така и за Windows. Преди да започнете процеса на инсталация ще са Ви нужни следните допълнителни програми:

- git
- CMake (по желание заедно с `ninja`¹)
- C++ компилатор (g++ или MSVC)

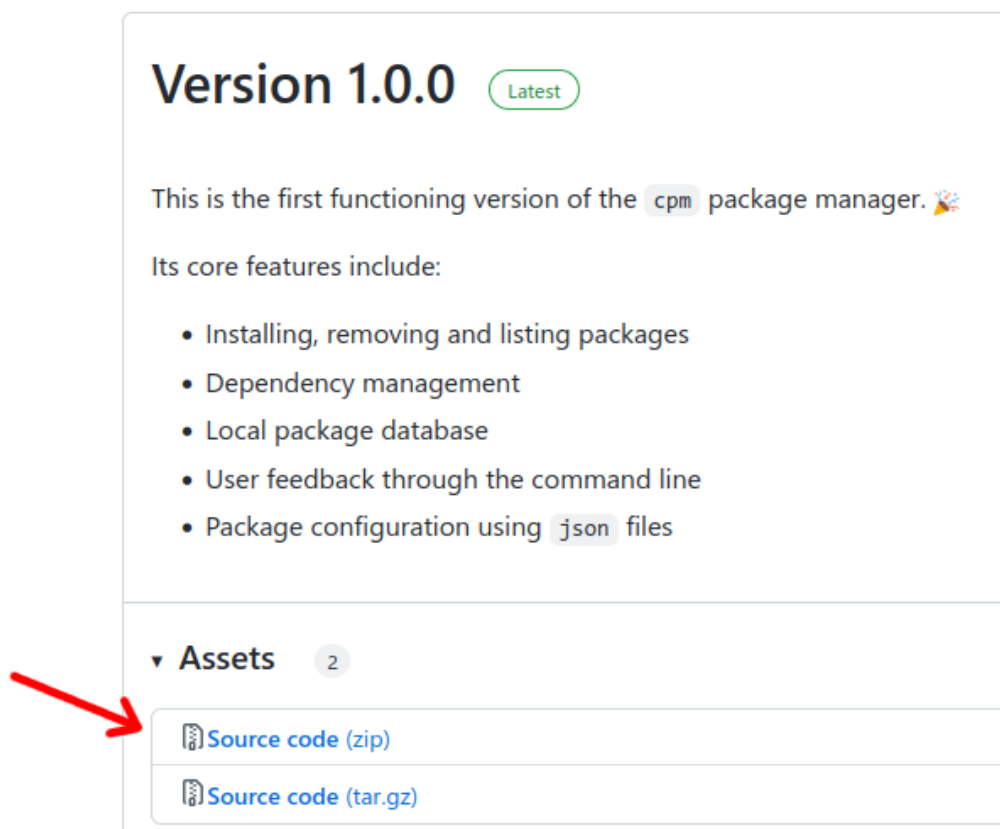
4.1.1 Сдобиване с кода на програмата

Кодът на програмата може да изтеглите от GitHub хранилището на проекта: <https://github.com/YassenEfremov/cpm>. Най-удобният и бърз начин е да клонирате хранилището директно:

```
$ git clone https://github.com/YassenEfremov/cpm
```

Като алтернатива може да изтеглите кода под формата на zip архив от страницата с официални публикации (фиг. 4.1):

¹Може да използвате build системата `ninja` за по-бърза компилация



Фигура 4.1: Теглене на кода под формата на zip архив

4.1.2 Конфигуриране и компилиране на програмата

След като сте се сдобили с кода може да преминете към компилиране на програмата. Стъпките за това са следните:

1. Инициализиране на git submodules - проектът използва git submodules за управление на зависимостите. Преди да можете да компилирате основната програма, трябва да изтеглите всички нейни зависимости. Това става лесно със следната команда:

```
$ git submodule update --init --progress
```

2. Конфигуриране на проекта - за да можете да компилирате проекта трябва първо да генерирате build файловете, необходими за вашата конкретна операционна система. За целта е препоръчително да създадете нова директория, в която ще бъдат генерирани съответните build файлове:

```
$ mkdir build
$ cd build
```

След като сте в новосъздадената директория можете да генерирате съответните build файлове като изпълните командата **cmake** и подадете директорията, съдържаща основния **CMakeLists.txt** файл на проекта, в този случай това е предишната директория, обозначена с две точки:

```
$ cmake ..
```

Това автоматично ще избере подходящата build система в зависимост от текущата операционна система. Ако желаете да използвате конкретна build система можете да подадете на **cmake** параметъра **-G** и името на build системата. За текущия проект е препоръчително да използвате build системата **ninja** за по-бърза компилация:

```
$ cmake .. -G Ninja
```

3. Компилиране на програмата - компилирането на програмата става като се изпълнят build файлове, генерирани в предишната стъпка. Следната команда автоматично разпознава генерираните build файлове и ги изпълнява:

```
$ cmake --build
```

4. Инсталиране на програмата - за да можете да достъпвате програмата от която и да е папка на компютъра трябва да я инсталирате. Това изисква администраторски права и става чрез следната команда:

```
$ sudo cmake --install
```

4.2 Синтаксис

След като сте инсталирали програмата, можете да я пуснете чрез изпълнимия файл **cm**. Ако не подадете никакви параметри се из-

вежда обобщено описание на програмата (фиг. 4.2):

```
^ ~/example-package $ cpm
Usage: cpm [--help] [--version] {create,install,list,remove,sync}

Optional arguments:
  -h, --help      shows help message and exits
  -v, --version   prints version information and exits

Subcommands:
  create          Create a new package
  install         Install the specified package/s
  list            List installed packages
  remove          Remove the specified package/s
  sync            Synchronize dependencies with the current package config
^ ~/example-package $
```

Фигура 4.2: Обобщено описание на програмата

По-горното описание показва начина на използване на програмата и параметрите, които приема. Това съобщение може да бъде показано и като подадете параметъра `--help` (или по-краткото `-h`). Може да видите коя версия на програмата е инсталирана като подадете параметъра `--version` (или по-краткото `-v`) (фиг. 4.3):

```
^ ~/example-package $ cpm --version
1.0.0
^ ~/example-package $
```

Фигура 4.3: Показване на версията

4.2.1 Инсталиране на пакети

Инсталирането на пакети става чрез командата `install`. Може да видите нейното обобщено описание като подадете параметъра `--help` (или по-краткото `-h`) (фиг. 4.4):

Командата приема произволен брой параметри, всеки от които идентифицира даден пакет, който трябва да бъде инсталиран. При подаване на пакети се използва синтаксиса `<име на пакет>@<версия>`. Ако версията не е подадена програмата автоматично намира последната такава. Пакетите могат да бъдат инсталирани по два начина:

- локално (по подразбиране) - пакетите се инсталират в папката `<cwd>/lib/` (където `<cwd>` е текущата работна директория) и се добавят като зависимости на текущия пакет. Този начин на инсталация е предназначен за библиотеки, които единствено текущият пакет ще използва.

```

^ ~/example-package $ cpm install --help
Usage: install [--help] [--global] packages

Install the specified package/s

Positional arguments:
  packages      Packages to install [nargs: 1 or more] [required]

Optional arguments:
  -h, --help    shows help message and exits
  -g, --global  install package/s globally
^ ~/example-package $

```

Фигура 4.4: Обобщено описание на командата за инсталиране

- глобално - пакетите се инсталират в глобалната директория на програмата и не се добавят като зависимости на текущия пакет. Този начин на инсталация е предназначен за библиотеки, които ще бъдат използвани от повече от един пакет. Може да бъде избран чрез подаване на флага `--global` (или по-краткото `-g`).

Ето как изглежда инсталацията на примерния пакет `histogram` и неговата зависимост - пакетът `ppm` (фиг. 4.5):

```

^ ~/example-package $ cpm install histogram
Resolving version for package histogram ... found latest: 0.1.2
Installing package into /home/yassen/example-package/lib/histogram/ ...
v Obtaining lockfile ... done.
+ histogram@0.1.2 (Extracting)
[##### ] 8/14
v ppm@0.1.0 (Downloading)
[ #] 83.41 KB

```

Фигура 4.5: Инсталиране на пакет и неговите зависимости

По време на инсталиране на пакетите се показват анимирани индикатори за напредък. Всеки пакет има собствен индикатор, състоящ се от заглавие, определящо извършващото се действие, и лента, опресняваща се всеки път, когато има напредък. Синият символ “v” пред индикатора означава теглене на данни. Зеленият символ “+” пред индикатора означава декомпресиране на данни.

След като пакетът е инсталиран, програмата предоставя на потребителя инструкции за това как да го използва чрез build системата CMake (фиг. 4.6). Инструкциите са оцветени в жълто, за да бъдат лесно различни от останалите съобщения.


```

~/example-package $ cpm install histogram
Resolving version for package histogram ... found latest: 0.1.2
Installing package into /home/yassen/example-package/lib/histogram/ ...
v Obtaining lockfile ... done.
+ histogram@0.1.2          (Extracting)
[   done   ] 14/14
+ ppm@0.1.0                (Extracting)
[   done   ] 13/13
cpm_pack.json: modified 1 record/s

**Hint**
To use the package/s add the following commands to your CMakeLists.txt:
    add_subdirectory(${CMAKE_CURRENT_SOURCE_DIR}/lib/histogram)

    target_include_directories(<target>
        PRIVATE
            ${CMAKE_CURRENT_SOURCE_DIR}/lib/histogram/include
    )

    target_link_libraries(<target>
        PRIVATE
            histogram
    )

Finished in 3.085s
~/example-package $

```

Фигура 4.6: Инструкции за използване на инсталираните пакети

Ако някой от подадените пакети е вече инсталиран, командата сигнализира на потребителя като извежда грешка и отново показва инструкции относно начина на използване на пакета (фиг. 4.7):

```

~/example-package $ cpm install histogram
Resolving version for package histogram ... found latest: 0.1.2
error: histogram: package already installed!
cpm_pack.json: modified 0 record/s

**Hint**
To use the package/s add the following commands to your CMakeLists.txt:
    add_subdirectory(${CMAKE_CURRENT_SOURCE_DIR}/lib/histogram)

    target_include_directories(<target>
        PRIVATE
            ${CMAKE_CURRENT_SOURCE_DIR}/lib/histogram/include
    )

    target_link_libraries(<target>
        PRIVATE
            histogram
    )

Finished in 990ms
~/example-package $

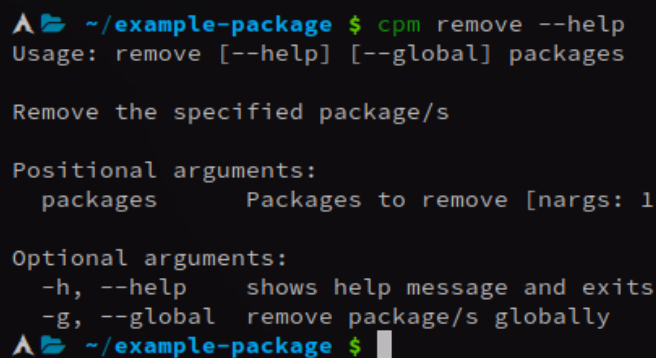
```

Фигура 4.7: Опит за инсталиране на вече инсталиран пакет

За момента програмата не позволява инсталирането на повече от една версия на един и същ пакет.

4.2.2 Премахване на пакети

Премахването на пакети става чрез командата **remove**. Може да видите нейното обобщено описание като подадете параметъра **--help** (или по-краткото **-h**) (фиг. 4.8):



```
~/example-package $ cpm remove --help
Usage: remove [--help] [--global] packages

Remove the specified package/s

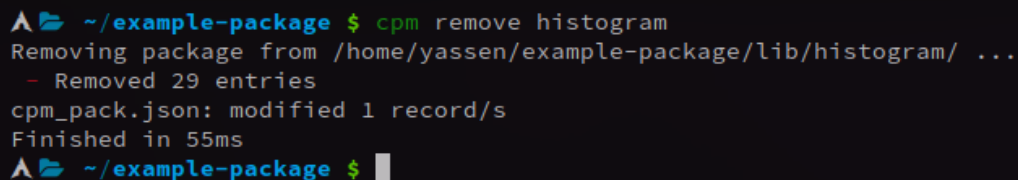
Positional arguments:
  packages      Packages to remove [nargs: 1 or more] [required]

Optional arguments:
  -h, --help    shows help message and exits
  -g, --global  remove package/s globally
~/example-package $
```

Фигура 4.8: Обобщено описание на командата за премахване

Също като командата за инсталиране, командата за премахване приема произволен брой параметри, всеки от които идентифицира даден пакет, който трябва да бъде премахнат. Отново пакетите могат да бъдат премахнати по два начина - локално (по подразбиране) и глобално, като параметрите към програмата са аналогични (**--global** или **-g**).

Ето как изглежда премахването на примерния пакет **histogram** и неговите зависимости (фиг. 4.9):



```
~/example-package $ cpm remove histogram
Removing package from /home/yassen/example-package/lib/histogram/ ...
- Removed 29 entries
cpm_pack.json: modified 1 record/s
Finished in 55ms
~/example-package $
```

Фигура 4.9: Премахване на пакет и неговите зависимости

Командата за премахване не показва анимиран индикатор за напредък, а единствено броя на премахнатите файлове. Червеният символ “-” пред индикатора сигнализира изтриване на данни.

Ако някой от подадените пакети не е инсталиран, командата сигнализира на потребителя като извежда грешка (фиг. 4.10):

```

^ ~/example-package $ cpm remove histogram
error: histogram: package not installed!
cpm_pack.json: modified 0 record/s
Finished in 0ms
^ ~/example-package $

```

Фигура 4.10: Опит за премахване на пакет, който не е инсталиран

4.2.3 Изброяване на пакети

Изброяването на пакети става чрез командата `list`. Може да видите нейното обобщено описание като подадете параметъра `--help` (или по-краткото `-h`) (фиг. 4.11):

```

^ ~/example-package $ cpm list --help
Usage: list [--help] [--global] [--all]

List installed packages

Optional arguments:
  -h, --help      shows help message and exits
  -g, --global    list globally installed package/s
  -a, --all       list all packages, including dependencies
^ ~/example-package $

```

Фигура 4.11: Обобщено описание на командата за изброяване

Командата за изброяване не приема пакети като параметри. Тя извежда имената и версиите на инсталираните пакети. Също като предишните две команди и тази работи в два режима - локален (по подразбиране) и глобален - като съответно в първия случай изброява пакетите инсталирани в текущата директория, а във втория случай тези, инсталирани глобално в глобалната директория на програмата. Параметрите към програмата са аналогични (`--global` или `-g`).

По подразбиране командата изброява единствено директно инсталираните пакети. Ако искате да видите всички инсталирани пакети и техните зависимости трябва да подадете параметъра `--all` (или по-краткото `-a`) (фиг. 4.12). Пакетите се изреждат един под друг, като всяка зависимост е подравнена по-навътре от предишната.

4.2.4 Създаване на пакети

Създаването на нов пакет става чрез командата `create`. Може да видите нейното обобщено описание като подадете параметъра `--help` (или по-краткото `-h`) (фиг. 4.13):

```

^ ~/example-package $ cpm list -a
Packages in /home/yassen/example-package/lib/:
  histogram@0.1.2
  ppm@0.1.0

Total: 1 installed, 0 not installed, 0 unspecified
^ ~/example-package $

```

Фигура 4.12: Изброяване на всички инсталирани пакети и техните зависимости

```

^ ~/example-package $ cpm create --help
Usage: create [--help]

Create a new package

Optional arguments:
  -h, --help  shows help message and exits
^ ~/example-package $

```

Фигура 4.13: Обобщено описание на командата за създаване

Тази команда не приема допълнителни параметри и работи само в локален режим. При изпълнението ѝ се създава нов `cpm_pack.json` в текущата директория с генерирани стойности по подразбиране (фиг. 4.1):

```

1 {
2   "name": "example-package",
3   "version": "0.1.0",
4   "url": "",
5   "description": "",
6   "author": "",
7   "license": ""
8 }

```

Списък 4.1: `cpm_pack.json` по подразбиране

4.2.5 Синхронизиране на пакети

Зависимостите на текущия пакет могат да бъдат променени и ръчно като директно се редактира файлът `cpm_pack.json`. За да влязат в действие тези промени обаче трябва да бъде изпълнена командата `sync`. Може да видите нейното обобщено описание като подадете параметъра `--help` (или по-краткото `-h`) (фиг. 4.14):

Тази команда също не приема никакви допълнителни параметри и отново работи само в локален режим. При нейното изпълнение се инсталират всички липсващи пакети, изброени във файла

```

^ ~/example-package $ cpm sync --help
Usage: sync [--help]

Synchronize dependencies with the current package config

Optional arguments:
  -h, --help    shows help message and exits
^ ~/example-package $

```

Фигура 4.14: Обобщено описание на командата за синхронизиране

`cpm_pack.json`, и се премахват всички излишни пакети, които не присъстват във файла `cpm_pack.json`. Командата използва същите индикатори за напредък, които използват и командите за инсталиране и премахване на пакети (фиг. 4.15):

```

^ ~/example-package $ cpm sync
Synchronizing packages with cpm_pack.json ...
Checking package ppm@0.1.0 ... found.
ppm: Already installed, skipping ...
histogram: Not specified, removing ...
- Removed 16 entries
Synchronization complete.
Installed 0 new, removed 1 unspecified

**Hint**
To use the package/s add the following commands to your CMakeLists.txt:
  add_subdirectory(${CMAKE_CURRENT_SOURCE_DIR}/lib/ppm)

  target_include_directories(<target>
    PRIVATE
      ${CMAKE_CURRENT_SOURCE_DIR}/lib/ppm/include
  )

  target_link_libraries(<target>
    PRIVATE
      ppm
  )

Finished in 1.128s
^ ~/example-package $

```

Фигура 4.15: Синхронизиране на пакети

4.3 Други функционалности

В глобалната директория се съхраняват и някои други полезни файлове:

- `log/cpm.log` - Файл, в който се записва диагностична информация относно действията, извършени от програмата. Този

файл може да бъде от полза при отстраняване на грешки, възникнали по време на изпълнението на програмата.

- **package_locations.json** - Файл, в който са изброени имената на GitHub профилите/организациите, от които програмата се опитва да тегли пакети. Файлът съдържа **JSON** масива **package_locations**, в който по подразбиране е посочена примерната организация **cpm-examples** (фиг. 4.2). Тя може да бъде премахната или могат да бъдат добавени колкото и да е и каквито и да е други профили/организации като единственото ограничение е те да бъдат публично достъпни.

```
1 {  
2   "package_locations": [  
3     "cpm-examples"  
4   ]  
5 }
```

Списък 4.2: package_locations.json по подразбиране

Заклучение

Настоящата дипломна работа реализира базов вариант на система за управление на пакети за програмните езици C и C++. Системата бе написана на C++, като е достъпна за платформите Linux и Windows. Тя предоставя конзолен потребителски интерфейс, посредством който могат да бъдат изпълнени команди, позволяващи инсталирането, премахването, изброяването, създаването и синхронизирането на софтуерни пакети. Като регистър за съхранение на пакетите бе използвана платформата GitHub. Системата предоставя механизъм за автоматично управление на зависимостите на пакетите посредством символични връзки (symlinks). Други допълнителни функционалности са воденето на диагностични записки и показването на анимирани индикатори за напредък по време на изпълнението на програмата.

Възможностите за бъдещо развитие на проекта са многобройни: следващата голяма цел би била разработката на собствен регистър за съхранение на пакети, предоставящ специално разработени точки за достъп (endpoints), позволяващи по-лесно сдобиване с кода на пакетите и техните зависимости; други цели са автоматична компилация на кода при теглене на пакетите, поддръжка за други build системи освен CMake (например Meson и Bazel) и много допълнителни команди, улесняващи потребителя - команда за търсене на пакети, команда за актуализация на вече инсталирани пакети, команда за автоматично публикуване на новосъздадени пакети и други.

Библиография

- [1] Tom Preston-Werner. *Semantic Versioning 2.0.0*. URL: <https://semver.org/>.
- [2] Fabian Sauter и Tim Stack. *Curl for People*. URL: <https://docs.libcpr.org/>.
- [3] *An Introduction To The SQLite C/C++ Interface*. URL: <https://www.sqlite.org/cintro.html>.
- [4] Niels Lohmann. *JSON for Modern C++*. URL: <https://json.nlohmann.me/>.
- [5] *How npm3 Works*. URL: <https://npm.github.io/how-npm-works-docs/npm3/how-npm3-works.html>.
- [6] Roger Leigh. *Support for version suffixes*. URL: <https://gitlab.kitware.com/cmake/cmake/-/issues/16716>.
- [7] Erich Gamma и др. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994.
- [8] GitHub Docs. *Download a repository archive (zip)*. URL: <https://docs.github.com/en/rest/repos/contents?apiVersion=2022-11-28#download-a-repository-archive-zip>.
- [9] Robert C. Martin. “Java and C++: A critical comparison”. B: (1997).
- [10] Waldo Bastian и др. *XDG Base Directory Specification*. URL: <https://specifications.freedesktop.org/basedir-spec/basedir-spec-latest.html>.

Съдържание

Използвани съкращения	3
Увод	5
1 Методи за управление на софтуерни пакети	6
1.1 Основни концепции	6
1.1.1 Видове системи за управление на пакети	6
1.1.2 Видове софтуерни пакети	7
1.1.3 Моделът клиент - регистър	7
1.1.4 Описание на пакетите (метаданни)	8
1.1.5 Управление на зависимостите	8
1.2 Съществуващи системи	10
1.3 Съществуващи системи за C и C++	10
2 Проектиране на система за управление на пакети	12
2.1 Функционални изисквания	12
2.2 Подбор на средства за разработка	13
2.2.1 Програмен език	13
2.2.2 Build система	14
2.2.3 Система за управление на версиите	15
2.2.4 Среда за разработка	15
2.2.5 Непрекъсната интеграция (CI)	15
2.2.6 Използвани библиотеки	16
2.2.7 Алгоритъм за управление на зависимостите . .	17
3 Реализация на проекта	19
3.1 Структура на проекта	19
3.2 Файлове с общо предназначение	20
3.3 Диагностика (Logging)	20
3.4 Пакети	21
3.4.1 Инициализация на пакет	22

3.5	Версии	26
3.6	Команди	27
3.6.1	Регистриране на команди	29
3.6.2	Обработка на подадените параметри	29
3.6.3	Команда за инсталиране	31
3.6.4	Команда за премахване	40
3.6.5	Команда за изброяване	42
3.6.6	Команда за създаване	44
3.6.7	Команда за синхронизиране	45
3.7	Съхранение на пакетите	46
3.7.1	Локално инсталирани пакети	48
3.7.2	Съхранение на графа на зависимостите (Lockfile)	49
3.7.3	Глобално инсталирани пакети	52
4	Ръководство за потребителя	55
4.1	Инсталиране на програмата	55
4.1.1	Сдобиване с кода на програмата	55
4.1.2	Конфигуриране и компилиране на програмата	56
4.2	Синтаксис	57
4.2.1	Инсталиране на пакети	58
4.2.2	Премахване на пакети	61
4.2.3	Изброяване на пакети	62
4.2.4	Създаване на пакети	62
4.2.5	Синхронизиране на пакети	63
4.3	Други функционалности	64
	Заклучение	66