VIPACK Manager
by
Ruben Khachatryan

BA in Computer Science, National Polytechnic University of Armenia, 2016

A thesis submitted in partial satisfaction of

the requirements for the degree of

Master of Science

in

Computer & Information Science

in the

**Zaven and** Sonia **Akian** College of Science and Engineering

of the

AMERICAN UNIVERSITY OF ARMENIA

Supervisor:Norayr Chilingaryan
Signature: ____________________ Date: ____________

Committee Member: Aram Hajian
Signature: ____________________ Date:______________

Committee Member: ______________________
Signature: ____________________ Date:______________

Committee Member: ______________________

Signature: ____________________ Date:______________

## Abstract:

Modern trend with programming tools is having a tool to manage libraries and library dependencies. To name a few, those are pip for python, npm or yarn for nodejs, golang has several (godep, wgo), composer for php, etc... Package managers currently have big scale of usage and for some platforms are an inseparable part of the ecosystem(like npm in NodeJS or bundler in Ruby).

We are going to explore how to create one for **Oberon** ecosystem.

## Author:

Ruben Khachatryan

## Keywords:

Oberon programming language, Package manager, package managers, package manager in modular programming languages, module management

# Licenses for Software and Content

## Software Copyright License (to be distributed with software developed for masters project)

Copyright (c) 2019, Ruben Khachatryan

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

(This license is known as “The MIT License” and can be found at http://opensource.org/licenses/mit-license.php)

**Content Copyright License (to be included with Technical Report)**

LICENSE

Terms and Conditions for Copying, Distributing, and Modifying

Items other than copying, distributing, and modifying the Content with which this license was distributed (such as using, etc.) are outside the scope of this license.

1. You may copy and distribute exact replicas of the OpenContent (OC) as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the OC a copy of this License along with the OC. You may at your option charge a fee for the media and/or handling involved in creating a unique copy of the OC for use offline, you may at your option offer instructional support for the OC in exchange for a fee, or you may at your option offer warranty in exchange for a fee. You may not charge a fee for the OC itself. You may not charge a fee for the sole service of providing access to and/or use of the OC via a network (e.g. the Internet), whether it be via the world wide web, FTP, or any other method.

2. You may modify your copy or copies of the OpenContent or any portion of it, thus forming works based on the Content, and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified content to carry prominent notices stating that you changed it, the exact nature and content of the changes, and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the OC or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License, unless otherwise permitted under applicable Fair Use law.

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the OC, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the OC, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it. Exceptions are made to this requirement to release modified works free of charge under this license only in compliance with Fair Use law where applicable.

3. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to copy, distribute or modify the OC. These actions are prohibited by law if you do not accept this License. Therefore, by distributing or translating the OC, or by deriving works herefrom, you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or translating the OC.

NO WARRANTY

4. BECAUSE THE OPENCONTENT (OC) IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE OC, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE OC "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK OF USE OF THE OC IS WITH YOU. SHOULD THE OC PROVE FAULTY, INACCURATE, OR OTHERWISE UNACCEPTABLE YOU ASSUME THE COST OF ALL NECESSARY REPAIR OR CORRECTION.

5. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MIRROR AND/OR REDISTRIBUTE THE OC AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE OC, EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

(This license is known as “OpenContent License (OPL)” and can be found at http://opencontent.org/opl.shtml)

# VIPACK Manager

(Vishap package manager for Oberon)

Ruben Khachatryan
Supervisor: Norayr Chilingaryan

Abstract: 1
Author: 1
Keywords: 1
Licenses for Software and Content 2
Software Copyright License (to be distributed with software developed for masters project) 2
**Introduction(intention) 8**
What is the concept of package manager? 8
What is Oberon? 8
Choice of the development platform. 8
Why it is important? 9
General and specific Goals of the Thesis: 9
Defining the granularity of the package. 10
Hypothesis 10
Testing 10
Validation 10
Quality Measurement 11
Things required to study. 11
**Literature review 11**
Areas to look in 11
Questions 12
Searching terms 12
Criteria 12
The list of papers 12
**The literature review conclusion 13**
Comparison between gentoo vs debian package managers 13
How Package Manager works? 14
How can we use Oberon to create Package manager? 15
What speed we expect from the package manager? 15
Do we need optimizations in package install/uninstall strategies? 15
Which kind of problems people encounter in current implementations of package managers? 15
How to maintain security of the package manager? 16
**Architecture 16**
Introduction 16

Definition of a package 17
Package dependencies 17
What are the use cases for the final product? 17
Server side file system hierarchy diagram 18
**Package File Content** **18**
Version file 19
* versioning 20
**Approach** **21**
**Package manager** **21**
Architecture 22
Modules description 22
Dependency tree 23
Why do we use JSON as manifest file? 23
The Lack of libraries 23
Description 23
JSON parsing module architecture and OOP 23
Code of the JSON class definition 24
HTTP module 25
**Comparison** **27**
**Possible enhancements** **27**
Circular Dependencies prevention 27
Package manager allows to build package when creating dependency tree 28
**Conclusion** **28**

# Introduction(intention)

## What is the concept of package manager?

Package manager is a piece of software that helps developers to keep track of external dependencies and libraries for a project.

- Package manager is used as a collaboration tool
- Package manager is needed to implement big scale projects which are very hard to implement all by hand
- Package manager allows to reduce the size of the project and not keep big modules and inside the project itself
- Version controlling

## What is Oberon?

Oberon is a highly efficient, general-purpose programming language, descendant of Pascal and Modula-2. It is simpler yet more powerful than its predecessors. Oberon programs are structured, modular and type-safe. Object-oriented programming is supported through the type extension mechanism, single inheritance, procedural variables, type-bound methods, data hiding and encapsulation, and garbage collection.
https://modulaware.com/mdltws.htm

"*Oberon is a general-purpose programming language that evolved from Modula-2. Its principal new feature is the concept of type extension. It permits the construction of new data types on the basis of existing ones and to relate them.*"

*The Programming Language Oberon*

Defining "Oberon" is not trivial. There are several revisions of the language, made by its author Niklaus Wirth and described in corresponding language reports (88, 07, 13). He also created the compilers for these languages that run under Oberon operating system. Then there is Oberon-2, its report was signed by Wirth and his student Mossenbock, and there are some relative programming languages that preserve the spirit, like Component Pascal and Active Oberon. As we mentioned, Oberon can also describe the operating system, written in the eponymous language.
Thus the question needs to be answered, what exactly do we mean by "Oberon package manager".

## Choice of the development platform.

We decided to use one of the Oberon-2 compilers, which runs on several architectures and under different operating systems - voc.
One of those is our primary working platform - Linux, so we decided to use the compiler and the libraries bundled with it under Linux operating system, and see later is it feasible/desirable to port it to work under one of Oberon operating systems.
Keeping that in mind explains the choice of object model - we decided to stick to the classical Oberon object model, though voc compiler supports Oberon-2 dialect which provides "methods", or type-bound procedures, in Oberon terminology.

What striked us instantly when studying the Oberon language is its approach to design: Oberon removed several features of its ancestor Modula-2, but became more powerful and flexible, by having smaller and simpler syntax. While other languages evolved by keeping and adding features, Oberon's approach was to keep only necessary and essential features: every feature of the language had to have strong reasons to take part of the design.

It is hard for us to not notice the parallels with modernist movements: the form follows functions, and excesses are not welcomed. Therefore we will have a courage to describe the Oberon language as the modernistic programming language.

## Why is it important?

- Oberon is a fully featured minimalistic language which deserves to have package manager which will benefit the community and will help to grow the auditorium
- An open source tool will benefit enthusiasts explore the field to see how solutions might be implemented
- Package managers are currently in trend
- Package managers make package repositories smaller as they do not include whole libraries but references to them
- Libraries that are downloaded with package managers are commonly tested by community

## General and specific Goals of the Thesis:

- Decide on package definition
- Design Architecture for storing packages
- Design Architecture package manager
- Decide which part of the architecture is critical in the scope of this thesis to prove that it is feasible
- Implement the package manager
- Implement the server architecture
- Test
- Compare with other ready solutions
- Think about the future(vision for the platform)

# Approach

Any language has its own specifics, and the package managers have their own approaches. We have to analize already existing approaches and solutions, to decide what features exist that can be helpful in our case, as well as explore specifications of Oberon language as the implementation heavily depends on the technology.

Creating libraries and modules which will help to prove that creating package manager for Oberon is possible.

### Defining the granularity of the package.

In Oberon we have very granular approach as concept of module that can be dynamically imported to running programs. The issue is that a single module is a very small unit and often can not be considered as a full package. For example, graphical editor software can contain several modules, that are responsible for its specific file formats, for its specific classes, objects, and tools. At the same time, the graphical editor can depend on the set of the libraries that work with the graphic files like tiff or png. As Oberon modules already have dependencies, we could decide to use module as a package unit. However we decided to define package as  a set of modules which are dependent from one another and are connected to each other logically, as well as keeping in mind single responsibility principle as much as possible.

## Hypothesis

It's possible to create a package manager ecosystem for modular programming languages such as Oberon, using only Oberon programming language, Oberon compiler and system APIs.

### Testing

- The application should be written using open source tools
- We will measure speed of run time in our local environment to get an idea on how it should perform in real life scenario

### Validation

1. Set up the server which stores package(manifest) files
2. Set up the server which stores packages that contains module sources
3. Compile the package manager
4. Create sample package file
5. Run the application with sample Package file, it will:
    a. Download package files
    b. Parse package files
    c. Create dependency tree

   d. Download dependency packages
6. Compile
   a. Dependency modules
   b. Our package
7. Run the compiled application

### Quality Measurement

We are going to evaluate efficiency of our program based on time that program runs, this will give us information about how can our package manager compete with other modern solutions.

1. Fast

   Program runs in less then a second.
2. Medium

   Program runs in in more than a second and less than 5 seconds.
3. Slow

   Program runs in more than 5 seconds.

## Things required to study.

- Oberon Language
- OS package management
- Software package management
- Current trends and solution for items described above

What are the steps, actions, literature research, experiments, etc... we will perform in order to accomplish the goals of the thesis. Which approaches are relevant to the topic and which approaches are correct once to be used. We will explore all of the corresponding topics below.

# Literature review

### Areas to look in

- Package manager in software development
- Package manager in Operating systems
- Oberon Programming language
- Modular architectures

## Questions

- How Package Manager works?
- How can we use Oberon to create Package manager?
- What can we expect from the speed of package manager run time?
- Which problems people see in current implementations of package managers?
- How to maintain security in package management?
- Do we need optimizations in package install/uninstall strategies?

## Searching terms

- Software package management
- Oberon
- Security of a package manager
- Npm, pip, godep
- Index file
- Manifest file
- Package Manager use cases

And mentioned word combinations.

## Criteria

- Relation to the topic
- Relatively new(as the field is evolving very fast)
- Authors(the authors should be known in the field)

## The list of papers


1. Forbes JA, Stone JD, Parthasarathy S, Toutonghi MJ, Sliger MV, inventors; Microsoft Corp, assignee. Software package management. United States patent US 6,381,742. 2002 Apr 30.
2. Forbes J, Stone J, Parthasarathy S, Toutonghi M, Sliger M, inventors; Microsoft Corp, assignee. Software package management. United States patent application US 10/071,526. 2002 Oct 3.
3. Staicu CA, Pradel M, Livshits B. SYNODE: understanding and automatically preventing injection attacks on NODE. JS. InNetwork and Distributed System Security (NDSS) 2018 Feb 20 (p. 1).
4. Abate P, DiCosmo R, Treinen R, Zacchiroli S. MPM: a modular package manager. InProceedings of the 14th international ACM Sigsoft symposium on Component based software engineering 2011 Jun 20 (pp. 179-188). ACM.
5. Abate P, Di Cosmo R, Treinen R, Zacchiroli S. A modular package manager architecture. Information and Software Technology. 2013 Feb 1;55(2):459-74.

6. Di Cosmo R, Zacchiroli S, Trezentos P. Package upgrades in FOSS distributions: Details and challenges. arXiv preprint arXiv:0902.1610. 2009 Feb 10.
7. Tucker C, Shuffelton D, Jhala R, Lerner S. Opium: Optimal package install/uninstall manager. In29th International Conference on Software Engineering (ICSE'07) 2007 May 20 (pp. 178-188). IEEE.
8. Cicchetti A, Di Ruscio D, Pelliccione P, Pierantonio A, Zacchiroli S. A model driven approach to upgrade package-based software systems. InEvaluation of Novel Approaches to Software Engineering 2009 May 9 (pp. 262-276). Springer, Berlin, Heidelberg.
9. Darwin IF. Android Cookbook: Problems and Solutions for Android Developers. " O'Reilly Media, Inc."; 2017 May 10.
10. Mugler J, Naughton T, Scott SL. OSCAR meta-package system. In19th International Symposium on High Performance Computing Systems and Applications (HPCS'05) 2005 May 15 (pp. 353-360). IEEE.
11. Guo PJ. CDE: Run Any Linux Application On-Demand Without Installation. InLISA 2011 Dec 4.
12. Wookey MJ, inventor; Sun Microsystems Inc, assignee. Apparatus and method for generating a software dependency map. United States patent application US 11/862,987. 2008 Aug 21.
13. Cappos J, Samuel J, Baker S, Hartman JH. Package management security. University of Arizona Technical Report. 2008 Jul:08-2.
14. Cappos J, Samuel J, Baker S, Hartman JH. A look in the mirror: Attacks on package managers. InProceedings of the 15th ACM conference on Computer and communications security 2008 Oct 27 (pp. 565-574). ACM.
15. Vidal SK, Antill J, inventors; Red Hat Inc, assignee. Determining when to update a package manager software. United States patent US 9,417,865. 2016 Aug 16.
16. Mössenböck H, Wirth N. The programming language Oberon-2. Structured Programming. 1991 Oct;12(4):179-96.

# The literature review conclusion

## Comparison between gentoo vs debian package managers

Debian (deb) and Redhat’s (rpm) packages, and package management systems have similarities. The package has a file which is used to build it from the source. It’s a .desc file in case of Debian and .spec file in case of Redhat system.
Debian’s package manager was called apt-get, and Redhat started to include a yum package manager, which works with rpm files later. Suse Linux distribution introduced other tool that manages rpm packages - zypper.
The packages are distributed via online repositories - usually accessible via ftp or http. There are source packages, and binary packages - source packages also include build scripts and patches.

Package manager works on a client machine. How does package manager resolve dependencies of the packages that reside on a remote file systems? For that package directories on the servers are indexed, and special index files are created.
Yum and apt package managers, that work on client machines, have a functionality to download package archive indexes (yum makecache / apt-get update). Then, by having the indexes, they also get the information about package dependencies. When the user asks the package manager to install some software, the package manager uses the indexes to calculate dependencies and asks the user if it's okay to download those.

Gentoo Linux approach is different. Gentoo uses package files called ebuilds. Those are plain text files, that describe the dependencies of the package, and how to build it. The files reside in so called 'portage tree' - file system hierarchy that contains ebuild files. There is also a separate file called 'Manifest' that keeps hashes for the files needed to be downloaded.
Package manager 'emerge' uses information stored in plain text ascii ebuild files locally to calculate the dependencies of the package. Then it downloads the packages from the URLs described in ebuild files. Though most of this software is copied to Gentoo repositories, the design of the package manager does not require any special repositories to be created. Once the user synced the portage tree, then the package manager is able to calculate dependencies and download the sources to build those. Sources themselves can be located everywhere in the Internet - often it is possible to find sources that are located at university sites and/or are not redistributable according to the licenses.

We decided to borrow some of the Gentoo package manager decisions and use plain text package files, store those on the local file system. Thus in principle there is no need in a central repository of downloadable packages.

## How Package Manager works?

Most general approach in package managers are used as following

- Install Strategy
  - Create a dependency tree for all dependencies for the project. Download all the necessary files
  - In some package management systems there are so-called "soft" dependencies which might be neglected, we do not intend such a logic
- Uninstall Strategy
  - Recursively delete all underlying dependencies for specified package.
- Update strategy
  - Update packages based on new version and rules setup in manifest file
- Automations
  - Some package managers are used for automating workflows

A good strategy for package is to have a manifest file which contains

- Version,

- sources of files of the package
- and underlying package description for ability to create a dependency tree.

## How can we use Oberon to create Package manager?

Oberon is a minimalistic programming language which has a modular architecture. Its design is beneficial for creating single module packages (one module per package), those can serve as foundations for bigger packages. As Oberon is a system level programming language, and usually Oberon compilers generate native code, it is fairly possible to create efficient programs. Also, Oberon programs (in case modules don't use SYSTEM module, which implements system specific logic) are guaranteed to be portable.

## What speed we expect from the package manager?

Even though classical Oberon language has no multithreading constructs, which means all the operations are executed sequentially, it is still is a low level programming language and should be up to the competition with other modern package managers if not outperform them.

## Do we need optimizations in package install/uninstall strategies?

**Opium**(Optimal package install uninstall manager) that is cited above, implements optimizations regarding install/uninstall strategies. Those optimizations especially relate to "uninstall" procedure, when package manager relies on statistics to decide which packages should not be uninstalled, and can be needed later(while installing other packages). Though it will save some processing time and internet bandwidth, we decided to neglect such optimizations at this time. Avoiding these kind of optimizations may be also more in line with Wirth's approach to avoid unnecessary optimizations that are making program output unpredictable.

*"Generate predictable code"*

*Compiler Construction*
*The Art of Niklaus Wirth*

## Which kind of problems people encounter in current implementations of package managers?

- Recursive package removal
    - Recursive package removal causes problems as many packages can be dependent on one package and that makes removing dependencies slow and inefficient, as the whole dependency tree should be scanned multiple times.
- Dependency tree build time
    - It is time consuming to build dependency trees.

    - It takes even more  compute time for larger projects which have interconnected libraries.

In both cases we see a place for improvement. We will consider improvements later in the scope of this project.

### How to maintain security of the package manager?

There are many aspects to consider in terms of security of the package manager. As it lets user to install code which will be executed on a computer, I consider only security vulnerabilities that can cause unpredictable program behavior.

We  observe 2 types of attacks below:

- Attacker replaces package files which leads to downloading insecure packages by the user.
- Package intentionally or unintentionally contains malicious code.

Both of the issues are resolvable but will not be discussed in context of this application as now our priority is a Proof of concept of a package manager and not fault tolerant product.
Possible solutions:

- Checking integrity of the files with cryptographic fingerprints.
- Use secure channels like HTTPS to prevent man in the middle attacks.
- Client can have plugin which will do static code analysis and will warn the user if the package has
    - command executions
    - file system accesses
    - data transfer

- Install only from trusted sources
    - Introduce a list of trusted sources.

# Architecture

## Introduction

This section discusses overall strategy and architecture of the application.

## Definition of a package

Package is considered as a group of modules related to each other logically or functionally. For example, package may define installation of a standalone program or library pack without dependencies or something that has a lot of dependencies. Package should have a “package file” which will define it(sometimes referred as manifest). There should be a “Version file” which will describe a version of a specific “package file”.

Example: Vipack package manager is a package

## Package dependencies

Packages may have a list of dependencies. It allows the manager to create the dependency tree. If package requires 2 dependencies of the same version, only one of them will be downloaded and installed. If the required versions are different, then both of the versions of the package should be downloaded.

## What are the use cases for the final product?

The main purpose of the product is dependency management for Oberon projects. However, it can be also used as a package manager of the Oberon OS itself (assuming that OS specific libraries, such as "Internet" library which is a wrapper over linux sockets, are implemented).

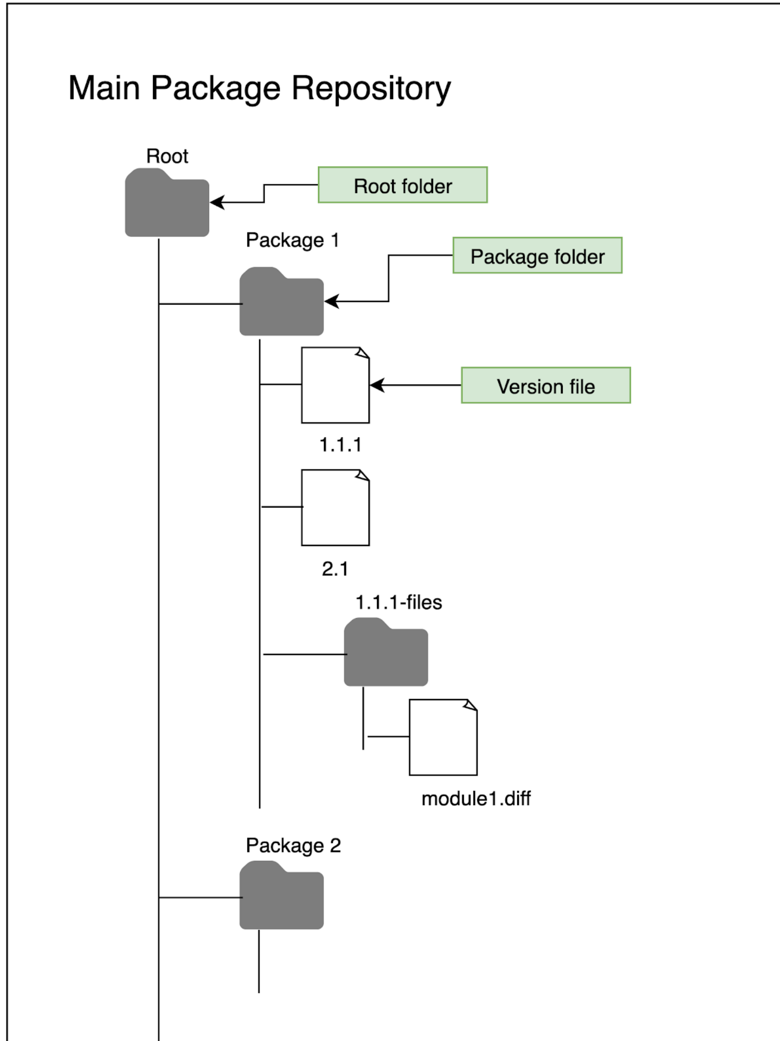
We believe it can be used as a generic package manager for any files as it does not implement any Oberon specific logic. As the app itself is a monolite client without any server logic, it is possible to use it with different custom servers.

CLI interactions

```
# command will initialize(generate package file for the project)
vipack init

# command will init dependencies
vipack install dependency-name@version

# command will run build, compilation, linking or whatever is written under "Scripts"
vipack run command

# generate version file
vipack generate version
```

## Server side file system hierarchy diagram

Server for our application is  a simply a file server with no special software to back the logic, it is visually described in a image below. Therefore any file server can be used(assuming we have corresponding protocol libraries).




## Package File Content

Below “package file” content example is shown. It will be considered as a manifest file standard.

```
{
  "Package": "Package-name",
  "Author": "Author Name",
  "License":"License name",
  "Version": "1.0.0",
  "Remote": "http://vishap.oberon.com",
  “Port”: 80,
  “Path”: “/path/10.0.0”,
  "Files": [
          "File1.Mod",
          "File2.Mod",
          "MakeFile"
  ],
```

```
 “Main”: “main.Mod”
 "Dependencies": {
         "Package-2-name": "2.5.*",
         "Package-3-name": "2.*",
         "Package-4-name": "2.1.4"
 },
 // if exists will be executed after dependency resolution
 "Patches": {
             “module0”: [“patch0.diff”, “patch1.diff”, “patch2.diff”],
             “module1”: [“patch_64bit_fix.diff”]
},
 "Scripts": {
         "build": "make build",
         "test": "make test",
         "run": "make run",
         "compile": "make compile"
 }
}
```

## Version file

“Version file” is the file that should be generated by the package manager itself. It contains all the package data as well as signatures for listed files to maintain security and reliability of the package.

```
{
 "Package": "Package-name",
 "Author": "Author Name",
 "License":"License name",
 "Version": "1.0.0",
 "Remote": "http://vishap.oberon.com",
 “Port”: 80,
 “Path”: “/path/10.0.0”,
 "Files": [
         "File1.Mod": “someHash",
         "File2.Mod": “someHash",,
         "MakeFile": “someHash",
 ],
 "Dependencies": {
         "Package-2-name": "2.5.*",
         "Package-3-name": "2.*",
```

```
        "Package-4-name": "2.1.4"
 },
 // if exists will be executed after dependency resolution
 "Patches": [“module0”: {patch0.diff, patch1.diff, patch2.diff},
             “module1”: {patch_64bit_fix.diff}
],
 "Scripts": {
        "build": "make build",
        "test": "make test",
        "run": "make run",
        "compile": "make compile"
 }
}
```

## * versioning

If a package has a version that contains a symbol “*”(asterisk), the package can be updated by the package manager replacing * with the maximal version of the package available in the remote repository. “*” can be placed only at the end of the version

Valid

- 2.*

Invalid

- 2.*.0

Same strategy(asterisk is replaced by the highest available version) should implemented while installing.
Example:

If package file lists version

- 1.*

and repository versions are

- 1.1.1
- 1.2.1
- 1.2.2

Then version 1.2.2 should be installed, as “*" is replaced with the maximal available value.

# Approach

- It might seem strange that we have remote and port fields in the “Version File”: this design aimed to provide packages for different cases like

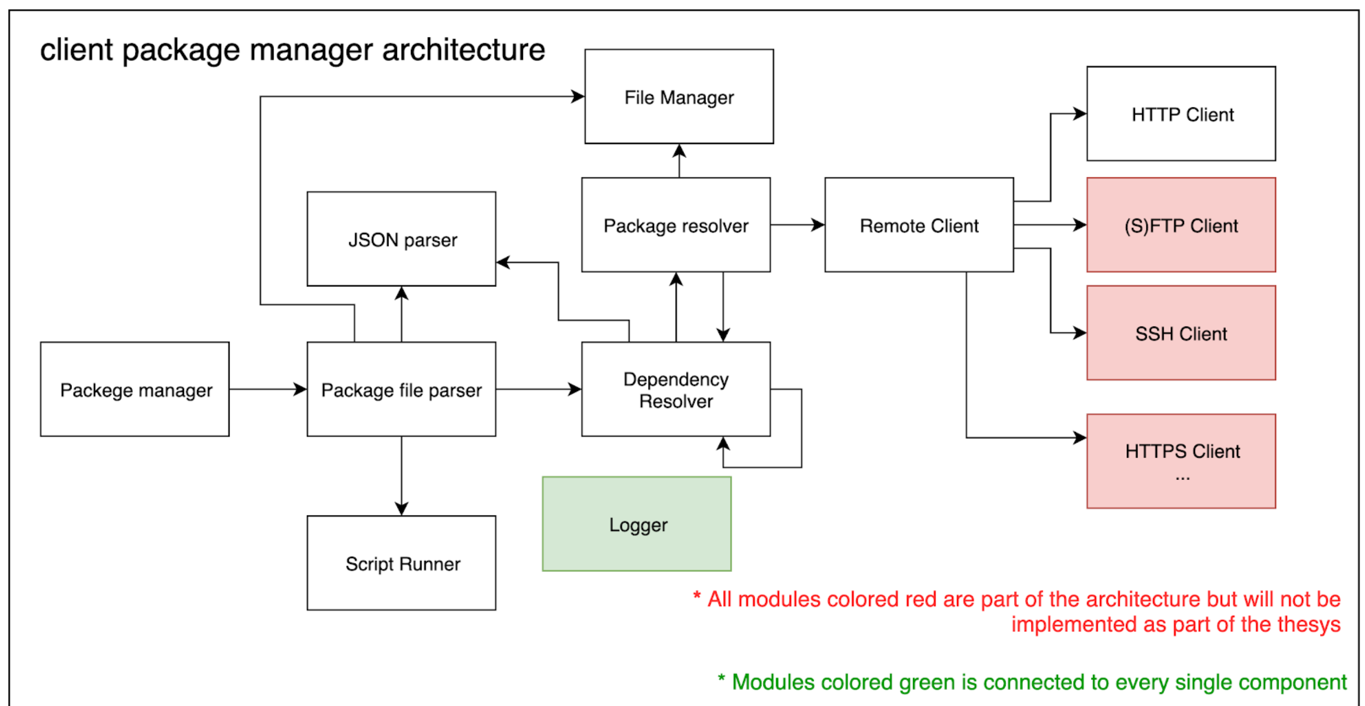
    - Sources listed in the package file may be located on other servers, for example because they are distributed under the licenses, that prevent redistribution.
    - This approach saves space in the repository server, as it is not necessary to host all the sources, but only version files.

- Why use 2 files (Version file and Package file)
    - Package file will be generated by the developer and is used at development stage.
    - Version files will be generated by the tool itself. It will contain version and signatures for the package.
- Why we use hashes(signatures)
    - Hashes are used to make sure that files downloaded by the package manager are correct, and will help us to maintain minimal amount of security without doing some fancy validations.

# Package manager

Package manager application is a client side program. This approach should help to maintain scalability, as no back end logic is applied. The approach is similar to Linux based package managers that were discussed higher, and will help us to focus on the functionality of the system rather than on spending time on scalable backend architectures.

# Architecture




## Modules description

- Package file parser is the package responsible for parsing the version file of the program
- JSON parser
    - This module is responsible for parsing JSON Files
- Package manager
    - Is the main module of the program which has entry point.
- Script Runner
    - This module is responsible for running scripts which are described in the “Script” section of version file.
    - It is also responsible for running patches.
- Logger
    - Logger is a module which logs information from other modules to standard output.
- Dependency resolver
    - This module is responsible for resolving dependencies for given package
- Package Resolver
    - Is responsible for resolving **single** package
- File manager
    - This module is used for writing and reading files from the file system.
- Remote Client

- This module used for downloading packages from the remote repositories.

### Dependency tree

To create a dependency tree it is necessary to traverse recursively through all project packages. The issue with tree like structure is that it will take too much time to traverse all packages with all the dependencies, which we consider not acceptable in terms of efficiency. We decided to use HashMap like structure to find redundant dependencies. With a good implementation we can expect ~O(1) access time.

## Why do we use JSON as manifest file?

Json file format was chosen for multiple reasons

- Json is a good way to describe an object
- Json is universal across many languages
- Json is simple to parse/generate as a file
- Json is human readable and easily editable

## The Lack of libraries

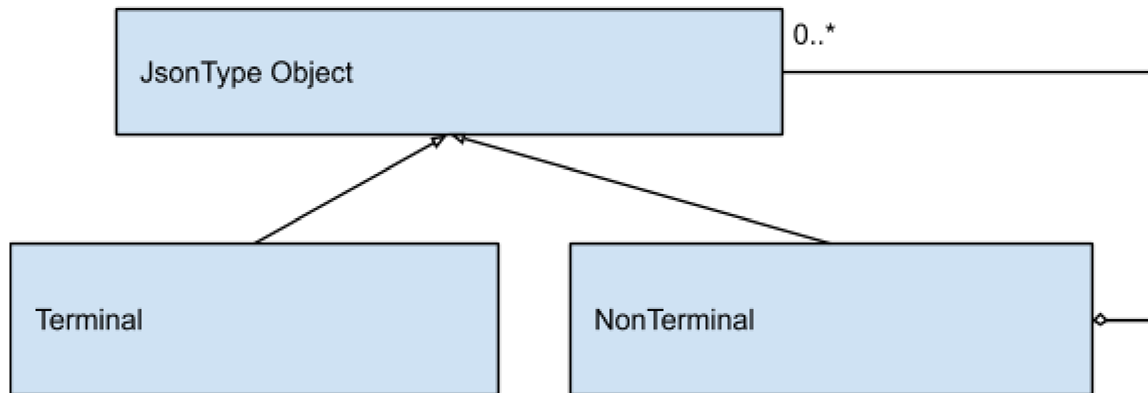
### Description

Oberon has very limited community and therefore there are no many libraries available. Though it was possible to use external Linux tools (like git and wget) or create bindings to existing libraries like curl, we decided to write some modules and contribute by this project to the Oberon ecosystem by writing native JSON and HTTP modules in Oberon in the scope of this project.

### JSON parsing module architecture and OOP

The concept of module is very similar to the concept of an Object so most of the modules can be treated as Objects, but not JsonParser . The point is that JsonParser needs to have state and can have multiple layers, having multiple states is impossible using Oberon Modules(as the state will be shared). Oberon has object structure which was be used in implementing OOP architecture.

I used **composite like** pattern architecture to implement it along with Push Down Automata approach to create JsonModule.




- Terminal is a final entity
- Non terminal is entity which contains of a JSON type object

All the Terminal and NonTerminal values are keeped in HashMap like structure.

## Code of the JSON class definition

```
  JsonTypePointer* = POINTER TO JsonType;

  JsonType* = RECORD
     GetTerminal* : PROCEDURE(self : JsonTypePointer; string : ARRAY OF CHAR; VAR
returnValue : ARRAY OF CHAR): BOOLEAN;
     GetNonTerminal* : PROCEDURE(self : JsonTypePointer; key : ARRAY OF CHAR):
JsonTypePointer;
     HasKey* : PROCEDURE(self : JsonTypePointer; key : ARRAY OF CHAR): BOOLEAN;
     TypeOfTheKey* : PROCEDURE(self : JsonTypePointer; key : ARRAY OF CHAR; VAR
returnValue : ARRAY OF CHAR);
     GetTerminalKeys* : PROCEDURE(self : JsonTypePointer; VAR destination : ARRAY OF
TString);
     GetTerminalValues* : PROCEDURE(self : JsonTypePointer; VAR destination : ARRAY OF
TString);
     GetNoneTerminalKeys* : PROCEDURE(self : JsonTypePointer; VAR destination : ARRAY OF
TString);
     GetTerminalNumber* : PROCEDURE(self : JsonTypePointer): LONGINT;
     GetNonTerminalNumber* : PROCEDURE(self : JsonTypePointer): LONGINT;

     TerminalKeys : ARRAY ArrayMaxNumber OF TString;
```

```
    TerminalValues* : ARRAY ArrayMaxNumber OF TString;
    TerminalNumber* : LONGINT;

    NonTerminalKeys : ARRAY ArrayMaxNumber OF TString;
    NonTerminalValues : POINTER TO ARRAY OF JsonTypePointer;
    NonTerminalNumber* : LONGINT;
  END;
```

## HTTP module

As in Oberon there is no http client, it was created by hand using tcp-sockets.

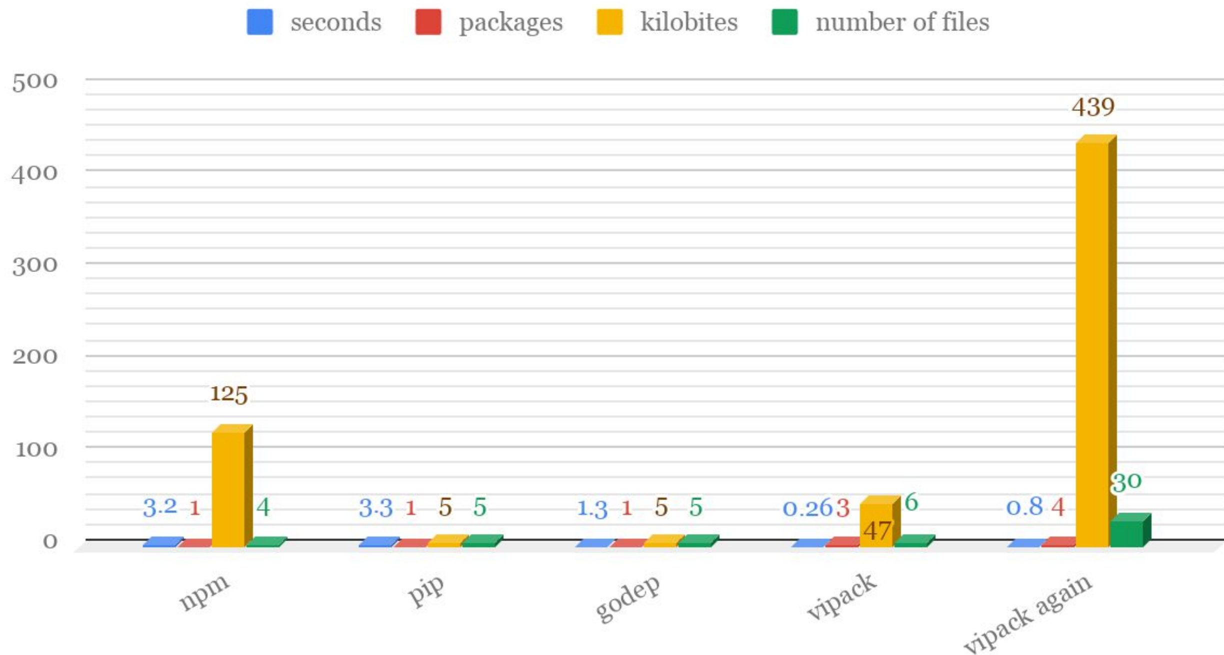
- Module can send HTTP 1.1 requests and
- Module is tested for
  - Apache
    
    and
  - nginx.
- Module is not working with a “python simple server”

Below is the implementation example of the fetching function that was implemented.

```
PROCEDURE get *(host, port, path: ARRAY OF CHAR; VAR buff: ARRAY OF CHAR);
VAR
  socket : Internet.Socket;
  connectionFlag: BOOLEAN;
  valueContentLength: REAL;
  send, valueContentLengthString: ARRAY MAXARRAYNUMBER OF CHAR;
  sendClean: PSTRING;
  httpTail: ARRAY 16 OF CHAR;
  endOfLine: ARRAY 3 OF CHAR;
  tmpBuff: ARRAY MAXARRAYNUMBER OF CHAR;
BEGIN
  Empty(buff);
  httpTail := " HTTP/1.1";

  connectionFlag := Internet.Connect(host, port, socket);
```

```
  send := "GET ";

  Strings.Append(path, send);
  Strings.Append(httpTail, send);



  AppendEOLAndClean(send, sendClean);
  connectionFlag := Internet.Write(socket, sendClean^);

  addHeader("HOST", host, sendClean);
  connectionFlag := Internet.Write(socket, sendClean^);

  addHeader("User-Agent", "oberon-http-client/1.0", sendClean);
  connectionFlag := Internet.Write(socket, sendClean^);

  addHeader("Accept", "*/*", sendClean);
  connectionFlag := Internet.Write(socket, sendClean^);

  AppendEOLAndClean("", sendClean);
  connectionFlag := Internet.Write(socket, sendClean^);
  REPEAT
    Empty( tmpBuff);

    connectionFlag := Internet.Read(socket, tmpBuff);
    Strings.Append(tmpBuff, buff);
    getHeader(buff, "Content-Length", valueContentLengthString);
    Strings.StrToReal(valueContentLengthString, valueContentLength);
    Logger.Log(buff);
  UNTIL ~connectionFlag OR (Strings.Length(buff) > valueContentLength);
  Internet.Disconnect(socket);
END get;
```

# Comparison




In result of our testing we can observe that our cli is very fast(though it lacks optimization).

- Keep in mind that testing was done in ideal conditions (we had just one user testing with the server). It can be seen from the chart "vipack" package manager is very fast and can compete with modern package managers.

# Possible enhancements

## Circular Dependencies prevention

Having circular links from one object to another is known to be an issue in many programming languages. We believe packages that reference each other should not be allowed when using package manager, as it will create an infinite loop while creating out dependency tree. That will mean that program will not stop. Though there is no need for implementing that kind of solution as we have no application for having circular dependency in package manager as in Oberon modules can not cross import(cross reference) each other, we came up with the idea of using a HashMap like structure which will allow us to resolve such issues.

Some part of  the vision for later updates and path described in the current section

- Package update strategies
- Recursive package removal
- Security(as described in a literature review section)
- Ability to add plugins - such as “static code analysis”
- Optimizations in code
    - Hash-table implementations (to work in ~O(1))
    - Make Json tree implementation based on **composite** pattern
        - Use Getters/Setters
    - Json module should be able to export a “text” in order to write to a “Version File”
    - Create character stack based on array(vector) and not list
- Tool that build version files
    - Version files should be build to generate hash
- Distinguish between recursive package updates
- Deciding which files are necessary to rebuild (based on some signature/timestamp)
- While downloading packages detect Modules with same names

### Package manager allows to build package when creating dependency tree

- While creating dependency tree(hashmap in our case) program can resolve dependency first, thus eliminating need for makefile.

- Add alias to vipack

```
alias vpk="vipack"
```

# Conclusion

All solutions and observations made above are taken from papers and personal experience.

As we can see it is possible to create a fast package manager which is shown in current research, our main point was not to focus on the server architecture and rather to focus on the package manager itself: having minimal file architecture on server to back it up. This is the other example of the application of ‘less is more’ principle: having no central server backend logic means that there’s no running server application to hack, and it contributes to scalability, because it is not necessary to store packages on the central server: they can be stored on different servers. The package/version files themselves can be stored on different servers, and user may choose to combine several “overlays” of package trees.

The sources are compiled and tested with the steps described in “Testing” section. By testing it we have shown that one can verify that program runs as “fast” as is described in Quality Measurement section.

The program runs in ~0.3 second, so we can safely say that quality measurement has possessed.

I deeply believe that the solution implemented is up to competition and will serve its purpose.