# Large Scale Programming with Modules

## Concepts of Programming Languages
## 7 December 2020

Bernhard Saumweber
Rosenheim Technical University

# Enterprise Programming

- Large codebases
- Conflicting dependencies
- Stable dependencies
- Compatiblity requirements

# What is a Module?

- A module is an unit of Programming, Testing, Distribution, Versioning and Visibility
- Languages have diffent interpretations and focus
- Decomposition into modules is a well understood

"The major advancement in the area of modular programming has been the development of coding techniques and assemblers which (1) allow one module to be written with little knowledge of the code in another module, and (2) allow modules to be reassembled and replaced without reassembly of the whole system."

On the Criteria To Be Used in Decomposing Systems into Modules, David Parnas 1972

(https://www.win.tue.nl/~wstomv/edu/2ip30/references/criteria_for_modularization.pdf)

# History

- Originally because of technical limitations
- Compile time was long and expensive
- Memory was limited
- Solution: compile smaller separate modules and link them together

## Today

- Large codebases
- Many developers on a single project
- Often even different companies involved
- Re-using code, libraries
- Structuring code into functional modules

## Visibility / Information Hiding Principle

- Makes sure that certain content of a module is private and can not be accessed be other modules
- The benefits are the same like OO encapsulation but at larger scale
- Other modules can only depend on the public (API) part of a module
- Modules hide complexity from clients (information hiding)

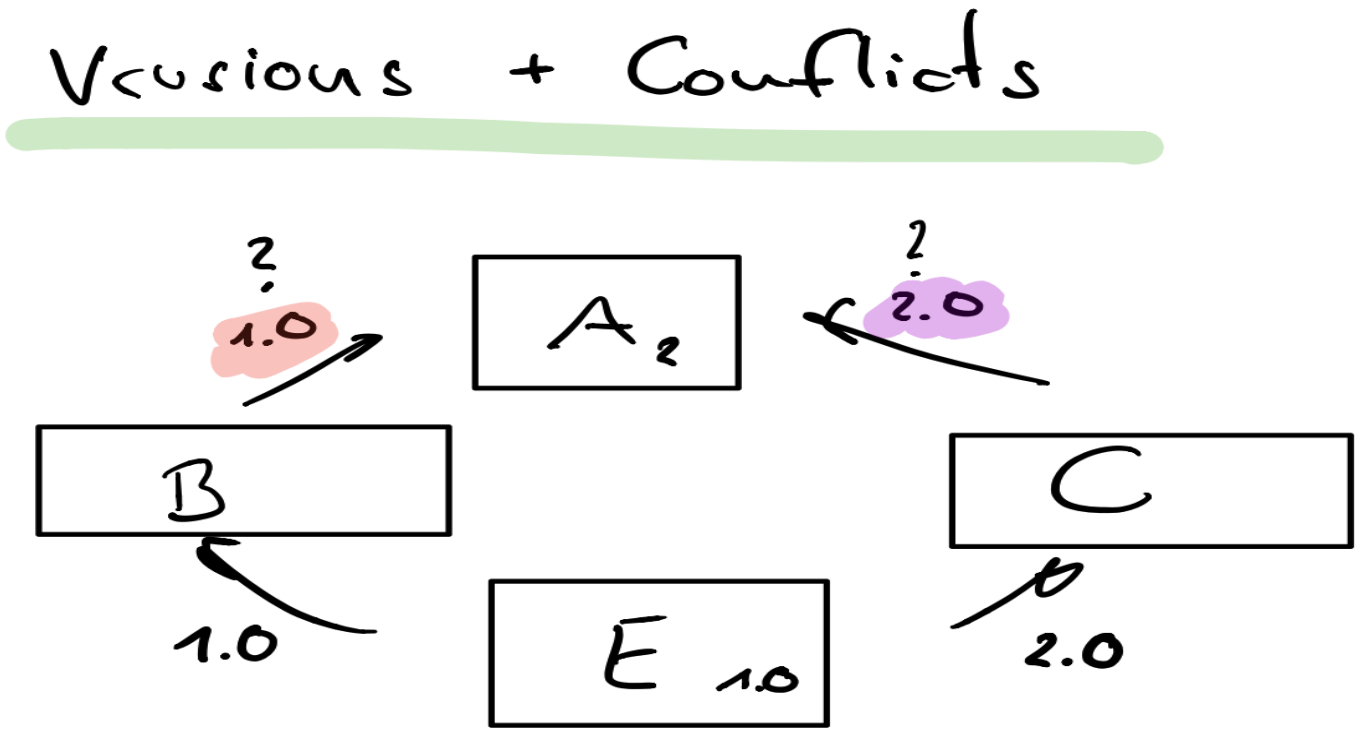
## Distribution and Replaceability

- Modules are monolithic deployment units
- Modules could be separate developed and released
- Languages differ in the way an application is assembled from modules (static or dynamic linking)
- Modules can be exchanged without affecting the whole system

## Versioning

- Modules should be versioned to guarantee stable dependencies
- The version should make the module immutable, so a module can be clearly identified by its version
- Languages differ greatly on that issue (From support of naming conventions to build tools (maven))

# Versions and Conflicts



# Module System in Java

- Java knows Modules since Java SE 9 (Project Jigsaw)
- Java Modules focus on Distribution and Visibility but not on Versioning!
- Java need 3rd party tools to implement versionized dependencies (Maven, Gradle ...)

www.sigs-datacom.de/uploads/tx_dmjournals/weigend_JS_05_16_6n6J.pdf (https://www.sigs-datacom.de/uploads/tx_dmjournals/weigend_JS_05_16_6n6J.pdf)

## Modules in other programming languages

- Rust: Cargo
- Javascript: NPM
- Ruby: Gems
- ...

# The Go Module System

www.youtube.com/watch?v=F8nrpe0XWRg (https://www.youtube.com/watch?v=F8nrpe0XWRg)

youtu.be/V8FQNen4WAA (https://youtu.be/V8FQNen4WAA)

youtu.be/aeF3l-zmPsY (https://youtu.be/aeF3l-zmPsY)

# The Go Module System

- There is an evolution in the Go module systems
- Before 1.11: Go dep was the tool of choice to implement safe, versioned dependencies
- Since 1.11: Go modules is the concept for the future
- Can be controlled by `$GO111MODULE` environment variable
- No need to develop on the `$GOPATH` anymore
- Usually one repository = one module, but also multiple modules possible

Go Module System (https://github.com/golang/go/wiki/Modules)

Go Module Reference (https://golang.org/ref/mod)

# The Principles of Versioning in Go

**Repeatability**
Make sure that build results never change over time !!!

**Compatibility**
Make sure that released code stays compatible. Make incompatible changes explicit visible to users of your code !!!

**Cooperation**
Make sure your code ist stable with the latest minor version of a dependent library

# Semantic Versioning

- Distinguish MAJOR, MINOR and PATCH version numbers
- A version number is of the form: <MAJOR>.<MINOR>.<PATCH> (e.g 1.2.0, 0.9.0-alpha)
- Semantic versioning defines several rules how and when to change the version number

semver.org (https://semver.org)

Change ...

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards compatible manner, and
- PATCH version when you make backwards compatible bug fixes.

# The go.mod File

Go module: a tree (directory) of source files with a go.mod file in the root directory

Example `go.mod` file:

```
module github.com/my/module/v3

go 1.14

require (
    github.com/some/dependency v1.2.3
    github.com/another/dependency v0.1.0
    github.com/additional/dependency/v4 v4.0.0
)
```

# How to create a new module

A module can be created with a simple command:

```
mkdir module
cd module
go mod init example.com/my/module
```

If your code is already under version control (e.g. git), you may simply run

```
go mod init
```

# Managing Dependencies

```
require (
    github.com/some/dependency v1.2.3
    github.com/another/dependency v0.1.0
    github.com/additional/dependency/v4 v4.0.0
)
```

- go.mod is automatically updated on `go get` and `go build`
- Semantic version numbers
- Version is part of the module path!
- Allows other versions to be used in the same build
- Code sharing between versions is possible
- v1 is implied

# Exclude and Replace Modules

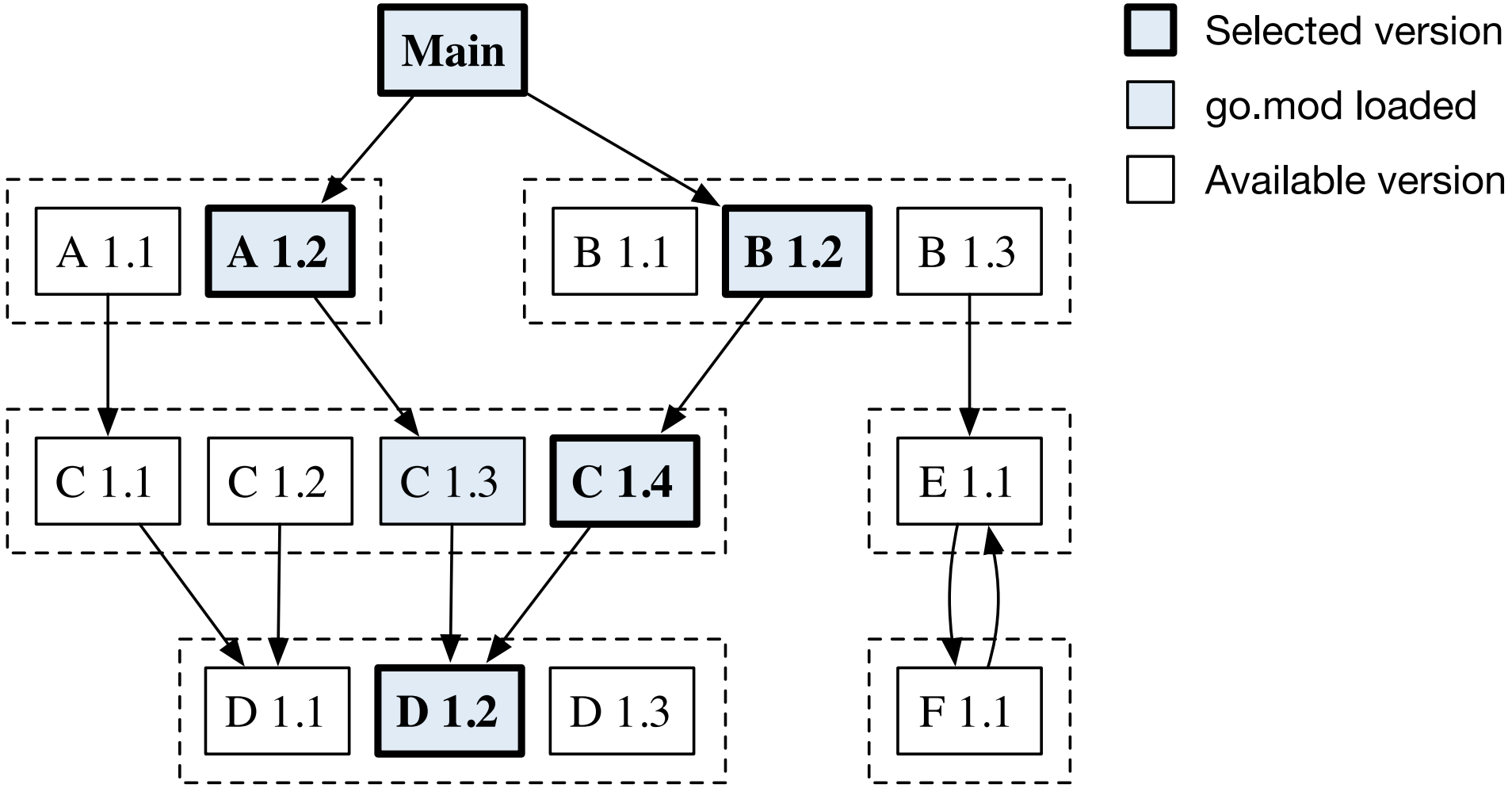
```
exclude (
    golang.org/x/crypto v1.4.5
    golang.org/x/text v1.6.7
)
```

- Useful to load the next higher non-excluded version instead
- Both release and pre-release versions are considered for this purpose

```
replace (
    golang.org/x/net v1.2.3 => example.com/fork/net v1.4.5
    golang.org/x/net => example.com/fork/net v1.4.5
    golang.org/x/net v1.2.3 => ../fork/net
    golang.org/x/net => ../fork/net
)
```

- Useful to specify a fork with a fixed bug
- Also possible to refer to a relative or absolute path on your file system
- gohack tool automates this process: github.com/rogpeppe/gohack

# Minimal Version Selection



# The go.sum File

```
github.com/golang/protobuf v1.3.2 h1:6nsPYzhq5kReh6QImI3k5qWzO4PEbvbIW2cwSfR/6xs=
github.com/golang/protobuf v1.3.2/go.mod h1:6lQm79b+lXiMfvg/cZm0SGofjICqVBUtrP5yJMmIC1U=
golang.org/x/image v0.0.0-20200927104501-e162460cd6b5 h1:QelT11PB4FXiDEXucrfNckHoFxwt8USGY1ajP1ZF5lM=
golang.org/x/image v0.0.0-20200927104501-e162460cd6b5/go.mod h1:FeLwcggjj3mMvU+oOTbSwawSJRM1uh48EjtB4UJZ
```

- Contains the cryptographic checksums of the module's dependencies
- Hashes of the module version's file tree and of the go.mod file
- h1 means sha256 (only hashing algorithm at the moment)
- Untagged revisions can be referred to using a "pseudo-version" like `v0.0.0-yyyymmddhhmmss-abcdefabcdef`, where the time is the commit time in UTC and the final suffix is the prefix of the commit hash

## Module Downloads

- Old versions of Go used `git clone` to install dependencies
- Now the `go.mod` file and a zip package of that version is downloaded
- This means modules are effectively immutable
- Hence a CDN or proxy for retrieving modules can be used
- Useful in corporate builds

## Hands On

- Write a Go module mail with API and impl
- The Mail implementation should log a message with logrus (https://github.com/sirupsen/logrus)
- Change your module dependency to a former logrus version
- Write a second Go module client which uses the mail module
- Make an incompatible change to your mail API and increase the major version number

# Plugins

## Golang and Libraries

- Go uses static linking (output is a single binary containing all libraries)
- Go support static and dynamic linking of C libraries
- Go supports dynamic loading of plugins

# Linking of C libraries

```
#include <stdio.h>
void myprint(char *s);
```

```
#include "mylib.h"
void myprint(char *s) {
    printf("You said: %s\n", s);
}
```

```
package main

// #cgo CFLAGS: -I.
// #cgo LDFLAGS: -L. -lmylib
// #include "mylib.h"
// #include <stdlib.h>
import "C"
import "unsafe"

func main() {
    s := C.CString("Hello Lib!")
    defer C.free(unsafe.Pointer(s))
    C.myprint(s)
}
```

# Dynamic linking of C libraries

```
dynamic:
    # compile the source into an object file (position independent code)
    gcc -fPIC -c mylib.c
    # convert the resulting object file into a shared library
    gcc -shared -o libmylib.so mylib.o
```

Deleting the `libmylib.so` throws error:

```
./main
dyld: Library not loaded: libmylib.so
  Referenced from: ./main
  Reason: image not found
zsh: abort      ./main
```

# Static linking of C libraries

```
static:
    # compile the source into an object file
    gcc -c mylib.c
    # convert the resulting object file into a library
    ar rc libmylib.a mylib.o
    # build an index inside the library
    ranlib libmylib.a
```

Deleting the `libmylib.a` is fine:

```
./main
You said: Hello Lib!
```

## Plugins

- Available since Go 1.8
- Before: launch separate programs and use e.g. OS exec calls, sockets or gRPC
- Plugins are shared objects (.so) library files
- Can be built using `-buildmode=plugin` directive
- Public functions and variables are exposed as ELF symbols that can be looked up and be bound to at runtime using the `plugin` package

ELF = Executable and Linkable Format: common standard file format for shared libraries

## Golang Plugin Example

Plugins can be opened using `Open()` provided in package `plugin`:

```
p, err := plugin.Open(os.Args[1])
```

Lookup searches for a symbol named `MyFunc` in plugin p. A `Symbol` is any exported variable or function.

Looking up a function:

```
// Code in plugin: func MyFunc() { return "Hello!" }
myFuncSymbol, err := p.Lookup("MyFunc")
var myFunc func() := myFuncSymbol.(func())
myFunc()
```

Looking up a variable:

```
// Code in plugin: var MyVar int
myVarSymbol, err := p.Lookup("MyVar")
var myVar *int = v.(*int)
*myVar = 42
```

## Golang Plugins Caveats

- Must be built with the exact same Golang version
- Must be built with the exact same libraries
- Only Linux, FreeBSD and macOS are supported at the moment
- Building inside a Docker container is recommended for reproducible builds
- Rather large size (static linking)

## Golang Plugin Examples

- KrakenD: fast API gateway
- Gosh: a pluggable command shell
- ...

## Hands On

- Export your Go mail module from the previous exercise to a mail plugin
- The Mail implementation should log a message with logrus (https://github.com/sirupsen/logrus)
- Write a third Go module client which uses the mail plugin and also logrus
- Make an incompatible change to your mail API and increase the major version number
- Try to use a different logrus version in the plugin (e.g. v1.7.0) and the client (e.g. v1.4.2)

Windows users: use Golang in your Windows Subsystem for Linux or a Docker container.

# Thank you

Bernhard Saumweber
Rosenheim Technical University
bernhard.saumweber@qaware.de (mailto:bernhard.saumweber@qaware.de)
http://www.qaware.de (http://www.qaware.de)