



Dealing with software dependencies

PETER BINDELS

KIKI DE ROOIJ

Introduction

- ▶ Peter Bindels
- ▶ 8 years at TomTom
 - Large code base maintenance
 - Wrote Hippomocks (before TomTom)
 - Wrote cpp-dependencies (with many others)
- ▶ Official open-source maintainer for
cpp-dependencies @ TomTom



Introduction

- ▶ Kiki de Rooij
- ▶ 2,5 years at TomTom
 - Part-time as student
 - Internship
 - Fulltime
- ▶ Internship about dependencies

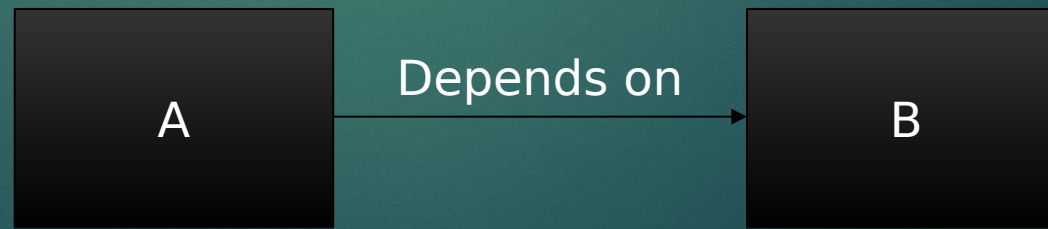


Table of content

- ▶ What are dependencies?
- ▶ Dependency types
- ▶ SOLID
- ▶ Why dependencies?
- ▶ What are modules?
- ▶ How to reduce dependencies?
- ▶ Tools to visualize dependencies

Dependency definition

- ▶ Relationship
- ▶ Change in B affects A
- ▶ Reusing A, B also needed



Direct vs Indirect

► Dependencies in code

Word.h

```
#include "String.h"

class Word {
public:
    // some functions
private:
    // some functions
};
```

String.h

```
#include "Chararray.h"

class String {
public:
    // some functions
private:
    // some functions
};
```

Chararray.h

```
class Chararray {
public:
    // some functions
private:
    // some functions
};
```

Direct vs Indirect

- Dependencies in code

Word.h

```
#include "String.h"
```

```
class Word {  
    public:  
        // some functions  
    private:  
        // some functions  
};
```

String.h

```
#include "Chararray.h"
```

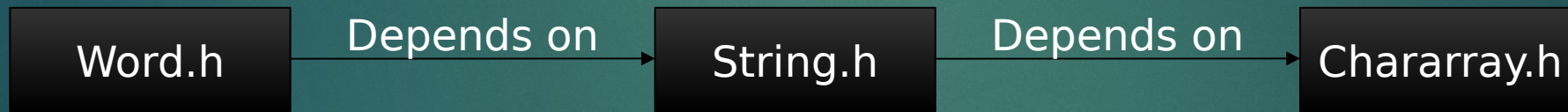
```
class String {  
    public:  
        // some functions  
    private:  
        // some functions  
};
```

Chararray.h

```
class Chararray {  
    public:  
        // some functions  
    private:  
        // some functions  
};
```

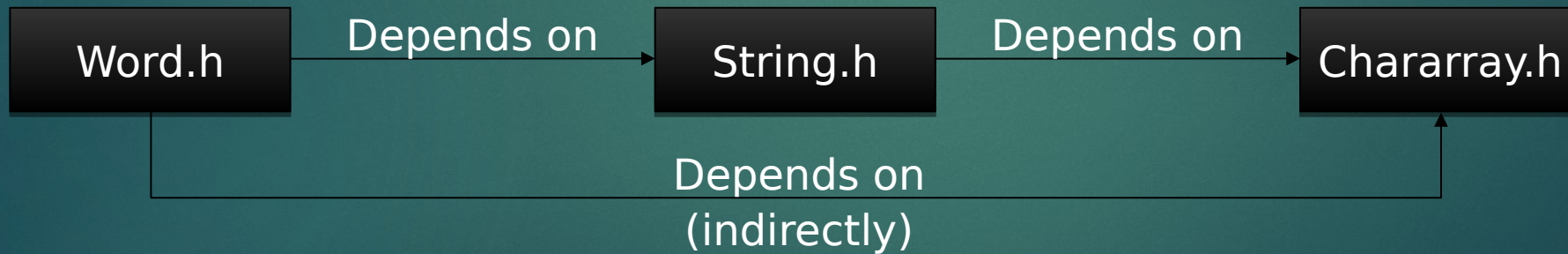

Direct vs Indirect

► Direct



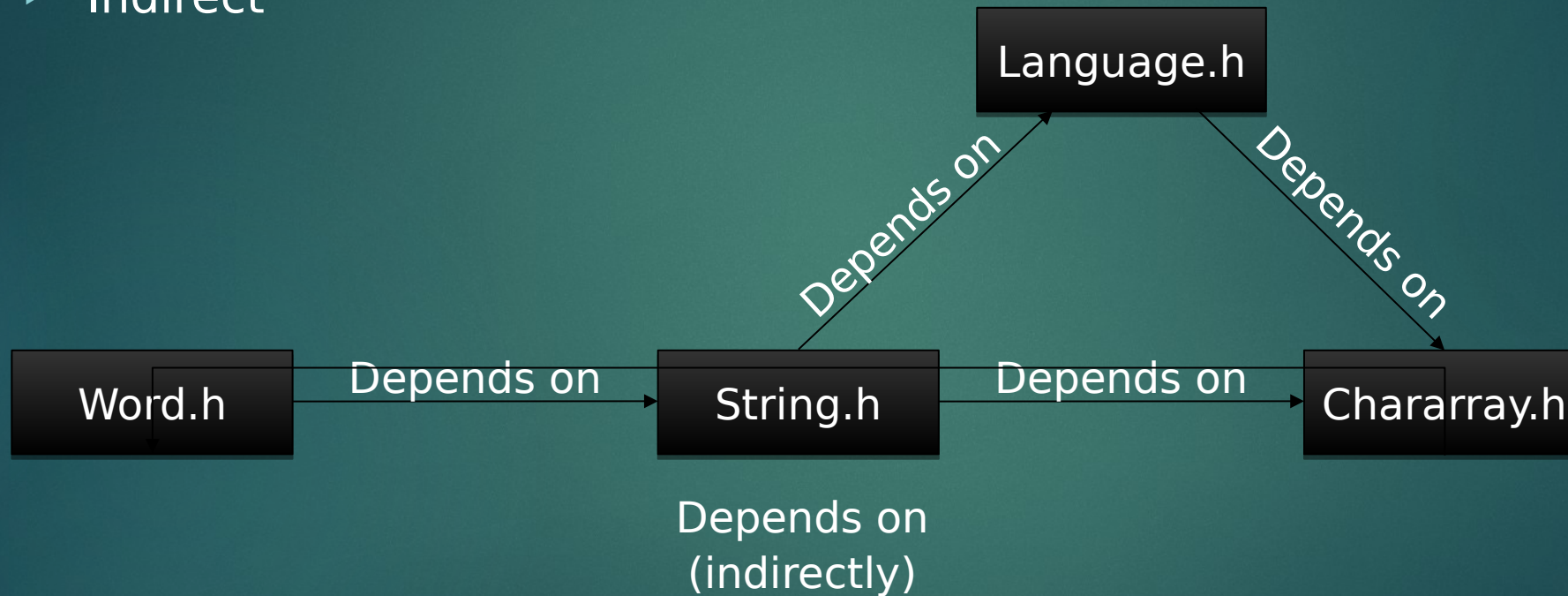
Direct vs Indirect

► Indirect



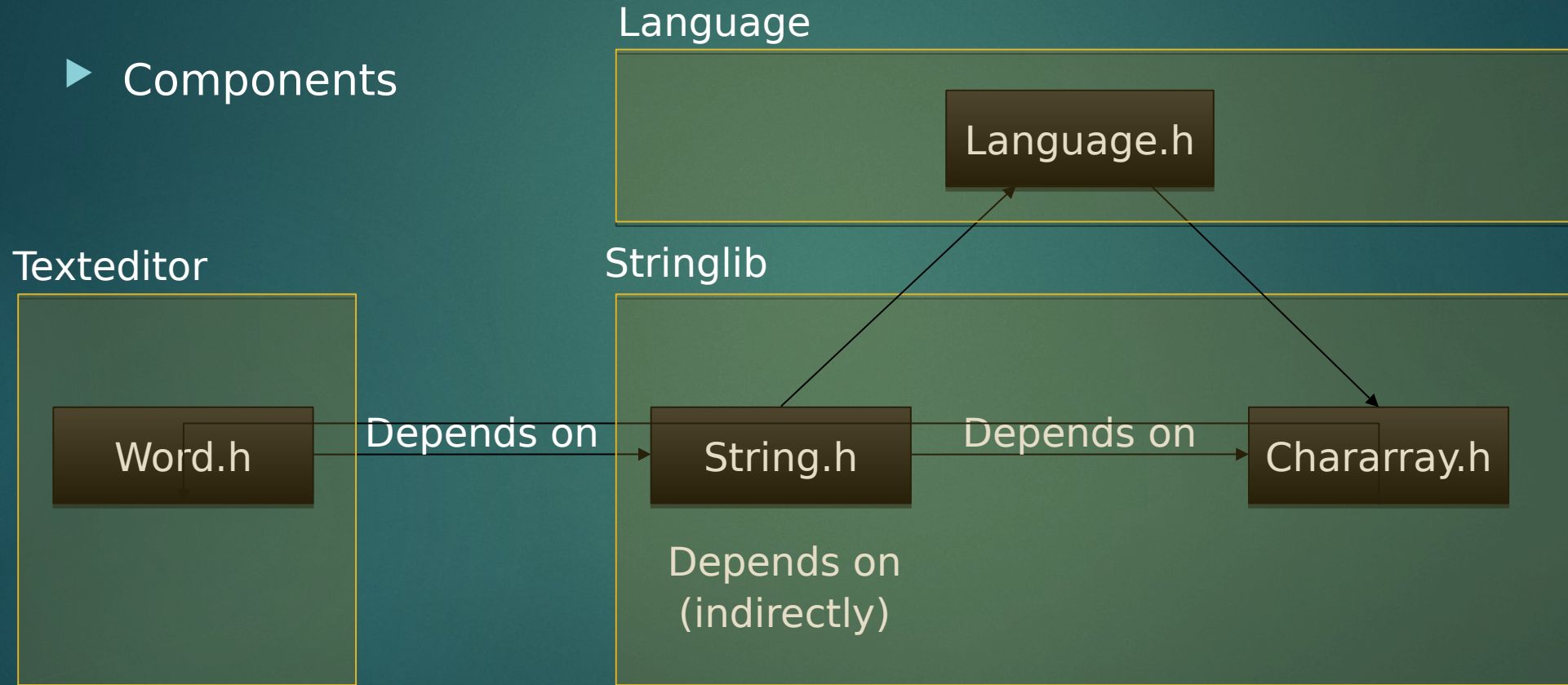
Direct vs Indirect

► Indirect



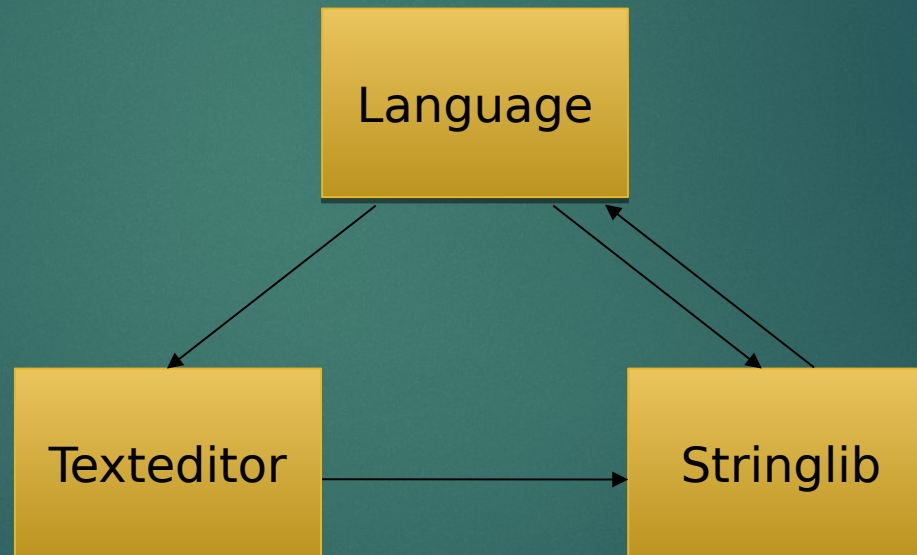
Direct vs Indirect

► Components



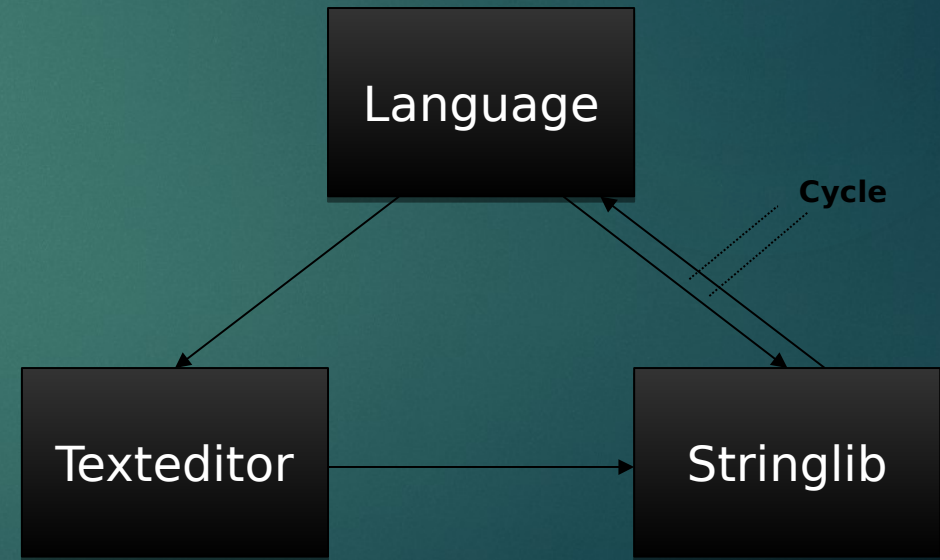
Direct vs Indirect

- Components



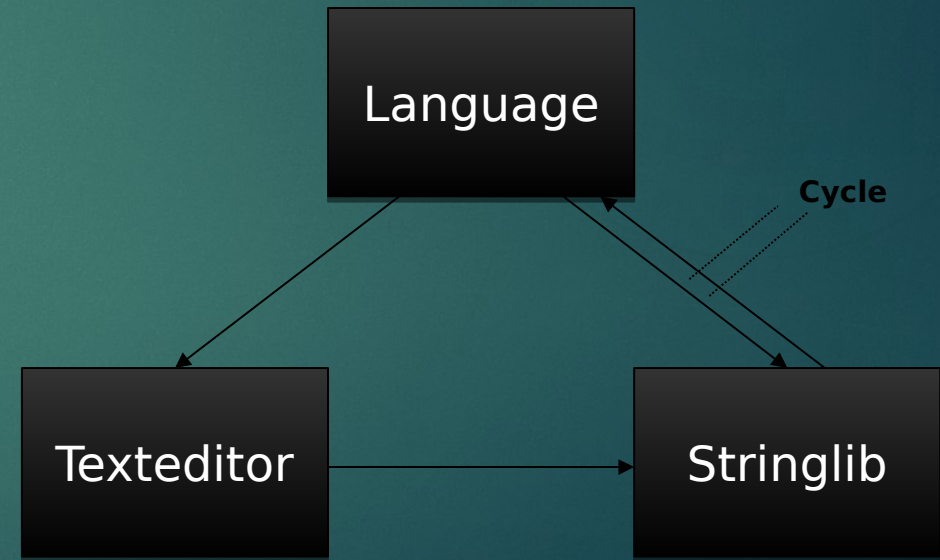
Cyclic vs Acyclic

- ▶ Cyclic
 - ▶ Components are dependent on each other



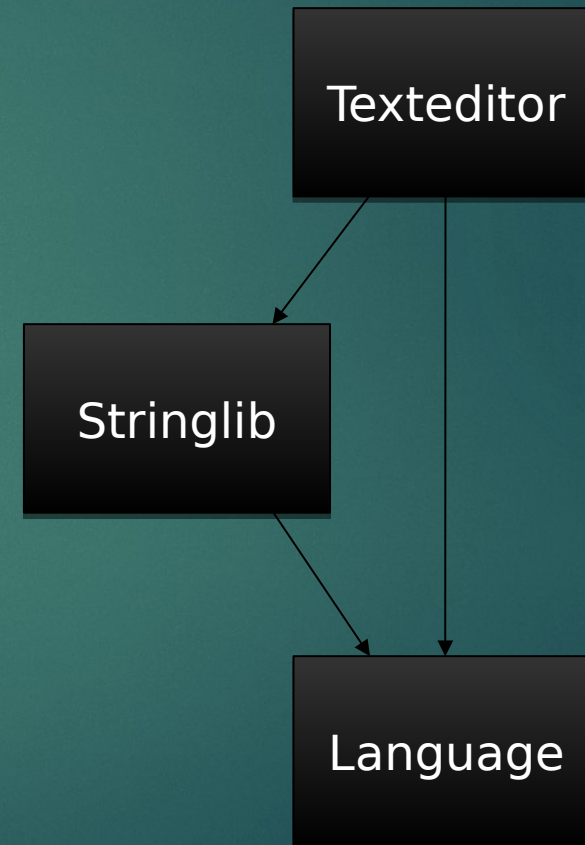
Cyclic vs Acyclic

- ▶ Cyclic
 - ▶ Limits individual testing
 - ▶ Cognitive load
 - ▶ Increased complexity



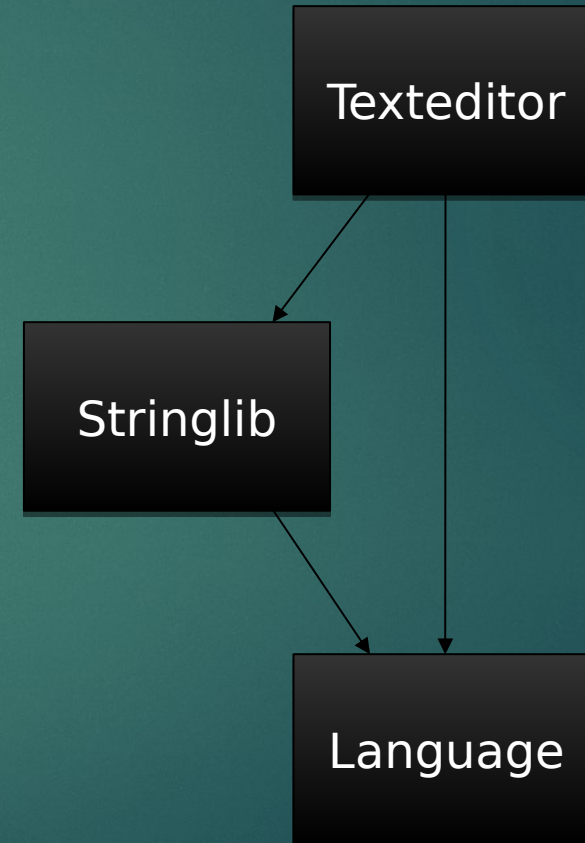
Cyclic vs Acyclic

- ▶ Acyclic
 - ▶ Individual testing possible
 - ▶ Decreases cognitive load
 - ▶ Decreases complexity



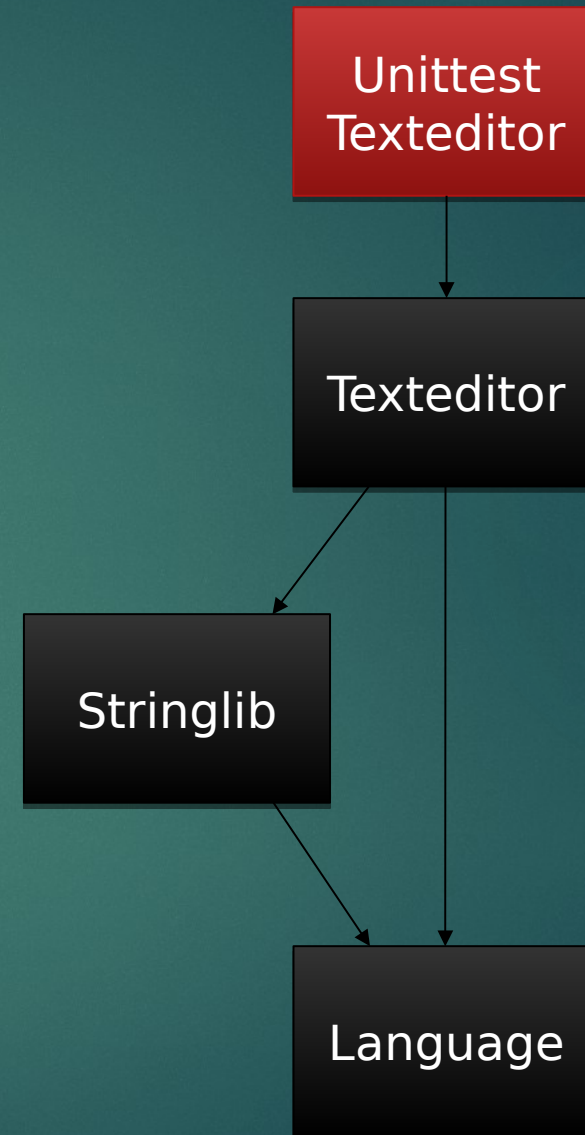
Transitive closure

- ▶ List of your dependencies
- ▶ Reachability



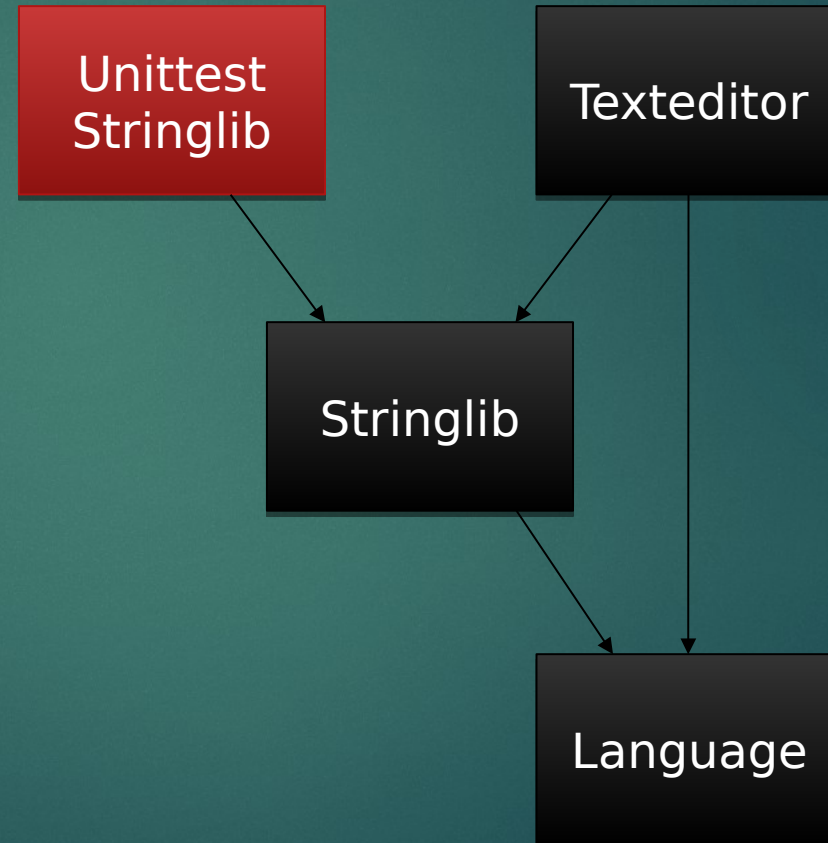
Transitive closure

- ▶ Texteditor
- ▶ Stringlib
- ▶ Language



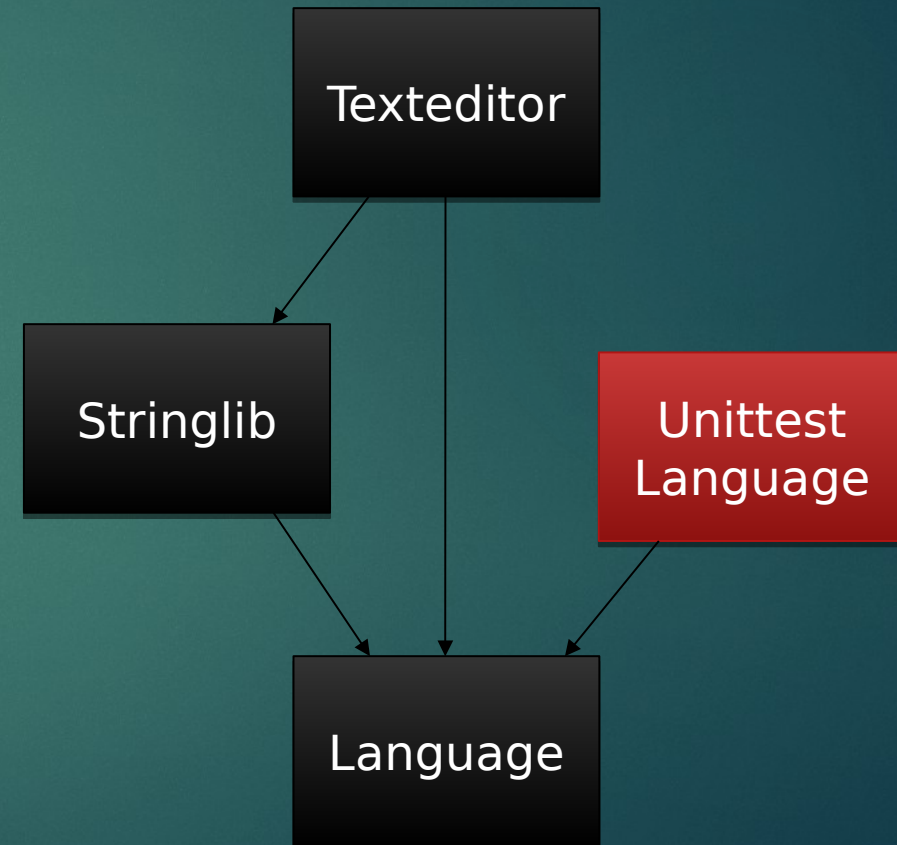
Transitive closure

- ▶ Stringlib
- ▶ Language



Transitive closure

- Language



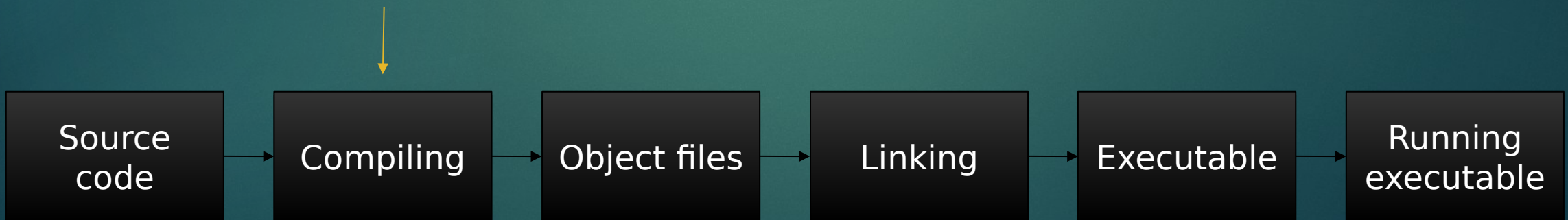
Dependency types

- ▶ Compile time
- ▶ Link time
- ▶ Run time



Dependency types

- ▶ Compile time dependencies
 - ▶ Creation of object files
 - ▶ Declaration of symbols

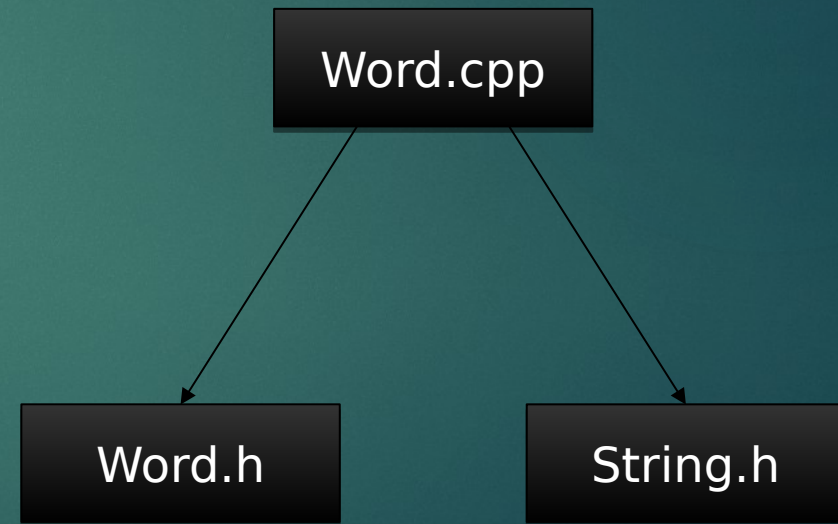


Dependency types

- ▶ Compile time dependencies
 - ▶ Private
 - ▶ Public

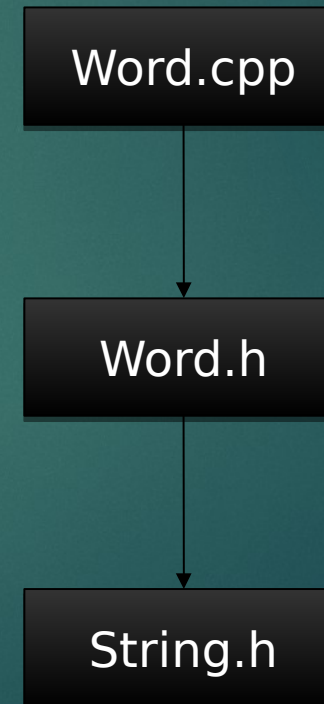
Dependency types

- ▶ Private compile time dependency

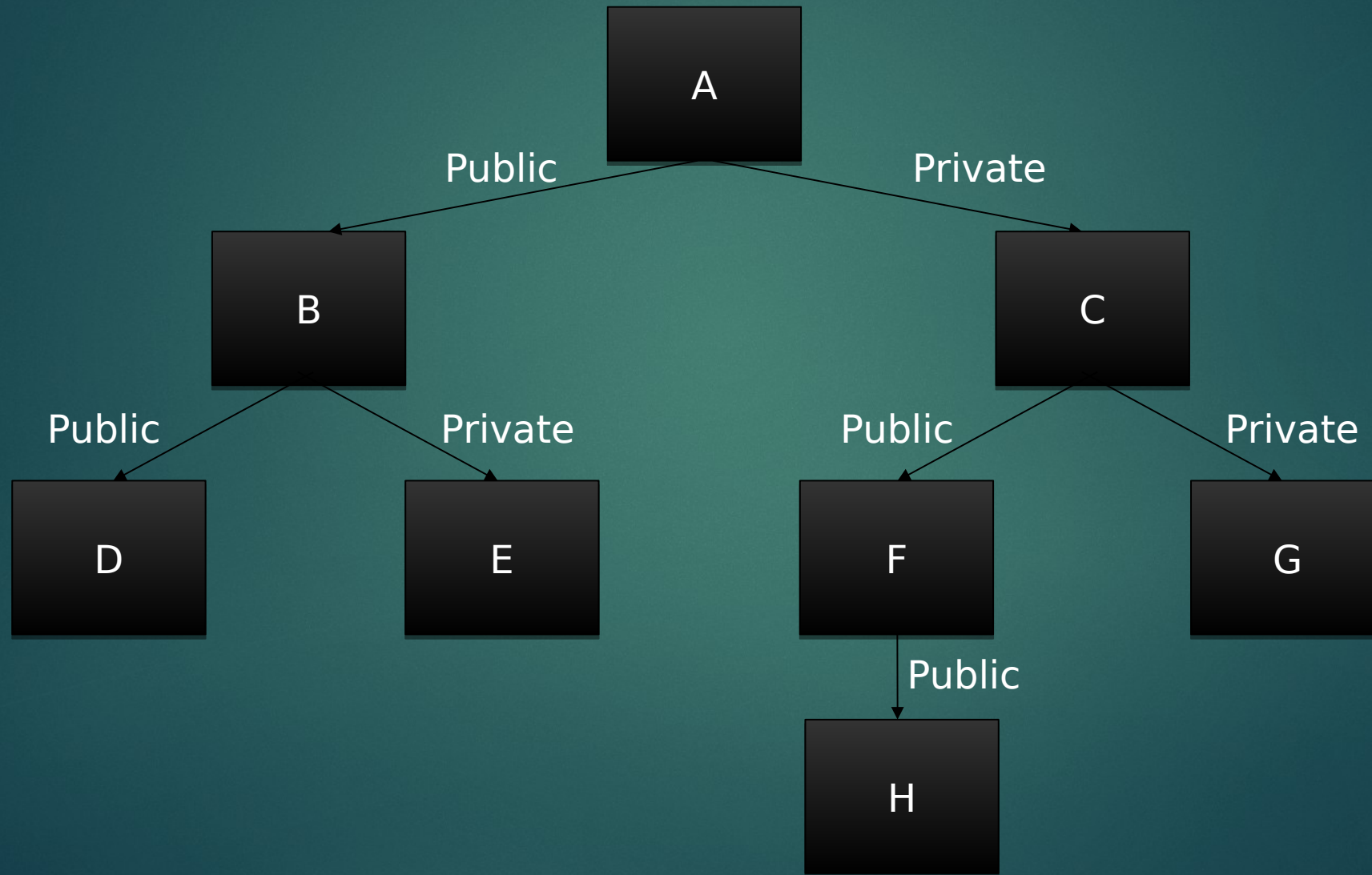


Dependency types

- ▶ Public compile time dependency



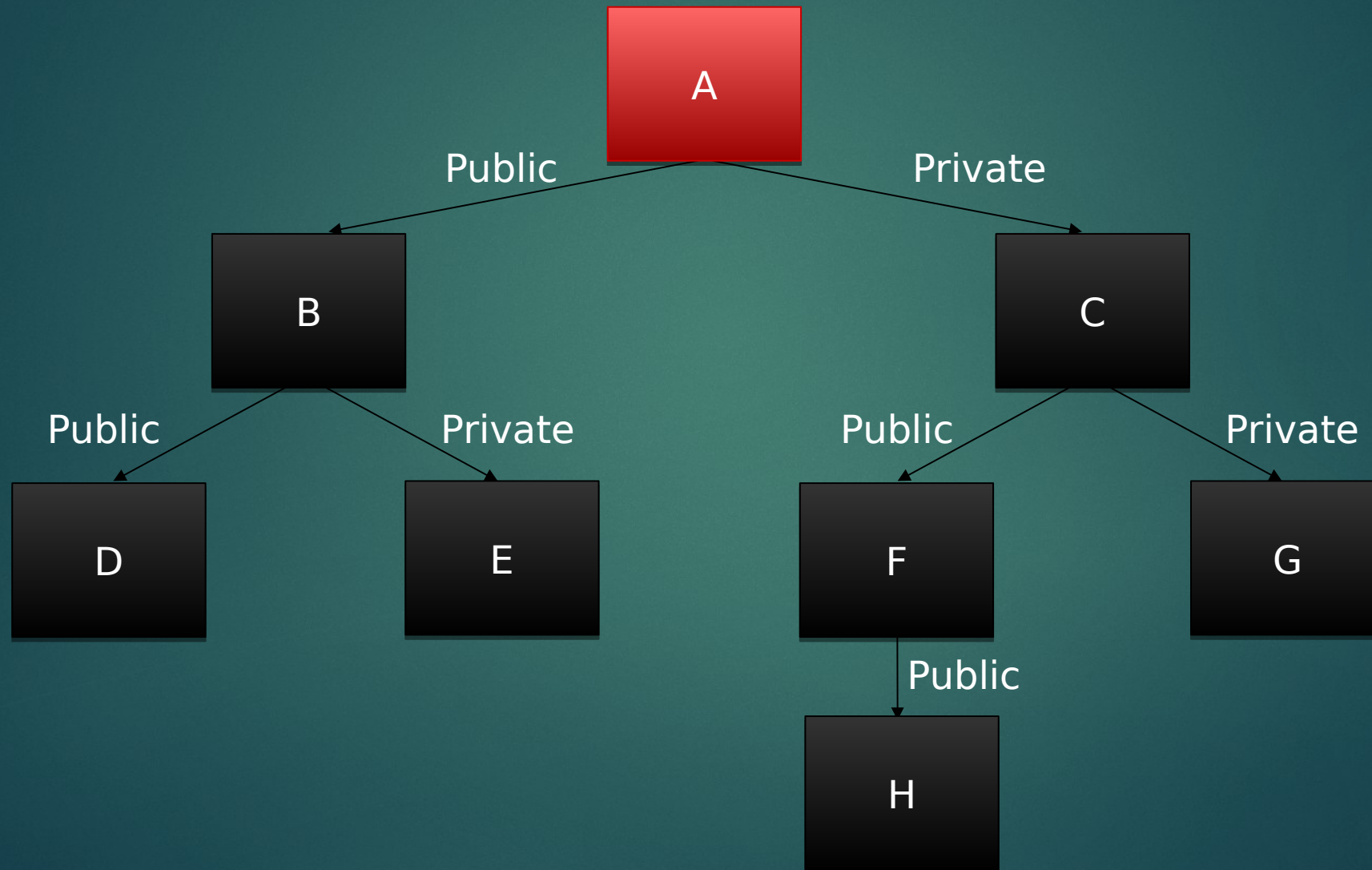
Dependency types



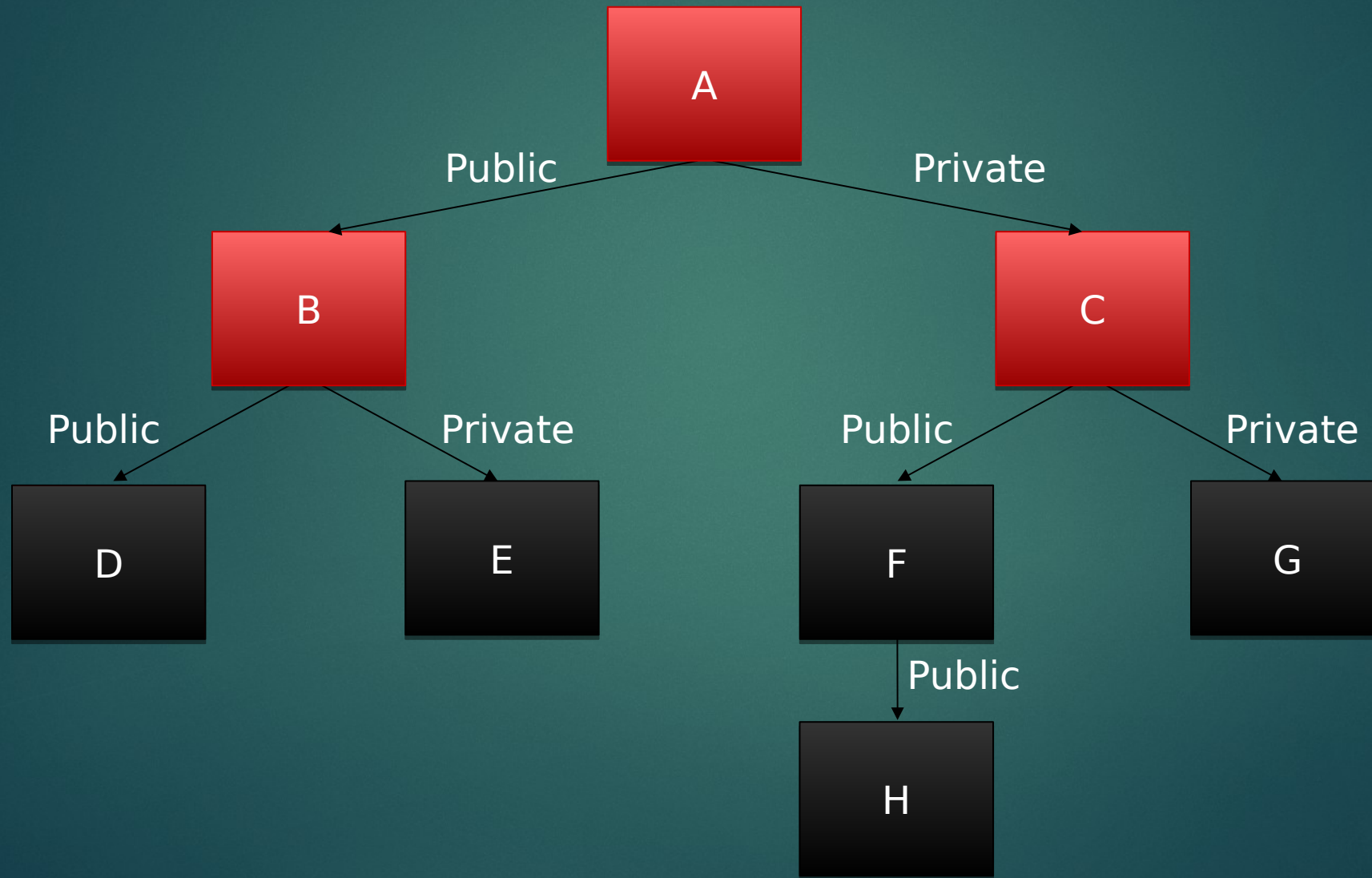
Dependency types

Compile-time include path of A

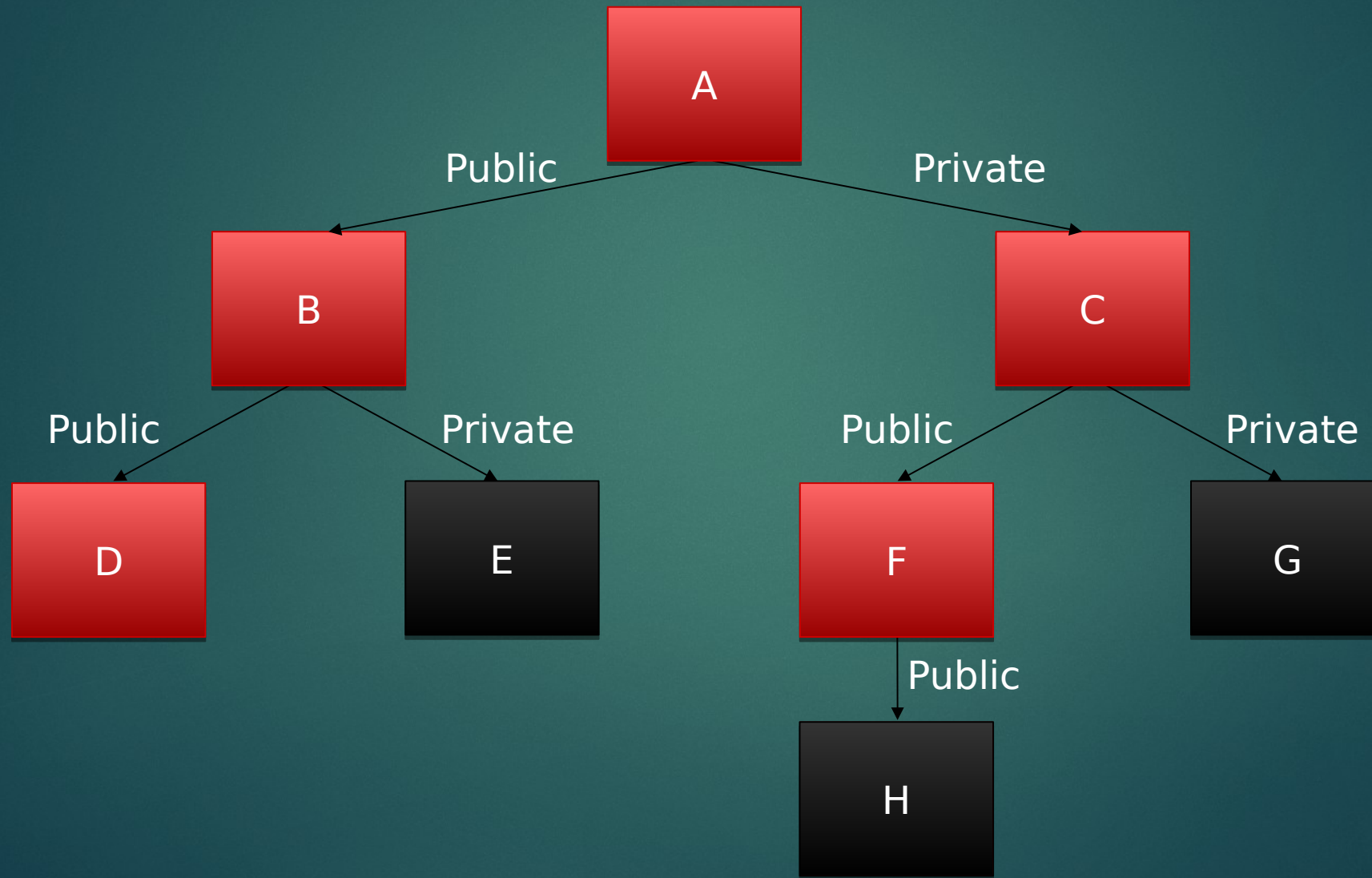
Dependency types



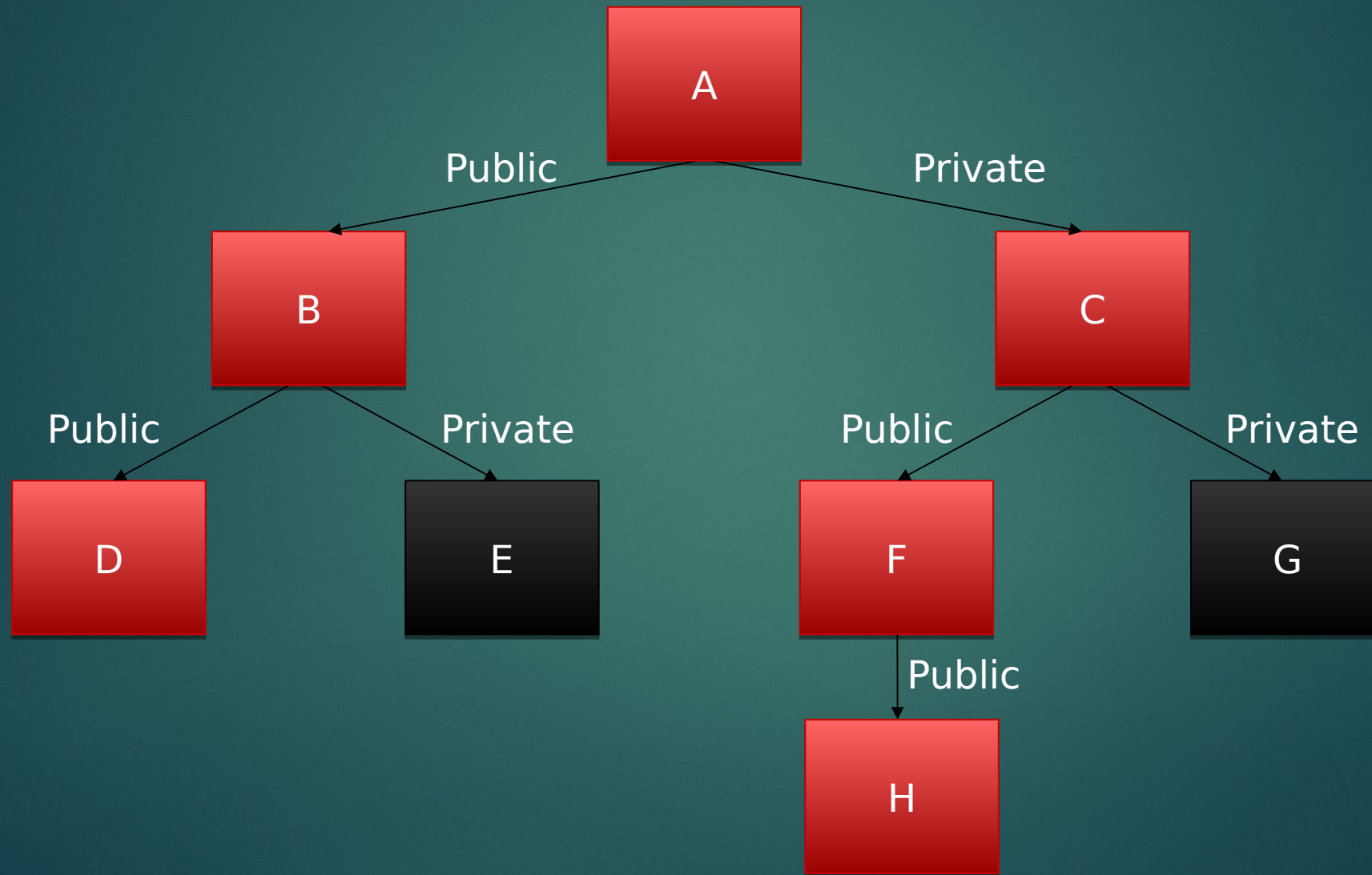
Dependency types



Dependency types



Dependency types

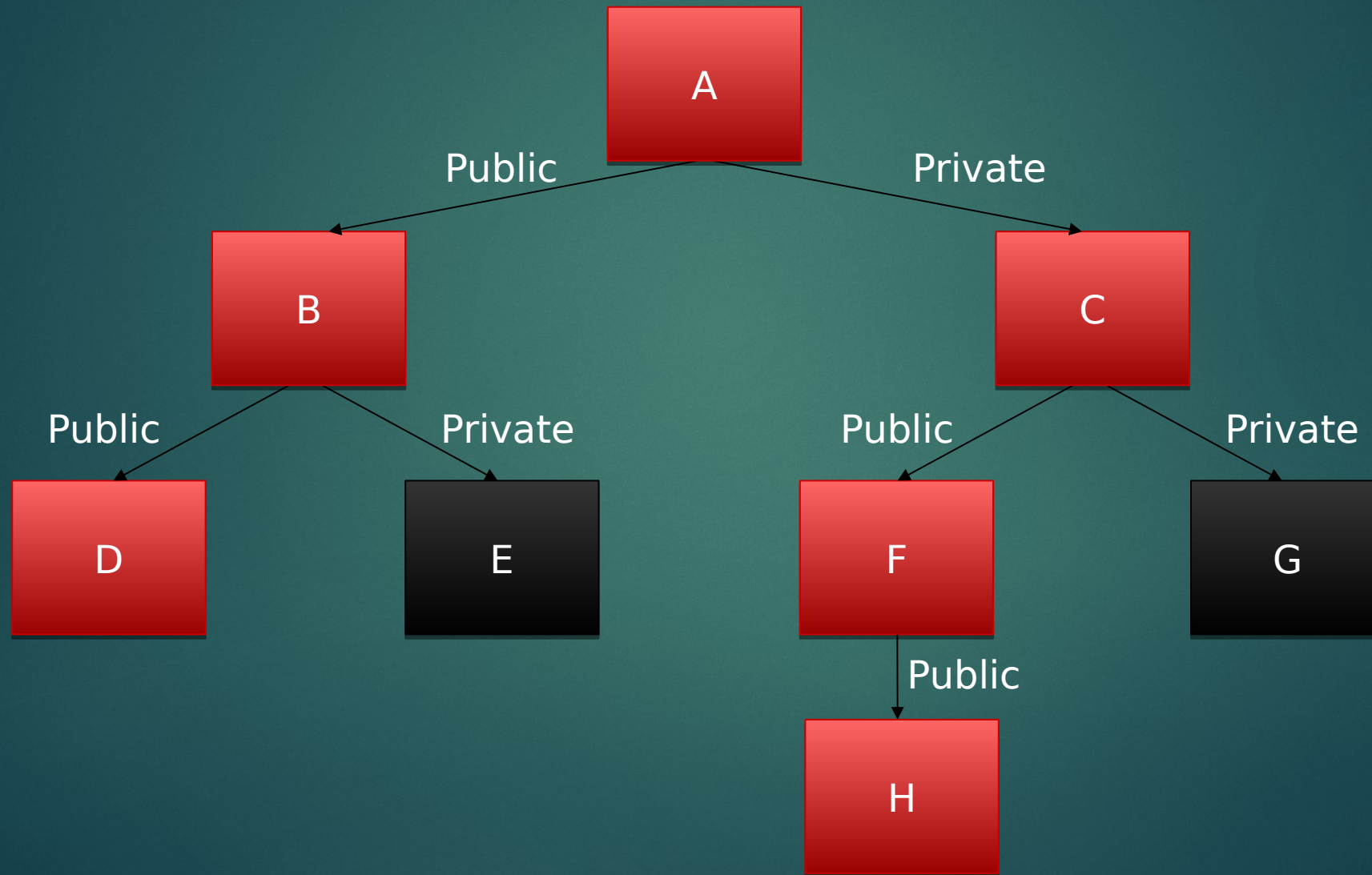


Dependency types

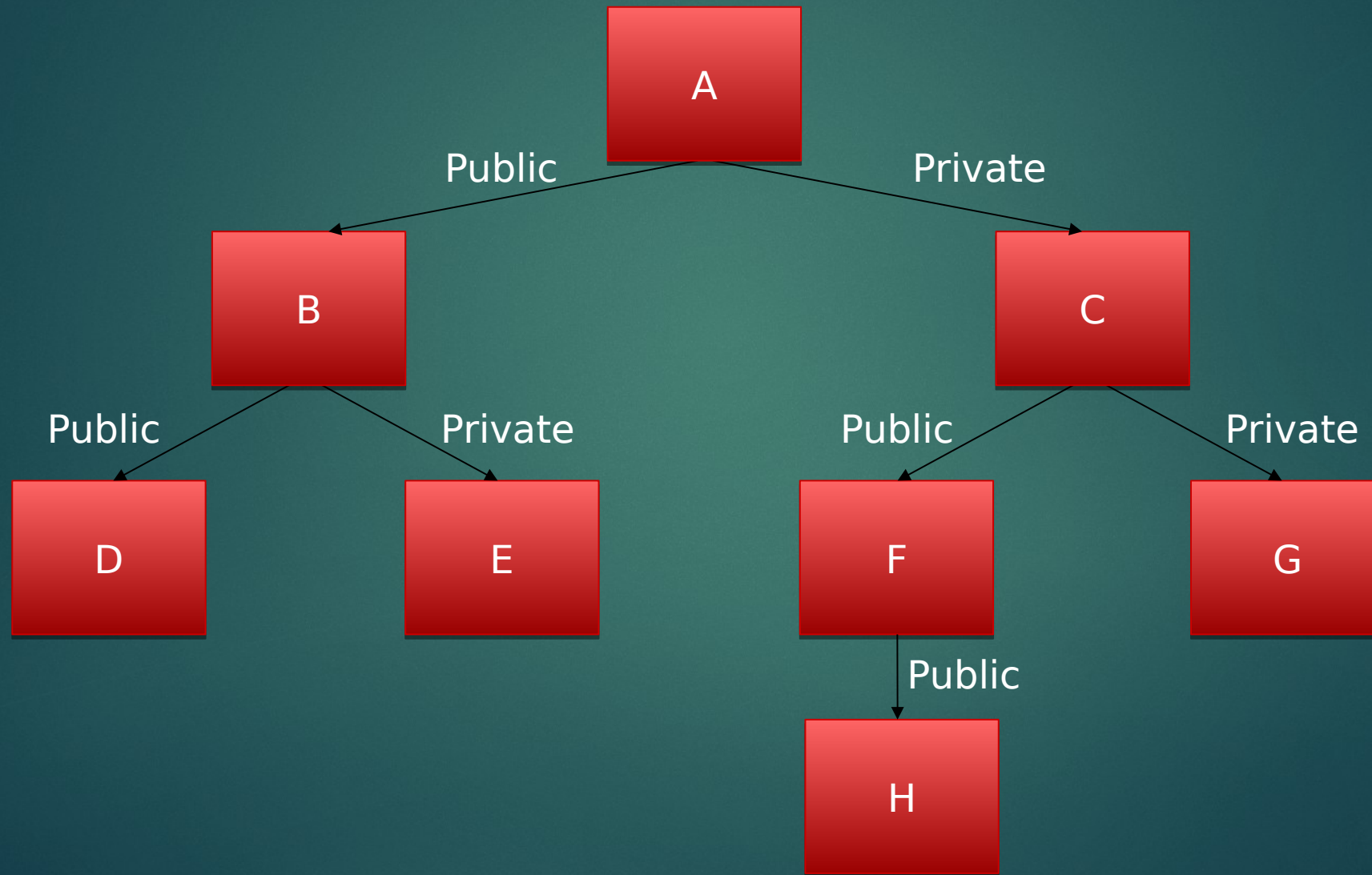


Link-time

Dependency types

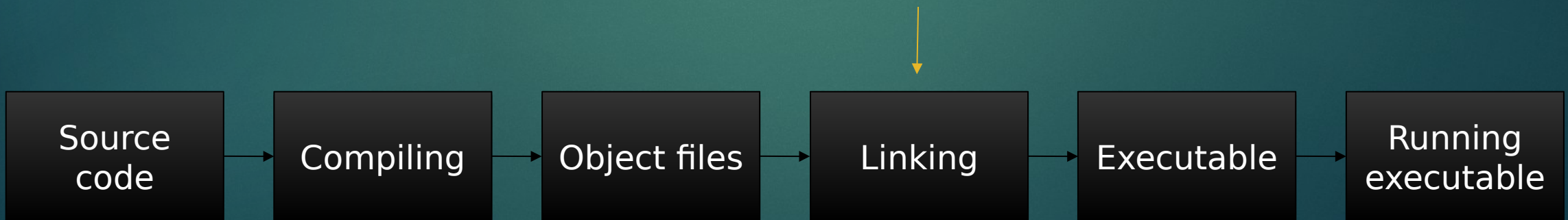


Dependency types



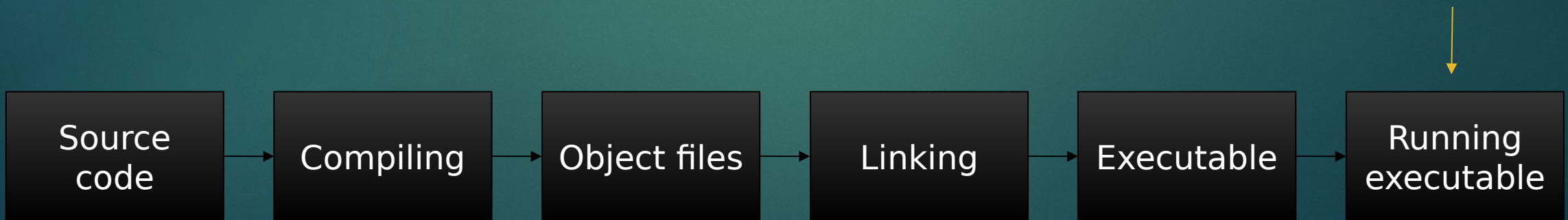
Dependency types

- ▶ Link time dependencies
 - ▶ Creation of executable
 - ▶ Definition of symbols



Dependency types

- ▶ Run time dependencies
 - ▶ Depending on other program
 - ▶ Depending on external interface



Dependency types

- ▶ Run time dependencies



SOLID



Single responsibility principle

Open/closed principle

Liskov substitution principle

Interface segregation

Dependency inversion

SOLID

SOLID



- Guidelines for making good software
 - Think “design smells”
- Good software has few smells, bad software has many

SOLID

SOLID



Single responsibility principle

Open/closed principle

Liskov substitution principle

Interface segregation

Dependency inversion

SOLID

Single responsibility principle

- Do one thing and do it well
 - Unix' base principle



Single responsibility principle

- Users for one of your responsibility do not get your indirect dependencies for your other responsibilities



Interface segregation

- Each interface has functions for one kind of user
- If you expose functions for multiple users, this is multiple interfaces
- Each interface is self-consistent



Interface segregation

- Users for one of your interfaces do not get your indirect dependencies for your other responsibilities

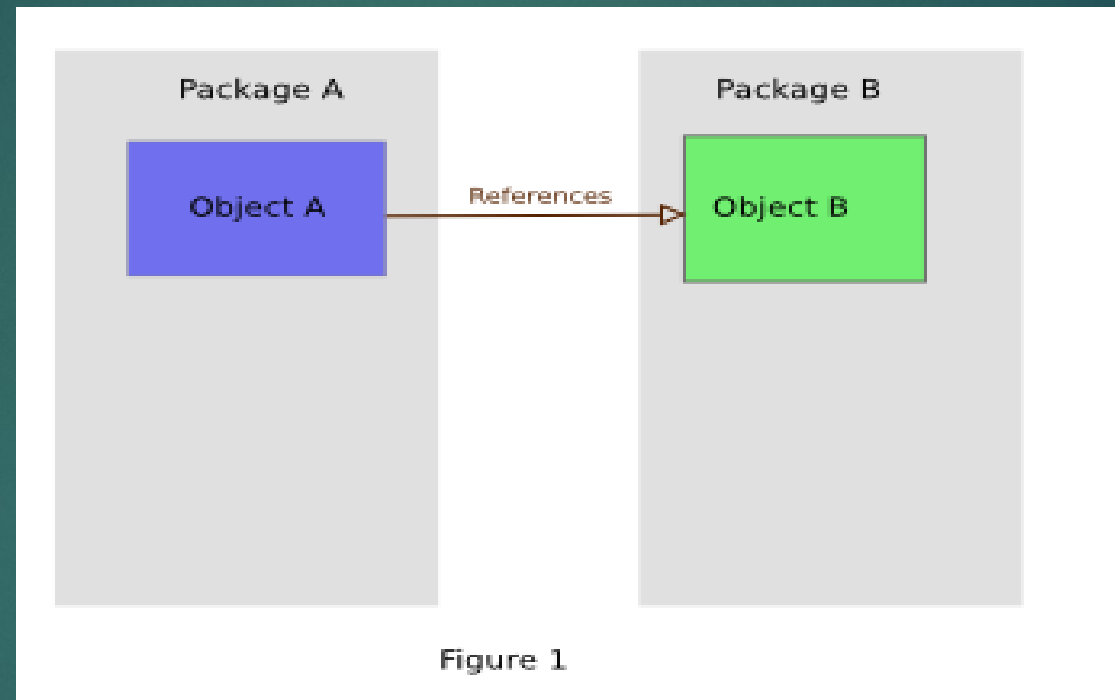
Dependency inversion

- Depend on abstractions (interfaces) instead of implementations

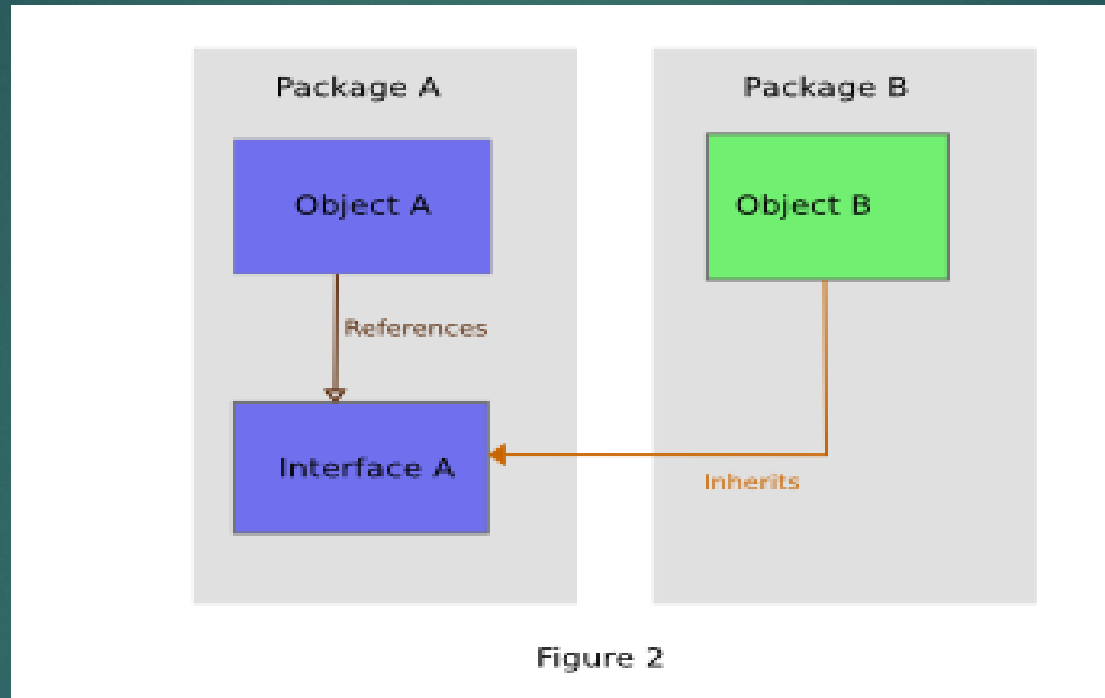
Dependency inversion

- Depend on abstractions (interfaces) instead of implementations
- We like it because it allows us to flip a dependency

Dependency inversion



Dependency inversion




```
class Settings {
    void setVolume(int newVolume) {
        volume = newVolume;
        AudioSystem::NotifyVolumeChange();
    }
    void setAcceleration(int newAcceleration) {
        acceleration = newAcceleration;
        Car::NotifyAccelerationChange();
    }
    int getVolume() {
        return volume;
    }
    int volume;
};


class AudioSystem {
    void NotifyVolumeChange() {
        Hardware::SetVolume(settings.getVolume());
    }
};
```



SOLID

```
class Settings {
    class VolumeListener {
        virtual void NotifyVolumeChange() = 0;
    };
    VolumeListener* volumeListener;
    void setVolumeListener(VolumeListener* listener) {
        volumeListener = listener;
    }
    void setVolume(int newVolume) {
        volume = newVolume;
        volumeListener->NotifyVolumeChange();
    }
    int getVolume() {
        return volume;
    }
    int volume;
    // Analogous for Car, details omitted
};
```

SOLID



```
class AudioSystem : VolumeListener {  
    AudioSystem(Settings& settings) {  
        settings->SetVolumeListener(this);  
    }  
    void NotifyVolumeChange() {  
        Hardware::SetVolume(settings.getVolume());  
    }  
};
```

SOLID

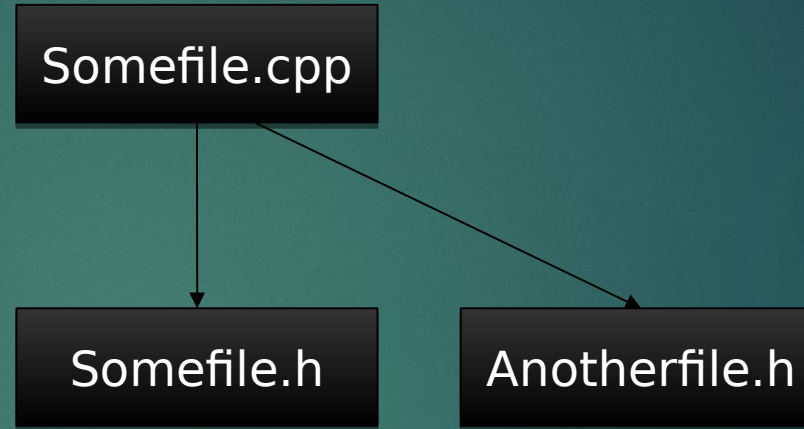
Upsides

- ▶ Ties program together
- ▶ Makes reuse possible

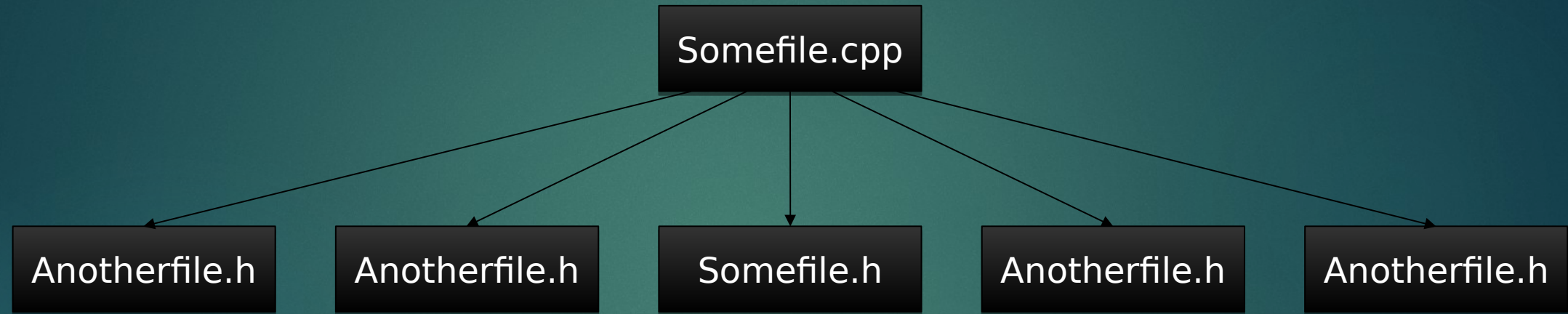
Downsides

- ▶ Slower builds
 - ▶ Preprocessing
 - ▶ Extended source files

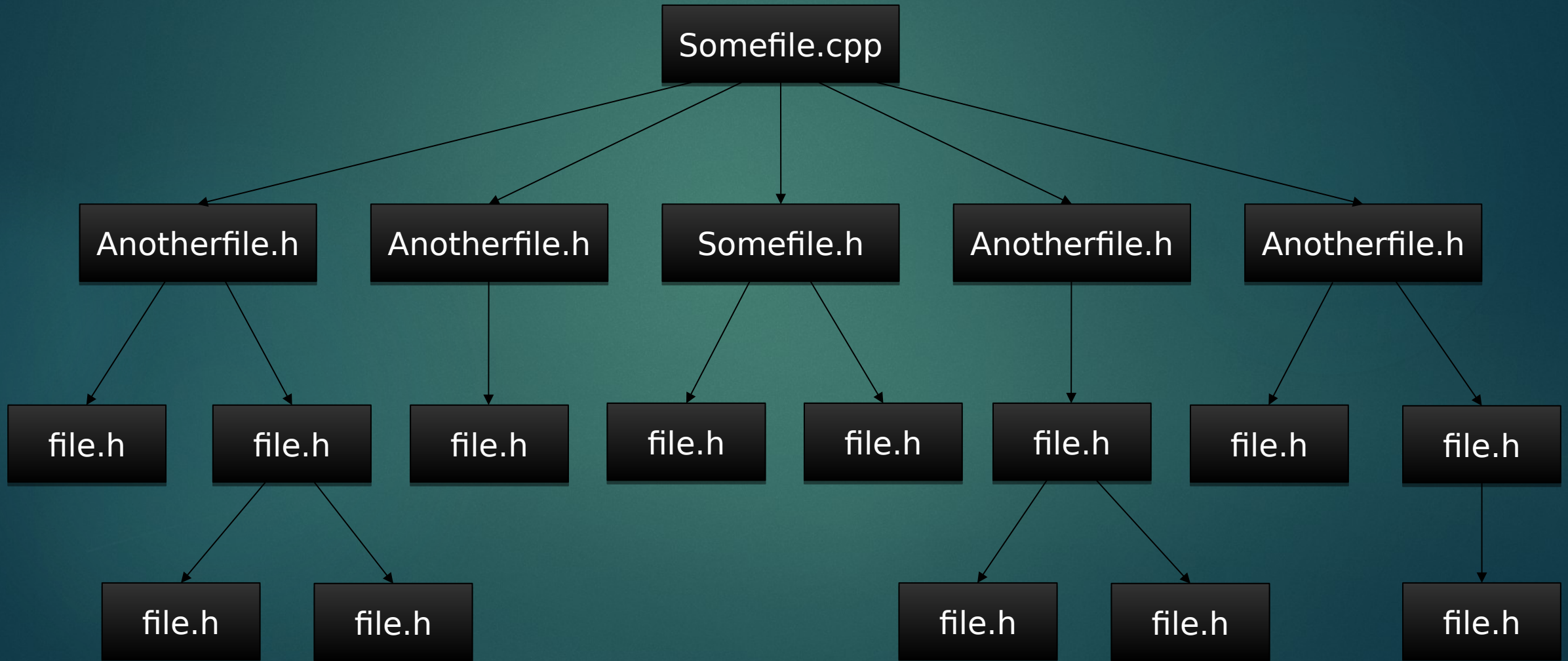
Downsides



Downsides



Downsides

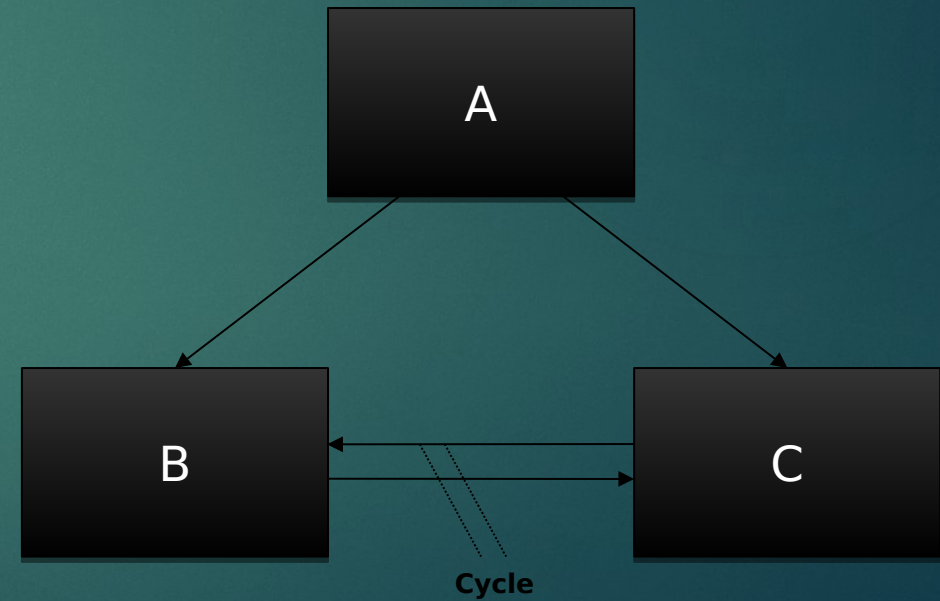


Downsides

- ▶ Recompiling by changes
 - ▶ All dependent files
 - ▶ Private members

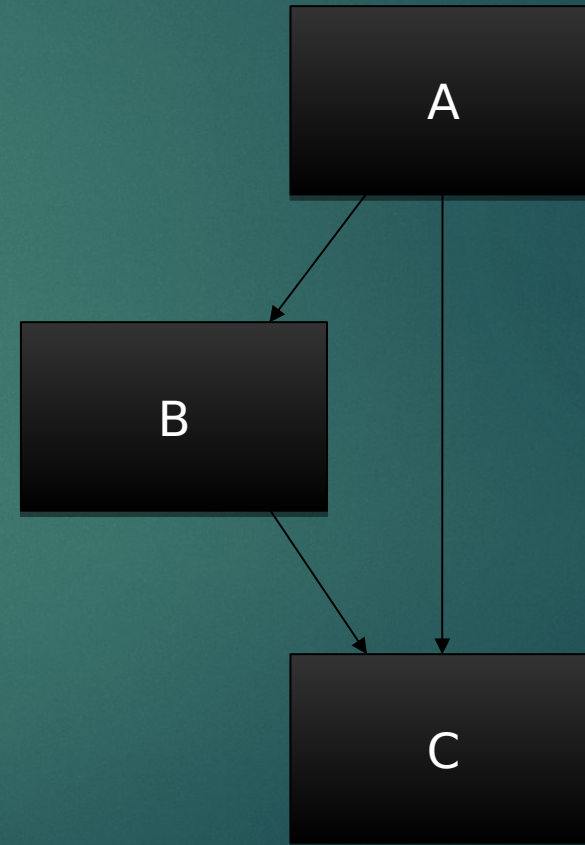
Downsides

- ▶ Testability
 - ▶ Individual testing



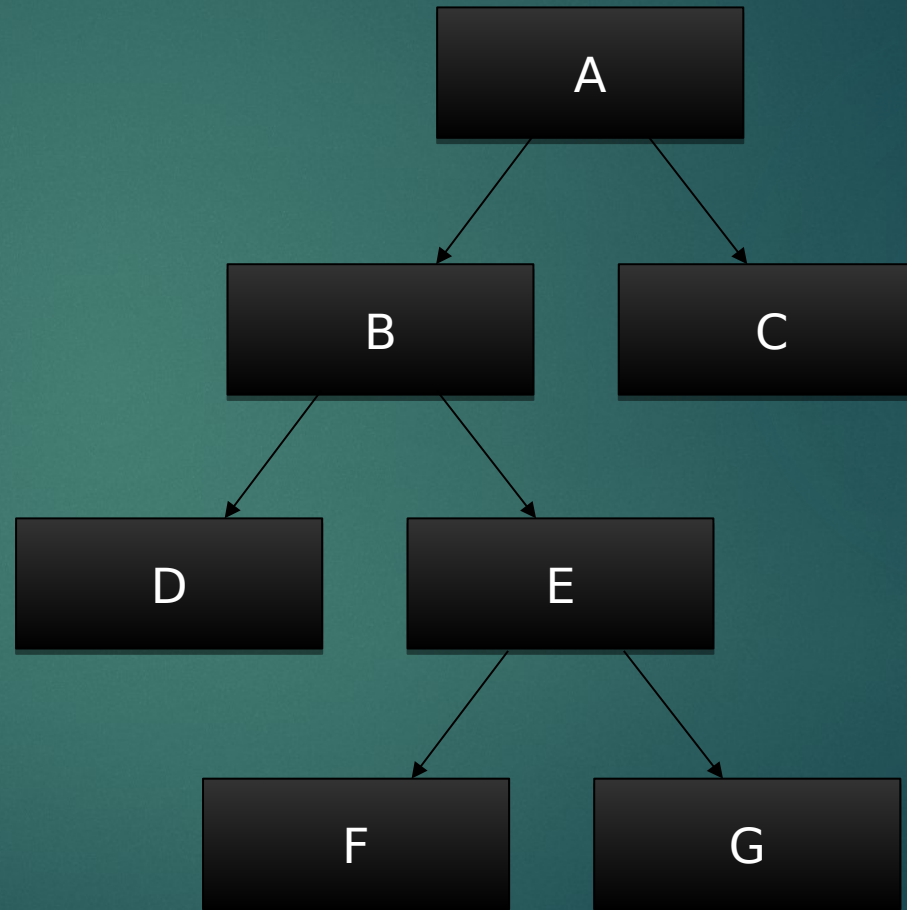
Downsides

- ▶ Testability
 - ▶ Test in certain order



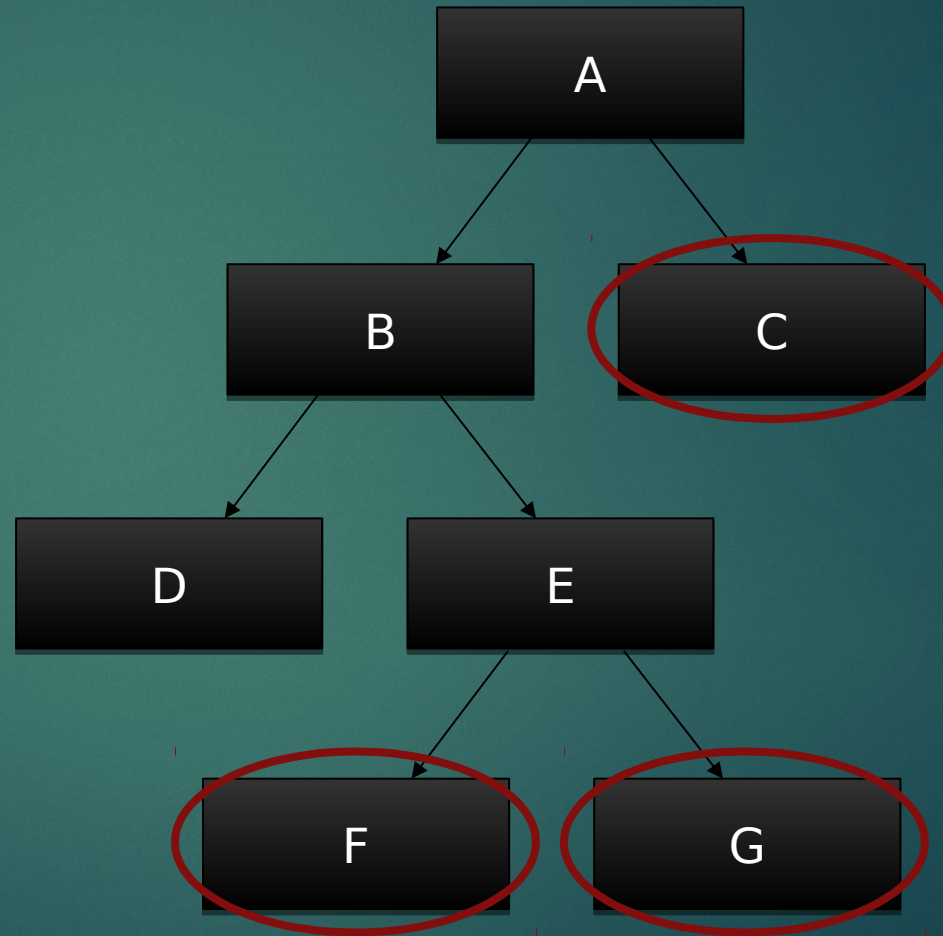
Downsides

- ▶ Reusability



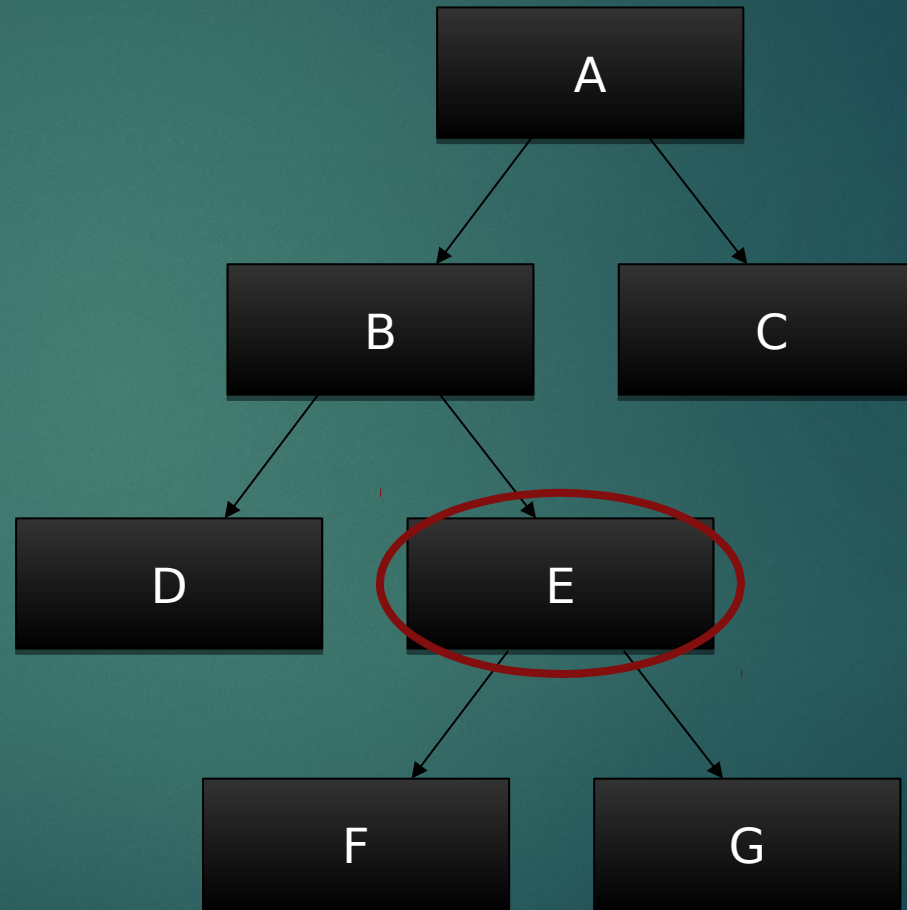
Downsides

- Reusability



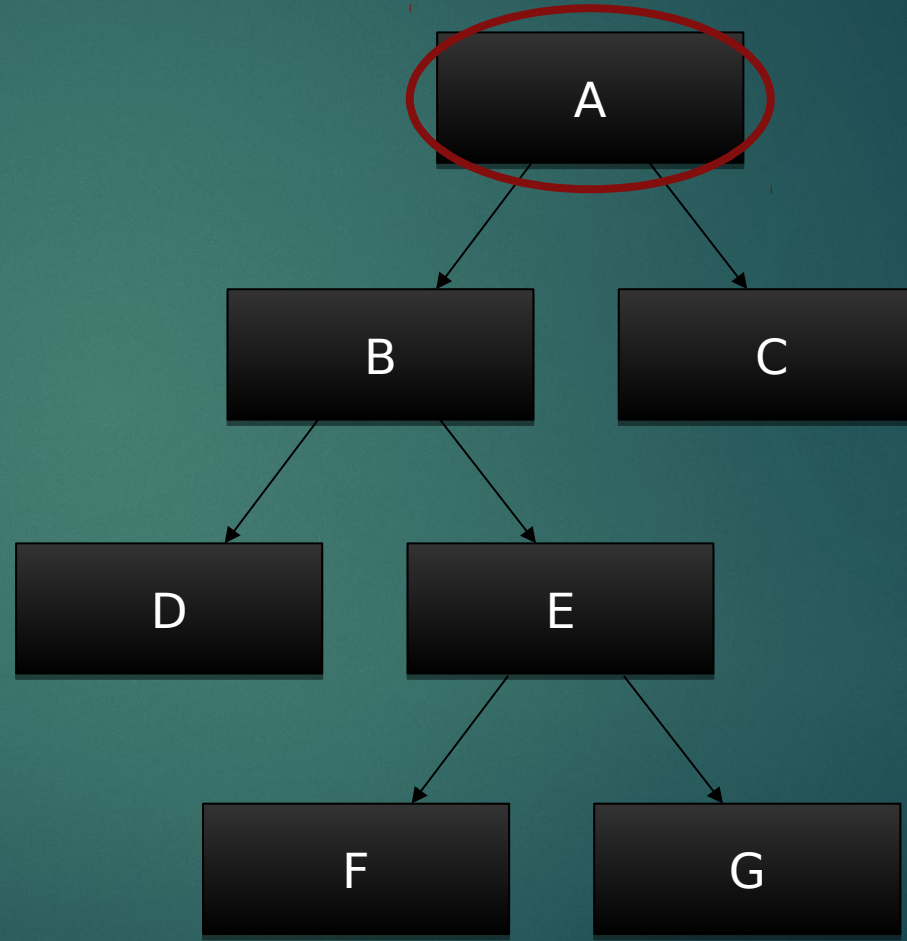
Downsides

- ▶ Reusability



Downsides

- ▶ Reusability





Modules

Modules

- ▶ N4047: [Modules] A Module System for C++ (by G. Dos Reis, M. Hall, G. Nishanov) (2014-05-27) (Related: EWG163)
- ▶ N4214: [Modules] A Module System for C++ (Revision 2) (by G. Dos Reis, M. Hall, G. Nishanov) (2014-10-13) (Related: EWG163)
- ▶ N4465: [Evolution] A Module System for C++ (Revision 3) (by Gabriel Dos Reis, Mark Hall, Gor Nishanov) (2015-04-13) (Related: EWG163)
- ▶ P0142R0: [WG21] A Module System for C++ (Revision 4) (by Gabriel Dos Reis) (2016-02-15)

Modules

- ▶ P0841R0: [Evolution] Modules at scale (by Bruno Cardoso Lopes, Adrian Prantl, Duncan P. N. Exon Smith) (2017-10-16)
- ▶ P0832R0: [Evolution, Core] **Module TS Does Not Support Intended Use Case** (by David Sankel) (2017-10-14)
- ▶ P0804R0: [Evolution] Impact of the Modules TS on the C++ tools ecosystem (by Tom Honermann) (2017-10-15)
- ▶ P0795R0: [SG14, Evolution] From Vulkan with love: **a plea to reconsider the Module Keyword to be contextual** (by Simon Brand, Neil Henning, Michael Wong, Christopher Di Bella, Kenneth Benzie) (2017-10-16)

Modules

- ▶ Been in development for a reasonable time
- ▶ Sort of consensus, but not in implementations
- ▶ Goal is to make the preprocessor unnecessary most of the time

What's wrong with macros and #include ?

- ▶ Unparsed inclusion
- ▶ Textual inclusion
- ▶ You cannot usefully pre-parse a header file
- ▶ #define makes it impossible to read any code without knowing what came before
- ▶ It's not theoretically possible to make always-correct refactoring tools, code layout tools, ...
- ▶ Refactoring tools do exist!

Root problem

- ▶ `#include` does textual inclusion
- ▶ `#define` does textual substitution
- ▶ Your includes (dependencies) may modify things outside of their official reach

Root problem

- ▶ To get around this problem we want to have the ability to say
 - "This code is what it is, no more, no less"
- ▶ Very easy to get ODR violations this way
- ▶ Good componentization has components that are independent
- ▶ Most of the software we write actively does not want this

[P0142]

- ▶ a module unit should be immune to macros and any preprocessor directives in effect in the translation unit in which it is imported
- ▶ [preprocessor] ... neither its eradication nor improvements of it.
- ▶ [modules should] not come equipped with new sets of name lookup rules
- ▶ In an ideal world with modules, the usage of the time-honored header files should be rare, if not inexistent

Actual implementations

- ▶ Clang: Uses a `module_map` to "convert" a header into a module, module map used to compile. Not as per (P0142), but used in production.
- ▶ MSVC: Uses modules, import & export (as per P0142). Existed since 2015. In production (by MSVC itself).
- ▶ GCC: Same as MSVC, existed since March 2017. Not production ready.

Dependency glasses

- ▶ **All dependencies we had, we still have**
- ▶ The compiler, post-parsing, must do exactly what it otherwise also had to do
- ▶ Modules enable faster template processing, but do not make it happen by itself
- ▶ Moving code generation later (around link time) makes this possible
- ▶ Still not everything avoidable, because you cannot read some code without looking up types, and thereby instantiating your templates

How to reduce dependencies

- Remove stale / unused dependencies
- Convert public dependencies to private
- Invert (some) dependencies

Remove stale / unused dependencies

- If you actually don't use an include / import in a given file, just remove it.
- Harder than it looks
 - Includes may result in subsequent errors
 - Build system may rely on incorrect dependencies


```
// A.h  
#include "B.h"
```

```
class A {  
    C c;  
};
```

```
// B.h  
#include "C.h"
```

```
class B {  
    // nothing relevant  
}
```

```
// C.h
```

```
class C {  
    // nothing relevant  
}
```

```
// A.h
// #include "B.h"
```

```
class A {
    C c;
};
```

```
// B.h
#include "C.h"
```

```
class B {
    // nothing relevant
}
```

```
// C.h
```

```
class C {
    // nothing relevant
}
```

SOLID

```
// A.h
#include "B.h"
```

```
class A {
    C c;
};
```

```
// B.h
```

```
//#include "C.h"
```

```
class B {
    // nothing relevant
}
```

```
// C.h
```

```
class C {
    // nothing relevant
}
```

SOLID


```
// A.h
#include "B.h"
#include "C.h"

class A {
    C c;
};
```

```
// B.h
//#include "C.h"

class B {
    // nothing relevant
}
```

```
// C.h

class C {
    // nothing relevant
}
```

```
// A.h
//#include "B.h"
#include "C.h"

class A {
    C c;
};
```

```
// B.h
//#include "C.h"

class B {
    // nothing relevant
}
```

```
// C.h

class C {
    // nothing relevant
}
```

Remove stale / unused dependencies

- In implementation files
 - Incomplete header files
 - Behavior change because of globals
- In headers
 - Include someone else relies on indirectly

Remove stale / unused dependencies

- In build system
 - Incomplete dependency trees – much like includes
 - Library link order changes

Remove stale / unused dependencies

- Any dependency tree works, as long as the transitive closure has all the required libraries in order.
 - Corollary: Nobody fixes it until it's too broken to fix
- Symptoms of broken dependency trees
 - `#include` statements with relative paths
 - `#include` statements not from component root
 - Random build failures when removing incorrect dependencies

Convert public dependencies to private

- Stop including headers in headers
 - Use forward declarations instead of includes
 - Use pImpl pattern
 - Use dependency injection

Forward declarations

- You need a full type when
 - You use the size of the object
 - You use a member of the object
 - You instantiate some templates (but not all)
- You do not need a full type otherwise
 - Pointers, references, most common templates

plmpl pattern

- Used to split a public dependency chain by making one link private
 - Create interface for your class
 - Have an opaque pointer to another class
 - Implement it, and that class, in a CPP file

Dependency Injection

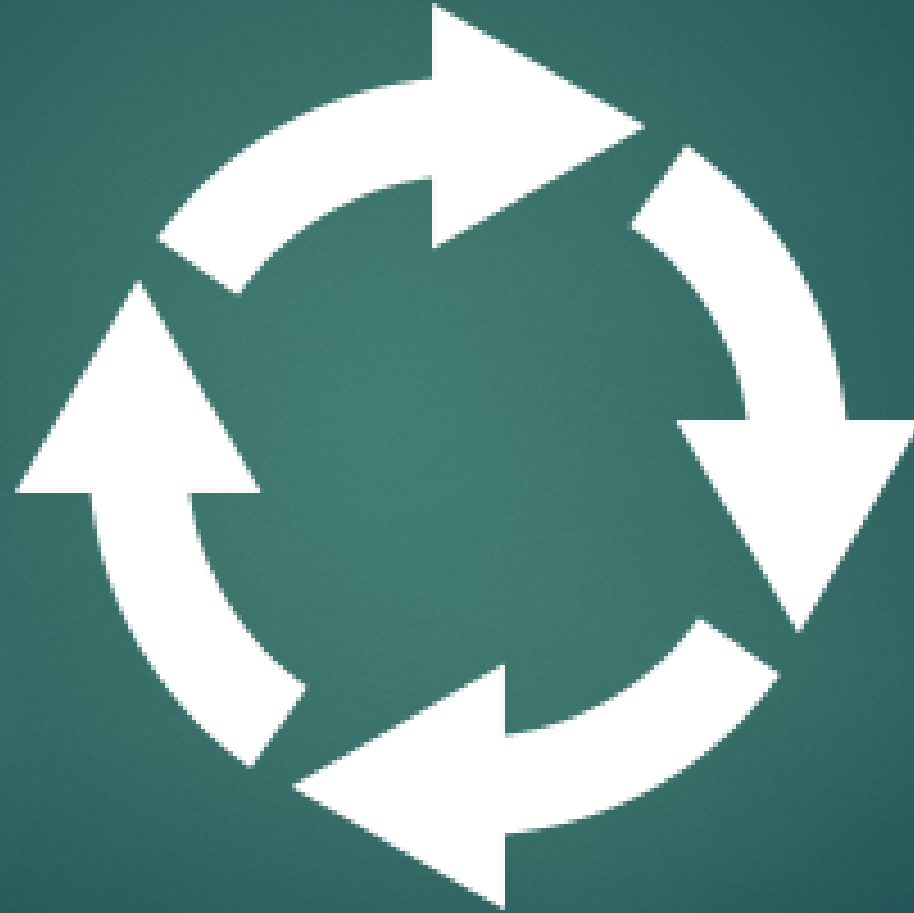
- Convert a compile-time dependency to a link-time dependency
 - More componentized code base
 - Many more interfaces
 - Explicit use of dependencies
 - May be hard to trace
 - Does not need libraries to work

Invert (some) dependencies



- Code often leads to natural cycles
 - Publish-subscribe
 - Register/callback
 - Graphs

Cyclic dependencies



Why fix cyclic dependencies

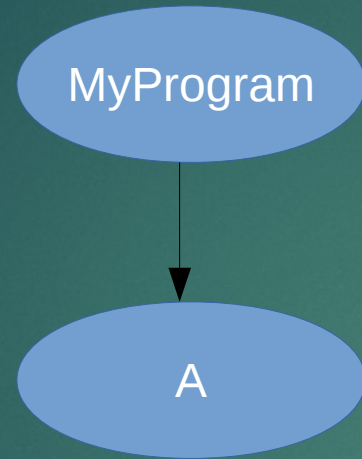
- Cycles contribute the most to your dependencies
 - Any dependency to anything in a cycle pulls in the whole cycle
 - Each component in the cycle has its dependencies
 - Each user will accumulate the sum of all dependencies of all parts in the cycle

Why fix cyclic dependencies

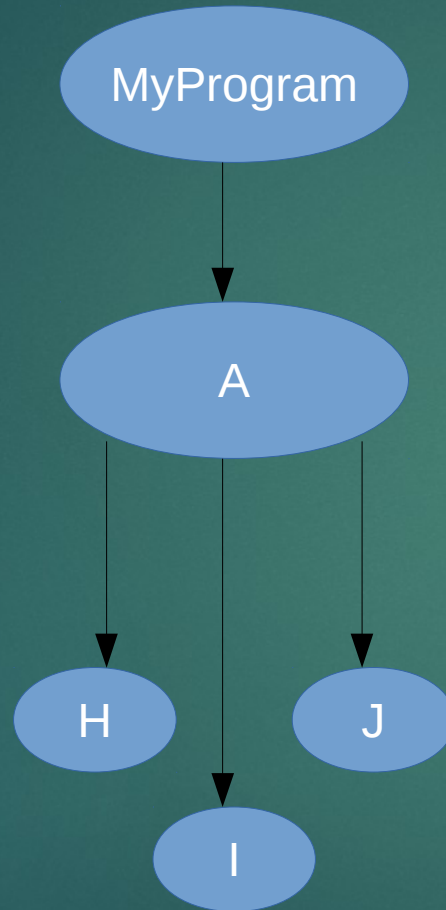


MyProgram

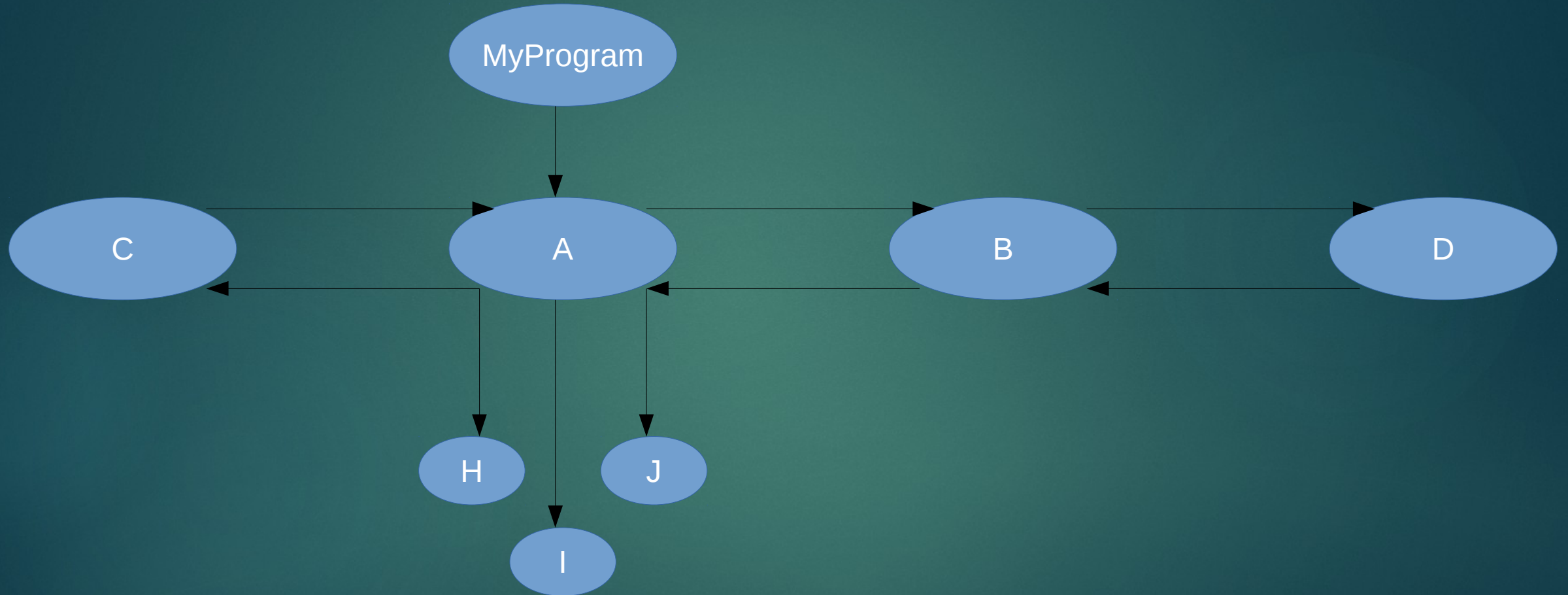
Why fix cyclic dependencies



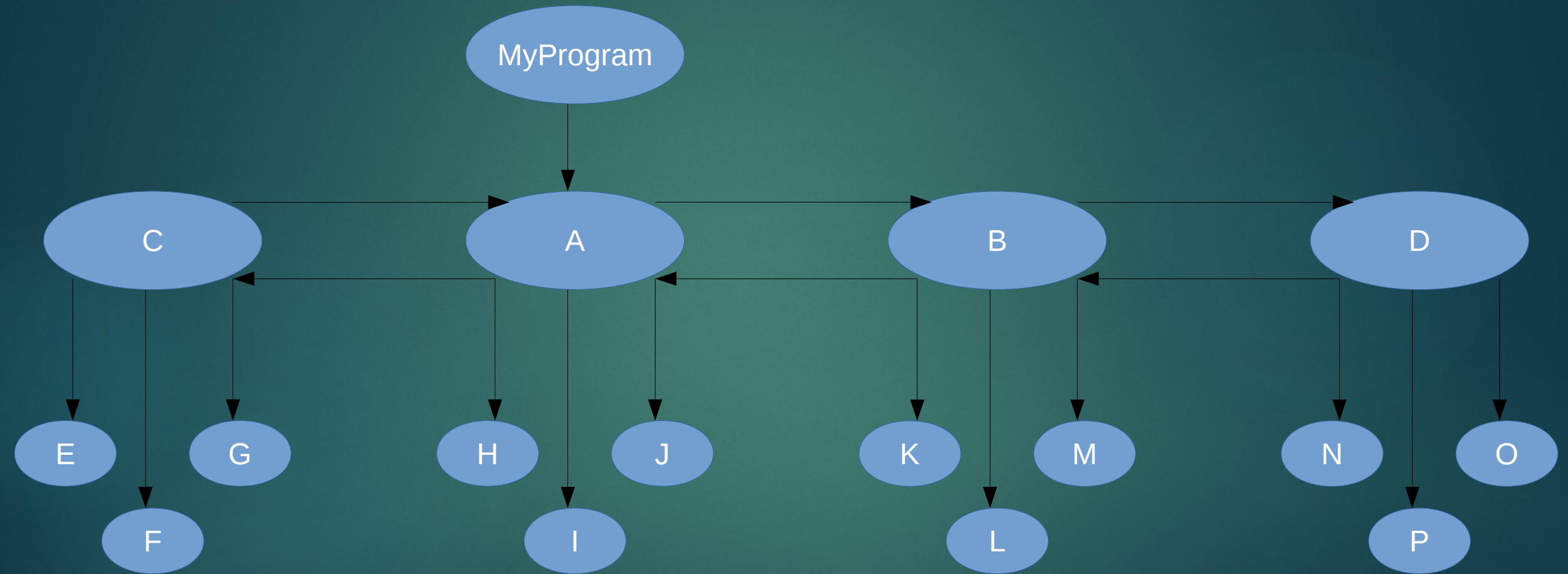
Why fix cyclic dependencies



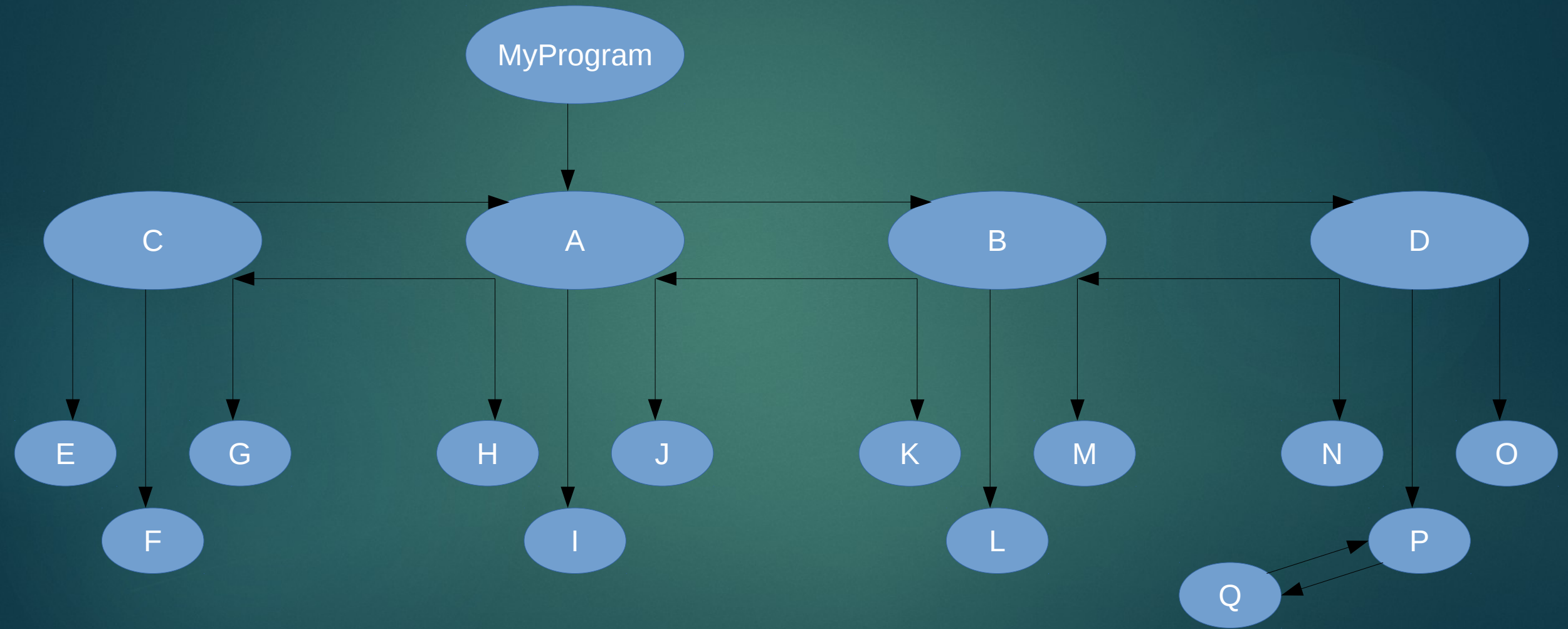
Why fix cyclic dependencies



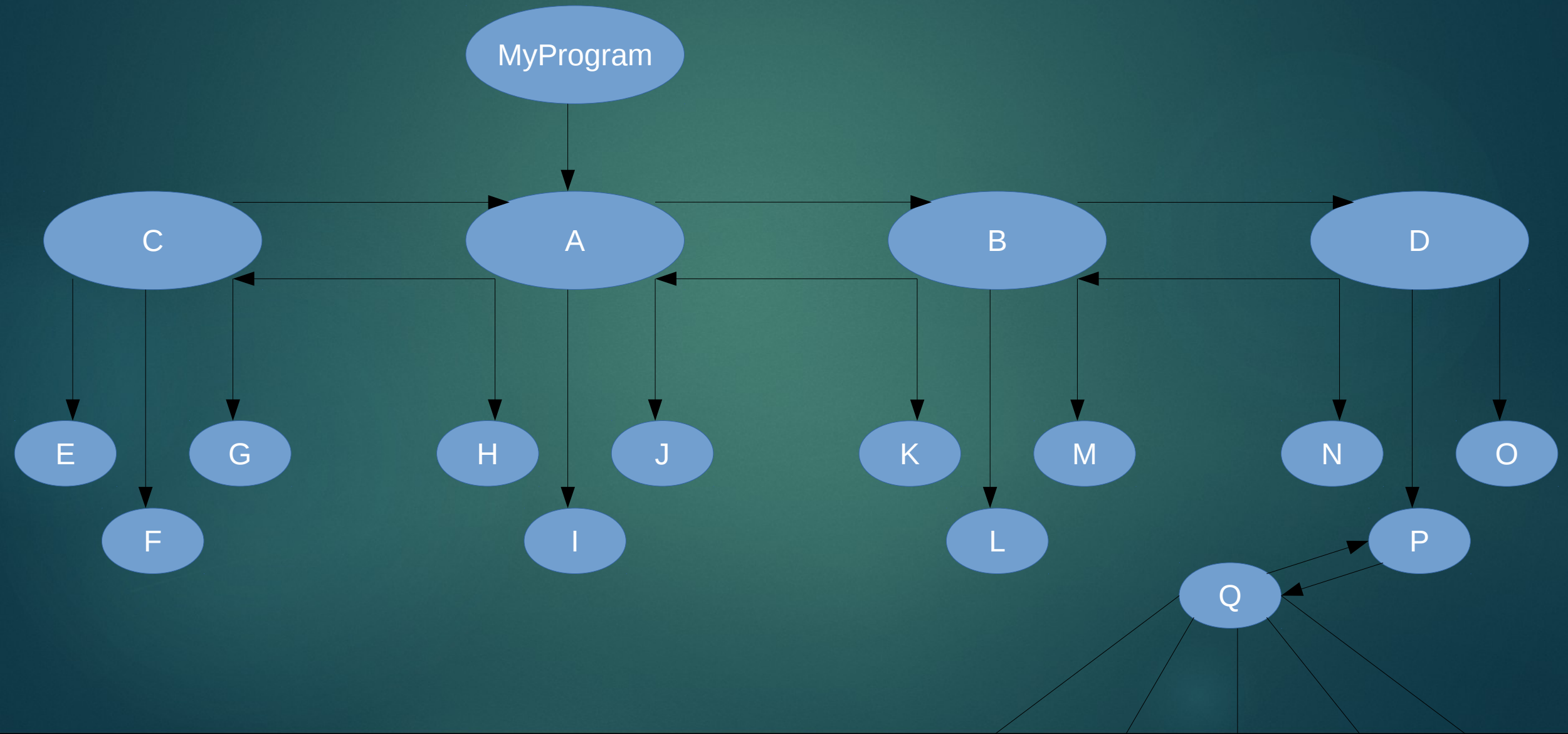
Why fix cyclic dependencies



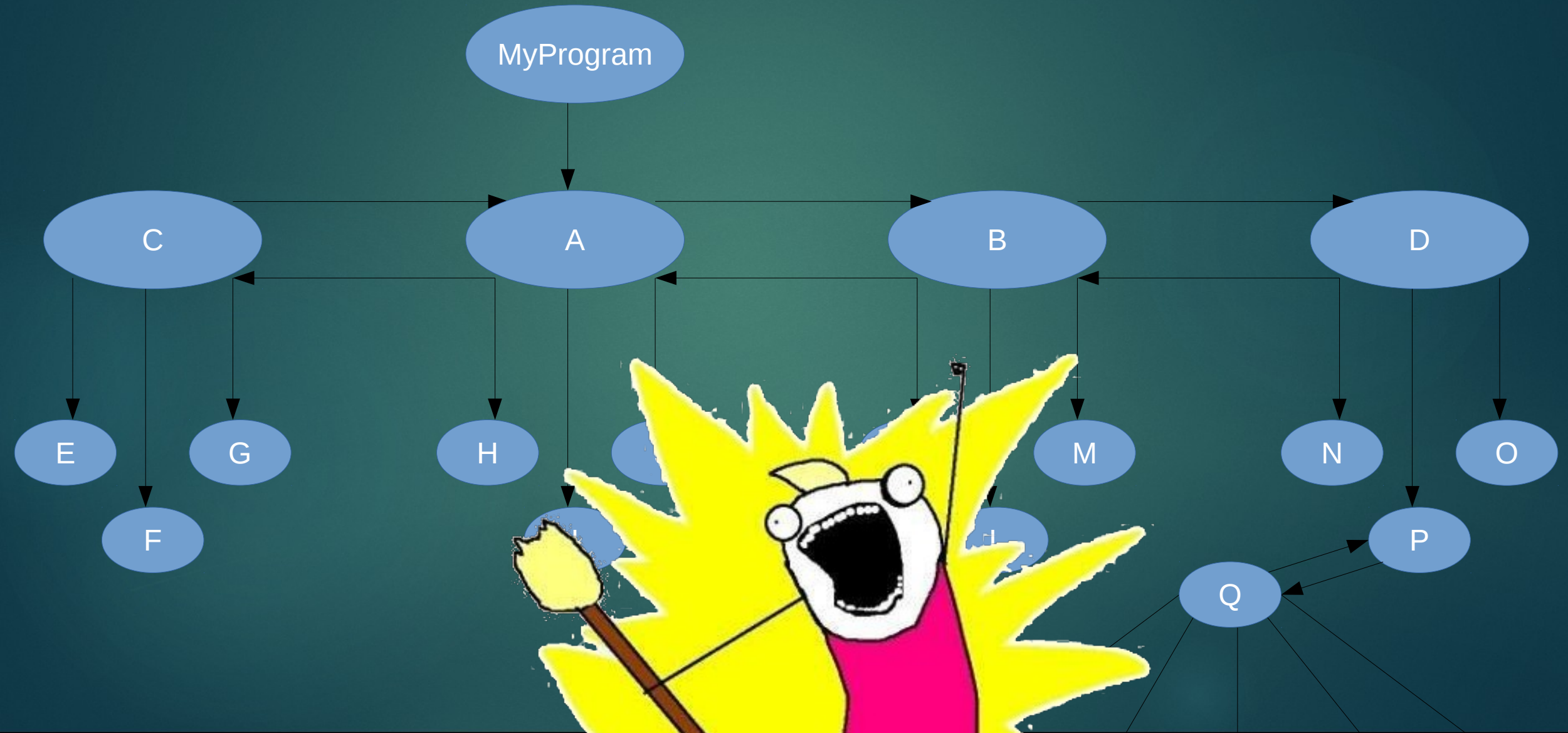
Why fix cyclic dependencies



Why fix cyclic dependencies



Why fix cyclic dependencies



Why fix cyclic dependencies

MyProgram



```
graph TD; A(MyProgram) --> B(ALL THE THINGS);
```

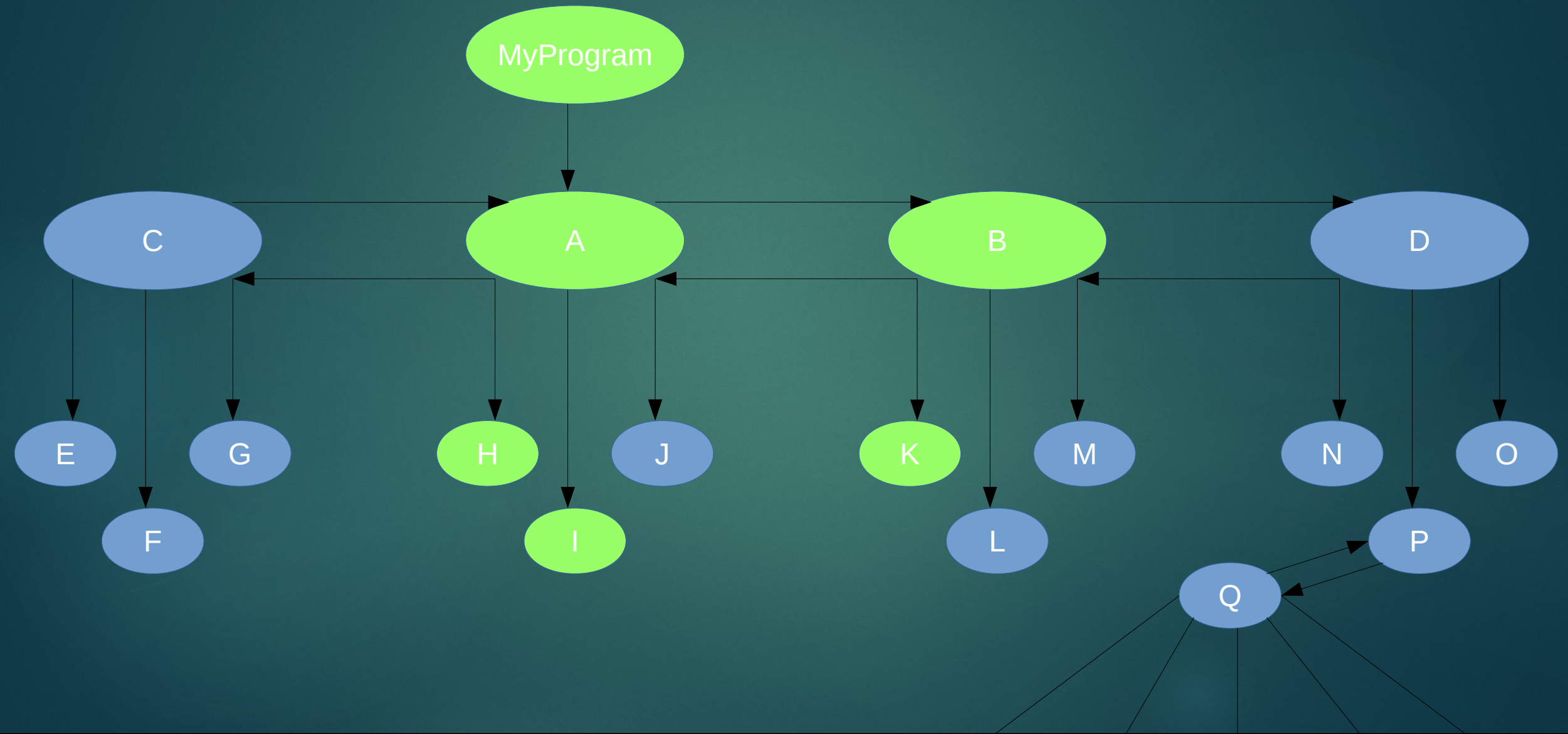
ALL THE THINGS



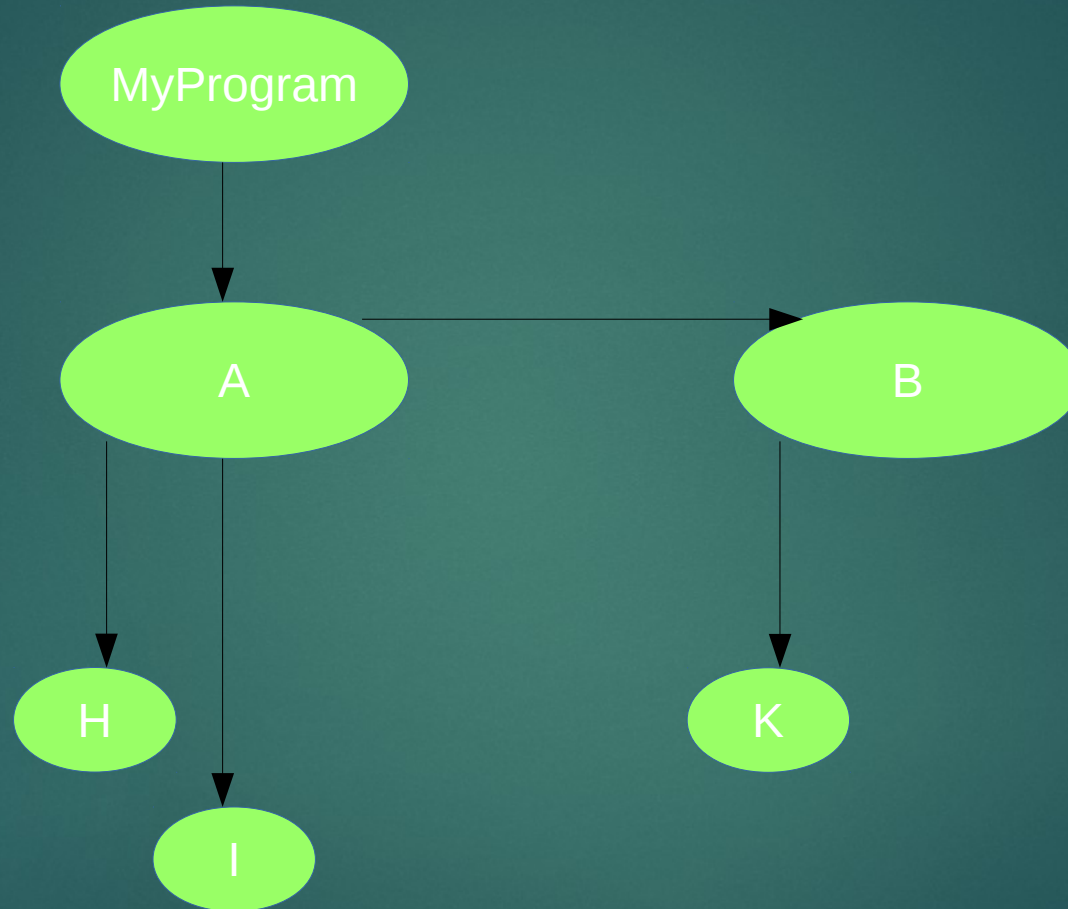


GCC C++ “Hello World” program -> .exe is 500kb big when compiled on Windows.

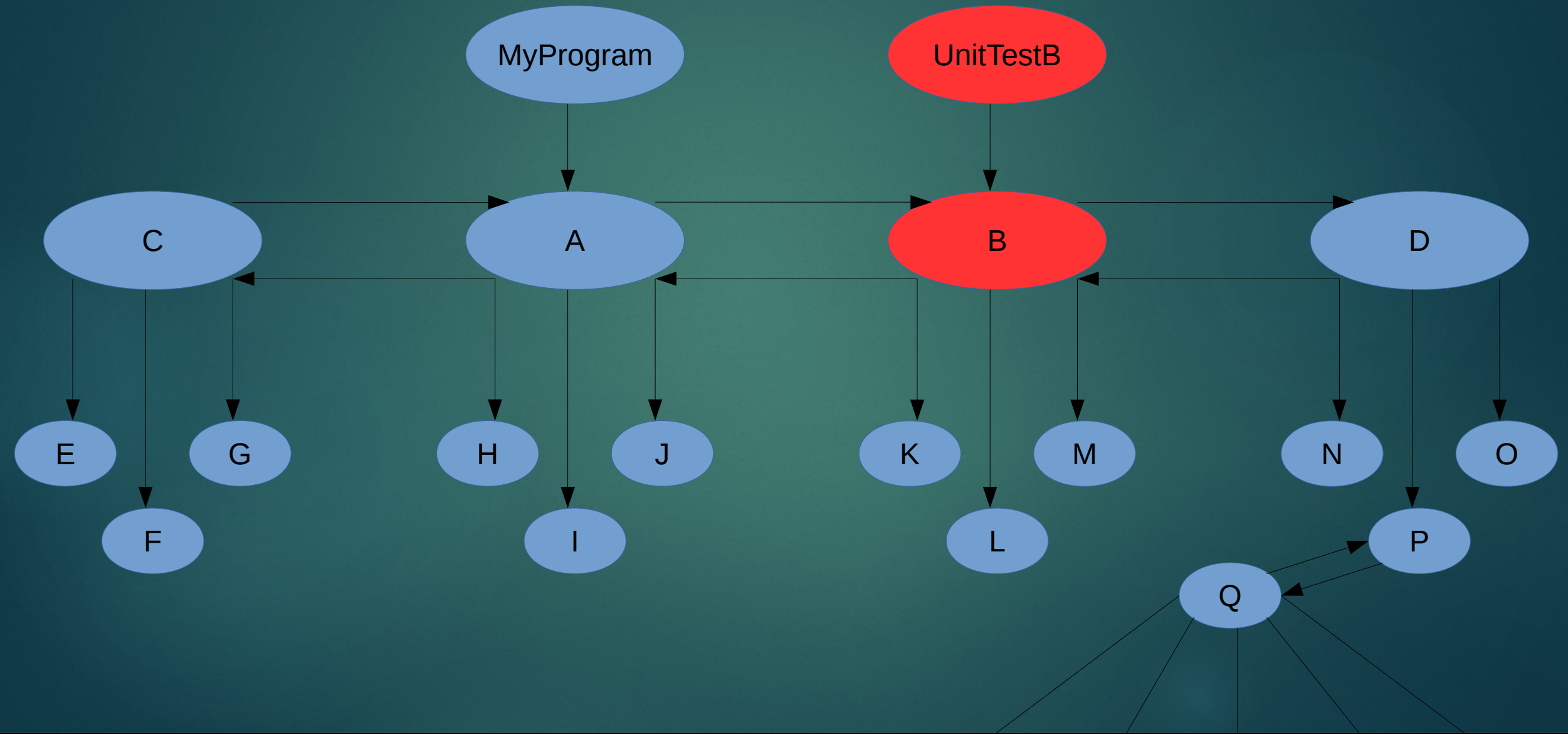
Why fix cyclic dependencies



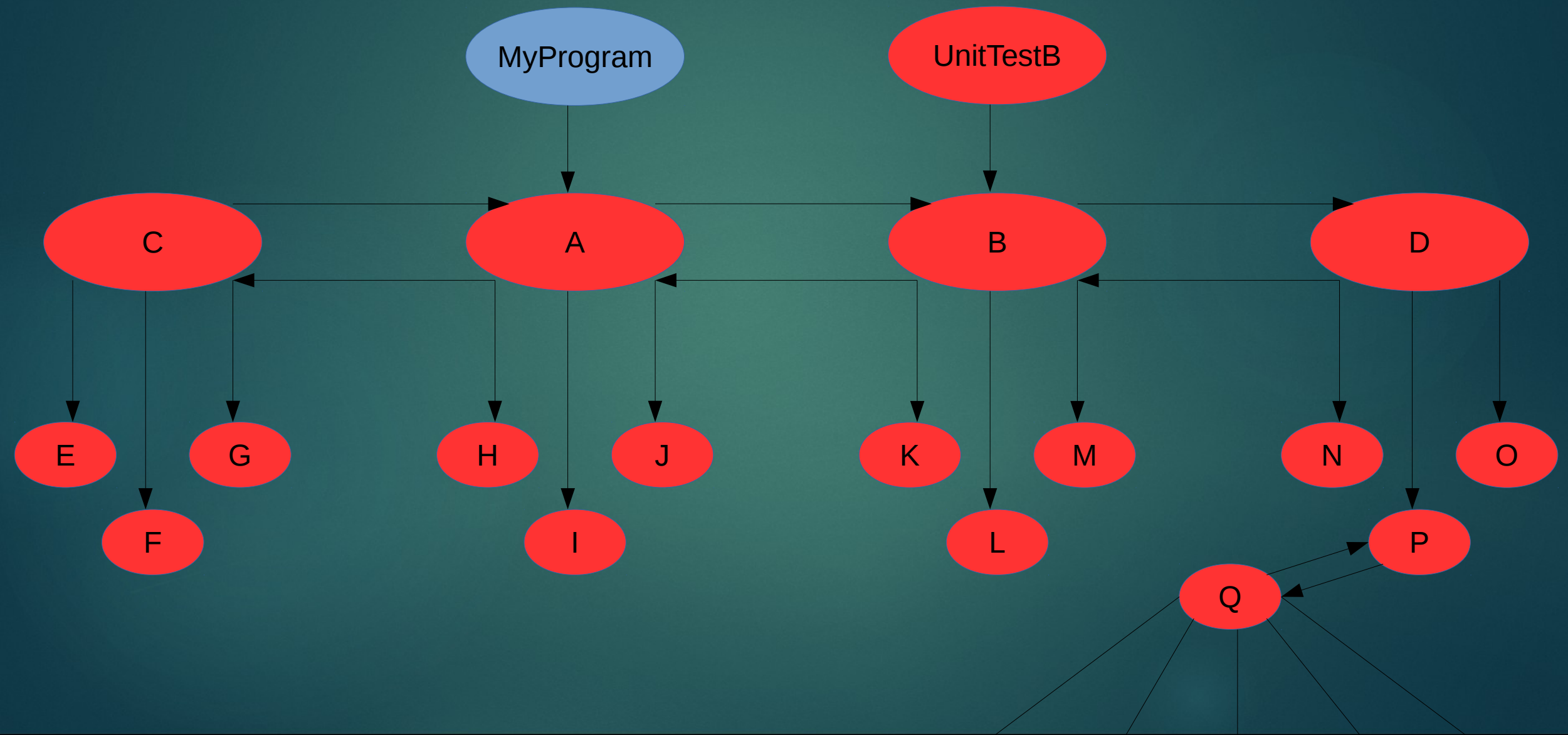
Why fix cyclic dependencies



Why fix cyclic dependencies



Why fix cyclic dependencies



Why fix cyclic dependencies

- Remove cycles to
 - Avoid false indirect dependencies
 - Reduce build tree size
 - Reduce change impact
 - Improve system testability
 - Avoid link order instability
 - Avoid link failures
 - Improve comprehensibility
 - Improve system stability
 - Make unit tests faster
 - Run less unit tests for a single change
 - Encourage reuse of smaller components
 - Speed up distributed builds
 - Speed up local builds

Why fix cyclic dependencies

- Remove cycles to
 - Avoid false indirect dependencies
 - Reduce build tree size
 - Reduce change impact
 - Improve system testability
 - Avoid link order instability
 - Avoid link failures
 - Improve comprehensibility
 - Improve system stability
 - Make unit tests faster
 - Run less unit tests for a single change
 - Encourage reuse of smaller components
 - Speed up distributed builds
 - Speed up local builds

There are so many reasons

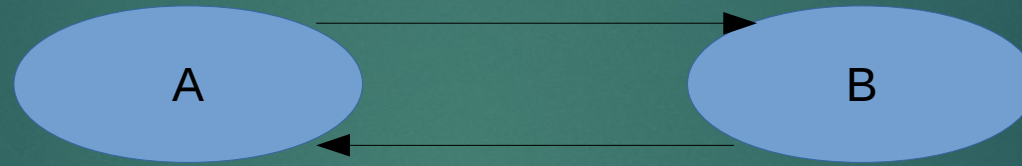


PARKING
ANY TIME

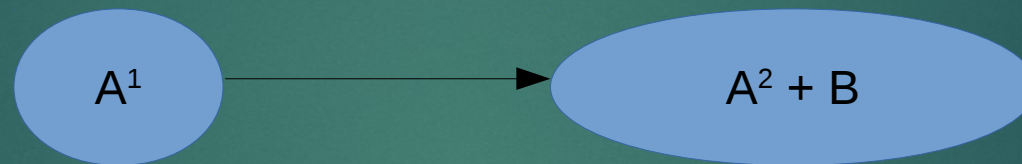
Gazetteer
Mexico

HONDA
TERESA
43 1606
5043 AS2
DOT 876
430 00011

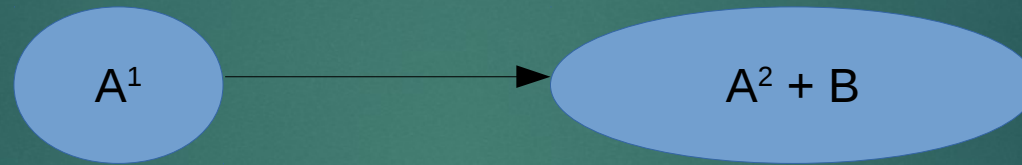
How to fix cyclic dependencies



1. Code motion

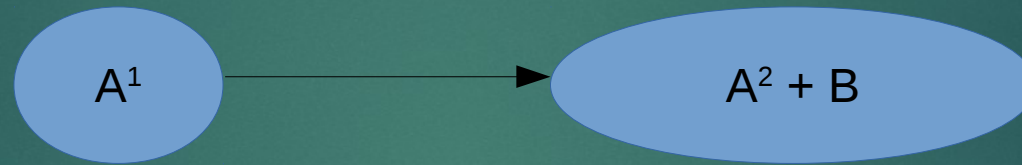


1. Code motion



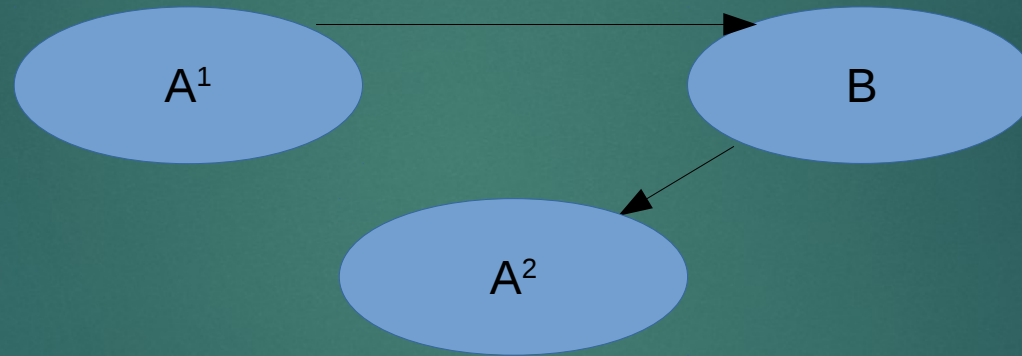
- Oops, put the file in the wrong place
- Unclear componentization

1. Code motion

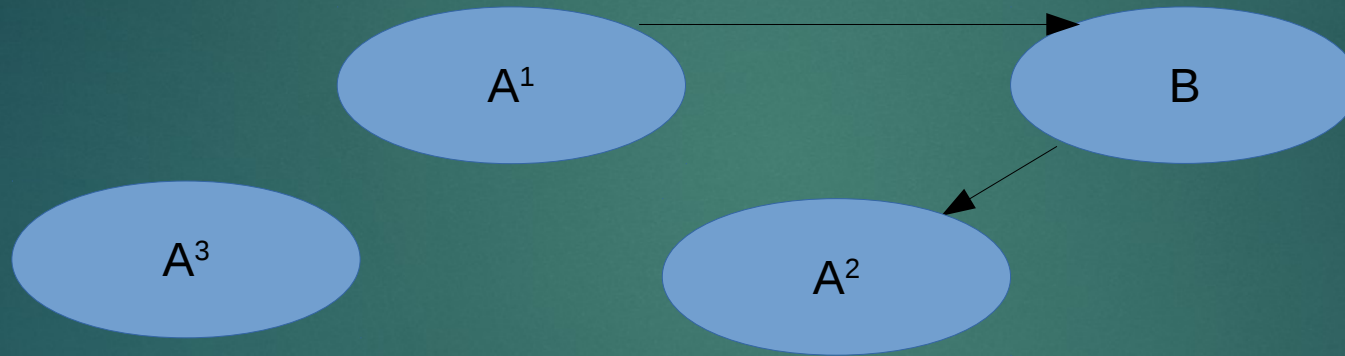


- Simplest change
- Very common

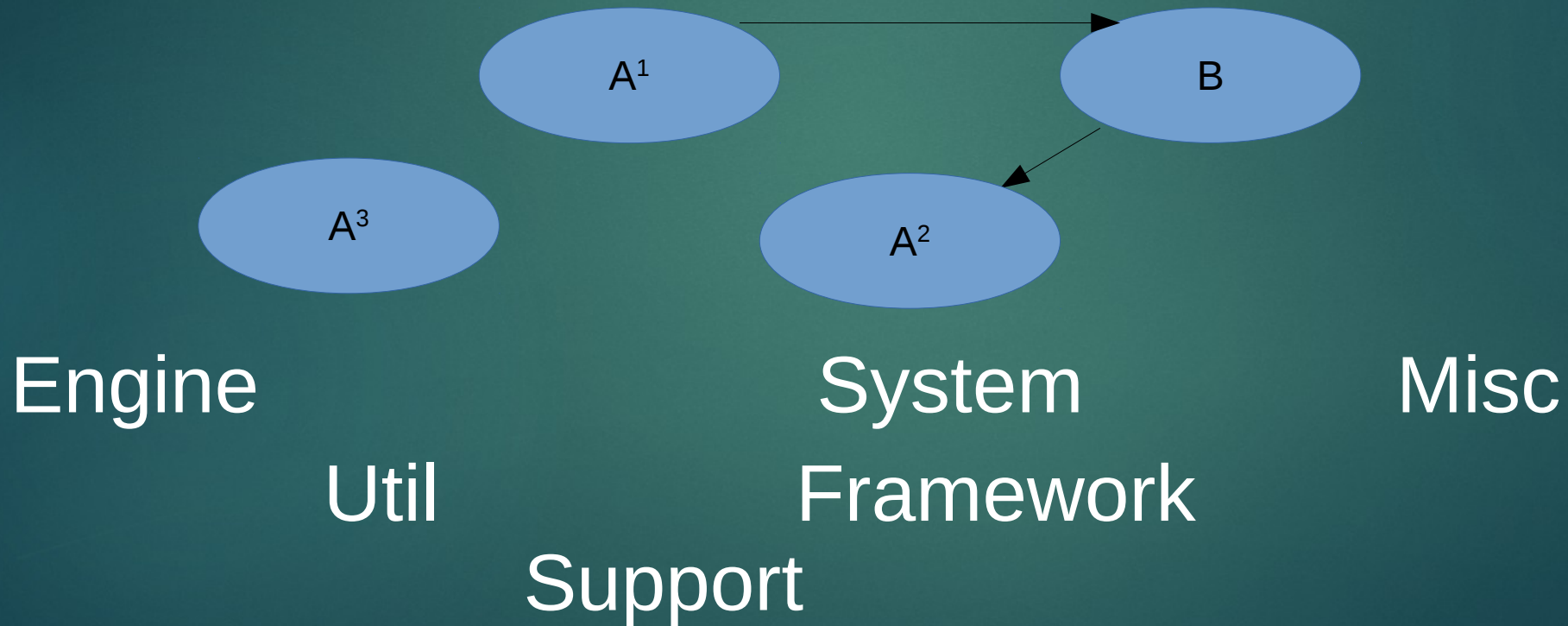
2. Split component



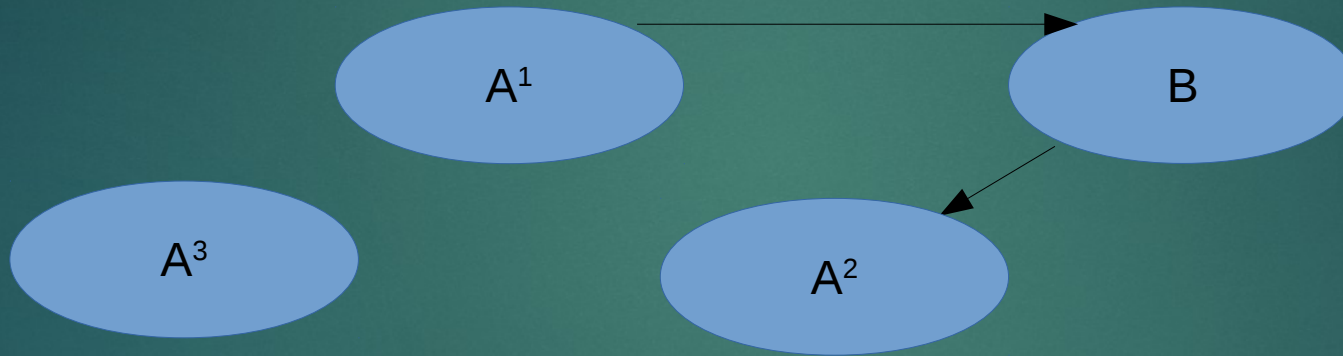
2. Split component



2. Split component

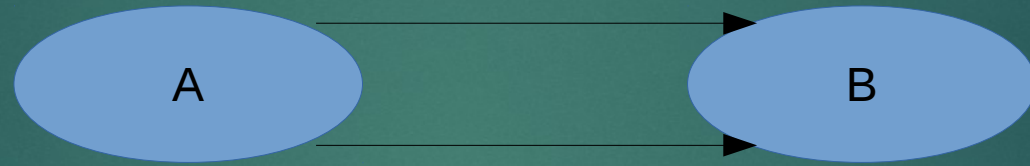


2. Split component

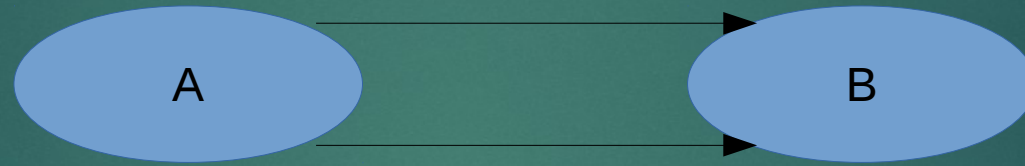


- Also very common
- Split off one responsibility at a time

3. Invert dependency

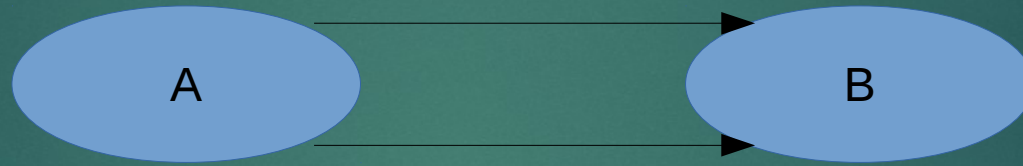


3. Invert dependency



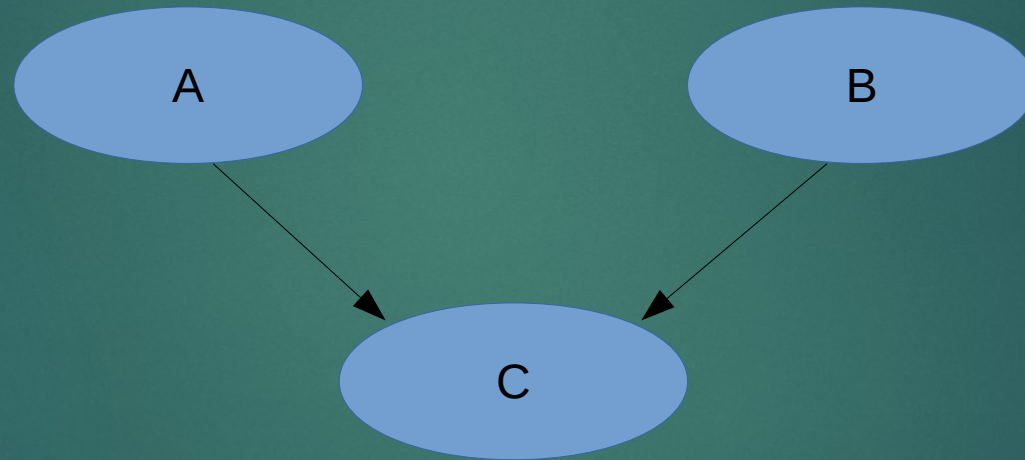
- Settings
- Configuration
- Callbacks

3. Invert dependency

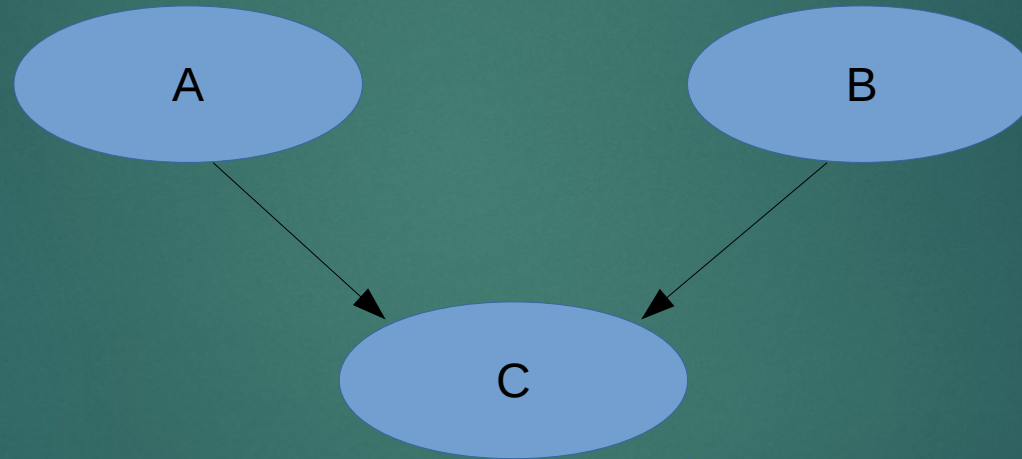


- Most effective in making code independent
- Requires major source code changes
- Will fix many cycles in one go

4. Extract common component

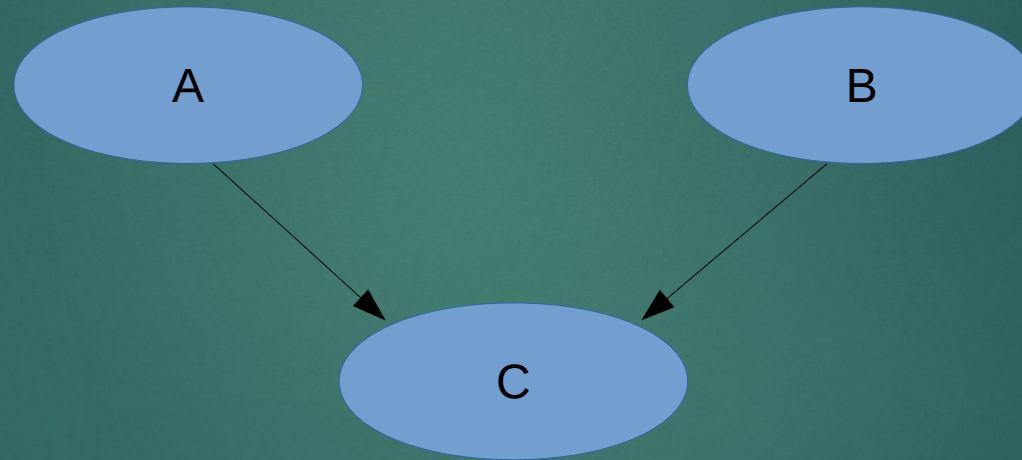


4. Extract common component



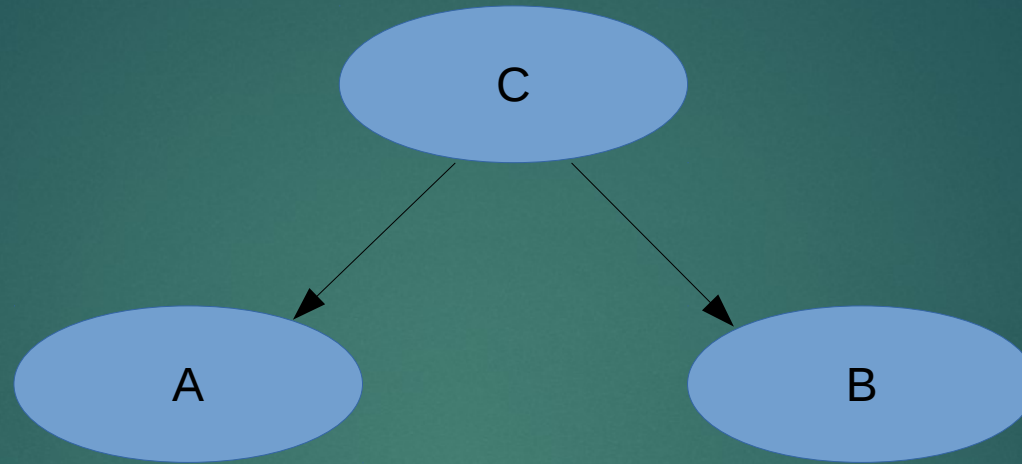
- Organic code growth
- Refactoring to be done, but not done

4. Extract common component

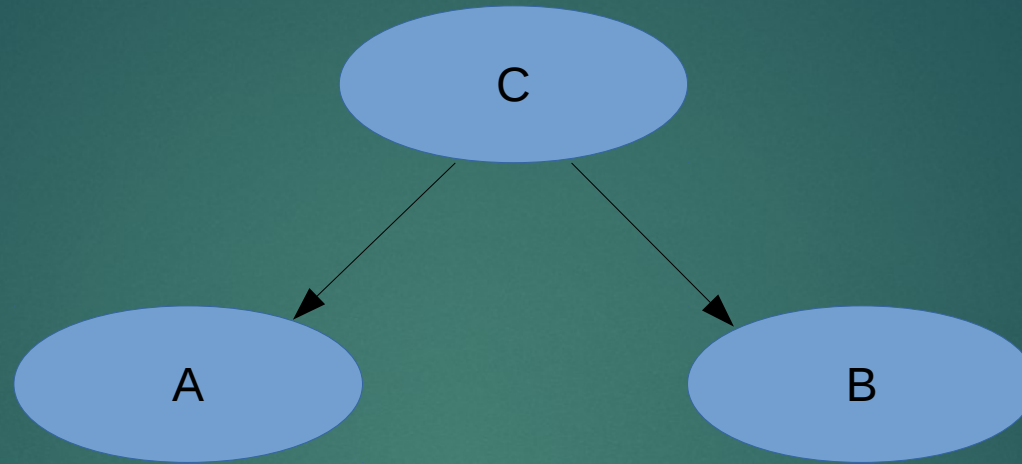


- Caused by SRP violation

5. Extract driver

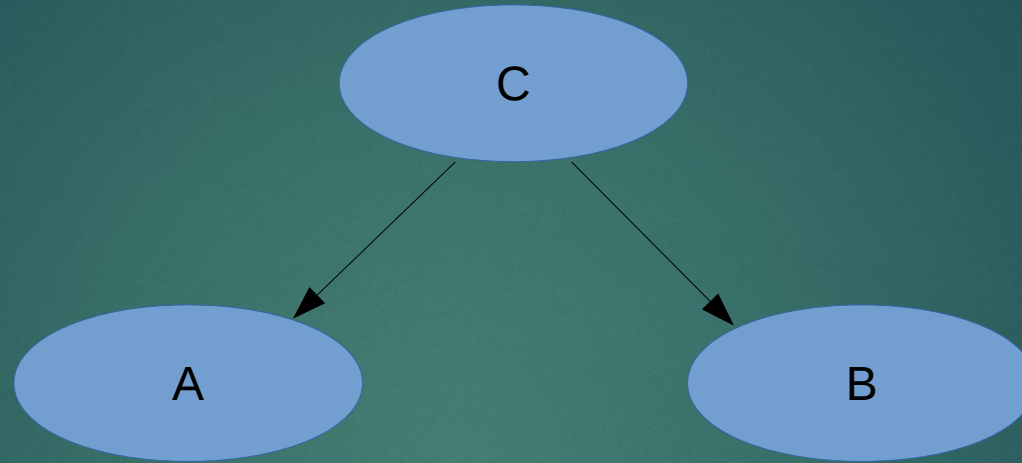


5. Extract driver



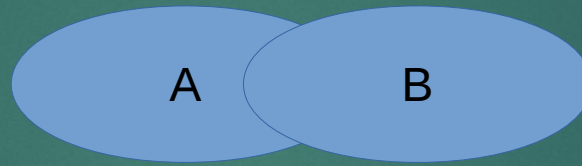
- Organic code growth
- Refactoring to be done, but not done

5. Extract driver

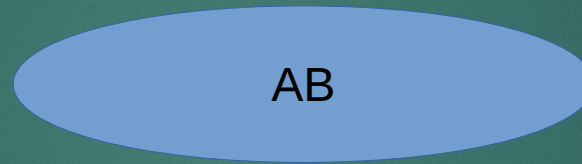


- Move external API to new component
- Interface Segregation violation

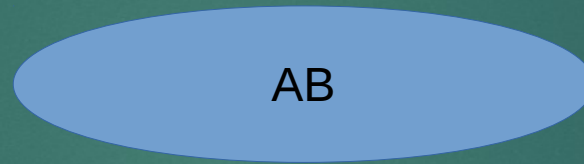
6. Merge components



6. Merge components

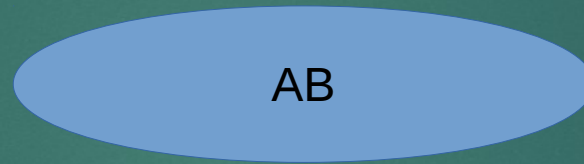


6. Merge components



- Same dependency results
- More honest code base

6. Merge components



- Least useful
- Usually not the right solution

Dependency visualization

- ▶ Overview of dependencies in project
- ▶ Cpp-Dependencies
- ▶ Spindel

Cpp-Dependencies

- ▶ Analyze compile-time dependencies automatically
 - Cyclic dependencies
 - Command-line interface
 - Multiple analysis methods to help fix dependencies
- Not interactive
- Not newbie friendly

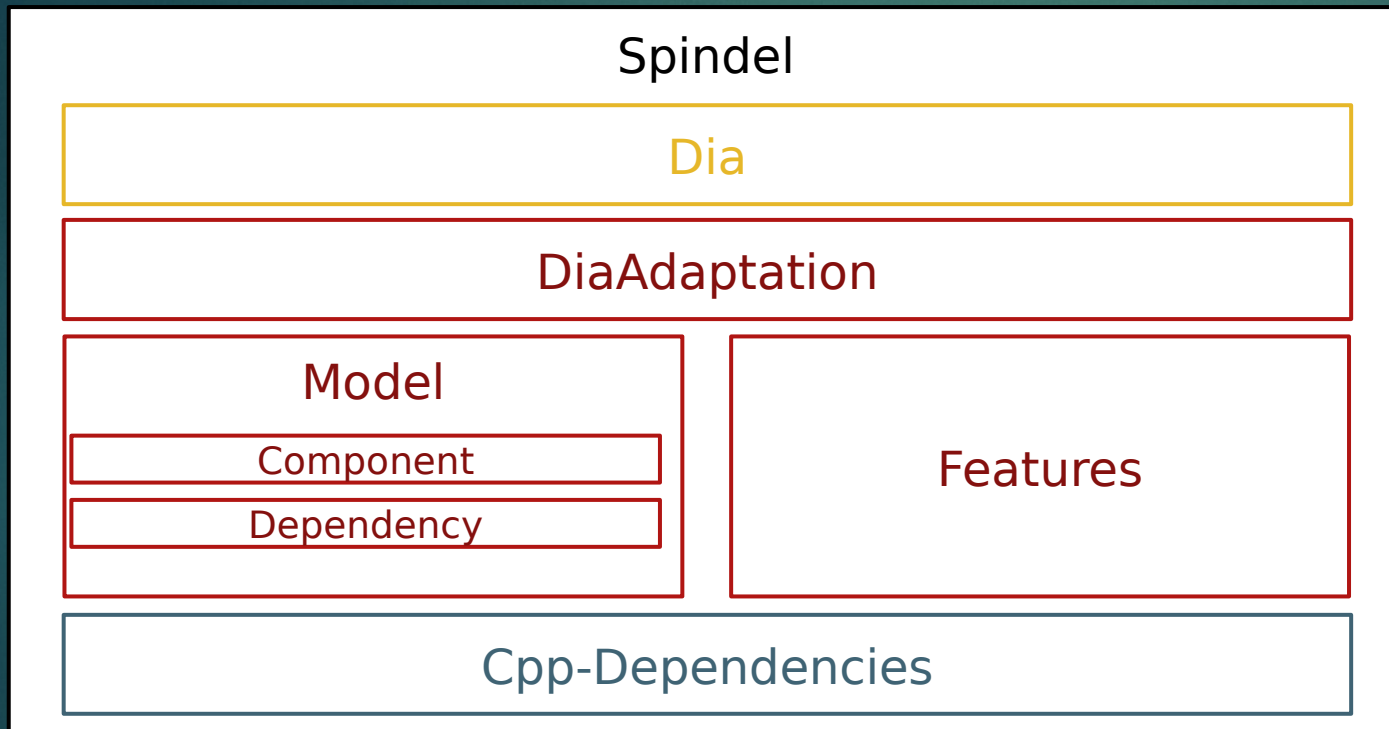
Spindel

- ▶ Interactive visualization of dependencies within project
 - ▶ Public and private compile time dependencies

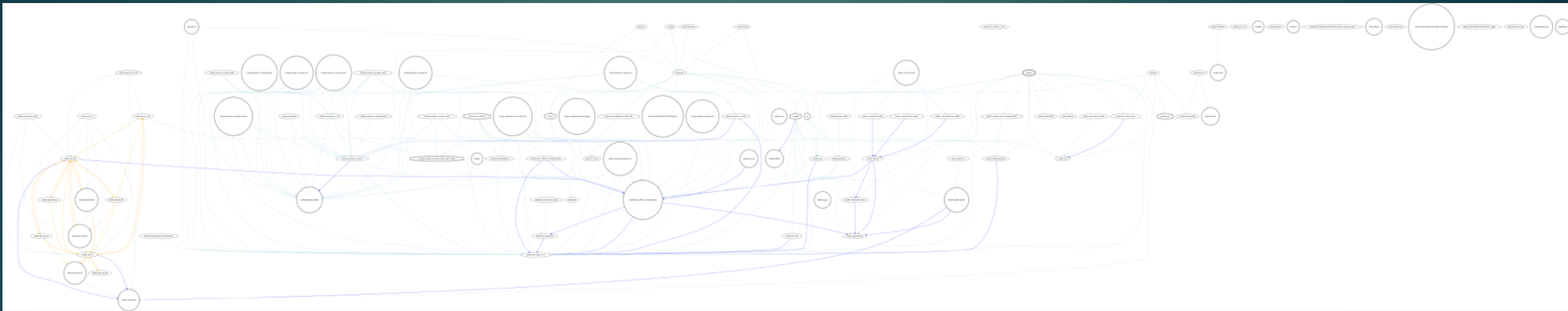
Spindel

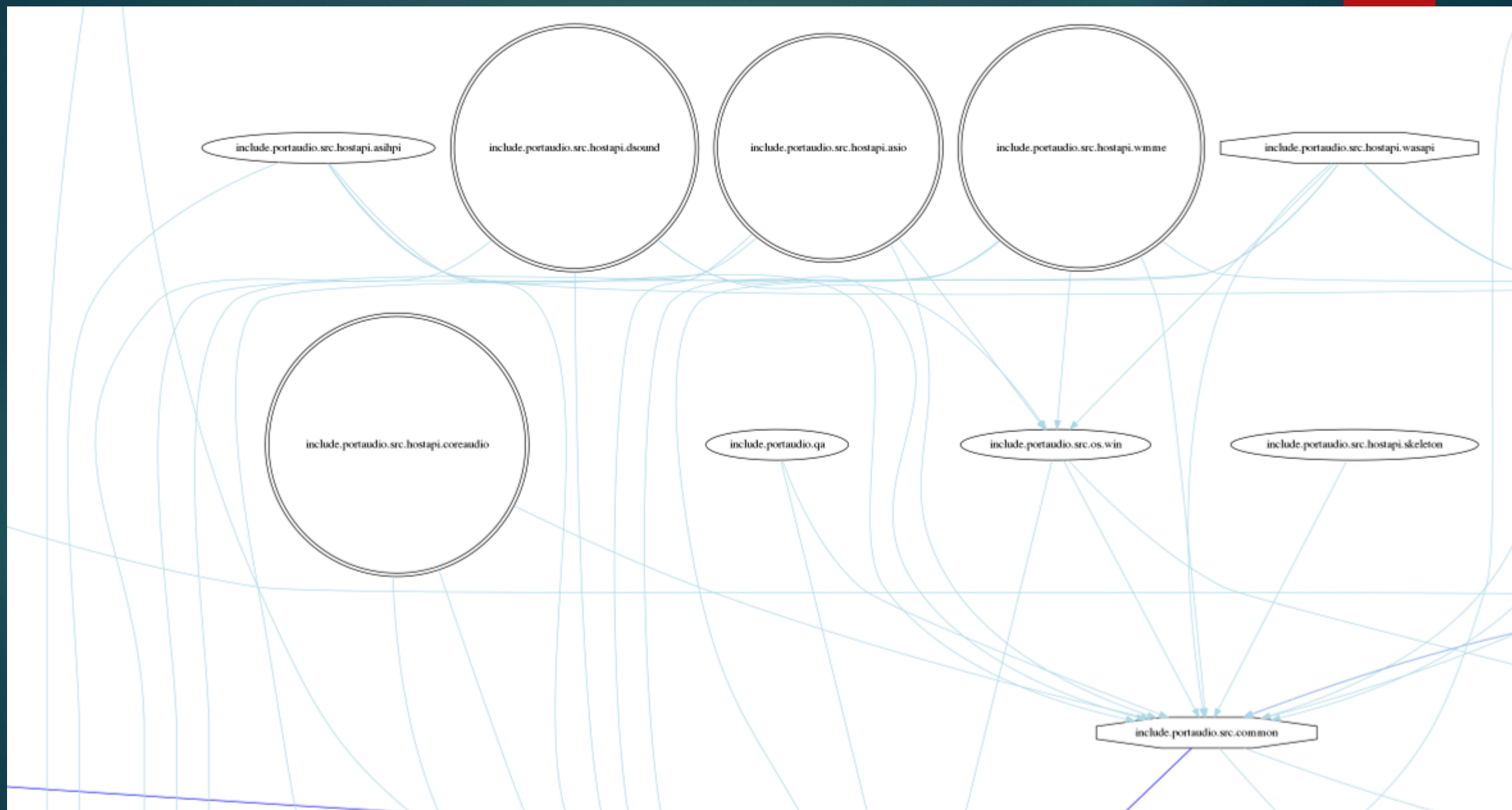
- ▶ Combination of:
 - ▶ Cpp-Dependencies
 - ▶ Dia Diagram editor

Spindel

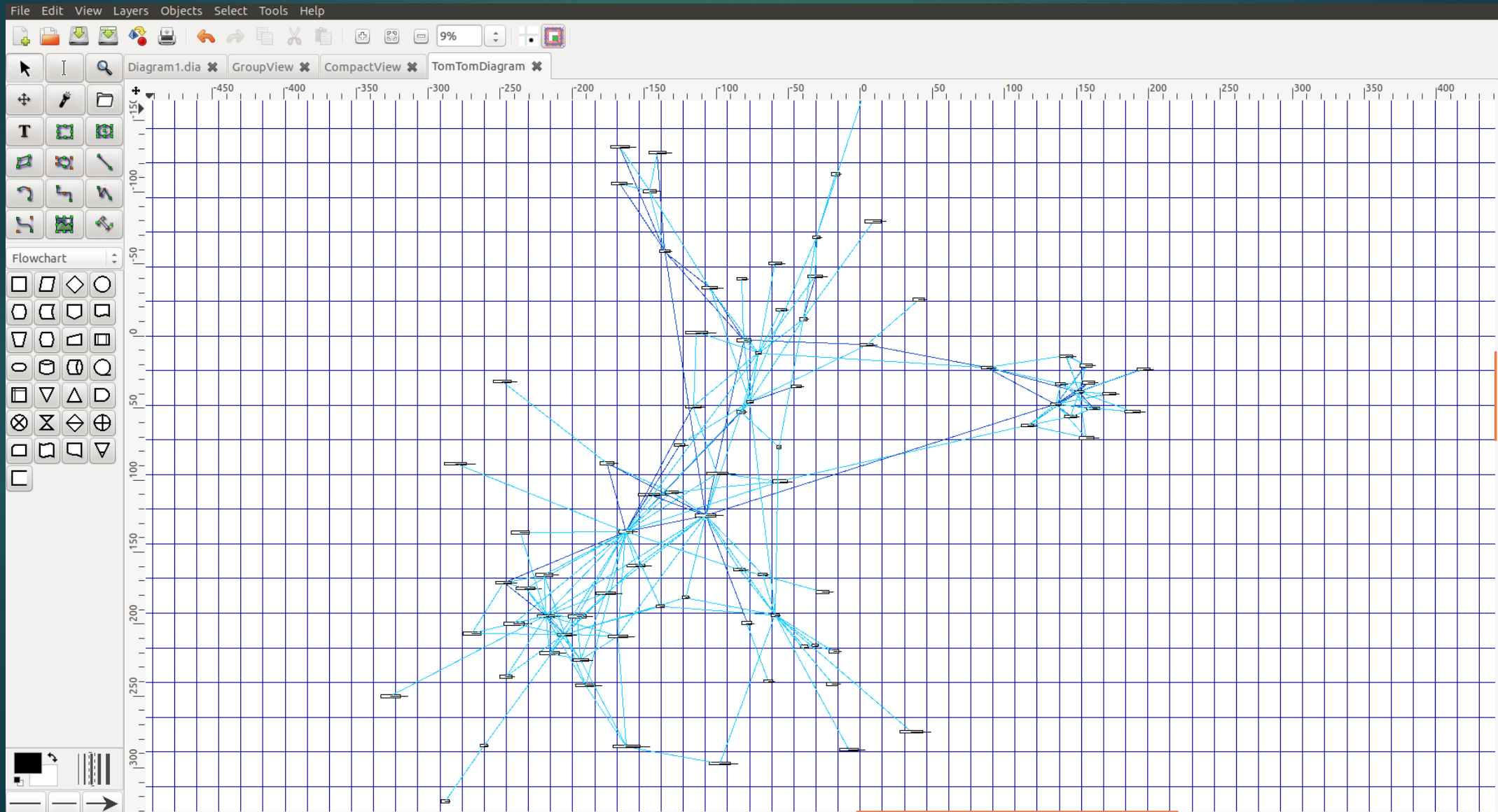


Cpp-dependencies output

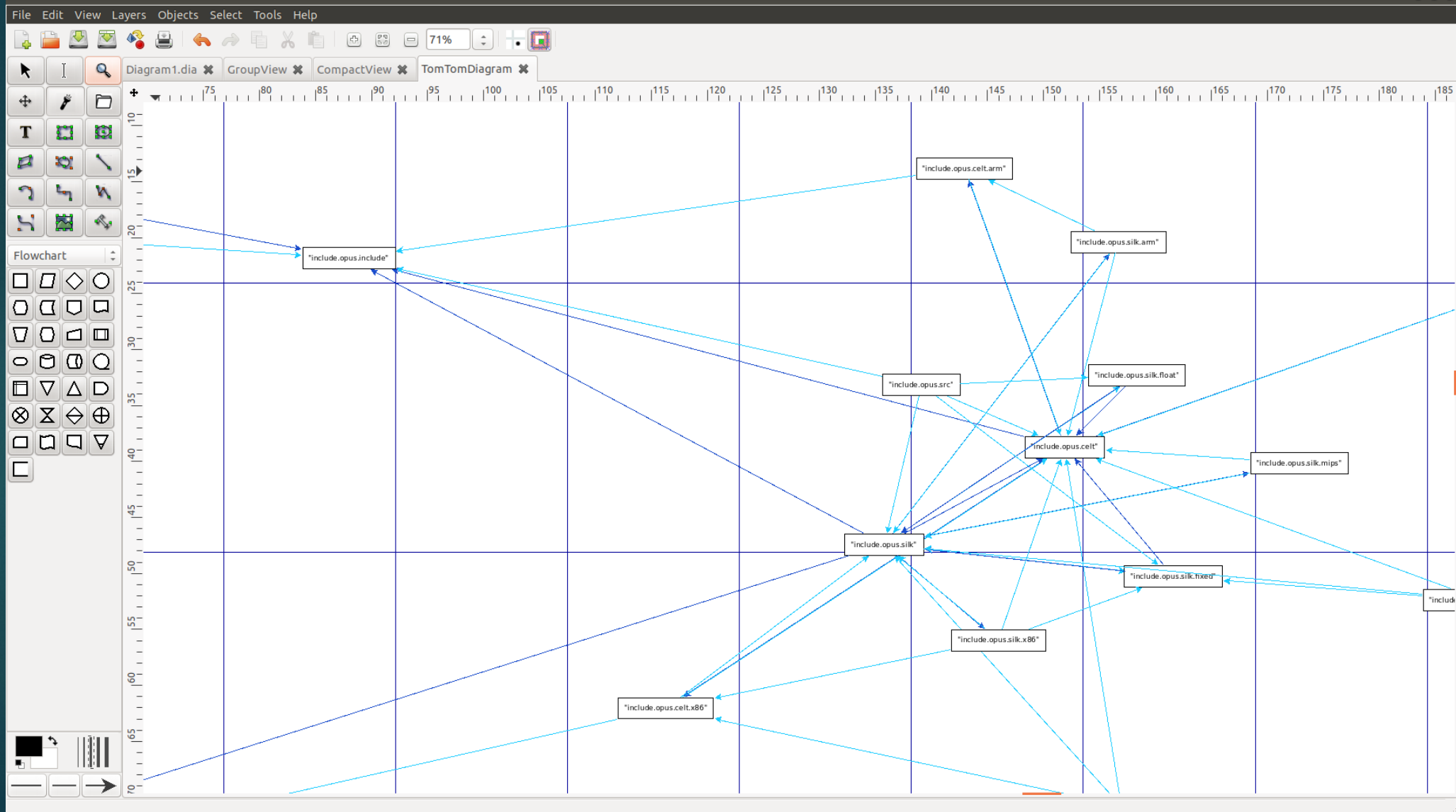




Spindel view



Spindel view



Spindel



▶ Features

- Highlighting
- More compact overview
- Folder structure overview

▶ Demo

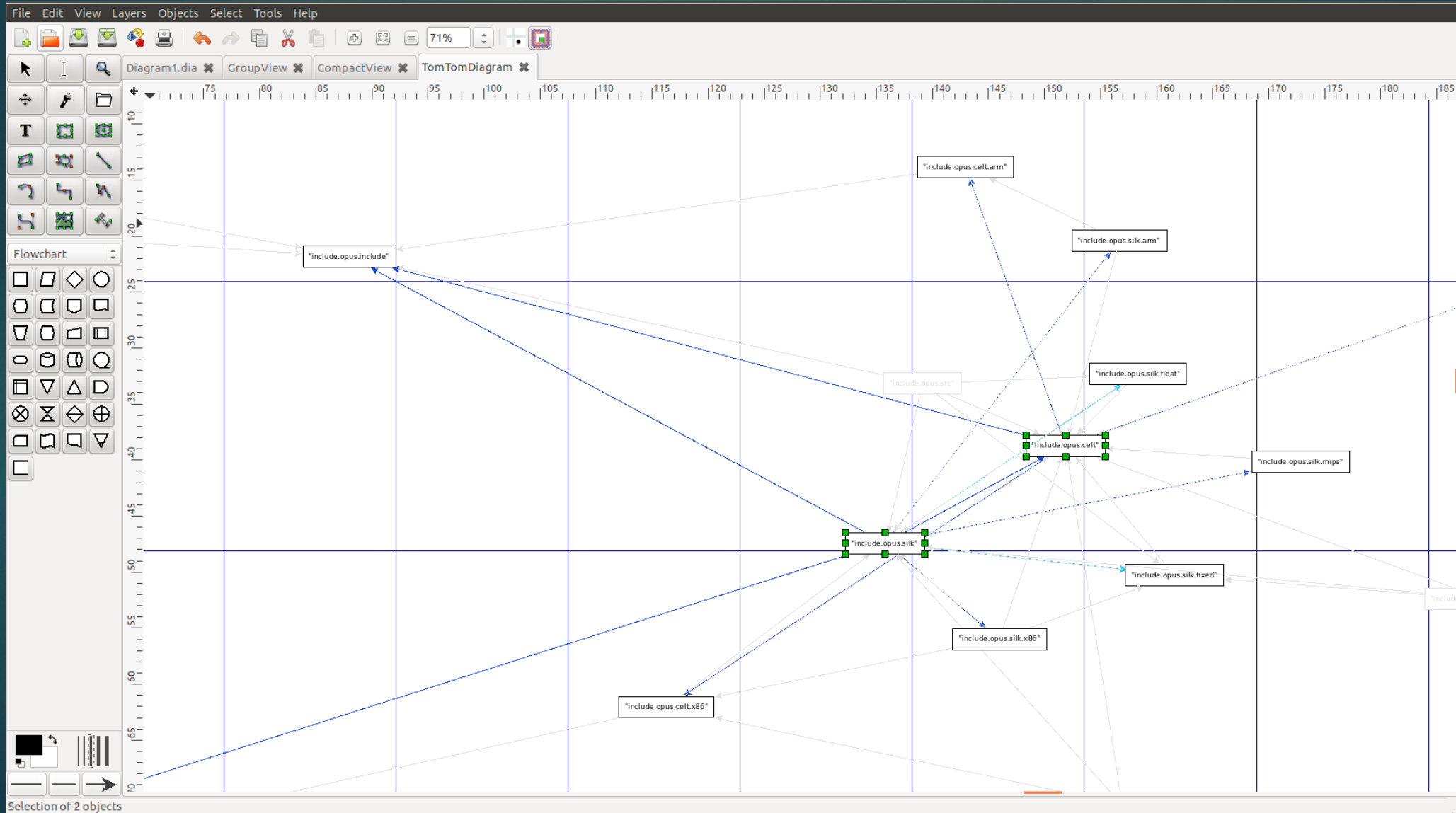
References

- <https://github.com/tomtom-international/cpp-dependencies>
- Kiki de Rooij
 - <https://www.linkedin.com/in/kiki-de-rooij-38bba882/>
- Peter Bindels
 - @dascandy42 (twitter)

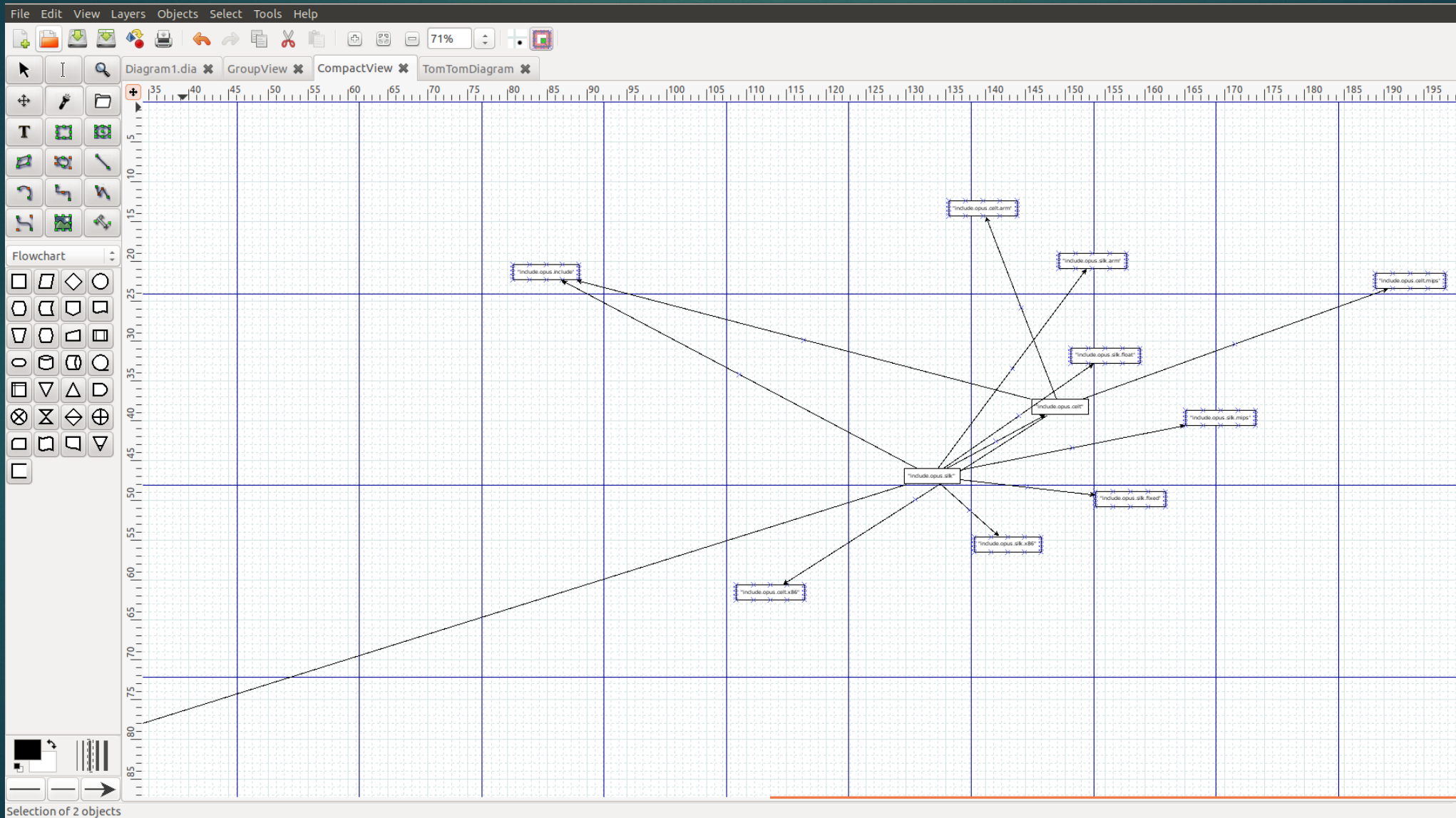


Questions

Spindel features



Spindel features



Spindel features

