

CPPNOW 2019

# DEPENDENCY INJECTION

A 25-DOLLAR TERM FOR A 5-CENT  
CONCEPT

Kris Jusiak, Quantlab Financial

[kris@jusiak.net](mailto:kris@jusiak.net) | [@KrisJusiak](https://twitter.com/@KrisJusiak) | [LINKEDIN.COM/IN/KRIS-JUSIAK](https://www.linkedin.com/in/kris-jusiak)

# MOTIVATION/GOAL/REQUIREMENT

**SCENARIO: NUMBER OF ATTENDEES WILLING TO USE DI  
SHOULD INCREASE**

# SCENARIO: NUMBER OF ATTENDEES WILLING TO USE DI SHOULD INCREASE

Given There is a talk about DI at CppNow-2019 (✓)

# SCENARIO: NUMBER OF ATTENDEES WILLING TO USE DI SHOULD INCREASE

Given There is a talk about DI at CppNow-2019 (✓)

And The number of attendees is greater than zero (✓)

# SCENARIO: NUMBER OF ATTENDEES WILLING TO USE DI SHOULD INCREASE

Given There is a talk about DI at CppNow-2019 (✓)

And The number of attendees is greater than zero (✓)

And There are attendees not using DI (?)

# SCENARIO: NUMBER OF ATTENDEES WILLING TO USE DI SHOULD INCREASE

Given There is a talk about DI at CppNow-2019 (✓)

And The number of attendees is greater than zero (✓)

And There are attendees not using DI (?)

When The talk has been given by a speaker (~)

# SCENARIO: NUMBER OF ATTENDEES WILLING TO USE DI SHOULD INCREASE

Given There is a talk about DI at CppNow-2019 (✓)

And The number of attendees is greater than zero (✓)

And There are attendees not using DI (?)

When The talk has been given by a speaker (~)

Then The number of attendees willing to use DI should increase (?)

# AGENDA

# AGENDA

- DESIGN

# AGENDA

- DESIGN
- PRINCIPLES

# AGENDA

- DESIGN
  - PRINCIPLES
  - DISCOVERY

# AGENDA

- DESIGN
  - PRINCIPLES
  - DISCOVERY
- DEPENDENCY INJECTION (DI)

- DI LIBRARIES

---

- DI LIBRARIES
  - HYPODERMIC
-

- DI LIBRARIES
    - HYPODERMIC
    - GOOGLE.FRUIT
-

- DI LIBRARIES
    - HYPODERMIC
    - GOOGLE.FRUIT
    - [BOOST].DI
-

- DI LIBRARIES
    - HYPODERMIC
    - GOOGLE.FRUIT
    - [BOOST].DI
  - SUMMARY
-

- DI LIBRARIES
  - HYPODERMIC
  - GOOGLE.FRUIT
  - [BOOST].DI
- SUMMARY

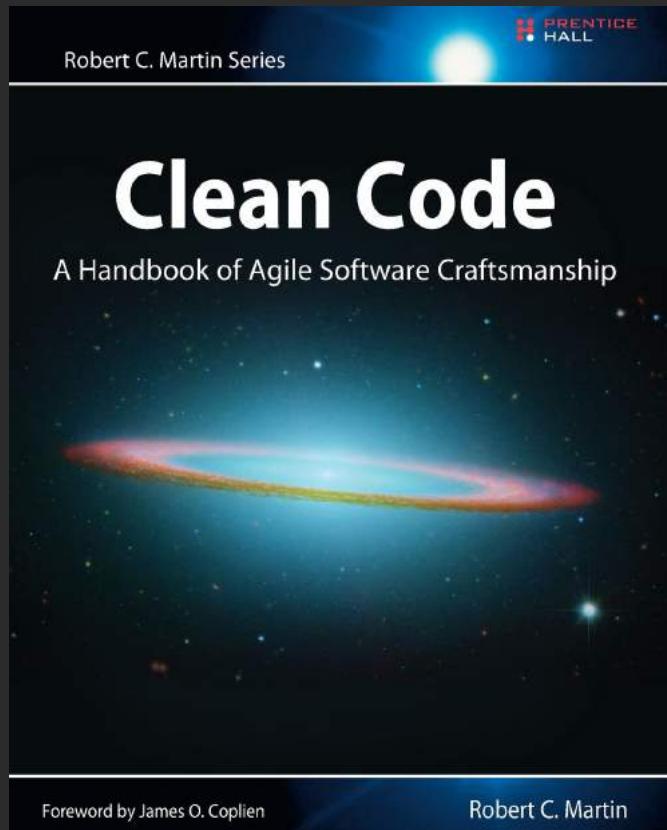
---

darkblue background - something to remember ✓

# DESIGN PRINCIPLES

**"THE ONLY WAY TO GO FAST IS TO GO WELL", UNCLE BOB**

"THE ONLY WAY TO GO FAST IS TO GO WELL", UNCLE BOB



# FLEXIBLE

# FLEXIBLE

*"Nothing is certain in Software Development except for bugs and constatly changing requirements",  
Franklin rule*

# SCALABLE

# SCALABLE

*Easy to extend, maintain, reuse*

# SCALABLE

*Easy to extend, maintain, reuse*



# TESTABLE

# TESTABLE

*"If you liked it then you should have put a test on it", Beyonce rule*

# SOLID VS 'STUPID'

# SOLID VS 'STUPID'

S Single Responsibility  
O Open-close  
L Liskov substitution  
I Interface segregation  
D Dependency inversion

S Singleton  
T Tight Coupling  
U Untestability  
P Premature Optimization  
I Indescriptive Naming  
D Duplication

# DESIGN DISCOVERY

# DESIGN DISCOVERY

*Flexible/Scalable/Testable*

KISS - ~~KEEP IT SIMPLE~~, 'STUPID'

# KISS - ~~KEEP IT SIMPLE~~, 'STUPID'

```
int main() {
```

```
}
```

# KISS - ~~KEEP IT SIMPLE~~, 'STUPID'

```
int main() {  
  
    const auto before = attendees_ask();  
    speaker_talk("Kris", "Dependency Injection");  
    const auto after = attendees_ask();  
    return after - before;  
  
}
```

# KISS - ~~KEEP IT SIMPLE~~, 'STUPID'

```
int main() {  
  
    const auto before = attendees_ask();  
    speaker_talk("Kris", "Dependency Injection");  
    const auto after = attendees_ask();  
    return after - before;  
  
}
```

*Code is read much more often than it's  
written*

KISS - ~~KEEP IT SIMPLE~~, 'STUPID'

# KISS - ~~KEEP IT SIMPLE~~, 'STUPID'

- FLEXIBLE?

# KISS - ~~KEEP IT SIMPLE~~, 'STUPID'

- FLEXIBLE?
- NO: TIGHTLY COUPLED

# KISS - ~~KEEP IT SIMPLE~~, 'STUPID'

- FLEXIBLE?
  - NO: TIGHTLY COUPLED
- SCALABLE?

# KISS - ~~KEEP IT SIMPLE~~, 'STUPID'

- FLEXIBLE?
  - NO: TIGHTLY COUPLED
- SCALABLE?
  - NO: HARD TO EXTEND

# KISS - ~~KEEP IT SIMPLE~~, 'STUPID'

- FLEXIBLE?
  - NO: TIGHTLY COUPLED
- SCALABLE?
  - NO: HARD TO EXTEND
- TESTABLE?

# KISS - ~~KEEP IT SIMPLE~~, 'STUPID'

- FLEXIBLE?
  - NO: TIGHTLY COUPLED
- SCALABLE?
  - NO: HARD TO EXTEND
- TESTABLE?
  - NO: IMPLEMENTATION IN MAIN

LET'S MAKE IT somewhat TESTABLE 

# BEHAVIOR DRIVEN DEVELOPMENT (BDD) / GUNIT

# BEHAVIOR DRIVEN DEVELOPMENT (BDD) / GUNIT

```
GSTEPS("Number of attendees willing to use DI should increase") {  
    auto result = 0;
```

```
}
```

# BEHAVIOR DRIVEN DEVELOPMENT (BDD) / GUNIT

```
GSTEPS("Number of attendees willing to use DI should increase") {  
    auto result = 0;
```

```
    Given("There is a talk about DI at CppNow-2019") = [&] {  
        cppnow_talk sut{};
```

```
}
```

```
}
```

# BEHAVIOR DRIVEN DEVELOPMENT (BDD) / GUNIT

```
GSTEPS("Number of attendees willing to use DI should increase") {  
    auto result = 0;
```

```
    Given("There is a talk about DI at CppNow-2019") = [&] {  
        cppnow_talk sut{};
```

```
        When("The talk has been given by a speaker") = [&] {  
            result = sut.run();  
        };
```

```
}
```

```
}
```

# BEHAVIOR DRIVEN DEVELOPMENT (BDD) / GUNIT

```
GSTEPS("Number of attendees willing to use DI should increase") {  
    auto result = 0;
```

```
    Given("There is a talk about DI at CppNow-2019") = [&] {  
        cppnow_talk sut{};
```

```
        When("The talk has been given by a speaker") = [&] {  
            result = sut.run();  
        };
```

```
        Then("Then The number of attendees willing  
              to use DI should increase") = [&] {  
            EXPECT(result > 0);  
        };
```

```
}
```

```
}
```

# BDD - REFACTOR

# BDD - REFACTOR

```
class cppnow_talk {  
public:
```

```
} ;
```

# BDD - REFACTOR

```
class cppnow_talk {  
public:
```

```
    auto run() {
```

```
}
```

```
} ;
```

# BDD - REFACTOR

```
class cppnow_talk {
public:

    auto run() {

        const auto before = attendees_ask();
        speaker_talk("Kris", "Dependency Injection");
        const auto after = attendees_ask();
        return after - before;

    }

};
```



# CONSIDER TEST DRIVING YOUR CODE (BDD/TDD)



# CONSIDER TEST DRIVING YOUR CODE (BDD/TDD)

- BDD, uses automated examples to guide us towards **building the right thing**



## CONSIDER TEST DRIVING YOUR CODE (BDD/TDD)

- BDD, uses automated examples to guide us towards **building the right thing**
- TDD uses unit tests to guides us towards **building it right**

LET'S MAKE IT more FLEXIBLE 

# SINGLE RESPONSIBILITY PRINCIPLE (SRP)

# SINGLE RESPONSIBILITY PRINCIPLE (SRP)



## Single Responsibility Principle

Just because you *can* doesn't mean you *should*.

# SINGLE RESPONSIBILITY PRINCIPLE (SRP)



## Single Responsibility Principle

Just because you *can* doesn't mean you *should*.

*A class should have only one reason to change*

# SINGLE RESPONSIBILITY PRINCIPLE (SRP)

# SINGLE RESPONSIBILITY PRINCIPLE (SRP)

```
/**  
 * Responsibility: Give a talk  
 */  
class speaker {  
    static constexpr auto name = "Kris"; // Tightly coupled  
  
public:  
    void talk();  
};
```

# SINGLE RESPONSIBILITY PRINCIPLE (SRP)

```
/**  
 * Responsibility: Give a talk  
 */  
class speaker {  
    static constexpr auto name = "Kris"; // Tightly coupled  
  
public:  
    void talk();  
};
```

```
/**  
 * Responsibility: Participate  
 */  
class attendees {  
    std::vector names = {"Lenny", "Shea", ...}; // Tightly coupled  
  
public:  
    auto ask();  
};
```

# SINGLE RESPONSIBILITY PRINCIPLE (SRP)

# SINGLE RESPONSIBILITY PRINCIPLE (SRP)

```
class cppnow_talk {
```

```
} ;
```

# SINGLE RESPONSIBILITY PRINCIPLE (SRP)

```
class cppnow_talk {  
  
    speaker speaker_{ };      // Tightly coupled  
    attendees attendees_{ };  // Tightly coupled
```

```
} ;
```

# SINGLE RESPONSIBILITY PRINCIPLE (SRP)

```
class cppnow_talk {  
  
    speaker speaker_{ };          // Tightly coupled  
    attendees attendees_{ };      // Tightly coupled  
  
public:  
    auto run() {  
        const auto before = attendees.ask();  
        speaker.talk();  
        const auto after = attendees.ask();  
        return after - before;  
    }  
};
```

# SINGLETON RESPONSIBILITY PRINCIPLE (SRC)

# SINGLETON RESPONSIBILITY PRINCIPLE (SRC)

- FLEXIBLE?

# SINGLETON RESPONSIBILITY PRINCIPLE (SRC)

- FLEXIBLE?
  - NOT REALLY: TIGHTLY COUPLED

# SINGLETON RESPONSIBILITY PRINCIPLE (SRC)

- FLEXIBLE?
  - NOT REALLY: TIGHTLY COUPLED
- SCALABLE?

# SINGLETON RESPONSIBILITY PRINCIPLE (SRC)

- FLEXIBLE?
  - NOT REALLY: TIGHTLY COUPLED
- SCALABLE?
  - NOT REALLY: NOT DON'T REPEAT YOURSELF (DRY)

# SINGLETON RESPONSIBILITY PRINCIPLE (SRC)

- FLEXIBLE?
  - NOT REALLY: TIGHTLY COUPLED
- SCALABLE?
  - NOT REALLY: NOT DON'T REPEAT YOURSELF (DRY)
- TESTABLE?

# SINGLETON RESPONSIBILITY PRINCIPLE (SRC)

- FLEXIBLE?
  - NOT REALLY: TIGHTLY COUPLED
- SCALABLE?
  - NOT REALLY: NOT DON'T REPEAT YOURSELF (DRY)
- TESTABLE?
  - NOT REALLY: HARD TO FAKE



**CONSIDER CLASSES TO HAVE ONLY ONE REASON TO  
CHANGE**

---



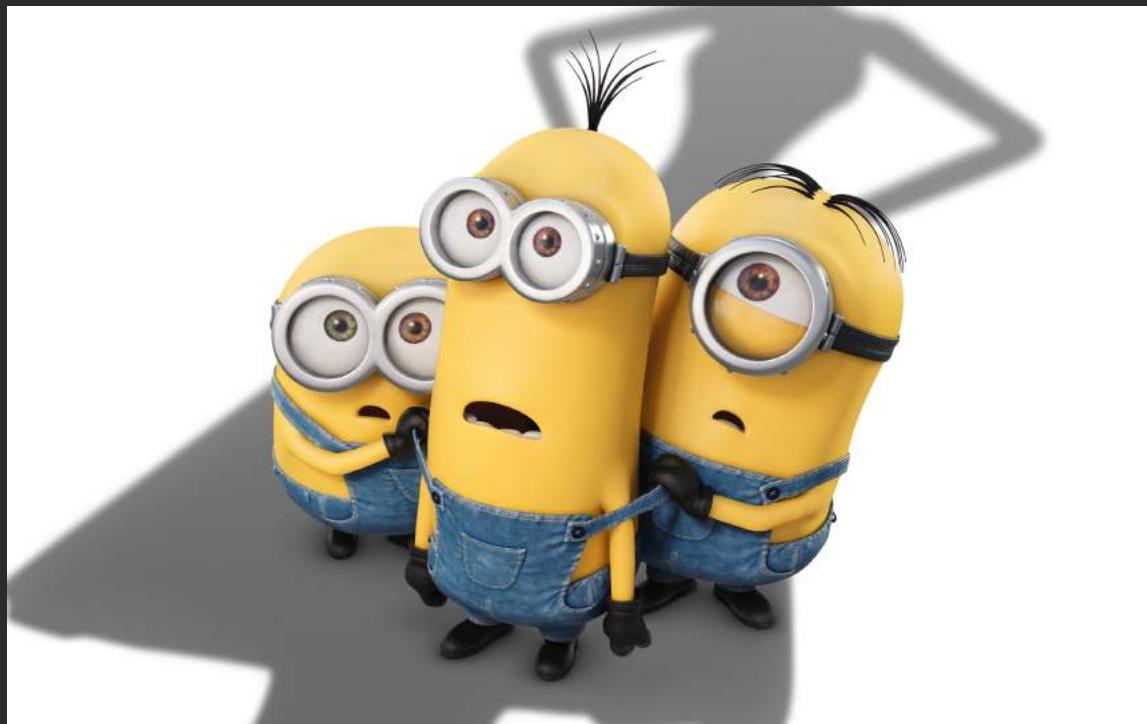
**CONSIDER CLASSES TO HAVE ONLY ONE REASON TO  
CHANGE**

---

**BUT WHAT ABOUT THE COUPLING?**

# DEPENDENCY INJECTION (DI) ?

# DEPENDENCY INJECTION (DI) ?



# DEPENDENCY INJECTION (DI) ?

---

---

# DEPENDENCY INJECTION (DI) ?

---

Design A way to reduce coupling...

---

# DEPENDENCY INJECTION (DI) ?

---

Design A way to reduce coupling...

---

C++ Constructors (simplified)

 **WHETHER DI IS DONE RIGHT DEPENDS ON WHAT AND HOW WILL BE PASSED INTO CONSTRUCTORS**



**DI DOESN'T IMPLY USING A LIBRARY/FRAMEWORK**

# TIGHT COUPLING - NO DI

# TIGHT COUPLING - NO DI

```
class speaker {  
    static constexpr auto name = "Kris"; // Tightly coupled  
  
public:  
    auto talk();  
};
```

# TIGHT COUPLING - NO DI

```
class speaker {  
    static constexpr auto name = "Kris"; // Tightly coupled  
  
public:  
    auto talk();  
};
```

```
class cppnow_talk {  
    speaker speaker_{ }; // Tightly coupled  
    attendees attendees_{ }; // Tightly coupled  
  
public:  
    auto run();  
}
```

# LESS COUPLING - CONSTRUCTOR DI

# LESS COUPLING - CONSTRUCTOR DI

```
class speaker {  
    std::string name_{};
```

```
    auto talk();  
};
```

# LESS COUPLING - CONSTRUCTOR DI

```
class speaker {  
    std::string name_{};  
  
public:  
    // ⚡ Dependency Injection!!!  
    explicit speaker(std::string_view name)  
        : name_{name}  
    {}  
  
    auto talk();  
};
```

# LESS COUPLING - CONSTRUCTOR DI

# LESS COUPLING - CONSTRUCTOR DI

```
class cppnow_talk {  
    speaker speaker_; // Tightly coupled?  
    attendees attendees_; // Tightly coupled?
```

```
    auto run();  
};
```

# LESS COUPLING - CONSTRUCTOR DI

```
class cppnow_talk {  
    speaker speaker_;      // Tightly coupled?  
    attendees attendees_; // Tightly coupled?  
  
public:  
    // 👍 Dependency Injection!!!  
    cppnow_talk(speaker speaker, attendees attendees)  
        : speaker_{speaker}, attendees_{attendees}  
    {}  
  
    auto run();  
};
```

# LESS COUPLING - CONSTRUCTOR DI

```
class cppnow_talk {  
    speaker speaker_;      // Tightly coupled?  
    attendees attendees_; // Tightly coupled?  
  
public:  
    // 👍 Dependency Injection!!!  
    cppnow_talk(speaker speaker, attendees attendees)  
        : speaker_{speaker}, attendees_{attendees}  
    {}  
  
    auto run();  
};
```

*"Don't call us, we'll call you", Hollywood principle*

# CONSTRUCTOR DI - WIRING

# CONSTRUCTOR DI - WIRING

```
int main() {
```

```
}
```

# CONSTRUCTOR DI - WIRING

```
int main() {  
  
    const auto speaker = speaker{"Kris"};  
    const auto attendees = attendees{"Lenny", "Shea", ...};  
  
}
```

# CONSTRUCTOR DI - WIRING

```
int main() {  
  
    const auto speaker = speaker{"Kris"};  
    const auto attendees = attendees{"Lenny", "Shea", ...};  
  
    cppnow_talk track{speaker, attendees};  
    return track.run();  
  
}
```

# WIRING

# WIRING

👉 SEPARATES THE CREATION LOGIC FROM THE BUSINESS LOGIC

# WIRING

👉 SEPARATES THE CREATION LOGIC FROM THE BUSINESS LOGIC

*No raw new/make\_unique/etc...  
except in the wiring*

# COMPOSITION ROOT

# COMPOSITION ROOT

*A unique location in an application where modules are composed together*

# COMPOSITION ROOT

# COMPOSITION ROOT

```
class cppnow_talk {
public:
    explicit cppnow_talk(speaker);
    auto run() {
        const auto attendees = attendees{"Lenny", "Shea", ...}; // ⚡
        // ...
    }
};
```

# COMPOSITION ROOT

```
class cppnow_talk {  
public:  
    explicit cppnow_talk(speaker);  
    auto run() {  
        const auto attendees = attendees{"Lenny", "Shea", ...}; // ⚡  
        // ...  
    }  
};
```

```
int main() {  
    const auto speaker = speaker{"Kris"};  
    cppnow_talk track{speaker};  
    // ...  
}
```

# COMPOSITION ROOT

```
class cppnow_talk {  
public:  
    explicit cppnow_talk(speaker);  
    auto run() {  
        const auto attendees = attendees{"Lenny", "Shea", ...}; // 👎  
        // ...  
    }  
};
```

```
int main() {  
    const auto speaker = speaker{"Kris"};  
    cppnow_talk track{speaker};  
    // ...  
}
```

---

```
int main() {  
    const auto speaker = speaker{"Kris"};  
    const auto attendees = attendees {"Lenny", "Shea", ...};  
    cppnow_talk track{speaker, attendees}; // 👍  
    // ...  
}
```

# CONSTRUCTOR DI - IMMEDIATE BENEFITS

# CONSTRUCTOR DI - IMMEDIATE BENEFITS

```
int main() {
```

```
}
```

# CONSTRUCTOR DI - IMMEDIATE BENEFITS

```
int main() {  
  
    cppnow_talk track1{ {"Kris"}, {"Lenny", "Shea", ...} };  
  
}
```

# CONSTRUCTOR DI - IMMEDIATE BENEFITS

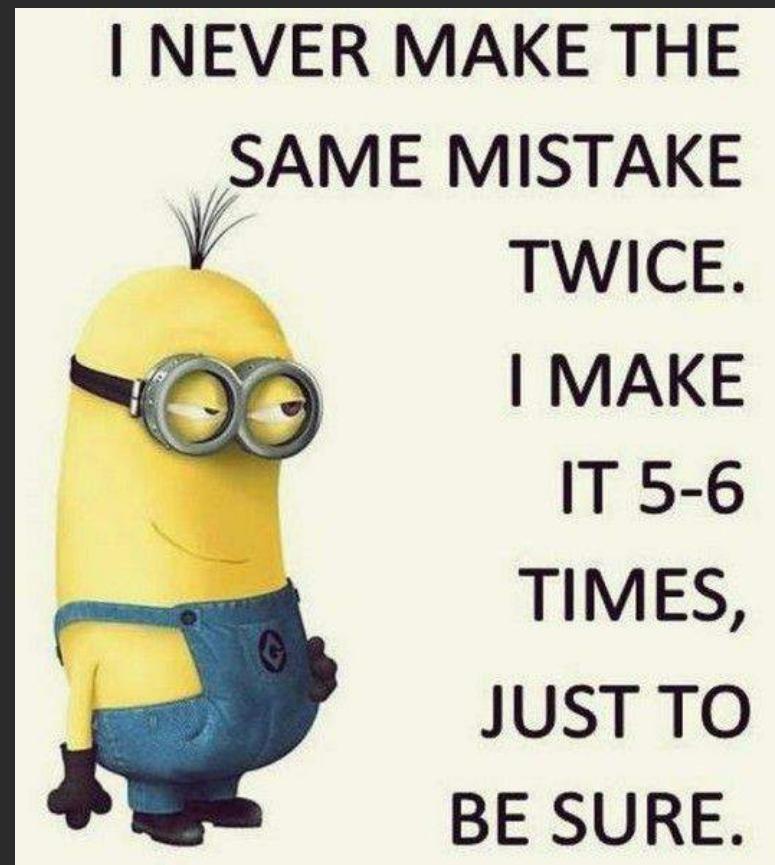
```
int main() {  
  
    cppnow_talk track1{ {"Kris"}, {"Lenny", "Shea", ...} };  
  
    cppnow_talk track2{ {"Timur"}, {"Ben", "Brian", ...} };  
  
}
```

# CONSTRUCTOR DI - IMMEDIATE BENEFITS

```
int main() {  
  
    cppnow_talk track1{{"Kris"}, {"Lenny", "Shea", ...}};  
  
    cppnow_talk track2{{"Timur"}, {"Ben", "Brian", ...}};  
  
    co_await track1.run();  
    co_await track2.run();  
  
}
```

# CONSTRUCTOR DI - GOTCHAS

# CONSTRUCTOR DI - GOTCHAS



# NOT USING CONSTRUCTORS CONSISTENTLY

# NOT USING CONSTRUCTORS CONSISTENTLY

```
class cppnow_talk {  
    speaker speaker_;                                // Tightly coupled?  
  
    auto run();  
};
```

# NOT USING CONSTRUCTORS CONSISTENTLY

```
class cppnow_talk {  
    speaker speaker_; // Tightly coupled?
```

```
public:  
    cppnow_talk() : speaker_("Kris") {} // Tightly coupled
```

```
    auto run();  
};
```



**CONSIDER USING CONSTRUCTOR DEPENDENCY  
INJECTION CONSISTENTLY**



# CONSIDER USING CONSTRUCTOR DEPENDENCY INJECTION CONSISTENTLY

```
cppnow_talk() : speaker_("Kris") {} // ⚡
```



# CONSIDER USING CONSTRUCTOR DEPENDENCY INJECTION CONSISTENTLY

```
cppnow_talk() : speaker_("Kris") {} // ⚡
```

```
cppnow_talk(speaker speaker) : speaker_{speaker} {} ; // 🌟
```

# USING SINGLETONS

# USING SINGLETONS

```
class cppnow_talk {  
public:  
};
```

# USING SINGLETONS

```
class cppnow_talk {
public:

    auto run() {
        ...
        speakers::instance().get("Kris").talk(); // how to test?
        ...
    }
};
```



**CONSIDER AVOIDING SINGLETONS (OR INJECT THEM  
VIA CONSTRUCTOR)**



# CONSIDER AVOIDING SINGLETONS (OR INJECT THEM VIA CONSTRUCTOR)

```
speakers::instance().get("Kris")
```

// 



# CONSIDER AVOIDING SINGLETONS (OR INJECT THEM VIA CONSTRUCTOR)

```
speakers::instance().get("Kris")
```

// 

```
cppnow_talk(speaker speaker) : speaker_{speaker} {};
```

// 

# CARRYING DEPENDENCIES

# CARRYING DEPENDENCIES

```
class cppnow_talk {  
    speaker speaker_; // Tightly coupled
```

```
    auto run();  
};
```

# CARRYING DEPENDENCIES

```
class cppnow_talk {  
    speaker speaker_; // Tightly coupled  
  
public:  
    // ⚡ Leaky abstraction  
    explicit cppnow_talk(string_view name)  
        : speaker_{name}  
    {}  
  
    auto run();  
};
```

 **CONSIDER PASSING INITIALIZED OBJECTS INSTEAD OF  
PARAMETERS TO INITIALIZE THEM**



# CONSIDER PASSING INITIALIZED OBJECTS INSTEAD OF PARAMETERS TO INITIALIZE THEM

```
explicit cppnow_talk(string_view name) : speaker{name} {} ; // 🙅
```



# CONSIDER PASSING INITIALIZED OBJECTS INSTEAD OF PARAMETERS TO INITIALIZE THEM

```
explicit cppnow_talk(string_view name) : speaker{name} {}; // 🙅
```

```
explicit cppnow_talk(speaker speaker) : speaker_{speaker} {}; // 👍
```

# CARRYING DEPENDENCIES WITH INHERITANCE

# CARRYING DEPENDENCIES WITH INHERITANCE

```
class cppnow_talk : speaker { // Tightly coupled to `speaker` API
```

```
    auto run();  
};
```

# CARRYING DEPENDENCIES WITH INHERITANCE

```
class cppnow_talk : speaker { // Tightly coupled to `speaker` API  
  
public:  
    explicit cppnow_talk(string_view name) // Common with CRTP?  
        : speaker{name}  
    {}  
  
    auto run();  
};
```



# PREFER COMPOSITION OVER INHERITANCE



# PREFER COMPOSITION OVER INHERITANCE

```
class cppnow_talk : speaker // ⚡
```



# PREFER COMPOSITION OVER INHERITANCE

```
class cppnow_talk : speaker // ⚡
```

```
class cppnow_talk { speaker speaker_; // 👍
```

# TALKING TO YOUR DISTANT FRIENDS

# TALKING TO YOUR DISTANT FRIENDS

```
class cppnow_talk {  
    speaker speaker_; // Tightly coupled
```

```
    auto run();  
};
```

# TALKING TO YOUR DISTANT FRIENDS

```
class cppnow_talk {
    speaker speaker_; // Tightly coupled

public:
    explicit cppnow_talk(talk_manager& mgr)
        // 👋 Distant friends
        : speaker_{mgr.get_speakers().get("Kris")}; // difficult to test
    { }

    auto run();
};
```



**CONSIDER TALKING ONLY TO YOUR IMMEDIATE  
FRIENDS (LAW OF DEMETER)**



# CONSIDER TALKING ONLY TO YOUR IMMEDIATE FRIENDS (LAW OF DEMETER)

```
speaker_{mgr.get_speakers().get("Kris")} // 🙋
```



# CONSIDER TALKING ONLY TO YOUR IMMEDIATE FRIENDS (LAW OF DEMETER)

```
speaker_{mgr.get_speakers().get("Kris")} // 👍
```

```
speaker_{speaker}; // 👍
```

# NOT USING STRONG TYPES

# NOT USING STRONG TYPES

```
// 🤦 Weak API  
speaker(string_view first_name, string_view last_name);
```

# NOT USING STRONG TYPES

```
// 🤦 Weak API  
speaker(string_view first_name, string_view last_name);  
  
speaker{"Kris", "Jusiak"}; // 👍 Okay
```

# NOT USING STRONG TYPES

```
// 🙅 Weak API  
speaker(string_view first_name, string_view last_name);
```

```
speaker{"Kris", "Jusiak"}; // 👍 Okay
```

```
speaker{"Jusiak", "Kris"}; // 🙅 Oops
```

# STRONG TYPES FOR STRONG INTERFACES

# STRONG TYPES FOR STRONG INTERFACES

The image shows a video player interface. On the left, a large slide is displayed with the title "STRONG TYPES FOR STRONG INTERFACES" in large, bold, orange and green letters. Below the title, the author's name "Jonathan Boccara" and handle "@JoBoccara" are shown. The slide also features the text "Fluent {C++}" and the Murex logo. On the right side of the video player, there is a dark sidebar with white text. The top part of the sidebar reads "Meeting C++ 2017" and "Jonathan Boccara". Below that, the title of the presentation is listed as "Strong Types For Strong Interfaces". At the bottom of the sidebar, a smaller video frame is visible, showing a person speaking at a podium with a screen behind them displaying the text "Put static type safety in code. The type system is there for you." The video player has a progress bar at the bottom left indicating it is at 0:03 / 51:33. At the bottom right, there are standard video control icons.

Meeting C++ 2017  
Jonathan Boccara

Strong Types For  
Strong Interfaces

Fluent {C++}

MUREX

Put static type safety in code.  
The type system is there for you.

51:00

▶ ▶ 🔍 0:03 / 51:33 CC HD ⌂

# STRONG TYPES FOR STRONG INTERFACES

# STRONG TYPES FOR STRONG INTERFACES

```
using first_name = named<string_view, "first name">;  
using last_name = named<string_view, "last name">;
```

# STRONG TYPES FOR STRONG INTERFACES

```
using first_name = named<string_view, "first name">;  
using last_name = named<string_view, "last name">;
```

```
// 👍 Strong API  
speaker(first_name, last_name);
```

# STRONG TYPES FOR STRONG INTERFACES

```
using first_name = named<string_view, "first name">;  
using last_name = named<string_view, "last name">;
```

```
// 👍 Strong API  
speaker(first_name, last_name);
```

```
speaker{first_name{"Kris"}, last_name{"Jusiak"} }; // 👍 Okay
```

# STRONG TYPES FOR STRONG INTERFACES

```
using first_name = named<string_view, "first name">;  
using last_name = named<string_view, "last name">;
```

```
// 👍 Strong API  
speaker(first_name, last_name);
```

```
speaker{first_name{"Kris"}, last_name{"Jusiak"}}; // 👍 Okay
```

```
speaker{last_name{"Jusiak"}, first_name{"Kris"}}; // 👎 Compile error
```

 CONSIDER USING STRONG TYPES

 **CONSIDER USING STRONG TYPES**

```
speaker(string_view first_name, string_view last_name); // ⌘
```

 **CONSIDER USING STRONG TYPES**

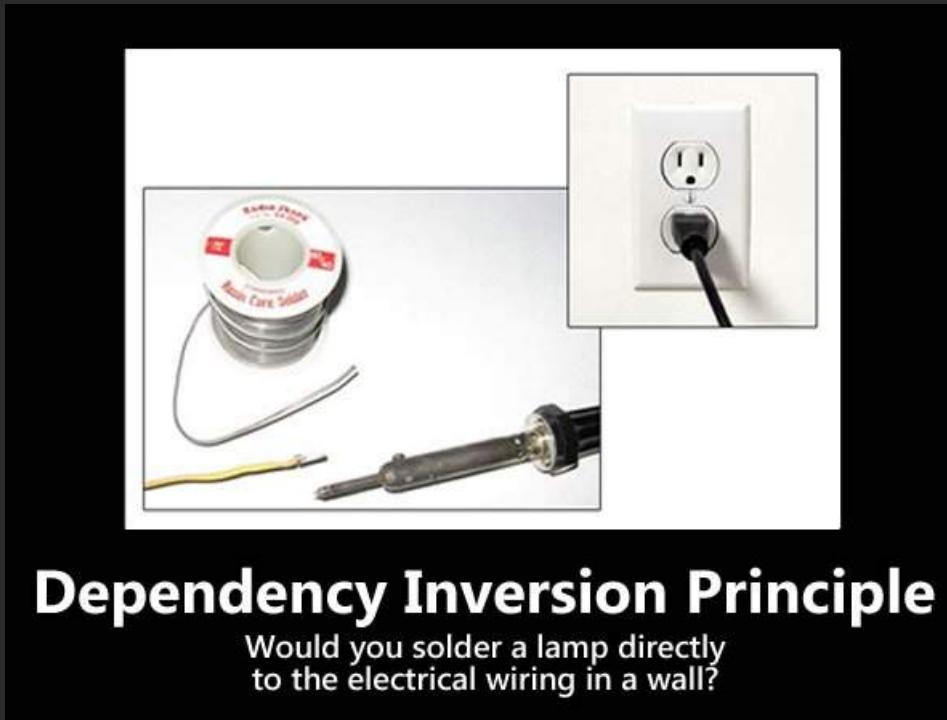
```
speaker(string_view first_name, string_view last_name); // 🤡
```

```
speaker(first_name, last_name); // 👍
```

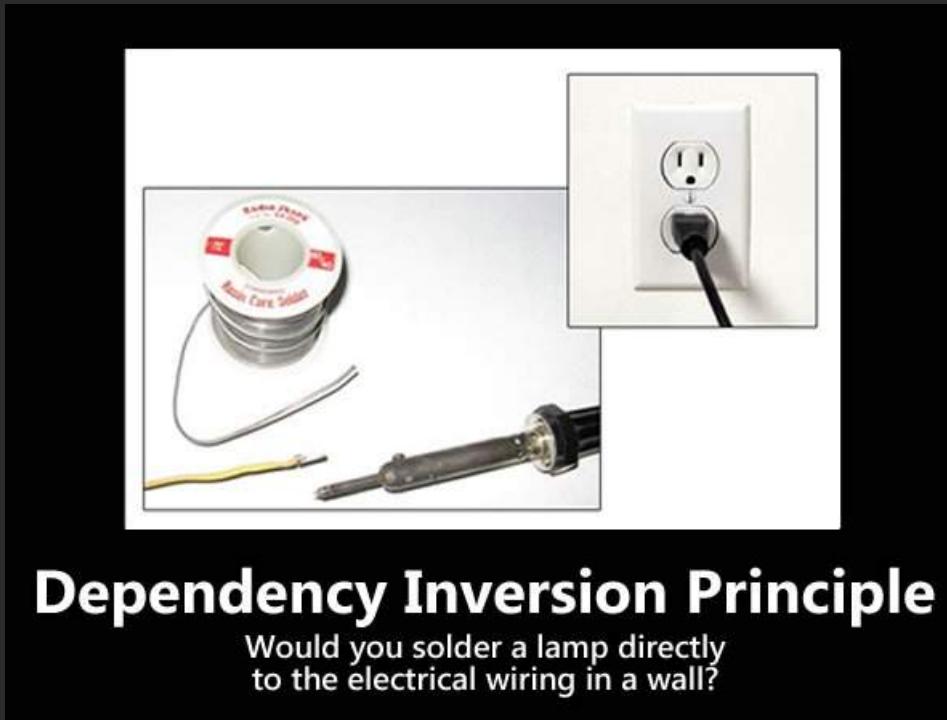
**LET'S MAKE IT** even more **FLEXIBLE** 

# DEPENDENCY INVERSION PRINCIPLE (DIP)

# DEPENDENCY INVERSION PRINCIPLE (DIP)



# DEPENDENCY INVERSION PRINCIPLE (DIP)



## Dependency Inversion Principle

Would you solder a lamp directly  
to the electrical wiring in a wall?

*Depends on abstractions, not on  
implementations*

# POLYMORPHISM IN C++

# POLYMORPHISM IN C++

- INHERITANCE

# POLYMORPHISM IN C++

- INHERITANCE
- TYPE-ERASURE

# POLYMORPHISM IN C++

- INHERITANCE
- TYPE-ERASURE
- STD::VARIANT/STD::ANY (C++17)

# POLYMORPHISM IN C++

- INHERITANCE
- TYPE-ERASURE
- STD::VARIANT/STD::ANY (C++17)
- TEMPLATES

# POLYMORPHISM IN C++

- INHERITANCE
- TYPE-ERASURE
- STD::VARIANT/STD::ANY (C++17)
- TEMPLATES
- CONCEPTS (C++20)

# POLYMORPHISM IN C++

- INHERITANCE
- TYPE-ERASURE
- STD::VARIANT/STD::ANY (C++17)
- TEMPLATES
- CONCEPTS (C++20)
- ...

# OBJECT ORIENTED DESIGN WITH DYNAMIC POLYMORPHISM

# OBJECT ORIENTED DESIGN WITH DYNAMIC POLYMORPHISM

```
class SpeakerLike {  
public:  
    virtual ~SpeakerLike() noexcept = default;  
    constexpr virtual void talk() = 0; // C++20  
};
```

# OBJECT ORIENTED DESIGN WITH DYNAMIC POLYMORPHISM

```
class SpeakerLike {  
public:  
    virtual ~SpeakerLike() noexcept = default;  
    constexpr virtual void talk() = 0; // C++20  
};
```

```
class regular_speaker : public SpeakerLike { // Coupling ↪  
public:  
    regular_speaker(first_name, last_name);  
    void talk() override final;  
};
```

# OO DESIGN WITH DYNAMIC POLYMORPHISM

# OO DESIGN WITH DYNAMIC POLYMORPHISM

```
class cppnow_talk {  
    std::unique_ptr<SpeakerLike> speaker;  
    std::shared_ptr<AttendeesLike> attendees;  
  
    auto run();  
};
```

# OO DESIGN WITH DYNAMIC POLYMORPHISM

```
class cppnow_talk {
    std::unique_ptr<SpeakerLike> speaker;
    std::shared_ptr<AttendeesLike> attendees;

public:
    cppnow_talk(std::unique_ptr<SpeakerLike> speaker,
                std::shared_ptr<AttendeesLike> attendees)
        : speaker_{std::move(speaker)}, attendees_{std::move(attendees)}
    {}

    auto run();
};
```

# OO DESIGN WITH DYNAMIC POLYMORPHISM - WIRING

# OO DESIGN WITH DYNAMIC POLYMORPHISM - WIRING

```
int main() {
```

```
}
```

# OO DESIGN WITH DYNAMIC POLYMORPHISM - WIRING

```
int main() {  
  
    auto speaker =  
        std::make_unique<regular_speaker>(first_name{ "Kris" },  
                                         last_name{ "Jusiak" } );  
  
}
```

# OO DESIGN WITH DYNAMIC POLYMORPHISM - WIRING

```
int main() {  
  
    auto speaker =  
        std::make_unique<regular_speaker>(first_name{ "Kris" },  
                                         last_name{ "Jusiak" }) ;  
  
    auto attendees =  
        std::make_shared<awesome_attendees>("Lenny", "Shea", ...);  
  
}
```

# OO DESIGN WITH DYNAMIC POLYMORPHISM - WIRING

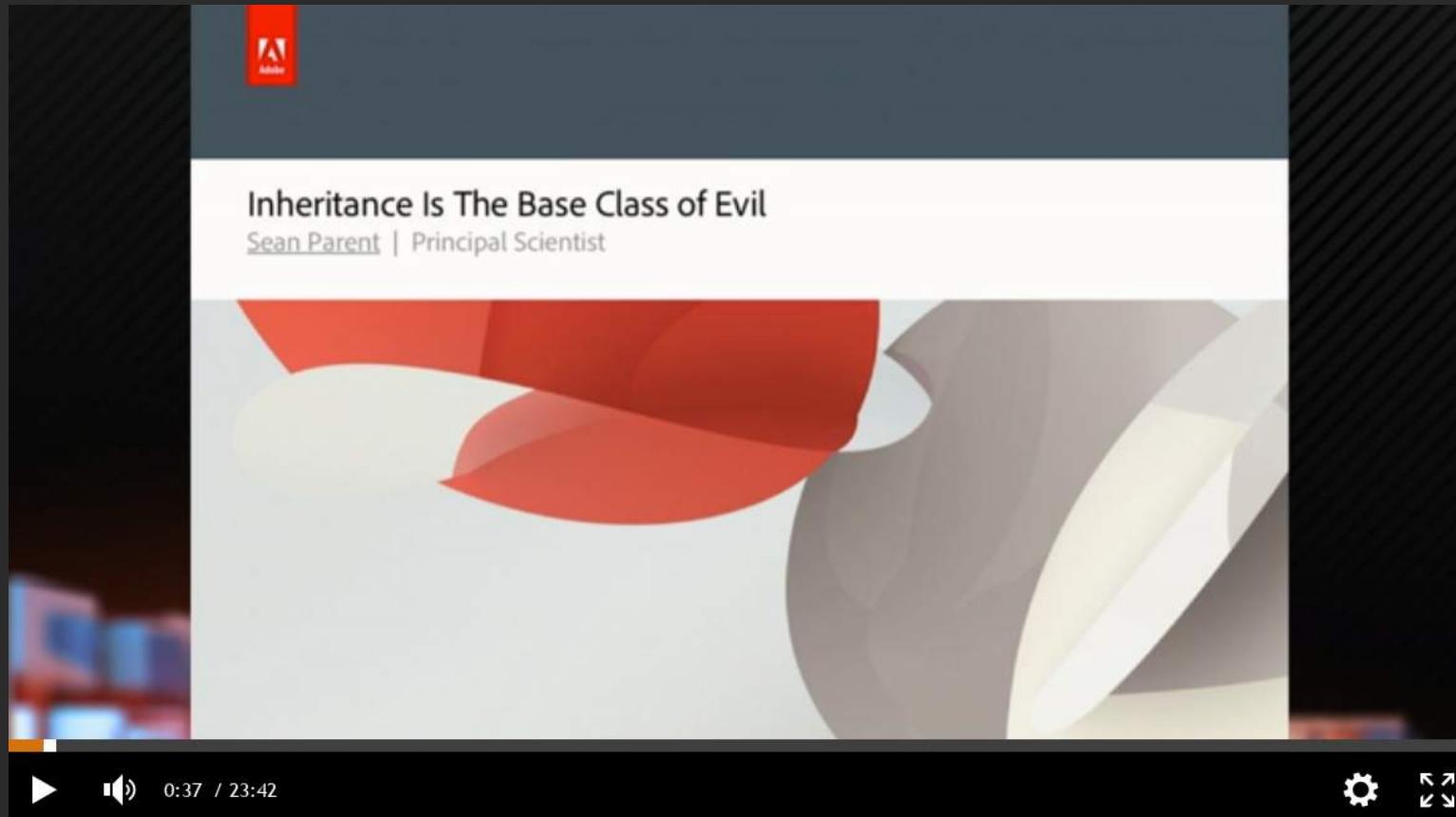
```
int main() {  
  
    auto speaker =  
        std::make_unique<regular_speaker>(first_name{"Kris"},  
                                         last_name{"Jusiak"});  
  
    auto attendees =  
        std::make_shared<awesome_attendees>("Lenny", "Shea", ...);  
  
    auto track =  
        cppnow_talk(std::move(speaker), attendees);  
  
}
```

# OO DESIGN WITH DYNAMIC POLYMORPHISM - WIRING

```
int main() {  
  
    auto speaker =  
        std::make_unique<regular_speaker>(first_name{ "Kris" },  
                                            last_name{ "Jusiak" }) ;  
  
    auto attendees =  
        std::make_shared<awesome_attendees>("Lenny", "Shea", ...);  
  
    auto track =  
        cppnow_talk(std::move(speaker), attendees);  
  
    return track.run();  
  
}
```

# INHERITANCE IS THE BASE CLASS OF EVIL, SEAN PARENT

# INHERITANCE IS THE BASE CLASS OF EVIL, SEAN PARENT





**CONSIDER NOT USING INHERITANCE FOR DYNAMIC  
POLYMORPHISM**

# TYPE-ERASURE - STD::FUNCTION

# TYPE-ERASURE - STD::FUNCTION

```
std::function speaker_talk = [](first_name, last_name) { ... };
```

# TYPE-ERASURE - STD::FUNCTION

```
std::function speaker_talk = [](first_name, last_name) { ... };  
std::function attendees_ask = [] { ... };
```

# TYPE-ERASURE - STD::FUNCTION

```
std::function speaker_talk = [](first_name, last_name) { ... };  
  
std::function attendees_ask = [] { ... };  
  
std::function cppnow_talk = [](auto talk, auto ask) { ... };
```

# FUNCTIONAL PROGRAMMING DESIGN PATTERNS, SCOTT WLASCHIN

# FUNCTIONAL PROGRAMMING DESIGN PATTERNS, SCOTT WLASCHIN



The image shows a video frame from the NDC 2014 conference. On the left, the NDC logo is displayed with the text "2014 new DevelopersConference(); 1-5 December · London, UK". Below the logo, the text "Inspiring Developers SINCE 2008" is visible. In the center, a man with a beard, identified as Scott Wlaschin, is speaking at a podium. The podium has the NDC logo on it. The video frame includes standard controls at the bottom: a play button, a progress bar showing 4:21 / 1:05:43, and a set of icons for volume, full screen, and other media functions.

OO pattern/principle	FP equivalent
• Single Responsibility Principle	• Functions
• Open/Closed principle	• Functions
• Dependency Inversion Principle	• Functions, also
• Interface Segregation Principle	• Functions
• Factory pattern	• You will be assimilated!
• Strategy pattern	• Functions again
• Decorator pattern	• Functions
• Visitor pattern	• Resistance is futile!

The list of patterns and principles is as follows:

- Single Responsibility Principle
- Open/Closed principle
- Dependency Inversion Principle
- Interface Segregation Principle
- Factory pattern
- Strategy pattern
- Decorator pattern
- Visitor pattern

The "Dependency Inversion Principle" and "Interface Segregation Principle" are circled in red. The "Functions" entries under "FP equivalent" are also circled in red.

# TYPE-ERASURE - STD::FUNCTION - WIRING

# TYPE-ERASURE - STD::FUNCTION - WIRING

```
int main() {
```

```
}
```

# TYPE-ERASURE - STD::FUNCTION - WIRING

```
int main() {  
  
    std::function talk = [first_name = "Kris", last_name = "Jusiak"] {  
        speaker_talk(first_name, last_name);  
    };  
  
}
```

# TYPE-ERASURE - STD::FUNCTION - WIRING

```
int main() {  
  
    std::function talk = [first_name = "Kris", last_name = "Jusiak"] {  
        speaker_talk(first_name, last_name);  
    };  
  
    std::function ask = [] { return attendees_ask(); };  
  
}
```

# TYPE-ERASURE - STD::FUNCTION - WIRING

```
int main() {  
  
    std::function talk = [first_name = "Kris", last_name = "Jusiak"] {  
        speaker_talk(first_name, last_name);  
    };  
  
    std::function ask = [] { return attendees_ask(); };  
  
    std::function track = [] { cppnow_talk(talk, ask); }  
  
}
```

# TYPE-ERASURE - STD::FUNCTION - WIRING

```
int main() {  
  
    std::function talk = [first_name = "Kris", last_name = "Jusiak"] {  
        speaker_talk(first_name, last_name);  
    };  
  
    std::function ask = [] { return attendees_ask(); };  
  
    std::function track = [] { cppnow_talk(talk, ask); }  
  
    return track();  
  
}
```

# TYPE-ERASURE

# TYPE-ERASURE

```
class regular_speaker { // No inheritance 👍
public:
    regular_speaker(first_name, last_name);
    void talk();
};
```

# TYPE-ERASURE - VIRTUAL CONCEPTS

# TYPE-ERASURE - VIRTUAL CONCEPTS

```
class cppnow_talk {  
    virtual speaker speaker_;           // Type-erased  
    virtual attendees attendees_;      // Type-erased
```

```
    auto run();  
};
```

# TYPE-ERASURE - VIRTUAL CONCEPTS

```
class cppnow_talk {
    virtual speaker speaker_;           // Type-erased
    virtual attendees attendees_;      // Type-erased

public:
    cppnow_talk(virtual speaker speaker,
                virtual attendees attendees)
        : speaker_(speaker), attendees_(attendees)
    {}

    auto run();
};
```

# TYPE-ERASURE - WIRING

# TYPE-ERASURE - WIRING

```
int main() {
```

```
}
```

# TYPE-ERASURE - WIRING

```
int main() {  
  
    speaker speaker =  
        regular_speaker{first_name{"Kris"}, last_name{"Jusiak"} } ;  
  
}
```

# TYPE-ERASURE - WIRING

```
int main() {  
  
    speaker speaker =  
        regular_speaker{first_name{"Kris"}, last_name{"Jusiak"} } ;  
  
    attendees attendees = awesome_attendees{"Lenny", "Shea", ...} ;  
  
}
```

# TYPE-ERASURE - WIRING

```
int main() {  
  
    speaker speaker =  
        regular_speaker{first_name{"Kris"}, last_name{"Jusiak"} } ;  
  
    attendees attendees = awesome_attendees{"Lenny", "Shea", ...} ;  
  
    auto track = cppnow_talk{speaker, attendees};  
  
}
```

# TYPE-ERASURE - WIRING

```
int main() {  
  
    speaker speaker =  
        regular_speaker{first_name{"Kris"}, last_name{"Jusiak"} } ;  
  
    attendees attendees = awesome_attendees {"Lenny", "Shea", ...} ;  
  
    auto track = cppnow_talk{speaker, attendees} ;  
  
    return track.run();  
  
}
```

# TYPE-ERASURE - WIRING

```
int main() {  
  
    speaker speaker =  
        regular_speaker{first_name{"Kris"}, last_name{"Jusiak"}};  
  
    attendees attendees = awesome_attendees{"Lenny", "Shea", ...};  
  
    auto track = cppnow_talk{speaker, attendees};  
  
    return track.run();  
  
}
```

*Virtual Concepts aren't part of C++ ISO*

# STD::VARIANT (C++17)

# STD::VARIANT (C++17)

```
class regular_speaker { // No inheritance 👍
public:
    regular_speaker(first_name, last_name);
    void talk();
};
```

# STD::VARIANT (C++17)

```
class regular_speaker { // No inheritance 👍
public:
    regular_speaker(first_name, last_name);
    void talk();
};
```

*Same as with Type-Erasure/Templates/Concepts!*

# STD::VARIANT (C++17)

# STD::VARIANT (C++17)

```
using SpeakerLike =  
    std::variant<regular_speaker, keynote_speaker>;  
  
using AttendeesLike =  
    std::variant<regular_attendees, awesome_attendees>;
```

# STD::VARIANT (C++17)

```
using SpeakerLike =
    std::variant<regular_speaker, keynote_speaker>;  
  
using AttendeesLike =
    std::variant<regular_attendees, awesome_attendees>;  
  
class cppnow_talk {  
  
    auto run(); // visit  
};
```

# STD::VARIANT (C++17)

```
using SpeakerLike =
    std::variant<regular_speaker, keynote_speaker>;  
  
using AttendeesLike =
    std::variant<regular_attendees, awesome_attendees>;  
  
class cppnow_talk {  
  
public:  
    cppnow_talk(SpeakerLike speaker, AttendeesLike attendees)
        : speaker_{speaker}, attendees_{attendees}
    {}  
  
    auto run(); // visit
};
```

# **STD::VARIANT (C++17) - WIRING**

# STD::VARIANT (C++17) - WIRING

```
int main() {
```

```
}
```

# STD::VARIANT (C++17) - WIRING

```
int main() {  
  
    auto speaker =  
        regular_speaker{first_name{"Kris"}, last_name{"Jusiak"} };  
  
}
```

# STD::VARIANT (C++17) - WIRING

```
int main() {  
  
    auto speaker =  
        regular_speaker{first_name{"Kris"}, last_name{"Jusiak"} };  
  
    auto attendees = awesome_attendees{"Lenny", "Shea", ... };  
  
}
```

# STD::VARIANT (C++17) - WIRING

```
int main() {  
  
    auto speaker =  
        regular_speaker{first_name{"Kris"}, last_name{"Jusiak"} } ;  
  
    auto attendees = awesome_attendees{"Lenny", "Shea", ...} ;  
  
    auto track = cppnow_talk{speaker, attendees};  
  
}
```

# STD::VARIANT (C++17) - WIRING

```
int main() {  
  
    auto speaker =  
        regular_speaker{first_name{"Kris"}, last_name{"Jusiak"} } ;  
  
    auto attendees = awesome_attendees{"Lenny", "Shea", ...} ;  
  
    auto track = cppnow_talk{speaker, attendees};  
  
    return track.run();  
  
}
```

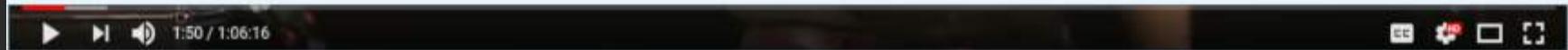
# POLICY DESIGN / DESIGN BY INTROSPECTION

# POLICY DESIGN / DESIGN BY INTROSPECTION

## Design by Introspection DConf 2017

Andrei Alexandrescu, Ph.D.

2017-05-06



# STATIC POLYMORPHISM

# STATIC POLYMORPHISM

```
class regular_speaker { // No inheritance ↗
public:
    regular_speaker(first_name, last_name);
    void talk();
};
```

# STATIC POLYMORPHISM

```
class regular_speaker { // No inheritance 👍
public:
    regular_speaker(first_name, last_name);
    void talk();
};
```

*Same as with Type-Erasure!*

# **STATIC POLYMORPHISM**

# STATIC POLYMORPHISM

```
template<class TSpeaker, class TAttendees>
class cppnow_talk {
    TSpeaker speaker_;
    TAttendees attendees_;

    auto run();
};
```

# STATIC POLYMORPHISM

```
template<class TSpeaker, class TAttendees>
class cppnow_talk {
    TSpeaker speaker_;
    TAttendees attendees_;

public:
    cppnow_talk(TSpeaker speaker, TAttendees attendees)
        : speaker_(speaker), attendees_(attendees)
    {}

    auto run();
};
```

# STATIC POLYMORPHISM - WIRING

# STATIC POLYMORPHISM - WIRING

```
int main() {
```

```
}
```

# STATIC POLYMORPHISM - WIRING

```
int main() {  
  
    auto speaker =  
        regular_speaker{first_name{"Kris"}, last_name{"Jusiak"} };  
  
}
```

# STATIC POLYMORPHISM - WIRING

```
int main() {  
  
    auto speaker =  
        regular_speaker{first_name{"Kris"}, last_name{"Jusiak"} };  
  
    auto attendees = awesome_attendees{"Lenny", "Shea", ... };  
  
}
```

# STATIC POLYMORPHISM - WIRING

```
int main() {  
  
    auto speaker =  
        regular_speaker{first_name{"Kris"}, last_name{"Jusiak"} };  
  
    auto attendees = awesome_attendees{"Lenny", "Shea", ...};  
  
    auto track = cppnow_talk<decltype(speaker), decltype(attendees)>{  
        speaker, attendees  
    };  
  
}
```

# STATIC POLYMORPHISM - WIRING

```
int main() {  
  
    auto speaker =  
        regular_speaker{first_name{"Kris"}, last_name{"Jusiak"} } ;  
  
    auto attendees = awesome_attendees {"Lenny", "Shea", ...} ;  
  
    auto track = cppnow_talk<decltype(speaker), decltype(attendees)>{  
        speaker, attendees  
    } ;  
  
    return track.run();  
  
}
```

# STATIC POLYMORPHISM - WIRING

```
int main() {  
  
    auto speaker =  
        regular_speaker{first_name{"Kris"}, last_name{"Jusiak"} };  
  
    auto attendees = awesome_attendees{"Lenny", "Shea", ...};  
  
    auto track = cppnow_talk<decltype(speaker), decltype(attendees)>{  
        speaker, attendees  
    };  
  
    return track.run();  
  
}
```

*Most dependencies are known at compile time!*

# CONCEPTS (C++20)

# CONCEPTS (C++20)

```
template <class TSpeaker>
concept SpeakerLike = requires(TSpeaker speaker) {
    { speaker.talk() } -> void;
};
```

# CONCEPTS (C++20)

```
template <class TSpeaker>
concept SpeakerLike = requires(TSpeaker speaker) {
    { speaker.talk() } -> void;
};
```

```
class regular_speaker { // No inheritance 👍
public:
    regular_speaker(first_name, last_name);
    void talk();
};
```

# CONCEPTS (C++20)

```
template <class TSpeaker>
concept SpeakerLike = requires(TSpeaker speaker) {
    { speaker.talk() } -> void;
};
```

```
class regular_speaker { // No inheritance 👍
public:
    regular_speaker(first_name, last_name);
    void talk();
};
```

*Same as with Type-Erasure/Templates!*

# CONCEPTS (C++20)

# CONCEPTS (C++20)

```
template<SpeakerLike TSpeaker, AttendeesLike TAttendees>
```

# CONCEPTS (C++20)

```
template<SpeakerLike TSpeaker, AttendeesLike TAttendees>
```

```
class cppnow_talk {
    TSpeaker speaker_;
    TAttendees attendees_;
```

```
    auto run();
};
```

# CONCEPTS (C++20)

```
template<SpeakerLike TSpeaker, AttendeesLike TAttendees>
```

```
class cppnow_talk {
    TSpeaker speaker_;
    TAttendees attendees_;

public:
    cppnow_talk(TSpeaker speaker, TAttendees attendees)
        : speaker_{speaker}, attendees_{attendees}
    { }
```

```
    auto run();
};
```

# CONCEPTS (C++20) - WIRING

# CONCEPTS (C++20) - WIRING

```
int main() {
```

```
}
```

# CONCEPTS (C++20) - WIRING

```
int main() {  
  
    SpeakerLike speaker =  
        regular_speaker{first_name{"Kris"}, last_name{"Jusiak"}};  
  
}
```

# CONCEPTS (C++20) - WIRING

```
int main() {  
  
    SpeakerLike speaker =  
        regular_speaker{first_name{"Kris"}, last_name{"Jusiak"} };  
  
    AttendeesLike attendees = awesome_attendees {"Lenny", "Shea", ...};  
  
}
```

# CONCEPTS (C++20) - WIRING

```
int main() {  
  
    SpeakerLike speaker =  
        regular_speaker{first_name{"Kris"}, last_name{"Jusiak"} };  
  
    AttendeesLike attendees = awesome_attendees{"Lenny", "Shea", ...};  
  
    auto track = cppnow_talk{speaker, attendees};  
  
}
```

# CONCEPTS (C++20) - WIRING

```
int main() {
```

```
    SpeakerLike speaker =  
        regular_speaker{first_name{"Kris"}, last_name{"Jusiak"}};
```

```
    AttendeesLike attendees = awesome_attendees{"Lenny", "Shea", ...};
```

```
    auto track = cppnow_talk{speaker, attendees};
```

```
    return track.run();
```

```
}
```

**LET'S MAKE IT TESTABLE** 

# TEST DRIVEN DEVELOPMENT (TDD + DIP)

# TEST DRIVEN DEVELOPMENT (TDD + DIP)

```
"should return true if number of attendees  
willing to using DI increased"_test = [] {  
  
};
```

# TEST DRIVEN DEVELOPMENT (TDD + DIP)

```
"should return true if number of attendees  
willing to using DI increased"_test = [] {
```

```
EXPECT(track.run());
```

```
};
```

# TEST DRIVEN DEVELOPMENT (TDD + DIP)

```
"should return true if number of attendees  
willing to using DI increased"_test = [] {
```

```
    EXPECT_CALL(speaker(talk)).Times(1);  
    EXPECT_CALL(attendees(ask)).Times(2).WillOnce(Return(1))  
        .WillOnce(Return(2));
```

```
    EXPECT(track.run());
```

```
};
```

# TEST DRIVEN DEVELOPMENT (TDD + DIP)

```
"should return true if number of attendees  
willing to using DI increased"_test = [] {  
  
fake_speaker speaker{}; // Wiring  
fake_attendees attendees{}; // Wiring  
cppnow_talk track{speaker, attendees}; // Wiring  
  
EXPECT_CALL(speaker(talk)).Times(1);  
EXPECT_CALL(attendees(ask)).Times(2).WillOnce(Return(1))  
    .WillOnce(Return(2));  
  
EXPECT(track.run());  
  
};
```

# DEPENDENCY INVERSION PRINCIPLE

# DEPENDENCY INVERSION PRINCIPLE

- FLEXIBLE?

# DEPENDENCY INVERSION PRINCIPLE

- FLEXIBLE?
  - YES: LOOSELY COUPLED

# DEPENDENCY INVERSION PRINCIPLE

- FLEXIBLE?
  - YES: LOOSELY COUPLED
- SCALABLE?

# DEPENDENCY INVERSION PRINCIPLE

- FLEXIBLE?
  - YES: LOOSELY COUPLED
- SCALABLE?
  - ???

# DEPENDENCY INVERSION PRINCIPLE

- FLEXIBLE?
  - YES: LOOSELY COUPLED
- SCALABLE?
  - ???
- TESTABLE?

# DEPENDENCY INVERSION PRINCIPLE

- FLEXIBLE?
  - YES: LOOSELY COUPLED
- SCALABLE?
  - ???
- TESTABLE?
  - YES: WE CAN INJECT MOCKS/FAKES/STUBS



# CONSIDER USING PROPER ABSTRACTIONS FOR YOUR PROJECT/MODULE (DIP)



# CONSIDER USING PROPER ABSTRACTIONS FOR YOUR PROJECT/MODULE (DIP)

- Templates/Concepts



# CONSIDER USING PROPER ABSTRACTIONS FOR YOUR PROJECT/MODULE (DIP)

- Templates/Concepts
  - Dependencies known at compile time



# CONSIDER USING PROPER ABSTRACTIONS FOR YOUR PROJECT/MODULE (DIP)

- Templates/Concepts
  - Dependencies known at compile time
- Type-Erasure



# CONSIDER USING PROPER ABSTRACTIONS FOR YOUR PROJECT/MODULE (DIP)

- Templates/Concepts
  - Dependencies known at compile time
- Type-Erasure
  - Run-Time dependency



# CONSIDER USING PROPER ABSTRACTIONS FOR YOUR PROJECT/MODULE (DIP)

- Templates/Concepts
  - Dependencies known at compile time
- Type-Erasure
  - Run-Time dependency
- Inheritance



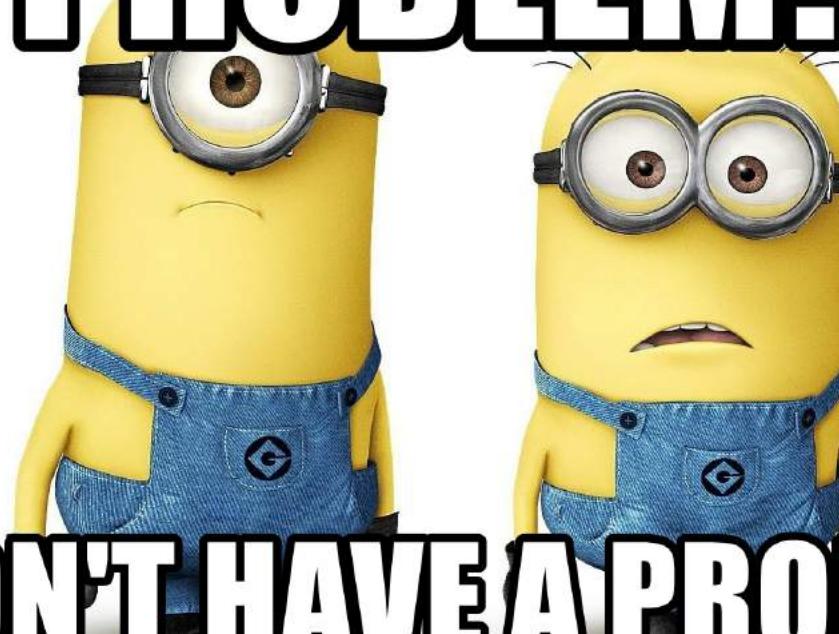
# CONSIDER USING PROPER ABSTRACTIONS FOR YOUR PROJECT/MODULE (DIP)

- Templates/Concepts
  - Dependencies known at compile time
- Type-Erasure
  - Run-Time dependency
- Inheritance
  - Never?

# PROBLEM?

PROBLEM?

PROBLEM?



I DON'T HAVE A PROBLEM

memegenerator.net

# WHAT ALL THESE SOLUTIONS HAVE IN COMMON?

# **MANUAL WIRING**

# MANUAL WIRING

```
int main() {  
    auto speaker = std::make_unique<regular_speaker>(  
        first_name{"Kris"}, last_name{"Jusiak"}); // Wiring  
    auto attendees = std::make_shared<awesome_attendees>(  
        "Lenny", "Shea", ...); // Wiring  
    auto track = cppnow_talk(std::move(speaker), attendees); // Wiring  
}
```

# MANUAL WIRING

```
int main() {  
    auto speaker = std::make_unique<regular_speaker>(  
        first_name{"Kris"}, last_name{"Jusiak"}); // Wiring  
    auto attendees = std::make_shared<awesome_attendees>(  
        "Lenny", "Shea", ...); // Wiring  
    auto track = cppnow_talk(std::move(speaker), attendees); // Wiring  
}
```

```
int main() {  
    std::function talk = bind(speaker_talk, "Kris", "Jusiak"); // Wiring  
    std::function ask = attendees_ask; // Wiring  
    auto track = bind(cppnow_talk, talk, ask); // Wiring  
}
```

# **MANUAL WIRING**

# MANUAL WIRING

```
int main() {  
    speaker speaker = regular_speaker{  
        first_name{"Kris"}, last_name{"Jusiak"} } ; // Wiring  
    attendees attendees = awesome_attendees{  
        "Lenny", "Shea", ... } ; // Wiring  
    auto track = cppnow_talk{speaker, attendees} ; // Wiring  
}
```

# MANUAL WIRING

```
int main() {
    speaker speaker = regular_speaker{
        first_name{"Kris"}, last_name{"Jusiak"} }; // Wiring
    attendees attendees = awesome_attendees{
        "Lenny", "Shea", ... }; // Wiring
    auto track = cppnow_talk{speaker, attendees}; // Wiring
}
```

```
int main() {
    SpeakerLike speaker = regular_speaker{
        first_name{"Kris"}, last_name{"Jusiak"} }; // Wiring
    AttendeesLike attendees = awesome_attendees{
        "Lenny", "Shea", ... }; // Wiring
    auto track = cppnow_talk{speaker, attendees}; // Wiring
}
```

# WIRING MESS

# WIRING MESS

```
void Configure()
{
    auto vbFactory = CreateVirusBasesFactory();
    auto vbConfig = LoadVirusBasesConfiguration();
    auto virusBases = vbFactory->CreateVirusBases(vbConfig);
    auto recoveryManager = TryCreateRecoveryManager();
    if (!recoveryManager)
        LogWarningRecoveryManagerNotAvailable();
    auto amDetector = make_shared<AntimalwareDetector>(virusBases);
    auto arDetector = make_shared<AntiRootkitDetector>(virusBases, amDetector);
    auto webAnalyzer = make_shared<WebAnalyzer>(amDetector, recoveryManager);
    auto mailAnalyzer = make_shared<MailAnalyzer>(amDetector, recoveryManager, webAnalyzer);
    auto trafficProcessor = make_unique<TrafficProcessor>(arDetector, webAnalyzer, mailAnalyzer);
    // ...
}
```

# WIRING MESS

```
void Configure()
{
    auto vbFactory = CreateVirusBasesFactory();
    auto vbConfig = LoadVirusBasesConfiguration();
    auto virusBases = vbFactory->CreateVirusBases(vbConfig);
    auto recoveryManager = TryCreateRecoveryManager();
    if (!recoveryManager)
        LogWarningRecoveryManagerNotAvailable();
    auto amDetector = make_shared<AntimalwareDetector>(virusBases);
    auto arDetector = make_shared<AntiRootkitDetector>(virusBases, amDetector);
    auto webAnalyzer = make_shared<WebAnalyzer>(amDetector, recoveryManager);
    auto mailAnalyzer = make_shared<MailAnalyzer>(amDetector, recoveryManager, webAnalyzer);
    auto trafficProcessor = make_unique<TrafficProcessor>(arDetector, webAnalyzer, mailAnalyzer);
    // ...
}
```

- BOILERPLATE / TEDIOUS

# WIRING MESS

```
void Configure()
{
    auto vbFactory = CreateVirusBasesFactory();
    auto vbConfig = LoadVirusBasesConfiguration();
    auto virusBases = vbFactory->CreateVirusBases(vbConfig);
    auto recoveryManager = TryCreateRecoveryManager();
    if (!recoveryManager)
        LogWarningRecoveryManagerNotAvailable();
    auto amDetector = make_shared<AntimalwareDetector>(virusBases);
    auto arDetector = make_shared<AntiRootkitDetector>(virusBases, amDetector);
    auto webAnalyzer = make_shared<WebAnalyzer>(amDetector, recoveryManager);
    auto mailAnalyzer = make_shared<MailAnalyzer>(amDetector, recoveryManager, webAnalyzer);
    auto trafficProcessor = make_unique<TrafficProcessor>(arDetector, webAnalyzer, mailAnalyzer);
    // ...
}
```

- BOILERPLATE / TEDIOUS
- HARD TO MAINTAIN (ORDER IS IMPORTANT)

# WIRING MESS

```
void Configure()
{
    auto vbFactory = CreateVirusBasesFactory();
    auto vbConfig = LoadVirusBasesConfiguration();
    auto virusBases = vbFactory->CreateVirusBases(vbConfig);
    auto recoveryManager = TryCreateRecoveryManager();
    if (!recoveryManager)
        LogWarningRecoveryManagerNotAvailable();
    auto amDetector = make_shared<AntimalwareDetector>(virusBases);
    auto arDetector = make_shared<AntiRootkitDetector>(virusBases, amDetector);
    auto webAnalyzer = make_shared<WebAnalyzer>(amDetector, recoveryManager);
    auto mailAnalyzer = make_shared<MailAnalyzer>(amDetector, recoveryManager, webAnalyzer);
    auto trafficProcessor = make_unique<TrafficProcessor>(arDetector, webAnalyzer, mailAnalyzer);
    // ...
}
```

- BOILERPLATE / TEDIOUS
- HARD TO MAINTAIN (ORDER IS IMPORTANT)
- MULTIPLE CONFIGURATIONS (RELEASE, DEBUG, TESTING)

# WIRING MESS

# WIRING MESS

- SINGLE RESPONSIBILITY

# WIRING MESS

- SINGLE RESPONSIBILITY
  - A LOT OF CLASSES =>

# WIRING MESS

- SINGLE RESPONSIBILITY
  - A LOT OF CLASSES =>
    - DEPENDENCY INVERSION/INJECTION =>

# WIRING MESS

- SINGLE RESPONSIBILITY
  - A LOT OF CLASSES =>
    - DEPENDENCY INVERSION/INJECTION =>
    - WIRING MESS =>

# WIRING MESS

- SINGLE RESPONSIBILITY
  - A LOT OF CLASSES =>
    - DEPENDENCY INVERSION/INJECTION =>
    - WIRING MESS =>
    - HARD TO MAINTAIN + DEADLINES / PRESSURE =>

# WIRING MESS

- SINGLE RESPONSIBILITY
  - A LOT OF CLASSES =>
    - DEPENDENCY INVERSION/INJECTION =>
    - WIRING MESS =>
    - HARD TO MAINTAIN + DEADLINES / PRESSURE =>
    - HACKS / WORKAROUNDS =>

# WIRING MESS

- SINGLE RESPONSIBILITY
  - A LOT OF CLASSES =>
    - DEPENDENCY INVERSION/INJECTION =>
    - WIRING MESS =>
    - HARD TO MAINTAIN + DEADLINES / PRESSURE =>
    - HACKS / WORKAROUNDS =>
      - ~~SRP / DI~~

# SOLUTION

# SOLUTION



# SOLUTION



SIMPLIFY/REMOVE THE WIRING MESS BY AUTOMATING DEPENDENCY INJECTION

# DEPENDENCY INJECTION LIBRARIES



# DEPENDENCY INJECTION LIBRARIES



*Often called Inversion of  
Control Containers*

# GOAL

# GOAL

- REMOVE BOILERPLATE/WIRING MESS

# GOAL

- REMOVE BOILERPLATE/WIRING MESS
  - CONSTRUCTOR PARAMETERS DEDUCTION

# GOAL

- REMOVE BOILERPLATE/WIRING MESS
  - CONSTRUCTOR PARAMETERS DEDUCTION
  - MAINTAIN/DEDUCE OBJECTS LIFE TIME

# GOAL

- REMOVE BOILERPLATE/WIRING MESS
  - CONSTRUCTOR PARAMETERS DEDUCTION
  - MAINTAIN/DEDUCE OBJECTS LIFE TIME
  - CREATE OBJECT GRAPH

# WRITING A DI LIBRARY AIN'T EASY IN C++

# WRITING A DI LIBRARY AIN'T EASY IN C++

- NO REFLECTION

# WRITING A DI LIBRARY AIN'T EASY IN C++

- NO REFLECTION
- RVALUES/LVALUES/PVALUES/...

# WRITING A DI LIBRARY AIN'T EASY IN C++

- NO REFLECTION
- RVALUES/LVALES/PVALUES/...
- EAST CONST/WEST CONST

# WRITING A DI LIBRARY AIN'T EASY IN C++

- NO REFLECTION
- RVALUES/LVALES/PVALUES/...
- EAST CONST/WEST CONST
- ERROR REPORTING (CONSTRUCTORS)

# WRITING A DI LIBRARY AIN'T EASY IN C++

- NO REFLECTION
- RVALUES/LVALES/PVALUES/...
- EAST CONST/WEST CONST
- ERROR REPORTING (CONSTRUCTORS)
- DON'T PAY FOR WHAT YOU DON'T USE

# WRITING A DI LIBRARY AIN'T EASY IN C++

- NO REFLECTION
- RVALUES/LVALES/PVALUES/...
- EAST CONST/WEST CONST
- ERROR REPORTING (CONSTRUCTORS)
- DON'T PAY FOR WHAT YOU DON'T USE
- ...

# APPROACHES

# APPROACHES

- COMPILE-TIME (COMPILE TIME ERROR)

# APPROACHES

- COMPILE-TIME (COMPILE TIME ERROR)
- RUNTIME/COMPILE-TIME (SOMETIMES COMPILE TIME ERROR/SOMETIMES EXCEPTIONS)

# APPROACHES

- COMPILE-TIME (COMPILE TIME ERROR)
- RUNTIME/COMPILE-TIME (SOMETIMES COMPILE TIME ERROR/SOMETIMES EXCEPTIONS)
- RUNTIME (EXCEPTIONS)

 **PREFER COMPILE- AND LINK-TIME ERRORS TO RUN-TIME ERRORS**

# OVERVIEW

# HYPODERMIC

---

# HYPODERMIC

---

- HEADER ONLY
-

# HYPODERMIC

---

- HEADER ONLY
  - LICENSE: MIT
-

# HYPODERMIC

---

- HEADER ONLY
  - LICENSE: MIT
  - DEPENDENCIES: STL, BOOST
-

# HYPODERMIC

---

- HEADER ONLY
  - LICENSE: MIT
  - DEPENDENCIES: STL, BOOST
  - STD: C++11
-

# HYPODERMIC

---

- HEADER ONLY
  - LICENSE: MIT
  - DEPENDENCIES: STL, BOOST
  - STD: C++11
  - SUPPORTED APPROACHES: RUN-TIME
-

# GOOGLE.FRUIT

---

# GOOGLE.FRUIT

---

- LIBRARY
-

# GOOGLE.FRUIT

---

- LIBRARY
  - LICENSE: APACHE-2
-

# GOOGLE.FRUIT

---

- LIBRARY
  - LICENSE: APACHE-2
  - DEPENDENCIES: STL, [BOOST]
-

# GOOGLE.FRUIT

---

- LIBRARY
  - LICENSE: APACHE-2
  - DEPENDENCIES: STL, [BOOST]
  - STD: C++11
-

# GOOGLE.FRUIT

---

- LIBRARY
  - LICENSE: APACHE-2
  - DEPENDENCIES: STL, [BOOST]
  - STD: C++11
  - SUPPORTED APPROACHES: SEMI COMPILE-TIME
-

# [BOOST].DI

---

# [BOOST].DI

---

- HEADER ONLY (SINGLE HEADER)
-

# [BOOST].DI

---

- HEADER ONLY (SINGLE HEADER)
  - LICENSE: BOOST 1.0
-

# [BOOST].DI

---

- HEADER ONLY (SINGLE HEADER)
  - LICENSE: BOOST 1.0
  - DEPENDENCIES: NONE (NEITHER STL NOR BOOST)
-

# [BOOST].DI

---

- HEADER ONLY (SINGLE HEADER)
  - LICENSE: BOOST 1.0
  - DEPENDENCIES: NONE (NEITHER STL NOR BOOST)
  - STD: C++14
-

# [BOOST].DI

---

- HEADER ONLY (SINGLE HEADER)
  - LICENSE: BOOST 1.0
  - DEPENDENCIES: NONE (NEITHER STL NOR BOOST)
  - STD: C++14
  - SUPPORTED APPROACHES: COMPILE-TIME/RUN-TIME
-

# [BOOST].DI

---

- HEADER ONLY (SINGLE HEADER)
  - LICENSE: BOOST 1.0
  - DEPENDENCIES: NONE (NEITHER STL NOR BOOST)
  - STD: C++14
  - SUPPORTED APPROACHES: COMPILE-TIME/RUN-TIME
- 

**Disclaimer** [Boost] . DI is not an official Boost library

LET'S MAKE IT SCALABLE



# LET'S MAKE IT SCALABLE



*With IoC containers*

# HYPODERMIC - 00

# HYPODERMIC - 00

```
class regular_speaker : public SpeakerLike {  
public:  
    regular_speaker(std::shared_ptr<first_name>,  
                    std::shared_ptr<last_name>);  
    void talk() override final;  
};
```

# HYPODERMIC - 00

```
class regular_speaker : public SpeakerLike {  
public:  
    regular_speaker(std::shared_ptr<first_name>,  
                    std::shared_ptr<last_name>);  
    void talk() override final;  
};
```

```
class cppnow_talk {  
    std::shared_ptr<SpeakerLike> speaker_;  
    std::shared_ptr<AttendeesLike> attendees_;
```

```
    auto run();  
}
```

# HYPODERMIC - 00

```
class regular_speaker : public SpeakerLike {  
public:  
    regular_speaker(std::shared_ptr<first_name>,  
                    std::shared_ptr<last_name>);  
    void talk() override final;  
};
```

```
class cppnow_talk {  
    std::shared_ptr<SpeakerLike> speaker_;  
    std::shared_ptr<AttendeesLike> attendees_;  
  
public:  
    cppnow_talk(const std::shared_ptr<SpeakerLike> speaker  
                , const std::shared_ptr<AttendeesLike> attendees)  
        : speaker_(speaker), attendees_(attendees)  
    {}
```

```
    auto run();  
}
```

# **HYPODERMIC - 00 - WIRING**

# HYPODERMIC - OO - WIRING

```
int main() {
```

```
}
```

# HYPODERMIC - OO - WIRING

```
int main() {  
  
    Hypodermic::ContainerBuilder builder{};  
  
}
```

# HYPODERMIC - OO - WIRING

```
int main() {  
  
    Hypodermic::ContainerBuilder builder{};  
  
    builder.registerType<SpeakerLike>().as<regular_speaker>();  
    builder.registerType<AttendeesLike>().as<awesome_attendees>();  
  
}
```

# HYPODERMIC - OO - WIRING

```
int main() {  
  
    Hypodermic::ContainerBuilder builder{};  
  
    builder.registerType<SpeakerLike>().as<regular_speaker>();  
    builder.registerType<AttendeesLike>().as<awesome_attendees>();  
  
    auto container = builder.build();  
  
}
```

# HYPODERMIC - OO - WIRING

```
int main() {  
  
    Hypodermic::ContainerBuilder builder{};  
  
    builder.registerType<SpeakerLike>().as<regular_speaker>();  
    builder.registerType<AttendeesLike>().as<awesome_attendees>();  
  
    auto container = builder.build();  
  
    // throws if not bound  
    return container->resolve<cppnow_talk>()->run();  
  
}
```

# GOOGLE.FRUIT - 00

# GOOGLE.FRUIT - 00

```
class regular_speaker : public SpeakerLike {  
public:  
    INJECT(regular_speaker(first_name, last_name)) = default;  
    void talk() override final;  
};
```

# GOOGLE.FRUIT - 00

```
class regular_speaker : public SpeakerLike {  
public:  
    INJECT(regular_speaker(first_name, last_name)) = default;  
    void talk() override final;  
};
```

```
class cppnow_talk {  
    SpeakerLike* speaker_{ };  
    AttendeesLike* attendees_{ };
```

```
    auto run();  
}
```

# GOOGLE.FRUIT - 00

```
class regular_speaker : public SpeakerLike {  
public:  
    INJECT(regular_speaker(first_name, last_name)) = default;  
    void talk() override final;  
};
```

```
class cppnow_talk {  
    SpeakerLike* speaker_{ };  
    AttendeesLike* attendees_{ };  
  
public:  
    INJECT(cppnow_talk(SpeakerLike* speaker  
                        , AttendeesLike* attendees))  
        : speaker_{speaker}, attendees_{attendees}  
    {}
```

```
    auto run();  
}
```

# GOOGLE.FRUIT - OO - WIRING

# GOOGLE.FRUIT - 00 - WIRING

```
int main() {
```

```
}
```

# GOOGLE.FRUIT - 00 - WIRING

```
fruit::Component<cppnow_talk> get_component() {
    return fruit::createComponent()
        .bind<SpeakerLike, regular_speaker>()
        .bind<AttendeesLike, awesome_attendees>();
}
```

```
int main() {
```

```
}
```

# GOOGLE.FRUIT - OO - WIRING

```
fruit::Component<cppnow_talk> get_component() {
    return fruit::createComponent()
        .bind<SpeakerLike, regular_speaker>()
        .bind<AttendeesLike, awesome_attendees>();
}
```

```
int main() {
    // compile error if not bound
    fruit::Injector<cppnow_talk> injector(get_component);
}
```

# GOOGLE.FRUIT - OO - WIRING

```
fruit::Component<cppnow_talk> get_component() {
    return fruit::createComponent()
        .bind<SpeakerLike, regular_speaker>()
        .bind<AttendeesLike, awesome_attendees>();
}
```

```
int main() {
    // compile error if not bound
    fruit::Injector<cppnow_talk> injector(get_component);

    auto* track = injector.get<cppnow_talk*>();

    }
}
```

# GOOGLE.FRUIT - OO - WIRING

```
fruit::Component<cppnow_talk> get_component() {
    return fruit::createComponent()
        .bind<SpeakerLike, regular_speaker>()
        .bind<AttendeesLike, awesome_attendees>();
}
```

```
int main() {
    // compile error if not bound
    fruit::Injector<cppnow_talk> injector(get_component);

    auto* track = injector.get<cppnow_talk*>();

    return track->run();
}
```

[BOOST].DI

# [BOOST].DI

```
class SpeakerLike;           // 👍 No changes required
class AttendeesLike;         // 👍 No changes required
class regular_speaker;       // 👍 No changes required
class awesome_attendees;     // 👍 No changes required
class cppnow_talk;           // 👍 No changes required
...
...
```

**[BOOST].DI - 00 - RUN-TIME - WIRING**

# [BOOST].DI - OO - RUN-TIME - WIRING

```
int main() {
```

```
}
```

# [BOOST].DI - OO - RUN-TIME - WIRING

```
int main() {  
  
    extension::injector injector{};  
  
}  
}
```

# [BOOST].DI - OO - RUN-TIME - WIRING

```
int main() {  
  
    extension::injector injector{};  
  
    injector.bind<SpeakerLike>.to<speaker>();  
    injector.bind<AttendeesLike>.to<awesome_attendees>();  
  
}
```

# [BOOST].DI - 00 - RUN-TIME - WIRING

```
int main() {  
  
    extension::injector injector{};  
  
    injector.bind<SpeakerLike>.to<speaker>();  
    injector.bind<AttendeesLike>.to<awesome_attendees>();  
  
    // throws if not bound  
    return di::create<cppnow_talk>(injector).run();  
  
}
```

# [BOOST].DI - 00 - COMPILE-TIME - WIRING

# [BOOST].DI - 00 - COMPILE-TIME - WIRING

```
int main() {
```

```
}
```

# [BOOST].DI - OO - COMPILE-TIME - WIRING

```
int main() {  
  
    auto injector = di::make_injector(  
        di::bind<SpeakerLike>.to<speaker>(),  
        di::bind<AttendeesLike>.to<awesome_attendees>()  
    );  
  
}
```

# [BOOST].DI - 00 - COMPILE-TIME - WIRING

```
int main() {  
  
    auto injector = di::make_injector(  
        di::bind<SpeakerLike>.to<speaker>(),  
        di::bind<AttendeesLike>.to<awesome_attendees>()  
    );  
  
    // compile error if not bound  
    return di::create<cppnow_talk>(injector).run();  
  
}
```

# [BOOST].DI - POWER OF THE WIRING

# [BOOST].DI - POWER OF THE WIRING

```
auto injector = di::make_injector(  
    di::bind<SpeakerLike>.to<speaker>(),  
    di::bind<AttendeesLike>.to<awesome_attendees>()  
) ;
```

# [BOOST].DI - TYPE-ERASURE - STD::FUNCTION

# [BOOST].DI - TYPE-ERASURE - STD::FUNCTION

```
class regular_speaker {  
public:  
    speaker(first_name, last_name);  
    auto operator()(); // talk  
};
```

# [BOOST].DI - TYPE-ERASURE - STD::FUNCTION

```
class regular_speaker {  
public:  
    speaker(first_name, last_name);  
    auto operator()(); // talk  
};
```

```
class cppnow_talk {
```

```
    auto operator()(); // talk  
};
```

# [BOOST].DI - TYPE-ERASURE - STD::FUNCTION

```
class regular_speaker {  
public:  
    speaker(first_name, last_name);  
    auto operator()(); // talk  
};
```

```
class cppnow_talk {  
  
public:  
    cppnow_talk(speaker speaker, attendees attendees);  
  
    auto operator()(); // talk  
};
```

# [BOOST].DI - TYPE-ERASURE - STD::FUNCTION - WIRING

# [BOOST].DI - TYPE-ERASURE - STD::FUNCTION - WIRING

```
int main() {  
    // 👍 The same wiring as with Dynamic Polymorphism  
  
    return di::create<cppnow_talk>(injector)();  
}
```

# [BOOST].DI - TYPE-ERASURE - WIRING

# [BOOST].DI - TYPE-ERASURE - WIRING

```
using SpeakerLike    = te::poly<speaker>;      // Using [Boost].TE
using AttendeesLike = te::poly<attendees>;     // Using [Boost].TE
```

# [BOOST].DI - TYPE-ERASURE - WIRING

```
using SpeakerLike    = te::poly<speaker>;      // Using [Boost].TE  
using AttendeesLike = te::poly<attendees>;     // Using [Boost].TE
```

```
cppnow_talk(SpeakerLike speaker, AttendeesLike attendees);
```

# [BOOST].DI - TYPE-ERASURE - WIRING

```
using SpeakerLike    = te::poly<speaker>;      // Using [Boost].TE  
using AttendeesLike = te::poly<attendees>;     // Using [Boost].TE
```

```
cppnow_talk(SpeakerLike speaker, AttendeesLike attendees);
```

```
int main() {  
    // 👍 The same wiring as with Dynamic Polymorphism,  
    //           std::function  
  
    return di::create<cppnow_talk>(injector).run();  
}
```

# [BOOST].DI - STD::VARIANT - WIRING

# [BOOST].DI - STD::VARIANT - WIRING

```
using SpeakerLike =  
    std::variant<regular_speaker, keynote_speaker>;  
  
using AttendeesLike =  
    std::variant<regular_attendees, awesome_attendees>;
```

# [BOOST].DI - STD::VARIANT - WIRING

```
using SpeakerLike =
    std::variant<regular_speaker, keynote_speaker>;  
  
using AttendeesLike =
    std::variant<regular_attendees, awesome_attendees>;  
  
int main() {
    // 👍 The same wiring as with Dynamic Polymorphism,
    //           std::function,
    //           Type-Erasure  
  
    return di::create<cppnow_talk>(injector).run();
}
```

# [BOOST].DI - STATIC POLYMORPHISM - WIRING

# [BOOST].DI - STATIC POLYMORPHISM - WIRING

```
template<class TSpeaker    = class SpeakerLike,  
         class TAttendees = class AttendeesLike>  
class cppnow_talk;
```

# [BOOST].DI - STATIC POLYMORPHISM - WIRING

```
template<class TSpeaker    = class SpeakerLike,
         class TAttendees = class AttendeesLike>
class cppnow_talk;
```

```
int main() {
    // 👍 The same wiring as with Dynamic Polymorphism,
    //           std::function,
    //           Type-Erasure,
    //           std::variant

    return di::create<cppnow_talk>(injector).run();
}
```

# [BOOST].DI - CONCEPTS - WIRING

# [BOOST].DI - CONCEPTS - WIRING

```
template<SpeakerLike TSpeaker      = class SpeakerLike,  
          AttendeesLike TAttendees = class AttendeeLike>  
class cppnow_talk;
```

# [BOOST].DI - CONCEPTS - WIRING

```
template<SpeakerLike TSpeaker      = class SpeakerLike,  
          AttendeesLike TAttendees = class AttendeeLike>  
class cppnow_talk;
```

```
int main() {  
    // 👍 The same wiring as with Dynamic Polymorphism,  
    //                                         std::function,  
    //                                         Type-Erasure,  
    //                                         std::variant,  
    //                                         Templates  
  
    return di::create<cppnow_talk>(injector).run();  
}
```



**IOC CONTAINERS AREN'T ONLY ABOUT OO DESIGN AND  
DON'T IMPLY PERFORMANCE OVERHEAD**

# DI LIBRARIES ADDITIONAL BENEFITS

# DI LIBRARIES ADDITIONAL BENEFITS



# REFACTORING WITH DI

# REFACTORING WITH DI

```
cppnow_talk(speaker, attendees);
```

# REFACTORING WITH DI

```
cppnow_talk(speaker, attendees);
```

```
cppnow_talk1(speaker, std::shared_ptr<attendees>);
```

# REFACTORING WITH DI

```
cppnow_talk(speaker, attendees);
```

```
cppnow_talk1(speaker, std::shared_ptr<attendees>);
```

```
cppnow_talk2(attendees, std::unique_ptr<SpeakerLike>);
```

# MANUAL WIRING - REFACTORING FOR \$\$\$

# MANUAL WIRING - REFACTORING FOR \$\$\$

```
{  
    auto speaker = regular_speaker();  
    auto attendees = awesome_attendees();  
    auto track0 = cppnow_talk(speaker, attendees);  
}
```

# MANUAL WIRING - REFACTORING FOR \$\$\$

```
{  
    auto speaker = regular_speaker();  
    auto attendees = awesome_attendees();  
    auto track0 = cppnow_talk(speaker, attendees);  
}
```

```
{  
    auto speaker = regular_speaker();  
    auto attendees = std::make_shared<awesome_attendees>();  
    auto track1 = cppnow_talk(speaker, attendees);  
}
```

# MANUAL WIRING - REFACTORING FOR \$\$\$

```
{  
    auto speaker = regular_speaker();  
    auto attendees = awesome_attendees();  
    auto track0 = cppnow_talk(speaker, attendees);  
}
```

```
{  
    auto speaker = regular_speaker();  
    auto attendees = std::make_shared<awesome_attendees>();  
    auto track1 = cppnow_talk(speaker, attendees);  
}
```

```
{  
    auto speaker = std::make_unique<regular_speaker>();  
    auto attendees = awesome_attendees();  
    auto track2 = cppnow_talk(std::move(speaker), attendees);  
}
```

# [BOOST].DI - REFACTORING FOR FREE

# [BOOST].DI - REFACTORING FOR FREE

```
{  
    track0 = di::create<cppnow_talk>(injector); // 👍 Okay  
    track1 = di::create<cppnow_talk1>(injector); // 👍 Okay  
    track2 = di::create<cppnow_talk2>(injector); // 👍 Okay  
}
```

# [BOOST].DI - EASY MOCKING FOR UNIT TESTING / **GUNIT**

# [BOOST].DI - EASY MOCKING FOR UNIT TESTING / GUNIT

```
"should return true if number of attendees  
willing to using DI increased"_test = [] {  
  
};
```

# [BOOST].DI - EASY MOCKING FOR UNIT TESTING / GUNIT

```
"should return true if number of attendees  
willing to using DI increased"_test = [] {
```

```
    EXPECT(track.run());
```

```
};
```

# [BOOST].DI - EASY MOCKING FOR UNIT TESTING / GUNIT

```
"should return true if number of attendees  
willing to using DI increased"_test = [] {
```

```
EXPECT_CALL(mocks<speaker>(talk)).Times(1);  
EXPECT_CALL(mocks<attendees>(ask)).Times(2).WillOnce(Return(1))  
    .WillOnce(Return(2));
```

```
EXPECT(track.run());
```

```
} ;
```

# [BOOST].DI - EASY MOCKING FOR UNIT TESTING / GUNIT

```
"should return true if number of attendees  
willing to using DI increased"_test = [] {  
  
    auto [track, mocks] = testing::make<cppnow_talk, StrictGMock>();  
  
    EXPECT_CALL(mocks<speaker>(talk)).Times(1);  
    EXPECT_CALL(mocks<attendees>(ask)).Times(2).WillOnce(Return(1))  
        .WillOnce(Return(2));  
  
    EXPECT(track.run());  
  
};
```

# [BOOST].DI - EASY FAKING FOR INTEGRATION TESTING

# [BOOST].DI - EASY FAKING FOR INTEGRATION TESTING

```
"Number of attendees willing to use DI should increase"_test = [] {
```

```
} ;
```

# [BOOST].DI - EASY FAKING FOR INTEGRATION TESTING

```
"Number of attendees willing to use DI should increase"_test = [] {
```

```
// EXPECT_CALL...  
  
auto track = di::create<cppnow_talk>(injector);  
  
EXPECT(track.run());  
  
};
```

# [BOOST].DI - EASY FAKING FOR INTEGRATION TESTING

```
"Number of attendees willing to use DI should increase"_test = [] {  
  
    auto test_injector = di::make_injector(  
        std::move(injector),  
        di::bind<AttendeesLike>.to<fake_attendees>() [di::override]  
    );  
  
    // EXPECT_CALL...  
  
    auto track = di::create<cppnow_talk>(injector);  
    EXPECT(track.run());  
  
};
```

# [BOOST].DI - QUALITY ENFORCEMENT

---

# [BOOST].DI - QUALITY ENFORCEMENT

```
cppnow_talk3(speakerLike,    AttendeesLike);
```

---

# [BOOST].DI - QUALITY ENFORCEMENT

```
cppnow_talk3(speakerLike, AttendeesLike);
```

```
cppnow_talk4(speakerLike*, AttendeesLike);
```

---

# [BOOST].DI - QUALITY ENFORCEMENT

```
cppnow_talk3(speakerLike, AttendeesLike);
```

```
cppnow_talk4(speakerLike*, AttendeesLike);
```

---

```
auto policy_injector =  
    di::make_injector<no_raw_pointers_policy>(std::move(injector));
```

# [BOOST].DI - QUALITY ENFORCEMENT

```
cppnow_talk3(speakerLike, AttendeesLike);
```

```
cppnow_talk4(speakerLike*, AttendeesLike);
```

```
auto policy_injector =  
    di::make_injector<no_raw_pointers_policy>(std::move(injector));
```

```
{  
    track0 = di::create<cppnow_talk>(policy_injector); // 👍 Okay  
    track1 = di::create<cppnow_talk1>(policy_injector); // 👍 Okay  
    track2 = di::create<cppnow_talk2>(policy_injector); // 👍 Okay  
    track3 = di::create<cppnow_talk3>(policy_injector); // 👍 Okay
```

# [BOOST].DI - QUALITY ENFORCEMENT

```
cppnow_talk3(speakerLike, AttendeesLike);
```

```
cppnow_talk4(speakerLike*, AttendeesLike);
```

```
auto policy_injector =  
    di::make_injector<no_raw_pointers_policy>(std::move(injector));
```

```
{  
    track0 = di::create<cppnow_talk>(policy_injector); // 👍 Okay  
    track1 = di::create<cppnow_talk1>(policy_injector); // 👍 Okay  
    track2 = di::create<cppnow_talk2>(policy_injector); // 👍 Okay  
    track3 = di::create<cppnow_talk3>(policy_injector); // 👍 Okay  
  
    // 👎 Compile time error (policy not satisfied)  
    track4 = di::create<cppnow_talk4>(policy_injector);  
}
```



**CONSIDER USING A DI FRAMEWORK TO LIMIT THE  
BOILERPLATE AND REMOVE THE WIRING MESS**

# DI LIBRARIES - GOTCHAS

# DI LIBRARIES - GOTCHAS

Please Cancel my  
subscription to your  
issues.



# USING A GLOBAL INJECTOR/CONTAINER

# USING A GLOBAL INJECTOR/CONTAINER

```
class cppnow_talk {  
    std::unique_ptr<SpeakerLike> speaker_;  
    std::shared_ptr<AttendeesLike> attendees_;
```

```
} ;
```

# USING A GLOBAL INJECTOR/CONTAINER

```
class cppnow_talk {
    std::unique_ptr<SpeakerLike> speaker_;
    std::shared_ptr<AttendeesLike> attendees_;

public:
    cppnow_talk()
        : speaker_{
            container::instance().resolve<std::unique_ptr<SpeakerLike>>()
        }
        , attendees_{
            container::instance().resolve<std::shared_ptr<SpeakerLike>>()
        }
    {}
};

};
```



# CONSIDER AVOIDING GLOBAL INJECTOR/CONTAINERS



# CONSIDER AVOIDING GLOBAL INJECTOR/CONTAINERS

```
speaker_{container::instance().resolve<SpeakerLike>() } // 🙅
```



# CONSIDER AVOIDING GLOBAL INJECTOR/CONTAINERS

```
speaker_{container::instance().resolve<SpeakerLike>() } // 👎
```

```
speaker_{speaker} // 👍
```

# USING SERVICE LOCATOR/GOD OBJECT

# USING SERVICE LOCATOR/GOD OBJECT

```
class cppnow_talk {  
    SpeakerLike& speaker_;          // Tightly coupled  
    AttendeesLike& attendees_;     // Tightly coupled  
};
```

# USING SERVICE LOCATOR/GOD OBJECT

```
class cppnow_talk {
    SpeakerLike& speaker_;           // Tightly coupled
    AttendeesLike& attendees_;      // Tightly coupled

public:
    explicit cppnow_talk(ServiceLocator& container)
        : speaker_{container.resolve<SpeakerLike&>()}
        , attendees_{container}
    { }

};
```

 CONSIDER SERVICE LOCATOR TO BE ANTI PATTERN

 **CONSIDER SERVICE LOCATOR TO BE ANTI PATTERN**

```
explicit cppnow_talk(ServiceLocator& container) // 🤦
```

# 👉 CONSIDER SERVICE LOCATOR TO BE ANTI-PATTERN

```
explicit cppnow_talk(ServiceLocator& container) // 👎
```

```
explicit cppnow_talk(speaker& speaker) // 👍
```

# COUPLING TO A SPECIFIC DI LIBRARY

# COUPLING TO A SPECIFIC DI LIBRARY

```
class cppnow_talk {  
};
```

# COUPLING TO A SPECIFIC DI LIBRARY

```
class cppnow_talk {  
  
public:  
    // Coupling to the DI framework via INJECT macro  
    INJECT(cppnow_talk(speaker, attendees));  
  
};
```

# COUPLING TO A SPECIFIC DI LIBRARY

# COUPLING TO A SPECIFIC DI LIBRARY

```
class cppnow_talk {  
};
```

# COUPLING TO A SPECIFIC DI LIBRARY

```
class cppnow_talk {  
  
public:  
    // Only shared_ptrs can be injected  
    cppnow_talk(std::shared_ptr<speaker>,  
                std::shared_ptr<attendees>);  
  
};
```



**CONSIDER NOT COUPLING CODE TO A SPECIFIC DI  
FRAMEWORK**



# CONSIDER NOT COUPLING CODE TO A SPECIFIC DI FRAMEWORK

```
INJECT(cppnow_talk(speaker, attendees)); // 👎
```



# CONSIDER NOT COUPLING CODE TO A SPECIFIC DI FRAMEWORK

```
INJECT(cppnow_talk(speaker, attendees)); // 🤦
```

```
cppnow_talk(speaker, attendees); // 👍
```

# SUMMARY

# GOOD PRACTISES ARE GOOD PRACTICES FOR A REASON!

# GOOD PRACTISES ARE GOOD PRACTICES FOR A REASON!

SOLID >> STUPID

# DEPENDENCY INJECTION

# DEPENDENCY INJECTION

- (+) LOOSELY COUPLED CODE

# DEPENDENCY INJECTION

- (+) LOOSELY COUPLED CODE
- (+) EASY TO TEST CODE

# DEPENDENCY INJECTION

- (+) LOOSELY COUPLED CODE
- (+) EASY TO TEST CODE
- (-) WIRING MESS

# DEPENDENCY INJECTION LIBRARIES

# DEPENDENCY INJECTION LIBRARIES

- (+) PROMOTES LOOSELY COUPLED CODE

# DEPENDENCY INJECTION LIBRARIES

- (+) PROMOTES LOOSELY COUPLED CODE
- (+) REMOVES / CLEANUP THE WIRING MESS

# DEPENDENCY INJECTION LIBRARIES

- (+) PROMOTES LOOSELY COUPLED CODE
- (+) REMOVES / CLEANUP THE WIRING MESS
- (+) SIMPLIFY REFACTORING / ADDING NEW FEATURES

# DEPENDENCY INJECTION LIBRARIES

- (+) PROMOTES LOOSELY COUPLED CODE
- (+) REMOVES / CLEANUP THE WIRING MESS
- (+) SIMPLIFY REFACTORING / ADDING NEW FEATURES
- (+) MAKES TESTING / ENFORCING GUIDELINES EASIER

# DEPENDENCY INJECTION LIBRARIES

- (+) PROMOTES LOOSELY COUPLED CODE
- (+) REMOVES / CLEANUP THE WIRING MESS
- (+) SIMPLIFY REFACTORING / ADDING NEW FEATURES
- (+) MAKES TESTING / ENFORCING GUIDELINES EASIER
- (-) STEEPER LEARNING CURVE





- DI IS JUST A FANCY TERM FOR USING CONSTRUCTORS (25 DOLLAR CONCEPT...)



- DI IS JUST A FANCY TERM FOR USING CONSTRUCTORS (25 DOLLAR CONCEPT...)
- DI CAN BE EASILY MISUSED



- DI IS JUST A FANCY TERM FOR USING CONSTRUCTORS (25 DOLLAR CONCEPT...)
- DI CAN BE EASILY MISUSED
- DI DOESN'T REQUIRE A LIBRARY/FRAMEWORK



- DI IS JUST A FANCY TERM FOR USING CONSTRUCTORS (25 DOLLAR CONCEPT...)
- DI CAN BE EASILY MISUSED
- DI DOESN'T REQUIRE A LIBRARY/FRAMEWORK
- DI LIBRARY/FRAMEWORK HELPS WITH THE WIRING MESS

# LET'S INJECT ALL THE THINGS!

---

Slides

<http://boost-experimental.github.io/di/cppnow-2019>

---

Hypodermic <https://github.com/ybainier/Hypodermic>

---

Google.Fruit <https://github.com/google/fruit>

---

[Boost].DI  
<https://github.com/boost-experimental/di>

---

-

[kris@jusiak.net](mailto:kris@jusiak.net) | [@krisjusiak](https://twitter.com/krisjusiak) | [LINKEDIN.COM/IN/KRIS-JUSIAK](https://www.linkedin.com/in/kris-jusiak)