# AW OF DEMETER

## A PRACTICAL GUIDE TO LOOSE COUPLING

S JUSIAK

kris@jusiak.net | @krisjusiak | linkedin.com/in/kris-jusiak
https://www.quantlab.com/careers

# Agenda

# Agenda

- Motivation

# Agenda

- Motivation
- Loose Coupling

# Agenda

- Motivation
- Loose Coupling
    - Law of Demeter

# Agenda

- Motivation
- Loose Coupling
    - Law of Demeter
        - [S]ingle Responsibility

# Agenda

- Motivation
- Loose Coupling
  - Law of Demeter
    - [S]ingle Responsibility
    - ...

# Agenda

- Motivation
- Loose Coupling
  - Law of Demeter
    - [S]ingle Responsibility
    - ...
    - [D]ependency Inversion/Injection

# Agenda

- Motivation
- Loose Coupling
    - Law of Demeter
        - [S]ingle Responsibility
        - ...
        - [D]ependency Inversion/Injection
- Summary

# Agenda

- Motivation
- Loose Coupling
    - Law of Demeter
        - [S]ingle Responsibility
        - ...
        - [D]ependency Inversion/Injection
- Summary

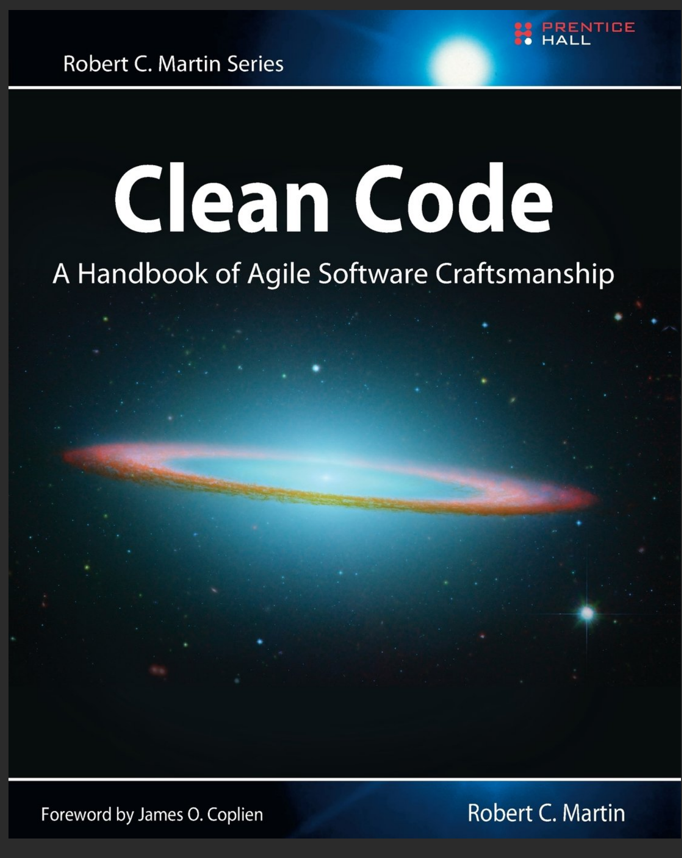
---

darkblue background - something to remember ✓

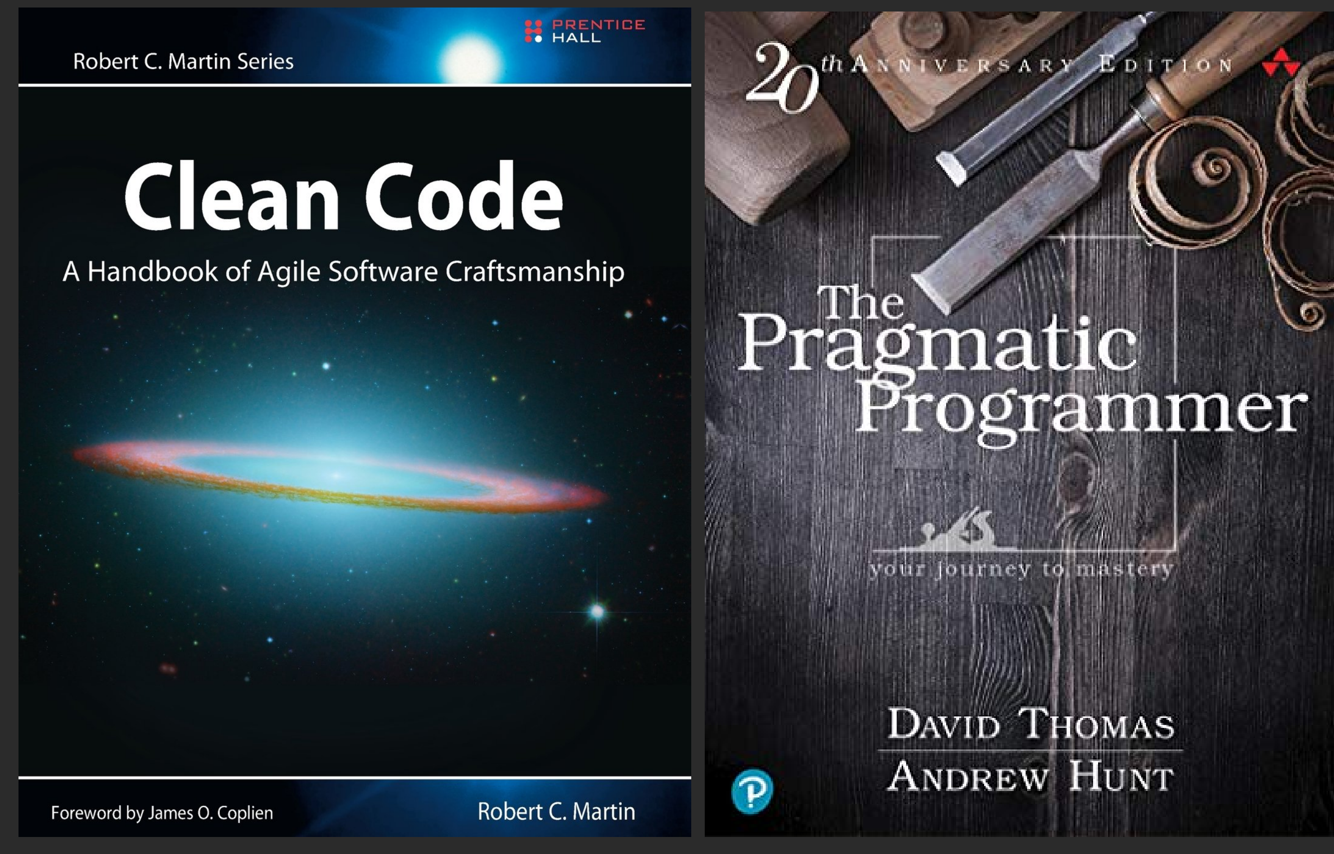
# Motivation

# "The only way to go fast is to go well", Uncle Bob

# "The only way to go fast is to go well", Uncle Bob



# "The only way to go fast is to go well", Uncle Bob



# The Essence of Good Design

# The Essence of Good Design

*Good Design is Easier to Change Than Bad Design (ETC)*

# 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

> *"Test Your Software, or Your Users Will"*

## Testable

*"Test Your Software, or Your Users Will"*

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

# Loose Coupling / Easier To Change

# Loose Coupling / Easier To Change

*By Example*

# KISS - ~~Keep it simple~~, 'STUPID'

# KISS - ~~Keep it simple~~, 'STUPID'

```
class accu_talk {
 public:
```

```
};
```

# KISS - ~~Keep it simple~~, 'STUPID'

```
class accu_talk {
 public:
```

```
  [[gnu::always_inline]] auto run() { // Premature Optimization
```

```
  }
```

```
};
```

# KISS - ~~Keep it simple~~, 'STUPID'

```
class accu_talk {
 public:
```

```
  [[gnu::always_inline]] auto run() { // Premature Optimization
```

```
    // Indescriptive Naming - What's Manager responsibility?
    // Singleton            - Manager::instance() - global variable
    // Untestability        - How to fake Manager?
    // Tight Coupling       - Can we change the speaker?
    const auto& speaker = Manager::instance().get_speakers().get();
```

```
  }
```

```
};
```

# KISS - ~~Keep it simple~~, 'STUPID'

```
class accu_talk {
 public:

  [[gnu::always_inline]] auto run() { // Premature Optimization

    // Indescriptive Naming - What's Manager responsibility?
    // Singleton            - Manager::instance() - global variable
    // Untestability        - How to fake Manager?
    // Tight Coupling       - Can we change the speaker?
    const auto& speaker = Manager::instance().get_speakers().get();

    // Duplication - Manager::instance(), same access pattern
    const auto& attendees = Manager::instance().get_attendees().get();

  }

};
```

# KISS - ~~Keep it simple~~, 'STUPID'

```
class accu_talk {
 public:

  [[gnu::always_inline]] auto run() { // Premature Optimization

    // Indescriptive Naming - What's Manager responsibility?
    // Singleton            - Manager::instance() - global variable
    // Untestability        - How to fake Manager?
    // Tight Coupling       - Can we change the speaker?
    const auto& speaker = Manager::instance().get_speakers().get();

    // Duplication - Manager::instance(), same access pattern
    const auto& attendees = Manager::instance().get_attendees().get();

    speaker.talk();
    attendees.ask();

  }

};
```

# KISS - ~~Keep it simple~~, 'STUPID'

# KISS - ~~Keep it simple~~, 'STUPID'

- Flexible?

# KISS - ~~Keep it simple~~, 'STUPID'

- Flexible?
  - Not really: Tightly coupled

# KISS - ~~Keep it simple~~, 'STUPID'

- Flexible?
  - Not really: Tightly coupled
- Scalable?

# KISS - ~~Keep it simple~~, 'STUPID'

- Flexible?
  - Not really: Tightly coupled
- Scalable?
  - Not really: Hard to extend

# KISS - ~~Keep it simple~~, 'STUPID'

- Flexible?
  - Not really: Tightly coupled
- Scalable?
  - Not really: Hard to extend
- Testable?

# KISS - ~~Keep it simple~~, 'STUPID'

- Flexible?
  - Not really: Tightly coupled
- Scalable?
  - Not really: Hard to extend
- Testable?
  - Not really: Hard to fake

# KISS - ~~Keep it simple~~, 'STUPID' | Problems

# KISS - ~~Keep it simple~~, 'STUPID' | Problems

- [S]ingleton

# KISS - ~~Keep it simple~~, 'STUPID' | Problems

- [S]ingleton
- [T]ight Coupling

# KISS - ~~Keep it simple~~, 'STUPID' | Problems

- [S]ingleton
- [T]ight Coupling
- [U]ntestability

# KISS - ~~Keep it simple~~, 'STUPID' | Problems

- [S]ingleton
- [T]ight Coupling
- [U]ntestability
- [P]remature Optimization

# KISS - ~~Keep it simple~~, 'STUPID' | Problems

- [S]ingleton
- [T]ight Coupling
- [U]ntestability
- [P]remature Optimization
- [I]ndescriptive Naming

# KISS - ~~Keep it simple~~, 'STUPID' | Problems

- [S]ingleton
- [T]ight Coupling
- [U]ntestability
- [P]remature Optimization
- [I]ndescriptive Naming
- [D]uplication

# KISS - ~~Keep it simple~~, 'S([T]ight Coupling)UPID'

# KISS - ~~Keep it simple~~, 'S([T]ight Coupling)UPID'

```
obj1.obj2.obj3.objN...
```

# KISS - ~~Keep it simple~~, 'S([T]ight Coupling)UPID'

```
obj1.obj2.obj3.objN...
```

aka

```
Manager::instance().get_speakers().get()...
```

# KISS - ~~Keep it simple~~, 'S([T]ight Coupling)UPID'

```
obj1.obj2.obj3.objN...
```

aka

```
Manager::instance().get_speakers().get()...
```

> *Is breaking the law!*

# Law of demeter

# Law of demeter

*Only talk to your immediate friends!*

---

# Law of demeter

> *Only talk to your immediate friends!*

---

```
Manager::instance().get_speakers().get() // Don't chain method calls 👎
```

## *Why that's so bad?*

## *Why that's so bad?*

- Tightly coupled

## *Why that's so bad?*

- Tightly coupled
- Almost impossible to test

## *Why that's so bad?*

- Tightly coupled
- Almost impossible to test
- Really hard to extend/reuse

*How to fix it?*

# Not a fix!

# Not a fix!

```
const auto* speaker = Manager::instance()->get_speakers()->get(); // 👎

speaker->talk();
```

# Not a fix!

```
const auto* speaker = Manager::instance()->get_speakers()->get(); // 👎

speaker->talk();
```

---

```
auto& manager       = Manager::instance();     // 👎
auto& speakers      = manager.get_speakers(); // 👎
const auto& speaker = speakers.get();          // 👎

speaker.talk();
```

# How to fix it?

# How to fix it?

- By applying `SOLID` principles the proper way!

# How to fix it?

- By applying `SOLID` principles the proper way!
- By applying Test Driven Development (TDD) / Behaviour Driven Development (BDD)!

# SOLID vs 'STUPID'

# SOLID vs 'STUPID'

| | | | |
|---|---|---|---|
| S | Single Responsibility | S | Singleton |
| O | Open-close | T | Tight Coupling |
| L | Liskov substitution | U | Untestability |
| I | Interface segregation | P | Premature Optimization |
| D | Dependency inversion | I | Indescriptive Naming |
| | | D | Duplication |

# Let's fix it then, shall we 👌

# Single Responsibility Principle (SRP)

# Single Responsibility Principle (SRP)

# Single Responsibility Principle (SRP)

> *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)

```cpp
/**
 * Responsibility: Give a talk
 */
class speaker {
  static constexpr auto name = "Kris"; // Tightly coupled

 public:
  void talk();
};
```

```cpp
/**
 * Responsibility: Ask questions
 */
class attendees {
  std::vector names = {"John", "Mike", ...}; // Tightly coupled

 public:
  auto ask();
};
```

# Single Responsibility Principle (SRP)

# Single Responsibility Principle (SRP)

```
class accu_talk {
```

```
};
```

# Single Responsibility Principle (SRP)

```
class accu_talk {

  speaker speaker_{};       // Tightly coupled
  attendees attendees_{};   // Tightly coupled

};
```

# Single Responsibility Principle (SRP)

```
class accu_talk {

  speaker speaker_{};       // Tightly coupled
  attendees attendees_{};   // Tightly coupled

 public:
  auto run() {
    speaker.talk();
    attendees.ask();
  }

};
```

# Singleton Responsibility Principle (SRC)

# Singleton Responsibility Principle (SRC)

- Flexible?

# Singleton Responsibility Principle (SRC)

- Flexible?
  - Better but still coupled

# Singleton Responsibility Principle (SRC)

- Flexible?
  - Better but still coupled
- Scalable?

# Singleton Responsibility Principle (SRC)

- Flexible?
  - Better but still coupled
- Scalable?
  - A bit easier to change (ETC) as we can change components in separation

# Singleton Responsibility Principle (SRC)

- Flexible?
  - Better but still coupled
- Scalable?
  - A bit easier to change (ETC) as we can change components in separation
- Testable?

# Singleton Responsibility Principle (SRC)

- Flexible?
  - Better but still coupled
- Scalable?
  - A bit easier to change (ETC) as we can change components in separation
- Testable?
  - Still hard to fake but can be unit-tested

# Consider classes to have only one reason to change

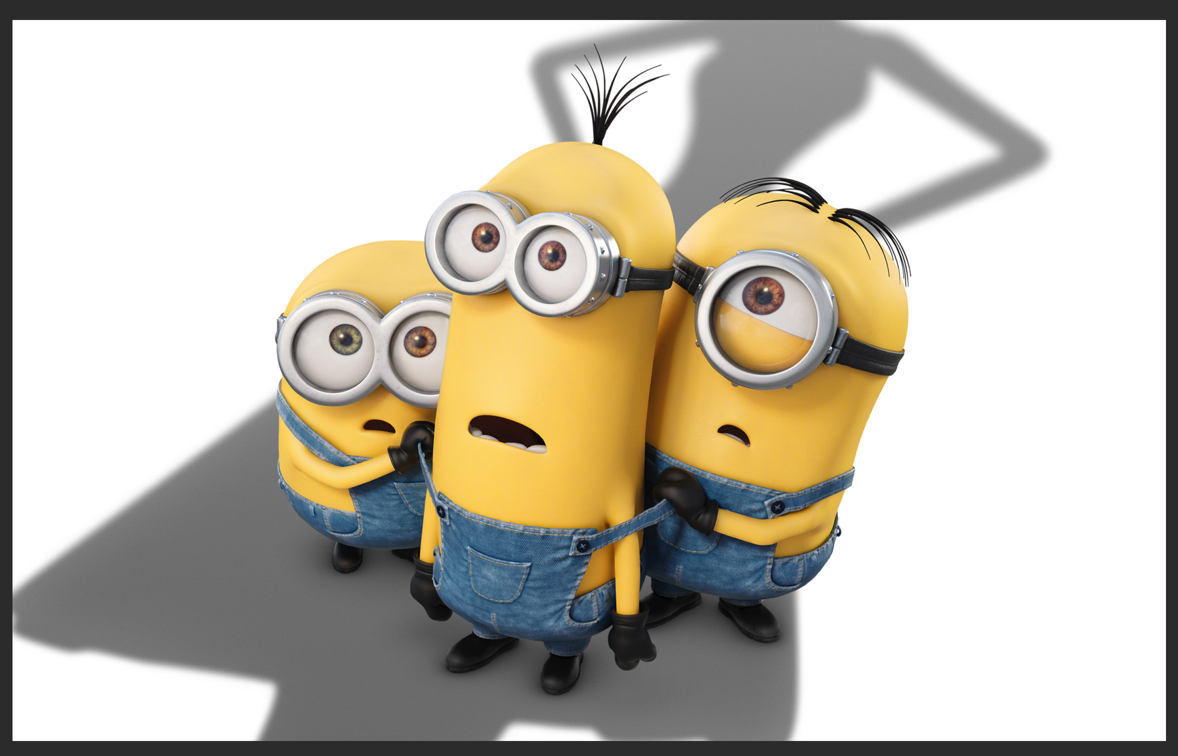
# Consider classes to have only one reason to change

---

# 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

# 👍 DI doesn't imply using a library/framework

*DI libraries may help in the large scale!*

# 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 accu_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 name)
     : name_{name}
   { }
```

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

# Less coupling - Constructor DI

# Less coupling - Constructor DI

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

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

# Less coupling - Constructor DI

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

```
 public:
  // 👍 Dependency Injection!!!
  accu_talk(speaker speaker, attendees attendees)
      : speaker_{speaker}, attendees_{attendees}
  { }
```

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

# Less coupling - Constructor DI

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

```
 public:
  // 👍 Dependency Injection!!!
  accu_talk(speaker speaker, attendees attendees)
      : speaker_{speaker}, attendees_{attendees}
  { }
```

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

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

# Constructor DI - gotchas

# Constructor DI - gotchas

# Not using constructors consistently

# Not using constructors consistently

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

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

# Not using constructors consistently

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

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

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

# 👍 Consider using constructor Dependency Injection consistently

# 👍 Consider using constructor Dependency Injection consistently

```
accu_talk() : speaker_{"Kris"} {}                    // 👎
```

# 👍 Consider using constructor Dependency Injection consistently

```
accu_talk() : speaker_{"Kris"} {}                        // 👎
```

```
accu_talk(speaker speaker) : speaker_{speaker} {}        // 👍
```

# Using singletons

# Using singletons

```
class accu_talk {
 public:
```

```
};
```

# Using singletons

```
class accu_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")                                    // 👎
```

```
accu_talk(speaker speaker) : speaker_{speaker} {}               // 👍
```

# Carrying dependencies

# Carrying dependencies

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

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

# Carrying dependencies

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

```
 public:
  // 👎 Leaky abstraction
  explicit accu_talk(std::string 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 accu_talk(std::string name) : speaker{name} {}      // 👎
```

# 👍 Consider passing initialized objects instead of parameters to initialize them

```
explicit accu_talk(std::string name) : speaker{name} {}       // 👎
```

```
explicit accu_talk(speaker speaker)  : speaker_{speaker} {}  // 👍
```

# Carrying dependencies with inheritance

# Carrying dependencies with inheritance

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

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

# Carrying dependencies with inheritance

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

```
 public:
  explicit accu_talk(std::string name) // Common with CRTP?
    : speaker{name}
   {}
```

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

# 👍 Prefer composition over inheritance

# 👍 Prefer composition over inheritance

```
class accu_talk : speaker                         // 👎
```

# 👍 Prefer composition over inheritance

```
class accu_talk : speaker                     // 👎
```

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

# Talking to your distant friends

# Talking to your distant friends

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

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

# Talking to your distant friends

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

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

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

👍 Consider talking only to your immediate friends

# 👍 Consider talking only to your immediate friends

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

# 👍 Consider talking only to your immediate friends

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

```
speaker_{speaker};                               // 👍
```

# 👍 Consider talking only to your immediate friends

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

```
speaker_{speaker};                             // 👍
```

*Law of demeter!*

# Not using strong types

# Not using strong types

```cpp
// 👎 Weak API
speaker(std::string first_name, std::string last_name);
```

# Not using strong types

```
// 👎 Weak API
speaker(std::string first_name, std::string last_name);
```

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

# Not using strong types

```
// 👎 Weak API
speaker(std::string first_name, std::string last_name);
```

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

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

# Strong types for strong interfaces

# Strong types for strong interfaces

```
using first_name = named<std::string, "first name">;
using last_name  = named<std::string, "last name">;
```

# Strong types for strong interfaces

```
using first_name = named<std::string, "first name">;
using last_name  = named<std::string, "last name">;
```

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

# Strong types for strong interfaces

```
using first_name = named<std::string, "first name">;
using last_name  = named<std::string, "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<std::string, "first name">;
using last_name  = named<std::string, "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(std::string first_name, std::string last_name);     // 👎
```

## 👍 Consider using strong types

```
speaker(std::string first_name, std::string last_name);     // 👎
```

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

# Combine required parameters together

# Combine required parameters together

```
struct speaker_info {



};
```

# Combine required parameters together

```
struct speaker_info {

  first_name first_name;
  last_name  last_name;
  ...

};
```

# Consider combining required paremeters together

# 👍 Consider combining required paremeters together

```
make_speaker(std::string first_name, std::string last_name, ...); // 👎
```

# 👍 Consider combining required paremeters together

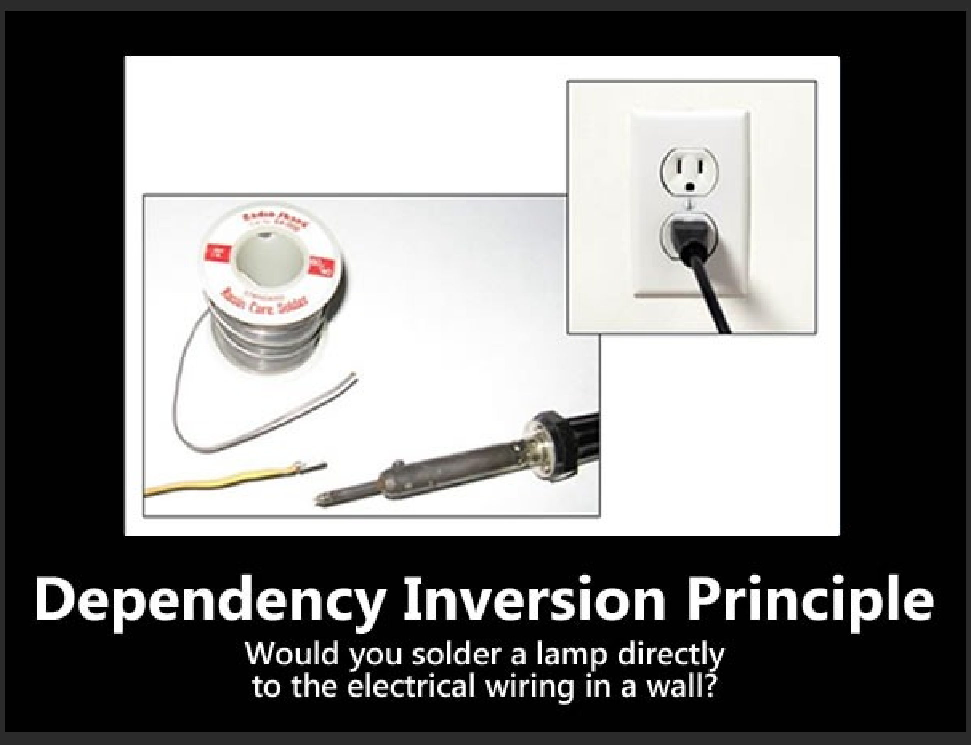
```
make_speaker(std::string first_name, std::string last_name, ...); // 👎
```

```
make_speaker(speaker_info);                                        // 👍
```

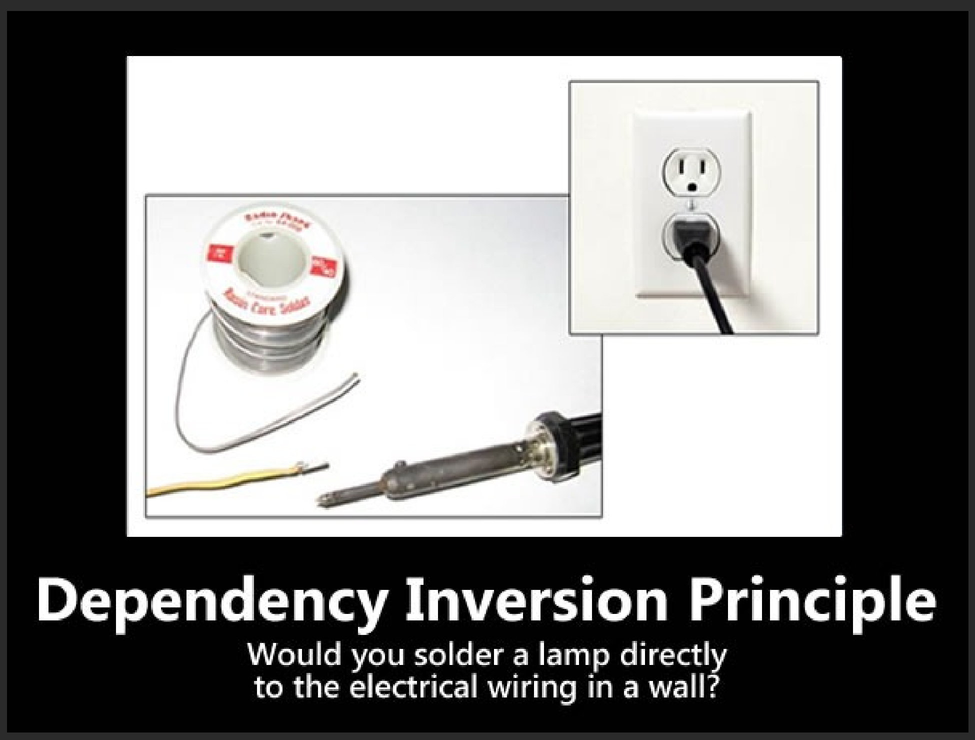
# Let's make it `actually` flexible 👌

# Dependency Inversion Principle (DIP)

# Dependency Inversion Principle (DIP)



# Dependency Inversion Principle (DIP)




> *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++

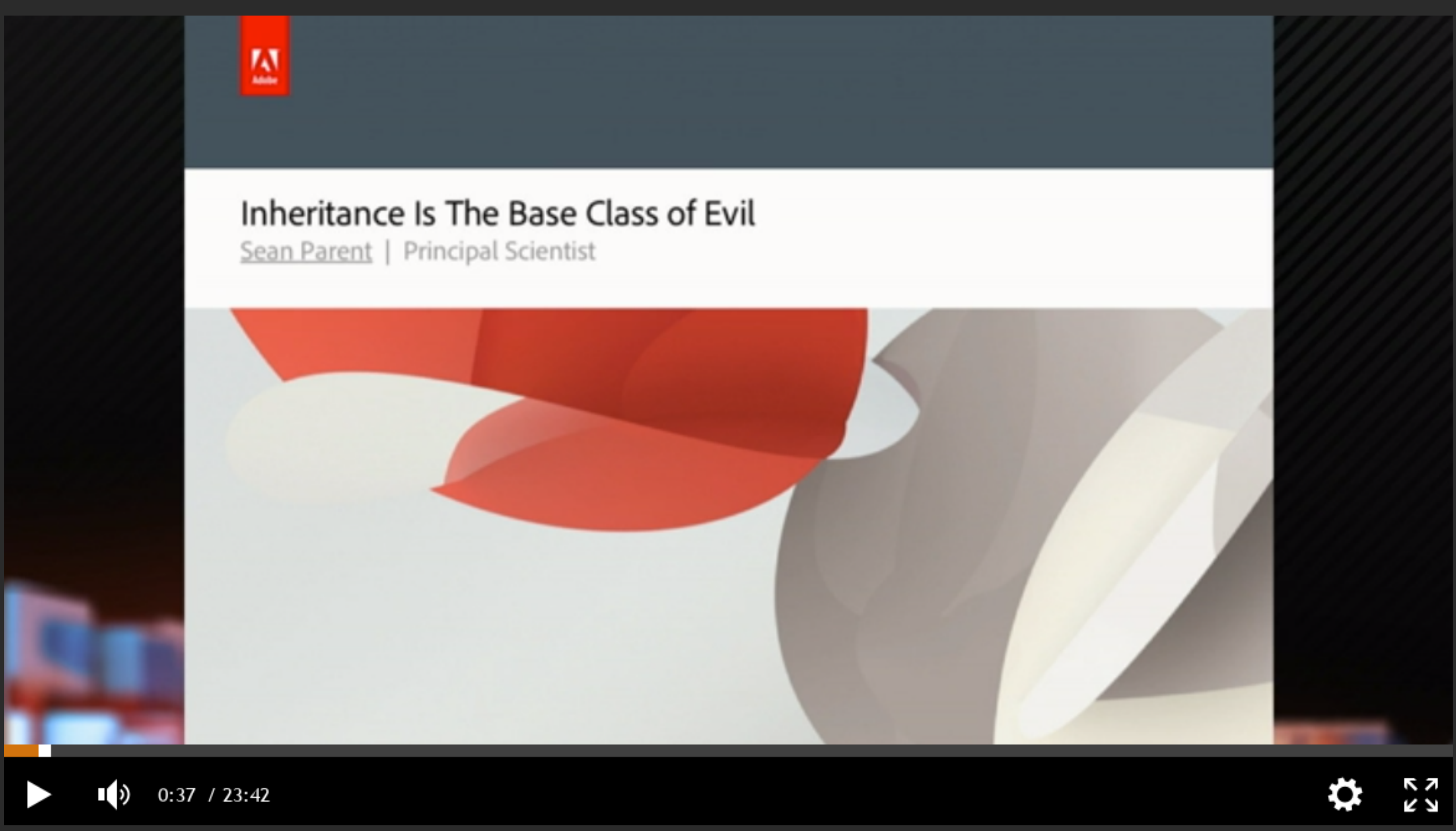
- Inheritance
- Type-Erasure
- std::variant/std::any (C++17)
- Templates
- Concepts (C++20)

# Polymorphism in C++

- Inheritance
- Type-Erasure
- std::variant/std::any (C++17)
- Templates
- Concepts (C++20)
- ...

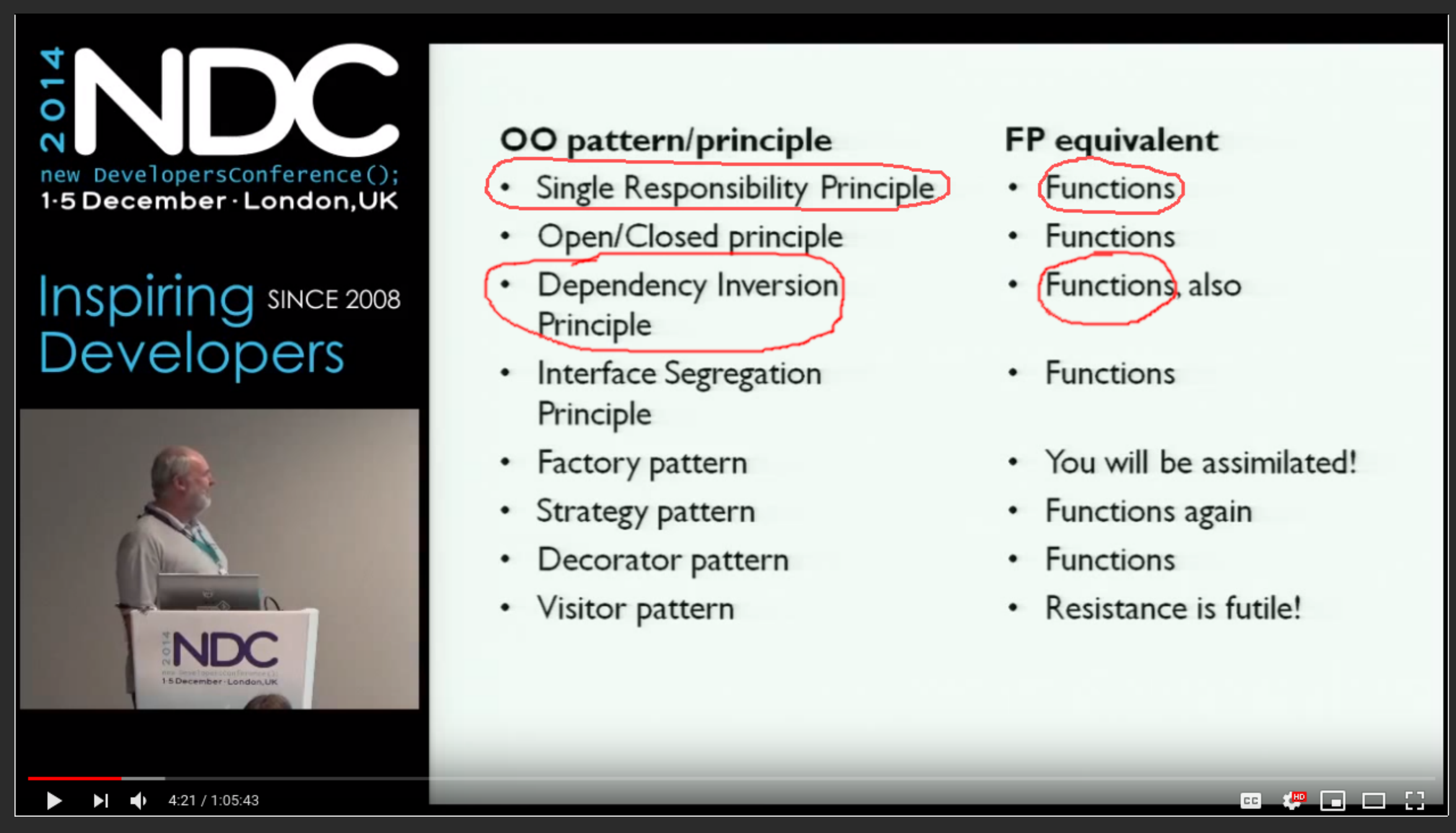
# Inheritance Is The Base Class of Evil, Sean Parent

# Inheritance Is The Base Class of Evil, Sean Parent



# Functional programming design patterns, Scott Wlaschin

# Functional programming design patterns, Scott Wlaschin



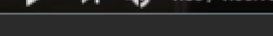
# Policy Design / Design by introspection

# Policy Design / Design by introspection

DConf 2017

Andrei Alexandrescu, Ph.D.

2017-05-06



# 👍 Consider using proper abstractions for your project

- Templates/Concepts
- Type-Erasure
- Inheritance

## 👍 Consider using proper abstractions for your project

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

# 👍 Consider using proper abstractions for your project

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

# 👍 Consider using proper abstractions for your project

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

# Concepts (C++20)

# Concepts (C++20)

```
template <class TSpeaker>
concept Speaker = requires(TSpeaker speaker) {
  { speaker.talk() } -> std::same_as<void>;
};
```

# Concepts (C++20)

```
template <class TSpeaker>
concept Speaker = requires(TSpeaker speaker) {
  { speaker.talk() } -> std::same_as<void>;
};
```

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

# Concepts (C++20)

# Concepts (C++20)

```
template<Speaker TSpeaker, Attendees TAttendees>
```

# Concepts (C++20)

```
template<Speaker TSpeaker, Attendees TAttendees>
```

```
class accu_talk {
  TSpeaker& speaker_;
  TAttendees& attendees_;
```

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

# Concepts (C++20)

```
template<Speaker TSpeaker, Attendees TAttendees>
```

```
class accu_talk {
  TSpeaker& speaker_;
  TAttendees& attendees_;
```

```
 public:
  accu_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() {
```

```
  Speaker speaker      = regular_speaker{"Kris", "Jusiak"};
```

```
}
```

# Concepts (C++20) - Wiring

```
int main() {
  Speaker speaker     = regular_speaker{"Kris", "Jusiak"};
  Attendees attendees = awesome_attendees{"John", "Mike", ...};




}
```

# Concepts (C++20) - Wiring

```
int main() {
  Speaker speaker     = regular_speaker{"Kris", "Jusiak"};
  Attendees attendees = awesome_attendees{"John", "Mike", ...};
  auto talk           = accu_talk{speaker, attendees};


}
```

# Concepts (C++20) - Wiring

```
int main() {
  Speaker speaker     = regular_speaker{"Kris", "Jusiak"};
  Attendees attendees = awesome_attendees{"John", "Mike", ...};
  auto talk           = accu_talk{speaker, attendees};
  talk.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*

*(aka wired together)*

# 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?
  - ETC applies

# Dependency Inversion Principle

- Flexible?
  - Yes: Loosely coupled
- Scalable?
  - ETC applies
- Testable?

# Dependency Inversion Principle

- Flexible?
  - Yes: Loosely coupled
- Scalable?
  - ETC applies
- Testable?
  - Yes: We can inject fakes/stubs/mocks

# Can we go too far?

---

---

# Can we go too far?

---

# Absolutely!

---

# Can we go too far?

---

# Absolutely!

---

> *Good Design is Easier to Change Than Bad Design (ETC)*

# Can we go too far?

# Can we go too far?

```
struct ispeaker_talk {
  virtual ~ispeaker_talk() noexcept = default;
  virtual auto talk() -> void = 0;
}
```

# Can we go too far?

```
struct ispeaker_talk {
  virtual ~ispeaker_talk() noexcept = default;
  virtual auto talk() -> void = 0;
}
```

```
struct ispeaker_say {
  virtual ~ispeaker_say() noexcept = default;
  virtual auto say() -> void = 0;
}
```

# Can we go too far?

```
struct ispeaker_talk {
  virtual ~ispeaker_talk() noexcept = default;
  virtual auto talk() -> void = 0;
}
```

```
struct ispeaker_say {
  virtual ~ispeaker_say() noexcept = default;
  virtual auto say() -> void = 0;
}
```

```
...
```

---

# Can we go too far?

```
struct ispeaker_talk {
  virtual ~ispeaker_talk() noexcept = default;
  virtual auto talk() -> void = 0;
}
```

```
struct ispeaker_say {
  virtual ~ispeaker_say() noexcept = default;
  virtual auto say() -> void = 0;
}
```

```
...
```

---

```
Aiming for 100% coverage! // just for the sake of coverage
```

# 👎 Going too far is a risk!

# 👎 Going too far is a risk!

> *Like with any approach there is a risk of going too far without much benefits*

---

# 👎 Going too far is a risk!

*Like with any approach there is a risk of going too far without much benefits*

---

*ETC to the rescue!*

# And what about testing?

# Behavior Driven Development (BDD) / ut

# Behavior Driven Development (BDD) / ut

```
given("I have a accu talk") = [] {
```

```
};
```

# Behavior Driven Development (BDD) / ut

```
given("I have a accu talk") = [] {
```

```
  auto speaker   = fake_speaker{};
  auto attendees = fake_attendees{};
  auto talk      = accu_talk{speaker, attendees};
```

```
};
```

# Behavior Driven Development (BDD) / ut

```
given("I have a accu talk") = [] {

  auto speaker   = fake_speaker{};
  auto attendees = fake_attendees{};
  auto talk      = accu_talk{speaker, attendees};

  when("I run the talk") = [&] {
    talk.run();

  };

};
```

# Behavior Driven Development (BDD) / ut

```
given("I have a accu talk") = [] {

  auto speaker   = fake_speaker{};
  auto attendees = fake_attendees{};
  auto talk      = accu_talk{speaker, attendees};

  when("I run the talk") = [&] {
    talk.run();

  then("The speaker should give a talk") = [&] {
    expect_call(speaker.talk);
  };

  then("The attendees should ask questions") = [&] {
    expect_call(attendees.ask);
  };

  };

};
```

# BDD/TDD

# BDD/TDD

- Flexible?

# BDD/TDD

- Flexible?
  - Yes: Loosely coupled

# BDD/TDD

- Flexible?
  - Yes: Loosely coupled
- Scalable?

# BDD/TDD

- Flexible?
  - Yes: Loosely coupled
- Scalable?
  - Yes, BDD/TDD drives ETC

# BDD/TDD

- Flexible?
  - Yes: Loosely coupled
- Scalable?
  - Yes, BDD/TDD drives ETC
- Testable?

# BDD/TDD

- Flexible?
  - Yes: Loosely coupled
- Scalable?
  - Yes, BDD/TDD drives ETC
- Testable?
  - Well, Yeah!

# 👍 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**

# Summary

# Good practises are good practices for a reason!

# Good practises are good practices for a reason!

SOLID >> `STUPID`

# Law of demeter

# Law of demeter

- Promotes loosely coupled code

# Law of demeter

- Promotes loosely coupled code
- Makes testing easier

# Law of demeter

- Promotes loosely coupled code
- Makes testing easier
- TDD/BDD, Single Responsibility, Dependency Injecition/Inversion is the way to go!

# Law of demeter

- Promotes loosely coupled code
- Makes testing easier
- TDD/BDD, Single Responsibility, Dependency Injecition/Inversion is the way to go!
- DI can be easily misused and doesn't require a library/framework

# Law of Demeter: A Practical Guide to Loose Coupling

https://www.quantlab.com/careers

# Let's

# "only talk to our immediate friends"

https://www.quantlab.com/careers

# Let's

# !

# "only talk to our immediate friends"

https://www.quantlab.com/careers