

# Design Principles and Design Patterns

**D. Ryan Bartling**

January 19, 2019

# Outline

Introduction

Symptoms of Rotting Design

Principles of Object Oriented Class Design

Package Design

Architecture Design

Conclusion

# Introduction

## Introduction

## Symptoms of Rotting Design

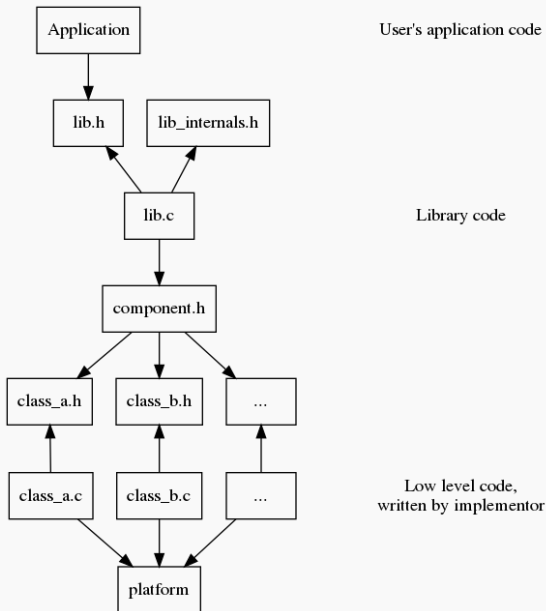
## Principles of Object Oriented Class Design

## Package Design

## Architecture Design

## Conclusion

# Architecture and Dependencies



# Symptoms of Rotting Design

Introduction

**Symptoms of Rotting Design**

Principles of Object Oriented Class Design

Package Design

Architecture Design

Conclusion

# Symptoms of Rotting Design

# Symptoms of Rotting Design

## 1. Rigidity

# Symptoms of Rotting Design

1. Rigidity
2. Fragility



# Symptoms of Rotting Design

1. Rigidity
2. Fragility
3. Immobility

# Symptoms of Rotting Design

1. Rigidity
2. Fragility
3. Immobility
4. Viscosity



Rigidity

# Rigidity

- ▶ Deficient in or devoid of flexibility

# Rigidity

- ▶ Deficient in or devoid of flexibility
- ▶ Software for which extra effort is expended in order to make changes.

# Rigidity

- ▶ Deficient in or devoid of flexibility
- ▶ Software for which extra effort is expended in order to make changes.
- ▶ The system is hard to change because every change forces many other changes to other parts of the system.



# Rigidity

How it happens

# Rigidity

How it happens

- ▶ Code written in a procedural way



# Rigidity

How it happens

- ▶ Code written in a procedural way
- ▶ Lack of abstractions

# Rigidity

How it happens

- ▶ Code written in a procedural way
- ▶ Lack of abstractions
- ▶ Solving a generic problem with implementation specific details

# Rigidity

How it happens

- ▶ Code written in a procedural way
- ▶ Lack of abstractions
- ▶ Solving a generic problem with implementation specific details
- ▶ Spreading a single responsibility throughout several parts

# Rigidity

How it happens

- ▶ Code written in a procedural way
- ▶ Lack of abstractions
- ▶ Solving a generic problem with implementation specific details
- ▶ Spreading a single responsibility throughout several parts
- ▶ When components need a lot of knowledge about each other in order to function

# Rigidity

```
1  #include <stdint.h>
2
3  #define ADC_BITS (12)
4  #define ADC_DATA_SHIFT (2)
5  #define ADC_SIGN_CONVERSION (1)
6  #define RAW_ADC_BITS (15) // Sum of the above bits
7
8  #define LFSR_LENGTH (4)
9  #define LFSR_REPEATS (2)
10 #define CORRELATED_BITS (20) // ADC bits + lfsr length + log2(repeats)
11
12 typedef int16_t rpo_raw_adc_t;
13 typedef int24_t rpo_correlated_int_t;
14
15 #if sizeof(rpo_raw_adc_t) < (RAW_ADC_BITS / 2 + 1)
16 #error "rpo_raw_adc_t is too small to store ADC results"
17 #endif
18
19 #if sizeof(rpo_correlated_int_t) < (RAW_ADC_BITS / 2 + 1)
20 #error "rpo_correlated_int_t is too small to store correlated adc results"
21 #endif
```



# Rigidity

```
1  #include <stdint.h>
2
3  #define ADC_BITS (14) // Changing this
4  #define ADC_DATA_SHIFT (2)
5  #define ADC_SIGN_CONVERSION (1)
6  #define RAW_ADC_BITS (17) // Changes this
7
8  #define LFSR_LENGTH (4)
9  #define LFSR_REPEATS (2)
10 #define CORRELATED_BITS (22) // Changes this
11
12 typedef int24_t rpo_raw_adc_t; // Changes this
13 typedef int24_t rpo_correlated_int_t;
14
15 #if sizeof(rpo_raw_adc_t) < (RAW_ADC_BITS / 2 + 1)
16 #error "rpo_raw_adc_t is too small to store ADC results"
17 #endif
18
19 #if sizeof(rpo_correlated_int_t) < (RAW_ADC_BITS / 2 + 1)
20 #error "rpo_correlated_int_t is too small to store correlated adc results"
21 #endif
```

# Rigidity

## Refactor to reduce rigidity

```
1  #include <stdint.h>
2
3  #define ADC_BITS (14)
4  #define ADC_DATA_SHIFT (2)
5  #define ADC_SIGN_CONVERSION (1)
6  #define RAW_ADC_BITS (ADC_BITS + ADC_DATA_SHIFT + ADC_SIGN_CONVERSION)
7  typedef_min_int(rpo_raw_adc_t, RAW_ADC_BITS);
8
9  #define LFSR_LENGTH (4)
10 #define LFSR_REPEATS (2)
11 #define CORRELATED_BITS (RAW_ADC_BITS + LFSR_LENGTH + log_2(LFSR_REPEATS))
12 typedef_min_int(rpo_correlated_int_t, CORRELATED_BITS);
```

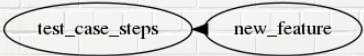
# Rigidity



new\_feature



# Rigidity

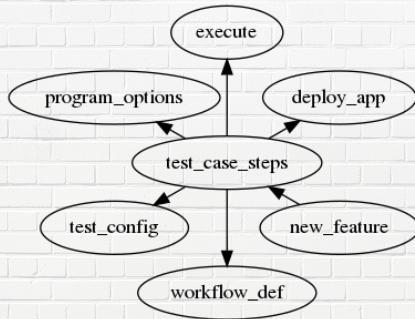


The diagram shows two overlapping ovals, one labeled 'test\_case\_steps' on the left and one labeled 'new\_feature' on the right. A small black arrow points from the 'test\_case\_steps' oval to the 'new\_feature' oval, indicating a dependency or constraint that represents rigidity in the design.

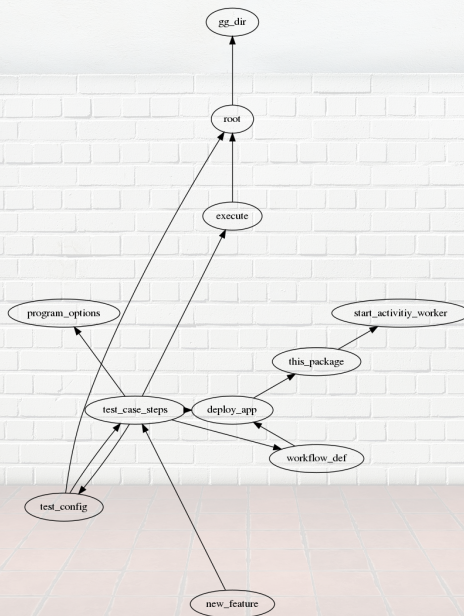
test\_case\_steps

new\_feature

# Rigidity



# Rigidity



# Rigidity

How to avoid it

# Rigidity

How to avoid it

- ▶ Break the code into smaller, self-contained concepts



# Rigidity

How to avoid it

- ▶ Break the code into smaller, self-contained concepts
- ▶ Solve the details and provide a problem oriented abstraction

# Rigidity

How to avoid it

- ▶ Break the code into smaller, self-contained concepts
- ▶ Solve the details and provide a problem oriented abstraction
- ▶ Solving a generic problem with implementation specific details

# Rigidity

How to avoid it

- ▶ Break the code into smaller, self-contained concepts
- ▶ Solve the details and provide a problem oriented abstraction
- ▶ Solving a generic problem with implementation specific details
- ▶ Write DRY code (Don't repeat yourself)



# Rigidity

How to avoid it

- ▶ Break the code into smaller, self-contained concepts
- ▶ Solve the details and provide a problem oriented abstraction
- ▶ Solving a generic problem with implementation specific details
- ▶ Write DRY code (Don't repeat yourself)
- ▶ Define the code in logical pieces. Set boundaries and responsibilities.



# Fragility

# Fragility

- ▶ Easily broken or destroyed

# Fragility

- ▶ Easily broken or destroyed
- ▶ Software for which extra risk is incurred in order to make changes.

# Fragility

- ▶ Easily broken or destroyed
- ▶ Software for which extra risk is incurred in order to make changes.
- ▶ Changes cause the system to break in places that have no conceptual relationship to the part that was changed.



# Fragility

How it happens

# Fragility

How it happens

- Implicit dependencies

# Fragility

How it happens

- ▶ Implicit dependencies
- ▶ Unmanaged shared resources



# Fragility

How it happens

- ▶ Implicit dependencies
- ▶ Unmanaged shared resources
- ▶ Relying on implementation details

# Fragility

How it happens

- ▶ Implicit dependencies
- ▶ Unmanaged shared resources
- ▶ Relying on implementation details
- ▶ Relying upon side effects of operations

# Fragility

How it happens

- ▶ Implicit dependencies
- ▶ Unmanaged shared resources
- ▶ Relying on implementation details
- ▶ Relying upon side effects of operations
- ▶ Reaching past abstraction layers

# Fragility

How it happens

- ▶ Implicit dependencies
- ▶ Unmanaged shared resources
- ▶ Relying on implementation details
- ▶ Relying upon side effects of operations
- ▶ Reaching past abstraction layers
- ▶ Unmanaged complexity

# Fragility

```
1  void sdcard_init(void) {  
2      spi_init(mode_0, card_cs_pin);  
3      fat_init();  
4  }  
5  
6  void sensor_init(void) {  
7      spi_init(mode_0, sensor_cs_pin);  
8      spi_write(SENSOR_CONFIGURATION, sensor_cs_pin);  
9  }
```

# Frailty

Changing the sensor to use mode 1...

```
1  void sdcard_init(void) {  
2      spi_init(mode_0, card_cs_pin);  
3      fat_init();  
4  }  
5  
6  void sensor_init(void) {  
7      spi_init(mode_1, sensor_cs_pin); // Breaks the sd card  
8      spi_write(SENSOR_CONFIGURATION, sensor_cs_pin);  
9  }
```

...Breaks the sd card (when sensor is initialized after the sd card)



# Fragility

How to avoid it

# Fragility

How to avoid it

- ▶ Implicit dependencies

# Fragility

How to avoid it

- ▶ Implicit dependencies
- ▶ Law of Demeter: principle of least knowledge

# Fragility

How to avoid it

- ▶ Implicit dependencies
- ▶ Law of Demeter: principle of least knowledge
- ▶ Avoid side effects, and don't rely on the side effects of other modules

# Fragility

How to avoid it

- ▶ Implicit dependencies
- ▶ Law of Demeter: principle of least knowledge
- ▶ Avoid side effects, and don't rely on the side effects of other modules
- ▶ Rely on the published API

# Fragility

How to avoid it

- ▶ Implicit dependencies
- ▶ Law of Demeter: principle of least knowledge
- ▶ Avoid side effects, and don't rely on the side effects of other modules
- ▶ Rely on the published API
- ▶ Invent and **simplify**



# Immobility



# Immobility



# Immobility

- ▶ Incapable of being moved

# Immobility

- ▶ Incapable of being moved
- ▶ Software for which extra effort is required in order to reuse.

# Immobility

- ▶ Incapable of being moved
- ▶ Software for which extra effort is required in order to reuse.
- ▶ It is hard to disentangle the system into components that can be reused in other systems.



# Immobility

How it happens

- ▶ Direct dependency on things you don't own



# Immobility

How it happens

- ▶ Direct dependency on things you don't own
- ▶ Too many responsibilities

# Immobility

How it happens

- ▶ Depend upon the concept, not the details

# Immobility

How it happens

- ▶ Depend upon the concept, not the details
- ▶ Reduce responsibilities to solve distinct problems

## Viscosity





# Viscosity

- ▶ Having or characterized by a high resistance to flow

# Viscosity

- ▶ Having or characterized by a high resistance to flow
- ▶ Software projects in which design preserving changes are more difficult than hacks.



## ▶ Viscous Design

# Viscosity

- ▶ Viscous Design
- ▶ Viscous Environment

# Viscosity

- ▶ Viscous Design
  - ▶ When making changes, preserving the design is difficult
- ▶ Viscous Environment

# Viscosity

- ▶ Viscous Design
  - ▶ When making changes, preserving the design is difficult
  - ▶ When a more correct solution is not the easier solution
- ▶ Viscous Environment

# Viscosity

- ▶ Viscous Design
  - ▶ When making changes, preserving the design is difficult
  - ▶ When a more correct solution is not the easier solution
  - ▶ *"That is the right way to do this, but we can't do that in this project"*
- ▶ Viscous Environment



# Viscosity

- ▶ Viscous Design
  - ▶ When making changes, preserving the design is difficult
  - ▶ When a more correct solution is not the easier solution
  - ▶ *"That is the right way to do this, but we can't do that in this project"*
- ▶ Viscous Environment
  - ▶ Long builds can prevent people from making the appropriate change since it will trigger a longer build.



# Viscosity

- ▶ Viscous Design
  - ▶ When making changes, preserving the design is difficult
  - ▶ When a more correct solution is not the easier solution
  - ▶ *"That is the right way to do this, but we can't do that in this project"*
- ▶ Viscous Environment
  - ▶ Long builds can prevent people from making the appropriate change since it will trigger a longer build.
  - ▶ Slow/unreliable Tests *"I can't run these tests after each change, I'd get no work done. Besides, they always fail anyway."*

# Viscosity

- ▶ Viscous Design
  - ▶ When making changes, preserving the design is difficult
  - ▶ When a more correct solution is not the easier solution
  - ▶ *"That is the right way to do this, but we can't do that in this project"*
- ▶ Viscous Environment
  - ▶ Long builds can prevent people from making the appropriate change since it will trigger a longer build.
  - ▶ Slow/unreliable Tests *"I can't run these tests after each change, I'd get no work done. Besides, they always fail anyway."*
  - ▶ Slow/cumbersom tools (e.g. if checking in files )

# Viscosity

- ▶ Viscous Policies
  - ▶ Management steps in to avoid the issues above

# Viscosity

- ▶ Viscous Policies
  - ▶ Management steps in to avoid the issues above
  - ▶ *"We cannot afford to have anyone touch the Fobnicator stack, because too many things depend upon it"*



# Viscosity

## ► Viscous Policies

- Management steps in to avoid the issues above
- *"We cannot afford to have anyone touch the Fobnicator stack, because too many things depend upon it"*
- Policies can remain long after the original problem was solved.

# Viscosity

## ► Viscous Policies

- Management steps in to avoid the issues above
- *"We cannot afford to have anyone touch the Fobnicator stack, because too many things depend upon it"*
- Policies can remain long after the original problem was solved.
- Process can also result in viscosity. If a more correct solution triggers a heavier round of reviews, the incorrect solution that can get by with less review and documentation will be favored by the developers. E.g. Creating a new module requires upfront design review. Adding the same code inside an existing module requires only the normal code review.



# Viscosity

Software develops along the path of least resistance. If hacks are easier, that's what your project will consist of.

# Principles of Object Oriented Class Design

Introduction

Symptoms of Rotting Design

Principles of Object Oriented Class Design

Package Design

Architecture Design

Conclusion

# Principles of Object Oriented Class Design

## SOLID Principles

# Principles of Object Oriented Class Design

## SOLID Principles

- ▶ *Single Responsibility Principle* (SRP)

# Principles of Object Oriented Class Design

## SOLID Principles

- ▶ *Single Responsibility Principle* (SRP)
- ▶ *Open Closed Principle* (OCP)

# Principles of Object Oriented Class Design

## SOLID Principles

- ▶ *Single Responsibility Principle* (SRP)
- ▶ *Open Closed Principle* (OCP)
- ▶ *Liskov Substitution Principle* (LSP)



# Principles of Object Oriented Class Design

## SOLID Principles

- ▶ *Single Responsibility Principle* (SRP)
- ▶ *Open Closed Principle* (OCP)
- ▶ *Liskov Substitution Principle* (LSP)
- ▶ *Interface Segregation Principle* (ISP)

# Principles of Object Oriented Class Design

## SOLID Principles

- ▶ *Single Responsibility Principle* (SRP)
- ▶ *Open Closed Principle* (OCP)
- ▶ *Liskov Substitution Principle* (LSP)
- ▶ *Interface Segregation Principle* (ISP)
- ▶ *Dependency Inversion Principle* (DIP)

# Single Responsibility Principle

Responsibility

# Single Responsibility Principle

Responsibility

▶ Cohesion

# Single Responsibility Principle

Responsibility

- ▶ Cohesion
- ▶ Reason to change

# Single Responsibility Principle

## Responsibility

- ▶ Cohesion
- ▶ Reason to change
- ▶ Axis of change



# Single Responsibility Principle

```
1  class modem
2  {
3      public:
4          void dial();
5          void hangup();
6          void send();
7          void rcv();
8  }
```

# Single Responsibility Principle

```
1  class modem
2  {
3      public:
4          void dial();    // Connection management
5          void hangup();  // Connection management
6          void send();
7          void rcv();
8  }
```

# Single Responsibility Principle

```
1  class modem
2  {
3      public:
4          void dial();
5          void hangup();
6          void send(); // Data Management
7          void rcv();  // Data Management
8  }
```

# Single Responsibility Principle

```
1  class modem_connection
2  {
3      public:
4          void dial();
5          void hangup();
6      }
7
8  class modem_data
9  {
10     public:
11         void send();
12         void rcv();
13     }
14
15     class modem_impl
16     {
17     private:
18         modem_connection connection;
19         modem_data      data;
20     }
```

# Single Responsibility Principle

Caution:

# Single Responsibility Principle

Caution:

- ▶ Too much splitting of modules can lead to an overly complicated design.



# Single Responsibility Principle

Caution:

- ▶ Too much splitting of modules can lead to an overly complicated design.
- ▶ If the code does not change in a way that the two responsibilities change at different times, then there's no need to separate.

# Open Closed Principle

# Open Closed Principle

- ▶ "Open for Extension"

# Open Closed Principle

- ▶ "Open for Extension"
  - ▶ Behavior of the module can be modified through extension

# Open Closed Principle

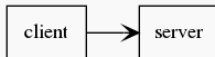
- ▶ "Open for Extension"
  - ▶ Behavior of the module can be modified through extension
- ▶ "Closed for Modification"

# Open Closed Principle

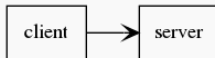
- ▶ "Open for Extension"
  - ▶ Behavior of the module can be modified through extension
- ▶ "Closed for Modification"
  - ▶ Extending the behavior requires no change in source code or binary executables.



# Open Closed Principle

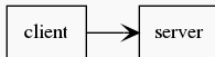


# Open Closed Principle



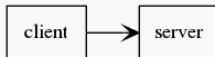
- ▶ Client depends on server

# Open Closed Principle



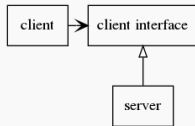
- ▶ Client depends on server
- ▶ Changing server requires modification of client

# Open Closed Principle

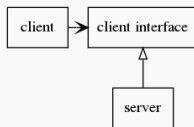


- ▶ Client depends on server
- ▶ Changing server requires modification of client
- ▶ Use of clients with different servers requires duplication of code

# Open Closed Principle



# Open Closed Principle



- ▶ Enables client implementations for multiple servers



# Open Closed Principle

```
1  // shape.h //////////////////////////////////////
2
3  enum shape_type_t { circle, square };
4  struct shape_s {
5      shape_type_t shape_type;
6  }
7
8  // circle.h //////////////////////////////////////
9  #include "shape.h"
10 struct circle_s {
11     shape_type_t shape_type;
12     double      radius;
13     point       center;
14 }
15
16 void drawCircle(struct circle_s *);
17
18 // square.h //////////////////////////////////////
19 #include "shape.h"
20 struct square_s {
21     shape_type_t shape_type;
22     double      side;
23     point       top_left;
24 }
25
26 void drawSquare(struct square_s *);
```

# Open Closed Principle

```
1 // shape.h ////////////////////////////////////////
2 // Adding a new shape, requires modification of enum
3 enum shape_type_t { circle, square };
4 struct shape_s {
5     shape_type_t shape_type;
6 }
7
8 // circle.h ////////////////////////////////////////
9 #include "shape.h"
10 struct circle_s {
11     shape_type_t shape_type;
12     double       radius;
13     point        center;
14 }
15
16 void drawCircle(struct circle_s *);
17
18 // square.h ////////////////////////////////////////
19 #include "shape.h"
20 struct square_s {
21     shape_type_t shape_type;
22     double       side;
23     point        top_left;
24 }
25
26 void drawSquare(struct square_s *);
```

# Open Closed Principle

```
1 // draw_all_shapes.c //////////////////////////////////////
2
3 typedef struct shape_t *shape_pointer_t;
4
5 void DrawAllShapes(shape_pointer_t *shapes, int n)
6 {
7     for (int i = 0; i < n; i++) {
8         struct shape_s *s = shapes[i];
9         switch (shape->shape_type) {
10            case circle:
11                drawCircle((struct circle_s *)shape);
12                break;
13            case square:
14                drawSquare((struct square_s *)shape);
15                break;
16        }
17    }
18 }
```

# Open Closed Principle

```
1 // shape.h ////////////////////////////////////////
2 enum shape_type_t { circle, square };
3 void (*DrawFunction)(void *);
4 struct shape_s {
5     DrawFunction draw;
6 }
7 void DrawShape(void *);
8
9 // shape.c ////////////////////////////////////////
10 void DrawShape(void * shape_in){
11     shape = (struct shape_s *) shape_in;
12     shape.draw(shape_in);
13 }
14
15 // circle.h ////////////////////////////////////////
16 struct circle_s {
17     DrawFunction draw;
18     double radius;
19     point center;
20 }
21
22 void drawCircle(struct circle_s *);
23
24 // square.h ////////////////////////////////////////
25 struct square_s {
26     DrawFunction draw;
27     double side;
28     point top_left;
29 }
30
31 void drawSquare(struct square_s *);
```

# Open Closed Principle

```
1 // draw_all_shapes.c //////////////////////////////////////
2
3 typedef struct shape_t *shape_pointer_t;
4
5 void DrawAllShapes(shape_pointer_t *shapes, int n)
6 {
7     for (int i = 0; i < n; i++) {
8         struct shape_s *shape = shapes[i];
9         DrawShape(shape);
10    }
11 }
```

# Liskov Substitution Principle

# Interface Segregation Principle

# Dependency Inversion Principle

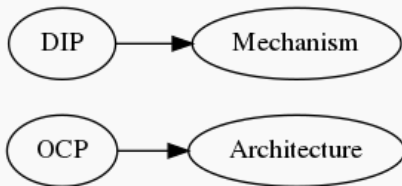
Depend upon abstractions.



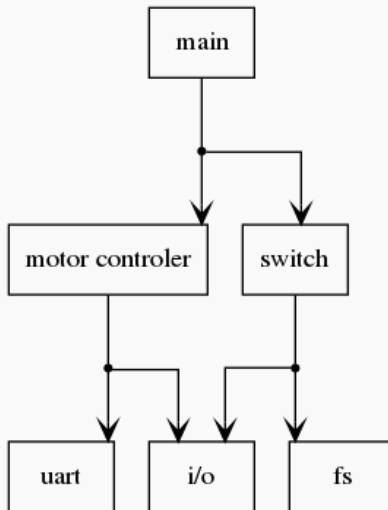
# Dependency Inversion Principle

Depend upon abstractions. Do not depend upon concretions.

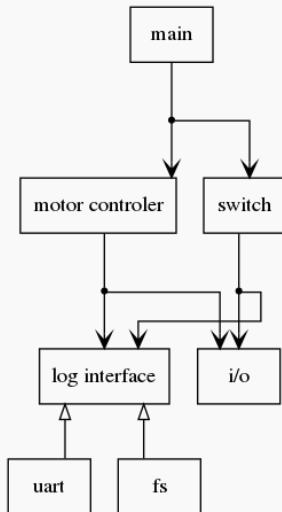
# Dependency Inversion Principle



# Dependency Inversion Principle



# Dependency Inversion Principle



# Package Design

---

Introduction

Symptoms of Rotting Design

Principles of Object Oriented Class Design

**Package Design**

Architecture Design

Conclusion

# Principles of Package Architecture

## Package Principles

# Principles of Package Architecture

## Package Principles

- ▶ Package Cohesion

# Principles of Package Architecture

## Package Principles

- ▶ Package Cohesion
- ▶ Package Coupling



# Principles of Package Architecture

## Package Principles

- ▶ Package Cohesion
  - ▶ Release Reuse Equivalency Principle (REP)
- ▶ Package Coupling

# Principles of Package Architecture

## Package Principles

- ▶ Package Cohesion
  - ▶ Release Reuse Equivalency Principle (REP)
  - ▶ Common Closure Principle (CCP)
- ▶ Package Coupling

# Principles of Package Architecture

## Package Principles

- ▶ Package Cohesion
  - ▶ Release Reuse Equivalency Principle (REP)
  - ▶ Common Closure Principle (CCP)
  - ▶ Common Reuse Principle (CRP)
- ▶ Package Coupling

# Principles of Package Architecture

## Package Principles

- ▶ Package Cohesion
  - ▶ Release Reuse Equivalency Principle (REP)
  - ▶ Common Closure Principle (CCP)
  - ▶ Common Reuse Principle (CRP)
- ▶ Package Coupling
  - ▶ Acyclic Dependencies Principle (ADP)

# Principles of Package Architecture

## Package Principles

- ▶ Package Cohesion
  - ▶ Release Reuse Equivalency Principle (REP)
  - ▶ Common Closure Principle (CCP)
  - ▶ Common Reuse Principle (CRP)
- ▶ Package Coupling
  - ▶ Acyclic Dependencies Principle (ADP)
  - ▶ Stable Dependencies Principle (SDP)

# Principles of Package Architecture

## Package Principles

- ▶ Package Cohesion
  - ▶ Release Reuse Equivalency Principle (REP)
  - ▶ Common Closure Principle (CCP)
  - ▶ Common Reuse Principle (CRP)
- ▶ Package Coupling
  - ▶ Acyclic Dependencies Principle (ADP)
  - ▶ Stable Dependencies Principle (SDP)
  - ▶ Stable Abstractions Principle (SAP)

# Dependency Inversion Principle

# Architecture Design

---

Introduction

Symptoms of Rotting Design

Principles of Object Oriented Class Design

Package Design

**Architecture Design**

Conclusion



# Principles of Package Architecture

# Conclusion

---

Introduction

Symptoms of Rotting Design

Principles of Object Oriented Class Design

Package Design

Architecture Design

Conclusion

# Principles of Package Architecture

## References

- ▶ [https://fi.ort.edu.uy/innovaportal/file/2032/1/design\\_principles.pdf](https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf)
- ▶ <http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>
- ▶ <http://notherdev.blogspot.com/2013/07/code-smells-rigidity.html>
- ▶ <https://dev.to/bob/how-do-you-know-your-code-is-bad>
- ▶ [http://staff.cs.utu.fi/~jounsmed/doos\\_06/slides/slides\\_060321.pdf](http://staff.cs.utu.fi/~jounsmed/doos_06/slides/slides_060321.pdf)
- ▶ <https://softwareengineering.stackexchange.com/questions/357127/clear-examples-for-code-smells>

# Questions