

# Design Principles and Design Patterns

**D. Ryan Bartling**

March 23, 2019

# Outline

Introduction

Symptoms of Rotting Design

Principles of Object Oriented Class Design

Package Design

Architecture Design

Conclusion

# Introduction

## Introduction

## Symptoms of Rotting Design

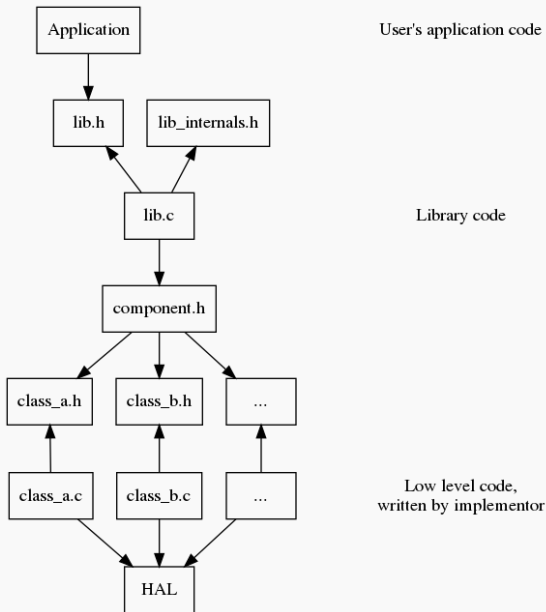
## Principles of Object Oriented Class Design

## 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

- ▶ Overly procedural code



# Rigidity

How it happens

- ▶ Overly procedural code
- ▶ Lack of abstractions

# Rigidity

How it happens

- ▶ Overly procedural code
- ▶ Lack of abstractions
- ▶ Solving a generic problem with implementation specific details

# Rigidity

How it happens

- ▶ Overly procedural code
- ▶ Lack of abstractions
- ▶ Solving a generic problem with implementation specific details
- ▶ Spreading a single responsibility throughout several parts

# Rigidity

How it happens

- ▶ Overly procedural code
- ▶ 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);
```

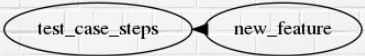
# Rigidity



new\_feature



# Rigidity

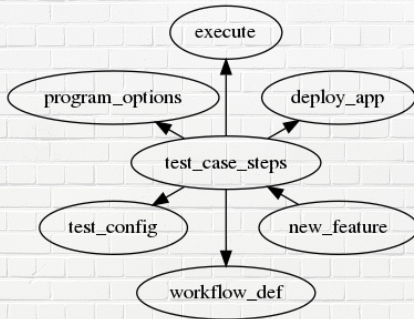


A diagram illustrating rigidity in software design. It features two overlapping ovals on a white brick wall background. The left oval is labeled 'test\_case\_steps' and the right oval is labeled 'new\_feature'. A small black arrow points from the 'test\_case\_steps' oval to the 'new\_feature' oval, indicating a dependency or constraint that makes adding new features difficult.

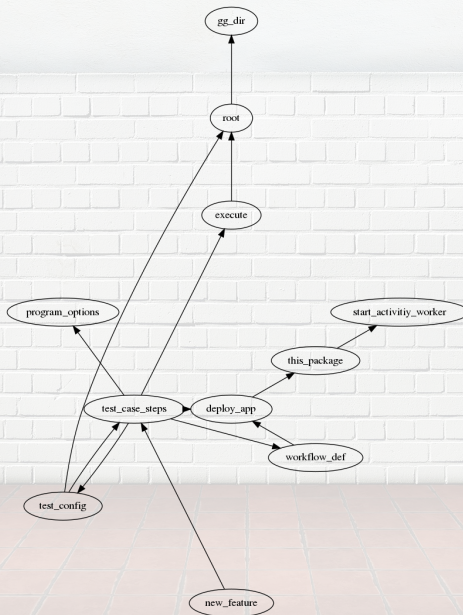
test\_case\_steps

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

# Frailty

```
1  void
2  sdcard_init(void) {
3      spi_init(mode_0, card_cs_pin);
4      fat_init();
5  }
6
7  void
8  sensor_init(void) {
9      spi_init(mode_0, sensor_cs_pin);
10     spi_write(SENSOR_CONFIGURATION, sensor_cs_pin);
11 }
```

# Frailty

Changing the sensor to use mode 1...

```
1  void
2  sdcard_init(void) {
3      spi_init(mode_0, card_cs_pin);
4      fat_init();
5  }
6
7  void
8  sensor_init(void) {
9      spi_init(mode_1, sensor_cs_pin); // Breaks the sd card
10     spi_write(SENSOR_CONFIGURATION, sensor_cs_pin);
11 }
```

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



# Frailty

We can fix it with dynamic resource allocation...

```
1  void
2  sdcard_init(void) {
3      if(spi_success != spi_acquire(mode_0, card_cs_pin)) { return; }
4      fat_init();
5      spi_release();
6  }
7
8  void
9  sensor_init(void) {
10     if(spi_success != spi_acquire(mode_1, sensor_cs_pin)) { return; }
11     spi_write(SENSOR_CONFIGURATION, sensor_cs_pin);
12     spi_release();
13 }
```



# Frangility

If multi threaded, we could spin lock...

```
1  void
2  sdcard_init(void) {
3      while(spi_success != spi_acquire(mode_0, card_cs_pin)) {}
4      fat_init();
5      spi_release();
6  }
7
8  void
9  sensor_init(void) {
10     while(spi_success != spi_acquire(mode_1, sensor_cs_pin)) {}
11     spi_write(SENSOR_CONFIGURATION, sensor_cs_pin);
12     spi_release();
13 }
```

# Fragility

We could also have a common allocation and assert correctness...

```
1  void
2  sys_init(void) {
3      spi_init(mode_0);
4  }
5
6  void
7  sdcard_init(void) {
8      assert(mode_0 == spi_mode_get() && "Wrong spi mode for sdcard");
9      fat_init();
10 }
11
12 void
13 sensor_init(void) {
14     assert(mode_0 == spi_mode_get() && "Wrong spi mode for sensor");
15     spi_write(SENSOR_CONFIGURATION, sensor_cs_pin);
16 }
```

# Fragility

How to avoid it

# Fragility

How to avoid it

- ▶ Explicit dependencies

# Fragility

How to avoid it

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

# Fragility

How to avoid it

- ▶ Explicit 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

- ▶ Explicit 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

- ▶ Explicit 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

```
1  #include "temperature_sensor.h"
2
3  #include <mcu.h>
4  #include <stdint.h>
5
6  uint16_t
7  oven_temperature(void) {
8      adccon |= 1 << 3;           // Start adc conversion
9      while(!(adccon & (1 << 0))) {} // While not done
10     return ((adsamp * 53) / 7);
11 }
```

# Immobility

```
1  #include "temperature_sensor.h"
2
3  #include <mcu.h>
4  #include <stdint.h>
5
6  uint16_t
7  oven_temperature(void) {
8      ADC1_start_conversion();
9      while(!ADC1_done()) {}
10     return ((ADC1_sample_get() * 53) / 7);
11 }
```

# Immobility

```
1  #include "temperature_sensor.h"
2
3  #include <mcu.h>
4  #include <stdint.h>
5
6  // TPS = Temperature Sensor
7
8  static uint16_t const
9  TPS_get_adc_sample(void) {
10     ADC1_start_conversion();
11     while(!ADC1_done()) {}
12     return ADC1_sample_get();
13 }
14
15 static uint16_t const
16 TPS_adc_counts_to_F(uint16_t const adc_sample) {
17     return ((ADC1_sample_get() * 53) / 7);
18 }
19
20 uint16_t
21 TPS_oven_temperature_F(void) {
22     uint16_t sample = TPS_get_adc_sample();
23     return TPS_adc_counts_to_F(sample);
24 }
```



# Immobility

```
1  #include "temperature_sensor.h"
2
3  #include <mcu.h>
4  #include <stdint.h>
5
6  // TPS = Temperature Sensor
7
8  static uint16_t const
9  TPS_get_adc_sample(void) {
10     ADC1_start_conversion();
11     while(!ADC1_done()) {}
12     return ADC1_sample_get();
13 }
14
15 static int const
16 TPS_adc_counts_to_F(uint16_t const adc_sample) {
17     return ((ADC1_sample_get() * 53) / 7);
18 }
19
20 int
21 TPS_temperature_F(void) {
22     uint16_t sample = TPS_get_adc_sample();
23     return TPS_adc_counts_to_F(sample);
24 }
```

# Immobility

```
1  #include "temperature_sensor.h"
2
3  #include <mcu.h>
4  #include <stdint.h>
5
6  // TPS = Temperature Sensor
7
8  static uint16_t const
9  TPS_get_adc_sample(void) {
10     ADC1_start_conversion();
11     while(!ADC1_done()) {}
12     return ADC1_sample_get();
13 }
14
15 static int const
16 TPS_adc_counts_to_F(uint16_t const adc_sample) {
17     return ((ADC1_sample_get() * 53) / 7);
18 }
19
20 int
21 TPS_temperature_F(void) {
22     uint16_t sample = TPS_get_adc_sample();
23     return TPS_adc_counts_to_F(sample);
24 }
25
26 int
27 TPS_temperature_C(void) {
28     int temperature_F = TPS_temperature_F();
29     return ((temperature_F - 32) * 5) / 9
30 }
```

# Immobility

```
1  #include "temperature_sensor.h"
2
3  // TPS = Temperature Sensor
4
5  static int const
6  TPS_adc_counts_to_F(int const adc_sample) {
7      return ((ADC1_sample_get() * 53) / 7);
8  }
9
10 static int const
11 TPS_F_to_C(int const temperature_F) {
12     return ((temperature_F - 32) * 5) / 9;
13 }
14
15 int
16 TPS_temperature_F(int const adc_sample) {
17     return TPS_adc_counts_to_F(adc_sample);
18 }
19
20 int
21 TPS_temperature_C(int const adc_sample) {
22     int temperature_F = TPS_temperature_F(adc_sample);
23     return TPS_F_to_C(temperature_F);
24 }
```

# Immobility

```
1 // temperature_sensor.h
2
3 typedef int (*counts_to_F_function)(int const /*adc_counts*/);
4 ...
5
6 // temperature_sensor.c
7 #include <assert.h>
8 #include <stddef.h>
9
10 static counts_to_F_function adc_counts_to_degrees_F = NULL;
11
12 void
13 TPS_set_temperature_conversion(counts_to_F_function user_function) {
14     adc_counts_to_degrees_F = user_function;
15 }
16
17 int
18 TPS_temperature_F(int const adc_sample) {
19     assert(
20         (NULL != adc_counts_to_degrees_F)
21         && "You must call TPS_set_temperature_conversion first");
22     return adc_counts_to_degrees_F(adc_sample);
23 }
```

# Immobility

How to prevent immobility

- ▶ Depend upon the concept, not the details

# Immobility

How to prevent immobility

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

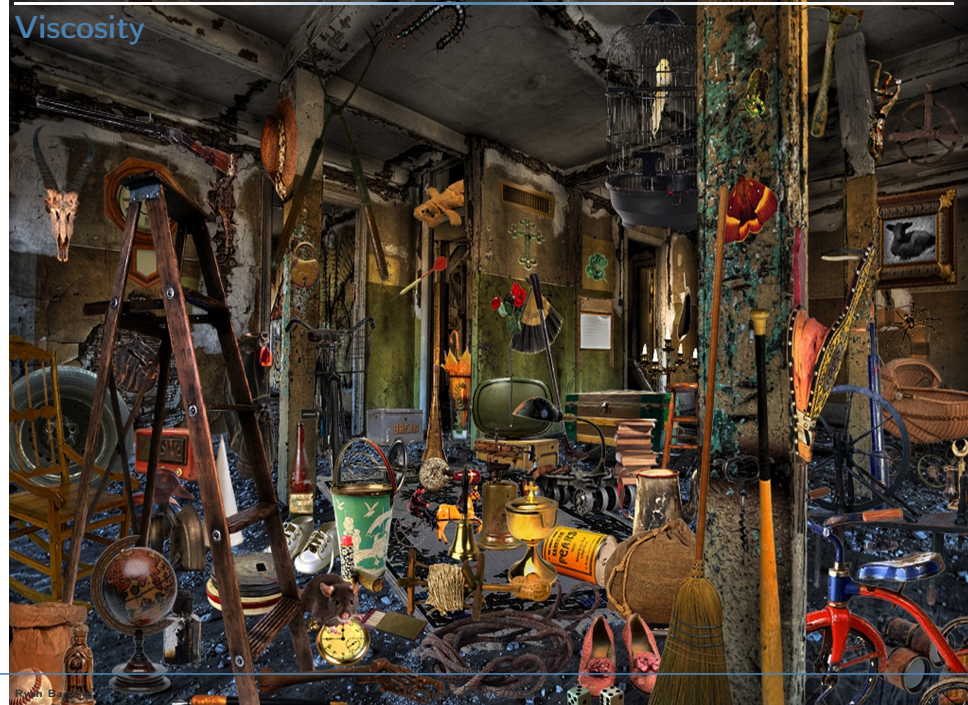


# Immobility

How to prevent immobility

- ▶ Depend upon the concept, not the details
- ▶ Reduce responsibilities to solve distinct problems
- ▶ Write unit tests for the module at the time that you write the module.

## Viscosity



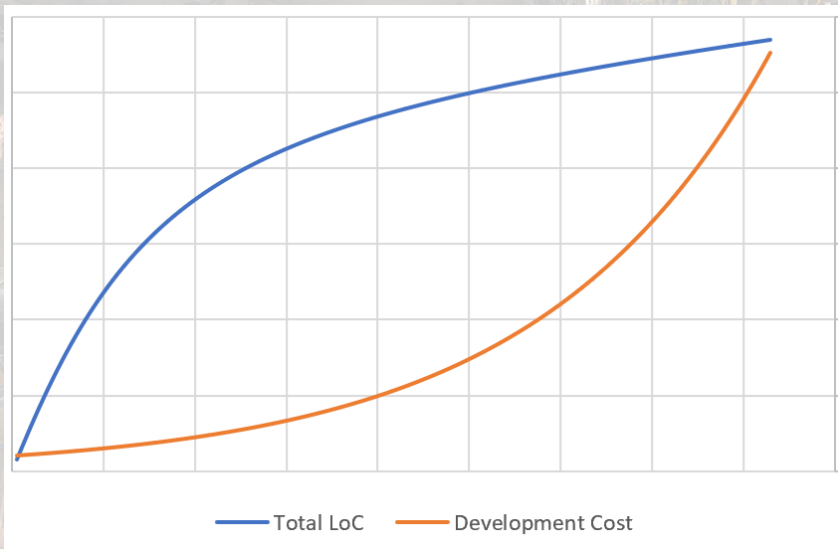
# Viscosity

- ▶ 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.

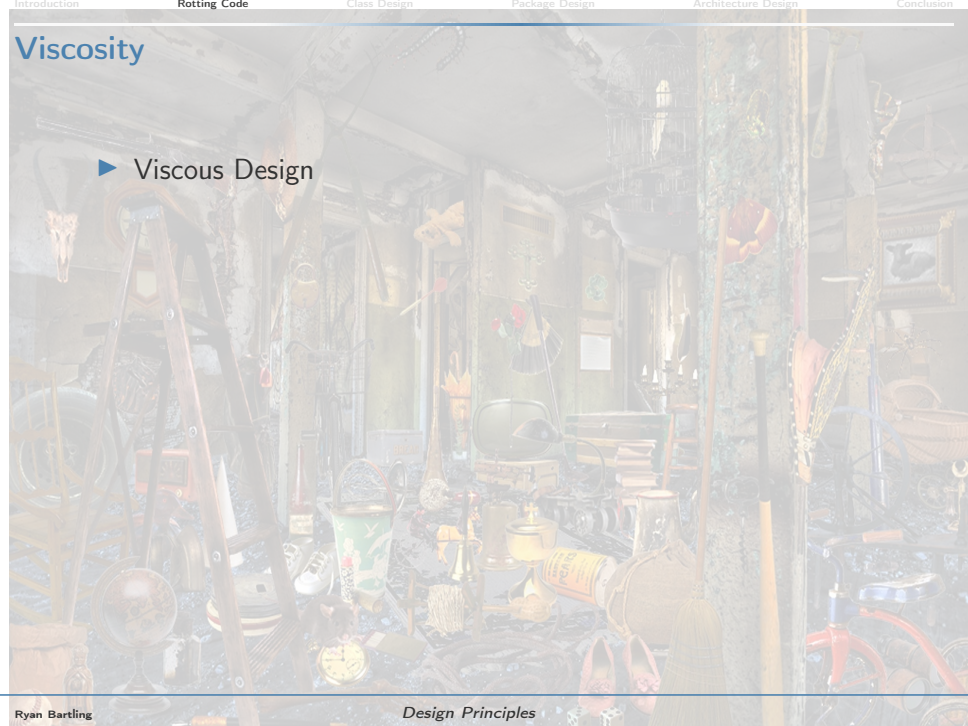
# Viscosity





# Viscosity

## ► 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/cumbersome tools (e.g. large complicated files may require longer static analysis)

# 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.

# 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.
    - ▶ What code changes require stricter review?



# 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.
    - ▶ What code changes require stricter review?
    - ▶ What code changes require new or updated documentation?

# 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.
    - ▶ What code changes require stricter review?
    - ▶ What code changes require new or updated documentation?
    - ▶ When does a code revision require upfront design?

# 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  public:  
3      void  
4      dial();  
5      void  
6      hangup();  
7      void  
8      send();  
9      void  
10     rcv();  
11 }
```

# Single Responsibility Principle

```
1  class modem {  
2  public:  
3      void  
4      dial(); // Connection management  
5      void  
6      hangup(); // Connection management  
7      void  
8      send();  
9      void  
10     rcv();  
11 }
```

# Single Responsibility Principle

```
1  class modem {  
2  public:  
3      void  
4      dial();  
5      void  
6      hangup();  
7      void  
8      send(); // Data Management  
9      void  
10     rcv();  // Data Management  
11 }
```

# Single Responsibility Principle

```
1  class modem_connection {
2  public:
3      void
4      dial();
5      void
6      hangup();
7  }
8
9  class modem_data {
10 public:
11     void
12     send();
13     void
14     rcv();
15 }
16
17 class modem_impl {
18 private:
19     modem_connection connection;
20     modem_data      data;
21 }
```



# 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



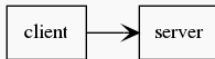
# Open Closed Principle

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

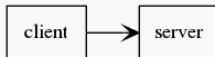
# Open Closed Principle

- ▶ "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

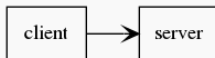


# Open Closed Principle



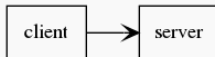
- ▶ Client depends on server

# Open Closed Principle



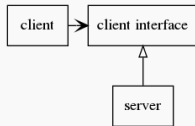
- ▶ Client depends on server
- ▶ Changing server requires modification of client

# Open Closed Principle



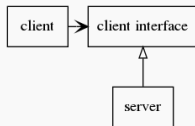
- ▶ Client depends on server
- ▶ Changing server requires modification of client
- ▶ Use of clients with different servers requires duplication of code

# 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  // circle.h ////////////////////////////////////////
8  #include "shape.h"
9  struct circle_s {
10     shape_type_t shape_type;
11     double      radius;
12     point       center;
13 }
14
15 void
16 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
27 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  // circle.h ////////////////////////////////////////
8  #include "shape.h"
9  struct circle_s {
10     shape_type_t shape_type;
11     double      radius;
12     point       center;
13 }
14
15 void
16 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
27 drawSquare(struct square_s *);
```

# Open Closed Principle

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

# 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 } void
7 DrawShape(void *);
8
9 // shape.c ////////////////////////////////////////
10 void
11 DrawShape(void *shape_in) {
12     shape = (struct shape_s *)shape_in;
13     shape.draw(shape_in);
14 }
15
16 // circle.h ////////////////////////////////////////
17 struct circle_s {
18     DrawFunction draw;
19     double radius;
20     point center;
21 }
22
23 void
24 drawCircle(struct circle_s *);
25
26 // square.h ////////////////////////////////////////
27 struct square_s {
28     DrawFunction draw;
29     double side;
30     point top_left;
31 }
32
```

# Open Closed Principle

```
1 // draw_all_shapes.c //////////////////////////////////////
2
3 typedef struct shape_t *shape_pointer_t;
4
5 void
6 DrawAllShapes(shape_pointer_t *shapes, int n) {
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

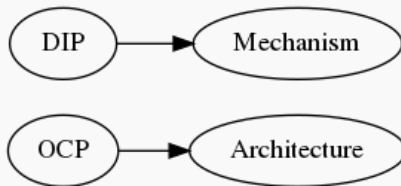
# Dependency Inversion Principle

Depend upon abstractions.

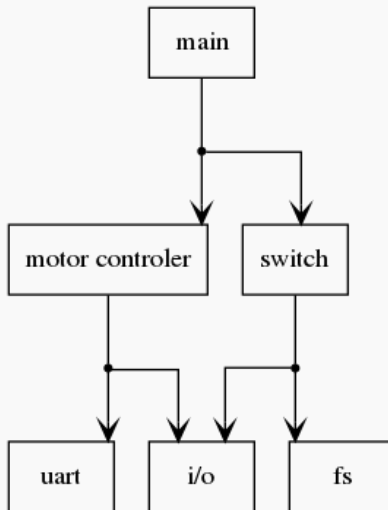
# Dependency Inversion Principle

Depend upon abstractions. Do not depend upon concretions.

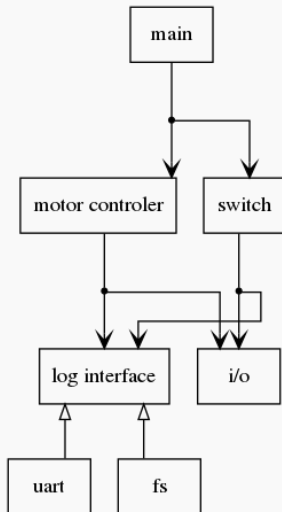
# Dependency Inversion Principle



# 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