

# Design Principles and Design Patterns

D. Ryan Bartling

February 2, 2020

---

# Outline

Introduction

Symptoms of Rotting Design

Principles of Object Oriented Class Design

Package Design

Architecture Design

Conclusion

---

# Introduction

## Introduction

## Symptoms of Rotting Design

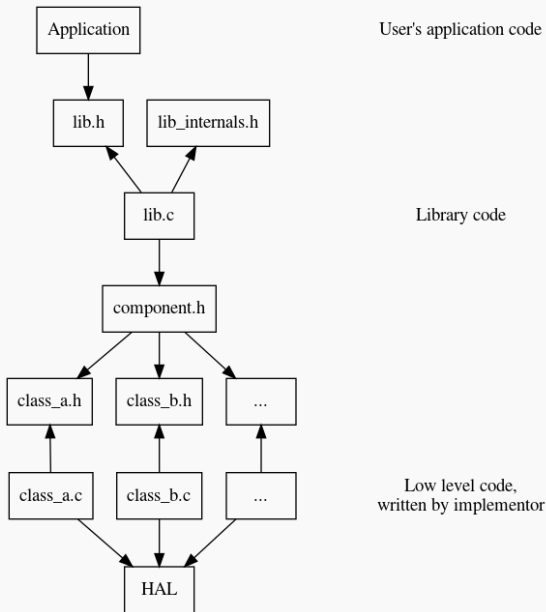
## Principles of Object Oriented Class Design

## 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  #include <stdlib.h>
3  #ifndef INT24_MAX
4  typedef int32_t int24_t;
5  #endif
6
7  #define ADC_BITS (12)
8  #define ADC_DATA_SHIFT (2)
9  #define ADC_SIGN_CONVERSION (1)
10 #define RAW_ADC_BITS (15) // Sum of the above bits
11
12 #define LFSR_LENGTH (4)
13 #define LFSR_REPEATS (2)
14 #define CORRELATED_BITS (20) // ADC bits + lfsr length + log2(repeats)
15
16 typedef int16_t rpo_raw_adc_t;
17 typedef int24_t rpo_correlated_int_t;
```



# Rigidity

```
1  #include <stdint.h>
2  #ifndef INT24_MAX
3  typedef int32_t int24_t;
4  #endif
5
6  #define ADC_BITS (14) // Changing this
7  #define ADC_DATA_SHIFT (2)
8  #define ADC_SIGN_CONVERSION (1)
9  #define RAW_ADC_BITS (17) // Changes this
10
11 #define LFSR_LENGTH (4)
12 #define LFSR_REPEATS (2)
13 #define CORRELATED_BITS (22) // Changes this
14
15 typedef int24_t rpo_raw_adc_t; // Changes this
16 typedef int24_t rpo_correlated_int_t;
```

# Rigidity

## Refactor to reduce rigidity

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


# Rigidity



new\_feature



# Rigidity

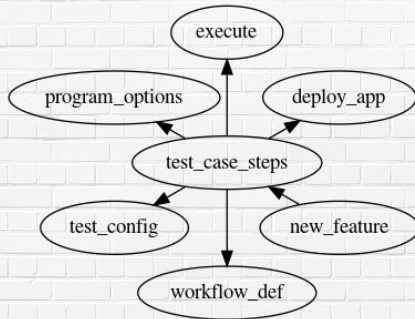


The diagram shows two overlapping ovals, one labeled 'test\_case\_steps' on the left and one labeled 'new\_feature' on the right. A small black arrow points from the 'new\_feature' oval to the 'test\_case\_steps' oval, indicating a dependency or relationship between the two components.

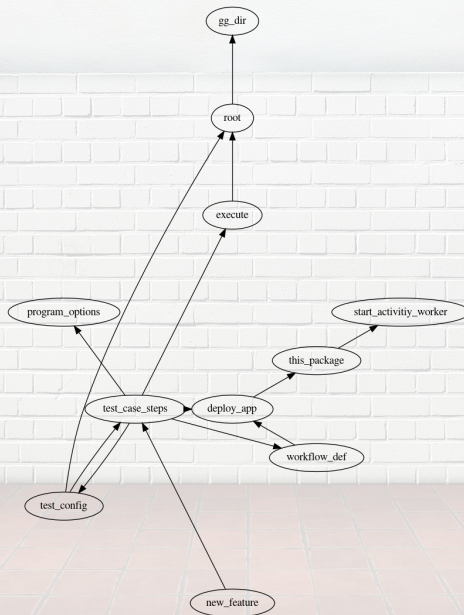
test\_case\_steps

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

# Frangility

```
1  #include "mcu.h"
2
3  void sdcard_init(void) {
4      spi_init(mode_0);
5      fat_init();
6  }
7
8  void sensor_init(void) {
9      spi_init(mode_0);
10     spi_write(SENSOR_CONFIGURATION, sensor_cs_pin);
11 }
```

# Fragility

Changing the sensor to use mode 1...

```
1  #include "mcu.h"
2
3  void sdcard_init(void) {
4      spi_init(mode_0, card_cs_pin);
5      fat_init();
6  }
7
8  void 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)



# Fracture

We can fix it with dynamic resource allocation...

```
1  #include "mcu.h"
2
3  void sdcard_init(void) {
4      if (spi_success != spi_acquire(mode_0, card_cs_pin)) { return; }
5      fat_init();
6      spi_release();
7  }
8
9  void 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  #include "mcu.h"
2
3  void sdcard_init(void) {
4      while (spi_success != spi_acquire(mode_0, card_cs_pin)) {}
5      fat_init();
6      spi_release();
7  }
8
9  void 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  #include "mcu.h"
2
3  void sys_init(void) { spi_init(mode_0); }
4
5  void sdcard_init(void) {
6      assert(mode_0 == spi_mode_get() && "Wrong spi mode for sdcard");
7      fat_init();
8  }
9
10 void sensor_init(void) {
11     assert(mode_0 == spi_mode_get() && "Wrong spi mode for sensor");
12     spi_write(SENSOR_CONFIGURATION, sensor_cs_pin);
13 }
```

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

# Immobility

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

# Immobility

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



# Immobility

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

# Immobility

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

# Immobility

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

# 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 TPS_set_temperature_conversion(counts_to_F_function user_function) {
13     adc_counts_to_degrees_F = user_function;
14 }
15
16 int TPS_temperature_F(int const adc_sample) {
17     assert((NULL != adc_counts_to_degrees_F) &&
18         "You must call TPS_set_temperature_conversion first");
19     return adc_counts_to_degrees_F(adc_sample);
20 }
```

# 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



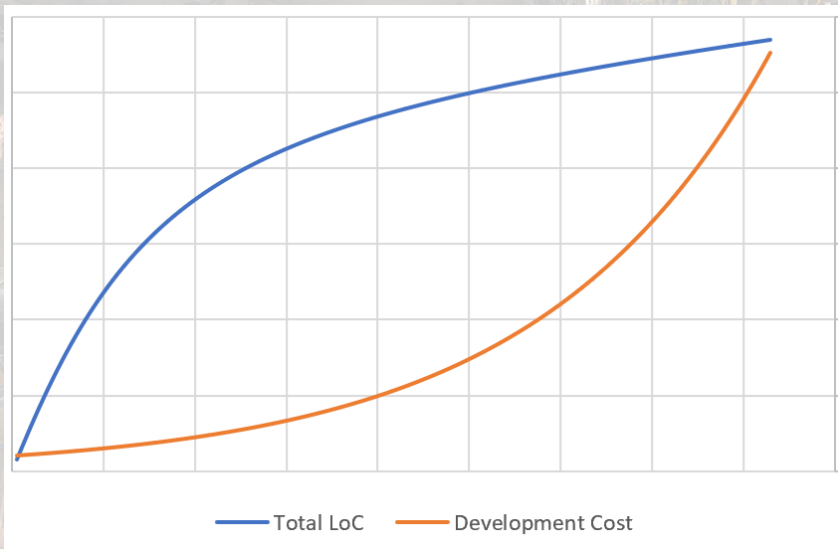
# Viscosity

- ▶ 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.

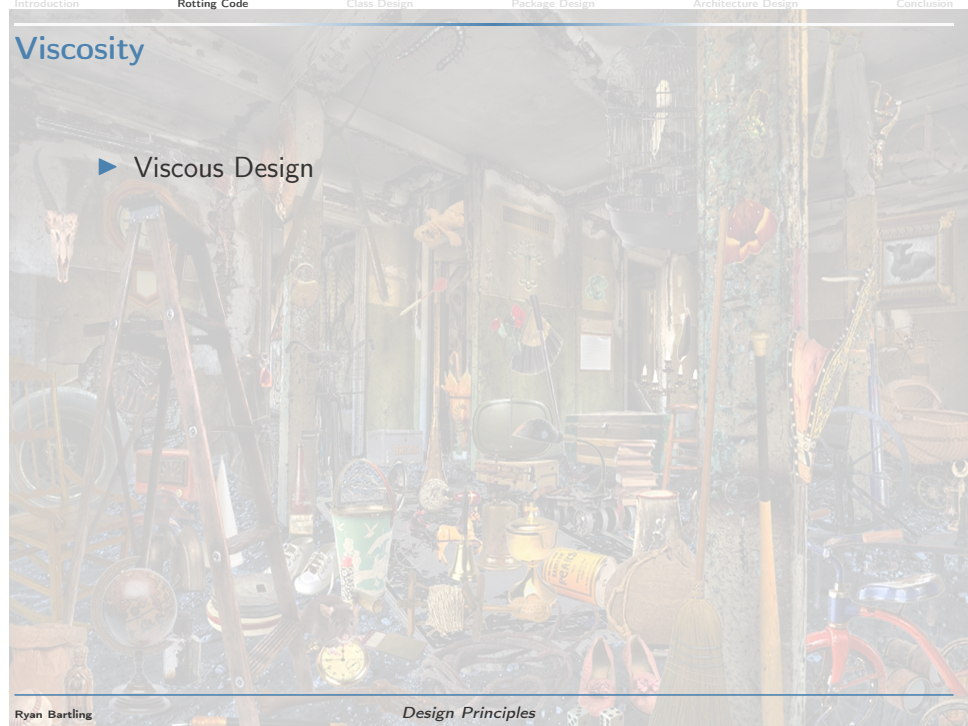
# Viscosity





# 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 dial();  
4          void hangup();  
5          void send();  
6          void rcv();  
7  };
```

# Single Responsibility Principle

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

# Single Responsibility Principle

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

# Single Responsibility Principle

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



# 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



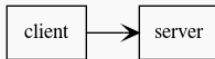
# Open Closed Principle

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

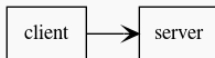
# Open Closed Principle

- ▶ "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

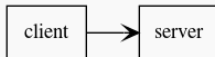


# Open Closed Principle



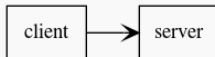
- ▶ Client depends on server

# Open Closed Principle



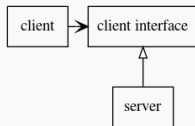
- ▶ Client depends on server
- ▶ Changing server requires modification of client

# Open Closed Principle



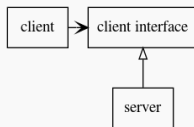
- ▶ Client depends on server
- ▶ Changing server requires modification of client
- ▶ Use of clients with different servers requires duplication of code

# Open Closed Principle





# Open Closed Principle



- ▶ Enables client implementations for multiple servers

# Open Closed Principle

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

# Open Closed Principle

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

# Open Closed Principle

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

# Open Closed Principle

```
1 // shape.h ////////////////////////////////////////
2 typedef void (*draw_function_t)(void *);
3 typedef struct point_s {
4     double x;
5     double y;
6 } point;
7 struct shape_s {
8     draw_function_t draw;
9 };
10 void draw_shape(void *);
11 // shape.c ////////////////////////////////////////
12 void draw_shape(void *shape_in) {
13     struct shape_s *shape = (struct shape_s *)shape_in;
14     shape->draw(shape);
15 }
16
17 // circle.h ////////////////////////////////////////
18 struct circle_s {
19     draw_function_t draw;
20     double radius;
21     point center;
22 };
23 void drawCircle(struct circle_s *);
24 // square.h ////////////////////////////////////////
25 struct square_s {
26     draw_function_t draw;
27     double side;
28     point top_left;
29 };
30 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     for (int i = 0; i < n; i++) {
7         struct shape_s *shape = shapes[i];
8         DrawShape(shape);
9     }
10 }
```

# Liskov Substitution Principle

Subtypes are substitutable for their base types.



# Liskov Substitution Principle

Subtypes are substitutable for their base types.

If A is a base class, and B inherits from A, then B can be used as A.

# Liskov Substitution Principle

Subtypes are substitutable for their base types.

If A is a base class, and B inherits from A, then B can be used as A.

Don't surprise users with unexpected changes in behavior.

# Liskov Substitution Principle

```
1 // rectangle.h ////////////////////////////////////////
2 typedef struct rectangle_s {
3     double width;
4     double height;
5     double (*area)(struct rectangle_s *);
6     void (*set_width)(struct rectangle_s *, double);
7     void (*set_height)(struct rectangle_s *, double);
8 } rectangle_t;
9 rectangle_t *REC_construct(void);
10 // rectangle.c ////////////////////////////////////////
11 static void REC_set_width(struct rectangle_s *r, double w) { r->width = w; }
12 static void REC_set_height(struct rectangle_s *r, double h) { r->height = h; }
13 // square.h ////////////////////////////////////////
14 typedef struct rectangle_s square_t;
15 square_t *SQ_construct(void);
16 // square.c ////////////////////////////////////////
17 #include <stdlib.h>
18 static void SQ_set_side(square_t *sq, double s) {
19     sq->width = s;
20     sq->height = s;
21 }
22 square_t *SQ_construct(void) {
23     square_t *sq = calloc(1, sizeof(square_t));
24     sq->set_width = SQ_set_side;
25     sq->set_height = SQ_set_side;
26     return sq;
27 }
```

# Liskov Substitution Principle

```
1  #include "lsp_1.c"
2  void bar(rectangle_t *r);
3
4  void foo(void) {
5      square_t *sq = SQ_construct();
6      bar(sq);
7  }
8
9  void bar(rectangle_t *r) {
10     r->set_height(r, 3);
11     r->set_width(r, 4);
12 }
```

# Liskov Substitution Principle

```
1  #include "lsp_1.c"
2
3  #include <assert.h>
4
5  void bar(rectangle_t *r);
6
7  void foo(void) {
8      square_t *sq = SQ_construct();
9      bar(sq);
10 }
11
12 void bar(rectangle_t *r) {
13     r->set_height(r, 3);
14     r->set_width(r, 4);
15     assert(r->area(r) == 12);
16 }
```

# Liskov Substitution Principle

Contract for `set_height()`:

# Liskov Substitution Principle

Contract for `set_height()`:

- ▶ Pre-conditions:



# Liskov Substitution Principle

Contract for `set_height()`:

- ▶ Pre-conditions:
  - ▶ Valid object pointer

# Liskov Substitution Principle

Contract for `set_height()`:

- ▶ Pre-conditions:
  - ▶ Valid object pointer
  - ▶  $0 \leq \text{new height value}$

# Liskov Substitution Principle

Contract for `set_height()`:

- ▶ Pre-conditions:
  - ▶ Valid object pointer
  - ▶  $0 \leq \text{new height value}$
- ▶ Post-conditions:

# Liskov Substitution Principle

Contract for `set_height()`:

- ▶ Pre-conditions:
  - ▶ Valid object pointer
  - ▶  $0 \leq \text{new height value}$
- ▶ Post-conditions:
  - ▶ Height matches the new value

# Liskov Substitution Principle

Contract for `set_height()`:

- ▶ Pre-conditions:
  - ▶ Valid object pointer
  - ▶  $0 \leq \text{new height value}$
- ▶ Post-conditions:
  - ▶ Height matches the new value
  - ▶ Width is unchanged

# Interface Segregation Principle

Allow users to use the parts of your library they need without concern over the parts they don't need.

# Dependency Inversion Principle

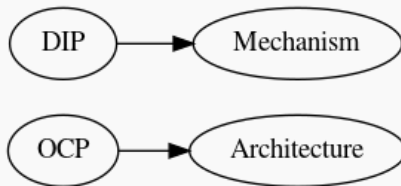
Depend upon abstractions.



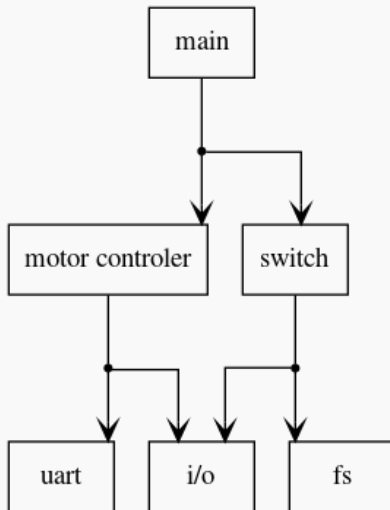
# Dependency Inversion Principle

Depend upon abstractions. Do not depend upon concretions.

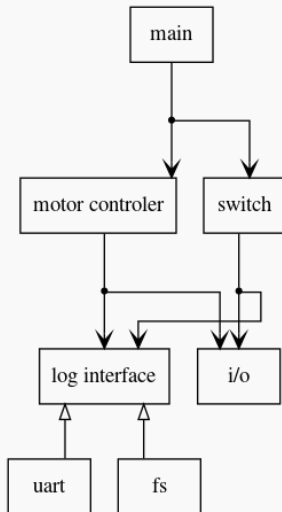
# Dependency Inversion Principle



# 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