Design Principles and Design Patterns

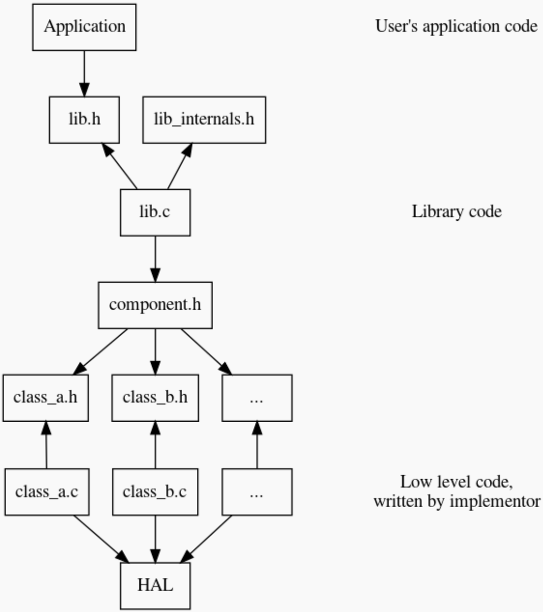
D. Ryan Bartling

February 6, 2020

Outline

Introduction

Architecture and Dependencies



Symptoms of Rotting Design

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

```c
#include <stdint.h>
#include <stdlib.h>
#ifndef INT24_MAX
typedef int32_t int24_t;
#endif

#define ADC_BITS (12)
#define ADC_DATA_SHIFT (2)
#define ADC_SIGN_CONVERSION (1)
#define RAW_ADC_BITS (15) // Sum of the above bits

#define LFSR_LENGTH (4)
#define LFSR_REPEATS (2)
#define CORRELATED_BITS (20) // ADC bits + lfsr length + log2(repeats)

typedef int16_t rpo_raw_adc_t;
typedef int24_t rpo_correlated_int_t;
```

Rigidity

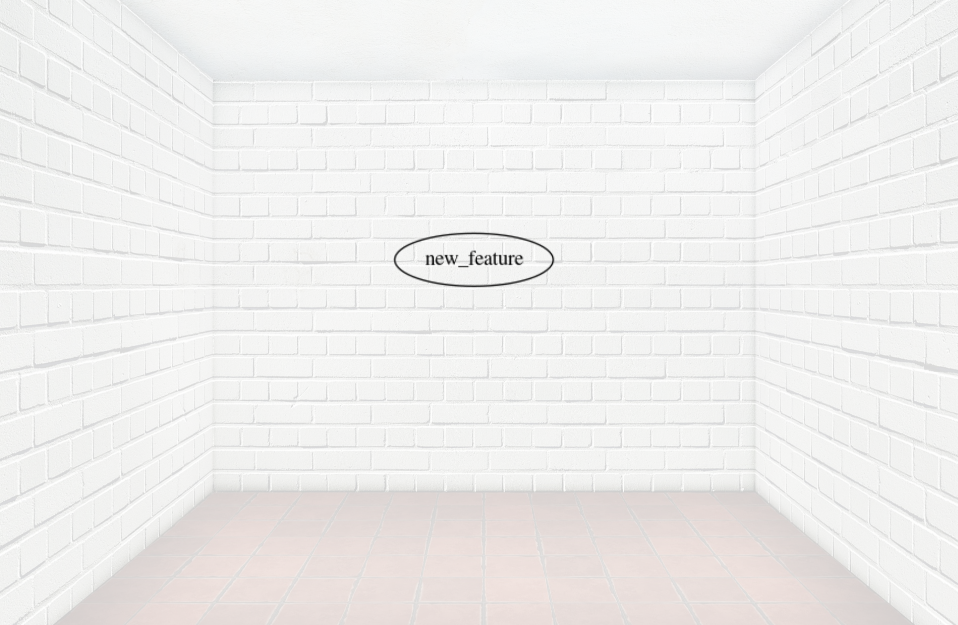
```c
#include <stdint.h>
#ifndef INT24_MAX
typedef int32_t int24_t;
#endif

#define ADC_BITS (14) // Changing this
#define ADC_DATA_SHIFT (2)
#define ADC_SIGN_CONVERSION (1)
#define RAW_ADC_BITS (17) // Changes this

#define LFSR_LENGTH (4)
#define LFSR_REPEATS (2)
#define CORRELATED_BITS (22) // Changes this

typedef int24_t rpo_raw_adc_t; // Changes this
typedef int24_t rpo_correlated_int_t;
```

Rigidity

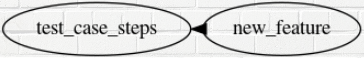
Refactor to reduce rigidity

```
#include "mcu.h"
#include <stdint.h>

#define ADC_BITS (14)
#define ADC_DATA_SHIFT (2)
#define ADC_SIGN_CONVERSION (1)
#define RAW_ADC_BITS (ADC_BITS + ADC_DATA_SHIFT + ADC_SIGN_CONVERSION)
typedef_min_int(rpo_raw_adc_t, RAW_ADC_BITS);

#define LFSR_LENGTH (4)
#define LFSR_REPEATS (2)
#define CORRELATED_BITS (RAW_ADC_BITS + LFSR_LENGTH + log_2(LFSR_REPEATS))
typedef_min_int(rpo_correlated_int_t, CORRELATED_BITS);
```

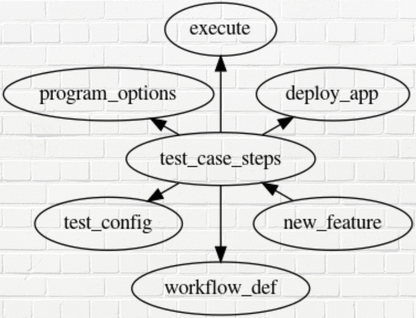
Rigidity



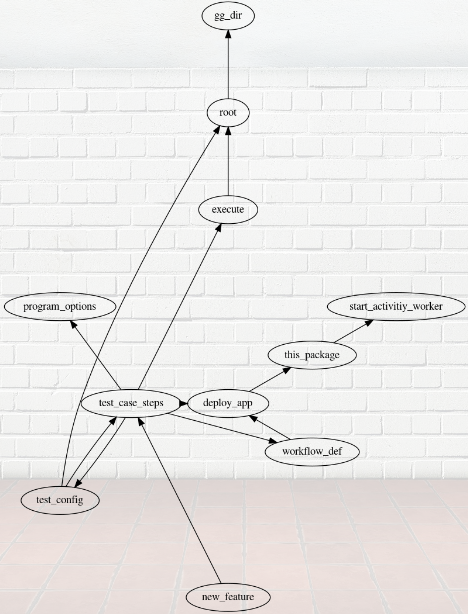
Rigidity



Rigidity


execute
program_options
deploy_app
test_case_steps
test_config
new_feature
workflow_def

Rigidity


gg_dir
root
execute
program_options
start_activitiy_worker
this_package
test_case_steps
deploy_app
workflow_def
test_config
new_feature

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

```c
#include "mcu.h"

void sdcard_init(void) {
    spi_init(mode_0);
    fat_init();
}

void sensor_init(void) {
    spi_init(mode_0);
    spi_write(SENSOR_CONFIGURATION, sensor_cs_pin);
}
```

Fragility

Changing the sensor to use mode 1...

```c
#include "mcu.h"

void sdcard_init(void) {
    spi_init(mode_0);
    fat_init();
}

void sensor_init(void) {
    spi_init(mode_1); // Breaks the sd card
    spi_write(SENSOR_CONFIGURATION, sensor_cs_pin);
}
```

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

Fragility

We can fix it with dynamic resource allocation...

```c
#include "mcu.h"

void sdcard_init(void) {
    if (spi_success != spi_acquire(mode_0, card_cs_pin)) { return; }
    fat_init();
    spi_release();
}

void sensor_init(void) {
    if (spi_success != spi_acquire(mode_1, sensor_cs_pin)) { return; }
    spi_write(SENSOR_CONFIGURATION, sensor_cs_pin);
    spi_release();
}
```

Fragility

If multi threaded, we could spin lock...

```c
#include "mcu.h"

void sdcard_init(void) {
    while (spi_success != spi_acquire(mode_0, card_cs_pin)) {}
    fat_init();
    spi_release();
}

void sensor_init(void) {
    while (spi_success != spi_acquire(mode_1, sensor_cs_pin)) {}
    spi_write(SENSOR_CONFIGURATION, sensor_cs_pin);
    spi_release();
}
```

Fragility

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

```c
#include "mcu.h"
#include <assert.h>

void sys_init(void) { spi_init(mode_0); }

void sdcard_init(void) {
    assert(mode_0 == spi_mode_get() && "Wrong spi mode for sdcard");
    fat_init();
}

void sensor_init(void) {
    assert(mode_0 == spi_mode_get() && "Wrong spi mode for sensor");
    spi_write(SENSOR_CONFIGURATION, sensor_cs_pin);
}
```

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

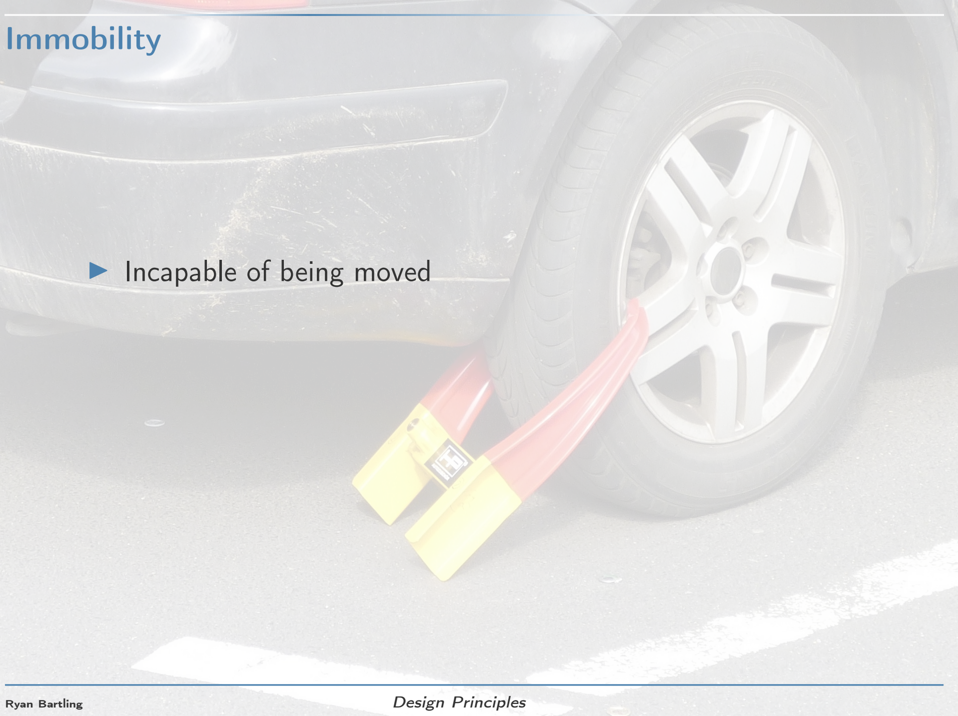
How to avoid it

- 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

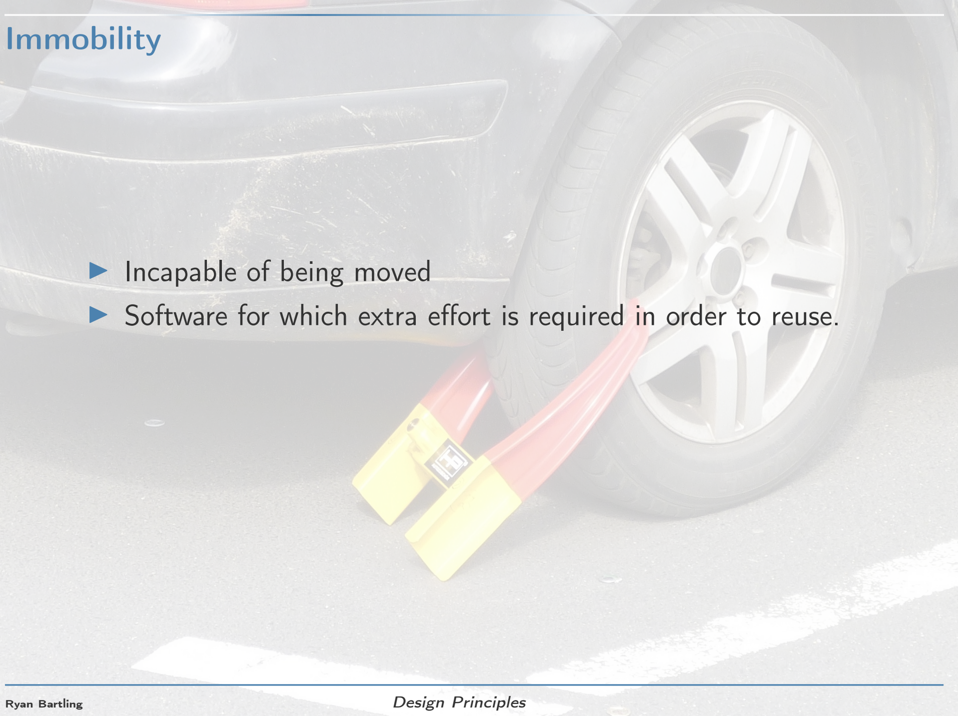
Immobility

Immobility



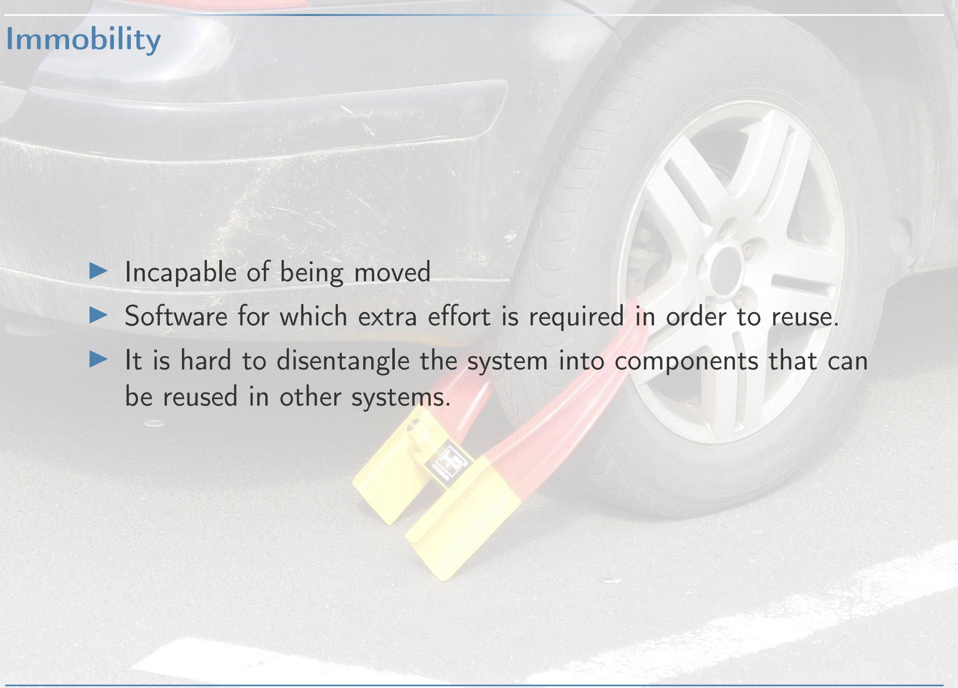
- Incapable of being moved

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

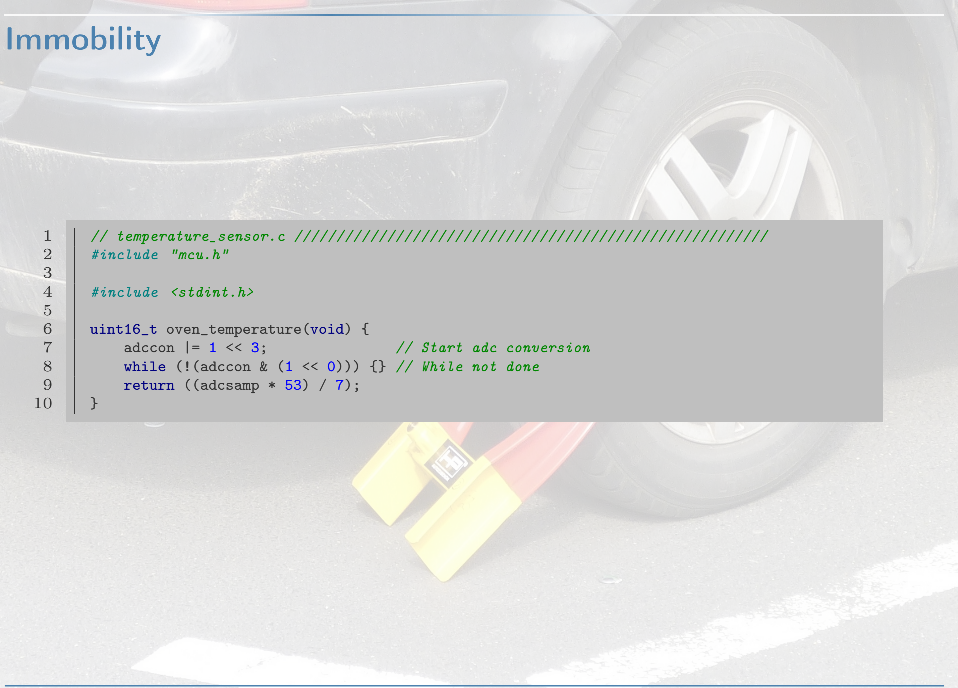
- Direct dependency on things you don't own

Immobility

How it happens

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

Immobility



```c
// temperature_sensor.c ////////////////////////////////////////////////////////
#include "mcu.h"

#include <stdint.h>

uint16_t oven_temperature(void) {
    adccon |= 1 << 3;               // Start adc conversion
    while (!(adccon & (1 << 0))) {} // While not done
    return ((adcsamp * 53) / 7);
}
```

Immobility

```c
#include "mcu.h"
#include <stdint.h>

uint16_t oven_temperature(void) {
    ADC1_start_conversion();
    while (!ADC1_done()) {}
    return ((ADC1_sample_get() * 53) / 7);
}
```

Immobility

```c
#include "mcu.h"
#include <stdint.h>

// TPS = Temperature Sensor

static uint16_t const TPS_get_adc_sample(void) {
    ADC1_start_conversion();
    while (!ADC1_done()) {}
    return ADC1_sample_get();
}

static uint16_t const TPS_adc_counts_to_F(uint16_t const adc_sample) {
    return ((ADC1_sample_get() * 53) / 7);
}

uint16_t TPS_oven_temperature_F(void) {
    uint16_t sample = TPS_get_adc_sample();
    return TPS_adc_counts_to_F(sample);
}
```

Immobility

```c
#include "mcu.h"
#include <stdint.h>

// TPS = Temperature Sensor

static uint16_t const TPS_get_adc_sample(void) {
    ADC1_start_conversion();
    while (!ADC1_done()) {}
    return ADC1_sample_get();
}

static int const TPS_adc_counts_to_F(uint16_t const adc_sample) {
    return ((ADC1_sample_get() * 53) / 7);
}

int TPS_temperature_F(void) {
    uint16_t sample = TPS_get_adc_sample();
    return TPS_adc_counts_to_F(sample);
}
```

Immobility

```c
#include "mcu.h"
#include <stdint.h>

// TPS = Temperature Sensor

static uint16_t const TPS_get_adc_sample(void) {
    ADC1_start_conversion();
    while (!ADC1_done()) {}
    return ADC1_sample_get();
}

static int const TPS_adc_counts_to_F(uint16_t const adc_sample) {
    return ((ADC1_sample_get() * 53) / 7);
}

int TPS_temperature_F(void) {
    uint16_t sample = TPS_get_adc_sample();
    return TPS_adc_counts_to_F(sample);
}

int TPS_temperature_C(void) {
    int temperature_F = TPS_temperature_F();
    return ((temperature_F - 32) * 5) / 9;
}
```

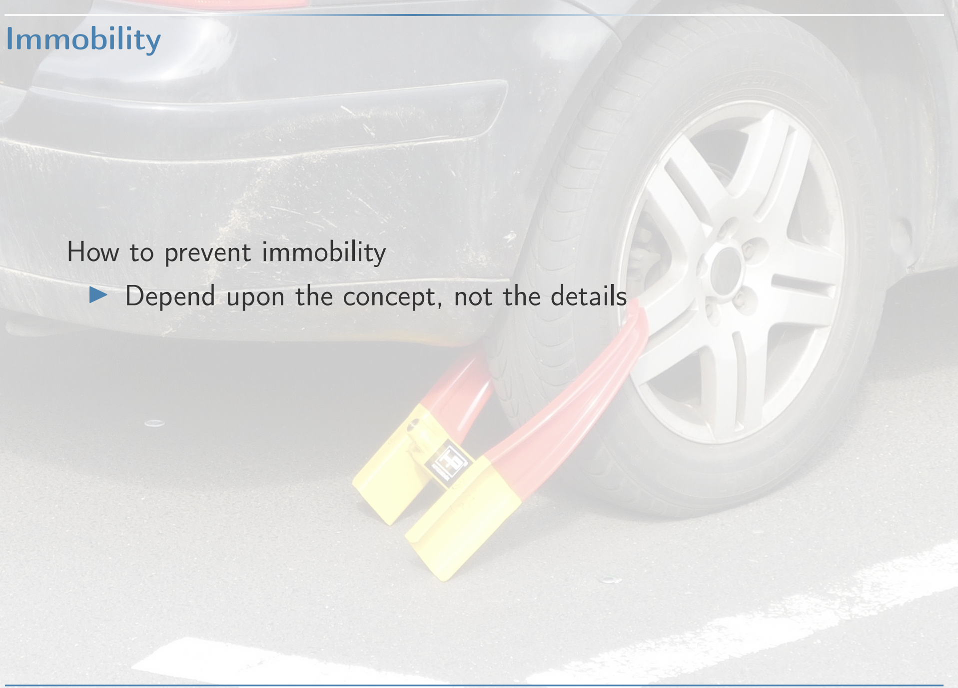
Immobility

```c
#include "mcu.h"
#include <stdint.h>

// TPS = Temperature Sensor

static int const TPS_adc_counts_to_F(int const adc_sample) {
    return ((ADC1_sample_get() * 53) / 7);
}

static int const TPS_F_to_C(int const temperature_F) {
    return ((temperature_F - 32) * 5) / 9;
}

int TPS_temperature_F(int const adc_sample) {
    return TPS_adc_counts_to_F(adc_sample);
}

int TPS_temperature_C(int const adc_sample) {
    int temperature_F = TPS_temperature_F(adc_sample);
    return TPS_F_to_C(temperature_F);
}
```

Immobility

```c
// temperature_sensor.h

typedef int (*counts_to_F_function)(int const /*adc_counts*/);
//...

// temperature_sensor.c
#include <assert.h>
#include <stddef.h>

static counts_to_F_function adc_counts_to_degrees_F = NULL;

void TPS_set_temperature_conversion(counts_to_F_function user_function) {
    adc_counts_to_degrees_F = user_function;
}

int TPS_temperature_F(int const adc_sample) {
    assert((NULL != adc_counts_to_degrees_F) &&
           "You must call TPS_set_temperature_conversion first");
    return adc_counts_to_degrees_F(adc_sample);
}
```

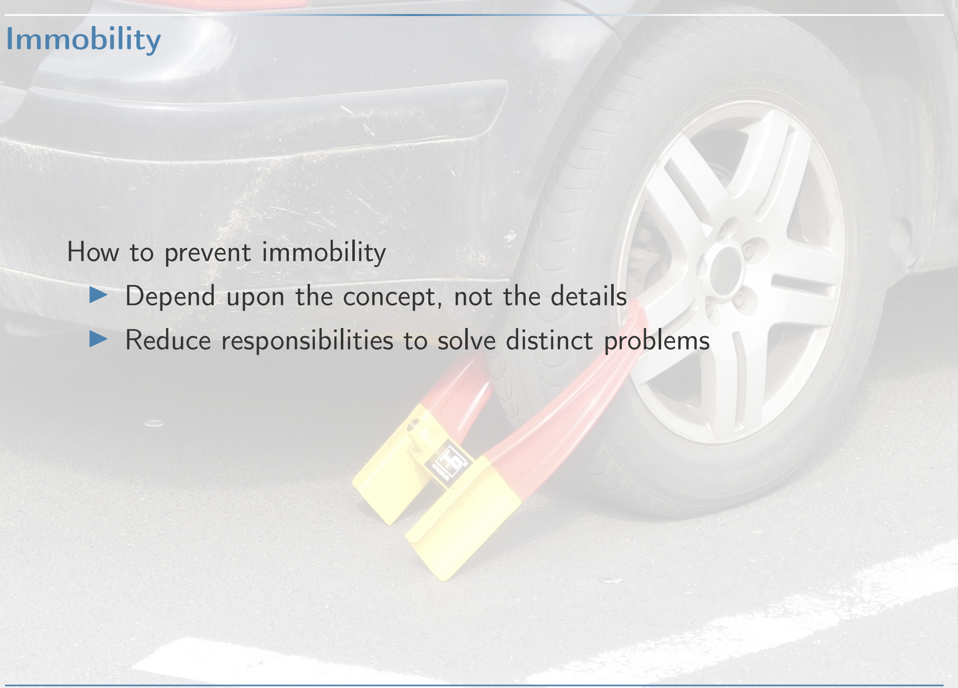
Immobility



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

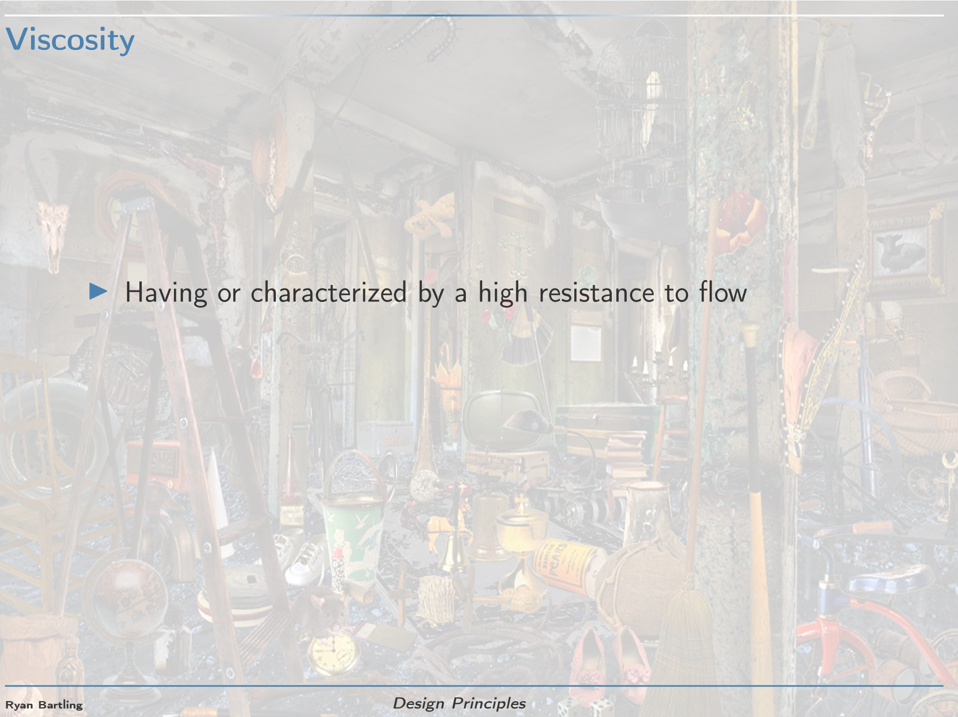
How to prevent immobility

- 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

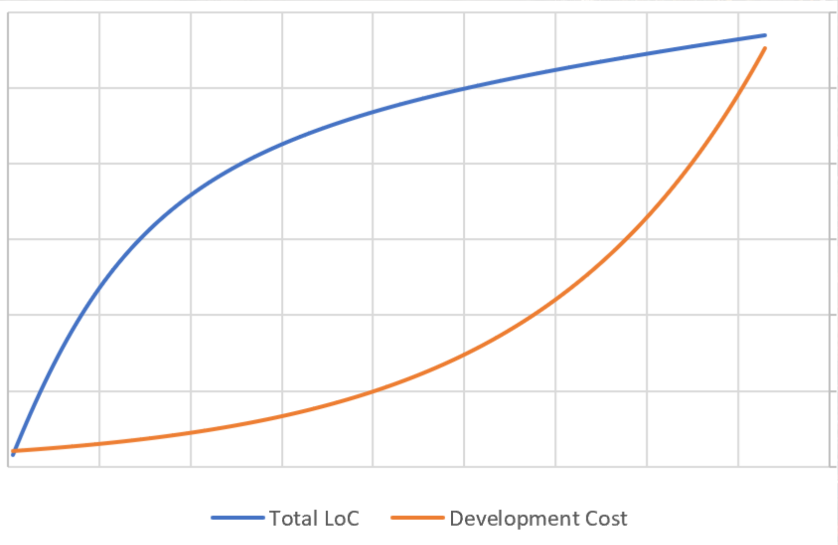
- Having or characterized by a high resistance to flow



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

Principles of Object Oriented Class Design

SOLID Principles

Principles of Object Oriented Class Design

SOLID Principles

- **S***ingle Responsibility Principle* (SRP)

Principles of Object Oriented Class Design

SOLID Principles

- **S***ingle Responsibility Principle* (SRP)
- **O***pen Closed Principle* (OCP)

Principles of Object Oriented Class Design

SOLID Principles

- **S***ingle Responsibility Principle* (SRP)
- **O***pen Closed Principle* (OCP)
- **L***iskov Substitution Principle* (LSP)

Principles of Object Oriented Class Design

SOLID Principles

- **S***ingle Responsibility Principle* (SRP)
- **O***pen Closed Principle* (OCP)
- **L***iskov Substitution Principle* (LSP)
- **I***nterface Segregation Principle* (ISP)

Principles of Object Oriented Class Design

SOLID Principles

- **S***ingle Responsibility Principle* (SRP)
- **O***pen Closed Principle* (OCP)
- **L***iskov Substitution Principle* (LSP)
- **I***nterface Segregation Principle* (ISP)
- **D***ependency 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

```
class modem {
  public:
    void dial();
    void hangup();
    void send();
    void rcv();
};
```

Single Responsibility Principle

```cpp
class modem {
  public:
    void dial();   // Connection management
    void hangup(); // Connection management
    void send();
    void rcv();
};
```

Single Responsibility Principle

```
class modem {
  public:
    void dial();
    void hangup();
    void send(); // Data Management
    void rcv();  // Data Management
};
```

Single Responsibility Principle

```
class modem_connection {
  public:
    void dial();
    void hangup();
};

class modem_data {
  public:
    void send();
    void rcv();
};

class modem_impl {
  private:
    modem_connection connection;
    modem_data data;
};
```

Single Responsibility Principle

Cohesion

Single Responsibility Principle

Cohesion

- Measure of average percentage of member variables used by each function/method

Single Responsibility Principle

Cohesion

- Measure of average percentage of member variables used by each function/method
- Can help reveal multiple responsibilities

Single Responsibility Principle

Cohesion

- Measure of average percentage of member variables used by each function/method
- Can help reveal multiple responsibilities
- How many functions make full use of your class's member variables?

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

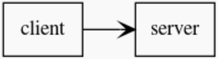
Open Closed Principle

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

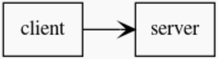
Open Closed Principle

- "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

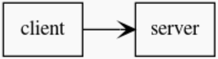


Open Closed Principle



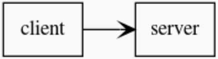
- Client depends on server

Open Closed Principle



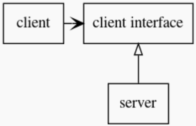
- Client depends on server
- Changing server requires modification of client

Open Closed Principle

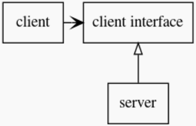


- Client depends on server
- Changing server requires modification of client
- Use of clients with different servers requires duplication of code

Open Closed Principle



Open Closed Principle



- Enables client implementations for multiple servers

Open Closed Principle

```c
// shape.h //////////////////////////////////////////////////////////////////////
typedef struct point_s {
    double x;
    double y;
} point;

enum shape_type { circle, square };
struct shape_s {
    enum shape_type shape_type;
};
// circle.h /////////////////////////////////////////////////////////////////////
struct circle_s {
    enum shape_type shape_type;
    double radius;
    point center;
};
void draw_circle(struct circle_s *);
// square.h /////////////////////////////////////////////////////////////////////
struct square_s {
    enum shape_type shape_type;
    double side;
    point top_left;
};
void draw_square(struct square_s *);
```

Open Closed Principle

```c
// shape.h ///////////////////////////////////////////////////////////////
typedef struct point_s {
    double x;
    double y;
} point;
// Adding a new shape, requires modification of enum
enum shape_type { circle, square };
struct shape_s {
    enum shape_type shape_type;
};
// circle.h //////////////////////////////////////////////////////////////
struct circle_s {
    enum shape_type shape_type;
    double radius;
    point center;
};
void drawCircle(struct circle_s *);
// square.h //////////////////////////////////////////////////////////////
struct square_s {
    enum shape_type shape_type;
    double side;
    point top_left;
};
void drawSquare(struct square_s *);
```

Open Closed Principle

```c
#include "shape.c"

// draw_all_shapes.c ///////////////////////////////////////////////////////////
typedef struct shape_s *shape_pointer_t;

void DrawAllShapes(shape_pointer_t *shapes, int n) {
    for (int i = 0; i < n; i++) {
        struct shape_s *s = shapes[i];

        switch (s->shape_type) {
        case circle: draw_circle((struct circle_s *)s); break;
        case square: draw_square((struct square_s *)s); break;
        }
    }
}
```

Open Closed Principle

```c
#include "shape.c"

// draw_all_shapes.c /////////////////////////////////////////////////////////////
typedef struct shape_s *shape_pointer_t;

void DrawAllShapes(shape_pointer_t *shapes, int n) {
    for (int i = 0; i < n; i++) {
        struct shape_s *s = shapes[i];
        // Adding a new shape would require modification of this switch
        switch (s->shape_type) {
        case circle: draw_circle((struct circle_s *)s); break;
        case square: draw_square((struct square_s *)s); break;
        }
    }
}
```

Open Closed Principle

```c
// shape.h /////////////////////////////////////////////////////////////////////
typedef void (*draw_function_t)(void *);
typedef struct point_s {
    double x;
    double y;
} point;
struct shape_s {
    draw_function_t draw;
};
void draw_shape(void *);
// shape.c /////////////////////////////////////////////////////////////////////
void draw_shape(void *shape_in) {
    struct shape_s *shape = (struct shape_s *)shape_in;
    shape->draw(shape);
}

// circle.h ////////////////////////////////////////////////////////////////////
struct circle_s {
    draw_function_t draw;
    double radius;
    point center;
};
void draw_circle(struct circle_s *);
// square.h ////////////////////////////////////////////////////////////////////
struct square_s {
    draw_function_t draw;
    double side;
    point top_left;
};
void draw_square(struct square_s *);
```

Open Closed Principle

```c
#include "shape_fix.c"

// draw_all_shapes.c ////////////////////////////////////////////////////////////
typedef struct shape_t *shape_pointer_t;

void DrawAllShapes(shape_pointer_t *shapes, int n) {
    for (int i = 0; i < n; i++) {
        struct shape_s *shape = (struct shape_s *)shapes[i];
        draw_shape(shape);
    }
}
```

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

```c
// rectangle.h //////////////////////////////////////////////////////////////////
typedef struct rectangle_s {
    double width;
    double height;
    double (*area)(struct rectangle_s *);
    void (*set_width)(struct rectangle_s *, double);
    void (*set_height)(struct rectangle_s *, double);
} rectangle_t;
rectangle_t *REC_construct(void);
// rectangle.c //////////////////////////////////////////////////////////////////
static void REC_set_width(struct rectangle_s *r, double w) { r->width = w; }
static void REC_set_heigth(struct rectangle_s *r, double h) { r->height = h; }
// square.h /////////////////////////////////////////////////////////////////////
typedef struct rectangle_s square_t;
square_t *SQ_construct(void);
// square.c /////////////////////////////////////////////////////////////////////
#include <stdlib.h>
static void SQ_set_side(square_t *sq, double s) {
    sq->width = s;
    sq->height = s;
}
square_t *SQ_construct(void) {
    square_t *sq = calloc(1, sizeof(square_t));
    sq->set_width = SQ_set_side;
    sq->set_height = SQ_set_side;
    return sq;
}
```

Liskov Substitution Principle

```c
#include "lsp_1.c"
void bar(rectangle_t *r);

void foo(void) {
    square_t *sq = SQ_construct();
    bar(sq);
}

void bar(rectangle_t *r) {
    r->set_height(r, 3);
    r->set_width(r, 4);
}
```

Liskov Substitution Principle

```c
#include "lsp_1.c"

#include <assert.h>

void bar(rectangle_t *r);

void foo(void) {
    square_t *sq = SQ_construct();
    bar(sq);
}

void bar(rectangle_t *r) {
    r->set_height(r, 3);
    r->set_width(r, 4);
    assert(r->area(r) == 12);
}
```

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 <= new height value

Liskov Substitution Principle

Contract for set_height():

- Pre-conditions:
  - Valid object pointer
  - 0 <= new height value
- Post-conditions:

Liskov Substitution Principle

Contract for set_height():

- Pre-conditions:
  - Valid object pointer
  - 0 <= new height value
- Post-conditions:
  - Height matches the new value

Liskov Substitution Principle

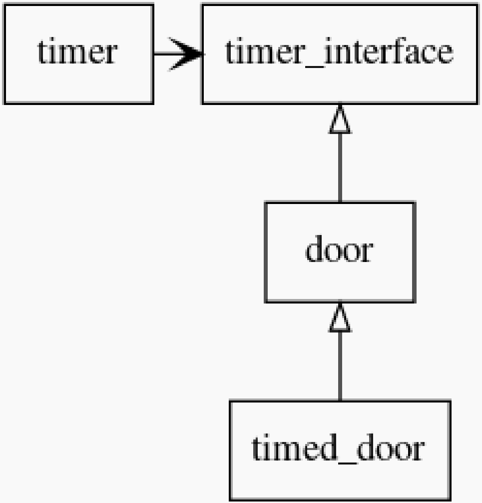
Contract for set_height():

- Pre-conditions:
  - Valid object pointer
  - 0 <= new height value
- Post-conditions:
  - Height matches the new value
  - Width is unchanged

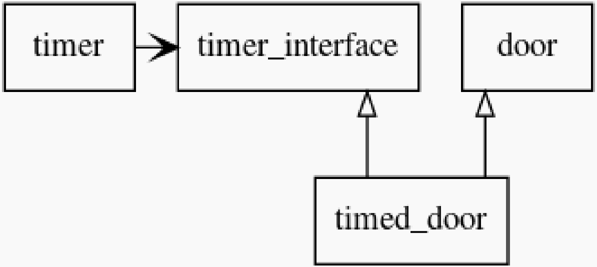
Interface Segregation Principle

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

Interface Segregation Principle



Interface Segregation Principle



Dependency Inversion Principle

Depend upon abstractions.

Dependency Inversion Principle

Depend upon abstractions. Do not depend upon details.

Dependency Inversion Principle

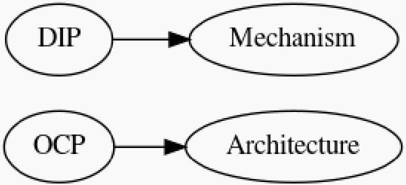
Dependency Inversion Principle

- High level moduls should not depend on low-level modules. Both should depend on abstractions.

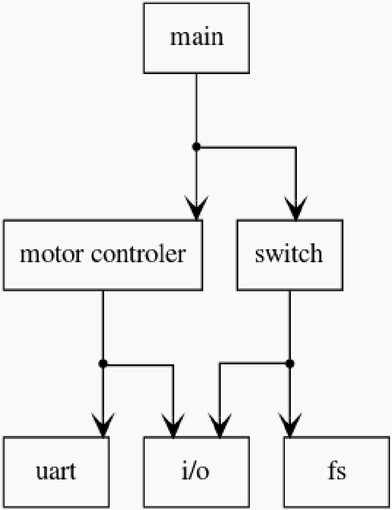
Dependency Inversion Principle

- High level moduls should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

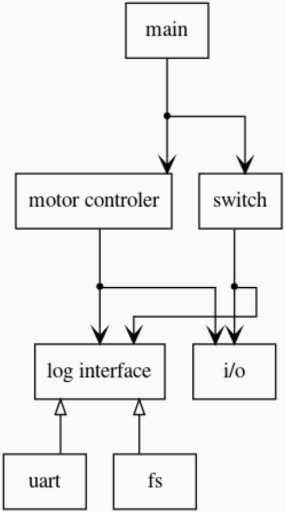
Dependency Inversion Principle



Dependency Inversion Principle



Dependency Inversion Principle



Package Design

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

Principles of Package Architecture

Conclusion

Introduction

Symptoms of Rotting Design

Principles of Object Oriented Class Design

Package Design

Architecture Design

Conclusion

References

- Agile Software Development, Robert C. Martin
- 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
- https://softwareengineering.stackexchange.com/questions/357127/clear-examples-for-code-smells

Questions