Design Principles and Design Patterns

D. Ryan Bartling

June 6, 2018

Outline

Introduction

Symptoms of Rotting Design

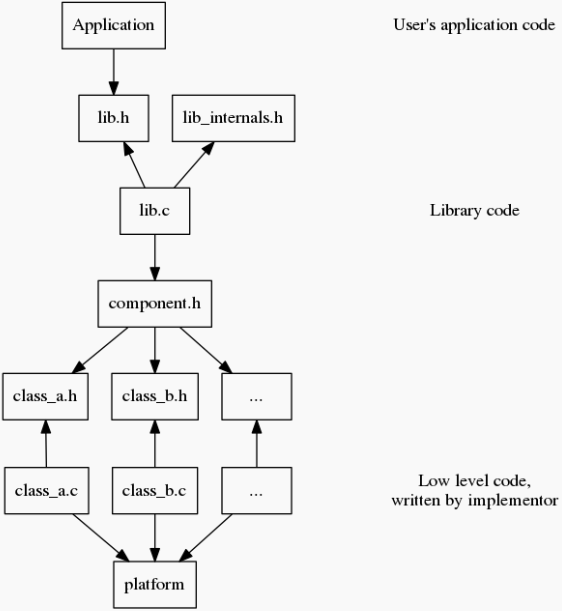
Principles of Object Oriented Class Design

Package Design

Architecture Design

Conclusion

Architecture and Dependencies



Symptoms of Rotting Design

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

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

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

Rigidity

```c
#include <stdint.h>

#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;

#if sizeof(rpo_raw_adc_t) < (RAW_ADC_BITS / 2 + 1)
#error "rpo_raw_adc_t is too small to store ADC results"
#endif

#if sizeof(rpo_correlated_int_t) < (RAW_ADC_BITS / 2 + 1)
#error "rpo_correlated_int_t is too small to store correlated adc results"
#endif
```

Rigidity

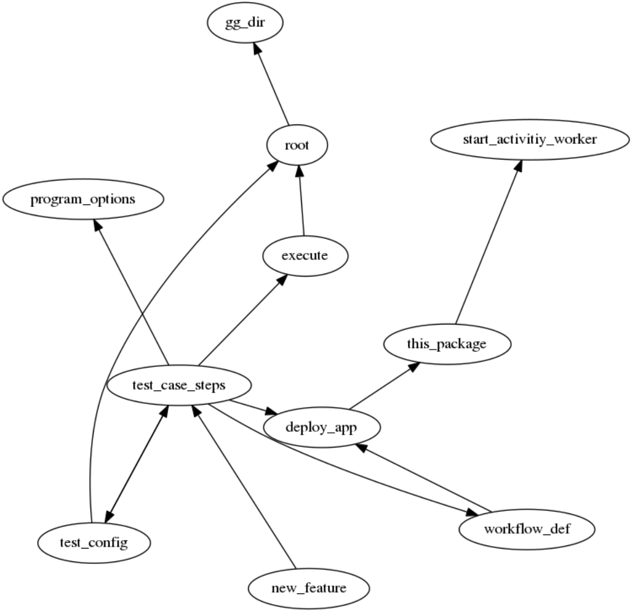
```c
#include <stdint.h>

#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;

#if sizeof(rpo_raw_adc_t) < (RAW_ADC_BITS / 2 + 1)
#error "rpo_raw_adc_t is too small to store ADC results"
#endif

#if sizeof(rpo_correlated_int_t) < (RAW_ADC_BITS / 2 + 1)
#error "rpo_correlated_int_t is too small to store correlated adc results"
#endif
```

Rigidity

Refactor to reduce rigidity

```
#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


gg_dir
root
start_activitiy_worker
program_options
execute
this_package
test_case_steps
deploy_app
workflow_def
test_config
new_feature

Rigidity

Notes:
Customer wants a feature: be able to test a new feature
Create the code to test the new feature
This requires new test case steps
Which in turn requires a new workflow to execute the new test case steps
Also new execution command
Also new test configurations
Also new command line arguments
Etc.

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

- 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

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

Fragility

```
void sdcard_init(void) {
    spi_init(mode_0, card_cs_pin);
    fat_init();
}

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

Fragility

Changing the sensor to use mode 1...

```c
void sdcard_init(void) {
    spi_init(mode_0, card_cs_pin);
    fat_init();
}

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

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

Fragility

How to avoid it

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

Immobility

- 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
- Too many responsibilities

Immobility

How it happens

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

Viscosity

- Having or characterized by a high resistance to flow
- Software for which extra effort is required in order to reuse.

Viscosity

Code that takes effort to maintain correctly

- Viscous Design
  - When changing, preserving the design is difficult
- Viscous Environment
  - Long builds
  - Slow Tests

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

- Cohesion
- Reason to change
- Axis of change

Single Responsibility Principle

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

Single Responsibility Principle

```
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

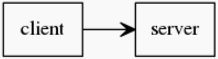
Caution:

- 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

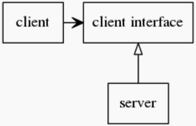
- "Open for Extension"
    - Behavior of the module can be extended through extension
- "Closed for Modification"
    - Extending the behavior requires no change in source code or binary executables.

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



- Enables client implementations for multiple servers

Open Closed Principle

```c
// shape.h ////////////////////////////////////////////////////////////////////
enum shape_type_t { circle, square };
struct shape_s {
    shape_type_t shape_type;
}

// circle.h ///////////////////////////////////////////////////////////////////
#include "shape.h"
struct circle_s {
    shape_type_t shape_type;
    double       radius;
    point        center;
}

void drawCircle(struct circle_s *);

// square.h ///////////////////////////////////////////////////////////////////
#include "shape.h"
struct square_s {
    shape_type_t shape_type;
    double       side;
    point        top_left;
}

void drawSquare(struct square_s *);
```

Open Closed Principle

```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 < nl i++) {
        struct shape_s *s = shapes[i];
        switch (shape->shape_type) {
        case circle:
            drawCircle((struce circle_s *)shape);
            break;
        case square:
            drawSquare((struce square_s *)shape);
            break;
        }
    }
}
```

Open Closed Principle

```c
// shape.h /////////////////////////////////////////////////////////////////////
enum shape_type_t { circle, square };
void (*DrawFunction)(void *);
struct shape_s {
    DrawFunction draw;
}
void DrawShape(void *);

// shape.c /////////////////////////////////////////////////////////////////////
void DrawShape(void * shape_in){
    shape = (struct shape_s *) shape_in;
    shape.draw(shape_in);
}

// circle.h ////////////////////////////////////////////////////////////////////
struct circle_s {
    DrawFunction draw;
    double       radius;
    point        center;
}

void drawCircle(struct circle_s *);

// square.h ////////////////////////////////////////////////////////////////////
struct square_s {
    DrawFunction draw;
    double       side;
    point        top_left;
}

void drawSquare(struct square_s *);
```

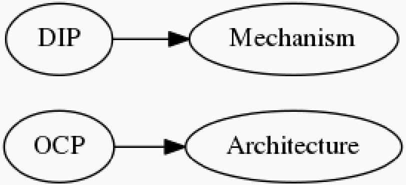
Open Closed Principle

```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 < nl i++) {
        struct shape_s *shape = shapes[i];
        DrawShape(shape);
    }
}
```

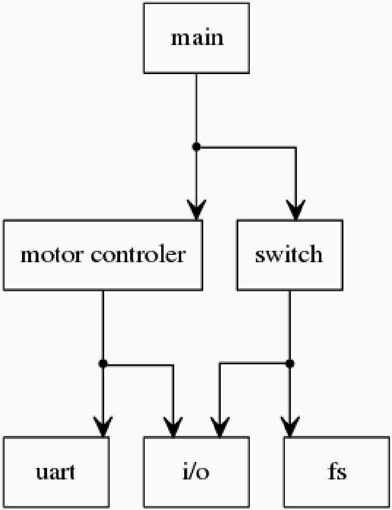
Dependency Inversion Principle

Depend upon abstractions. Do not depend upon concretions.

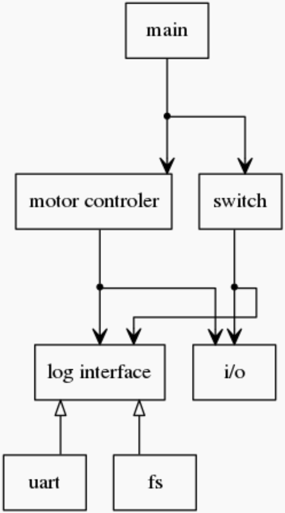
Dependency Inversion Principle



Dependency Inversion Principle



Dependency Inversion Principle



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

Principles of Package Architecture

Principles of Package Architecture

References

- 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