



00 BOOTCAMP

Bowen Wang





软件设计

软件设计



思考什么是好的“软件设计”？

为什么做软件？

满足客户需求



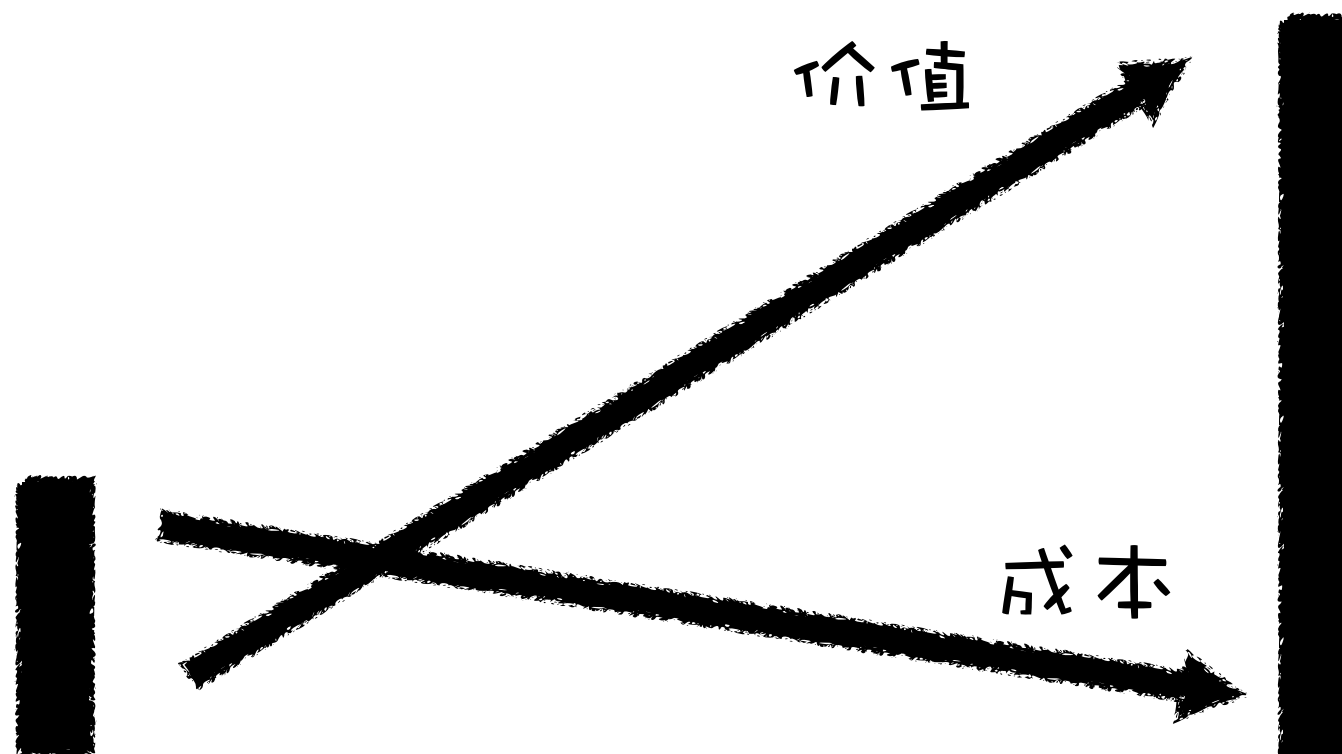
商业逻辑

revenue

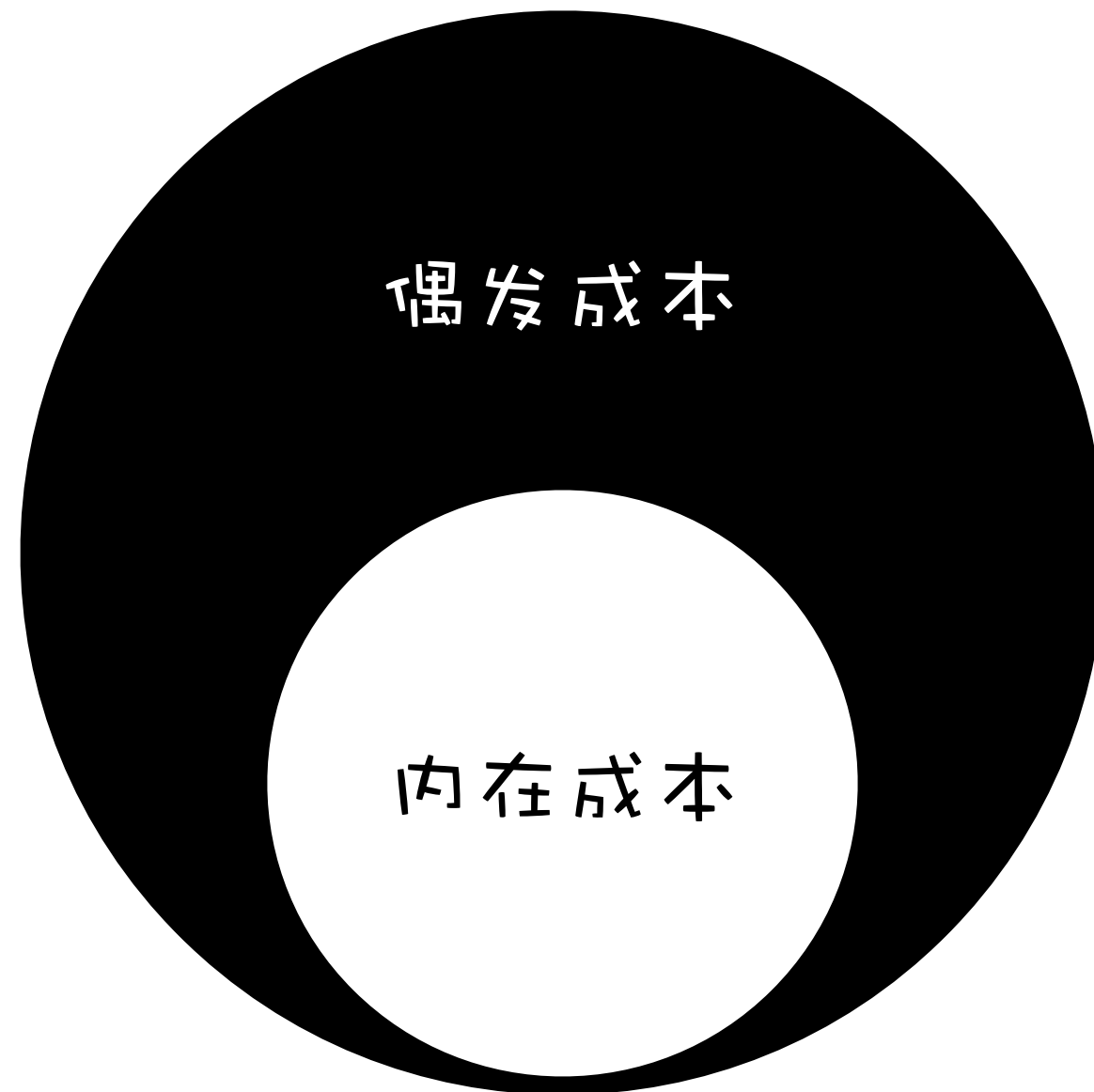
— cost

profit

所以...



成本？



例子: **HELLO WORLD**

```
std::cout << "Hello, World" << std::endl;
```


偶发成本

```
for(int i=0; i < ::strlen("Hello, World\n"); i++)  
{  
    4  ::putc("Hello, World\n"[i]);  
}
```

偶发成本

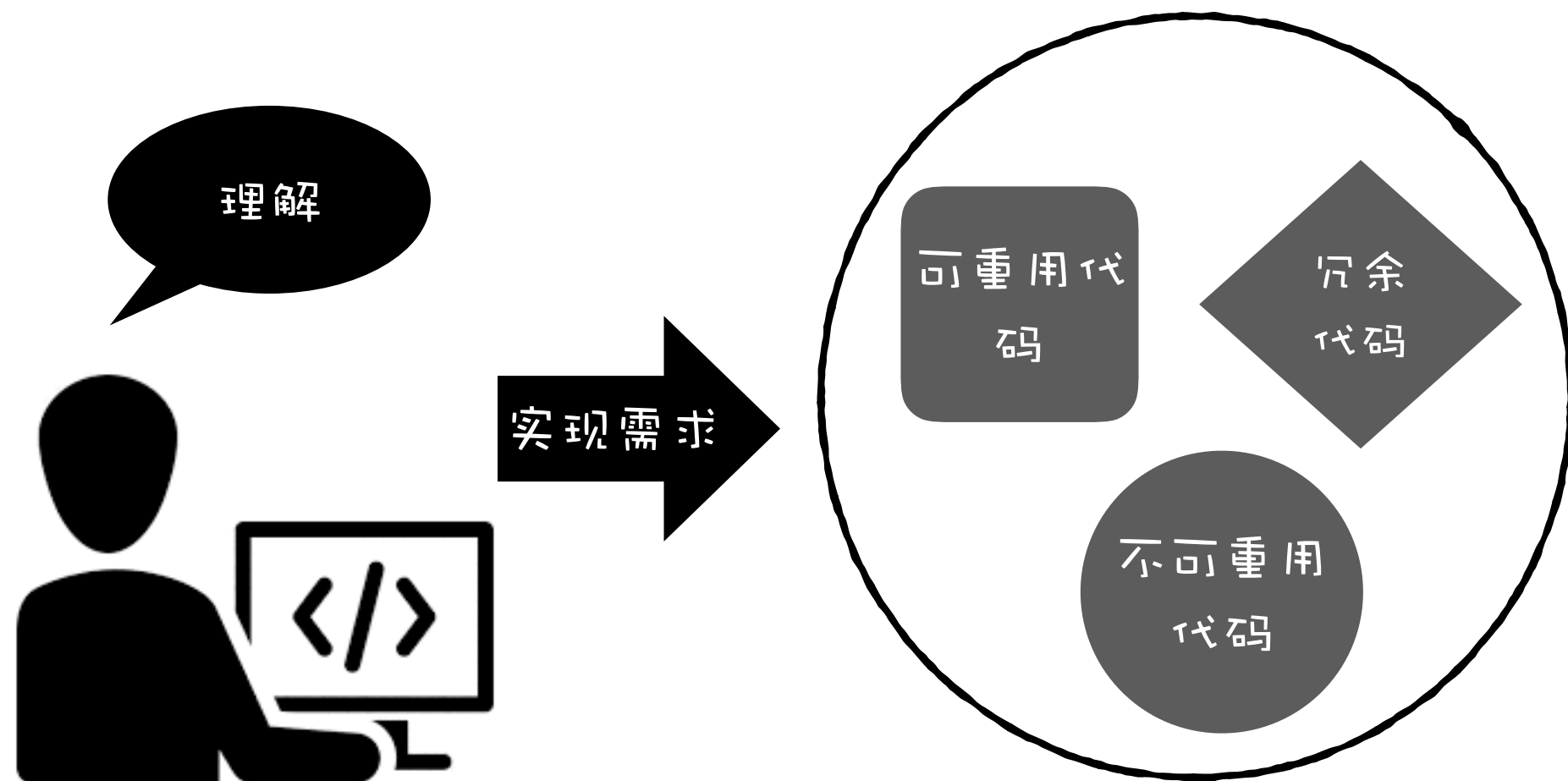
```
void alien_say(char* p)
{
    while(::putc(*(p += *(p+1) - *p)));
}

int main()
{
    return alien_say("BETH0! Altec oh liryom(a
                      loadjudas!) dowl."), 0;
}
```

偶发成本

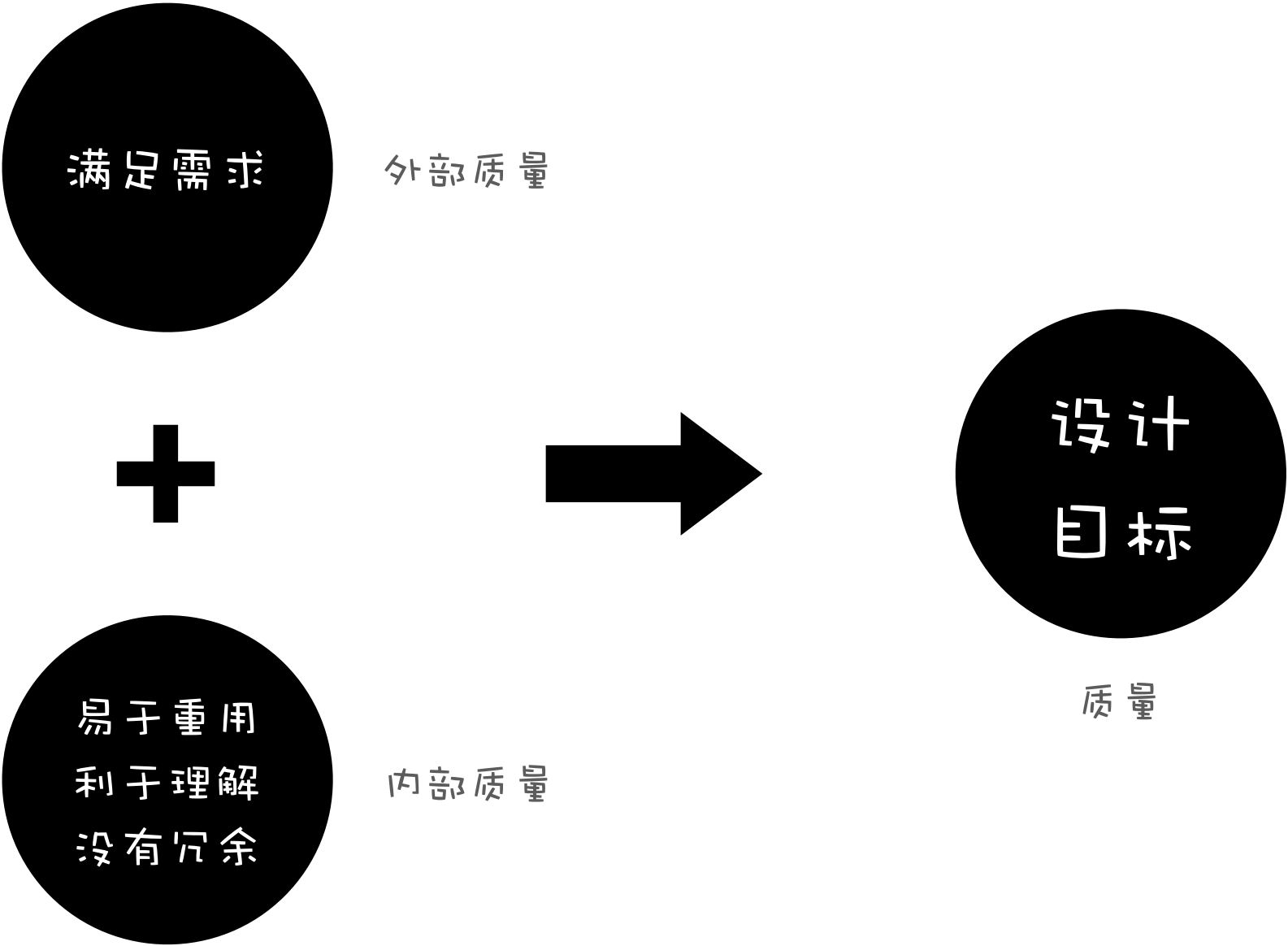
```
Insert new slide  
unsigned int i = 0;  
std::cout << "Hello, World" << std::endl;
```

软件开发中的“偶发成本”

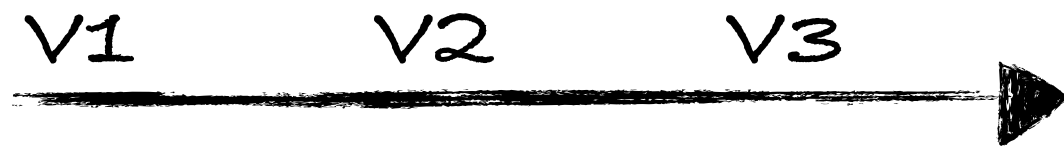


软件设计目标

.....

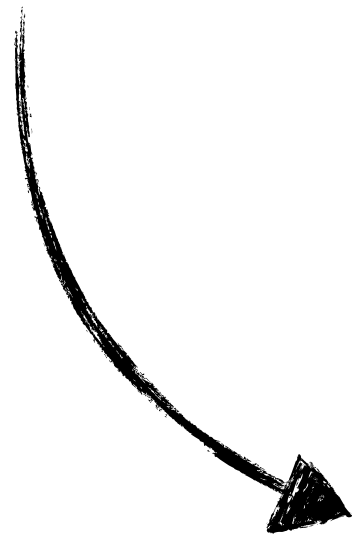


软件设计的价值



满足需求的情况下,

让软件在长期更低成本应对变化!



"Design is there to enable you to keep **changing** the software easily in the long term" -- Kent Beck.

简单设计 - KENT BECK

1. 通过所有测试 (Passes its tests)
2. 尽可能消除重复 (Minimizes duplication)
3. 尽可能清晰的表达 (Maximizes clarity)
4. 尽可能减少代码元素的数量 (Has fewer elements)

以上四个原则的重要程度依次降低!

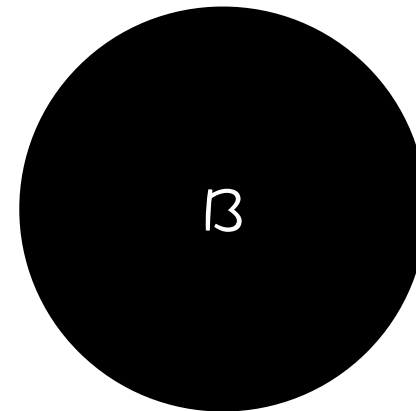
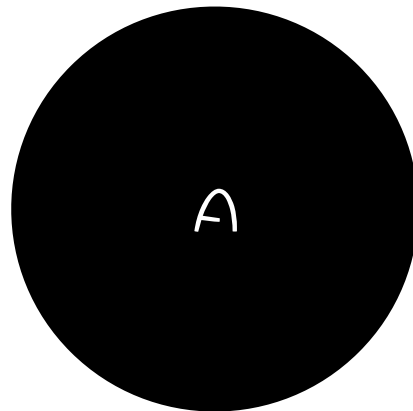




消除重复

— 提高软件可重用性

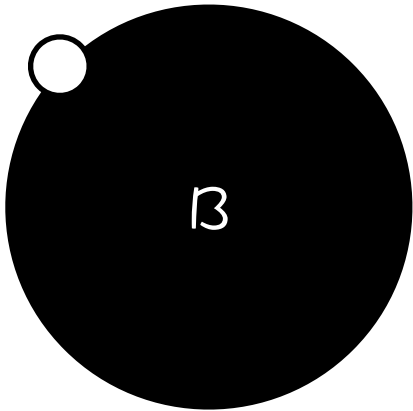
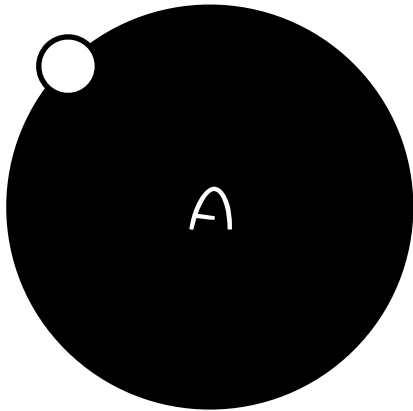
重复：完全重复



完全重复：例子

```
////////////////////////////////////  
const unsigned int max_num_of_allowed_connections = 1000;  
  
////////////////////////////////////  
const unsigned int max_num_of_allowed_connections = 1000;
```

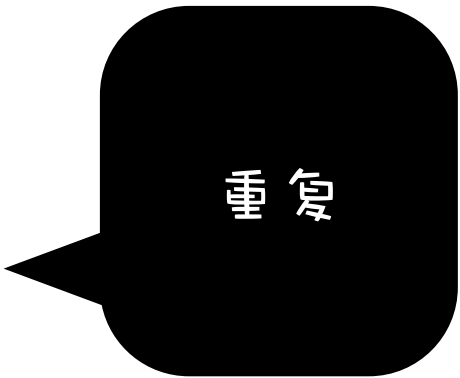
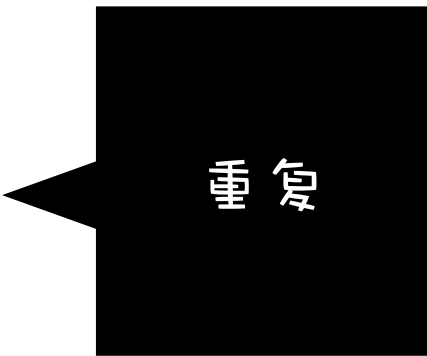
重复： 参数型重复



参数型重复：例子

```
////////////////////////////////////  
strcpy(buf, packet->src_address);  
buf += strlen(packet->src_address) + 1;  
  
////////////////////////////////////  
strcpy(buf, packet->dest_address);  
buf += strlen(packet->dest_address) + 1;
```

重复：功能型重复



功能型重复： 例子1

.....

```
//////////////////////////////////////
const unsigned int max_num_of_allowed_connections = 1000;
//////////////////////////////////////
#define MAX_ALLOWED_CONNECTIONS ((unsigned int)1000)
//////////////////////////////////////
unsigned int get_max_num_of_allowed_connections()
{
    return 1000;
}
```

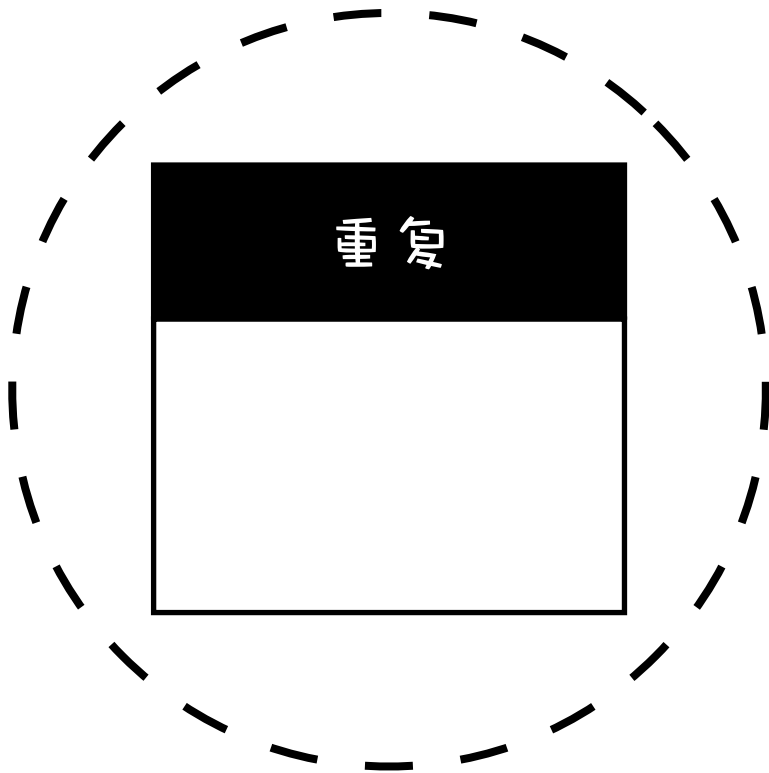
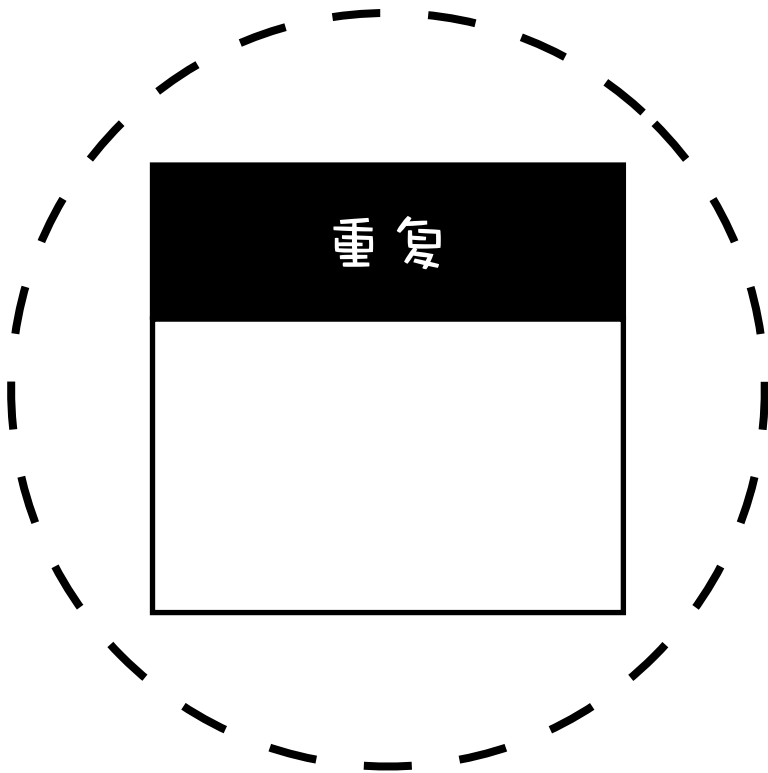
功能型重复： 例子2

.....

```
////////////////////////////////////  
void say_hello_world()  
{  
    std::cout << "Hello, World" << std::endl;  
}  
  
////////////////////////////////////  
void say_hello_world()  
{  
    for(int i=0; i<::strlen("Hello, World\n"); i++)  
    {  
        ::putc("Hello, World\n"[i]);  
    }  
}
```

功能型

重复：结构型重复

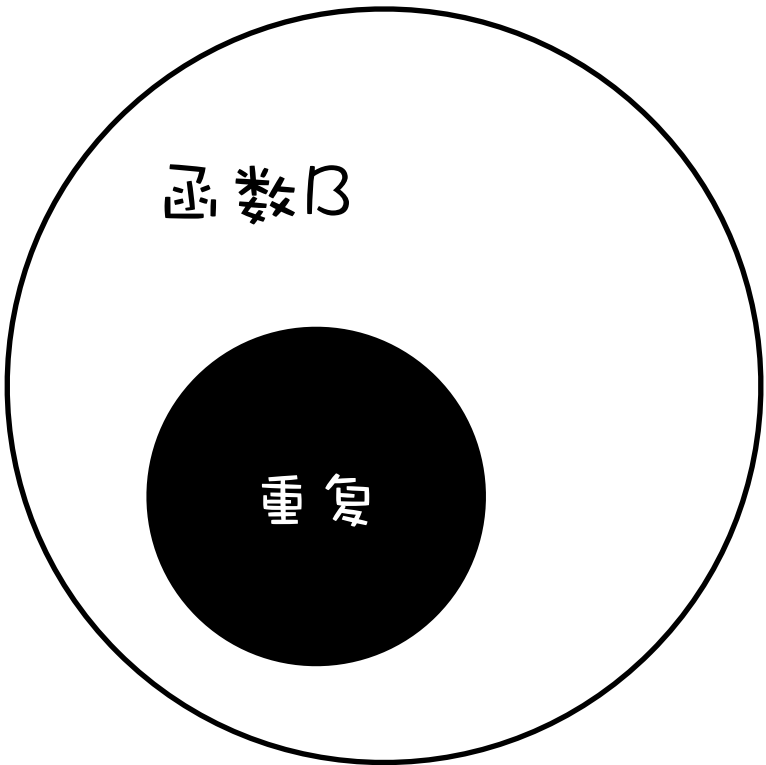
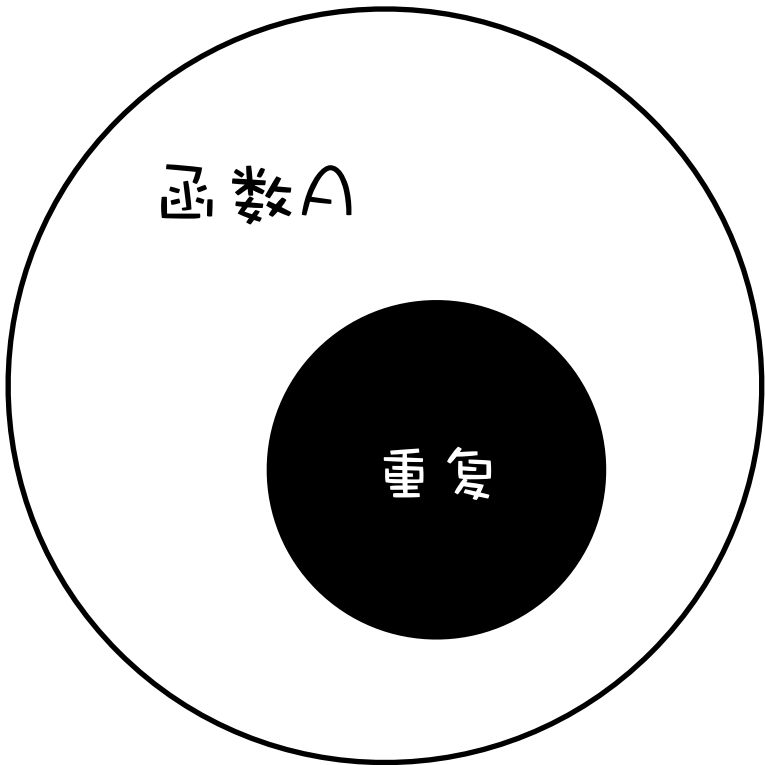


结构型重复： 例子

```
class Foo
{
public:
    void action1();
    void action2();
    void action3();
private:
    int data1;
    int data2;
};
```

```
class Bar
{
public:
    void action1();
    void action2();
    void action4();
private:
    int data1;
    int data3;
};
```

重复：调用型重复



调用型重复：例子

.....

```
//////////////////////////////////////
void foo()
{
    while(num-- > 0) if(num == packet->pin_num) break;

    strcpy(buf, packet->dest_address);
    buf += strlen(packet->dest_address) + 1;

    if(get_sys_cfg() == SEND) send(buf);
}

//////////////////////////////////////
void bar()
{
    if(isAllowed()) return;

    strcpy(buf, packet->dest_address);
    buf += strlen(packet->dest_address) + 1;
}
```

完全重

strcpy(bu
buf += st

重复：回调型重复



回调型重复：例子

.....

```
////////////////////////////////////  
void foo()  
{  
    while(num-- > 0) if(num == packet->pin_num) break;  
    save_to_database();  
  
    if(get_sys_cfg() == SEND) send(buf);  
}  
////////////////////////////////////  
void bar()  
{  
    while(num-- > 0) if(num == packet->pin_num) break;  
    strcpy(buf, packet->dest_address);  
    buf += strlen(packet->dest_address) + 1;  
    if(get_sys_cfg() == SEND) send(buf);  
}  
////////////////////////////////////
```

回调型重复： 例子

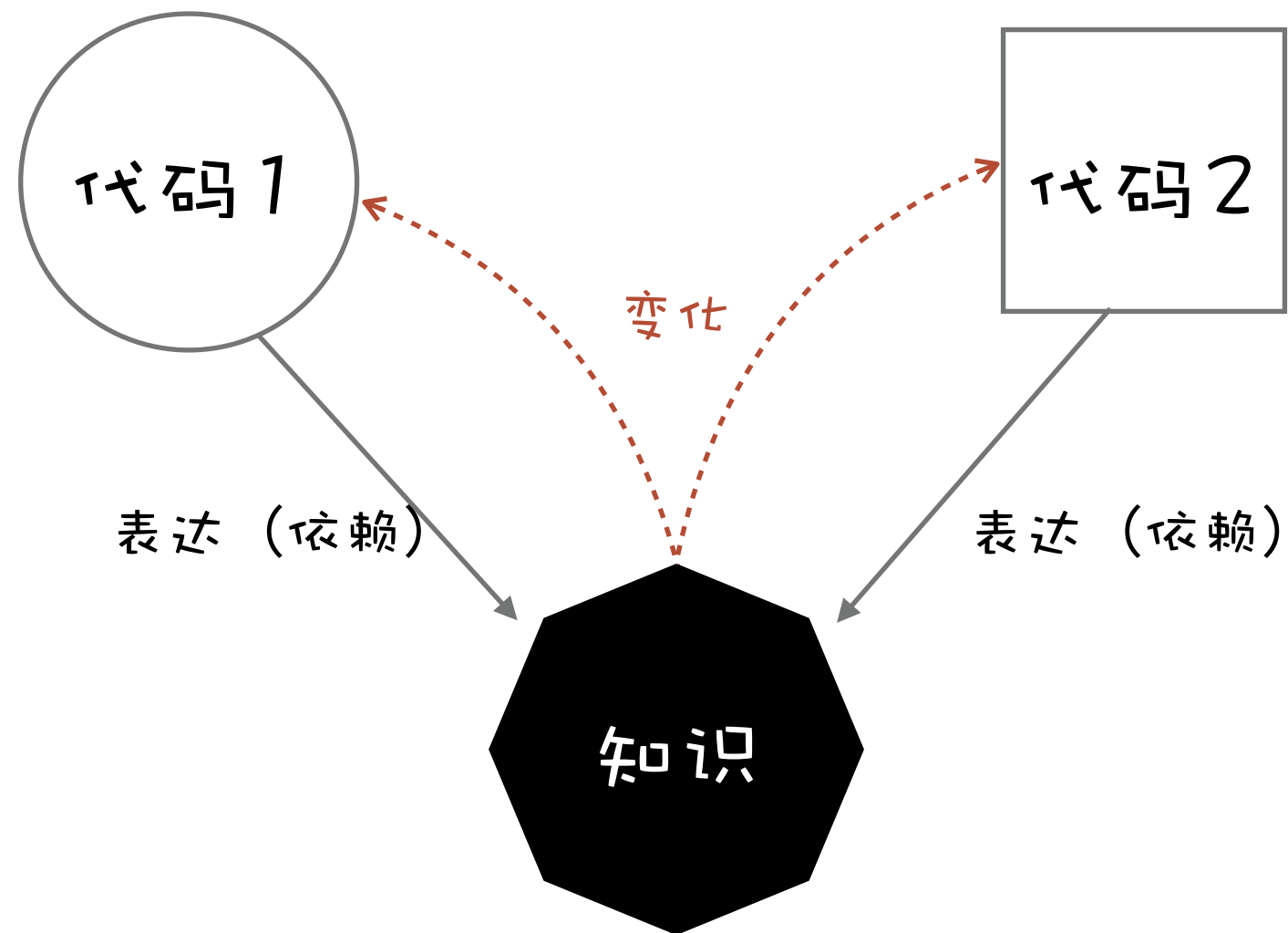
.....

```
////////////////////////////////////  
void foo()  
{  
    while(num-- > 0) if(num == packet->pin_num) break;  
    save_to_database();  
  
    if(get_sys_cfg() == SEND) send(buf);  
}  
////////////////////////////////////  
void bar()  
{  
    while(num-- > 0) if(num == packet->pin_num) break;  
    strcpy(buf, packet->dest_address);  
    buf += strlen(packet->dest_address) + 1;  
    if(get_sys_cfg() == SEND) send(buf);  
}  
////////////////////////////////////
```

算不算重复？

```
////////////////////////////////////  
#define PI ((double)3.1415)  
  
////////////////////////////////////  
const double MY_PI = PI;  
  
////////////////////////////////////  
inline double getPI() {  
    return MY_PI;  
}
```

重复的本质 - 知识的重复



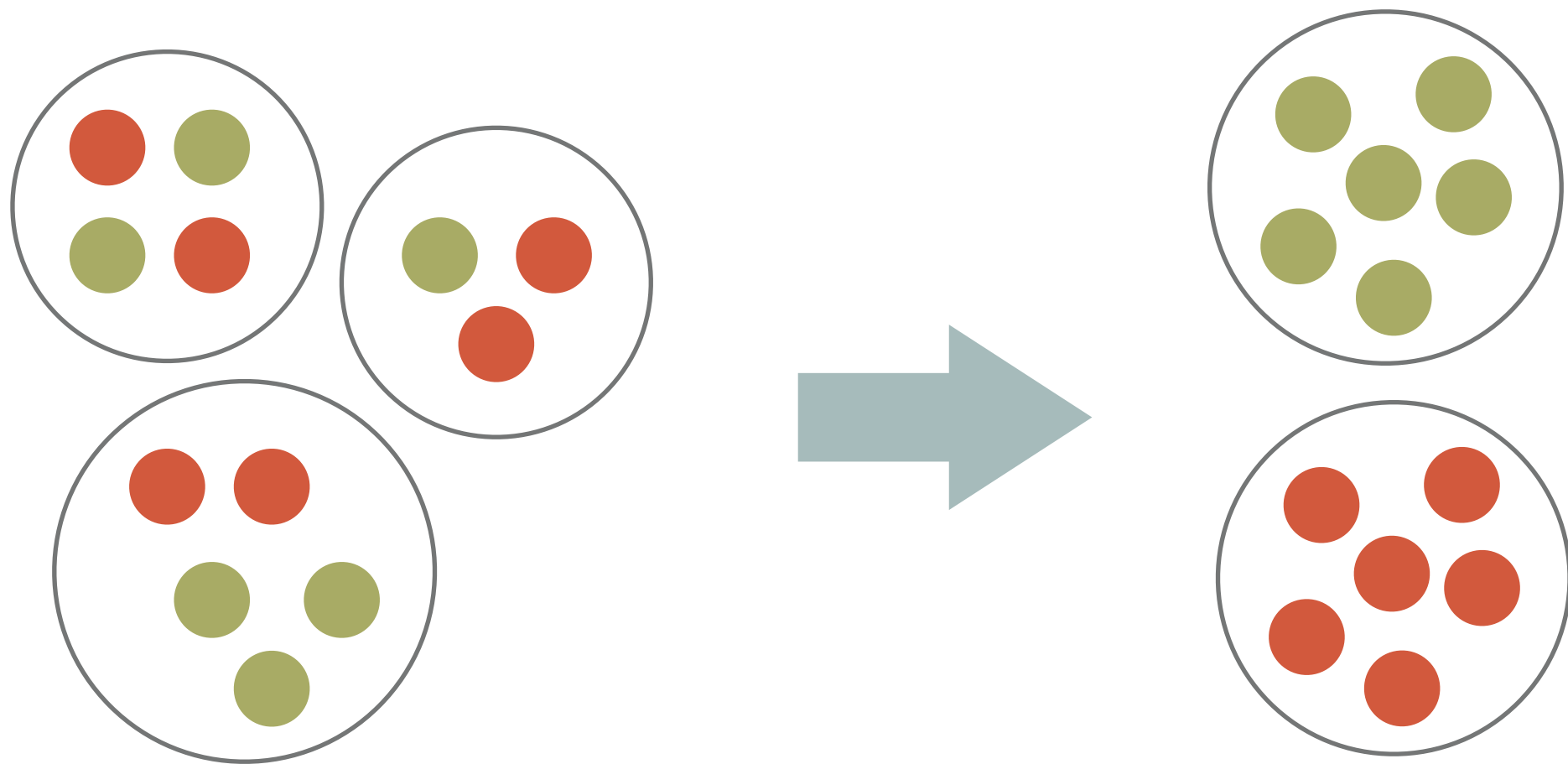
重复的危害



重复破坏了软件的“高内聚低耦合”

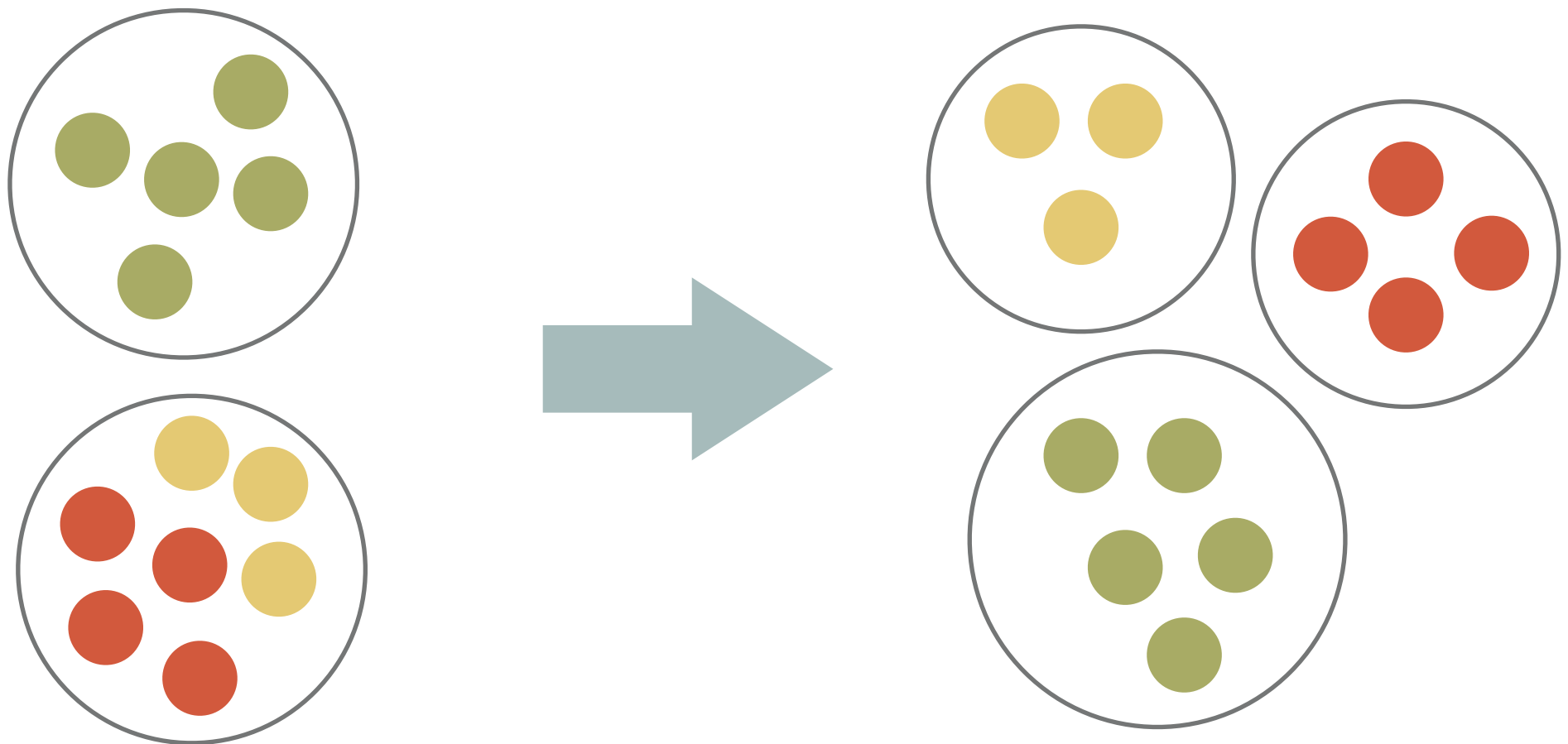
重复降低了软件的“可重用性”

高内聚



紧密关联的事物应该被放在一起！

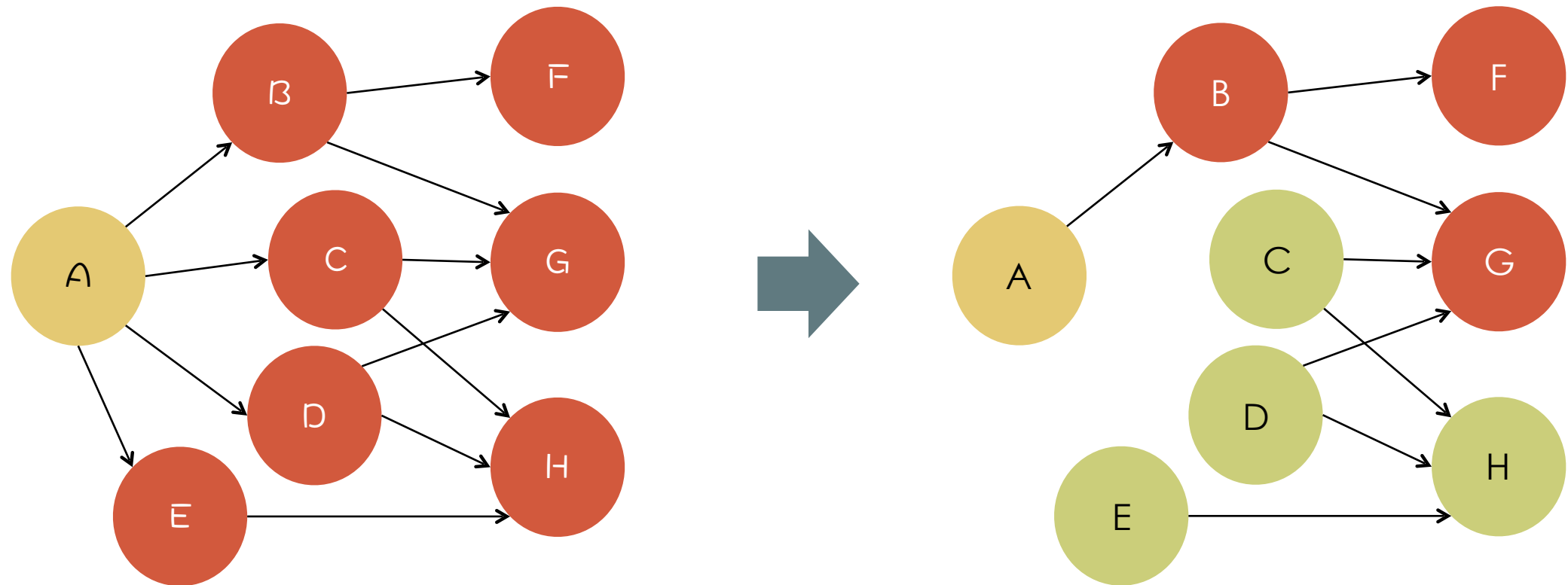
高内聚



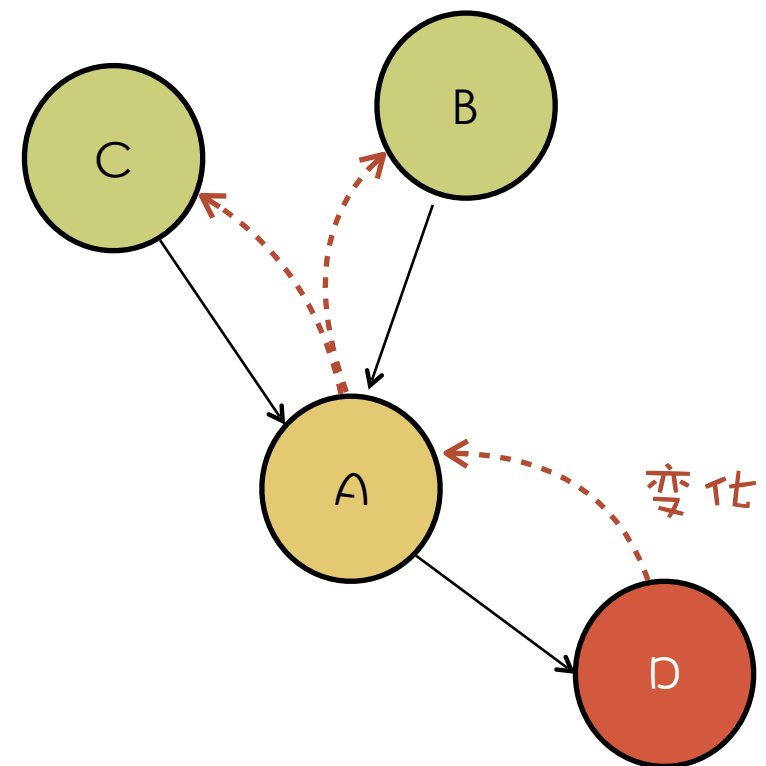
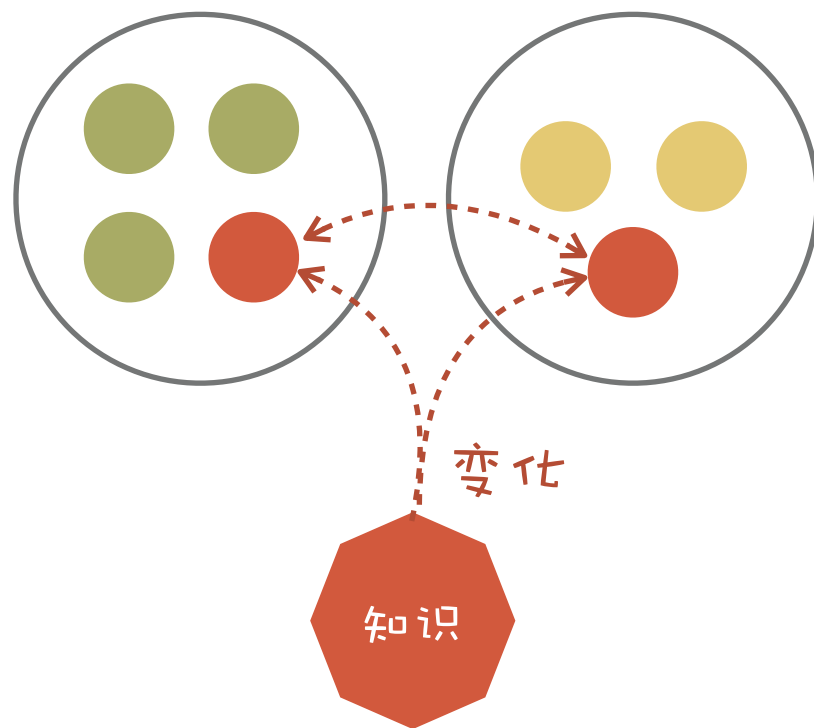
只有紧密关联的事物才应该被放在一起！

DO ONE THING, GO IT WELL !

低内聚

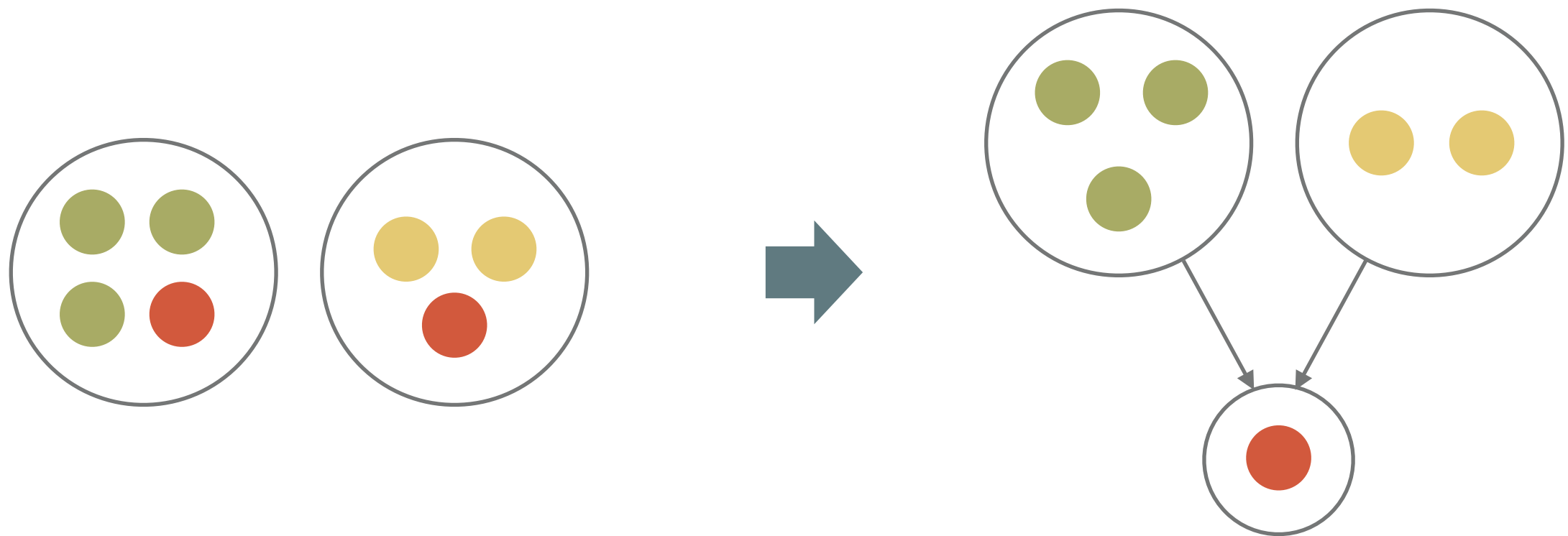


重复的危害



重复降低了内聚性、提高了耦合性
让软件出现“散弹式修改”的坏味道！

通过消除重复提高可重用性

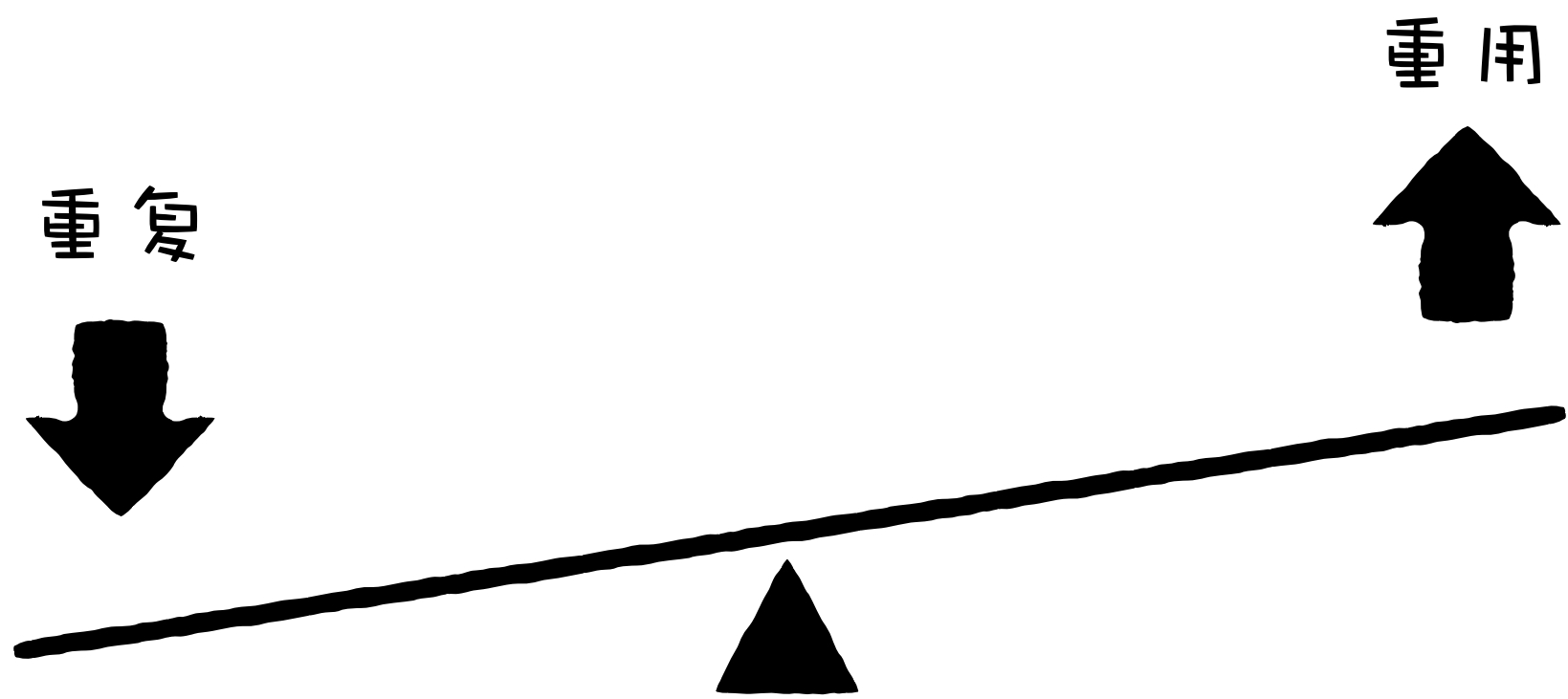


可重用性

可重用： 如果一块代码C，对于满足某个需求R，无须修改，无须copy & paste，就可以被使用，那么我们就称C对于R可重用。

可重用性： 一个软件单元被重用的可能性。一个软件单元被可重用的问题域越广，那么它的可重用性越高。

重复与重用



原则：消除重复

DRY: (Don't Repeat Yourself)

对于任何一项知识，系统中应该只存在一个明确而权威的表示！



权威

如何预防重复

WHY?

- 重构是有成本的!
- 重复的模式被隐藏其中，不易被识别!

例子：需求



- 将所有学生按身高从低到高排序

实现

```
struct Student
{
    char    name[MAX_NAME_LEN];
    unsigned int height;
};

void sort_students_by_height( Student students[]
                              , size_t num_of_students)
{
    for(size_t y=0; y < num_of_students-1; y++)
    {
        for(size_t x=1; x < num_of_students - y; x++)
        {
            if(students[x].height > students[x-1].height)
            {
                SWAP(students[x], students[x-1]);
            }
        }
    }
}
```

仅仅从需求实现的角度来看，

可能的变化方向

- 排序算法
- 排序对象
- 排序规则



重构后的代码

.....

```
template <typename T, typename GreaterThan>
void bulb_sort( T objects[]
               , size_t num_of_objects
               , const GreaterThan& greater_than )
{
    for(size_t y=0; y < num_of_objects - 1; y++)
    {
        for(size_t x=1; x < num_of_objects - y; x++)
        {
            if(greater_than(objects[x], objects[x-1]))
            {
                SWAP(objects[x], objects[x-1]);
            }
        }
    }
}
```

```
bool student_taller_than(const Student& lhs, const Student& rhs)
{
    return lhs.height > rhs.height;
}
```

事实上，大多数类型（包括整数，浮点
“>”。为了避免让这些类型各自重复的

原则：分离不同的变化方向

-

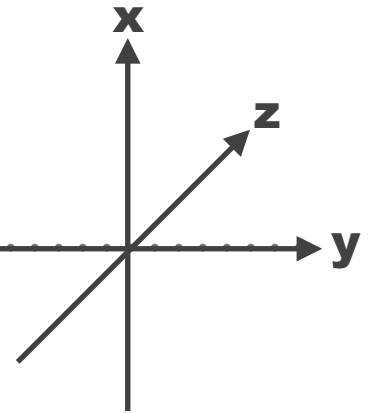
原则：减少依赖范围

-

原则：向稳定的方向依赖

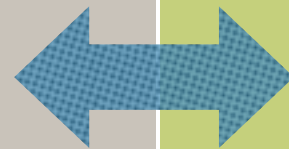
-

正交设计原则



消除重复

1个被动策略



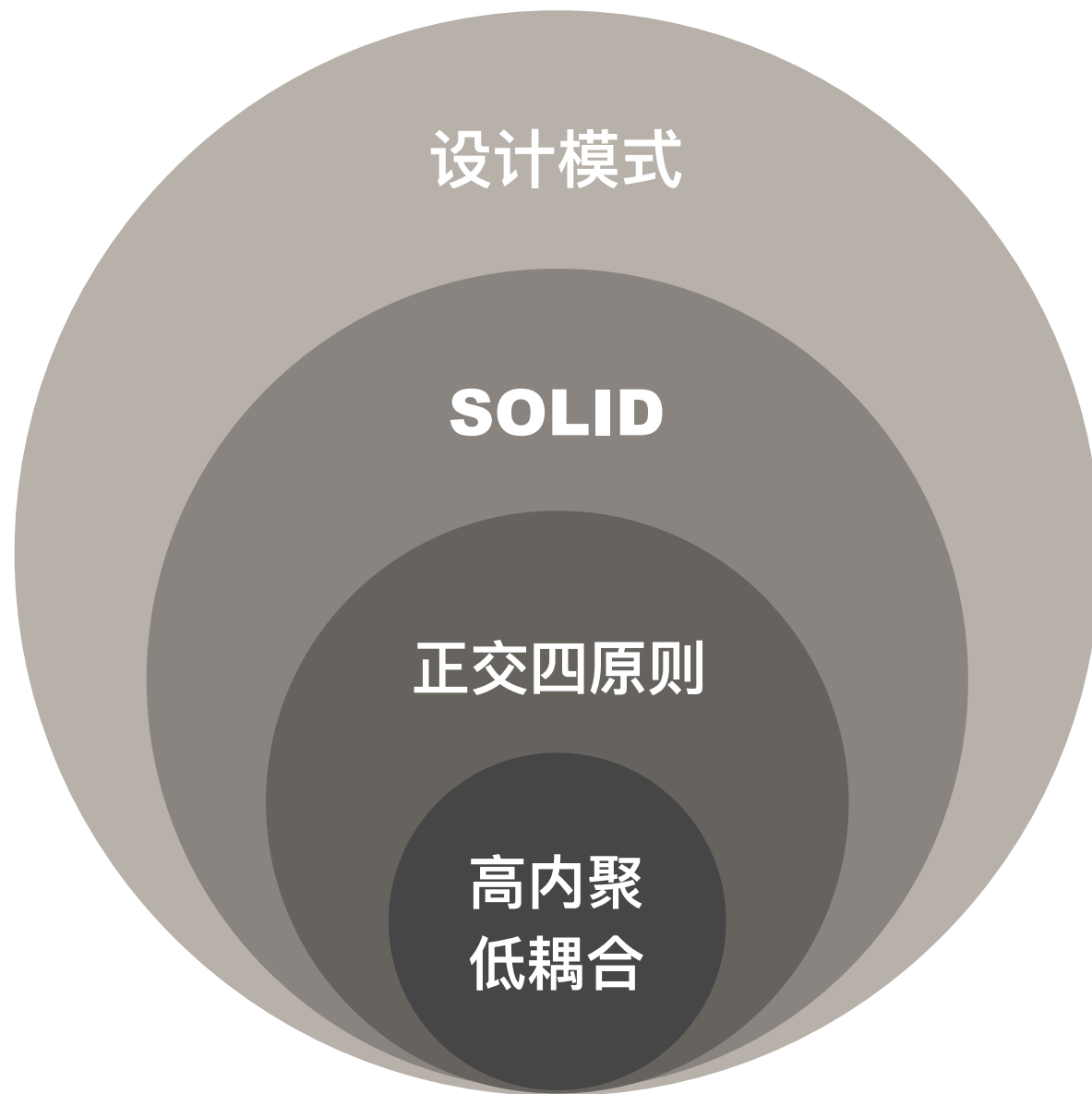
分离变化方向

缩小依赖范围

向稳定的方向依赖

3个主动策略

设计原则



*“Design is there to enable you
to keep **changing** the software
easily in the long term”*

—— Kent Beck



提高代码表达力



减少冗余



Questions?

ThoughtWorks

THANKS

王博

e.wangbo@gmail.com