Robot Cleaner

Programming in Modern C++11

刘光聪

This page is intentionally left blank.

目录

Do the simplest thing that could possibly work.

- Kent Beck

1 Background

1.1 问题提出

扫地机器人 (robot cleaner)，又称自动打扫机、智能吸尘器等。能凭借一定的人工智能，自动在房间内完成地板清理工作。

Bosch 公司是领先的扫地机器人制造商。Bosch 的工程师研发了一款机器人，它接受远端遥感指令，并完成一些简单的动作。

为了方便控制机器人的导航，工程师使用三元组 (x, y, d) 来表示机器人的位置信息；其中 (x,y) 表示机器人的坐标位置，d 表示机器人的方向 (包括 East, South, West, North 四个方向)。

假设机器人初始位置为 (0, 0, N)，表示机器人处在原点坐标、并朝向北。

1.2 技能要求

在实现需求的前提下，增加如下一些技能要求。

1. 每个迭代 45 分钟
2. 结对编程
3. 不允许使用鼠标
4. 不允许使用 IDE
5. 坚持使用 TDD 开发
6. 持续重构代码
7. 坚持良好的代码提交习惯
8. 教练点评的坏味道，下一迭代必须纠正
9. 参与互相点评的环节
10. 互相重构不同 pair 的代码

1.3 参考实现

本文所使用的代码，读者可以从 Gitlab 上自由下载、修改和传播。

软件设计是一门艺术，因作者的经验、能力都非常有限，此实现仅作供参考。如果大家发现了问题，或者有更加简单、漂亮的设计，请及时反馈给我，我将感激不尽。

Git 地址：git clone https://gitlab.com/horance/robot-cleaner-cpp.git

Email 地址：horance@outlook.com

This page is intentionally left blank.

I'm not a great programmer; I'm just a good programmer with great habits.

- Kent Beck

2 TurnTo

2.1 需求

1. 当 Robot 收到 LEFT 指令后，Robot 向左转 90 度；
2. 当 Robot 收到 RIGHT 指令后，Robot 向右转 90 度；

例如：

1. Robot 原来在 (0, 0, N)，执行 LEFT 指令后，新的位置在 (0, 0, W)；
2. Robot 原来在 (0, 0, N)，执行 RIGHT 指令后，新的位置在 (0, 0, E)；

2.2 破冰之旅

2.2.1 第一个测试用例

从第一个测试用例开始，Robot 初始位置应该在 (0, 0, N)。

示例代码 2-1 test/robot-cleaner/TestRobotCleaner.h

```
#include "testngpp/testngpp.hpp"

FIXTURE(RobotCleaner)
{
    RobotCleaner robot;

    TEST("at the beginning, the robot should be in at the initial position")
    {
        ASSERT_EQ(Position(0, 0, NORTH), robot.getPosition());
    }
};
```

2.2.2 通过编译

为了试图通过编译，先创建一个枚举类型 Orientation，用于表示方向。

示例代码 2-2 include/robot-cleaner/Orientation.h

```
#ifndef H9811B75A_15B3_4DF0_91B7_483C42F74473
#define H9811B75A_15B3_4DF0_91B7_483C42F74473

enum Orientation { EAST, SOUTH, WEST, NORTH };

#endif
```

使用随机的生成的宏保护符，使得重命名类和文件名变得极为便捷。

另外为 Orientation 新建一个头文件，即使它仅仅只有一行代码；而没有将所有代码实体都塞在一个头文件内，制造一个巨型头文件。如此实现给设计带来了诸多的好处：

1. 领域内的的 Orientation 概念，提供了更好的可复用性；
2. 职责更单一，从而可以得到更好的物理依赖；
3. 避免了上帝头文件，多人协作并行使用 GIT, SVN 等代码管理工具时，冲突的概率极大地被减低。

然后再创建一个 Position 类，用于描述 Robot 所在的地理位置信息。

示例代码 2-3 include/robot-cleaner/Position.h

```
#ifndef H722C49C6_D285_4885_88D1_97A11D669EE1
#define H722C49C6_D285_4885_88D1_97A11D669EE1

#include "robot-cleaner/Orientation.h"

struct Position
{
    Position(int x, int y, Orientation orientation);

    bool operator==(const Position&) const;
};

#endif
```

是的，现在最紧急的目标是让代码编译通过，没有考虑 Position 是怎么实现的，从而没有考虑 Position 应该包含哪些成员变量。

此外，Position 重载了操作符 ==，使得它更加具有 Value Object 的特性；另外 Position.h 直接包含了 Orientation.h，让 Position.h 具有自满足性，从而使用户无需为包含关系而烦恼。

最后创建一个 RobotCleaner 类，用于描述 Robot 主体。就目前测试用例而言，默认构造函数即可，附加提供一个 getPosition 成员函数即可。

示例代码 2-4 include/robot-cleaner/RobotCleaner.h

```
#ifndef H632C2FEA_66F0_4B68_B047_121B6CE1482E
#define H632C2FEA_66F0_4B68_B047_121B6CE1482E

struct Position;

struct RobotCleaner
{
    Position getPosition() const;
};

#endif
```

因为 getPosition 是一个查询函数，所以它被声明为 const 成员函数；此外，Position 仅作为返回值，前置声明即可，无需包含它的头文件，从而降低了物理依赖性。

2.2.3 通过链接

为了尽快看到 make 的反馈，实现 RobotCleaner::getPosition，及其 Position 类时，直接做了快速的伪实现，毕竟现在的目标只是让链接通过而已。

示例代码 2-5 src/robot-cleaner/RobotCleaner.cpp

```
#include "robot-cleaner/RobotCleaner.h"
#include "robot-cleaner/Position.h"

Position RobotCleaner::getPosition() const
{
    return Position(0, 0, NORTH);
}
```

示例代码 2-6 src/robot-cleaner/Position.cpp

```
#include "robot-cleaner/Position.h"

Position::Position(int x, int y, Orientation orientation)
{}

bool Position::operator==(const Position& rhs) const
{
    return false;
}
```

2.2.4 通过测试

此刻链接已经通过，但测试运行失败。为了尽快让测试通过，也只需将 Position 的比较逻辑快速实现。

示例代码 2-7 include/robot-cleaner/Position.h

```
#ifndef H722C49C6_D285_4885_88D1_97A11D669EE1
#define H722C49C6_D285_4885_88D1_97A11D669EE1

#include "robot-cleaner/Orientation.h"

struct Position
{
    Position(int x, int y, Orientation orientation);

    bool operator==(const Position&) const;
};

#endif
```

示例代码 2-8 include/robot-cleaner/Position.h

```
#ifndef H722C49C6_D285_4885_88D1_97A11D669EE1
#define H722C49C6_D285_4885_88D1_97A11D669EE1

#include "robot-cleaner/Orientation.h"

struct Position
{
    Position(int x, int y, Orientation orientation);

    bool operator==(const Position&) const;

private:
    int x;
    int y;
    Orientation orientation;
};

#endif
```

实现比较逻辑也较为简单，创建 Position.cpp 直接实现。

示例代码 2-9 src/robot-cleaner/Position.cpp

```
#include "robot-cleaner/Position.h"

Position::Position(int x, int y, Orientation orientation)
  : x(x), y(y), orientation(orientation)
{}

bool Position::operator==(const Position& rhs) const
{
    return x == rhs.x
        && y == rhs.y
        && orientation = rhs.orientation;
}
```

2.2.5 重构

显式实现

首先消除 RobotCleaner 的伪实现，创建一个 Position 的成员变量，并将 getPosition 的返回值类型修正为 const 引用。

示例代码 2-10 include/robot-cleaner/RobotCleaner.h

```
#ifndef H632C2FEA_66F0_4B68_B047_121B6CE1482E
#define H632C2FEA_66F0_4B68_B047_121B6CE1482E

struct Position;

struct RobotCleaner
{
    Position getPosition() const;
};

#endif
```

示例代码 2-11 include/robot-cleaner/RobotCleaner.h

```
#ifndef H632C2FEA_66F0_4B68_B047_121B6CE1482E
#define H632C2FEA_66F0_4B68_B047_121B6CE1482E

#include "robot-cleaner/Position.h"

struct RobotCleaner
{
    RobotCleaner();

    const Position& getPosition() const;

private:
    Position position;
};

#endif
```

最后并将初始化的代码移植到构造函数中去。

示例代码 2-12 src/robot-cleaner/RobotCleaner.cpp

```
#include "robot-cleaner/RobotCleaner.h"

Position RobotCleaner::getPosition() const
{
    return Position(0, 0, NORTH);
}
```

示例代码 2-13 src/robot-cleaner/RobotCleaner.cpp

```
#include "robot-cleaner/RobotCleaner.h"

RobotCleaner::RobotCleaner()
  : position(0, 0, NORTH)
{}

const Position& RobotCleaner::getPosition() const
{
    return position;
}
```

2.3 left 指令

2.3.1 测试用例

示例代码 2-14 test/robot-cleaner/TestRobotCleaner.h

```
TEST("the robot should be in (0, 0, W) when I send left instruction")
{
    robot.turnLeft();
    ASSERT_EQ(Position(0, 0, WEST), robot.getPosition());
}
```

2.3.2 通过测试

为了快速通过测试用例，为 RobotCleaner 新增一个 turnLeft 成员函数。

示例代码 2-15 include/robot-cleaner/RobotCleaner.h

```
#ifndef H632C2FEA_66F0_4B68_B047_121B6CE1482E
#define H632C2FEA_66F0_4B68_B047_121B6CE1482E

#include "robot-cleaner/Position.h"

struct RobotCleaner
{
    RobotCleaner();

    const Position& getPosition() const;

private:
    Position position;
};

#endif
```

示例代码 2-16 include/robot-cleaner/RobotCleaner.h

```
#ifndef H632C2FEA_66F0_4B68_B047_121B6CE1482E
#define H632C2FEA_66F0_4B68_B047_121B6CE1482E

#include "robot-cleaner/Position.h"

struct RobotCleaner
{
    RobotCleaner();

    void turnLeft();

    const Position& getPosition() const;

private:
    Position position;
};

#endif
```

此处，对于实现 turnLeft 还没有更多的想法，先伪实现，确保测试尽快通过。

示例代码 2-17 src/robot-cleaner/RobotCleaner.cpp

```
void RobotCleaner::turnLeft()
{
    position = Position(0, 0, WEST);
}
```

2.3.3 重构

显式实现

先将 turnLeft 委托给 Position。

示例代码 2-18 src/robot-cleaner/RobotCleaner.cpp

```
void RobotCleaner::turnLeft()
{
    position.turnLeft();
}
```

在 make 之前，先将 Position::turnLeft() 伪实现。

示例代码 2-19 src/robot-cleaner/Position.cpp

```
void Position::turnLeft()
{
    orientation = WEST;
}
```

确保测试通过后，继续消除 Position 的伪实现，为此再写一个测试，让问题变得更加明确。

示例代码 2-20 test/robot-cleaner/TestRobotCleaner.h

```
TEST("the robot should be in (0, 0, S) when I send left instruction with 2 times")
{
    robot.turnLeft();
    robot.turnLeft();
    ASSERT_EQ(Position(0, 0, SOUTH), robot.getPosition());
}
```

为了快速让测试通过，先使用 switch-case 快速实现。

示例代码 2-21 src/robot-cleaner/Position.cpp

```
void Position::turnLeft()
{
    orientation = WEST;
}
```

示例代码 2-22 src/robot-cleaner/Position.cpp

```
namespace
{
    Orientation toLeft(Orientation orientation)
    {
        switch (orientation)
        {
        case WEST:  return SOUTH;
        case NORTH: return WEST;
        default: break;
        }
        return orientation;
    }
}

void Position::turnLeft()
{
    orientation = toLeft(orientation);
}
```

重复这个微小的循环，覆盖所有的逻辑，确保自己永不犯错，每次通过测试都是修改非常少的代码而完成的，即使犯错也是非常容易定位和修正。

示例代码 2-23 test/robot-cleaner/TestRobotCleaner.h

```
TEST("the robot should be in (0, 0, E) when I send left instruction with 3 times")
{
    robot.turnLeft();
    robot.turnLeft();
    robot.turnLeft();

    ASSERT_EQ(Position(0, 0, EAST), robot.getPosition());
}

TEST("the robot should be in (0, 0, N) when I send left instruction with 4 times")
{
    robot.turnLeft();
    robot.turnLeft();
    robot.turnLeft();
    robot.turnLeft();

    ASSERT_EQ(Position(0, 0, NORTH), robot.getPosition());
}
```

toLeft 的实现也就被驱动完成了。

示例代码 2-24 src/robot-cleaner/Position.cpp

```
namespace
{
    Orientation toLeft(Orientation orientation)
    {
        switch (orientation)
        {
        case EAST:  return NORTH;
        case SOUTH: return EAST;
        case WEST:  return SOUTH;
        case NORTH: return WEST;
        default: break;
        }
        return orientation;
    }
}

void Position::turnLeft()
{
    orientation = toLeft(orientation);
}
```

也许与你共鸣，这样的设计非常不理想，已经存在很多明显的坏味道了。

1. switch-case 语句也让设计变得僵化；
2. Position 的 turnLeft 具有副作用；
3. 测试用例存在明显的重复代码；
4. turnLeft 具有严重的传染性，从 RobotCleaner 开始一直传递到 Position。

但不管怎么样，我们依然还要继续，因为到目前为止，需求我们只完成了一半。此刻更需要拒绝诱惑，在测试通过之前就将设计做大做全，不仅仅损伤自信心，而且更让设计没完没了，走不到尽头。是的，让测试通过是压倒一切的中心任务。为了快速通过测试的过程之中所使用的一切不光彩的行为，要不了多长时间，我们会回来的。

2.4 right 指令

因为我们对于问题域已经基本明确，此外抛开设计良好与否，turnRight 的实现基本上是通过 copy-paste 完成的。所以在 turnLeft 实现的经验之上，这次将步子稍微迈大一点，保证我们开发慢慢步入快速车道。总之需要根据实际情况，保持进度有快有慢，收放自如。

但无论怎么样，还是要坚持如上述实现 toLeft 的步骤，小步实现 toRight，这是一种良好的习惯。

2.4.1 快速实现

示例代码 2-25 test/robot-cleaner/TestRobotCleaner.h

```
TEST("the robot should be in (0, 0, E) when I send right instruction")
{
    robot.turnRight();
    ASSERT_EQ(Position(0, 0, EAST), robot.getPosition());
}

TEST("the robot should be in (0, 0, S) when I send right instruction with 2-times")
{
    robot.turnRight();
    robot.turnRight();

    ASSERT_EQ(Position(0, 0, SOUTH), robot.getPosition());
}

TEST("the robot should be in (0, 0, W) when I send right instruction with 3-times")
{
    robot.turnRight();
    robot.turnRight();
    robot.turnRight();

    ASSERT_EQ(Position(0, 0, WEST), robot.getPosition());
}

TEST("the robot should be in (0, 0, N) when I send right instruction with 4-times")
{
    robot.turnRight();
    robot.turnRight();
    robot.turnRight();
    robot.turnRight();

    ASSERT_EQ(Position(0, 0, NORTH), robot.getPosition());
}
```

turnRight 也非常简单，直接显示地实现。

示例代码 2-26 src/robot-cleaner/RobotCleaner.cpp

```
void RobotCleaner::turnRight()
{
    position.turnRight();
}
```

示例代码 2-27 src/robot-cleaner/Position.cpp

```
namespace
{
    Orientation toRight(Orientation orientation)
    {
        switch (orientation)
        {
        case EAST:  return SOUTH;
        case SOUTH: return WEST;
        case WEST:  return NORTH;
        case NORTH: return EAST;
        default: break;
        }
        return orientation;
    }
}

void Position::turnRight()
{
    orientation = toRight(orientation);
}
```

到目前为止，设计已经充满着坏味道，已经令人窒息。

1. turnLeft 与 turnRight 存在重复代码，并且犹如病毒一样，并在调用链上传递，从而使每一类都不能封闭修改；

2. Position 的 turnLeft，turnRight 具有副作用；

3. 测试代码也存在明显的重复设计；

4. 两个 switch-case 进一步导致设计的僵化性。

2.5 改善设计

在这之前所有的实现，都是为了快速地实现需求，即使重构也做得非常小，每次改动都能保证测试的通过。但在实现需求的路上，也欠下了很多技术债务，为此在测试保护的情况下需要进一步改善即有的设计。

2.5.1 消除重复

当实现 turnRight 时，已经释放出强烈的坏味道，turnLeft 与 turnRight 高度重复。消除 turnLeft 与 turnRight 之间的重复迫在眉睫。

先改进 Position 的设计，使用 Position::turnTo 合并实现 Position::turnLeft 与 Position::turnRight 的实现。

示例代码 2-28 src/robot-cleaner/Position.cpp

```
namespace
{
    Orientation toLeft(Orientation orientation)
    {
        switch (orientation)
        {
        case EAST:  return NORTH;
        case SOUTH: return EAST;
        case WEST:  return SOUTH;
        case NORTH: return WEST;
        default: break;
        }
        return orientation;
    }

    Orientation toRight(Orientation orientation)
    {
        switch (orientation)
        {
        case EAST:  return SOUTH;
        case SOUTH: return WEST;
        case WEST:  return NORTH;
        case NORTH: return EAST;
        default: break;
        }
        return orientation;
    }
}

void Position::turnLeft()
{
    orientation = toLeft(orientation);
}

void Position::turnRight()
{
    orientation = toRight(orientation);
}
```

示例代码 2-29 src/robot-cleaner/Position.cpp

```
namespace
{
    Orientation toLeft(Orientation orientation)
    {
        switch (orientation)
        {
        case EAST:  return NORTH;
        case SOUTH: return EAST;
        case WEST:  return SOUTH;
        case NORTH: return WEST;
        default: break;
        }

        return orientation;
    }

    Orientation toRight(Orientation orientation)
    {
        switch (orientation)
        {
        case EAST:  return SOUTH;
        case SOUTH: return WEST;
        case WEST:  return NORTH;
        case NORTH: return EAST;
        default: break;
        }
        return orientation;
    }

    Orientation to(bool left, Orientation orientation)
    {
       return left ? toLeft(orientation) : toRight(orientation);
    }
}

void Position::turnTo(bool left)
{
    orientation = to(left, orientation);
}
```

修改原来对 Position::turnLeft 与 Position::turnRight 的所有调用点，保证测试运行通过。

示例代码 2-30 src/robot-cleaner/RobotCleaner.cpp

```
void RobotCleaner::turnLeft()
{
    position.turnLeft();
}

void RobotCleaner::turnRight()
{
    position.turnRight();
}
```

示例代码 2-31 src/robot-cleaner/RobotCleaner.cpp

```
void RobotCleaner::turnLeft()
{
    position.turnTo(true);
}

void RobotCleaner::turnRight()
{
    position.turnTo(false);
}
```

虽然有晦涩的 true/false 接口，但暂时不要考虑，继续将 RobotCleaner::turnLeft 与 RobotCleaner::turnRight 之间的重复彻底消除。

示例代码 2-32 src/robot-cleaner/RobotCleaner.cpp

```
void RobotCleaner::turnLeft()
{
    position.turnTo(true);
}

void RobotCleaner::turnRight()
{
    position.turnTo(false);
}
```

示例代码 2-33 src/robot-cleaner/RobotCleaner.cpp

```
void RobotCleaner::turnTo(bool left)
{
    position.turnTo(left);
}
```

修改原来对 RobotCleaner::turnLeft 与 RobotCleaner::turnRight 的所有调用点，确保测试快速通过。

示例代码 2-34 test/robot-cleaner/TestRobotCleaner.h

```
FIXTURE(RobotCleaner)
{
    RobotCleaner robot;

    TEST("the robot should be in (0, 0, W) when I send
      instruction left")
    {
        robot.turnLeft();
        ASSERT_EQ(Position(0, 0, WEST), robot.getPosition());
    }

    TEST("the robot should be in (0, 0, E) when I send
      instruction right")
    {
        robot.turnRight();
        ASSERT_EQ(Position(0, 0, EAST), robot.getPosition());
    }
};
```

示例代码 2-35 test/robot-cleaner/TestRobotCleaner.h

```
FIXTURE(RobotCleaner)
{
    RobotCleaner robot;

    TEST("the robot should be in (0, 0, W) when I send
      instruction left")
    {
        robot.turnTo(true);
        ASSERT_EQ(Position(0, 0, WEST), robot.getPosition());
    }

    TEST("the robot should be in (0, 0, E) when I send
      instruction right")
    {
        robot.turnTo(false);
        ASSERT_EQ(Position(0, 0, EAST), robot.getPosition());
    }
};
```

晦涩的 true/false 用户接口表现非常不佳，引来了很多反对的声音。有人建议将接口改回去，因为他们受不了这样的用户接口。

事实上，消除 turnLeft 和 turnRight 重复优先级要更高，晦涩的 true/false 用户接口的表达力可以等待进一步改善和提高。

此刻的重构只是一个开始，重构可是一个循序渐进的过程，我们无法在一个时刻解决所有的问题。不要担心目前设计的存在的一些问题，关键在于我们是否有持续演进的勇气和决心。

2.5.2 改善表达力

改善 bool 接口的调用

首先要识别出 bool 接口到底存在什么样的坏味道？当用户调用 true 或者 false 的时候，不知道是什么含义，尤其调用点和被调用的距离越远，则问题越发突出。

但是，当传递 bool 变量，而非字面值 true/false 的时候，如果变量的名字具有明显 bool 语义，例如携带 is, has, should 等前缀，则不存在接口调用时晦涩的问题。此刻 bool 变量就是一个普通的变量而已，与其他变量传递没有其他特别之处。

所以不要盲目地排斥 bool 接口，有时候合理运用，则能消除大量的重复代码。对于目前的问题，首先需要改善晦涩的 true/false 的调用点，提高表达力。

示例代码 2-36 test/robot-cleaner/TestRobotCleaner.h

```
FIXTURE(RobotCleaner)
{
    RobotCleaner robot;

    TEST("the robot should be in (0, 0, W) when I send
      instruction left")
    {
        robot.turnTo(true);
        ASSERT_EQ(Position(0, 0, WEST), robot.getPosition());
    }

    TEST("the robot should be in (0, 0, E) when I send
      instruction right")
    {
        robot.turnTo(false);
        ASSERT_EQ(Position(0, 0, EAST), robot.getPosition());
    }
};
```

示例代码 2-37 test/robot-cleaner/TestRobotCleaner.h

```
FIXTURE(RobotCleaner)
{
    RobotCleaner robot;

    TEST("the robot should be in (0, 0, W) when I send
      instruction left")
    {
        robot.exec(LEFT);
        ASSERT_EQ(Position(0, 0, WEST), robot.getPosition());
    }

    TEST("the robot should be in (0, 0, E) when I send
      instruction right")
    {
        robot.exec(RIGHT);
        ASSERT_EQ(Position(0, 0, EAST), robot.getPosition());
    }
};
```

为此新建 Instruction.h 的头文件，用于表示 RobotCleaner 使用的指令集。

示例代码 2-38 include/robot-cleaner/Instruction.h

```
#ifndef HC37C3D94_43F1_4677_BD56_34CB78EFEC75
#define HC37C3D94_43F1_4677_BD56_34CB78EFEC75

enum Instruction { LEFT, RIGHT };

#endif
```

修改 RobotCleaner 的接口设计和实现。

示例代码 2-39 include/robot-cleaner/RobotCleaner.h

```
void RobotCleaner::turnTo(bool left)
{
    position.turnTo(left);
}
```

示例代码 2-40 include/robot-cleaner/RobotCleaner.h

```
void RobotCleaner::exec(Instruction instruction)
{
    position.turnTo(instruction == LEFT);
}
```

true ／ false 字面值的之间调用，被替换为 instruction == LEFT 的调用，在局部上改进了用户的表达力。但到目前位置，我所力所能及了。至于是否存在更好的解决方案，等测试更加充分，需求变得更加明确时，在进一步改善。

改善用例的表达力

使用 BDD 改善用例的可读性。不仅如此，BDD 让你更好地实践 TDD。

示例代码 2-41 test/robot-cleaner/TestRobotCleaner.h

```
FIXTURE(RobotCleaner)
{
    RobotCleaner robot;

    TEST("at the beginning, the robot should be in at the initial
      position")
    {
        ASSERT_EQ(Position(0, 0, NORTH), robot.getPosition());
    }

    TEST("the robot should be in (0, 0, W) when I send
      instruction left")
    {
        robot.exec(LEFT);
        ASSERT_EQ(Position(0, 0, WEST), robot.getPosition());
    }

    TEST("the robot should be in (0, 0, E) when I send
      instruction right")
    {
        robot.exec(RIGHT);
        ASSERT_EQ(Position(0, 0, EAST), robot.getPosition());
    }
};
```

示例代码 2-42 test/robot-cleaner/TestRobotCleaner.h

```
FIXTURE(RobotCleaner)
{
    RobotCleaner robot;

    void WHEN_I_send_instruction(Instruction instruction)
    {
        robot.exec(instruction);
    }

    void AND_I_send_instruction(Instruction instruction)
    {
        robot.exec(instruction);
    }

    void THEN_the_robot_cleaner_should_be_in(const Position&
      position)
    {
        ASSERT_EQ(position, robot.getPosition());
    }

    TEST("at the beginning")
    {
        THEN_the_robot_cleaner_should_be_in(Position(0, 0,
          NORTH));
    }

    TEST("left instruction")
    {
        WHEN_I_send_instruction(LEFT);
        THEN_the_robot_cleaner_should_be_in(Position(0, 0, WEST));
    }

    TEST("right instruction")
    {
        WHEN_I_send_instruction(RIGHT);
        THEN_the_robot_cleaner_should_be_in(Position(0, 0, EAST));
    }
};
```

2.5.3 消除条件分支语句

快速实现 Position 时，遗留两个 switch-case，让设计瞬间失去了光芒。透过对“方向”概念本质的理解，对于方向 turnTo 的操作，可以使用表驱动，即可消除糟糕的 switch-case 实现。

示例代码 2-43 src/robot-cleaner/Position.cpp

```
namespace
{
    Orientation toLeft(Orientation orientation)
    {
        switch (orientation)
        {
        case EAST:  return NORTH;
        case SOUTH: return EAST;
        case WEST:  return SOUTH;
        case NORTH: return WEST;
        default: break;
        }

        return orientation;
    }

    Orientation toRight(Orientation orientation)
    {
        switch (orientation)
        {
        case EAST:  return SOUTH;
        case SOUTH: return WEST;
        case WEST:  return NORTH;
        case NORTH: return EAST;
        default: break;
        }
        return orientation;
    }

    Orientation to(bool left, Orientation orientation)
    {
       return left ? toLeft(orientation) : toRight(orientation);
    }
}

void Position::turnTo(bool left)
{
    orientation = to(left, orientation);
}
```

示例代码 2-44 src/robot-cleaner/Position.cpp

```
namespace
{
    Orientation orientations[] = { EAST, SOUTH, WEST, NORTH };

    inline Orientation toLeft(Orientation orientation)
    {
        return orientations[(orientation + 3) % 4];
    }

    inline Orientation toRight(Orientation orientation)
    {
        return orientations[(orientation + 1) % 4];
    }

    inline Orientation to(bool left, Orientation orientation)
    {
        return left ? toLeft(orientation) : toRight(orientation);
    }
}

void Position::turnTo(bool left)
{
    orientation = to(left, orientation);
}
```

toLeft 与 toRight 算法实现依然存在重复，进一步消除重复。

示例代码 2-45 src/robot-cleaner/Position.cpp

```
namespace
{
    Orientation orientations[] = { EAST, SOUTH, WEST, NORTH };

    inline Orientation toLeft(Orientation orientation)
    {
        return orientations[(orientation + 3) % 4];
    }

    inline Orientation toRight(Orientation orientation)
    {
        return orientations[(orientation + 1) % 4];
    }

    inline Orientation to(bool left, Orientation orientation)
    {
        return left ? toLeft(orientation) : toRight(orientation);
    }
}

void Position::turnTo(bool left)
{
    orientation = to(left, orientation);
}
```

示例代码 2-46 src/robot-cleaner/Position.cpp

```
namespace
{
    Orientation orientations[] = { EAST, SOUTH, WEST, NORTH };

    inline int numOfTurnTo(bool left)
    { return left ? 3 : 1; }
}

void Position::turnTo(bool left)
{
    orientation = orientations[(orientation + numOfTurnTo(left))
      % 4];
}
```

重构完毕，测试通过，设计也变得较为清晰。

2.6 遗留问题

至此，TurnTo 已经实现了，但存在一些明显的坏味道，将其记录在 to-do list 中，后续再做改善。

1. Position 是一个典型的 Value Object，turnTo 的实现具有副作用；

2. Orientation 的枚举成员顺序与 Position.cpp 中定义的 orientations 数组元素的顺序，它们都是“方向顺序”这一知识的两种不同表现形式，具有设计的重复性；更为严重的是 turnTo 的算法实现强依赖于这样的约定，设计具有脆弱性；

3. RobotCleaner::exec 过于僵硬，扩展支持实现其他的指令时，问题更为突出；

4. turnTo 的传染性，让设计变得更加耦合；

5. 测试具有重复性。

Write programs for people first, computers second.

- Steve McConnell

3 MoveOn

3.1 需求

1. 当 Robot 收到 FORWARD 指令后，Robot 保持方向，并向前移动一个坐标；
2. 当 Robot 收到 BACKWARD 指令后，Robot 保持方向，并向后移动一个坐标；

例如：

1. Robot 原来在 (0, 0, N)，执行 FORWARD 指令后，新的位置在 (0, 1, N)；
2. Robot 原来在 (0, 0, N)，执行 BACKWARD 指令后，新的位置在 (0, -1, N)；

3.2 forward 指令

3.2.1 测试用例

示例代码 3-1 test/robot-cleaner/TestRobotCleaner.h

```
TEST("forward instruction")
{
    WHEN_I_send_instruction(FORWARD);
    THEN_the_robot_cleaner_should_be_in(Position(0, 1, NORTH));
}
```

3.2.2 引入 Command 模式

为了试图通过测试用例，简单伪实现 RobotCleaner::exec。

示例代码 3-2 src/robot-cleaner/RobotCleaner.cpp

```
void RobotCleaner::exec(Instruction instruction)
{
    position.turnTo(instruction == LEFT);
}
```

示例代码 3-3 src/robot-cleaner/RobotCleaner.cpp

```
void RobotCleaner::exec(Instruction instruction)
{
    if (instruction == FORWARD)
    {
        position = Position(0, 1, NORTH);
        return;
    }

    position.turnTo(instruction == LEFT);
}
```

测试虽然通过了，但设计已经变得无法容忍的境地了。此时需要分离出 Instruction 的变化方向，将它与执行指令的主体 RobotCleaner 相分离。为此引入 Command 模式，将指令执行的算法封装在独立的对象中，从而方便地支持指令集的扩展。

首先将 Instruction 从枚举升级为接口类；为了快速恢复测试的运行，暂时保留诸如 LEFT, RIGHT, FOWARD 等常量。

示例代码 3-4 include/robot-cleaner/Instruction.h

```
#ifndef HC37C3D94_43F1_4677_BD56_34CB78EFEC75
#define HC37C3D94_43F1_4677_BD56_34CB78EFEC75

struct Instruction { LEFT, RIGHT, FORWARD };

#endif
```

示例代码 3-5 include/robot-cleaner/Instruction.h

```
#ifndef HC37C3D94_43F1_4677_BD56_34CB78EFEC75
#define HC37C3D94_43F1_4677_BD56_34CB78EFEC75

#include "base/Role.h"

struct Position;

DEFINE_ROLE(Instruction)
{
    ABSTRACT(void exec(Position& position));
};

Instruction* left();
Instruction* right();
Instruction* forward();

#define LEFT    left()
#define RIGHT   right()
#define FORWARD forward()

#endif
```

修改原来对 Instruction 的所有调用，包括测试用例，此刻需要传递指针类型了；另外，还需要修改 RobotCleaner::exec 的接口设计和实现。

示例代码 3-6 src/robot-cleaner/RobotCleaner.cpp

```
#include "robot-cleaner/RobotCleaner.h"
#include "robot-cleaner/Instruction.h"

void RobotCleaner::exec(Instruction instruction)
{
    if (instruction == FORWARD)
    {
        position = Position(0, 1, NORTH);
        return;
    }

    position.turnTo(instruction == LEFT);
}
```

示例代码 3-7 src/robot-cleaner/RobotCleaner.cpp

```
#include "robot-cleaner/RobotCleaner.h"
#include "robot-cleaner/Instruction.h"

void RobotCleaner::exec(Instruction* instruction)
{
    instruction->exec(position);
    delete instruction;
}
```

此刻链接是失败的，为了快速链接和测试用例，创建 Instruction.cpp 的文件，实现所有的工厂方法，并将 ForwardInstruction 暂时进行伪实现。

示例代码 3-8 src/robot-cleaner/Instruction.cpp

```
#include "robot-cleaner/Instruction.h"
#include "robot-cleaner/Position.h"

namespace
{
    struct TurnToInstruction : Instruction
    {
        explicit TurnToInstruction(bool left)
          : left(left)
        {}

    private:
        OVERRIDE(void exec(Position& position))
        {
            position.turnTo(left);
        }

    private:
        bool left;
    };
}

Instruction* left()
{ return new TurnToInstruction(true); }

Instruction* right()
{ return new TurnToInstruction(false); }

namespace
{
    struct ForwardInstruction : Instruction
    {
        OVERRIDE(void exec(Position& position))
        {
            position = Position(0, 1, NORTH);
```

```
        }
    };
}

Instruction* forward()
{ return new ForwardInstruction; }
```

3.2.3 消除重复

Instruction 中指令存在重复，例如 left 和 LEFT 事实上是同一个概念。宏定义的存在主要是为了让测试尽快通过而暂时保留的产物。

示例代码 3-9 include/robot-cleaner/Instruction.h

```
#ifndef HC37C3D94_43F1_4677_BD56_34CB78EFEC75
#define HC37C3D94_43F1_4677_BD56_34CB78EFEC75

#include "base/Role.h"

struct Position;

DEFINE_ROLE(Instruction)
{
    ABSTRACT(void exec(Position& position));
};

Instruction* left();
Instruction* right();
Instruction* forward();

#define LEFT    left()
#define RIGHT   right()
#define FORWARD forward()

#endif
```

首先直接使用工厂方法替代宏定义表示指令，确保测试快速通过。

示例代码 3-10 test/robot-cleaner/TestRobotCleaner.h

```
TEST("left instruction")
{
    WHEN_I_send_instruction(LEFT);
    THEN_the_robot_cleaner_should_be_in(Position(0, 0, WEST));
}

TEST("right instruction")
{
    WHEN_I_send_instruction(RIGHT);
    THEN_the_robot_cleaner_should_be_in(Position(0, 0, EAST));
}

TEST("forward instruction")
{
    WHEN_I_send_instruction(FORWARD);
    THEN_the_robot_cleaner_should_be_in(Position(0, 1, NORTH));
}
```

示例代码 3-11 test/robot-cleaner/TestRobotCleaner.h

```
TEST("left instruction")
{
    WHEN_I_send_instruction(left());
    THEN_the_robot_cleaner_should_be_in(Position(0, 0, WEST));
}

TEST("right instruction")
{
    WHEN_I_send_instruction(right());
    THEN_the_robot_cleaner_should_be_in(Position(0, 0, EAST));
}

TEST("forward instruction")
{
    WHEN_I_send_instruction(forward());
    THEN_the_robot_cleaner_should_be_in(Position(0, 1, NORTH));
}
```

然后从 Instruction.h 删除指令的所有宏定义，运行测试保证通过。

示例代码 3-12 include/robot-cleaner/Instruction.h

```
#ifndef HC37C3D94_43F1_4677_BD56_34CB78EFEC75
#define HC37C3D94_43F1_4677_BD56_34CB78EFEC75

#include "base/Role.h"

struct Position;

DEFINE_ROLE(Instruction)
{
    ABSTRACT(void exec(Position& position));
};

Instruction* left();
Instruction* right();
Instruction* forward();

#define LEFT    left()
#define RIGHT   right()
#define FORWARD forward()

#endif
```

示例代码 3-13 include/robot-cleaner/Instruction.h

```
#ifndef HC37C3D94_43F1_4677_BD56_34CB78EFEC75
#define HC37C3D94_43F1_4677_BD56_34CB78EFEC75

#include "base/Role.h"

struct Position;

DEFINE_ROLE(Instruction)
{
    ABSTRACT(void exec(Position& position));
};

Instruction* left();
Instruction* right();
Instruction* forward();

#endif
```

3.2.4 实现 forward 指令

先消除 ForwardInstruction::exec 的伪实现, 将 ForwardInstruction::exec 委托给 Position。

示例代码 3-14 src/robot-cleaner/Instruction.cpp

```
namespace
{
    struct ForwardInstruction : Instruction
    {
        OVERRIDE(void exec(Position& position))
        {
            position = Position(0, 0, E);
        }
    };
}
```

示例代码 3-15 src/robot-cleaner/Instruction.cpp

```
namespace
{
    struct ForwardInstruction : Instruction
    {
        OVERRIDE(void exec(Position& position))
        {
            position.forward();
        }
    };
}
```

为了让测试尽快通过，然后对 Position::forward 进行伪实现。

示例代码 3-16 src/robot-cleaner/Position.cpp

```
void Position::forward()
{
    y += 1;
}
```

测试用例

示例代码 3-17 test/robot-cleaner/TestRobotCleaner.h

```
TEST("left, forward instructions")
{
    WHEN_I_send_instruction(left());
    WHEN_I_send_instruction(forward());

    THEN_the_robot_cleaner_should_be_in(Position(-1, 0, WEST));
}
```

然后对 Position::forward 进行伪实现，简单地使用 switch-case 实现。

示例代码 3-18 src/robot-cleaner/Position.cpp

```
void Position::forward()
{
    switch (orientation)
    {
    case NORTH:
        x += 0; y += 1;
        break;
    case WEST:
        x -= 1; y += 0;
        break;
    default: break;
    }
}
```

测试通过了，重构非常成功。循环这个微小的 TDD 环，将场景列举一下，将 Position::forward 用最笨的方法驱动实现出来。

示例代码 3-19 test/robot-cleaner/TestRobotCleaner.h

```
TEST("2-left, forward instructions")
{
    WHEN_I_send_instruction(left());
    AND_I_send_instruction(left());
    AND_I_send_instruction(forward());

    THEN_the_robot_cleaner_should_be_in(Position(0, -1, SOUTH));
}

TEST("3-left, forward instructions")
{
    WHEN_I_send_instruction(left());
    AND_I_send_instruction(left());
    AND_I_send_instruction(left());
    AND_I_send_instruction(forward());

    THEN_the_robot_cleaner_should_be_in(Position(1, 0, EAST));
}

TEST("4-left, forward instructions")
{
    WHEN_I_send_instruction(left());
    AND_I_send_instruction(left());
    AND_I_send_instruction(left());
    AND_I_send_instruction(left());
    AND_I_send_instruction(forward());

    THEN_the_robot_cleaner_should_be_in(Position(0, 1, NORTH));
}
```

最终，Position::forward 的实现如下，的确很不光彩，但这也让我信心倍增，毕竟我通过了所有的测试。

示例代码 3-20 src/robot-cleaner/Position.cpp

```
void Position::forward()
{
    switch (orientation)
    {
    case EAST:
        x += 1; y += 0;
        break;
    case SOUTH:
        x += 0; y -= 1;
        break;
    case WEST:
        x -= 1; y += 0;
        break;
    case NORTH:
        x += 0; y += 1;
        break;
    default: break;
    }
}
```

测试通过后，接下来将这个刺眼的 Position::forward 重构掉。类似于 turnTo 的实现，forward 也存在一个简单、有效的算法，来替换这个脆弱的 switch-case 实现。

示例代码 3-21 src/robot-cleaner/Position.cpp

```
void Position::forward()
{
    switch (orientation)
    {
    case EAST:
        x += 1; y += 0;
        break;
    case SOUTH:
        x += 0; y -= 1;
        break;
    case WEST:
        x -= 1; y += 0;
        break;
    case NORTH:
        x += 0; y += 1;
        break;
    default: break;
    }
}
```

示例代码 3-22 src/robot-cleaner/Position.cpp

```
namespace
{
    const int offsets[] = { 1, 0, -1, 0 };
}

void Position::forward()
{
    x += offsets[orientation];
    y += offsets[3 - orientation];
}
```

测试通过了，重构非常成功，终于将 switch-case 干掉了。

3.3 backward 指令

3.3.1 快速实现

backward 与 forward 逻辑差不多，它与 forward 具有逻辑反的关系。鉴于 forward 的经验，每次一个微小的 TDD 循环，小步快跑地实现了 backward 指令。

示例代码 3-23 test/robot-cleaner/TestRobotCleaner.h

```
TEST("backward instruction")
{
    WHEN_I_send_instruction(backward());
    THEN_the_robot_cleaner_should_be_in(Position(0, -1, NORTH));
}

TEST("1-left, backward instruction")
{
    WHEN_I_send_instruction(left());
    AND_I_send_instruction(backward());

    THEN_the_robot_cleaner_should_be_in(Position(1, 0, WEST));
}

TEST("2-left, backward instruction")
{
    WHEN_I_send_instruction(left());
    AND_I_send_instruction(left());
    AND_I_send_instruction(backward());

    THEN_the_robot_cleaner_should_be_in(Position(0, 1, SOUTH));
}

TEST("3-left, backward instruction")
{
    WHEN_I_send_instruction(left());
    AND_I_send_instruction(left());
    AND_I_send_instruction(left());
    AND_I_send_instruction(backward());

    THEN_the_robot_cleaner_should_be_in(Position(-1, 0, EAST));
}

TEST("4-left, backward instruction")
{
    WHEN_I_send_instruction(left());
    AND_I_send_instruction(left());
    AND_I_send_instruction(left());
    AND_I_send_instruction(left());
    AND_I_send_instruction(backward());

    THEN_the_robot_cleaner_should_be_in(Position(0, -1, NORTH));
}
```

快速实现 backward 关键字，并将职责委托给 Position::backward。

示例代码 3-24 src/robot-cleaner/Instruction.cpp

```
namespace
{
    struct BackwardInstruction : Instruction
    {
        OVERRIDE(void exec(Position& position))
        {
            position.backward();
        }
    };
}

Instruction* backward()
{
    return new BackwardInstruction;
}
```

按照 forward 的实现，backward 也很容易实现，因为它只是 forward 的逻辑反。

示例代码 3-25 src/robot-cleaner/Position.cpp

```
namespace
{
    const int offsets[] = { 1, 0, -1, 0 };
}

void Position::forward()
{
    x += offsets[orientation];
    y += offsets[3 - orientation];
}

void Position::backward()
{
    x -= offsets[orientation];
    y -= offsets[3 - orientation];
}
```

至此测试通过了。

3.3.2 消除重复

接下来需要处理将 Forward 与 Backward 之间的重复设计，重构第一步先将二者合并为 MoveOn 操作。

示例代码 3-26 src/robot-cleaner/Instruction.cpp

```
namespace
{
    struct ForwardInstruction : Instruction
    {
        OVERRIDE(void exec(Position& position))
        {
            position.forward();
        }
    };
}

Instruction* forward()
{
    return new ForwardInstruction;
}

namespace
{
    struct BackwardInstruction : Instruction
    {
        OVERRIDE(void exec(Position& position))
        {
            position.backward();
        }
    };
}

Instruction* backward()
{
    return new BackwardInstruction;
}
```

示例代码 3-27 src/robot-cleaner/Instruction.cpp

```
namespace
{
    struct MoveOnInstruction : Instruction
    {
        explicit MoveOnInstruction(bool forward)
          : step(forward ? 1 : -1)
        {}

    private:
        OVERRIDE(void exec(Position& position))
        {
            position.moveOn(step);
        }

    private:
        int step;
    };
}

Instruction* forward()
{ return new MoveOnInstruction(true); }

Instruction* backward()
{ return new MoveOnInstruction(false); }
```

同样地，将 Position 中原来的 forward 与 backward 接口合并为一个 moveOn 接口。

示例代码 3-28 src/robot-cleaner/Position.cpp

```
void Position::forward()
{
    x += offsets[orientation];
    y += offsets[3 - orientation];
}

void Position::backward()
{
    x -= offsets[orientation];
    y -= offsets[3 - orientation];
}
```

示例代码 3-29 src/robot-cleaner/Position.cpp

```
void Position::moveOn(int step)
{
    x += step*offsets[orientation];
    y += step*offsets[3-orientation];
}
```

3.3.3 改善表达力

offsets[3-orientation] 的逻辑有些晦涩，一眼看不出来这是什么东西。第一个感觉给它加一个注释，但最好的策略是改善代码本身的可读性。

事实上，offsets[orientation] 其实表示的是：当面朝 orientation 时，Robot 执行 forward 指令时，在 X 轴上的偏移量；offsets[3-orientation] 则表示：当面朝 orientation 时，Robot 执行 forward 指令时，在 Y 轴上的偏移量。

示例代码 3-30 src/robot-cleaner/Position.cpp

```
void Position::moveOn(int step)
{
    x += step*offsets[orientation];
    y += step*offsets[3-orientation];
}
```

示例代码 3-31 src/robot-cleaner/Position.cpp

```
inline int Position::getOffsetOfX()
{
    return offsets[orientation];
}

inline int Position::getStepOfY()
{
    return offsets[3 - orientation];
}

void Position::moveOn(int step)
{
    x += step*getStepOfX();
    y += step*getStepOfY();
}
```

至此测试通过，重构完毕。

3.4 遗留问题

至此，MoveOn 已经实现了，但依然存在一些明显的坏味道，将其记录在 to-do list 中，后续再做改善。

1. Position 是一个典型的 Value Object，turnTo 的实现具有副作用；

2. Orientation 的枚举成员顺序，Position.cpp 中定义的 orientations, offsets 数组元素的顺序，它们都是“方向顺序”这一知识的三种不同表现形式，具有设计的重复性；更为严重的是 turnTo, moveOn 的算法实现强依赖于此约定，设计具有脆弱性；

3. turnTo, moveOn 具有严重的传染性，让设计变得更加耦合；

4. 测试用例存在重复设计。

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

4 Round

4.1 需求

1. 当 Robot 收到 ROUND 指令后，Robot 顺时针旋转 180 度掉头；
2. 例如 Robot 原来在 (0, 0, N)，执行 ROUND 指令后，新的位置在 (0, 0, S)；

4.2 round 指令

4.2.1 测试用例

示例代码 4-1 test/robot-cleaner/TestRobotCleaner.h

```
TEST("round instruction")
{
    WHEN_I_send_instruction(round());
    THEN_the_robot_cleaner_should_be_in(Position(0, 0, SOUTH));
}
```

4.2.2 实现 round 指令

示例代码 4-2 include/robot-cleaner/Instruction.h

```
#ifndef HC37C3D94_43F1_4677_BD56_34CB78EFEC75
#define HC37C3D94_43F1_4677_BD56_34CB78EFEC75

#include "base/Role.h"

struct Position;

DEFINE_ROLE(Instruction)
{
    ABSTRACT(void exec(Position& position));
};

Instruction* round();

#endif
```

编译通过了，这次让步子迈大了一点，为了快速让链接和测试通过，先伪实现。

示例代码 4-3 src/robot-cleaner/Instruction.cpp

```
namespace
{
    struct RoundInstruction : Instruction
    {
        OVERRIDE(void exec(Position& position))
        {
            position = Position(0, 0, SOUTH);
        }
    };
}

Instruction* round()
{
    return new RoundInstruction;
}
```

测试通过了，接下来改善设计。经过分析，round 指令只不过是连续的两次 right 指令，为此引入 RepeatedInstruction 的概念，用于修饰 right 指令，它将 right 指令连续执行 2 次。

示例代码 4-4 src/robot-cleaner/Instruction.cpp

```
namespace
{
    struct RoundInstruction : Instruction
    {
        OVERRIDE(void exec(Position& position))
        {
            position = Position(0, 0, SOUTH);
        }
    };
}

Instruction* round()
{
    return new RoundInstruction;
}
```

示例代码 4-5 src/robot-cleaner/Instruction.cpp

```
namespace
{
    struct RepeatedInstruction : Instruction
    {
        RepeatedInstruction(Instruction* instruction, int n)
         : instruction(instruction), n(n)
        {}

    private:
        OVERRIDE(void exec(Point& point, Orientation&
          orientation) const)
        {
            for (auto i=0; i<n; i++)
            {
                instruction->exec(point, orientation);
            }
        }

    private:
        std::unique_ptr<Instruction> instruction;
        int n;
    };
}

Instruction* round()
{
    return new RepeatedInstruction(right(), 2);
}
```

测试通过了。

Premature optimization is the root of all evil.

– Donald Knuth

5 MoveOn(n)

5.1 需求

1. 当 Robot 收到 FORWARD(n) 指令后，Robot 保持方向，向前移动 n 个坐标；
2. 当 Robot 收到 BACKWARD(n) 指令后，Robot 保持方向，向后移动 n 个坐标；
3. 其中 n 在 [1..10] 之间；

例如：

1. Robot 原来在 (0, 0, N)，执行 FORWARD(10) 指令后，新的位置在 (0, 10, N)；
2. Robot 原来在 (0, 0, N)，执行 BACKWARD(10) 指令后，新的位置在 (0, -10, N)；

5.2 forward_n 指令

5.2.1 测试用例

示例代码 5-1 test/robot-cleaner/TestRobotCleaner.h

```
TEST("forward(n) instruction")
{
    WHEN_I_send_instruction(forward_n(10));
    THEN_the_robot_cleaner_should_be_in(Position(0, 10, NORTH));
}
```

5.2.2 实现 forward_n 指令

我很有自信地、大胆地将 forward 委托给 forward_n，步子虽然有点大，而且冒着让已有的用例失败的可能性，但就问题域而言，还在自己控制范围之内。

示例代码 5-2 include/robot-cleaner/Instruction.h

```
#ifndef HC37C3D94_43F1_4677_BD56_34CB78EFEC75
#define HC37C3D94_43F1_4677_BD56_34CB78EFEC75

#include "base/Role.h"

struct Position;

DEFINE_ROLE(Instruction)
{
    ABSTRACT(void exec(Position& position));
};

Instruction* forward_n(int n);

inline Instruction* forward()
{ return forward_n(1); }

Instruction* backward();

#endif
```

为了保证即有测试通过，此处 backward 的实现，进行了简单的处理。

示例代码 5-3 src/robot-cleaner/Instruction.cpp

```
namespace
{
    struct MoveOnInstruction : Instruction
    {
        explicit MoveOnInstruction(int n, bool forward)
          : step(prefix(forward)*n)
        {}

    private:
        OVERRIDE(void exec(Position& position))
        {
            position.moveOn(step);
        }

    private:
        static int prefix(bool forward)
        {
            return forward ? 1 : -1;
        }

    private:
        int step;
    };
}

Instruction* forward_n(int n)
{ return new MoveOnInstruction(n, true); }

Instruction* backward()
{ return new MoveOnInstruction(1, false); }
```

5.3 backward_n 指令

5.3.1 测试用例

示例代码 5-4 test/robot-cleaner/TestRobotCleaner.h

```
TEST("forward(n) instruction")
{
    WHEN_I_send_instruction(backward_n(10));
    THEN_the_robot_cleaner_should_be_in(Position(0, -10, NORTH));
}
```

5.3.2 实现 backward_n 指令

同样的道理，将 backward 内联，委托给 backward_n 指令。

示例代码 5-5 include/robot-cleaner/Instruction.h

```
#ifndef HC37C3D94_43F1_4677_BD56_34CB78EFEC75
#define HC37C3D94_43F1_4677_BD56_34CB78EFEC75

#include "base/Role.h"

struct Position;

DEFINE_ROLE(Instruction)
{
    ABSTRACT(void exec(Position& position));
};

Instruction* forward_n(int n);
Instruction* backward_n(int n);

inline Instruction* forward()
{ return forward_n(1); }

inline Instruction* backward()
{ return backward_n(1); }

#endif
```

在将 backward_n 显式实现。

示例代码 5-6 src/robot-cleaner/Instruction.cpp

```
namespace
{
    struct MoveOnInstruction : Instruction
    {
        explicit MoveOnInstruction(int n, bool forward)
         : step(prefix(forward)*n)
        {}

    private:
        OVERRIDE(void exec(Position& position))
        {
            position.moveOn(step);
        }

    private:
        static int prefix(bool forward)
        {
            return forward ? 1 : -1;
        }

    private:
        int step;
    };
}

Instruction* forward_n(int n)
{
    return new MoveOnInstruction(n, true);
}

Instruction* backward_n(int n)
{
    return new MoveOnInstruction(n, false);
}
```

5.4 边界

5.4.1 测试用例

示例代码 5-7 src/robot-cleaner/Instruction.cpp

```
TEST("forward(n) instruction: n out of bound")
{
    WHEN_I_send_instruction(forward_n(11));
    THEN_the_robot_cleaner_should_be_in(Position(0, 0, NORTH));
}

TEST("backward(n) instruction: n out of bound")
{
    WHEN_I_send_instruction(backward_n(11));
    THEN_the_robot_cleaner_should_be_in(Position(0, 0, NORTH));
}
```

5.4.2 快速实现

示例代码 5-8 src/robot-cleaner/Instruction.cpp

```
namespace
{
    struct MoveOnInstruction : Instruction
    {
        explicit MoveOnInstruction(int n, bool forward)
          : step(prefix(forward)*n)
        {}

    private:
        OVERRIDE(void exec(Position& position))
        {
            position.moveOn(step);
        }

    private:
        static int prefix(bool forward)
        {
            return forward ? 1 : -1;
        }

    private:
        int step;
    };
}

Instruction* forward_n(int n)
{
    return new MoveOnInstruction(n, true);
}

Instruction* backward_n(int n)
{
    return new MoveOnInstruction(n, false);
}
```

示例代码 5-9 src/robot-cleaner/Instruction.cpp

```
namespace
{
    const int MIN_MOVE_NUM = 1;
    const int MAX_MOVE_NUM = 10;

    struct MoveOnInstruction : Instruction
    {
        explicit MoveOnInstruction(int n, bool forward)
          : step(prefix(forward)*protect(n))
        {}

    private:
        OVERRIDE(void exec(Position& position))
        {
            position.moveOn(step);
        }

    private:
        static int prefix(bool forward)
        {
            return forward ? 1 : -1;
        }

        static int protect(int n)
        {
            return (MIN_MOVE_NUM <= n
              && n <= MAX_MOVE_NUM) ? n : 0;
        }

    private:
        int step;
    };
}

Instruction* forward_n(int n)
{
    return new MoveOnInstruction(n, true);
}

Instruction* backward_n(int n)
{
    return new MoveOnInstruction(n, false);
}
```

测试通过了，但此处处理的手段还不够光彩，有待进一步改善。

Make it work, make it right, make it fast.

- Kent Beck

6 Sequential

6.1 需求

当 Robot 收到一系列组合指令时，能够依次按指令完成相应的动作。例如收到指令序列：[LEFT, FORWARD, RIGHT, BACKWARD, ROUND, FORWARD(2)]，将依次执行：向左转 90 度；保持方向并向前移动一个坐标；向右转 90 度；保持方向并向后退一个坐标；顺时针旋转 180 度掉头；保持方向向前移动 2 个坐标。

6.2 sequential 指令

6.2.1 测试用例

示例代码 6-1 test/robot-cleaner/TestRobotCleaner.h

```
TEST("sequential instruction")
{
    WHEN_I_send_instruction(sequential(left(), forward(), round()));
    THEN_the_robot_cleaner_should_be_in(Position(-1, 0, EAST));
}
```

6.2.2 实现 sequential 指令

示例代码 6-2 include/robot-cleaner/Instruction.h

```
#ifndef HC37C3D94_43F1_4677_BD56_34CB78EFEC75
#define HC37C3D94_43F1_4677_BD56_34CB78EFEC75

#include "base/Role.h"
#include <initializer_list>

struct Position;

DEFINE_ROLE(Instruction)
{
    ABSTRACT(void exec(Position& position));
};

Instruction* __sequential(std::initializer_list<Instruction*>);

#define sequential(...) __sequential({ __VA_ARGS__ })

#endif
```

此处使用了 C++11 的特性，从而是接口变得更加直观、漂亮。

示例代码 6-3 src/robot-cleaner/Instruction.cpp

```
#include "robot-cleaner/Instruction.h"
#include <vector>

namespace
{
    struct SequentialInstruction : Instruction
    {
```

```
        SequentialInstruction(std::initializer_list<Instruction*> instructions)
          : instructions(instructions)
        {}

        ~SequentialInstruction()
        {
            for(auto instruction : instructions)
            {
                delete instruction;
            }
        }

    private:
        OVERRIDE(void exec(Position& position))
        {
            for(auto instruction : instructions)
            {
                instruction->exec(position);
            }
        }

    private:
        std::vector<Instruction*> instructions;
    };
}

Instruction* __sequential(std::initializer_list<Instruction*> instructions)
{
    return new SequentialInstruction(instructions);
}
```

SequentialInstruction 采用了组合模式，整合了“一多”之分，在此使用也非常恰当。

Controlling complexity is the essence of computer programming.

- Brian Kernighan

7 Repeat

7.1 需求

当 Robot 收到 REAPT(instruction, n) 指令后，它会循环执行 instruction 指令 n 次。原来在 (0, 0, N)，执行 REAPT(FORWARD(2), 2) 指令后，新的位置为 (0, 4, NORTH)；其中 n 在 [1..10] 之间。

7.2 repeat 指令

7.2.1 测试用例

示例代码 7-1 test/robot-cleaner/TestRobotCleaner.h

```
TEST("repeat instruction")
{
    WHEN_I_send_instruction(repeat(forward_n(2), 2));
    THEN_the_robot_cleaner_should_be_in(Position(0, 4, NORTH));
}
```

7.2.2 实现 repeat 指令

提供 repeat 关键字，并将 round 指令内联。

示例代码 7-2 include/robot-cleaner/Instruction.h

```
#ifndef HC37C3D94_43F1_4677_BD56_34CB78EFEC75
#define HC37C3D94_43F1_4677_BD56_34CB78EFEC75

#include "base/Role.h"

struct Position;

DEFINE_ROLE(Instruction)
{
    ABSTRACT(void exec(Position& position));
};

Instruction* repeat(Instruction* instruction, int n);

inline Instruction* round()
{ return repeat(right(), 2); }
```

示例代码 7-3 src/robot-cleaner/Instruction.cpp

```
namespace
{
    struct RepeatedInstruction : Instruction
    {
        RepeatedInstruction(Instruction* instruction, int n)
         : instruction(instruction), n(n)
        {}

    private:
        OVERRIDE(void exec(Point& point, Orientation&
          orientation) const)
        {
            for (auto i=0; i<n; i++)
            {
                instruction->exec(point, orientation);
            }
        }

    private:
        std::unique_ptr<Instruction> instruction;
        int n;
    };
}

Instruction* repeat(Instruction* instruction, int n)
{
    return new RepeatedInstruction(instruction, n);
}
```

测试通过，功能实现了。

7.2.3 边界

测试用例

示例代码 7-4 test/robot-cleaner/TestRobotCleaner.h

```
TEST("repeat instruction: n out of bound")
{
    WHEN_I_send_instruction(repeat(forward_n(2), 11));
    THEN_the_robot_cleaner_should_be_in(Position(0, 0, NORTH));
}
```

快速实现

示例代码 7-5 src/robot-cleaner/Instruction.cpp

```
namespace
{
    struct RepeatedInstruction : Instruction
    {
        RepeatedInstruction(Instruction* instruction, int n)
         : instruction(instruction), n(n)
        {}

    private:
        OVERRIDE(void exec(Point& point, Orientation&
          orientation) const)
        {
            for (auto i=0; i<n; i++)
            {
                instruction->exec(point, orientation);
            }
        }

    private:
        std::unique_ptr<Instruction> instruction;
        int n;
    };
}

Instruction* repeat(Instruction* instruction, int n)
{
    return new RepeatedInstruction(instruction, n);
}
```

示例代码 7-6 src/robot-cleaner/Instruction.cpp

```
namespace
{
    const int MIN_REPEATED_NUM = 1;
    const int MAX_REPEATED_NUM = 10;

    struct RepeatedInstruction : Instruction
    {
        RepeatedInstruction(Instruction* instruction, int n)
         : instruction(instruction), n(n)
        {}

    private:
        OVERRIDE(void exec(Point& point, Orientation&
          orientation) const)
        {
            for (auto i=0; i<n; i++)
            {
                instruction->exec(point, orientation);
            }
        }

    private:
        static int protect(int n)
        {
            return (MIN_REPEATED_NUM <= n
              && n <= MAX_REPEATED_NUM) ? n : 0;
        }

    private:
        std::unique_ptr<Instruction> instruction;
        int n;
    };
}

Instruction* repeat(Instruction* instruction, int n)
{
    return new RepeatedInstruction(instruction, n);
}
```

测试通过了，但实现的很不光彩，而且与 MoveOnInstruction 的边界处理产生了重复设计，不仅仅是实现上的重复，关键在与对越界处理的知识的重复（越界时，规整为 0），需要进一步消除重复。

Null Object

为了统一 MoveToInstruction 与 RepeatedInstruction 的边界处理，此处可引入 Null Object 模式，将边界处理得更加自然。

示例代码 7-7 src/robot-cleaner/Instruction.cpp

```
namespace
{
    const int MIN_MOVE_NUM = 1;
    const int MAX_MOVE_NUM = 10;

    struct MoveOnInstruction : Instruction
    {
        explicit MoveOnInstruction(int n, bool forward)
          : step(prefix(forward)*protect(n)) {}

    private:
        OVERRIDE(void exec(Position& position))
        {
            position.moveOn(step);
        }

    private:
        static int prefix(bool forward)
        {
            return forward ? 1 : -1;
        }

        static int protect(int n)
        {
            return (MIN_MOVE_NUM <= n
              && n <= MAX_MOVE_NUM) ? n : 0;
        }

    private:
        int step;
    };
}

Instruction* forward_n(int n)
{
    return new MoveOnInstruction(n, true);
}

Instruction* backward_n(int n)
{
    return new MoveOnInstruction(n, false);
}

namespace
{
    const int MIN_REPEATED_NUM = 1;
    const int MAX_REPEATED_NUM = 10;

    struct RepeatedInstruction : Instruction
    {
        RepeatedInstruction(Instruction* instruction, int n)
         : instruction(instruction), n(protect(n))
        {}

    private:
        OVERRIDE(void exec(Position& position))
        {
            for (auto i=0; i<n; i++)
            {
                instruction->exec(position);
            }
        }

    private:
        static int protect(int n)
        {
            return (MIN_REPEATED_NUM <= n
              && n <= MAX_REPEATED_NUM) ? n : 0;
        }

    private:
        std::unique_ptr<Instruction> instruction;
        int n;
    };
}

Instruction* repeat(Instruction* instruction, int n)
{ return new RepeatedInstruction(instruction, n); }
```

示例代码 7-8 src/robot-cleaner/Instruction.cpp

```
namespace
{
#define ASSERT_BETWEEN(num, min, max)   \
    do {                                \
       if (num<min || num>max)          \
          return new EmptyInstruction   \
    } while(0)

    struct EmptyInstruction : Instruction
    {
        OVERRIDE(void exec(Position&)) {}
    };
}

namespace
{
    struct MoveOnInstruction : Instruction
    {
        explicit MoveOnInstruction(int n, bool forward)
          : step(prefix(forward)*n) {}

    private:
        OVERRIDE(void exec(Position& position))
        {
            position.moveOn(step);
        }

    private:
        static int prefix(bool forward)
        {
            return forward ? 1 : -1;
        }

    private:
        int step;
    };

    const int MIN_MOVE_NUM = 1;
    const int MAX_MOVE_NUM = 10;

    Instruction* move_on(bool forward, int n)
    {
        ASSERT_BETWEEN(n, MIN_MOVE_NUM, MAX_MOVE_NUM);
        return new MoveOnInstruction(forward,  n);
    }
}

Instruction* forward_n(int n)
{
    return move_on(true, n);
}

Instruction* backward_n(int n)
{
    return move_on(true, n);
}

namespace
{
    struct RepeatedInstruction : Instruction
    {
        RepeatedInstruction(Instruction* instruction, int n)
         : instruction(instruction), n(n)
        {}

    private:
        OVERRIDE(void exec(Position& position))
        {
            for (auto i=0; i<n; i++)
            {
                instruction->exec(position);
            }
        }

    private:
        std::unique_ptr<Instruction> instruction;
        int n;
    };

    const int MIN_REPEATED_NUM = 1;
    const int MAX_REPEATED_NUM = 10;
}

Instruction* repeat(Instruction* instruction, int n)
{
    ASSERT_BETWEEN(n, MIN_REPEATED_NUM, MAX_REPEATED_NUM);
    return new RepeatedInstruction(instruction, n);
}
```

测试通过了，重构相当成功。

7.2.4 消除测试用例的重复

用例之间存在重复代码，可以进一步进行提取和封装，消除重复代码。

示例代码 7-9 test/robot-cleaner/TestRobotCleaner.h

```
FIXTURE(RobotCleaner)
{
    RobotCleaner robot;

    TEST("left instruction: 1-times")
    {
        WHEN_I_send_instruction(left());
        THEN_the_robot_cleaner_should_be_in(Position(0, 0, WEST));
    }

    TEST("left instruction: 2-times")
    {
        WHEN_I_send_instruction(left());
        AND_I_send_instruction(left());

        THEN_the_robot_cleaner_should_be_in(Position(0, 0,
          SOUTH));
    }

    TEST("left instruction: 3-times")
    {
        WHEN_I_send_instruction(left());
        AND_I_send_instruction(left());
        AND_I_send_instruction(left());

        THEN_the_robot_cleaner_should_be_in(Position(0, 0, EAST));
    }

    TEST("left instruction: 4-times")
    {
        WHEN_I_send_instruction(left());
        AND_I_send_instruction(left());
        AND_I_send_instruction(left());
        AND_I_send_instruction(left());

        THEN_the_robot_cleaner_should_be_in(Position(0, 0,
          NORTH));
    }
};
```

示例代码 7-10 test/robot-cleaner/TestRobotCleaner.h

```
FIXTURE(RobotCleaner)
{
    RobotCleaner robot;

    TEST("left instruction: 1-times")
    {
        WHEN_I_send_instruction(left());
        THEN_the_robot_cleaner_should_be_in(Position(0, 0, WEST));
    }

    TEST("left instruction: 2-times")
    {
        WHEN_I_send_instruction(repeat(left(), 2));
        THEN_the_robot_cleaner_should_be_in(Position(0, 0,
          SOUTH));
    }

    TEST("left instruction: 3-times")
    {
        WHEN_I_send_instruction(repeat(left(), 3));
        THEN_the_robot_cleaner_should_be_in(Position(0, 0, EAST));
    }

    TEST("left instruction: 4-times")
    {
        WHEN_I_send_instruction(repeat(left(), 4));
        THEN_the_robot_cleaner_should_be_in(Position(0, 0,
          NORTH));
    }
};
```

测试通过，重构完毕。

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Flower

8

Orientation

8.1 改善设计

8.1.1 重复设计

至此，所有的需求和测试均以通过，但还存在一个重要的问题还未解决。关于 Orientation 枚举成员顺序的知识重复。目前已经存在 3 个地方暗喻了这个知识的存在，而且更为严重的是，MoveOn 和 TurnTo 的算法都与它们紧密关联。

示例代码 8-1 include/robot-cleaner/Orientation.h

```
#ifndef H9811B75A_15B3_4DF0_91B7_483C42F74473
#define H9811B75A_15B3_4DF0_91B7_483C42F74473

enum Orientation { EAST, SOUTH, WEST, NORTH };

#endif
```

示例代码 8-2 src/robot-cleaner/Position.cpp

```
namespace
{
    Orientation orientations[] = { EAST, SOUTH, WEST, NORTH };
    const int offsets[] = { 1, 0, -1, 0 };
}
```

8.1.2 安全枚举类型

首先将 Orientation 从枚举提升为类，并快速恢复测试。

示例代码 8-3 include/robot-cleaner/Orientation.h

```
#ifndef H9811B75A_15B3_4DF0_91B7_483C42F74473
#define H9811B75A_15B3_4DF0_91B7_483C42F74473

enum Orientation { EAST, SOUTH, WEST, NORTH };

#endif
```

示例代码 8-4 include/robot-cleaner/Orientation.h

```
#ifndef H9811B75A_15B3_4DF0_91B7_483C42F74473
#define H9811B75A_15B3_4DF0_91B7_483C42F74473

#include "base/EqHelper.h"

struct Orientation
{
    operator int() const;

#define DECL_ORIENTATION(orientation) \
    static const Orientation orientation;

    DECL_ORIENTATION(east)
    DECL_ORIENTATION(south)
    DECL_ORIENTATION(west)
    DECL_ORIENTATION(north)

    DECL_EQUALS(Point);

private:
    Orientation(int order);

private:
    int order;
};

#define EAST  Orientation::east
#define SOUTH Orientation::south
#define WEST  Orientation::west
#define NORTH Orientation::north

#endif
```

此处暂时定义一个 Orientation::operator int，让 Orientation 工作起来，就犹如普通的枚举类型一样参与整数运算，保证测试用例尽快运行成功。

此时编译通过了，但链接不过，快速实现 Orientation.cpp。

示例代码 8-5 src/robot-cleaner/Orientation.cpp

```
#include "robot-cleaner/Orientation.h"

Orientation::operator int() const
{
    return order;
}

inline Orientation::Orientation(int order)
  : order(order)
{}

#define DEF_ORIENTATION(orientation, order) \
const Orientation Orientation::orientation(order);

DEF_ORIENTATION(east,  0)
DEF_ORIENTATION(south, 1)
DEF_ORIENTATION(west,  2)
DEF_ORIENTATION(north, 3)

DEF_EQUALS(Orientation)
{
    return FIELD_EQ(order);
}
```

测试通过了，此处 Orientation 维护了一个 order 变量，类似于普通枚举成员的值。

最后将原来 Orientation 的所有 pass-by-value 的地方都换成 pass-by-ref-to-const，保证测试运行通过。

示例代码 8-6 src/robot-cleaner/Position.cpp

```
Position::Position(int x, int y, Orientation orientation)
  : x(x), y(y), orientation(orientation)
{
}
```

示例代码 8-7 src/robot-cleaner/Position.cpp

```
Position::Position(int x, int y, const Orientation& orientation)
  : x(x), y(y), orientation(orientation)
{
}
```

如此设计，将 Orientation 枚举成员的顺序内聚在了一块，包括名称、编号。但它由原来单纯的枚举

类型升级为类，将获得更大弹性的设计空间。

8.1.3 搬迁职责

将 Position 中关于 orientation 的一些逻辑搬迁至 Orientation 类中去，

搬迁 TurnTo

将 TurnTo 的逻辑从 Position.cpp 搬迁过来，从而完全地将 Orientation 设计为值对象。此处用了一个小技巧，通过 Orientation 的构造函数，将它自动地注册给了 orientations 数组中。

示例代码 8-8 src/robot-cleaner/Orientation.cpp

```
namespace
{
    Orientation* orientations[4] = { nullptr };

    inline int numOfTurnTo(bool left)
    { return left ? 3 : 1; }
}

Orientation& Orientation::turnTo(bool left) const
{
    return *orientations[(order + numOfTurnTo(left)) % 4];
}

inline Orientation::Orientation(int order)
  : order(order)
{
    orientations[order] = this;
}

#define DEF_ORIENTATION(orientation, order) \
const Orientation Orientation::orientation(order);

DEF_ORIENTATION(east,  0)
DEF_ORIENTATION(south, 1)
DEF_ORIENTATION(west,  2)
DEF_ORIENTATION(north, 3)
```

保证测试通过后，再将 Position 中的 TurnTo 的逻辑委托给 Orientation 类，然后将 Position.cpp 冗余的代码删除掉，确保测试通过。

示例代码 8-9 src/robot-cleaner/Position.cpp

```
namespace
{
    Orientation orientations[] = { EAST, SOUTH, WEST, NORTH };

    inline int numOfTurnTo(bool left)
    { return left ? 3 : 1; }
}

void Position::turnTo(bool left)
{
    orientation = orientations[(orientation + numOfTurnTo(left))
      % 4];
}
```

示例代码 8-10 src/robot-cleaner/Position.cpp

```
void Position::turnTo(bool left)
{
    orientation = orientation.turnTo(left);
}
```

测试通过，重构完毕。

搬迁 MoveOn

先为 Orientation 对象增加 offset 字段，当执行 forward 时，用于表示在 X 轴移动的单位数目。

示例代码 8-11 src/robot-cleaner/Orientation.cpp

```
#include "robot-cleaner/Orientation.h"

namespace
{
    Orientation* orientations[4] = { nullptr };
}

inline Orientation::Orientation(int order)
  : order(order)
{
    orientations[order] = this;
}

#define DEF_ORIENTATION(orientation, order) \
const Orientation Orientation::orientation(order);

DEF_ORIENTATION(east,  0)
DEF_ORIENTATION(south, 1)
DEF_ORIENTATION(west,  2)
DEF_ORIENTATION(north, 3)
```

示例代码 8-12 src/robot-cleaner/Orientation.cpp

```
#include "robot-cleaner/Orientation.h"

namespace
{
    Orientation* orientations[4] = { nullptr };
}

inline Orientation::Orientation(int order, int offset)
  : order(order), offset(offset)
{
    orientations[order] = this;
}

#define DEF_ORIENTATION(orientation, order, offset) \
const Orientation Orientation::orientation(order, offset);

DEF_ORIENTATION(east,  0, 1)
DEF_ORIENTATION(south, 1, 0)
DEF_ORIENTATION(west,  2, -1)
DEF_ORIENTATION(north, 3, 0)
```

将 MoveOn 的逻辑从 Position 搬迁到 Orientation 中来，快速通过测试用例。

示例代码 8-13 src/robot-cleaner/Orientation.cpp

```
inline int Orientation::diff(int order) const
{
    return orientations[order]->offset;
}

inline int Orientation::xOffset() const
{
    return diff(order);
}

inline int Orientation::yOffset() const
{
    return diff(3-order);
}

void Orientation::moveOn(int& x, int& y, int step) const
{
    x += step*xOffset();
    y += step*yOffset();
}
```

其中，Orientation::moveOn 表示将 (x, y) 沿着该方向移动 step。

测试通过后，将 Position::moveOn 委托给 Orientation::moveOn，确保测试通过后，删除 Position 中无用的代码。

示例代码 8-14 src/robot-cleaner/Position.cpp

```
namespace
{
    const int offsets[] = { 1, 0, -1, 0 };
}

inline int Position::xOffset()
{
    return offsets[orientation];
}

inline int Position::yOffset()
{
    return offsets[3 - orientation];
}

void Position::moveOn(int step)
{
    x += step*xOffset();
    y += step*yOffset();
}
```

示例代码 8-15 src/robot-cleaner/Position.cpp

```
void Position::moveOn(int step)
{
    orientation.moveOn(x, y, step);
}
```

测试通过，重构完毕。最后将 Orientation::operator int 删除，将其完全升级为安全的枚举类型，确保测试通过。

抽取 Point

设计 Orientation::moveOn 时，传递了基本数据类型 (x, y)，这显然是不明智的，需要进一步处理。通过分析，其实 (x, y) 代表的是 Robot 的坐标信息，在此提取一个 Point 概念。

先建立了 Point 的概念，暂时将成员变量公开，以便测试尽快通过。

示例代码 8-16 include/robot-cleaner/Point.h

```
#ifndef H65A54D80_942C_4AB0_846B_A0568EA5200D
#define H65A54D80_942C_4AB0_846B_A0568EA5200D

#include "base/EqHelper.h"

sturct Point
{
    Point(int x, int y);

    DECL_EQUALS(Point);

    int x;
    int y;
};

#endif
```

示例代码 8-17 src/robot-cleaner/Point.cpp

```
#include "robot-cleaner/Point.cpp"

Point::Point(int x, int y)
  : x(x), y(y)
{}

DEF_EQUALS(Point)
{
    return FIELD_EQ(x) && FIELD_EQ(y);
}
```

测试通过后，将 Point 的概念加入到 Position 类中，让所有测试运行通过。

示例代码 8-18 src/robot-cleaner/Position.cpp

```
Position::Position(int x, int y, const Orientation& orientation)
  : x(x), y(y), orientation(orientation)
{}

void Position::moveOn(int step)
{
    orientation.moveOn(x, y, step);
}

DEF_EQUALS(Position)
{
    return FIELD_EQ(x) && FIELD_EQ(y) && FIELD_EQ(orientation);
}
```

示例代码 8-19 src/robot-cleaner/Position.cpp

```
Position::Position(int x, int y, const Orientation& orientation)
  : point(x, y), orientation(orientation)
{}

void Position::moveOn(int step)
{
    orientation.moveOn(point, step);
}

DEF_EQUALS(Position)
{
    return FIELD_EQ(point) && FIELD_EQ(orientation);
}
```

其中，orientation.moveOn(point, step) 表示：将 point 沿着 orientation 方向移动 step 步，但表达力明显下降。如果换成 point.moveOn(step, orientation)，则更能体现这个含义。暂时这样处理，接下来再做进一步处理。

接下来，为了修正编译错误，尽快恢复测试，再将 Orientation::moveOn 的入参换为 Point 类型。

示例代码 8-20 src/robot-cleaner/Orientation.cpp

```
void Orientation::moveOn(int& x, int& y, int step) const
{
    x += step*xOffset();
    y += step*yOffset();
}
```

示例代码 8-21 src/robot-cleaner/Orientation.cpp

```
void Orientation::moveOn(Point& point, int step) const
{
    point.x += step*xOffset();
    point.y += step*yOffset();
}
```

测试通过后，再将 Orientation::moveOn 中运算搬迁到 Point::move 中去。

示例代码 8-22 src/robot-cleaner/Orientation.cpp

```
void Orientation::moveOn(Point& point, int step) const
{
    point.x += step*xOffset();
    point.y += step*yOffset();
}
```

示例代码 8-23 src/robot-cleaner/Orientation.cpp

```
void Orientation::moveOn(Point& point, int step) const
{
    point.move(step, xOffset(), yOffset());
}
```

而 Point 增加 move 接口，完成最后的偏移运算。

示例代码 8-24 src/robot-cleaner/Point.cpp

```
void Point::move(int step, int xOffset, int yOffset)
{
    x += step*xOffset;
    y += step*yOffset;
}
```

测试通过后，再将 Point::x, Point::y 隐藏起来，保证测试通过。

示例代码 8-25 include/robot-cleaner/Point.h

```
#ifndef H65A54D80_942C_4AB0_846B_A0568EA5200D
#define H65A54D80_942C_4AB0_846B_A0568EA5200D

#include "base/EqHelper.h"

struct Point
{
    Point(int x, int y);

    void move(int xOffset, int yOffset);

    __DECL_EQUALS(Point);

    int x;
    int y;
};

#endif
```

示例代码 8-26 include/robot-cleaner/Point.h

```
#ifndef H65A54D80_942C_4AB0_846B_A0568EA5200D
#define H65A54D80_942C_4AB0_846B_A0568EA5200D

#include "base/EqHelper.h"

struct Point
{
    Point(int x, int y);

    void move(int xOffset, int yOffset);

    __DECL_EQUALS(Point);

private:
    int x, y;
};

#endif
```

测试通过了，此刻彻底消除了“方向顺序”的知识在三个地方存在三种表示的重复设计，将三张表规整为一张表。不仅在物理位置上的统一，也将归属于自身的职责也搬迁过来了，知识更加内聚。

示例代码 8-27 src/robot-cleaner/Orientation.cpp

```
DEF_ORIENTATION(east,  0, 1)
DEF_ORIENTATION(south, 1, 0)
DEF_ORIENTATION(west,  2, -1)
DEF_ORIENTATION(north, 3, 0)
```

8.1.4 依恋情节

至此，还存在几个明显的坏味道有待进一步改善。

1. 从职责归属关系看，让 Position 承担 turnTo, moveOn 的语义有些晦涩；

2. Position 与 Orientation 之间存在不必要的依赖关系，两者因为增加新的指令都不能独立变化，未能做到彻底的修改的封闭性。

3. Instruction 的 turnTo，moveOn 实现具有副作用。

TDA

首先，修改 Instruction 的接口。

示例代码 8-28 include/robot-cleaner/Instruction.h

```
struct Position;

DEFINE_ROLE(Instruction)
{
    ABSTRACT(void exec(Position&));
};
```

示例代码 8-29 include/robot-cleaner/Instruction.h

```
struct Point;
struct Orientation;

DEFINE_ROLE(Instruction)
{
    ABSTRACT(void exec(Point&, Orientation&));
};
```

先修改 RobotCleaner::exec 的实现。

示例代码 8-30 src/robot-cleaner/RobotCleaner.cpp

```
void RobotCleaner::exec(Instruction* instruction)
{
    instruction->exec(position);
    delete instruction;
}
```

示例代码 8-31 src/robot-cleaner/RobotCleaner.cpp

```
void RobotCleaner::exec(Instruction* instruction)
{
    instruction->exec(point.getPoint(),
      orientation.getOrientation());
    delete instruction;
}
```

最后删除 Position 的 turnTo, moveOn 接口，并为其增加新的接口。

示例代码 8-32 src/robot-cleaner/Position.cpp

```
Point& Position::getPoint()
{
    return point;
}

Orientation& Position::getOrientation()
{
    return orientation;
}
```

接下来，修改和实现 Instruction 所有的子类实现。

示例代码 8-33 src/robot-cleaner/Instruction.cpp

```
namespace
{
    struct EmptyInstruction : Instruction
    {
        OVERRIDE(void exec(Position& position)) {}
    };

    struct TurnToInstruction : Instruction
    {
        explicit TurnToInstruction(bool left)
          : left(left)
        {}

    private:
        OVERRIDE(void exec(Position& position))
        {
            position.turnTo(left);
        }

    private:
        bool left;
    };

    struct MoveOnInstruction : Instruction
    {
        explicit MoveOnInstruction(bool forward, int n)
          : step(prefix(forward)*n)
        {}

    private:
        OVERRIDE(void exec(Position& position))
        {
            position.moveOn(step);
        }

    private:
        static int prefix(bool forward)
        {
            return forward ? 1 : -1;
        }

    private:
        int step;
    };

    struct RepeatedInstruction : Instruction
    {
        RepeatedInstruction(Instruction* instruction, int n)
         : instruction(instruction), n(n)
        {}

    private:
        OVERRIDE(void exec(Position& position))
        {
            for (auto i=0; i<n; i++)
            {
                instruction->exec(position);
            }
        }

    private:
        std::unique_ptr<Instruction> instruction;
        int n;
    };

    struct SequentialInstruction : Instruction
    {
        SequentialInstruction(std::initializer_list<Instruction*>
          instructions)
         : instructions(instructions)
        {}

        ~SequentialInstruction()
        {
            for(auto instruction : instructions)
            {
                delete instruction;
            }
        }

    private:
        OVERRIDE(void exec(Position& position))
        {
            for(auto instruction : instructions)
            {
                instruction->exec(position);
            }
        }

    private:
        std::vector<Instruction*> instructions;
    };
}
```

示例代码 8-34 src/robot-cleaner/Instruction.cpp

```
namespace
{
    struct EmptyInstruction : Instruction
    {
        OVERRIDE(void exec(Point&, Orientation&)) {}
    };

    struct TurnToInstruction : Instruction
    {
        explicit TurnToInstruction(bool left)
          : left(left)
        {}

    private:
        OVERRIDE(void exec(Point&, Orientation& orientation))
        {
            orientation = orientation.turnTo(left);
        }

    private:
        bool left;
    };

    struct MoveOnInstruction : Instruction
    {
        explicit MoveOnInstruction(bool forward, int n)
          : step(prefix(forward)*n)
        {}

    private:
        OVERRIDE(void exec(Point& point, Orientation&
          orientation))
        {
            orientation.moveOn(point, step);
        }

    private:
        static int prefix(bool forward)
        {
            return forward ? 1 : -1;
        }

    private:
        int step;
    };

    struct RepeatedInstruction : Instruction
    {
        RepeatedInstruction(Instruction* instruction, int n)
         : instruction(instruction), n(n)
        {}

    private:
        OVERRIDE(void exec(Point& point, Orientation&
          orientation))
        {
            for (auto i=0; i<n; i++)
            {
                instruction->exec(point, orientation);
            }
        }

    private:
        std::unique_ptr<Instruction> instruction;
        int n;
    };

    struct SequentialInstruction : Instruction
    {
        SequentialInstruction(std::initializer_list<Instruction*>
          instructions)
         : instructions(instructions)
        {}

        ~SequentialInstruction()
        {
            for(auto instruction : instructions)
            {
                delete instruction;
            }
        }

    private:
        OVERRIDE(void exec(Point& point, Orientation&
          orientation))
        {
            for(auto instruction : instructions)
            {
                instruction->exec(point, orientation);
            }
        }

    private:
        std::vector<Instruction*> instructions;
    };
}
```

最后删除 Position 的 turnTo, moveOn 接口，保证测试通过。

测试通过了，此刻 Position 与 Orientation 之间，及其 Position 与新增 Instruction 之间的耦合度被

降低了；另外，Position 也不承担晦涩的 turnTo, moveOn 的职责。

这是 TDA, Tell, Don't Ask 原则一个具体运用，但也为此付出了另外的代价：Position 不得不提供两个 get 接口，而且是可修改的引用类型。

改善表达力

MoveInstruction 实现 moveOn 时，委托给了 Orientation，让 Orientation 承担 moveOn 的职责有些晦涩，应该将两者的语义换一下。

示例代码 8-35 src/robot-cleaner/Instruction.cpp

```
struct MoveOnInstruction : Instruction
{
    explicit MoveOnInstruction(bool forward, int n)
      : step(prefix(forward)*n)
    {}

private:
    OVERRIDE(void exec(Point& point, Orientation& orientation))
    {
        orientation.moveOn(point, step);
    }

private:
    static int prefix(bool forward)
    {
        return forward ? 1 : -1;
    }

private:
    int step;
};
```

示例代码 8-36 src/robot-cleaner/Instruction.cpp

```
struct MoveOnInstruction : Instruction
{
    explicit MoveOnInstruction(bool forward, int n)
      : step(prefix(forward)*n)
    {}

private:
    OVERRIDE(void exec(Point& point, Orientation& orientation))
    {
        point = point.moveOn(step, orientation);
    }

private:
    static int prefix(bool forward)
    {
        return forward ? 1 : -1;
    }

private:
    int step;
};
```

point.moveOn 表示 point 沿着 orientation 方向移动了 step 步后到了一个新的位置。

然后将 Point::move 重构为 Point::moveOn，保证测试通过。

示例代码 8-37 src/robot-cleaner/Point.cpp

```
void Point::move(int step, int xOffset, int yOffset)
{
    x += step*xOffset;
    y += step*yOffset;
}
```

示例代码 8-38 src/robot-cleaner/Point.cpp

```
Point Point::moveOn(int step, const Orientation& orientation)
{
    return Point( x+step*orientation.getXOffset()
                , y+step*orientation.getYOffset());
}
```

但需要暴露 Orientation::getXOffset，Orientation::getYOffset。这不是一个很好的设计，可以将 Point::x, Point::y 传递给 Orientation，根据 Orientation 自身所处的方向，将 x, y 向前移动 step，并返回一个新的 Point 位置。

示例代码 8-39 src/robot-cleaner/Point.cpp

```
Point Point::moveOn(int step, const Orientation& orientation)
{
    return Point( x+step*orientation.getXOffset()
                , y+step*orientation.getYOffset());
}
```

示例代码 8-40 src/robot-cleaner/Point.cpp

```
Point Point::moveOn(int step, const Orientation& orientation)
{
    return orientation.moveOn(x, y, step);
}
```

最后重构实现 Orientation::moveOn 的逻辑。

示例代码 8-41 src/robot-cleaner/Orientation.cpp

```
void Orientation::moveOn(Point& point, int step) const
{
    point.move(step, xOffset(), yOffset());
}
```

示例代码 8-42 src/robot-cleaner/Orientation.cpp

```
Point Orientation::moveOn(int x, int y, int step) const
{
    return Point(x+step*xOffset(), y+step*yOffset());
}
```

测试通过了。接下来将 Position 提供的两个 get 接口消除掉，使用多重继承更好可以表达它们之间的聚合关系。

示例代码 8-43 src/robot-cleaner/Orientation.cpp

```
#include "robot-cleaner/Point.h"
#include "robot-cleaner/Orientation.h"

struct Position
{
    Position(int x, int y, const Orientation& orientation);

    DECL_EQUALS(Position);

    Point& getPoint();
    Orientation& getOrientation();

private:
    Point point;
    Orientation orientation;
};
```

示例代码 8-44 src/robot-cleaner/Orientation.cpp

```
#include "robot-cleaner/Point.h"
#include "robot-cleaner/Orientation.h"

struct Position : Point, Orientation
{
    Position(int x, int y, const Orientation& orientation);

    DECL_EQUALS(Position);
};
```

先修改 RobotCleaner::exec 的实现。

示例代码 8-45 src/robot-cleaner/RobotCleaner.cpp

```
void RobotCleaner::exec(Instruction* instruction)
{
    instruction->exec(point.getPoint(),
      orientation.getOrientation());
    delete instruction;
}
```

示例代码 8-46 src/robot-cleaner/RobotCleaner.cpp

```
void RobotCleaner::exec(Instruction* instruction)
{
    instruction->exec(SELF(position, Point), SELF(position,
      Position));
    delete instruction;
}
```

测试通过了，重构完毕。

8.2 最后的话

首先看 Instruction 的设计，它将指令执行的算法封装起来，独立地随这指令集变化而变化；并且提供了工厂方法，对外提供 DSL，将它的子类实现全部隐藏到了实现文件中，对外暴露的只有 Instruction 的抽象接口，在提高表达力的同时，增强了信息隐藏的机制。

示例代码 8-47 test/robot-cleaner/TestRobotCleaner.h

```
#ifndef HC37C3D94_43F1_4677_BD56_34CB78EFEC75
#define HC37C3D94_43F1_4677_BD56_34CB78EFEC75

#include "infra/base/Role.h"
#include <initializer_list>

struct Point;
struct Orientation;

DEFINE_ROLE(Instruction)
{
    ABSTRACT(void exec(Point& point, Orientation& orientation) const);
};

Instruction* left();
Instruction* right();

Instruction* forward_n(int n);
Instruction* backward_n(int n);
Instruction* repeat(Instruction*, int n);
```

```
inline Instruction* forward()
{ return forward_n(1);  }

inline Instruction* backward()
{ return backward_n(1); }

inline Instruction* round()
{ return repeat(right(), 2); }

Instruction* __sequential(std::initializer_list<Instruction*>);

#define sequential(...) __sequential({ __VA_ARGS__ })

#endif
```

再来看 Position 与 Point，Orientation 之间的关系。Position 是由 Point 和 Orientation 聚合而成的。Point 具有 MoveOn 的语义，而 Orientation 具有 TurnTo 的语义；另外，Point::moveOn 时，是以 Orientation 的方向来决定该怎么移动的，所以 Orientation 将协作 Point 完成 MoveOn 的移动过程。

接着看 Instruction 与 Point,Orientation 的关系。TurnToInstruction 将于 Orientation 交互完成转向；MoveOnInstruction 将与 Point 交互，在 Orientation 的协作下完成移动。

需要强调的是，Instruction 与 Position 没有任何关系，它在领域内只是对 Point，Orientation 聚合而已。

最后重要的是，TurnLeft 与 TurnRight 之间，Forward 与 Backward 之间，犹如正反两方面。如果存在 TurnLeftInstrction, TurnRightInstructin 两个概念，并且它们的逻辑实现没有依赖一个更本质的抽象，则它们之间是存在重复设计的。

我们将 TurnLeftInstrction, TurnRightInstructin 两个概念合并为 TurnToInstruction，而将 ForwardInstrction, BackwardInstructin 两个概念合并为 MoveOnInstruction，采用了简单 bool 接口消除它们逻辑上的重复。

当然，此处 bool 接口的使用没有影响用户友好性，因为用户使用的是抽象了的、友好性的 DSL 接口，例如 left() 或者 right() 指令；另外，调用字面值 true/false 时，与定义实现并不远，也不存在晦涩难懂，需要用户跳转代码等难题。