第9章
健全性

この章の目的

◆これまで紹介してきた悪しき構造以外の様々な悪しきコードを知る

◆基本的に守る必要が多く、重要な内容である

内容

内容とキーワード

◆デッドコード
◆YAGNI原則
◆マジックナンバー
◆文字列型執着
◆グローバル変数
◆null問題
◆例外の握りつぶし
◆メタプログラミング
◆技術駆動パッケージング

◆サンプルコードのコピペ
◆銀の弾丸

デッドコード

デッドコード

どんな条件であっても決して実行されないコードをデッドコードもしくは到達不能コードと呼ぶ。

```
if (level > 99) {
    level = 99;
}

// 中略

if (level == 1) {
    intHitPoint();
    intMagicPoint();
    intEquipments();
} else if (level == 100) {
    addSqecialAbility();
}
```

levelは最大99に設定している

levelは最大99なので
このコードは絶対実行されない！

何だこのコード？
実行されることあるのか？
何の意味があるの？

デッドコードは読み手を混乱させるだけで無く、仕様変更により急に到達可能になり意図しない挙動をすることがある

デッドコード

デッドコードは見つけ次第削除しよう。

◆デッドコードは可読性を下げる

◆デッドコードは急に蘇る（まるでゾンビ）

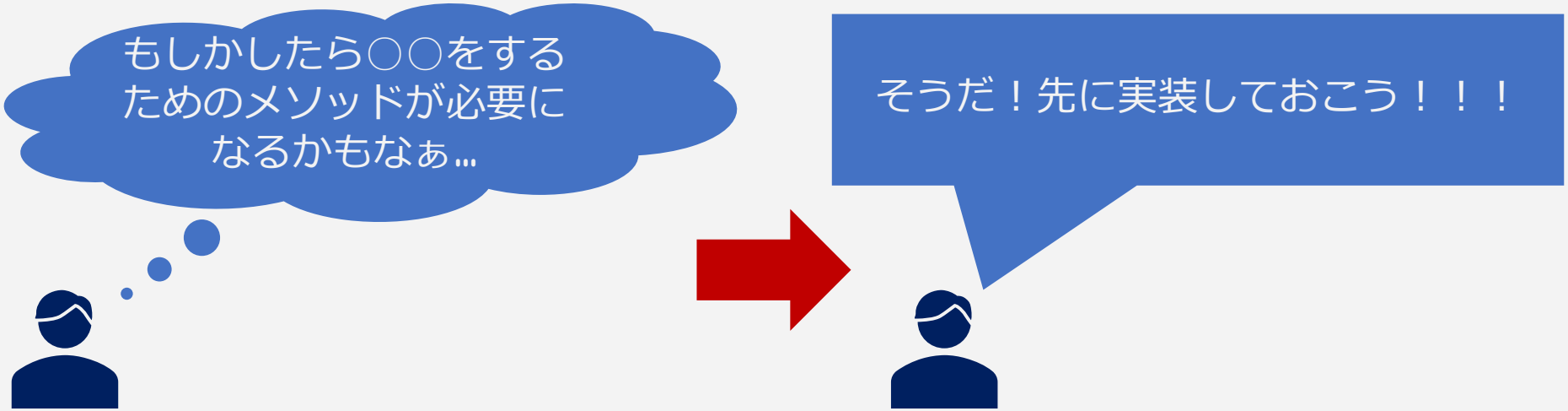
◆デッドコードは削除しよう

■せっかく書いたコードを消すことに不安があるかも知れないが、GitHub等のGitツールを利用すればその心配は無い。

YAGNI原則

YAGNI原則

◆こんなことを考えてプログラムを書いてませんか？



先読みしてコードを書いてしまう

YAGNI原則

YAGNI原則に則って今必要の無いコードは書かないようにしよう。

◆先回りして書いたコードは現実はほとんど使われない上にバグの原因にもなる。

◆YAGNI原則

- ■「You aren't going to need it.」の略で、「必要ないでしょう」と訳される。
- ■実際に必要になったときのみ実装せよという意味。
- ■使われないコードはデッドコードと同じ。

◆今必要な機能だけを作り構造をシンプルにしよう。

マジックナンバー

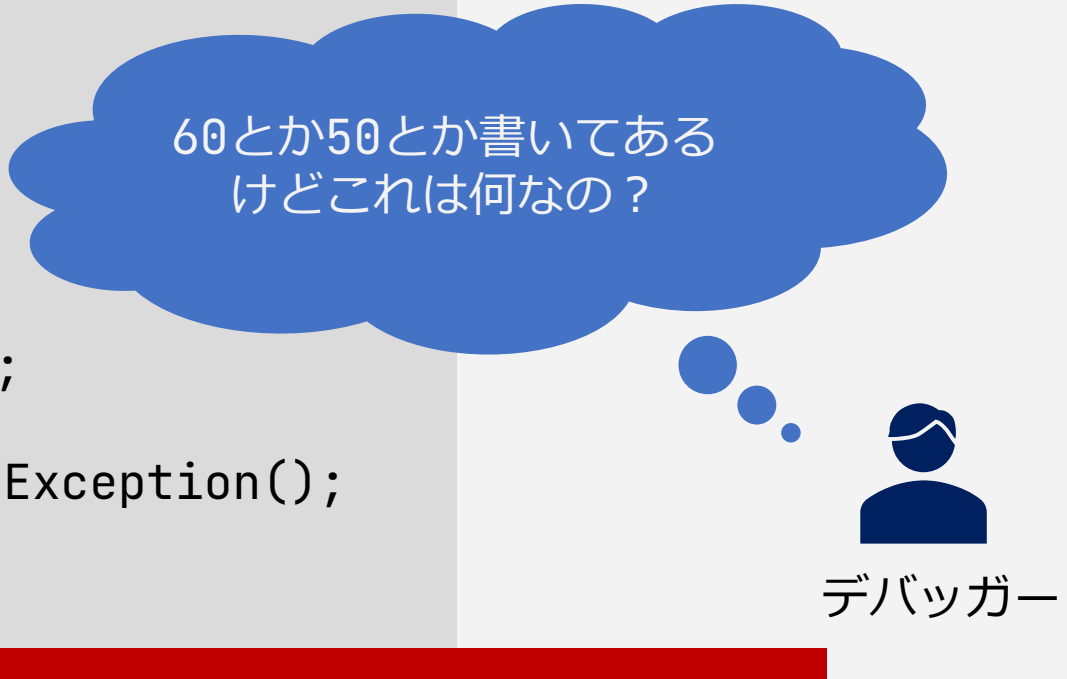
マジックナンバー

ロジック内に直接書き込まれている数値をマジックナンバーと呼び、意図不明な場合が多い。

以下はwebコミックサービスを実装する際のコードだが…

```
class ComicManager {
    // 中略
    boolean isOk() {
        return 60 <= value;
    }

    void tryConsume() {
        int tmp = value - 50;
        if (tmp < 0) {
            throw new RuntimeException();
        }
        value = tmp;
    }
}
```



数値そのままだと意図が伝わらない

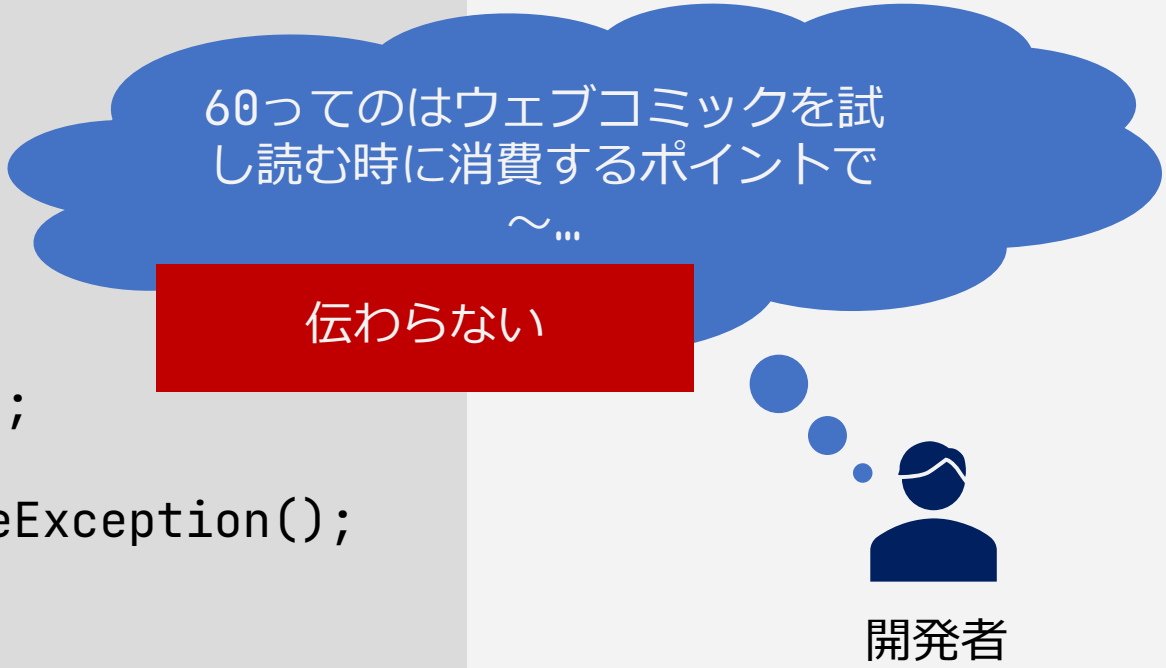
マジックナンバー

ロジック内に直接書き込まれている数値をマジックナンバーと呼び、意図不明な場合が多い。

以下はwebコミックサービスを実装する際のコードだが…

```
class ComicManager {
    // 中略
    boolean isOk() {
        return 60 <= value;
    }

    void tryConsume() {
        int tmp = value - 50;
        if (tmp < 0) {
            throw new RuntimeException();
        }
        value = tmp;
    }
}
```



マジックナンバー

マジックナンバーの代わりに定数を用意しましょう。

以下はwebコミックサービスを実装する際のコードだが…

```
class ReadingPoint {
    private static final int MIN = 0;
    private static final int TRAIL_READING_POINT = 60;
    final int value;

    ReadingPoint(final int value) {
        if (value < MIN) {
            throw new IlligalArgumentException();
        }
        this.value = value;
    }

    //  省略
}
```

適切な名前をつければ
意図が伝わる

意図が伝わるように定数を定義しよう

文字列執着型

文字列型執着

◆単一の文字列にコンマなどを使って数値を区切り格納しているものを見かけることがある。

```
// ラベル文字列、色表示（RGB）、上限文字数
String title = “タイトル,255,250,240,64”
// 以降でsplitメソッド等を使用してデータを切り出す
```

◆読み込んだCSVからデータを取り出すためにSplitメソッドを使う場合があるが、そういった意図がないのに文字列型変数に意味が異なる変数を詰め込むのは可読性が下がり良くない。

意図が伝わるようにそれぞれの変数を定義しよう

グローバル変数

グローバル変数

どこからでも参照可能な変数をグローバル変数と呼ぶ。これらの変数は不必要に影響範囲が広い。

◆可変なグローバル変数はどこからでも参照可能で、操作可能な変数であるため、どのタイミングで値が書き換わったのか把握することは非常に困難。

◆非常に多くのロジックに関わる場合、排他的制御を慎重に設計する必要があり、その結果パフォーマンスが落ちる。

◆Javaにはグローバル変数は無いが、`public static`で変数を宣言すると、実質グローバル変数である。

影響範囲を必要最低限にしよう！

Null問題

Null問題

nullは何もない状態を表すのに使われていることがある。

```
class Member {
    private Equipment head;
    private Equipment body;
    private Equipment arm;
    private int defense;

    int totalDefense {
        int total = defense;
        total += head.defense;
        total += body.defense;
        return total;
    }
    void takeOffAllEquipment {
        head = null;
        body = null;
        arm = null;
    }

}
```

何も装備していない = null

Null問題

null前提だとあちこちでnullかどうかのチェックが必要になる。

```
void showBodyEquipment() {
    showParam(body.name);
    showParam(body.defence);
    showParam(body.magicDefence);
}
```

NullPointerException
が発生することがある

あちこちでtry-catchによる例外処理
または
nullチェックが必要になってしまう

Null問題

◆Nullが入り込む前提でロジックを作成すると…
- ■あちこちでNullチェックを行い、コードが増える
- ■コードが増えた影響で、可読性が下がる
- ■Nullチェックが漏れるとバグにになる

◆そもそもNullって何？
- ■未初期化状態のメモリに対するアクセスを防ぐしくみ
- ■メモリアクセストラブルを防止するものでしかない
- ■Null自体は無効な扱い

Nullを返さない、渡さない

◆何も無い状態は、立派な状態のひとつです

■「何も無い」という状態を作成しよう

```
void takeOffAllEquipments() {
    head = Equipment.EMPTY
    body = Equipment.EMPTY
    arm = Equipment.EMPTY
}
```

◆上のコードはEMPTYという変数を用意している

Null安全

Null安全とは、Nullが原因のエラーを発生させないしくみである。

◆そもそもKotlinなどはNullが代入できない

■これをNull非許容型と呼ぶ

```
val name: String = null
```

コンパイルエラー！

例外の握りつぶし

例外の握りつぶし

例外処理を適当にしてしまうと、バグの発生源が分からなくなり、意図しないエラーを引き起こす原因になる。

◆catch文に何も書かないと…

```
try {
    reservations.add(product);
}
catch (Exception e) {
}
```

何もしていない（握りつぶしている）

危険！

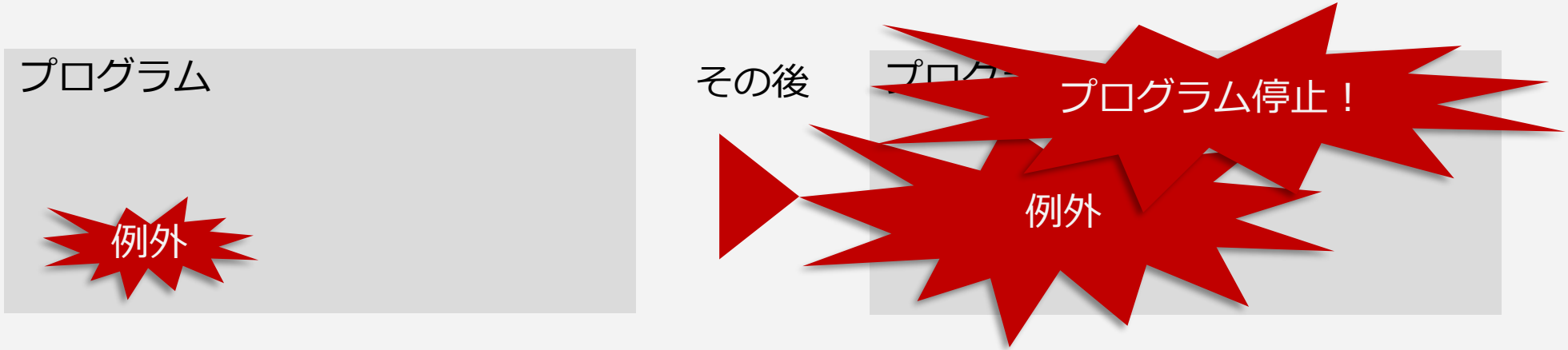
どこで例外が発生したか、分からなくなる

例外の握りつぶし

例外処理を適当にしてしまうと、バグの発生源が分からなくなり、意図しないエラーを引き起こす原因になる。

◆最初は問題ないこともあるが…

■時間が経つにつれて、影響が拡大することも



◆例外は検出した際にすぐに気づけるようにしよう

```
catch (Exception e) {
    reportError(e);
    requestNotifyError(“エラー詳細”)
}
```

技術行動パッケージング

サンプルコードのコピペ

サンプルコードのコピペ

よくありがちなサンプルコードのコピペ。

◆Web上には様々なサンプルコードが載っているが…
- ■多くは技術を理解するためのもである
- ■そのためそそまま写してしまうと設計上良くない

◆サンプルコードはあくまで理解を深めるためのもの
- ■サンプルコードのコピペはやめる
- ■あるべきクラス構造を自ら設計する

銀の弾丸

◆新しい技術を知ったとき、すぐに使いたくなるが…

■新しい技術は画期的に見える

■複雑な問題も解決してくれそうに思えるが…

■闇雲に使おうとするとかえって問題が複雑になることも

◆特効薬的な表現で「銀の弾丸」というものがあるが

■ソフトウェアには銀の弾丸は無い

■課題と目的に応じて

次の章に向けて