Technologic Arts Incorporated

ケント・ベックの実装パターン

（株）テクノロジックアート
長瀬　嘉秀

Technologic Arts Incorporated

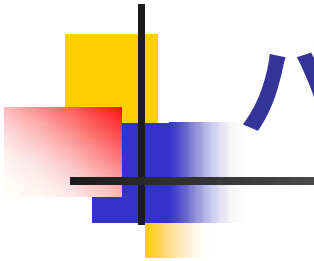
実装パターン

- Java
- プログラミングパターン
- 書籍
  - 実装パターン
    - 【著作】 ケント・ベック
    - 【監訳】長瀬嘉秀、永田渉
    - 【翻訳】株式会社テクノロジックアート
    - 【出版】 ピアソン・エデュケーション
    - 【ISBN】 4-89471-287-4

Technologic Arts Incorporated

本日説明するパターン

- クラス
  - バリューオブジェクト
  - サブクラス
  - 条件分岐
  - 委譲
- 状態
  - 引数
  - コレクティングパラメータ
  - パラメータオブジェクト
  - 初期化
- 振る舞い
  - 反転メッセージ
  - 説明メッセージ
  - ガード条件



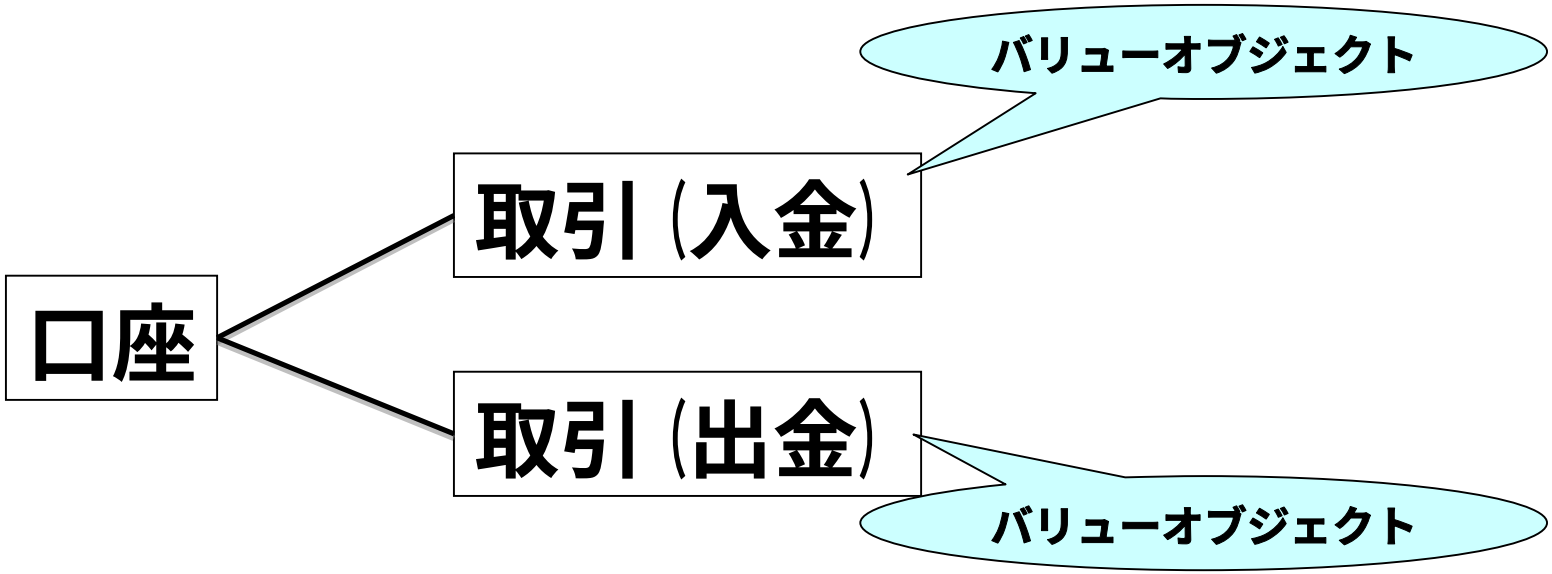
バリューオブジェクト

- 整数のように振る舞うオブジェクト
- 変化する状態ではない

Technologic Arts Incorporated

バリューオブジェクトの例

- 「口座」オブジェクトに関連する「取引」オブジェクト



Technologic Arts Incorporated

バリューオブジェクトの特徴

- オブジェクトの操作で、つねに新しいオブジェクトが返される

バリューオブジェクト

```
Bounds = bouds.translateBy(10, 20);
```

非バリューオブジェクト

```
bouds.translateBy(10, 20);
```

バリューオブジェクトのメリット／デメリット

Technologic Arts Incorporated

- パフォーマンス　新しいオブジェクトを生成
- バリュー型の考えに不慣れ
- オブジェクトが変化しない部分との間に、境界線を引くのが難しい

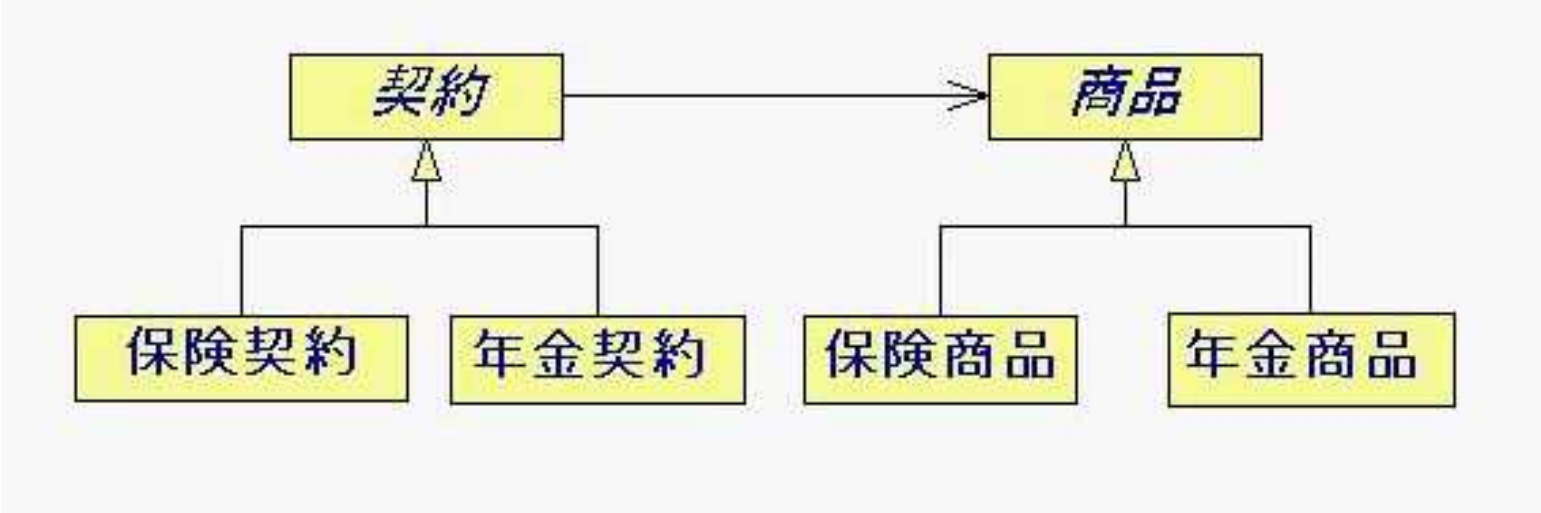
- 不変、可変を分けることでわかりやすく、よい設計のプログラムになる

サブクラス

- 分類
- バリエーション
- 複雑になると分からなくなる
- 継承階層が深くなると分からなくなる

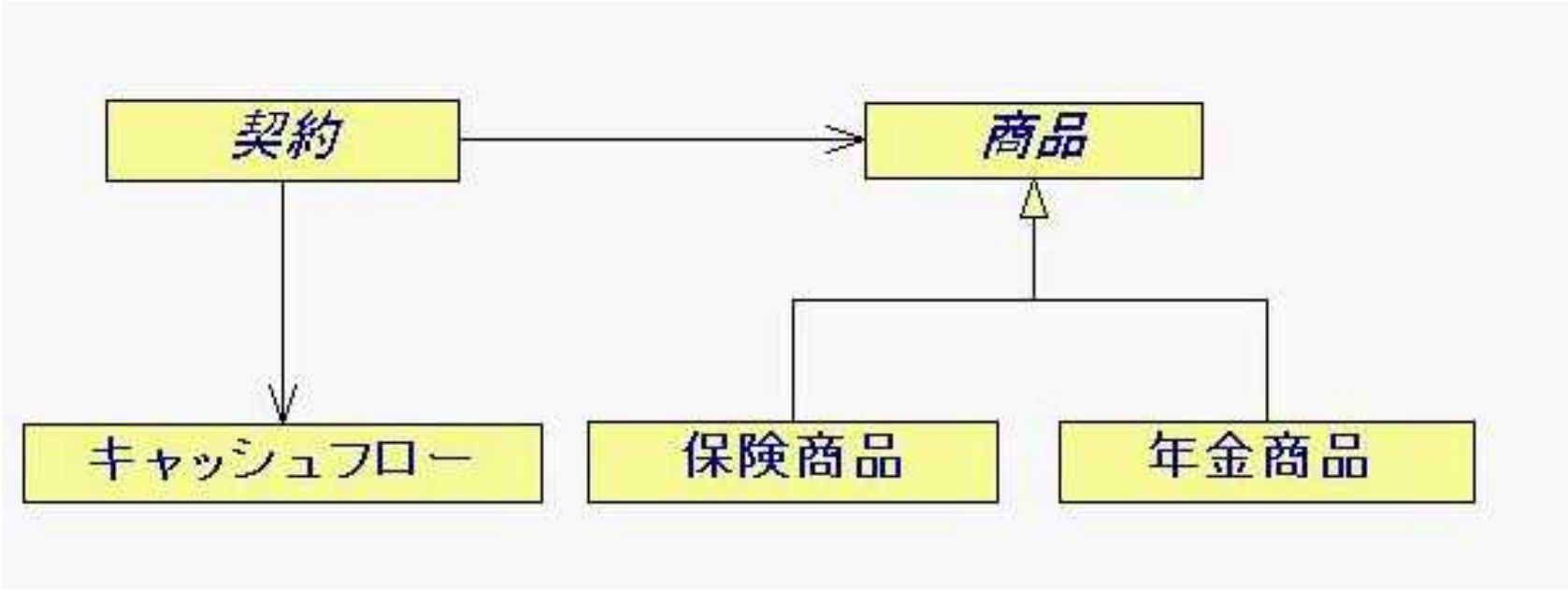
サブクラスの例

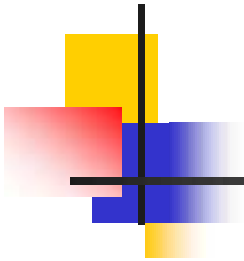
- 保険契約が年金商品を参照できない



解決策

- バリエーションを商品だけにした





サブクラスのコツ

- スーパークラス中のロジックを、１つの仕事だけを行なうメソッドに徹底的に分解する
- １つのメソッドだけをオーバーライドできるようにする
- サブクラスにコピーされたコードを修正すると変更が難しくなる

条件分岐

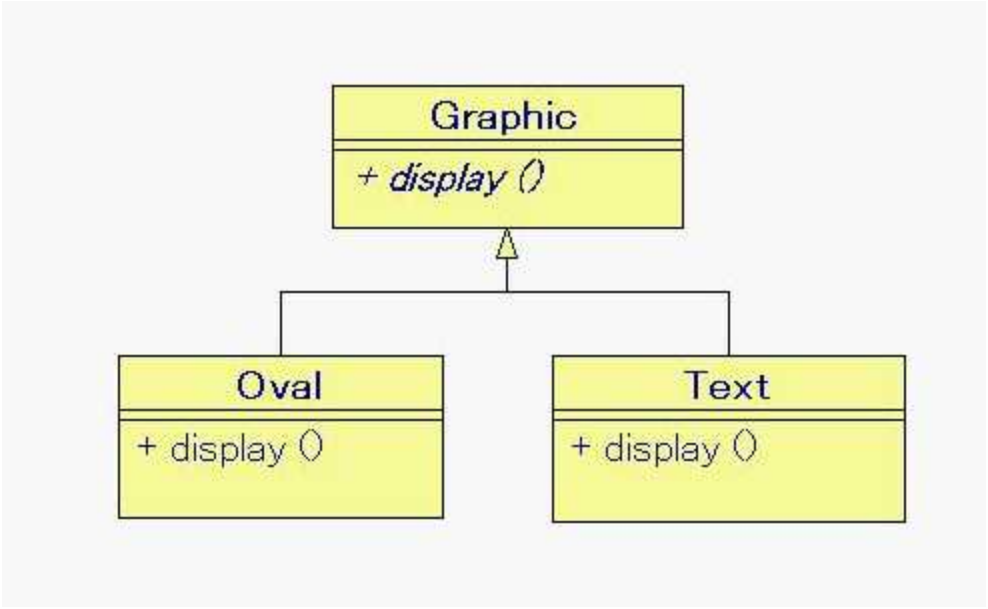
- 条件文が複数のメソッドに出てくる
- 重複

```
public  void display() {
   switch (getType()) [
      case RECTANGLE :
         // …
         break;
      case OVAL :
         // …
         break;
      case TEXT :
         // …
         break;
      default :
         break;
   }
}
```

```
public  boolean contains(Point p) {
   switch (getType()) [
      case RECTANGLE :
         // …
         break;
      case OVAL :
         // …
         break;
      case TEXT :
         // …
         break;
      default :
         break;
   }
}
```

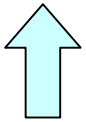
重複の削除

- サブクラス化または委譲




```
Graphic g;

g = new Oval();
g.display();
```

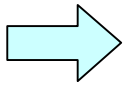


条件文はなくなる

委譲

- 条件分岐の重複削除

```
public  void mouseDown() {
   switch (getTool()) [
      case SELECTING :
         // …
         break;
      case CREATING_RECTANGLE :
         // …
         break;
      case EDITING_TEXT :
         // …
         break;
      default :
         break;
   }
}
```

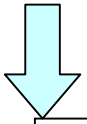


```
public  void mouseDown() {
   getTool(). mouseDown();
}
```

Technologic Arts Incorporated

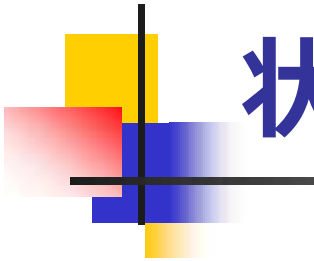
委譲２

```
GraphicEditor
private GraphicEditor editor;
public  void mouseDown() {
   switch (getTool()) [
   case CREATING_RECTANGLE :
         editor.add(new RectangleFigure());
         break;
      case EDITING_TEXT :
         // …
         break;
      default :
         break;
   }
}
```



```
GraphicEditor
public  void mouseDown() {
   tool.mouseDown();
}
```

```
RectangleTool
private GraphicEditor editor;
public RectangleTool(GraphicEditor editor) {
   this.editor = editor;
}
public void mouseDown() {
   editor.add(new RectangleFigure());
}
```



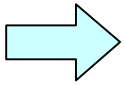
状態に関するパターン

- 引数
- コレクティングパラメータ
- パラメータオブジェクト
- 初期化
  - 早期初期化
  - 遅延初期化

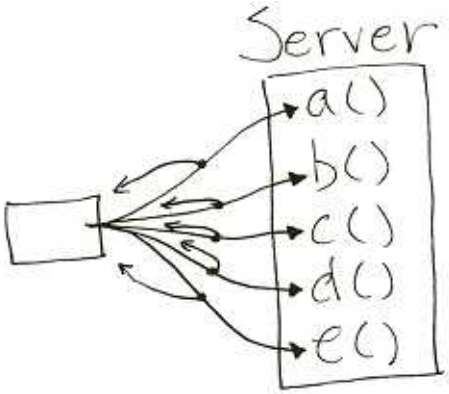
Technologic Arts Incorporated

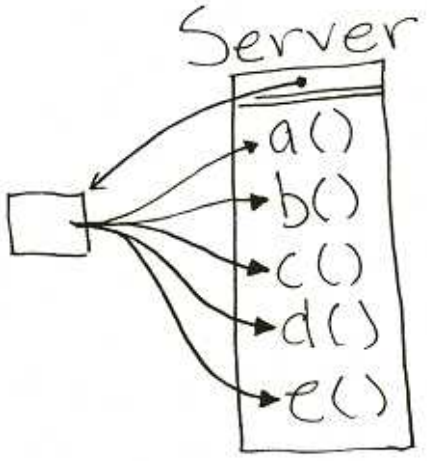
引数

```
Server s = new Server();
s.a(this);
s.b(this);
s.c(this);
s.d(this);
s.e(this);
```



```
Server s = new Server(this);
s.a();
s.b();
s.c();
s.d();
s.e();
```





引数の繰返しによって増加する結合度

参照によって減少する結合度

コレクティングパラメータ

- メソッド呼び出しを複数回行ない、その結果を収集する計算

```
Node
int size() {
    int result = 1;
    for (Node each: getChildren())
      result += each.size();
    return result;
```

シンプルな例

Technologic Arts Incorporated

コレクティングパラメータ２

- 複雑な場合

```
Node
asList() {
    List results = new ArrayList();
    addTo(results);
    return results;
}
```

result

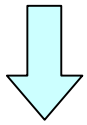
```
addTo(List elements) {
    elements.add(getValue());
    for (Node each: getChildren())
        each.addTo(elements);
}
```

Technologic Arts Incorporated

パラメータオブジェクト

- 同じ引数が複数のメソッドにある場合

```
setOuterBounds(x, y, width, height);
setInnerBounds(x + 2, y + 2, width - 4, height -4);
```



```
setOuterBounds(bounds);
setInnerBounds(bounds.expand(-2));
```

早期初期化

- 宣言時の初期化

```
class Library {
    List<Person> members = new ArrayList<Person>();
    ...
}
```

- コンストラクタでの初期化

```
class Point {
    int x, y;
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

遅延初期化

- リソースが限られている
- 起動時間を早くする
- コードが読みづらくなる

```
Library
Collection<Person> getMembers() {
    if (members == null)
        members = new ArrayList<Person>();
    return members;
}
```

振る舞いに関するパターン

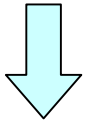
- 反転メッセージ
- 説明メッセージ
- ガード条件

Technologic Arts Incorporated

反転メッセージ

- 対称性により、コード可読性が改善する

```
void compute() {
    input();
    helper.process(this);
    output();
}
```



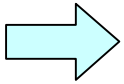
```
void process(Helper helper) {
    helper.process(this);
}
void compute() {
    input();
    process(helper);
    output();
}
```

Technologic Arts Incorporated

反転メッセージ・・・さらに

- 「美学的」要求
- コードに対する美学を養えば、コードから受ける美学的な印象、コードの品質に関する貴重なフィードバックとなる

```
void input(Helper helper) {
    helper.input(this);
}
void output(Helper helper) {
    helper.output(this);
}
```

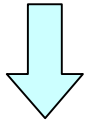


```
void compute() {
    new Helper(this).compute();
}
Helper
void compute() {
    input();
    process();
    output();
}
```

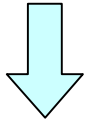
説明メッセージ

- 意図と実装の区別

```
void highlight(Rectangle area) {
    reverse(area);
}
```



```
flags |= LOADED_BIT; // ロード済ビットを立てる
```

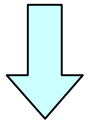


```
setLoadedFlag();
```

```
void setLoadedFlag() {
    flags |= LOADED_BIT;
}
```

ガード条件

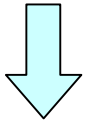
```
void initialize() {
    if (!isIntialized()) {
        ...
    }
}
```



```
void initialize() {
    if (isIntialized())
        return;
    ...
}
```

ガード条件・・・複雑な例

```
void compute() {
    Server server = getServer();
    if (server != null) {
        Client client = server.getClient();
        if (client != null) {
            Request current = client.getRequest();
            if (current != null)
                processRequest(current);
        }
    }
}
```



```
void compute() {
    Server server = getServer();
    if (server == null)
        return
    Client client = server.getClient();
    if (client == null)
        return;
    Request current = client.getRequest();
    if (current == null)
        return;
    processRequest(current);
}
```

ガード条件・・・技術者の常識

- 「各ルーチンは単一の入口と単一の出口を持たねばならぬ」

  FORTRAN、アセンブラ
- Javaではそうとは限らない
  - Javaは小さなメソッド、大部分のローカルなデータで構成されている

```
while (line = reader.readLine()) {
    if (line.startsWith('#') || line.isEmpty()
        continue;
    // 通常の処理がここに入る
}
```