オブジェクト指向設計

2023年度前期 第12回
九州産業大学 理工学部

講義計画と評価方法

第12～14回の計画

第12回	デザインパターン（入門） デザインパターン（適用）
第13回	まとめと模擬テスト
第14回	まとめと期末テスト

評価方法

- 演習とレポート 60%
- 期末テスト 40%

第12回：デザインパターン（入門）

今回の流れ

1. デザインパターン：Façade
2. デザインパターン：AbstractFactory

デザインパターン

デザインパターンの利用

オブジェクト間の共通の相互作用をカタログにしたもの

- 過去に成功した設計の事例を一般化している
- 特に拡張や変更（再利用）に配慮している
- 他の開発者にプログラムを簡潔に説明できる

Design Patterns - Elements of Reusable Object-Oriented Software （1995）

- GoF（Gang of Four） : Erich Gamma, Richard Helm, Ralph Jophnson, John Vlissides
- 23のデザインパターンを紹介している

23のパターン

生成に関するパターン

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton

構造に関するパターン

6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy

振る舞いに関するパターン

13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor

今回はFacadeパターンを
利用する

Façadeパターン

Facadeパターン

ファサード（フランス語）

- 建物を特徴付ける、装飾などが施された正面の構造

目的

- 既存システムの使用方法を簡素化したい
- 独自のインタフェースを提供する必要がある

問題

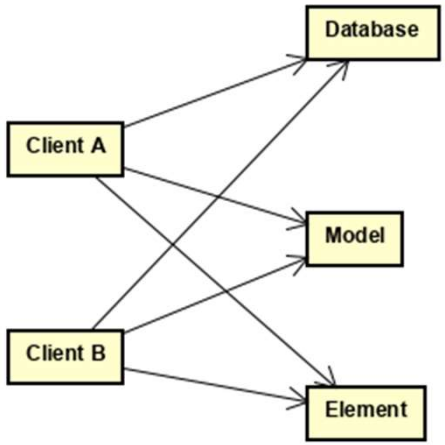
- 複雑なシステムの一部だけを使用する必要がある
- 特定の方法でシステムとやり取りを行う必要がある

解決策

- Facadeによって、既存システムを使用するクライアント向けの新たなインタフェースを作成する

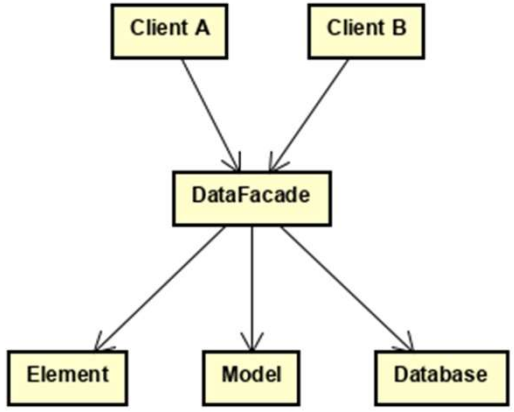
Facadeパターンのクラス図

Facade使用前



それぞれに関連がある

Facade使用後



Facadeによる関係の整理

図書館システムにおけるFacadeパターンの利用

図書館の蔵書情報にアクセスして、借りたい本があるかを検索する

蔵書

- 著書と本（Book、Copy）
- 雑誌（Journal）

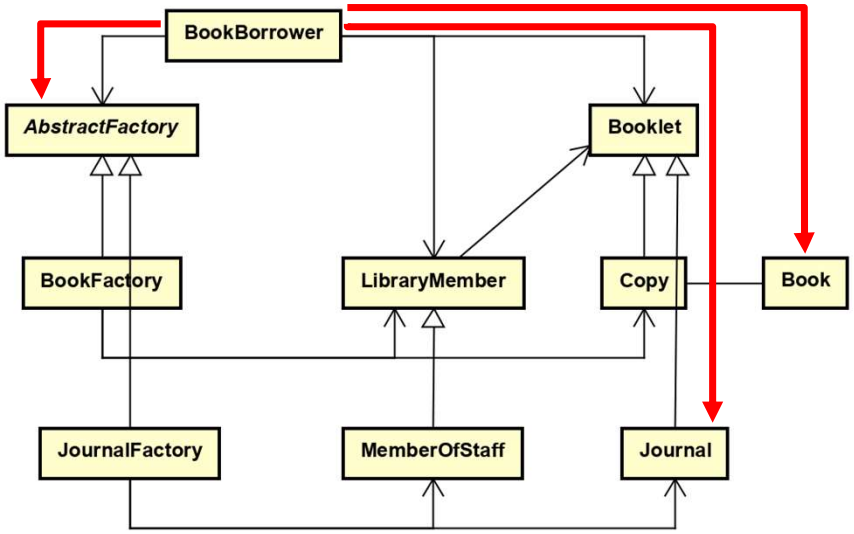
借りたい本

- 今回は本の名前を使い検索する

複数のクラスを扱うことなく検索したい

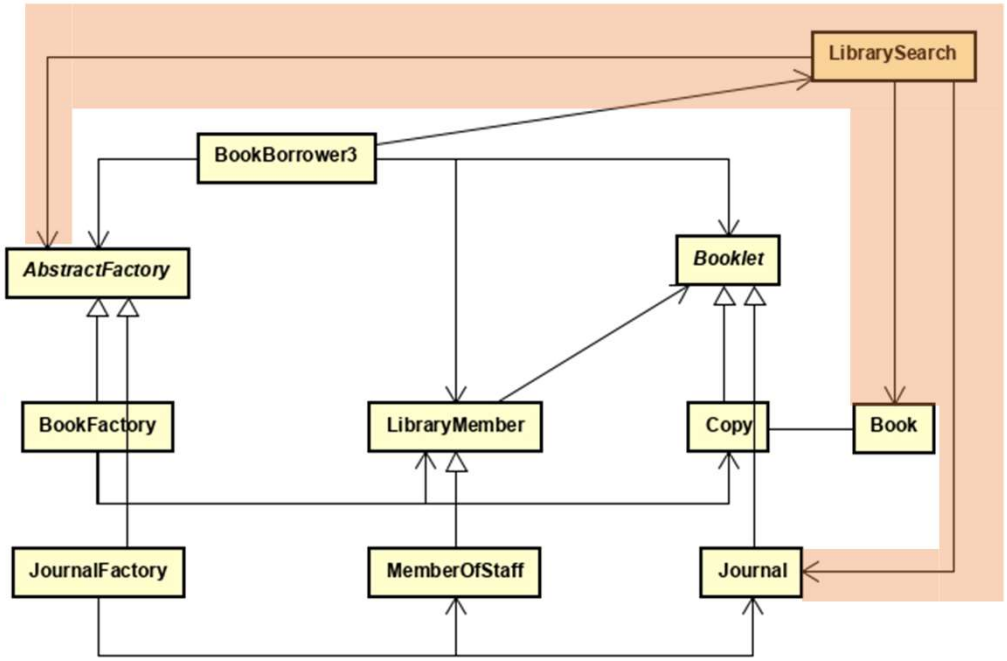
図書館システムに当てはめる（クラス図）

Facade使用前



BookとJournalのオブジェクトのリスト
BookBorrowerが管理し検索しなければならない

Facade使用後



LibrarySearchでリストが管理され検索できる
BookBorrowerは検索の依頼をするだけでよい

課題1

第4回のプログラムに学生を学籍番号で検索する機能を追加する

Sample.javaのmainメソッドの最後に検索するコードを書く

例：学籍番号237番を検索するコード

```
int targetNumber = 237;
if (mine.getNumber() == targetNumber) {
    System.out.println("学籍番号" + targetNumber + "番は" + mine.getName() + "さんです。");
} else if (friend1.getNumber() == targetNumber) {
    System.out.println("学籍番号" + targetNumber + "番は" + friend1.getName() + "さんです。");
} else if (friend2.getNumber() == targetNumber) {
    System.out.println("学籍番号" + targetNumber + "番は" + friend2.getName() + "さんです。");
} else if (friend3.getNumber() == targetNumber) {
    System.out.println("学籍番号" + targetNumber + "番は" + friend3.getName() + "さんです。");
} else if (friend4.getNumber() == targetNumber) {
    System.out.println("学籍番号" + targetNumber + "番は" + friend4.getName() + "さんです。");
} else if (friend5.getNumber() == targetNumber) {
    System.out.println("学籍番号" + targetNumber + "番は" + friend5.getName() + "さんです。");
} else {
    System.out.println("学籍番号" + targetNumber + "番はいませんでした。");
}
```

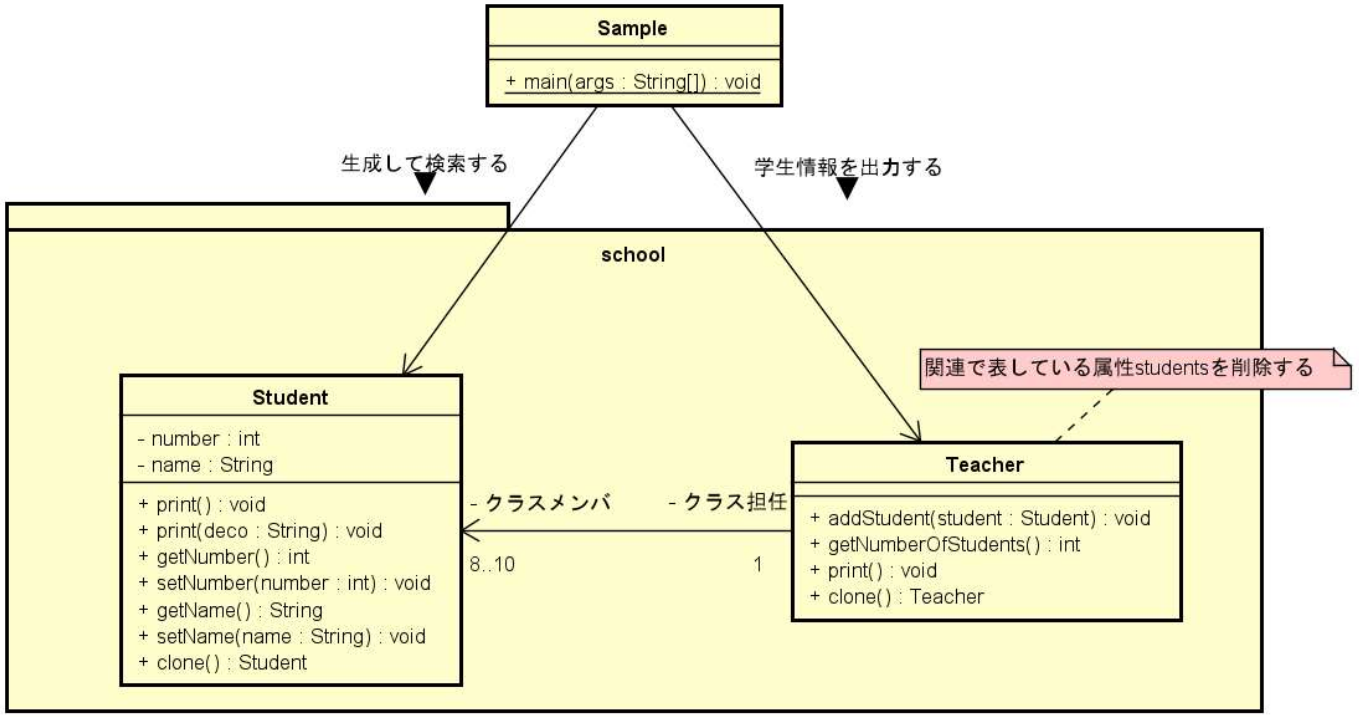
検索しているコード

Facadeパターンの活用

Sampleクラスで検索するのではなく、Studentクラスのインスタンスを管理しているクラスに検索を依頼する

学生について
- 情報出力はTeacherクラス
- 検索はSampleクラス

Teacherクラスにまとめる

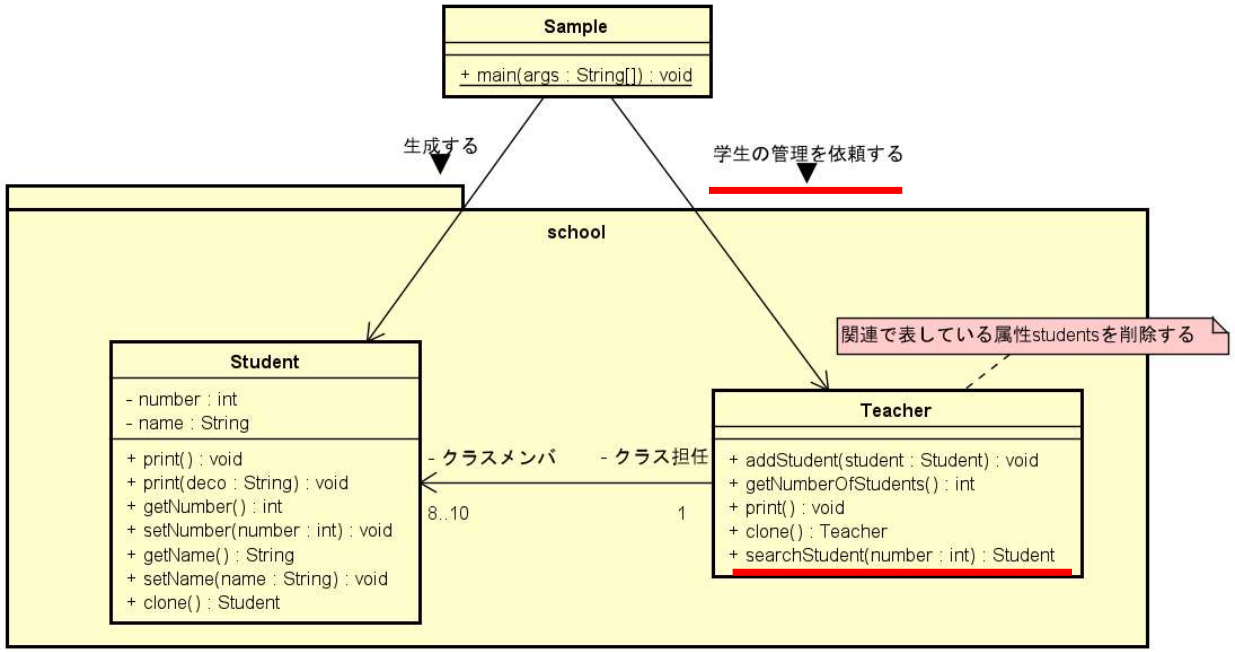


第4回のプログラムの構造（クラス図）

Teacherクラスを管理クラスにする

Teacherクラスが学生に関することを担う

検索する操作
「searchStudent」を
新たに持つ



第4回のプログラムの構造（クラス図）から
Teacherクラスに操作を追加

Teacherクラスへの操作追加

Teacher.javaに操作「searchNumber」を追加する

```
public Student searchNumber(int number) {
    for (Student s: students) {
        if (s.getNumber() == number) {
            return s;
        }
    }
    return null;
}
```

searchNumberメソッド

Sampleクラスの変更

Sample.javaのmainメソッドの最後に検索するコードを書く

例：学籍番号237番を検索するコード

```
int targetNumber = 237;
Student targetStudent = teacher.searchNumber(targetNumber);
if (targetStudent != null) {
    System.out.println("学籍番号" + targetNumber + "番は" + targetStudent.getName() + "さんです。");
} else {
    System.out.println("学籍番号" + targetNumber + "番はいませんでした。");
}
```

検索しているコード

AbstractFactoryパターン

23のパターン

生成に関するパターン

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton

構造に関するパターン

6. Adapter
7. Bridge
8. Composite
9. Decorator
10. Facade
11. Flyweight
12. Proxy

振る舞いに関するパターン

13. Chain of Responsibility
14. Command
15. Interpreter
16. Iterator
17. Mediator
18. Memento
19. Observer
20. State
21. Strategy
22. Template Method
23. Visitor

今回はAbstract Factoryパターンを
利用する

Abstract Factoryパターン

関連があるオブジェクトの生成を１カ所で行う

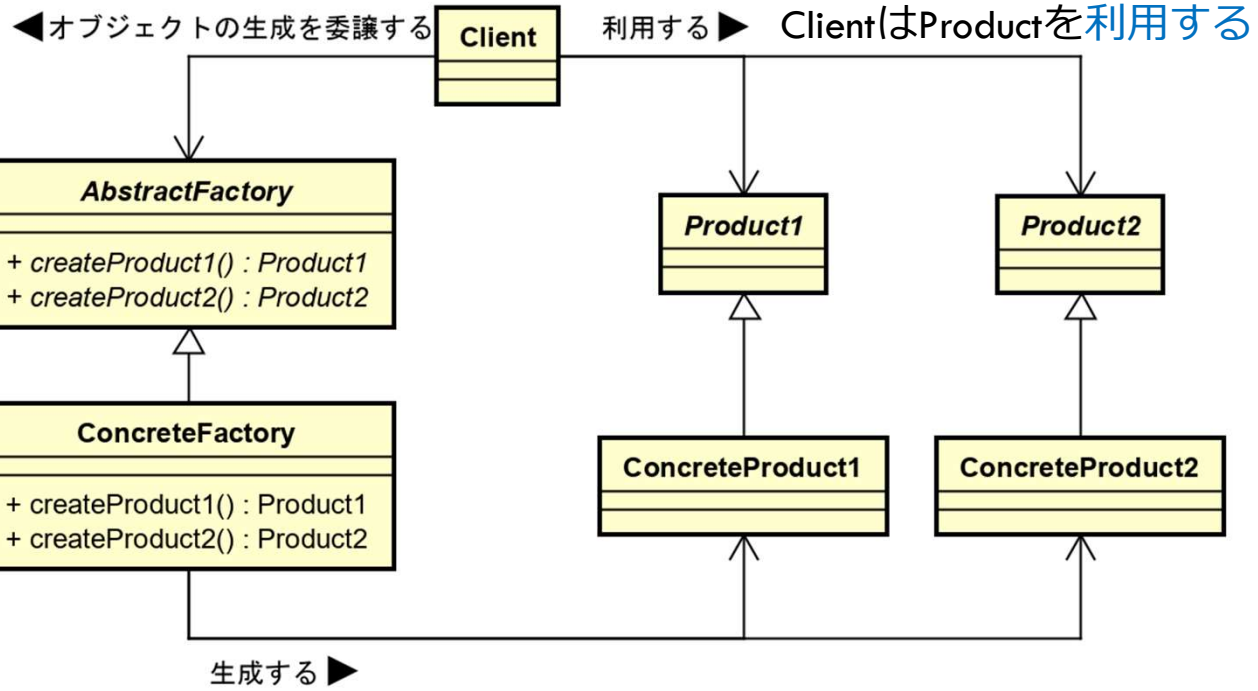
- オブジェクトを生成するための工場（Factory）がある
- 工場は依頼を解釈してオブジェクトを作る

このパターンが有効なケース

- 同じ系統で異なる種類のオブジェクトを多数使うとき
- 生成するオブジェクトをグループ単位で入れ替えるとき

Abstract Factoryパターンのクラス図（１）

ClientはFactoryに
オブジェクトの生成を委譲する
委譲：譲って任せる



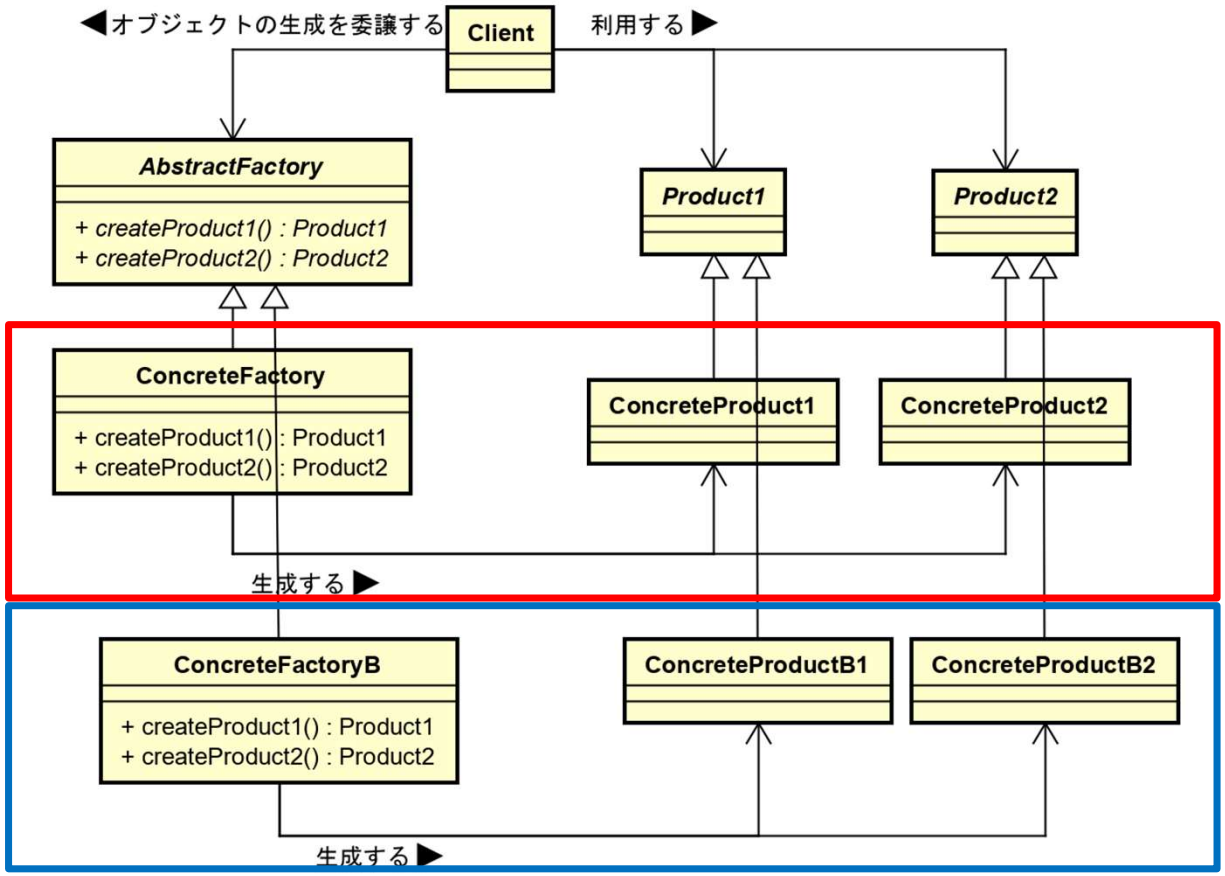
オブジェクト生成工場の抽象クラス

オブジェクト生成工場の具象クラス

生成される製品の抽象クラス

生成される製品の具象クラス

Abstract Factoryパターンのクラス図（２）



オブジェクト
生成工場の
具象クラスを追加

図書館システムにおける AbstractFactoryパターンの利用

図書館利用者（LibraryMember）
- 図書（Copy）を借りることができる

一般関連クラス
- LibraryMember
- Copy
- Book

方針

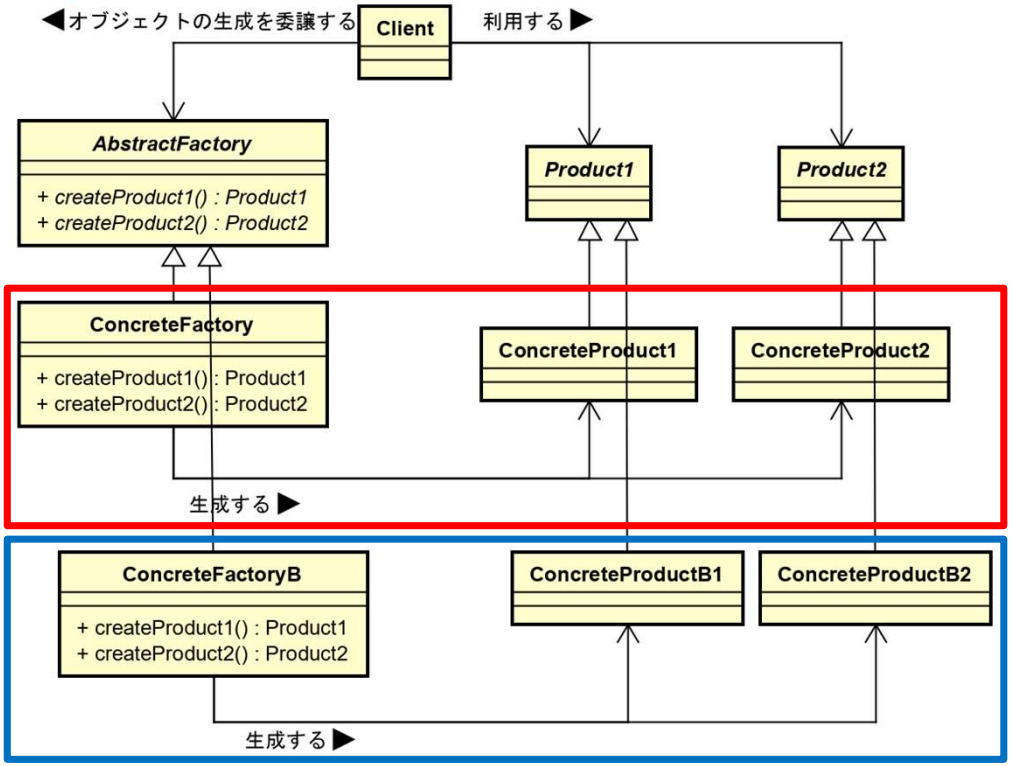
一般関連のオブジェクトを生成するFactoryを準備

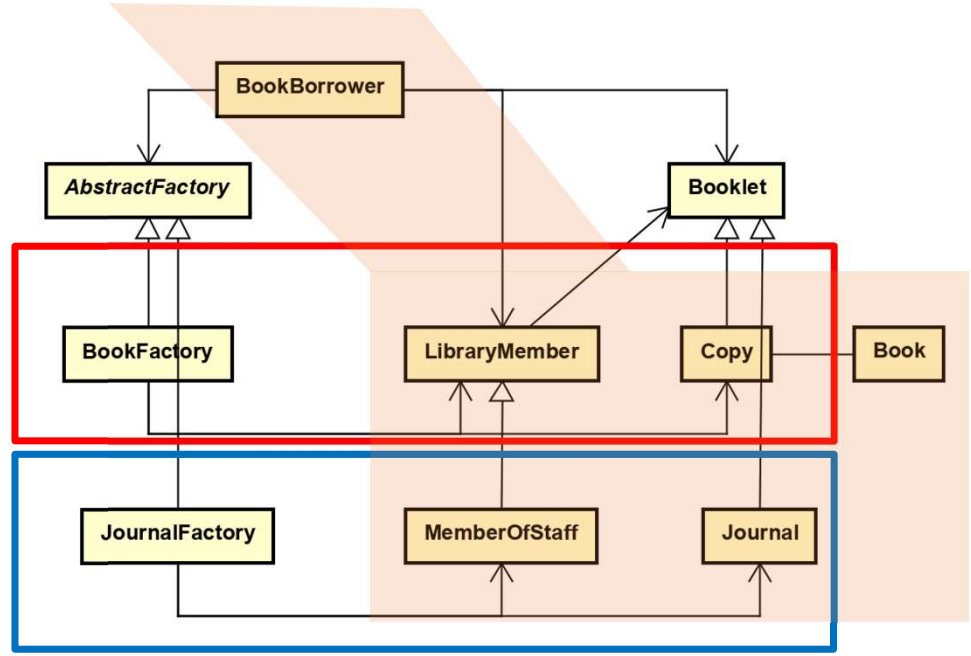
職員（MemberOfStaff）
- 図書（Copy）を借りることができる
- 雑誌（Journal）を借りることができる

職員関連クラス
- MemberOfStaff
- Journal

職員関連のオブジェクトを生成するFactoryを準備

図書館システムに当てはめるクラス図





課題2

課題1のプログラムに学生をまとめて生成する機能を追加する

Sample.javaのmainメソッドにオブジェクトを生成するコードが多くある

例：Studentクラスのインスタンスを生成するコードの一部

```
Student friend1 = new Student(235);
friend1.setName("香椎花子");
friend1.print();

Student friend2 = new Student("松香台五郎");
friend2.setNumber(236);
friend2.print();

Student friend3 = new Student(237, "東廉太郎");
friend3.print();
```

Studentクラスのインスタンスを生成しているコード

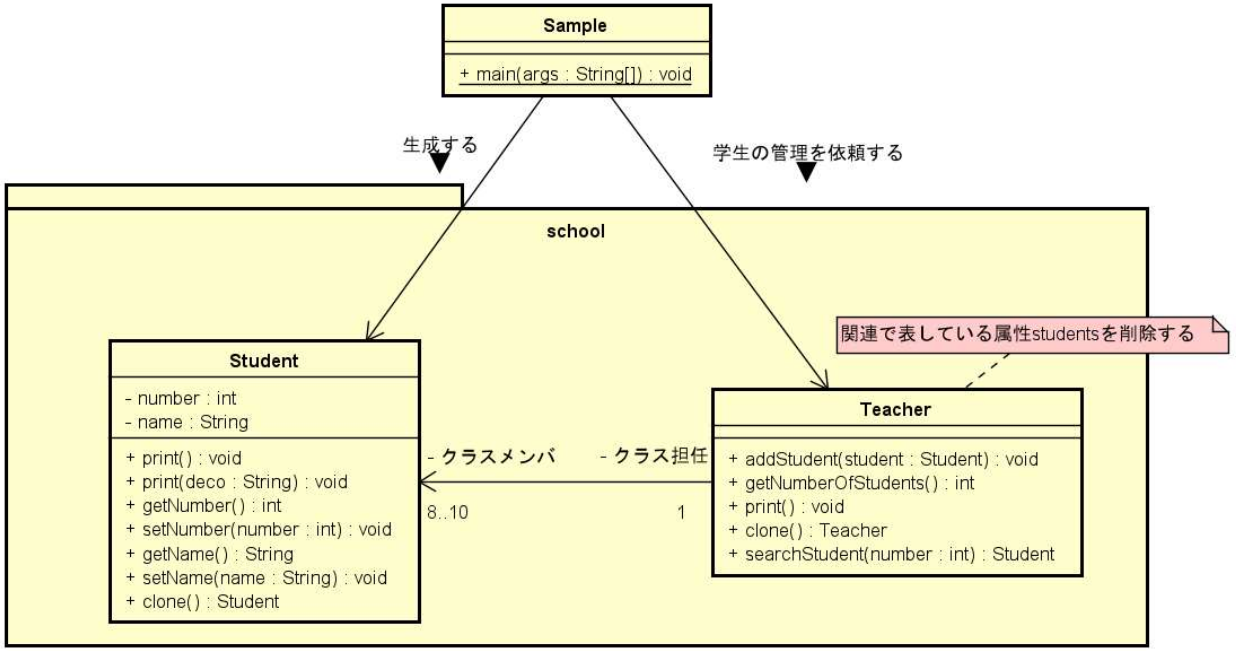
Factoryパターンの活用

SampleクラスでStudentクラスのインスタンスを生成するのではなく、新たなクラスに生成を依頼する

Sampleクラスでインスタンスを生成する

新規にStudentFactoryクラスを作り、インスタンス生成をまとめる

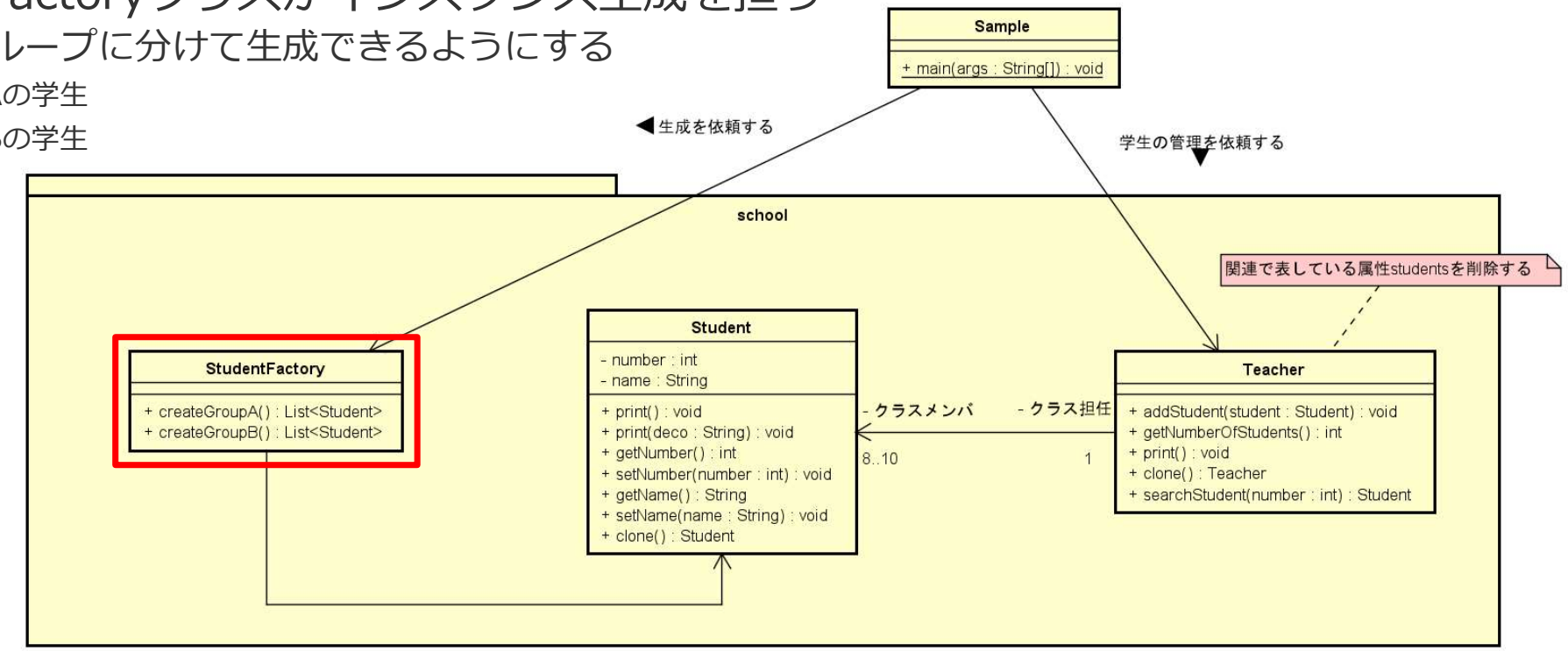
課題1の構造（クラス図）



Factoryクラスを追加する

StudentFactoryクラスがインスタンス生成を担う

- 学生をグループに分けて生成できるようにする
  - グループAの学生
  - グループBの学生



StudentFactoryクラスの新規作成

例：

- グループA
  234番 九産太郎
  235番 香椎花子
  236番 松香台五郎
  237番 東廉太郎
- グループB
  238番 福岡京子
  239番 日本卑弥呼

自分自身のSampleクラスに
合わせてグループを作ること

```
package school;

import java.util.ArrayList;
import java.util.List;

public class StudentFactory {

    // グループAの学生たちを生成する。
    public List<Student> createGroupA() {
        List<Student> students = new ArrayList<Student>();
        students.add(new Student(234, "九産太郎"));
        students.add(new Student(235, "香椎花子"));
        students.add(new Student(236, "松香台五郎"));
        students.add(new Student(237, "東廉太郎"));
        return students;
    }

    // グループBの学生たちを生成する。
    public List<Student> createGroupB() {
        List<Student> students = new ArrayList<Student>();
        students.add(new Student(238, "福岡京子"));
        students.add(new Student(239, "日本卑弥呼"));
        return students;
    }
}
```

StudentFactoryクラス

Sampleクラスの変更

Sample.javaのmainメソッドのインスタンス生成部分を置き換える

例：
グループAと
グループBを
生成するコード

補足：
学生番号の変更や
クローンは省略して
良い

```java
import school.Student;
import school.StudentFactory;
import school.Teacher;

public class Sample {

    public static void main(String[] args) {

        StudentFactory factory = new StudentFactory();
        Teacher teacher = new Teacher();
        teacher.addStudent(factory.createGroupA());

        System.out.println("学生は" + teacher.getNumberOfStudents() + "人です。");
        teacher.print();

        teacher.addStudent(factory.createGroupB());

        System.out.println("学生は" + teacher.getNumberOfStudents() + "人です。");
        teacher.print();

        int targetNumber = 237;
        Student targetStudent = teacher.searchNumber(targetNumber);
        if (targetStudent != null) {
            System.out.println("学籍番号" + targetNumber + "番は" + targetStudent.getName() + "さんです。");
        } else {
            System.out.println("学籍番号" + targetNumber + "番はいませんでした。");
        }
    }
}
```

Sampleクラスのコード

課題の提出

提出内容

- 課題2まで適用したプログラムを提出する

3つのファイルを提出する

1. Sample.java
2. Teacher.java
3. StudentFactory.java

(Student.javaは不要)

ファイル名：1. OOD23-12_Sample.java-学生番号

2. OOD23-12_Teacher.java-学生番号

3. OOD23-12_StudentFactory.java-学生番号

提出期限：

土曜日18:00まで、期限に遅れたときは最大60%の評価となります。