第6章（前半）
条件分岐

この章の目的

◆条件分岐はプログラミングの基本制御であり，条件分岐のおかげで複雑な判断を高速かつ正確に実行できる.

◆一方で条件分岐を杜撰に扱うと，開発者を苦しめる悪魔になる.

◆この章で，正しい分岐条件の扱い方について学びましょう.

内容

内容とキーワード

- ◆早期return
- ◆Switch文の重複
- ◆インタフェース
- ◆ストラテジパターン

早期return

条件分岐のネストによる可読性低下

◆第1章でも取り上げたIF文のネスト

※プログラムの一部

```
if(条件1) {
    何らかの処理（数十行）
    if(条件2) {
        何らかの処理（数十行）
        if(条件3) {
            何らかの処理（数十行）
        }
        何らかの処理（数十行）
    }
     何らかの処理（数十行）
}
```

条件がごちゃごちゃしている
（可読性が低い）

条件分岐のネストによる可読性低下

◆例：魔法発動可能かを判定するロジック

※プログラムの一部

```
if(0 < member.hitPoint) {

    if(member.canAct()) {

        if(magic.costMagicPoint <= member.magicPoint) {
            member.consumeMagicPoint(magic.costMagicPoint);
            member.chant(magic);
        }
    }
}
```

生存しているか？

行動可能か？

魔法力が残存しているか？

早期return

条件を満たしていない場合に，ただちにreturnで抜けてしまう手法を早期returnと呼ぶ.

◆例：魔法発動可能かを判定するロジック

※プログラムの一部

```
if(member.hitPoint <= 0) return;
if(!member.canAct()) return;
if(member.magicPoint < magic.costMagicPoint) return;

member.consumeMagicPoint(magic.costMagicPoint);
member.chant(magic);
```

条件に合わないなら，
さっさと抜けてしまおう！

⇒ 先ほどまでのネストが解消された！

早期return

早期returnを使えば，条件や実行ロジックの追加が簡単.

◆例：魔法発動可能かを判定するロジック

※プログラムの一部

```
if(member.hitPoint <= 0) return;
if(!member.canAct()) return;
if(member.magicPoint < magic.costMagicPoint) return;
if(member.technicalPoint < magic.costTechnicalPoint) return;

member.consumeMagicPoint(magic.costMagicPoint);
member.chant(magic);
member.gainTechnicalPoint(magic.incrementTechnicalPoint);
```

ここに条件を追加すればOK!

ここに実行ロジックを追加すればOK!

else句と早期return

else句も見通しを悪化させる要因の一つであり，早期returnを用いて見通しの良さを確保しよう．

※HPの残存割合から，生命状態を判定するロジック

```
float hitPointRate = member.hitPoint / member.maxHitPoint;

HealthCondition currentHealthCondition;
if (hitPointRate == 0) {
    currentHealthCondition = HealthCondition.dead;
} else if (hitPointRate < 0.3) {
    currentHealthCondition = HealthCondition.danger;
} else if (hitPointRate < 0.5) {
    currentHealthCondition = HealthCondition.caution;
} else {
    currentHealthCondition = HealthCondition.fine;
}

return currentHealthCondition;
```

死亡

危険

注意

良好

なんだか見にくい…

else句と早期return

else句も見通しを悪化させる要因の一つであり，早期returnを用いて見通しの良さを確保しよう.

※HPの残存割合から，生命状態を判定するロジック

```
float hitPointRate = member.hitPoint / member.maxHitPoint;

if (hitPointRate == 0) return HealthCondition.dead;
if (hitPointRate < 0.3) return HealthCondition.danger;
if (hitPointRate < 0.5) return HealthCondition.caution;

return HealthCondition.fine;
```

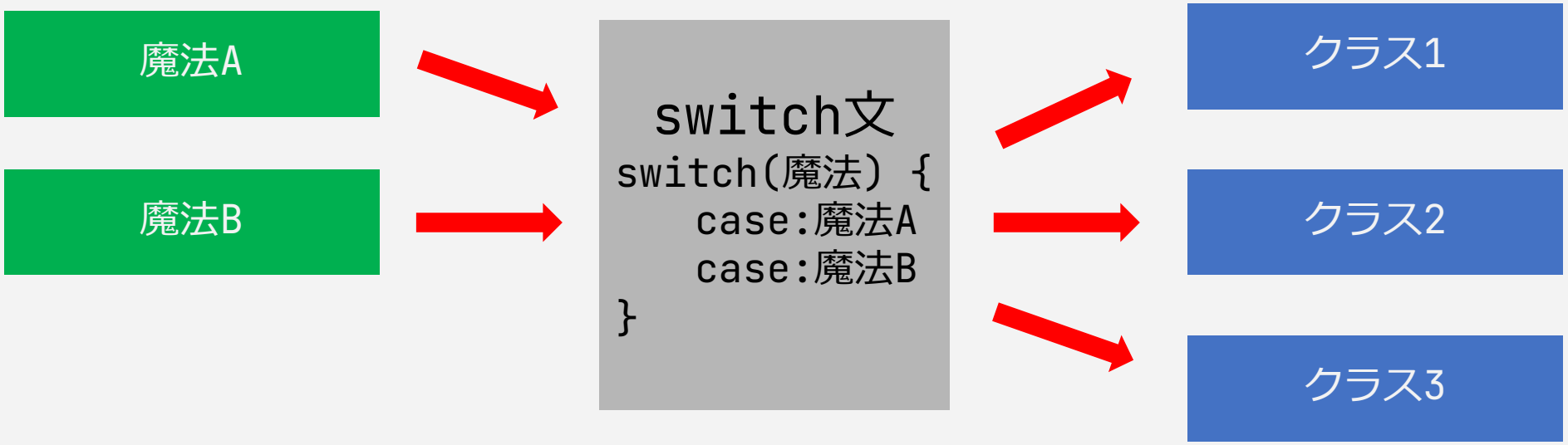
⇒ 早期returnを活用することでスッキリした見た目に！

switch文の重複

switch文の重複

switchで実装される条件分岐は，複数箇所で同じ内容が実装されがちで，
爆発的にコードが増加する．

◆例えば複数の魔法を実装するとき…



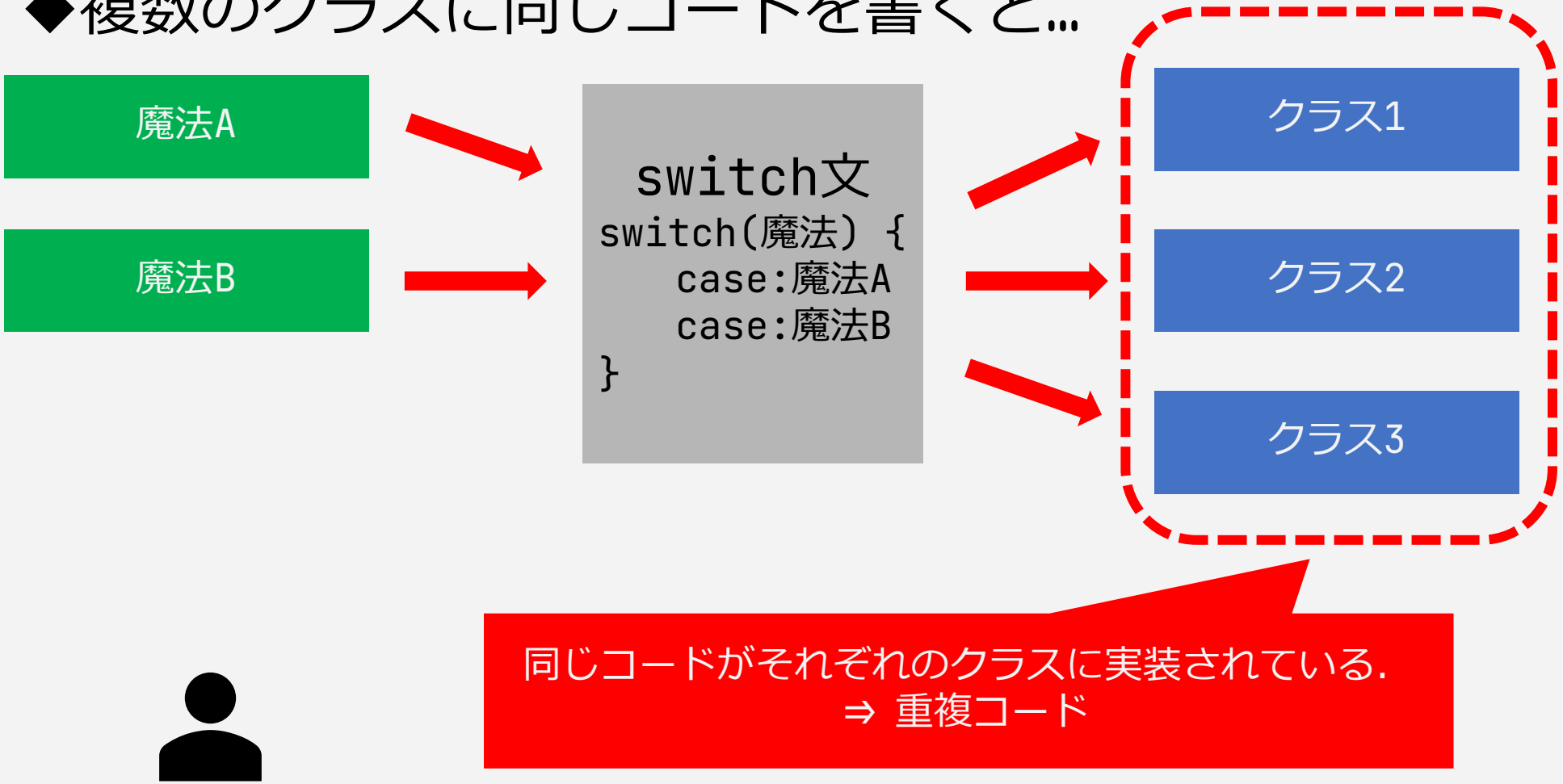
使う魔法によって操作を変えるぞ！
switch文をいろいろなクラスに実装する必要があるな！

⇒条件分岐が必要な場面でそれぞれ書いてしまう．
（同じコードを使い回している）

switch文の重複

switchで実装される条件分岐は，複数箇所で同じ内容が実装されがちで，
爆発的にコードが増加する．

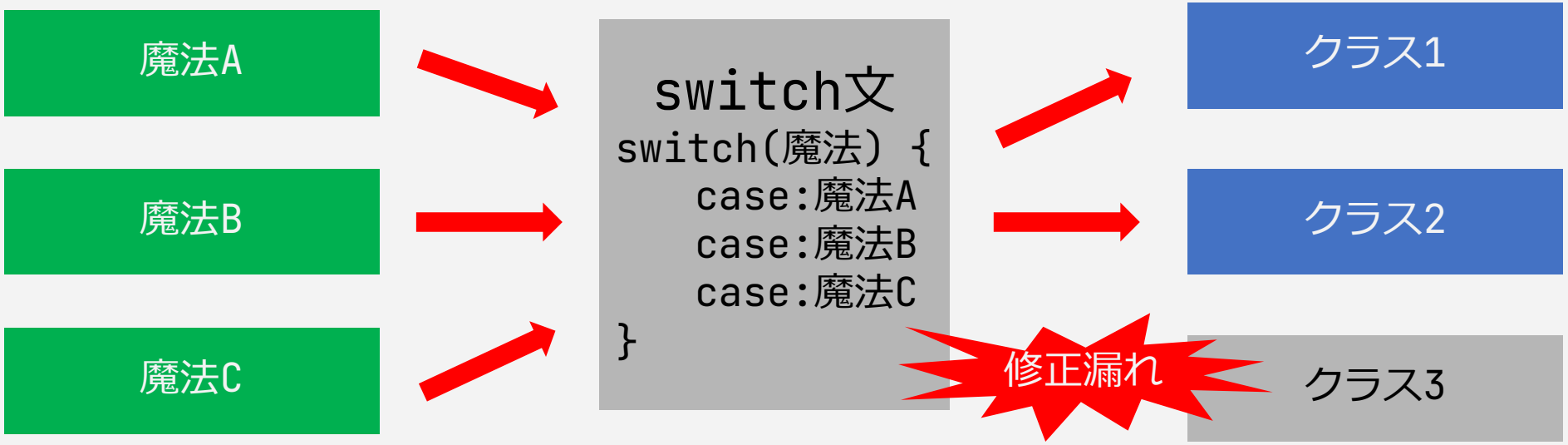
◆複数のクラスに同じコードを書くと…

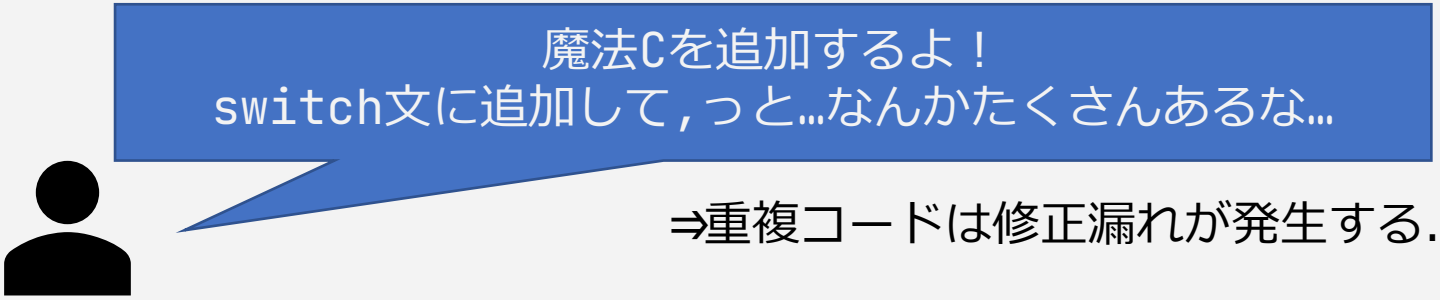


switch文の重複

重複コードが多いと，修正漏れが発生することは，第1章で学びましたね.

◆実装部分を全部チェックして直すのは困難

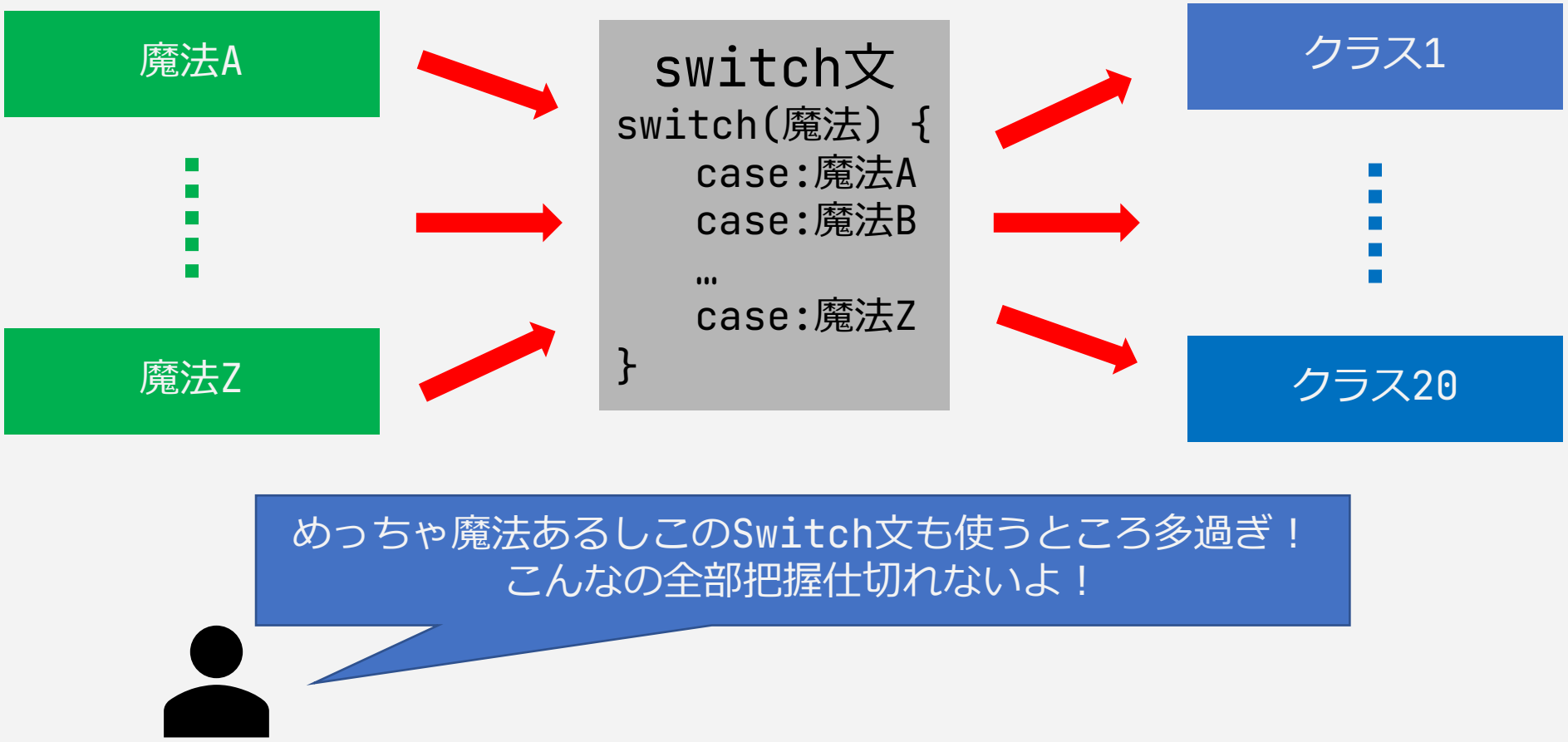




⇒重複コードは修正漏れが発生する.

switch文の重複

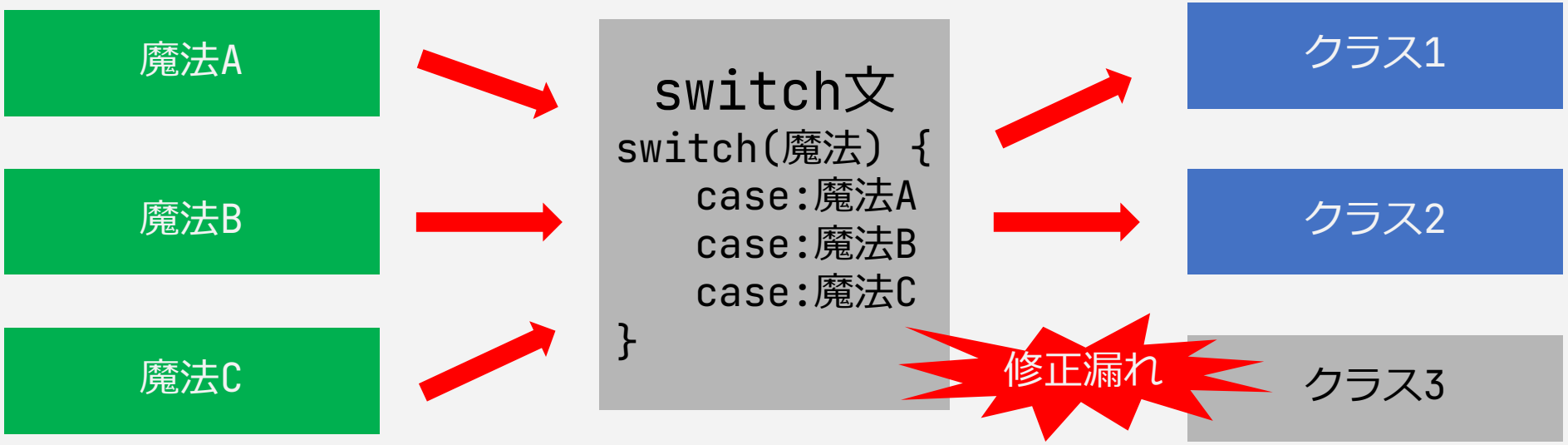
◆さらに実装する魔法やクラスが増えるとなおさら



switch文の重複

修正漏れがあるコードを，誰かが参考にしてしまうかも.

◆問題は他にも



さて，テクニカルポイントについて実装したいんだけど，
Switch文はクラス3のやつを参考にすればいいよね！

⇒ 修正漏れがあるコードを参考に実装してしまう.

switch文クローン問題

◆switch文をその都度実装していると，重複コードが発生し，修正漏れが発生してしまう.

◆数が増えると人間の注意力では全てを把握することは困難になる.

◆仕様変更の際，同じswitch文がたくさんあると，膨大な数のswitch文から修正箇所を探さなければならない. （可読性が低い）

◆種類によって処理を変えたいケースは，どのようなソフトウェアにもあり得るため，重要な問題.

じゃあどうするか？

条件分岐を一カ所にまとめよう

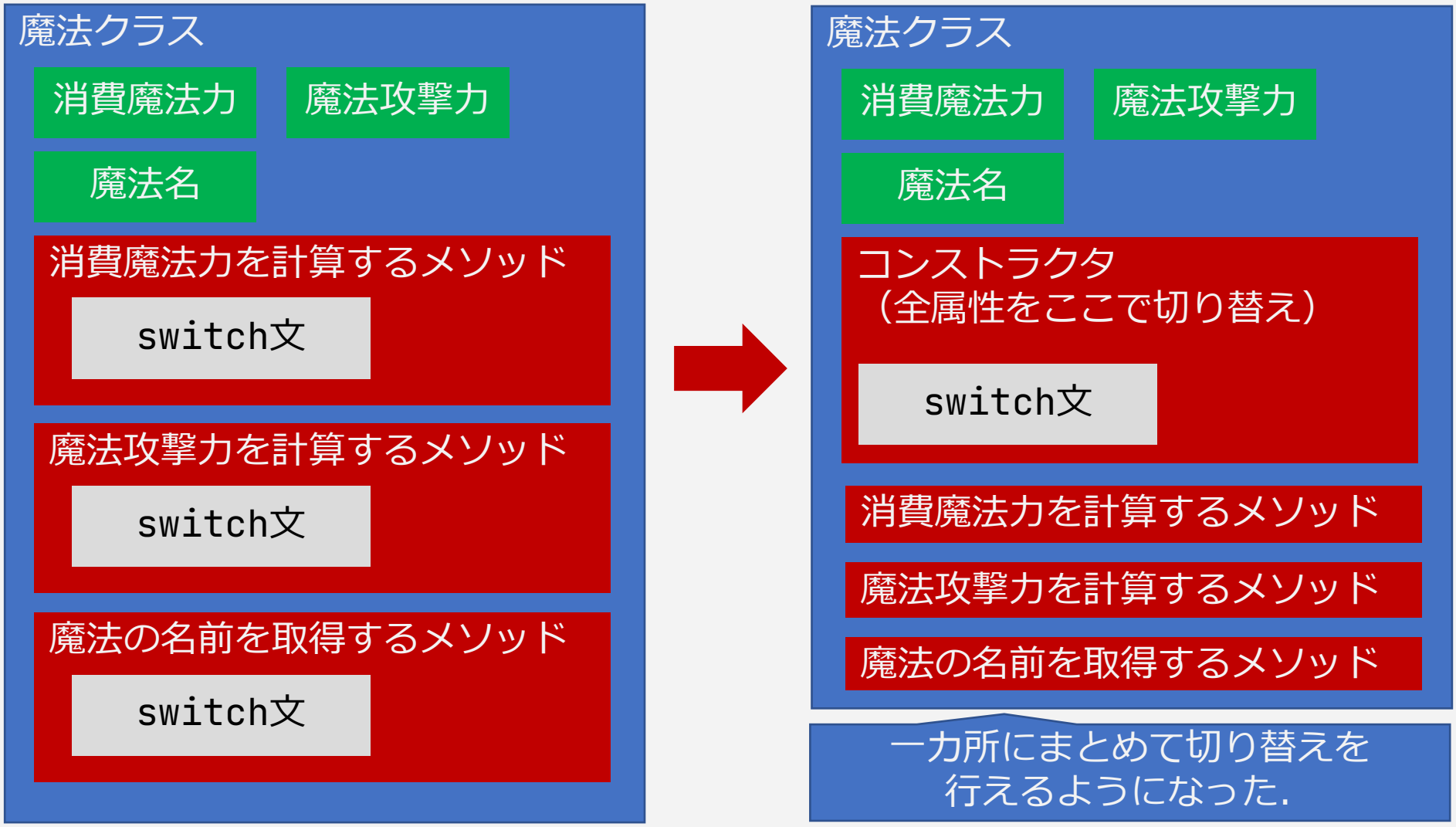
switch文の重複を解消するには，単一責任選択の原則の考え方が重要.

◆単一責任選択の原則とは

■ソフトウェアシステムが選択肢を提供しなければならないとき，そのシステムの中の1つのモジュールがその選択肢の全てを把握すべきである.

■同じ条件式の条件分岐を複数書かず，一カ所にまとめよう.

具体的には…


魔法クラス
消費魔法力
魔法攻撃力
魔法名
消費魔法力を計算するメソッド
switch文
魔法攻撃力を計算するメソッド
switch文
魔法の名前を取得するメソッド
switch文
魔法クラス
消費魔法力
魔法攻撃力
魔法名
コンストラクタ
（全属性をここで切り替え）
switch文
消費魔法力を計算するメソッド
魔法攻撃力を計算するメソッド
魔法の名前を取得するメソッド
一カ所にまとめて切り替えを
行えるようになった.

よりスマートに
switch文重複を解決

interfaceを使おう

switch文の重複は減ったが…

◆単一責任選択の原則により，switch文は一カ所にまとまったが…

◆切り替えたいものが増えた場合，クラスに書くコードがどんどん増えていく．

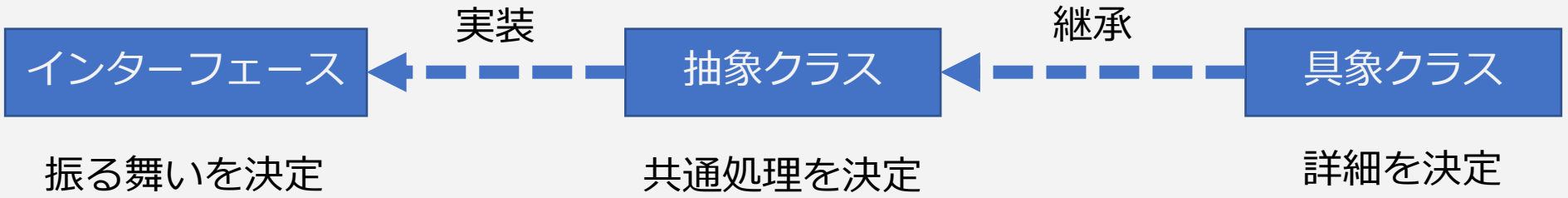
◆クラスが巨大になるとデータとロジックの関係性がわかりにくくなり，可読性・保守性が下がる．

⇒ この問題を解決するのが`interface`！

Interface（インターフェース）って？

インターフェースは実装したクラスに実装すべきメソッドを提示したり，実装したクラスを同じ型として利用できるようにする機能.

◆抽象クラスのうち，（※）抽象メソッドしか持たないものをインターフェースと呼ぶ.



※デフォルトメソッドなどを持つ場合もある.

インターフェースの機能

図形の面積を計算したいが…

円クラス
面積を計算するメソッド

四角クラス
面積を計算するメソッド

面積の計算をするには，
instanceofで型を判別して，
それぞれのメソッドを呼び出す．

図形インターフェース
面積が計算できる

円クラス
面積を計算するメソッド

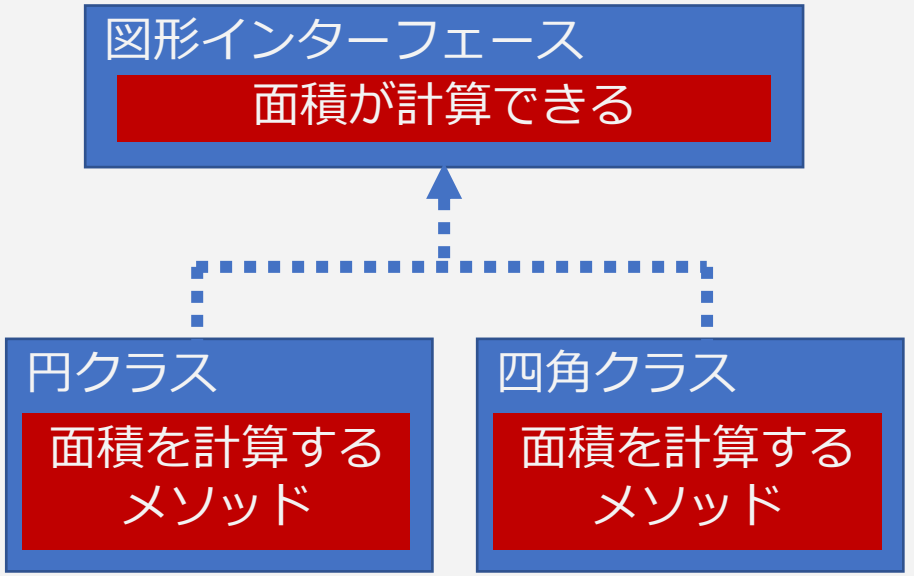
四角クラス
面積を計算するメソッド

図形型として受け取って，
面積を計算するメソッドを呼び出せる．

インターフェースの機能

具体的には？


```
void showArea(Object shape) {
    if (shape instanceof 四角) {
        ((四角) shape).area();
    }
    if (shape instanceof 円) {
        ((円) shape).area();
    }

}
```




```
void showArea(図形 shape) {
    // shapeが円でも四角でもOK
    shape.area();
}
```

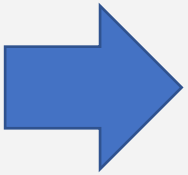
ストラテジパターン

ストラテジパターン

種類をクラス化しましょう.

◆種類ごとに切り替えたい機能のinterfaceのメソッドとして定義する.

○ 切り替えたい機能
1. 名前
2. 消費魔法力
3. 攻撃力
4. 消費テクニカルポイント



```
interface Magic {
    String name();
    int costMagicPoint();
    int attackPower();
    int costTechnicalPoint();
}
```

ストラテジパターン

種類をクラス化しましょう.

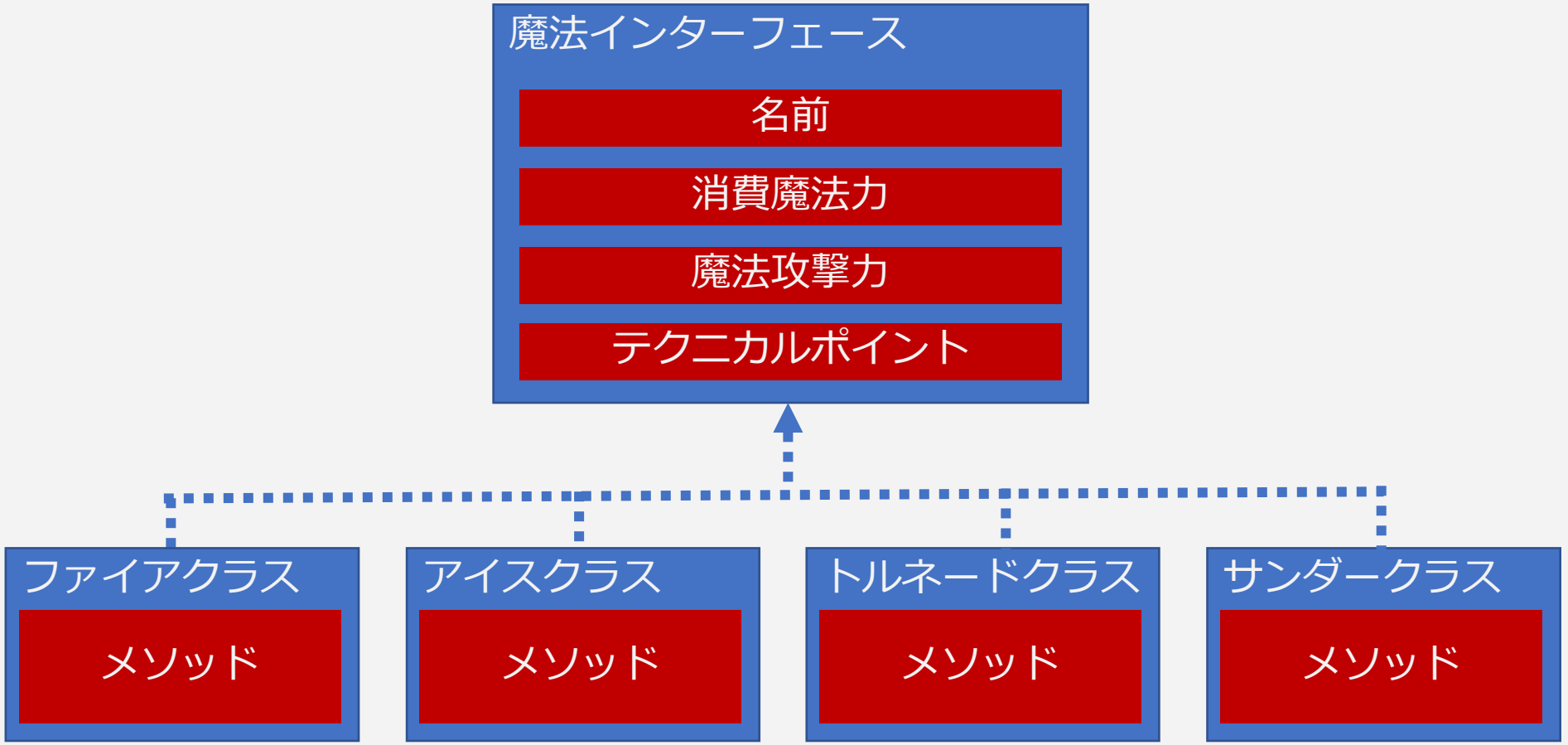
◆種類ごとに切り替えたい機能のinterfaceのメソッドとして定義する.

```
interface Magic {
    String name();
    int costMagicPoint();
    int attackPower();
    int costTechnicalPoint();
}
```

このように他の魔法についても
同様に実装する.

```
interface Fire implements Magic {
    private final Member member;
    Fire(final Member member) {
        this.member = member;
    }
    public String name() {
        return “ファイア”;
    }
    public int costMagicPoint(){
        return 2;
    }
    // 省略
}
```

ストラテジパターン


魔法インターフェース
名前
消費魔法力
魔法攻撃力
テクニカルポイント
ファイアクラス
メソッド
アイスクラス
メソッド
トルネードクラス
メソッド
サンダークラス
メソッド
全て同じ魔法型として扱うことができる！

ストラテジパターン

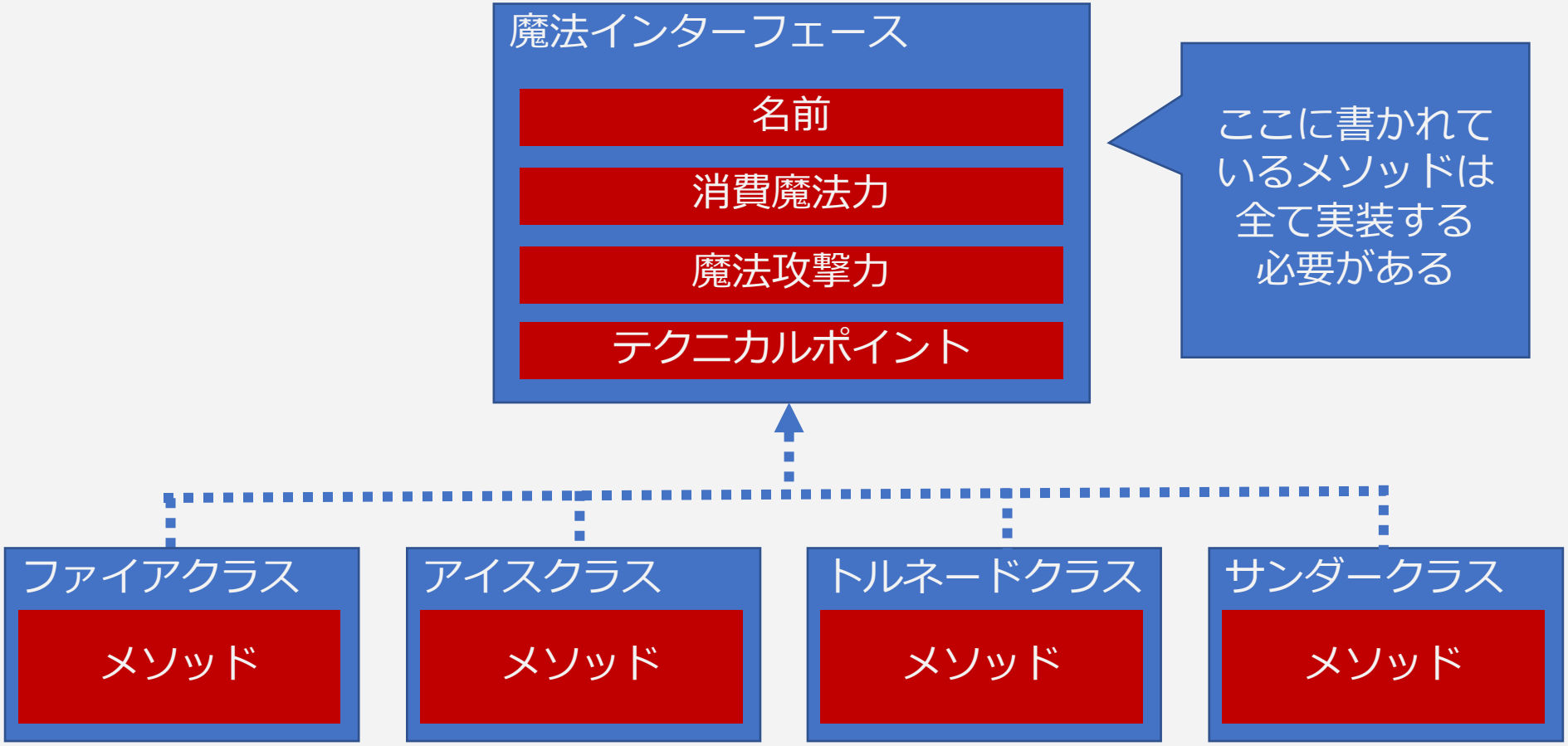
◆処理の切り替えをMapと列挙型で実装

```
final Map<MagicType, Magic> magics = new HashMap<>();

final Fire fire = new Fire(member);
final Ice ice = new Ice(member);
final Tornado tornado = new Tornado(member);
final Thunder thunder = new Thunder(member);

magics.put(MagicType.fire, fire);
magics.put(MagicType.ice, ice);
magics.put(MagicType.tornado, tornado);
magics.put(MagicType.thunder, thunder);
```

```
int magicAttack(final MagicType magicType) {
    final Magic usingMagic = magics.get(magicType);
    usingMagic.attackPower();
}
```

インタフェースのもう一つの役割

インタフェースのメソッドを全て実装しないと，コンパイルエラーになるため，実装漏れを防ぐことができる.



次の章に向けて

- ◆Switch文をいちいち書くのではなく，インタフェースでの設計を試みる.
- ◆ストラテジパターンを活用しよう.