ITCollege

無断転載、転用禁止

ソフトウェア設計手法

進化し続ける柔軟なソフトウェアの育て方

Ver. 1.0

copyright ©ITCollege All Rights Reserved

アジェンダ

- 設計とかテストの前に…
- ソフトウェア設計手法
  - 概要
  - 言語の種類
  - オブジェクト指向設計
  - 設計の道具
  - 設計の行為
- SOLID
  - Single Responsibility
  - Open-Closed
  - Liskov Substitution
  - Interface Segregation
  - Dependency Inversion
  - SOLIDまとめ
- 実際の設計例
- デザインパターン
- UML
- クラス図
- 補足
- シーケンス図
- デザインパターン
  - Singleton
  - TemplateMethod
  - FactoryMethod
  - Iterator
- まとめ

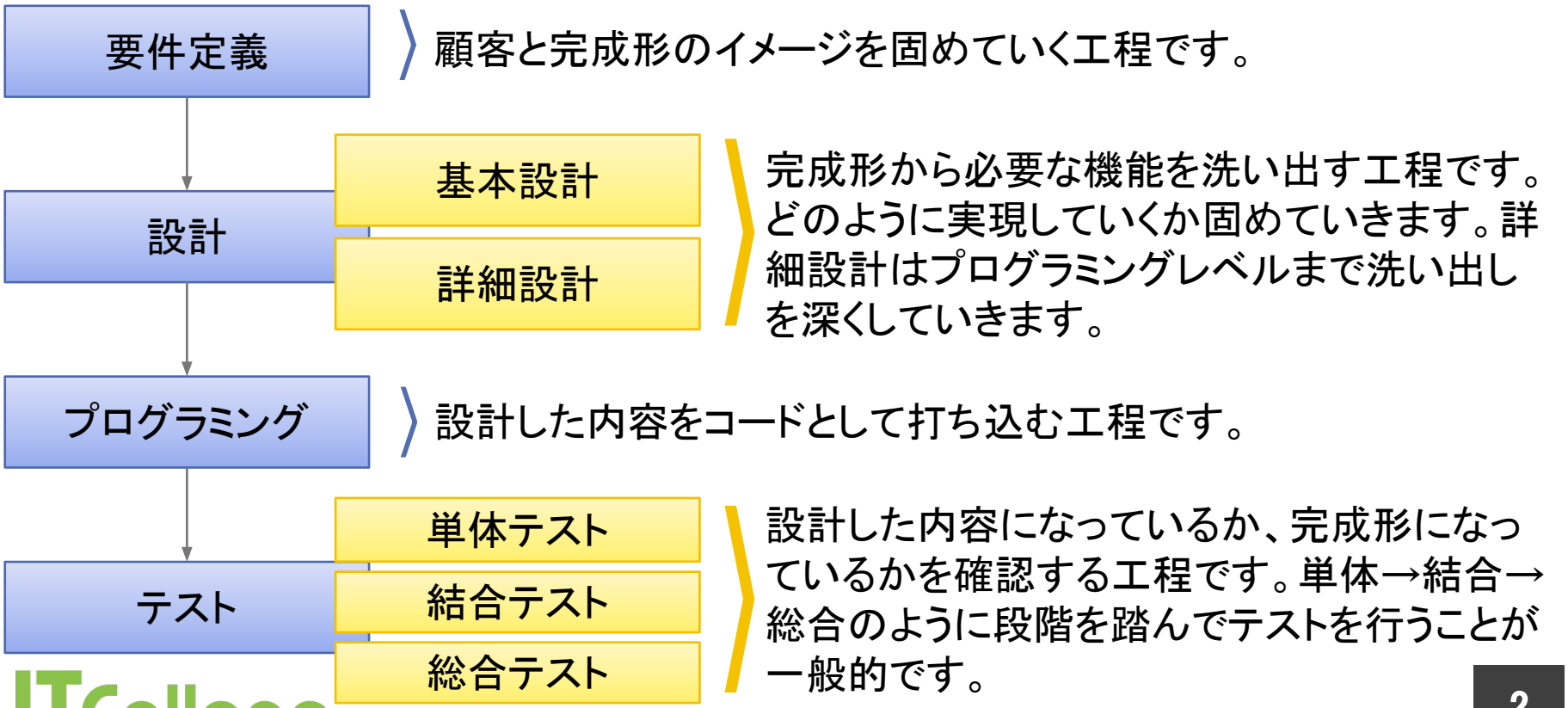
ITCollege copyright ©ITCollege All Rights Reserved

設計とかテストの前に…

- **システム開発での作業**

システム開発を行う上で工程というものがあり、開発現場ではその工程を順番に消化することで効率よく開発をしています。

ではどんな工程があるのか見ていきましょう。



ITCollege
copyright ©ITCollege All Rights Reserved

設計とかテストの前に…

- **工程の進め方**

システム開発を行う上で工程の進め方がいくつかあります。プロジェクト管理において以下のようなリスクが担保できればどんな進め方でもよいですが…

・スケジュールの欠陥

・要件の増大(変更)

・人員の離脱

・仕様の崩壊

進め方のメリット・デメリットを検討した上で選択できれば、プロジェクトとしての成功に近づいていきます。

※現場によって進め方が決まっているところもありますが・・・

その中で代表的な**ウォーターフォール**と**アジャイル**について紹介します。

ITCollege copyright ©ITCollege All Rights Reserved

設計とかテストの前に…

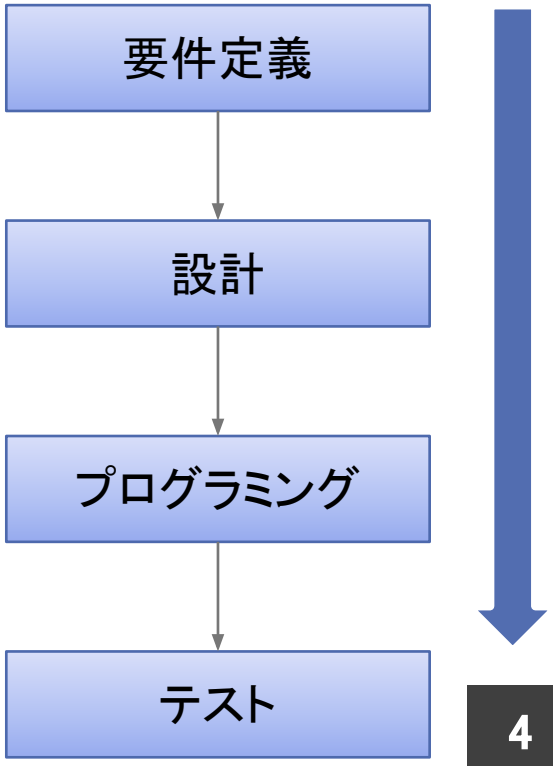
- **ウォーターフォール**

従来のシステム開発としては最もポピュラーな開発の進め方といえます。
工程を以下のように順番通りに進めていきます。

滝から水が落ちるように、前工程の成果物に基づき、次の工程の作業を行い、前工程への手戻りを想定していません。この様からウォーターフォールと呼ばれます。

一度、前工程への手戻りが発生するようなことが
発生した場合は、工程への波及範囲が広く、
取り戻すことは困難を極めます。

そこでこの進め方の弱点を克服すべく登場したのが、
反復開発を軸とするアジャイルという方法でした。



copyright ©ITCollege All Rights Reserved

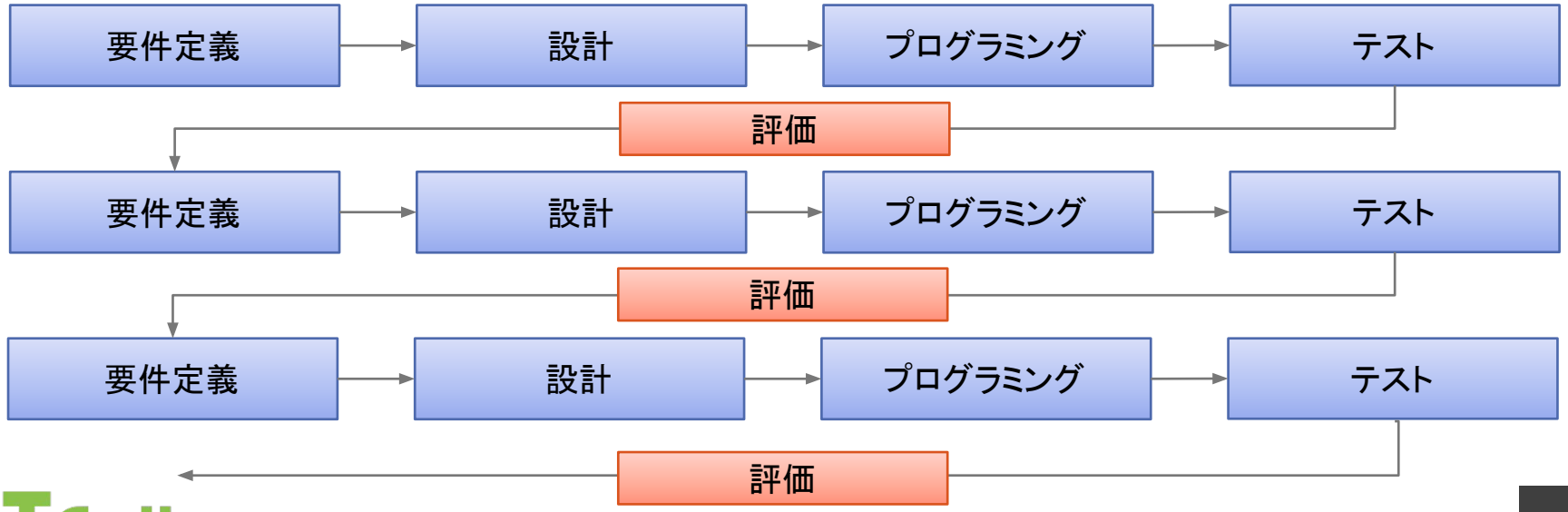
設計とかテストの前に…

- **アジャイル**

アジャイル開発では1つの開発サイクルを小さくし、プロジェクトを完了するまで反復する方法です。各サイクルを回すごとに評価を行い、手戻りをなくし、リスクを最小化しようとしています。これだけだと前述の問題をすべて解決したように感じますが、以下が成立しないとアジャイルは崩壊します。

・メンバー間の意思疎通　・問題の理解　・解決策の提案

そのため、プロジェクトリーダに負担がかかりやすく、成功までのプロセスは困難を極めます。



ITCollege copyright ©ITCollege All Rights Reserved

では本題の・・・

ソフトウェア設計手法
へ移ります

ITCollege copyright ©ITCollege All Rights Reserved

概要

- **ソフトウェア開発の現状**

開発現場において、コードが育ち、作成中にあるソフトウェアの要件が変化するにつれ、ロジックがさらに追加されていきます。仕様追加や不具合対応、タスク消化によって実装が追加される時、

・初心者はWebからサンプルコードのコピーで済ます

・全体を気にせず何となく書いて動く

のようなコードが多々出てきてしまい、読み返すと煩雑なコードだったり、メンテナンス性の低いコードになることが多くあります。

そこで偉大な先人達は様々な方法で設計について考えてきました。

その設計についてこれから学習していきましょう。

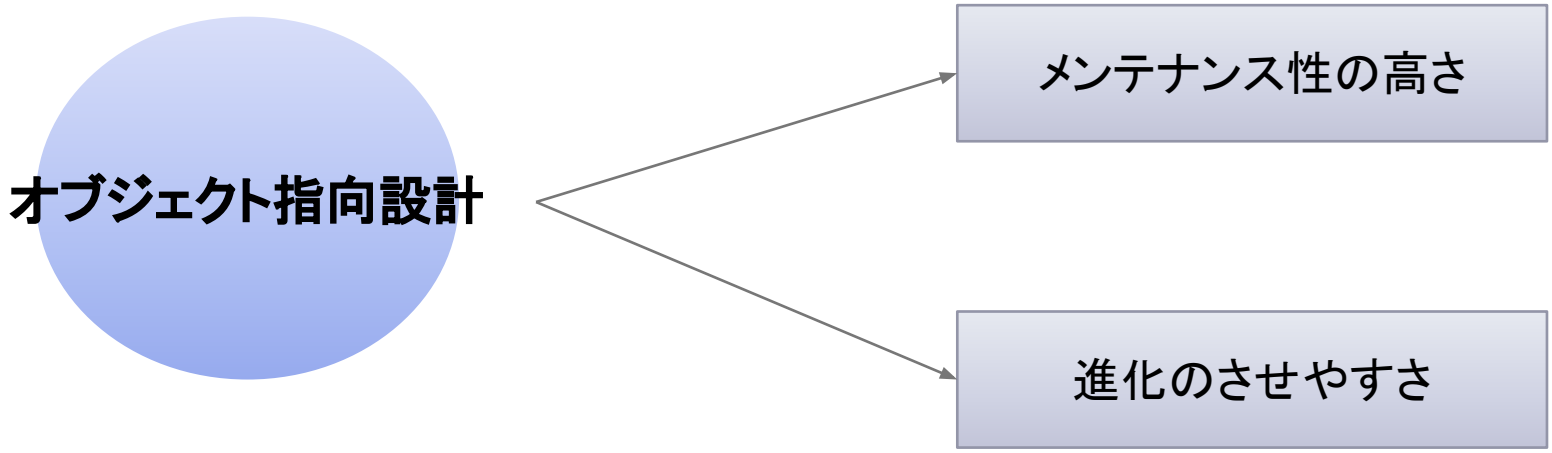
ITCollege copyright ©ITCollege All Rights Reserved

概要

- **オブジェクト指向設計**

まず、オブジェクト指向設計に着目してみましょう。

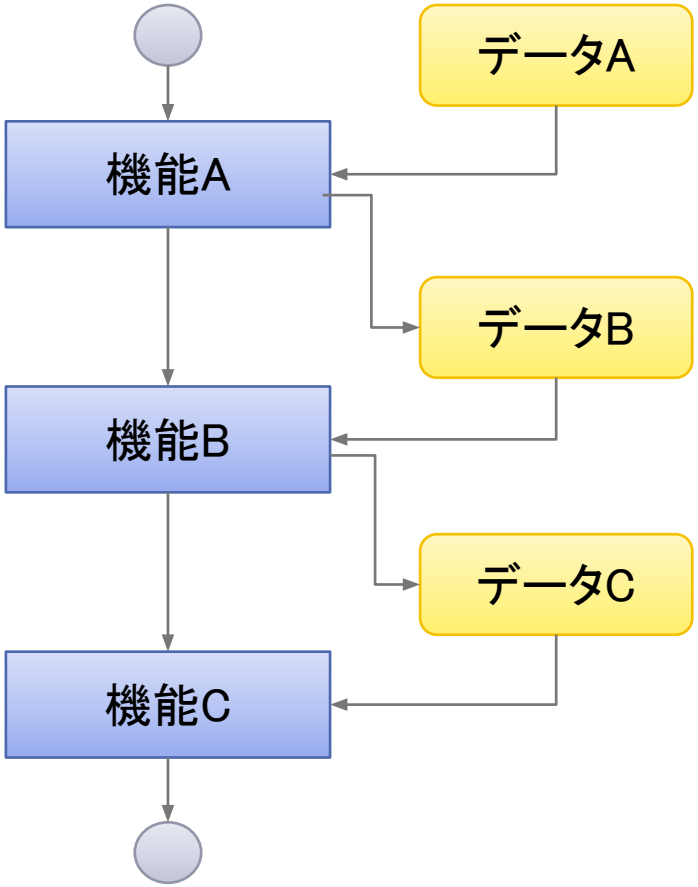
オブジェクト指向設計が約束するのは、コードのメンテナンス性の高さと進化のさせやすさです。オブジェクト指向設計の基礎をおさえ、次の段階である成熟したプログラマを目指しましょう。



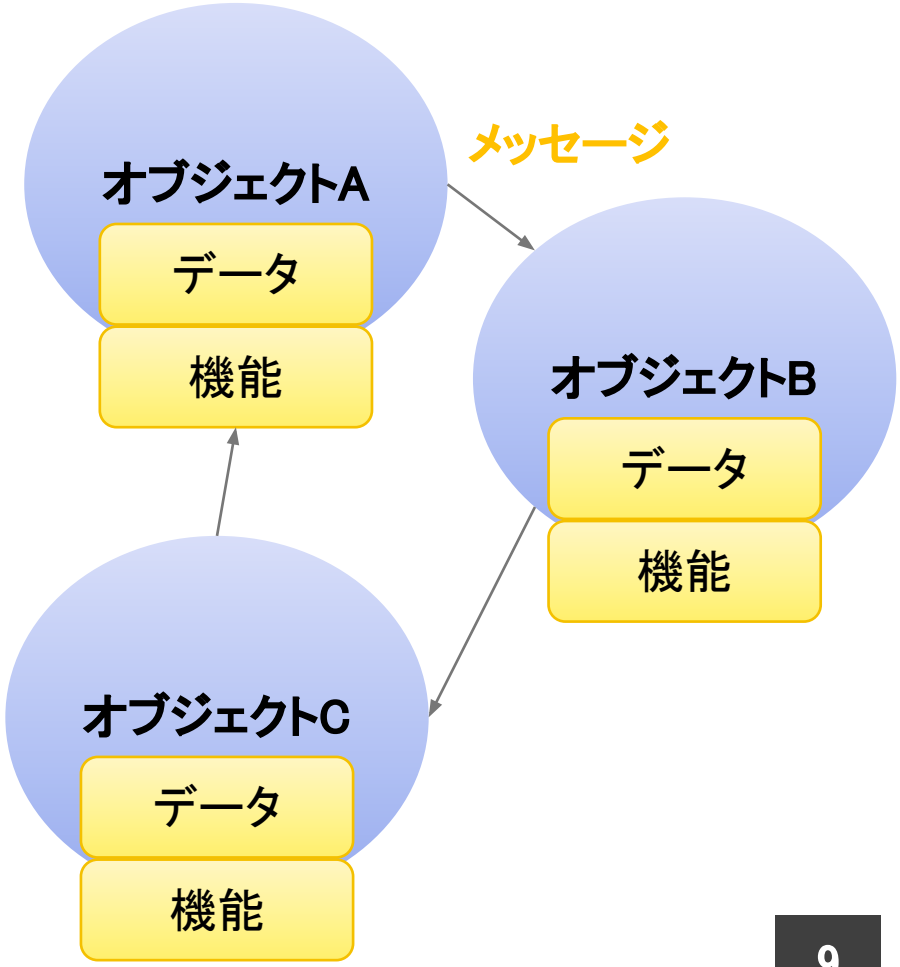
ITCollege copyright ©ITCollege All Rights Reserved

言語の種類

手続き型言語



オブジェクト指向言語



ITCollege copyright ©ITCollege All Rights Reserved

オブジェクト指向設計

- **設計が解決する問題**

設計が不要	完全で正確に仕様を満たしており、永久に変化しないアプリケーション
設計が必要	仕様に抜けがあり、変化するアプリケーション （依頼者のニーズ理解不足、より良い方法の構築など）

- **なぜ必要か？**

あらゆる面で完璧なアプリケーションでさえ、安定はしません。例えば大成功したアプリケーションの場合は、依頼者からさらなる要望が寄せられることでしょう。このように変化から逃げることはできません。

この**変更の必要性**こそが設計を必要かつ重要とするのです。

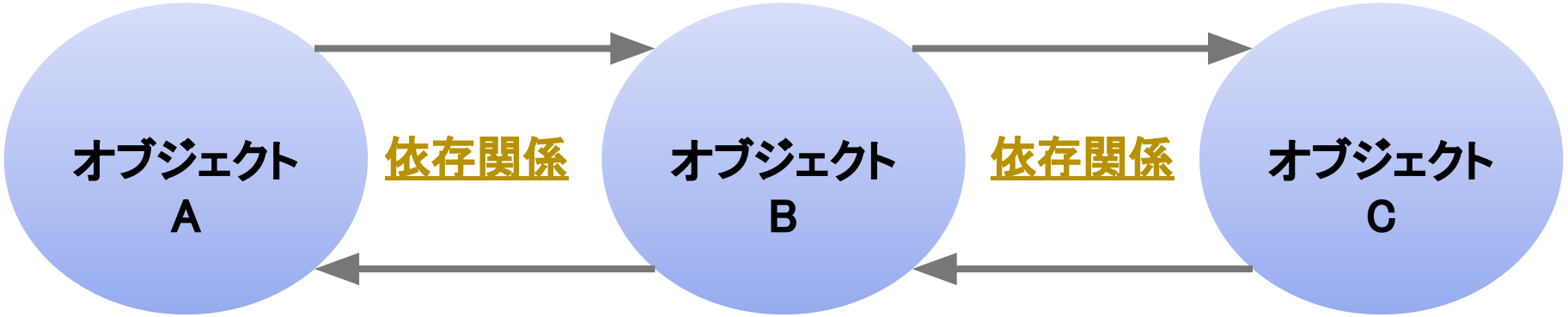
そのため、変更を容易にすることでプログラムを書くにも拡張するにも楽しんでできるようにしましょう。

ITCollege copyright ©ITCollege All Rights Reserved

オブジェクト指向設計

- **変更が困難な理由**

先ほどの図の通り、オブジェクト指向のアプリケーションは
部品(オブジェクト)が相互に作用しあい、全体の振舞いが生まれます。



この依存関係を管理することが肝になります。オブジェクト指向設計とは、オブジェクトが変更を許容できるようなかたちで、依存関係を構築するためのコーディングテクニックが集まったものです。設計がないと管理されていない依存関係が大混乱を起こします。

ITCollege copyright ©ITCollege All Rights Reserved

オブジェクト指向設計

- **設計の実用的な定義**

問題が発生するという未来を推測することは無理です。そうではなく、将来何かが起こるということを認めた上で、動くための余地を設計者に残すものです。

つまり、設計の目的は…「あとにでも」設計をできるようにすることであり、第一の目標は変更コストの削減です。

ITCollege copyright ©ITCollege All Rights Reserved

設計の道具

- **設計原則**

SOLIDと呼ばれる最も有名な5つの原則を紹介します。

S	ingle Responsibility	単一責任
O	pen-Closed	オープン・クローズド
L	iskov Substitution	リスコフの置換
I	nterface Segregation	インタフェース分離
D	ependency Inversion	依存性逆転

他にもDRYやデメテルの法則といったものもあります。これらの原則は研究結果に基づき、信頼できるデータも取れています。

つまり、原則に従えば自身のコードを改善できる可能性が高いということです。

ITCollege copyright ©ITCollege All Rights Reserved

設計の道具

- **設計(デザイン)パターン**

デザインパターンは、「オブジェクト指向ソフトウェア設計において遭遇する様々な問題に対して、簡単でかつ明瞭な解を与える」ものであり、「設計プロダクトの柔軟性、モジュール性、再利用性、および理解のしやすさをより高める」ために使えるとあります。

デザインパターンを用いることでプログラマー達は世代を問わず、プログラムの書き方について共通のコミュニケーションと共同作業を可能にしました。

デザインパターンとSOLIDに関しては、また後程詳細を説明します。

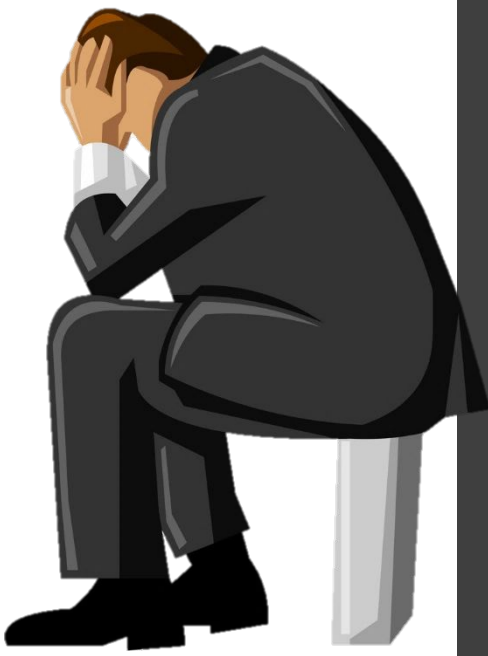
ITCollege copyright ©ITCollege All Rights Reserved

設計の行為

- **設計が失敗する原因**

①設計不十分

設計の初歩を知らなくても動くアプリケーションが作れてしまうことが要因です。このようなアプリケーションは動く状態にはなりますが、次第に変更ができなくなることでしょう。



②設計しすぎる

経験を積んだプログラマーにありがちですが、手法は知っているものの適用方法が分かっていないプログラマーです。

よかれと思って設計していったものが、不適切な場所に原則を適用し、存在しないところにパターンを見出すのです。そうなってしまうと変更要求に対して、複雑で美しいコードの城を築きあげた後に、石の壁に閉じ込められていることに気付いて苦しむことになります。

copyright ©ITCollege All Rights Reserved

設計の行為

- **設計をいつ行うのか？**

「いつ」という回答は、開発手法によって異なります。

業務系システムで一般的なものはウォーターフォール開発でプログラミングの前にしっかりと設計を行う形で進めます。また、アジャイル開発という開発方法もあり、設計・作成してすぐ顧客に見せるようにしています。これは、「顧客は自身の求めるソフトウェアをその目で見るまではっきりとは分からない」という考えが前提にあるためです。

そういった理由からアジャイル開発では、以下のようにも考えられています。

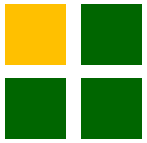
①全体の詳細設計を前もってつくる意味がない(→顧客に見せるたびに変わる)

②完成時期は誰にも予測できない(→顧客が納得するまで終わらない)

プログラマーはプログラムを常に変更し続け、顧客の要望に応えていきます。アジャイル開発では要望に応えるため「変更が起きることを約束」します。つまり、将来「設計」しやすい現在の「設計」にする必要がある開発スタイルともいえます。

どちらがよいかはメリット/デメリットを考え選択すると良いです。

copyright ©ITCollege All Rights Reserved



設計の行為

- SOLID

前ページにもあったように開発プロセスに合わせて設計を行います。今回はSOLIDの原則に沿って一緒に設計を考えていきましょう。

Single Responsibility :単一責任

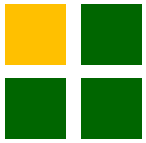
Open-Closed :オープンクローズド

Liskov Substitution :リスコフの置換

Interface Segregation :インタフェース分離

Dependency Inversion:依存性逆転

copyright ©ITCollege All Rights Reserved



SOLID

Single Responsibility

Open-Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

ITCollege

copyright ©ITCollege All Rights Reserved

Single Responsibility

- **Single Responsibility(単一責任のクラス)**

  システムの構造で最もはっきりしていて、目立つのは「クラス」です。

  ここではそのクラスに属するものをどのように決めるかについて取扱います。

  ①メソッドをグループに分けクラスにまとめる。

  ②**変更が簡単なよう**にコードを組成する。

  - 変更は副作用をもたらさない
  - 要件の変更が小さければ、コードの変更も相応して小さい
  - 既存のコードはかんたんに再利用できる
  - 最も簡単な変更方法はコードの追加

    ただし、そのコード自体変更が容易なものとする

ITCollege copyright ©ITCollege All Rights Reserved

Single Responsibility

「変更が簡単」というコードは以下のような性質を伴うべきとされています。

・見通しが良い(*Transparent*):

変更するコードにおいても、そのコードに依存する別の場所のコードにおいても、変更がもたらす影響が明白である。

・合理的(*Reasonable*):

どんな変更であっても、かかるコストは変更がもたらす利益にふさわしい。

・利用性が高い(*Usable*):

新しい環境、予期していなかった環境でも再利用できる。

・模範的(*Exemplary*):

コードに変更を加える人が、上記の品質を自然と保つようなコードになっている。

上記、それぞれの頭文字をとってTRUEコードと呼ばれます。

ITCollege copyright ©ITCollege All Rights Reserved

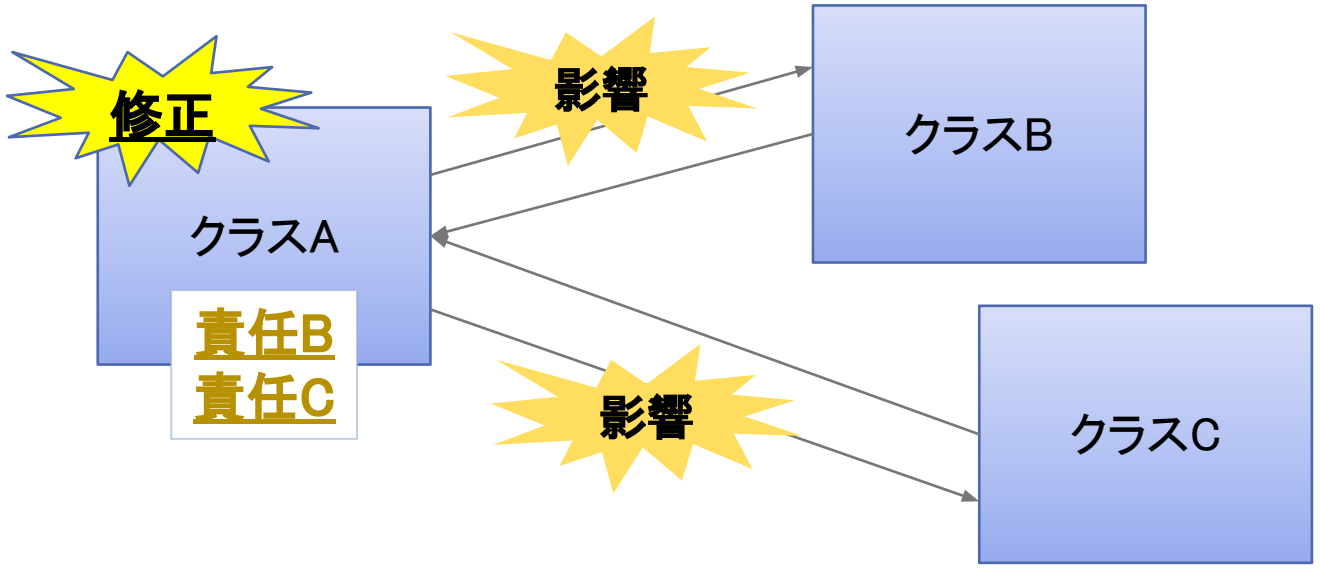
Single Responsibility

- **なぜ単一責任が重要なのか？**

  変更が簡単なクラス≒再利用が簡単なクラス

  ここで2つ以上の責任を持つクラスは簡単に再利用できません。多岐にわたる責任は内部構造に絡みついてしまいがちです。

  そして、いくつかの責任が絡み合った状態になっているため、変更が起こる理由はいくらでもあるのです。目的の用途とは関係のない理由で変更される可能性もあり、そのクラスに依存するすべてのクラスを破壊する可能性があります。



copyright ©ITCollege All Rights Reserved

Single Responsibility

- **クラスが単一責任か見極める**

①あたかもそれに知覚があるかのように仮定して問い正す。

クラスの持つメソッドを質問に言い換えた時に、意味を成す質問になっているべきです。

②1文でクラスを説明してみる。

クラスはできる限り最小で有用ということは、その説明は簡単にできるべきです。

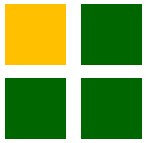
◆説明時の危険ワード

「それと」・・・おそらくクラスは2つ以上の責任を負っています。

「または」・・・クラスは2つ以上の責任を負い、互いに関連もしない責任を負っています。

もし、取り除けない余計な責任を見つけたら、隔離しましょう。

ITCollege copyright ©ITCollege All Rights Reserved



Single Responsibility

- **変更を歓迎するコードを書く**

①データではなく、振舞いに依存する。

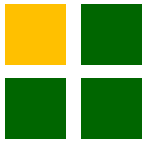
カプセル化が良い例ですね。フィールドを隠蔽化してメソッドによりデータの変更や取得を行います。

②あらゆる箇所を単一責任にする。

理由はクラスの時とまったく同じで、単一責任であることによって、メソッドの変更も再利用も簡単になるからです。

次ページで単一責任のメソッドがもたらす恩恵を記載します。

ITCollege copyright ©ITCollege All Rights Reserved



Single Responsibility

- **単一責任のメソッドのメリット**

・隠蔽されていた性質を明らかにする。

メソッドが単一の目的を果たすことによって、クラスが行うこと全体がより明確になります。

・コメントをする必要がない。

過去のものになったコメントを見たことは何度あるでしょうか？もしメソッド内のコードにコメントが必要ならそのコードを別のメソッドに抽出しましょう。その新しいメソッドの名前が当初のコメントの目的を果たします。

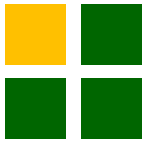
・再利用を促進する。

小さなメソッドはアプリケーションにとって健康的なコードの書き方を促進し、他のプログラマは再利用することでしょう。そういった流れにより小さく作るコードの書き方はおのずと広まっていきます。

・他のクラスへの移動が簡単。

小さなメソッドは簡単に移動できます。結果、振舞いの再編成も容易になります。

copyright ©ITCollege All Rights Reserved



Single Responsibility

- **単一責任に分離してみましょう。**

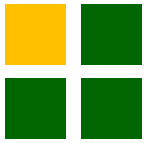
  今までのお話を踏まえて以下のインタフェースを分離してみましょう。

  処理の中身は考えなくて問題ありません。どのように分離するかメモ程度残しておけば大丈夫です。

  今回はコメントも記載しました。

```
interface Modem {
    //接続の開始
    public void dial(String pno);
    //接続の終了
    public void hangup();
    //送信
    public void send(char c);
    //受信
    public char recv();
}
```


copyright ©ITCollege All Rights Reserved



Single Responsibility

- Sample

  あくまで一例(※)ですが、接続管理と送受信というレベルで分離しました。

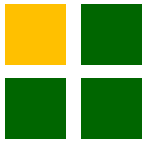
  ※今後このアプリケーションがどのように変更されるかによって変わってきます。

```
interface DataChannel {
    public void send(char c);
    public char recv();
}
```

```
interface Connection {
    public void dial(String pno);
    public void hangup();
}
```

```
interface ModelImpl {
    //抽象メソッドの実装
}
```

ITCollege copyright ©ITCollege All Rights Reserved



SOLID

Single Responsibility

Open-Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

copyright ©ITCollege All Rights Reserved

Open-Closed

- **閉鎖開放原則**

拡張	Open	**[開いている]**：機能を追加
変更	Closed	**[閉じている]**：コードを変更しない

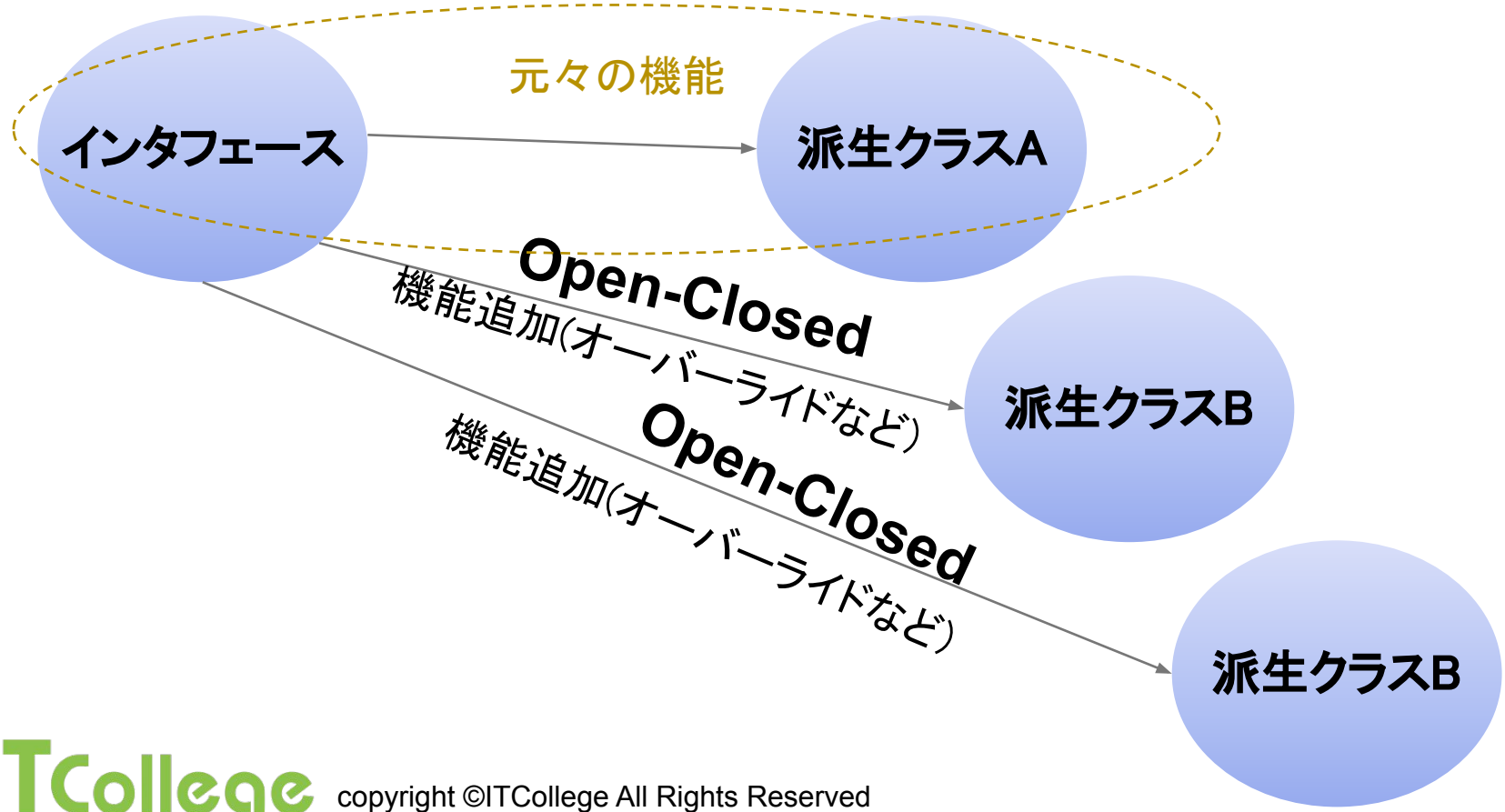
例）ユーザから機能拡張リクエストがあった場合、

ITCollege copyright ©ITCollege All Rights Reserved

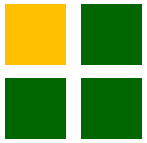
Open-Closed

- **抽象に依存する**

この原則はメソッドをある固定した「抽象」に従属させておけば修正に対してコードを閉じることができます。理由は、コードを修正しなくても派生クラスを新たに追加するだけでメソッドの振舞いを拡張できるからです。



ITCollege copyright ©ITCollege All Rights Reserved



Open-Closed

- **抽象を行う時の注意点**

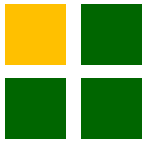
色々な意味で、この原則はオブジェクト指向の核心ともいえます。この原則に従うことでオブジェクト指向技術から得られる最大の利益(柔軟性、再利用性、保守性)を享受できます。一方で何でもかんでも「抽象」化しないことも「抽象」を使うのと同等に重要なことと言われています。

そして一番重要な概念としては、

「変更が起きた時に修正が少なくすむように設計する」

これに尽きると思います。

ITCollege copyright ©ITCollege All Rights Reserved



SOLID

Single Responsibility

Open-Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

copyright ©ITCollege All Rights Reserved

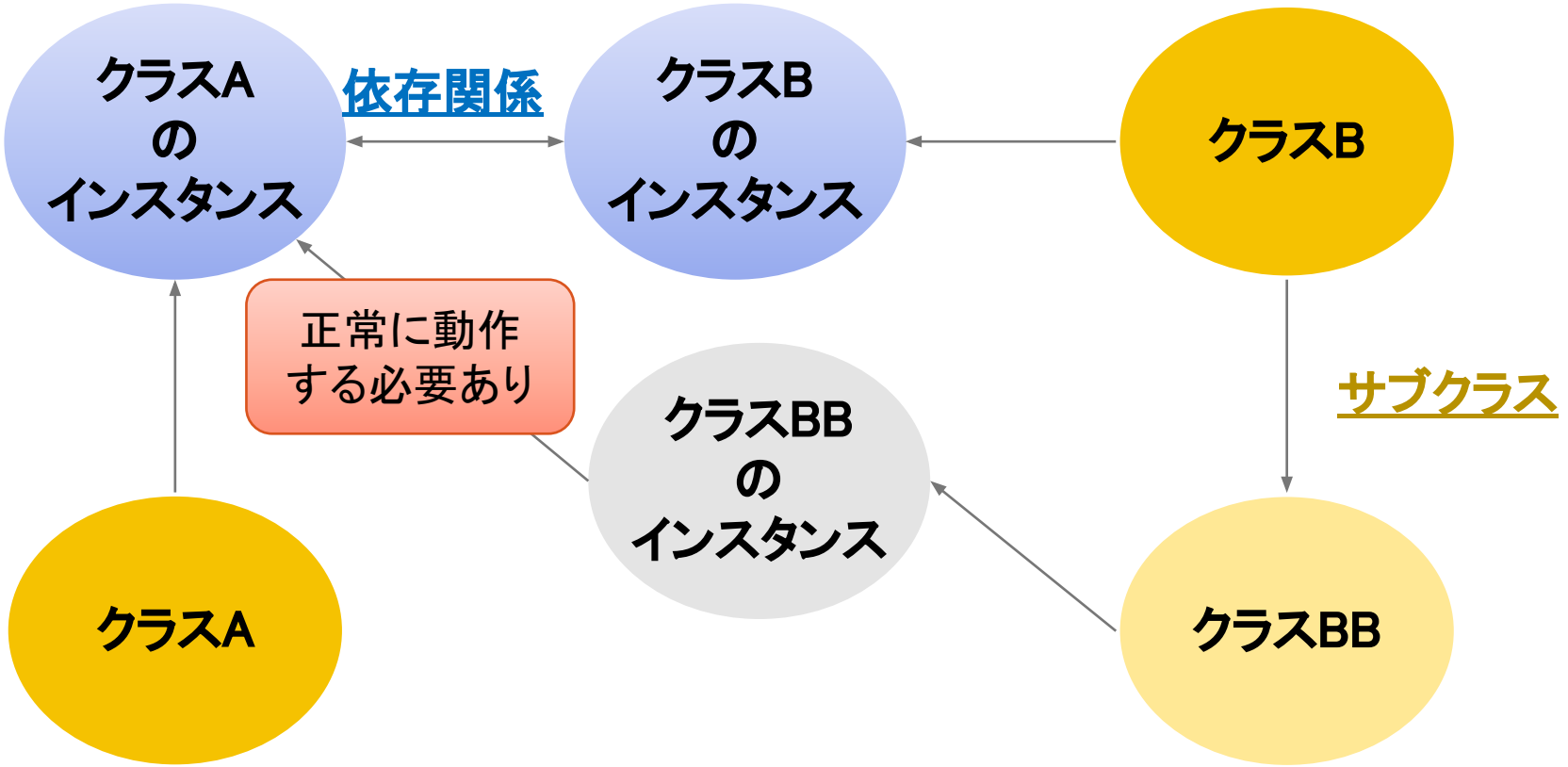
Liskov Substitution

copyright ©ITCollege All Rights Reserved

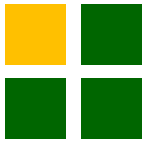
Liskov Substitution

- **置換するもの**

一言でいえばスーパークラスをサブクラスで置換できなければならないという原則です。



ITCollege copyright ©ITCollege All Rights Reserved



Liskov Substitution

前ページの例ではオーバーライドされた結果、動作が変わってしまう可能性があるようなメンバを公開してはならないため、公開する必要がなければprivateにしましょう。もちろんJavaの場合、finalも選択肢に入れてもよいでしょう。

そして、サブクラス作成時に注意することは、

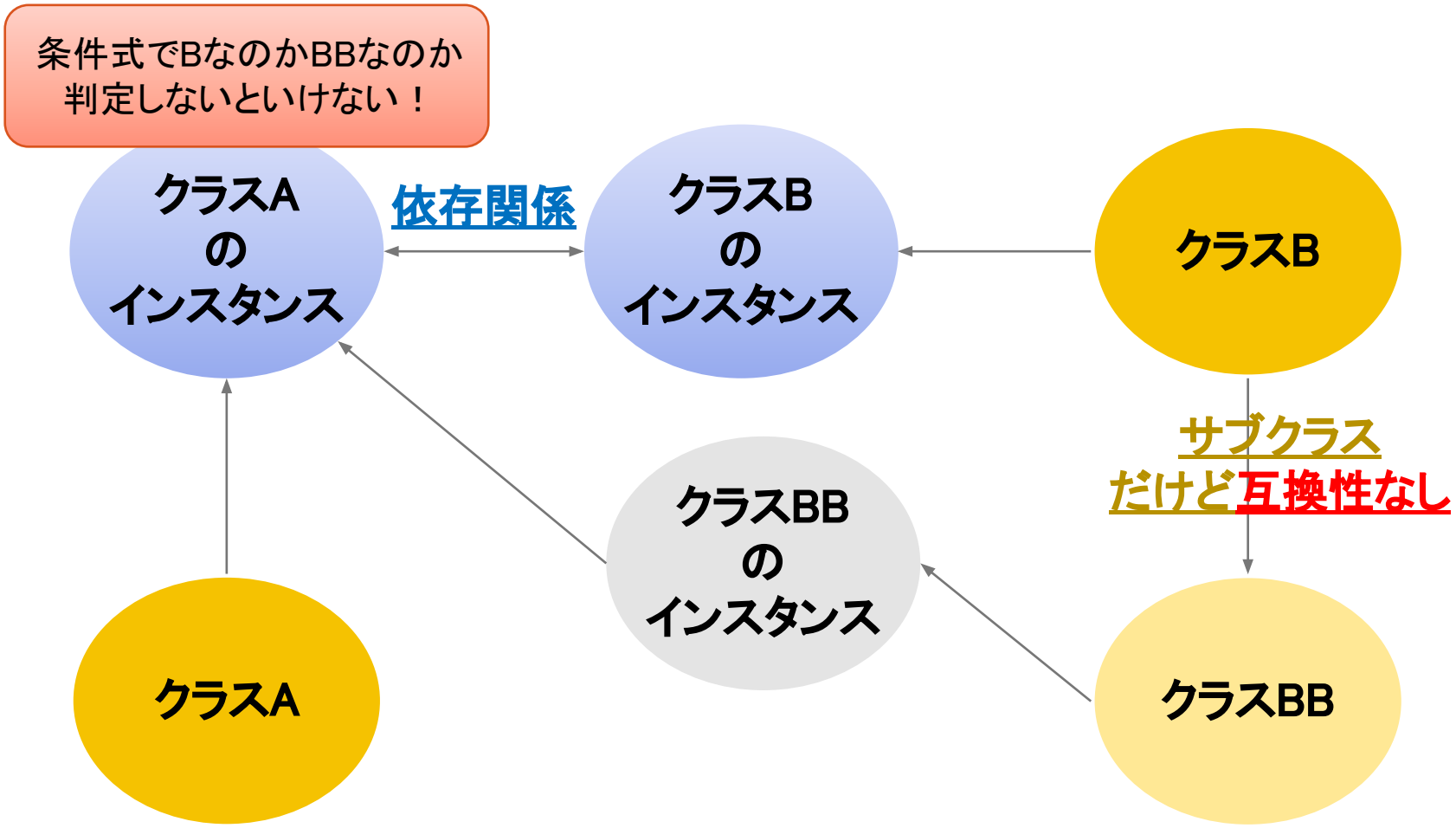
スーパークラスの責任から大きく外れるようなオーバーライドはしないということです。これはスーパークラスの質とアプリケーションがどのように変更されるかも考えなければいけません。

このようにサブクラスでもスーパークラスでも渡されたときにうまく動作することを必ず達成しなければなりません。これが置換可能ということです。

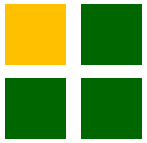
copyright ©ITCollege All Rights Reserved

Liskov Substitution

さらにこの原則を守ることは、Open-Closedの原則を守るために必要なことでもあります。



ITCollege copyright ©ITCollege All Rights Reserved



SOLID

Single Responsibility

Open-Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

ITCollege

copyright ©ITCollege All Rights Reserved

Interface Segregation

- **メソッドへの依存性を強制してはならない**

インタフェースは本来それぞれ独立し、**変更の影響を受けない**という特徴を持ちます。

例えば、会社での会話の仕方と自宅での会話の仕方は異なりますよね。

話し方に厳しい新しい上司が入って、会社で話し方を変えたからといって、自宅での会話の仕方は変わりませんよね。これは人間の脳内でそれぞれが分離されているからです。インタフェースも同じことがいえます。

このように使うクライアントによってインタフェースは分離されているべきというのがこの原則になります。

ITCollege copyright ©ITCollege All Rights Reserved

Interface Segregation

- **実装として考えてみる**

インタフェースが分離できず肥大化してしまう問題は、継承階層のあり方にも影響します。継承階層の上位にあるクラスが一部のクライアントしか利用しないメソッドを持っていると全てのサブクラスがそれを実装しなければならなくなります。

右記のように使わない場合は、

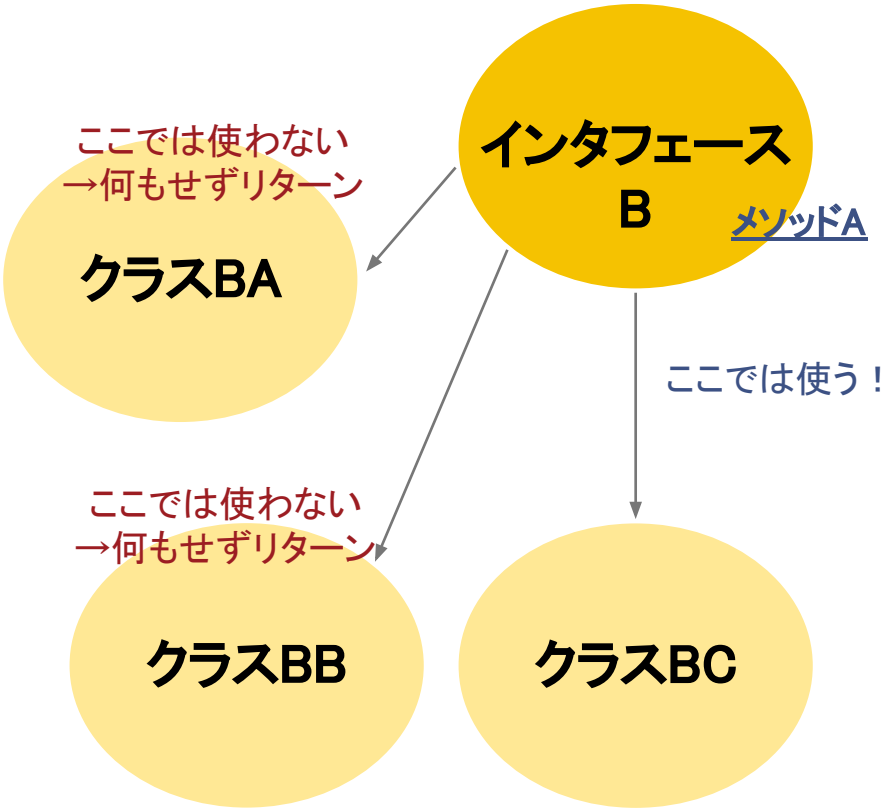
例外を投げたり、

何もせずリターンすることに

なります。

これはインタフェースの退化で

リスコフの原則にも違反します。

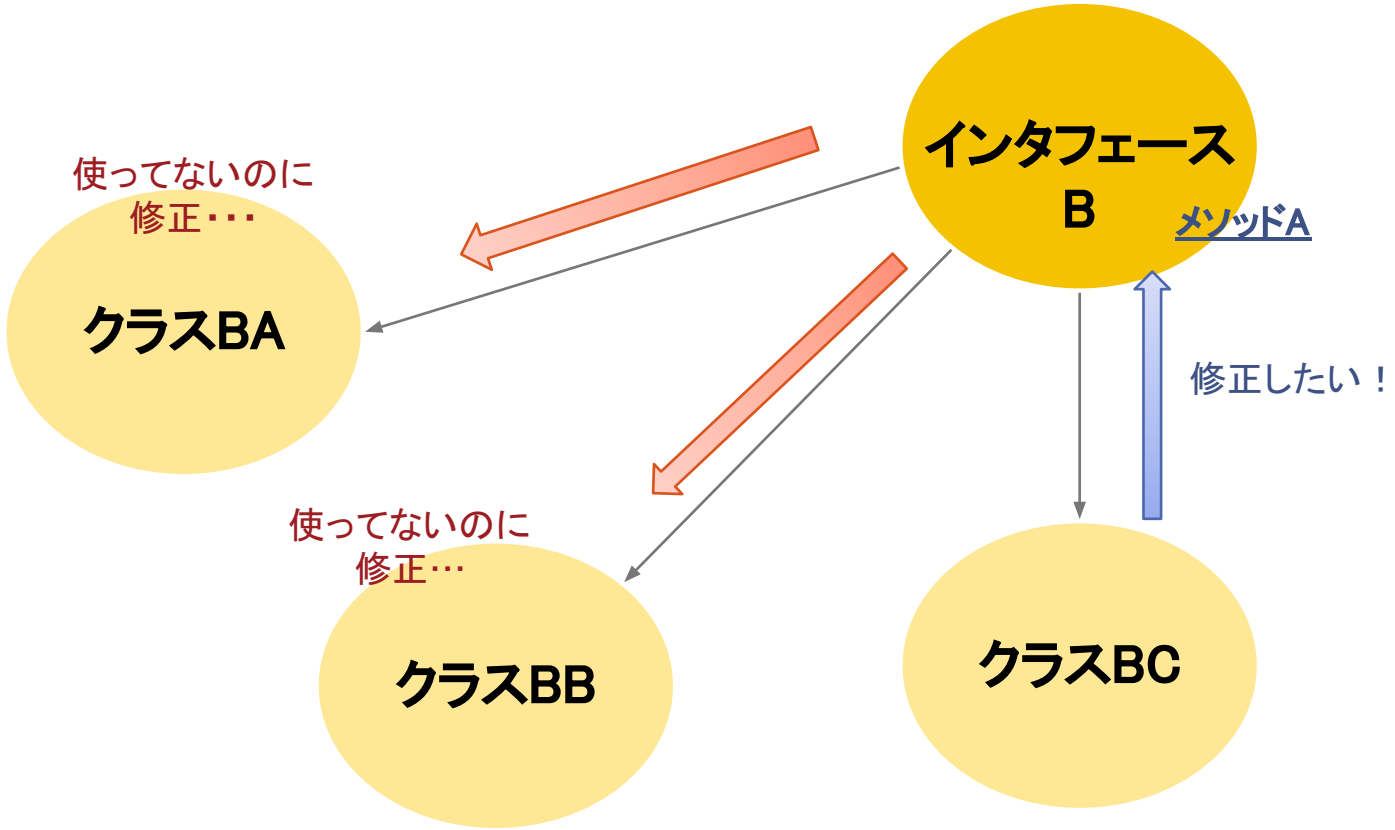


copyright ©ITCollege All Rights Reserved

Interface Segregation

ここで一番怖いのは、クラスBCでメソッドを変更したいとなった場合、実装しているクラスBA、クラスBBにも影響が波及していく悪夢のような状況となってしまうことです。

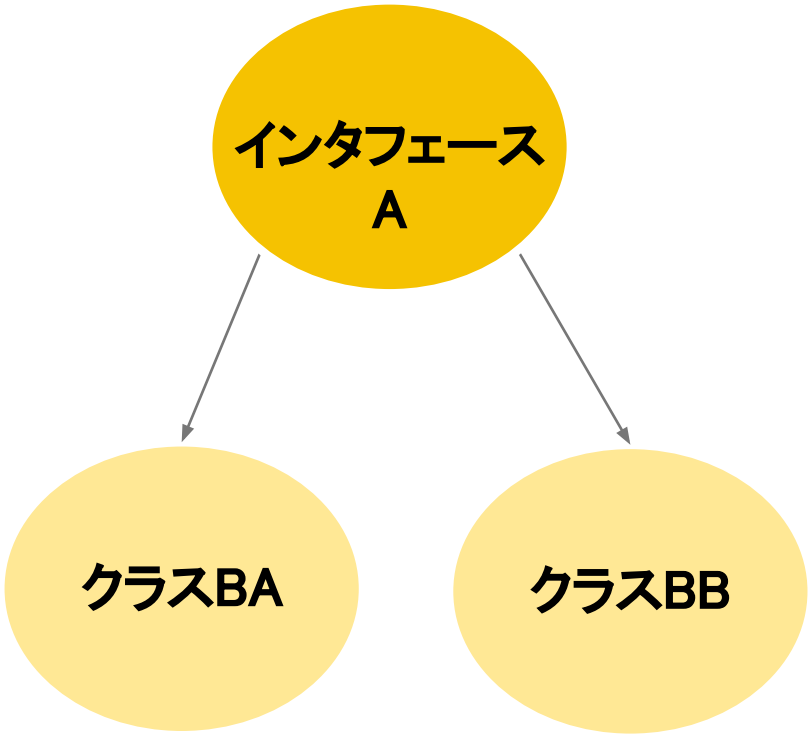
こうならないようにしないといけませんね。

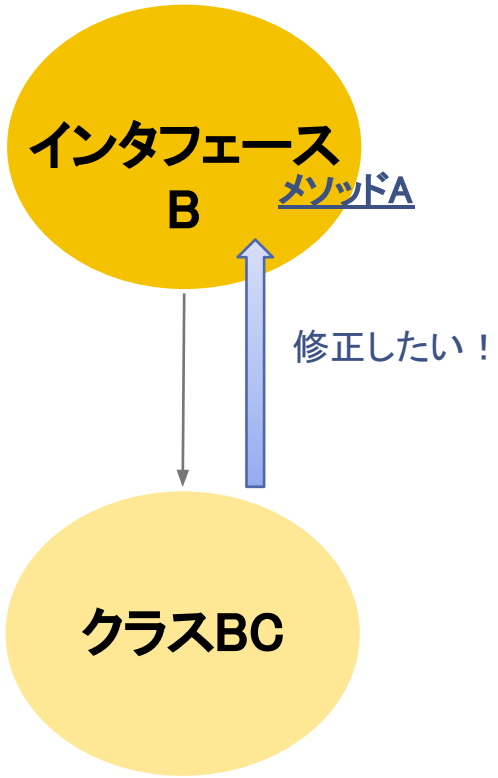


ITCollege copyright ©ITCollege All Rights Reserved

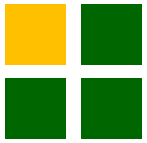
Interface Segregation

以下のように使うものと使わないものを明確にし、インタフェースの役割を適切に分離しましょう。





ITCollege copyright ©ITCollege All Rights Reserved



SOLID

Single Responsibility

Open-Closed

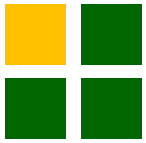
Liskov Substitution

Interface Segregation

Dependency Inversion

ITCollege

copyright ©ITCollege All Rights Reserved



Dependency Inversion

- **依存関係逆転の法則**

現在ではSeasar2やSpringなどの有名フレームワークでDI(依存性注入)の考え方が取り入れられていますが、これはこの依存関係逆転の考え方を一歩進めたものです。フレームワークの意義を理解するためにも理解しておきたい原則となります。

原則は以下の通りです。

①上位のモジュールは下位のモジュールに依存してはならない。どちらのモジュールも「抽象」に依存すべきである。

②「抽象」は実装の詳細に依存してはならない。実装の詳細が「抽象」に依存すべきである。

これだと少しわかりづらいですね。もうちょっと実生活を伴った例を説明していきます。

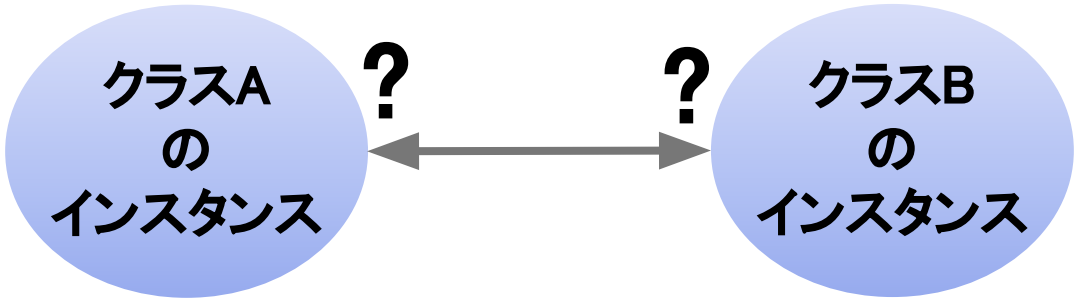
copyright ©ITCollege All Rights Reserved

Dependency Inversion

- **依存の方向**

原則の説明の前に・・・・

オブジェクトはお互いに依存しあっていることで力を発揮しています。ただし、その依存には方向というものがあります。



AインスタンスがBインスタンスと依存していたとして、AインスタンスがBインスタンスに依存しているのか、それとも逆にBインスタンスがAインスタンスに依存しているのかという関係があります。

これが依存の方向となります。覚えておきましょう。

copyright ©ITCollege All Rights Reserved

Dependency Inversion

- **実生活の例**

実生活ではたいてい細かいものが大きなものに依存しているという方向があります。その一例としてチェストを見てみます。

右記のチェストは枠があって、引出しが3つあります。

この場合依存関係はどうなっているでしょうか？

枠→引出し？　引出し→枠？

・・・どっちも違います。最初に設計図があると思いませんか？

設計図には枠のどこにどういう穴を作成しておけば、引出しがその穴に当てはまるように作られるかが書いてあります。

この時、穴が空っぽの空間のため、設計図には書いてあるけど(引出し穴の想定のため)実際には存在しない抽象的なものになります。枠はこの想定した引出し穴というインタフェース(抽象的なもの)に依存しています。引出し自体もこの穴に依存しています。

copyright ©ITCollege All Rights Reserved

Dependency Inversion

- **実生活の例**

つまり、以下のことがいえます。

①チェストの枠は引出しに依存してはならない。どちらも引出し穴に依存すべきである。

②引出し穴は枠や引出しに依存してはならない。枠や引出しが引出し穴に依存すべきである。

この内容はさきほどの原則の内容と一致します。

パッと見、引出しと枠が依存しあっているように見えますが、設計図上にしか存在しない引出し穴(インタフェース)に依存しているといえます。

そのため前ページで出た文を言い換えてみます。

「細かいオブジェクトは大きなオブジェクトが持っているインタフェースに依存している」

というようにいえます。

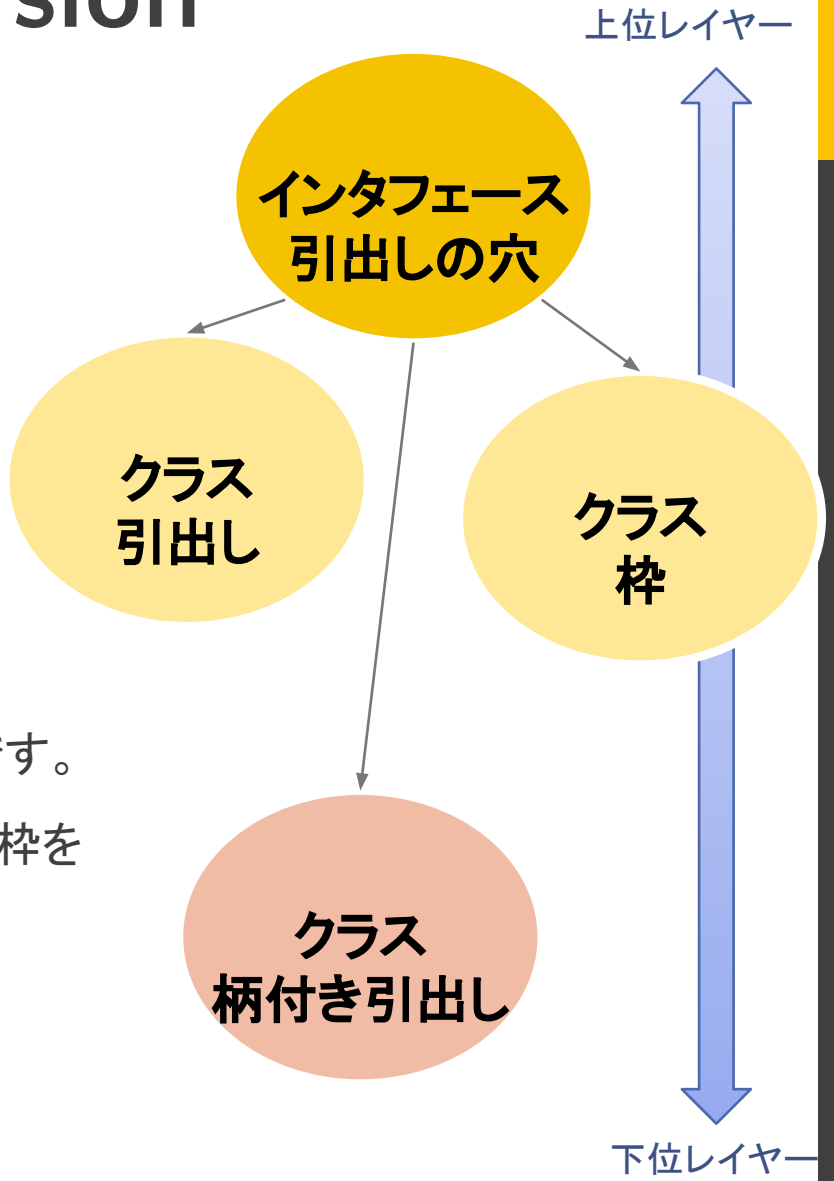
ITCollege copyright ©ITCollege All Rights Reserved

Dependency Inversion

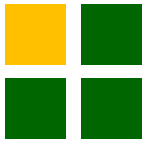
- **重要なこと**

今の例でもわかるように上位レイヤー(層)は、
「自分が必要とするものはこれ！」と
宣言するためにインタフェースを持ち、
下位レイヤーはそれに従って
詳細を作っていきます。

つまり主導権は上位レイヤーが握るべきなのです。
そうすれば右記のように別の色(柄)の引出しや枠を
用意することも容易です。



ITCollege copyright ©ITCollege All Rights Reserved



SOLIDまとめ

最初に説明したようにどんなプログラムも変更が付いて回ってきます。

そこで変更した場所以外には影響を与えないように設計することが重要です。

それを実現するためには、

①オブジェクト指向設計の理解

変更が容易なコードが適切に設計されているといえます。そのためには、オブジェクト指向設計をよく理解している必要があります。

②コードのリファクタリングを継続的に行う

ソフトウェアの外部の振る舞いを保ったままで内部の構造を改善していくことが重要です。継続的かつ正確に行うことで新たな要件にまた対応しやすくなります。

copyright ©ITCollege All Rights Reserved

実際の設計例

- **今と過去の設計について**

今は私達プログラマが必死に設計内容を考えてシステムを開発しています。

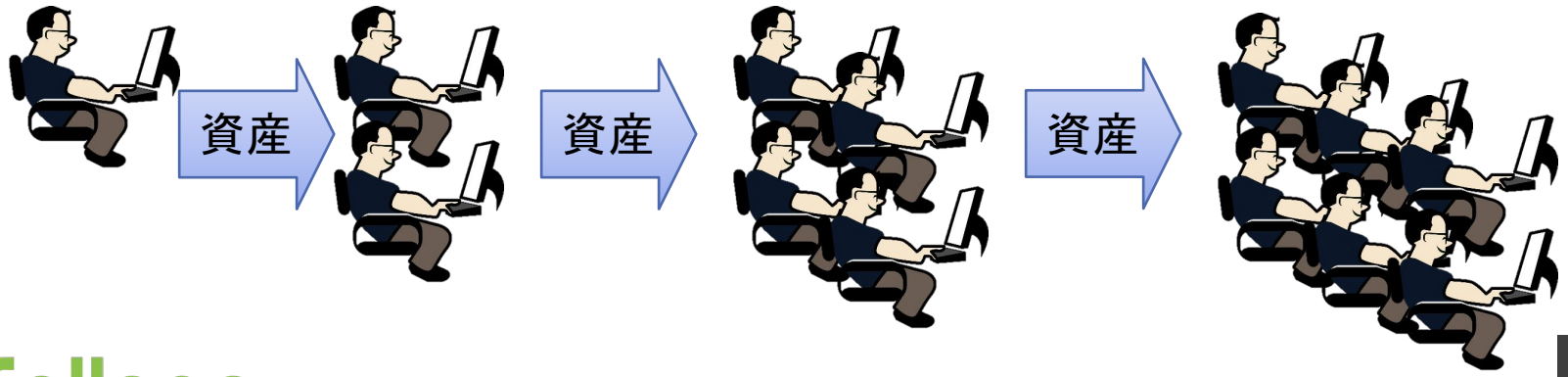
しかし、先人達はどのように行っていたのでしょう。

基本的には私達と一緒です。

必死に考え付いた設計内容というものが存在します。

そして、それをパターン(テンプレートのようなもの)として残しています。

こういったものを**デザインパターン**といい、便利な使い方を残してくれています。
これを使うかどうかは私達次第ですが、せっかくなので偉大な先人達の知恵を拝借していきましょう。



ITCollege copyright ©ITCollege All Rights Reserved

デザインパターン

- **目的**

①再利用性の高い柔軟な設計をできるようにするため

②技術者同士の意思疎通を容易にするため

上記目的を達成をすることために今までの開発者達が蓄積していったものです。デザインパターンを理解することで「頻出問題とその対応策になる設計」の知識を増やし、再利用性が高く、柔軟な設計を行うことができます。

全て覚えることは困難なため、比較的利用頻度の高いパターンに着目していきます。

copyright ©ITCollege All Rights Reserved

UML

- **まずはじめに・・・**

デザインパターンに登場するクラスとインスタンスの関係を表現するために
UML(Unified Modeling Language:統一モデリング言語)というものを使います。

UMLとは、システムを視覚化したり、仕様や設計を文書化したりするための表現方法です。

ここではUMLのほんの一部だけ使って説明していきます。

詳細をもっと知りたい方は以下ページを参照してください。

・UML Resource Page

http://www.uml.org/

ITCollege copyright ©ITCollege All Rights Reserved

UML

- **様々な表現方法**

構造図

:静的なシステム構造をモデルで表現します。

(例)クラス図、コンポーネント図など

振る舞い図

:システムの振る舞いをモデルで表現します。

(例)アクティビティ図、ユースケース図など

相互作用図

:振る舞い図の一種ではあるがオブジェクト間のデータ受け渡しをモデルで表現します。

(例)シーケンス図など

ITCollege copyright ©ITCollege All Rights Reserved

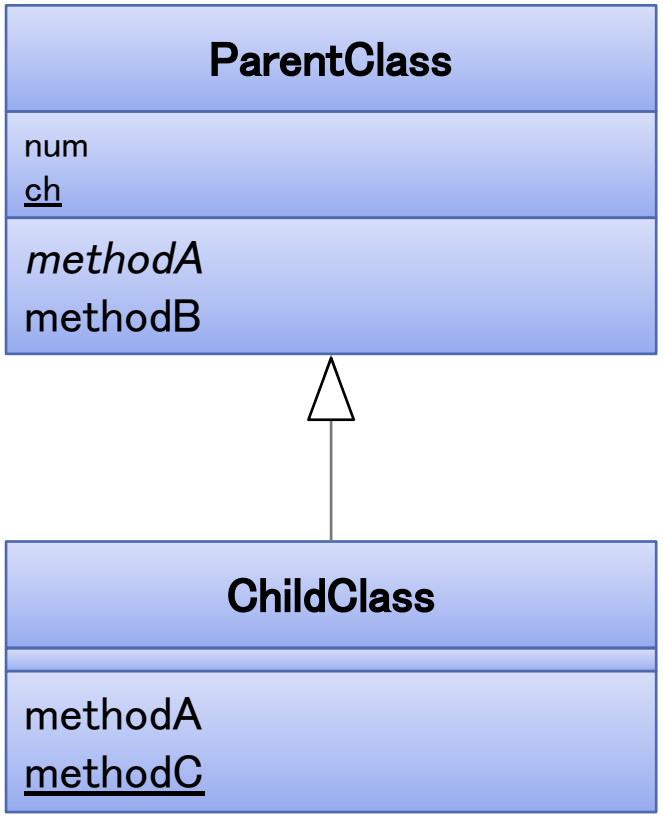
クラス図

- **クラス図(Class Diagram)**

  クラスやインスタンス、インタフェースなどの静的な関係を表現したものです。クラス図という名前で呼ばれていますが、登場するのはクラスだけではありません。

```
abstract class ParentClass{
    int num;
    static char ch;
    abstract void methodA();
    double methodB(){
    }
}
```

```
class ChildClass extends ParentClass{
    void methodA(){
    }
    static void methoC(){
    }
}
```



ITCollege copyright ©ITCollege All Rights Reserved

クラス図

この図はParentClassとChildClassという2つのクラスの関係を表しています。白抜きの△が付いた実線の矢印はクラスの階層関係を表しています。矢印はサブクラスからスーパークラスへ向かっています。(いうなれば、extendsの矢印)

クラスの持っている情報は長方形の水平線で分割され、

・クラスの名前

・フィールドの名前

・メソッドの名前

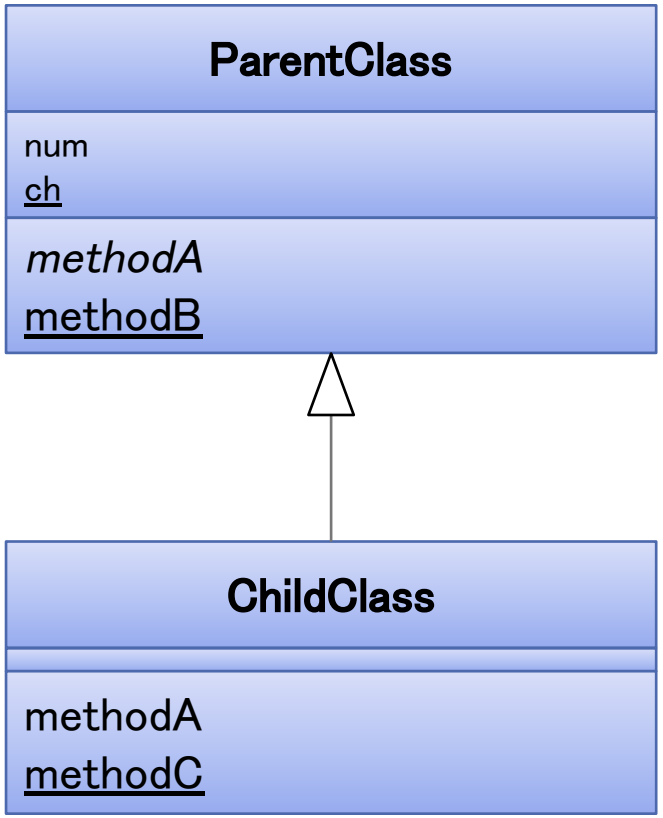
が表記されます。

プログラムと見比べれば分かるように、

抽象クラス、抽象メソッドは斜字体で書きます。

クラスフィールド、クラスメソッドは

下線を書きます。



ITCollege copyright ©ITCollege All Rights Reserved

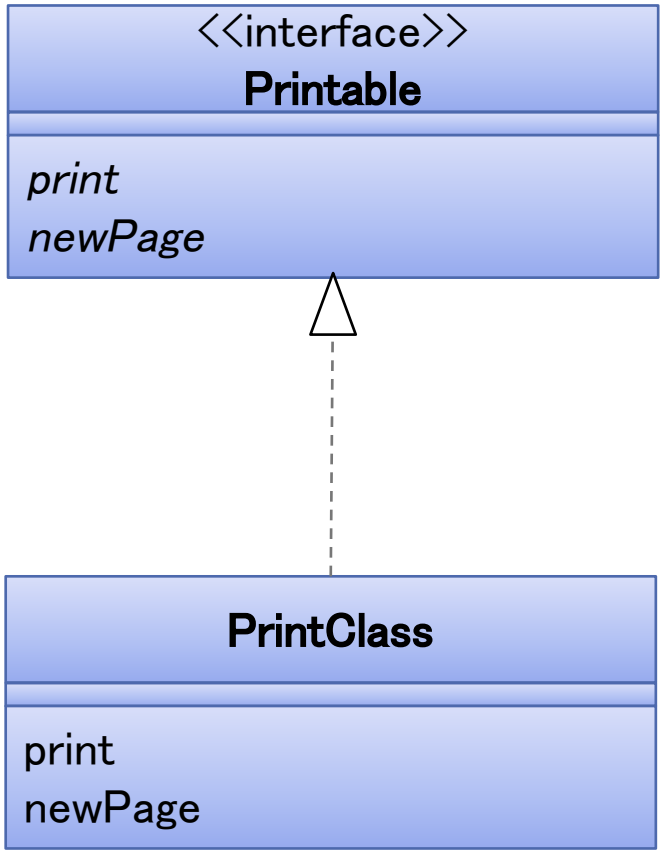
クラス図

- **インタフェースと実装**

次にインタフェースの表現方法を見てみましょう。

```
interface Printable{
    abstract void print();
    abstract void newPage();
}
```

```
class PrintClass impements Printable{
    void print(){
    }
    void newPage(){
    }
}
```



ITCollege copyright ©ITCollege All Rights Reserved

クラス図

- **集約**

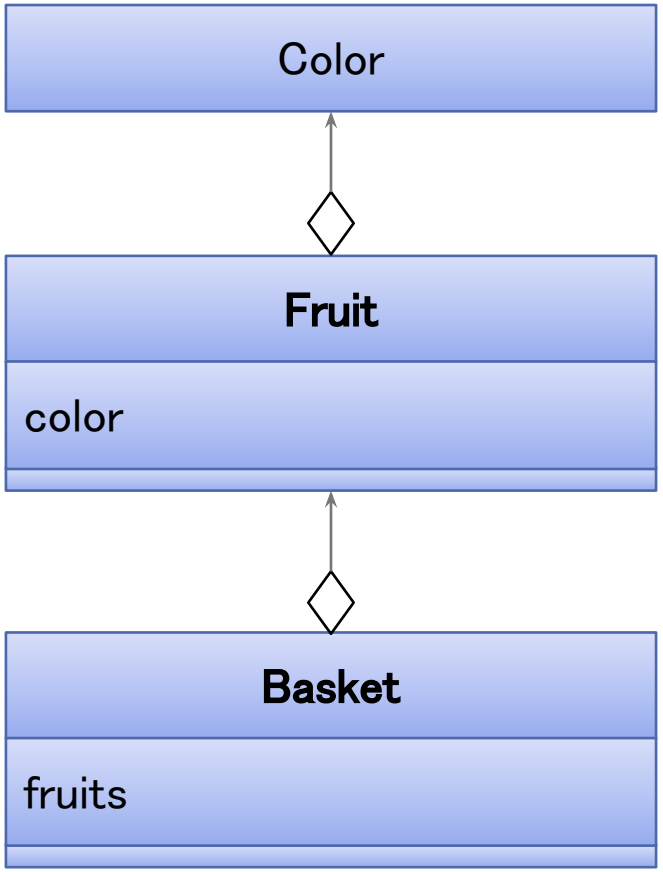
  各クラスを元に生成されたインスタンスがフィールドとしてインスタンスを持っている場合、「集約」という関係になります。

  以下のように個数に関係なく関係は「集約」です。

```
class Color{
    //--------
}
```

```
class Fruit{
    Color color;
    //--------
}
```

```
class Basket{
    Fruit[] fruits;
    //--------
}
```



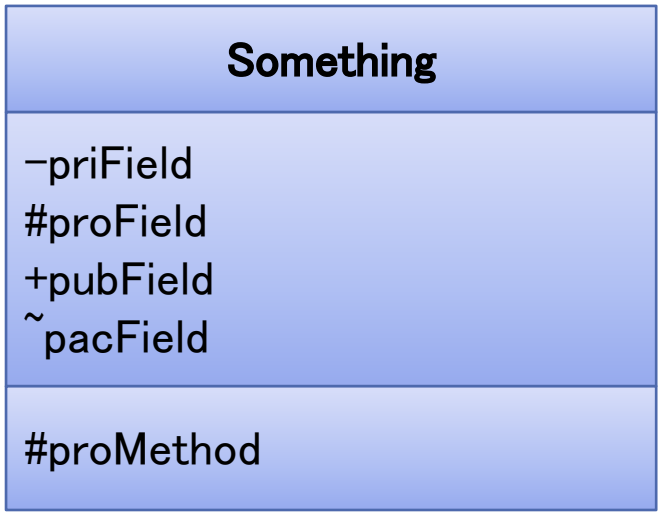
ITCollege copyright ©ITCollege All Rights Reserved

クラス図

- **アクセス制御**

  アクセス制御を表現したい場合、メソッドやフィールドの名前の前に記号を付けます。-がprivate、#がprotected、+がpublic、~が付いている場合は、同一パッケージからのみアクセスできるメソッドやフィールドを表します。

```
class Something{
    private int priField;
    protected int proField;
    public int pubField;
    int pacField;
    protected void proMethod(){
    }
}
```

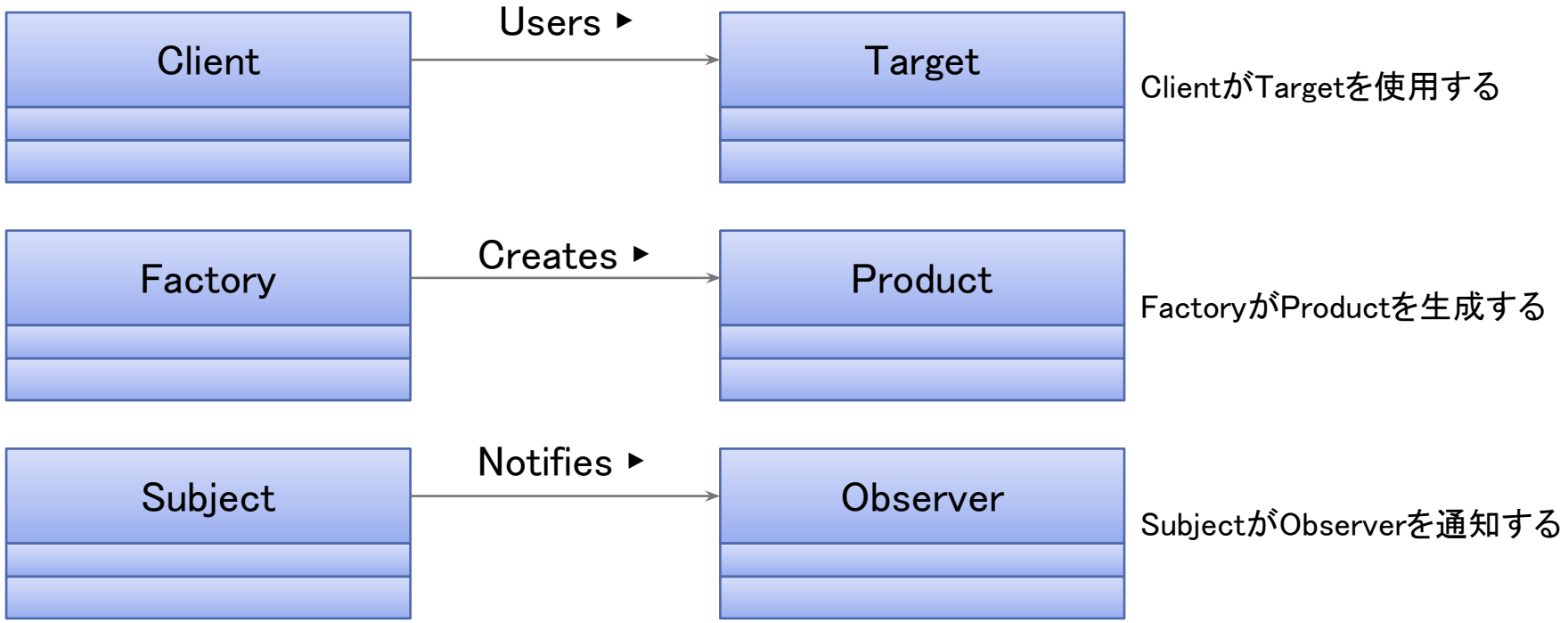


ITCollege copyright ©ITCollege All Rights Reserved

クラス図

- **クラスの関連**

  クラスの関連を示すために黒三角をつけて表記することができます。



ITCollege copyright ©ITCollege All Rights Reserved

補足

- **クラス図の矢印の向き**

  サブクラスからスーパークラス

  が基本の向きになります。

  この時にプログラムを作成したことがある人なら感じると思いますが、スーパークラスを元にサブクラスを作成することから、逆の方が理解しやすいと感じる人もいるかもしれません。

  そのため、ちょっとしたコツですが、サブクラスを定義するとき、extendsでスーパークラスを指定します。そのため、サブクラスは必ずスーパークラスを知っています。しかし、スーパークラスはサブクラスを知っているとは限りません。

  「相手を指さすことができるのは相手を知っている時だけ」

  このような考えを持って作成してみてはいかがでしょう？

copyright ©ITCollege All Rights Reserved

シーケンス図

- **目的**

プログラムが動くときに、どのメソッドがどういう順番で実行されるか、どのような事象がどういう順番で起きるかを表現したものです。

クラス図は時間で変化しないものを表現するのに対して、シーケンス図は「時間に従って変化するもの」を表現します。

```
class Client{
    Server server;
    void work(){
        server.open();
        server.print( “Hello” );
        server.close();
    }
}
```

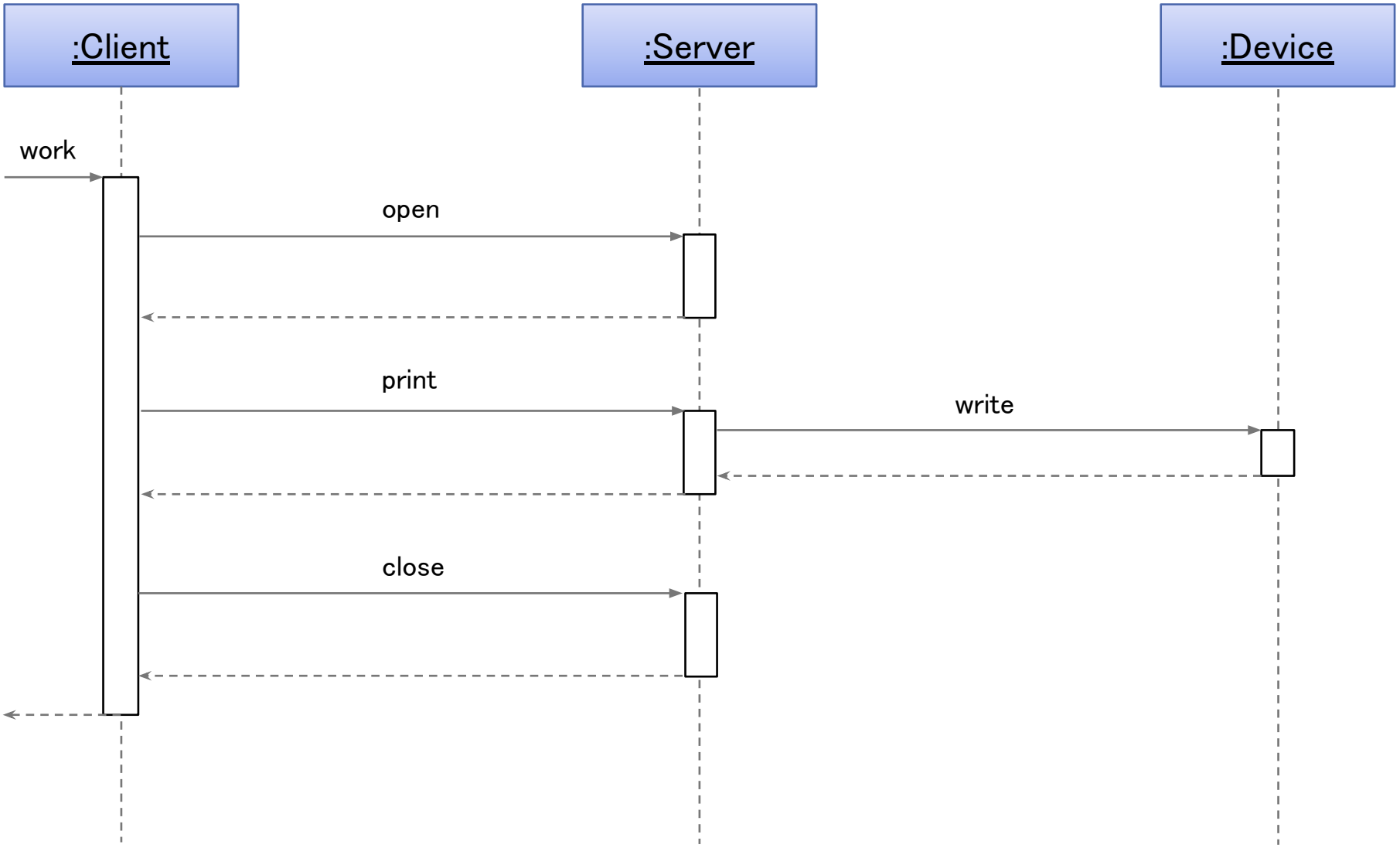
```
class Server{
    Device device;
    void open(){
    }
    void print(String s){
        device.write(s);
    }
    void close(){
    }
}
```

```
class Device{
    void write(String s){
    }
}
```

上記プログラムの断片を次ページでシーケンス図として記載していきます。

ITCollege copyright ©ITCollege All Rights Reserved

シーケンス図



ITCollege copyright ©ITCollege All Rights Reserved

シーケンス図

コロン(:)の後にクラス名が書かれ、下線が付けられています。これはそれぞれのインスタンスを表現しています。

インスタンスを表現する長方形の下に破線が伸びています。

これは**ライフライン**と言って、インスタンスが存在する間だけ存在します。そのライフライン上に設置している長方形は、オブジェクトが活動中であることを表現しています。

続いて横方向の→についてですが、先端が黒塗りになっている三角形で表現しているものはメソッドの呼び出し、対して破線に矢印はメソッドからのリターン(戻り)を表現しています。

ITCollege copyright ©ITCollege All Rights Reserved

デザインパターン

- **種類**

デザインパターンは数多くありますが、比較的使用頻度の高いものだけ紹介していきます。

Singleton(ｼﾝｸﾞﾙﾄﾝ)

:たった1つのインスタンス

TemplateMethod(ﾃﾝﾌﾟﾚｰﾄﾒｿｯﾄﾞ)

:具体的な処理をサブクラスに任せる

FactoryMethod(ﾌｧｸﾄﾘﾒｿｯﾄﾞ)

:インスタンス作成をサブクラスに任せる

Iterator(ｲﾃﾚｰﾀ)

:1つ1つ数え上げる

ITCollege copyright ©ITCollege All Rights Reserved

デザインパターン

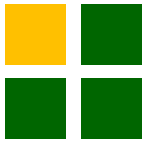
Singleton(ｼﾝｸﾞﾙﾄﾝ)

TemplateMethod(ﾃﾝﾌﾟﾚｰﾄﾒｿｯﾄﾞ)

FactoryMethod(ﾌｧｸﾄﾘﾒｿｯﾄﾞ)

Iterator(ｲﾃﾚｰﾀ)

ITCollege copyright ©ITCollege All Rights Reserved



Singleton

- **目的**

「このクラスのインスタンスはたった1つしか作らないし、作りたくない」という時に用います。

つまり

・指定クラスのインスタンスが絶対に1個しか存在しないことを保証したい

・インスタンスが1個しか存在しないことをプログラム上で表現したい

といえます。

複数のインスタンスを作る必要がない（複数あると困る）ようなクラスに使われるパターンです。

DBコネクションやファイル入出力アクセスなど、複数あるとトランザクションやファイルロックが複雑になるときなどに使われます。

ITCollege copyright ©ITCollege All Rights Reserved

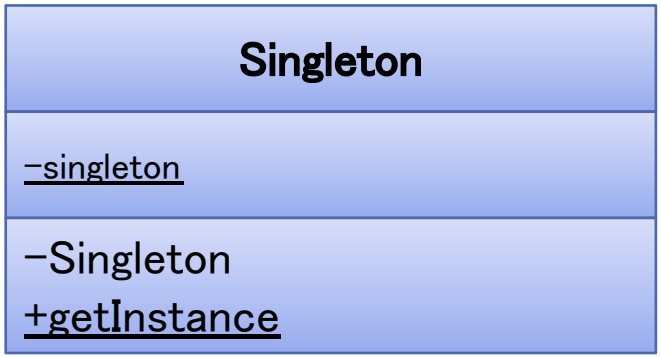
Singleton

- **クラス図**

実現するにはコンストラクタとフィールドをプライベートにします。そして唯一のインスタンスを保持するフィールドとそのフィールドにアクセスするメソッドをstaticにします。

これにより外部クラスからコンストラクタ呼び出しが不可能になり、外部からインスタンス生成できなくなります。

外部からこの唯一のインスタンスにアクセスするにはgetInstanceメソッドを使うしかできません。




```
public class Singleton{
    private static Singleton s = new Singleton();
    private Singleton(){
    }
    public static Singleton getInstance(){
        return singleton;
    }
}
```

ITCollege copyright ©ITCollege All Rights Reserved

デザインパターン

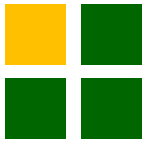
Singleton(シングルトン)

TemplateMethod(テンプレートメソッド)

FactoryMethod(ファクトリメソッド)

Iterator(イテレータ)

ITCollege copyright ©ITCollege All Rights Reserved



TemplateMethod

- **目的**

スーパークラスの方にテンプレートとなるメソッドが定義されています。

そして、サブクラス側で具体的な処理を決定します。

つまり、

・スーパークラスで処理の枠組みを定め、サブクラスで具体的内容を定める

といえます。

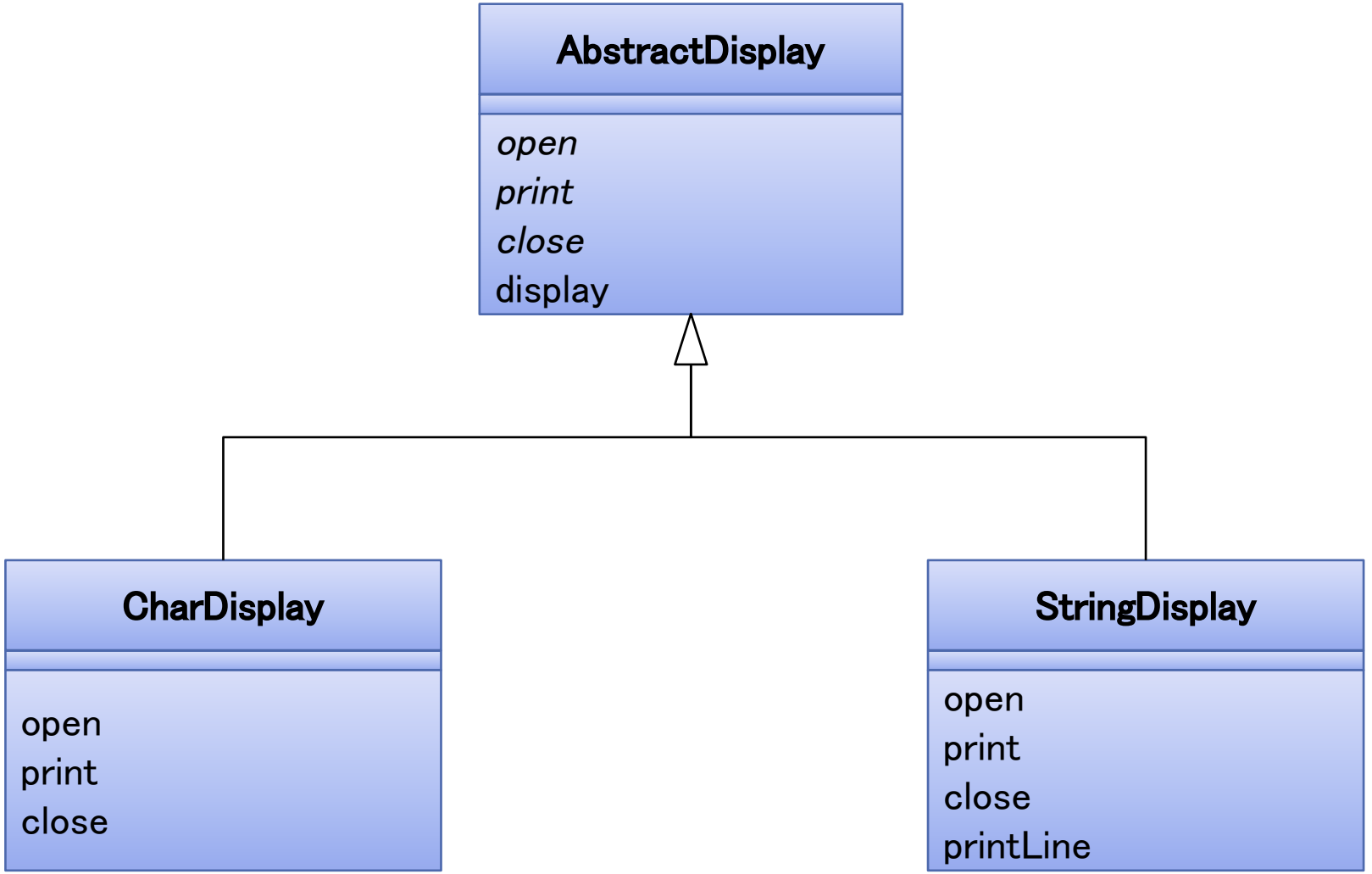
例えば、Aというクラスで【① → ② → ③ → ⑥】という処理があり、Bというクラスで【① → ② → ④ → ⑤ → ⑥】という処理を行っていたとします。

【① → ② → ～ → ⑥】は共通しているため、Sクラスで共通部分を作成、A、Bクラスは Sクラスを継承して、異なる部分を実装するという形がこのデザインパターンです。

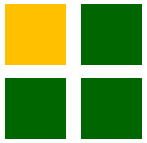
copyright ©ITCollege All Rights Reserved

TemplateMethod

- クラス図



ITCollege copyright ©ITCollege All Rights Reserved



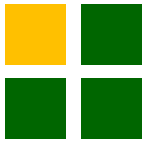
TemplateMethod

```
public abstract class AbstractDisplay{
    public abstract void open();
    public abstract void print();
    public abstract void close();
    public void display(){
        open();
        for(int i = 0;i < 5;i++){
            print();
        }
        close();
    }
}
```

```
public class CharDisplay extends AbstractDisplay{
    private char ch;
    public CharDisplay(char ch){
        this.ch = ch;
    }
    public void open(){
    }
    public void print(){
         //フィールドchを参照して表示
    }
    public void close(){
    }
}
```

```
public class StringDisplay extends AbstractDisplay{
    private String str;
    public StringDisplay(String str){
        this.str = str;
    }
    public void open(){
    }
    public void print(){
         //フィールドstrを参照して表示
    }
    public void close(){
    }
}
```

ITCollege copyright ©ITCollege All Rights Reserved



TemplateMethod

AbstractDisplayクラスではdisplayメソッドが定義されています。そして、displayメソッドの中ではopen、print、closeのメソッドが使われています。この3つのメソッドは抽象メソッドになっています。

ここでは、抽象メソッドを使っているdisplayメソッドが**テンプレートメソッド**になります。

どんなメリットがあるか考えてみましょう！

・テンプレートメソッドにアルゴリズムが書かれているため、ロジックが共通化できる。そして、修正もテンプレートメソッドに行えば実装しているクラスに反映される。

・サブクラスに実装を強制するため、実装漏れを防止できる。

・抽象クラスの段階で処理の流れを形作ることができる。

copyright ©ITCollege All Rights Reserved

デザインパターン

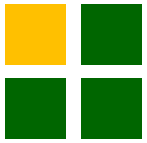
Singleton(シングルトン)

TemplateMethod(テンプレートメソッド)

FactoryMethod(ファクトリメソッド)

Iterator(イテレータ)

copyright ©ITCollege All Rights Reserved



FactoryMethod

- **目的**

インスタンスの作り方をスーパークラスの側で定めますが、具体的なクラス名までは定めません。具体的な内容はすべてサブクラス側で行います。

これにより、インスタンス生成の枠組み(フレームワーク)と実際のインスタンス生成のクラスとを分けて考えることができるようになります。

また、具体的なクラス名を定めないことで他のクラスでも使い回しすることができます。

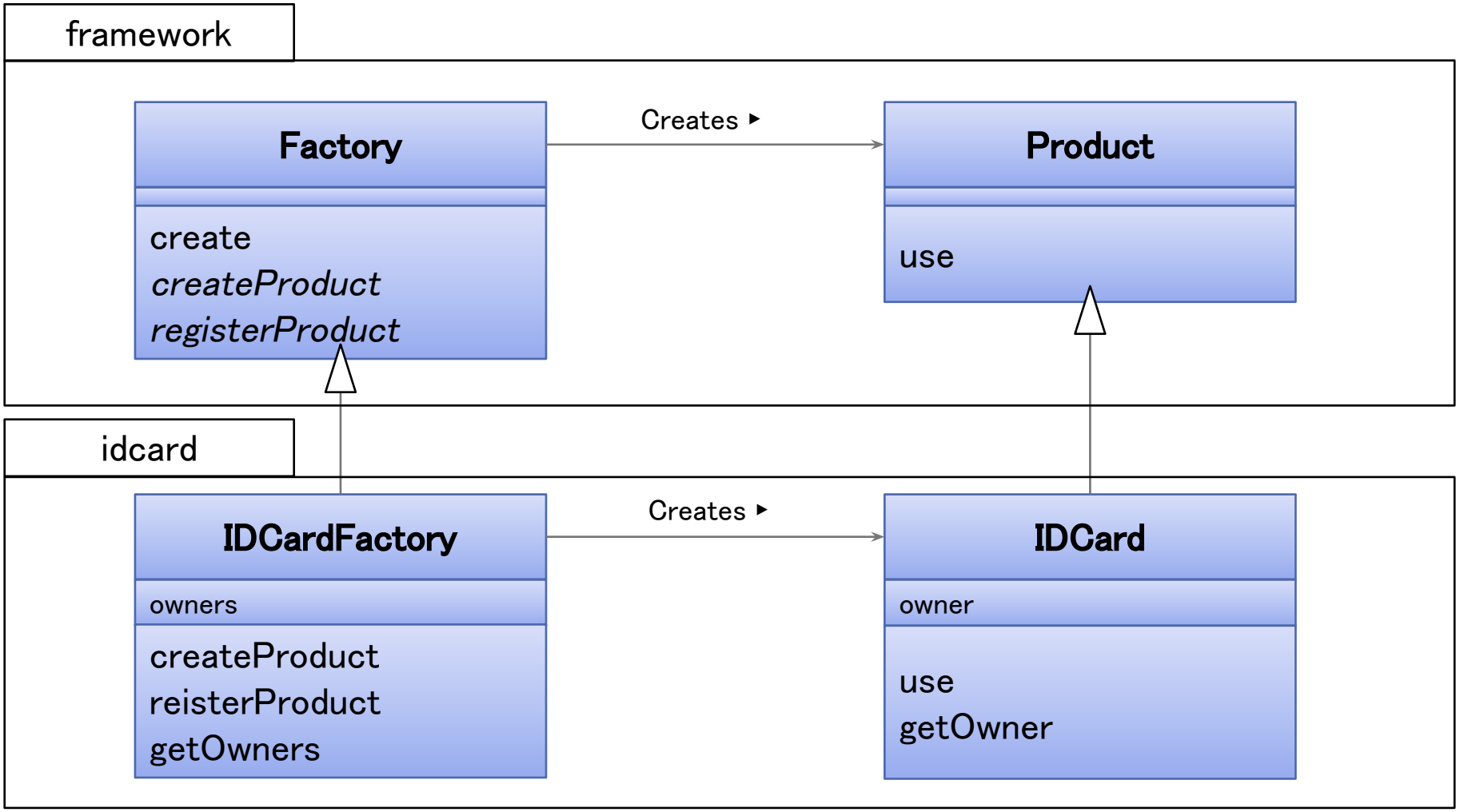
インスタンスを作成する場合、通常は【new A()】としますが、直接 new をするわけではなく、newをするためのクラスを作り、メソッドでインスタンスを取得するイメージです。
（ただし戻り値はスーパークラスとして、多態性を利用する）

インスタンス生成を1箇所に固めることで、仕様変更に耐えやすく依存度を低くする効果があります。

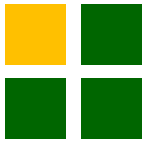
copyright ©ITCollege All Rights Reserved

FactoryMethod

- クラス図



ITCollege copyright ©ITCollege All Rights Reserved



FactoryMethod

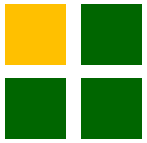
```
package framework;
public abstract class Product{
    public abstract void use();
}
```

```
package framework;
public abstract class Factory{
    public final Product create(String owner){
        Product p = createProduct(owner);
        registerProduct(p);
        return p;
    }
    protected abstract Product createProduct(String owner);
    protected abstract void registerProduct(Product product);
}
```

```
package idcard;
import framework.*;
public class IDCard extends Product{
    private String owner;
    IDCard(String owner){
        //カード作りますと表示
        this.owner = owner;
    }
    public void use(){
        //カード使いますと表示
    }
    public String getOwner(){
        return owner;
    }
}
```

```
package idcard;
import framework.*;
import java.util.*;
public class IDCardFactory extends Factory{
    private List owners = new ArrayList();
    protected Product createProduct(String owner){
        return new IDCard(owner);
    }
    protected void registerProduct(Product product){
        owner.add(((IDCard)product).getOwner());
    }
    public List getOwners(){
        return owners;
    }
}
```

ITCollege copyright ©ITCollege All Rights Reserved



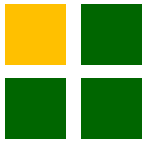
FactoryMethod

FactoryクラスとProductクラスはframeworkパッケージに属しています。この2つのクラスがインスタンス生成のための枠組みの役割を果たします。

idcardというパッケージに属しているIDCardクラスとIDCardFactoryクラスは実際の肉付けを行います。

これにより、newによる実際のインスタンス生成をインスタンス生成のためのメソッド呼び出しに代えることで、**具体的なクラス名による束縛からスーパークラスを解放**することができます。

ITCollege copyright ©ITCollege All Rights Reserved



FactoryMethod

- **フレームワークと肉付け**

前述でframeworkパッケージとidcardパッケージに分かれている例を出しました。

これは「フレームワーク」と「肉付け」という単位に分けています。

ここで同じフレームワークを使ってまったく別の「製品」と「工場」を作るとしましょう。

例えば・・・テレビのクラスTelevisionとテレビ工場TelevisionFactoryとします。

この場合、frameworkパッケージをimportした別のtelevisionパッケージを作ることになるでしょう。ここでframeworkパッケージの中身を修正せずに、まったく別の「製品」と「工場」を作れるという点に注目してみましょう。

理由は、frameworkパッケージの中では、idcardパッケージをimportしていません。ProductクラスやFactoryクラスの中には、IDCardやIDCardFactoryという具体的なクラス名は書いてありません。

そのため、televisionパッケージをimportするようなframeworkパッケージの修正はまったく必要がありません。このことを「**frameworkパッケージはidcardパッケージに依存していない**」と表現します。

copyright ©ITCollege All Rights Reserved

デザインパターン

Singleton(シングルトン)

TemplateMethod(テンプレートメソッド)

FactoryMethod(ファクトリメソッド)

Iterator(イテレータ)

ITCollege copyright ©ITCollege All Rights Reserved

Iterator

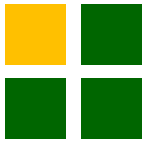
- **目的**

何かがたくさん集まっている時に、それを順番に指し示していき、全体をスキャンしていく処理を行うためのものです。

イメージとして

データが集合していて1つずつ上から下に順番に調べていきたい時に使用します。上記のようなイメージを持つことがとても重要です。

ITCollege copyright ©ITCollege All Rights Reserved



Iterator

- **特徴**

どうしてIteratorパターンなんて面倒なものを考える必要があるのでしょう。配列だったらfor文でくるくる回せば？と思ってしまいますよね。

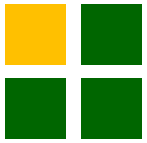
その大きな理由はIteratorを使うことで実装とは切り離して数え上げを行うことができることです。

```
while(it.hasNext()){
    Book book = (Book)it.next();
    System.out.println(book.getName());
}
```

上記のプログラムではIteratorのメソッドしか呼び出していないため、Iteratorインタフェースを実装したクラスには依存しないということがいえます。

これにより抽象クラスやインタフェースを使ってプログラミングをすることが可能になり、クラス間の結合を弱めることができます。

copyright ©ITCollege All Rights Reserved



Iterator

意味を間違えないようにしたいのはnextメソッドです。「現在の要素を返しつつ、次の位置に進める」ということを行います。

そして最後も間違えやすいので注意しましょう。

hasNext()メソッドは最後の要素を得た後はfalseを返します。最後の1個を返し損ねる危険がある点は注意しましょう。

また、応用として、以下のようなバリエーションをつけることができます。

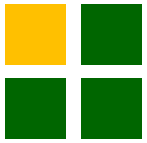
・最後尾から開始して逆方向に進む

・順方向にも逆方向にもいく

・番号を指定していきなりそこにジャンプする

このようなクラスを作ることも可能です。

ITCollege copyright ©ITCollege All Rights Reserved



Iterator

- **登場人物**

Iterator(反復子)の役

要素を順番にスキャンしていくインタフェースを定める。

ConcreteIterator(具体的な反復子)の役

Iterator役が定めたインタフェース(API)を実際に実装する役。

Aggregate(集合体)の役

Iterator役を作り出すインタフェースを定める役。

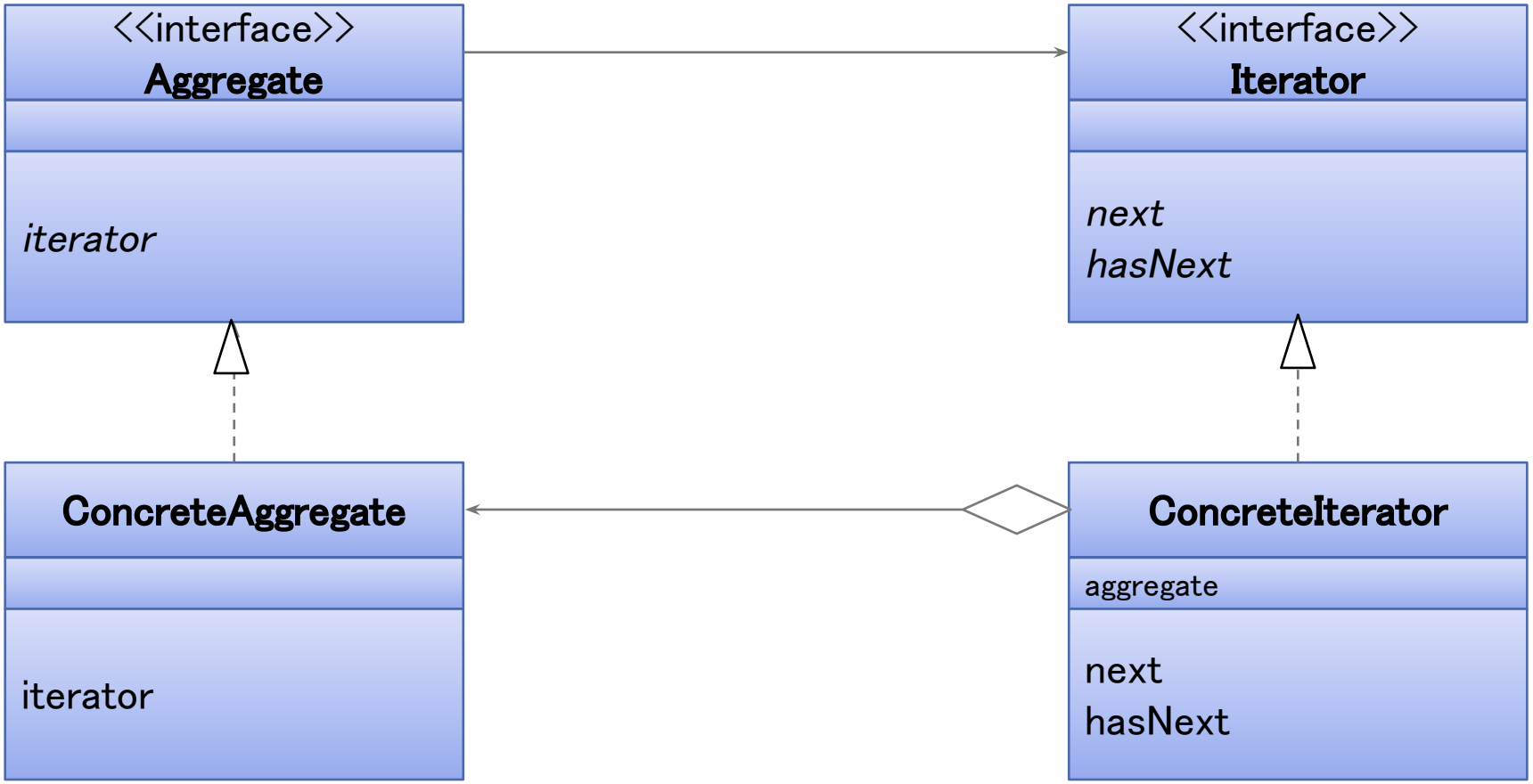
ConcreteAggregate(具体的な集合体)の役

Aggregate役が定めたインタフェースを実際に実装する役。

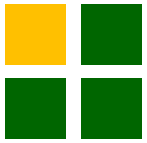
ITCollege copyright ©ITCollege All Rights Reserved

Iterator

クラス図は以下の通りです。



ITCollege copyright ©ITCollege All Rights Reserved



まとめ

- **設計原則とデザインパターンについていえること**

  自分以外の誰かが自分の作成したプログラムと関連した時に自分の修正と相手の修正が互いに影響を及ぼさないようにすることが重要です。

  それを実現するにはインタフェースや抽象クラスを用い、抽象的にプログラムを作成していき、クラス同士の関連を減らすように作っていくことが重要です。

  オブジェクト指向という概念を活用するなら積極的に使っていきましょう。

copyright ©ITCollege All Rights Reserved