応用プログラム言語II / 演習II
#06
C++で学ぶオブジェクト指向プログラミング（その3）

情報システム学科
坂本政祐
sakapon@sit.ac.jp

はじめに（#04復習）

復習

applied-progII #06

- OOPには三大要素というものがあります。
  - クラス（済）
  - ポリモーフィズム（未）
  - 継承（未）
- 今日はこの残り2つについて学んでいきます。

期待通りに振る舞ってくれると嬉しい: ポリモーフィズム

ポリモーフィズムの前に

applied-progII #06

- ポリモーフィズムの説明の前に，少し前置きの話をします。
- 「サブルーチン」（CやC++では「関数」）というのは元々何のために作るか?

サブルーチンを作る意義

applied-progII #06

- サブルーチンを何故作るか?
  - 理由1: 処理に名前をつけて独立させることで，何をやる部分なのかを明確化。
  - たとえ一度しか呼び出さないとしても，処理単位と名前によってわかりやすくなる。

applied-progII #06

```
#include <stdio.h>
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc_c.h>
#include <opencv2/highgui/highgui_c.h>

#define SOBEL_X_APERTURE_SIZE 3
#define SOBEL_Y_APERTURE_SIZE 3
#define LAPLACIAN_APERTURE_SIZE 3

#define SCALE 1
#define SHIFT 0

int main(void)
{
  int key;
  CvCapture *cap;
  IplImage *frameImage;
  IplImage *grayImage;
  IplImage *edgeImage16;
  IplImage *edgeImage8;

  char winNameCamera[] = "Camera";
  char winNameEdge[] = "Edge";

  cap = cvCreateCameraCapture(0);

  frameImage  = cvQueryFrame(cap);
  grayImage   = cvCreateImage(cvGetSize(frameImage), IPL_DEPTH_8U, 1);
  edgeImage16 = cvCreateImage(cvGetSize(frameImage), IPL_DEPTH_16S, 1);
  edgeImage8  = cvCreateImage(cvGetSize(frameImage), IPL_DEPTH_8U, 1);

  cvNamedWindow(winNameCamera, CV_WINDOW_AUTOSIZE);
  cvNamedWindow(winNameEdge, CV_WINDOW_AUTOSIZE);

  while(1){
    frameImage = cvQueryFrame(cap);

    cvCvtColor(frameImage, grayImage, CV_BGR2GRAY);
    cvLaplace(grayImage, edgeImage16, LAPLACIAN_APERTURE_SIZE);
    cvConvertScaleAbs(edgeImage16, edgeImage8, SCALE, SHIFT);

    cvShowImage(winNameCamera, frameImage);
    cvShowImage(winNameEdge, edgeImage8);

    key = cvWaitKey(1);
    if (key == 'q' || key == 0x1B){ break; }
  }

  cvReleaseCapture(&cap);
  cvReleaseImage(&grayImage);
  cvReleaseImage(&edgeImage16);
  cvReleaseImage(&edgeImage8);

  cvDestroyWindow(winNameCamera);
  cvDestroyWindow(winNameEdge);

  return 0;
}
```

- サブルーチン（関数）を作らずにすべて main() 関数に書いた場合
  - 読みにくい
  - どこで何をやっているかがわかりづらい（この例ではあえてコメントを入れていません）

- サブルーチン（関数）を作って使った例
  - 処理に名前がついてどこで何をしているかわかりやすい

サブルーチン定義

```
#define SCALE 1
#define SHIFT 0

void initializeCamera(CvCapture *cap)
{
  cap = cvCreateCameraCapture(0);
}

void initializeImagesAndWindows(CvCapture *cap,
  IplImage *gray, IplImage *edge16, IplImage *edge8,
  const char *winCamera, const char *winEdge)
{
  IplImage *frameImage;

  frameImage  = cvQueryFrame(cap);
  gray   = cvCreateImage(cvGetSize(frameImage), IPL_DEPTH_8U, 1);
  edge16 = cvCreateImage(cvGetSize(frameImage), IPL_DEPTH_16S, 1);
  edge8  = cvCreateImage(cvGetSize(frameImage), IPL_DEPTH_8U, 1);

  cvNamedWindow(winCamera, CV_WINDOW_AUTOSIZE);
  cvNamedWindow(winEdge, CV_WINDOW_AUTOSIZE);
}

void detectEdge(CvCapture *cap,
  IplImage *gray, IplImage *edge16, IplImage *edge8,
  const char *winCamera, const char *winEdge)
{
  frameImage = cvQueryFrame(cap);

  cvCvtColor(frameImage, grayImage, CV_BGR2GRAY);
  cvLaplace(grayImage, edgeImage16, LAPLACIAN_APERTURE_SIZE);
  cvConvertScaleAbs(edgeImage16, edgeImage8, SCALE, SHIFT);

  cvShowImage(winNameCamera, frameImage);
  cvShowImage(winNameEdge, edgeImage8);
}
```

```
void initializeImagesAndWindows(CvCapture *cap,
  IplImage *gray, IplImage *edge16, IplImage *edge8,
  const char *winCamera, const char *winEdge)
{
  Capture(&cap);
  mage(&gray);
  mage(&edge16);
  mage(&edge8);

  cvDestroyWindow(winCamera);
  cvDestroyWindow(winEdge);
}

int main(void)
{
  int key;
  CvCapture *capture;
  IplImage *frameImage;
  IplImage *grayImage;
  IplImage *edgeImage16;
  IplImage *edgeImage8;

  char winNameCamera[] = "Camera";
  char winNameEdge[] = "Edge";

  initializeCamera(capture);

  initializeImagesAndWindows(capture,
                grayImage, edgeImage16, edgeImage8,
                winNameCamera, winNameEdge);

  while(1){
    detectEdge(capture,
          grayImage, edgeImage16, edgeImage8,
          winNameCamera, winNameEdge);

    key = cvWaitKey(1);
    if (key == 'q' || key == 0x1B){ break; }
  }

  finalizeImagesAndWindows(capture,
                grayImage, edgeImage16, edgeImage8,
                winNameCamera, winNameEdge);

  return 0;
}
```

サブルーチン呼び出し

サブルーチンを作る意義

applied-progII #06

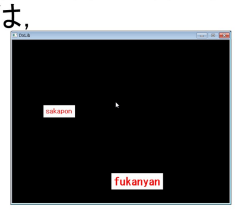
- サブルーチンを何故作るか?
  - 理由2: 何度も同じ仕事をする場合に，それをその都度その都度ソースに書くのではなく，1つにまとめて書いてそれを繰り返し呼び出せる。
  - つまり，呼び出される側のロジックを1つにまとめるものである。

サブルーチンを作る意義

applied-progII #06

■例えば，DXライブラリ[*1]というライブラリで，画面上に「白い四角を作って，その上に赤でセンタリングして任意の文字列を描く」という処理をやりたいとする。これを実現するには，

- ■フォントサイズを指定する
- ■描画位置を指定する
- ■描く文字列を指定する
- ■四角を描く
- ■文字列を描く



という手順が必要である。ので，これを何個もやろうとすると…

*1 ゲーム製作なんかでよく使われるライブラリです。


```
ClsDrawScreen();

  ...

int font_size1 = 20;
int x1 = 100, y1 = 200;
char *str1 = "sakapon";
SetFontSize(font_size1);
int w1 = GetDrawStringWidth(str1, strlen(str1));
DrawBox(x1 - 8, y1 - 8, x1 + w1 + 8, y1 + font_size1 + 8, WHITE, TRUE);
DrawString(x1, y1, str1, RED);

  ...

int font_size2 = 32;
int x2 = 300, y2 = 400;
char *str2 = "fukanyan";
SetFontSize(font_size2);
int w2 = GetDrawStringWidth(str2, strlen(str2));
DrawBox(x2 - 8, y2 - 8, x2 + w2 + 8, y2 + font_size2 + 8, WHITE, TRUE);
DrawString(x2, y2, str2, RED);

  ...

ScreenFlip();
```

■こんな感じで，ほぼ同じことしかしていないのに，それを複数箇所同じように書くことになる。


```
void DrawStringBox(int font_size, int x, int y, const char *str)
{
  SetFontSize(font_size);
  int w1 = GetDrawStringWidth(str, strlen(str));
  DrawBox(x - 8, y - 8, x + w1 + 8, y + font_size + 8, GetColor(255,255,255), TRU
  DrawString(x, y, str, GetColor(255,0,0));
}

  ...

  ClsDrawScreen();

  ...

  DrawStringBox(20, 100, 200, "sakapon");

  ...

  DrawStringBox(32, 300, 400, "fukanyan");

  ...

  ScreenFlip();
```

■サブルーチン（関数）を作れば，「呼び出される側」のロジックを1つにまとめることができる。

一方，ポリモーフィズムは

applied-progII #06

■（注）ここまでの話はまだOOPと何の関係もありません。

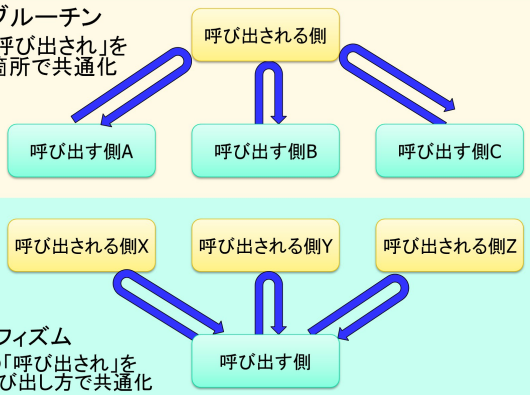
■一方，ポリモーフィズムは，ちょうどこの逆で，呼び出す側のロジックを1つにまとめる。

■つまり，サブルーチンの「理由2」が「共通サブルーチン」を作ることだったのに対して，ポリモーフィズムは「共通メインルーチン」を作ることに相当する。

イメージ図

applied-progII #06

■共通サブルーチン
- ■同じ「呼び出され」を複数箇所で共通化



■ポリモーフィズム
- ■複数の「呼び出され」を同じ呼び出し方で共通化

#04の再掲

クラスを説明する題材

applied-progII #06

■これらの効能を，以下では「シューティングゲームのキャラ」を題材にして見ていきます。

■想定するシューティングゲーム:
- ■自機1機画面上にいます。 未実装
- ■敵が複数画面上にいます。

#04の再掲

まずTekiクラスを思い出す

applied-progII #06

■Tekiクラスはこんな実装だった。

リスト6-1

```
class Teki
{
private:
  int handle;
  void effect(void) { /* ロジックは省略 */ }

public:
  int x, y;

  void yomikomu(std::string file_mei) { /* ロジックは省略 */ }
  void ugokasu(void) { /* ロジックは省略 */ }
  void hyouji(void) { /* ロジックは省略 */ }
};
```


Jikiクラスを作る

applied-progII #06

■自機のクラス「class Jiki」に必要な機能は何か?
- ■自機位置の座標の管理
- ■自機のキャラクター画像のファイル名を指定して，それを内部的にはハンドルという数値として扱うための準備
- ■自機の座標を更新
- ■自機の表示

■あれ，ということは敵と一緒じゃん，ということで…。

Jikiクラスを作る

- Jikiクラスにはこんな実装にしよう。

リスト6-2

```
class Jiki
{
private:
  int handle;
  void effect(void) { /* ロジックは省略 */ }

public:
  int x, y;

  void yomikomu(std::string file_mei) { /* ロジックは省略 */ }
  void ugokasu(void) { /* ロジックは省略 */ }
  void hyouji(void) { /* ロジックは省略 */ }
};
```


TekiクラスとJikiクラス

- 省略しているロジック部分は，敵と自機とでは同じ場合もあるし，違う場合もあるだろう。
  - 例えば，ugokasu() 関数の中身は，
    - 敵: ランダムに動く
    - 自機: キーボードやジョイパッドからの入力によって動く
- しかしいずれにしろ，クラスの外に見えている関数名はどっちもまったく一緒。
- だとすると…。

TekiクラスとJikiクラスを使う

リスト6-4

```
class Teki
{
  /* 省略 */
}

class Jiki
{
  /* 省略 */
}

int main(void)
{
  Teki zako;
  Teki boss;
  Jiki jiki;

  ...

  while(true) /* ゲームループ */
  {
    zako.ugokasu();
    boss.ugokasu();
    jiki.ugokasu();

    zako.hyouji();
    boss.hyouji();
    jiki.hyouji();
  }
}
```

Point

Point

- TekiクラスとJikiクラスが出来上がったので，後はそれをmain()関数で実際に使おう。
- すると，TekiとJikiで関数名を同じにしておいたおかげで，敵も自機もすべてugokasu() で動くし，すべてhyouji() で表示できる。
- これがポリモーフィズム。

ポリモーフィズム

- つまりポリモーフィズムというのは，同じ関数名にしておけば，同じような振る舞いを期待して呼び出せるよ，という概念。
- え，そんだけ!?
  - 特に新しい文法が出てきたわけじゃないし，同じ関数名にしただけじゃん，と思うかも知れないし，
  - しかも，「共通サブルーチン」がコード量を減らすことにも役立ったのに，「ポリモーフィズム」はコード量減ってないじゃん（実際，zako.ugokasu()とjiki.ugokasu()は別々に呼び出しているので），と思うかも知れない。

ポリモーフィズム

- 実際その通りですが:
  - 関数名については:
    - これが非OOPL（例えばC）だったら，zako_ugokasu()関数とjiki_ugokasu()関数を別々に作らなければならない。しかもグローバルに。
    - それを，各クラスにメンバ関数として持たせて，インスタンス zako でもインスタンス jiki でも，どちらも ugokasu() 関数ならきっと期待どおりに振る舞ってくれるだろう，ということになるわけです。
    - 実はこの振る舞いは，皆さんはすでに「better C としてのC++」範囲で体験しています。そう，std::string にも，std::vector にも，std::map にも，全部 size() があり，期待通りに振る舞ってくれたのです!

ポリモーフィズム

- 実際その通りですが:
  - コード量が減っていないことについては:
    - ポリモーフィズムは，「そういう『概念』があれば，変に命名に悩んだり，名前の衝突に苦労したり，teki_ugokasu() とugokasu_jiki() のように命名に一貫性が無いことに泣き笑いしなくて良いね」「そういう『概念』があれば，コードの書きやすさと読みやすさが100倍上がるね」というあくまで「概念」なので，直接的にはコード量を減らす仕組みではないのです。
    - が，OOPLそれぞれの言語において，ポリモーフィズムがより便利に使えるような仕組みが提供されています。
    - これについては，継承と関わってくるので，この後さらに継承を説明したあと次回に再び説明します。

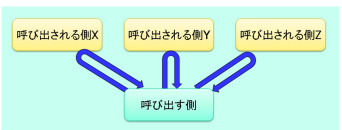
イメージ図と対応させると

リスト6-4

```
class Teki
{
  /* 省略 */
}

class Jiki
{
  /* 省略 */
}

int main(void)
{
  Teki zako;
  Teki boss;
  Jiki jiki;

  ...

  while(true) /* ゲームループ */
  {
    zako.ugokasu();
    boss.ugokasu();
    jiki.ugokasu();

    zako.hyouji();
    boss.hyouji();
    jiki.hyouji();
  }
}
```

Point

Point



- さきほどのこのイメージ図と対応させるとこんな感じ。

ポリモーフィズムについての小まとめ

- ポリモーフィズムは，サブルーチンを呼び出す側のロジックを一本化できる仕組み。
- すなわち，共通メインルーチンを作る仕組み。
- 同じ関数名にしておくと同じ振る舞いが期待できるね，という概念。もしくは，そうなってくれるようにクラス側とその中の関数名を適切に作ること。

性質を引き継ぐことで楽をしよう: 継承

継承とは

- クラスの最初の説明で，クラスの利点として，
  - まとめて
  - 隠して
  - たくさん作れる

  ということを言いました。

- 継承は，そうして作ったクラスから，さらに共通部分をくくり出し，それを引き継ぐ仕組みです。

継承とは

- 例えばTekiクラスとJikiクラスでは:
  - （前述のように）ugokasu() 関数は，敵と自機ではまったくことなるはず。
  - 一方で，
    - yomikomu() 関数（ファイルから画像を読みだしてそれに対応した数値（ハンドル）を1つ決める）とか，
    - hyouji() 関数（ugokasu()関数で決めた座標にキャラを表示）

    などは一字一句変わらないはず。

継承とは

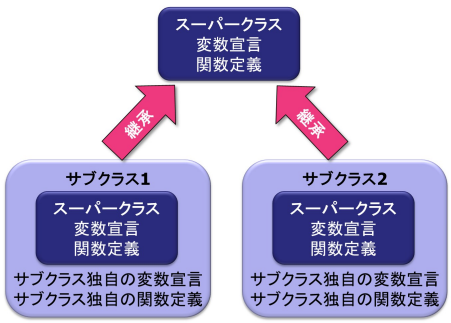
- だったら，くくりだそう。
- 日本語的には，こう考える。
  - 敵も自機も，もうちょっと広い視点（一段上に上がった視点）で眺めてみると，どっちも「キャラ」の一種だな。
  - キャラの一種ではあるけど，細部ではことなる部分もあるな（動かし方など）。
- C++的には，それをこう書く。
  - class Teki と class Jiki の共通点を class Kyara としてくくりだす。
  - くくり出せないものだけ，class Teki や class Jiki に残す。
  - class Teki や，class Jiki は class Kyara を継承することで，その機能を取り込む。
  - 継承すると，スーパークラス側の関数や変数は，サブクラス側では，それがサブクラスで定義したかのようにすべて使える。

スーパークラスとサブクラス

- そうすることで，似たような処理の重複記述を避けることができる。
- 継承される側: スーパークラス
- 継承する側: サブクラス
- is-a の関係になっているものを，スーパークラスとサブクラスにするべき。
  - この is-a（イズ　ア）は「～の一種である」と訳すと一番しっくりくる。
  - 例1: 敵 is a キャラクタ（敵はキャラクタの一種である）: 敵=サブクラス，キャラクタ=スーパークラス
  - 例2: プリウス is a 車（プリウスは車の一種である）: プリウス=サブクラス，車=スーパークラス

継承関係の図

- 継承関係



C++では継承をどう書くか

- C++での継承の書き方

リスト6-4

```
#include <string>

// スーパークラス
class Kyara
{
private:
  int handle;

public:
  int x, y;
  void yomikomu(std::string file_mei)
  {
    /* どんなキャラにも共通なファイル読み込み処理 */
  }
  void hyouji(void)
  {
    /* どんなキャラにも共通な表示処理 */
  }
};
```

（続く）

C++では継承をどう書くか

リスト6-4（続き）

```
// サブクラス1                    Point
class Teki : public Kyara
{
public:
  void ugokasu(void)
  {
    /* 敵特有な動き処理（ランダムとか） */
  }
};

// サブクラス2                    Point
class Jiki : public Kyara
{
public:
  void ugokasu(void)
  {
    /* 自機特有な動き処理（キーボードからの入力とか） */
  }
};
```

（続く）

C++では継承をどう書くか

applied-progII #06

リスト6-4（続き）

```
int main(void)
{
  Jiki jiki;
  Teki boss;

  jiki.yomikomu("jiki.bmp");
  boss.yomikomu("boss.bmp");

  while(true) /* ゲームループ */
  {
    boss.ugokasu();
    jiki.ugokasu();

    boss.hyouji();
    jiki.hyouji();
  }
}
```

スーパークラスの関数も呼び出せる

33

C++での継承の書き方

applied-progII #06

- C++での継承の書き方
  - スーパークラスは普通に書く。
  - サブクラスは，class クラス名 の後に，「: public スーパークラス名」をつけると，そのスーパークラスを継承したことになる。
    - ちなみに，「:」の部分は他のOOPLだと，Java だと「extends」，Ruby だと「<」，Python だと「()」など，言語によってまちまちです。

34

継承についての補足

applied-progII #06

- あれ，クラスとインスタンスの関係も is-a なんじゃないか? と思うかも知れない。
- クラスはあくまで設計図，抽象的なものであり，インスタンスは具体化したものであることが違う。
  - なのでより正確には，
    - プリウス is a 車 というのは，
    - プリウスの設計図 is a 車の設計図 ということであり，
    - 「坂本が買ったプリウス」（＝インスタンス）というのは，その設計図を元に量産されたものの1つ，ということ。
  - そういう意味では，クラス名は，
    - class Prius や class Kuruma とかではなく，
    - class PriusSekkeizu とか class KurumaSekkeizu とかにした方が最初のうちはわかりやすいかも知れない。
    - ただ，クラスというのは元々ほぼすべて設計図なのだから，それをいちいち名前に入れるのはみんなあまりしない，ということ。

35

継承についての補足

applied-progII #06

- 継承関係を作る意味がある状況:
  - サブクラスが2個以上考えられる状況
    - これは今作っているTekiクラスとJikiクラスがそうですね。
  - 設定や構成の大部分は決まってるんだけど，その一部はカスタマイズしたいような場面
    - GUIのアプリケーションではこの手の手法をよく使います。基本的な画面設計はライブラリ側で用意しておき，プログラマはそれを継承して，さらにボタンや背景画像をカスタマイズする，という感じです。

36

継承についての補足

applied-progII #06

- 書籍によっては，スーパークラスのことを親クラス，サブクラスのことを子クラスと書いてあったりしますが，「継承は is-a 関係であって親子関係ではない」ということに注意してください。
  - サブクラスはスーパークラスのすべての性質を引き継げるが，現実の親子は親と子が顔がまったく同じということはないわけです。
  - ていうか，子 is a 親ではないですよね。

37

継承についての補足

applied-progII #06

- ポリモーフィズムのところで書いたように，継承の仕組みを利用するとポリモーフィズムがさらに活きてきます。
- なんてスマートな書き方なんだ！とウケること請け合いですが，それはまた次回。今回の課題は継承に集中します。

38

継承についての小まとめ

applied-progII #06

- 継承は，複数のクラス定義の共通部分をくくり出すことで，コードの重複をなくす仕組み。
- 継承される側をスーパークラス，継承する側をサブクラスと呼ぶ。
- サブクラス is-a スーパークラスの関係。（サブクラス○○はスーパークラス△△の一種である。）

39

今日の課題

applied-progII #06

- 課題1: 先週までに自分が作ってきたクラスをサブクラスとして，何かスーパークラスになれるものは無いか考え，それを実装してください。
  - そして，今まで自分が作ってきたクラスはそれを継承するように書き換えて下さい。
  - また，今まで自分が作ってきたクラス以外に，もう1つサブクラスを作って，それも上記スーパークラスを継承するようにしてください。
  - 今まで自分が作ってきたクラスが，どうしても継承にふさわしくない場合もあるかも知れません。その場合，運が悪かったとあきらめて，また新たに別のクラスを作っても構いません。

40

今日の課題

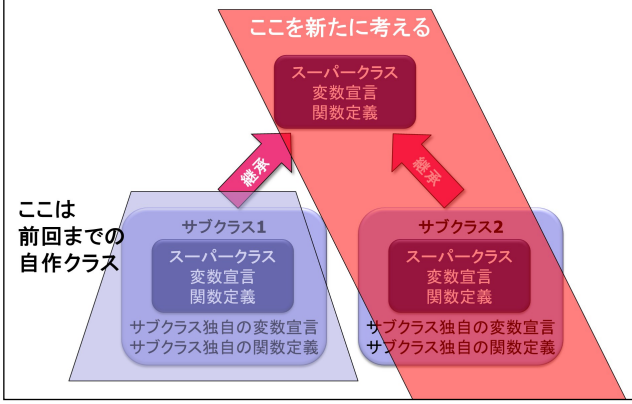
applied-progII #06

- 課題1の続き: 今までの自作クラスで「コンストラクタ」を作ってあった人は，今回の課題では，コンストラクタはサブクラス側でのみ作ってください。

41

今週の課題

applied-progII #06



42

今日の課題

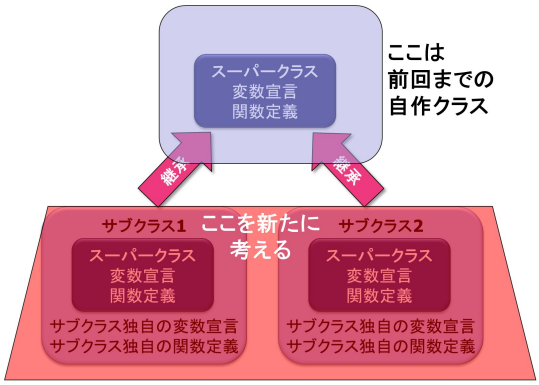
applied-progII #06

- 課題1続き: どうしても，自作クラスに対するスーパークラスが思いつかない場合は，自作クラスの方をスーパークラスとしたサブクラスを考える，というのでも良いことにします。
  - その場合も，サブクラスは2個考えましょう。

43

今週の課題（こうでも良いことにする）

applied-progII #06



44

今週の落穂拾い

applied-progII #06

- 継承は is-a だ，ということだが，現実世界のモノやコトは，ひとつのものが色々なものの is-a である場合もある。これを OOPL で表現するときは多重継承という仕組みを使う。
  - 例: 坂本 is a 大学教員，かつ，坂本 is a 男
  - ただし多重継承はOOPLによってはできたりできなかったりする。

45

今週の落穂拾い

applied-progII #06

- 今日の課題は，今まで継承なんて気にせずに作ってきたクラスに突然追加で継承の仕組みを導入しなさいという課題です。ただ，問題文にも書きましたが，世の中のありとあらゆるクラスが何かの継承であるべきだ，ということはありません。
- ですので，OOPの三大要素である，クラス，ポリモーフィズム，継承ですが，
  - クラスはどんなOOPコードにも必要不可欠であるのに対して，
  - ポリモーフィズムと継承は，それを使うと劇的に効率良く書けることがあるが，不可欠なものではない，と思って下さい。

46