プログラミング言語設計論

2014年度

第6回: 階層分類の先

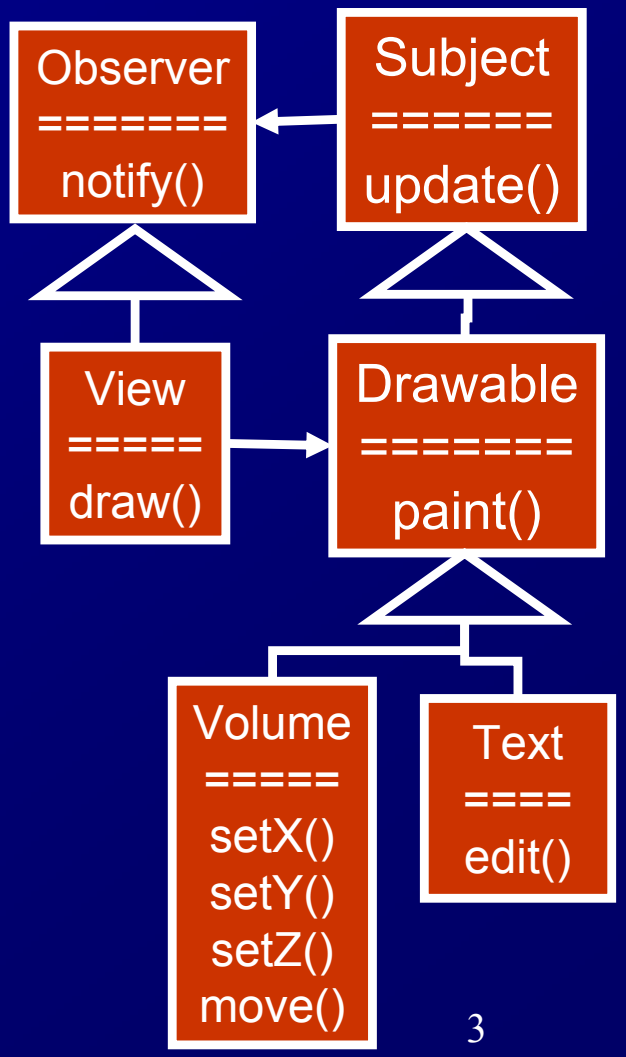
担当: 増原英彦

問題(1/2) 銀行の顧客を表現するクラス階層を設計せよ (10分)

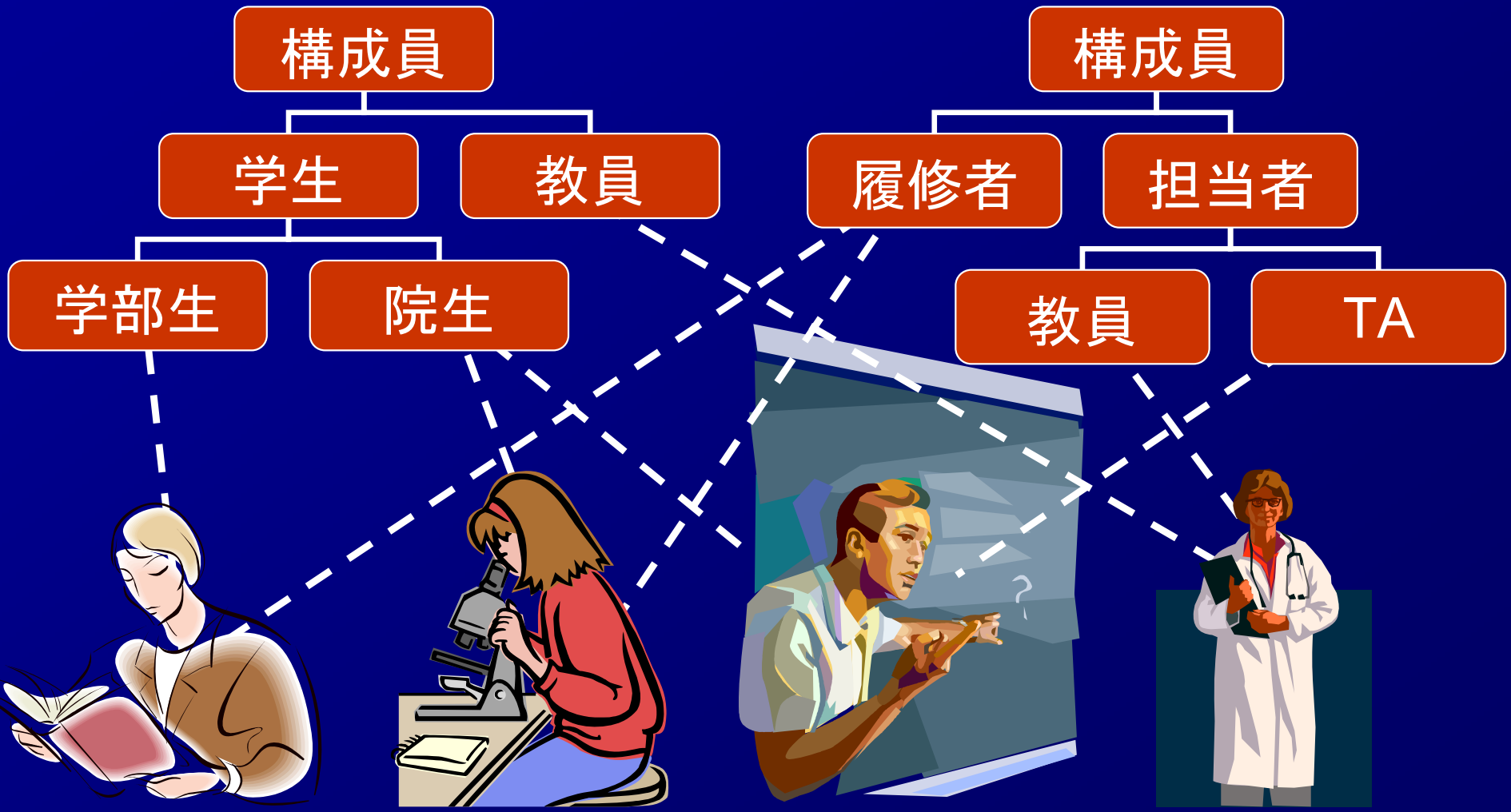
- 分類軸1: 法人、優遇個人、個人。法人は法人名と担当者を持つ。優遇個人と個人は氏名と生年月日を持つ。
  - 手数料の計算方法がそれぞれ違う
  - 法人は通知のときの宛名の書き方が違う
- 分類軸2: 国内と海外。国内の場合は住所として郵便番号、都道府県、市町村区、番地を持つ、海外の場合は国名と自由書式の住所を持つ
  - 通知を送る方法が違う
  - 税務処理が違う
  - 海外は手数料に追加料金が発生する

問題(2/2) Observerパターン批判(10分)

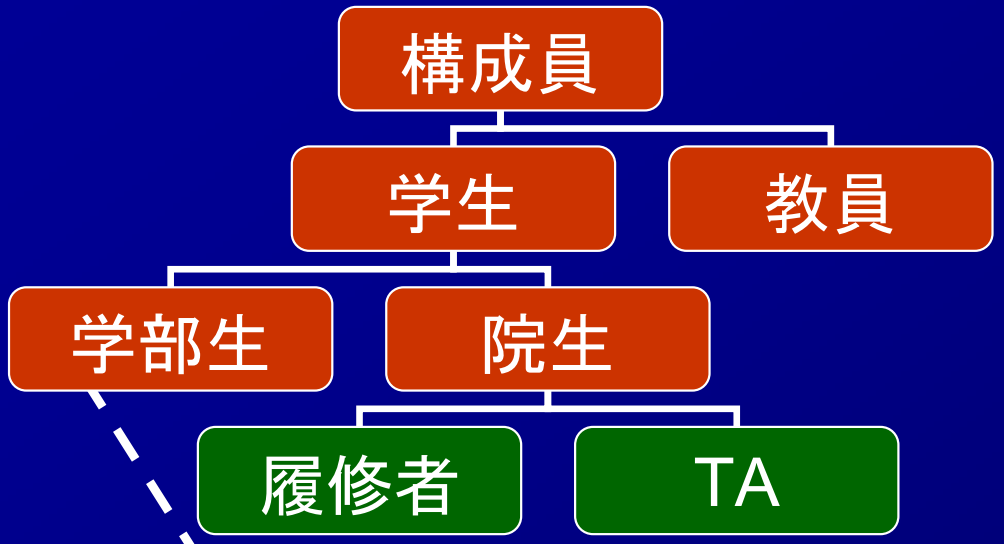
Observerパターンに基づく
右の設計を批判せよ
■実際にはVolume, Text以外にも沢山のDrawableがある
■VolumeがsetX, setY, setZ,moveしたら、またTextがeditされたらupdateを呼び、Viewにdrawさせる
■将来の拡張: 高速editモードを追加
——その間はVolumeが変化してもViewのdrawを実行しない



視点によって異なる階層的分類 [HO93]

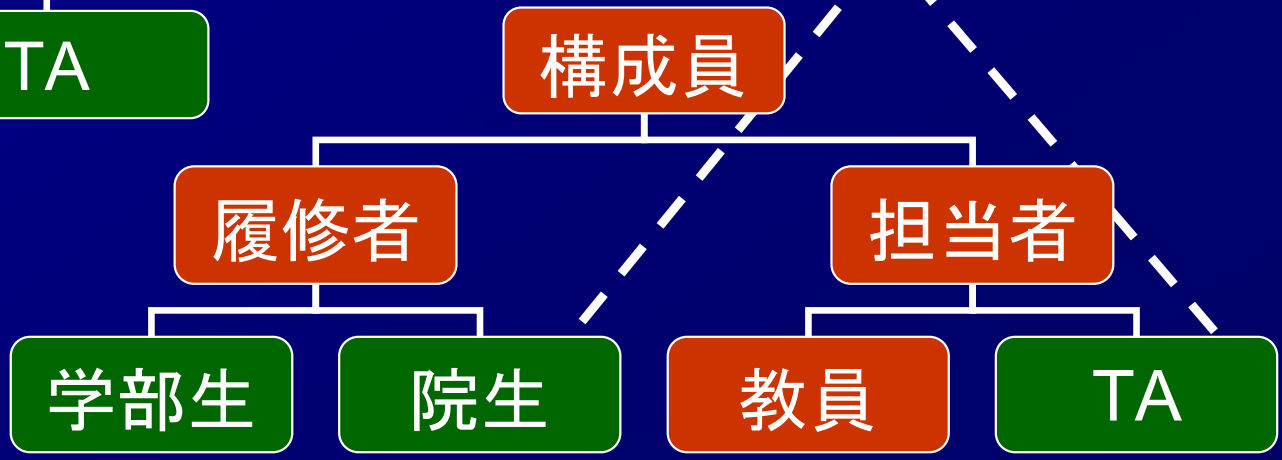


視点によって異なる階層的分類

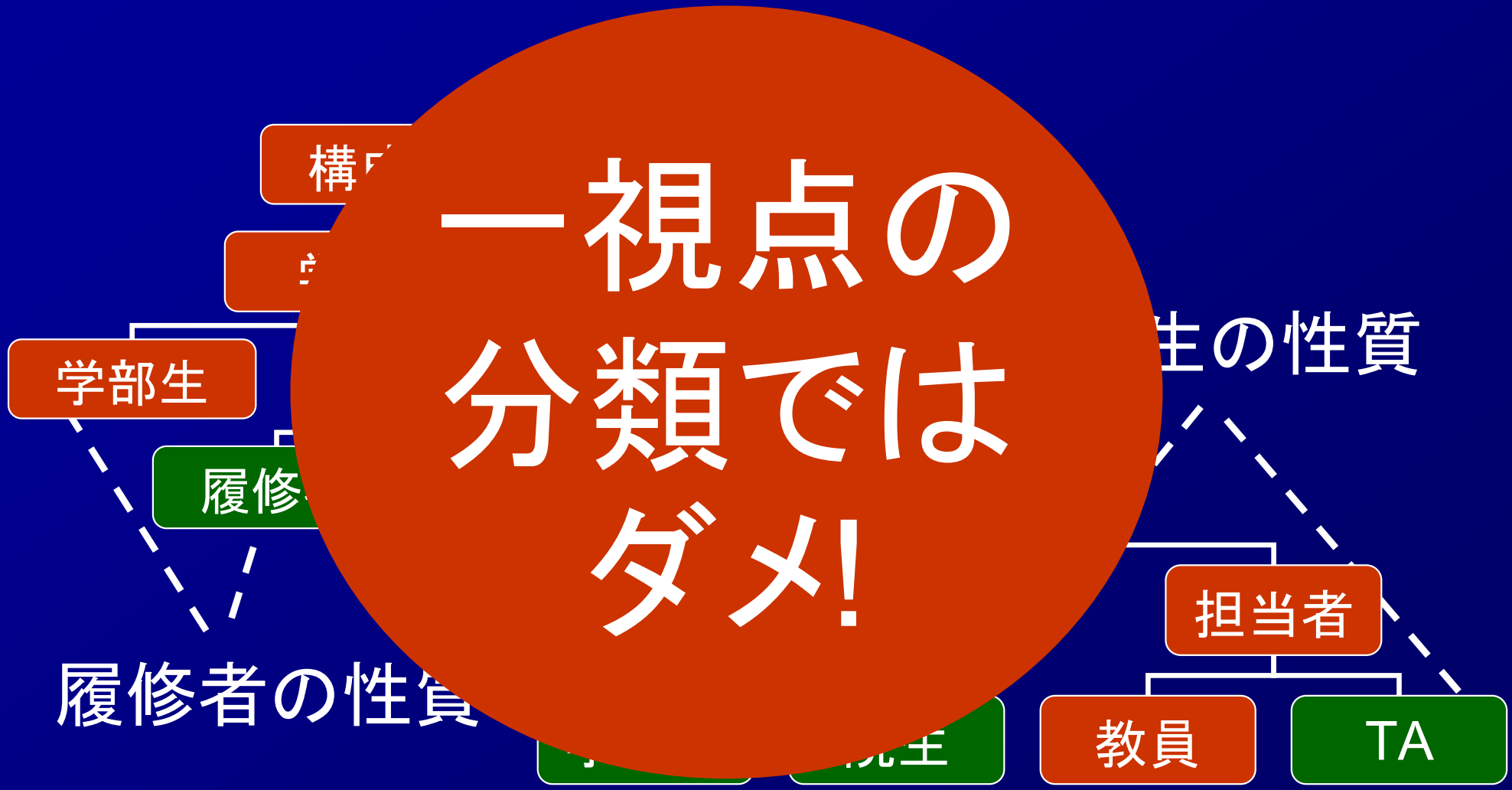


履修者の性質

院生の性質



視点によって異なる階層的分類



一視点の分類ではダメ!

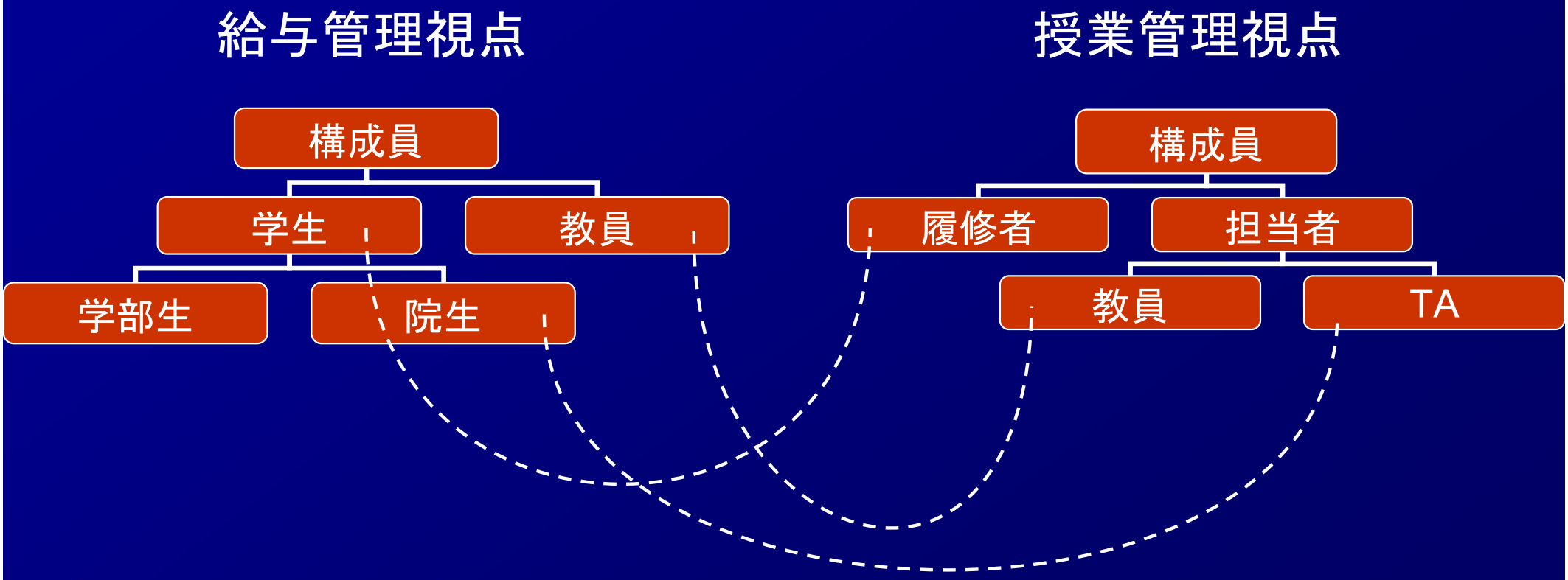
Subject指向プログラミング [HO93]

- 視点によって異なる階層的分類を扱う
- アプローチ: クラス階層の合成
- 言語: Hyper/J

Hyper/J言語

- Java言語のSubject指向拡張
- 視点ごとにクラス階層を作成する
  - 異なる視点に同じクラスが登場してよい
  - 「同じクラス」は多対多の関係でよい
- クラス階層どうしを合成する
  - クラス間の対応を指定する
  - メソッド間の対応を指定する
    - ◆overrideする、直列に実行する等
- (既存のプログラムからクラス階層を切り出す)

Hyper/Jによる大学構成員管理



Subject-oriented programingの問題

- クラス・メソッドの対応関係を記述する手間(良い記述言語がない)
- 明確でない意味論
  - 合成するクラス階層どうしが矛盾する可能性
  - 合成後を想像しないと動きが分からない場合

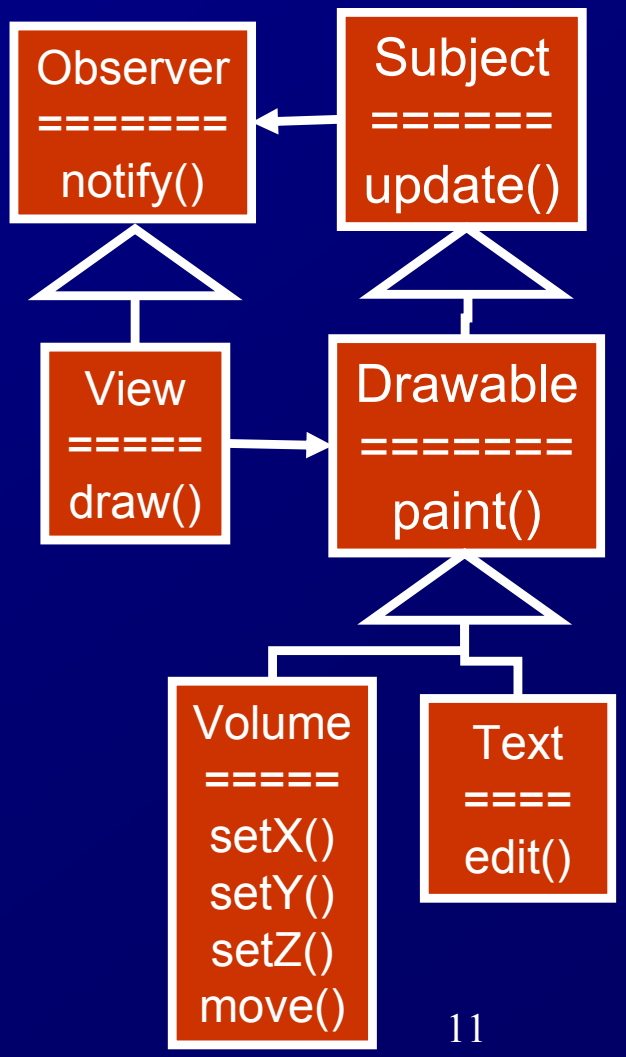
問題(2/2) Observerパターン批判(10分)

Observerパターンに基づく
右の設計を批判せよ

■実際にはVolume, Text以外にも沢山のDrawableがある

■VolumeがsetX, setY, setZ,moveしたら、またTextがeditされたらupdateを呼び、Viewにdrawさせる

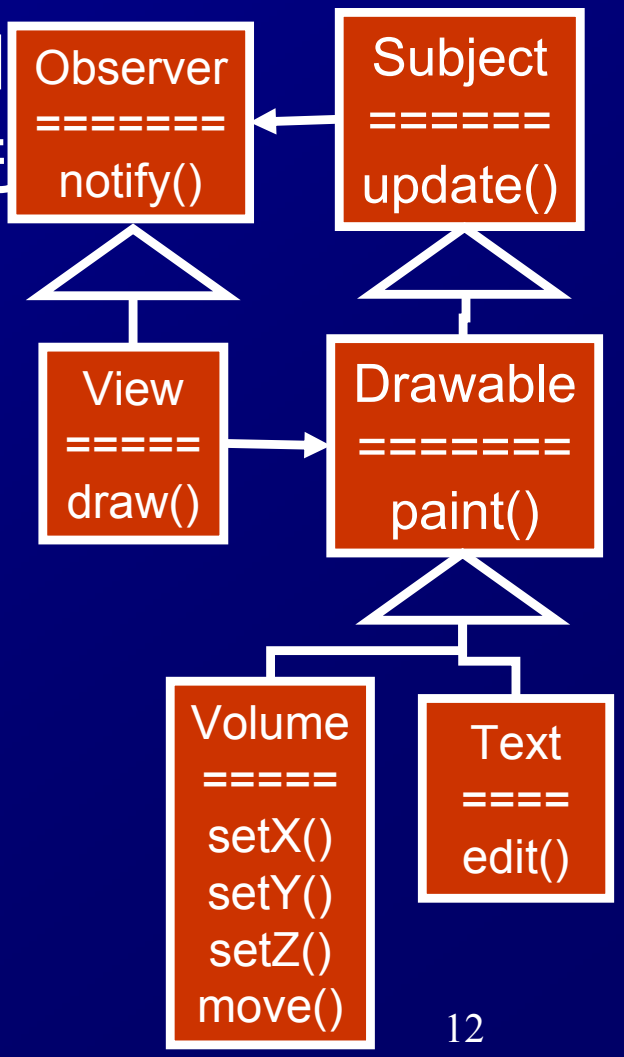
■将来の拡張: 高速editモードを追加
——その間はVolumeが変化してもViewのdrawを実行しない



階層的でない拡張

- 将来の拡張: 高速editモードを追加
  ——その間はVolumeが変化しても
  Viewのdrawを実行しない

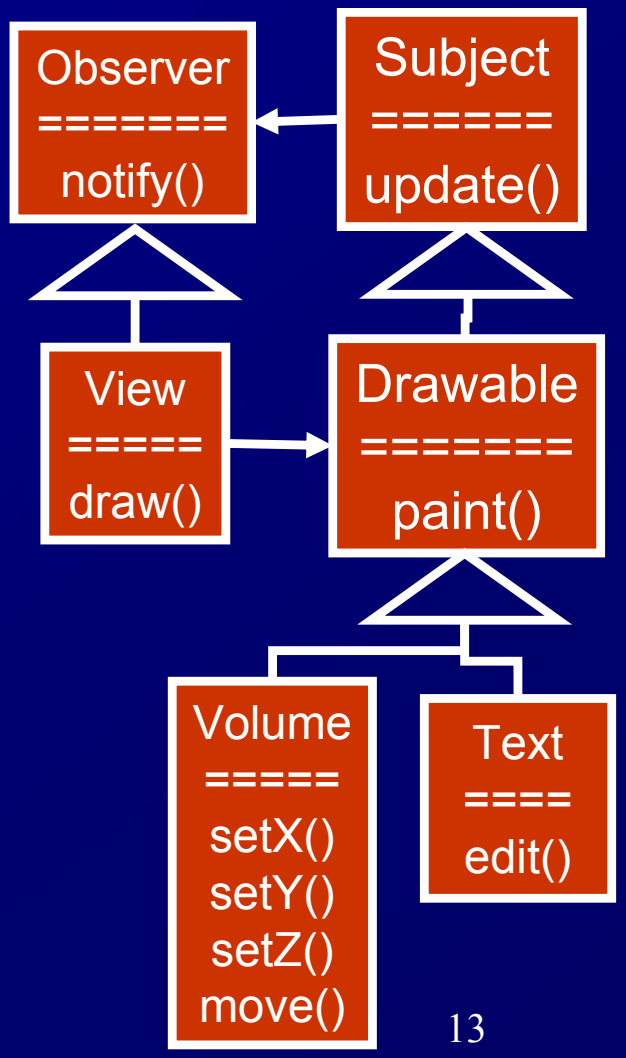
→「何をしたらupdateするか」
  という方針の拡張
  - Observerパターンはupdateを
    埋め込んでいる
    →方針がhard code



階層的でない拡張

- VolumeがsetX, setY, setZ,moveしたら、またTextがeditされたらupdateを呼び、Viewにdrawさせる
  - moveの中でsetX, setY, setZを呼び出している
  - setX, setY, setZの中でupdateを呼び出している
  - move1回で3回updateしたくない

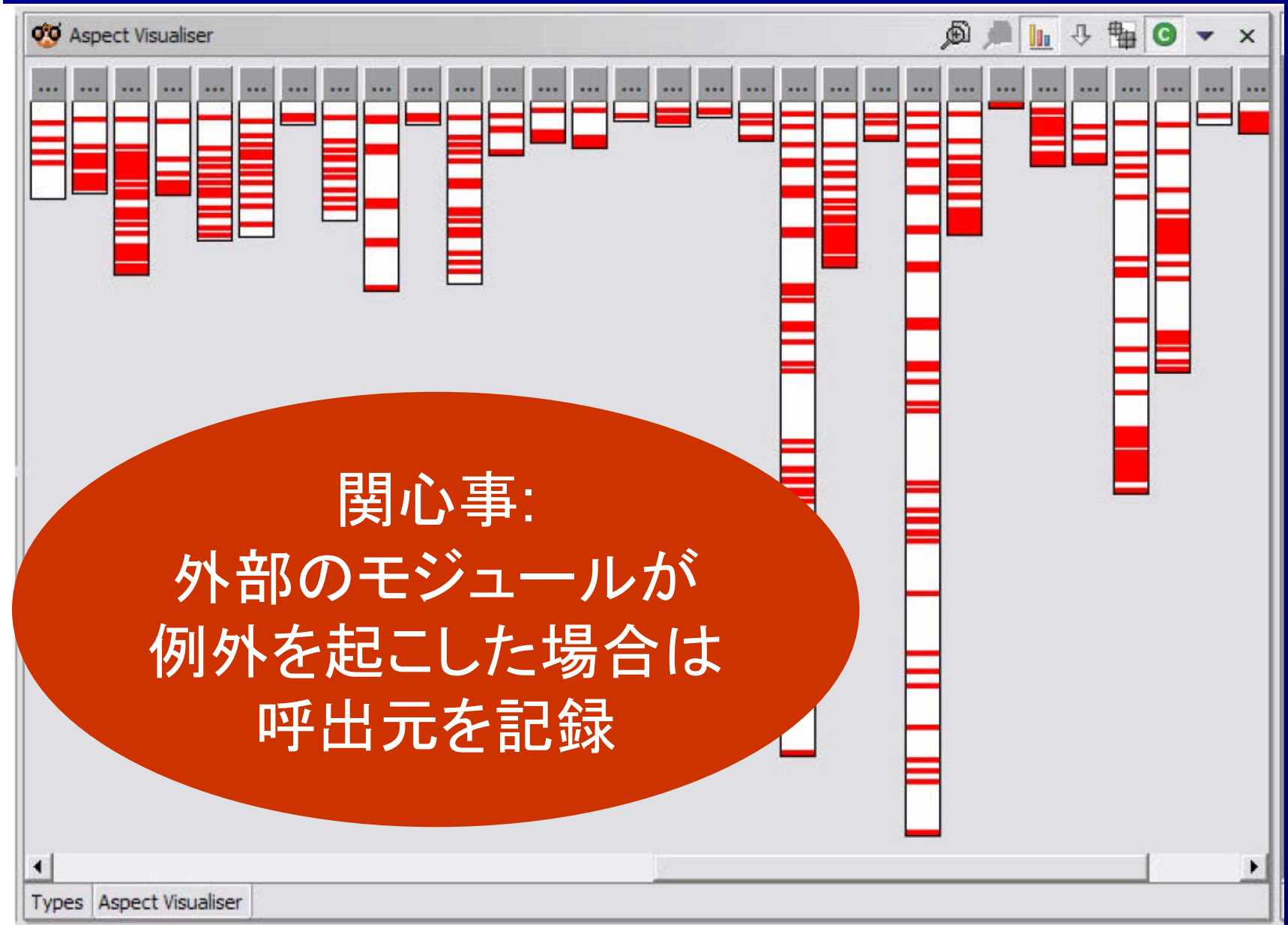
→「何をしたらupdateするか」がhard codeされている



横断的関心事
記述の散在ともつれ

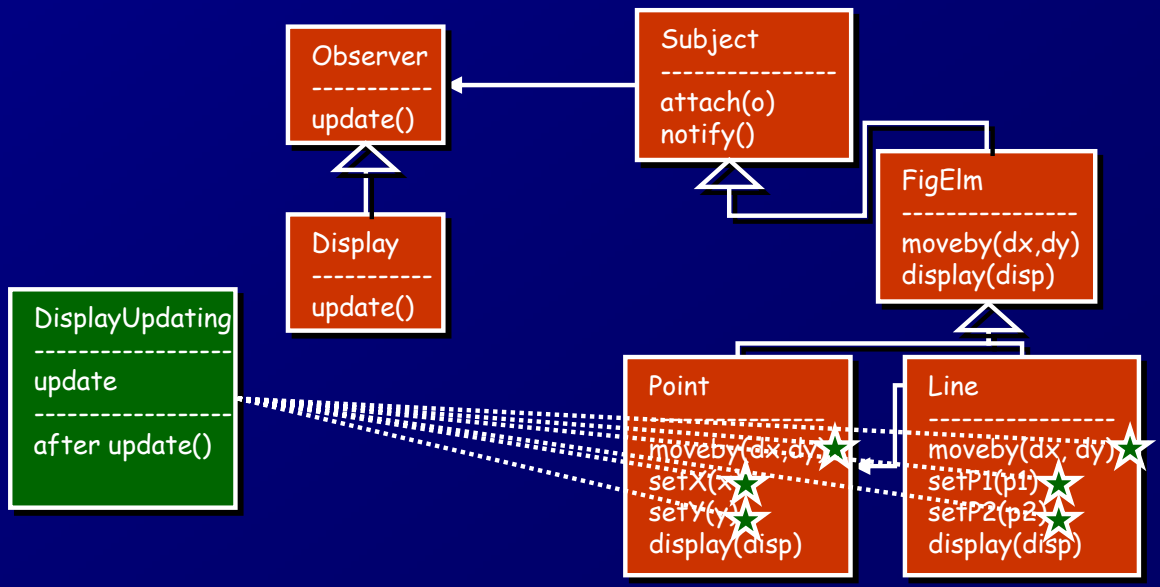
- 横断的関心事(crosscutting concerns): 1つの関心事が複数のモジュールにまたがってしまうこと
  - 「いつ画面更新するか」という方針: 複数のクラスの色々なメソッドへのhard codeとして実現
- 関心事: 人が「1まとまり」と考えるソフトウェア中の単位 (機能単位、性能への要求など)
  - 例:「要素が変更されたら画面更新をする」
- 記述の散在(scattering): 1つの関心事の実現がが色々なモジュールの記述に出現すること
- 記述のもつれ(tangling): 1つのモジュールの記述に複数の関心事が出現すること

記述の散在 [Colyer+03]



アスペクト指向プログラミング [Kiczales+97]

- クラス階層に対して横断的な追加・変更を行うモジュール = アスペクト
  - 特徴: 横断的な指示をするpointcut言語



AspectJ [Kiczales+01]

- AOPの代表的な言語
  - Javaの拡張
- 産業界での応用も多い
- 類似機能を持つ言語・システムも多い
  - JBoss AOP, AspectWerks, Spring AOP, Seasar 2 AOP, etc.
  - C, C++, C#, Python, Ruby, Smalltalk, ...

アスペクトの例:
「図形要素が動いたら画面更新」

- ポイントカット:「move = Point:: setXの呼出 または・・・」
- アドバイス:「move後にredraw()を実行」
- 型間宣言: 既存クラスにメソッド・フィールドを追加
- アスペクト: 1つのモジュール

```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.moveby(int,int)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int)) ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));

  after() : move() { Display.redraw(); }

  void FigElm.draw(Display d);
  void Point.draw(Display d) { … }
  … }
```

AspectJの基本: アドバイス (advice declarations)

追加・代替の動作を記述

- どんな動作の(ポイントカット)
- 前/後/かわりに(修飾子)
- 何をするか(本体)
  - Javaの文

```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.moveby(int,int)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int)) ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));

  after() : move() {
    Display.redraw();
  }

  void FigElm.draw(Display d);
  void Point.draw(Display d) { … }
  … }
```

move後に redraw()せよ

バリエーション(後述)

AspectJの基本: ポイントカット (pointcuts)

何かが起きたときを指定

- 動作の種類 (メソッド呼出etc.)
- シグネチャ
- 合成

```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.moveby(int,int)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int)) ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));

  after() : move() { Display.redraw(); }

  void FigElm.draw(Display d);
  void Point.draw(Display d) { ... }
  ... }
```

FigElm.movebyまたはPoint.setXまたは・・・を呼出すとき

AspectJの基本: 型間宣言 (inter-type declarations)

既存の型に外から宣言を追加

- 既存の型
  - クラス・インタフェース
- 追加される宣言
  - メソッド・フィールド
  - extends・implements節

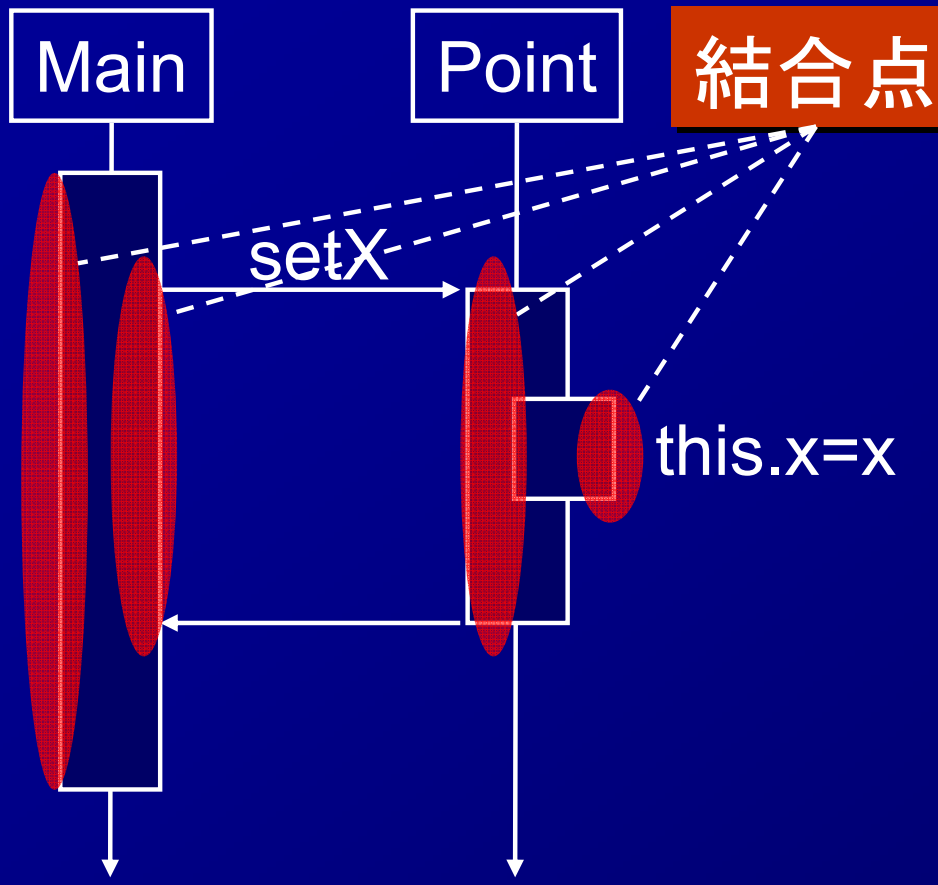
```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.move...
    call(...
    ...
    call(...

  after() : move() { Display.redraw(); }

  void FigElm.draw(Display d);
  void Point.draw(Display d) { ... }
  ... }
```

FigElmクラスにdrawメソッドを追加

```
declare parent MyTask:
    implements Runnable;
public void MyTask.run() { init(); }
```

MyTaskクラスにRunnableを実装

AspectJの基本: 結合点モデル (join point model)



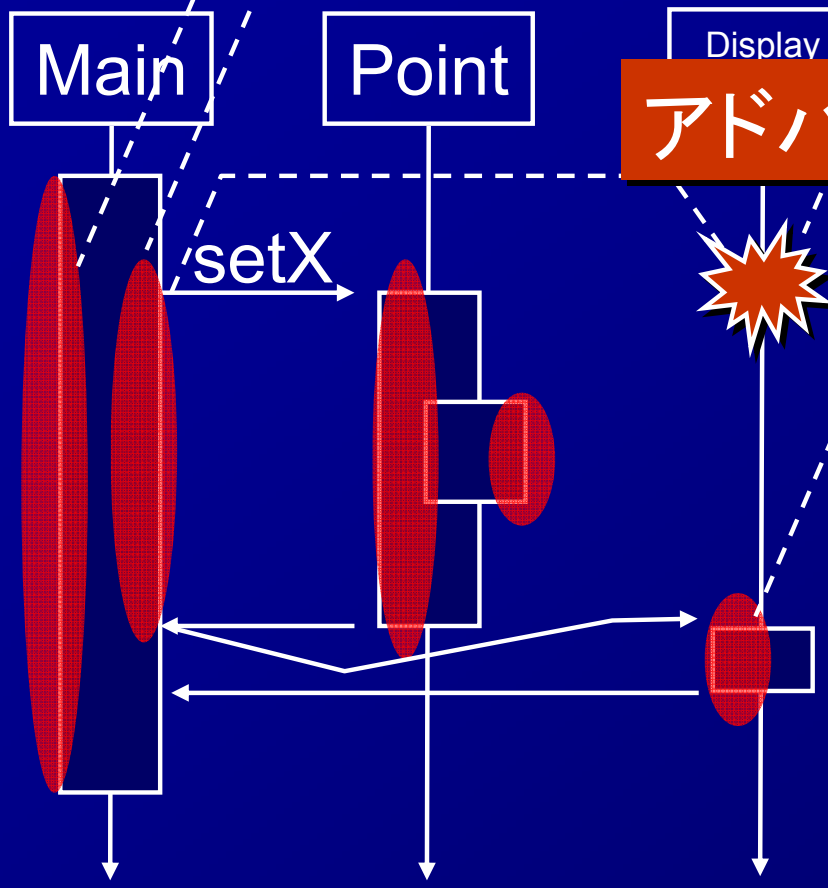
結合点 (join point)

= メソッド呼出、実行、フィールド代入などの実行中の動作

```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.moveby(int,int)) || ...;
  after() : move() { ... }
  void FigElm.draw(Display d);
  void Point.draw(Display d) { ... }
  ... }
```

結合点モデル

あくまで概念的なモデル



setXの呼出 ポイントカットに合致

アドバイスの実行

結合点ができる度にポイントカットが合致するアドバイスを探す

→beforeの本体;
本来の結合点;
afterの本体の順に実行

```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.moveby(int,int)) || ...;
  after() : move() { ... }
  void FigElm.draw(Display d);
  void Point.draw(Display d) { ... }
  ... }
```

ポイントカット

- AOP固有の機能
- 結合点を横断的に抽象化
  - 横断的: 複数のモジュールに関して
  - 抽象化:
    - 結合点の詳細を気にしなくてよい
    - 名前を付けられる
- 色々な指定方法

```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.moveby(int,int)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int)) ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));

  after() : move() { Display.redraw(); }
  … }
```

ポイントカットの種類

- 結合点の種類・シグネチャを指定:
  call, execution, set, get, initialization, preinitialization, staticinitialization, handler, adviceexecution
- 文脈情報の型検査+取り出し:
  this, target, args, @annotation
- スコープを限定: within, withincode, cflow
- 任意の条件式: if
- 論理演算: &&, ||, !

ポイントカット: ワイルドカード

- 図形要素の変化:「FigElmのmoveByまたはサブクラスのメソッド“set*”が呼び出された」とき

```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.moveby(int,int)) ||
    call(* FigElm+.set*(..));

  after() : move() { Display.redraw(); }
  ... }
```

ポイントカット: 様々な条件

- 色々な種類の条件
  - within(myapp.db..*): DBパッケージ内のみ
  - set(int Point.x): Point.xへの代入
  - withincode, execution, get, handler, initialization, static initialization
- 組み合わせる
  - call(* javax.swing..*(..)) && !within(myapp.ui..*)): UIパッケージ以外からのSwing呼出し

ポイントカット: 文脈情報の取り出し

- 文脈情報: 結合点に含まれる値
  - 例: 呼出元・レシーバ・引数等
- アドバイスへの引数になる

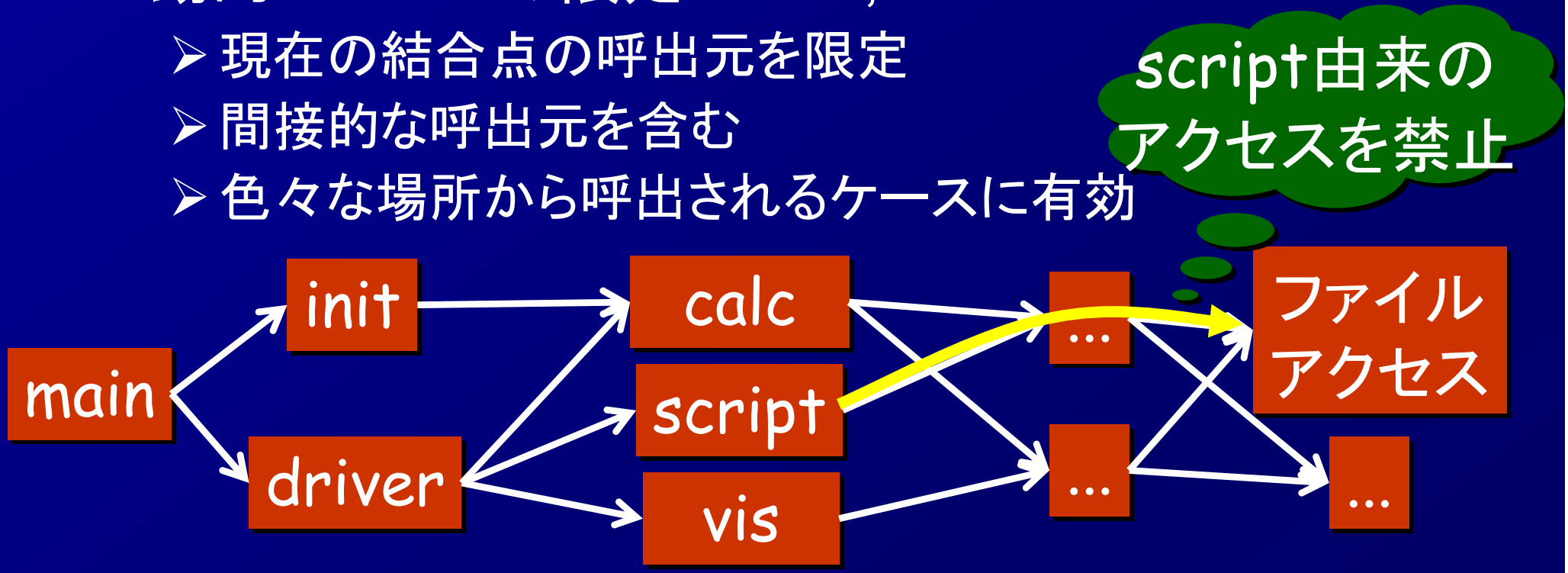
```
aspect DisplayUpdating {
  pointcut move(FigElm fig) :
    (call(int FigElm.moveby(int,int)) ||
     call(void FigElm+.set*(..)))
    && target(fig);

  after(FigElm fig) : move(fig) {
    Display d = fig.getDisp();
    d.redraw(fig);
  }
...
```

変更される図形が表示されている画面のみ更新

レシーバオブジェクトを変数figに束縛

ポイントカット: スコープの限定

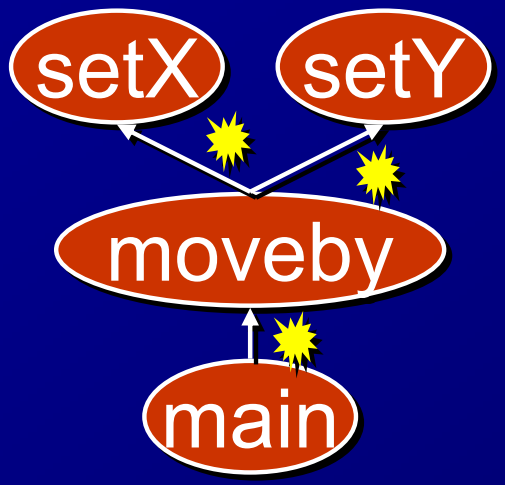
- 静的スコープの限定: within, withincode
  - 特定のパッケージ・クラス・メソッド中で起きた結合点だけを指定
- 動的スコープの限定: cflow, cflowbelow
  - 現在の結合点の呼出元を限定
  - 間接的な呼出元を含む
  - 色々な場所から呼出されるケースに有効



ポイントカット:
動的スコープの限定(1/2)

- p.moveby(2,3);は
  3回redrawを
  呼んでしまう

```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.moveby(int,int)) ||
    call(* FigElm+.set*(..));

  after() : move() { Display.redraw(); }
  ... }
```

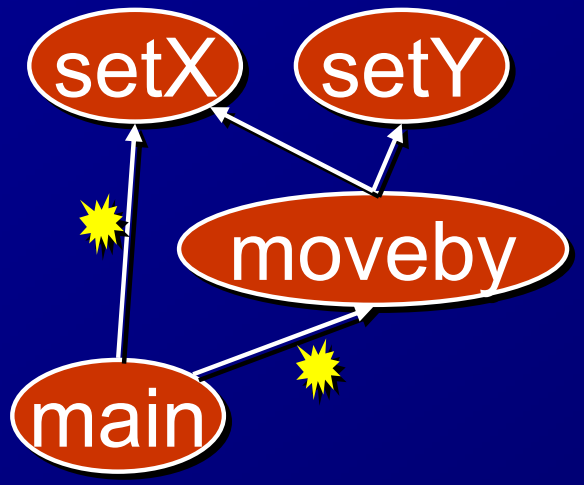



```
class Point ... {
  void moveby(int dx, int dy) {
    setX(getX()+dx);
    setY(getY()+dy); } }
```

ポイントカット:
動的スコープの限定(2/2)

- 呼出元を「外」に限定
  → 「setX, setYが movebyなどから呼出されていないとき」

```
aspect DisplayUpdating {
  pointcut move() :
    call(int FigElm.moveby(int,int)) ||
    call(* FigElm+.set*(..));
  after() : move() &&
        !cflowbelow(move()) { … }
  … }
```




```
class Point … {
  void moveby(int dx, int dy) {
    setX(getX()+dx);
    setY(getY()+dy);  } }
```

表現力の高いポイントカット

- 条件ポイントカット: 任意の条件式が書け、それが成立するときのみ一致
  - 「setXメソッドの引数が負の場合」は、引数を0に置き換える
  - 「compareメソッドのレシーバがnullの場合」は、DefaultComparotorに置き換える
- データフローポイントカット [MK03]: 値の間の依存関係を書ける
  - 「HTML.append()の引数がUserRequest.get()の返り値に依存する場合」は、引数をquoteしたものに置換
- Tracematch [Allan+05]: 連続する操作列に対する条件を正規表現風に書ける
  - 「あるFileReaderオブジェクトに、open(); (read() | write())*; close(); !close()が行われた場合」

アスペクトの再利用: 抽象アスペクト

- 一般的なアスペクトを定義し、それを再利用できる
  - 抽象アスペクト:「○○という挙動を追加・変更するが、どこに適用するかを決めない」

cf. 抽象クラスと抽象メソッド

- 抽象クラス:「○○という操作があるが、実際にどんな振舞かを決めない」

アスペクトの再利用: ログ取りアスペクトの例

- 抽象アスペクト:
  - どのようにしてログを取るか
  - 「どこ」でログを取るか →抽象ポイントカット
- 具体アスペクト:
  - ポイントカットを具体化
  - ログの取り方を気にしなくてよい

```
abstract aspect AbstractLogging {
  abstract pointcut logPoint();
  after() : logPoint() {
     …ログ取り動作… }
  … }
```

```
aspect DBLogging
        extends AbstractLogging {
  pointcut logPoint():
        call(* myapp.db..*.*(..));
}
```

参考文献

[HO93] Harrison and Ossher. Subject-Oriented Programming: A Critique of Pure Objects. In OOPSLA'93. pp.411-428. 1993.

[Colyer+03] Colyer, et al. Using AspectJ for component integration in middleware. In Practitioner Report, OOPSLA'03, 2003.

[Kiczales+97] Kiczales et al. Aspect-Oriented Programming. In ECOOP'97. pp.220-242, 1997.

[Kiczales+01] Kiczales et al. An Overview of AspectJ. In ECOOP'01. pp.327-353, 2001.

[MK03] Masuhara and Kawauchi, Dataflow Pointcut in Aspect-Oriented Programming, In APLAS'03, pp.105-121, 2003.

[Allan+05] Allan et al., Adding trace matching with free variables to AspectJ, In OOPSLA'05, pp. 345—364, 2005.