オブジェクトの広場

[Happy Squeaking!!]

― オブジェクト指向再入門 ―

[第一回] [第二回] [第三回] [第四回] [第五回]

第五回：デザインパターン事始め

ＩＮＤＥＸ

記事に対するご意見・ご感想をお待ちしています。
umezawa@tyo.otc.ogis-ri.co.jp
気軽にお寄せください。

「Happy Squeaking!! 第5回」をPDF形式でご覧になれます。

※ 本ページはスタイルシートを使用していますが、
PDF形式では一部スタイルが反映されず、
若干表示が異なりますのでご了承ください。

ご覧になるには、Adobe Acrobat Reader4.0が必要です。
以下のボタンをクリックしてダウンロードしてください(無料)。

HOME TOP

オブジェクトの広場

[Happy Squeaking!!]

１．オブジェクト指向設計とは

オブジェクト指向のメリットは、何といっても「もの」という人間的な認識の単位でソフトウェアを構築できるというところにあります

0と1のビット列で物事を考えているわけでない人間にとって、これは非常に有益なことと言えます。理想的には、分析者の頭の中に浮かんだ「ナチュラルなもの」が、そのままコンピュータ上に写像され、様々な問題解決処理が自動的に行われてしまえばこれに越したことはありません。

しかしそれほど甘くないのが世の中です。実際には、「現実世界のもの」に加えて*「現実世界には存在しない技巧的なもの」もコンピュータ上に用意してあげなければ、本当に役に立つシステムは作れない*のです。

通常ソフトウェアには、何らかの問題を解決するための要求が課せられますが、これはユーザの視点から見た場合、大きく分けて2種類に大別されます。

例えば、「銀行口座の預金を行わせたい」、「商品の受注を行わせたい」など、業務に比較的近い観点からの要求があります。これはシステムに対して論理的に実施してもらいたい「純粋な機能」といえます。

一方で、論理的な機能が実現されただけでは、ユーザが納得することはありません。「口座の預金処理のレスポンスは10秒以内でなければならない」「受注情報は二重化されたデータベースに保存すること」といった、実行性能や堅牢性などについての要求もあります。この場合は、機能とは別個のいわば「非機能的」な事項をシステムに求めていると考えられます。

また、今度はソフトウェアを使う側でなく、作る視点に立って考えてみましょう。「機能的」「非機能的」の両方について、ユーザの要求はたえず変化します。開発者側は、こうした変化する要求に対処するため、ソフトウェアの製作や、変更にかかるコストを最小限にしようと考えます。既存のライブラリの使用を検討したり、たとえ自分で作り込む量が多くなったとしても、なるべく次期のプロジェクトでも使えるように再利用性の高い部品としてソフトウェアを作りたいと考えるのが自然でしょう。こうした再利用性、保守性に関する要求は、「開発者側要求」とでも呼べるものです。

このように、現在のソフトウェアが解決しなければならない問題は、多くの側面を持っています。単に「機能的な要求」のみを満足させることを考えた場合には、その問題領域となる業務自体を熱心に分析することによって、オブジェクトとしてモデル化の作業を進めていくことができます。しかし「非機能的」な要求は、ネットワークのバンド幅やデータベースといった、純粋な業務とは別個の物理的なものに深く根差しているため、ナイーブな、業務だけを純粋に見ようという姿勢だけでは不十分なところがでてきます。この場合は視点を「業務」よりは「情報システム自体」に移さないと、問題解決の手助けとなるオブジェクトを切

り出していくことができません。更に、移り変わりの激しい現在のビジネスにおいては、ソフトウェアも常に変化していかねばならない宿命を背負っています。変化する部分をきちんと見定め、全体を作り替えずとも特定部分の取り替えによってシステムを容易に拡張できるような工夫を行っておかないと、すぐに使えないものなってしまいます。そのために、現在扱っている対象領域から、多少独立性の高い「プラグイン部品」としてオブジェクトをとらえていくことも時には必要になるのです。

従来のオブジェクト指向分析、設計方法では、比較的ナイーブな視点に立ち、「機能的な要求」のみを中心事項として扱い、その他の視点に関してはあまり注目していなかったのが実状です。しかし、オブジェクト指向による大規模、本格的なシステムの構築事例が増えてくるに従い、こうしたアプローチだけでは不十分なことが徐々に分かってきました。

「機能的」な要求に関しては、基本的に業務分析を行っていくことで対処できますが、それ以外の要求は範囲の外に置かれてしまいがちです。従来のやり方では、設計の段階で、各人の職人的な技量によって「なんとなく」設計用のオブジェクトが補完され、「機能的」以外の要求が充足されていくやり方が蔓延していました。(充足されない場合もありました)。

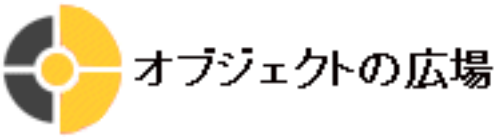
しかしソフトウェア作りを産業として見た場合には、その開発はもっと戦略的に行われていかねばなりません。

旧来のナイーブなやり方の反省として、Unified Processによるオブジェクト指向分析/設計方法では、「アーキテクチャ中心」という考え方が提唱されています。
この方式では分析の段階から、ユースケースによる業務分析に並行して、システムの実現に必要になるであろうアーキテクチャについての検討をしっかりと行います。これは業務分析に対してアーキテクチャ分析と呼ばれます。設計段階では、アーキテクチャ設計として更に検討が行われ、設計用オブジェクトがドメインモデルと依存性の少ない形で統合されていきます。
このようにUnified Processでは従来のOO開発プロセスに比べ「設計」の重要性が強調されています。ソフトウェア開発にまつわる様々な要求を実現させていくには、単純な業務のみの分析では不十分で、「設計」という作業を意識的に行っていかなければならないのです。

問題領域の分析だけで、一人前のシステムが出来上がらないといってなげくことはありません。「オブジェクト指向で現実世界そのものをあらわす」といっても、本当に必要なのは現実世界「そのもの」ではなく、現実世界をあらわして問題解決を行うソフトウェアという一種の「機械」だという認識が肝心です。機械には、それを動作させるための機構(メカニズム)があるのが当然であり、そのうまい機構や、機構を実現するための部品を作り上げていく「設計」作業は不可欠であると言えます。

一見、泥臭く、オブジェクト指向の基本から離れているようですが、この態度こそが現実を見据えており、正しい世界のとらえ方をしているということになるのです。

Index Next



[Happy Squeaking!!]

2．2つの基本概念

オブジェクト指向で設計を行っていく際に、不可欠となる概念として、「アーキテクチャ」、「フレームワーク」、「デザインパターン」があります。これらは一種のはやり言葉(buzzword)のように扱われ、相互に誤用されている傾向があります。これからオブジェクト指向設計を本格的に行っていこうという人は、これらの概念をしっかりと整理し、無用な混乱に巻き込まれないようにすることが肝要です。そこでまず、「アーキテクチャ」、「フレームワーク」について少し見てみることにしましょう。

オブジェクトの広場

[Happy Squeaking!!]

2. 2つの基本概念

2.1 アーキテクチャ

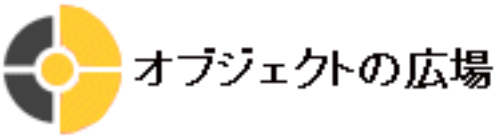
もともとは建築の用語で、ギリシャ、ロマネスクなど建築の様式を現すのに使われています。ある建築物が作られる際に、その建築がとるべき典型的な構造、装飾を示すものです。ある程度の規模の建築物を作ろうとする際には、居間、寝室、トイレといった各部屋を、バラバラに建てていってもうまくいかないことは目に見えています。部屋の移動しやすい配置、日当たり、風通し、収納、防音、耐久性など全体の構成についての配慮を最初に行っていかないと、建てた後で「見かけはいいが実は耐久性がすごく悪い」ということになってしまいます。アーキテクチャが事前にきちんと整理されていればこのようなことを防ぐことができます。

ソフトウェアの世界でも同じようなことがいえます。アーキテクチャは、ソフトウェアの開発プロセスにわたって横断的に存在し、まさにソフトウェア全体を支える骨格として機能しているのです。まずクラスの作り込みといった細部に入る前に、どんなDBを使うのか（RDB、ODB）、どんな通信形態を取るのか(クライアントサーバ、ピアツーピア)、どんな処理形態をとるのか(バッチ、並列処理、並行処理)といった方針をあらかじめ決めておかねばまともなシステムにはなりません。(こうした、比較的上流工程で行われる大きなレベルでの決定はマクロアーキテクチャと呼ばれます)。また、開発が進み、クラスを洗練させようといった段階に入った後にも、クラスやパッケージ間の構成を秩序だって決めていくといった、もう少し小さいスケールでのアーキテクチャ(ミクロアーキテクチャ)を選択していくことが重要なのです。

更に、ソフトウェアの世界でのアーキテクチャは、単に典型的構造にのみ焦点を当てているわけではなく、振る舞いについても規定していることに注意する必要があります。
再び一種の問題解決の機械としてソフトウェアを考えてみましょう。機械は、機能を実現するために何らかの流れを制御するメカニズム持ちます。この部分がまずい機械は通常うまく動作しません。各部品の修理や、組み替えもしにくく、耐久性や、拡張性も非常に悪いのです。
例えば、CDプレーヤの場合には、まずトレーがCDプレーヤをデッキの内部に取り込み、回転軸にCDをセットします。次にモータが回転し始め、ピックアップがレンズをCDの特定トラックに移動させて再生を開始します。このように各部品での処理担当と流れが明確に機構として定まっているのが、望ましい機械のあり方です。再生ができないという事態に陥ったときには、モータの回転段階までは正常だが、トラックへの移動が上手くいっていないのでピックアップの不具合だろうとすぐにわかります。
ソフトウェアも、どのようなデータが流れ、どのようにプロセスが起動し、どのモジュールが処理を行っていくのかという、典型的な振る舞いの規定があることによって、堅牢なものになりえます。

バッチ的に定められた入力データを加工して出力として返していく形式なのか、よりインタラクティブに、ユーザのリクエストに逐一反応するビューを提供するものなのかなど、どのような流れで全体の処理が進んでいくべきなのかについての決定も、アーキテクチャの一候補なのです。

アーキテクチャについては、マクロなものとしてとらえるか、ミクロなものとしてとらえるか、構造重視か振る舞い重視かなど、オブジェクト指向の設計者の間でも微妙な解釈の違いがあり、きちんとした定義としてはまだ不明瞭な部分があります。UMLの3-Amigosの一人、Boochさんの解説をもとに端的にまとめると、「*ソフトウェアの全体的な構造と振る舞いに関する戦略的な決定*」となります。



[Happy Squeaking!!]

2． 2つの基本概念

2．2 フレームワーク

フレームワークは、アーキテクチャにくらべ、実際の具体物が存在するという点で、より明確にとらえることができます。通常は、*「特定のアーキテクチャに基づき、実装された半完成のシステム」*を指します。「アーキテクチャの実例」といってもいいでしょう。

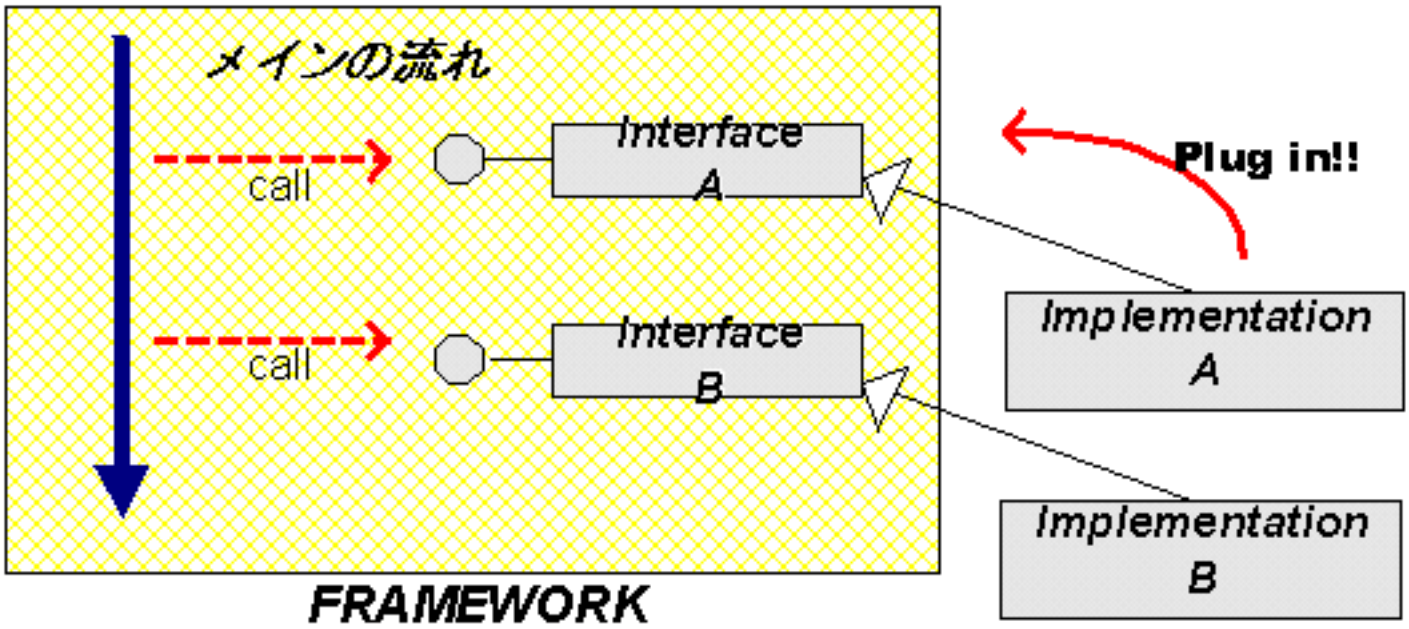
プレハブ住宅の建築法で、「ユニット工法」というのがあります。これは壁、床、天井、設備などをあらかじめ組み上げたユニットを作っておき、ユニットを全体の枠組みの中で相互に結び付けることで、家を作ってしまうといったものです。従来の、まず土台を用意し、柱と梁で支えと作って、筋交いで強度を高め、というやり方に比べ非常に短期間で作成でき、品質が大工さん個人の職人的技量に左右されないというメリットがあります。

フレームワークは、この「ユニット工法」の考えと非常に似ています。

アーキテクチャに基づいてソフトウェアを作るといっても、一からすべてを構築していったのでは非常に大変です。そこで、特定のアーキテクチャに基づいて実装されたできあいの半完成システムに対し、プラグインとして作成した(もしくはどこからか入手した)特定のソフトウェア部品をはめ込むことで、望みのシステムを手速く、安定したものとして作れるようにするのです。

フレームワークの代表例としては、古くは、柔軟なGUIを簡単に構築するMVCライブラリ、最近では、DBに接続するロジック層の構築を容易にするEJB(Enterprise Java Beans)、IBM SanFranciscoなどがあります。Appletの仕組みもまた、Thin-Clientを実現するための一種のフレームワークといえます。開発者はAppletのサブクラスを作成(もしくは入手)するだけで、WWWブラウザを通じ、サーバから動的にAppletを受け取ってクライアントとして実行させていくことが可能になります。

フレームワーク開発者側では、なんらかのプラグを用意しておき、プラグイン開発側は、そのプラグにうまくはまり込むような、カスタム部品を作成することになります。オブジェクト指向によるフレームワークの場合は、通常、フレームワークで提供する抽象クラスをサブクラス化したクラスを作り、はめ込むことになります。

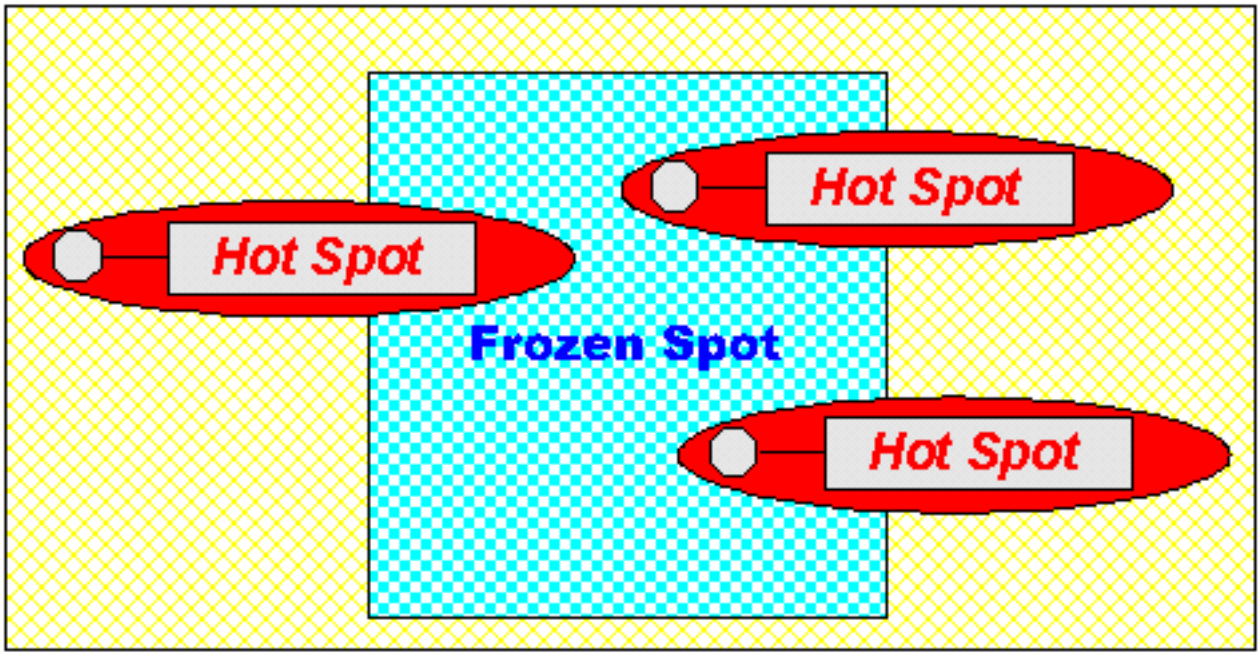


フレームワークの概念図

メインの流れはあくまでフレームワークが支配しています。この流れからプラグインが呼び出され、処理が進んでいきます。こうしたことで全体を制御しつつ柔軟なカスタマイズ性を実現するのです。

また、フレームワークで大事なのは、どの部分をプラグインできるようにし、どの部分を出来合いのものとして固定化するかということです。この選択を間違えると、手の込んだだけの使えないフレームワークになってしまいますので注意が必要です。

メタパターン(フレームワークを構成するためのデザインパターン)を定義したWolfgang Preeは、フローズンスポットとホットスポットという言葉で、このことを表現しています。フレームワークを作成する際に、ドメイン分析の結果、固定化しており、今後の変更が起こらないと思われる部分は、強固な流れを規定するフローズンスポットとなり、一方で新たな要求の発生が予想され、その変更に対し柔軟に対処できるようプラグを用意した部分がホットスポットとなります。



フローズンスポットとホットスポットの図

どの部分がフローズンスポットであり、どの部分がホットスポットであるかの区別は自動的にできるものではありません。この分割を行っていくには、開発者側の都合を考えただけではだめで、ユーザから得るアプリケーションのドメインに関する知識が不可欠です。ユーザの要求の中で変化しやすい部分とそうでない部分を見定めて「分けていく」作業は、まさに「分析」作業ということなのです。
ただし変化しやすい部分をホットスポットとして、実装コードの中に適切に表現していくには、単にこの切り分けができているだけでは不十分です。実際にフローズンスポット、ホットスポットをコードとして組み上げていくには、そのための道具といったものが必要です。

そこで、登場するのがデザインパターンということになります。

デザインパターンの説明に入る前に、今までの話の具体的なイメージをつかむため、Squeakを題材にMVCアーキテクチャ、更にそのフレームワークとしての実装を見てみましょう。

オブジェクトの広場

[Happy Squeaking!!]

3．事例：MVCアーキテクチャ

3．2　MVCとは

MVCは、「対話型のアプリケーションを作成するためのアーキテクチャ」といえます。Squeakに限らず、多くのSmalltalkが、このMVC(もしくはその発展形)と呼ばれるアーキテクチャを用い、高度にインタラクティブなGUIを実現しています。

MVCはそれぞれ、Model、View、Controllerを意味しています。Smalltalkはもともと、単なるオブジェクト指向言語というよりは、DynaBookという、あらゆる人が鉛筆、紙、本のように使える理想のコンピューター用のOS言語として考え出された経緯が有り、「対話性の実現」を非常に重視していました。

ビットマップディスプレイやマウスを使うというアイデアも、現在では当たり前となりましたが、当時としては奇抜なものでした。これは友達のように話し掛けられるコンピュータとしてDynaBookを作り上げるため、必要不可欠なものとして考え出されたのです。もともとコンピュータが扱っていたデータ(一種のModel)に加えて、ディスプレイ(一種のView)、マウス(一種のController)をどのように上手く扱っていけばいいのかということは、当時Smalltalkに課せられた重要な要求だったわけです。

時は過ぎ、Smalltalkによって始めて導入されたMVCアーキテクチャは、その後のMacApp、ET++、MFCのDoc-View、JavaのSwingといった具合に、様々なGUIフレームワーク上で少しづつ形を変えて採用され、その有効性が実証されてきました。(MVCが万能というわけではなく、この期間に欠点を補うための試行錯誤もされています)。MVCアーキテクチャは現在ではGUIだけでなく、より広い意味で使われる傾向にあります。UMLでのEntity、Boundary、Controlは、オリジナルのMVCとは性質を変えていますが、基本的な発想は紛れもなくMVCから来たものです。

このようにMVCには様々なバリエーションがありますが、以後は、クラシックなMVCについて説明することにしましょう。

オブジェクトの広場

[Happy Squeaking!!]

3．事例：MVCアーキテクチャ

3．2 MVCの3つ組

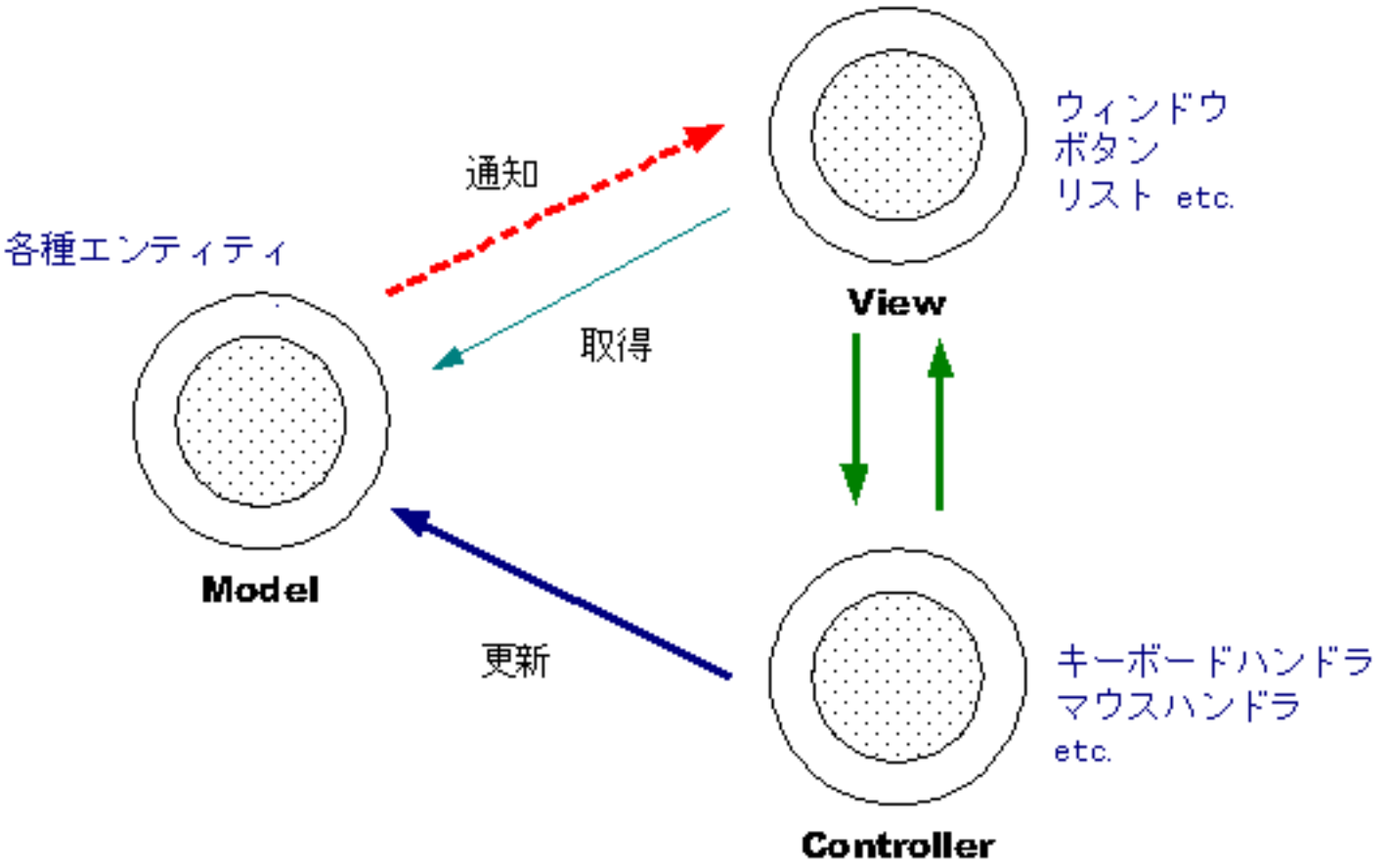
MVCでは、対話的なアプリケーションを実現するため、アプリケーションの構成要素を、処理、表示、入力の3つ組みに分割することに決めまています。この分割が、MVCアーキテクチャの基本的な構造を規定しています。

処理を受け持つ部分は、Modelと呼ばれます。アプリケーションで必要となる実際のデータを保持しており、業務に特化した処理を実行します。オブジェクト指向の用語を使うと、エンティティやビジネスオブジェクトと呼ばれるものを指します。具体例としては、銀行口座オブジェクト、受注オブジェクトといったものが該当するでしょう。

Modelの状態を表示する部分はViewになります。ビットマップディスプレイ上に表示されるウィンドウや、その中のウィジェット群(メニュー、ボタン、リスト、スライダなど)がビューの候補になります。ビューは、モデルと比べユーザの好みといったものが介在します。そのためモデルに比べて、非常に多様で移り変わりが激しくなる傾向があります。例としてテレビの選挙速報をイメージしてみてください。候補者の現在の獲得投票数といった同じモデルについて、各局で、わかりやすさやインパクトを狙い、実に様々な表示を行っているのを見たことがあるでしょう。時にはグラフによる表示が好まれ、また時には数値表示が好まれる場合もあります。こうした性質から、ビューはなるべくモデルとは分割されているのが望ましいということになります。分割を適切に行うには、ビューの役割は単に表示を行うのみに限定される必要があります。

3つ組みの最後はControllerです。コントローラは、マウスやキーボードからの入力をイベントとして受け取る役割を受け持っています。GUIの見栄えと操作感覚を現す言葉にLook&Feelという言葉がありますが、MVCにあてはめるとLookがビューでFeelがコントローラということになります。コントローラはOSからのマウスやキーボードのクリックといったプリミティブなイベントの中から、自分に関係のあるものを選択、処理し、適切な形に変換してモデルに伝達します。古典的なMVCでは、イベントの処理に加えてアプリケーション全体の制御を行うこともしばしばあります。

Model-View-Controllerの3つ組みの関係は、図で示すと以下のようになります。



MVCの3つ組

コントローラが、OSからイベントを受け取り、モデルにメッセージを送り、モデルの状態を更新します。モデルの状態変化はビューに通知され、通知されたビューは、モデルの最新の値を取得し、再表示を行うというのが基本的な流れになります。

3つ組は、それぞれ独立しているのが望ましく、特にモデルとビューの切り離しは非常に重要です。「*ビューやコントローラはモデルを知っているが、モデルはその他の要素については何も知らない*」という状態になるのが理想です。そうすることでモデル部分に一切の変更を加えることなく、様々な種類のビューやコントローラをプラグインできるようになります。

とはいうものの、少なくともモデルからビューに対して状態変化を通知する仕組みができていないと、ビューは再表示を行うことができません。従って、この「*モデルからビューへの通知用のメカニズム*」は別途考える必要があります。この通知をできる限り「間接的」におこなわせることによって、相互の独立性を高めることができるのです。

モデルからビューへの、状態変化に伴う間接的な通知の仕組みはいろいろな形で実装することができます。特にSmalltalkでは、*dependency*という仕組みを利用して実現しています。(後に説明します)。

オブジェクトの広場

［Happy Squeaking!!］

4．Squeak演習: MVCアプリケーションの作成

4．1　MVCのブラウズ

それでは、Squeakに実装されている「MVCフレームワーク」を利用して、実際にMVCアーキテクチャにのっとったアプリケーションを作成してみることにしましょう。.

ちなみに今回からは最新のSqueak2.4を使用します。
MVCは「枯れた」フレームワークとしてSqueakに実装されていますので、その部分での大きな変更はないと思いますが、古いバージョンをお使いの方はこの際バージョンアップしてみてください。本連載では紹介しきれないすばらしい機能が盛りだくさんです。

また、以後のビットマップは読みやすさのため、デフォルトのフォントサイズを一段階大きくしています。(ワークスペースで "TextStyle changeDefaultFontSizeBy: 1"と実行)。

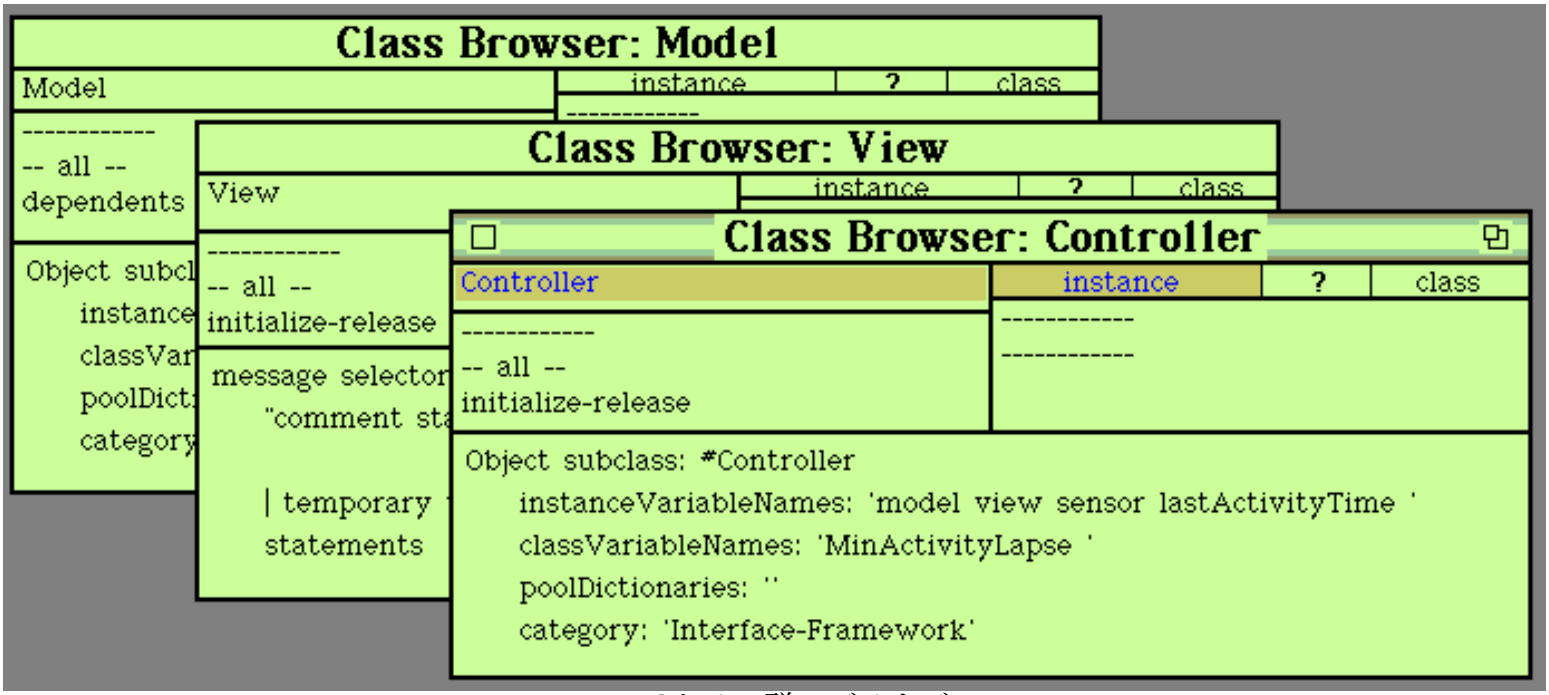
Squeakには、Model、View、Controllerという抽象クラスが用意されており、必要に応じてそれぞれをサブクラス化してカスタマイズ化することで、自らの望むGUIアプリケーションを作っていくことができます。

まずは、各クラスを軽くブラウズしてみましょう。

以下の式をワークスペースで順番に"*do it*"します。

```
Browser newOnClass: Model.
Browser newOnClass: View.
Browser newOnClass: Controller.
```

Model、View、Controllerの各クラスについてのブラウザが立ち上がります。

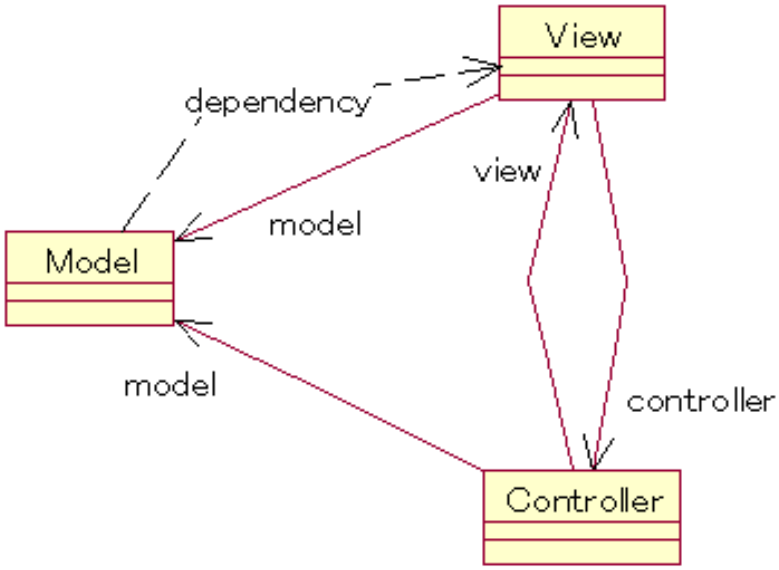


MVCクラス群のブラウズ

それぞれ非常に多くの操作と振る舞いが定義されています。これらはMVCの仕組みを実現するための基本的な機能を実現するために提供されているものです。

MVCに準拠したアプリケーションを開発する側は、通常はこれらクラスの中身の詳細について知る必要はありません。MVC各担当のそれぞれの役割、大体の動作を把握し、サブクラスで典型的にオーバーライドするメソッドを知るだけで十分です。

SqueakでのMVCをUMLのクラス図で書くと以下のようになります。(詳細は省略しています)。



SqueakでのMVC

モデルからビューへの間接通知がdependencyという依存関連で示されていますが、基本構造はオーソドックスなMVCそのものです。

それでは、Model、View、Controllerについてそれぞれサブクラスを定義していくことにしましょう。

オブジェクトの広場

[Happy Squeaking!!]

4．Squeak演習: MVCアプリケーションの作成

4．2　Counterアプリケーションの作成

今回はMVCの基本的な仕組みを把握するためになるべくシンプルなアプリケーションを作成することとします。題材とするのは、数値を数えるカウンターです。
以下のような簡単な仕様とします。

> カウンターは、数を数えるためのアプリケーションである。
> ユーザの入力に従い、整数値を上下させることができ、現在の値をユーザが確認できる。

オブジェクトの広場

[Happy Squeaking!!]

4．Squeak演習: MVCアプリケーションの作成

4．3　Counterモデルの作成

まずはCounterのモデル部分を作成します。
Counterモデルは内部の属性としてdataという名前の変数を持ち、基本的な操作として、increase(増加)、decrease(減少)の二つを持つとします。

Modelのサブクラスとして Counterクラスを以下のように定義します。

```
Model subclass: #Counter
        instanceVariableNames: 'data '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'MVC tutorial'
```

dataを設定、取得するための操作としてdata、data:の二つを定義します。

(Counter >> accessingカテゴリ)

```
data
        ^data
data: anInteger
        data := anInteger.
        self changed: #data
```

data:のメソッドの実装の末尾に注目してください。self changed: #dataという何やら怪しげな記述があります。実はこれが「ビューに対する間接的なメッセージ送信」を行わせるためのSqueak(Smalltlak)での書き方になります。data変数に値の更新があったことを通知はするが、どのViewに対してかは知らないといった書き方になっています。詳しくは後に説明します。

次に、increase、decreaseの基本的な操作を定義しましょう。

(Counter >> actionsカテゴリ)

```
increase
        self data: self data + 1.
decrease
        self data: self data - 1.
```

更にdata変数の初期化用メソッド、initializeを定義します。

Counter >> initialize-releaseカテゴリ)

```
initialize
        data := 0
```

最後は、インスタンス生成用のメソッドnewを"*class*側"に定義します。これで生成時に、自動的にdata変数の初期化が行われるようになります。

(Counter class>> instance creationカテゴリ)

```
new
        ^super new initialize
```

以上でCounterモデルの実装は終わりです。data変数の更新時に、data変数の更新時に、self changed: #data.というおまじないを書く以外は、今までのクラスの実装と何ら違っていないことに気づかれるでしょう。

さて、今までのソースです。

FileIn: Counter.st (<=Click)

オブジェクトの広場

[Happy Squeaking!!]

4．Squeak演習: MVCアプリケーションの作成

4．4　Counterコントローラの作成

次はCounterモデルを外部から操作するためのコントローラを作成していくことにします。

まず、抽象クラスControllerクラスの定義をみてみましょう。
Controller definition. "*print it*"

```
Object subclass: #Controller
        instanceVariableNames: 'model view sensor lastActivityTime '
        classVariableNames: 'MinActivityLapse '
        poolDictionaries: ''
        category: 'Interface-Framework'
```

となっています。重要なのはmodelとviewとsensorの各属性です。
コントローラはmodelとviewの変数によってモデルとビューの両方を参照できます。(*"SqueakでのMVC"*の図を参照のこと)。Modelへの参照は、コントローラがモデルに更新の動作を起こさせるために必要です。また、viewへの参照はコントローラ側からモデルを経由しないでビューの再表示をさせる場合に使われます。(典型的にはウィンドウのリサイズなどです。この時はモデルそのものとは無関係に表示をドラッグにあわせて変えなければなりません)。

一方sensorは、OSから渡されるイベントを捕らえるために使用されます。全てのコントローラは、イベントをキャッチするためのセンサを持つことになります。センサに問い合わせることによって、今どのような入力イベントが起こっているかを知ることができます。センサからはキーボードに関するイベントや、マウスに関するイベントを取得できます。(どのようなイベントが取得できるかに興味のある方は、InputSensorクラスをブラウズしてみてください。実際にはこのクラスのインスタンスがsensor変数にセットされることになります)。

さて、Modelの場合と同様、このControllerから直にサブクラス化して、Counter用のコントローラを定義したいところですが、ここではもう少し高レベルで扱いやすいStandardSystemControllerから継承することにします。StandardSystemControllerは、Squeakにウィンドウとして表示されるアプリケーションのためのコントローラとしての機能を持っています。MVCで作成されたSqueakのウィンドウはむしろ生のControllerでなく、こちらのStandardSystremControllerの方を使用しています。

ここではStandardSystremControllerの詳しい説明については、省略します。フレームワークを使う側はその細部について全てを知っている必要はありません。各ブラウザ用に実装されているApplet用のClassLoaderや、SecurityManagerの細かな実装を把握していなくともAppletを使ったアプリケーションが作成できるのと同じことです。

Counter用のコントローラはできるだけシンプルなものを目指します。ユーザからの入力は、主にキーからのものとマウスからのものが考えられますが、もっとも基本的な、キーからのイベントを受け付けて処理するコントローラにします。u(up)のキーで増加、d(down)のキーで減少という仕様にしましょう。

クラス定義は以下のようになります。

```
StandardSystemController subclass: #CounterKeyboardController
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'MVC tutorial'
```

操作としては以下の一つを定義するだけです。

(CounterKeyboardController >> control activityカテゴリ)

```
controlActivity
        | key |
        super controlActivity.
        key := self sensor keyboard.
        key == $u ifTrue: [self model increase].
        key == $d ifTrue: [self model decrease].
```

controlActivityは、コントローラがアクティブと判断されたとき(通常は、カーソルがそのコントローラの管理するビュー上でクリックされた時など)に自動的に起動されます。

OSからVM経由でコントローラに制御権が渡されていく、スケジューリングの仕組みについては、もちろんSqueakで実装されていますが、今回はMVCの基本に集中してもらうためあえて説明を避けます。

controlActivityメソッドのソースをコメントつきで解説すると以下のようになります。

```
controlActivity
        | key |
        super controlActivity.

        "センサからキーボードについてのイベントを取得"
        key := self sensor keyboard.

        "キーの値が$u(up)の場合はmodelに増加のリクエスト"
        key == $u ifTrue: [self model increase].
        "キーの値が$d(down)の場合はmodelに減少のリクエスト"
        key == $d ifTrue: [self model decrease].
```

最初のsuper controlActivityは、uかdのキーが入力された以外のユーザからの入力のハンドリングを適切に行うために必要です。具体的には、マウスで、ウィンドウのタイトルバーの最小化ボタンを押したときや、リサイズを行うためドラッグした時の処理などをスーパークラ

ス(StandardSystemController)のcontrolActivityで適切に行っているのです。

以上でCounter用コントローラの実装は終わりです。

ソースは以下のようになります。

FileIn: CounterKeyboardController.st (<=Click)

オブジェクトの広場

[Happy Squeaking!!]

4．Squeak演習: MVCアプリケーションの作成

4．5　Counterビューの作成

それでは3つ組の最後としてビューを作成することにしましょう。

まずは抽象クラスViewのクラス定義をみてみます。
View definition. "*print it*"

```
Object subclass: #View
        instanceVariableNames: 'model controller superView subViews
 transformation viewport window displayTransformation insetDisplayBox
 borderWidth borderColor insideColor boundingBox '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Interface-Framework'
```

伝統的なMVCの場合、Viewの役割はかなり広範です。ここでもすべてを把握する必要はありません。重要なのはmodel、controllerの各属性です。model変数によるモデルへの参照は、モデル更新時に最新の値を取得するために使われ、controller変数によるコントローラの参照は、ウィンドウビューがリストビューを含んでいるなど、ビューが入れ子関係になっている時に、子供のビューのコントローラに制御権を渡す場合などに役立ちます。(Viewの入れ子については今回のサンプルでは取り扱いません)。

Counter用のビューですが、ウィンドウとして表示されるようにしたいので、例によってViewそのものではなく、そのための機能が実装されたStandardSystemViewのサブクラスとして作成します。
表示の仕方についてはいろいろ考えられますが、ここではやはりシンプルに、テキスト形式で現在のモデルの値が表示されるのみとします。

クラス定義は次のようになります。

```
StandardSystemView subclass: #CounterTextView
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'MVC tutorial'
```

カウンタの値についての情報はCounterモデルで管理されているので、View側では一切Counterの値について情報をもつ必要がありません。結果としてCounterTextViewは属性な

しのクラス定義となっています。

ビューで行うことは何といっても"表示"ですから、そのための操作を定義します。

(CounterTextView >> displayingカテゴリ)

```
displayView
        | countStr text displayText |
        super displayView.
        self clearInside.
        countStr := self model data printString.
        text := Text string:  countStr attribute: (TextEmphasis bold).
        displayText := text asDisplayText.
        displayText foregroundColor: Color black backgroundColor: Color white.
        displayText displayAt: self insetDisplayBox center
```

といった具合になります。
幾つか新しい事項が出てきています。順番に説明します。
まずdisplayViewですが、ビューの再描画が必要になった時に自動的に起動されます。デフォルトでは、モデルの状態とは無関係に、ウィンドウのリサイズや移動を行った際に呼ばれるようになっています。

新たなビューを定義した場合には、最低でもこのメソッドを実装しないと適切な表示を行わせることができません。

displayViewの中身ですが、コメントつきで示すと以下のようになります。

```
displayView
        | countStr text displayText |
        "デフォルトの再表示処理を実行"
        super displayView.

        "ビュー内部の表示をリフレッシュする"
        self clearInside.

        "モデルから、Counterの値を文字列形式で取得"
        countStr := self model data printString.

        "ボールド体のテキストを生成する"
        text := Text string:  countStr attribute: (TextEmphasis bold).

        "テキストを表示テキストに変換"
        displayText := text asDisplayText.

        "表示テキストに色をつける"
        displayText foregroundColor: Color black backgroundColor: Color white.

        "ビュー内部の中心部に表示テキストの描画"
        displayText displayAt: self insetDisplayBox center
```

説明が特に必要なのはTextと思います。
TextはStringと違い、単なる文字列でなく、ボールドである、イタリックであるなどの属性

を持つ文字列です。Textはそのままではビットマップ表示を行うことができず、Textを一旦DisplayTextに変換しなければなりません。DisplayTextはDisplayObjectのサブクラスで、displayAt: やdisplayOn: at: を使うことでビットマップディスプレイに表示を行わせることができます。

コード例を示すと以下のようになります。

```
| text disText |
"イタリックのテキスト属性を生成"
att := TextEmphasis italic.

"イタリック属性の設定されたテキストを生成する"
text := Text string: 'HelloSqueak' attribute: att.

"DisplayTextのインスタンスへと変換"
disText := text asDisplayText.

"現在のカーソル位置に表示"
disText displayOn: Display at: Sensor cursorPoint
```

Displayは、Squeakビットマップ画面の全体を指すグローバルなオブジェクトです。Sensorも、Squeak全体に関するグローバルなセンサです。

さらに興味のある方は Text、TextEmphasis、DisplayObject、DisplayTextの各クラスをブラウズしてみてください。

SmallTip:Textは、ボールド、イタリックなどの属性をもつ文字列

SmallTip: DisplayObjectは、displayAt: によって表示を行わせることができる

次に、モデルからデータの更新があったときに呼ばれるupdate: メソッドを定義します。これは、Counterの値の変更があった段階でSqueakのdependency機構の働きによって自動的に起動されます。

モデルからの通知があった際に、ビューはアップデートを行うべきなので、以下のように定義します。

(CounterTextView >> updatingカテゴリ)

```
update: anAspect
        Transcript cr; show: 'now update'.
        "先に定義したdisplayViewを呼び出す"
        self displayView.
```

引数anAspectはとりあえず無視してください。ビューのupdateをより細かい単位で行う際に使用されます。

最後に、このビューがデフォルトで使うコントローラを示すためのメソッドを定義します。CounterTextViewは、先ほど作成したCounterKeyboardControllerと協調して動作しますので、以下のようなメソッドを作成することになります。ビューは、外部からコントローラが設定されていない場合、このメソッドを呼び出して自身が使うコントローラとします。

(CounterTextView >> controller accessカテゴリ)

```
defaultControllerClass
        ^CounterKeyboardController
```

これでビューの作成も終わりました。
以下は全体のソースです。

FileIn: CounterTextView.st (<=Click)

オブジェクトの広場

[Happy Squeaking!!]

4．Squeak演習: MVCアプリケーションの作成

4．6　Counterアプリケーションの起動

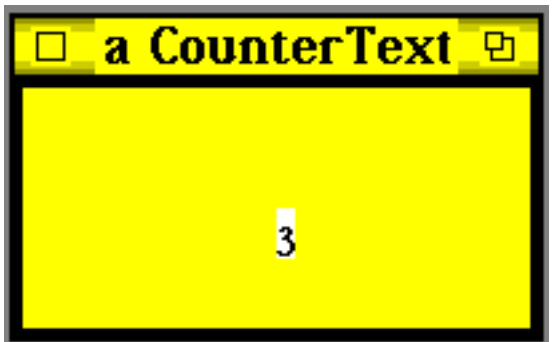
CounterについてのMとVとCができたので、それぞれを結び付けて起動してみましょう。

以下の式をワークスペースで実行してみてください。

```
| counterView |
        counterView := CounterTextView new.
        counterView model: (Counter new).
        counterView label: counterView name.
        counterView maximumSize: 200 @ 100.
        counterView minimumSize: 50 @ 30.
        counterView borderWidth: 5.
        counterView backgroundColor: Color yellow.
        counterView controller open.
```

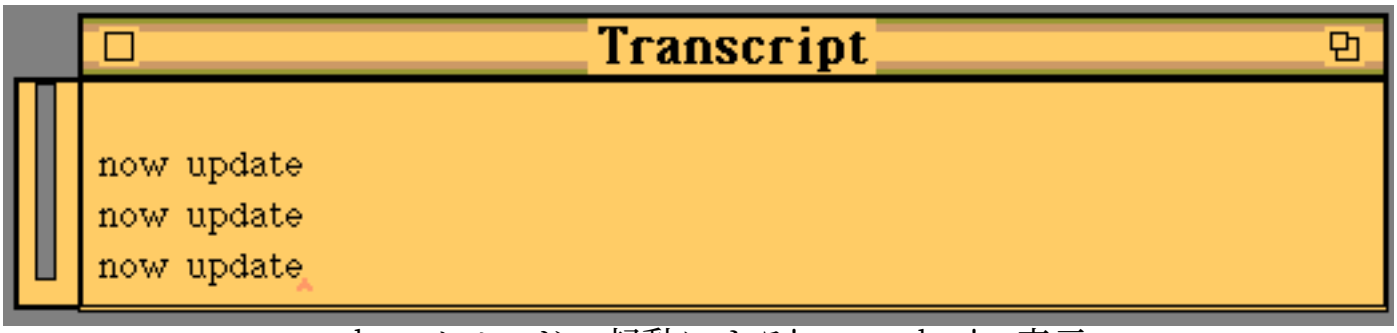
基本的には、CounterTextViewを作成して、モデル(Counter)を設定して、起動しているだけです。コントローラは、明示的に設定していないのですが、CounterTextViewのdefaultControllerClassメソッドを定義しているので、自動的にCounterKeyboardControllerのインスタンスが生成されセットされた状態で立ち上がります。

以下のような画面になります。



Counterアプリケーションの起動

キーボードのuとdキーで値が変化するのが確認できます。
Transcriptにはビューの再描画ごとに'now update'と表示されていきます。



update:メソッドの起動による'now update'の表示

MとVとCと、合わせてコード量は2k足らずです。あらかじめMVCのアーキテクチャに基づいたフレームワークが提供されていることにより、このような生産効率の高さが実現されているのです。

また、各オブジェクトが明確に分割されているので、後の拡張も非常に容易です。次の演習では、実際に拡張を行っていきます。

Prev. Index Next

オブジェクトの広場

[Happy Squeaking!!]

5. デザインパターンとは

5．1　パターン誕生まで

デザインパターンは、もともと、フレームワークの作成過程の中で、なるべくソフトウェアを再利用性が高く、効率も良く、また柔軟なものとしたいという開発者の要求から、自然発生的に生じてきたものです。
実装としてのフレームワークは、特定の言語によるクラス定義とその関連から構成されています。これを楽曲にたとえると、その中には、その全体としての効果を高めるため、繰り返し特定のフレーズが現れて出ているのです。
80年代の後半ごろから、このフレーズ(パターン)をうまく取り出して、整理することで、後のフレームワーク作りの構築要素として役立てることができるのではと設計者は徐々に気づき始めました。
そして1987年、OOPSLAというオブジェクト指向に関するカンファレンスで、記念碑的な論文が発表されます。Kent Beck、Ward Cunninghamの二人による、"Using Pattern Languages for Object-Oriented Programs" です。
この論文は、建築家である、Christopher Alexanderが、クライアントの要求にかなった建築を作り出すために抽出した「パターン」の考えをソフトウェアにも応用できるのではと示唆したものでした。

Christopher Alexanderは、建築はそこに実際に住む人の要求を満たした形で作られなければならないとしています。しかし、実際の作業を行っていくためには、建築に対する詳細かつ専門的な知識が必要になり、そのままでは単に理想を述べただけのものとなってしまいます。そこでAlexanderが提唱したのは、「パターン」の考えでした。建築の過程で生じてくる様々なトレードオフや問題、そしてそれに対する典型的な解決の方法を、誰もがわかりやすいパターンとし、カタログ化して示すことにより、設計者が自由にパターンを選択して、より要求にかなった建築の手がかりとしていくという方式を提唱したのです。

ソフトウェアも、要求をかなえるための成果物を構築しなければならないという点において、建築の世界と非常に似た部分があります。
開発の際にありがちな問題に何度も遭遇している設計者がいて、その問題は大抵の場合、同じような対処方法で解決されていたのです。しかし、そうした解決策は一部のスキルのある技術者の頭の中にノウハウとして貯えられているのみで、きちんとドキュメント化されることはありませんでした。
そこでSmalltalkのMVCフレームワーク上でGUIアプリケーション開発を続けてきた、Kent BeckとWard Cunninghamは、GUIの構築について幾つかの「パターン」をカタログ化して提示することで、MVCの複雑な全貌を理解しなくとも、一定の基準を維持したアプリケーションを作成できるのではと主張したのでした。

フレームワークについて全ての知識がなくとも、特定パターンに準拠することで不慣れな技術者でも正しい方向に導いていくことができます。フレームワークを正しく発展させるための構築要素として、まずパターンは産声をあげたのです。

オブジェクト指向の再利用というと、とかく「クラスライブラリの再利用」といった"what"に関する再利用性ばかりがクローズアップされてしまう傾向がありました。しかしデザインパターンは、今まできちんと形あるものとして示されなかった「*whatを生み出すためのknow-how*」("how")の再利用について焦点をあてたところが画期的だったのです。

オブジェクトの広場

[Happy Squeaking!!]

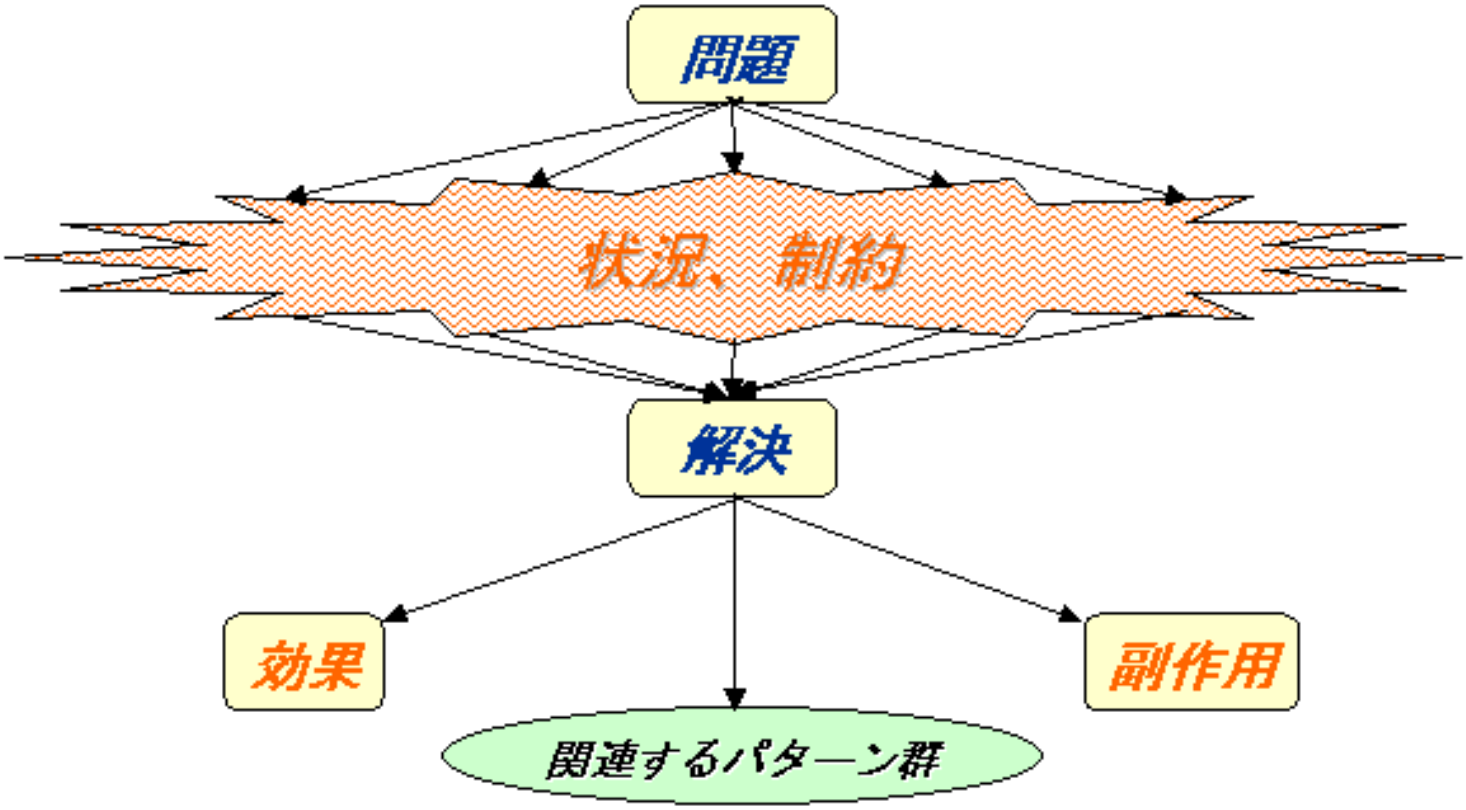
5. デザインパターンとは

5．2　パターンの構成要素

ボトムアップ的に生まれてきたオブジェクト指向での「パターン」の概念は、年月を経て徐々に整理され発展してきました。現在では「パターン」という言葉はもう少し広い意味で使われる傾向にあります。

「パターン」という考え方は、オブジェクト指向の設計のみに適用できるというものではなく、実際にはもっと上流の分析、また下流の実装においても有効なものということがわかってきています。
この広義の「パターン」を一言でいえば、「*ある状況下で発生する、典型的な問題についての有効な解決策に名前をつけたもの*」となります。

概念的な図で示すと以下のようになります。



パターンの概念図

基本的には、「問題」とそれに対する「解決策」が示されていればいいのですが(これがパターンの最小構成要素となります)、ドキュメントとしての厳密さやわかりやすさを狙って、通常はもう少し細かな単位でパターンを記述します。

まず、「*問題*」があります。問題は、典型的なものであり、「今回に限って」という性質のものであってはいけません。特定の状況で常に発生する問題についてでなければ、再利用の対象にはならないからです。

問題には「*特定の状況*」が付随しており、更に、問題からは様々な「*力、制約(Forces)*」が発生します。制約は相反する方向に働き、通常トレードオフが発生します。(柔軟性を高めるようとすると、パフォーマンスが悪くなるなど)。様々な方向に発散している制約のうち、どれを優先度の高いものとして解決していくかが示されないと、パターンを誤った場合に適用してしまう可能性があります。良いパターンとは、問題に伴う制約をなるべく広い範囲で洗い出しているものです。(全てを解いているものではありません。それは通常不可能なことです)。

次に、発散している制約のうち、対象とするものについて、典型的な「*解決策*」を示します。解決策は、有効性が実証されたものでなければなりません。「今回に限って、たまたまうまくいった」というものでは再利用できないからです。３回程度は実施に移され、成功が認められたものであるべきです。解決策は、解決に必要な要素間の*構造的な側面*に加えて、それらの*振る舞いの仕方*についても述べておかないと完全とはいえません。

パターン適用の後には、実際の効果と、未解決の制約と、トレードオフによる副作用が残されます。これはいわば「*新たな状況*」の発生であり、その状況から次の問題、次に適用すべき「*関連するパターン*」が生まれてきます。

以上がパターンの大体の構成要素になります。最後に、これら全てを説明しなくとも、どのパターンについての話をしているのかがすぐにわかるように、「*インパクトのある名前*」をつけておきます。ボトムアップ的に複数の異なる分野からパターンが生まれてきた場合には、特定の分野にとってわかりやすい「*別名*」を幾つかつけておく場合もあります。

オブジェクトの広場

[Happy Squeaking!!]

5. デザインパターンとは

5．3　パターン分類とデザインパターンの位置づけ

パターンは、それが扱う問題のスケールや種類によっていくつかに分類することができます。

「ソフトウェアアーキテクチャ」(トッパン、ISBN 4-8101-9007-2)の中では以下のようにパターンが分類されています。

- アーキテクチャパターン
- デザインパターン
- イディオム

アーキテクチャパターンは、システム全体を支配するような大きな問題解決のために、特定のアーキテクチャの適用方法やトレードオフを説明したものです。ここでのアーキテクチャはマクロアーキテクチャのことを指し、この分類中では、もっともスケールの大きな問題を取り扱うことが意図されています。サブシステム間の巧みな構成、振る舞いといったものがメインテーマです。MVCなどがパターン例として示されています。

デザインパターンは、おそらく皆さんがもっとも聞いたことのある言葉でしょう。パターンの元祖とも言えるもので、主にミクロアーキテクチャ的な視点から、クラス間の関係や相互作用について示したものです。フレームワークの構築ブロックとして役立ちます。後に説明するObserverパターンなどが代表例となります。

最後が、イディオムと呼ばれるものです。これは、特定のオブジェクト指向で有効な実装のルールやスタイルをパターンとして扱ったものです。C++でメモリ管理の煩わしさを避けるために、リファレンスカウンタを使うなどといった例があります。

また、こうしたスケールによる分類の他、対象とする問題領域の種類や、記述のスタンスの違いによって、パターンが分割される場合も見られます。

例えば「プロセスパターン」といったものがあります。これはソフトウェア自体というよりは、マネージメント的な視点から、ソフトウェアの開発プロセスの有効なパターンを指摘しているものです。(スパイラル開発、プロトタイピングなどがパターンとして示されています)。
「CORBAデザインパターン」は分散オブジェクト技術を利用した場合に有効なパターンについて述べてあり、「アンチパターン」は適用するとまずいことになるパターンとそこからの脱出策を反面教師的に述べています。

このようにパターンをとりまく動きは、非常に活発化しており、現在多様化の傾向にありま

す。

しかし、「設計者」という観点からみた場合には、まず皆さんが最初に本腰をあげて学ぶべきなのは、やはりいわゆる「デザインパターン」です。

「デザインパターン」という言葉をオブジェクト指向の技術者に広く知らしめたのは、Erich Gamma, Richard Helm, Ralph Johnson, John Vilissides (通称Gang of Four-GoF)らによる"Design Patterns"です。(訳: 「デザインパターン」 ソフトバンク、ISBN 4-89052-797-4)。この本がOO設計者の間で一種のベストセラーとなり、パターン運動を活性化するきっかけになったのです。出版は1995年ですが、現在でも、その実用性、有効性は変わりません。この本ではミクロアーキテクチャレベルでの「デザインパターン」について紹介しています。「生成」、「構造」、「振る舞い」という分類で23のパターンが理路整然とカタログ化されています。
まずはこの本に書かれた「デザインパターン」についてまずしっかりと把握することが、優れたオブジェクト指向技術者への重要な通過点となります。

ただし、元来「デザインパターン」は、コーディングも含めた日々のシステム開発の努力の中、ボトムアップ的に築きあげられていったものです。そのため単に机上のものとしてパターンのカタログを鵜呑みにするだけではありがたみもわからず、有効に活用できない危険性も出てきます。プロジェクト経験を積みながら、現在直面している問題と照らし合わせて繰り返し参照し、自分のものにしていくようにしましょう。
また、パターンは「どこまで覚えれば終わり」といったものではなく、システム開発がある限り、次々新たなものが発見され研鑽されていく過程を経ることになります。上級者の方には自らパターンを抽出し、パターンコミュニティに一石を投じていくという姿勢も望まれます。

オブジェクトの広場

［Happy Squeaking!!］

6. 事例:Observerパターン

ここでは「デザインパターン」の例として、Observerパターンを取り上げることにします。これはデザインパターンに関する多くの書籍で必ずといって紹介されるオーソドックスなパターンです。

パターンは通常「問題」「解決策」などの構成要素が明確になるように、テンプレートの穴埋め形式で記述されるのが普通です。ここでもそれに従ってみることとします。

問題

ある一つのオブジェクトの状態変化に対し、他の複数のオブジェクトが同時に依存している場合がある。例えばあるモデルのデータの最新情報を表示するビューは、モデルの状態変化に応じて再表示を行わなければならない。これら複数の異なるビューが、一つのモデルを参照している。
モデルの状態とビューの表示を一貫性のある形にするには、相互の協調が不可欠だが、モデルがあらかじめ多様に発展していくビューの全て知っており、それぞれに応じて再表示の要求を送るのは困難である。またモデルがビューに直接更新を要求すると、モデルとしてのクラスの独立性が失われてしまう。

制約

- ある一つのオブジェクトの状態変化を、複数のオブジェクトに通知しなければならない
- 状態変化を受け取るオブジェクトは多様なものになる可能性があり、状態変化を起こす側では、あらかじめそれを知ることができない
- あるタイミングで、状態変化を通知されていた側が、その通知を必要としなくなる場合がある

解決策

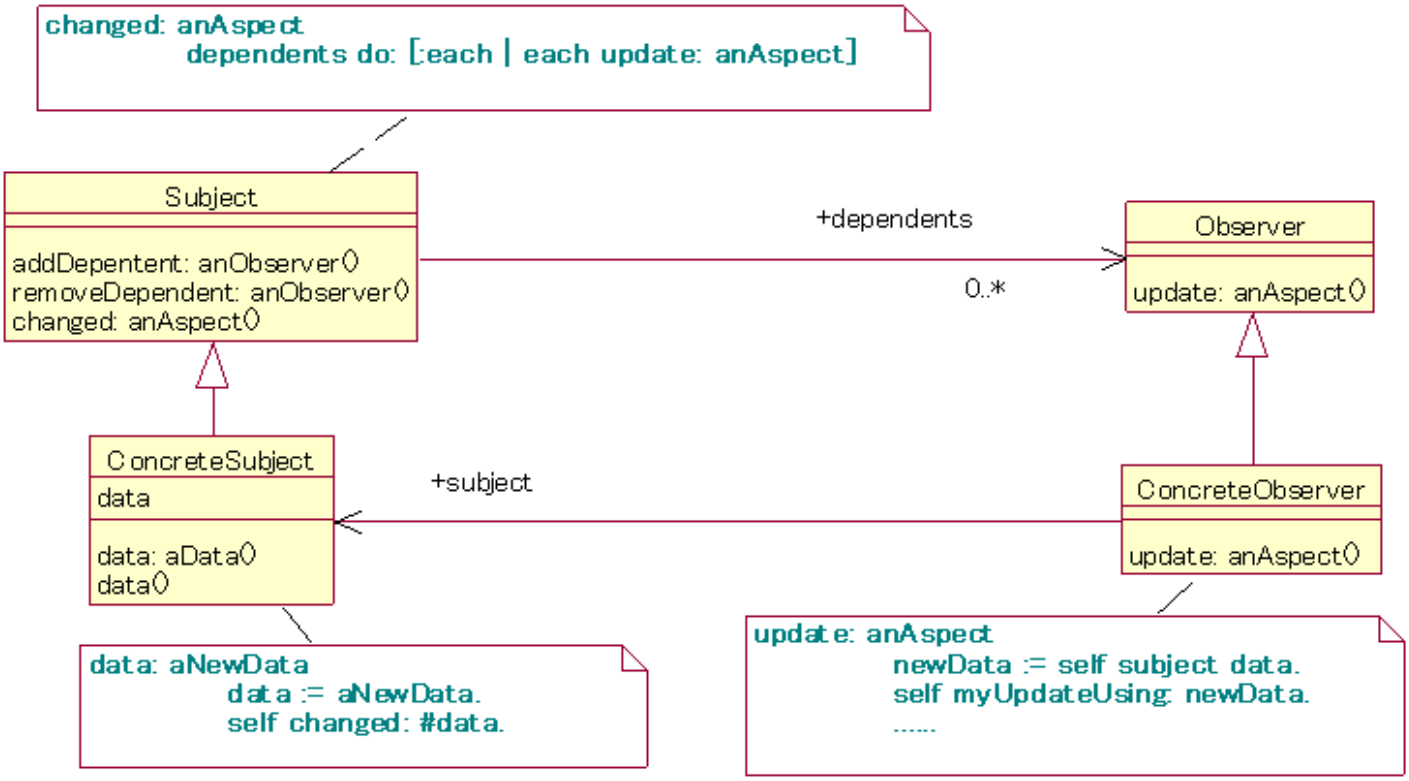
あるオブジェクトの状態変化を受け取る側は、変更を観察するもの、Observerとして抽象的にとらえられる。変更の起こる側は、Observerにとって関心のあるSubjectである。ObserverとSubjectといった抽象レベルで、Subjectに対するObserverの登録、また登録されたObserverに対する通知の仕組みを用意することで、相互の密な結合を防ぐ。

構造

以下のような構成要素から成り立つ

- Observerクラス
  - Subjectの状態変化を監視する
  - Subjectからの状態変化に対し通知受け取りのインターフェースを提供する
- Subjectクラス
  - Observerを登録、削除する機能を提供する
  - 登録されたObserverに対して状態変化を通知するインターフェースを提供する
- ConcreteObserverクラス
  - ConcreteSubjectを参照している
  - ConcreteSubjectからの状態変化に対し、通知を受け取り、自身を更新する具体的な処理を実装する
- ConcreteSubjectクラス
  - ConcreteObserverにとって有益な情報を保持する
  - 状態変化の際にSubjectで規定された、変更通知操作を呼び出す

以下のようなクラス図となる。



振る舞い(協調関係)

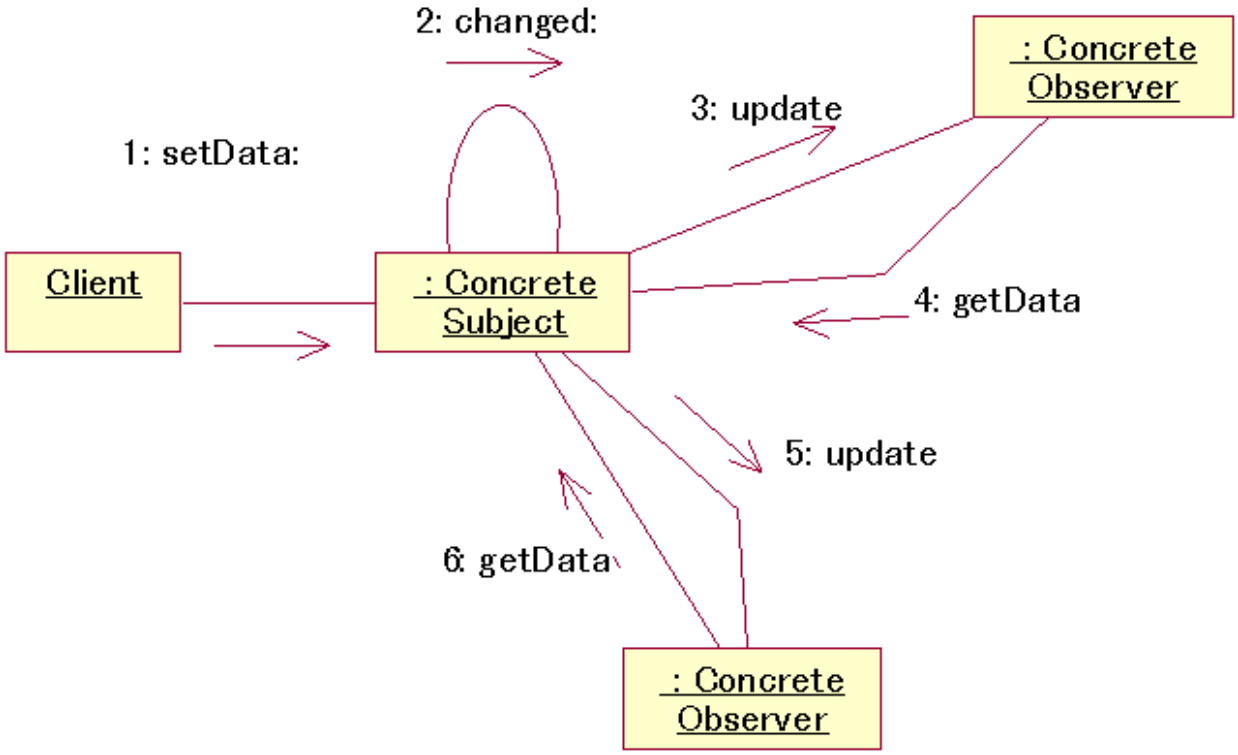
登録

1. 関心のあるConcreteSubjectに対してConcreteObserverの登録(addObserver:)が行われる
2. ConcreteSubjectは自身のリスト(dependents)にConcreteSubjectを追加する

通知

1. ConcreteSubjectの状態が、更新操作の起動により変化する。
2. ConcreteSubjectは自身の状態変化に伴い、通知用の操作(changed:)を起動する
3. ConcreteSubjectに登録した全てのConcreteObserverの通知受け取り操作(update:)が起動される
4. 通知受け取り操作でConcreteObserverは最新ConcreteSubjectの最新状態を取得し適切な処理を行う

以下のようなコラボレーション図となる。



効果、副作用

- ObserverのSubjectへの登録、登録されたObserverへのSubjectからの通知が、抽象レベルで定義されており、ConcreteSubjectのConcreteObserverに対する結合の度合いが低くなる
- Subjectに対するObserverの登録により、同時に複数のObserverに対し変更の通知が可能になる
- Subjectに対する変更に関心がなくなったObserverは、登録から削除することによって変更の通知を受けなくすることができる
- 登録された全てのObserverに対して通知されるため、Subjectの状態の変更が非常に多い場合には、Observer側の更新受け取り動作が必要以上に起動される可能性がある
- Subjectの状態変化に対してObserverが最新のSubjectの最新状態を取得するが、Subjectのどの属性についての変更かを示す関する情報がないと、全ての属性の値についてObserver側が取得せねばならず、全体の効率を落とすことになる。

別名

Publisher-Subscriber
　(Subject、Observerにそれぞれ該当する。定期購読申込者に最新の雑誌が届く連想から)
Dependents
　(ObserverはSubjectに対し、通常の関連に比べ結合度の弱い依存関係を持つため)

パターンテンプレートを使った書き方は、慣れていないと読みにくいかもしれません。しかし、きちんとしたドキュメントとしてパターンを残していくという点では有効な方法といえるでしょう。

Prev. Index Next

オブジェクトの広場

[Happy Squeaking!!]

7. Squeak演習:Observerパターン

7．1　Dependencyメカニズム

パターンは、それが扱う問題のスケールや種類によっていくつかに分類することができます。

Smalltalkの世界では、Observerがパターンとして認識されるよりもはるか昔から、Observerの仕組みを実現していました。
1983年の"Smalltalk-80: The Language and Its Implementation"においてオブジェクト間の3種類の関係として以下のような定義がされています。

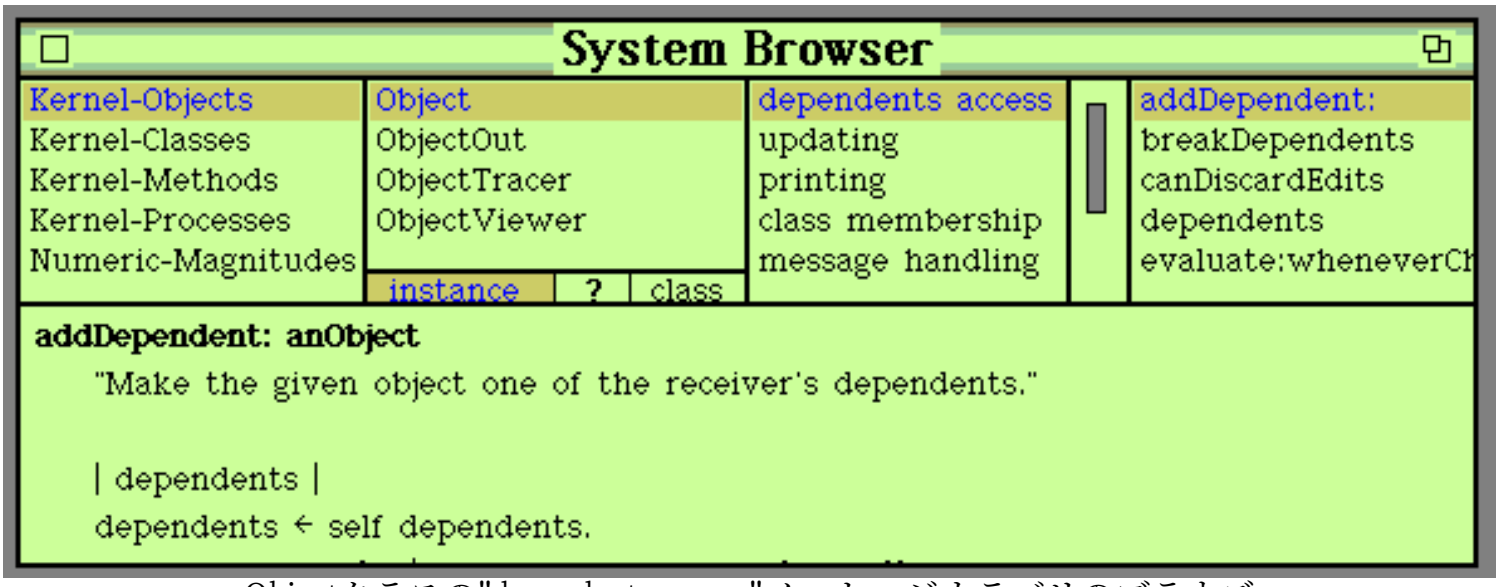
1. インスタンス間の関係(インスタンス変数による通常の参照)
2. クラス間の関係(クラスからスーパークラス、クラスからメタクラスへの関係)
3. dependency(依存)関係(他のオブジェクトに自らを依存物として登録する関係)

3番目の関係がまさにObserverがSubjectを監視する関係を表しています。
異なるオブジェクト間での協調動作をスマートに行うやり方としてdependencyは考え出されたのです。
dependencyは主にMVCフレームワークにおいて、MからVに対して間接的に変更通知を行う際に使われていますが、適用範囲はなにもMVCに限る必要はありません。
dependencyメカニズムの基本実装は、Objectクラスで定義されています。これは、全てのオブジェクトが、ObserverにもSubjectにもなれるということを示しています。(Modelクラスでは、より効率的な実装としてdependencyメカニズムを提供しています)。

SmallTip: Smalltalkでは、Observerパターンの実装としてdependencyメカニズムが提供されている

ここでは基本的なObserverの実装として、MVCとは独立させて、dependencyの仕組みを試してみることにしましょう。

まずはObjectクラスをブラウズしてみます。



Objectクラスの"dependents access"メッセージカテゴリのブラウズ

まず、ObjectをSubjectとして考えましょう。
SubjectはObserverを登録、削除できなければなりません。
従って、メッセージカテゴリ "dependents access"には、Dependent(Observer)の登録、削除に関する操作が格納されています。(addDependtdent: 、removeDependent: など)

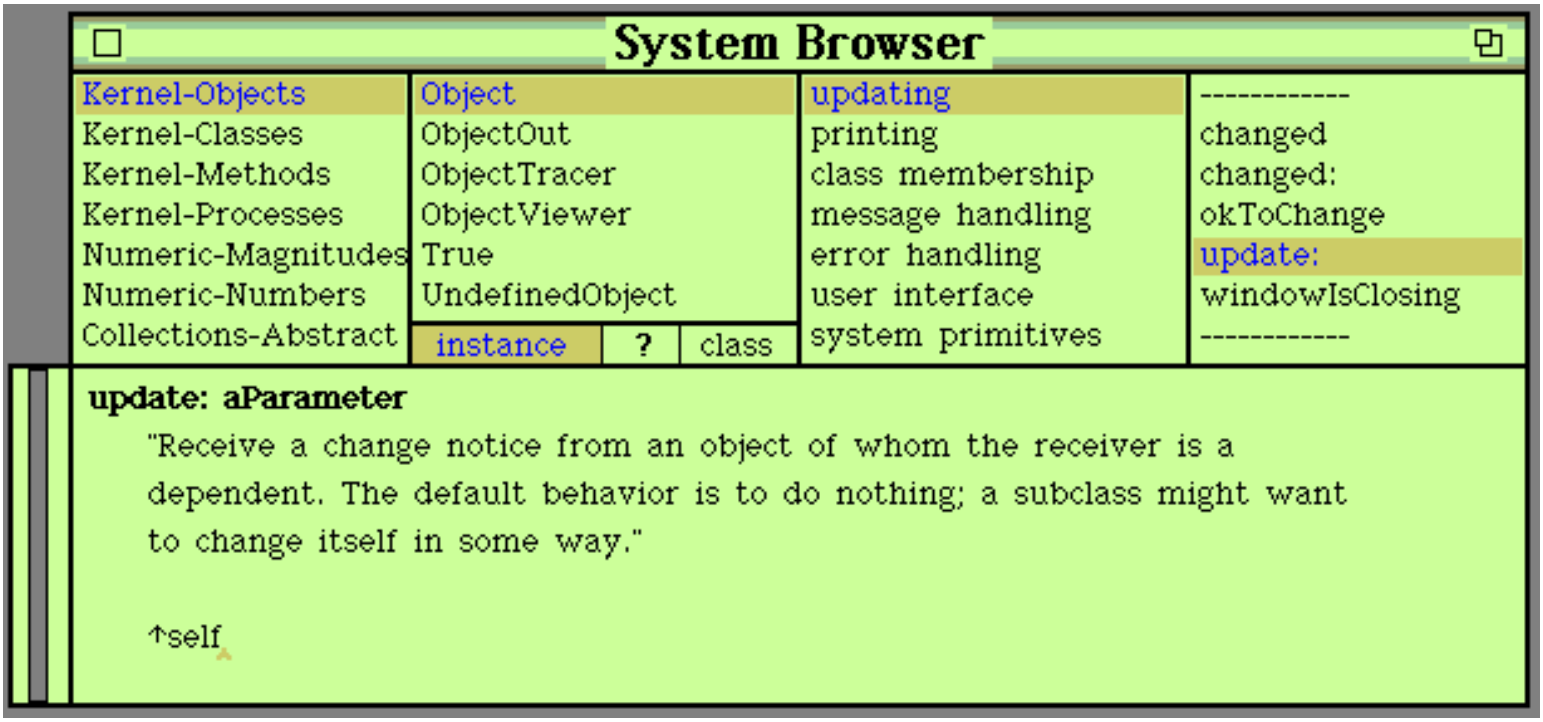
また、Subjectは、自分の状態変更を通知できる必要があります。
メッセージカテゴリ updatingには、通知用のメソッド changed:を見ることができます。

(Object >> updatingカテゴリ)

```
changed: aParameter
        "…コメントは省略..."
        self dependents do: [:aDependent | aDependent update: aParameter]
```

依存物リスト(dependents)に登録された全てのDependent(Observer)に対してupdate: のメッセージを送るように実装されています。

次にObjectをObserverとして考えましょう。
Observerは、Subjectの変更通知を受け取る操作を持っていなければなりません。これが、changed内で送られていたupdate:になります。



Objectクラスの"updating"メッセージカテゴリのブラウズ

デフォルトではupdate:の中身は空になっています。Dependent(Observer)は、update: をオーバイライドして、適切な処理を実行します。

更新受け取り操作のupdate: は引数をとります。引数は何でもいいのですが、通常は、Subjectのどの属性が変化したのかを示す情報を渡します.
(Smalltalkの世界ではこれをAspect-側面と呼びます)。

SmallTip: 変更通知はchanged:メッセージを自身に送る

SmallTip: 変更の受け取りはupdate:メッセージ

オブジェクトの広場

[Happy Squeaking!!]

7. Squeak演習:Observerパターン

7．2　Dependencyサンプル:CoffeePotとTemperatureAlarm

それではDependencyメカニズムを利用した簡単なサンプルを作ってみましょう。

まずは、Subject側としてCoffeePotなるものを定義します。属性としてquantity(量)、temperature(温度)を持つものとします。

以下のようなクラス定義となります。

```
Object subclass: #CoffeePot
        instanceVariableNames: 'temperature quantity '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Dependency tutorial'
```

依存物リスト(dependents)に登録された全てのDependent(Observer)に対してupdate: のメッセージを送るように実装されています。

次にアクセス用の操作を定義していきます。
状態変化のあった場合に、最後にchanged:を送るところに注目してください。

(CoffeePot >> accessingカテゴリ)

```
quantity
        ^quantity
quantity: anInteger
        quantity := anInteger.
        self changed: #quantity.
temperature
        ^temperature
temperature: anInteger
        temperature := anInteger.
        self changed: #temperature.
```

以上でCoffeePotの実装は終わりです。

次に、Dependent(Observer)側として、TemperatureAlarmを定義します。

```
Object subclass: #TemperatureAlarm
```

```
        instanceVariableNames: 'target '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Dependency tutorial'
```

targetは温度を測る対象です。

アクセス用として以下を定義します。

(TemperatureAlarm >> accessingカテゴリ)

```
target
        ^target
target: anObject
        target := anObject
```

次は更新受け取り用の操作を定義します。

(TemperatureAlarm >> accessingカテゴリ)

```
update: anAspect
        anAspect == #temperature
                ifTrue:
                        [Transcript cr; show: 'Temperature is '.
                        Transcript show: self target temperature printString].
```

TemperatureAlarmは温度の変更にのみ関心があります。そこで引数anAspectとして渡されるシンボルが#temperatureの場合にのみ、トランスクリプトに温度を表示するように実装しています。

今までのソースです。

FileIn: DependencyTutorial.st (<=Click)

ではサンプルを実際に動かしてみましょう。

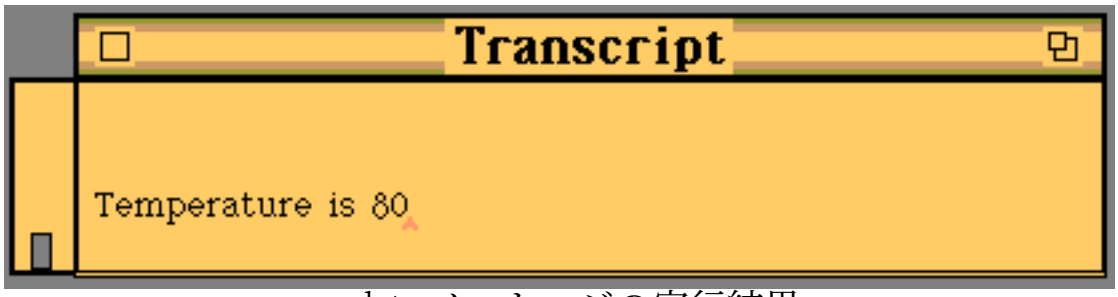
ワークスペースで以下を実行します。

```
| pot alarm |
pot := CoffeePot new.
alarm := TemperatureAlarm new.
alarm target: pot.
pot addDependent: alarm.
pot quantity: 100.
pot temperature: 80.
```

addDependent:によって、alarmをpotのObserver(Dependent)として登録しています。

変更通知メッセージ(changed:)がquantity:、temperature:の内部で共に起動されます。この際に引数anAspectにどの属性が変化したかの情報がシンボル文字列として渡されています。結果として、変更受け取り(update:)メッセージを受け取ったalarmは、温度が変化した場合にのみ反応することになります。

Transcriptには以下のように表示されます。



update:メッセージの実行結果

具象レベルではCoffeePot(Subject)はTemperatureAlarm(Observer)のことを一切知らず、抽象的な変更通知(changed:)によって変更が伝播されていくという点が重要です。

Prev. Index Next

オブジェクトの広場

[Happy Squeaking!!]

8. Squeak演習:マルチビューの実現

ここではSqueakを使い、複数のビュー(Observer)が一つのモデル(Subject)を参照するアプリケーションを作成することにします。

まず先ほど作成したMVCアプリケーションのCounterに、別のViewを定義し、同一モデルに対してマルチビューが提供できるように拡張します。
Observerパターンが使われていることよって、多様なビューが新たに追加されていったとしても、オリジナルのモデル部分のコードにまったく変更を加えずに済んでいることを確認して下さい。

また、Observerパターンの利点として、複数のObserverに対して、同時に変更通知を送れる(ブロードキャスト)ということがあります。モデルを参照するビューが同時に複数存在しても、やはりモデル側はまったく影響を受けないのです。

オブジェクトの広場

[Happy Squeaking!!]

8. Squeak演習:マルチビューの実現

8．1　新たなCounterビューの作成

CounterTextViewは、ボールド体文字列でCounterの値を表示しただけでしたが、今回は新たなViewであるCounterLineView を定義し、線の位置によって値を確認できるようにします。

まずは、CounterLineViewというクラスを定義します。

```
StandardSystemView subclass: #CounterLineView
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'MVC tutorial'
```

表示用としてdisplayView操作を定義します。

(CounterLineView >> displayingカテゴリ)

```
displayView
        | count box max min line |
        super displayView.
        count := self model data.
        box := self insetDisplayBox.
        max := box width.
        min := 0.
        (count > max) ifTrue: [count := max].
        (count < min) ifTrue:[count := min].
        self clearInside.
        line := Line new.
        line beginPoint: (box topLeft + (Point x: count y:0)).
        line endPoint: (box bottomLeft + (Point x: count y:0)).
        line displayOn: Display.
```

Squeakのグラフィックライブラリの詳細についての解説ではないので、詳細については説明を避けます。基本的には、DisplayTextのかわりにLine(DisplayObjectのサブクラス)を使い、モデルから取得した現在のカウンタ値をx座標として直線を引いているだけです。

次に更新受け取りメッセージを作成します。

(CounterLineView >> updatingカテゴリ)

```
update: anAspect
```

```
        anAspect == #data
                ifTrue:
                        [Transcript cr; show: 'now update'.
                        self displayView].
```

前の演習でアスペクトの使い方を習ったので、引数anAspectの指定により更新のフィルタリングを行うようにしています。しかし基本的には、CounterTextViewのupdate: と同様、ビューの再表示用にdisplayViewを呼び出すだけです。

最後はビューが使うデフォルトのコントローラを示すメソッドです。

これはCounterTextViewと同じように実装します。

(CounterLineView >> control accessカテゴリ)

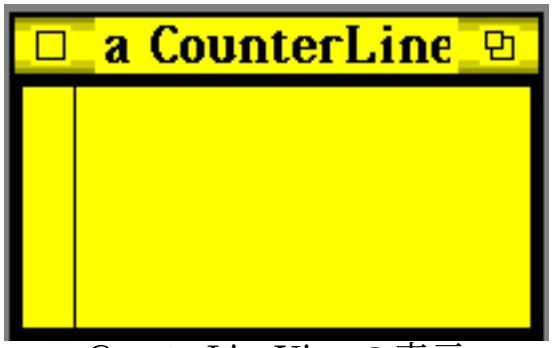
```
defaultControllerClass
        ^CounterKeyboardController
```

では、ひとまずシングルビューで、新しく作られたCounterLineViewの見栄えを確認してみましょう。

ワークスペースで以下の式を"*do it*"します。

```
| counterView |
        counterView := CounterLineView new.
        counterView model: (Counter new).
        counterView label: counterView  name.
        counterView maximumSize: 200 @ 100.
        counterView minimumSize: 50 @ 30.
        counterView borderWidth: 5.
        counterView backgroundColor: Color yellow.
        counterView controller open.
```

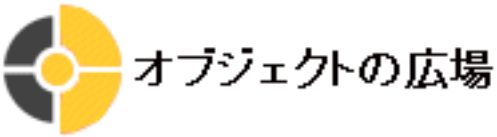
キーボードのuとdで操作しましょう。線の表示位置が変化していきます。



CounterLineViewの表示

今までのソースです。

FileIn: CounterLineView.st (<=Click)



[Happy Squeaking!!]

8. Squeak演習:マルチビューの実現

8．2　CounterLauncherの作成

マルチビューとしてCounterアプリケーションを起動するにあたって便利なユーティリティとなるクラスを作成しましょう。これでワークスペースにビュー起動用のメッセージ群をいちいち書かずに済むようになります。

以下のようにクラス定義を行います。

```
Object subclass: #CounterLauncher
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'MVC tutorial'
```

CounterLauncherは、ユーティリティクラスとして、簡単に使えることを考えています。そこでインスタンスをいちいち生成しなくともCounterLauncherが使えるように、クラス操作やクラス属性を追加していくことにします。

まず、クラスインスタンス変数として、counterを定義します。ここにはビューが同時に参照するモデルとしてCounterのインスタンスが入ります。
ブラウザを"*class*"側に合わせて以下のようにテンプレートに記入し、"accept"します。

```
CounterLauncher class
        instanceVariableNames: 'counter '
```

次に操作を追加します。(これらもすべて"*class*"側で行います)

まずは定義したcounterの初期化操作です。

(CounterLauncher class >> class initializationカテゴリ)

```
initialize
        counter := Counter new
```

アクセス用の操作も定義しましょう。

次に更新受け取りメッセージを作成します。

(CounterLauncher class >> accessingカテゴリ)

```
counter
        ^counter
```

```
counter: aCounter
        counter := aCounter
```

最後に起動用の操作を定義します。引数としてビューのインスタンスを受け取り、同一モデル(counter)を参照させて起動するようにします。

(CounterLauncher class >> actionsカテゴリ)

```
launchBy: aCounterView
        aCounterView model: self counter.
        aCounterView label: aCounterView  name.
        aCounterView maximumSize: 200 @ 100.
        aCounterView minimumSize: 50 @ 30.
        aCounterView borderWidth: 5.
        aCounterView backgroundColor: Color yellow.
        aCounterView controller open.
```

これで完成です。

FileIn: CounterLauncher.st (<=Click)

オブジェクトの広場

[Happy Squeaking!!]

8. Squeak演習:マルチビューの実現

8．3　マルチビューCounterの起動

それでは、マルチビューでCounterアプリケーションを立ち上げてみましょう。

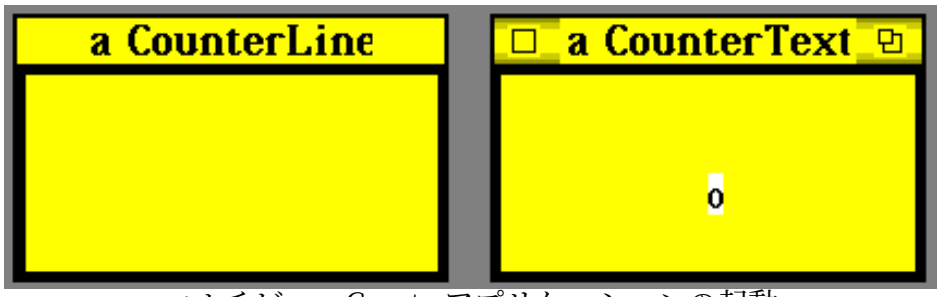
まずはCounterLauncherのクラスインスタンス変数counterをセットします。
ワークスペースで*CounterLauncher initiallize*と"do it"して下さい。

CounterLauncherにlaunchBy:で異なるビューを与えると、同一Counterモデルを参照するビューが複数立ち上がります。

一行ずつ"do it"して下さい。

```
CounterLauncher launchBy: (CounterLineView new).
CounterLauncher launchBy: (CounterTextView new).
```

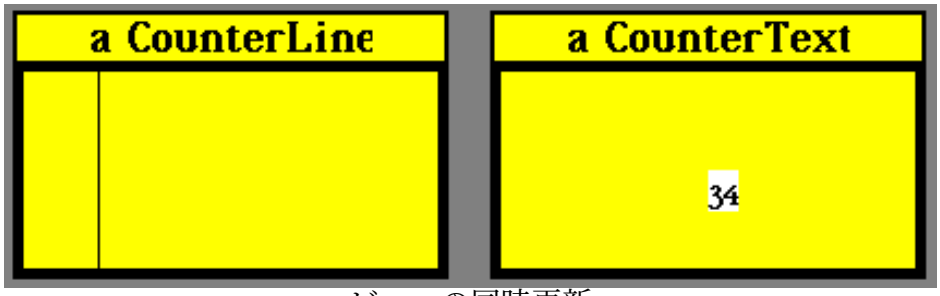
マルチビューでCounterアプリケーションが立ち上がります。



マルチビューCounterアプリケーションの起動

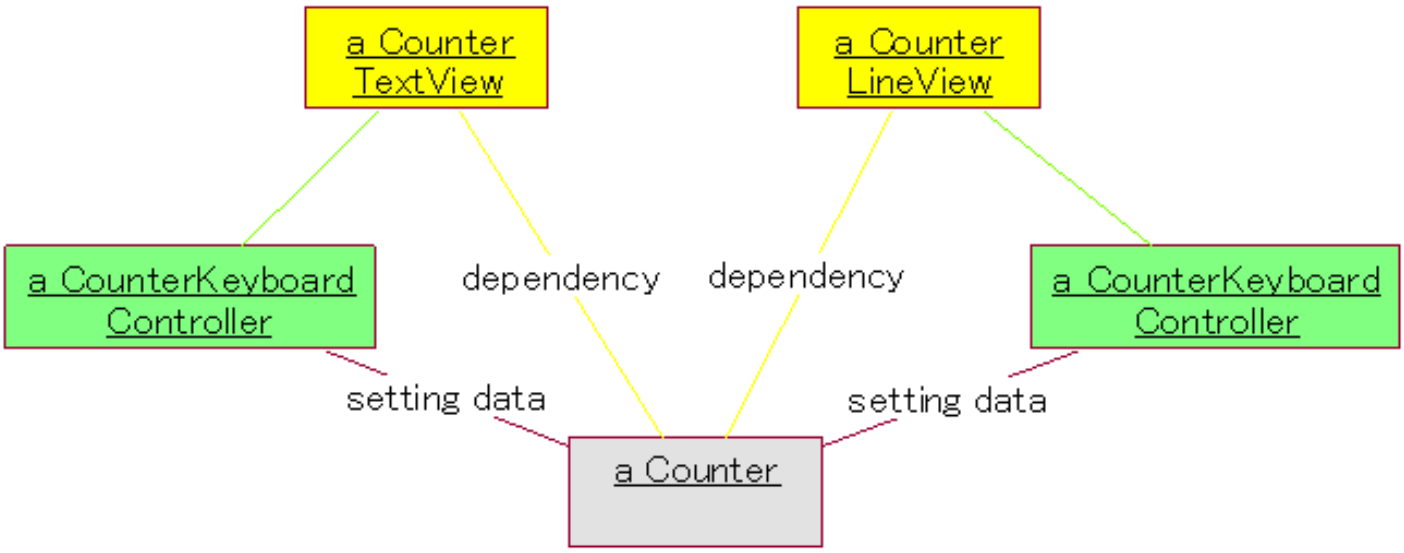
どちらのビューでもかまいませんので、カーソルを合わせてu、dのキーボード操作を行ってみてください。

ビューが同時に更新されます。



ビューの同時更新

念のためオブジェクト図で動作イメージを示します。



マルチビューCounterアプリケーションのオブジェクト図

マルチビューなので、二つと言わず、更にビューの数を増やしたり減らしたりして楽しんでみてください。

###

今回もなかなかのボリュームでした。しかし最後まで読み通したあなたは、デザインパターンについて基本をしっかり把握し、一歩先ゆくOO技術者を目指す足がかりを得たのです。次回は、デザインパターンのPart2です。今後のトレンド間違いなしのリファクタリングというトピックを扱います。ご期待ください。

Prev. Index Next

オブジェクトの広場

[Happy Squeaking!!]

9. 参考文献

●和書

「Smalltalkイディオム」 ソフトリサーチセンター 1997

日本のSmalltalkerとしては第一人者である青木淳さんの書き下ろし。スレッドや例外処理などアドバンスな内容を含む。

「デザインパターン」 ソフトバンク 1995

GoFの"DesignPatterns"の訳。デザインパターンを学ぶにははずせない本。

「デザインパターンプログラミング」 トッパン 1998

フレームワーク構築のためのパターン、「メタパターン」を紹介している。HotSpot、FrozenSpotの解説は非常に重要。

「ソフトウェアアーキテクチャ」 トッパン 1999

"Pattern-Oriented Software Architecture"の訳。通称POSA本と呼ばれるデザインパターンについての一種のバイブル。MVCアーキテクチャについて詳しい解説がある。

● 洋書

"Smalltalk, Objects, and Design" Prentice Hall 1996

Smalltalkを学びながら、オブジェクト指向、デザインパターンについて学 んでいく。本講座のスタンスに最も近い。入門書でありながら洞察は深い。

"DesignPatterns Smalltalk Companion" Addison Wesley 1998

オリジナルのGoF デザインパターン本をSmalltlkerからの観点から再考察 した本。GoFのDPとSmalltalkとの関係をより深く知ることができる。

● Web

Patterns Home Page

パターンコミュニティの総本山、Hillsideグループのページ。パターン全般についての詳しい情報が得られる。
http://hillside.net/patterns/

JPLoP設立準備委員会

日本のパターンコミュニティ。日本人によるパターンリポジトリが参照できる。
http://www.kame-net.com/jplop/Default.htm

Let's "do it"!!

Prev. Index