アスペクト指向開発適用記

技術部
ITスペシャリスト
友野 晶夫
Masao Tomono

システム開発では、画面遷移、認証、アクセス制限、データベースアクセス、トランザクション管理、データベースキャッシュといった、さまざまなフィーチャ（feature）に取り組む必要がある。一般に、これらのフィーチャをサブルーチンやオブジェクトなどのモジュール化技法のみで適切なモジュール化を行うことは極めて困難であり、総じてフィーチャへの小さな変更でさえ、アプリケーションプログラムの広範囲にわたり影響が及ぶ。

アスペクト指向開発（AOSD:Aspect Oriented Software Development）は、こうしたフィーチャを効果的にモジュール化する技術として非常に有望であり、また実践的である。

本稿では、Webアプリケーションの構築にAOSDを実適用した際の成功と失敗の事例をケーススタディとして示し、アスペクト指向設計とも呼ぶべき観点から「高度な関心の分離（ASoC: Advanced Separation of Concern）」の重要性と、アスペクト指向開発の可能性を具体的に示す。また、アスペクト指向開発の取りうる形態と従来技法との差異にも言及する。

1. 用語定義

(1) アスペクト指向プログラミング[6)]

システムの広域に影響を及ぼす（crosscutting）関心を分離する技法。実現するための言語系としてはAspectJが広く知られている。

(2) 高度な関心の分離

ASoCと略称されることが多い。「関心の分離（SoC）」という規範自体はParnas[4)] やDijkstra[5)] に遡るが、その拡張となる「高度な関心の分離」では、多重の、しかも輻湊した関心の取り扱いが主題とされる。

(3) コンテキスト

対話の背景をなすもののこと。本文では、Webアプリケーションの例が取り上げられるが、javax.servlet.ServletContextのことではないので注意。

(4) パターン

くり返し発生する一定の出来事、ないしそれを生成する空間的形態のこと。そうした空間的形態を生み出すパターンもあり、再帰的に存在する。

(5) パターンライティング

パターンを記述すること。ソフトウェアに関するパターンコミュニティでは、パターンライティングのための方法が編み出されている。

2. AOSD概要

ここでは、構造化からオブジェクト指向を経てAOSDへと至る経緯を示すことで、AOSDが何を目的とするもので、どのようなものなのかを説明する。

構造化パラダイムは、構造に着目することの重要性を主張するものであった。この主張は、設計に大きな力を発揮し、オブジェクト指向、AOSDにおいても受け継がれている。しかし、構造化パラダイムにおいて焦点が当てられたのは、モジュールやデータといったプログラムの世界を記述する要素のなす構造に限定されていた。

後に現れたオブジェクト指向パラダイムは、実世界の「事物」の構造に着目することを主張した。これにより、要求や分析といった人間活動をモデリングすることにも同様の技法を用いることが可能になり、これら人間活動モデルとシステム設計との距離が縮まった。

ところで、「事物」というのは視点から切り離されて存在するものだが、これを正確にモデリングすることは実は困難である。たとえば、今、筆者の手元にある清涼飲料水の缶をモデリングするとしよう。この缶を運搬することを考えれば、その重さや容積が制約となるだろう。リサイクルすることを考えると、素材が何かを記述する必要がある。当然、この内容物を飲むのだが、その際に、どの程度のことを記述する必要があるだろう。単に飲んだかどうかの事実を把握しておきたいなら、「飲む」という操作があるとだけ記述すればよいだろう。一方で、ロボットのようなものにこれを飲ませることを考えるなら、開封し、手にもち、傾けるといった、飲むための方法を記述する必要がある。これでもすべてを網羅しているとはいえない。このように、視点から切り離されたものとしてモデリングすることは、事実上不可能なのである。したがって、オブジェクト指向モデリングは、何らかの視点を定めた上で行われることとなる。この視点の関心事をdominant concernと呼ぶ。

さて、分析時にはシステムを用いる人間のビジネスがdominant concernとなるが、設計時にさまざまなシステム実装上の関心事が現れることとなる。たとえば、後に事例で触れているように、各メソッドへのアクセス権限をユーザごとに割り振る場合、すべての公開されたメソッドの開始時にアクセス権のチェックを行わなければならない。

構造を与えるということは、同時に概念要素に切り分ける作業でもある。このアクセス制御の例では、dominant concernに基くモデリングにより、アクセス制御の関心事が散り散りに切り分けられてしまった。このような関心事をcross-cutting concernと呼ぶ。もちろん、cross-cutting concernは実装に関する関心事だけではない。ビジネスシステムでは、ビジネス上の関心事も複数あるのが通常であり、それらがcross-cutting concernとなる場合もある。

この点を是正することを目指しているのがAOSDである。現在、大きく分けて２つの流派がある。一つはアスペクト指向プログラミング（AOP:Aspect Oriented Programming）と呼ばれるものである。これは、dominant concernのプログラムにツールによってcross-cutting concernを「編みこむ(weave)」という技法である。編みこみは、リフレクション(自己反映計算)の応用であり、言語セマンティクスにcross-cutting concernを埋め込むことで、dominant concernと分離する。現時点では、こちらの流派が主流である。

もう一つは、MDSoC[7)]（Multi-Dimensional Separation of Concerns）と呼ばれるもので、オブジェクトを各関心

事に分離して捉えるものである。

ただし、AOSDは、目的を完全に達成するには至っていない。AOPは、いわばdominant concernとcross-cutting concernの織り成す現象に直接対処する方法であるため、dominant concernとcross-cutting concernの立場が固定的でになるので、cross-cutting concern側の領域が肥大化した場合の対処が難しい。特に、複数のcross-cutting concernを編みこむ場合には、どのような順序でcross-cutting concernを編みこむべきかを判定できない（特に、安全でない順序があることも分かっている！）という、リフレクションに関して古くから未解決なまま残っている問題に直面する。また、MDSoCでは、すべての関心事を平等なものとして捉える点でAOPよりも本質的なアプローチといえる。しかし、関心事間の結合をオブジェクト識別子で結び付けているため、視点によって概念の粒度が異なる場合（ビジネス領域の関心事にはこうしたものがよく見られる）に対処できない。

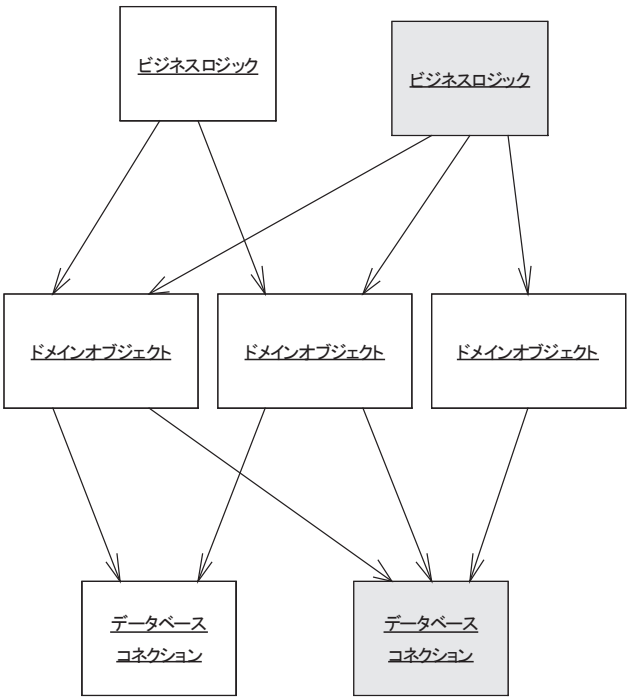

図１データベースコネクション

3. ケーススタディ

AOSDのケーススタディとして５つの事例を示し、成功へ導いた要因と、その過程で遭遇した失敗要因を述べる。

当該Webアプリケーションシステムは、バックエンドにデータベースが配置されており、そこにユーザの入力情報が蓄積されている。この蓄積されている情報を閲覧する機能がある。ただし、それはユーザ権限により入力情報の開示範囲が限定される。

3.1. 事例１－データベースアクセスとトランザクション境界

あるシステムでは、ドメインオブジェクトの各メソッドにてデータベースアクセスを行うべく設計されていた。そのため、各ドメインオブジェクトにデータベースコネクションを与える必要があった。

【問題】：　データベースアクセスの整合性を保つため、各メソッドで使用されるデータベースコネクションは、トランザクション境界内で同一のものとなるようにしなければならない。これを図１に示す。トランザクション境界は、必ずしもドメインオブジェクトの各メソッドで完結する訳ではない。トランザクション境界として完結する一連の手続き(たとえばビジネスロジックのどれか)において、データベースコネクションを取得し、それをドメインオブジェクトへの各メソッドコールに引数として渡せばよいのだが、そうすると、大半のドメインオブジェクトのメソッドにデータベースコネクションを受け取る引数が必要となり、これはビジネスオブジェクトの本来の意図からは明らかに逸脱した「関心」事項となる。さらに、必ずしも単一のビジネスロジック内でトランザクション境界が完結するとは限らないため、ビジネスロジックのインタフェース自体が不安定となる。

【解法】：　ドメインオブジェクトのメソッド引数からデータベースコネクションを取り除くことができれば、ビジネスロジックのプログラマにとっては、データベースアクセスについての関心を分離することができ、また、ビジネスロジックのインタフェースも安定させることができる。

ビジネスロジックからデータベースコネクションを引数として渡すことが必要となるのは、システム全体として複数のデータベースコネクションを扱う必要があるからだが、それは、複数のセッションが同時に実行されるためである。単一のセッションにおいて同時に扱うデータベースコネクションは、たかだか一つである。

当該システムでは、ビジネスロジックとドメインオブジェクトの間で行われる対話のコンテキストをセッションにより識別できる*。そこで、コンテキストからデータベースコネクションへのナビゲーションを可能とすることにし、また、コンテキストを図１の全領域にて指示可能とすることにより、引数を使わずに、ドメインオブジェクト内部にて、属しているコンテキストからそれに呼応するデータベースコネクションを取得させることとした。結果として、ビジネスロジックの部分およびインタフェースからデータベースアクセスに関する関心を明確に分離することができた。また、トランザクション管理も、同様の方法で分離して実現することができた。

3.2. 事例２－認証、アクセス制限

あるシステムでは、サーブレットへのアクセスを認証結果により制限する必要がある。

【問題】： 認証結果を保持する部分については、データベースアクセスと同様にコンテキストに保持させることとしたが、しかし、依然として全サーブレットのdoGet()およびdoPostの先頭にて、認証結果に当たるユーザが該当サーブレットにアクセス権を有するか否かの判定を行うコードを書く必要がある。これを図２に示す。

アクセス制御に関するコードは、どのサーブレットにとっても均質であり、変更する際にはすべてのサーブレットを変更しなければならない。

【解法】： 本システムでは、ブラウザからサーブレットへのアクセスにインターセプタ[1)]と呼ぶ単一のサーブレット[2)]を介在させ、このインターセプタにおいてアクセス制御を行い、アクセス可能であると判定された後に、実際のサーブレットにforwardさせた。これにより、サーブレットでの処理ロジックから、アクセス制御に関する関心を明確に分離することができた。

3.3. 事例３－画面遷移

本システムでは、アンカーリンクにて遷移先のページを指させていた。

【問題】： 画面遷移は各ページに散逸してしまい、画面遷移の変更は容易ではなかった。

【解法】： 各ページのアンカーリンクでは遷移のイベントを示すこととし、実際の画面遷移は、インターセプタにて状態遷移表に照らして決定するようにした。こうすることで、どこに遷移しなければならないかという関心を、表

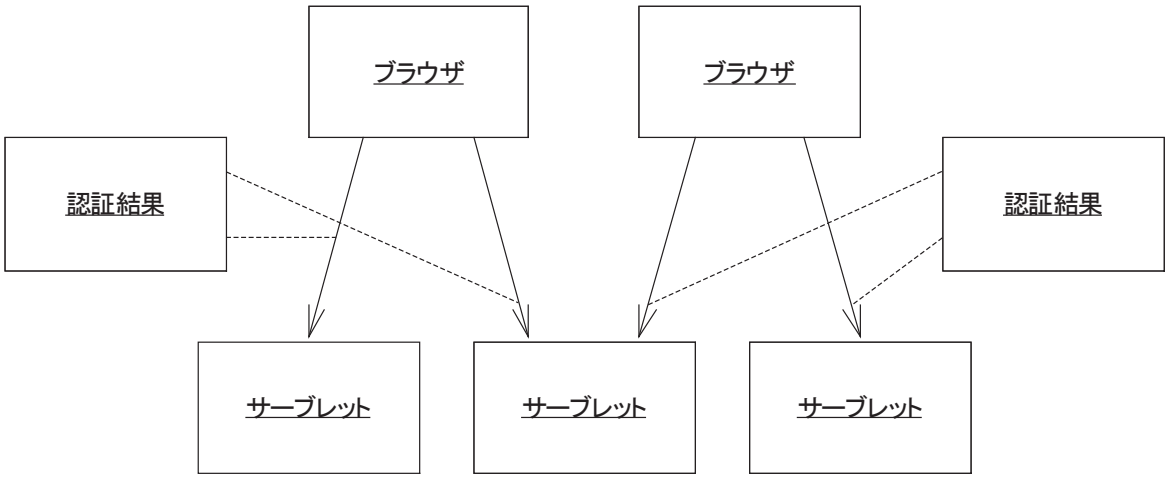

図２ アクセス制御

＊ コンテキストは、Sessionそのものではない。Sessionは、プログラム全域では扱えないため。また、他のプラットフォームに移行することを考えても、Session そのものを用いるのは得策ではない。

示コンテンツ（HTMLの記述）から明確に分離することができた。

3.4. 事例４. ーデータベースキャッシュ

事例１、２のシステムでは、さまざまなコンポーネントをコンテキストからナビゲーション可能とするため、Singleton の変種を用いていた。データベースキャッシュについても同様に、そのような方法ではじめは実装した。

【問題】： データベースキャッシュの方針は単一ではなく、システムで必要とされる隔離性水準とパフォーマンスのトレードオフにより方針の切り替えを余儀なくされる。

幸い、それらのインタフェースは変化しないので、それらのコンポーネントのSingletonを書き換えれば済むのだが、こうした変更を行うたびにコードを変更しなければならない。

【解法】： 本システムでは、コンフィギュレータとしてこれらのコンポーネントをコンテキストからナビゲートする部分を分離し、XMLスクリプトにより構成を指定できるようにした。これにより、各コンポーネントのコードからその構成についての関心を明確に分離することができ、また、構成を一元的に管理できるようになった。

また、画面遷移やアクセス制限も同様の方法で整理できた。

3.5. 事例５ークラスごとのキャッシュ方針変更

事例３では､すべてのクラスについて一様な構成を用いた。

【問題】： データベースアクセスのチューニングをより細かに行うことになり、データベースキャッシュの方針をドメインオブジェクトのクラスごとに設定する必要が生じた。コンフィギュレータに複数のデータベースキャッシュ方針を用意しておき、クラスごとに使うことを分けることでも回避できるが、どちらの方針を用いるかは、メモリ容量とのトレードオフもあるため、クラスによって必ず決まるわけではない。また、各クラスで用いるデータベースキャッシュ方針を切り替えるには、各ドメインオブジェクトからコンフィギュレータにアクセスする部分を変更しなければならない。

【解法】： コンフィギュレータにて、希望する機能（フィーチャ）とコンポーネントのマッピングを管理し、コンテキストごとに切り替え可能とした。また、AspectJを用い、コンフィギュレータにアクセスする部分で、マッピングを切り替えるようにした。

これにより、システムの既存コードには一切、影響を与えることなく、局所的にデータベースキャッシュ方針を切り替えることが可能になった。

4. 開発プロセス

これら事例に示した関心の分離は、リファクタリング[3)]プロセスを通じて得られたものであり、当初から想定された訳ではない*。

リファクタリングプロセスは、大まかに次のようであった。

(1) 対処する状況、フィーチャを追加する。
　まずは、関心の分離は考えない。この段階では、一般的な細かいレベルのリファクタリングを行っている。
(2) 状況により不安定でメンテナンスしにくそうなコードフラグメントに注目。
　大量に現れる同一のコードフラグメントもその一つ。
(3) 解決策を、パターンフォーマットで表現してみる。
　このとき、実際には、いくつかの解決策を考え、フォースの分析を通じて適した解を選択している。
(4) 解決策のコーディングを行う。
　適用してみて、見逃していたフォースに気付くこともあるので、積極的にtrialを行う。同時に、コンポーネントのプログラミングマニュアルを記述する。

＊ 例外的に､画面遷移についてだけは､当初から状態遷移によって表現するというアプローチを取った。画面遷移は、本質的に状態遷移で記述できることが予め分かっていたからである。当初は、インターセプタというアイデアがなかったので、HTMLとしてブラウザへ返す前に、JSPにて、ありうる状態遷移を評価していた。この方法よりは、事例に示した方法の方がリンクの記述をHTMLで完結できるため、関心の分離はより高度に行なえる。

これがユースケースの働きをする。

(5) パターンから既知の構造が見出せた場合、それを新たな言語系として構築する。

たとえば、コンフィギュレータを構築した事例では、以前に取り組んだBOM (Bill Of Material) システムの問題と等価であることに気付き、新たな言語系を作成できた。

(6) 元の言語セマンティクスとして捉えられる部分は、AspectJにて隠蔽する。

一方で、失敗要因としては、次のようなものがあった。どちらも「関心の分離」に注意を払うこと無く、安易に特定の解法を前提としてアプローチし、失敗した。

(1) 言語セマンティクスへの埋め込みを想定する。

簡易OODBを構築する際、言語セマンティクスに永続化のロジックを埋め込むためのプリコンパイラからアプローチしてしまった。永続化領域を簡単なもので済ませるという予断からそのようなアプローチを取ったのだが、マッピング方法を言語プリミティブごとに一貫したものとするのが難しく、結局、断念。その後、永続化領域を中心にアプローチし、非常に簡単なルールをオブジェクトの記述時に課すことで成功した。プリコンパイラも不要なほど簡単なルールだったので、最終的にプリコンパイラは作成しなかったが、実際に作るとしてもそれほど難しくはない。

(2) jAspectJによる解決を想定する。

認証とアクセス制御について、最初からAspectJで解決するものとしてアプローチし、失敗した。このとき、当初はJSPからアクセスするオブジェクトのレイヤを決めておき、そのレイヤに属するpublicなメソッドにてACLにアクセスさせることを想定していた。しかし、パフォーマンスが求められたとき、JSPへのアクセスの段階でアクセス制御を行うことが必要となった。しかし、JSP自身がJavaのランタイムプリコンパイラであるため、AspectJを使用することができなかった。

5. 考察

5.1. アスペクト指向開発の目指すもの

事例に見られるように、AOSDは必ずしもアスペクト指向プログラム言語の利用を前提とする必要はなく、使用しなくとも広範囲に関与する関心事(crosscutting concern)をモジュールとして分離するという、当初の「目的」を達成することができた。

サブルーチンやオブジェクトといったモジュール化技法による関心の分離は、関数コールとプログラム実体による分離から来ている。これと、今回取った方法を照らし合わせると、関心あるいはアスペクトとして明示的に抽出された「コンテキスト」の使用が大きな違いである。プログラム実体同士がランタイムにおいて対話を形成していると捉えたとき、関数コールは対話における「発話と受話」にあたる。しかし、「発話と受話」以外にもコンテキストは対話の構成要素として重要である。

たとえば、事例１に挙げた初期の解決法のまずかった点は、データベースコネクションのような対話のコンテキストに属するようなものを「発話と受話(つまり関数コール)」に混入した点に起因していると考えられる。それにより、「発話と受話」とコンテキスト間の関心の分離が妨げられたのである。

事例２においては、さらに、この「発話と受話」とコンテキストの分離を、インターセプタを用いて積極的に行っており、これはアスペクト指向アーキテクチャとでも呼べる、関心の分離を積極的に支援するアーキテクチャパターンとして用いられている。これは、インターセプタにより、「発話と受話」が立脚するコンテキストを構成する言語セマンティクス（HTTPによるサーブレットの起動）を変容させている、と捉えることができる。

事例３、４では、それぞれ、画面遷移をリンクにより表現していたものを、状態遷移という言語系に置き換えたり、Singletonによる構成記述というある種の言語系をコンフィギュレータによる構成記述という言語系に置き換えたりすることで、関心の分離を行っている。これも、「発話と受話」がおかれるコンテキストとしての言語系を変容させることで、「発話と受話」から関心を分離しているといえる。これにより、事例３では、「発話と受話」の内容自体から、どこに遷移するかという情報を取り除くことが可能となっており、事例４では、どのクラスに生成させるかという情報が「発話と受話」の内容自体から取り除かれている。

事例４、５では、コンポーネントの構成が結果的にシステムとデータベース間の対話のセマンティクスに影響を与

えることを見た。

ここまでの考察から、「発話と受話」を取り囲むコンテキストは、筆者が知る限りの範囲では、次のような次元軸で整理される。

(1) 束縛時

ランタイム、コンパイルタイム、配置時

(2) 言語レベル

コンポーネント構成、言語セマンティック、言語系

事例から、これらの次元軸で示されるどの領域であってもコンテキストと「発話と受話」の分離という視点は、crosscutting concernに対処する上で有効であり、一方、どの領域に解決策を求めるかは問題に依存していることが分かった。

5.2. アスペクト指向開発のプロセス

成功、失敗事例から、アスペクト指向開発を適用する上で重要なことは、従来よりもかなり慎重で徹底的な検討を要する点である、と主張できる。これは、アスペクト指向開発による関心の分離は、従来の「発話と受話」による関心の分離と比較して、分離後に関与する部分が各段に広く、分離の境界が変化した場合に影響範囲が広いことに起因している。事実、当初からこれらの分離を想定することは、一度でも同種の分離について経験がなければ難しいことと思われる。また､失敗要因の(2)にあるように、同種の分離について経験があっても、状況によってはうまく行かない場合も見られる。その点で、リファクタリングとパターンライティングによる問題の明確な把握と、それを取り巻くフォースの分析は有効であった。

本稿で提示したアスペクト指向開発の適用プロセスは、開発プロジェクトにおける贅沢品として見なされるかもしれない。しかし、実際には、これらの作業は、アプリケーションサイズを拡大する前に行っており、その後の開発をよりスムーズに行うために必須のことと筆者は考える。特にアーキテクチャ中心に開発を進める上では、言語系として完結したコンポーネントが得られる点で再利用の観点からも有利であると考えている。

6. 提言

6.1. アスペクト指向設計の追求

現在、「アスペクト指向」と名のつくもので、主に注目を集めているのは、プログラム言語としての記述方法の部分だが、より広く、設計方法として取り組むことが強く望まれる。

たとえば、既に、どのようにアスペクトを実現するか、あるいは関心の分離をアーキテクチャとして支援するかは、そのための技法として、インターセプタといった技法が確立しているが、どのようにアスペクトを配置すべきか、という問題については未だ明らかでないと筆者は認識している。

上記考察を傍証とし、コンテキストと「発話と受話」の分離およびコンテキストの構造という点に着目し、設計原理として再構成できるのではないか、と考えている。

6.2. 有効性と適用可能性

アスペクト指向開発の技法のうち、特に、言語セマンティクスを変容させるものについては、現在のところ無分別に使用した場合、従来の技法よりも深い混迷をもたらすものであることだけは分かっているが、適用可能性がどこまであるのかが明らかではない。今後とも、適用可能な領域（スコープ）についての蓄積が必要と考えられるが、その際に、現在行なわれているような個別のフィーチャについてではなく、より内包的にその適用可能性を示すことはできないだろうか。

たとえば、上記考察を傍証とし、適用可能領域としてコンテキストと「発話と受話」の分離に起因するのであれば、その分離をアスペクトとすることが、一般的に有効であるといえるのではないだろうか。また、コンテキストと「発話と受話」の分離以外の側面で言語セマンティクスの変容が有効である場合を、今後、知見として蓄積していくべきではないかと考えている。これが、アスペクト指向開発の発展に多いに寄与するであろう。

6.3. 「高度な関心の分離」のための道具としてのパターンライティング

アスペクト指向開発を適用するプロセスにおいて、パターンライティングのように、周辺との因果関係を明らかにする技法は必須のものと筆者は考える。また、その際にパターンランゲージとして、より「高度な関心の分離」へとエスカレート可能な構造を明らかにすることは、言語系と

してより完結したコンポーネントを構築する契機になるのではないかと考えている。事例で示したように、適切なAOSDがなされた際に得られる利得は非常に大きい。AOSDに就いての経験がパターンとして残されることは、この大きな可能性を安全に享受するためにも必須といえよう。

参考文献

1) Schmidt, D., et at. Pattern-Oriented Software Architecture Volume2, Wiley & Sons, Ltd, 2000.
2) Sun Microsystems, Inc., "Front Controller". http://java.sun.com/blueprints/patterns/j2ee_patterns/front_controller/index.html.
3) Fowler, M. Refactoring, Addison Wesley Longman, Inc., 1999.
4) D. L. Parnas. On the Criteria to be used in Decomposing Systems into Modules. In CACM. Vol. 15. No. 12. December 1972, pp. 1053-1058.
5) E.W. Dijkstra, A Discipline of Programming. Prentice Hall, Englewood Cliffs, NJ, 1976.
6) G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming. In proceedings of ECOOP '97, Springer-Verlag LNCS 1241. June 1997. http://aosd.net/
7) H. Ossher and P. Tarr, Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer, 2000. http://www.research.ibm.com/hyperspace/

＜問い合わせ先＞

技術部

Tel. 044-540-2141 友野 晶夫

E-mail : masao-tomono@exa-corp.co.jp