# Caterpillar's Fate : 分析から設計への変形についてのパターン言語

著 Norman L. Kerth
訳 井関知文、友野晶夫

## 概要

Caterpillar's Fate は、精密な分析ドキュメントから初期のソフトウェア設計への変形に対応するために使われるパターンランゲージです。変態の概念が手品のような蝶の羽化を説明するのに使われるように、Caterpillar's Fate は、オブジェクトとは無関係な分析から、オブジェクトのシステムが構築されるまでのタネあかしを探るものです。

## １．導入

Caterpillar's Fate というのは、ソフトウェア工学のアクティビティとしては風変わりな名前ですが、このパターンランゲージの役割をもっとも正確に表現しています。Caterpillar's Fate とは変態を指します。地面を這う生き物が空を舞うようになるという手品のような変形です。Caterpillar's Fate パターンランゲージは、システムの分析段階から設計段階への変形という、やはり手品のような変形を手引きするのに使われます。

過去の構造化技法は、分析から設計への変態に苛まれました。このようなパターンランゲージがなかったことが一因であると思います。筆者は、何人かのメソドロジストが分析から設計への移行方法を記述しようとするのを見てきました。初心者には、彼らが移行の際にしていることが行き当たりばったりのように見えるでしょう。実際には、メソドロジストはこの変形を支えるために、学んだ多くの知恵を使っているはずです。パターンランゲージのアプローチは、この問題に理想的な解決を与えてくれます。パターンランゲージは、学ばれた知恵を元に組み立てられているからです。

普及したオブジェクト指向方法論であっても、その多くは、この変形に関する問題を避けるために、設計の問題を分析フェーズで取り上げています（たとえば、オブジェクトの識別、クラス定義、継承の記録といったことです）。これは、多くの大規模プロジェクトをリスクに曝してきました。提案するシステムが本当は何をすべきかがはっきりしないうちに、複雑な設計判断を行うことになるからです。

この失敗や再作業のリスクを避けるために、Caterpillar's Fate のようなパターンランゲージを簡潔な方法論の一部として確立する必要がありました。この方法論には、分析フェーズと設計フェーズがあります。分析フェーズではいかなるオブジェクトのバイアスもかかりません。オブジェクトが使われるところやその方法は設計フェーズに委ねられます。設計フェーズではさまざまな設計戦略がとられます。たとえば、オブジェクト指向、オブ

ジェクト主導、構造化、フィルタツール主導、アルゴリズム的、多様な複合的手法、といったものが、技術的、または、非技術的な要因により選択されます。

オブジェクトと無関係な分析フェーズは、システムが「何を」すべきかという根本的な質問に答えるためのモデリングアクティビティからなります。ここでは、「どのように」それが実装されるかには言及しません（表１をご覧ください）。この方法論については、初期の論文[1, 2]でより詳細に論じられます。

## ２．Caterpillar's Fate パターンランゲージ

Caterpillar's Fate は、筆者とその顧客が精密な分析モデルから設計ソリューションを開発した際に学んだ知恵を捉えたものです。これは、その移行において筆者がしたことを文書化したものです。そこでは、「どうやってオブジェクトを見つけるか」というよくある問題にも取り組んでいます。

Caterpillar's Fate の形式は、意図的に Christopher Alexander[3, 4]のものをそのまま用いましたが、2 つほど違いがあります。設計者への「提案」の節を付け加えました。また、「スケッチ」はすべてのパターンにはありません。

提案は、重要なので無視できません。そのため、Alexander の形式を破りました。Alexander のパターンには一つひとつにスケッチが付されていますし、それは重要なことと思います。しかし、この論文の初期の版では、ページ数の関係から取り除きました。現在スケッチを追加している途中です。この版の論文には、もっとも重要なスケッチだけが付されています。スケッチの記法についての説明は付録にあります[1]。

| 分析の質問 | モデリング技法 |
|---|---|
| どんな情報が問題領域において重要か。 | 情報モデリング |
| この情報は、時とともにどう変化するか。また、どんな出来事によって変化するか。 | エンティティ状態モデリング |
| ユーザはだれか。また、どんな作業を達成するためにシステムを使うか。 | ユーザ作業リスト |
| 各作業におけるデータの流れはどうなっているか。 | ユーザ作業別データフロー図 |
| ヒューマンインタフェースは正確にどのように動作するか。 | 4 次元ヒューマンインタフェース見取図 |

**表１　分析フェーズにおける設計に関する質問とモデル技法**

[1] 訳注：PloPD1 には、ついていない。

Alexander のスケッチは、抽象的なものではありません。ある実際の問題の解を示すものです。この考え方に固執したところと抽象的な形式をとったところがあります。これについては、実験途上であり、ご意見、ご感想を歓迎します。

それでは、Caterpillar's Fate を説明しましょう。まずは、Alexander を真似て、最初に取り組むべき設計判断を挙げます。次の 3 つです。

**分析ドキュメントからアーキテクチャの設計を出発するにあたり、並列性に関する問題に最初に取り掛かるべきだろう。この問題に関するパターンは次の通り。**

1. **並列実行スレッド**（Concurrent Threads of Execution）
2. **並列スレッドの同期**（Synchronization of Concurrent Threads）
3. **協調作業パケット**（Collaborative Work Packets）

## パターン１：並列実行スレッド（Concurrent Threads of Execution）

同時に、または、（オペレーティングシステムが提供するタスクスイッチングのような）擬似的に同時に実行されるプロセスがシステムにあるなら、注意深くことを進める必要があります。

要求文書において、機能のことが話題になることはよくありますが、どのプロセスからどんな機能が利用できるかを正確に論ずることは滅多にありません。実際、そのようなドキュメントでは、機能の配置方法に言及すべきではありません。それは、設計判断によるものだからです。

**そこで：**他に存在するスレッドとは独立に存在しうる実行スレッドを識別します。これらのスレッドは、別のマシン上にある場合もあるし、各自の課題を担って同一マシン上で作業している、ユーザを表現している場合もあります。（このパターンにおいて、「ユーザ」とは、「システム外部にあって、システムにサービスを要求する実体」のことです。ユーザは人である場合もあるし、装置や単なる時間の経過である場合もあります。）

### 提案

各スレッドに名前をつけ、各スレッドのシステムにおける目的を記述します。

並列実行スレッドが識別されたら、並列スレッドの同期（Synchronization of Concurrent Threads(2)）と協調作業パケット（Collaborative Work Packets(3)）の問題に取り掛かります。設計に並列実行スレッドが一つしかなければ、プログラムの形（Shape of

Program(9)）に取り掛かれます。

## パターン2：並列スレッドの同期（Synchronization of Concurrent Threads）

並列実行スレッド（Concurrent Threads of Execution(1)）が識別されたら、並列スレッド間に必要な同期の識別に取り掛かれます。

並列スレッドの実行において、あるスレッドがある協定された状態に達するまで、他の並列スレッドは、処理を停止するか、ある処理形態にとどまる必要がある場合がよくある。

**そこで：**各並列実行スレッドについて、その実行ライフサイクルに渡って調べ、シグナルが他の並列スレッドに送られたり、他の並列スレッドからシグナルを受け取ったりする点を識別します。シグナルが表現する状況にあわせて、シグナルの名前を付け、また、そのシグナルの送信者および受信者を識別します。本パターンでは、シグナルは情報を伴わないものとします。それが届いたか届かないかだけがわかるものとします。

### スケッチ

図 1 は、家庭暖房システムを示したものです。それには、4 つの並列実行スレッドがあり、そのうちの 3 つが同期に関与します。

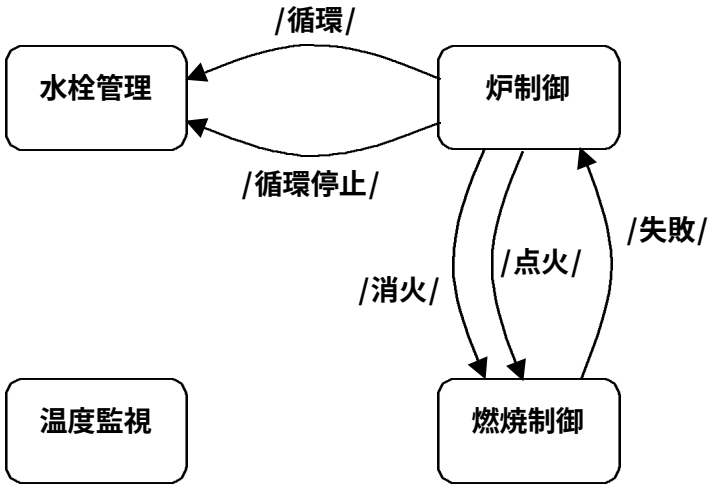


**図1： 家庭暖房システム**

### 提案

あるシステムに大量の同期があるように思えるなら、それは、さまざまなスレッドに大量の相互依存関係があることを示しています。「大量」であることは、各シグナルのシステム全体における目的とやり取りを理解するのに必要な集中力に注目して判定します。集

中力のレベルにレビュワーが耐えられなくなってきたら、欠陥を作りこむ可能性が増します。拡張にかかるコストも同様に増します。そのような場合、並列実行スレッド（Concurrent Threads of Execution(1)）に戻り、スレッドの数を変えたり、並列スレッドの同期（Synchronization of Concurrent Thread(2)）による設計判断が理解しやすくなるように、それらの目的を変えたりします。

そして、並列スレッドが互いにどのように同期するかが明確に理解できたら、プログラムの形（Shape of Program(9)）と危険領域の保護（Critical Region Protection(17)）に進めます。

## パターン3：協調作業パケット（Collaborative Work Packets）

システムにどのような並列スレッドがあるかが明確にわかれば、それらの間で移動する作業パケットの調査に取り掛かれます。多くのシステムでは、処理をプロセッサに割り振ったり、別の時点で行われる作業を定義およびスケジュールしたりする必要があります。この例は、多くのビジネスアプリケーションで見られるトランザクション処理システムです。

本パターン言語では、作業パケットの生産者と消費者が並列実行スレッドとなります。時を経ると、生産者と消費者は変化します。行われる作業の性質も変化します。そのため、並列実行スレッドと、その間で渡される作業とを明確に分離するよう設計することが重要です。

**そこで：**並列スレッドの各組について、各スレッドのライフサイクルを調べ、あるスレッドで開始され、次のスレッドで継続される具体的な作業を識別します。作業パケットが管理または隠蔽する情報および作業ユニットにあわせてその名前を付けます。各作業パケットについて、生産者および消費者となりうる並列スレッドを識別します。

### 提案

各作業パケットが担う責任は簡単に記述できるでしょう。責任は、通常「_____について知られるすべてのこと」を示します。

並列実行スレッド間でやり取りされる作業パケットを設計したら、作業パケットの内容（Work Packet Contents(4)）やプログラムの形（Shape of Program(9)）の設計に取り掛かれます。

作業パケット（これを「トランザクション」と呼ぶ環境もある）の設計には注意を要する。「並列性」のような設計は、初期に考慮しておく必要がある。それは、「大局図」の視点から、並列サブシステムや並列実行スレッドがどのように協調するかを表現するからであ

**る。作業パケットの詳細の大部分は、設計プロセスの初期に取り組むべきものである。これは、システムの設計が明らかになってきたときに再作業をしなくて済むようにするためである。本節には次のパターンがある。**

4. **作業パケットの内容**（Work Packet Contents）
5. **作業パケットの状態レポート**（Work Packet Status Report）
6. **作業パケットの完了レポート**（Work Packet Completion Report）
7. **作業パケットの優先順位**（Work Packet Priority）
8. **作業パケットのセキュリティ**（Work Packet Security）

## パターン4：作業パケットの内容（Work Packet Contents）

**協調作業パケット**（Collaborative Work Packet(3)）**を発見し名前を付け、その責任を記録し、それぞれの生産者および消費者にあたる並列実行スレッド**（Concurrent Threads of Execution(1)）**が識別されたら、作業パケットの内容についての設計に取り掛かれます。**

**作業パケットには、情報やさまざまな機能に対する指示が含まれます。作業パケット内部を設計する際に、それを用心深く組み立てないと、混乱に陥ると思われます。**

**そこで：各作業パケットについて、作業を遂行する上で必要な情報をすべて設計します。そのような情報は、データ、状態、要求記述、履歴、といったものです。データは、固有の処理で用いられます。状態は、そのときに作業パケットが置かれた状況や条件を示すものです。要求記述は、ある目標を達成するため、消費者スレッドに補助を依頼するものです。履歴には、作業パケットが過去に経てきたことに関する情報があります。これには、過去にそのパケットが経由した並列スレッドへの参照を含むことがよくあります（起源を知るデータ**（Data Knows Its Roots(21)））。

**作業パケットの状態レポート**（Work Packet Status Report(5)）**、または、作業パケットの完了レポート**（Work Packet Completion Report(6)）**がこの設計に有効なら、作業パケットには、識別子か名前を与える必要があります。作業パケットの優先順位**（Work Packet Priority(7)）**および作業パケットのセキュリティ**（Work Packet Security(8)）**を取り入れると、やはり内容に影響します。**

**生産者または消費者が変化した際に作業パケットに与えられる影響を最小限にするため、作業パケットには、将来に関する情報を置かないこととします。つまり、消費者がどのように処理するかを記述してはいけません。消費者は、作業パケットに、データ、状態、要求、履歴を問い合わせることで、処理の種類を決定できなければなりません。たとえば、SQL 句は要求記述です。実際の処理は、消費者に任されます。一方、特定の機能を実行し、作業パケットを別の消費者に渡すように指示するのは、将来に渡る制御をしようとするこ**

とにあたります。

### 提案

作業パケットの設計は、分析ドキュメントに従います。作業パケットに置くデータは、分析フェーズで構築された情報モデルから決定できます。状態の情報は、エンティティ状態モデルから導き出せます。履歴は、分析ドキュメントからはわからないと思います。それは、並列スレッドのような設計判断に依存するからです。

作業パケットが大きく思えてきたら、作業パケットの変化を注意して観察してください。作業パケットの種類が多ければ、作業パケットをサブタイプ化することを検討してください。それにより、作業パケットに置かれるデータ、状態、履歴の量が減らせるかもしれません。

数種類の作業パケットが同一の消費者に渡って処理され、処理がパケットごとに異なるようなら、処理の違いが作業パケットに隠蔽されるように設計してください。それを判別するには、作業パケットがその型を答える必要があるかどうかに注意すればよいです。もしそうなら、設計を変更しましょう。(この場合、オブジェクト指向設計者なら、ポリモーフィズムを使うことを考慮するでしょう。)

作業パケットが定義されたら、他の設計判断に取り組む必要があります。作業パケットの状態レポート(Work Packet Status Report(5))、作業パケットの完了レポート(Work Packet Completion Report(6))、作業パケットの優先順位(Work Packet Priority(7))、作業パケットのセキュリティ(Work Packet Security(8))、起源を知るデータ(Data Knows Its Roots(21))をご覧下さい。具体的な作業パケットをその生産者および消費者とともに識別できたら、プログラムの形(Shape of Program(9))に取り掛かれます。

## パターン5:作業パケットの状態レポート(Work Packet Status Report)

協調作業パケット(Collaborative Work Packet(3))および作業パケットの内容(Work Packet Contents(4))についての設計判断をしたら、作業パケットの状態についてのレポートを生産者のために作成する必要があるかどうかがわかるはずです。

設計によっては、設計者の並列実行スレッドが作業パケットを作り、それを消費者に送った後は、その作業パケットについて関知しない、とする場合もあります。その場合、本パターンは有効ではありません。

しかし、ユーザの要求により、生産者に作業パケットの状態を監視させたいと思うことがよくあります。こうした設計においては、どんな状態情報を生産者に返させるか、どんな刺激によってレポートを返させるか、作業パケットの果たす役割は何か、消費者である並列実行スレッドの果たす役割は何か、といったことに答えなければなりません。

**そこで**：分析で識別された要求を評価し、次の問いに対する答えに基づき設計判断を行います。

1．レポートが送られるのはいつですか。次の選択肢があります。
- 生産者の要求によってのみ
- 常に作業パケットの状態が変化したとき
- 消費者での処理が一定の目標を達成したとき
- ある時間が経過したとき

2．レポートにはどのような情報が必要ですか。作業パケットは識別子を要しますか。識別子は一意である必要がありますか。情報の一部を組み合わせると（たとえば、ユーザ名と送付時刻）作業パケットの一意な識別子になりますか。レポートの状態情報はどんなものですか。データ情報はどんなものですか。履歴情報はどんなものですか。レポートはカスタマイズできますか。

3．作業パケットが消費者に渡るときに、レポートが必要かどうかわかることがありますか。それなら、作業パケットに生産者へレポートを戻す責任を担わせましょう。作業パケットには生産者の記録が残っているからです。

作業パケットを送った後でレポートが必要になることがあるなら、消費者にレポートを生成する責任を担わせるよう設計してください。そうしたレポートは、一つの作業パケットについてのものではなく、消費者の制御下にある作業パケットおよび消費者が知っている作業パケットすべての集計です。

4．状態レポートのセキュリティに関する要求を決めます。ある並列実行スレッドが、他の並列スレッドが作成した作業パケットについての状態レポートを要求したり自動的に受け取ったりしてもよいですか。この要求を利用可能にする特権を与えますか。パスワードで保護しますか。状態レポートは暗号化する必要がありますか。

作業パケットと消費者のどちらにレポートを作成して送る責任を主に担わせるかを決めます。そして、残りの一方にどんな補助を求めるかを決めます。上述の設計に関する問いに対する答えに対応するように、並列実行スレッド間で通信する仕組みを設計します。

**提案**

作業パケットの状態レポートについての設計は、可能な限り単純にしてください。余分な「おまけ」はこの部分の設計には加えません。システムは状態レポートを作らなければ

ならないとしても、それは主目的ではありません。その主目的は作業パケットを処理することです。設計の創造力をそこに集中します。

これらの設計判断がすんだら、作業パケットの設計に関することが明確になります。特化されたプログラムの形（Shape of Program(9)）が識別されたすべての状態レポートの受領に対応していることを確認してください。

## パターン6：作業パケットの完了レポート

協調作業パケット（Collaborative Work Packet(3)）、作業パケットの内容（Work Packet Contents(4)）、並列スレッドの同期（Synchronization of Concurrent Thread(2)）について、設計判断ができていれば、消費者の並列実行スレッドで、作業パケットがその処理を完了したとき、どうすべきかについての設計判断に取り掛かれます。

作業パケットの処理が完了したことの応答については、さまざまな設計の選択肢があります。ここでは、これらの応答について考慮します。

**そこで：**分析で識別された要求を評価し、次の問いに基づき設計判断を行います。

1．何が返りますか。成功／失敗のメッセージですか。作業パケットにあった問い合わせへの答えですか。データ、状態、履歴が変更された、作業パケットそのものですか。

2．失敗のレポートが返るとき、失敗についてどの程度のことを伝えますか。場合によって、エラーの性質や、他にどんなやり方があるかの示唆を得たいと思うこともあります。

しかし、システムによっては、失敗そのものの詳細を生産者スレッドと共有すると、セキュリティ上の要請が破られることもあるので、注意してください。

3．セキュリティのために、応答を暗号化する必要がありますか。パスワードによる保護は必要ですか。

4．応答を送る準備ができても、生産者の並列実行スレッドが利用不能な場合に、応答をどう扱うかの方針を設計します。これは、分散システムにおいて起こります。原因は、ネットワークが停止したり、生産者のコンピュータがオフラインになったり、それ以外にもさまざまあります。応答を要求している生産者が、システムに永遠に戻ってこないこともあります。この状況をどう扱いますか。

### 提案

作業パケットの状態レポート（Work Packet Status Report(5)）の提案がここにも当てはまります。完了レポートは簡潔明瞭にしておきます。

これらの設計判断が済めば、作業パケットに関することが明確になります。特化されたプログラムの形（Shape of Program(9)）が識別されたすべての完了レポートの受領に対応していることを確認してください。

## パターン7：作業パケットの優先順位

協調作業パケット（Collaborative Work Packets(3)）についての設計が済み、一般的なプログラムの形（Shape of Program(9)）を変形して特殊なものにする際に、作業パケットが消費される順番を決定する必要があります。

作業パケットを後で処理するようにキューに蓄えるシステムもあります。こうしておけば、消費者の処理リソースが空いたり、手に入りやすくなったりするため、または、作業パケットがある時点に集中し、システムの処理容量を越えることがあるためです。そこで、処理待ちの作業パケットが複数あるときに作業パケットを選択する方法について設計判断を下さなければなりません。

**そこで**：処理待ちの作業パケットから一つを選択する方針を明確にしておきます。（これは、ほとんどの分析ドキュメントでは見過ごされることです。）「先入れ先出し」でことは足りるかもしれません。しかし、一般的な性能仕様として、要求が隠されていることがあります。これを見つけるのは簡単ではありません。次のような選択方法を心に留めてドキュメントを調べてください。

1．先入れ先出し。
2．割り振られた優先順位。作業パケットの型に優先順位を割り当ててもよいし、作業パケットを作った生産者に特権的な優先順位を割り当ててもよい。
3．締め切り指向。各作業パケットには完了の締め切りがあり、締め切りを破る作業パケットの数が最小になるよう作業を行う。
4．コスト-ペナルティ分析。各作業パケットには完了の締め切り以外に、締め切りを破った場合のコストがあり。このコストを最小にするように選択する。

これらは、基本的な設計方針です。これらのアプローチの組み合わせや、別の選択方針を必要とするシステムもありえます。重要なのは、この選択方針が意識的に決定され、消費者に特化されたプログラムの形（Shape of Program(9)）に設計として組み込まれていることです。採られる方針によって、作業パケットの内容（Work Packet Contents(4)）に戻って、選択方針に対応するようにする必要があるかもしれません（作業パケットに優先順位や締め切りを追加する必要があるでしょう）。

## 提案

選択システムは、可能な限り簡単なものにしておきます。将来、提案されうるいかなる種類の変更にも備えておきましょう。そのため、ここではシステムに「フック」を追加したりしません。また、予断を排し、将来の成長を妨げるようなショートカットを作らないようにします。なぜ「フックを作らない」かというと、将来は予想通りになるとは限らないからです。たとえそうなったとしても、システム中のフックは、現時点で稼動するコードのほどにはテストされていないでしょう。これは疑いを買うことにつながり、その後のメンテナンスでも、使われないフックが顧られることはないでしょう。

この「フックを作らない」という提案は、本パターン以外にも設計プロセス全体に当てはまるものです。ここで述べたのは、ここが初めにこの助言を必要とするところだったからです。他のところでは繰り返さないようにしますが、繰り返し申し上げたくなるほどのことです。

これらの設計判断が済めば、作業パケットの設計が明確になります。消費者に固有のプログラムの形（Shape of Program(9)）が選択方針の設計判断に対応していることを確認してください。また、生産者のプログラムの形（Shape of Program(9)）と作業パケットの内容（Work Packet Contents(4)）が十分洗練されていることを確認してください。

## パターン8：作業パケットのセキュリティ

作業パケットの内容（Work Packet Contents(4)）および協調作業パケット（Collaborative Work Packets(3)）に対処したら、作業パケットのセキュリティ問題に取り掛かれます。

デリケートな内容の情報や作業指示を含む作業パケットが、あるマシンで組み立てられ、ネットワークを通じて他のマシンに送られる場合、生産者および消費者の設計とともに、作業パケットの設計でも、いくつかのセキュリティの問題に取り組む必要があります。

**そこで：**次の設計上の問題について考慮しましょう。

- パケットの情報は暗号化する必要がありますか。その答えは、（あるハードウェア構成を仮定したときに）誰かが作業パケットにアクセスすることで利益を得ることがあるかどうかによります。誰かがその情報にアクセスできるかどうかではありません。ネットワークが機密保持に対応していると信じていても、将来ハードウェア構成が変化したときにもそうである保証はどこにもありません。

- システムで、ある作業パケットの生産者が、他のものにはない特権（情報のアクセスまたは機能の実行権限）をもっている場合、生産者に不適切な情報または機能へのア

クセスの予防措置を設計します（ヒューマンインタフェースのため）。また、消費者内部でセキュリティチェックが行われるよう設計します。作業パケットは、消費者がセキュリティチェックを行うのに必要な情報を運ぶ必要があります。

- あるシステムでは、消費された作業パケットを定期的に検査し、異常を探知します。また、あるシステムでは、何が行われたかを追跡するための監査証跡を取ります。用件から、これらのセキュリティに関する要求が示されるなら、必要な情報とアクティビティのログをとるような設計を行いましょう。

**提案**

通常、セキュリティは、考慮されないか、せいぜい、ほとんどの要件および分析ドキュメントで曖昧に述べられている程度です。高度なセキュリティに対する顧客の要望は、この問題とコストについての理解が浅いため、「あればいいな」程度のことがほとんどでしょう。そうした状況で、セキュリティが重大な問題ならば、設計のアクティビティをすべて停止し、分析に戻り、効果的なセキュリティモデルを打ち立ててください。

ここまで、暗号、パスワード、ログといったセキュリティに関する問題についての取り組みを見てきました。本パターンが成熟するにつれ、論じるべきセキュリティの話題が増えていくことでしょう。

**この節では、あるプログラムに関して起こるうる設計早期の問題について見ていきます。この問題には、プログラムの形に関しての、早い段階での意思決定が含まれます。それは、どのように、刺激を受けとめ、それに反応するか、また、ヒューマンインターフェースがどのように、残りのシステムに影響を与えることなく、プログラムの形に加えらるのかといったことです。また、この節では、危険領域を見つけるために、並列性に関する問題を再検討するパターンも扱います。この節で取り上げるパターンは、次の通りです。**

9. **プログラムの形**（Shape of Program）
10. **システムの住人の役割**（System Citizen's Role）
11. **意思決定者の役割**（Decision Maker's Role）
12. **労働者の役割**（Worker’s Role）
13. **インタフェースの役割**（Interface’s Role）
14. **伝達者の役割**（Informational Role）
15. **小さな集団によるシステム**（Small Family Systems）

[2] 訳注：PloPD1 には、ついていない。

16. **対話により達成される作業**（Work Accomplished Through Dialogs）
17. **危険領域の保護**（Critical Region Protection）
18. **イベントの取得**（Event Acquisition）
19. **イベントの伝送**（Event Routing）
20. **特殊なインタフェースとしてのヒューマンインタフェースの役割**（Human Interface Role Is a Special Interface Role）

## パターン9：プログラムの形（Shape of Program）

システムの並列実行スレッド（Concurrent Threads of Execution(1)）と並列実行スレッドの同期（Synchronization of Concurrent Threads (2)）が明確になり、協調作業パケット(3)の詳細が固まれば、プログラムの全体の形に、特定の形式を適用しはじめられます。１つしか並列実行スレッド（Concurrent Threads of Execution(1)）がないシステムでは、このパターンから考え始めれば良いでしょう。

要求ドキュメントをソフトウェアの設計に持っていく初期の段階では、考えるべき多くの問題が存在します。通常、それは、一度に頭に詰め込めないほどの量でしょう。そのため、システムの狭い範囲における設計項目の最適化に注力する方が容易です。ただし、「全体図」が危険にさらされます。

**そこで：**「典型的な全体図設計」が役に立つ限り（小さな集団によるシステム（Small Family Systems(15)）、対話により達成される作業（Work Accomplished Through Dialogs(16)）といったパターンを使わない限り）は、それを使っておきます。このパターンは、典型的なプログラムの形を提供します。プログラムは、さまざまな形を取り得ますが、このパターンは、筆者がこれまでに関与した全てのシステムについて、出発点として十分に機能してきました。

このプログラムの形を表現するには、階層構造が最適です。各層のオブジェクトは、それと同じか下位の層にある設計コンポーネントのサービスをコールできますが、その上位の層にはコールできません。

### スケッチ

図 2 において、1 番目の層にあるオブジェクトは、各自の課題に向けて作業している並列スレッドのコミュニティにおいて、この並列実行スレッドが善良な住人であることを保証する責任を持ちます。システムの住人の役割（System Citizen's Role(10)）をご覧ください。

2 番目の層にあるオブジェクトは、構築中のシステムの意思決定に対して責任を持ちます。意志決定者の役割（Decision Makers' Role(11)）をご覧ください。3 番目の層にあるオブジェクトは、システムで必要な作業を遂行します。労働者の役割（Workers' Role(12)）をご覧ください。4 番目の層にあるオブジェクトは、外部エンティティから、インタフェースを隠す責任を持ちます。インタフェースの役割（Interface's Role(13)）をご覧ください。

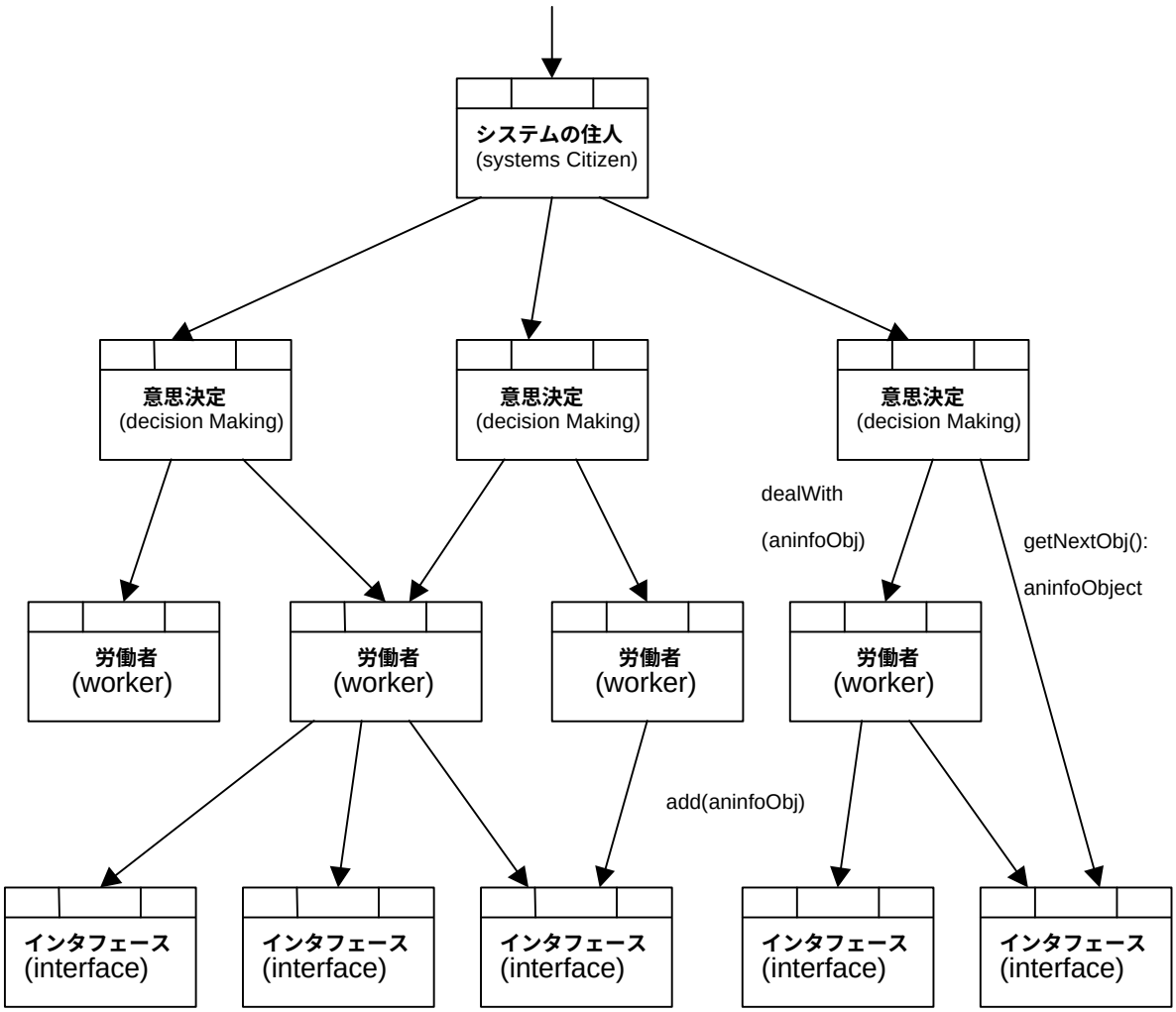


**図 2：** 図 2 の 1 番目の層にあるオブジェクトは、各自の課題に向けて作業している並列スレッドのコミュニティにおいて、この並列実行スレッドが善良な住人であることを保証する責任を持ちます。システムの住人の役割（System Citizen's Role(10))をご覧ください。

この階層構造のシステムには、階層構造のシステム中を移動し、他のオブジェクトを訪れてまわる型のオブジェクトが必要となります。この移動するオブジェクトは、カスタマ

イズされた機能とともに、ふさわしい情報を提供するする責任を持ちます。オブジェクト指向の考え方に慣れている人は、これらのオブジェクトはポリモーフィックであると言うでしょう。それらのオブジェクトは、パラメータとして送られます。このオブジェクトは、図２では、anInfoObj という言葉で表されています。伝達者の役割（Informational Role(14)）をご覧ください。

このパターンを使用してなされた設計判断は、一般的なプログラムの形を、分析ドキュメントにある成果を反映した固有の形に変形します。この変形をなしとげるには、階層構造のプログラムの形を使うと同時に、次のパターンを検討すると有効でしょう。システムの住人の役割（System Citizen's Role(10)）、意思決定者の役割（Decision Makers' Role(11)）、労働者の役割（Workers' Role(12)）、インタフェースの役割（Interface's Role(13)）、伝達者の役割（Informational Role(14)）といったものです。階層構造でうまく行かなければ、小さな集団によるシステム（Small Family Systems(15)）および対話により達成される作業（Work Accomplished Through Dialog(16)）を検討しましょう。マウスクリックやキーストロークなどを扱う方法を考え始めたら、イベントの取得（Event Acquisition(18)）、イベントの伝送（Event Routing(19)）が役立つでしょう。

## 提案

オブジェクト間のサービスは、1 つの層の中でコールされるかもしれません。しかし、これは、最低限にすべきです。その理由は、層によって変わります。一般に、同じ層のオブジェクト同士の相互作用が増えると、オブジェクト単独で再利用することができなくなります。これは特に、インタフェースオブジェクトおよび労働者オブジェクトに言えることです。

意思決定オブジェクトは、アプリケーション固有の知識を持っているため、再利用されないでしょう。そのため、これについては、再利用では説明になりません。意思決定者間に相互作用があると、メンテナンスエンジニアは、どちらか一方の意思決定者を変更するに当たり、両方の意思決定者の操作を正確に理解する必要があります。これは、ある程度まではうまくいきます。しかし、度を越すと、本当の問題を引き起こすことになります。そのような場面に出会ったら、これら 2 種類のオブジェクトの境界と責任を定義した分析および設計判断を再検討してください。

大きなシステムでは、4 つの階層で十分でないように思われます。筆者は、ここで述べたオブジェクトの役割に、プログラムの形が再帰的に使われているのを見たことがあります。ほとんどの場合、それが行われるのは、意思決定者オブジェクトと労働者オブジェクトでした。また、これらのオブジェクトの内部動作が、小さな集団によるシステム（Small Family System(15)）の形式となっていることもありました。

一般的なプログラムの形からシステム固有の形へと変形する際、並列スレッドの同期

（Synchronization of Concurrent Threads(2)）で識別された全てのシグナルと協調作業パケット（Collaborative Work Packet(3)）で識別された全ての作業パケットが勘案されていることを確認してください。

一般的な階層構造から分析ドキュメントの成果を反映した固有なものへと変形したら、この初期的なアーキテクチャについてのパターンランゲージでは対象としていない、性能等の品質項目に関するシステムの洗練に取り掛かれます。

### パターン10：システムの住人の役割（System Citizen's Role）

並行実行スレッド（Concurrent Threads of Execution(1)）が明確になり、プログラムの形（Shape of Program(9)）を一般的なものから固有のものへと移行させ始めたら、システムの住人オブジェクトを一つ作ることを考慮します。

ほとんどのプログラムは、同時にいくつかのアプリケーションが動作しているようなプラットフォームに存在します。（例えば、Mac OS 7.X、OS/2、Winodows 3.X など）これらの動作環境は、各アプリケーションが自身の上で実行可能であることを想定しています。各動作環境は、それぞれ独自の方法でアプリケーションを動作させます。プラットフォーム間で簡単に移植できるアプリケーションを設計するのは、ビジネス上の理由からでしょう。

**そこで**：アプリケーションには、アプリケーションのコミュニティにおける「善良な住人」として従わなければならないプロトコルがあります。これを知る責任を担う単一のオブジェクトを設計します。特定の動作環境がどのように、マウスクリックやキーストローク等を扱うか考える上で、イベントの取得（Event Acquisition(18)）とイベントの伝送（Event Routing(19)）が役立つでしょう。

### 提案

システムの住人の責任についての情報は、分析ドキュメントにはありません。そういった情報は、通常、選択した動作環境のプログラマリファレンスマニュアルかチュートリアルマニュアルで見つかります。

これらの知識を全て 「単一のオブジェクト」に持たせると、そのオブジェクトが非常に大きくなるのでは、というコメントをレビュワーから頂いたことがあります。筆者は、システムの住人オブジェクトの内部設計が多くのオブジェクトから構成されることを思い描いています。その設計の目標は 、プラットフォームが想定するアプリケーションの振る舞いに関する知識を 1 つのユニットに局所化することです。

このパターンを乗り越えたら、意思決定者の役割（Decision Makers' Role(11)）および特殊なインタフェースとしてのヒューマンインタフェースの役割（Human Interface Role

Is a Special Interface Role(20)）の適用に取り掛かれます。

## パターン11：意思決定者の役割（Decision Makers' Role）

プログラムの形（Shape of Program(9)）を一般的なものから固有なものへと移行させ始め、システムの住人の役割（System Citizen's Role(10)）を定義したら、アプリケーションに関する高次の方針決定を設計し始められます。

「実使用される」システムには、アプリケーションの全操作の制御に関する設計がつきものです。この設計は、関心をそそるものです。これは、ワークフロー、制御といった、アプリケーションの最上位の振る舞いについての意思決定[3]となります。顧客がアプリケーションを競合他社のものと比較する上で、この意思決定がものをいいます。

制御そのものに関心が向けられないと、システムの方針／意思決定に関する動作は、多くのオブジェクトに散らばってしまうでしょう。そして、メンテナンスが困難になります。方針と意思決定が散らばっていると、方針または意思決定オブジェクトを変更する際に、メンテナンスエンジニアは、多くのオブジェクトの動作を習得しなければなりません。

**そこで**：主として、アプリケーションの意思決定に関する動作について責任を担うオブジェクトを少なくとも 1 つ作ります。つまり、方針を決定する動作と方針を実行に移す仕組みとを分離します。

### 提案

意思決定オブジェクトは、変なオブジェクトに見えます。外部メソッドが doIt 1 つだけの ことも よくあります。しかしそれらは、メンテナンスしやすく再利用可能なシステムを構築する上で重要です。再利用できそうもないオブジェクトの集合に、再利用できそうもないオブジェクト（つまり、あるアプリケーションに関しての方針／意思決定）を加えることで、再利用可能性が得られるのです。意思決定オブジェクトはアプリケーションに固有なため、同じアプリケーションを他の環境に移植でもしない限り、再利用されません。

方針／意思決定オブジェクトに配置すべきものは、通常、処理仕様からわかります。それより下位の全ての操作は、労働者オブジェクト（パターン 12 をご覧ください）、伝達者オブジェクト（パターン 14 をご覧ください）、インタフェースオブジェクト（パターン 13 をご覧ください）に割当てられるます。したがって、その残りが方針／意思決定オブジェクトに配置すべきものです。

ここまで来れば、アプリケーションがどのようになすべき作業を制御するか、明確にわかるでしょう。これで、労働者の役割（Workers' Role(12)）オブジェクト、インタフェースの役割（Interface's Role(13)）オブジェクト、特殊なインタフェースとしてのヒューマ

ンインタフェースの役割（Human Interface Role Is A Special Interface Role (20)）オブジェクトをつなぎ合わせ始められます。意思決定者は、労働者オブジェクトおよびインタフェースオブジェクトとの間でやりとりされる、多くの伝達者の役割（Informational Role(14)）オブジェクトを扱うことになると思われます。

### パターン12： 労働者の役割（Workers' Role）

プログラムの形（Shape of Program(9)）を一般的なものから固有なものへと移行させ始めたら、アプリケーションをその目標に近づけるために役立つオブジェクトを設計し始められます。

いくつかの設計判断の下、多くのプログラムで、アプリケーションをその目標に近づけるのに役立つ関連サービスを用意することがあります。

**そこで**：意思決定者による責任の遂行を補助する、具体的なオブジェクトを設計します。コレクションオブジェクトは、典型的な労働者オブジェクトです。

### 提案

労働者オブジェクトについて、第 1 の目標は、意思決定者が行っている、直接アプリケーションの制御に関係のない作業をすべて労働者オブジェクトに肩代わりさせることです。第 2 の目標は、類似した別のアプリケーションで再利用されるような労働者オブジェクトを作ることです。

労働者オブジェクトは、通常、イベント[4]別 DFD を調べれば、簡単に見つかります。それらは、よくデータストアに隠れています。処理仕様から、それらが労働者オブジェクトであることが確認され、そのメソッドが識別されます。

労働者オブジェクトを識別したら、これらのオブジェクトの内部設計を検討し始められます。全体図が局所的な図にぴったりとはまるのは、気持ちよいものです。ここから先、それが設計者の関心となります。

### パターン13　インタフェースの役割（Interface's Role）

プログラムの形（Shape of Program(9)）を一般的なものから固有なものへと移行させ始めたら、システム外部のエンティティに固有な振る舞いを隠蔽するオブジェクトの設計に取り掛かれます。

孤立したシステムが構築されることはありません。相互作用する他者がいるはずです。

---

[3] 訳注：ワークフローを意思決定プロセスとみなしていると思われる。
[4] 訳注：このイベントはユーザ作業に対応すると思われる。

それは、設計者の支配下にないハードウェアやソフトウェアかもしれません。その場合、これらの外部エンティティの振る舞いは、おかまいなしに変化するかもしれません。

**そこで：**構築中のシステムを守るため、外部エンティティの振る舞いをあるオブジェクトに隠蔽します。そのオブジェクトは、システムの残りの部分に対して、高水準で抽象的[5]なサービスを提供する責任を持ちます。

### 提案

インタフェースオブジェクトは、イベント別 DFD で見つかります。それは、通常、外部エンティティです。抽象的サービスは、処理仕様を調べれば決定できます。

インタフェースオブジェクトを識別したら、これらのオブジェクトの内部設計に取り掛かれるでしょう。全体図が局所的な図にぴったりとはまるのは、気持ちよいものです。ここから先、それが設計者の関心となります。意思決定者オブジェクトからヒューマンインタフェースを隠蔽する特殊な種類のオブジェクトも存在します。それについては、特殊なインタフェースとしてのヒューマンインタフェースの役割（Human Interface Role Is a Special Interface Role (20)）をご覧ください。

## パターン14：伝達者の役割（Informational Role）

プログラムの形（Shape of Program(9)）を一般的なものから固有なものへと移行させ始めたら、意思決定者の指示のもとにシステム中を移動し、インタフェースオブジェクトおよび労働者オブジェクトを訪れてまわる、オブジェクトの設計に取り掛かれます。

ポリモーフィズムを使うと、多くの効果が得られることは知られています。プログラムの形（Shape of Program(9)）は、多くのオブジェクトを柔軟性のない階層システムとして構造化します。これは、古くからの構造化設計法の考え方と似ています。そうした構造には、いくつかの利点がありますが、ポリモーフィズムは扱えません。構造化技法において、もっとも近いことをするには、「放浪データ（tramp data）[6]」として知られる複雑なデータ構造を渡すことの効果を認める必要があります。これは、アプリケーション全体にデータ構造についての知識が必要となるため、最善の設計手法とは考えられませんでした。オブジェクト指向の考え方は、この問題を解決する方法を与えてくれます。オブジェクトを渡せばよいのです。

**そこで：**具体的な情報の構造を知られずに、それをアプリケーション内の適切の場所に

---

[5] 訳注：外部のソフトウェアやハードウェアが提供するインタフェースという、システムの主たる目的からは些末なことが捨象されているという意味である。

[6] 訳注：目的のモジュールにたどり着くためだけに、それを必要としないモジュールに渡されるデータ。設計上まずいこととされている。

届ける責任を担うオブジェクトを作ります。これは、伝達者オブジェクトとして知られます。

### 提案

設計上、多くの IF 文や SWITCH 文等で分岐先を決めるため、伝達者オブジェクトにそのオブジェクトの型を尋ねる必要があることに気付いたら、ともかく立ち止まって、型に関する作業を伝達者オブジェクトの中に埋め込む方法を探してください。

伝達者オブジェクトは、イベント別 DFD の情報の流れに注目すれば見つかります。有用で意味のある情報があり、インタフェースの役割（Interface's Role(13)）に流れ込むか、そこから流れ出て、関心のある処理（処理仕様で述べられている）に結びついていれば、それは伝達者オブジェクトと思われます。特に、この情報について多くの型とバリエーションがあることがデータ辞書からわかれば、これが当てはまります。

Smalltalk の専門家は、伝達者オブジェクトと Model-Viewer-Controller モデルの 3 つのモデルコンポーネントとの類似性を認めています。これについては、特別なインタフェースとしてのヒューマンインタフェースの役割（Human Interface Role Is a Special Interface Role (20)）で、さらに述べています。

伝達者オブジェクトを識別できたら、これらオブジェクトの内部設計の検討に取り掛かれます。全体図が局所的な図にぴったりとはまるのは、気持ちよいものです。ここから先、それが設計者の関心となります。

## パターン15： 小さな集団によるシステム

プログラムの形（Shape of Program(9)）を一般的なものから固有なものへと移行させ始めたら、システムが階層であるために、設計がぎこちなくなることがあるでしょう。そのような場合も、プログラムの形を捨てる必要はありません。「小さな」形をシステムの関心のある部分に付け加えるだけです。特別なインタフェースとしてのヒューマンインタフェースの役割 （Human Interface Role Is a Special Interface Role (20)）から得られた内部設計にはこのパターンが有効なことがよくあります。

特に、チームとしてうまく協調動作する少数のオブジェクトを作ることが設計上有用となる部位があることに気づくでしょう。この設計上の部位に固有の重要な特徴は、この「小さな集団」のオブジェクトの中では、どのオブジェクトも他のオブジェクトを制御する責任を負わないことです。

**そこで：**1 家族のような小さな集団を設計します。しかし、それが「健全な」集団であることは確認する必要があります。「健全な」集団において、それぞれのオブジェクトには明確な責任があり、それは、その集団にいる他のメンバの責任と簡単に区別できます。こ

の集団がシステムに提供する各サービスにつき、それぞれ１つのオブジェクトが、制御に関する責任を持ちます。そのオブジェクトは、目標を達成する上で他のオブジェクトに助けを求めます。古典的な Model－View－Controller モデルは、小さな集団によるシステムの一例 です。

集団のメンバ間で責任の境界を決して破らないことを保証してください。その境界を破ると、あるアクティビティを成し遂げる責任がどのオブジェクトにあるかが明確でない状況が生まれます。その場合、メンテナンス作業で過ちを犯しやすくなります。その結果、あるアクティビティが 2 度実行されるようになったり、別のアクティビティが全く実行されなくなったりします。

使用言語や動作環境によっては、小さな集団によるシステムをこのように実装することはできないかもしれません。設計の視点では、オブジェクトの小さな集団がプログラムの形（Shape of Program(9)）によって作られた階層システムの中にあるとき、それを 1 つのオブジェクトとして考えることができます。

### 提案

小さな集団によるシステムは、数種類のオブジェクトのみで構成されます。それは、2、3 種類、まれに、4 種類ぐらいといったところです。筆者は、4 種類以上のオブジェクトが、小さな集団としてうまく機能するのを見たことがありません。

小さな集団は通常、分析ドキュメントからは識別できません。それらは、通常、多くの設計目標を同時に満たすため、または、階層システムに現れるぎこちない構造を簡潔にするために作られます。それらは多くの場合、過去の経験を当てはめてみることで得られる、巧みな発明品です。

筆者は、小さな集団によるシステムを、階層システムに対する 副次的な設計戦略として見ています。1 つの集団がどのように動作するか理解するために、メンテナンスエンジニアは、かなりの作業量を要するからです。御自身の設計判断や文書化計画を手引きするために、この尺度をご利用ください。

小さな集団によるシステムが必要であると判断したなら、これらオブジェクトの内部設計に取り掛かれます。全体図が局所的な図にぴったりとはまるのは、気持ちよいものです。ここから先、それが設計者の関心となります。

## パターン16：対話により達成される作業（Work Accomplished Through Dialogs）

プログラムの形（Shape of Program(9)）を一般的なものから固有なものへと移行させ始めたら、2 つのオブジェクトが目標を達成するため、互いのメソッド起動を要する場面に出会うでしょう。その場合も、プログラムの形（Shape of Program(9)）を使って開発

した階層構造のシステムを捨てる必要はなく、むしろ、オブジェクトがメソッドを互いにコールしてもよいこととする設計判断を付け加えればよいです。

システムが成し遂げなければならない作業を行う上で、2 つのオブジェクトの責任を融合させる必要がある場合があります。どちらのオブジェクトにも、要求を満たすために必要な情報、または、処理能力がすべてはそろっていません。2 つのオブジェクトを組み合わせるのが正しい考えとは思えないかもしれません。しかし、各オブジェクトの責任は明確に表され、それらの相互作用を制御する意思決定者オブジェクトがこれから見つかる見込みもないようです。

**そこで：**2 つのオブジェクトが互いに対話してよいこととする設計を選択します。第 1 のオブジェクトが、第 2 のオブジェクトのメソッドを起動し、そのメソッドでの処理から、第 1 のオブジェクトのメソッドを呼びます。第 2 のオブジェクトが 第 1 のオブジェクトとの対話を始めるはずです。そうでなければ、単なる階層構造とすればよいです。

**提案**

2 つのオブジェクト間の対話は、ある制限の下では簡単に理解できます。対話により起動されたメソッド（つまり、第 2 のオブジェクトが 第 1 のオブジェクトをコールして起動されたメソッド）が問い合わせ的である場合、それは簡単に理解できます。問い合わせ的なメソッドの唯一の目的は、第 2 のオブジェクトがその作業を達成する助けとなる情報や回答を提供することです。

問い合わせ的の反対は制御的です。制御的なメソッドは、第 1 のオブジェクトの状態を変化させます。最悪の場合、それにより 第 1 のオブジェクトがさらに別のオブジェクトと制御的に相互作用します。制御的なメソッドを通じた対話による深い影響を把握するのは困難です。レビュワーとメンテナンスエンジニアは、微妙な影響を見逃すでしょう。こうした対話は避けてください。

2 つ以上のオブジェクト間での対話は、不幸の手紙対話と呼ばれます。これらは、どんな犠牲を払っても、避けるべきです。あるオブジェクトが 第 2 のオブジェクトにメッセージを送るような設計を想像してください。第 2 のオブジェクトは、第 3 のオブジェクトのメソッドを呼び、あるとき、第 n のオブジェクトが 第 1 のオブジェクトに情報を求めます。その場合、**n**個全てのオブジェクトに依存関係が存在します。**n**個全てのオブジェクトについて考慮し終えるまで、どのオブジェクトにもメンテナンス上の変更を加えることができません。このことは、情報隠蔽や関心の分離として知られること全てに違反します。不幸の手紙対話は、絶対しないでください。

問い合わせ的なメソッド間で行われる対話を慎重に使用し、プログラムの形（Shape of Program(9)）から行ってきた設計を洗練したら、プログラムの形を一般的なものから固有の形に変換する作業に戻れます。

## パターン17：危険領域の保護（Critical Region Protection）

並列実行スレッド（Concurrent Threads of Execution(1)）が識別され、それらのプログラムの形（Shape of Program(9)）を一般的なものから固有なものへと移行させ始めたら、潜在的な危険領域の問題を探し始められます。

並列システムは、インタフェースオブジェクト(13)、伝達者オブジェクト(14)、労働者オブジェクト(15)の形式で表された同じ資源を使用することがよくあります。それらは、意思決定者オブジェクト(11)を共有することもあります。2 つの並列実行スレッドが同じオブジェクトに同時にアクセスすると、害を引き起こす恐れがあります。この害は、共有するオブジェクト、それぞれのスレッドで実行されるべき作業、まれにメソッドを起動している並列実行スレッド自身の作業にも影響を及ぼします。

**そこで：**各並列実行スレッドについて、プログラム固有の形を調べ、共有されるオブジェクトを識別します。共有される各オブジェクトについて、同時アクセスからの保護が必要かどうか判定します。その結果、必要なら、安全な利用を保証してくれる保護の仕組みを開発します。

### 提案

この問題に対しては、設計解法がいくつかあります。これらは、どんな OS の教科書にも見られます。次のようなものです。

2 つ目のスレッドが危険領域に入るのをブロックする方法。

危険領域に入る前に、それぞれのスレッドに許可を取らせる方法。

高い優先度をもつスレッドが危険領域に入った時には、低い優先度のスレッドの作業を終了させる方法。

それぞれのスレッドに独自のデータ記憶空間を与えて、危険領域を取り去る方法。

これらは、見つけることが非常に難しい潜在的な欠点ですが、このパターンに取り組めば間違いなく避けられます。

## パターン18：イベントの取得（Event Acquisition）

プログラムの形（Shape of Program(9)）を一般的なものから固有なものへと移行させ始め、システムの住人の役割（System Citizen's Role(10)）を考慮しているとき、このパターンは、ある設計上の問題を解決するのに有用です。

すべてのプラットフォームは、独自の方法でイベントを扱っています。「イベント」とは、マウスクリック、キーストローク等が発生したことを示すシグナルのことです。これ

について、かなりの支援をしてくれる動作環境もある一方、ほとんどシステム設計者任せのものもあります。ともかく、プログラムの形は、首尾一貫した設計を保証するように適応させる必要があります。

**そこで：**イベント取得機能に関する設計を調べます。アプリケーションにとって有意義で、プログラムの形を洗練するのに役立つ機能を選びます。設計上考慮すべき重要な質問は、次の通りです。

- 正確には、どのようにイベントが取得されますか。動作環境がイベントを検出し、それを取得および解読しますか。それとも設計者にその作業の一部または全体が委ねられますか。すべてのイベントは、一意のものとして扱われますか。それとも違う方法ですか。動作環境が用意するイベント取得機能をオーバライドできますか。どんなときにそれができればよいですか。

- どんな動作環境の機能まで適応したいですか。それは、プログラムの形（Shape of Program(9)）に、どう影響しますか。これらの機能は、システムの住人オブジェクト(10)に収められますか。そうでなければ、アプリケーションを異なるプラットフォームに移すとき、どうなりますか。

- 自分でイベント取得機能の設計を用意するのは、どんなときですか。それは、プログラムの形（Shape of Program(9)）に、どう影響しますか。キャンバス上でグラフィックの輪郭をドラッグするときというのが一例です 。プラットフォームのイベント取得機構が不適切なのは、どんなときですか（例えば、性能や機能に不満がある場合）。

- 動作環境は、ハードウェア割込みに対応しますか。動作環境は、「ハードウェアドライバ」に特別な設計をすることを想定していますか。これらの設計判断を満たすために、プログラムの形（Shape of Program(9)）には、どのような修正が必要ですか？

- 動作環境は、「イベント消費オブジェクト」の概念に対応しますか。これは、特定のイベントを登録しておくオブジェクトです。そうなら、そうした強力な機能を利用すると、プログラムの形はどう変化しますか。この登録機能を使用しないのは、どんなときですか。

**提案**

ある動作環境でのアプリケーション設計が初めてなら、さまざまなプログラムの形を粗く描いてみて、それらをチームで検討してもらい、いくつか試作品を作ってみてください。

そして、各試作品の長所と短所を評価します。その成果を文書化しておいてください。これは、次に同じことを考慮する人のためでもあり、習熟曲線以上に作業を要した場合に自身で再検討するためでもあります。

このパターンに取り組めば、設計の要求により適合するようにプログラムの形（Shape of Program(9)）が洗練されるでしょう。プログラムの形（Shape of Program(9)）に戻り、プログラムの形を一般的なものから固有の形に変換する作業を続けることもできます。将来、初期段階においてシステムを適合させる方法に取り組むパターンをここから参照するでしょう。

## パターン19：イベントの伝送（Event Routing）

プログラムの形（Shape of Program(9)）を一般的なものから固有なものへと移行させ始め、システムの住人の役割（System Citizen's Role(10)）を考慮しているとき、イベント取得（Event Acquisition(18)）に加えてこのパターンも、ある設計上の問題を解決するのに有用です。

すべてのプラットフォームは、独自の方法でイベントを扱っています。「イベント」とは、マウスクリック、キーストローク等が発生したことを示すシグナルのことです。こうしたイベントが起きたときに、そのイベントが受け取られ、応答すべきすべてのオブジェクトがイベントに従って動作することを保証するためのさまざまなアプローチがあります。

**そこで：**イベントがどう消費されるかを示すために、プログラムの形を洗練します。次の設計上の問題について考慮すべきです。

- 同じイベントが起こったとき、各イベントは1つのオブジェクトだけに送られますか。それとも、同じイベントの発生通知を待ち受けるオブジェクトが複数ありそうですか。

- どうやって伝送しますか。それは、割込みハンドラのように固定的ですか。ネゴシエーション作業を行いますか。その場合、あるイベントが到着したとき、イベント消費オブジェクト（通常は、意思決定者かヒューマンインタフェースオブジェクト）がそのイベントを理解するかどうか、または、そのイベントに関心があるかどうかを答えます。アプリケーションにとって、レジストレーションは適切な設計ではないですか。（レジストレーションは、オブジェクトがイベントをディスパッチャに登録するという設計方針です。イベントが到着したとき、ディスパッチャがそのイベントを登録しているオブジェクトに、そのイベントを送ります。）

- 動作環境のイベント伝送経路を迂回しようと考えるのは、どんな状況ですか。

**提案**

イベントの伝送はできるだけ簡単にしておきます。イベントの動的レジストレーションとは、システムが運用される間、断続的に行われるレジストレーションのことです。これを使うと、コードを読んでもシステムが理解できなくなります。メンテナンスエンジニアは、イベントが断続的に登録、抹消されるところを、デバッガを使って監視しなければなりません。その場合、全てのありうる組み合わせを調べたかどうか、決してわかりません。

このパターンに取り組めば、設計の要求により適合するようにプログラムの形（Shape of Program(9)）が洗練されるでしょう。プログラムの形（Shape of Program(9)）に戻り、プログラムの形を一般的なものから固有の形に変換する作業を続けることもできます。将来、初期段階においてシステムを適合させる方法に取り組むパターンをここから参照します。

## パターン20：特別なインタフェースとしてのヒューマンインタフェースの役割（Human Interface Role Is a Special Interface Role）

プログラムの形（Shape of Program(9)）を一般的なものから固有なものへと移行させる上で、意思決定者オブジェクト(11)が人間から指示を得る方法が、普段よりも気になるでしょう。

ヒューマンインタフェースは、アプリケーションに情報と指示を提供する手段です。それは、アプリケーション中のところどころに見られます。ヒューマンインタフェースが無計画にアプリケーション全体に充満するのを許容してきたとしたら、あるプラットフォームから他のプラットフォームへのシステムを移植がどれほど困難か考慮すべきです。したがって、プログラムの形を洗練し、システムの重要な部分から分離したヒューマンインタフェースコンポーネントを配置する必要があります。

**そこで：**ヒューマンインタフェースに固有の設計判断を隠蔽する特殊なインタフェースオブジェクト(13)を作ります。人間の入力を必要とするのは、ほぼ間違いなく意思決定者オブジェクト(11)です。プログラムの形を洗練し、意思決定者オブジェクトからアクセスされるヒューマンインタフェースオブジェクトを配置します。ヒューマンインタフェースオブジェクトではヒューマンインタフェースやプラットフォームに固有の問題を扱い、より抽象的な意思決定や制御処理は意思決定者オブジェクトに任せます。

労働者オブジェクトやインタフェースオブジェクト(13)も、それらの下位の作業を補助するため、同様に特別なヒューマンインタフェースオブジェクトを要するかもしれません。これは、ダイアログボックス等の形になって現れるでしょう。また、伝達者オブジェクトにも特別なヒューマンインタフェースオブジェクトがいくつかついているかもしれません。伝達者オブジェクトについている各ヒューマンインタフェースオブジェクトは、それぞれ

伝達者オブジェクトのデータを異なる視点から見せます。

**提案**

ヒューマンインタフェースオブジェクトの振る舞いは、4 次元ヒューマンインタフェース見取図の細部に見られます。何をヒューマンインタフェースの役割を担うオブジェクトの責任とするか、かつ、何を意思決定者オブジェクトの責任とするかの判断には、慎重を期すべき場合があります。代替設計案をいくつか作り、同僚とそれらについて議論してください。他の誰かに設計を説明する上での簡潔さと簡単さという指針となる原則に則り、最善の設計を選択します。

このパターンは、設計者がヒューマンインタフェースの機能をプログラムの形（Shape of Program(9)）に追加する際に役立ちます。プログラムの形（Shape of Program(9)）に戻り、プログラムの形を一般的なものから固有の形へとさらに洗練してもよいです。ヒューマンインタフェースオブジェクトの内部設計を検討してもよいです。小さな集団によるシステム（Small Family Systems(15)）がこの内部設計に極めて適切であること、Model-View-Controller モデルが完成した小さな集団によるシステムであること、意思決定者オブジェクトがヒューマンインタフェースオブジェクトに Model に当たるものを伝達者オブジェクト(14)として渡すことがわかるでしょう。Model、Controller、View の実際の振る舞いは、このヒューマンインタフェースオブジェクトに隠蔽されます。イベント取得(18)およびイベント伝送(19)についてなされた設計判断にも従うことができます。

この節では、データの設計に取り掛かります。このパターンランゲージは、作成途上ですので、この節は、現時点で１つしかパターンがありません。この節は拡張する必要があります。しかし、この１つのパターンは重要なため、放っておきたくなかったのです。そのパターンは、次のものです。

21．起源を知るデータ（Data Knows Its Roots）

## パターン21：起源を知るデータ（Data Knows Its Roots）

協調作業パケットや伝達者の役割を担うオブジェクトといった他の情報を包含する「エンティティ」を識別したら、そうしたデータの起源を知る必要があるかどうかを判断できます。

優れたシステムは、かなりの期間に渡り運用されます。そして、時とともに、システムは変化します。アップグレード、機能の追加、新しい市場などがシステムを変化させる理

由です。新たなバージョンをリリースする際には、作業中のデータを、その状態に関わらず、新たなバージョンに収容することが重要です。これにより、古いバージョンからの透過的な移行を保証します。それと同時に、新たなバージョンを設計する際、拡張される機能により、作業パケットや伝達者オブジェクト等を自由に改良できなければなりません。

**そこで：**ある情報は、それを作ったプログラムの運用期間を超えて存在し続けます。また、並列実行スレッド間を移動する情報もあります。そのような情報を包含するすべての「エンティティ」が次のことをするように設計します。

- その型とバージョンを記録する
- そのエンティティを作り出したプログラムのバージョン、作成時刻、アクセス可能な環境についてのすべての情報（マシンのタイプ、マシンの構成、OS のバージョン、ファイルシステムのバージョン等）を記録する
- 最新のアクセス時刻と最終変更時刻を記録する
- 訪れたすべての並行実行スレッドについて、そのスレッドのバージョンと最新訪問時刻を記録する
- この情報にアクセスするすべてのアプリケーションについて、そのアプリケーションのバージョンと最新訪問時刻を記録する。
- この情報を蓄積するところに、蓄積されるデータ構造の形式に関する知識を埋め込む。

**提案**

大きなエンティティにとって、この付加的な「起源」情報は、時間的にも空間的にもほとんどオーバヘッドになりません。非常に小さなエンティティにとっては、そのオーバヘッドが重大事となるかもしれません。それでも少数であれば、この情報は有効です。小さくて数の多いエンティティについては、この設計方針を捨てる前に、創造的になりましょう。この情報を記録するトランザクションログの設計を検討してください。そして、処理の少ない時間帯に情報を圧縮しておき、処理のピーク時を救うのです。

このパターンの適用が完了したら、システムが発展する際に必要となる情報を永続的なオブジェクトが保持していることが保証されます。同時に、作業中のデータを、その状態に関わらず、新たなバージョンに収容できることが保証されます。

## ３．Caterpillar's Fate の使い方

読者は、Caterpillar's Fate を使えば再作業やイテレーションを経ずに移行が可能になると思われるかもしれません。それは大きな間違いです。分析から設計への移行において、分析ドキュメントの重大な抜けや矛盾が見つかることもあります。設計が出来上がってく

るにつれて、ある設計項目が全く考慮外になっていることが明らかになることもあります。Caterpillar's Fate を使うと、さまざまなイテレーションを体験することになります。これは、方法論、パターン言語、プロジェクトのいずれの不備によるものでもありません。大規模システムの構築には、システムが何をするものかを理解することが一部を占めるのです。多くの場合、理解はイテレーションを通じて行われます。欠陥を作り出すもの、それは、イテレーション、再分析、再設計の必要性の無理解です。

## ４．Caterpillar's Fate の状態

本ドキュメントには、筆者の気づくところとなり、また、現在使っているパターンのおよそ 3 分の 1 を収めました。残りについては、ページ数の問題から省きました。「気づくところとなり」という回りくどい言い方をしたのは、顧客と筆者が Caterpillar's Fate を使うたびに、より多くの一般的な設計の知恵が見つかるからです。さらに、使うことによって得られた知見から、既に書いたパターンについて再作業することがあります。そのため、Caterpillar's Fate は作成途上のパターンと捉えられます。

それでも、この作業は、高度に成熟しています。筆者の顧客は、実使用されるシステムを構築するのに Caterpillar's Fate を 1992 年から使っています。それらはたとえば、投資銀行のシステム、対話型音声反応システム、ハンドヘルドコンピュータのアプリケーション、電力制御システムといったものです。チームの人数は、3 人から 20 人でした。いつもは、パターン言語を方法論のコースとして伝え、必要に応じて個人的に指導しました。これらの概念を書かれたものとしてだけ伝えようとしたのは、本論文が初めてです。

## ５．問題

本パターン言語を文章にしてみて、システム分析から初期の設計に変換するときに、すばやくこれらのパターンについて考慮するようになったことに驚きました。このことから、筆者が手に入れた知恵を誰かが使えるようになるまで、この言語を習得するのにどれだけの労力が必要か、という筆者の関心事にたどり着きました。これは、プログラム開発に関するパターン言語を探求するための新たな実験の一部です。パターン言語は著者以外の人にも、精通した先生の助けなしで使えるでしょうか。これが可能であることがわかれば、パターン言語として書いておく意味があります。そのためにこの論文を書きました。これは、Caterpillar's Fate のようなパターン言語が広く敷衍されうるかどうかを調べるためのテストベッドとして機能します。

Caterpillar's Fate について、筆者のもう一つの関心事は、それが設計パラダイムとしてオブジェクトに依存していることです。これは、パターン言語がそれを実行するにあたり使う素材とは無縁であるという Alexander の意見と相反します。彼に従い、筆者は、オブ

ジェクトと無縁な分析から設計戦略のいくつかへと変態させられるかを知りたいと思います。Caterpillar's Fate に収めたパターンには、並列実行スレッド（Concurrent Threads of Execution(1)）のように、オブジェクトのバイアスなしに表現されているものも、対話により達成される作業（Work Accomplished Through Dialog(16)）のように、くっきりとオブジェクトのバイアスのかかっているものもあります。筆者には、獲得された知恵のバイアスから解き放たれた表現が謎のままです。

また、筆者を含めすべてのパターン言語の著者が解決すべきもう一つのことは、他の著者の作品を尊重しながら組み入れる方法です。あるパターン言語を解体して、その一部分を Caterpillar's Fate に組み込みたいのです。原著者の作品を尊重しつつこれを行うことは可能でしょうか。アイデアにあたること以外はすべてリライトしたくなるかもしれません。それにより筆者が意図した通りに筆者の考えが表現されないかもしれません。コミュニティとして、これおよびこれに関連する事柄を議論する必要があります。

最後に、公の攻撃への恐れにどう対処するかという問題を挙げます。Caterpillar's Fate は、普段、筆者がどのようにプログラムを設計するかについて詳細に述べています。「筆者ならこうする」と明言するのには危険が伴います。筆者は、Caterpillar's Fate が、筆者の同僚に見せるのが恥ずかしくなるようなプログラムを生み出す可能性があることを確信しています。筆者は、適切でないと思ったときには、このパターンに則らない権利を主張します。これは、筆者が現時点で設計に用いている方法です。筆者は、新たにさまざまな設計方法を学び、その結果、新たな版の Caterpillar's Fate を作成する権利を主張します。

## 参考文献


[1] Kerth, N. "A Structured Approach to Object-Oriented Design." Addendum to the OOPSLA Proceedings, 1991.

[2] Kerth, N., Rhodes, R., and Burley, J. "How to Deliver 20,000 Lines of Code with Only Four Defects for under $2.00 Per Line of Code." Invited paper, Pacific Northwest Software Quality Conference, Fall 1990.

[3] Alexander, C. The Timeless Way of Building. New York: Oxford University Press, 1979.

[4] Alexander, C., et al. A Pattern Language. New York: Oxford University Press, 1977.



Norman L. Kerth の連絡先

Elite Systems, P. O. Box 2205, Beaverton, OR 97075;

72073.3222@compuserve.com.