原文："Designing Component-Based Frameworks Using Patterns in the UML"
Communication of the ACM, Vol.42, No.10, pp.38-45（Oct. 1999）より許可を得て翻訳

UMLパターンを使用した コンポーネントベースの フレームワーク設計

フレームワーク・ソリューションは，再利用可能で使いやすい
ソフトウェア・アーキテクチャの構築に役立つ.

Grant Larsen　glarsen@blueprint-technologies.com
翻訳：安藤　進　sando@twics.com

入り口を抜けると，ある建築物が目にとまった．タージマハル（インドにある霊廟）だ．遠目には対称的な線だけの単純なものに見えたが，その回りの広場や小道をたどると，簡単に近寄れる．間近で見ると，表面に深い彫りや微細な装飾があり，たくさんの宝石が周囲の光を受けて輝く様子に深い感銘を受けた．周囲の状況の変化に見事に適応しているのだ.

現在私たちは，このような建築が備えている性質をどんどんソフトウェアに取り込みたいと考えている．一定の規則に従った美しい線と境界をソフトウェアのアーキテクチャにも反映させたい．既存のインタフェースを再利用しながらも，簡単に使えるものが望ましい．さらに，環境や実装の変化に適応できる能力があれば，今日各企業が求めている多くの要望に対応できるアーキテクチャが実現できるだろう.

コンポーネントや実績のある既存のソリューションを使ってシステムを設計すれば，開発者は抽象度の高いレベルで仕事ができる．このような設計手法によってもたらされる2つの効果は，生産性と品質の向上である．生産性を上げるには，分析，設計，開発の抽象度を上げればよい．品質を上げるには，すでによく知られており実績のあるソリューションとソリューションを実装するコンポーネントを再利用すればよい.

業界のニーズに対応する自動化手法には，a）ターンキー・ソリューション，b）フレームワーク・ソリューション，c）カスタム開発ソリューションなどがある．ここでは，フレームワーク・ソリューションに重点を置いて述べる.

実績のあるソリューション（パターン）を使ってコンポーネントベースのフレームワークを設計するためのUMLの使用法について述べる．まず，パターン，フレームワーク，コンポーネントのそれぞれの関係について簡単に説明する．次に，一連のモデルと，フレームワークを作成し拡張するコードを紹介する.

生産性を向上させるパターン

パターンやフレームワークなど実績のあるソリューションを使ってシステムを設計すれば，ソフトウェア開発者の生産性を向上できる．生産性が向上し成果物がよくなれば，企業や組織の関係者にも直接的間接的によい影響が出る．ここでいう関係者には，ソフトウェア・アーキテクト，プログラム開発者，ITマネージャ，事業管理者だけではなく，顧客も含まれる.

ソフトウェア・パターンは，建築家であるChristopher Alexanderの影響を色濃く受けている．Christopher Alexanderはパターンに固有の性質について言及することが多いが，特にその性質に名前をつけていない．だが名前がなくても，パターンが，役立ち，分かりやすく，お互いに結びついており，活用できることは間違いない.

ソフトウェア業界で成果物とパターンの再利用可能性をはじめて明確に提示した専門家が何人かいる．主としてヒューマンインタフェースにかかわる初期のパターンについては，Ward CunninghamとKent Beckによる1987年の研究がある．その数年後に，OOPSLAなどの会議のディスカッションに登場したErich GammaとRichard

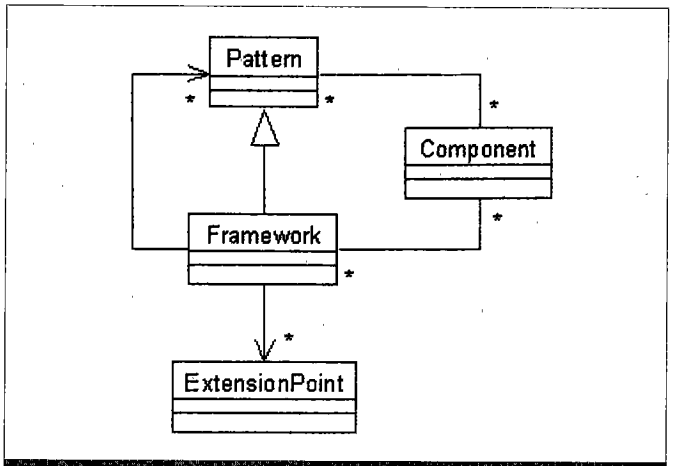

図-1 パターン，コンポーネント，フレームワークの関係

Helmがいる．1990年代の初頭には，いわゆる「4人衆(Gang of Four)」が各自の体験を明確化しまとめる共同作業を開始した．そのほかには，Desmond D'Souza, Norm Kerth, Peter Coad, Bruce Andersonらがいる．

パターンとフレームワークを使用すると，実績のあるソリューションの再利用が可能になり，開発者はソリューションについてより抽象度の高いレベルで議論したり開発したりすることができる．パターンやフレームワークをパッケージ化したものがコンポーネントである．したがって，コンポーネントを使用すれば，コンテンツ，パターン，フレームワークの再利用が可能になる．

Grady Boochはコンポーネントを「一連のインタフェースに準拠し，実現しようとしているシステムの一部であり，交換可能な物理的部分である」[1]と定義している．メタデータをコンポーネントに付加すれば，ほかの開発者によるコンポーネントの再利用がさらに簡単で容易になる．このメタデータには，「コンポーネントの利点は何か，コンポーネントの欠点は何か，コンポーネントが依存するものは何か」といった質問に対する解答が含まれる．

OMGに提出されたUMLの最初の文書では「パターンは設計パターンの構造を記述したテンプレート・コラボレーション（オブジェクト間の協調的振舞い）と同義である．設計パターンには，テンプレート適用のヒントや長所，短所など，構造化できないものが多い．そのようなものはUMLでモデル化せずに，テキストや表の形で表現するとよい」[5]と書いてある．簡単にいうと，繰り返し起こる問題の解決法を記述したものがパターンである．

フレームワークはパターンと密接な関係があり，UMLではステレオタイプとして表現される．Grady Boochはフレームワークを「特定のドメインのアプリケーションを開発するための拡張可能なテンプレートを提供するアーキテクチャ・パターン」[1]と定義している．図-1に示す概念モデルでは，フレームワークという名称と位置づけを与えることで，正式に市民権を認めた．

図-1のモデルは，フレームワークも繰り返し発生する問題の解となるが，単にそれだけにはとどまらない．フレームワークは拡張点（extension point）の集合であり，これらの拡張点を利用してフレームワークの振舞いを拡張[校閲者注1]できる．フレームワークは，自分自身の中にいくつでもパターンを取り込める．

拡張点は，別な言い方をすると，「フレームワークを特定のコンテキストに適合させるために調整しなければならないスロット，ノブ，ダイヤル」[1]に相当する．つまり，拡張点は，フレームワークを拡張（カスタマイズ）する場所と方法を記述したものである．拡張点については，Observable Party Accountフレームワークの項で説明する．

パターンを適用するには，通常，そのパターンの役割／参加者を，その役割を担う実際のモデル要素にマッピングする．1つのモデル要素は，ある時点で複数の役割を演じる参加者になれる．フレームワークの適用方法も同様であるが，フレームワークを拡張（修正）して元々の振舞いを変更できる．図-1に示すように，1つのコンポーネントは複数のパターンやフレームワークを使える．1つのパターンまたはフレームワークは，複数のコンポーネントを利用できる．

アーキテクチャ・パターンとフレームワークは，ソフトウェア・アーキテクチャを記述したものである．フレームワークは，通常，ソフトウェアを構築するときにアーキテクチャ上の主要な意思決定ポイントを提供してくれるものである．UMLでは，ソフトウェア・アーキテクチャを分析する視点がいくつかあり，ソフトウェア・アーキテクチャの5つのビュー（view）と呼ばれている．ここでは，Observable Party Accountフレームワークに対するアーキテクチャの論理ビューだけについて示す．

フレームワークの性質は水平特性と垂直特性の両方を含むので，両者合わせてhertical（horizontal+vertical）と呼ばれることもある．水平特性はフレームワークが複数のコンテキストにまたがって使えることであり，垂直特性はフレームワークがそれぞれの業界のドメイン知識を反映していることである．

システム開発では，システムの要素を拡張可能なフレームワークとして作成できないものかと常に考えている．第1はフレームワークを作成する作業であり，第2はフレームワークをパッケージ化する作業である．しかし，適切なフレームワークを適切な方法で選択することも重要な作業である．拡張可能なフレームワークの設計，構築，パッケージ化をするには，開発時間とコストが余計にかかる．したがって，事前に生産性の向上や再利用が可能なども考慮に入れなければならない．

フレームワークの設計と実装には，次のような作業がある．

1. フレームワークを適用するドメインを特定する．
2. フレームワークでサポートする主要なユースケースを決定する．
3. フレームワークと相互にやり取りするアクターの集合を洗い出す．
4. フレームワーク開発に役立つ既存のパターンや実績の

校閲者注1 ここで「拡張」とは，クラスの継承を用いてメソッドを変更することなどをいう．

あるソリューションを洗い出す.

5. フレームワークの主要なインタフェースとコンポーネントを設計し，役割とアクターをインタフェースにマッピングする.
6. フレームワークにインタフェースのデフォルトを実装する.
7. フレームワークの拡張点を文書でまとめる.
8. フレームワークのテストケースとプランを作成する.

上記の準備作業を行うと，いくつかの成果物が得られる．フレームワークをパッケージ化し，ほかの開発者にも使えるようにするために役立つ成果物の一部を以下に示す.

a) アーキテクチャ文書
b) フレームワークを実装するコンポーネント
c) 拡張点の定義
d) フレームワーク特性
e) フレームワーク・コード
f) フレームワーク品質基準
g) 要件文書とデータベース
h) モデル
i) スナップショット
j) テストケース
k) テストデータとテストドライバ

ここでは個々の成果物についてあまり細かい話はしない．成果物によってそれぞれ重要度は異なるが，開発者にフレームワークを使用してもらい，一貫性，品質，利用しやすさを保証するのが目的であることに注意してほしい．これは再利用という点で最も重要なことだ．従来は「再利用可能な」項目の選択，理解，適用が容易でないという問題があった．再利用可能なものを，使いやすく，予測可能で，分かりやすいものにすれば，フレームワーク，コンポーネント，パターンの再利用も拡大する.

フレームワークの設計と構築

既存のパターンを利用してコンポーネントベースのフレームワークを設計する目的でUMLを使用するプロセスを図で示す．フレームワークには多くのパターンが実装されているので，Gammaら，Hay, Fowler[2)〜4)]などさまざまな著者からパターンを集めて，それらを相互運用が可能なコンポーネントとしてモデル化し，Observable Party Accountフレームワークを構築した.

ここではモデルの一部を紹介するにとどめ，そのほかの成果物には触れていない．しかし，フレームワークの設計と構築に関する主要な作業は示してある.

Observable Party Accountは，さまざまな業務要素と企業会計に対して会計取引を計上し処理する要素を管理するフレームワークである．このフレームワークは，基本的なビジネスオブジェクト間の関係を示すものであり，拡張やカスタマイズが可能な仕組みでもある．このフレームワークをインスタンス化することで，料金請求システムやそのほかの関連システムの主要な要素を作成することができる．これで**フレームワークを適用するドメインを特定する**作業1ができたことになる.

このフレームワークに関連したすべてのユースケースについて説明するつもりはない．ここでは，次のユースケースのなかで「happy day」というシナリオを取り上げる.

- 勘定残高の検索
- 勘定操作の監視

これで**フレームワークでサポートする主要なユースケースを決定する**作業2ができた.

ユースケースに影響を与えるアクターは次のとおりである.

- 顧　客
- 顧客サービス係

これで**フレームワークと相互にやり取りするアクターの集合を洗い出す**作業3ができた.

ここで，David HayからPartyパターン[4)]の要素を，Martin FowlerからAccount[2)]パターンを，GammaらからObserver[3)]パターンをそれぞれ借用させてもらう．これらのパターンの静的構造は，インタフェースとそれらを実装したコンポーネントで示す．これらのコンポーネントと定義済みの拡張点でフレームワークを構築する．紙幅の関係でフレームワークの動的なセマンティクスは割愛してあるが，このビューはフレームワーク開発で非常に重要なものである.

これで**フレームワーク開発に役立つ既存のパターンや実績のあるソリューションを洗い出す**作業4ができた.

David HayのPartyパターンを簡略化すると，Party, Organization, Personの3つのクラスで構成される．OrganizationとPersonはPartyを継承する．Organizationは複数のPersonと関連（association）を持つ．パターンを構成するインタフェースとそれを実装したクラスを**図-2a**に示す．ここで注目すべき関係はUML実装関係であり，これは〈〈Interface〉〉を指し示すすべての関係のことである．Party, Organization, Personは，それぞれのインタフェースを実現しているものである．IPersonとIOrganizationはIPartyを継承する．これらのインタフェースを選択したのは，任意の時点で多くのエンティティが果たす主要な役割を，それぞれのインタフェースが表現しているからである.

これで**フレームワークの主要なインタフェースとコンポーネントを設計し，役割とアクターをインタフェースにマッピングする**作業5ができた．ここで紹介した事例では，各コンポーネントごとにこの作業を繰り返す.

このパターンの実装例を**図-5a**に示す．Partiesコンポーネントは，これらのインタフェースを実装したクラスで構成される．このコンポーネントは，（ほかのパターンを実装している）そのほかのコンポーネントと協調して，フレームワークの基本機能を実現する.

FowlerのAccountパターンを修正した基本的なクラスと関係を**図-2b**に示す．Accountクラスは常にAccountTypeの1つであり，どの時点においても一連の入金／引出しの残高勘定を保持する.

図-2bのパターンは，図-5aのAccountsコンポーネント

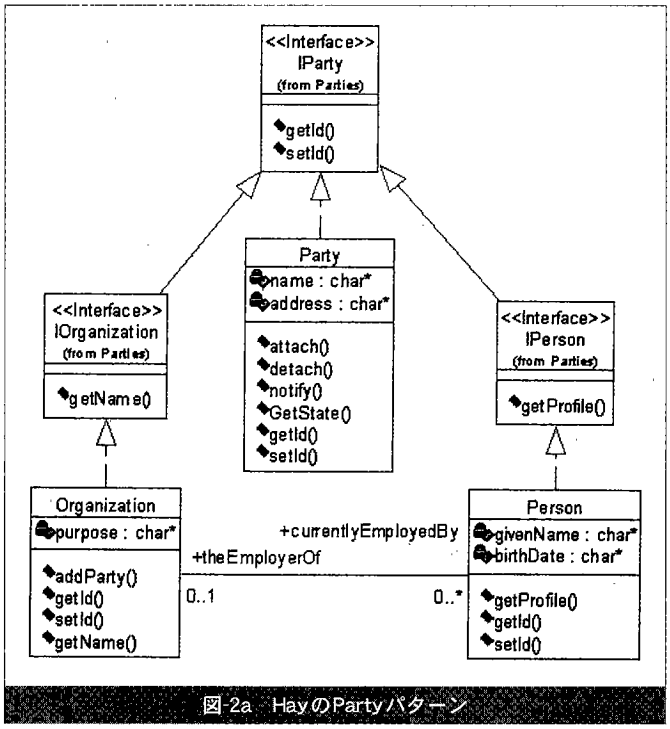

図-2a　HayのPartyパターン

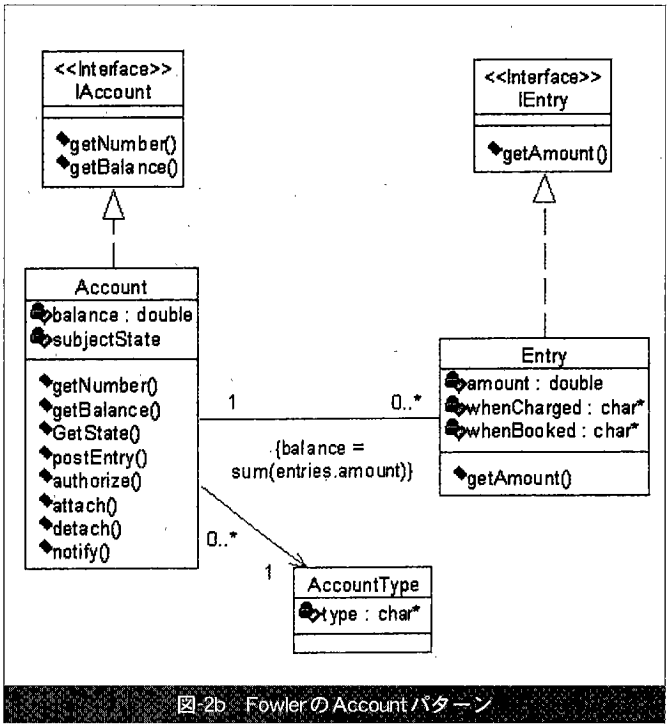

図-2b　FowlerのAccountパターン

として実装されている．Accountsコンポーネントは，そのほかのコンポーネント（ほかのパターンを実装する）と協調して，フレームワークの基本機能を実現する．

もう1つの要件は，Accountオブジェクトの動きを監視する機能をフレームワークで実現する必要があることだ．そこで私は，GammaらのObserverパターンを使用して，どのクラスを監視するのか，だれが監視をするのかを宣言した．GammaらのObserverパターンの参加者と役割をフレームワークの既存のクラスにマッピングし，このフレームワークを構成する一部のクラスで図-3のIObservableとIOserverインタフェースを実装した．IObservableインタフェースを実装するクラスは，それらのインタフェースに関連しているIOserverインタフェー

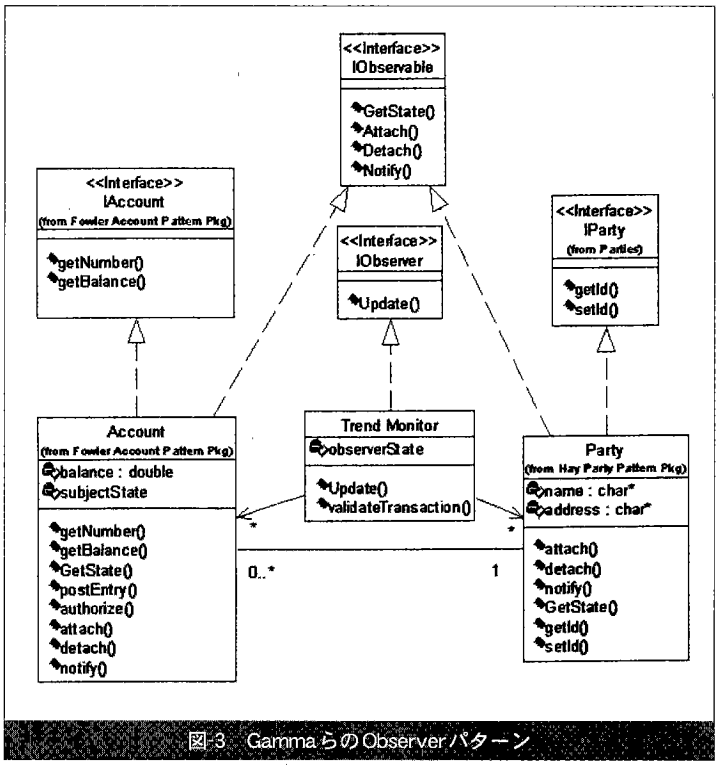

図-3　GammaらのObserverパターン

スを使って通知や更新を送ることができる．

図-3で，Trend MonitorクラスはConcrete Observerの役割を，AccountクラスとPartyクラスはGammaらのObserverパターンから具象Subjectクラスの役割をそれぞれ演じる．ここで私は，AccountクラスとPartyクラスでIObservableインタフェースを実装することにした．もっとも，IAccountとIPartyの各インタフェースからIObservableインタフェースを継承させることも可能なのだが，関連性を弱くしたほうがよいと考えており，「監視可能（observable）」であることがAccountクラスとPartyクラス，そしてIAccountとIPartyの各インタフェースを実装したほかのクラスのセマンティクスに密接に関連づけるほうがよいというほどの確信がなかったからである．

この決定はプログラムとして実装する上で少なからぬ影響を与える．たとえば，実装クラスでキーワードを余分に書く必要がある．

```
public class Account implements IAccount, IObservable
{
...
}
```

しかし，この決定は，ビジネス・オブジェクトのインタフェースに固有の部分に監視可能（observable）セマンティクスを強要しないので，柔軟性と潜在的な再利用可能性が増す．私はこのパターンを専用のコンポーネントに実装するのではなく，既存のコンポーネントのクラスとインタフェースがパターンのさまざまな役割を果たせるようにした．

フレームワークを構成するすべてのインタフェースとクラスを図-4に示す．この図の中ほどにある太線の上側にあ

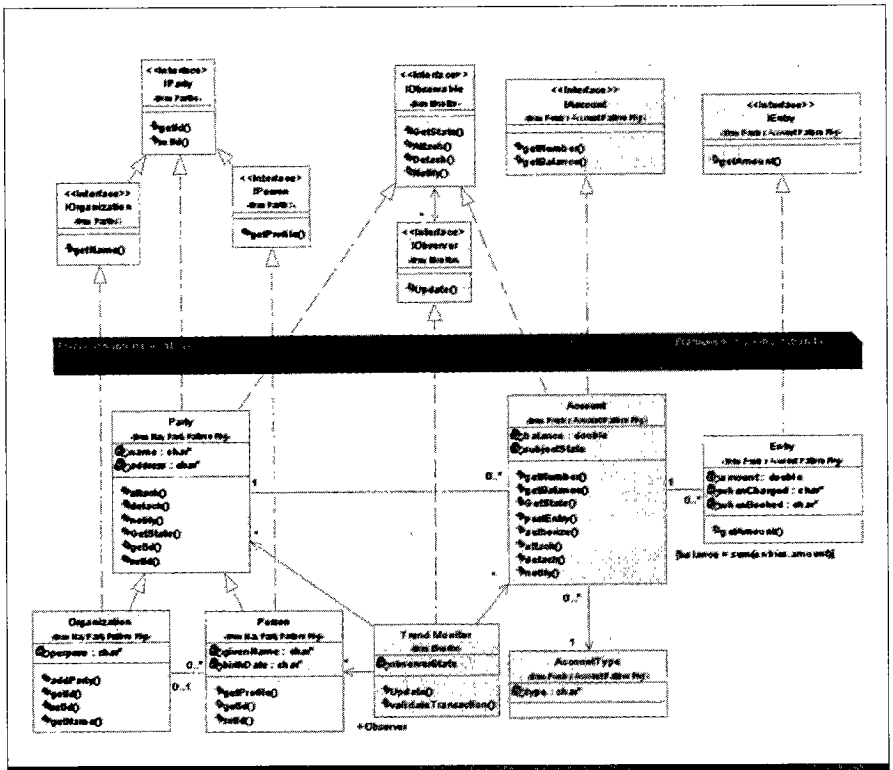

図-4　フレームワークのインタフェースとクラス

るのがコンポーネント・インタフェースである．太線の下側にあるのがこれらのインタフェースを実装したものであり，フレームワークの基本的な振舞いを示している．

これで**フレームワークにインタフェースのデフォルトを実装する**作業6ができた．

インタフェース仕様と実装なしのコンポーネントだけでフレームワークを提供することも可能だが，フレームワークのクライアントが特別な実装要件を持つ場合に限られるだろう．フレームワークは，通常，基本的な実装を持つと考えてよい．

フレームワークの振舞いを拡張するには，次の手法がある．

- フレームワークにインタフェースを直接実装する．これは提供されている実装を使わないという意味である．
- フレームワークのクラスから継承によって振舞いを拡張する．
- 1つのフレームワーク・クラスを集約し，その振舞いを拡張する．

インタフェースを使ってフレームワークを拡張する

フレームワークの標準の振舞いを無効にするには，独自の実装を作成すればよい．このフレームワークをSavingsAccountクラスで拡張する手法を以下に示す．

```
public class SavingsAccount implements IAccount
{
...
}
```

フレームワークを継承によって拡張する

フレームワークを拡張しカスタマイズする最も簡便な手法である．このフレームワークにSavingsAccountクラスを新規に参加させるには，フレームワークのAccountクラスを継承させればよい．この手法でフレームワークを拡張する例を以下に示す．

```
public class SavingsAccount extends Account
{
...
}
```

フレームワークを集約によって拡張する

フレームワークを拡張する手法の中では，最も関連性の低いものである．委譲（delegation）とコンポジッション（composition）を使ってフレームワークのクラスの1つを別のクラスに包含させることで，フレームワークを拡張する．たとえば，フレームワークのAccountクラスのインスタンスをSavingsAccountクラスに包含させる．ただし，イベント・ループやフレームワークのそのほかのインタフェースへは参加できなくなる可能性がある．この手法でフレームワークを拡張するには，次のようにすればよい．

```
import Account;
public class SavingsAccount
{
   private:
        Account acct = null;
}
```

フレームワークを拡張する方法は，フレームワークが提供される形態によって異なる．たとえば，フレームワークがブラックボックス（ソースコードがなくバイナリコードだけ）として提供された場合，フレームワークを拡張する方法はインタフェースの実装に限定される．一方，フレームワークがホワイトボックス（ソースコードがある）として提供された場合は，継承や委譲などの手法を使ってソースコードレベルの拡張が可能である．

これで**フレームワークの拡張点を文書でまとめる**作業7ができた．なお，説明の煩雑さを防ぐためにテストケースの作業8は割愛する．

フレームワークのコンポーネントとインタフェースを図-5aに示す．Trend Monitorは，PartiesとAccountsの各コンポーネントのアクティビティを監視するコンポーネントである．Accountsは，特定の当事者（party）の勘定残高（account balance）を操作できるコンポーネントである．先ほど紹介したいくつかのパターンを実装したコ

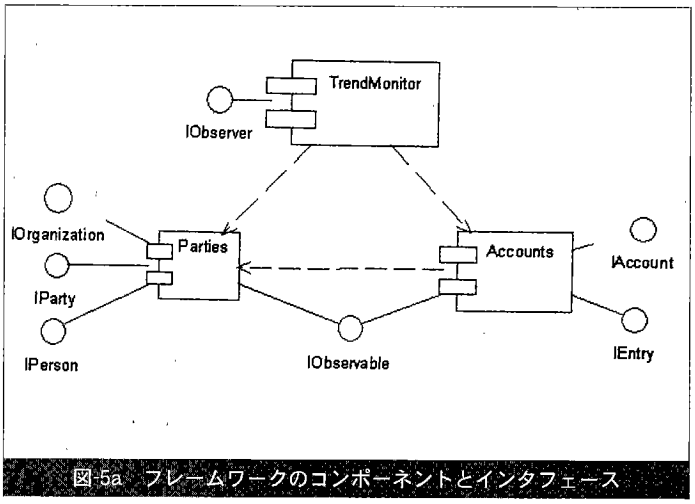

図-5a フレームワークのコンポーネントとインタフェース

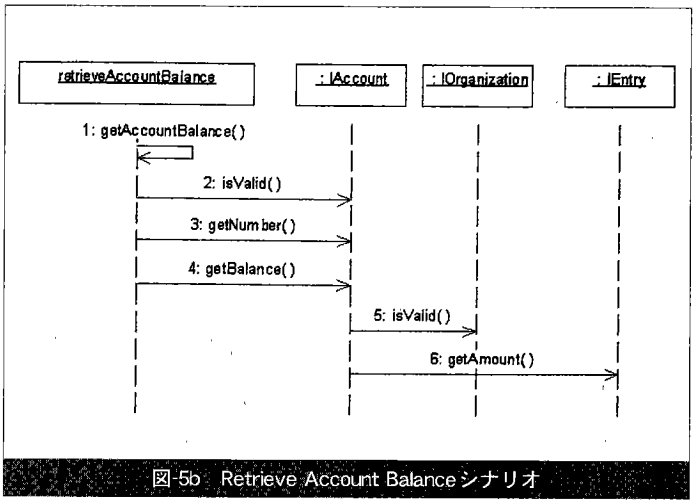

図-5b Retrieve Account Balanceシナリオ

ンポーネント同士が協調することで，フレームワークのユースケースとシナリオを実現する．Retrieve Account Balanceシナリオを図-5bに示す．図-5bには，勘定残高を検索するコンポーネントのインタフェースを時系列で示してある．

フレームワークの利用

これまでフレームワークの設計，構築，拡張について述べてきた．これからフレームワークの利用へ話を進める．特定のドメインに適用するために，継承によってフレームワークを拡張する形態を使用する．ここで，企業の会計業務に基づいてマーケティング資料を作成する料金請求システムについて考えてみよう．請求書を作成するたびにマーケティング資料も作成したい．そのためには，Observable Party Accountフレームワークを拡張し，CommercialOrgとLiteratureMonitorという2つのクラスを新規に作成する．

```
class CommercialOrg extends Organization
{
//Partyから派生したクラス（フレームワークの一部に
//なっている）のAccountsを評価するときに，それぞ
//れのAccountのgetBalance()メソッドを呼び出す．
//たとえば，勘定残高が5000ドル超過していた場合，
//関連する監視Observerオブジェクトのnotify()メソ
//ッドを呼び出す．
...
}

class LiteratureMonitor extends TrendMonitor
{
//このクラスのupdate()メソッドを実装して，指定され
//たCommercialOrgのAccountアクティビティのレ
//ルに基づいてマーケティング資料を生成する．
...
}
```

上記は簡単な拡張点であるが，フレームワークの2つの拡張点を使用することで，既存のフレームワークを再利用しながら，特定のアプリケーションに新しい機能を追加できる．ここで紹介したObservable Party Accountフレームワークをみれば，再利用可能なパターンを適用しUMLでモデル化することで，コンポーネントベースのフレームワークを設計し構築する手法の一端を理解してもらえたと思う．

まとめ

冒頭でタージマハルの話を紹介したが，近づいて表面をよく見ると，宝石が埋め込まれていることに気がついた．近づくほどに奥が深く，遠方から見ると対称性の美しさと細部を抽象化する能力に驚嘆した．その振舞いは，光，時刻，天候に自然に適応するのだった．

UMLを使ったフレームワークとコンポーネントのモデリングも，タージマハルと同じ効果を演出する．鳥瞰的に眺めることもできるし，精細に捉えることもできる．モデリングは面倒な仕事かもしれないが，UMLが複雑度を抑え，再利用可能でカスタマイズ可能なフレームワークの設計に役立つことは確かだ．

ここで紹介した技法でシステム設計に取り組めば，現場における再利用の可能性が広がり，生産性や投資利益率（ROI）も向上し，多くの関係者にも喜ばれるはずだ．

謝辞　日立教育部の田口昭二氏には拙訳に目を通していただき貴重な示唆を受けた．校閲者の青山幹雄先生には不適切な表現や誤訳を指摘していただいた．著者のGrant Larsen氏には訳者の疑問点についてEメールで教えていただいた．各氏に感謝申し上げる．

参考文献
1) Booch, G.: The Unified Modeling Language User Guide, Addison Wesley, Reading, MA (1998).
2) Fowler, M.: Analysis Patterns: Reusable Object Models, Addison Wesley, Reading, MA (1996).
3) Gamma et al.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, Reading, MA (1995).
4) Hay, D.: Data Model Patterns: Conventions of Thought, Dorset House, NY (1995).
5) Rational Software and UML Partners: UML Semantics and Appendices (Nov. 19, 1997).

(平成12年1月28日受付)