修 士 論 文

開発プロジェクトに応じて拡張可能な版管理システム構成法

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

早坂 良

2003 年 3 月

修 士 論 文

開発プロジェクトに応じて拡張可能な版管理システム構成法

指導教官　落水 浩一郎 教授

審査委員主査　落水 浩一郎 教授
審査委員　篠田 陽一 教授
審査委員　権藤 克彦 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

010089 早坂 良

提出年月: 2003 年 2 月

概 要

ソフトウェア開発プロジェクトが版管理システムに要求するプロセス支援機能は, 開発プロジェクトによりさまざまである. この機能要求に応えるためには, 機能拡張性が版管理システムに求められる. 既存の版管理システムの機能拡張アプローチでは, あらゆる開発プロジェクトの機能要求に応えることは難しい.

本論文では, 開発プロジェクトの要求するプロセス支援機能を必要に応じて容易に追加できる, 拡張可能な版管理システムの構成法を提案する. システムの機能拡張性の実現のために, オブジェクト指向アプリケーション・フレームワークを利用したレイヤアーキテクチャを用いる. 各レイヤの提供するインタフェースを組み合わせることにより必要な拡張機能を実装できる. その際, フレームワークのホットスポット部分の実装を行うが, サブクラス化やオブジェクトコンポジションなどの差分プログラミングにより, 低いプログラミングコストで記述できる.

また, CVS (Concurrent Versions System) を対象としたプロトタイプシステムの設計 / 実装法を示す. さらに, 2 種類のアクセス権管理機能をオブジェクト指向フレームワークの拡張機能として設計 / 実装し, 本システムの拡張容易性を評価する.

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　背景

版管理とは, 同一の識別子に対し 2 つのバージョンが付けられたときに生じる混乱を避けるために, 変更が起こったら新しくユニークな識別子を発行することである. 識別子の個数が増大した場合でも, それらの間の関係や共通のプロパティを記録し, 管理することが要求される [12].

ソフトウェア開発プロジェクトは, 普通, 開発の過程で生成された中間成果物 (ソースコード, 各種ドキュメント, 自動ビルドを支援するスクリプトなど) のバージョンを管理するために, 版管理システムを用いている. これを利用することにより, 開発者は一つの中間成果物について行われた変更を自分で管理する必要がなくなり, 特定のバージョンを指定すればいつでもそのバージョンの中間成果物を取り出すことができる. さらに, 二つのバージョンを指定することによりバージョン間の比較が行えるようになる. ある中間成果物にバグが発見された場合, そのバグが開発のどの時点のどのような変更により発生したのかを追跡できる. また, バージョンの変更履歴を追うことにより, プロジェクトの進捗状況も把握できるようになる. このように, ソフトウェア開発プロジェクトが版管理システムを使うことには多くの利点がある. 現在広く使われているオープンソース [26] ソフトウェアの版管理システムには, SCCS [31], RCS [33], PRCS [22], CVS [4] などがある.

1.2　問題点

一般に, ソフトウェア開発プロジェクトは独自の開発プロセスを持っている. 開発プロセスは, ソフトウェア開発における一連の活動のことであり, 高品質のソフトウェアを短期間で開発すべく適用され実行される. その選考にあたっては, プロジェクトの規模, コスト, 納期, 開発しようとしているソフトウェアの種類, 開発者の能力, チーム形態, 版管理システムの種類, プロジェクトの開発ポリシーなどのさまざまな要因が考慮される.

たとえば, ソースコードを公開し世界中の開発者が参加して開発を進めていくオープンソースソフトウェア開発プロジェクトの採用する開発プロセスは, 組織構造, 権限の構造, リポジトリへのアクセス方法, 変更要求の処理方法, 版管理システムの使い方, プロジェクトの運営ポリシーなどの点で大きく異なる.

現状では, 版管理システムにはプロセス支援機能がないため, これらのほとんどはプロジェクト管理者によって手作業で行われており, プロジェクトの規模が大きい場合や版管

理システムで管理される中間成果物が増大した場合, プロジェクト管理者の負荷が大きくなってしまいプロジェクトの運営とソフトウェア開発に支障をきたしてしまう.

いくつかの版管理システムは, 拡張性を提供する機能またはカスタマイズ機能を持っている. しかし, それらの機能は汎用性に欠けており, あらかじめ想定された機能しか追加できないし, カスタマイズ性も低い.

版管理システムのソースコード自体に変更を加えてプロセス支援機能を実現する方法もある. しかし, この方法では実現にかかるプログラミングコストが問題になる. 一般に設計段階に再利用性や拡張性が考えられていないソフトウェアのソースコードを変更するには, 困難を伴うことが多い. なぜならば, モジュール (関数など) 間の依存性が強く, 一部分に変更を加えるとその影響が広範囲に及ぶことが多く, 簡単には変更を加えることができないからである. したがって, 既存の版管理システムにおいてプロセス支援機能を実現することは, 非常に困難である.

まとめると, 開発プロジェクトが既存の版管理システムを使用する上で, 以下の問題がある:

- **プロセス支援機能が不足している.**

  版管理システムに開発プロジェクトの開発プロセスに対応したプロセス支援機能があると, より効率的に開発プロセスを実行できる. たとえば, ソースコードを公開しユーザからのフィードバックを受けて開発が進行するオープンソースソフトウェア開発では, ユーザからの変更要求をいかに取り込むかが重要になる. 変更要求は開発者がレビューを行い, リポジトリへコミットされる. その際, 開発者に対するリポジトリへのアクセス権限の授与や剥奪を版管理システムが支援すると, より効率的に開発を行えるようになる.

- **プロセス支援機能を拡張機能として追加する汎用的なメカニズムが存在しない.**

  版管理システムは, ある程度の拡張性やカスタマイズ性を提供してはいるが, 十分ではない. たとえば, CVS にはあるイベントが起こった時にスクリプトを呼び出す機能がある. しかし, そのスクリプトが呼び出されるタイミングは固定しているし, スクリプトの内部からはごく限られた版管理システムの情報しかアクセスできない.

- **開発プロジェクトの要求に応じたプロセス支援機能を容易に組み込めない.**

  プロジェクトの目的は, 版管理システムを拡張することではなく, ソフトウェアを開発することである. そのため, 少ないプログラミングコストで拡張機能を実現できる必要がある. たとえば, 既存の版管理システムでプロセス支援機能を実現しようと考えると, ほとんどの場合スクラッチからのプログラミングを要求されてしまう. さらに, プロセス支援機能は開発プロジェクトに良く合った開発プロセスを実現しなければならない. 開発プロジェクトに良く合った開発プロセスは生産性とソフトウェアの品質を向上させるが, そうでない開発プロセスを強制されるとその結果はプロセス支援がないより悪くなってしまう.

一方, 版管理機能の他に, プロセス支援機能, 変更管理機能, 分散開発支援機能などの高度な機能を持ったシステムにソフトウェア構成管理 (Software Configuration Management: SCM) システムがある. 製品としては, Rational ClearCase [34, 29], Microsoft Visual SourceSafe [25] などが有名である. プロセス支援機能を持つ SCM システムを使えば, 1 つ目の問題は解決する. しかし, 残りの二つの問題は依然として解決しない. 一般に SCM システムは, 機能拡張をサポートするよりも, SCM ベンダがあらかじめ対象とする開発プロセスを定義しておき, それをカスタマイズ可能にして個々の開発プロジェクトに対応させるアプローチをとっている. たとえば ClearCase の場合, UCM (Unified Change Management) [28] という, ベンダがあらかじめ定義している開発プロセスがよく利用される[1]. 開発プロジェクトの管理者は, 自らの開発プロジェクトに合うようにこれをカスタマイズして使用する. しかしながら, SCM システムの対象とする開発プロセスから外れるような開発プロジェクトの要求するプロセス支援機能があった場合, それを実現することは困難である. 実際, UCM では一人の開発者だけがアクセスできる開発ブランチを一つだけ持つと定義されているために, 複数の開発者間で一つの開発ブランチを共有するような開発プロセスを持つ開発プロジェクトを支援できない状況が発生する.

また, 豊富な機能を持つ SCM システムには, シンプルに使えない, 各種コストが高い (導入のためのトレーニング, メンテナンス, 配置), 計算機資源の消費量が多い, 処理が重い (実行が遅い) などといった問題がある [7]. 特にシンプルに使えないという問題は重要で, 依然として単純な版管理システムが広く使われている原因の一つとなっている.

1.3 目的

本研究の目的は, 様々な開発プロジェクトが要求するプロセス支援機能を必要に応じて容易に追加できる, 拡張可能な版管理システムを実現することである. 本研究では, 版管理システムを対象とする. 多機能だが複雑な SCM システムに対し, シンプルに使える版管理システムを拡張可能にし, プロセス支援機能を必要に応じて追加できるシステムについて研究することには意義があると考えられる.

1.2 節で述べた問題点の解決のために, 拡張可能な版管理システムとしてオブジェクト指向フレームワークを用いたレイヤアーキテクチャを用いる手法を提案する. 図 1.1 にレイヤアーキテクチャの概念図を示す.

以下, 各層の特徴を上から順に説明する.

- **拡張機能層**

  プロジェクトが要求するプロセス支援機能のうち, 実際に実装を行う部分である. 下層のオブジェクト指向フレームワーク中のホットスポット部分を差分プログラミングすることにより実現する.

[1] より正確には, UCM が定義されているのは ClearCase に統合されているプロセスエンジン ClearGuide である.

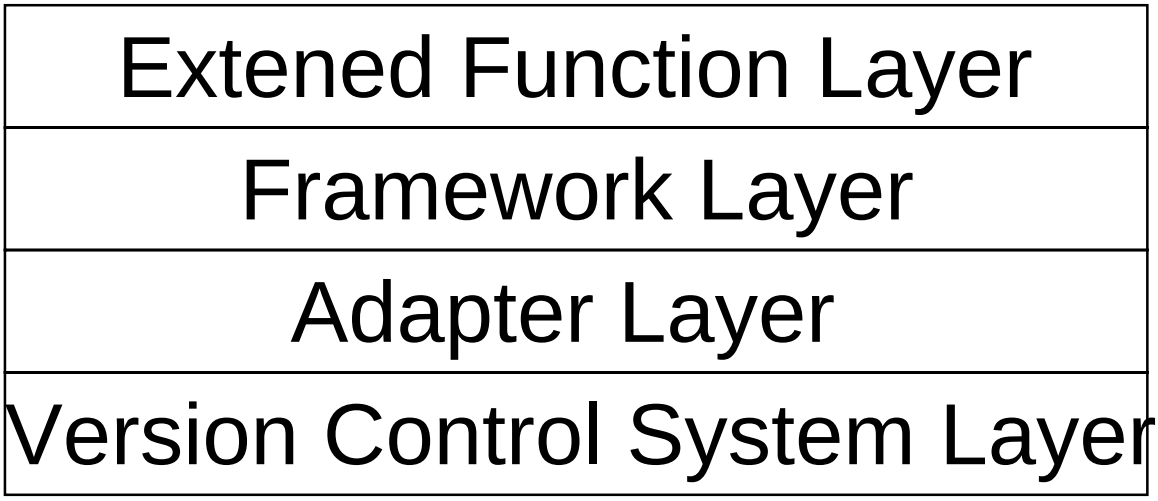

図 1.1: レイヤアーキテクチャの概念図

- **フレームワーク層**

  オブジェクト指向フレームワークで設計 / 実装される. オブジェクト指向フレームワークとは, 半完成のアプリケーションでありホットスポット部分の実装を残しておくことにより拡張性を実現できる. いくつかの拡張機能を実現するにあたり, 汎用的 / 再利用可能な部分や共通に現れる部分をあらかじめフレームワークで実装しておく. これにより, 少ないプログラミングコストで拡張機能を実現できる.

- **アダプタ層**

  版管理システムのラッパーとなる. 版管理システムのインタフェースとオブジェクト指向フレームワークのインタフェースとの間の変換が役割となる. この層があることにより, これより上層では版管理システムに直接依存せずに, 拡張機能の開発に集中できる.

- **版管理システム層**

  版管理システムを再利用して, 版管理機能を提供する. 版管理システムには変更を加えないので, 本システムの実装コストを削減できる.

このレイヤアーキテクチャを用いると, フレームワーク層のオブジェクト指向フレームワークの拡張メカニズムを利用して, 開発プロジェクトの要求に応じた拡張機能を拡張機能層に実現できる. さらに, オブジェクト指向フレームワークのホットスポット部分を差分プログラミングすることにより, 少ないプログラミングコストでの実装を可能にする.

提案するシステムの設計 / 実装にあたり, 版管理システムとして CVS を対象とする. CVS は現在広く利用されている版管理システムであり, ロックを用いない並行処理制御モデル, クライアント / サーバによる非集中開発, ネットワークを介して行われるソースコード公開機能などの特徴がある.

単に CVS の機能不足を補うことが本研究の目的ではないことを強調しておく. CVS の機能不足を補うことを目的とした各種パッチやサードパーティソフトウェアは, 多数存在する. たとえば, CVS の重大な機能的欠点をいくつか改良した Meta-CVS [19], CVS のネットワーク関係のコードをセキュアなものに書き直した cvs-nserver [23], CVS の pserver モードのセキュアなラッパーである cvsd [2], CVS の提供している構成管理プロセスを改良した trug [10] などである. それらとは異なり, 本研究では, まず版管理システムのアーキテクチャを汎用的に拡張可能な構成にすることに焦点を当て, 開発プロジェクトごとに異なるプロセス支援機能を容易に追加可能にすることに重点を置く.

本論文ではレイヤアーキテクチャを用いる本手法の有効性を評価するために, プロトタイプシステムを開発し, 拡張機能としてプロセス支援に関する 2 種類のアクセス権管理機能を実装する. この拡張機能のプログラミングコストを考えることにより, 評価を行う.

1.4 論文の構成

本論文は, 以下の 7 つの章から構成される.

第 2 章で拡張フレームワーク層で用いられるオブジェクト指向フレームワークの概要, 特徴, 利点, 設計手法について述べる. また, 一般に拡張性を実現するメカニズムについても議論する. 次に, 第 3 章では, レイヤアーキテクチャを導入し, 各層について詳しく議論する.

第 4 章 では, 提案したレイヤアーキテクチャを用いるプロトタイプシステムを実装する上で, 本研究のとるアプローチとプロトタイプシステム全体の概要を述べる. また, オブジェクト指向技術を用いた, プロトタイプシステムの設計と実装について詳しく議論する. 第 6 章では, アクセス権管理機能に関する 2 種類の拡張機能を考え, その設計と実装について議論する.

7 章で, 本論文のまとめを行い, 今後の課題 / 展望について述べる.

第2章　拡張性の実現メカニズムとオブジェクト指向フレームワーク

本研究では, 開発プロジェクトの要求するプロセス支援機能を容易に拡張できる版管理システムの構成法を提案する. その拡張性をどのようなメカニズムによって実現するのかが重要な点となる. この章では, まずソフトウェアを拡張可能にする基本メカニズムについて議論する. 次に, オブジェクト指向フレームワークを導入し, 本研究の用いるオブジェクト指向フレームワークを利用した拡張性の実現法について議論する. 最後に, 良いオブジェクト指向フレームワークを設計するための手法について述べる.

2.1　拡張性を実現する基本メカニズム

ソフトウェアの拡張可能な構成とは, どういった基本メカニズムによって実現されているのかについて議論する. 拡張性を実現する基本メカニズムとして, 図 2.1 に示すような単純化したモデルを考える.

アプリケーションはイベントループを持っており, 外部からのイベントの入力を待っている状態にある. また, アプリケーションは**エントリポイント**を複数持っている. エントリポイントは, 新たな機能を拡張できるように設計 / 実装されている部分であり, ある特定のイベントと対応している. 拡張機能である**フック関数**は, エントリポイントに登録される. エントリポイントに登録されたフック関数は, ある特定のイベントが起こった時にアプリケーションからそのイベントハンドラとして呼び出される (コールバック). その際, アプリケーションの方からフック関数の方へ制御が移る. 制御がフック関数に移っている間, アプリケーションは制御がフック関数から戻ってくるのを待つ. アプリケーションはフック関数の実行が終わり制御が戻ってくると実行を再開する. また, フック関数が登録されていないエントリポイントに対応するイベントが起こった場合には, イベントハンドラとして何も実行されない.

アプリケーションの拡張性とは, 新たな機能の追加, 既存の機能の変更, 既存の機能の削除の各操作ができる能力またはシステム構成のことと考えられる. 同時に, このメカニズムには汎用性がある. このモデルにおいて, これらの操作はエントリポイントとフック関数との間の操作に対応づけることができ, それにより拡張性を説明することができる. 以下にその対応を示す:

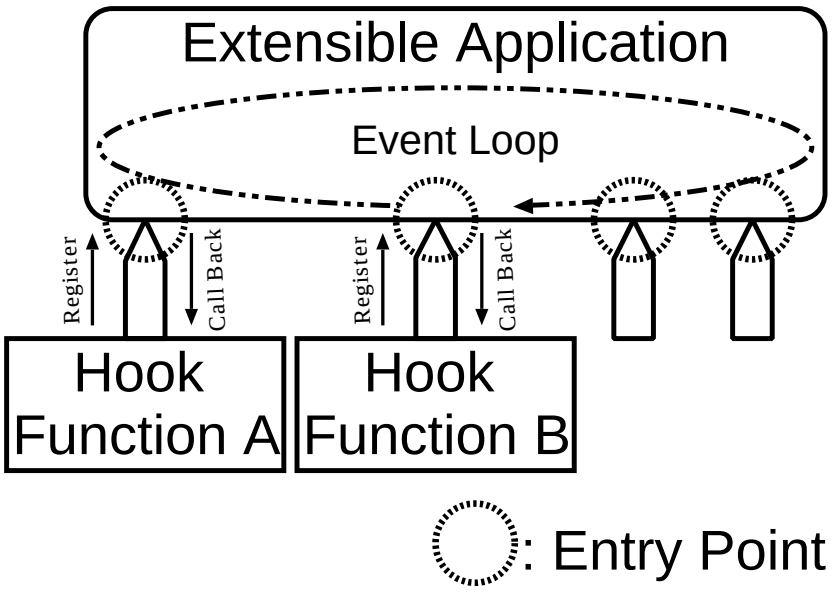

図 2.1: 拡張性を実現する基本メカニズムの概念モデル

- **新たな機能の追加**

  エントリポイントに新たなフック関数を登録する.

- **既存の機能の変更**

  エントリポイントに登録されているフック関数を別のフック関数で置き換える.

- **既存の機能の削除**

  エントリポイントに登録されているフック関数を取り除く.

ここで, 既存の機能の変更が可能であれば, 既存の機能の削除も可能である. それには, 実装の中身が空 (何もしないで終了する) のフック関数を用意し, それで既存のエントリポイントに登録されているフック関数を置き換えれば実現できる.

したがって, “新たな機能の追加” と “既存の機能の変更” の二つについて議論すればよい. 以下の章では, 本研究で用いるオブジェクト指向フレームワークがこの二点を満たすことを示す.

2.2 オブジェクト指向フレームワークの概要

オブジェクト指向フレームワークとは, カスタマイズされたアプリケーションを作り出す, 再利用可能で半完成のアプリケーションである [13]. そのため, 設計と実装のコストを大幅に削減できソフトウェアの品質を向上できる. クラスライブラリを用いた細粒度の再利用法に比べて, オブジェクト指向フレームワークは特定のアプリケーションドメインに対応した粗粒度のコンポーネントを単位として再利用が可能である.

オブジェクト指向フレームワークには主に以下の 4 つの利点がある:

- **モジュラ性**

  オブジェクト指向フレームワークは, その実装を公開されたインタフェースにカプセル化する. 設計や実装が変わってもその影響がインタフェースの外に波及しない. このモジュラー性は, 開発者のオブジェクト指向フレームワークに対する理解を助け, メンテナンス性を向上させる.

- **再利用性**

  オブジェクト指向フレームワークのインタフェースを拡張 / カスタマイズすることにより, オブジェクト指向フレームワークの実装を再利用した複数のアプリケーションを作ることができる. つまり, 設計と実装の再利用が可能である. これにより, 開発者の生産性を向上させることができる.

- **拡張性**

  オブジェクト指向フレームワークには, 拡張可能な部分であるホットスポットと固定部分であるフローズンスポットがある. ホットスポット部分をアプリケーションのコンテキストに応じて実装することにより, さまざまなアプリケーションを実現できる. この拡張性については, 2.3 章にて詳しく議論する.

- **制御の反転**

  アプリケーションは普通, 開発者がイベントループを含むメインアプリケーションを開発し, イベントが起きた際にそれに対応するサブコンポーネントを呼び出すという制御の流れを持っている. そのサブコンポーネントはライブラリなどから再利用する. これと反対に, オブジェクト指向フレームワークでは再利用するフレームワークの方がイベントループを持っており, イベントが起きたら対応するイベントハンドラを呼び出すという制御の流れになる. 開発者はこのイベントハンドラのみを実装すればよい. つまり, 開発者はドメインに特化した (特定のイベントに対する) 実装のみに集中できる.

オブジェクト指向フレームワークの基本は抽象クラスである. 抽象クラスは, オブジェクトのコラボレーションの基本的な設計であり, アルゴリズムのテンプレートを実装して

いる. 抽象クラスにアプリケーション固有のカスタマイズを行い具象クラスをつくる. したがって, 抽象クラスと具象クラスの組み合わせからオブジェクト指向フレームワークは構成されている.

一般にオブジェクト指向フレームワークは, デザインパターン [17] を用いて設計される. デザインパターンは, ソフトウェア開発の設計段階において特定のコンテキストで繰り返し発生するソフトウェア開発における問題の解法をパターンとしてまとめたものである. オブジェクト指向フレームワークがオブジェクト指向プログラミング言語によって記述されており実行可能であるのに対して, デザインパターンはプログラミング言語で記述されていないのでそのままでは実行できない. デザインパターンをオブジェクト指向フレームワークのホットスポット部分の設計に適用し, そのドメインに特化した実装を行う.

2.3 オブジェクト指向フレームワークの拡張性

オブジェクト指向フレームワークは, フローズンスポット部分とホットスポット部分の 2 つの部分から成る. フローズンスポット部分は固定的で, アプリケーションの処理の流れ, 抽象的な振る舞い, オブジェクト間の関連を実装している部分である. ホットスポット部分は拡張可能で未実装のまま残されている部分である. 拡張性の実現にはホットスポット部分の設計が重要となる.

つまり, 図 2.1 とオブジェクト指向フレームワークとの関係を次のように考えることができる. アプリケーションがオブジェクト指向フレームワーク, エントリポイントはフローズンスポット部分, フック関数はホットスポット部分に対応する.

ホットスポット部分の設計をする際に重要な役割を担うのが抽象クラスである. 抽象クラスは, そのままではインスタンスを作ることができないが, サブクラスに未実装のメソッドの実装を任せることにより, サブクラスではそのメソッドを拡張することができる. それによってインスタンスを作ることができるようになる. また, オブジェクト指向フレームワークは, 実装しているオブジェクト指向プログラミング言語のメソッドの動的束縛機能に強く依存している.

たとえば, 図 2.2 のような基本的でシンプルなクラス図[1]を考えてみる. クラス *B* は, m1, *m2*, m3 の 3 つのメソッドを持っている. メソッド *m2* が抽象メソッドなのでクラス *B* は抽象クラスである. メソッド *m2* はインタフェースのみ提供している. メソッド m1 はその実装の中でメソッド *m2* とメソッド m3 を呼び出している.

ここでポイントとなるのは, メソッド m1 の実装に抽象メソッド *m2* の呼び出しが入っているところである. 抽象メソッド *m2* の実装をサブクラスに任せることにより, メソッド m1 の実装のコンテキストにおける拡張が可能になっている. さらに, B1 は, 必要であればクラス *B* のメソッド m3 をオーバーライドすることもできる. オーバーライドとは実装の上書きであり, メソッド m3 が呼び出されたときクラス *B* のメソッド m3 の実装ではなく,

[1]UML 図中のノートに示したメソッドのコード例は, オブジェクト指向プログラミング言語 Ruby [24] の文法で記述している. 以後, 本文中に出てくるコード例も特に断りのない限り同様である.

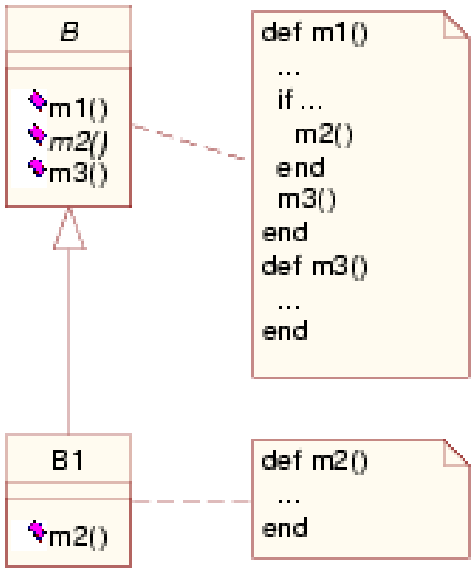

図 2.2: ホットスポット部分の基本構造 1

クラス *B* のサブクラス B1 のメソッド m3 の実装が実行される.

クラス *B* を継承したクラス B1 は, スーパークラス *B* の抽象メソッドであったメソッド m2 を実装しており, 具象クラスとなるのでインスタンスを生成できる. クラス B1 のインスタンスを b1 とする.

この設計を実装するオブジェクト指向プログラミング言語のメソッドの動的束縛機能により, b1 へのメソッド m1 の呼び出し b1.m1() は, クラス *B* のメソッド m3 とクラス B1 のメソッド m2 を用いて実行される.

クラスの構造に注目すると, クラス *B* (もしくはより詳細に, クラス *B* のメソッド m1) がオブジェクト指向フレームワーク中フローズンスポット部分になっており, クラス B1 (もしくはより詳細に, クラス B1 のメソッド m2) がホットスポット部分となっている.

これを図 2.1 で用いた言葉で言い換えると, フローズンスポット部分とホットスポット部分はそれぞれエントリポイントとフック関数であったから, クラス *B* がエントリポイント, クラス B1 がフック関数となる.

以上から, オブジェクト指向フレームワークは拡張性を実現するために必要な以下の 2 点を満たす.

- **新たな機能の追加**

  クラス *B* の抽象メソッド *m2* のところに, サブクラス B1 はメソッド m2 実装して, b1.m1() を実行した際に *m2* ではなく m2 が実行されるようになった. すなわち, エン

トリポイントに新たなフック関数を登録できたと考えられる. 確かに, このオブジェクト指向フレームワークを含むアプリケーション全体からみれば, メソッド m2 という新たな機能が追加されたことになる.

- **既存の機能の変更**

  クラス *B* はメソッド m3 を実装しているが, サブクラス B1 が *B* の m3 をオーバーライドして新たに B1 の m3 を実装した場合, b1.m1() を実行する際に *B* の m3 ではなく B1 の m3 が実行されるようになった. すなわち, エントリポイント登録されているフック関数を別のフック関数で置き換えることができたと考えられる. 確かに, このオブジェクト指向フレームワークを含むアプリケーション全体からみれば, クラス *B* の m3 という既存の機能が変更されたことになる.

  3 つ目の “既存の機能の削除” についても, B1 の m3 を

  ```
  def m3
  end
  ```

  というように空の実装を行えば, “既存の機能の削除” 機能を用いて同様に満たすことができる.

ここまでの議論により, 前節で述べた “新たな機能の追加” と “既存の機能の変更” の 2 つの機能が実現できる. よって, オブジェクト指向フレームワークを拡張性を実現するためのメカニズムとして用いる本研究のアプローチは, 適用するアプリケーションに汎用的な拡張性を与えることができる.

GoF は, オブジェクト指向フレームワークの実装言語による拡張手法として**サブクラス化**と**オブジェクトコンポジション**をあげている [17]. 図 2.2 は, このサブクラス化の手法を用いて拡張を行っている典型的な例となっている. このクラス構造は, デザインパターンカタログにおける Template Method パターンの構造 (図 2.3) と同様である. Template Method パターンは, 抽象クラスで定義されているアルゴリズムの各ステップを後に拡張できるようになっている. クラス ConcreteClass (または, そのメソッド PrimitiveOperation) がホットスポット部分である. 共通のアルゴリズムと処理の各ステップをテンプレートとして抽象クラスで実装し, 各ステップの処理の詳細の実装を具象クラスに任せている.

次に, 図 2.4 では, もう一方のオブジェクトコンポジションの手法を用いている基本的でシンプルな例をあげる.

抽象クラス *B* とそのサブクラス B1 は図 2.2 と同様である. サブクラス B2 は B1 とは異なるメソッド m2 の実装を行っている. クラス *B* は B1 や B2 以外にも, メソッド m2 を独自に実装している複数のサブクラス (B3, B4, ...) を持っているものとする. クラス A は抽象クラス *B* への参照 bRef を持っており, メソッド a_m で bRef の参照先のインスタンスのメソッド m1() を呼んでいる.

たとえば, bRef が B1 のインスタンス b1 を参照しているとすると, クラス A のインスタンス a のメソッド a_m の呼び出し a.a_m() は, bRef.m1() を呼び出す. bRef は b1 を参照し

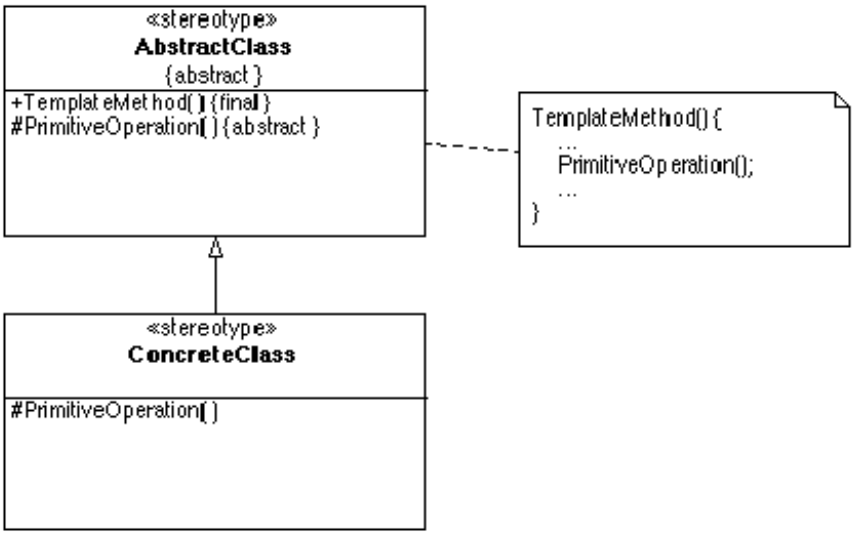

図 2.3: Template Method パターン

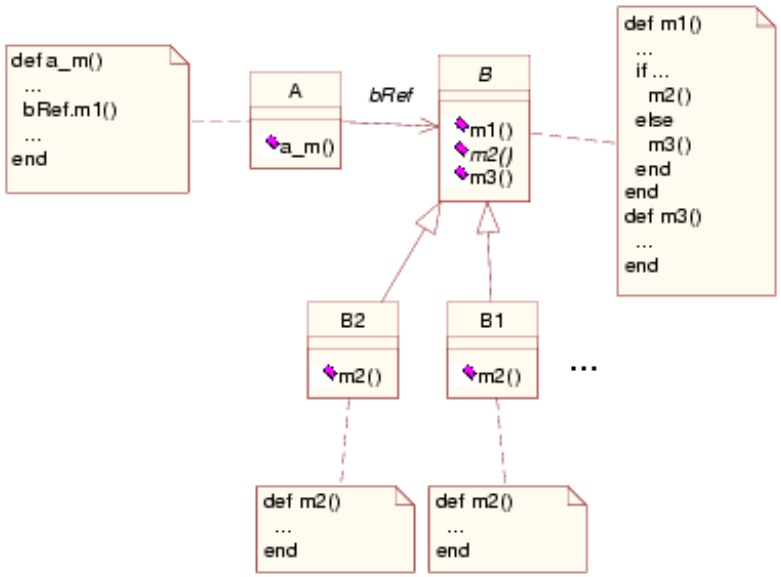

図 2.4: ホットスポット部分の基本構造 2

ているから, b1.m1() が呼び出される. b1.m1() の実装は, スーパークラスである *B* で行われている. *B* のメソッド m1 では抽象メソッド m2 とメソッド m3 を使用している. 抽象メソッド m2 の実装はクラス B1 にあるので b1.m2() が実行される. メソッド m3 に関しては, 抽象クラス *B* が実装している m3 を実行する.

この構造で重要な点は, クラス A のメソッド a_m の中の bRef が抽象クラス *B* への参照であることである. bRef は, 抽象クラス *B* を継承しているすべてのサブクラスを参照できる. これは, 継承によりすべてのサブクラスが抽象クラス *B* のインタフェースを持っているからである. 抽象クラス *B* のサブクラスは, それぞれの必要に応じてメソッド m2 を実装できる. クラス A は目的に応じて動的に抽象クラス *B* のサブクラスのインスタンス m1, m2, ... のメソッド m2 の実装を含んだ bRef.m1() の機能を利用できる. つまり, この手法がオブジェクトコンポジションである.

この図において, フローズンスポット部分はクラス A のメソッド a_m とクラス *B* のメソッド m1 であり, ホットスポット部分はクラス *B* のメソッド m1 とクラス B1, B2, ... のメソッド m2 である. ここで注意したいのは, *B* のメソッド m1 がホットスポット部分でもありフローズンスポット部分でもあることである. *B* のメソッド m1 は, クラス A のメソッド a_m に対してホットスポット部分であるが, クラス B1, B2, ... のメソッド m2 に対してはフローズンスポット部分となっている. このように, オブジェクト指向フレームワークの拡張可能な部分を構成するクラス間の関係を個々のメソッドの呼び出しまで細かくみていくと, 見方によってはホットスポット部分でもありフローズンスポット部分でもあるような部分もあることがわかる. この構成は, あるフローズンスポット部分に対するホットスポット部分であったところが, 別のホットスポット部分に対するフローズンスポット部分になっているという, 2 段の拡張が可能になっている例である. この連鎖が複数段にわたる, より複雑な構成をとる場合もある.

デザインパターンカタログの Strategy パターン (図 2.5) と Command パターン (図 2.6) のクラス構造は, 基本的に図 2.4 の構造と同様である. Strategy パターンは, アルゴリズムの選択を自由に行うことができる点に拡張性がある. ConcreteStrategy クラスのメソッド AlgorithmInterface がホットスポット部分となる. Command パターンは, 命令をオブジェクトにカプセル化し, いつどのように命令を実行するかを自在に設定できる点に拡張性がある. ConcreteCommand クラスのメソッド Execute がホットスポット部分となる.

図 2.2 と図 2.4 で説明したホットスポット部分の基本構造は, 実際のオブジェクト指向フレームワークの中のエントリポイント / フック関数の設計にそのまま現れたり, またはより複雑な構造の部分として現れたりする. したがって, これらの図で示したクラス構造をホットスポット部分の設計を行う際の基本設計として用いることにより, オブジェクト指向フレームワークの拡張性を実現できる.

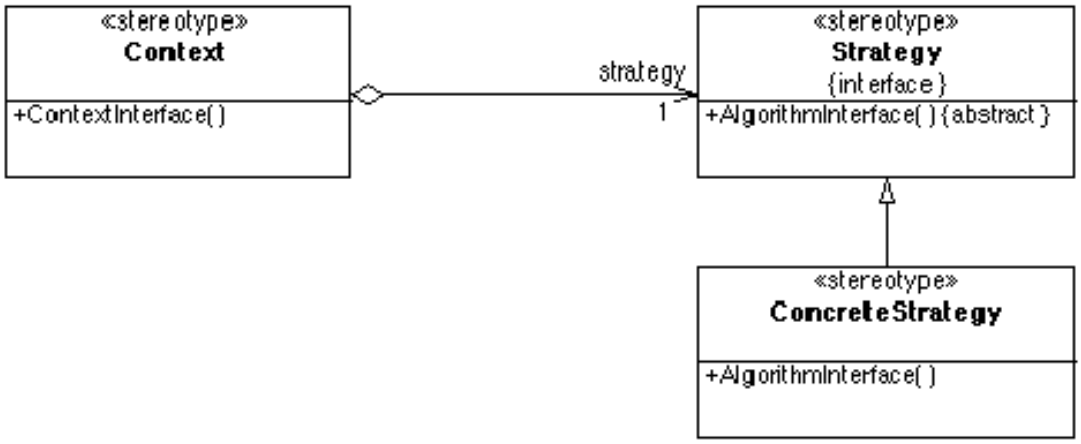

図 2.5: Strategy パターン

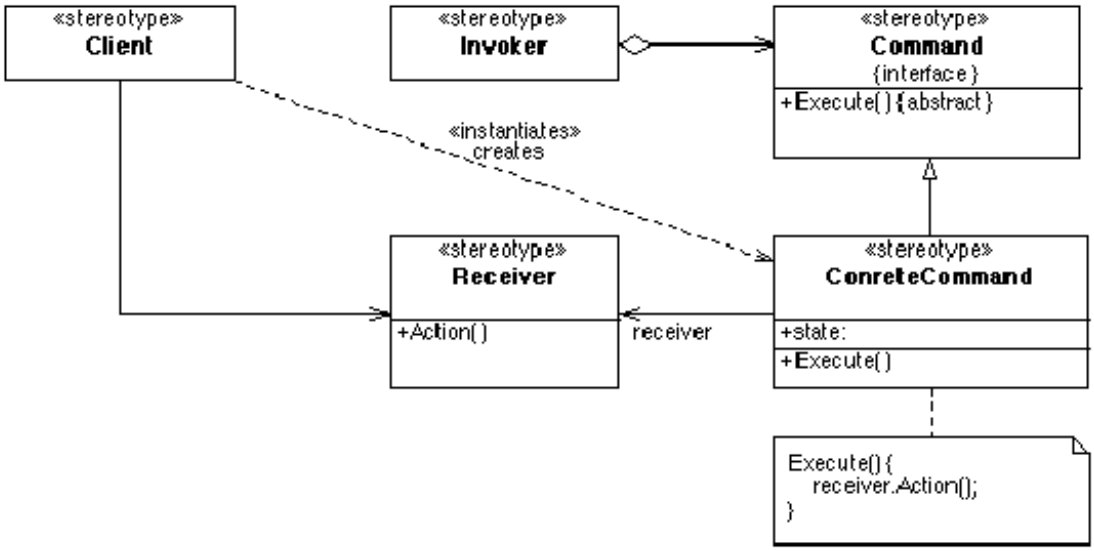

図 2.6: Command パターン

2.4 デザインパターンを用いたオブジェクト指向フレームワークの構成手法

一般に, 良いオブジェクト指向フレームワークの設計を行うことは難しい作業である. 良いオブジェクト指向フレームワークには良いホットスポット部分の設計が行われていなければ, 高い拡張性を提供できないからである.

ホットスポット部分の設計をするには, ドメインに特化した知識が必要となる. つまり, そのドメインにおいて, オブジェクト指向フレームワークがどのように利用され, オブジェクト指向フレームワークのどの部分に柔軟性が必要で, 将来にわたってそこがどのように拡張できるべきなのかということを完全に明らかしなければいけないからである.

しかしながら, そのようなドメイン分析が必要なことにはかわりがないが, 拡張の要求が明確になった後, その拡張要求をどのようなソフトウェア構成で実現するかについていくつかの有用な手法がある.

特に, デザインパターンを用いる手法は有用である. デザインパターンはオブジェクト指向フレームワークより抽象度が高く, 特定のドメインに特化したものではない. あるコンテキストで頻繁に起こる問題に対する解法をパターンの形で与えるものである. デザインパターンの目的, 動機, 結果, 適応可能性を考慮したうえで, マッチするパターンをオブジェクト指向フレームワークの設計に適用すれば, 拡張性や再利用性を高めることができる.

以下の小節では, デザインパターンを用いてオブジェクト指向フレームワークの設計を支援する 2 つの手法について説明する.

2.4.1 デザインパターンのホットスポットとフローズンスポット

GoF のデザインパターンカタログは, 生成に関するパターン, 構造に関するパターン, 振る舞いに関するパターンの 3 種類に分類されており, 合計して 23 個のパターンが記述されている. 各パターンには, 目的, 動機, 結果, 適応可能性などの項目があり, どのパターンをどのようなコンテキストで用いればどのような結果が得られるか分かるようになっている.

しかし, オブジェクト指向フレームワークの構成を目的とした場合には, 必ずしも GoF のパターンカタログの記述方法が最良であるわけではない. なぜならば, 個々のパターンのクラス構成を詳細に検討しなければ, どの部分にどのような拡張性があるのかがひと目では分かりにくいからである.

C. W. Irving らは, デザインパターンを拡張性の視点から分類し整理した [18]. 表 2.4.1 にその抜粋を示す.

図中の Semantics とは, クラスの “意味” のことであり, そのクラスの定義とすべてのスーパークラスの定義を含んだものである. オブジェクト指向フレームワークの提供するクラスの意味を継承によって変更できる場合, Semantics ホットスポットになる. Composition とは, オブジェクト指向フレームワークの提供するがクラスのインスタンスを集約できる

表 2.1: デザインパターンの拡張性の視点からの分類 [18] (抜粋)

Name	Hot spot	Frozen spot	Hot spot participants
Factory Method	S	T, P, C	ConcreteCreator, ConcreteProduct
Adapter	T, P	S, C	Adapter, Adaptee
Composite	S, C	T, P	Composite, Leaf
Command	S, C	T, P	ConcreteCommand
Strategy	S	T, P, C	ConcreteStrategy
Template Method	S	T, P, C	ConcreteClass

S : Semantics; C : Composition; T : Type compatibility; P : Protocol

場合, Composition がホットスポットになる. Type compatibility は, オブジェクトの型の互換性のことであり, Protocol はクラスのインタフェース (呼びだし可能なメソッドの集合) のことである. 変数に型があるプログラミング言語の場合, Type compatibility と Protocol の間の区別はなくなる. オブジェクト指向フレームワーク中のクラスのインスタンス間の呼びだし関係を変更できる場合に, これらがホットスポットであるという.

この表から分かることは, Semantics & Composition と Type compatibility & Protocol はトレードオフの関係にあるということである. 両方の要求を同時に満たすことはできない.

前章で述べたとおり, サブクラス化を行っている Strategy と Template Method パターンは, Semantics がホットスポットとなっている. オブジェクトコンポジションを行っている Command パターンは, Semantics と Composition がホットスポットとなっている. Command パターンに Semantics の拡張性が含まれているのは, 図 2.6 の抽象クラス Command とそれのサブクラス ConcreteCommand のところの構造が, 図 2.2 に示したホットスポット部分の基本構造になっているためであると考えられる.

この表をを用いると, それぞれのデザインパターンのホットスポットとフローズンスポットの箇所とホットスポットの構成要素が明確になり, 何を拡張可能にしているのかがはっきりする. したがって, オブジェクト指向フレームワークの拡張性を実現するために適用するデザインパターンの選択を支援することができる.

2.4.2 メタパターン

W. Pree は, オブジェクト指向フレームワークを構成する際に適用するメタパターンを提案している [27]. メタパターンはデザインパターンのメタレベルの集合であり, ドメイン非依存のオブジェクト指向フレームワーク構成法を示すものである. 特に, フレームワークのホットスポット部分のクラス構造を形式的に解析している.

オブジェクト指向フレームワークのホットスポット部分とフローズンスポット部分を,

メタパターンではそれぞれ *hook* と *template*[2] の概念で説明する. hook と template を基本原理としたメタパターンは, オブジェクト指向フレームワークの全ての概念を表現できると主張している.

オブジェクト指向フレームワークの拡張性を実現するメカニズムの基本構成要素として *template method*[3] と *hook method* が重要な役割を担う. template method は, 抽象的な振る舞いや制御の流れを定義している. hook method は 3 種類あり, abstract, regular, template である. 例をあげてこれらの概念を説明する.

図 2.2 において, クラス *B* のメソッド m1 が template method であり, クラス B1 のメソッド m2 が regular hook method (普通に実装を行っているメソッド) である. クラス B1 のメソッド m2 が, さらにクラス *B* のメソッド m1 のように他の機能を呼び出す雛形として実装されている場合クラス B1 のメソッド m2 を template hook method と呼び, クラス *B* のサブクラスでありながら, まだメソッド m2 を実装せずサブクラスに実装を任せている場合 (たとえば抽象クラス *B3* の抽象メソッド *m2*) を abstract hook method という.

次に, *template class* と *hook class* を導入する. template method を含んでいるクラスを template class という. hook class は, 対応する template class をパラメタ化するクラスである. メタパターンは template class と hook class を構成要素として, Unification, 1:1 Connection, 1:N Connection, 1:1 Recursive Connection and 1:1 Recursive Unification, 1:N Recursive Connection and 1:N Recursive Unification の 5 種類のパターンに大きく分類される.

例として図 2.7 に, Unification メタパターン (a), 1:1 Connection メタパターン (b), 1:N Recursive Connection メタパターン (c) を示す. 図中の表記で, T が template class, H が hook class, t が template method, h が hook method を表す. TH または th と表記されているところは, template でありかつ hook でもあるクラスまたはメソッドである.

図 2.2 のクラス構造は, Unification メタパターンの例となっている. 一つのクラスが template method と hook method の両方を持っており, ホットスポット部分であるサブクラスの hook method を拡張できる. 図 2.4 のクラス構造は, 1:1 Connection メタパターンの例になっている. hook class であるクラス *B* の hook method *b* を拡張できる. 1:N Recursive Connection メタパターンの例として, 図 2.8 に Composite パターンのクラス図を示す. template class が hook class を集約しているところに特徴があり, クラスのツリー構造を実現している. hook class のリストである hList に含まれるクラス (Composite や Leaf) を拡張できるようになっている.

メタパターンを用いると, デザインパターンを用いたオブジェクト指向フレームワークのホットスポット部分のクラス構造を形式的に知ることができる. これにより, オブジェクト指向フレームワークのホットスポット部分の設計者が拡張可能なクラス構造の設計を行う支援ができる.

[2]C++ の template とは全く関係ない.
[3]GoF のデザインパターンカタログの Template Method パターンとは関係ない.

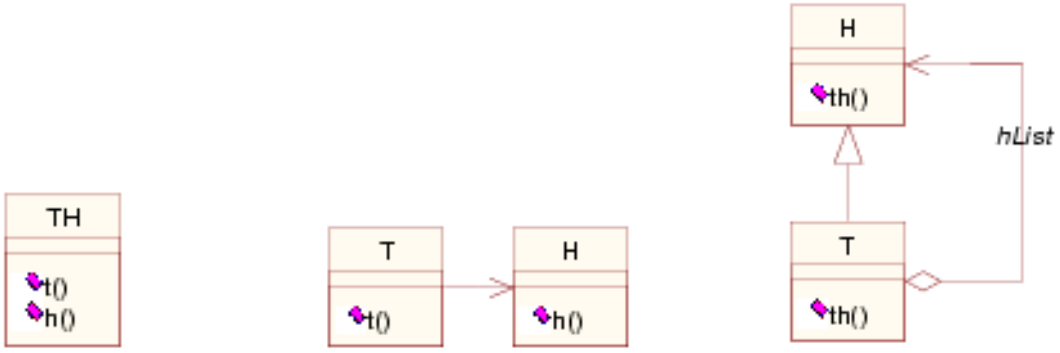

図 2.7: Unification メタパターン (a), 1:1 Connection メタパターン (b), 1:N Recursive Connection メタパターン (c)

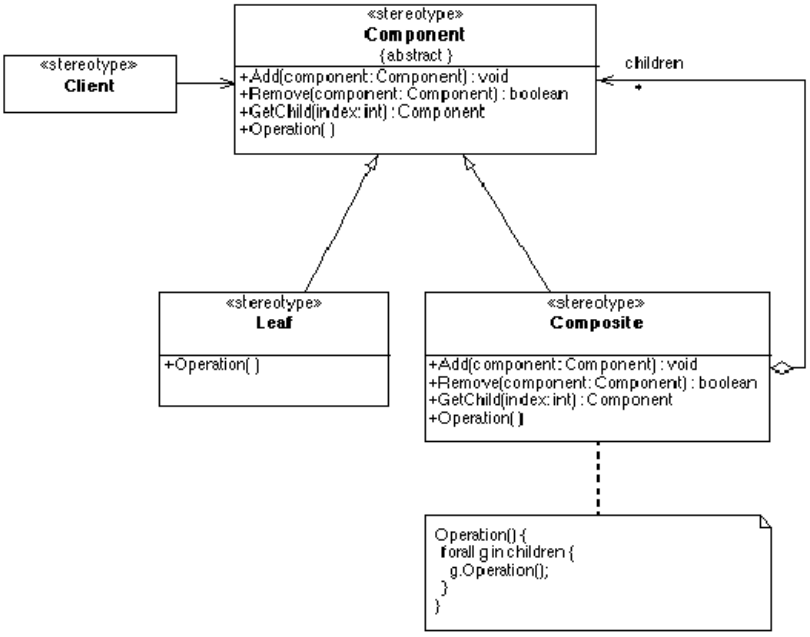

図 2.8: Composite パターン

2.5　本章のまとめ

オブジェクト指向フレームワークの拡張性は, 拡張機能の追加, 変更, 削除を行うことができる汎用的なものである. その実現にはデザインパターンがよく用いられる. 拡張性を実現するクラス構造にはいくつか基本的なものがあり, パターンとして記述できる. オブジェクト指向フレームワークを実装をしているオブジェクト指向プログラミング言語の提供するサブクラス化とオブジェクトコンポジション技術が, 拡張性の実現のための基本メカニズムである.

また, オブジェクト指向フレームワークは, 粗粒度の再利用性も提供する. 個々の機能を再利用するクラスライブラリを用いる手法に比べ, アプリケーション全体のアーキテクチャや大部分の設計 / 実装を再利用できる. さらに, オブジェクト指向プログラミング言語の提供するサブクラス化とオブジェクトコンポジション技術により差分プログラミングが可能となり, 拡張機能の実現のためのプログラミングコストを低く押さえられる.

以上のような特徴や利点を持つオブジェクト指向フレームワークを本研究で採用することにより, 1.2 節で述べた問題点のうち拡張性に関するものを解決できると考えられる.

第3章　レイヤアーキテクチャ

この章では, 開発プロジェクトの要求するプロセス支援機能を追加できる拡張可能なシステムのレイヤアーキテクチャを提案する.

3.1　レイヤアーキテクチャの概要

本研究では, 版管理システムとして CVS を採用している. CVS を採用した理由は 3 点ある. 1 つめは, CVS が現在広く利用されている版管理システムであるということである. オープンソース開発プロジェクトでは, ほぼすべてのプロジェクトが CVS を版管理システムとして利用している[1] CVS はオープンソースソフトウェアであり, だれでも無償で使用することができる. さまざまなプラットフォームで CVS は動作し, GUI (Graphical User Interface) を用いた CVS クライアントも多数作られている [8]. これらが CVS が広く使われている理由である.

2 つめの理由は, CVS は中間成果物の版管理のために行う操作がシンプルであり, かつ版管理システムとして基本的な機能を持っていることである. CVS には, ロックを用いない並行処理制御モデル, クライアント / サーバによる非集中開発, ネットワークを介して行われるソースコード公開機能などの特徴がある. これらの特徴は, オープンソースソフトウェア開発に代表されるような分散共同ソフトウェア開発を良く支援する. CVS にはいくつかの欠点がある[2]のだが, これらの特徴を備えている CVS は, 分散共同ソフトウェア開発において必要な基本的機能を最低限持っているといえる.

3 つめの理由は, CVS はオープンソースソフトウェアであり, すべての仕様とソースコード, ドキュメントが無償で公開されている. 特にソースコードが公開されているので, CVS の挙動が明らかである. ドキュメントにすべてソフトウェアの挙動を記述することは困難なので, 挙動が不鮮明なときにはソースコードを読めば解決する. さらに, CVS 内部の動作 (アルゴリズム, データ構造, 制御の流れなど) を詳細に知ることができる. もしクローズドソースのソフトウェアならば, ドキュメントにない挙動をした場合にバグなのか仕様なのか区別がつかないが, オープンソースソフトウェアならばソースを読むことによって解決

[1] Linux [21] は, カーネルの版管理システムとして BitKeeper [5] を使用しており, CVS を利用していないオープンソース開発プロジェクトの代表例である.

[2] メタデータやディレクトリの版管理機能がない, ファイルの名前変更機能がない, コミット操作がアトミックではない, ブランチ作成やタグを打つ操作の処理が重い, バイナリファイルをうまく扱えない, クライアント / サーバプロトコルが効率的でないなどといった欠点が指摘されている.

できる. このようなソフトウェアを用いると, 研究を行っていく上で都合が良い. 以上の理由から, 本研究では版管理システムとして CVS を対象としている.

レイヤアーキテクチャ [6] を用いる最大の利点は, “関心事の分離 (Separation of Concerns)” を行えることにある. システム全体の複雑さが各層にカプセル化され, 各層の提供する機能の実装の詳細を隠蔽できる. その結果各層は, 下層の機能の実装方法に関係なく, その機能のインタフェースを使って下層の提供する機能を利用できる.

提案するアーキテクチャは, 図 3.1 に示すとおり次の 6 層からなる: CVS の版管理データモデルである RCS 形式で中間成果物を保存する **RCS ファイル層**; CVS の機能と RCS ファイルを操作するための機能を提供する **CVS / RCS ライブラリ層**; 版管理システムとオブジェクト指向フレームワークとの間のインタフェースの変換をする **CVS アダプタ層**; 基本的な機能の拡張のためのオブジェクト指向フレームワークを提供する **基本システムフレームワーク層**; 拡張機能の実現ために, ドメインに特化した機能を実装するオブジェクト指向フレームワークである**拡張機能フレームワーク層**; 開発プロジェクトの要求するプロセス支援機能を実装する **拡張機能層**.

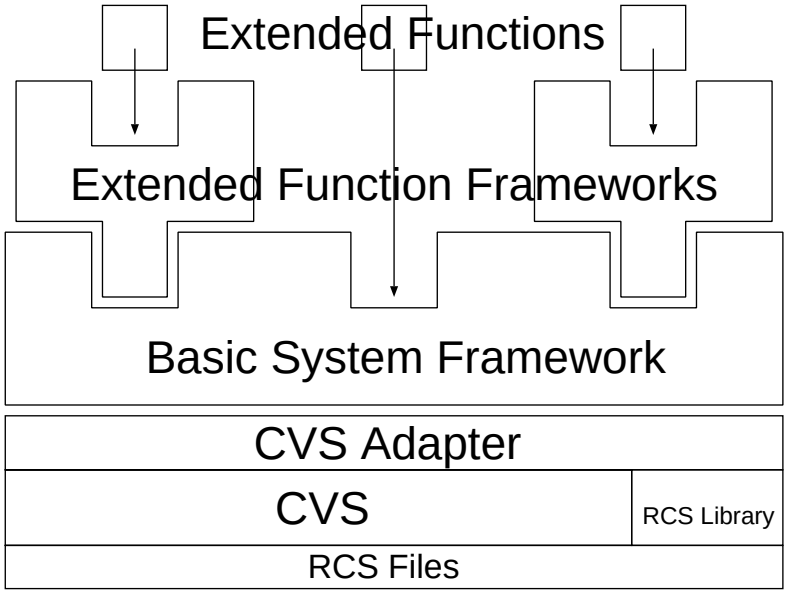

図 3.1: 6 層レイヤアーキテクチャ

図 1.1 の概念図とこのレイヤアーキテクチャの 6 つの層は, 下から順に次のように対応する: 版管理システム層は, RCS ファイル層と CVS / RCS ライブラリ層; アダプタ層は, CVS アダプタ層; フレームワーク層は, 基本システムフレームワーク層と拡張機能フレームワーク層; 拡張機能層は, 拡張機能層.

このような層を持つレイヤアーキテクチャには, 3 つの重要な特徴がある. 第一の特徴は, アダプタ層を境目に下層の版管理システム層と上層のフレームワーク層を明確に分離する構造である. アダプタ層は上層に対し下層の CVS システムの実装の詳細を隠蔽する. これにより, フレームワーク層が CVS の実装の詳細に直接依存することがなくなり, 拡張機能の実現という目的のためだけにフレームワークの設計 / 実装に集中できる. たとえば, CVS の実装ではなく CVS の提供する機能のインタフェースだけに注目してフレームワークの開発を行える. また, CVS アダプタにはインタフェース変換の役割がある. つまり, “CVS の提供するユーザ / 拡張インタフェース” から “オブジェクト指向フレームワークとしての API (Aplication Programming Interface)” への変換である. フレームワーク層の実装にはオブジェクト指向フレームワークを用いるので, CVS の提供する機能をオブジェクト指向 API として利用できる必要がある. CVS アダプタがオブジェクト指向技術を用いて CVS の提供する機能をラップすることにより, この必要性を満たすことができる.

レイヤアーキテクチャには, 上層へ提供するインタフェースを変えない限りにおいて, その層の実装を別の実装に置き換えることができるという性質がある. アダプタ層の存在によりフレームワーク層に変更を加えることなく, 版管理システム層を RCS ファイル層と CVS / RCS ライブラリ層以外のものに置き換えることができる構造になっている. 将来的には, CVS といった特定の版管理システムに限らず, 開発プロジェクトが好む版管理システムを本アーキテクチャで使えるようにしたいと考えている. これが第二の特徴である.

第三の特徴は, フレームワーク層に階層型オブジェクト指向フレームワークを導入している点である. 階層型オブジェクト指向フレームワークとは, 対象とするアプリケーションドメインを分割して, その部分ドメインに応じてフレームワークの層をつくり, 階層化する手法である [3]. 一般に広いドメインをカバーする単一のフレームワークをつくることには大変なコストがかかるが, 本手法を用いると各層のフレームワークは, 分割されたより狭いドメインをカバーするように設計されれば良いので, 開発コストが低くてすむし設計もしやすい. 本アーキテクチャにおいては, 拡張のための基本的なメカニズムを提供する基本システムフレームワークと, 開発プロジェクトが要求する個々の拡張機能のドメインに特化した拡張機能フレームワークの 2 層に階層化することにより, 開発コストとシステム全体の複雑さの低減をねらっている.

以下の節では, 各層の役割と提供する機能について下の層から順に説明する.

3.2 RCS ファイル層

この層では, RCS 形式ファイルのストレージ機能を提供する. CVS が内部的に RCS の機能を用いて実装されているため, この層が存在する.

CVS の版管理データモデルは RCS 形式である. 版管理データモデルとは, 版管理システムに保管される中間成果物の種類の抽象的な表現のことであり, 中間成果物間の関係を表している [12]. 版管理データモデルは, 開発中のソフトウェア製品の構成を定義するのに使われる. すなわち, CVS は中間成果物であるファイルを RCS 形式で保管する.

RCS 形式の中間成果物は, 通常オペレーティングシステムが提供するファイルシステム上に置かれる. つまり, 中間成果物はファイルシステムの提供する “ファイル” として表現され, 中間成果物のファイル名に “,v” を追加した形式の名前で管理される. 一つの中間成果物に対するバージョンは, デルタメカニズム (diff [11], patch [16]]) を用いて一つのファイル内に保存される. メタデータ (作者, バージョンナンバ, 日付, コメント, タグ情報など) も同様にそのファイルに保存される.

3.3 CVS / RCS ライブラリ層

この層は, 2 つの構成要素から成る.

3.3.1 CVS 層

この層では, CVS の持つ版管理機能をそのまま提供する. CVS 自体には一切変更を加えない. 既存の版管理システムを基本的な版管理機能として再利用する構成を用いているので, 本アーキテクチャを実装するシステムの実装コストを大きく削減できる.

CVS は分散共同開発に適した, 次に述べるようないくつかの特徴を持っている. 版管理システムをそのまま再利用する本手法には, このような版管理システムの特徴を損なうことなく, アーキテクチャに取り込めるという利点もある.

RCS は, “ロック-修正-アンロック” モデルを採用している. これは, ファイルを修正する前にそのファイルにロックをかけ, 他の人が変更できないようにしておき, 修正が終わったらその内容をリポジトリにコミットし, 最後にロックを解除するというものである. このモデルでは, 一人の開発者が修正中のファイルには他の開発者が修正を加えられないので, 複数の開発者による共同開発を効率良く行うことが困難である. そのため, 個人使用が奨励されている.

一方, CVS はロックを用いない “コピー-修正-マージ” モデルを採用しており, 複数の開発者による同時平行開発が可能である. このモデルでは, 修正を行いたい開発者はリポジトリよりファイルのコピーを行い, 手元で修正を加えて, その修正差分をリポジトリにマージする. RCS とは異なり修正中のファイルにロックをかけず, 開発者がある一つのファイルに対するリポジトリよりのコピーを行う数に制限がないことが重要である. これにより, 複数の開発者がそれぞれ独立して同時平行的に開発を行うことができ, 効率良く開発が行える. しかしこのモデルでは当然, 開発者がそれぞれ行った修正の間に競合が起こる可能性がある. CVS ではそのような場合, 先にコミットした開発者の修正がまずコミットされ, 後からコミットする開発者にはコミット前に手作業でその競合を解決することが求められる. 競合が解決された後, はじめてコミットが許される.

CVS はバージョン 1.5 より, クライアント / サーバによる非集中開発をサポートしてい

る[3]. この機能により, ネットワークを介した分散共同開発が可能である. 本研究ではこの機能に注目し, クライアントとサーバの通信の間に CVS プロキシを導入して本システムを実現するアプローチをとる. 詳しくは第 4 章で議論する.

3.3.2 RCS ライブラリ層

この層は, RCS ファイルを操作するのに便利な機能一式を提供する. RCS ファイルを操作する機能を実装することを考えた場合, それ自体は普通のテキストファイルであるが, RCS 形式を解析できる機能が利用可能であると実装コストはずっと低くなる. この機能は, 上層が下層の RCS ファイルに CVS 層を通さずに直接アクセスする場合に, 容易に RCS ファイルの操作を行えるようにする目的がある. これにより, CVS が管理するよりも細粒度で中間成果物である RCS ファイルを管理することが可能となり, CVS が持っていないより高度な機能の実現を支援できる.

そのような高度な機能の例として, CVS リポジトリの階層型分散構成機能 [36] が考えられる. オープンソースソフトウェア開発プロジェクトは目的の異なるサブプロジェクトを持つ場合があり, そのサブプロジェクトではマスタプロジェクトと異なるリポジトリの運用方針や管理方針がとられることが多い. そうした場合に, CVS リポジトリの階層型分散構成機能があると, マスタとスレーブのリポジトリ間の関連を維持しながら, リポジトリごとに異なる運用方針や管理方針が設定可能になる. RCS ライブラリ層を用いると, RCS ファイルのリビジョン番号を書き換えたり, リビジョンヒストリを追加するといった, CVS を通してはできないような機能を容易に実装できるようになる. CVS リポジトリの階層型分散構成機能の実現にはこの他にもさまざまな機能が必要になると考えられるが, その中でも RCS ファイルを容易に操作できるようになる RCS ライブラリ層の存在は重要であると考えられる.

別の例としては, 管理下にある中間成果物についての情報を提供するアカウンティング機能 [9] がある. この機能は, 各種統計を計算したり, 中間成果物の状態を調査したり, 中間成果物やプロセスに関するレポートを作成したりする. 中間成果物である RCS ファイルを RCS ライブラリの提供する機能を使用して直接解析すれば, アカウンティング機能を比較的容易に実現できる.

一般のライブラリは本アーキテクチャの外に存在し, 各層は実装の際に必要に応じて各種ライブラリを利用する. しかし RCS ライブラリには上記のような本アーキテクチャが持つ拡張性の幅を大きく左右する特別な役割があるため, 層としてアーキテクチャ内に存在する意味がある.

RCS 形式のファイルを解析する機能を実装したライブラリは JRCS [1] などいくつか存在するので, 本研究のために新たに開発する必要はない.

[3]執筆現在, CVS の最新リリースバージョンは 1.11.5 である.

3.4 CVS アダプタ層

CVS のインタフェースから上層の基本システムフレームワーク層のインタフェースへの変換が主な役割である. つまり, この層は下層の CVS / RCS ライブラリ層の提供する機能のオブジェクト指向ラッパーとなる. この構成は Wrapper Facade [32] パターンに当てはまる. 図 3.2 に Wrapper Facade パターンのクラス図を示す. 図中の client クラスが基本システムフレームワーク, Wrapper Facade クラスが CVS アダプタ, Functions クラスが CVS / RCS ライブラリ層の提供する機能にそれぞれ対応する. client が Wrapper Facade オブジェクトのメソッドを呼ぶと, Wrapper Facade オブジェクトはカプセル化している Functions クラスの (場合によっては複数の) 機能 (関数) に転送する.

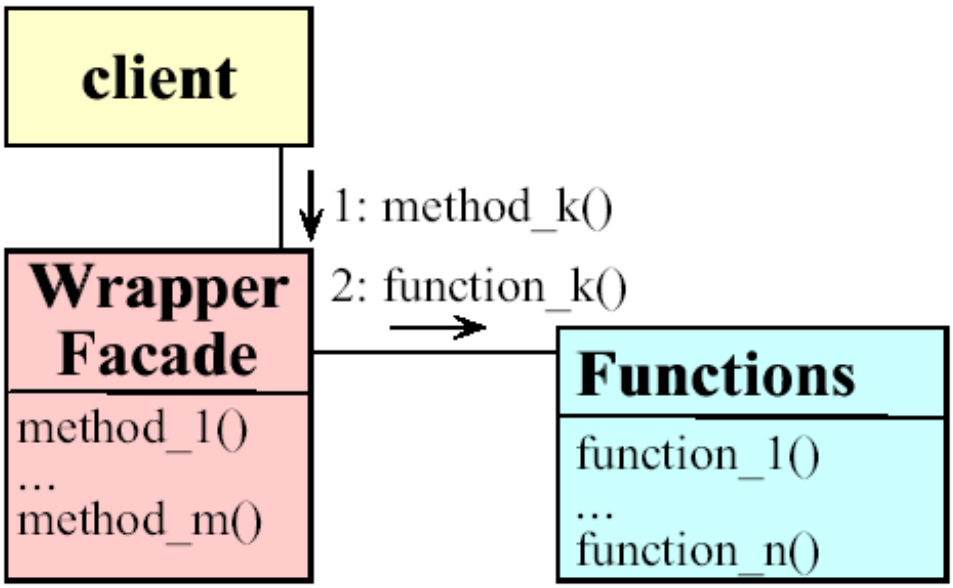

図 3.2: Wrapper Facade パターン

Wrapper Facade パターンは, 既存の非オブジェクト指向 API で提供される機能 (関数) やデータを, より簡潔性, ロバスト性, ポータビリティ, メンテナンス性の高いオブジェクト指向インタフェースにカプセル化する. 本アーキテクチャは CVS の設計や実装に直接依存しないように基本システムフレームワークを構成するため, および CVS と RCS ライブラリの提供する API を統一的に提供するために, このパターンを用いている. よって, 本層より上層ではオブジェクト指向技術を用いた開発手法 (方法論, 分析, 設計, プログラミング言語, その他ツールなど) を用いて, 下層の提供する機能の実装方法の詳細に引きずられることなく, 開発を行うことができる.

3.5 基本システムフレームワーク層

この層の役割は, 本アーキテクチャを採用するシステムの持つ基本的な機能を実装しつつ, その機能を拡張可能な構成にすることにより上層に対しその機能の拡張性を提供することである. つまり, 上層は新たな機能の追加や既存の機能のカスタマイズを行うことができる. その実現には, 第 2 章で議論したオブジェクト指向フレームワークの持つ拡張メカニズムが用いられる.

この層で実装する本アーキテクチャを採用するシステムの持つべき基本的な機能とは, 以下のようなものである. 図 2.1 に示した拡張性を実現する基本メカニズムを実現するには, 最低これらの機能が必要になる:

- **制御の流れを管理する機能**

  イベントループを持ち, システム全体の大きな制御の流れを管理する機能. システム外部もしくは内部で発生するイベントに反応する. システムの起動時や終了時に行う操作もここで実装する.

- **イベント定義機能**

  システムが反応すべきイベントを定義できる機能. ここで定義されているイベントに対してのみシステムは反応する.

- **タスクマネージャ機能**

  イベントハンドラの登録 / 削除を行う機能. イベントが起こった場合にそのイベントに対応したイベントハンドラを “適切に” 呼び出す機能. “適切に” という意味は, 一つのイベントに対応するイベントハンドラが複数あった場合それらの間の依存関係を考慮して適切な順番で呼び出さなければいけなかったり, 呼び出したイベントハンドラの実行に時間がかかる場合それを別スレッドで起動して処理をするなどといったタスク管理をうまく行うということである.

- **イベントハンドラ機能**

  イベントが発生した際に起動されるイベントハンドラの実装. タスクマネージャに登録する. 該当するイベントが発生すると, タスクマネージャからコールバックされる.

オブジェクト指向フレームワークはフローズンスポット部分とホットスポット部分の 2 つから成っており, これらのデフォルトの基本機能がフローズンスポット部分として実装される. ここでデフォルトとは, システムがあらかじめ提供しているという意味である.

同時に, これらの基本機能はホットスポット部分にもなっている. デフォルトの基本機能を実装しつつ, これらの基本機能のそれぞれの設計は機能拡張が可能な構造になっている. そのため, 新たな処理を追加できたり, デフォルトの処理を書き換えて, 基本機能の振る舞いを変化させることができる. この実装は差分プログラミングによって行われるので, プログラミングコストを削減できる.

また, ここで挙げた基本機能以外の基本機能を追加することもできる. それには, 一つ目の基本である “制御の流れを管理する機能” を拡張して, イベントループから新たにその基本機能を適当なタイミングで呼び出すようにすればよい.

基本機能以外にも, 上層が共通して使う機能をこの層に作り込んでおけば[4], 上層が機能拡張を行う際のプログラミングコストをさらに削減できる. 上層が共通に使う機能とは, ドメイン非依存の機能ということである. このように, 階層型オブジェクト指向フレームワークを採用して対象とするドメインによって層を分けることにより, 上層の拡張のためのプログラミングコストを削減することができるメリットが生まれる.

すなわち, この層は以上のようなホットスポット部分を上層に提供する. 上層では, 新たな機能の追加や既存の機能のカスタマイズが可能となる.

ここで強調しておきたいのは, システムを実現する上でこれらの基本機能の拡張が必須ではないということである. システムが実装しているデフォルトの振る舞いで満足できない時だけ必要に応じて拡張を行えばよい.

このような拡張可能な構成を実現することは, とりわけ困難なことではない. 2.3 節で議論したホットスポット部分の基本構造を用いれば個々の機能拡張は実現可能であるし, 2.4 節で議論したデザインパターンを用いたホットスポット部分の設計手法を用いればより柔軟で規模の大きいホットスポット部分を構成できる. 第 4 章では, 具体例を挙げて詳しく議論する.

3.6 拡張機能フレームワーク層

拡張機能の対象とするドメインに特化した機能を実装している, 階層型オブジェクト指向フレームワークの層である. したがって, この層の実装している内容および上層へ提供する機能は, 個々の開発プロジェクトによってさまざまである.

この層は, 前節の基本機能のリストの最後の “イベントハンドラ機能” の拡張である. 拡張機能フレームワークを用いて実装された拡張機能は, 基本システムフレームワーク層のタスクマネージャに登録される. そして, 対応するイベントが起こった際にその拡張機能が実行される.

この層以上の層の開発者には, ドメイン固有の知識が必要となる. この層の開発者は, 対象とするドメインのみをカバーするオブジェクト指向フレームワークを作成すればよく, そのドメインの分析に集中できる. 良い分析が行えれば良いホットスポット部分を設計することができるので, より狭いドメインの分析に集中できることはフレームワークの設計と実装のコストを低減している.

この層の役割は, 拡張機能の対象とするドメインに特化したイベントハンドラの実装のためのフレームワークを上層に提供することである. 上層では, そのホットスポット部分

[4] オブジェクト指向フレームワークの設計時に, あらかじめ組み込んでおくべき機能を完全に見極めるのは難しい. そのため, まず複数の拡張機能を作成してみて, それらの拡張機能に共通する機能を括り出してフレームワークに組み込むという手法を使うことが多い.

に差分プログラミングを行うことにより, 拡張機能を容易に実現できる.

本論文では第 6 章で, 拡張機能フレームワークの例として, アクセス権管理フレームワークを考える. このフレームワークは 2 つのアクセス制御モデルを実装しており, 開発プロジェクトはポリシーに応じてアクセス制御モデルを選択できる. それを拡張 / カスタマイズすることにより個々の開発プロジェクトの開発プロセスに適した具体的で実行可能なアクセス制御機能を実装できる.

3.7 拡張機能層

図 2.1 におけるフック関数にあたる層である. 言い換えると, オブジェクト指向フレームワークのホットスポット部分の実装である. 基本システムフレームワークや拡張機能フレームワークの提供する機能を利用し, 開発プロジェクトの要求するプロセス支援機能を実現する.

本アーキテクチャでは拡張機能の実現のために, 拡張機能フレームワークを使わずに直接基本システムフレームワークに拡張機能を追加する方法, あらかじめ用意されている拡張機能フレームワークに拡張機能を追加する方法, 拡張機能フレームワークを作成しそれに拡張機能を追加する方法の 3 通りの方法をサポートしている.

- **拡張機能フレームワークを使わずに直接基本システムフレームワークに拡張機能を追加する方法**

  比較的簡単な機能を迅速に拡張したい場合に有効である. 一般に再利用性や拡張性の高いフレームワークを作成することには時間と労力がかかる. 開発プロジェクトの方針や拡張機能の種類によっては, そういった “良い” フレームワークを時間をかけてつくるよりも拡張機能を迅速に実現してソフトウェア開発を押し進めたいことも多々ある. たとえば, 短期間で終了することがわかっている開発プロジェクトや, コードが汚くてもよいのでとにかく動くコードをなるべく早く実装して開発に使用し, 開発が終わってしまえばそのコードを捨ててしまうような場合である.

- **あらかじめ用意されている拡張機能フレームワークに拡張機能を追加する方法**

  一番推奨される方法である. 多くの開発プロジェクトに必要なプロセス支援機能の主なものをあらかじめ拡張機能フレームワークに用意しておき, 開発プロジェクトがそれを拡張 / カスタマイズを行う. この方法はプログラミングコストを押さえながらかつ比較的大きな機能を実現できる.

- **拡張機能フレームワークを作成しそれに拡張機能を追加する方法**

  拡張機能フレームワークに用意されていないある程度の規模のプロセス支援機能を開発プロジェクトが要求する場合である. 拡張機能開発者が自ら拡張機能フレームワークを作成する. この場合, 1 つ目の方法の様に基本システムフレームワークのホッ

トスポット部分に書き捨てのコードを書いて拡張機能を実現することも可能だが, 状況が許すのならオブジェクト指向フレームワークのかたちで再利用性や拡張性の高い設計 / 実装を行っておけば, 将来このフレームワークを拡張機能フレームワークとして再利用することができる. この方法は, 実現したい拡張機能のドメインが広い場合有効である. たとえば, そのドメインにおける有力な解法が複数ありどれを選択するかの決断を拡張機能実装者に委ねたい場合や, そのドメインにおける解法をフレームワークとして構成した方が将来, 再利用性, 拡張性, メンテナンス性などの面で有利になる場合などが考えられる.

これらの方法には, 拡張機能の実現のためにかかるコストと拡張機能のソフトウェア品質とのトレードオフがある. 開発プロジェクトはこのトレードオフを考慮し, プロジェクトの目的に応じて拡張方法を選ぶことができる. いずれの方法をとったとしても, オブジェクト指向フレームワークの利点は生きており, 拡張性の実現と差分プログラミングによるプログラミングコストの削減を行うことができる.

第4章　プロトタイプシステムの設計

この章では, 前章で提案したレイヤアーキテクチャを用いたシステムが実現可能であることを示すために, プロトタイプシステムを設計 / 実装した. このプロトタイプシステムは, 前章で議論したレイヤアーキテクチャのすべての特徴を完全に実装したわけではないが, 現段階の設計 / 実装でオブジェクト指向フレームワークを利用してシステムの拡張性を実現する本手法の利点を示すことができていると思われる. 機能拡張の容易さについては, 次章で議論する.

本章で論じるプロトタイプシステムの設計と実装は, 図 3.1 の第 1 層 (RCS ファイル層) から第 4 層 (基本システムフレームワーク層) までである. 特に, 第 3 層の CVS アダプタ層と第 4 層の基本システムフレームワーク層の設計と実装を中心に議論する. 基本システムフレームワーク層の設計には, 第 2 章で議論したオブジェクト指向フレームワークのホットスポット部分の設計手法 (デザインパターン, メタパターンなど) を用い, 十分な拡張性を持つクラス構成を提案する.

まず, プロトタイプシステムの実現のために, 本研究のとった設計方針について説明し, 実装アーキテクチャを示す. 次に, CVS プロキシの設計と基本システムフレームワーク層にあたるオブジェクト指向フレームワークの設計 (クラス図) を示す. そのフレームワーク内の主なホットスポット部分について, 具体的なコード例を示しながら, 基本システムフレームワーク層の提供する拡張性について議論する.

4.1　CVS のクライアント / サーバ機能と CVS プロトコル

最初に、本章で述べる設計の前提となっている CVS のクライアント / サーバ機能と CVS プロトコルについて説明する.

CVS は, ネットワークを介した分散共同開発を可能にするためにクライアント/ サーバ機能を持っている. 開発者たちはネットワーク上のどこからでもリポジトリにアクセスできる. これにより, 複数の開発者たちによる同時開発が可能となっている.

CVS は, 集中型リポジトリモデルで実装されている. つまり, 開発プロジェクトは中央に一つのリポジトリをもち, クライアントはそのリポジトリから中間成果物をチェックアウト (checkout) して, ネットワークを通じて中間成果物のコピーをクライアントのマシンのワーキングディレクトリの中に持ってくる. 開発者は, 自分のローカルマシンで開発を行う. 開発終了後, ネットワークを通じてサーバと通信をして, サーバのマシン上のリポジトリに作業内容をコミット (commit) する. つまり, 3.3.1 節で議論した, “コピー-修正-マー

ジ” モデルの “コピー” と “修正” をネットワークを通じて行っていることになる.

クライアントとサーバは, CVS プロトコルによって通信を行う. CVS は初期設計からネットワークに対応した設計になっていなかった. その後, 初期設計を引きずったままクライアント / サーバ機能を追加したために, もともと一つであった CVS のクライアントインタフェースと処理の実行部分をクライアントとサーバに分けたかたちとなっている. そのため, クライアントとサーバ間の通信に使われる CVS プロトコルは, あまり効率の良いものとはなっていない.

CVS プロトコルは, クライアントからの通信である “リクエスト” とサーバからの通信である “レスポンス” からなる. CVS プロトコルは, 一部の例外を除いて基本的に次のような形式で通信が行われる.

```
COMMAND[ ARGUMENT]\n
```

COMMAND は, CVS のインタフェースである checkout や commit などのコマンドではなく, プロトコル内部で使われる内部コマンドである. また, COMMAND は引数をとる場合もある. コマンドは必ず “\n” で終わる.

クライアントからのリクエストは, 大文字で始まるコマンド (Root, Valid-responses, Directory など) はサーバの応答を期待しないコマンドで, 小文字で始まるコマンド (ci, diff, log など) はサーバの応答を期待するコマンドである. サーバからの応答は, Updated, Created, Merged などのコマンドを用いて, サーバで行われたリクエストに対応する処理の結果をクライアントに帰す.

このプロトコルの特徴は, CVS のインタフェースレベルがサポートしているコマンドがクライアントからサーバに送られる際に, CVS プロトコルレベルのより細かい内部コマンドの組み合わせになって CVS サーバに送られるということである. たとえば, あるプロジェクトのリポジトリからチェックアウトしてローカルのワーキングディレクトリにコピーしてきたばかりの (何も変更を加えていない), foo.h というファイルに対して update コマンドを次のように発行すると,

```
$ cvs update foo.h
```

クライアント (C と略す) からサーバ (S) への通信は, CVS プロトコルで定義された内部コマンドを用いて以下のように行われる[1].

```
...
C: UseUnchanged\n
C: Argument -u\n
C: Argument --\n
C: Directory .\n/src/FooProject\n
```

[1]実際には, この通信の前にリポジトリに対するクライアントのユーザ認証を行う通信が入る. この例では省略した.

```
C: Entry /foo.h/1.74///\n
C: Unchanged foo.h\n
C: Argument foo.h\n
C: update\n
S: ok\n
```

4.2 設計方針

プロトタイプシステムの実装アーキテクチャとして, CVS プロキシを用いた構成をとる. CVS のクライアント / サーバ機能に注目しクライアントとサーバとの間にプロキシをはさみ込むことにより, レイヤアーキテクチャにおける CVS アダプタ層が上層のフレームワーク層と下層の版管理層を分離している構成を実現する. CVS アダプタ層に相当するプロキシ内部のオブジェクトが, 拡張性の実現のためのフレームワーク構成部分に対し CVS クライアントと CVS サーバへの通信の詳細を隠蔽する. これにより, CVS の通信プロトコルの詳細に依存せずに, フレームワークの開発を行うことができる. CVS プロキシをオブジェクト指向フレームワークで実装し, 拡張可能なホットスポット部分に機能を追加していくという実装アプローチは, 本プロトタイプシステムの大きな特徴である.

本研究では, CVS を版管理システムとして採用したレイヤアーキテクチャを実装するプロトタイプシステムを設計するために, 2 つの方針をとった.

一つめは, CVS には変更を加えないということである. 拡張機能を追加できるように CVS 自体のソースコードを変更して CVS のシステムを再構成するということはしない. CVS は前小節で述べたように, 分散共同ソフトウェア開発をサポートする良い特徴を備えている. これらの CVS の持つ版管理システムとしての機能をブラックボックスとして本プロトタイプシステムに再利用する. これにより, 版管理システムとして持っていなければならない多数の機能を本プロトタイプシステムが実装しなくてよくなり, 実装コストを大幅に削減できる. また, 現在広く利用されている CVS に対し全く変更を加えないで拡張可能な版管理システムが実現可能であれば, 将来, 本手法を用いたソフトウェアが CVS を利用している多くの開発プロジェクトで広く利用される可能性があり, CVS に変更を加えるアプローチよりも社会的インパクトが大きいと考えられる.

二つめに, CVS クライアントとサーバとの間にプロキシを入れる実装アーキテクチャを採用する. 図 4.1 に実装アーキテクチャを示す. プロキシは, クライアントとサーバ間で行われる通信をすべて解析し, その内容により必要に応じて拡張機能を呼び出す. その拡張機能は基本システムフレームワークによりオブジェクト指向フレームワークのホットスポット部分を拡張して実装されるために, プログラミングコストが少なくてすむ. CVS クライアントが動作しているホスト A はプロキシが動作しているホスト B とは別だが, ホスト B は CVS サーバが動作しているホスト C と同一であっても別であってもよい. 同一のホストの場合には, プロキシとサーバはネットワークのサービスポート番号が異なる番号を使う必要がある.

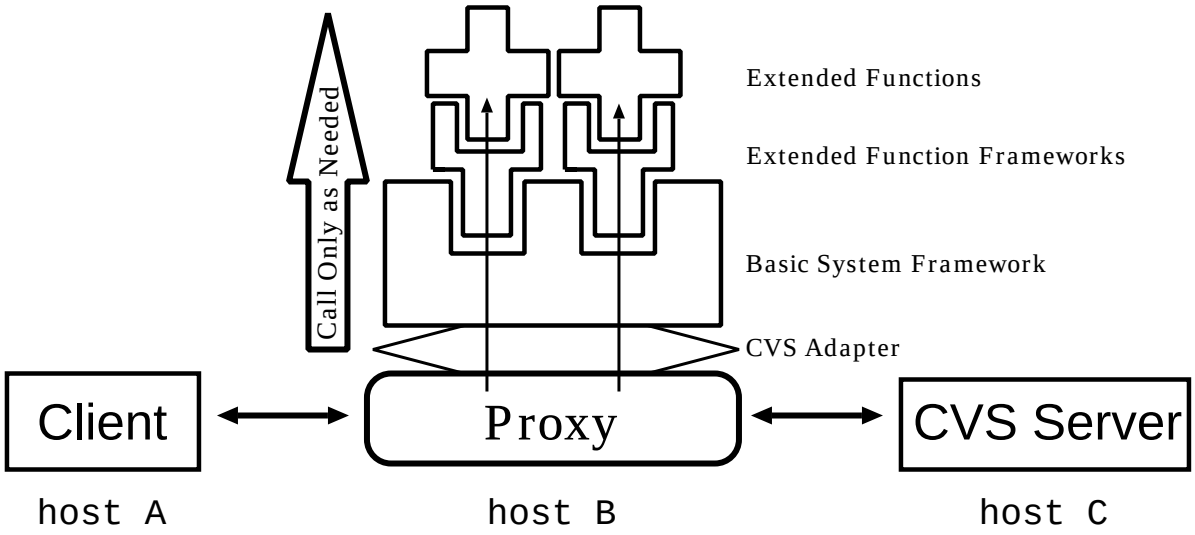

図 4.1: プロトタイプシステムの実装アーキテクチャ

プロキシを拡張可能な構成にして張機能を追加できるようにすることにより, 本プロトタイプシステムは, 一つめの設計方針である CVS には変更を加えないでその機能を再利用することができる. つまり, CVS クライアントとサーバは, プロキシの存在を意識する必要はない. お互いに直接通信しているのと同じ状況である.

前小節で議論したように CVS においては, CVS プロトコルでは CVS インタフェースのコマンドよりも細かい内部コマンドを使用しているので, プロキシにおいてきめ細かい制御が可能となる.

この実装アーキテクチャにおいて, 第 3 章で提案したレイヤアーキテクチャの 1 層と 2 層が CVS クライアントとサーバに相当することになる.

プロキシを用いる本実装アーキテクチャをとることにより, 既存の CVS で分散共同開発を行っているプロジェクトは, 本プロトタイプシステムに簡単に移行できる. CVS サーバの代わりにプロキシをおき, CVS サーバを別のマシンに移すだけで本論文で提案している機能が利用できるようになる.

次節では, これらの設計方針に従った CVS プロキシの設計を示す.

4.3 オブジェクトを用いた設計の概要

まず, 図 4.2 にオブジェクトを用いたプロトタイプシステムの設計 (概念図) を示す. この図は, 本システムの動作モデルにもなっている. 大まかな動作は, クライアントと CVS

サーバとの間にプロキシを置き, プロキシがその通信内容である CVS プロトコルを解析する. その際, 必要に応じて拡張機能を起動するという流れになる.

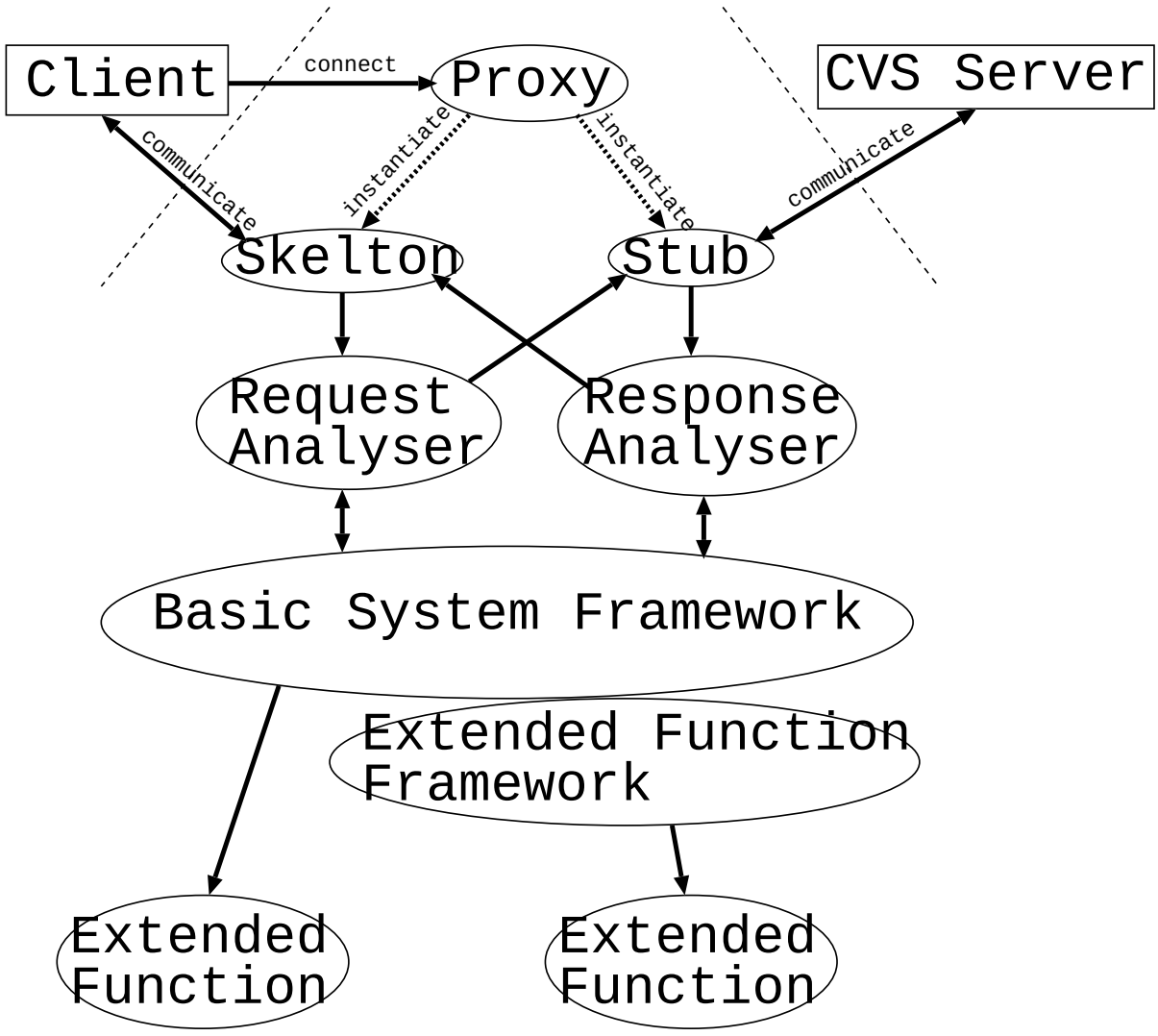

図 4.2: オブジェクトを用いたプロトタイプシステムの設計 (概念図)

プロキシオブジェクトが生成されシステムが起動すると, プロキシオブジェクトは, リクエストアナライザオブジェクト, レスポンスアナライザオブジェクト, 基本システムフレームワークを構成するオブジェクト群, 拡張機能フレームワークを構成するオブジェクト群, 拡張機能オブジェクトをそれぞれ生成する.

プロキシオブジェクトは, クライアントからの接続要求を受けたらスケルトオブジェクトンとスタブオブジェクトを生成する. スケルトンオブジェクトは, クライアントとの通信

方法の詳細をカプセル化したオブジェクトであり, プロキシのモデルの中でクライアントそのものとして振る舞う. 同様に, スタブオブジェクトは CVS サーバとの通信方法の詳細をカプセル化しており, プロキシのモデルの中で CVS サーバそのものとして振る舞う. これらのオブジェクトは, 一つのクライアントからの要求を処理し終わると開放される. クライアントからの接続があるたびにこれらのオブジェクトは生成される. クライアントまたは CVS サーバとの接続の期間とオブジェクトの生存期間を同じにすることにより, プロキシオブジェクトの設計ドメインにおいてスケルトンオブジェクト / スタブオブジェクトをクライアント / サーバそのものとしてモデル化できる.

スケルトンオブジェクトはクライアントから送信されたリクエストを, CVS プロトコルを解析するリクエストアナライザオブジェクトに送る. ここで CVS プロトコルの内部コマンドレベルで解析が行われる. 一つの内部コマンドの解析を終了したらその結果を基本システムフレームワークに送る. 基本システムフレームワークでは, その解析結果と登録されている拡張機能との対応を検討し, 必要に応じて拡張機能を呼び出して実行する. これをクライアントからのリクエストをすべて解析し終わるまで繰り返す. 拡張機能オブジェクトは, 基本システムフレームワークから直接呼び出される場合と, 拡張機能フレームワークを介して呼び出される場合がある. 後者は, 拡張機能オブジェクトが拡張機能フレームワークを利用して実装されている場合である. リクエストの解析と呼び出すべき拡張機能オブジェクトをすべて実行した後, 基本システムフレームワークは, リクエストアナライザオブジェクトから受け取ったリクエストをもう一度リクエストアナライザオブジェクトに帰す. リクエストアナライザオブジェクトは基本システムフレームワークから渡されたリクエストをスタブオブジェクトに渡し, スタブオブジェクトは CVS サーバにそれを転送する. これでクライアントからのリクエストはプロキシ経由で CVS サーバに送られたことになる.

CVS サーバからのレスポンスは, クライアントからのリクエストをプロキシが処理したのと同じ要領で処理が行われる. CVS サーバは, 普通のクライアントから直接リクエストを送られた場合と同じように内部で処理を行い, その結果をスタブオブジェクトに送る. スタブオブジェクトはそれをレスポンスアナライザオブジェクトに送り, そこで CVS プロトコルの内部コマンドレベルまで解析される. その結果を基本システムフレームワークに送り, 必要に応じて拡張機能が実行される. その後, レスポンスが基本システムフレームワークからレスポンスアナライザオブジェクトを経由してスケルトンオブジェクトに渡され, 最終的にクライアントに転送される.

一見, リクエスト / レスポンスアナライザオブジェクトがクライアントからのリクエスト内容を基本システムフレームワークまで転送するのは無駄のようにみえる. だが, このようにオブジェクトのコラボレーションを構成する二つの理由がある. 一つめは, クライアントからのリクエストや CVS サーバからのレスポンスに応じて拡張機能を呼び出すだけではなく, そのリクエスト / レスポンス自体を拡張機能を実行することにより変更することを可能にするためである. クライアントと CVS サーバ間で通信内容に CVS プロトコルレベルで変更を加えることにより, クライアントに対し現在の CVS が提供している機

能を拡張できる. もちろん, CVS プロトコルがサポートしていない通信を行えば, 既存の CVS クライアントはそれに対応できない. したがって, このような CVS プロトコルレベルの拡張を行う場合には CVS クライアントに変更を加えなければならなくなる. これは前節で述べた, CVS 自体には変更を加えないという設計方針に反する. しかし CVS クライアントには変更を加えず, それに拡張プロトコルをサポートするラッパーを被せるアプローチをとるとよって, 設計方針に反しないで CVS クライアントとプロキシとの間の通信プロトコルを拡張する方法を実現できる.

二つめの理由は, 基本システムフレームワークが CVS に強く依存することを避けるためである. 本構成では, 基本システムフレームワークが依存しているのはリクエスト / レスポンスアナライザオブジェクトの解析後の出力だけである. リクエスト / レスポンスアナライザオブジェクトには, CVS に強く依存した部分と, 基本システムフレームワークや拡張機能フレームワークのオブジェクト指向フレームワーク部分との間の関連を粗結合にしている効果がある. つまり, リクエスト / レスポンスアナライザオブジェクトは, CVS アダプタ層の役割を果たしている. スケルトンオブジェクトとスタブオブジェクトは CVS 層, 基本システムフレームワークは基本システムフレームワーク層, 拡張機能フレームワークは拡張機能フレームワーク層, 拡張機能オブジェクトは拡張機能層の役割をそれぞれ果たしている. 本プロトタイプシステムのオブジェクト構成は, 第 3 章で提案したレイヤアーキテクチャをまさに体現したものとなっている.

4.4 実装方針

本プロトタイプシステムでは, 以下を前提として開発を行った.

CVS のアクセス方式

CVS をクライアント / サーバで動作させる際, リポジトリへのアクセス方式を指定しなければならない. CVS はいくつかのアクセス方式をサポートしている. 将来的には複数のアクセス方式に対応したいが, 実装上のコストを考えて本プロトタイプシステムでは pserver 方式のみを対象とする.

CVS プロキシの I/O モデルと並行制御

本プロトタイプシステムは, CVS プロキシを用いる実装アーキテクチャを用いている. CVS プロキシの I/O モデルとして, 基本的に I/O Multiplexing モデルを用い, 通信処理の効率向上のために Nonblocking I/O モデルを部分的に組み合わせている. 本プロトタイプシステムがクライアントからの同時接続要求を受けられるのは 1 つのみである. つまり, あるクライアントがプロトタイプシステムと通信中だった場合, 他のクライアントはプロト

タイプシステムと通信できない. 実装上のコストを考えて本プロトタイプシステムでは複数のクライアントからの同時接続をサポートしていない.

拡張機能の間の連携 / 依存関係

拡張機能が増えてくると, 拡張機能間の依存関係や連携の問題が発生する. 将来的には何らかの管理機能が必要となると思われるが, 本プロトタイプシステムでは, そのような問題はすべてフレームワーク開発者の責任において行われるものとする.

拡張機能のロードタイミング

ソフトウェアには, 以下のような拡張機能のロードタイミングがある:

- 静的
  - コンパイル時
    ソースコードに拡張機能を埋めこむ. 構成を変更するには, 再コンパイルが必要となる.
  - インストール時
    拡張機能をインストールする際に, データベースに登録し内容を更新する. 構成を変更するには, 再インストールが必要となる.
- 動的
  - 起動時
    システムを起動する際に拡張機能をロードする. OS の提供するダイナミックリンク機能や拡張機能の場所やシンボルが書かれている設定ファイルを使ったりする方法がこれに該当する. 構成を変更するには, 再起動が必要となる.
  - 実行時
    実行環境やプログラミング言語の提供するリフレクションまたはイントロスペクション機能を利用し, 拡張機能をロードする. システムが稼働したまま構成を変更することができる.

本プロトタイプシステムでは, 拡張機能をオブジェクト指向フレームワークに組み込むアプローチをとっているので, 拡張機能のロードタイミングはコンパイル時である.

4.5 実装環境と使用ツール

プロトタイプシステムの実装を以下ような環境やツールを用いて行った.

- 版管理システム: CVS 1.11.4 (client/server), CVS Client/Server Protocol 1.11
- プログラミング言語: Ruby 1.8.0
- 使用ライブラリ: ruby-zlib 0.5.1, ruby-tcpwrap 0.3
- OS: NetBSD/i386 1.6L

オブジェクト指向スクリプト言語 Ruby [24] を実装言語として選んだのには理由がある. 拡張性や再利用性の高いオブジェクト指向フレームワークを実現するには, 何度も設計と実装を繰り返すことが要求される [35]. 初期のフレームワーク設計段階ですべてのホットスポット部分とそのクラス構成を完全に読み切ることはほぼ不可能である. そのため, 設計と実装のサイクルを早く回すことができる方が開発効率が良い. コンパイル型言語よりインタプリタ型言語の方がプロトタイプシステムの実装には適していると考えられる. 実行効率より開発効率を優先し, スクリプト言語を選択した.

また, Ruby は次のような特徴を持つプログラミング言語である: 純粋なオブジェクト指向言語 (例外なくすべてがオブジェクト), 書きやすくて読みやすい文法, 豊富なクラスライブラリ, ネットワークプログラミングが得意, メモリ管理が不要, 移植性が高い. これらの特徴は, CVS プロキシとして動作するプロトタイプシステムの実装を大いに支援する.

第5章　CVS アダプタと基本システムフレームワーク

次に, CVS プロキシを用いた本プロトタイプシステムの設計のクラス図を図 5.1 に示す. 本プロトタイプシステムは, 拡張性の実現のために複数のホットスポット部分を持つオブジェクト指向フレームワークで設計されている. 図 4.2 とは異なり, 基本システムフレームワークのホットスポット部分のクラス構成を中心に表している.

図 5.1 中のクラスは, CVS プロトコルを中継するプロキシ機能を実現している部分と, 複数のホットスポット部分を持っているフレームワーク部分に大きく 2 つに分けられる. それらは, 第 3 章で提案したレイヤアーキテクチャの第 3 層と第 4 層にそれぞれ対応する. 以下では, 順にこれらの層が図中でどのように設計されているかについて述べる. さらに, それぞれの層の中をクラスの役割を基準に部分ごとに分けて, 拡張性を実現するクラス構成について個別に議論する.

5.1　CVS アダプタ

図中の以下のクラスがこの層に含まれる: *Adapter*, CVSAdapter, CVSProtocolAnalyser, CVSRequestAnalyser, CVSResponseAnalyser, CustomisedCVSRequestAnalyser.

Adapter は抽象クラスである. このクラスは, 下層の版管理システム層のインタフェースを上層のオブジェクト指向フレームワークのインタフェースに変換する “アダプタ” として持っているべき属性や操作のシグネチャを定義している. このクラスの *request* メソッドは, 下層の版管理システムからの出力である “メッセージ” を上層のオブジェクト指向フレームワークが理解できる “イベント” に変換する役割を持つ. このクラスを継承してさまざまな版管理システムに対応するアダプタをつくることができる. *Adapter* が抽象クラスであるのは, CVS だけに依存せずそれ以外の版管理システムに対しても本研究のとっている拡張アプローチを適用できるようにするためである. たとえば FooVCS という版管理システムがあった場合, FooVCS のインタフェースを変換するクラスを *Adapter* クラスを継承して FooVCSAdapter クラスとして作成すればよい. *Adapter* クラスの存在により, 本設計はアダプタクラスを作成できる限りにおいて, 版管理システムを交換可能にできる構成である. この拡張性は, Main クラスと *Adapter* クラスに対して, 図 2.7 (b) に示した 1:1 Connection メタパターンを適用したことによって実現している. *Adapter* が hook class である.

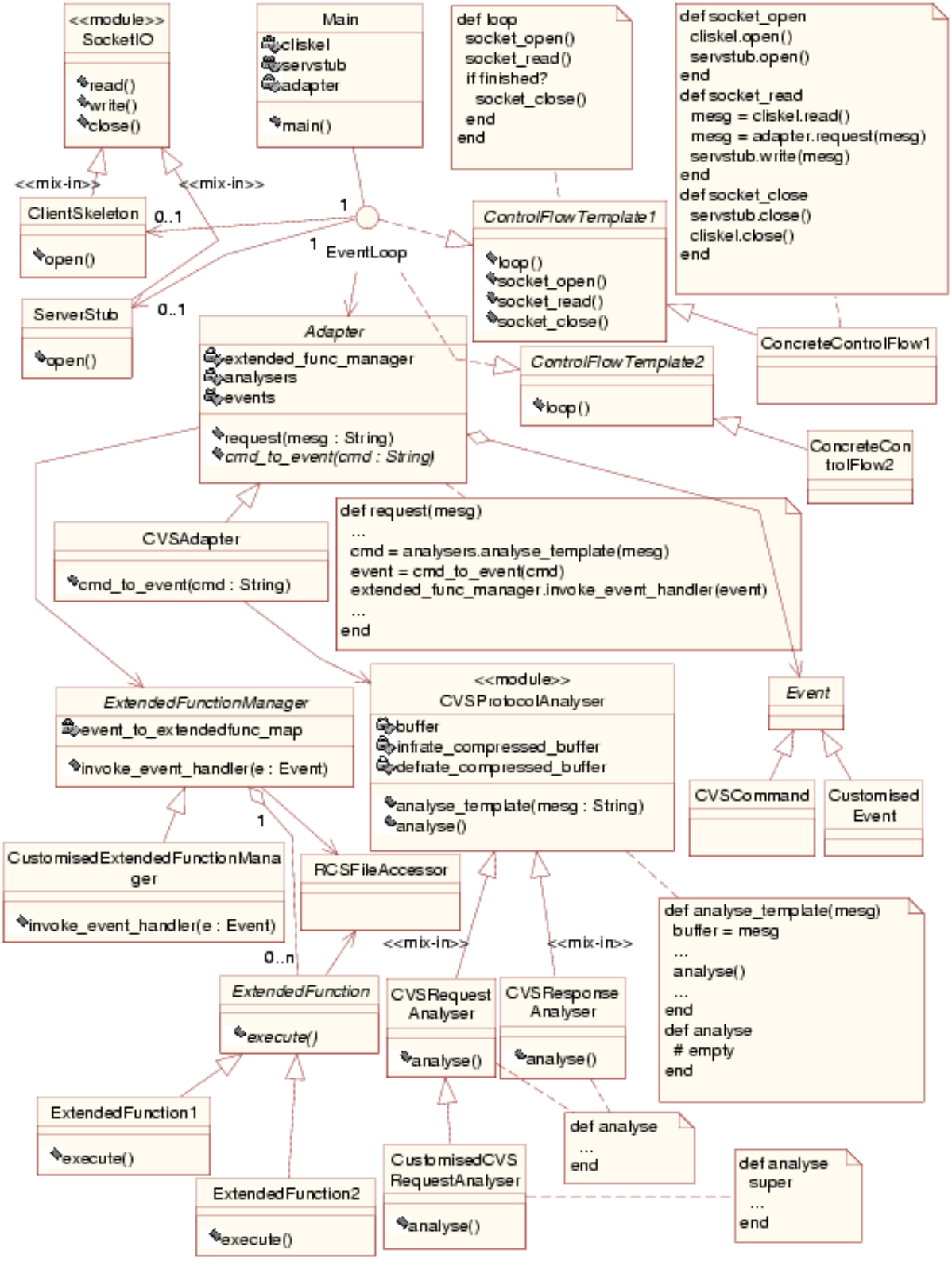

図 5.1: プロトタイプシステムの CVS アダプタ層と基本システムフレームワーク層の設計 (クラス図)

本研究では, 版管理システムとして CVS を対象としているので, *Adapter* クラスを継承して具象クラス CVSAdapter を定義している. 抽象クラス *Adapter* の *request* メソッドには図中のノートに示したコード断片のような複数のプリミティブなメソッド呼び出しの流れをテンプレートとして実装している. ここには, 図 2.3 に示した Template Method パターンと図 2.7 (a) の Unification メタパターンを適用している. つまり, *Adapter* クラスの *cmd_to_event* メソッドが hook method となっており, このメソッドの実装をサブクラスでカスタマイズできる.

CVSAdapter は下層の CVS のインタフェースを理解しなくてはならない. 本プロトタイプシステムにおいては, プロキシを用いる実装アーキテクチャを採用しているので, CVS プロトコルの解析ができればよい. その役割を負っているのが, CVSProtocolAnalyser クラスを継承している CVSRequestAnalyser と CVSResponseAnalyser である. 図 4.2 で示したように, CVSRequestAnalyser はクライアントからのリクエストを, CVSResponseAnalyser はサーバからのレスポンスをそれぞれ解析するクラスである. これらのクラスのスーパークラスである CVSProtocolAnalyser は, CVS プロトコルの解析に必要となる各種バッファや, 解析の基本的な処理の流れをテンプレートとして実装しているメソッド (analyse_template) を実装している. 実際の解析の処理内容はクライアントからのリクエストかサーバからのレスポンスかで異なるので, 図中のノートのコード断片のように, analyse メソッドを hook method として Template Method パターンを適用している. CVSRequestAnalyser と CVSResponseAnalyser はデフォルトで標準の CVS プロトコルを解析できる実装を持っているが, さらに CVSRequestAnalyser クラスを継承して CustomisedCVSRequestAnalyser クラスのように analyse メソッドをオーバーライドして独自に拡張した CVS プロトコルを解析できるようにすることができる. analyse メソッドの実装の際, 図中のノートのコード断片のように, メソッド内部で super を呼ぶことによりスーパークラス CVSRequestAnalyser が実装している標準の CVS プロトコルのリクエスト解析の実装を再利用しつつ独自プロトコルの解析の実装を追加することができる. このように差分プログラミングを行うことにより, 少ないプログラミングコストで CVS プロトコルの解析エンジンを拡張することができる.

CVSAdapter にとって CVSProtocolAnalyser が hook class となっており, さまざまな CVS プロトコル解析エンジンを利用することができる. この部分の拡張性の実現のためのクラス設計には, 1:N Connection メタパターンが適用されている.

まとめると CVS アダプタ層では, 版管理システムに対応したアダプタの種類と版管理システムのインタフェースを解析するエンジン部分が拡張可能なホットスポットとして設計されている.

5.2 基本システムフレームワーク

この層には, CVS アダプタ層に含まれていない残りのクラスが含まれる: Main, EventLoop, *ControlFlowTemplate1*, ConcreteControlFlow1, *ControlFlowTemplate2*, ConcreteControlFlow2, SocketIO, ClientSkeleton, ServerStub, *Event*, CVSCommand, CustomisedEvent, *ExtendedFunctionManager*, CustomisedExtendedfunctionManager, *ExtendedFunction*, ExtendedFunction1, ExtendedFunction2, RCSFileAccessor.

これらのクラスの多くは拡張性を提供するため, オブジェクト指向フレームワークのホットスポット部分を構成するように設計されている. 以下では, 3.5 節で述べた基本システムフレームワーク層の持つべき 4 つの機能ごとにクラス構成を説明する. これらの機能をオブジェクト指向フレームワークのホットスポット部分として設計することで, 拡張性の高い版管理システムを構成できる.

- **制御の流れを管理する機能**

  この機能の実現には, Main, EventLoop, *ControlFlowTemplate1*, ConcreteControlFlow1, *ControlFlowTemplate2*, ConcreteControlFlow2 のクラスが関係する.

  Main クラスは EventLoop インタフェースを利用することにより, 制御の流れをカスタマイズできる. この構造には, EventLoop が hook class であるような 1:1 Connection メタパターンを適用している. EventLoop インタフェースの実装には Template Method パターンを用いる. たとえば, 抽象クラス *ControlFlowTemplate1* ではメソッド loop 内でプリミティブなメソッド (socket_open, socket_read, socket_close など) を呼び出す制御の流れを定義している. これらのプリミティブなメソッドは, ConcreteControlFlow1 で図中のノートに示しているように, サブクラスで詳細に実装されカスタマイズが可能である.

  また, *ControlFlowTemplate1* とは違った制御の流れにしたい場合は, EventLoop インタフェースを実装して別の抽象クラス *ControlFlowTemplate2* をつくり, メソッド loop を定義すればよい. たとえば, メソッド loop は次のように定義できる.

```
def loop
  pre_process()
  process()
  post_process()
end
```

  この実装は, *ControlFlowTemplate1* のメソッド loop のソケット入出力操作に注目した制御の流れの実装とは異なり, 処理本体とその前処理と後処理という関係に注目した制御の流れを実装している. この場合, pre_process, process, post_process がプリミティブなメソッドとなっており, サブクラス ConcreteControlFlow2 でそれぞれ詳細が実装される.

- **イベント定義機能**

  この機能の実現には, *Event*, CVSCommand, CustomisedEvent が関係している.

  Adapter は抽象クラス *Event* を集約しており, *Event* を継承することにより, さまざまなイベントをカスタマイズできる. たとえば, checkout や commit などといった CVS のコマンドを単位としたイベントを定義している CVSCommand, CVS プロトコルの内部コマンドを単位とした CVSProtocolCommand, 本フレームワークの拡張者が独自に定義した CustomisedEvent などとして拡張できる. この拡張性の実現のための設計には, Strategy パターンと 1:N Connection メタパターンを *Adapter* と *Event* に適用している. 抽象クラス *Event* が hook class となっている.

- **タスクマネージャ機能**

  この機能は, *ExtendedFunctionManager* と CustomisedExtendedFunctionManager で実現される.

  抽象クラス *ExtendedFunctionManager* は, シンプルな機能をデフォルトで実装している. それは, あるイベントが起こった時その通知を受け取って, それに対応する拡張機能を呼び出すというものである. 拡張機能間の呼びだし順序やそれらの間の依存関係を考慮しなければならないなどといった, より複雑な制御機能が必要になる場合には, サブクラス CustomisedExtendedFunctionManager をつくりメソッド invoke_event_handler をオーバーライドして機能拡張を行う. *Adapter* と *Extended-FunctionManager* との関係には, *ExtendedFunctionManager* が hook class となる 1:1 Connection メタパターンを適用している.

- **イベントハンドラ機能**

  この機能の実現には, *ExtendedFunction*, ExtendedFunction1, ExtendedFunction2 が関係する.

  本フレームワークを利用する拡張機能は, 必ず抽象クラス *ExtendedFunction* のサブクラスでなければならない. *ExtendedFunction* は抽象メソッド execute を持っており, サブクラスはメソッド execute の実装として拡張機能を実装する. *ExtendedFunction-Manager* は *ExtendedFunction* をコンポジションしており, この設計には Command パターンと *ExtendedFunction* が hook class である 1:N Connection メタパターンを適用している.

特に重要なのは, 一つめの “制御の流れを管理する機能” である. 現在ソフトウェアに拡張性を与えるメカニズムとしてよく使われているものに, Apache [15] などが採用しているプラグインアーキテクチャ [20] がある. このアーキテクチャには, 実行時に動的にソフトウェアの構成を変更することができ機能拡張が可能であるという利点があるが, 2.1 節で議論したエントリポイントは固定しており, あらかじめ決められたポイントでしか機能を拡張することができない. しかし, オブジェクト指向フレームワークを利用して実装さ

れている基本システムフレームワークの提供する “制御の流れを管理する機能” により, 実行時に動的に変更することはできないが, エントリポイントの数や場所を自在に拡張できる. この機能により, さまざまな拡張機能要求に柔軟に対応することができる.

以上のホットスポット部分以外に, CVS プロキシとして動作するために必要なクラスが基本システムフレームワーク層には組み込まれている. その代表は SocketIO, ClientSkeleton, ServerStub と RCSFileAccessor である. SocketIO は, ネットワークソケットへの入出力を機能を実装している有用なメソッド群を持っているクラスである. これを継承して, クライアントとの通信を担当する ClientSkeleton, サーバとの通信を担当する ServerStub がモデル化されている. また, RCSFileAccessor は下層の RCS ライブラリ層の提供する機能へのインタフェースとなる機能を実装している. ExtendedFunctionManager と *ExtendedFunction* のサブクラスが必要に応じて RCSFileAccessor を利用して CVS リポジトリ内の RCS ファイルにアクセスを行う.

第6章　拡張機能フレームワークと拡張機能

この章では, 前章で述べたプロトタイプシステムを基本とするレイヤアーキテクチャの第5層と第6層の設計と実装を示す. 拡張機能としてリポジトリに対するアクセス制御機能を取り上げる. 複数のアクセス制御モデルをサポートするオブジェクト指向フレームワークを設計し, これを拡張機能フレームワークとして構成する. 拡張機能フレームワークを利用して実装する拡張機能として2種類のアクセス権管理機能の設計と実装を行い, そのプログラミングコストについて評価を行う. 最後に, 本プロトタイプシステムの対象とする拡張機能について明らかにする.

6.1　アクセス権管理機能の概要

アクセス権管理機能とは, どの開発者にリポジトリ内のどの中間成果物に対してどのようなアクセスを許可するのかをプロジェクト管理者が設定でき, かつその設定によって開発者のリポジトリに対するアクセス制御が行われる機能である. ソフトウェア開発は開発者がリポジトリ内の中間成果物にアクセスすることによって進んでいくので, そのアクセスを制御するアクセス権管理機能は重要な役割を持つ. 開発者に与えられるアクセス権は開発プロジェクトの組織構造, 権限の構造, 運営ポリシーなどによって決定され, それぞれの開発プロジェクトごとに必要なアクセス権管理機能は異なる.

例として, OSSD (Open Source Software Development) プロセスにおけるアクセス権管理の重要性と多様性を説明する. OSSDでは、比較的ゆるやかな制約のもとで開発が進行する. 一般にOSSDプロジェクトの権限の構造は、プロジェクト全体の意思決定 / 管理を行うコアチーム, モジュールの意思決定 / 管理を行うメンテナ, ソースコードの変更権限のみを持つコミッタ, 開発に貢献するユーザであるコントリビュータという構成を基本としながら, それぞれのプロジェクトによって権限の構造は大きく異なる. コアチームの権限が一番強く, コントリビュータの権限が一番弱い.

コミッタは自分の行った変更をリポジトリに反映させる際には, その変更に対応するモジュールのメンテナによるレビューと承認が必要である. そのプロセスはシステムによって駆動されるわけではなく, 暗黙の了解によってコミッタ自身が忘れずに駆動しなければならない. また, この権限の構造から, コアチームはリポジトリに対して行うさまざまな粒度のアクセス権限の授与 / 剥奪や, 保証人 / 継続した貢献が必要などといった条件のチェッ

クを手作業で行っている. コアチームはメンテナに特定のモジュールに対してのみ全権を委任しているわけで, 担当している以外のモジュールに対しての権限を持つべきではない. しかし, 現状では版管理システムが細粒度のアクセス制御をサポートしていないために, 他のモジュールに対しても変更権限を与えてしまうことになる[1]. 版管理システムにアクセス権管理機能があれば, 細粒度のアクセス制御が可能となり, コアチームはさまざまな条件の下で柔軟かつ安全に権限の委譲を行うことができる.

6.2 拡張機能フレームワーク層

図 6.1 に複数のアクセス制御モデルをサポートするアクセス権管理フレームワークの設計 (クラス図) を示す. この層に含まれるのは以下のクラスである: *ExtendedFunction*, *AccessController*, *User*, FileInRepository.

本フレームワークの中核となるクラスは, 抽象クラス *AccessController* である. このクラスは, あるユーザがリポジトリ内のあるファイルにアクセスする際に呼び出され, アクセス許可を与えるかどうかを判断するメソッド *check* を持っている. このため, *AccessController* は, 現在アクセスしようとしているリポジトリ内のファイルをモデル化した FileInRepository と現在アクセスしようとしているユーザをモデル化した *User* クラスと関連が張られている. 実際, どのようにそのアクセスを許可するかどうかの判断を行う処理は, サブクラスで実装する.

本フレームワークを用いての実装するアクセス権管理機能を拡張機能として基本システムフレームワークに登録するために, 抽象クラス *AccessController* は抽象クラス *ExtendedFunction* を継承して, 図中のノートに示したコード断片のようにメソッド execute をオーバーライドしている. これで, 基本システムフレームワークの ExtendedFunctionManager から本拡張機能フレームワークを通して具体的な拡張機能の実装を呼び出せるようになる. このフレームワークにより, ファイルを単位とした細粒度のアクセス制御が可能となっている.

6.3 拡張機能層

前節で述べたアクセス制御フレームワークを拡張して ACL と RBAC という 2 種類のアクセス制御モデルを実装した.

拡張機能実装者は, 抽象クラス *AccessController* を継承して拡張機能の実装を行う. このとき, *AccessController* のメソッド *check* を必ずオーバーライドしなければならない. このメソッドの役割は, あるファイルに対するユーザからのアクセスを許可するのかどうかの処理を実装することにある. つまり, *check* メソッドは hook method になっている.

以下では, ACL と RBAC のそれぞれについて設計と実装を述べる.

[1] ここの議論で使っている “モジュール” は, CVS のモジュールではないことに注意.

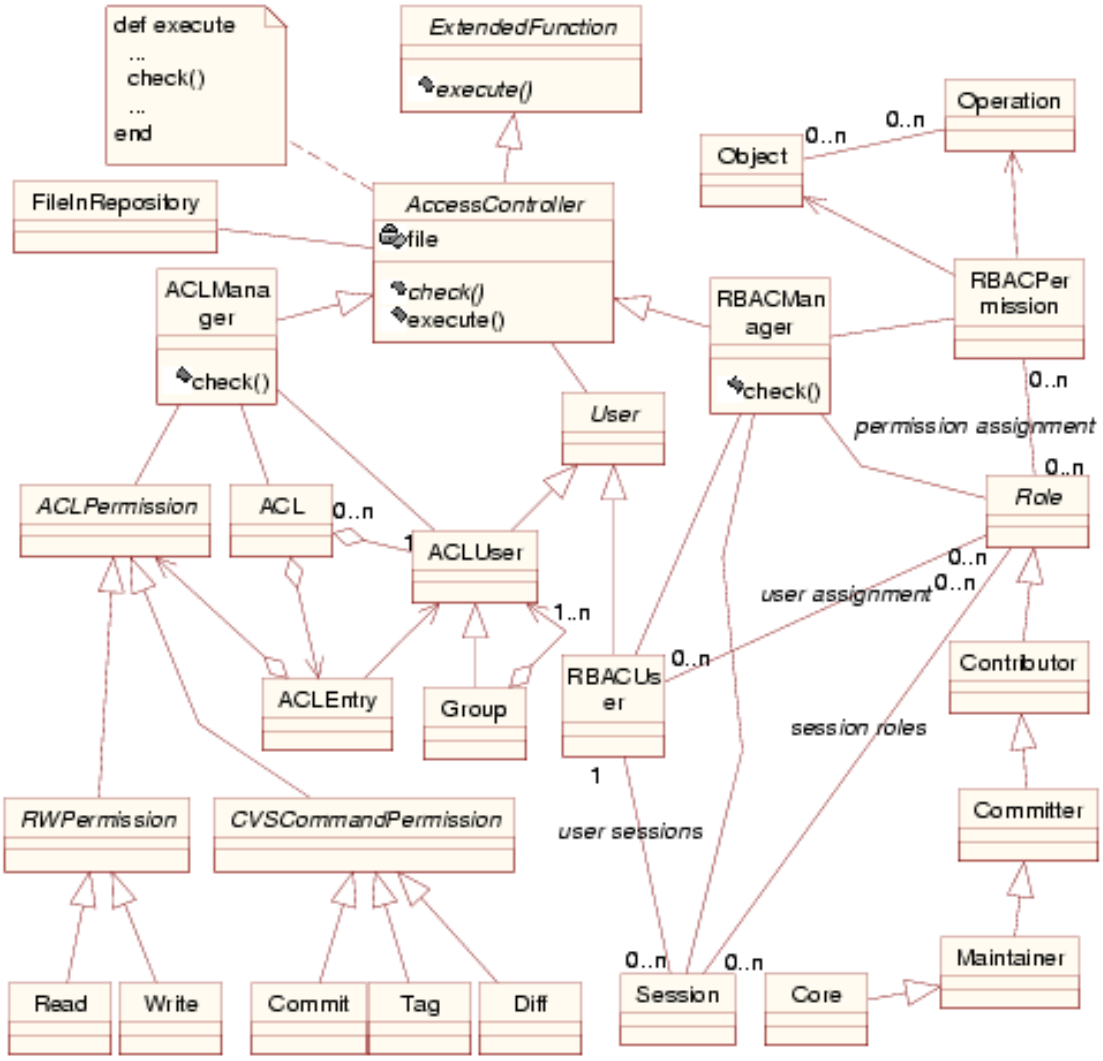

図 6.1: アクセス権管理機能の拡張機能フレームワーク設計 (クラス図)

6.3.1 Access Control List (ACL)

本論文で用いる ACL を以下のように定義する:

- ファイル 1 個につき 1 個の ACL を持つ.
- ACL はユーザまたはグループと, 権限との対応を保持するリストである.
- ACL の変更は, ファイルの所有者または所有グループのみ可能である.

たとえば, ユーザ / グループ定義が

```
user1:PASSWORD1
user2:PASSWORD2
group1:user1,user2
```

のようにされており, あるファイル file1 の ACL が

```
group1:rw
-user1:w
```

と定義されていた場合, file1 に対して user1 は読み込みのみ, user2 は読み書きが可能である.

図 6.1 において, ACL は複数の ACL エントリは持っており, 一つ一つの ACL エントリはユーザまたはグループと権限との組を保持している. この拡張機能フレームワークのホットスポット部分は, 抽象クラス *ACLPermission* である. このクラスを継承することにより, フレームワーク開発者は権限の種類を自由にカスタマイズできる.

たとえば, リポジトリ内のファイルに対して読み込みまたは書き込みの 2 種類の権限でアクセス制御する場合には, 図中の *RWPermission* のサブクラス Read, Write を作成すればよい. また, CVS のコマンドを単位としてファイルにアクセス制限をかけたい場合には, 図中の *CVSCommandPermission* のサブクラス Commit, Tag, Diff などのような拡張を行えばよい.

6.3.2 Role Based Access Control (RBAC)

RBAC [30, 14] では, 役割 (Role) を基本としてアクセス制御が行われる. 組織の中でのユーザの役割がアクセス制御に使われる.

病院内のユーザである, 医者と看護婦の役割を例として考えてみる. 病院内の資源へのアクセス権は役割ごとに決まっている. 医者は診療を行えるし, 薬を処方できる. しかし, 看護婦は診療や薬の処方はできない. このように, 役割と操作には関係がある.

図 6.1 において, ユーザはある役割でファイルにアクセスしようと試みる. その役割にはファイルに対して実行可能な操作が定義してあり, その役割に許されている操作だけを

実行できる. この拡張機能フレームワークのホットスポット部分は, 抽象クラス *Roles* である. これをサブクラス化することにより, 役割の階層で権限の強さを表現でき, フレームワーク開発者は権限の構造を自由に定義できる.

図中の Core, Maintainer, Committer, Contributer のクラス階層は, 6.1 節で説明した OSSD プロジェクトにおける権限の構造をモデル化した例である. OSSD における権限は, Core が一番強く Contributer が一番弱い. このモデルを使うと, 開発者がリポジトリにアクセスする際には, その操作に適切な役割を用いる. たとえば, あるメンテナがあるモジュールを担当していた場合, Maintainer クラスを継承してそのメンテナ専用の役割を設定する. そうすると, そのメンテナはその役割を使って担当しているモジュールのみしかアクセスできなくなる. これにより, 自分の担当外のモジュールを間違って書き込んでしまうといったミスを防止することができる.

このように, RBAC を用いると組織の権限の構造に応じた役割階層を自在に設定できる.

6.4 拡張容易性についての評価

本研究では, 拡張機能としてアクセス権管理機能を考え, 図 6.1 に示したように拡張機能フレームワークと拡張機能 である ACL と RBAC の実装を行った.

拡張機能の実装者からみると, アクセス制御を行う上で必要になる情報 (現在リポジトリにアクセスしようとしているユーザとファイル) は, 拡張機能フレームワークですでに実装されており, 拡張機能の実装者がこのことを設計したり実装したりしなくてもよい. また, 実装している拡張機能を基本システムフレームワークからどのように呼び出すかといったことも考える必要はない. なぜならば, *AccessController* クラスがすでにそれを実装しているからである. つまり, 拡張機能の実装者は, *AccessController* を差分プログラミングして, 自分のプロジェクトで使用したいアクセス制御モデル (ACL や RBAC) の設計 / 実装のみを行えばよい.

このように, 柔軟で汎用的な拡張が可能でありながら, 拡張機能の実装コストは拡張機能フレームワークを差分プログラミングすることにより, 最小限に押さえることができたといえる.

6.5 本プロトタイプシステムの対象とする拡張機能

開発プロジェクトの要求するプロセス支援機能を版管理システムに拡張できるソフトウェア構成についてこれまで議論してきた. 第 2 章で述べたように本プロトタイプシステムはオブジェクト指向フレームワークの拡張メカニズムを利用した汎用的な拡張が可能である. さらに, オブジェクト指向フレームワークのホットスポット部分にサブクラス化やオブジェクトコンポジションを用いて差分プログラミングを行うことにより低いプログラミングコストで容易に拡張できると主張してきた.

本アプローチによって設計されたプロトタイプシステムにおいて, 基本システムフレームワークを利用して開発プロジェクトの要求するさまざまなプロセス支援機能を追加することは可能である. しかし, その機能拡張の容易性を保証するにはそのドメインに対応する拡張機能フレームワークが前もって実現されている必要がある.

プロセス支援機能の中で, 拡張機能フレームワークを構成するのが困難であると推測されるものに変更管理機能がある. 変更管理機能はプロセス支援機能の中で中心的な機能であり, 状態遷移で変更要求を管理するモデルや, ワークフローとプロセスモデリング言語を用いるモデルなど複数提案されているが, その実装法が明らかでなかったりさまざまであったりする. このような比較的広いドメインを対象としたオブジェクト指向フレームワークの設計には非常に多くの開発時間とプログラミングコストがかかってしまうのが普通である.

したがって, プロセス支援機能の中でも本手法が有効な拡張機能は以下のような特徴を持つ機能であると考えられる: (1) 対象ドメインが比較的狭く, (2) 実装方法が明らかで, (3) 一般の開発プロジェクトに適用可能で, (4) 比較的小規模な機能. このような特徴を持つプロセス支援機能のみを本手法の対象としても, 以下のような開発プロジェクトに必要なプロセス支援機能を十分に拡張可能である.

ソフトウェア共同開発を行う上で, ありとあらゆる面で開発者間で合意をとらねばならない. たとえば, タグ名の命名規則とタグを打つタイミング, リリースエンジニアリングの方法 (ブランチの切り方やフィーチャーフリーズのやり方), コミット時のログの書き方, テストの方法 (テスティングフレームワークやレグレッションテストなど), コンパイルの方法, コーディング規則, ドキュメントの書き方など, これら以外にも多数ある. さらに, これらは開発プロジェクトの持つ “ポリシー” によって, プロジェクトごとに異なるものである.

現状では, ほとんどの場合プロジェクト管理者や開発者自身が注意を払いながら, “暗黙的” にこれらの規則を守りながら開発を進めている. より進んだ開発プロジェクトでは, これらの規則を文書化して Hacking Guide や Developer Documentation としてまとめている. しかしながら規則は文書化されて明らかになったが, プロジェクト管理者や開発者自身が規則を犯さないように注意を払いながら開発を進めていることにはかわりがない. そのため, 開発者が規則をうっかり破ってしまうこともよく起こる. そうした場合, プロジェクト管理者が手作業でそれを修正しなければならない. また, 開発者は活動を行うたびに規則を調べたり, 思い出したりしなければならず, ソフトウェア開発に集中できない.

本手法を用いると, これらの問題を解決することができる. 開発プロジェクトは, “ポリシー” に従ってこれらの規則を拡張機能として版管理システムに埋めこむことができる. つまり, これらの規則を開発者に強制させたり, 開発者が行う活動を支援することができる. これにより, プロジェクト管理者が手作業で管理を行う機会が減る. また, 開発者はソフトウェア開発に集中できるようになる.

本プロトタイプシステムの拡張機能として本章で取り上げたアクセス制御機能は, 上記 (1) から (4) の特徴を満たしている例となっている.

第7章　おわりに

7.1　まとめ

本論文では, 従来の版管理システムが持っていた以下の 3 つの問題点を解決する, ソフトウェア開発プロジェクトの要求するプロセス支援機能を容易に追加可能な版管理システムの構成法を示した: プロセス支援機能の不足, 汎用的な拡張メカニズムが存在しない, 拡張機能の実装コストが高い. これらの問題を解決するために, オブジェクト指向フレームワークの拡張メカニズムを利用したレイヤアーキテクチャを用いる手法を提案した.

オブジェクト指向フレームワークには未実装のまま残されているホットスポット部分があり, そこを拡張することによって機能拡張を実現できる. 拡張可能なホットスポット部分のクラス構成には, いくつかの基本構造がある. その設計にデザインパターン / メタパターンを適用することにより, より高い拡張性と汎用性を実現することができる. また, オブジェクト指向フレームワークを実装するオブジェクト指向プログラミング言語の提供するサブクラス化やオブジェクトコンポジション技術を利用することにより, 差分プログラミングが可能となり拡張機能の実装コストが低く押さえることができる.

6 層レイヤアーキテクチャは, アダプタ層の上層と下層で明確に分離する構成により, 版管理システムの実装を再利用しつつオブジェクト指向フレームワークを利用してシステムの拡張性を実現できる. また, 本手法を用いる拡張可能な版管理システムは, 以下の 4 つの機能がホットスポット部分として設計されている必要があることを示した: (1) 制御の流れを管理する機能, (2) イベント定義機能, (3) タスクマネージャ機能, (4) イベントハンドラ機能.

本アーキテクチャに基づいたプロトタイプシステムの設計と実装を行い, 本手法の実現可能性を示した. プロトタイプシステムの実装アーキテクチャとしてプロキシを採用することにより, 版管理システム層である CVS に変更を加えないで拡張可能な版管理システムを実現できる. フレームワーク層の設計に, デザインパターン / メタパターンを適用することにより, 上にあげた 4 つのホットスポット部分を含んだ拡張性の高いフレームワークのホットスポット部分の設計を示した.

プロトタイプシステムの拡張機能として, 2 種類のアクセス管理機能の設計を示した. プロトタイプシステムのオブジェクト指向フレームワークを差分プログラミングすることにより, プログラミングコストを大幅に削減できた.

よって, 本手法を用いた拡張可能な版管理システムを使うと, 開発プロジェクトはその開発プロセスに応じたプロセス支援機能を必要に応じて容易に追加することが可能となる.

これにより, 本手法が有用であることを示すことができたと考える.

7.2 今後の課題 / 展望

アクセス権管理機能以外の拡張機能を設計 / 実装し, その運用実験を行いたい. 拡張機能が増えてくると, 当然オブジェクト指向フレームワークで実装されている基本システムフレームワーク層と拡張機能フレームワーク層の設計を見直す必要が出てくるだろう. この設計の見直し作業を繰り返すことによって, より洗練されたフレームワーク層が実現できる.

謝辞

本研究を行なうにあたり, 終始御指導を賜わりました落水 浩一郎教授に深く感謝致します.

有益な御助言をいただきました情報科学センター 藤枝 和宏 助手に深く感謝致します.

本論文審査にあたり, 本学情報科学センター 篠田 陽一 教授, 権藤 克彦 助教授に深く感謝致します.

落水研究室の助手でありました 故 村越 広享 博士に深く感謝するとともに、故人のご冥福を心よりお祈り致します.

落水研究室の諸氏に厚く御礼申し上げます. 特に藤田 充典 氏には, 研究面だけではなく生活面でもお世話になりました. 感謝致します.

最後に, 院での生活を支援してくれた家族に感謝します.

参考文献

[1] Juancarlo Añez. Jrcs. http://www.suigeneris.org/jrcs/.

[2] Arthur. cvsd - a cvs pserver daemon. http://tiefighter.et.tudelft.nl/%7Earthur/cvsd/.

[3] D. Bäumer, G. Gryczan, R. Knoll, C. Linienthal, D. Riehle, and H. Züllighoven. Framework development for large systems. *Communications of the ACM*, Vol. 40, No. 10, pp. 53–59, October 1997.

[4] Brian Berliner. CVS II: Parallelizing software development. In USENIX, editor, *Proceedings of the Winter 1990 USENIX Conference, January 22–26, 1990, Washington, DC, USA*, pp. 341–352, Berkeley, CA, USA, 1990. USENIX.

[5] Bitkeeper - the scalable distributed software configuration management system. http://www.bitkeeper.com/.

[6] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[7] Ivica Crnkovic. Why do some mature organizations not use mature CM tools? In *System Configuration Management*, pp. 50–65, 1999.

[8] Cvsgui. http://cvsgui.sourceforge.net/.

[9] S. Dart. Spectrum of functionality in configuration management systems. Technical Report CMU/SEI-90-TR-11 ADA235753, Software Engineering Institute (Carnegie Mellon University), 1990.

[10] Cees de Groot. Trug - a software configuration management system. http://trug.sourceforge.net/scm/.

[11] Paul Eggert. Diffutils - finds differences between and among files. http://www.gnu.org/software/diffutils/diffutils.html.

[12] Jacky Estublier, David Leblang, Geoff Clemm, Reidar Conradi, Walter Tichy, Andre van der Hoek, and Darcy Wiborg-Weber. Impact of the research community on the field of software configuration management.

[13] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *the guest editorial for the Communications of the ACM, Special Issue on Object-Oriented Application Frameworks*, Vol. 40, No. 10, October 1997.

[14] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramoli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, Vol. 4, No. 3, pp. 224–274, 2001.

[15] The Apache Software Foundation. The apache software foundation. http://www.apache.org/.

[16] Christopher R. Gabriel. patch. http://www.fsf.org/software/patch/patch.html.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, Reading, 1996.

[18] C. W. Irving and D. Eichmann. Patterns and design adaptability. In *PLoP '96 Submission*, 1996.

[19] Kaz Kylheku. Home of meta-cvs. http://users.footprints.net/%7Ekaz/mcvs.html.

[20] Ben Laurie. Apache hook functions. http://httpd.apache.org/docs-2.0/developer/hooks.html.

[21] The linux home page at linux online. http://www.linux.org/.

[22] Josh MacDonald, Paul N. Hilfinger, and Luigi Semenzato. PRCS: The project revision control system. *Lecture Notes in Computer Science*, Vol. 1439, pp. 33–??, 1998.

[23] Alexey Mahotkin. cvs-nserver. http://cvs-nserver.sourceforge.net/.

[24] Yukihiro maz Matsumoto. The object-oriented scripting language ruby. http://www.ruby-lang.org/.

[25] Microsoft visual sourcesafe 6.0. http://www.microsoft.com/catalog/display.asp?subid=22&site=606&x=36&y=19.

[26] The open source definition. http://www.opensource.org/docs/definition.php.

[27] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, Wokingham, 1995.

[28] Unified change management. http://www.rational.com/leadership/initiatives/ucm.jsp.

[29] Rational clearcase. http://www.rational.com/products/clearcase/index.jsp.

[30] Role based access control. http://csrc.nist.gov/rbac/.

[31] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, Vol. 1, No. 4, pp. 364–370, December 1975.

[32] D. Schmidt. Wrapper facade: A structural pattern for encapsulating functions within classes, 1999.

[33] Walter F. Tichy. RCS - a system for version control. *Software - Practice and Experience*, Vol. 15, No. 7, pp. 637–654, 1985.

[34] Brian A. White. *Software Configuration Management Strategies and Rational ClearCase: A practical Introduction*. Addison-Wesley, 2000.

[35] Allan Wirfs-Brock. Ecoop/oopsla'90 panel on designing reusable frameworks. *Ecoop/oopsla'90*, October 1990.

[36] 嶋田大輔. オープンソースソフトウェア開発に適した cvs リポジトリの階層型分散構成法の研究. Master's thesis, 北陸先端科学技術大学院大学, March 2002.

本研究に関する発表論文

- 早坂 良, 藤枝 和宏, 落水 浩一郎. オープンソース開発プロジェクトのための *CVS* 機能拡張法. 平成 13 年度 電気関係学会北陸支部連合大会, pp.315, October 2001.
- 早坂 良, 藤枝 和宏, 落水 浩一郎. プロセス支援機能を組み込み可能な *CVS* プロキシ構成法. 電子情報通信学会ソフトウェアサイエンス研究会, 信学技報 SS2002-51, pp.19-24, March 2003.