修 士 論 文

OS入門用の教材を事例とした, アスペクト指向によるソースコードとドキュメントの関連づけ

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

大場 勝

2004年3月

修 士 論 文

OS 入門用の教材を事例とした, アスペクト指向によるソースコードとドキュメントの関連づけ

指導教官　片山卓也 教授

審査委員主査　片山卓也 教授
審査委員　二木厚吉 教授
審査委員　落水浩一郎 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

210012 大場 勝

提出年月: 2004 年 2 月

概 要

我々はソフトウェアに関係する多種のドキュメントをアスペクト指向を用いて整理する手法を提案する.

一般にプログラム理解において, 対応するドキュメントを照合することが必要である. しかし多くのドキュメントの中から対応するものを見つけることは煩雑である. 本研究のアイディアは, (i) ソースコード中にドキュメントへの参照を**関連**として形式的に記述し, (ii) その**関連**を横断する関心事についてアスペクト指向を用いて整理することである. これを基にドキュメント整理ツール ADIOS を実験的に実装し, OS 教材を例題として実験した. その結果ドキュメント間の関連をアスペクトに基づき検索できた.

目 次

図目次

表 目 次

第1章　はじめに

1.1　背景

我々はソフトウェアに関係する多種のドキュメントについて, アスペクト指向を用いて整理する手法を提案する. 一般にソフトウェアには多種多様なドキュメント (例えば, 仕様書やマニュアル, 開発メモ) が混在する. 開発者はここから, ドキュメントを必要に応じて検索し利用している [10]. これは, ソースコードを理解する上でドキュメントが必要不可欠だからである.

しかし, 多種のドキュメントが混在する中から必要なドキュメントを検索することは煩雑である. 従って検索し易くするためにはドキュメントの整理が必要である. 整理とは, 読み手の関心事ごとに, 必要なドキュメント断片の集合を 1 つにまとめる作業をいう. 多種のドキュメントを整理するには以下に挙げる問題点がある.

- ソースコードに読むべきドキュメントの記録がない.
- ドキュメントの分類は, 読み手の知りたい事 (関心事) によって異なるため整理が容易ではない.
- 既存のドキュメンテーションツールはこれらの点に関しては十分でない.

1.2　目的

本研究の目的はこれらの問題点をふまえ, ソフトウェアに関係するドキュメントを効率よく検索するためのドキュメント整理手法の提案と, その有効性を実験により評価することが目的である. 本研究の提案するドキュメント整理法は, 関連するソースコードとドキュメントの**追跡性**を向上させ, 効率のよいプログラム理解を実現する. **追跡性**とは, 読み手の関心事に応じて必要なドキュメントを取得できることをいう.

1.3　アプローチ

本研究は, ソースコード中にプログラム理解に必要なドキュメントへの参照を**関連**として形式的に記述し, アスペクト指向を用いてその**関連**を横断する関心事に基づいて整理する. **関連**とは, ソースコードとそのコードの理解に必要なドキュメントへの参照をいう.

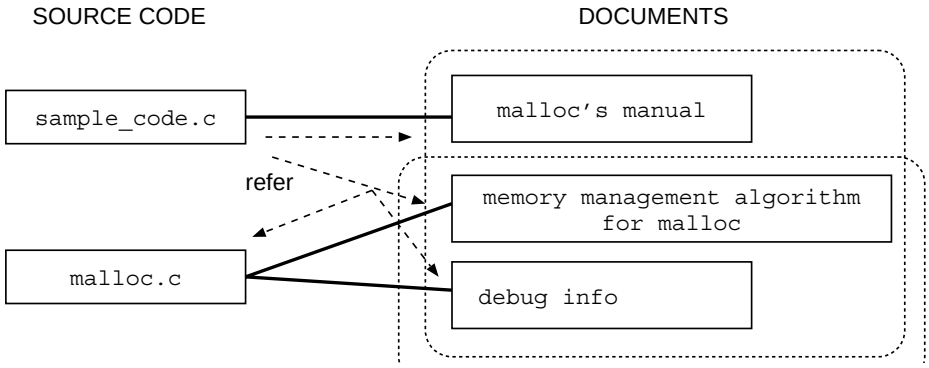

図 1.1: アスペクトの例

アスペクト指向の問題意識は, 1 つの視点でまとめた構造にはそれを横断する関心事が存在するということである. アスペクト指向は, この横断する関心事を 1 つのモジュールにする技術をいう. 本手法を適用することによって, 追跡性のあるドキュメントやソースコードを容易に取得することが可能になり, 効率のよいプログラム理解が可能になると考えた.

本研究では, **関連**を記録することによって, ソースコードからドキュメントへの追跡性を向上させる. さらに, 開発者が関連を横断する関心事を 1 つのアスペクトにすることで, **関連**の追跡性を向上させる. アスペクトは, プログラムの開発者自身が**関連**を関心事別に整理する. これによって, 読み手がソースコード中に埋め込んだ関連以外にも追跡性の高い関連を取得することが可能になる.

図 1.1 は,「malloc 関数のサンプルコード」と「malloc 関数の実装」が持つ**関連**を実線で表し, アスペクトによって分類されたドキュメントを点線四角で囲んだ図である.「malloc 関数のサンプルコード」から直接得られる関連は「malloc 関数のマニュアル」のみである. 図 1.1 のように,「malloc 関数のサンプルコード」の関連と「malloc 関数の実装」の関連を 1 つのアスペクトにまとめておけば,「malloc のサンプルコード」からより詳細なドキュメント「メモリ管理」や malloc の実装を取得することが可能になる. さらに「ページングの実装」などの関連をこのアスペクトにまとめておけば, メモリ管理に関するより詳細な情報を追跡することが可能になる.

例えば, 既存のドキュメンテーションツール Javadoc[17] では, このような追跡性の高い**関連**の取得を行うことは難しい. Javadoc はソースコード中に埋め込んだコメントから API リファレンスを生成するツールで, コメント中に URL のリンクを埋め込むこともできる. Javadoc をドキュメント整理ツールとしてみた場合, ソースコード中に直接リンクを埋め込んだドキュメントは取得できるが, アスペクトのように追跡性のある他の関連の取得はできない. また, ソースコードとの対応をとることもできない.

1.4 結果

本研究では, アスペクト指向によるドキュメントの整理法を適用したツール ADIOS[14] を実験的に実装した. ADIOS を使用して独自開発中の OS 教材 udos[18] を事例とし, ページングに関してドキュメントの整理を行った. OS 教材を事例としたのは, 多くのドキュメントをソースコードと照合しながら理解する必要があり, ドキュメント整理の効果を明確に評価できると考えたからである.

実験の結果, ソースコードから直接埋め込んだ関連以外に, 追跡性のある他の関連を取得することができた. この実験に限って, 我々の方法が複雑な関係をもつドキュメントをアスペクト整理法で, 上手く整理できたことを意味している. また, 本研究の問題点として, 読み手が関連を選ぶための説明文が難しいことがわかった. 我々がドキュメント整理を適用した事例が小規模だったため十分な評価が得られなかった. このため, 本手法のより大規模な適用を行い問題点の洗い出しをしなければならない.

1.5 本論文の構成

第 1 章 はじめに　ソフトウェア開発におけるドキュメント整理の必要性とドキュメント整理における問題点を述べ, 本研究のアプローチと結論を概説した.

第 2 章 ドキュメント整理の問題　ソフトウェア開発における種々のドキュメントについて簡単に述べ, それらを整理する場合の問題について触れる.

第 3 章 ドキュメント整理へのアスペクト指向の適用　アスペクト指向によるドキュメント整理法を提案する. アスペクト指向プログラミングと本研究の適用を対比しながら述べる.

第 4 章 ドキュメント整理ツール　前章で提案したドキュメント整理法を基に, ドキュメント整理ツール ADIOS を作成し, ADIOS の仕組みについて述べる.

第 5 ADIOS の OS 教材 udos への適用　我々が独自開発中の OS 教材 udos の概要を説明し, ドキュメント整理ツール ADIOS を OS 教材 udos に適用した例について述べる.

第 6 章 議論　我々が提案したドキュメントの整理法と udos の事例に基づいて ADIOS の実装に関する議論を行う.

第 7 章 関連研究　既存のドキュメンテーションツールの紹介と本研究との関連性を述べる.

第 8 おわりに　本研究のまとめと, 今後の展開について述べる. udos のページングにドキュメント整理法を部分的に適用した結果, 本手法でドキュメントの検索が有効であること. 今後さらに規模の大きい実験が必要であることを述べる.

第2章　ドキュメンテーションの問題点

2.1　ソフトウェア開発におけるドキュメント

ソフトウェア開発では, プログラムや仕様書などの多種にわたる文書が作成される [20]. 各開発プロセスのドキュメントの例を表 2.1 に示す. プログラム理解のためには複数のドキュメントを参照しなければならない. しかし, 各ドキュメントは作成時に適した構成であってプログラム理解の場合においてこの構成が有効に働くわけではない.

プログラム理解に必要なドキュメントは混在するドキュメントの中から検索しなければならない. 従って必要なドキュメントを検索する作業は煩雑である. そのため一度参照したドキュメントを記録し整理することはプログラム理解の効率の点で重要である. 整理とは, 1 度参照したドキュメントをプログラム理解に必要なドキュメントごとに 1 つの集合にまとめることをいう. 例えば, OS 教材 udos の開発に必要なドキュメントは PC/AT(PIC,DMA,FDC,ATA 等), インテルアーキテクチャ, 参考ソースコード (linux など), BIOS のマニュアル, その他のマニュアルが混在しており, 全部で 4000 ページ以上のドキュメントもある. 必要になるドキュメントのほとんどはプログラムを実装したときに 1 度参照されていることが多い. 例えば実装するときには関連する仕様書, マニュアル, 実装時のメモなどがプログラマによって参照されている.

2.2　ドキュメントの閲覧時の問題

前節ではプログラム理解のためのドキュメントへの参照を記録し, 整理しておくことが重要であることを述べた.

検索をするためにはドキュメントを整理する必要がある. しかし, 多種のドキュメントを整理するには以下に挙げる問題点がある.

(1) ソースコードに読むべきドキュメントの記録がない.

(2) 読むべきドキュメントの記録が残っていても, 効率よく検索するための分類がなく, それ自体も難しい.

(3) 既存のドキュメンテーションツール [17][19][12] はドキュメントの整理に関しては十分でない.

プロセス	ドキュメント
要求分析	要求仕様書
システム設計	設計仕様書
実装プロセス	アルゴリズム, 構成論などの参考書
	使用するツールやライブラリ等のマニュアル
テストプロセス	テストプログラム仕様書
	コードのテスト結果 (単体テスト, 統合テストなど)
	テストコード
保守	バグの報告書, バグ原因の結果報告書
	テストツールなどのマニュアル

表 2.1: ソフトウェア開発におけるドキュメント例

プログラマにとってドキュメントへの関連を保存することは煩雑である. 関連とはソースコードと, ソースコードを理解するために必要なドキュメントとのつながりのことをいう. しかし, ソースコード理解のためには, そのソースコードを実装した人が必要としたドキュメントを参照することは重要である. また, 開発者自身にとっても過去に参照したドキュメントすべてを記憶しているわけではなく, デバッグなどでソースコードの修正が必要になったとき, 過去に検索したドキュメントを再検索しなくてはならない.

ソースコード中に関連するドキュメントを記録しておくことによって, 過去に検索したドキュメントの再検索は必要なくなる. さらに, それらの記録同士の関連も 1 つにまとめられ, 必要なドキュメントを集合で取得することが可能になれば, 効率のよいドキュメントの取得が可能になる. 例えばドキュメント A を読むためにドキュメント B の理解が必要, ドキュメント B は別のソースコードとそのソースコードから関連しているドキュメント C を理解しなければならない場合 (図 2.1 参照) である. ドキュメント A を取得した場合にそれに付随するドキュメント B と C と他のソースコードが集合でとれれば, より効率のよいドキュメントの取得ができる.

しかしドキュメントの集合の作成 (すなわち分類) は, 検索時の読み手の知りたい事 (関心事) によって異なるため, 容易ではない. 1 つの視点で分類したドキュメントは, その視点からみれば良くまとまって使いやすい. しかしそれらのドキュメントを別の視点でみた場合, その分類を横断する参照をしなければならない.

例えば, 要求分析プロセスにおいては顧客からの要求を取得するために現状の把握のため「市場調査」や「アンケート」などがまとめられる. そこから得た情報から「機能, 構成要素, 性能, 信頼性, 納期, 予算」が検討され 1 つの要求仕様書としてまとめる. 設計プロセスでは, 要求仕様書を基づいて, 機能や性能の要求を満すプログラム部品の抽出を行い, それを設計仕様書にまとめる.

これらのまとめられたドキュメントを利用する場合, 関心事に応じて複数のドキュメントを参照することが多い. また, この関心事によって参照するドキュメントは様々である.

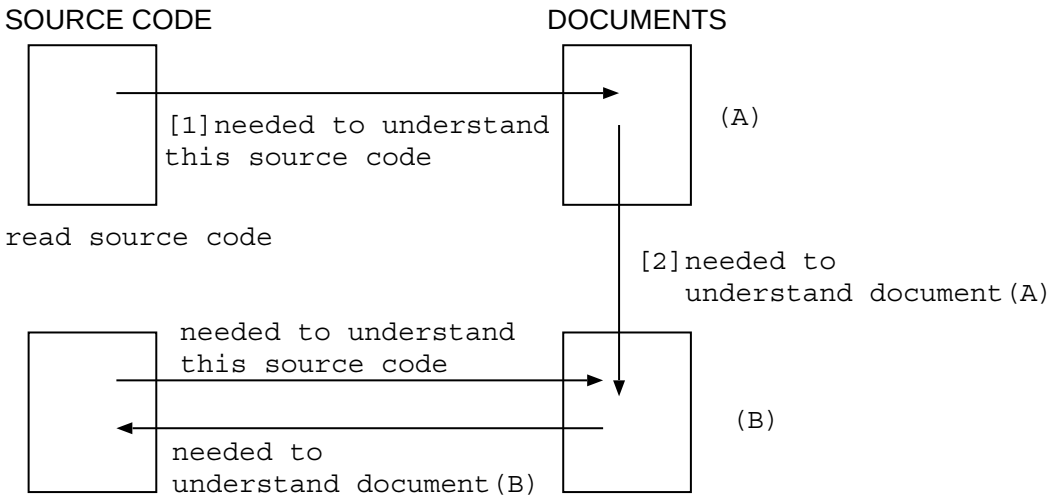

図 2.1: プログラム理解に必要なドキュメントとソースコードの関係

実装プロセスでは, 設計仕様書を基にプログラム言語で実装を行い, 実装に必要なドキュメント (ライブラリのマニュアルやメモ等) やコード断片を部分的に参照する. これが関心事によるドキュメントを横断するということである. 実装プロセスで参照したドキュメントを 1 つの分類と考える.

デバッグ時において, まず開発者が行わなければならないことは, デバッグ対象のコード理解である. しかし, (他人の) コード理解は自分で設計する以上に困難である [21]. このため, 各プロセスにおいて利用されたドキュメントを参照することが必要である. デバッグにおいて, 必要なドキュメントは多くある. 実装時では設計仕様書を参照すればよかったが, デバッグ時では顧客の要求をみたしているかどうか, それが設計仕様書で実現されているか, 実装のテスト結果が設計仕様書を満たしているかどうか, それぞれのドキュメントと照合してチェックする必要がある (図 2.2).

このように, それぞれの場面に必要なドキュメントを 1 つの分類にまとめようとしたとき, ドキュメントの分類は単純な排他関係ではなく, 関心事によってその分類が重なり合うことがある. つまり, 各ドキュメントを単純に線引きして分類することはできないため, ドキュメントの分類は難しい.

2.2.1 ドキュメントの追跡性

追跡性とは

図 2.1, 図 2.2 で示した例のように, プログラミング理解において問題解決のために必要なドキュメントやソースコードが必要であることを示した. 本論文では, 必要なドキュメントやソースコードを人が取得できることを**追跡性**と呼ぶ. プログラミング理解には追跡

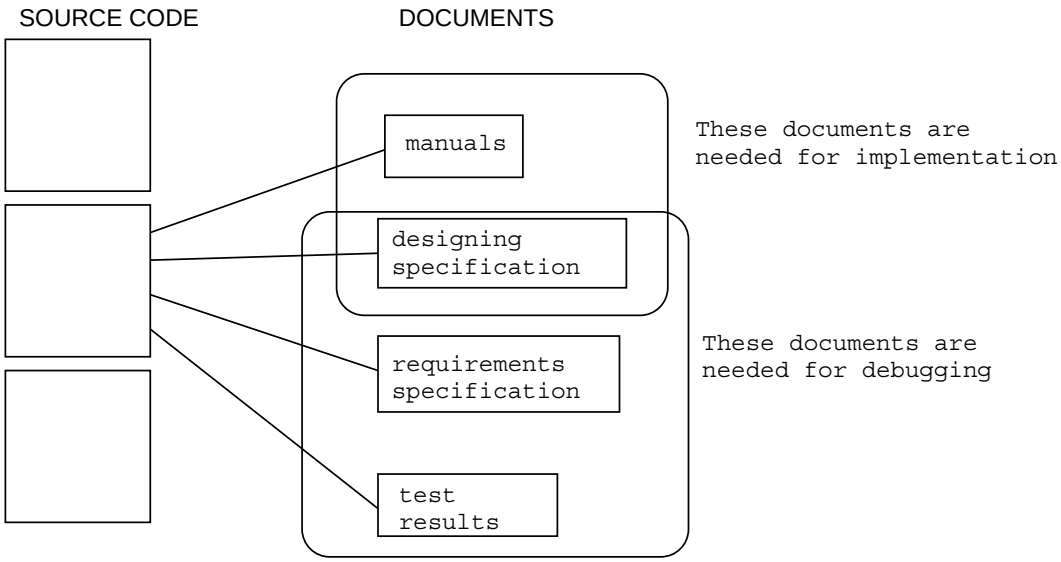

図 2.2: 実装時とデバッグ時に必要なドキュメントの違い

性のあるドキュメントとソースコードが必要である.

追跡の対象

我々は, プログラム理解で, 追跡する必要のなる対象をソースコードに付随するドキュメント (仕様書や開発メモなど) とソースコードであると考えた. さらに, 追跡の対象を以下の 2 種類に分類した.

1. プログラム理解の対象のソースコードから参照すべきドキュメント (図 2.1 [1]). 読み手がソースコードからドキュメントを直接取得できる.
2. 理解対象のソースコード以外に記述してあるドキュメントまたは, 必要な他のソースコード (図 2.1[2]).

1 は, ソースコード中にドキュメントの参照情報を埋め込むことによって, 読み手が取得可能であるという点で直接的である. また, 読み手が直接追跡性のあるドキュメントを読んだときに始めて追跡性が生じるドキュメントやソースコードは, 理解対象としているソースコードからみて間接的である.

ドキュメントの追跡方法

直接的なドキュメントへの参照は読み手によってソースコードから容易に取得可能である. しかし, 間接的な追跡性のある関連は, 読み手の関心事によって異なる. このため関心事別に関連を整理する必要がある.

読み手がドキュメントを追跡するには, (1) プログラム理解の対象となるソースコード中に追跡性のあるドキュメントの参照が埋め込まれている必要がある. さらに (2) 読み手の関心とドキュメントの内容が一致するかどうかを判断するための参照への説明がなければならない. これら情報を読み手が得ることができれば直接的な追跡性のあるドキュメントは取得可能である.

また, 間接的な追跡性は読み手がソースコードから, 関係の深いと思われるドキュメントやソースコードを自ら検索しなければならない. これは, 対象としている問題を理解していない読み手にとっては, 負荷の高い作業である. これを容易に検索するには, 関係の深いドキュメントやソースコードの参照を関心事別に分類し, その分類の説明をトピックとして読み手に提供する. これによって読み手は, 間接的な追跡作業の軽減が可能である.

2.3 ソースコードとドキュメンの関連の種類

ソースコードから必要なドキュメントや参考となるソースコードの参照を行う場合, 最も単純な関係はコード断片の理解に必要なドキュメントが 1 つの場合 (1 対 1) である. しかし, 前節であげた例のように, 1 つのコード断片に対して複数のドキュメントを読まなければならない場合が多い場合 (1 対多) がある. また, 図 2.1 のようにドキュメント以外にソースコードも参照し, これを 1 つの分類ととらえるとき, ソースコードとドキュメントの関係は多対多になる.

ソースコードとドキュメントのつながりを以下にまとめる.

- 1 対 1 : コード断片に対して, 読むべきドキュメントが 1 つでよい場合.
- 1 対多: コード断片に対して, 読まなければならないドキュメントが複数ある場合.
- 多対 1: 複数のコード断片から, 同一のドキュメントを参照する場合. このとき, ソースコード間に関係はない.
- 多対多: ソースコード理解に必要なコード断片が, あちこちに分散し, それぞれのコード断片に対して, それに付随するドキュメントがある場合.

2.4 ドキュメント整理ツール

既存のドキュメンテーションツール [17][19][12] は主にドキュメントの生成をサポートするものが多く, 「ドキュメンテーションツール=ドキュメントの生成サポート」という概念が強い. しかし, ドキュメンテーションは以下に示すように, 生成, 整理, 検索の要素の組み合わせによって実現するものである (図 2.3 参照).

- ドキュメントの生成.

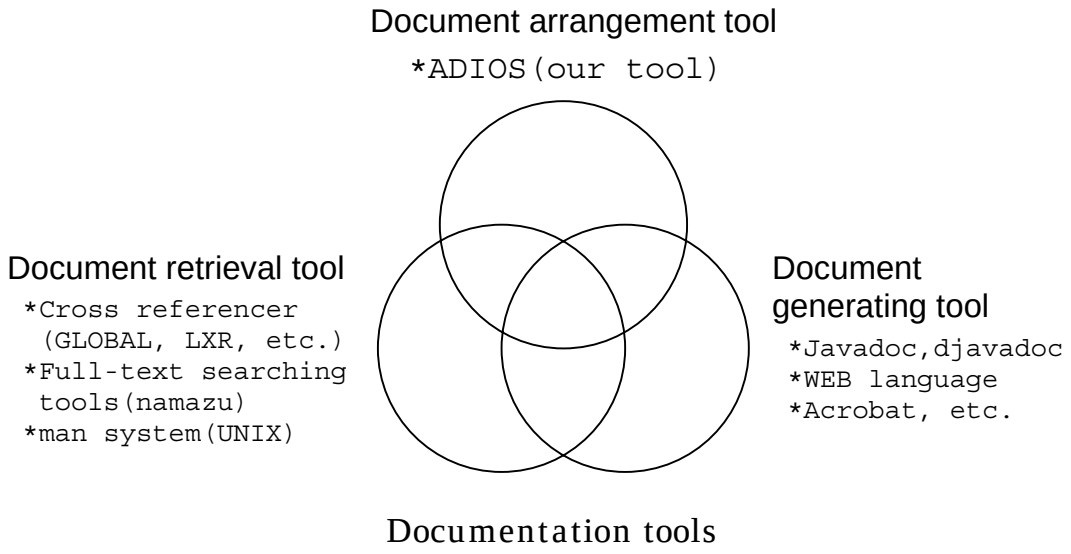

図 2.3: ドキュメンテーションツール間の関係

- ドキュメントの整理, 管理.
- ドキュメントの検索

これらの要素は階層構造をもち (図 2.4), ドキュメンテーションをサポートする. ドキュメントの検索 (retrieval) は, ドキュメントを分類し検索対象をしぼり込むための基盤が必要である. この基盤を担う階層がドキュメントの整理 (arrangement) を行うレイヤーである. ドキュメント整理の階層では, 整理する対象となるドキュメントそのものと, ソースコードやドキュメント同士の関連が必要である. これを提供する階層がドキュメント生成 (generating) の階層である.

我々が提唱する広義のドキュメンテーションの意味では, ソースコードのクロスリファレンスも, 関連し合うソースコードを相互に閲覧しコメントから有用なドキュメントや参考文献などを取得できる点でドキュメントの検索ツールとしてみることができる. 本研究では, これらのドキュメンテーションツールのうち, ドキュメントの整理に主眼をおく. 整理するとはソースコードに関連するドキュメントを分類わけし, ドキュメントの検索を行うための基盤を構築する作業である.

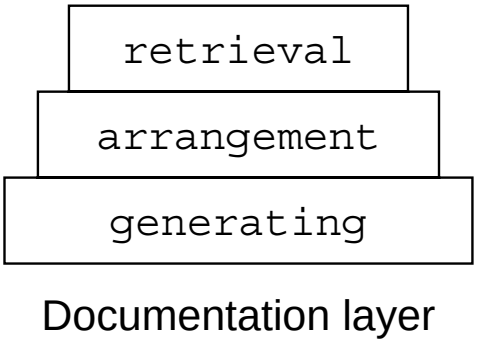

図 2.4: ドキュメンテーションのレイヤー構造

第3章　ドキュメント整理へのアスペクト指向の適用

3.1　ドキュメントの整理法

前章で示した通りソースコードとドキュメントのつながりは多対多の関係になり複雑になる. 本節では, 多対多の関連を容易に記述し, 前節で述べた問題点を解決するドキュメント整理法を提案する.

本研究でドキュメントを整理するためのアイディアは以下の通りである.

- ソースコード中にドキュメントへの参照を**関連**として形式的に記述する.
- その**関連**を横断する関心事についてアスペクト指向を用いて整理する.

関連は, ソースコードの位置とその理解に必要なドキュメントの位置を示し開発者が実装時に埋め込む. **関連**は, 形式的に記述することによって, 開発者による埋め込みでも機械処理による抽出も容易に行うことが可能である. 以下に**関連**中に記述する要素を挙げる.

- 参照するドキュメントの位置を表す URI
- その関連の説明
- キーワード (詳細は 3.4 節で説明)

上に挙げた要素を “[@関連の説明;URI; キーエワード]” のように形式的に記述する. これによって, ドキュメント整理ツールによる抽出と整理が可能になる. ソースコード中に関連を埋め込むことによって, ソースコードとドキュメントのつながりをドキュメントの URI だけで表現できるため記述コストをさげることができる. **関連**は常に 1 対 1 の関係にあり, ソースコードとドキュメントの最も小さい単位である. しかしほとんどの場合, 任意のコード断片と参照するドキュメントの関係は多対多になる. 多対多を 1 対 1 の集まりへ細かく**関連**を分けることによって, 分類を詳細に行うことができる. しかし, 細かい単位の関連を手作業によって整理することは負担が大きい. これを解決するために本研究では, 関連にキーワードを付加し, あらかじめ分類を行う.

3.2　アスペクト指向の概要

アスペクト指向プログラミングの基本的な問題意識は, 1 つの視点でまとめた構造にはそれを**横断する関心事**が存在してモジュール化ができないものがあるということである. この横断する関心事に基づいてモジュール化することをアスペクト指向プログラミングという. アスペクト指向とはこれを他の分野にも適用した概念であり, 近年, アスペクト指向に関する多くの研究が行われている [1]. モジュール化したアスペクトを扱うための機能を以下に挙げる.

- 分離したアスペクトを元の構造へ戻す作業をウィーブ (weave)
- ウィーブ (weave) する際に元に戻すためのポイントをジョインポイント (join point)

3.2.1　アスペクト指向プログラミング

アスペクト指向プログラミングは, オブジェクト指向によって 1 つにまとめられた構造 (クラスの階層構造) に対し, 横断的関心事に基づいてクラス間を横断する関心事を 1 つのアスペクトとしてまとめる. アスペクト指向プログラミングの仕組みをサポートする言語としては, AspectJ, AspectC++など数多くある.

AspectJ　AspectJ はアスペクト指向を Java 言語に適用した実装である. ジョインポイントはメゾットやコンストラクタの呼び出し前後, 例外ハンドラなどに設定することができる.

モジュールは言語要素 aspect として 1 つの場所に記述する. この記述内容をアドバイス (Advice) と呼ぶ. aspect 中にジョインポイントを設定しウィーブされるときに使用される. AspectJ の場合, このジョインポイントは集合として指定できる. これをポイントカット (Point-cut) という.

3.2.2　アスペクト指向のドキュメント整理への適用

本研究ではソースコードを基準に, 対応するドキュメントへの**関連**を扱う. 関連を横断する関心事を 1 つにまとめたものを**アスペクト**という. ドキュメンテーションにおいてウィーブするとは, ソースコードと関連するドキュメントを 1 つにみせることをいう.

ウィーブ時に必要なジョインポイントは関連を埋め込んだソースコードの位置にあたる. 開発者がコーディング中に参照したドキュメントをソースコード中に関連として残す作業を**関連づけ**という. アスペクト指向言語でいう横断的関心, アスペクト, ウィーブ, ジョインポイントは本研究では表 3.1 のように対応する.

用語	AOP	本研究
横断的関心	OOP を横断する	関連を横断する
アスペクト	プログラム	関連
ウィーブ	バイナリなど	ドキュメントとコードの合体
ジョインポイント	関数の前など	関連の位置

表 3.1: AOP と本研究の概念比較

3.3 アスペクト指向によるドキュメント整理法の提案

2.2 節ではドキュメントの閲覧時の問題点について述べた. 本節ではその問題点に対して, 3.1 節で述べた. ソースコードとドキュメントとのつながりを**関連**として形式的に記述し, アスペクト指向を適用することによって**関連**をアスペクトに整理することで問題の解決を行う.

3.3.1 関連

関連の種類

関連は, 理解対象となるソースコードとそのソースコードで必要なったドキュメントとのつながりである. 関連にはいくつかの種類がある (図 3.1 参照). 図 3.1 は, ソースコードとドキュメントと関連を示した図である. 図中ではソースコードが全部で 4 つあり, その中のソースコード A(source (A)) での 1 つの関心事によって現れる関連の種類を図示している. 従ってこれらすべてのソースコードの関連を考えればより複雑になる.

以下に関連の種類をあげる.

1. プログラム理解の対象となるソースコードから直接関係のあるドキュメントへのつながり (図 3.1(1)).
2. プログラム理解の対象となるソースコードから直接関係のあるソースコード (図 3.1(2)).
3. 直接関係のあるドキュメントを理解するための間接的なソースコード (図 3.1(3)).
4. 直接関係のあるドキュメントを理解するための間接的なドキュメント (図 3.1(4)).

1, 2 は, の種類の関連は, 読み手にとってソースコード中にある問題点解決に必要な最初の手掛りになる. 例えば, ドキュメントは関数の引数の書式などを記述したマニュアルなどはこれにあたり, ソースコード中で使用している関数の定義などのソースコードである. 3,4 はドキュメントの理解を補完するドキュメントやソースコードである. 例えば, 図 3.1

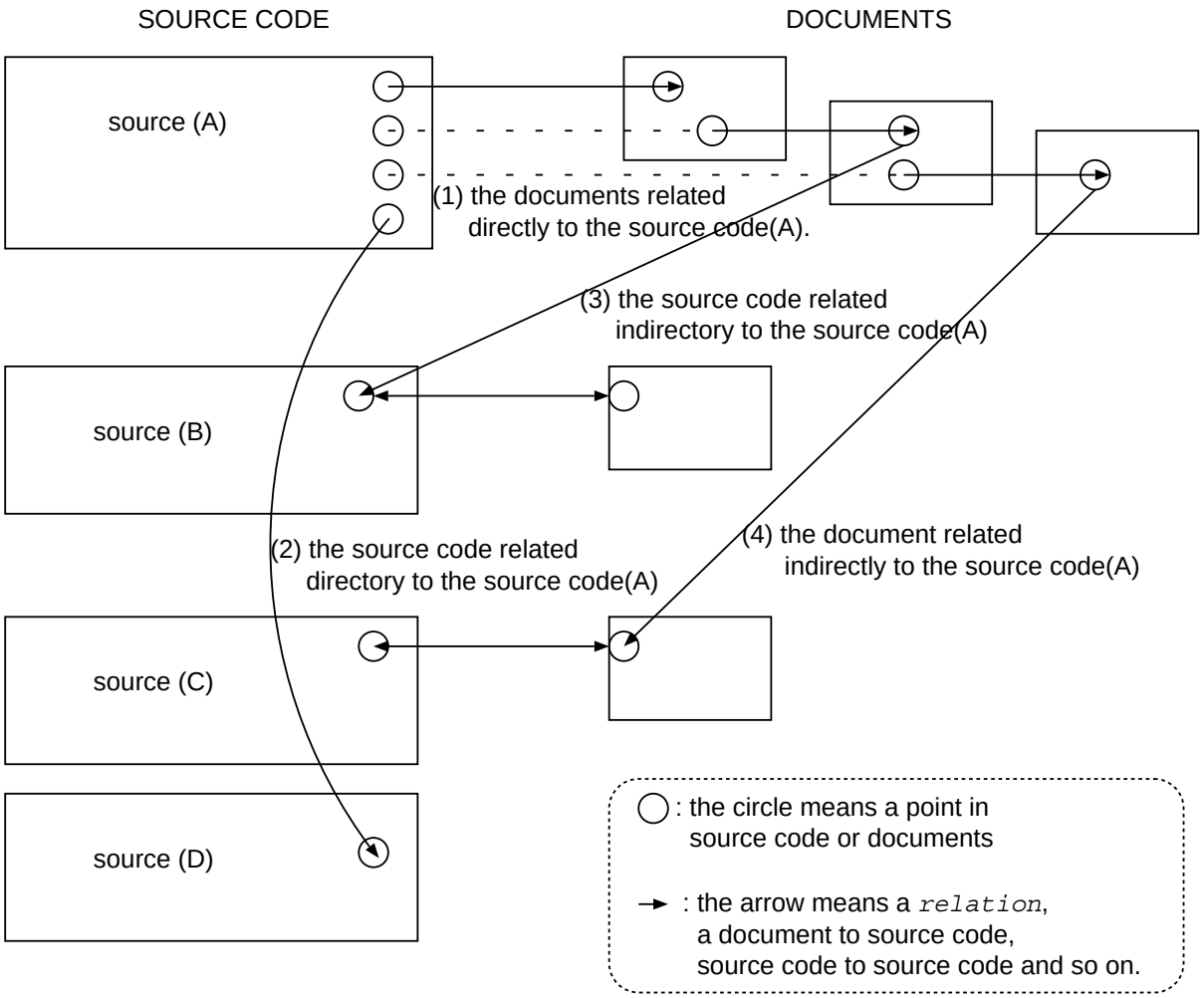

図 3.1: 関連の種類

のソースコード (1) から直接関連するドキュメントがチュートリアルであったとすれば, それに付随するサンプルコードや用語説明のドキュメントなどである.

このように, ソースコードとドキュメントの関係は多対多になる. これを開発者がすべての種類を区別して記述することは煩雑であり, 開発効率を下げてしまう. この問題を解決するために我々は, これらすべての種類の関連を少ない構成要素で記述するアプローチをとった. これによって開発効率の低下を防ぎつつ関連を残すことが可能になる.

関連の記述方法

我々は, この複雑な構造をもつ関連を容易に記述するための記述方法を提案する. またこの関連はドキュメントをアスペクトによって関心事別に整理するための基盤となる情報である. 関連の記述は開発者にとってプログラム理解を助ける為に有意義であるものの,

関連の記述コストはうまくやらなければ非常に高くなってしまう. 従って我々は, 関連の記述のコストを最小限に抑えなければならない. 我々が行った関連の記述コストを削減する為に以下のアプローチをとった.

- 関連の構成要素の削減. 関連の複雑な構成要素で最低限, 必要なものだけを残す.
- 開発者の負担を低減するために, ソースコード中に関連を記述する.
- 関連の書式をシンプルにする.

最低限の関連の構成要素　我々は複雑な関連の構成要素を以下の 3 つの要素で表現できると考えた.

- 関連づけの対象ソースコードの位置.
- 関連するドキュメントまたはソースコードの位置.
- 関連を説明する文章.

関連づけの対象ソースコードの位置と関連するドキュメントまたはソースコードの位置のみを残す. これによってソースコードから直接追跡性のあるドキュメントを残すことができる. つまり我々が言及している関連は狭義の意味で, 直接的な追跡性の情報のみをいう. 従って, 関連の情報から間接的な追跡性の情報は失われる. 間接的な関連を補うために我々はアスペクトを導入した. アスペクトは直接的な追跡性を意味のある 1 つの集合としてまとめる.

アスペクトは人が埋め込む. このため, 全体としての関連の記述コストは変らない. しかし, 関連さえ記述されていればいつでもアスペクトを作成することが可能である. つまり開発者の仕事に余裕がないときでも関連さえ残しておけば, 仕事の余裕がでたときにアスペクトを作成することが可能になる. アスペクトについては 3.3.2 節でより詳細に説明する.

関連を説明する文章は, ソースコードから関連するドキュメントを取得するときや, ドキュメント関心事に基づいて 1 つのアスペクトにまとめる作業のとき, に読み手が必要かそうでないかを判断するために必要である.

関連の記述位置 (ソースコード中 vs 独立)　関連の記述コストはその構成要素数だけではなく, 関連を記述する位置にも左右される. 関連は以下の位置に記述することができる.

- ソースコードやドキュメントとは独立した別のファイル.
- 各ドキュメント中.
- 各ソースコード中.

一般的には, ソースコード中のコメントに重要度の低い記述を残すと, 本当に重要なコメントの発見がしにくくなり, ソースコードの可読性を下げることになる. ソースコードとドキュメントと独立した別のファイルに記述すると, ソースコード中に関連が現れなくなる. これによってソースコードの可読性を下げずに関連の記述が可能である. しかし, 本来関連は, 機械的に埋め込むタグのように人が理解不能な記述ではなく人にとっても理解有用であり, 機械処理も可能な記述である. また, 関連をソースコード中に記述することによって, ソースコードとの照合もとりやすくなるため関連の保守性も向上する. 従って, 我々はソースコード中に記述することがより有意義であると判断する.

これと同様の理由で関連をドキュメント中に記述するとソースコードとの照合がとりにくくなる上, 仕様書などのドキュメントによっては, 記述の変更や追記などができないものも多く存在するため, 関連の記述場所としては相応しくない.

関連の書式　関連はソースコード中に以下の書式でコメントとして記述する.

[@**説明**; **ドキュメントへの** URI; **キーワード**]

関連はソースコード中に記述するため, 関連の要素「ソースコードの位置」は記述しなくてよい. 関連の説明 (@説明) には関連するドキュメントの内容を簡潔に自然言語で記述する. つまり, この部分は機械処理を行わず関連の検索者への表示にのみ使われる. 関連の説明はアスペクトの抽出時とドキュメント参照時に利用される. この書式は簡単な文字列マッチングによって抽出することができるため, ツールの実装が容易である.

キーワードは関連をアスペクトで容易にまとめるための機能で, 開発者によって設定される. 同一のキーワードに設定された関連はそれが 1 つのアスペクトとして表現される. このキーワードは機械的に処理することが可能で, かつ読み手がそのキーワードからどのような目的で 1 つにまとめたアスペクトであるか推測可能な文字列である. なお, キーワードの詳細については 3.4 節で説明する.

3.3.2　アスペクト

2 つのアスペクトの関係は以下の 3 つの種類がある.

- 排他的アスペクト:2 つのアスペクト間に共有するアスペクトが 1 つもない状態 (図 3.2 (1) 参照).
- 共有関係のアスペクト:2 つのアスペクト間に 1 つ以上の関連を共有する状態 (図 3.2 (2) 参照).
- 部分集合になるアスペクト:1 つのアスペクトが他方のアスペクトを含む状態 (図 3.2 (3) 参照).

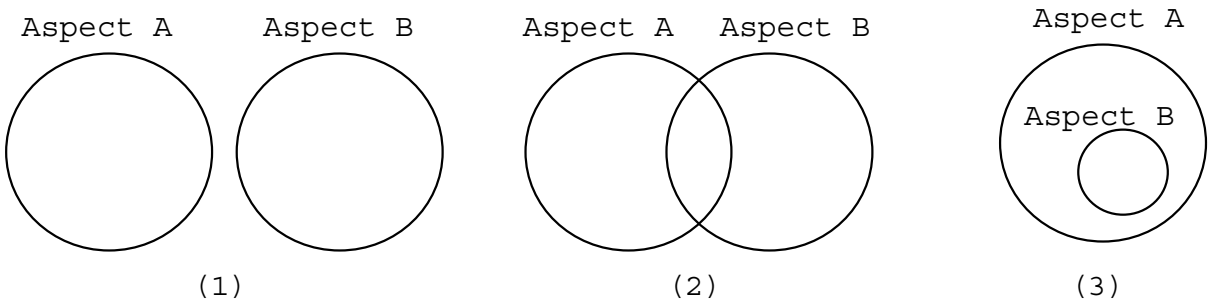

図 3.2: 2 つのアスペクト間の関係

排他的アスペクト

排他的アスペクトは, お互いのアスペクトに全く関連性はない. 読み手からみると, 互いのアスペクトの関連を参照し合うことがなく完全に分離している. また, 互いの排他的なアスペクトの関連が同一のドキュメントを参照していても, 双方向にアスペクトを参照しあうことはない. 例えば, 2 分木探索を用いた 2 つのプログラムから「2 分木探索のアルゴリズムを説明するドキュメント」を関連づけしているとき. ドキュメントは同一のものであっても, ソースコード間には関連性はない. このとき 2 つのプログラムが分類されているアスペクトは排他的である.

共有関係のアスペクト

共有関係のアスペクトは, 1 つの関連に対して 2 つの関心事があり, かつアスペクトの相互参照が有用であるときに共通部分を持つ. 例えば, 「テストデータ」に関するドキュメントは「性能」に関するアスペクトと「望みの動作をしているかどうか (機能)」というアスペクト 2 つをもつ. 通常, テストデータを評価するときは「性能」と「望みの動作をしているかどうか」という関心事は相互に見比べることは有用である.

部分集合のアスペクト

部分集合のアスペクトは, 比較的細かい関心事でまとめたアスペクトをより大きな関心事に基づいてまとめた場合である. この場合大きなアスペクトから詳細な情報を取得したいときに, 細かいアスペクトを取得し関連を限定していく. 例えば, バッファ機能などを備えた入出力関数を「標準入出力関数」というアスペクト (std_io とする) にし, 低水準な入出力システムコールを「低水準入出力関数」というアスペクト (low_io) にまとめたとき, アスペクト「低水準入出力関数」はアスペクト「標準入出力関数」のサブセットになる.

アスペクトが排他的アスペクトと部分集合のアスペクトだけの場合, 読み手が取得する関連は常に限定されていく. この構造は「本の目次」と同じ木構造である (図 3.3).

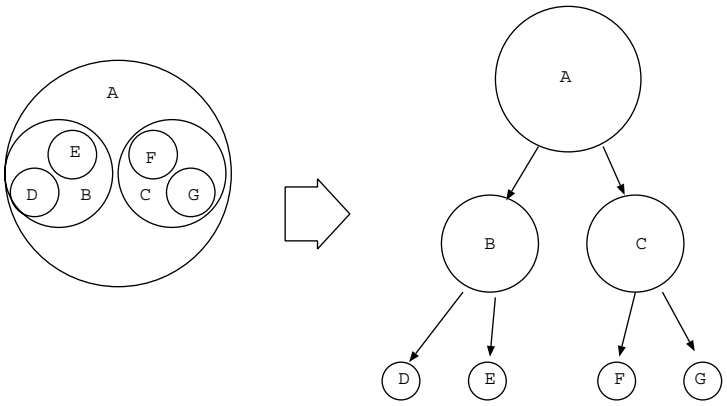

図 3.3: 排他的アスペクトと部分集合のアスペクトの組み合わせ

アスペクトの例 1(排他的アスペクト)

ここでは前述でも述べた「2 分木探索のプログラムとドキュメントの関連」の例を挙げる. 図 3.4 では 2 つの 2 分木探索を使うプログラムのソースコード (図 3.4(A) と (B)) に「アルゴリズムを説明する参考書」と「プログラム特有のメモ」を関連づけする.

プログラム (A) とプログラム (B) には全く関連性はない. そのために, この図ではアスペクトとしてそれぞれのプログラムで分離している. 例えばプログラム (A) を読んでいたときに, 「アルゴリズムを説明する参考書」を参照したときに, 「プログラム (A) 特有のメモ」が発見できれば理解に有用である. しかし「プログラム (B) 特有のメモ」の発見は読み手にとっては不要なものとする.

この場合, プログラム (A) と (B) からのびる「アルゴリズムを説明する参考書」へのそれぞれ関連から関連性がないため, 各プログラム別に「メモ」と「アルゴリズムを説明する参考書」とでアスペクトを作る (図 3.4 中央の◯). これによってプログラム (A) と (B) から参照される不要なドキュメントが除去できる.

アスペクトの例 2(共有関係のアスペクト)

共有関係のアスペクトは, 1 つ以上の関連を 2 つのアスペクト間で共有している状態である. この場合, 1 つ以上の関連から他方のアスペクトの関連を取得できる. これは, プログラム理解に必要な関連情報を関心事別で知ることが可能になる. プログラム理解のために有用である.

例えば, ソースコード (A) と (B) で構築したソフトウェアをデバッグをする場合を考える. プログラム (A) はプログラム (B) の関数を利用している. 各ソースコードには「プログラムのテストを行った結果」,「ソースコード (A) のデバッグに関するレポート」,「ソースコード (B) 実装時の開発メモ」が関連づけされている (図 3.5 参照).

デバッグを行う読み手は対象となるソースコード (A) を理解する必要がある. この時,

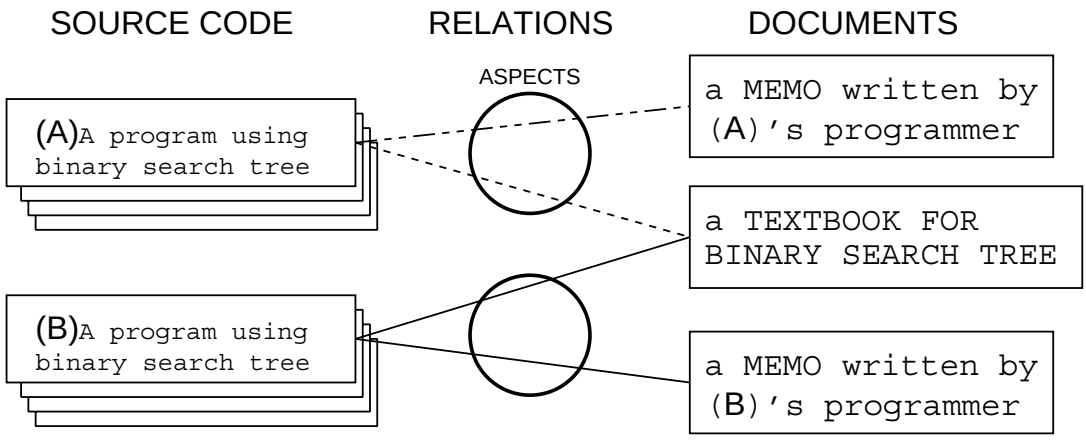

図 3.4: 排他的アスペクトの例

ソースコード (A) のアスペクトに付随する関連「デバッグに関するレポート」と「テストの結果」が参照できる. またソースコード (A) に付随するソースコード (B) があり, それらの理解も必要とするときソースコード (B) のアスペクトも参照することは, 読み手にとって有用である. この例ではソースコード (A) から, それに付随するドキュメントとソースコードをアスペクトから取得できる.

このように読み手の関心事によるアスペクトが共有関係である場合, 取得できるドキュメントは増える. つまり, 取得できる有用な関連を取得することができる. しかし, 共有部分を通してアスペクトを取得しつづけると読み手のもつ初期の関心事から離れた内容になっていく傾向がある.

アスペクトの例 3(部分集合になるアスペクト)

大きなアスペクトをより細かい視点で分類したアスペクトがこのような部分集合のアスペクトになる. 1 つの粗い粒度の関心事からより詳細なアスペクトへ情報をしぼることが可能である. 細かい視点で分類したアスペクトは詳細情報であり, そのドキュメントを参照する必要のある人が限定されている. このような関連をアスペクトでまとめ, プログラム理解の初期に必要以上に詳細な関連に始めによむべき概要の文章が埋もれることを防ぐ役割をする. 図 3.6 の例では, 全体の粗い関心事として「ハンドラ全般」についてアスペクトを作成し, より詳細なハンドラの「エラーコードのフォーマット」や「ハンドラの仕様」についてを「ハンドラの実装に必要な情報」として作成した. 読み手はハンドラに関心があった場合この「ハンドラ全般」のアスペクトを取得し, ソースコード (A) と (B) に付随するドキュメントとの関連「HOW TO USE HANDLER」とアスペクト「ハンドラ実装に必要な情報」を取得できる. このとき読み手はハンドラの概要 (使い方) のみに関心があったとき, 「ハンドラの使い方」以外のドキュメントはアスペクトとしてまとめられているため, 効率よく必要な関連を取得できる.

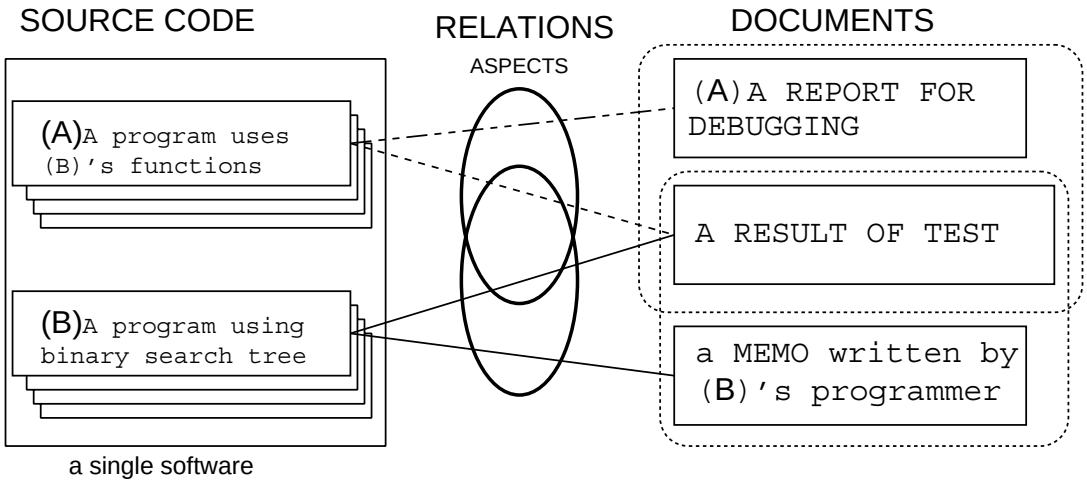

図 3.5: 共有関係のアスペクトの例

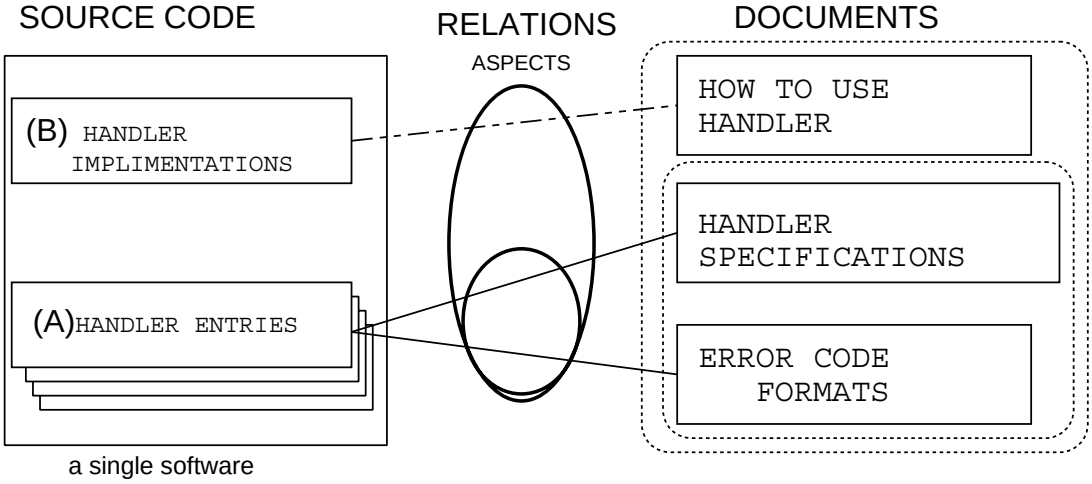

図 3.6: 部分集合になるアスペクトの例

3.3.3　アスペクトの作成例

前節ではアスペクトの種類について述べ, それぞれの種類に応じた例を示した. 本節では, 複数の種類のアスペクトが混合した例を示し, そこからドキュメントを取得するまでの流れを説明する.

関連の埋め込み

注目するソースコードを以下に挙げる.

1. 多重ループを持つソースコード (A と呼ぶ)
2. (A) を高速化するためのソースコード (B と呼ぶ)
3. (A) のテストコード (TA と呼ぶ)

またドキュメントを以下に挙げる.

1. ソースコード (A) のテスト仕様書 (TS と呼ぶ)
2. ソースコード (A) のテスト結果報告書 (TR と呼ぶ)
3. ソースコード (B) の最適化メモ (OM と呼ぶ)

ここに挙げたソースコードとドキュメントは関わりが深い. それぞれのソースコードに対応するドキュメントを関連として保存しておくことは, プログラム理解の点で有用である. この例はソフトウェアを構成する 1 つのモジュールを取り挙げる. ソースコード A は多重ループを持ち, システムのパフォーマンスのボトルネックになっている. ソースコード B はソースコード A のボトルネックを解消するためにループ中を最適化した代替のコードである. 開発者はコード A のテスト結果 (TR) から, 最適化のアイディアや考察などを記した最適化メモ (OM) を作成する. 以上の関連を図 3.7 に示す.

アスペクトの作成

図 3.7 では関連からアスペクトの生成をしている. アスペクトは開発者が関連をすべて埋め込んでから作成する.

プログラムのテストではテストコード作成のためにテスト仕様書 (TS) を参照し, その結果を (TR) に保存する. この 2 つの関連を「テストに関する資料」として 1 つのアスペクトにまとめる. ソースコード B の作成には, ソースコード A とソースコード A のテスト結果 (TR) を参照し, 最適化方法などのメモを残す. これを「最適化に必要な資料」として 1 つのアスペクトにまとめる. 最後に, すべての関連を 1 つのアスペクトとしてまとめる.

このようにアスペクトとしてまとめることによって関連を関心事別にまとめることができる.

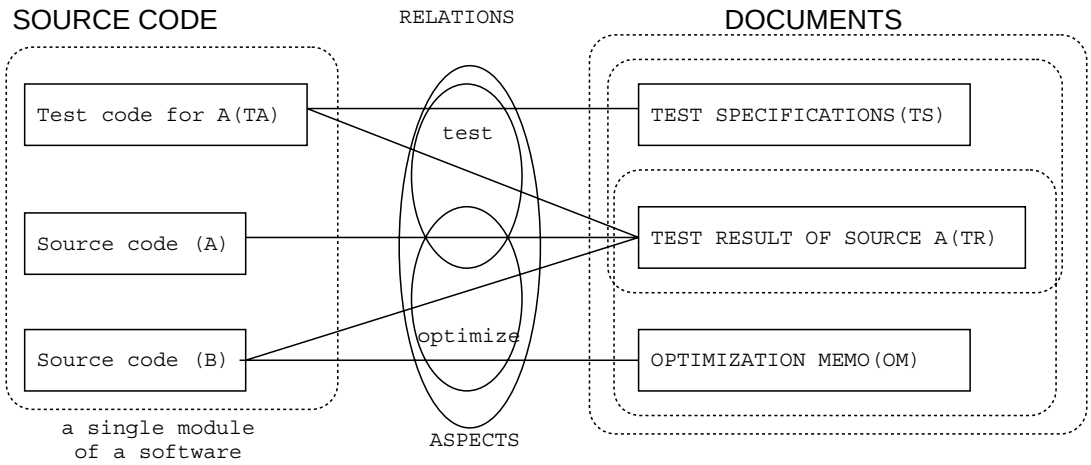

図 3.7: アスペクトの例

ドキュメントの取得

本節では, 前節でまとめたアスペクトと関連を使用して, ソースコード B のプログラム理解に必要なドキュメントの検索を行う.

ソースコード B ではソースコード A で行っている多重ループ中のボトルネックを最適化によって解消している. しかし, 一般的に最適化したコードにはプログラムの読み手にとってわかりにくいコードになることがよくある. 読み手はソースコード B を読解しているときにわかりにくい部分について, 開発者によって関連を利用して理解を進める. しかしソースコード B からたどれる関連はソースコード B の最適化メモ (OM) とソースコード A のテスト結果 (TR) だけである. テスト結果 (TR) では, テストの結果 (プロファイリングの結果やメモリの消費量) についての記述はあるがソースコード A のどこが問題であるか, またはソースコードへのリンクは記述されていない. このときにアスペクト「最適化に必要な資料」としてまとめてある関連を使えばソースコード A のボトルネックになっているコード断片も取得できる.

読み手は, 取得したソースコード A からさらに, アスペクト「テストに関する資料」を取得でき, テストに関する詳細な関連 (テストコード, テスト仕様書) を取得することができる.

3.4 キーワードによるアスペクトの作成

埋め込んだ関連の数が多くなると, その中からアスペクトを作成することが煩雑になる. これに対処するため, 関連の埋め込み時に, 予めアスペクトを作り, いくつかの関連をまとめておく必要がある. 本研究では, 容易にアスペクトを作るためにキーワードを導入した. キーワードは関連と一緒に埋め込まれ同一のキーワードをもつ関連を 1 つのアスペクトと

してまとめる. キーワードでまとめたアスペクトを他のアスペクトや関連と組み合わせることで, 新たなアスペクトを作成できる.

3.4.1 キーワード導入例

3.3.3 節で示した例にキーワードを適用する. 関連の埋め込みがすべて終ってから, アスペクトをまとめていた. しかし埋め込んだ関連の一覧からアスペクトをまとめることは煩雑である.

テストコード (TA) から「テスト仕様書 (TS)」と「テスト結果 (TR)」と「ソースコード (A)」への関連をキーワード「test」としてまとめ, ソースコード B からの関連「最適化メモ (OM)」,「テスト結果 (TR)」,「ソースコード (A)」をキーワード「optimize」によってまとめる. 「ソースコード (A)」にはキーワードが 2 つ設定されている. これは 2 つのキーワード「test」と「optimize」で作成されるアスペクトの共有部分であることを示す.

第4章　ドキュメント整理ツール ADIOS

3 章で提案したアスペクト指向によるドキュメントの整理法を適用したドキュメント整理ツール ADIOS を実験的に実装した.

本章では, ADIOS のシステム説明およびシステム実装のための関連技術について述べる.

ADIOS はソースコード中に埋め込んだ関連と, 作成したアスペクトをリポジトリへ, XML 形式で格納する. 読み手がソースコードからアスペクトを取得する場合, ADIOS を用いてリポジトリにアクセスする.

ADIOS での XML の利用

ADIOS は, リポジトリを格納するデータ形式として XML を採用した. 実装言語は Ruby 言語で約 1500 行で実装した.

Ruby を実装言語としたのは,

- 関連やドキュメントが日本語を中心としたテキストの処理が容易であったこと.
- XML パーサ REXML の実装があり, 日本語に対応していた.

からである.

本研究では, XML がテキストベースであり実装のデバッグに役にたつこと, XML を処理するための基盤技術が整備されていること, リポジトリの再利用性が向上することから, リポジトリのデータ形式として採用した.

4.1　リポジトリの設計

関連とアスペクトを保存するリポジトリは XML 文書である. XML 文書は論理構造をもつテキストデータであり, 人が直接読めることからデバグ時に有効である. また, XML 関連技術 (XML パーサ,DOM,SAX など) が多種のプラットフォームで実装されているため, ADIOS のリポジトリにアクセスする際に専用の API を作成しなくてすむ. これによって他のツールでのリポジトリの再利用が容易になる.

リポジトリは XML のルートノードとして “repository” ノードがあり, そのサブノードに, 関連のあつまりを格納する “relations” ノードとアスペクトのあつまりを保存する “aspects” ノードがある (図 4.1 参照).

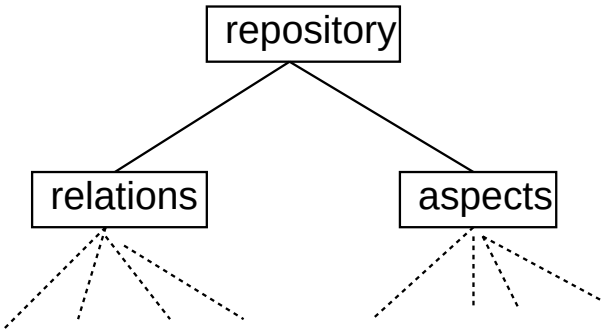

図 4.1: リポジトリ概観

関連　関連抽出ツールが抽出した関連は, リポジトリへ, ソースファイル別に格納する. 概念的な, リポジトリ中の関連は図 4.2 のようになる. リポジトリは, 関連をファイル別に XML による木構造で格納する. 以下にリポジトリに格納する関連の要素を挙げる.

- ソースファイルのパスとファイル名
- 関連識別子:一意に関連を識別するための値
- ドキュメントへの URI
- ソースへの URI
- キーワード
- 関連の説明

これらの要素は XML では図 4.3 のように記述する.

関連識別子は, 関連抽出ツールが自動的に付加する. この識別子はアスペクトが関連を指定する場合に利用される (次節で説明). 関連識別子の自動付加は以下の規則で作成する.

“ソースファイルへのパス” / “ファイル名” @ “ソース中のカウンタ番号”

アスペクトの記述　リポジトリに格納する要素を以下に挙げる.

- 関連識別子の集合
- アスペクトの説明
- アスペクト識別子

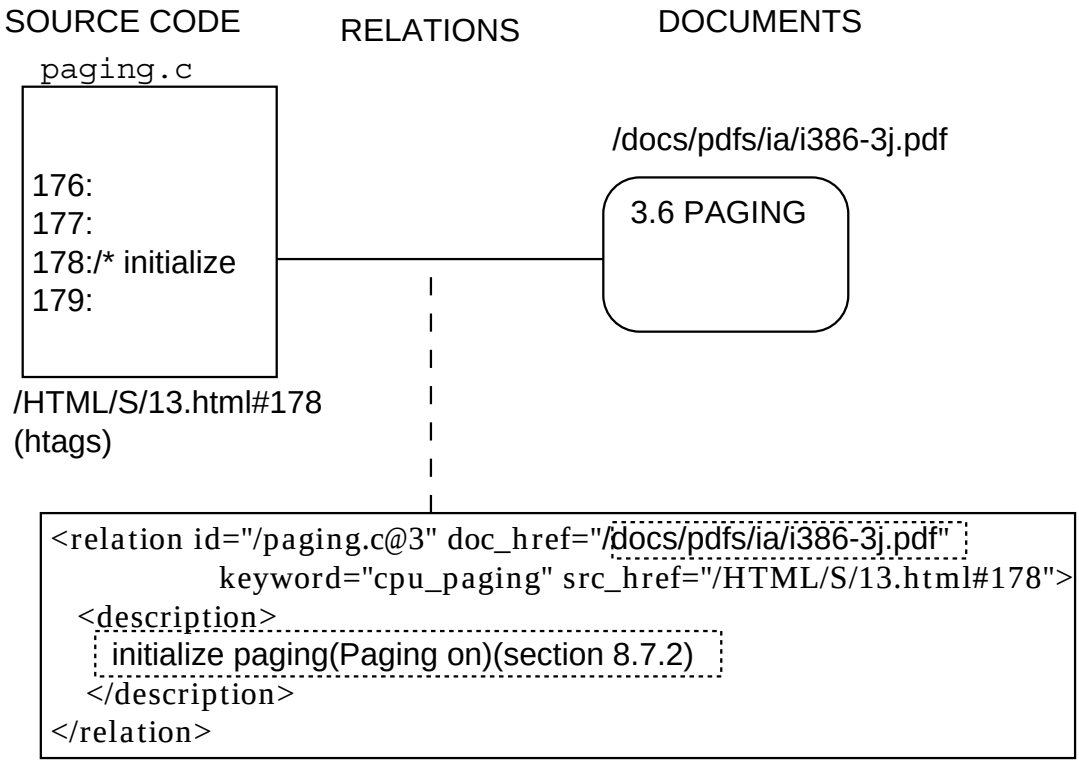

図 4.2: リポジトリ中の関連の表現 (例)

```
<relations name="プロジェクトのパス">
 <source name="ファイル名">
   <relation id="関連識別子" doc_href="ドキュメントへの URI"
             keyword="キーワード" src_href="ソースへの URI">
      <description>関連の説明</description>
   </relation>
 </source>
</relations>
```

図 4.3: リポジトリ中の関連のフォーマット

```
<aspects>
  <aspect id="アスペクト識別子">
    <description>アスペクトの説明</description>
    <relation_references ids="関連識別子の集合"/>
  </aspect>
</aspects>
```

図 4.4: リポジトリ中のアスペクトのフォーマット

これらの要素は XML では図 4.4 のように記述する.

関連識別子はリポジトリ中の関連に記述されている. アスペクトのメンバに含めたい関連識別子を, リポジトリ中に記述することによってアスペクトを表現する. また, アスペクトにはどのような関心事に基づいてアスペクトがまとめてあるのかを説明する簡潔な文が必要である. アスペクト識別子は, アスペクト表示ツール (後述) でこれらのアスペクトを操作するために必要である.

リポジトリに格納したアスペクトとソースコードとドキュメントのつながり (関連) との関係は図 4.5 のようになる. アスペクトの作成者は, アスペクトをリポジトリへ直接編集する. 関連抽出ツール (後述) によって作成される, デフォルトのアスペクトは, 関連中に埋め込まれたキーワードと同一ファイルにある関連でまとめたものである. これは自動的にリポジトリへ入力される. また, 実験的に同一ファイルにある関連をひとつにまとめたアスペクトもデフォルトのアスペクトとして作成する.

キーワードによるデフォルトのアスペクトのアスペクトの説明には, キーワードそのものが設定される. アスペクト識別子は関連抽出ツールが自動付加する. ADIOS は 1 つの関連に設定できるキーワードは 1 つである. 従ってキーワードのみによる. 共有関係のアス

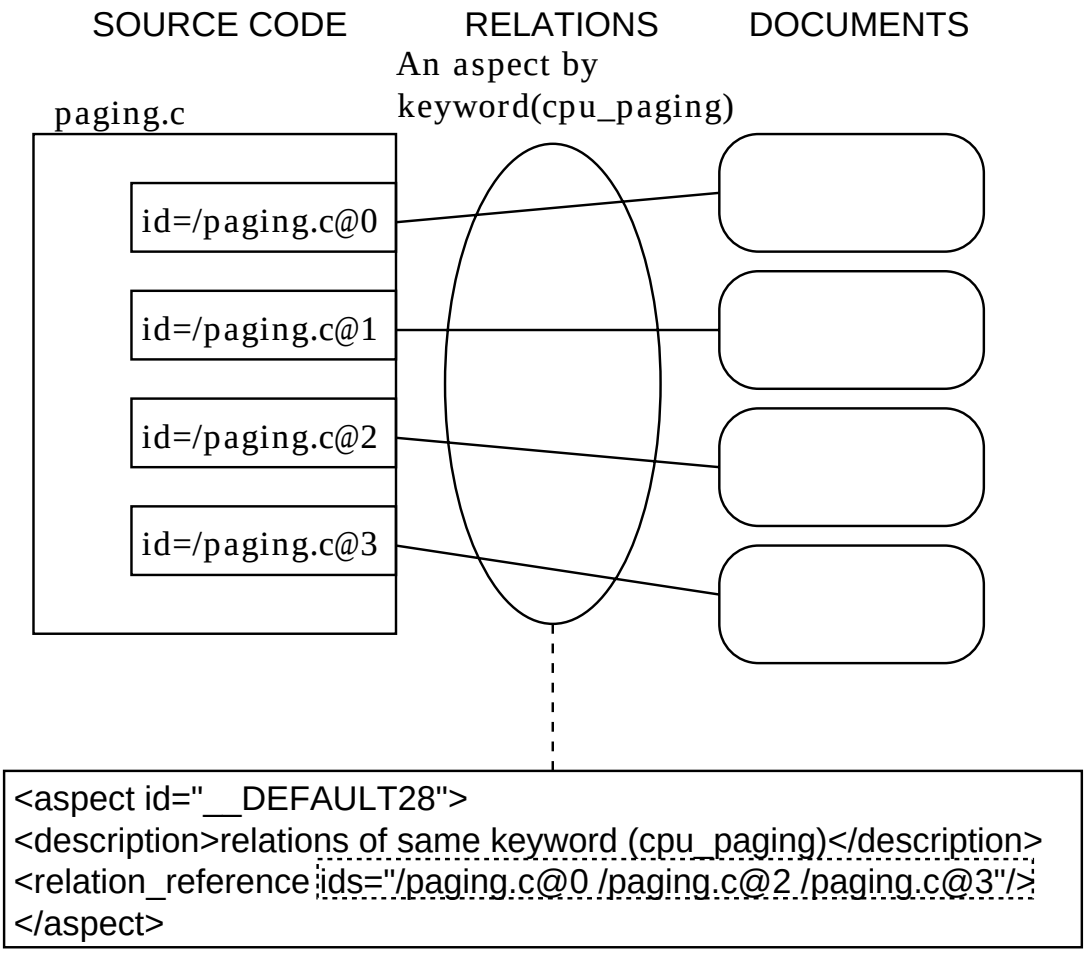

図 4.5: リポジトリ中のアスペクトの表現 (例)

ペクトの生成はリポジトリを直接編集で実現する.

4.2 ADIOS の実装

4.2.1 システム構成

ADIOS は以下に挙げる 2 つのツール

- 関連抽出ツール (Relations extractor): ソースコード中に埋め込まれた関連と, キーワードによるアスペクトを抽出してリポジトリに格納する (図 4.6 中 p1〜p4).
- アスペクト表示ツール (Repository Viewer): 関連抽出ツールで作成したリポジトリを操作し読み手のユーザーインターフェースを提供するツール (図 4.6 中 r1〜r3).

で構成されている.

ADIOS は関連とアスペクトの整理だけではなく, GNU GLOBAL[16] から生成されるソースコードの html を利用してクロスリファレンサとしても動作する. リポジトリ中の関連には 3.3.1 節で述べた通り, ソースコードの位置, ドキュメントの位置, 関連の説明からなる. 関連抽出ツールはソースコードと対応する GLOBAL の html ファイルに対応する位置の URI をリポジトリへ保存する.

関連抽出ツール

関連抽出ツールは, 1 つのプロジェクトを構成するソースコードをすべて読み込み, そこに埋め込まれている関連をリポジトリに格納する. 関連抽出ツールは, 同一のキーワードを持つ関連を 1 つのアスペクトとし, リポジトリ中の “aspects“ノード以下に格納する. それ以外のアスペクトは関連抽出後, リポジトリを編集してアスペクトを記述する. この時, アスペクト識別子は, 重複しない値をアスペクト作成者が決める.

アスペクト表示ツール

アスペクト表示ツールは, 小規模な Web サーバとして動作する. 読み手は汎用の Web ブラウザでこの Web サーバにアクセスする (図 4.7 参照).

図 4.7 では以下の作業が可能である.

1. ソースコード中からドキュメントとアスペクト一覧の取得 (図 4.7 上部フレーム)
2. アスペクトの一覧の中からアスペクト中の関連を取得 (図 4.7 下部フレーム)

新しいドキュメントを取得した場合, ADIOS は Web ブラウザの新しいウィンドウで表示する.

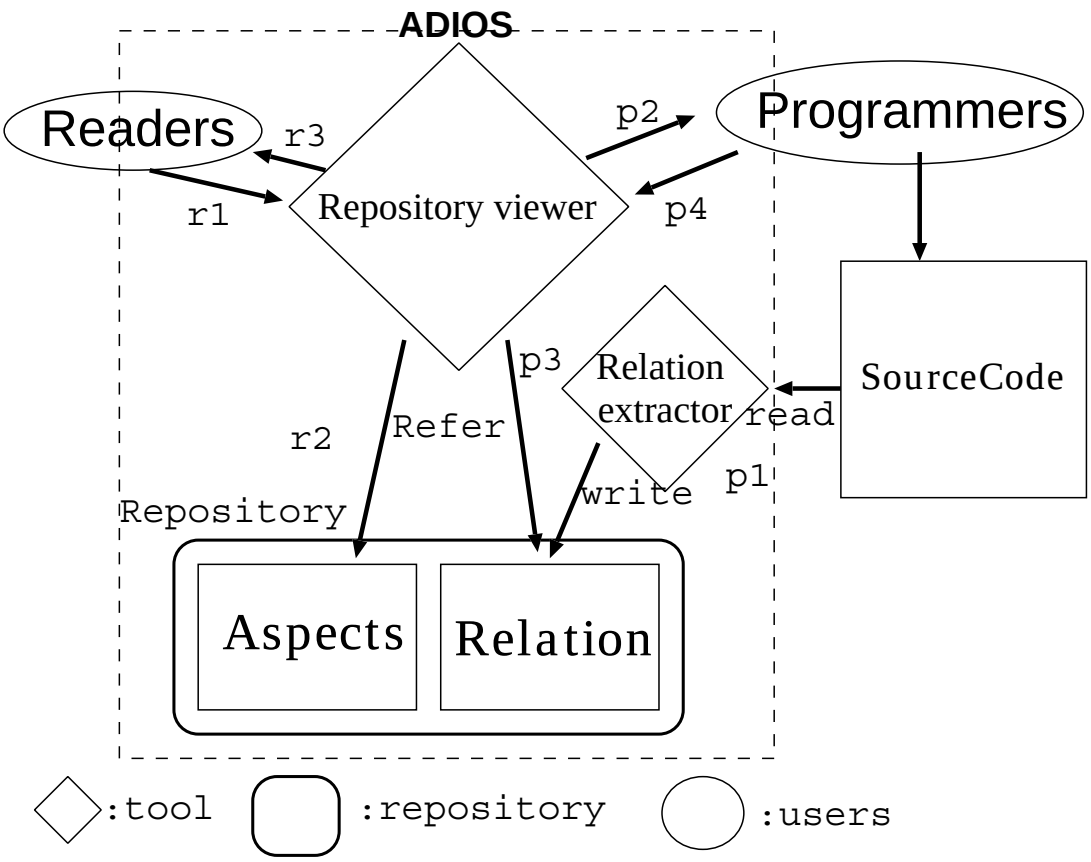

図 4.6: ドキュメント整理ツール ADIOS の概観

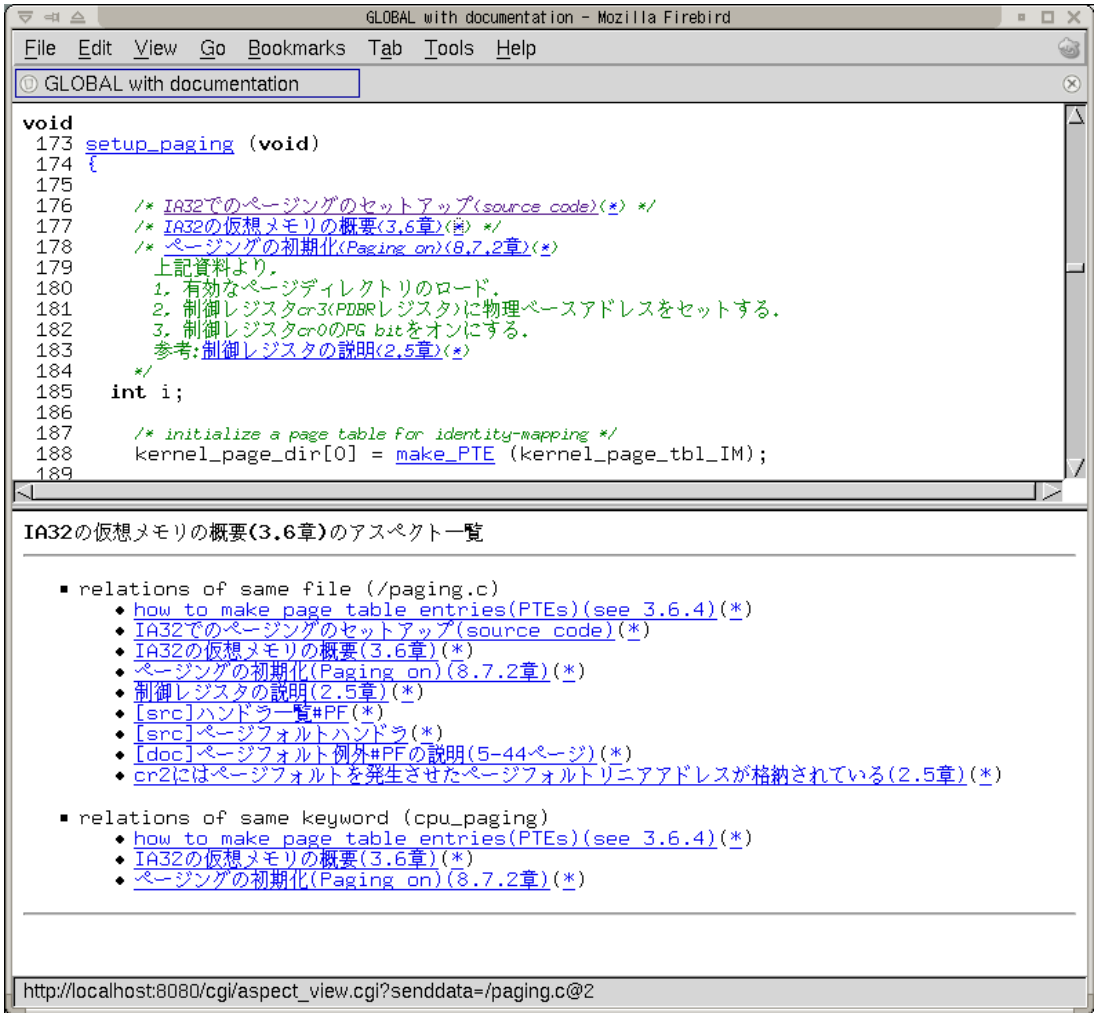

図 4.7: ADIOS の実行画面

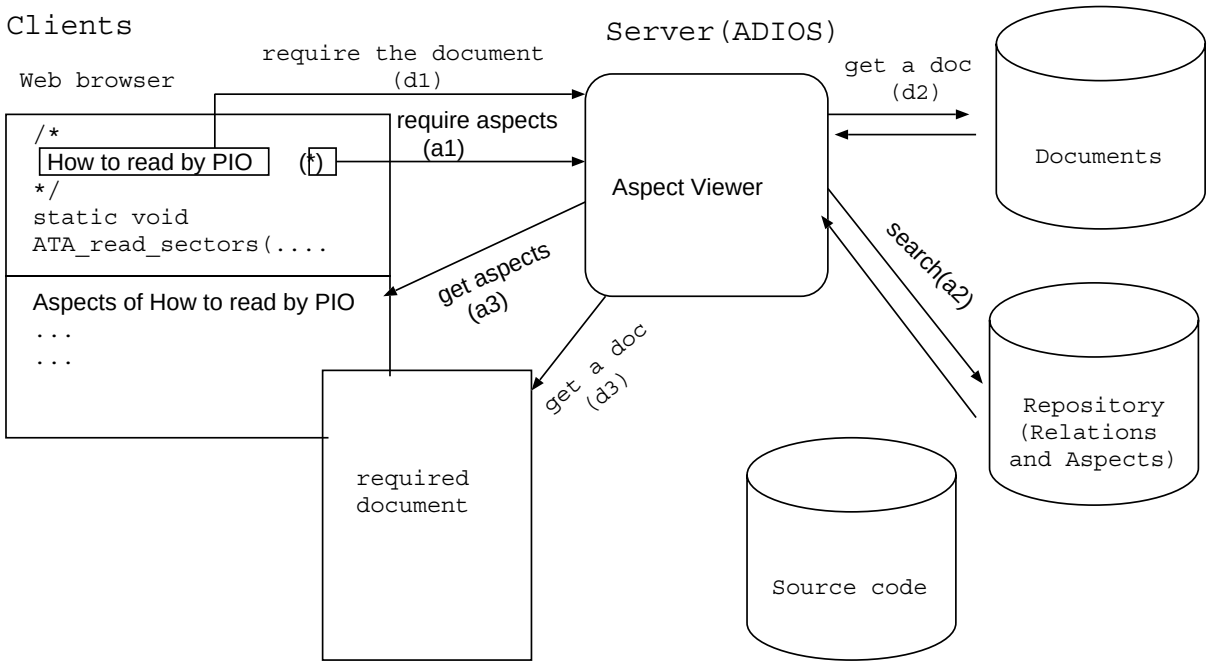

図 4.8: ソースコードからアスペクト一覧の取得

ソースコード中からのアスペクトとドキュメントの取得　始めて読み手からアクセスされたときアスペクト表示ツールは, ソースコードブラウザとして働く (このとき図 4.7 の下部フレームは無表示). Web からみたソースコードにはコメント中に埋め込まれた関連がハイパーリンクとして表現される. 読み手はここから関連するドキュメントとアスペクトを得ることができる.

アスペクトの取得をする場合, 読み手はコメント中の関連の「(*)」を選択する. ブラウザは読み手から「読み手の選択した関連のアスペクト一覧要求」としてアスペクト表示ツールへクエリとして送信する (図 4.8 (a1)). クエリを受けとるとアスペクト表示ツールは, リポジトリ中から対象となるアスペクトとそれに付随する関連の集合を取得し (図 4.8 (a2)), Web ブラウザの下部フレームにアスペクトと関連一覧を送る (図 4.8 (a3))

また, ソースコード中の関連からドキュメントを取得する場合は, 読み手の選択したドキュメントをアスペクト表示ツールが検索し, Web ブラウザの新しいウィンドウとして表示する (図 4.8 d1-d3).

アスペクトの一覧から関連の取得　読み手が図 4.7 の上フレームから, アスペクト表示ツールにアスペクトの一覧表示を要求し, 下部フレームにその一覧が表示される. 下部フレームには各アスペクトとそれに付随する関連の一覧が表示される. その一覧から, ドキュメントとソースコードの位置を取得することができる.

アスペクトに付随する関連からソースコードを取得する場合はアスペクト一覧中の「(*)」

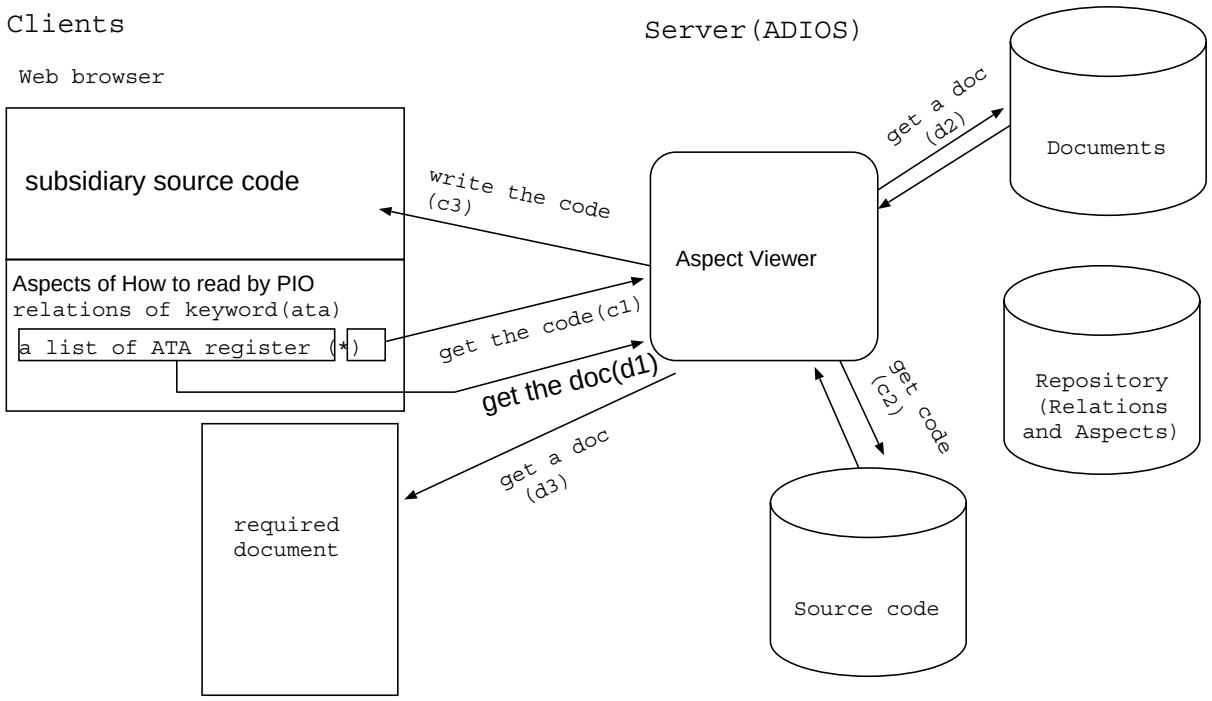

図 4.9: アスペクト一覧から関連取得の流れ

を選択することで得ることができる. ソースコードを取得する場合はクライアントから「読み手の選択した関連のソースコード取得要求」をクエリとしてペクト表示ツールに送信する (図 4.9 (c1)). アスペクト表示ツールはそのクエリを受けとるとソースコードを検索し (図 4.9 (c2)), 結果を, 図 4.7 上部フレームに返す (図 4.9 (c3)).

また, ドキュメントを取得する場合は, 読み手の選択したドキュメントをアスペクト表示ツールが検索し,Web ブラウザの新しいウィンドウとして表示する (図 4.9 d1-d3).

アスペクト表示ツールは動的にリポジトリや関連を検索する. これは, 現在の機能にはないが, 将来アスペクトを閲覧しながら抽出することを目的としているからである.

関連の取得例

図 4.7 中に示されているソースコードから「IA32 の仮想メモリの概要」を選択したときに得られるアスペクトの例を示す. 全体の流れを図 4.10 に示す. この関連が埋め込まれている関数 (setup_paging()) は, OS が仮想メモリを使用できるようにページングの初期化を行う.

読み手は (1) 仮想メモリ全体の把握したいという関心事に基づいて (2)「仮想メモリの概要」を選択し関連するドキュメントを選択する. ここには, インテルアーキテクチャ(IA32) の CPU 上で仮想メモリを扱い方の概要が述べられている. しかしページフォルトが起きたときに実際にどのような処理が, どの関数で行われるのかはまだわかない.

このとき (3) 選択した関連のアスペクトの一覧を取得すると以下のアスペクトが取得できる.

1. OS の基礎的な学習用のアスペクト.「仮想メモリについて」「ページングアルゴリズム」などの関連. ただしこれは対応するソースコードはないためページングのソースコードの先頭に埋め込んである.

2. 性能向上のテクニックについてのアスペクト.「トランスレーション・ルックアサイド・バッファ(TLB) を使ったページアクセスの高速化」に関する関連.

3. 他の関係する実装についてのアスペクト. 実際にページフォルトが発生したときに呼び出されるページフォルトハンドラとそれに付随するドキュメントの関連など.

(4) 読み手はページングについての疑問を解決するために 3 の関連から, ソースコードの位置を取得する. すると図 4.7 の上部に page_fault_handler 関数が表示される. (5,6) また, 3 のアスペクトからドキュメント「ページフォルト例外の説明」に関するドキュメントも取得できる. また, page_fault_handler 関数にジャンプすることによって, そのソースコードに埋め込まれているより詳細な情報, 例えば「ページフォルトハンドラのエラー引数の説明」に関するドキュメントなどが取得できる.

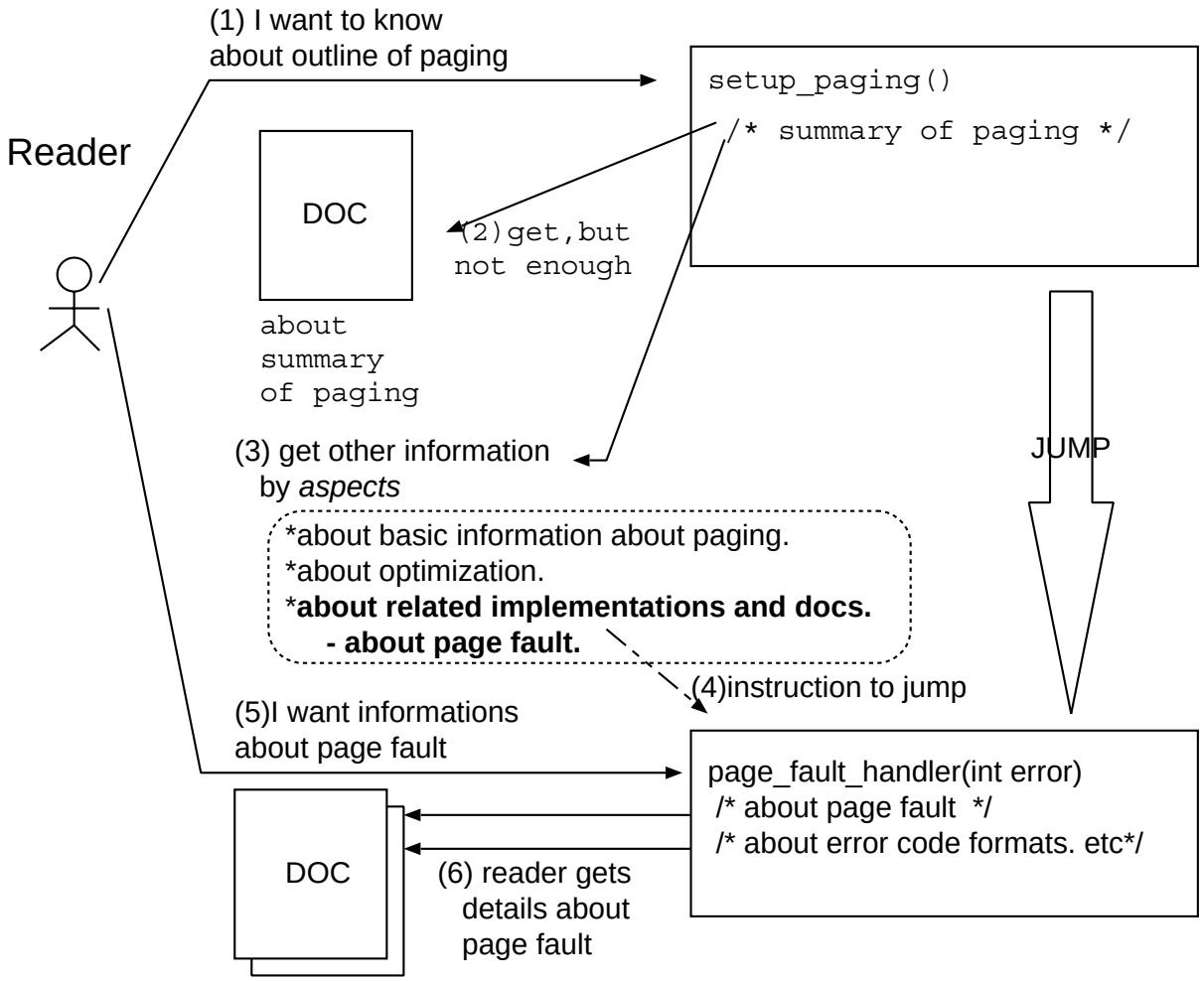

図 4.10: 例ページングに関する関連取得の流れ

第5章　ADIOSのOS教材 udosへの適用

本章では, 4 章で述べたドキュメント整理ツール ADIOS を使って, 独自開発中の OS 教材 udos[18] を事例に, ドキュメントの整理を行い, 予備評価を行う. 事例の対象としたのは OS 教材 udos のページングに関するドキュメントである. ページングに関するドキュメント整理だけでは規模的にみて, 十分に評価できなかったため予備評価とした. 予備評価の結果, 部分的ではあるがアスペクトを作成し, プログラム理解に有用なドキュメントを取得することができた.

5.1　udosの概要

udos は, 本研究で独自に開発中の OS 教材である. 開発コンセプトは, 32 ビットインテルアーキテクチャプロセッサ (以後 IA32) を使用し, マルチタスク, 仮想メモリ, I/O を備え最小限の実装 (現在約 2000 行) で実現を目指した.

OS のプログラム理解は, OS の概念を理解するだけではできない. これは概念で扱っているデータ構造やアルゴリズムを実装するためのドキュメントが, 非常に多いためである. 例えば, ページングを扱うだけでも, CPU の基本的な概念, CPU の仕様書, 実装するプラットフォーム (例えば PC/AT) に関する仕様書. ページ置換アルゴリズムの参考書, 性能測定の結果, OS の仕様書などが必要になる. またそれらが, 大きなドキュメントであることも多い.IA32 の仕様書 [6, 7, 5] は約 2000 ページ, BIOS のマニュアル [3, 2] は約 1000 ページもある.

5.1.1　仮想記憶の概念

仮想メモリに関するドキュメントを関連づけし, アスペクトでまとめる. 仮想メモリとはプログラム中のアドレス (仮想アドレス) と物理メモリのアドレス (物理アドレス) を分離独立する記憶方式である. 仮想記憶の利点は,

- 実メモリよりも大きなメモリ空間を透過的に扱え,
- プロセス毎にメモリ空間を別にできる.
- プログラム (バイナリ) の共有が可能になる.

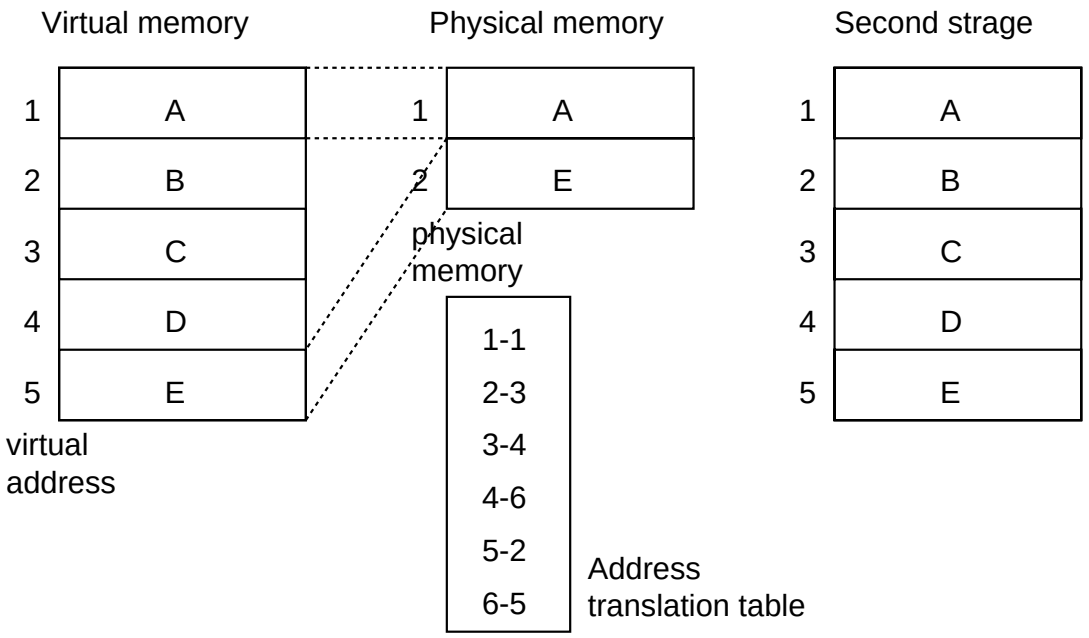

図 5.1: ページングの概念図

である.

仮想メモリを実現する方法として, スワッピングと要求時ページングがあるが, 本研究が扱うのは後者である. ページングの実現には, 仮想アドレスと物理アドレスの対応関係を明らかにしたアドレス変換表が必要である (図 5.1 参照). ページングには, アドレス変換表を基に物理アドレスと仮想アドレスの相互変換を行い, 未割り当ての仮想アドレスへのアクセスが要求 (ページフォルト) されたときに, OS に伝える MMU(memory management unit) が必須である. IA32 は MMU を CPU の機能としてもつ.

ページングに関して, プログラムを理解するには, 前節で述べた概念だけではなくより詳細なハードウェアの知識が必要である. 実際のハードウェアや OS は, メモリ効率や速度, 下位互換のための古い規則があり複雑である. 例えば, IA32 で仮想アドレスと物理アドレスとの変換をするテーブルは単純な配列ではなく, サイズ節約のためにカスケード接続になっている. さらに, PTE(Page Table Entry) などのハードウェア固有のフォーマットは, ドキュメントを参照しなれば理解できない. 従って関連によってソースコードからドキュメントを取得することはプログラム理解に有用である.

5.2 ページングに関するソースコード

クロスリファレンサを使って得られるページングの関数の呼び出し関係は図 5.2 のようになる. それぞれの関数は以下の処理を行う.

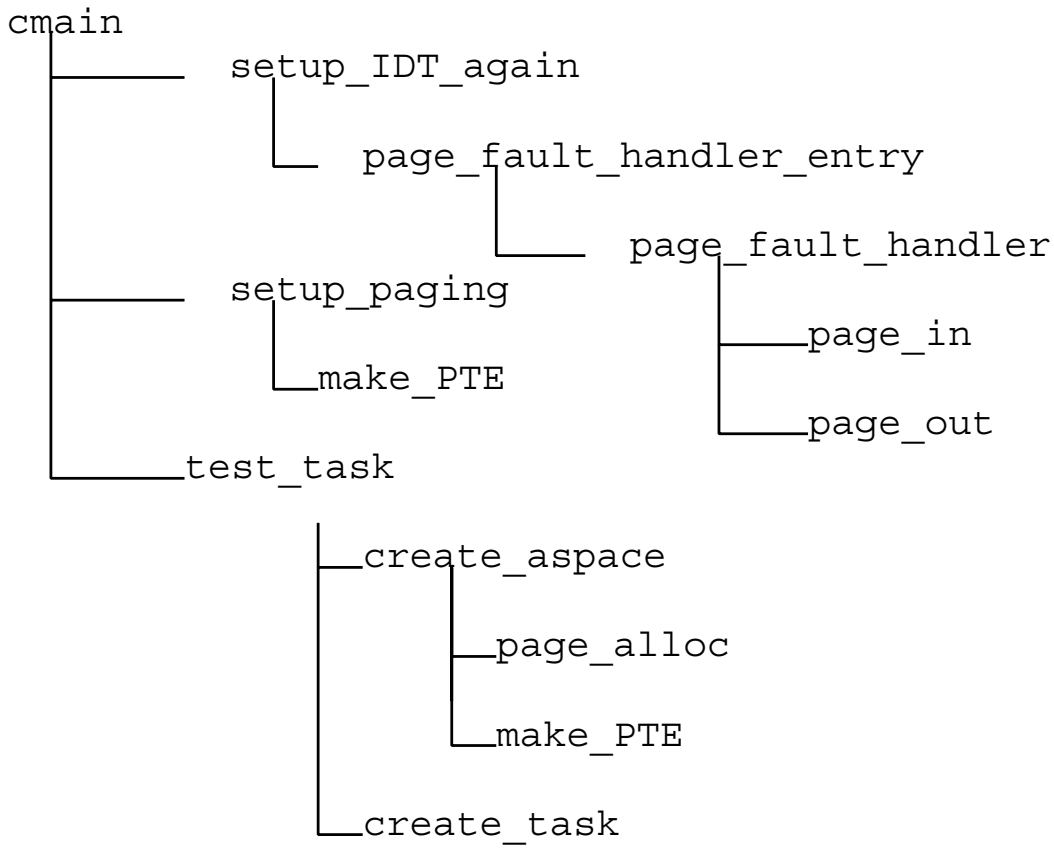

図 5.2: ページングに関する関数の呼び出し関係

- cmain カーネルの起動後, 初期設定を行う.
- setup_IDT_again 例外ハンドラを設定する.
- page_fault_handler_entry ハンドラを起動するためのエントリ
- page_fault_handler ページフォルトハンドラ本体
- page_out ATA ハードディスクにページを書き出す
- page_in ATA ハードディスクからページを読み込む
- setup_paging ページングの初期化
- make_PTE ページテーブルエントリを初期化する.
- create_task ユーザープロセス用のタスクステートセグメントの初期化
- create_aspace ユーザープロセス用のメモリの確保

- page_alloc ページの確保 (空き管理用のビットマップから空いている物理アドレスを得る)

5.3 関連の抽出

ページングのソースコード中に埋め込んだ関連を以下に挙げる.

1. setup_paging 関数:ページングの初期化を行う関数.
   (a) ページングの概要. (キーワード introduction)
   (b) (IA32) ページングの初期化手順.
   (c) (IA32) 制御レジスタの書式.
2. page_fault_handler 関数:ページフォルトが起きた時に呼ばれる関数.
   (a) ページ置換アルゴリズムの概要. (キーワード introduction)
   (b) ページフォルトハンドラの動作説明. (キーワード introduction)
   (c) ページイン/ページアウトのリファレンスマニュアル.
3. page_in/page_out 関数:ページイン/ページアウトを行う時の呼ばれる関数.
   (a) ATA ハードディスクのリファレンスマニュアル.

5.4 アスペクトの作成

ソースコード中に残した関連からアスペクトを作成した.

1. キーワード introduction によるアスペクト.
2. 関連「制御レジスタの書式」は, 関連「ページング初期化手順」でも知る必要があるため**「IA32 上でのページングの初期化方法と関連資料」**という関心事に基づいた 1 つのアスペクトとした.
3. 関連「ATA のハードディスクのリファレンスマニュアル」と関連「ページイン, ページアウトのリファレンスマニュアル」は**「ページイン/アウト時の補助記憶デバイスの扱い方」**という関心事に基づいた 1 つのアスペクトにした.

キーワード introduction によるアスペクトは, 関連抽出ツールによって自動的に作成されるアスペクトである.

setup_paging 関数では, カーネル空間のページテーブルを作成し, 制御レジスタ cr3 にベース物理アドレスを設定する. その後, 制御レジスタ cr0 の PG ビットをオンにすることでページングを有効化する. この関数を理解するためには, 各制御レジスタの書式, IA32 の仮想メモリ概要などの情報が必要である. これはソースコード中に埋め込んだ関連から取得することができる. またアスペクトから間接的な情報を取得することができる.

5.5 ドキュメントの取得

setup_paging 関数から読む例を考える. まず全体的な処理の流れをつかむために「ページングの概要」を読む. またこの関数からは直接呼び出さないが, この概要にはページフォルトに関する記述がある. キーワード introduction によってまとめられたアスペクトからドキュメント「ページフォルトハンドラの動作説明」とページフォルトハンドラ (page_fault_handler) を取得することができる.

「ページイン/アウトのリファレンスマニュアル」を読むとき, より詳細な情報を, アスペクト「ページイン/アウト時の補助デバイスの扱い方」から「ATA ハードディスクのリファレンスマニュアル」が取得できる.

5.6 予備評価

5.6.1 関連の埋め込み

関連はソースコードとドキュメントをつなぐ役割がある. しかし, 必要な関連はソースコードからドキュメントだけではなくソースコードからソースコードへの関連も必要であることがわかった. また, 関連づけするドキュメントがソースコードの特定の位置に付随するものではなく, ソースコード全体に付随するドキュメントがあることがわかった. 例えば, ページング概要を説明したドキュメントはページングのソースコードの特定の位置ではなく, 全体に関連するドキュメントである. 本手法ではこれらを完全にサポートすることができない. しかし, ソースコードからソースコードへの関連は, クロスリファレンスによって解決できると考えた. また, ソースコード全体に関連するソースコードは, 最も関連性の高い特定の位置に関連づけをした.

5.6.2 アスペクトの取得

アスペクト「キーワード introduction」中のドキュメント「ページングの概要」には, ページフォルトやページ置換アルゴリズムについても述べられている. 関連だけでは「ページングの概要」からページフォルトやページ置換アルゴリズムについて検索することはできない. アスペクト「キーワード introduction」によってページフォルトハンドラ関数のソー

スコードやそれに付随するドキュメントを取得できた.

第6章　議論

6.1　コメントと関連の関係

コードが本当の意味で難解な場合, 理解の手がかりを示すコメントを書くと読み手が助かる. 親切なコメントには参考文献が明記されている [11].

本研究では, ソースコード中のコメントは重要であると考える. しかし無駄の多いコメントはソースコードの可読性を下げるだけでなく, 本当に重要なコメントが埋もれてしまう. 特に関連するドキュメントのまる移しなどは, プログラムの理解を低下させる. 必要なドキュメントは「関連」として記述すれば, ソースコードの可読性を損なわず, プログラム理解のための情報を付加できる. 例えば, 複雑なアルゴリズムを実装する場合, 使用するデータ構造の説明, アルゴリズムと実装の説明などのドキュメントの関連が, プログラム理解を助ける.

例えばページングのセットアップに関していえば以下のようになる.

```
void
setup_paging(void)
{
  /*
  IA32 の仮想メモリの概要 (関連「IA32 の仮想メモリの扱い方」)
  ページングの初期化方法 (関連「IA32 の仮想メモリの初期化方法」)
  1, 有効なページディレクトリのロード.
  2, 制御レジスタ cr3(PDBR レジスタ) に物理ベースアドレスをセットする.
  3, 制御レジスタ cr0 の PG bit をオンにする.
  制御レジスタの説明 (関連「制御レジスタ書式」)
  */
  ...
```

このように, コメントと関連を組み合せて記述することで, ソースコードの可読性を下げずに, 有用な情報を付加すことができる.

6.2 ADIOS の実装について

6.2.1 ソースコードへの関連づけ

ADIOS ではソースコード中に埋め込んだドキュメントを関連づけする. しかし udos へ適用した結果, ソースコードからソースコードへの関連づけが必要であることがわかった. ソースコードへの関連づけを実現するときの問題は, ソースコードの位置をどのように指定するかということである. 解決方法は 2 つ考えられる.

1. 関連先のソースコードにタグを埋め込み, その場所への URI を記述する.
2. ソースコードへの URI を “ファイル名” と “行番号” の組で指定する.

1 は, 関連先のソースコード中に, ソースコードを説明する目的以外のコメント (タグ) が混入し, 可読性が落ちてしまうことを予想する. 2 は, ソースコードの変更に弱くなる.

6.2.2 関連識別子の作成ルールの問題

関連識別子は以下の規則で作成する.

“ソースファイルへのパス”“ファイル名”@ソース中のカウンタ番号”

この規則では, 関連の埋め込み位置の変更という観点でみれば, 行番号による生成よりは変更に強い. しかし, 関連の埋め込み順を変えるだけで対応ができなくなる. これは, 関連を作成しながら動的なアスペクト作成のサポートをする場合に問題になる. この解決策として, 関連を埋め込む人自身が関連識別子も記述することで利便性が下ってしまう. これは今後の課題である.

6.2.3 ソースの URI(src_href) の問題

ADIOS では, htags によって作成された html ファイルの URI をプロジェクトのルートディレクトリからの相対パスで表現している. しかしこの URI は htags への依存性を高めてしまう. これを解決するには, ソースコードの URI として識別子 (ファイル名など) で表現し, htags の URI との変換テーブルを用意することで解決できると予想する.

6.3 関連の埋め込み作業の妥当性

関連を形式的に記述する作業コストは, コメントとして参考文献を記述するコストと変らない. さらに形式的に記述することによって, 本研究で示したようにアスペクトによって分類でき, プログラム理解にとって有用である.

6.4 アスペクトの動的な作成

6.4.1 RDF の ADIOS への導入

セマンティック Web の研究で利用されている形式である. 現在ある Web の情報は人間には理解可能であるが, 機械が理解できる情報ではない. セマンティック Web は機械的に Web 上の情報を扱い, 推論による情報検索などを行う技術をいう. セマンティック Web では Web 上の情報にメタデータによって機械的に処理する. このメタデータを記述するためのデータモデルが RDF(Resource Description Framework) である. RDF での最も基本的なメタデータの表現は, を主語 (Subject), 述語 (Predicate), オブジェクト (Object) である.

例えば (RDF 仕様書 [4] より),

> “http://www.w3.org/Home/Lassila. の作成者は Ora Lassila ある.”

は RDF で表 6.1 のように表現できる.

主語	http://www.w3.org/Home/Lassila
述語	Creator
オブジェクト	“Ora Lassila”

表 6.1: RDF の表現例

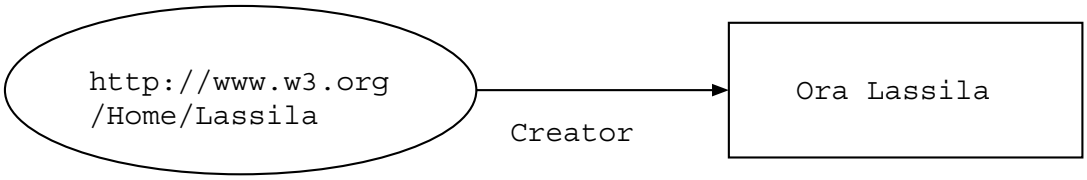

図 6.1: RDF の例

この RDF のデータモデルは, 本研究の関連の記述に適用することができる. 本研究での関連の説明には, 人間が理解でき, かつ, アスペクトの抽出ができるようドキュメントの内容を忠実に記述することが求められる. しかし, 現時点ではアスペクトの抽出は人間が行っている. メタデータの記述に RDF 使うことで, 機械的な処理と人間による意味の推測が同時に可能になる. これによってアスペクト作成のコストが下げられると予測する.

導入例

関連の説明の例を以下に挙げる

主語	http://hoge/doc/paging
述語	仕様書
オブジェクト	ページング

これを散文で書けば,

“リソース http://hoge/doc/paging/はページングの仕様書である”

となる.

RDF を適用する場合に考えられる問題は, メタデータの記述につかう語彙集を作らなければならないことである. またその語彙集がすべてのソフトウェアにおいて汎用的につかえるかどうかを検討しなければならない.

6.4.2 教材への応用

現在の関連の取り出し方は, 最初に読み手がソースコードを読んで, その中から関連を取り出し, 付随するアスペクトを取り出し, その中から関連を選択する流れである (図 6.2 参照). この流れは, ソースコード中で疑問に思ったことから, 関連するドキュメントを取得し, さらにアスペクトから有用な他の関連を取得する. しかし教材作成ツールとして, ADIOS をみた場合, ソースコードを読む前に必要なアスペクトを検索できる機能が教材作成ツールとして有用であると考える (図 6.3 参照). これを実現する為には, アスペクト検索用のクエリ言語が必要である. アスペクトの説明を RDF で記述できれば, クエリにマッチするアスペクトの機械的な検索が容易になると予想する.

6.5 仕様書のソースコード中への記述 vs 関連

WEB 言語 [12] では, 関連するすべての文書をコード中に残す. コード中にドキュメントを埋め込むことによって, ドキュメントに変更があったときにソースコードとドキュメントとの整合性がとりやすくなるというメリットがある. しかし, 本当に難解なソースコードに付随するドキュメントが, 他の重要性の低いドキュメントに埋もれてしまいソースコードの可読性が下る.

本研究では, ソースコード中に残すコメントは本当にプログラムが難解なものに限定し, それをさらに補足するものとしてドキュメントへの関連を残す. それによって, コメントの有用性が向上し, さらに補足するドキュメントも取得可能になる.

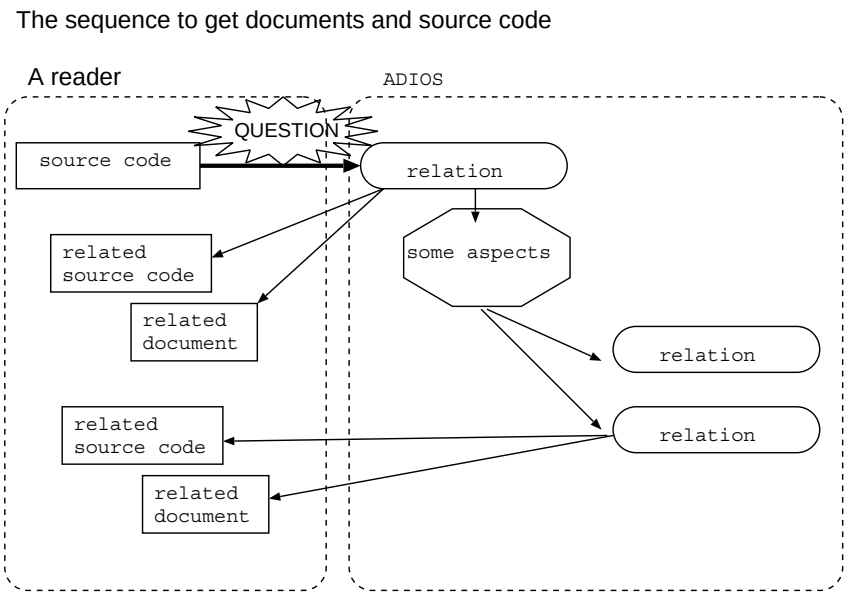

図 6.2: 現在の手法での関連抽出の流れ

6.6 キーワードを使った全文検索システム Namazu[15] によるドキュメントの検索

複数のドキュメントの中からドキュメントを検索する場合, 全文検索システム Namazu[15] が広く使われている. Namazu の一般的な使い方として, 文章中にキーワードを入力しておきそれを検索するといった手法がある. しかしソースコードとドキュメントの両方を, この手法によって関連づけする場合, ソースコードとドキュメントの両方にキーワードを埋め込まなければならず煩雑である. また同一のキーワードでアスペクトを作成するとき, Namazu で抽出したアスペクト中の関連には多対多の関係しか取り出せず, ソースコードとドキュメントの 1 対 1 の対応がわからなくなる (図 6.4 参照). ソースコードとドキュメントの対応をとるために, 関連ごとに 1 つずつキーワードを設定すると, アスペクトとして抽出するときにキーワードの演算 (インターセクション等) が煩雑になる.

本研究で提案する整理手法では関連として 1 か所にキーワードとドキュメントへの関連を書いておくだけで, アスペクトでまとめてもソースコードとドキュメントとの対応は失なわない.

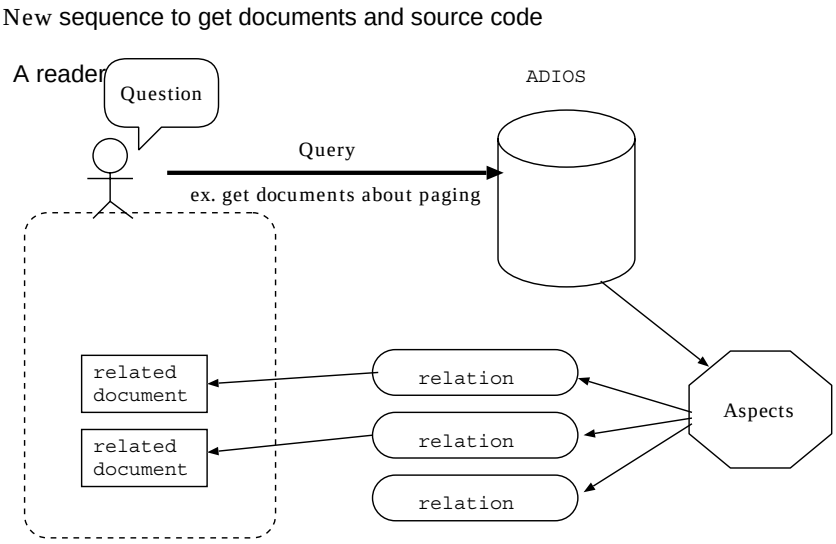

図 6.3: クエリによる関連抽出の流れ

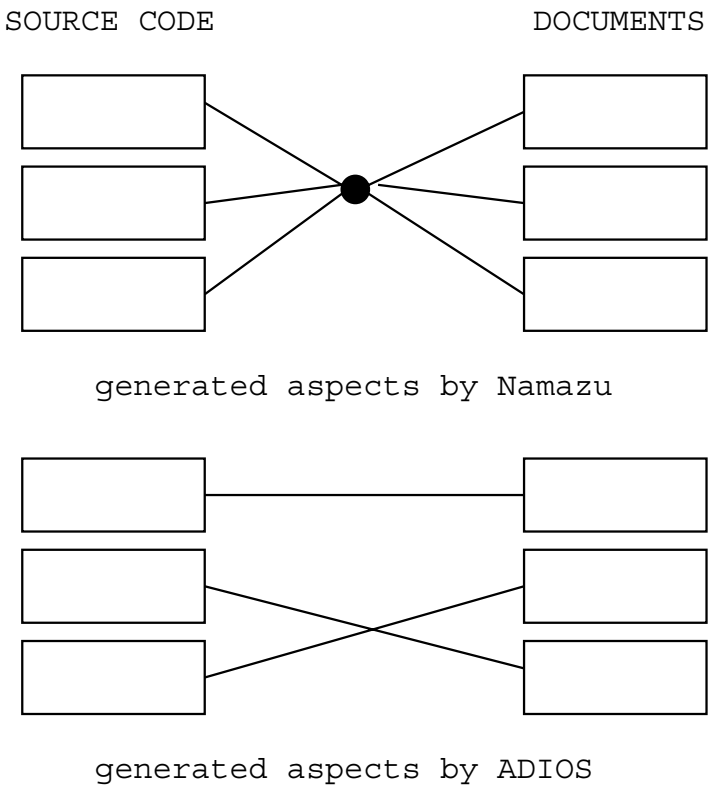

図 6.4: 全文検索システム Namazu との比較

第7章　関連研究

7.1　GNU GLOBAL[16]

GNU GLOBAL は C, C++, Yacc, Java, PHP, アセンブラのソースコードに対応するクロスリファレンサである. GLOBAL は静的にソースコードを解析し, 関数定義元などの検索を行うツールである. GLOBAL は, ソースコードの静的解析ツール gtags, ソースコードに関数定義などの検索用タグ付けした html を出力するツール htags, ユーザーのクエリ毎に検索する global(-r など) で構成されている.

7.1.1　クロスリファレンサの概要

クロスリファレンサとは, ソースプログラムの構成要素間をリンクし, それらの関係を明らかにするツールである. 例えば, 関数の定義部分や参照部分を見つけたり, 複数のソースコードから構成されるソースファイル間の参照を可能にする. 例えば, 既存のツールには GLOBAL[16] や Cxref[13] などがある.

GLOBAL は厳密にソースコードを構文解析していないので, typedef や関数ポインタを追跡できないなどの問題がある. 逆に厳密に解析を行えば詳細なクロスリファレンス情報を得ることができるが, 言語毎に厳密に解析しなくてはならなくなり実装コストの面で不利である.

本研究で提案する ADIOS は, コメントの位置さえわかればよいので多言語に対応できる. この長所を生かすために, 多言語に対応している GLOBAL を採用した. ADIOS の対応する言語は Java, C, アセンブラである.

htags によるクロスリファレンスによって, ADIOS は, 読み手の関数の呼び出し関係の理解をサポートする. さらに関連やアスペクトによって, ソースコードから抽出できない情報の取得をサポートする. これによって, クロスリファレンサの機能以上に効率よくプログラムの理解が可能になる.

7.2　XML と関連技術の概要

拡張可能なマークアップ言語 XML(eXtensible Markup Language)[9] は SGML(StandardGeneralized Markup Language) のサブセットとして規定されたメタタグ言語である. XML は 1998 年に

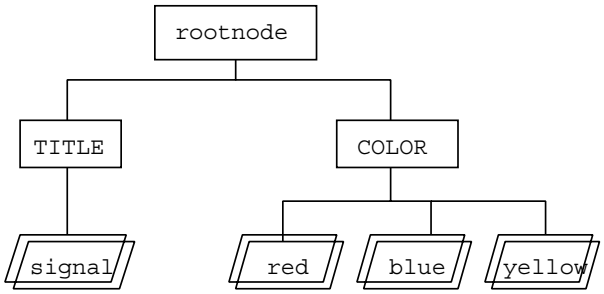

図 7.1: XML 文書の木構造

W3C(World Wide Web Consortium) によって制定された. XML の設計目標を以下に示す.

- XML は, インターネット上でそのまま利用可能である.
- XML は, 広範囲のアプリケーションをサポートする.
- XML は,SGML 互換である.
- XML 文書を処理するプログラムが簡単に書ける.
- XML における付加機能はできるだけ少なくし, 理想的には 1 つの存在しない.
- XML の設計は, 迅速に行う.
- XML の設計は, 形式的かつ簡潔である.
- XML のマークアップ数を減らすことは重要ではない.

XML によって記述されたドキュメントを XML 文書と呼ぶ. XML 文書は論理構造をもつテキストデータである. また,XML は論理構造に関する制約を記述するための機構を提供する.

XML 文書は, XML 宣言, 文書型定義 (DTD), 開始タグ, 終了タグ, 属性, 文字列から構成される. 開始タグから終了タグにかこまれた範囲を 1 つの要素と呼ぶ. 要素はその内部にさらに別の要素を含むことができる. このとき外側の要素を親要素, 内側の要素を子要素という. 親要素と子要素の関係によって XML 文書は全体で 1 つの木構造になる (図 7.1). XML 文書を処理するプログラムを書く場合, テキストとして与えられている XML 文章を解析し, メモリ中に論理構造としてアクセスできるように格納する XML パーザが必要である. また, 読み込んだ XML 文書を操作するインターフェースがある.

W3C によって XML 文書を操作するためのプログラミングインターフェースとして DOM(Document Object Model) と SAX(Simple API for XML) が用意されており, 多くの言語で実装されている.

7.3 Javadoc[17], DJavadoc[8], doxygen[19]

Javadoc[17], DJavadoc[8], doxygen[19] はドキュメント生成のサポートに主眼をおいたツールである. これらのツールは, 特殊なコメント中に関数やクラスを説明するドキュメントを記述し, それを抽出し, 整形して html で出力する (例, 図 7.2 参照). また, 関連するドキュメントの URI を埋め込むこともできる. しかし, ソースコードとドキュメントとの関連をまたがる横断的関心事に基づいて, ドキュメントを整理することはできない. また, ドキュメント生成用のコメントをスコード中に記述することは, ソースコードの可読性が低下する原因となる. これはドキュメント管理の利便性とのトレードオフである.

クラス Sample

java.lang.Object
└Sample

public class Sample
extends java.lang.Object

Javadocの出力サンプル Javadocはコメント中に記述したドキュメントを整形して

関連項目:
java.util.Vector

フィールドの概要

private static int	constant1 クラス変数

コンストラクタの概要

Sample() コンストラクタ.

メソッドの概要

int	getConstant1() クラス変数1をかえす

図 7.2: javadoc によるドキュメント生成の例

7.4 WEB[12]

WEB 言語 [12] はプログラム理解に主眼をおいたプログラミング言語である. ソースコード中にプログラムの詳細な説明を記述し, プログラム構成間をつなぐ. tangle, weave によってプログラムの生成とドキュメンテーションを 1 つのソースコードから行える. ソースコード中にすべて記述することによってプログラマはドキュメントを残しやすい.

コード中にドキュメントを埋め込むことによって, ドキュメントに変更があったときにソースコードとドキュメントとの整合性がとりやすくなるというメリットがある. しかし, 本当に難解なソースコードが, 他の重要性の低いドキュメントに埋もれてしまい可読性を低下させる.

7.5 セマンティック Web

現在の Web 技術は, Web 情報を人間が読み, 人間が理解し, 人間が操作することを前提としている. Web 情報から得られる情報は膨大であり, 検索エンジンによって抽出したドキュメントを, 人間が内容を読みさらに必要な情報であるかを判断しなくてはならない.

セマンティック Web は, Web 情報に機械処理できる意味情報を付加することで, 人間のかわりにより精度の高いドキュメントの検索を行う. セマンティック Web は, 図 7.3 に示すアーキテクチャを持つ.

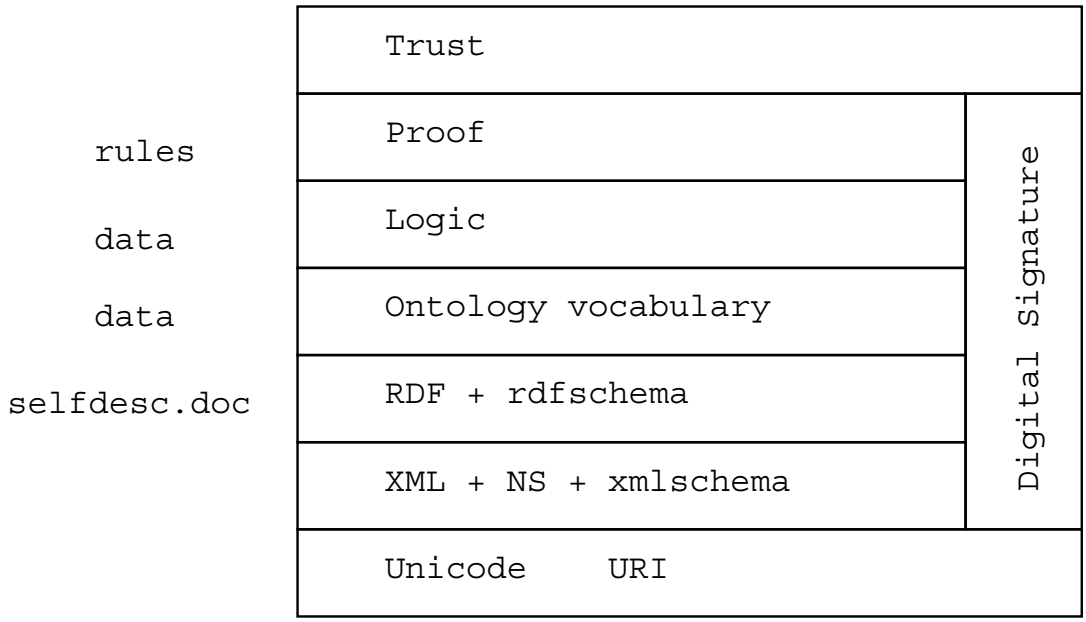

図 7.3: セマンティック Web のアーキテクチャ

図 7.3 のレイヤーについて以下で説明をする.

- XML: Web 作成者がタグを自ら定義し, テキストなどに付加することにより, 論理構造をもたせることができる. しかしここでは論理構造の意味については言及してない.
- RDF: RDF は XML に対して論理構造の意味を規定するものである. 意味の規定はメタデータを付加することで行う. メタデータは主語, 述語, オブジェクトの組み合わせで記述する.
- Ontorogy: 例えば表現方法の異なる 2 つのメタデータが同じ意味を持っている. 場合, 語彙間の共有化を行うための, 用語間の関係を正式に定義している文章またはファイルをオントロジーという.
- Logic: メタデータの論理的な処理することで Web ページを導きだす仕組みを提供するレイヤー. 論理的な処理とは, 述語論理や否定などである.

第8章　おわりに

8.1　まとめ

本研究で我々は多種のドキュメントをアスペクト指向を用いて整理する手法を提案した. 本手法を適用したドキュメント整理ツール ADIOS を実装し, 独自開発中の OS 教材 udos のドキュメントの整理法をページングにのみ適用し予備評価を行った. ADIOS を使うことによって, ソースコード中に埋め込んだ関連からドキュメントを取得することができた. 関連からアスペクトを取り出し, 関連性の深いソースコードとドキュメントを取得することができた. 関連をコメント中に形式的に記述することで, ソースコードとドキュメントの関連を容易に収集することができた. 関連の形式的な記述はコメント中に低コストで記述できたと考える. 関連を, アスペクトとしてまとめることで, 小規模な udos の事例に限っては, ソースコードからの関連以外にも関連を取得することを確認した. しかしアスペクトの作成は, キーワードによる省力化を提案し, ある程度アスペクトの抽出のコストを柔らげることができたが, より多くのアスペクトの整理には, まだ工夫が必要である. アスペクトの整理には関連の説明が必要不可欠である. しかし関連の説明は自然言語で記述している. このためアスペクトの整理に関しては手動で行わなればならない. この説明を RDF などを利用したメタデータとしての記述するアイディアで, アスペクトを整理する手間を省ける可能性がある.

8.2　今後の課題

本論文で示した例は小規模のドキュメント整理であり, 本手法における問題点は完全に明確になったわけではない. 今後, アスペクト間の関係 (共有関係など) を分析するため, より大規模な適用を行い. 問題点の明確化と本手法の有用性を確認しなければならない.

今後の課題として, RDF を使ったアスペクト抽出の省力化とソフトウェア開発における語彙集 (オントロジー) の検討. および RDF からエージェント等を用いたアスペクトの検索手法の提案を行う.

また, 本研究で実装した ADIOS は, 埋め込んだ関連をプログラミングを一旦中断して, リポジトリに取り出してから, リポジトリ中の関連を検討しなければならなかった. 今後, プログラミング中にも動的にアスペクトの追加を行えるような開発環境の提案を目指す.

謝辞

本論文の執筆にあたって始終熱心な御指導を頂きました片山卓也教授ならびに権藤克彦助教授, 貴重な御意見と有意義な議論を共にして頂きました川島勇人さん, 並びにソフトウェア基礎講座の皆様に感謝を申し上げます.

参考文献

[1] Home page for aspect-oriented software development(AOSD) community and conference.
http://aosd.net/.

[2] OADG テクニカル・リファレンス (DOS/V) バージョン 6 対応. 310 ページ.

[3] OADG テクニカル・リファレンス (ハードウェア). 310 ページ.

[4] Resource description framework(RDF) model and syntax specification.
http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[5] インテル・アーキテクチャ・ディベロッパーズ・マニュアル 下巻: システム・プログラミング・ガイド. 658 ページ, 資料請求番号 243192J.

[6] インテル・アーキテクチャ・ディベロッパーズ・マニュアル 上巻: 基本アーキテクチャ. 368 ページ, 資料請求番号 243190J.

[7] インテル・アーキテクチャ・ディベロッパーズ・マニュアル 中巻: 命令セットリファレンス. 850 ページ, 資料請求番号 243191J.

[8] Erik Berglund. *Use-Oriented Documentation in Software Development*. PhD thesis, Linkoping Studies in Science and Technology, 1999.

[9] Tim Bray, Jean Paoli, and C.M.Sperberg-McQueen. Extensible markup language(XML)1.0.
http://www.w3c.org/TR/1998/REC-xml-19980210.

[10] Andrew Forward and Timothy C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering*, pp. 26–33. ACM Press, 2002.

[11] Brian W. Kerninghan and Rob Pike. プログラミング作法. アスキー, 2000. 福崎 俊博 訳, ISBN4-7561-3649-4.

[12] Donald E. Knuth. 文芸的プログラミング. アスキー出版局, 1994. 有澤 誠 訳, ISBN4-7561-0190-9.

[13] Andrew M.Bishop. The Cxref homepage.
http://www.gedanken.demon.co.uk/cxref/.

[14] Masaru OHBA. Home page for Aspect oriented Documents Inspection System(ADIOS).
http://www.jaist.ac.jp/~m-ohba/adios/.

[15] Namazu project. Namazu, a Full-Text Search Engine.
http://www.namazu.org/.

[16] Yamaguchi S. Homepage for GNU Global-Source Code Tag System for C, C++, Java and Yacc.
http://www.gnu.org/software/global/.

[17] Sun Microsystems. Javadoc for generating API documentation.
http://java.sun.com/j2se/javadoc/.

[18] udos project. Home page for micro-disk-operating-system(udos).
http://www.jaist.ac.jp/~m-ohba/udos/.

[19] Dimitri van Heesch. Homepage for Doxygen, a documentation system.
http://www.doxygen.org/.

[20] 鯵坂恒夫, 佐伯元司. プロセスと環境トラック 方法論工学と開発環境. ソフトウェアテクノロジーシリーズ 7. 共立出版, 2001. ISBN4-320-02780-9.

[21] 鈴木正人. ソフトウェア工学. サイエンス社, 2003. ISBN4-7819-1022X.