保守性向上へ向けた設計モデルとソースコード間のトレーサビリティ解析

Traceability analysis between design model and source code toward maintainability improvement

伊藤 弘毅*　志水 理哉†　鷲崎 弘宜‡　波木 理恵子§　田邉 浩之¶　深澤 良彰‖

あらまし　ソフトウェア開発において，成果物間のトレーサビリティを確保することはソフトウェアの保守性を保つ上で重要である．本研究では，設計モデルとソースコード間のトレーサビリティの解析法と，その結果を利用して保守性向上を目指す改善プロセスを提案する．トレーサビリティの解析は，GQM 法を利用したトレーサビリティの定量的測定と，両者の対応箇所と相違箇所をグラフィカルに示すトレーサビリティモデルの生成の２つから成り立っている．本手法を適用することにより，両者の修正作業やリファクタリングを支援し，ソフトウェアの保守性向上を促進することができる．また，本手法の有用性を評価するために，現実の設計モデルとソースコードの組を利用してケーススタディを実施した．結果として，改善プロセスを通じて両者の修正箇所やリファクタリング箇所の候補が導かれ，効率よく保守性を向上できることを確認した．

1　はじめに

近年のソフトウェアはサービスの多様化に付随して，より一層大規模化および複雑化してきている．その中で，ソフトウェア開発におけるトレーサビリティ―開発プロセスで生成された複数の成果物間の関連度合 [1]―の低下が問題となっている．

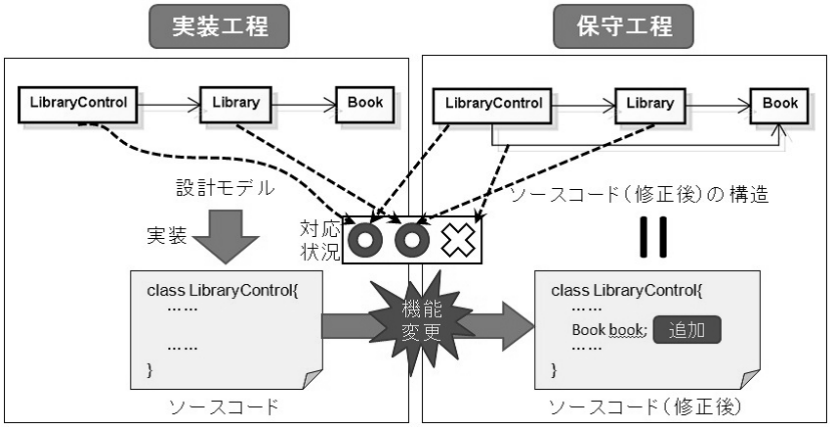

図 1　設計モデルとソースコードのトレーサビリティ低下の例

*Hiroki Itoh, 早稲田大学
†Masaya Shimizu, 早稲田大学
‡Hironori Washizaki, 早稲田大学
§Rieko Namiki, 株式会社オージス総研
¶Hiroyuki Tanabe, 株式会社オージス総研
‖Yoshiaki Fukazawa, 早稲田大学

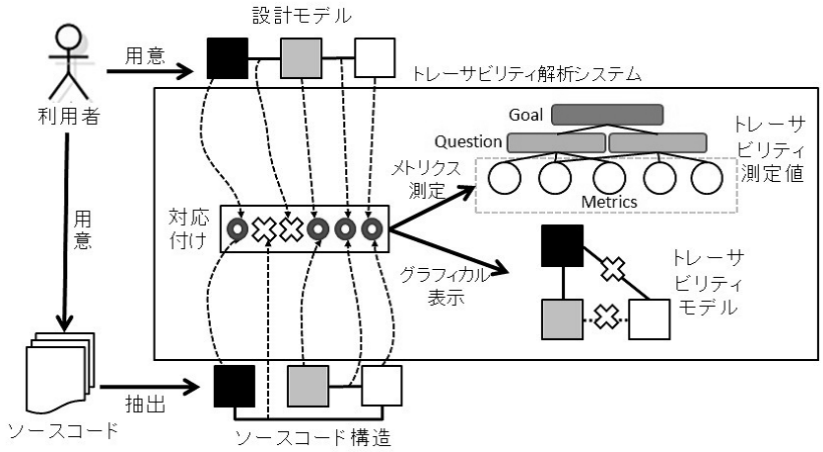

図 2　本手法の全体構成

図 1 に設計モデルとソースコード間のトレーサビリティが低下する具体的イメージを示す．設計段階で設計モデルを作成し，設計通りに実装する．この段階では，設計モデルの要素はソースコードの要素と対応付けられるため，両者のトレーサビリティは高い状態である（図の左側）．しかし，保守段階で仕様変更や機能修正などが生じた時，アドホックにソースコードを修正してしまうと，設計段階では考慮していなかった関連が追加される可能性がある（図の右側）．この時，併せて設計モデルを修正せずにいると，両者の間に相違が生まれてしまい（図の対応状況），トレーサビリティが低下してしまう．同様な形で相違が増えてくると，設計モデルが信頼できないものとなってしまい，ソースコードのみで保守しなければならなくなる．

このようにトレーサビリティが低下すると，ソフトウェアが理解しにくいものになり，今後の保守活動の効率が下がってしまう [2] ため，開発を通じてその状態を認識し，成果物間の対応をとり続けることが望ましい．しかし，開発の途中でそれを認識するには多くの時間と労力を要するため，実際の開発現場ではトレーサビリティが軽視される傾向にある．この問題を解決するためには，その負担を減らすことが求められ，実現するには自動化ツールのサポートが不可欠となる．

本研究では，設計モデル（UML [3] クラス図）とソースコード間のトレーサビリティを自動的に解析する手法を提案する．本手法を適用することで，容易にトレーサビリティの状態を認識できるほか，コストのかかる修正作業やリファクタリングを支援できるため，ソフトウェアの保守性を効率よく向上させることができる．

2　本手法の手順

2.1　本手法の全体像

本手法の全体像を図式化したものを図 2 に示す．

本手法では，ソフトウェアの保守性向上を目指して 2 つの成果物を生成し，利用する．1 つ目は設計モデルとソースコード間のトレーサビリティの得点である．トレーサビリティを定量的に示すことにより，両者の対応具合や傾向を客観的に認識することができるようになり，保守作業の方針を決定するのに役立てることができる．2 つ目は設計モデルとソースコードを比較して異なる部分を指摘するトレーサ

[0] トレーサビリティには同開発フェーズで生成された成果物間の関係を示す horizontal traceability と，成果物が開発を通じて持つ側面の関係を示す vertical traceability がある [4]．本手法では設計フェーズでの成果物・設計モデルと実装フェーズでの成果物・ソースコード間のトレーサビリティを解析するため vertical traceability を扱うこととなる．

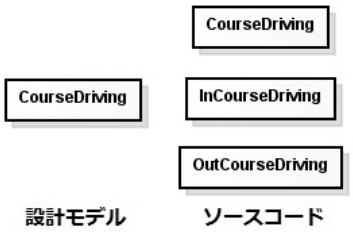

図 3　類似したクラス名の出現例

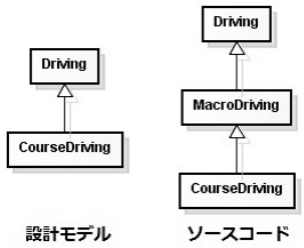

図 4　汎化関係にあるクラスの細分化の例

ビリティモデルである．トレーサビリティモデルを利用することにより，どの要素が異なっているかを一目で判断できるようになるため，モデルやソースの修正支援やリファクタリング箇所の特定に役立てることができる．

これら 2 つの成果物は以下の 3 つのステップから生成される．

Step1　設計モデルとソースコードの要素を対応付ける
Step2　設計モデルとソースコードの対応状況からトレーサビリティを測定する
Step3　設計モデルとソースコードの相違箇所を基にトレーサビリティモデルを作成する

これより Step1 から Step3 まで，順を追ってプロセスの詳細を記述していく．

2.2　設計と実装の対応付け

まずは Step1 の詳細を述べていく．

トレーサビリティを測定する際，設計モデルとソースコードの対応付けが困難となる．与えられる設計モデルは，実装にそのまま適用できる詳細なモデルから，主要要素のみを示す抽象的なモデルまで様々である [5]．特に抽象度の高い設計モデルから実装する時は，そのモデルに記載されているクラスをそのまま実装せず，それを細分化して実装することがある．この場合，設計モデルとソースコードを比べて，単純に要素の名称が一致するもの同士を比較するだけでは適切な結果が得られない．このように，設計モデルの抽象度の違いがトレーサビリティの測定に影響を及ぼすため，対応付けのルールを工夫する必要がある．

本節では上記の問題を克服するため，クラスの細分化の傾向を考察した結果と，その結果をもとに制定した両者の対応付けルールを示す．なお細分化の傾向を示すにあたり，早稲田大学鷲崎研究室が 2009 年に ET ロボコンに参加した時に作成した設計モデルとソースコード（以後，このデータを”ロボコン 2009”と表現する）を，その実例として示す．

まずは細分化の傾向についてである．

1. 類似したクラス名の出現
   設計モデルに存在するクラスを機能分割し，類似した名前のクラスに細分化するケースが存在する．具体的にロボコン 2009 では，図 3 のように CourseDriving クラスが類似した名称の 3 つのクラスに細分化されている．
2. 汎化関係にあるクラスの細分化
   設計モデルで記述されている汎化関係の間に新たなクラスを追加し，汎化関係を階層的にするようにクラスを細分化するケースが存在する．具体的にロボコン 2009 では，図 4 のように Driver クラスを細分化することで汎化関係を階層的にしている．

上に示した細分化の傾向から，設計モデルとソースコード間のクラスの対応付け手順を定義した．手順の詳細を以下に示す．

Step1　名称が完全一致するクラスの対応付け

*

Goal	Question	Metrics
S-G. システム視点での設計モデルとソースコードのトレーサビリティ	S-Q1. 設計モデルを構成するクラスやクラス間の関係はソースコードに対応付けられるか	S-M1. 設計モデルに記述されているクラスのうち，ソースコードに対応付けられるものの割合
		S-M2. 設計モデルとソースコードで置かれている名前空間が一致するクラスの割合
		S-M3. 設計モデルに記述されているクラス間関係のうち，ソースコードに対応付けられるものの割合
	S-Q2. ソースコードで実装されているクラスやクラス間の関係は抜けなく設計モデルで記述されているか	S-M4. 設計モデルに対応付けることができるソースコード中のクラスの割合
		S-M2. 設計モデルとソースコードで置かれている名前空間が一致するクラスの割合
		S-M5. 設計モデルに対応付けることができるソースコード中のクラス間関係の割合

表 1　S-G に対する GQM パラダイム

設計モデルとソースコードのクラスを比較して名前が完全一致するクラスの組が存在する場合，両者を対応付ける．

Step2　他クラスとの関係が一致するクラスの対応付け

設計モデルとソースコードの情報を比較し，関連・汎化・依存関係が完全に一致するクラスが存在する場合，これらのクラスを対応付ける．

Step3　名称が部分一致するクラスの対応付け

「類似したクラス名の出現」の法則から，設計モデルとソースコードを比較して名前が部分一致するクラスの組が存在する場合，両者を対応づける．ただし，この操作により 5 つ以上のソースコード中のクラスと対応付く設計モデルのクラスは Step3 の対応付け操作の対象から外す．これは，そのクラス名がシステムを通じ汎用的に用いられていると考えられるためである．

Step4　汎化関係に関係するクラスの対応付け

「汎化関係にあるクラスの細分化」の法則から，対応付けされていないソースコード中のクラスに対し，既に対応付けされたスーパークラスとサブクラスが存在する場合，このクラスをスーパークラスの対応付け先と対応付ける．

2.3　トレーサビリティ測定

続いて，プロセスにおける Step2 の詳細を述べる．

一口にトレーサビリティを測定するといっても，クラスの対応具合から属性の型の一致具合まで，それには様々な測定基準がある．そのため，これらを単一のメトリクスで表現しようとすると複雑になり，利用者にとって理解しにくいものとなってしまう．

この問題を解決するために，本手法ではトレーサビリティを測定するのに GQM 法を利用した．GQM 法 (The Goal Question Metric Approach) [6] とは，Goal に設定した内容を測定するメトリクスを Question，Metrics の順に，トップダウンに決定する手法であり，測定したい対象が持つ複数の側面を統括して扱うのに適している．このため GQM 法は，保守性などソフトウェアの品質を測定する研究において，その多面性を取り扱うために導入された例がある [7] [8]．

ここで，トレーサビリティの測定基準には“ 構成クラスが対応付いているか ”といったシステム全体を測定対象としたものと，“ 属性・メソッドが対応付いているか ”といった一つ一つのクラスの内容を測定対象としたものがある．そこで測定対象に応じて以下の 2 つの Goal を設定し，トレーサビリティの度合を導出することにした．

1. システム視点での設計モデルとソースコードのトレーサビリティ (S-G)
   S-G に対する GQM パラダイムを表 1 に示す．当パラダイムの Question の設

*

Goal	Question	Metrics
C-G. クラス視点での設計モデルとソースコードのトレーサビリティ	C-Q1. 設計モデル中に記述されているクラスの内容やクラス間関係はソースコード中に反映されているか	C-M1. 設計モデルに記述されている属性，メソッドのうち，ソースコードに対応付けられるものの割合
		C-M2.（対応付けられる属性・メソッドのうち）可視性・型が一致する属性，可視性・返り値の型・引数が一致するメソッドの割合
		C-M3. 設計モデルに記述されているクラス間関係のうち，ソースコードに対応付けられるものの割合
	C-Q2. ソースコード中に実装されているクラスの内容やクラス間関係は設計モデルに記述されているか	C-M4. ソースコード中に実装されている属性，メソッドのうち，設計モデルに対応付けられるものの割合
		C-M5.（対応付けられる属性・メソッドのうち）可視性・型が一致する属性，可視性・返り値の型・引数が一致するメソッドの割合
		C-M6. ソースコード中に実装されているクラス間関係のうち，設計モデルに対応付けられるものの割合

表 2　C-G に対する GQM パラダイム

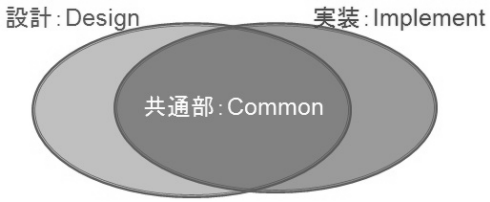

図 5　Question 設定のイメージ

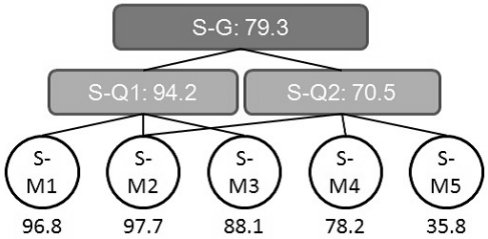

図 6　S-G,S-Q1,S-Q2 の導出例

定は，本手法で扱う要素が設計に存在する要素から成る Design 領域，実装に存在する要素から成る Implement 領域，さらに双方に存在する要素から成る Common 領域に属すると考えたとき（図 5），Q1 は Common ／ Design，Q2 は Common ／ Implement となるようにした．
この GQM パラダイムに対して，Question の得点をその Question に対する Metrics 値の平均値，Goal の得点を全ての Metrics 値の平均値と設定した．上記の Goal と Question の得点例を図式化したものを図 6 に示す．なお，全ての得点はパーセント表示されたものである．

2. クラス視点での設計モデルとソースコードのトレーサビリティ(C-G)
C-G に対する GQM パラダイムを表 2 に示す．これもシステム視点におけるトレーサビリティと同様の方針で Question を設定した．得点の算出に関しても C-G と同様に，Question の得点はその Question に対する Metrics の値の平均値，Goal の得点は全ての Metrics の得点の平均値と設定している．

2.4　トレーサビリティモデルの作成

最後に，プロセスにおける Step3 の詳細を述べる．

本手法では，トレーサビリティの測定のほかに，設計モデルとソースコードの相違箇所を色分けして指摘するモデル（トレーサビリティモデル）を提示する．設計モデルとソースコード間の相違は，以下の 3 つに大別して考えることができる．

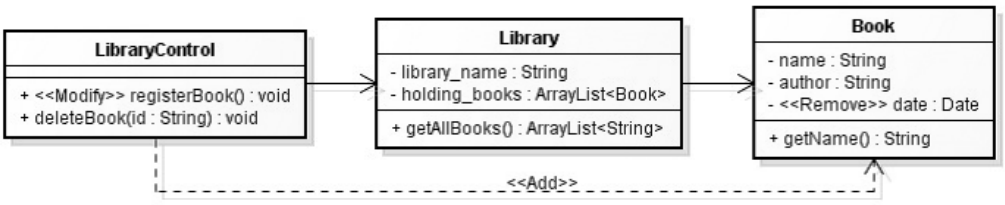

図 7　トレーサビリティモデルの出力例

Lv.	S-Q1	S-Q2	トレーサビリティの状態
3	高	高	設計モデルとソースコードの内容がほぼ一致している
2	高	低	ソースの修正が設計にあまり反映されていない
1	低	—	設計不在

表 3　システム視点でのトレーサビリティの得点の利用

- 除去 (remove)……設計モデルにある要素がソースコード中で実装されていない
- 追加 (add)……設計モデルにない要素がソースコード中で実装されている
- 修正 (modify)……設計モデルにある要素がソースコード中で実装されているが，その内容が変更されている

トレーサビリティモデルではこれら 3 つの分類を色分けしたり，ステレオタイプを付加することにより相違箇所を区別し，利用者に提示する．トレーサビリティモデルの出力例を図 7 に示す．

3　本手法の利用

本章では，第 2 章の成果物から両者のトレーサビリティの状態を判定し，ソフトウェアの保守性向上に向けた対応手順を提示する．なお，本節の内容は SESSAME WG2 の書籍 [9] [10] を参考にして考察したものである．

3.1　システム視点でのトレーサビリティの得点の利用

システム視点のトレーサビリティにおける Question から，トレーサビリティの全体的な傾向を判定することができる．具体的には，S-Q1 と S-Q2 の値の高低に応じてトレーサビリティの取れ具合を以下の 3 レベルに大別する（表 3）．

Lv.3 は S-Q1 と S-Q2 の得点の値がどちらも高い時であり，設計モデルの内容がソースコードに，またソースコードの内容が設計モデルに反映されているので，両者の対応は良く取れているということとなる．

Lv.2 は S-Q1 の得点は高いが，S-Q2 の得点が低い場合である．これは，設計モデルを参照して実装を行っているが，ソースコードに追加された実装情報が設計モデルのほうに反映されていないことを表している．これより Lv.2 は，ソースコードの修正が設計モデルにあまり反映されていない状態であると考えられる．

Lv.1 は S-Q1 の得点が低い場合である．S-Q1 の得点が低いのは，設計モデルに記述されている情報がソースコードに実装されていないことを表している．これより Lv.1 は，設計モデルに記述されている情報が実装に生かされていない，すなわち対象システムは設計不在で実装されていると考えられる．

3.2　クラス視点でのトレーサビリティの得点の利用

続いて，クラス視点でのトレーサビリティの得点の利用について考察する．この値が低いクラスは，クラス中で定義されている属性やメソッドに相違が生じていた

り，他クラスとの関係に想定外のものが追加されていることを示している．このような問題は，直接的には設計モデルやソースコードの修正漏れが原因で生じる．しかしさらに掘り下げて考察すると，要求の変更・ソフトウェアの機能性の向上・実装の困難など，設計段階では考慮されていなかった制約の発生が，修正の必要を生じさせると考えることができる．このような制約が発生すると，設計情報とは異なる情報の付加や，実装構造の複雑化を誘発する．

以上の内容から，トレーサビリティの得点が低いクラスは修正がないがしろになっているだけではなく，その内部構造が複雑になっている可能性がある．そのため，得点の低いクラスの中身を確認し構造を見直すことによって，クラス視点のトレーサビリティの得点を保守性の向上に利用することができる．

3.3 実際の環境への適用

本項では以上の考察を踏まえ，設計モデルとソースコードを対象に保守性向上を指向するプロセスを提案する．提案するプロセス（以後，これを改善プロセスと呼ぶ）は以下の 4 つから成る．

Step1 測定値からトレーサビリティの状況を認識
3.1 で述べた方法により，現状での設計モデルとソースコードのトレーサビリティの傾向を把握する．

Step2 Q1(S-Q1, C-Q1) の得点を改善する
最初に Q1 の得点を改善させる理由は，Q1 の得点が,「設計情報をもとにソフトウェアを実装しているかどうか」という根本を判定する指標となっているからである．具体的には Q1 の得点が低いクラスから順番に，トレーサビリティモデルにおける 3 種の相違のうち「除去」・「修正」をなくすようにする．

Step3 改善後の設計モデルとソースコードのトレーサビリティを再測定する
Step2 での改善を反映させるために，トレーサビリティを再び測定し，トレーサビリティモデルを更新する．

Step4 Q2(S-Q2, C-Q2) の得点を改善する
Step3 の結果，トレーサビリティの得点は向上し，トレーサビリティモデルで表示される相違はほとんどが「追加」の相違となっているはずである．Step4 では残された追加の相違をなくすようにモデルやソースコードの修正，また必要に応じてリファクタリングを実施して Q2 の得点を改善，全体のトレーサビリティを向上させる．

ただし，Step1 でトレーサビリティレベルが Lv.1 であると判定した場合，修正を行うのに多大なコストがかかることから，Step2 に進まずにリバース設計して得られたモデルを利用して保守する方針をとるべきである．

以上のプロセスを実際のサンプルに適用した結果を第 4 章で記述する．

4 ケーススタディ

本章では，本手法の有用性を検証するために，２つのサンプルに対して第 3 章で述べた改善プロセスを適用した結果を示す．１つ目のサンプルは，早稲田大学情報理工学科が 2008 年度の講義にて使用した，簡易航空便予約システムである．このサンプルのソースコードは，クラス数が 12，LOC の合計が 340 の規模である．２つ目のサンプルは，早稲田大学鷲崎研究室が 2010 年度に ET ロボコンに参加したときに作成した設計モデルとソースコードの組である．このサンプルのソースコードは，クラス数が 31，LOC の合計が 1802 の規模である．なお設計と実装の修正は，本サンプルがリアルタイム性をあまり問うていない点から，実行速度よりも今後の保守のしやすさを重視し，共通化できる部分はできる限り抽出してまとめるという方針を取ることとした．

	S-G	S-Q1	S-Q2	S-M1	S-M2	S-M3	S-M4	S-M5
改善前	76.1	87.9	72.2	90.9	100.0	72.7	84.6	32.0
改善後	97.3	100.0	95.5	100.0	100.0	100.0	100.0	86.4

表 4　システム視点でのトレーサビリティの得点（航空便予約システム）

C-G	C-Q1	C-Q2	C-M1	C-M2	C-M3	C-M4	C-M5	C-M6
77.6	100.0	62.7	100.0	100.0	100.0	71.4	100.0	16.7

表 5　クラス視点でのトレーサビリティの得点（ManagementControl・Q1 修正後）

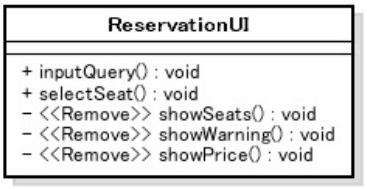

図 8　ReservationUI クラスのメソッド除去

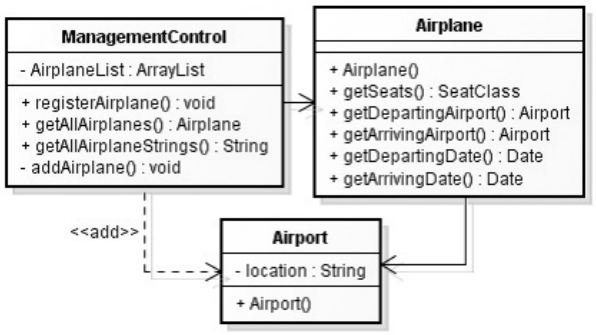

図 9　ManagementControl クラスの依存追加

4.1　航空便予約システムの事例

ここでは，１つ目のサンプル（簡易航空便予約システム）に対して，改善プロセスを適用した結果を示す.

Step1　測定値からトレーサビリティの状況を認識

サンプルに対してトレーサビリティを測定した結果を表 4 に示す．この測定結果（改善前）を判定すると，システム視点でのトレーサビリティにおいて Q1 の得点が 87 点と高いため，サンプルはきちんと設計をもとに実装されていることが分かる．一方 Q2 の得点は 72 点である．この値は必ずしも悪いものではないが，Q1 の得点よりも比較的低くなっている.

以上，Question の得点を 3.1 のトレーサビリティレベルにあてはめると Lv.2 となり，ソースコードの修正が設計モデルにあまり反映されていない状態であると認識できる.

Step2　Q1 の得点を改善する

Q1 の得点を向上させるには，検出された「除去」・「修正」の相違をなくすようにモデルやソースを修正すればよい.

例えば，出力されたトレーサビリティモデルを参照すると，ReservationUI クラスでは３つの private メソッドが除去されている（図 8）ことが分かる．上記の相違への原因を精査したところ,「除去」された private メソッドの内容が public メソッド中に直接記述されていることが判明したため，これらの内容をメソッド抽出することで，モデルの内容と対応付けた.

同様にほかの要素に対しても「除去」・「修正」の相違をなくすように修正を行うことにより，Q1 の得点を改善させた.

Step3　改善後の設計モデルとソースコードのトレーサビリティを再測定する

Step2 終了後，設計モデルとソースコードのトレーサビリティを再評価し，トレーサビリティモデルを更新した.

S-G	S-Q1	S-Q2	S-M1	S-M2	S-M3	S-M4	S-M5
40.2	40.3	41.7	69.6	45.5	5.9	54.8	25.0

表 6　システム視点でのトレーサビリティの得点（ロボコン 2010）

Step4　Q2 の得点を改善する

Q2 の得点を向上させるために，検出された「追加」の相違をなくすように修正を行う．

Step3 の再評価の結果，一番トレーサビリティの得点が低くなったのは ManagementControl クラス (MC) であった．MC のクラス視点でのトレーサビリティの得点を表 5 に示す．この得点の低さから MC に関連する「追加」の相違に着目すると，トレーサビリティモデル中に図 9 のような依存関係が追加されていることが確認された．ここで，MC はすでに Airplane クラスを通じて Airport クラスと繋がっているため，この依存関係を除去するようにリファクタリングを実施した．このように依存関係を除去することにより，トレーサビリティを向上させると同時に，構造を簡潔にすることができる．同様にほかの要素に対しても内容の修正やリファクタリングをすることによって，Q2 の得点を改善させた．

改善プロセス実施後のトレーサビリティの得点を示したのが表 4 の改善後の列である．改善前の得点と比べてみると，全ての項目において得点が上昇し，トレーサビリティが向上していることが分かる．

このように改善プロセスを実施することにより，設計モデルとソースコード間のトレーサビリティを向上させたほか，リファクタリングで依存関係を削減し，保守性を向上させることができた．

4.2　ET ロボコン 2010 の事例

ここでは，２つ目のサンプル（ロボコン 2010）に対して，改善プロセスを適用した結果を示す．

Step1　測定値からトレーサビリティの状況を認識

このサンプルに対してトレーサビリティを測定した結果を表 6 に示す．この測定結果から，S-Q1 の得点が 40.3 点と低いため，設計モデルの要素がソースコードにうまく反映されていないことが分かる．これを 3.1 のトレーサビリティレベルにあてはめると Lv.1 となり，もはや設計モデルはソースコードの構造を反映していないため，リバース設計で得られたモデルを今後の保守作業に利用すべきであると考えられる．

このようにロボコン 2010 の事例では，測定値からトレーサビリティの状況を判定することで，今後の保守作業の方針を提起することができた．

5　関連研究

Murphy らは，ユーザが定義したモジュール構造をソースコードに対応させ，相違箇所をグラフィカルなモデル (Reflexion Model) として示す手法を提案している [11]．Murphy らの提案手法は本手法におけるトレーサビリティモデルの提示に類似したものとなっているが，モジュール間関係の相違のみを検出し，クラスの属性やメソッドの相違を検出しない点で本手法と異なっている．また，モジュール構造とソースコード間の対応付けデータを別途用意しなければならないため手動操作が必須となり，自動で解析する本手法と比べると多くの時間がかかる．

トレーサビリティに関する研究に，ある成果物へ与えた修正が他の成果物にどのような影響を及ぼすかを解析する impact analysis がある．例えば，Briand らはトレーサビリティと修正の観念をモデル化し，抽象度の異なる UML クラス図の組を

対象に impact analysis をする手法を提案している [12] ほか，Hammad らはソースコードの変更が設計モデル (UML クラス図) にどのように影響を与えるかを，自動的に判別する手法を提案している [13]．impact analysis の研究では，成果物の変化を捉えるためにバージョンが異なる複数の成果物を用意する必要があるが，本手法では一組の設計モデルとソースコードを用意すれば手法を適用することができるため，導入が容易である．

6 おわりに

本研究では，設計モデルとソースコードの情報を抽出し，両者の間のトレーサビリティを解析する手法を提案した．利用者に設計と実装のトレーサビリティの度合を定量的に示すことで，設計モデルとソースコードの対応具合や傾向を客観的に示すことができるほか，両者の相違箇所を示すことにより，情報の同期支援やリファクタリング候補モジュールの特定に役立てることが期待される．また，実際にこれらの成果物を利用した保守性向上への改善プロセスを提案した．

本研究の今後の展望として以下の事柄が考えられる．

1. 設計モデルとソースコードの対応付け手法の拡充
   現状提案している対応付け方法では完全に設計モデルの抽象度の違いを吸収するには至っていないので，ほかに追加すべき対応付け方法があるかどうかを検討する必要がある．
2. 相違状況から導き出されるリファクタリング手法の提案
   本手法では，トレーサビリティの得点から設計モデルとソースコードの修正やリファクタリングの方向性を示すことはできたが，具体的な実施方法を提案するまでに至っていない．よって今後の展望として，トレーサビリティの得点と相違箇所の情報からもう少し踏み込んだリファクタリングの提案ができると，より実用的な手法になると考えている．

参考文献

[1] IEEE, "IEEE Standard Glossary of Software Engineering Terminology", IEEE Press, Piscataway, 1990.

[2] Stefan Winkler and Jens von Pilgrim, "A survey of traceability in requirements engineering and model-driven development", , VOL.9, NO.4, pp529-565, 2009.

[3] G.Booch, I.Jacobson and J.Rumbaugh, "The Unified Modeling Language User Guide", 2nd Edition, Addison Wesley, 2005.

[4] Shari Lawrence Pfleeger and Joanne M. Atlee, "Software engineering: theory and practice", Prentice Hall, 2009.

[5] Alexander Egyed, "Automated abstraction of class diagrams", ACM Transactions on Software Engineering and Methodology, Vol.11, No.4, 2002.

[6] Victor R. Basili and David M. Weiss, "A Methodology for Collecting Valid Software Engineering Data", IEEE Transactions on Software Engineering, VOL.SE-10, NO.6, pp728-738, 1984.

[7] Cristiane S. Ramos, Kathia M. Oliveira and Nicolas Anquetil, "Legacy software evaluation model for outsourced maintainer", CSMR'04, pp.48-57, 2004.

[8] 鷲崎ほか, "プログラムソースコードのための実用的な品質評価枠組み", 情報処理学会論文誌, Vol.48, No.8, pp.2637-2650, 2007.

[9] SESSAME WG2, "組込みソフトウェア開発のためのオブジェクト指向モデリング", 翔泳社, 2006.

[10] SESSAME WG2, "組込みソフトウェア開発のためのリバースモデリング", 翔泳社, 2007.

[11] Gail C. Murphy, David Notkin and Kevin J. Sullivan, "Software Reflexion Models: Bridging the Gap between Design and Implementation", IEEE Transactons on Software Engineering, VOL.27, NO.4, pp.364-380, 2001.

[12] Lionel C. Briand, Yvan Labiche and Tao Yue, "Automated traceability analysis for UML model refinements", Information and Software Technology, VOL.51, No.2, pp.512-527, 2009.

[13] Maen Hammad, Micheal L. Collard and Jonathan I. Maletic, "Automatically Identifying Changes that Impact Code-to-Design Traceability", ICPC'09, pp.20-29, 2009.