概念モデリングに基づく O/R マッピング手法に関する研究

村上 直

システム情報工学研究科
筑波大学

２０１４年　３月

謝辞

本研究を進めるにあたり，筑波大学大学院システム情報工学研究科 北川博之教授には，熱心かつ懇切丁寧なご指導を賜り，数々のご助言を頂きました．心より御礼を申し上げます．また，天笠俊之准教授には数々のご助言と議論を頂き，研究の指針を与えて頂きました．厚く御礼を申し上げます．

本論文の審査過程において，筑波大学大学院システム情報工学研究科 田中二郎教授，加藤和彦教授，前田敦司准教授には，適切なご助言を頂きました．謹んで御礼申し上げます．

筑波大学大学院システム情報工学研究科 北川研究室の皆様に御礼申し上げます．特に，川島英之講師，早瀬康裕助教，駒水孝裕氏には，ゼミの議論などで貴重なアドバイスを頂きました．

東京農工大学大学院 並木美太郎教授には，2006 年度下期 IPA 未踏ソフトウェア創造事業でご指導を賜り，本研究の成果の一部となりました．御礼申し上げます．横浜国立大学大学院 倉光君郎准教授には，折に触れ研究に関する示唆を頂きました．御礼申し上げます．

研究を進めるにあたり，様々な面でサポートを頂きました，高エネルギー加速器研究機構計算科学センターの皆様，特にセキュリティグループとネットワークグループの皆様に感謝致します．とりわけ金子敏明教授（センター長），

湯浅富久子准教授，鈴木聡准教授，野崎光昭教授（元センター長），川端節彌名誉教授（元センター長），渡瀬芳行名誉教授（元センター長），柴田章博研究機関講師には，研究を進めるための環境づくりにご尽力頂いたり，研究に関する示唆を頂いたり，様々な面でサポートを頂きました．また，本研究の実践の場となった DMZ User’s Portal に関わる皆様に感謝致します．利用者の皆様やセキュリティ管理部会の委員の皆様のご理解，ご意見により，長年にわたる運用を続けられました．また，J-PARC センター情報システムセクションの皆様，特に石川弘之氏と舘明宏氏には，DMZ User’s Portal の J-PARC への展開にあたり，テストなどで多大なる貢献を頂きました．

最後に，生活面や精神面で支えとなった，妻あゆみと家族に感謝します．

概要

ウェブアプリケーションやビジネス応用アプリケーションなどの，データベースの利用が不可欠なアプリケーション（DB アプリケーション）が普及している．DB アプリケーションでは，オブジェクト指向技術により設計や開発を行い，データベースとして関係データベース（RDB）を用いるケースが多いが，インピーダンスミスマッチ問題が知られており，開発の負担は大きくなる．そこで，オブジェクト関係マッピング（ORM）フレームワークが広く用いられている．ORM フレームワークは，永続化対象となるメモリ上のデータをオブジェクト指向言語で扱うための（a）永続化クラス，これを RDB 上で永続化するための（b）関係スキーマ，および，永続化クラスから RDB へクエリし応答を受けとる（c）RDB アクセスコードの三点について，効率的な設計や開発に貢献する．ORM フレームワークは二つのグループに大別できる．(1) 永続化クラスとこれを扱う関係テーブルについて，一対一を基本とした単純な対応関係（以下，単純な対応関係とよぶ）を簡易に実現できるフレームワーク，(2) 永続化クラスと関係テーブルについて，単純ではない対応関係（以下，複雑な対応関係とよぶ）を実現できるフレームワーク．

DB アプリケーションの開発方針としては，反復型開発やアジャイル開発のように，小規模なプロトタイプの開発と機能拡張を繰り返す方式がとられることが多い．ここで複雑な対応関係を実現できる ORM フレームワークを用いると，詳細な設計を得にくい開発初期の段階から開発者が永続化クラスとテー

ブルの詳細な対応関係を記述する必要が生じ，また，機能拡張の際にこの詳細な対応関係を修正する必要が頻繁に出てくるため，開発の負担が大きい．一方で単純な対応関係を簡易に実現できる ORM フレームワークを用いると，開発が進んで永続化クラスとテーブルの対応関係が複雑になった場合の対応が難しい．このように既存の ORM フレームワークは，単純な対応関係の簡易な実現と複雑な対応関係の実現についてサポートの両立が十分ではなく，単純な対応関係から複雑な対応関係へ移行する際の開発の負担もまた，大きくなる．

このような問題の克服のために，本研究では概念モデリングに基づく ORM 手法として，ORM フレームワーク DBPowder を提案する．本研究の特徴は，単純な対応関係の簡易な実現と複雑な対応関係の実現についてサポートを両立できる手法の提案と，この手法が単純な対応関係から複雑な対応関係に移行する開発を効率的にサポートできる点にある．詳細な設計を得にくい開発初期には，開発者は Extended Entity-Relationship (EER) モデルを記述して設計開発を進める．機能拡張の際には，開発者は EER モデル上の実体の走査経路と利用方法を指定した *ObjectView* を追加することで，アプリケーションロジックに対応した複雑な対応関係を実現する．EER モデルと連携させて ObjectView を併用することで，単純な対応関係の簡易な実現と複雑な対応関係の実現について，サポートを両立できる．ここで提案する ObjectView は，EER 概念モデルを基として，対象とする実体や関連を指定することで，追加的または修正的に複雑な対応関係をもつ永続化クラスを生成する手法であり，単純な対応関係から複雑な対応関係に移行する開発を効率的にサポートできる．

また，本研究で提案する ORM フレームワーク DBPowder が妥当な性能で実現可能であることを，DBPowder ORM システムの提案とそのプロトタイプ実装により示す.

提案手法 DBPowder の評価として，記述力の評価と開発に要する負担の評

価を実施する．記述力の評価では，単純な対応関係を簡易に実現できる Ruby on Rails（RoR）と複雑な対応関係を実現できる Hibernate を比較対象とする．DBPowder による ORM を記述するための言語 DBPowder-mdl の言語定義と Hibernate の ORM 記述を行う .hbm XML ファイルの文書型定義（DTD）を比較し，その結果を RoR の機能と照合することで，DBPowder の記述力を比較検証する．開発に要する負担の評価では，ORM の実現に要する開発の負担について二種類の比較評価を行う．一つは，DBPowder，Ruby on Rails（RoR），および Hibernate について，各々のフレームワークが ORM を実現する際に必要とするコードを用意し，各々のコードに含まれる単語数を数えることで，開発で要求される記述量を比較する．これを，単純な対応関係や複雑な対応関係について行う．もう一つは，DBPowder と Hibernate について，単純な対応関係のみで構成される ORM に対して複雑な対応関係を追加する際に，必要とされる編集回数や単語数を数え，比較する．

また DBPowder ORM システムの評価として，OO7 ベンチマークによる性能評価を，DBPowder，Hibernate，および RoR について実施する．

目 次

謝辞 **i**

概要 **iii**

図目次 **x**

表目次 **xiii**

第 1 章　序論 **1**

第 2 章　関連研究 **6**

2.1 オブジェクト指向言語とデータベースの連携方式 6

2.2 自動生成されるデータモデルに基づいた分類 9

2.2.1 Bottom-up マッピングアプローチ 12

2.2.2 Top-down マッピングアプローチ 14

2.2.3 Meet-in-the-middle マッピングアプローチ 15

2.2.4 Metadata マッピングアプローチ 16

2.3 永続化クラスと関係スキーマの対応関係に基づいた分類 18

2.3.1 本研究における単純な対応関係と複雑な対応関係の定義 18
2.3.2 単純な対応関係を簡易に実現できるアプローチ 22
2.3.3 複雑な対応関係を実現できるアプローチ 24
2.4 本研究の位置づけ 29

第 3 章 提案手法 DBPowder で用いるデータモデル 31

3.1 DBPowder で用いるオブジェクトモデル 31
3.2 DBPowder で用いる関係モデル 34
3.3 DBPowder で用いる EER モデル 35

第 4 章 DBPowder：概念モデリングに基づく O/R マッピング手法 37

4.1 はじめに 37
4.2 EER モデルによる単純な対応関係を簡易に実現できる ORM 手法 41
4.2.1 EER モデルを用いた ORM の手順 41
4.2.2 DBPowder-mdl による単純な対応関係の記述（導入） . . 44
4.2.3 単純な対応関係の実現例（user – register – host スキーマ） 45
4.3 ObjectView による複雑な対応関係を実現できる ORM 手法 . . . 47
4.3.1 ObjectView の導入 47
4.3.2 ObjectView の定義と ORM の内容 48
4.3.3 DBPowder-mdl による複雑な対応関係の実現（導入） . . 52
4.3.4 複雑な対応関係の実現例（user – register – host スキーマ） 53
4.3.5 複雑な対応関係の実現例（6.1 節 KEKapp の一部） . . . 57

4.4 記述言語 DBPowder-mdl の定義 60
4.4.1 形式的な定義 60
4.4.2 EER 宣言部の構文 61
4.4.3 EER 宣言部の例 64
4.4.4 ObjectView 宣言部の構文 65
4.4.5 ObjectView 宣言部の例 68
4.4.6 CoC の活用：実体名表記による名称の決定方法 69
4.4.7 CoC の活用：実体パラメータの決定方法 71
4.4.8 オブジェクトモデルの継承を関係モデルで扱う記述例と他の ORM フレームワークでの記述例 71
4.5 記述力の評価 74
4.5.1 評価の内容と方法 74
4.5.2 評価結果 75
4.6 開発に要する負担の評価：単語数を用いた記述量の比較 78
4.6.1 評価の内容と方法 78
4.6.2 評価結果 82
4.7 開発に要する負担の評価：複雑な対応関係に移行する開発で要した編集の回数や単語数の比較 85
4.7.1 評価の内容と方法 85
4.7.2 評価結果 86
4.8 まとめ 89

第 5 章　DBPowder ORM システム **92**

5.1 はじめに 92

5.2 ORM 機能をもつコードの生成機能 93

5.3 生成されたコードにより実現される ORM 機能 97

5.3.1 永続化に関する基本的な操作 97

5.3.2 永続化に関する内部的な処理 99

5.4 DBPowder ORM プロトタイプシステムの実装 102

5.5 性能評価 103

5.5.1 OO7 ベンチマークの導入 103

5.5.2 評価の内容と方法 106

5.5.3 評価結果 110

5.6 まとめ 112

第 6 章　応用事例 **114**

6.1 ウェブサイトの構築：サーバのセキュリティ管理ポータル . . . 114

6.2 構築済みウェブサイトへの追加開発 118

第 7 章　結論 **120**

参考文献 **124**

研究業績 **134**

図目次

2.1 単純な対応関係の例：標準の単純な対応関係 20

2.2 単純な対応関係の例：結合テーブルによる単純な対応関係 . . . 21

2.3 単純な対応関係の例：継承による単純な対応関係 23

3.1 DBPowder で用いるオブジェクトモデルの例 32

3.2 DBPowder で用いる関係モデルの例 34

3.3 DBPowder で用いる EER モデルの例 36

4.1 DBPowder による O/R マッピング（ORM）の概要 38

4.2 DBPowder における単純な対応関係の実現例（user – register – host スキーマ） 46

4.3 DBPowder-mdl の記述例：図 4.2 の EER モデル *eer* 46

4.4 DBPowder における複雑な対応関係の実現例（user – register – host スキーマ） 54

4.5 DBPowder-mdl の記述例：図 4.4 の ObjectView ov_1 と ov_2 54

4.6 永続化クラス `User` の統合例 56

4.7 DBPowder における複雑な対応関係の実現例（6.1 節 KEKapp の一部） 58

4.8 DBPowder-mdl の記述例：図 4.7 の ObjectView ov_1 と ov_2 58

4.9 DBPowder-mdl の BNF 記法による文法（EER 宣言部） 62

4.10 DBPowder-mdl の記述例：EER 宣言部 63

4.11 DBPowder-mdl の BNF 記法による文法（ObjectView 宣言部） . 66

4.12 DBPowder-mdl の記述例：ObjectView 宣言部 67

4.13 DBPowder-mdl の記述例：オブジェクトモデルの継承を関係モデルで扱う例 72

4.14 図 4.13 の Hibernate や Ruby on Rails での扱い 73

4.15 CaveatEmptor（オブジェクトモデル：クラス図） 80

4.16 CaveatEmptor（関係モデル：IDEF1x 表記） 80

4.17 OO7 ベンチマーク（ER モデル）と，ObjectView の適用 81

4.18 DBPowder-mdl の記述例：図 4.17 の ObjectView 81

4.19 開発に要する負担の評価結果：単語数を用いた記述量の比較 . . 83

5.1 DBPowder ORM システムの全体構成：ORM 機能をもつコードの生成と，生成されたコードが構成する ORM モジュール . . . 94

5.2 アプリケーションコード（acode）の例：応用開発者による永続化クラスのコード（gcode_pc）の利用 97

5.3 ActiveRecord クラスの導入による，永続化対象のオブジェクトと RDB 上のタプル値の一貫性管理 100

5.4 OO7 ベンチマークのデータ件数 104

5.5 性能評価の結果：OO7 ベンチマークで計測した，DBPowder と Hibernate の処理時間の比較 108

5.6 性能評価の結果：OO7 ベンチマークで計測した，DBPowder と Ruby on Rails (RoR) の処理時間の比較 109

6.1 DMZ User's Portal（KEKapp）のシステム全体図 115

6.2 DMZ User's Portal のページ（一部） 116

6.3 CRUD 機能を実現するウェブページ 117

表目次

2.1 自動生成されるデータモデルに基づいた ORM 手法の分類 . . . 10

2.2 データモデル設計変換における R，O，および ER モデルの要素の一般的な対応関係 11

2.3 表 2.1 と 表 2.2 の関係，および各々の変換方法を議論した文献一覧 11

2.4 自動生成されるデータモデルと O-R 間の対応関係に基づいた，ORM 手法の分類 19

4.1 eer, ov_n, rs, spc, cpc_n の主な対応関係 42

4.2 DBPowder-mdl における，設定より規約（CoC）による名称の決定方法 70

4.3 DBPowder-mdl の実体で指定できるパラメータと，そのデフォルト値 70

4.4 記述力の評価結果 (1)：単純な対応関係と複雑な対応関係 76

4.5 記述力の評価結果 (2)：表 4.4 以外の結果 77

4.6 開発に要する負担の評価：評価対象としたアプリケーション . . 79

4.7 開発に要する負担の評価結果：複雑な対応関係に移行する開発で要した，編集の回数および編集した単語数の比較 87

第1章　序論

近年，ウェブアプリケーションやビジネス応用アプリケーションなどの，データベースの利用が不可欠なアプリケーション（DB アプリケーション）が普及している．DB アプリケーションでは，オブジェクト指向技術により設計や開発を行うことが一般的である．ここで，永続化対象となるメモリ上のデータは，オブジェクトとして扱われる．一方でデータベースとしては関係データベース（RDB）[25] が広く用いられる．ここで，永続化対象のオブジェクトは RDB から検索され，また RDB に格納される．

DB アプリケーションの設計や開発に用いられるオブジェクト指向言語からデータの永続化に用いられる関係モデルを扱う際には，開発の負担が大きくなる．この問題は，インピーダンスミスマッチ問題 [28, 62, 16, 65, 41, 26, 27, 9] として知られている．その原因として，たとえば文献 [28, 9] では，データ利用の際にアプリケーションが求める論理スキーマと実際のデータソースがもつ物理スキーマが同一ではないことを指摘しており，また文献 [27] では，オブジェクト指向言語における命令型プログラム上から，これとは意味論的な基盤の異なる関係データベースの宣言型問い合わせが行われることを指摘している．DB アプリケーションに要求される仕様が複雑になる場合，永続化対象となるクラスの種類が増えるにしたがってこの問題は深刻化する．したがって，DB アプリケーションの設計や開発においては，予めこの問題を考慮に入れて対策を打っておくことが重要である．

このような問題に対処するため，オブジェクト関係マッピング（ORM）フレームワークが広く用いられる．ORM フレームワークは，永続化対象となるメモリ上のデータをオブジェクト指向言語で扱うための（a）永続化クラス，これを RDB 上で永続化するための（b）関係スキーマ，および，永続化クラスから RDB へクエリし応答を受けとる（c）RDB アクセスコードの三点について，効率的な設計や開発に貢献する． 既存の ORM フレームワークは以下の二つのグループに大別できる.（1）永続化クラスとこれを扱う関係テーブルについて，一対一を基本とした単純な対応関係（以下，単純な対応関係とよぶ）を簡易に実現できるフレームワーク [40, 36, 20, 56, 95],（2）永続化クラスと関係テーブルについて，単純ではない対応関係（以下，複雑な対応関係とよぶ）を実現できるフレームワーク [49, 2, 14].

例として，ホストと管理者を管理するアプリケーションを考える．ホストと管理者の関係が多対多のとき，関係テーブル（以下，単にテーブル）としては，ホストと管理者のほか，二者の関係を管理するテーブル（結合テーブル）の合計三つが必要である．これに永続化クラスを対応させる方法は様々に考えられる．代表的には，三つのテーブルに対応した三つの永続化クラスを用意する方法と，結合テーブルに対応したクラスを用意せずにホストと管理者に多対多の関連を直接もたせる方法が，考えられる．但し後者の方法は，結合テーブルが登録日などの属性をもつ場合に，その属性の管理が問題となる．このようなケースを考えたとき，単純な対応関係を簡易に実現できるフレームワークは，三つのテーブルに対応した三つの永続化クラスを用意するような開発で，開発の負担を小さくできる．しかし，結合テーブルが属性をもつような場合に複雑な対応関係を設計することは難しい．一方で，複雑な対応関係を実現できるフレームワークでは，単純な対応関係を簡易には実現できない代わりに，結合テーブルが属性をもつような場合でも，アプリケーション内の場面ごとで用いられるロジック（以下，アプリケーションロジック）を踏まえて永続化クラス

とテーブルの様々な対応関係を指定できる．

DB アプリケーションの開発動向に眼を向けると，反復型開発やアジャイル開発 [63] のような迅速な開発への要求が高まっている．その一方で，仕様を固めるのに時間を要するケースが増えている．ウォーターフォール型の開発 [84] とは異なり，反復型開発やアジャイル開発では，はじめに小規模なプロトタイプを開発し，これを反復して機能拡張しつつ洗練させることで，少しずつ目的の複雑なアプリケーションに近づけていく．このような開発で ORM フレームワークを用いる場合，詳細な仕様や設計が固まらない状況で小規模なプロトタイプを開発していく開発の初期段階では，単純な対応関係をもつ永続化クラスとテーブルのマッピングが求められる．一方，開発が進むにつれて徐々に複雑な対応関係をもつマッピングが求められるようになる．複雑な対応関係をもつマッピングはアプリケーションロジックからの要求により生じることが多く，これに ORM フレームワークが応じることで，複雑な DB アプリケーションの開発の負担は軽減される．

ここで, (1) のような単純な対応関係を簡易に実現できるフレームワークは，使いやすい一方で，複雑な対応関係をうまく扱えない．したがって，開発が進み仕様が複雑になり，永続化クラスとテーブルの複雑な対応関係が必要になった場合に十分に対応できない．一方で（2）のような複雑な対応関係を実現できるフレームワークでは，詳細な設計を得にくい開発初期の段階から開発者が永続化クラスとテーブルの詳細な対応関係を記述する必要が生じ，また，機能拡張の際にこの詳細な対応関係を修正する必要が頻繁に出てくる．したがって，迅速な開発が強く求められる開発初期の段階から，開発の負担が大きくなる．

また，単純な対応関係により設計された永続化クラスや関係スキーマに対して詳細な設計が得られた場合には，しばしば単純な対応関係から複雑な対応関係に移行する必要が生じる．よって，このような開発の効率的なサポートが望

まれる．しかし既存の ORM フレームワークは，単純な対応関係の簡易な実現と複雑な対応関係の実現についてサポートの両立が十分ではなく，単純な対応関係から複雑な対応関係へ移行する開発の負担は大きい．

このような問題の克服のために，本研究では概念モデリングに基づく ORM 手法として，ORM フレームワーク DBPowder を提案する．本研究の特徴は，単純な対応関係の簡易な実現と複雑な対応関係の実現についてサポートを両立できる手法を提案した点と，この手法が単純な対応関係から複雑な対応関係に移行する開発を効率的にサポートできる点にある．詳細な設計を得にくい開発初期には，開発者は Extended Entity-Relationship (EER) モデル [94] を記述して設計開発を進める．DBPowder は EER モデルに基づき，単純な対応関係をもつ永続化クラスと関係スキーマを生成する．機能拡張の際には，開発者は EER モデル上の実体の走査経路と利用方法を有向グラフベースで指定した *ObjectView* を追加することで，アプリケーションロジックの各場面に応じた複雑な対応関係を実現する．DBPowder は，記述された ObjectView の内容に基づいて，永続化クラスの追加や修正を実施する．

このように DBPowder では，EER モデルと連携させて ObjectView を併用することで，単純な対応関係の迅速な開発をサポートするとともに，複雑な対応関係による永続化クラスを簡易に追加できる．ここで提案する ObjectView は，EER 概念モデルを基として，対象とする実体や関連を指定することで，追加的または修正的に複雑な対応関係をもつ永続化クラスを生成する手法であり，単純な対応関係から複雑な対応関係に移行する開発を効率的にサポートできる．

また，本研究で提案する ORM フレームワーク DBPowder が妥当な性能で実現可能であることを，DBPowder ORM システムの提案とそのプロトタイプ実装により示す．

提案手法 DBPowder の評価として，記述力の評価と開発に要する負担の評価を実施する．記述力の評価では，単純な対応関係を簡易に実現できる Ruby on Rails（RoR）と複雑な対応関係を実現できる Hibernate を比較対象とする．DBPowder による ORM を記述するための言語 DBPowder-mdl の言語定義と Hibernate の ORM 記述を行う .hbm XML ファイルの文書型定義（DTD）を比較し，その結果を RoR の機能と照合することで，DBPowder の記述力を比較検証する．開発に要する負担の評価では，ORM の実現に要する開発の負担について二種類の比較評価を行う．一つは，DBPowder，Ruby on Rails（RoR），および Hibernate について，各々のフレームワークが ORM を実現する際に必要とするコードを用意し，各々のコードに含まれる単語数を数えることで，開発で要求される記述量を比較する．これを，単純な対応関係や複雑な対応関係について行う．もう一つは，DBPowder と Hibernate について，単純な対応関係のみで構成される ORM に対して複雑な対応関係を追加する際に，必要とされる編集回数や単語数を数え，比較する．

また DBPowder ORM システムの評価として，OO7 ベンチマークによる性能評価を，DBPowder，Hibernate，および RoR について実施する．

本論文の構成は以下の通りである．第 2 章 で関連研究について述べ，第 3 章 で提案手法 DBPowder で用いるデータモデルについて述べる．第 4 章 で概念モデリングに基づく O/R マッピング（ORM）手法 DBPowder を提案する．第 5 章 では，提案手法を実現する DBPowder ORM システム について述べる．第 6 章 では本研究の応用事例を紹介し，第 7 章 でまとめを述べる．

第2章　関連研究

本研究の主題である O/R マッピング（ORM）は，オブジェクト指向言語と関係データベースの連携に関する問題であり，古くから議論されてきた [62, 16]. はじめに 2.1 節 では，Carey ら [16] によるオブジェクト指向言語とデータベースの連携方式に関する議論を踏まえつつ，現在の ORM をとりまく状況を概観する．次に 2.2 節 では，ORM 手法が自動生成するデータモデルに基づいて，関連研究を分類する．これに加え 2.3 節 では，ORM 手法が扱う永続化クラスと関係スキーマの対応関係に基づいて，関連研究を分類する．これらを踏まえて，2.4 節 では本研究の位置づけを示す.

2.1　オブジェクト指向言語とデータベースの連携方式

Carey らは文献 [16] で，オブジェクト指向言語とデータベースの連携方式について 1996 年までの状況を整理している．ここで示されたアプローチをまとめると，以下の四つのグループに大別できる.

- グループ 1) オブジェクト指向データベース（OODB）システムおよび，これに多大な影響を与えた永続化プログラミング言語
- グループ 2) オブジェクト関係データベースシステムおよび，これの源流となった関係データベースシステムの拡張アプローチ

- グループ 3) オブジェクト指向クライアントラッパー
- グループ 4) データベースシステムのツールキット

グループ 1) の OODB は現在でも利用されているが，CAD/CAM など特定用途に限られており，関係データベースと比べて市場規模は極めて小さい [58]．その理由として，文献 [16, 58] ともに，OODB の標準規格が広まらなかったことを挙げている．標準規格 ODMG3.0 [19] は，2013 年現在において広範に実装されているとは言えない．関係データベースにおいて標準規格 SQL92 [48] が広く受け入れられ，その後に発表された標準規格についても各ベンダが準拠のための努力を続けているのとは対照的と言える．

グループ 2) のオブジェクト関係データベースは，関係データベースを拡張してオブジェクト指向技術に対応したデータベースであり，関係データベースを基盤としない OODB とはアプローチを異にする．現在広く用いられている関係データベース管理システム（RDBMS）のうち，Oracle Database [78]，IBM DB2 [45]，Microsoft SQL Server [67]，PostgreSQL [81] などがこの拡張を施している．しかしこれらの拡張は，それぞれの RDBMS が独自に行っているのが現状であり [41]，オブジェクト関係データベースの標準規格を定めた SQL99 [47] の内容とは差異がある．

OODB やオブジェクト関係データベースによるアプローチの課題として，たとえば 文献 [65] で，標準化が進んでいない技術を用いるとデータ移行時のコストが高くつく点と，ビジネスロジックをデータベースにもたせるとシステムとしてのスケールアウトが難しくなる点を指摘している．グループ 3) のオブジェクト指向クライアントラッパーによるアプローチでは，通常のオブジェクト指向言語を用いて関係データベースシステムのラッパーを実装することで ORM を実現し，オブジェクト指向技術とデータベース技術を連携させる．

OODB やオブジェクト関係データベースと異なり，既存の言語やデータベースシステムそのものの拡張を伴わないアプローチである．こんにちの ORM の多くはこの方式をとっている．本研究はこのグループを対象としており，2.2 節 以降で詳説する．

グループ 4) のデータベースシステムのツールキットによるアプローチでは，データベース管理システムを拡張してアプリケーションの実装まで担当させる．Carey らは，この方式は 1996 年までに終焉を迎えたと述べている．その理由として，一点目としてはこれらのツールキットを使うには熟練が必要となること，二点目としてはツールキットとして柔軟性や完成度を上げるのが難しかった点を挙げている．しかし 2013 年現在，この流れを汲むアプローチが少なくとも二種類ある．

一つのアプローチは，開発対象をデータ集中型ウェブアプリケーション [37] に絞ったツールキットが挙げられる．このうちデータ管理に関係データベースを用いるアプローチとして，WebRatio [99]，Araneus [66]，Autoweb [79] を挙げることができる．いずれも ER モデル [23] をデータモデル設計に用い，ER モデルをもとにして関係スキーマとアプリケーションを生成する．このうち WebRatio ではモデリング言語 WebML [21, 20, 12] を用いており，グループ 3) の ORM としても機能している．このカテゴリでは，ほかに Ruby on Rails (RoR) [40] を挙げることができる．RoR の中心コンポーネントである ActiveRecord [36] は，グループ 3) の ORM である．データ集中型ウェブアプリケーションが成功を収めた理由としては，ツールキットとしての対象領域をウェブアプリケーションに特化したことが原因の一つと考えられる．WebML と ActiveRecord については，2.2 節以降で議論する．

データベースシステムのツールキットによるもう一つのアプローチはモデル駆動型アーキテクチャ（MDA） [73] である．モデル駆動型アーキテクチャで

は，はじめにプラットフォーム非依存モデル（PIM）を設計する．次に，PIM を PSM（プラットフォーム依存モデル）に自動変換する．PSM は，使用 OS や言語，データベースシステムなどのミドルウェアを特定したモデルである．最後に，PSM に必要に応じて実装を加えることで実行可能なコードを生成する．代表的な実現方式として，UML [72, 85] を PIM として用いる Executable UML [90] が挙げられるが，実践は難しいという指摘がある [86, 88]．一方で，モデル駆動型 [32, 89] という開発アプローチの概念そのものは，現在広く受け入れられている．

2.2 自動生成されるデータモデルに基づいた分類

本節では，ORM を実現するために開発者が与えるべき入力と，その入力に基づいた ORM 手法の処理を整理し，ORM 手法が自動生成するデータモデルに基づいた分類を行う．分類結果は 表 2.1 の通りである．以下，この分類に至る経緯を示す．

入力には，関係モデル（R），オブジェクトモデル（O），R や O 以外のデータモデル（C），R-O 間の対応をとるための補助情報（META）を考えうる．なお入力では，実装可能な関係スキーマ（*rs*）や永続化クラス（*pc*）を入力にとることもあるが，ORM 手法はこれらを，内部的には R や O に変換して扱っている．したがって入力としての *rs* や *pc* は，R や O として考える．また出力としては，関係スキーマ（*rs*），永続化クラス（*pc*），および R-O 間の対応をとる RDB アクセスコード（*map*）が要求される．これは，序論で与えた ORM の定義による．

出力として *rs*，*pc*，*map* を得るためには，その基となるデータモデル R と O が必要である．入力時のデータモデルが不足する場合は，ORM 手法は不足

したデータモデルを自動生成して補う必要がある．こうして，入力または自動生成で得られた R や O が，出力における *rs* や *pc* に変換される．また *map* を得るためには，この R-O 間の対応を実際にとる必要がある．

また整理するにあたり，以下の二つの条件を与える．

- ORM を成立させるには，入力に少なくとも一つ以上のデータモデルと補助情報 META を必要とする．但し入力に C と META がある場合は，C を補助情報 META の一部とみなすことができる．
- 入力にプログラムコードをとることはない．つまり，*map* を入力にとることはない．

これらの条件を踏まえて整理すると，入力と出力の組み合わせは 表 2.1 に示す四通りとなる．それぞれの組み合わせをマッピングアプローチ名として，bottom-up，top-down，meet-in-the-middle，metadata と名付ける．なおここで

表 2.1: 自動生成されるデータモデルに基づいた ORM 手法の分類：入力を開発者が与えると，ORM 手法が出力を自動生成する

マッピング アプローチ名	入力 (R, O, C, META)	自動生成される データモデル
Bottom-up	R, META	O
Top-down	O, META	R
Meet-in-the-middle	R, O, META	-
Metadata	C, META	R, O

■R-O 間の対応をとる
■変換
R→*rs*, O→*pc*
出力
rs, *pc*, *map*

R :関係モデル
O :オブジェクトモデル
C :R や O 以外のデータモデル
META :R-O 間の対応をとるための補助情報

rs :関係スキーマ
pc :永続化クラス
map :RDB アクセスコード．永続化クラスから RDB へクエリし応答を受けとることで，R-O 間の対応をとる

得られた分類は，文献 [7, 8] に示される典型的な ORM の分類や名称と一致している．

Fahrner ら [35] は，ER モデル，R，O，ネットワークモデル [38, 103, 74]，および階層モデル [38, 103, 60] について，データモデルの設計を変換する方式について調査している．このうち，R，O，および ER モデルに関する変換を実

表 2.2: データモデル設計変換における R，O，および ER モデルの要素の一般的な対応関係

入力や出力のモデル	入力や出力のモデル同士で対応する要素			
ER: ER モデル	実体	主キー	属性	関連
R: 関係モデル	テーブル	主キー	属性	外部キー，結合テーブル (*1,2)
O: オブジェクトモデル	クラス	識別子	属性	関連

(*1) 関連の連結度/多重度が多対多以外でかつ次数が 2 の場合は，結合テーブルを用いない方法が一般的である．即ち，連結度/多重度が 1 側の実体やクラスに対応したテーブルの主キーを，連結度/多重度が多側に対応したテーブルに加え，この加えたキーを外部キーとすることで，外部キー制約を与える．

(*2) 関連の連結度/多重度が多対多または次数が 3 以上の場合は，関連を結合テーブルに変換し，関連に関係づけられる実体やクラスを変換したテーブルへの外部キーを結合テーブルに加え，これを主キーとする．

表 2.3: 表 2.1 と 表 2.2 の関係，および各々の変換方法を議論した文献一覧

マッピングアプローチ名	入力	出力	文献
Bottom-up	R	O	[35, 71, 82] など
Top-down	O	R	[35, 10, 43] など
Metadata	ER	R	[35, 94, 103] など
		O	[35, 71, 34] など

Meet-in-the-middle については，該当なし．Metadata については，ER を入力としないものについては該当なし

施する際の，モデル内の要素の一般的な対応関係をまとめたものを，表 2.2 に示す．また，表 2.1 と 表 2.2 の関係，および各々の変換方法を議論した文献一覧を，表 2.3 に示す．

以下，2.2.1 節，2.2.2 節，2.2.3 節，2.2.4 節 で，それぞれのアプローチを示す．これらの複数をサポートする ORM 手法については，2.2 節 では代表的な機能を示すにとどめ，複数アプローチのサポート状況は 2.3 節 で改めて述べる．

2.2.1　Bottom-up マッピングアプローチ

表 2.1 に示すとおり bottom-up マッピングアプローチでは，開発者が関係モデル R と補助情報 META を入力として与え，ORM 手法がその入力に基づきオブジェクトモデル O を自動生成し，R-O 間の対応をとり，これらに基づいて関係スキーマ *rs*，永続化クラス *pc*，および RDB アクセスコード *map* を出力する手法をとる．

ActiveRecord [36] は，関係スキーマと構造の一致した永続化クラスを開発者が簡易に構築できることを特徴とするアプローチであり，開発現場で広く用いられている．ActiveRecord クラスは永続化クラスであり，担当するテーブルを一つもつ．このテーブル上の各々の属性について，ActiveRecord クラスは対応する属性をもつ．ActiveRecord クラスをインスタンス化したオブジェクトは，テーブル上のタプルに対応する．この手法の利点はその単純さにある．特に Ruby on Rails (RoR) [40] の実装では，クラス，テーブル，属性，外部キーなどの命名規則に強い制約を設けることで開発者による設定の手間を省く，設定より規約（CoC[1]） [33, 22] という考え方を推し進めることで，リソースの重複し

[1] クラス名は英単語の単数形，テーブル名はクラス名の複数形，主キーは代理キーとして id という名前にする，など

た指定を大幅に回避（DRY: Don’t Repeat Yourself [44]）でき，結果として開発に要する負担を軽減できる．但し，よく知られた例（多対多関連と継承）を除き，基本的には一つの ActiveRecord クラスは一つのテーブルと対応づけることしかできず，記述力には制約がある．また CoC が定める文法から逸脱したスキーマ設計を試みると，逸脱した設計が関連する他の箇所に波及し，開発に要する負担の軽減が得られなくなる．なお RoR では，Scaffold ツールにテーブル名とテーブルが保持する属性名を与えると，関係スキーマと ActiveRecord クラスを同時に簡易に生成できる．この際，CRUD（Create，Retrieval，Update，Delete）機能をもつウェブアプリケーションも同時に生成される．これをテンプレートとしてカスタマイズすることで，所望のウェブアプリケーションを構築できる．RoR は，ORM を用いたアジャイル開発の現場で広く用いられており，実用およびビジネス分野で大きな成功を収めている [87] .

PhpClick [52, 92] は，関係データベースを用いたウェブアプリケーションを開発者が GUI から簡単に開発できることを目指したアプローチであり，開発機能 GUI もウェブアプリケーションで提供される．開発者はウェブ画面にテーブル名や属性名を入力して関係モデルを設計する．続いてウェブ画面遷移フローにて各々のウェブ画面で扱いたいテーブルを割り当て，テーブルの使い方を定義する．開発者は PhpClick が生成したソースコードを編集することで，画面に割り当てられたテーブル上のデータ操作を実装できる．

Thiran らはラッパー指向の ORM を提案している [96, 95, 42] ．提案では，関係スキーマを出発点として，フレームワークが定めた八種類のスキーマ変換ルールを開発者が適宜適用することで，開発者が望む永続化クラスを取得できるとしている．ORM 実現のためのラッパーは，このルールを適宜適用したものとして定義できる．関係スキーマと永続化クラスの構造上の違いに着目できるのが利点である一方，対応関係が変換ルールに基づくものに限られる制約が

ある．たとえば，永続化クラスにおける継承の関係モデルでの対応や，永続化クラスとテーブルの個数が異なる場合の対応をサポートしない．なおデータモデルの記述方法として，関係スキーマと永続化クラスを共通な文法のデータモデルで記述できる Generic Entity-Relationship model（GER）を提案している．

2.2.2 Top-down マッピングアプローチ

表 2.1 に示すとおり top-down マッピングアプローチでは，開発者がオブジェクトモデル O と補助情報 META を入力として与え，ORM 手法がその入力に基づき関係モデル R を自動生成し，R-O 間の対応をとり，これらに基づいて関係スキーマ *rs*，永続化クラス *pc*，および RDB アクセスコード *map* を出力する手法をとる．

RoR では，2.2.1 節 で述べた ActiveRecord クラスの代わりに Datamapper クラス [36, 91] を使うことができる．開発者が Datamapper クラス上に属性と関連を定義し，関係スキーマの生成を指示すると，Datamapper クラスごとに対応するテーブルが生成される．ActiveRecord 同様，よく知られた例（多対多関連と継承）を除き，基本的には一つの Datamapper クラスは一つのテーブルと対応づけることしかできず，記述力には制約がある．

Lodhi ら [59] は，オブジェクトモデルの関連を関係モデルへ対応づける際に，多重度に関わらずテーブルを用いたほうが実データをもつテーブルから外部キーが取り除かれ，インピーダンスミスマッチが緩和されると述べている．そして性能評価を実施した結果，大きな劣化は見られなかったと結論づけている．但しこの評価は，最大レコード数が 14 万件程度となっている．一方で Philippi ら [80] は，関係モデルへの対応づけの方式が複数種類ある場合に，それぞれの得失を理解度，保守性，消費容量，性能の四つの指標で開発者に提示

し，採用方式を選択させる CASE ツールを提案している．

Kowark ら [57] は，Smalltalk 言語 [53] の一実装である Squeak [93] 向けの ORM として，SqueakSave を提案している．SqueakSave では，プログラム実行時にオブジェクトが永続化されると，そのクラスに定義された属性が，クラスに対応するテーブルの属性として認識される．この認識されたテーブルや属性が関係スキーマ上で不足している場合には，テーブルや属性の自動追加が事前に実施される．このように，開発者は関係モデルを明示的に設計する必要はなく，関係データベースを意識せずにプログラミングが可能である．その一方でクラスとテーブルの対応は，単純かつ固定的な対応関係に限られる．

文献 [7] では，top-down マッピングアプローチが業務アプリケーションの本開発で用いられるケースは少ないと述べている．その背景として，業務アプリケーション開発時のほとんどのケースで既存の業務システムが稼働している点[2]と，完全な新規開発であってもデータスキーマの設計を簡略化すると，後に開発されるであろうアプリケーションがそのデータを利用するのが難しくなる点を指摘している．この問題を克服するため，top-down マッピングアプローチをサポートする ORM フレームワークは，他のアプローチとの併用で用いられるケースが多い．そのケースについては 2.3 節 で述べる．

2.2.3 Meet-in-the-middle マッピングアプローチ

表 2.1 に示すとおり meet-in-the-middle マッピングアプローチでは，開発者が関係モデル R，オブジェクトモデル O，および補助情報 META を入力として与え，ORM 手法がその入力に基づき R-O 間の対応をとり，これらに基づい

[2]関係モデルを自動生成するアプローチでは，業務システムで使用中の既存の関係スキーマや DB アプリケーションに副作用をもたらすと考えられる

て関係スキーマ *rs*，永続化クラス *pc*，および RDB アクセスコード *map* を出力する手法をとる．

Java Persistence API (JPA) [76] は，開発者が Java で記述した永続化クラス上のクラス定義やプロパティ定義に Java アノテーションを付与することで，関係スキーマとの対応関係を記述する規格である．JPA の実装として，EJB3 [75] や Hibernate annotations [49] などがある．JPA は top-down マッピングアプローチを併せてサポートする．JPA については 2.3.3 節で改めて述べる．

Melnik ら [65, 64] は，関係モデルとオブジェクトモデル間の制約を SQL の拡張言語を用いて列挙することで ORM を実現するアプローチを提案している．列挙した制約文はコンパイルされて Query View と Update View に分解され，それぞれがデータ参照と更新を担う．

Cabibbo ら [14, 13, 15] は，関係モデルとオブジェクトモデが設計済みの場合にクラスとテーブルの組を構成することで ORM を実現する方式を，提案している．この組は以下の制約をもつ．

- クラスとテーブルのいずれかは，一つから構成される．他方は一つ以上から構成される．
- クラスとテーブルのそれぞれについて，一つの *primary* が定義される必要がある．ここで，primary を始点として外部キーまたは関連を辿ったとき，その多重度が必ず 1 であるという条件を満たす必要がある．

2.2.4 Metadata マッピングアプローチ

表 2.1 に示すとおり metadata マッピングアプローチでは，開発者がデータ

モデル C と補助情報 META を入力として与え，ORM 手法がその入力に基づき関係モデル R とオブジェクトモデル O を自動生成し，R-O 間の対応をとり，これらに基づいて関係スキーマ *rs*，永続化クラス *pc*，および RDB アクセスコード *map* を出力する手法をとる．このアプローチは，R や O とは異なるモデルを間に挟むことでオブジェクトモデルと関係モデルのインピーダンスミスマッチを緩和し，ORM を実現する方式といえる．

Hibernate [49, 8] では開発者は，Hibernate が定めるデータモデルを，Hibernate mapping XML (.hbm) ファイルに記述する．Hibernate の生成ツールが .hbm ファイルを読み取り，*rs*，*pc*，および *map* を生成することで，ORM が成立する．Hibernate は bottom-up マッピングアプローチを併せてサポートする．

Microsoft ADO.NET Entity Data Model (EDM) [2, 11] では開発者は，概念スキーマ（CSDL），ストアスキーマ（SSDL），マッピング仕様（MSL）の三種類を設計し，XML 形式で記述する．これらを生成ツールにかけることで，*rs*，*pc*，および *map* が生成され，ORM が成立する．EDM は bottom-up と top-down マッピングアプローチを併せてサポートする．

Kensche らは，関係モデル，EER モデル，UML，OWL DL，および XML Schema の五つのモデルを記述する能力をもったモデル GeRoMe [54] および，GeRoMe モデル上の ORM である MAGIC [55] を提案している．

Hibernate，EDM，および MAGIC については，2.3.3 節で改めて述べる．

ER モデルを用いてデータスキーマを設計し，それに基づいて ORM を実現するアプローチがある．WebML [21, 20, 12, 99] および Enterprise Objects Framework (EOF) [56, 6] では，開発者が記述した ER モデルに基づいて一対の永続化クラスと関係スキーマが生成される．これらのアプローチは，各々のテーブルに対する永続化クラスの対応が固定的となる点で，記述力には制約が

ある．

2.3 永続化クラスと関係スキーマの対応関係に基づいた分類

ORM 手法が扱う永続化クラスと関係スキーマの対応関係に基づいた分類は，二者の単純な対応関係を簡易に実現できるか，あるいは複雑な対応関係を実現できるかという観点で行える．複雑な対応関係を実現する際には，ひとつのマッピングに対して指定するべき項目が多くなる傾向があると考えられる．このことは，二者の単純な対応関係の簡易な記述を妨げる．逆に，単純な対応関係の簡易な実現は，複雑な対応関係を実現する方法に制約を与えると考えられる．すなわちこの分類は，トレードオフを示すと考えられる．

表 2.4 に，自動生成されるデータモデルと O-R 間の対応関係に基づいて ORM 手法を分類した結果を示す．以下，2.3.1 節 で本研究における単純な対応関係と複雑な対応関係の定義を述べたのちに，表 2.4 に沿って，2.3.2 節 で単純な対応関係を簡易に実現できるアプローチについて述べ，2.3.3 節 で複雑な対応関係を実現できるアプローチについて述べる．

2.3.1 本研究における単純な対応関係と複雑な対応関係の定義

本節では，序論で述べた単純な対応関係と複雑な対応関係について，具体的な定義を与える．本論文では，断りなく単純な対応関係や複雑な対応関係とよぶときには，永続化クラスと関係スキーマの対応関係について述べているものとする．

本研究では，単純な対応関係にあてはまらない対応関係は，全て複雑な対応関係と定義する．よって以下，単純な対応関係について述べる．本研究で定義する単純な対応関係には，標準の単純な対応関係（例：図2.1），結合テーブルによる単純な対応関係（例：図2.2），継承による単純な対応関係（例：図2.3）の三種類がある．なおここに示す単純な対応関係の扱いは，文献 [36, 3, 8, 103, 38] などに示された内容を参考にしており，永続化クラスと関係スキーマの対応関係を扱う上で広く受け入れられている手法である．

標準の単純な対応関係は，以下に示す三条件を全て満たす対応関係である．

- テーブル一つに対して永続化クラスが一つで対応している．
- 対応している永続化クラスの属性とテーブルの属性が一対一に対応している．

表 2.4: 自動生成されるデータモデルと O-R 間の対応関係に基づいた，ORM 手法の分類

	単純な対応関係を簡易に実現	複雑な対応関係を実現できる
Bottom-up	ActiveRecord [36, 40], phpClick [52, 92], Thiran ら [96, 95, 42]	
Top-down	DataMapper [36, 91], Philippi ら [80], Lodhi ら [59], SqueakSave [57]	
Meet-in-the-middle		JPA [76], Melnik ら [65, 64], M^2ORM^2 [14, 13, 15]
Metadata	EOF [56, 6], WebML [21, 20, 12, 99]	Hibernate [49, 8], Microsoft EDM [2, 11], MAGIC [55], DBPowder

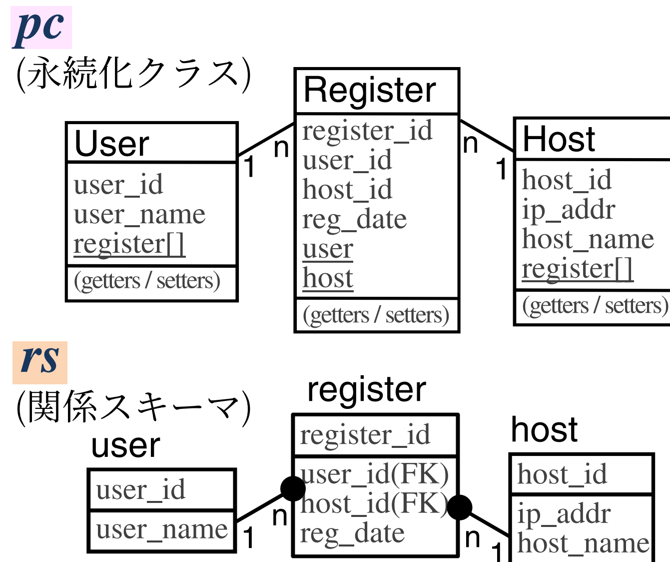


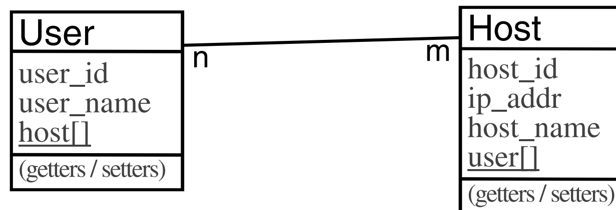
図 2.1: 単純な対応関係の例：標準の単純な対応関係

- 永続化クラスの関連とテーブルの外部キーが一対一に対応しており，扱える多重度が一致している．但し永続化クラスの関連は，片方向関連と双方向関連の両方が許される．なお多重度の扱いについては，表 2.2 の (*1,2) を参照のこと．

図 2.1 に，標準の単純な対応関係の例を示す．テーブル `user`，`register`，`host` に対してそれぞれ，永続化クラス `User`，`Register`，`Host` が対応している．対応している永続化クラスとテーブルの属性をみると，全て一対一に対応している．また，永続化クラス `User` – `Register` 間，`Register` – `Host` 間に双方向関連が定義されており，関係スキーマ上にこれに対応する外部キー `user_id` と `host_id` がある．したがって，図 2.1 に示す永続化クラスと関係スキーマは，単純な対応関係にある．

二つの永続化クラス間の多対多の関連は，永続化クラスに対応するテーブルを単純に二つ用意するだけでは扱うことができない．結合テーブルによる単純

pc
(永続化クラス)



rs
(関係スキーマ)

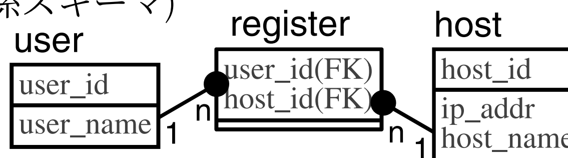


図 2.2: 単純な対応関係の例：結合テーブルによる単純な対応関係

な対応関係は，これを扱うための典型的なパターンであり，以下に示す四条件を全て満たす対応関係である．

- 二つの永続化クラスがあり，多対多の関連をもつ．
- これに対応して，二つのテーブルとこれらをつなぐ一つの結合テーブルがある．
- 結合テーブルは二つのテーブルの主キーの両方を外部キーとしてもっており，この外部キーの組を複合主キーとしている．
- 結合テーブルは，複合主キー以外の属性をもたない．

図 2.2 に，結合テーブルによる単純な対応関係の例を示す．永続化クラス `User` と `Host` は多対多関連をもつ．これに対して，関係スキーマではテーブル `user` と `host` のほかに結合テーブル `register` をもつことで対応している．

なお，結合テーブルが複合主キー以外の属性をもつ場合，その属性を扱うためには結合テーブルによる単純な対応関係を利用できない．

オブジェクトモデルにおける継承を関係モデルで直接扱うことはできない．継承による単純な対応関係は，これを扱うための典型的なパターンであり，単一テーブル継承（SR），クラステーブル継承（CR），具象テーブル継承（CCR）の三種類が知られている [36, 3, 8, 38, 14].

- SR: クラス階層に属する全てのクラスを，一つのテーブルにまとめて対応させる．
- CR: クラス階層上の各々のクラスを，一対一でテーブルに対応させる．
- CCR: クラス階層上の各々の具象クラスを，一対一でテーブルと対応させる．

図 2.3 に，継承による単純な対応関係の例を示す．永続化クラス `Bill` に対して，子クラス `CreditCard` と `BankAccount` がある．SR では，一つのテーブル `BILL` が三つのクラスに対応しており，属性も テーブル `BILL` にまとめられている．CR では，三つの各々のクラスが，それぞれ一対一でテーブルと対応している．CCR では，子クラス `CreditCard` と `BankAccount` にテーブル `CREDIT_CARD` と `BANK_ACCOUNT` が対応しており，各々のテーブルは親クラス `Bill` への対応をそれぞれに受けもっている．

2.3.2 単純な対応関係を簡易に実現できるアプローチ

単純な対応関係を簡易に実現できるアプローチのうち，bottom-up マッピングアプローチ [36, 40, 52, 92, 96, 95, 42] では関係モデルを起点とした ORM を

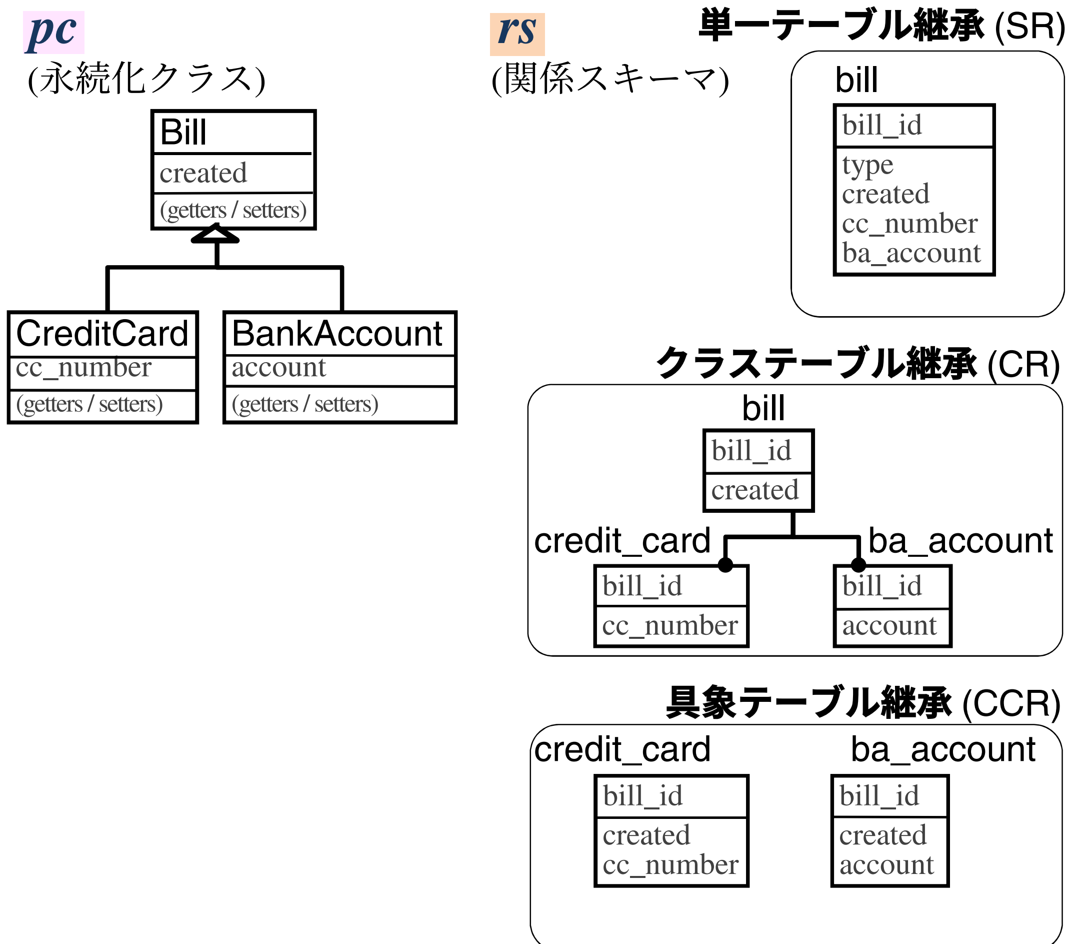


図 2.3: 単純な対応関係の例：継承による単純な対応関係

実施し，top-down マッピングアプローチ [36, 91, 80, 59, 57] ではオブジェクトモデルを起点とした ORM を実施する．したがって bottom-up や top-down では，テーブルと永続化クラスの対応関係が一対一ではない対応関係の記述は，特定のパターンを除いては困難である．たとえば単一テーブル継承（SR）のパターンと，永続化クラス同士の多対多関連を結合テーブルなしに実現するマッピングは，RoR の ActiveRecord，Thiran ら，DataMapper，および SqueakSave でサポートする．しかし，bottom-up や top-down を主とするフレームワークで

は，複数テーブルを結合して一つの永続化クラスで扱う対応関係をサポートできない．

Metadata マッピングアプローチのうち，単純な対応関係を簡易に実現できるアプローチ [56, 6, 21, 20, 12, 99] については，開発者が記述した概念モデルに基づいて一対の永続化クラスと関係スキーマを生成する手法をとるため，一つのテーブルに対して複数種類のクラス定義を与えることができない．なお Enterprise Objects Framework (EOF) については，既存の関係スキーマや永続化クラスを解析して EOF で用いる ER モデルに変換する機能がある．これを用いると bottom-up，top-down マッピングアプローチと併用できる．但し，開発者が記述する関係スキーマや永続化クラスについて EOF のサポート範囲を逸脱すると，開発済の内容が EOF によって上書きされる可能性がある点に注意が必要である．

以上の通り，単純な対応関係を簡易に実現できるアプローチでは，複雑な対応関係の実現が困難である．

2.3.3　複雑な対応関係を実現できるアプローチ

サポートするマッピングアプローチが単一のもの

複雑な対応関係を実現できるアプローチのうち，meet-in-the-middle マッピングアプローチについて，Melnik ら [65, 64] および M^2ORM^2 [14, 13, 15] のアプローチでは，事前に関係モデルとオブジェクトモデルが設計されていることが必須である．したがって，単純な対応関係を簡易には実現できない．

Metadata マッピングアプローチについて，MAGIC [55] は，五種類のデー

タモデル[3]を記述する能力をもったモデル GeRoMe 上で設計されている．したがって，関係モデルやオブジェクトモデルに加えて XML や OWL DL との相互運用が必要になる際に，設計が容易になると考えられる．一方で，マッピングに必要な情報量が増える上に要素の省略も困難になるため，単純な対応関係の簡易な実現を妨げる．

サポートするマッピングアプローチが複数のもの

複雑な対応関係を実現できるアプローチのうち meet-in-the-middle マッピングアプローチについて，JPA [76] では，永続化の対象となるクラスについて，Java アノテーションで `@Entity`（クラス），`@id`（識別子），`@one-to-many / @many-to-one`（関連），`@Inheritance`（継承）を Java ソースファイル上に指定する必要がある．識別子名を JPA の規約に従って決定[4]すれば，テーブルを自動生成させて関係モデルを意識せずに永続化クラスの設計が可能である．この点で，単純な対応関係については top-down マッピングアプローチで開発が可能である．但し JPA の言語仕様上，Java のソースコードが必要であり，永続化クラスを構成するプロパティや getter/setter メソッドを Java 言語で記述する必要がある．この点で，単純な対応関係を簡易には実現できない．また 2.2.2 節 で述べたとおり，top-down マッピングアプローチを業務アプリケーションの本開発に適用することには困難を伴う．

Metadata マッピングアプローチのうち Hibernate [49, 8] では，開発者は .hbm ファイルにオブジェクトモデルの主な要素としてクラス，プロパティ，識別子，関連を記述し，関係モデルの主な要素としてテーブル，属性，主キー，外部キーを記述する．この記述したスクリプトファイルを hbm2ddl ツールおよび

[3] 2.2.4 節 で述べた，関係モデル，EER モデル，UML，OWL DL，および XML Schema.
[4] クラス名に “_id” を付与した文字列を識別子名にする必要がある．

hbm2java ツールの入力として与えると，関係スキーマ *rs*，永続化クラス *pc*，および RDB アクセスコード *map* が生成され，ORM が実現される．この手法の利点は，オブジェクトモデルと関係モデルについて，一つ一つ要素の対応を記述できることである．一方，単純な対応関係に限定したデータモデル設計を行い，かつ名称をオブジェクトモデルの要素のものに倣うと決めた場合を考える．この場合 Hibernate では，オブジェクトモデルのクラス，プロパティ，識別子，関連と関係モデルの外部キーを省略できない．つまり .hbm ファイルを記述するためには，オブジェクトモデルと関係モデルの両方を意識する必要があり，単純な対応関係の簡易な実現を妨げる．

Hibernate では，開発者が既存のテーブルに対する永続化クラスの対応関係を明らかにする reveng.xml という XML ファイルを記述し，これを middlegen ツールの入力として与えることで，bottom-up マッピングアプローチを実現できる．但しこのアプローチは，テーブルと永続化クラスが一対一に対応する関係のみを扱う．この方式では .hbm ファイルが生成されるので，既存の関係スキーマを起点として Hibernate を導入する場合に有用である．但し開発者が .hbm ファイルを編集した後に middlegen ツールで .hbm ファイルを再生成すると編集内容は上書きされる．したがってこのアプローチは，hbm2java や hbm2ddl を使用した通常の開発方式を開始する前のみ，利用が可能である．

Metadata マッピングアプローチのうち Microsoft ADO.NET Entity Data Model (EDM) [2, 11] では，開発者は永続化クラス *pc* に対応する概念モデルである CSDL，関係スキーマ *rs* に対応する SSDL，および RDB アクセスコード *map* に対応する MSL の三種類の定義を XML で記述する必要がある．したがって，単純な対応関係を簡易に実現するためにはツールの援用が必須であり，また複雑な対応関係を実現する場合でも三種類の関連する言語を矛盾なく記述するのは容易ではないと考えられる．Microsoft では，EDM による ORM を実現

する枠組として，Entity Framework を提供している．Entity Framework のうち Entity Data Model ツール [29] を用いることで，bottom-up および metadata マッピングアプローチを実現できる．

- Entity Data Model ウィザードは，既存の関係スキーマを解釈し，CSDL，SSDL，MSL，および永続化クラスを生成する．
- モデルの更新ウィザードは，既存の関係スキーマを正として，CSDL，SSDL，MSL，および永続化クラスとの差分を解消する．
- データベース生成ウィザードは，CSDL の内容に基づいて，MSL，SSDL，および関係スキーマを自動生成する．既存の MSL と SSDL は上書きされる．
- Entity Data Model デザイナーは，CSDL，SSDL，MSL，および永続化クラスの連携した編集機能を提供する．但しこのツールには，既存の関係スキーマ上のテーブルや属性の追加や編集をサポートしない，複数のテーブルを結合して一つのクラスに対応づける場合は全てのテーブルが同じ主キーをもつ必要がある[5]，永続化クラスの継承を関係スキーマで扱う際の CCR パターンをサポートしない[6] などの制約がある．
- Entity Data Model ウィザード，モデルの更新ウィザード，および Entity Data Model デザイナーの組み合わせにより，関係モデルを起点とした ORM である bottom-up アプローチを提供する．Entity Framework ではこれを，*Database First* とよんでいる．
- データベース生成ウィザードと Entity Data Model デザイナーの組み合わせにより，metadata アプローチを提供する．Entity Framework ではこれ

[5]http://msdn.microsoft.com/en-US/library/vstudio/cc716779(v=vs.100).aspx
[6]http://msdn.microsoft.com/en-US/library/vstudio/cc716685(v=vs.100).aspx

を，*Model First* とよんでいる．

Database First と Model First のいずれも，ORM の能力としては Entity Data Model デザイナーの制約を受ける．特に Model First は，CSDL の変更内容を関係スキーマに同期させる手段をもたないため使いづらいという指摘がある [68]. 一方で Database First では，単純な対応関係を扱いつつ永続化クラス，テーブル，または属性の名称のみを違えたいケースであっても，永続化クラスの設計に Entity Data Model デザイナーを用いつつ関係スキーマの設計に SQL DDL 文などを用いる必要が生じ，meet-in-the-middle マッピングアプローチと類似した，開発に要する負担が高い状況が生じる．また，主キーを異にして複数のテーブルを結合して一つのクラスに対応づける場合や，CCR パターンの継承を扱いたい場合には Entity Data Model ツールの援用を受けられず，CSDL, SSDL，および MSL の三種類の言語を矛盾なく記述する必要が生じる．

また Entity Framework では，C# 言語 [30] または VB.NET 言語 [31] で記述された永続化クラスを入力として，CoC に基づいて単純な対応関係をもつ関係スキーマを自動生成する top-down マッピングアプローチも提供する．Entity Framework ではこれを，*Code First* とよんでいる．CoC から逸脱する命名が必要な場合や複雑な対応関係の記述が必要な場合は，データ注釈[7] または Fluent API[8]を用いる．このうち Fluent API は，独自の命名規則を生成するために命名規則の生成処理をフックするための仕様である．永続化クラスの継承を関係スキーマで扱う場合や複数のテーブルを結合して一つのクラスに対応づける場合などには Fluent API を使う必要があり，柔軟な命名が可能な一方で開発の負担は大きい．

---

[7]http://msdn.microsoft.com/en-US/data/jj193542 (6. Data Annotations),
http://msdn.microsoft.com/en-US/data/jj591583.aspx

[8]http://msdn.microsoft.com/en-US/data/jj591617.aspx

2.4 本研究の位置づけ

序論で述べたとおり，近年の DB アプリケーションの開発動向を踏まえると，ORM 手法が単純な対応関係を簡易に実現しつつ複雑な対応関係にも対応でき，かつ単純な対応関係から複雑な対応関係への移行を効率的にサポートできることが望ましい．しかし本章で見たように，既存の ORM 手法は単純な対応関係と複雑な対応関係について，そのいずれかの記述には適しているが，もう一方の記述には難がある．したがって，単純な対応関係から複雑な対応関係への移行について，効率的なサポートは難しい．

本研究で提案する ORM 手法 DBPowder では，概念モデリングに基づいて単純な対応関係を簡易に実現できる．これに連携させて ObjectView と名付けた有向グラフを併用することで，アプリケーションロジックを走査経路と利用方法の形で簡潔かつ柔軟に記述することで，複雑な対応関係を実現できる．本研究は，複雑な対応関係を実現する要求が主にアプリケーションの個別の利用状況を踏まえて生じることに着目し，各々のアプリケーションが永続化データを利用する際の実体上の走査経路を，複雑な対応関係の実現に用いた点に特徴がある．ここで ObjectView が概念モデル EER を参照して定義を再利用するため，ObjectView による追加開発の負担を軽減できる．この連携による併用方式により，単純な対応関係を簡易に記述できるうえに，単純な対応関係から複雑な対応関係に移行する開発を効率的にサポートできる．

DBPowder で記述する複雑な対応関係は，概念モデル EER で記述済の単純な対応関係に依存する．この点で，このような依存関係を要しない Hibernate，JPA，Microsoft EDM よりも記述力は劣る．一方でこの方式により概念モデルを活用でき，単純な対応関係から複雑な対応関係に移行する開発を効率的にサポートできる．

ActiveRecord は，関係スキーマを基準にして，属性など永続化クラスを構成する要素の命名規則を設け，設定すべき項目を減らすことにより単純な対応関係を簡易に記述しようというアプローチである．この命名規則に従えない場合，命名規則の違反が連鎖して設定すべき項目数が膨らむことが，しばしばある．これに対して DBPowder では，開発者が記述した EER モデルを自動変換することで実施される関係スキーマと永続化クラスの基本設計に対して，差分をパラメータとして設定する．DBPowder が定めるデフォルト名称に従えない場合でも，EER モデルが定めるデータ構造は影響を受けないため，命名規則の違反が連鎖するような問題は起こらない．

ActiveRecord や ER モデルを起点として ORM を実現する他のアプローチでは，複雑な対応関係を実現できないが，DBPowder では EER モデルとの連携による ObjectView の併用により，単純な対応関係における簡潔な記述を損ねることなく，複雑な対応関係を実現できる．これにより DBPowder は，複雑な対応関係を実現できるアプローチでありながら，単純な対応関係を簡易に実現できる．

自動生成されるデータモデルに基づいた分類に従うと，本研究は EER モデルを用いた metadata マッピングアプローチを主とする．既存の関係スキーマがある場合，DBPowder ではテーブル名を指定してその属性を自らの属性として取り込む機能を有しており，限定的に bottom-up マッピングアプローチをサポートする．DBPowder は top-down や meet-in-the-middle マッピングアプローチを直接にはサポートしないが，EER モデルが関係モデルやオブジェクトモデルに依存しないモデルであり，かつ UML への変換が容易である [38] ことを踏まえると，オブジェクトモデルを強く意識した開発スタイルをとることはできる．

第3章　提案手法 DBPowder で用いるデータモデル

3.1　DBPowder で用いるオブジェクトモデル

DBPowder で用いるオブジェクトモデルは，ODMG3.0 [19] のサブセットを基本とし，Java [39] を参考にした拡張を含む．図 3.1 に，オブジェクトモデルを図示した例を示す.

オブジェクトモデルは，オブジェクトとリテラルで構成される．オブジェクトは識別子をもち，リテラルは識別子をもたない．オブジェクトは状態とふるまいをもつ．状態はプロパティの集合により定義され，ふるまいは，操作の集合により定義される．プロパティは，属性または関連である．DBPowder のオブジェクトモデルでは，getter/setter メソッドの組によってプロパティを表現し，プロパティ値の参照や変更は getter/setter メソッドの呼出により行う．オブジェクトモデルの利用者は，プロパティ値のインスタンスがどのように管理され，参照や変更がどのように実現されるかを，意識しない．なお，関連とメソッドの定義は本節で後述する．

構造化リテラルはリテラルの集合により定義され，各々の要素リテラルは要素名をもつ．定義された構造化リテラルは，リテラルと同様に扱うことができる．

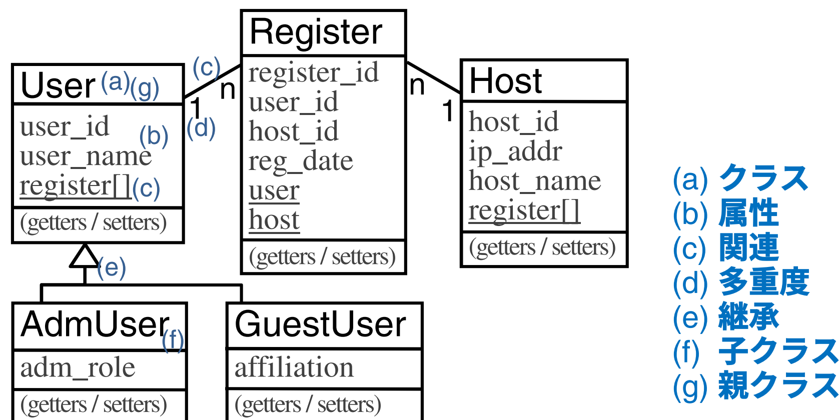


図 3.1: DBPowder で用いるオブジェクトモデルの例

型は，オブジェクトやリテラルを分類する．オブジェクト型は，インタフェース定義とクラス定義により構成される．リテラル型は，Java のものに準じる．オブジェクト型のふるまいのみを定義した仕様をインタフェース定義とよび，オブジェクト型の状態とふるまいを定義した仕様をクラス定義とよぶ．なお，インタフェース定義では仕様を実装しない．

次に，具象クラスと抽象クラスを以下のように定める．仕様を実装済のクラスを具象クラスとよぶ．記述された具象クラスにおいて，状態は属性や関連として全て実装されており，ふるまいはメソッドとして全て実装されている．仕様の全てまたは一部が実装済ではないクラスを，抽象クラスとよぶ．抽象クラスでは，クラス定義を構成する属性，関連，メソッドのいずれかについて，実装されていないものがある．以降，断りなくクラスとよぶときには，具象クラスを指すものとする．

型 A, B があるとき，型 A がもつ状態やふるまいを型 B が全て備え，型 B のオブジェクトの存在が同時に型 A のオブジェクトの存在を意味するとき，「型 B は型 A を継承する」という．クラスは，一つのクラスと一つ以上のインタフェー

スを継承できるが，複数クラスを多重継承することはできない．インタフェースは， 一つ以上のインタフェースを継承できるが，クラスを継承することはできない．

二つの型について，相方のオブジェクトを参照できる関係のことを，関連とよぶ．相互に参照できる関連を双方向関連とよび，片方向のみに参照できる関連を，片方向関連とよぶ．片方向関連は，型 A, B について，$A \to B$ の参照は可能だが，$B \to A$ の参照はできないといったものである．関連により参照できるオブジェクトの個数に応じて，1 または多の多重度を定義する．双方向関連の場合の多重度は，1:1, 1:n, n:1, n:m のいずれかとなる．多の多重度をもつ場合，関連を保持するプロパティの型はコレクション型となる．コレクション型は，List, Set, Map のいずれかとする[1]．

以後，is-a 関連を継承とよび，has-a 関連を関連とよぶこととする．

オブジェクトのプロパティを保存することで，オブジェクトを別のプロセスや計算機から復元可能にする操作のことを，永続化とよぶ．永続化に関する操作は，保存，ロード（復元），更新，削除の四種類がある．

図 3.1 に示した例では，(a) クラス `User` は，(b) 属性 `user_name` と，クラス `Register` への (c) 関連 `register` をもつ．またクラス `Register` は，クラス `User` への関連 `user` をもつ．したがって，クラス `Register` と `User` は双方向に関連をもつ．クラス `User` とクラス `Register` の (d) 多重度は `1:n` である．クラス `User` からクラス `Register` への多重度は多なので，クラス `User` がもつ関連プロパティ `register` は，コレクション型となる．クラス `AdmUser` と `GuestUser` は，クラス `User` を (e) 継承する．クラス `User` は，(f) 子クラス

[1] ODMG3.0 では，コレクション型は Set, Bag, List, Array, Dictionary のいずれかとしている．Java では，Bag や Array を List で代用するケースが多く，また，Dictionary は Map クラスにより実現される．そこで本研究におけるコレクション型は，List, Set, Map の三種類とする．

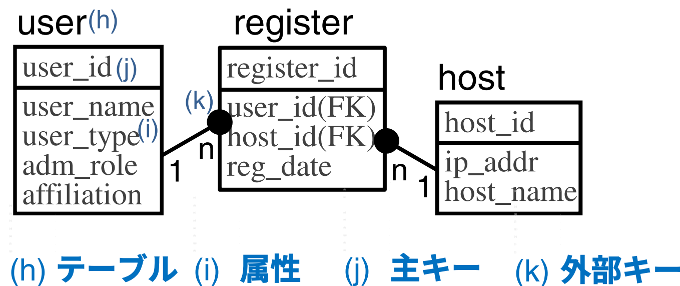


図 3.2: DBPowder で用いる関係モデルの例

AdmUser と GuestUser をもち，AdmUser と GuestUser の (g) 親クラスである．

3.2 DBPowder で用いる関係モデル

本節では，DBPowder で用いる関係モデル（R）について述べる．図 3.2 に，関係モデルを図示した例を示す．

属性 $A_1, A_2, ..., A_n$ に対して，各々の属性のドメインの直積 $D_1 \times D_2 \times ... \times D_n$ の有限部分集合を，関係テーブル T とする．関係テーブルを本論文では，単にテーブルとよぶ．テーブルの各要素を，タプルとよぶ．

次に，候補キー，主キー，外部キーを以下のように定める．テーブル中のタプルを一意に識別できる属性の部分集合で，要素数が極小のものを，候補キーとよぶ．候補キーからある一つを選択したものを，主キーとよぶ．主キーにおいては，各属性値は空値（null）を許さない．テーブル $T_1(..., A_{FK}, ...)$ と $T_2(A_{PK}, ...)$ について，任意の A_{FK} の値が，null または A_{PK} 上の ある値に一致するとき，A_{FK} を外部キーとよぶ．ここで特に $T_1 = T_2$ の場合，自己参照とよぶ．

DBPowder が扱う関係代数演算は，射影，選択，自然結合（等結合）である．

データ定義言語およびデータ操作言語としては，SQL を用いる．

図 3.2 に示した例では，(h) テーブル `user` は，(i) 属性 `user_name` と，(j) 主キー `user_id` をもつ．テーブル `register` は，テーブル `user` への (k) 外部キー `user_id` をもつ．

3.3 DBPowder で用いる EER モデル

本節では，DBPowder で用いる EER モデルについて述べる．図 3.3 に，EER モデルを図示した例を示す．

DBPowder で扱う EER モデルでは，属性 A，実体 E，関連 R，連結度 C の四つを基本的な要素とする．定義は以下による．まず，データモデルが対象としている，識別可能なことがらを e とする．e は一つ以上の性質をもち，そのそれぞれを属性 $A_1, A_2, ..., A_n$ とよぶ．この属性について，集合 $\{A_1, A_2, ..., A_n\}$ が同一である e を集めた集合が，実体 E である[2]．二つの実体 E_i, E_j について，要素間の連結度 C を定義できるとき，実体 E_i, E_j は関連 R_{ij} をもつとする．ここで連結度 C は，1:1, 1:n, n:1, n:m のいずれかとする．三つ以上の実体 $E_1, E_2, E_3, ...$ に関する関連は，以下に示すいずれかの方法で二つの実体間の関連に変換して扱う．

- 関連を扱うための実体 E' を別途導入し，E' に対する各々の実体 $E_1, E_2, E_3, ...$ との関連を扱うことで記述する [38].
- 三つの実体 E_1, E_2, E_3 を扱う関連の場合は，Jones ら [51] の示す方法で二

[2]Chen が示したオリジナルの ER モデル [23] では，ことがら e を「実体」，実体 E を「実体集合」とよんでいる．Teorey らが示した EER モデル [94] では，E を「実体」とよんでいる．本研究では E を「実体」とよぶ．

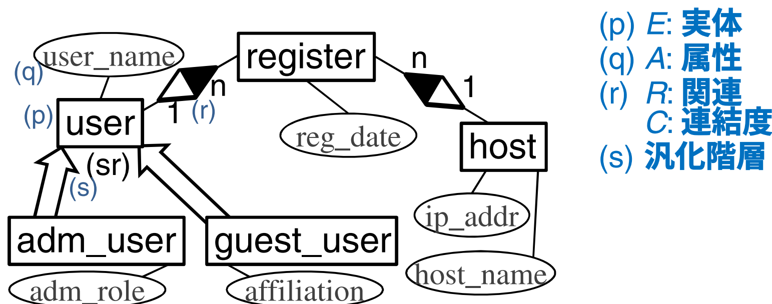


図 3.3: DBPowder で用いる EER モデルの例

つの実体間の関連を複数導入し，分解することで記述する．

次に，候補キーと主キーを定める．候補キーは，E 中で e を一意に識別できる属性の集合である．主キーは，候補キーのうち，モデルで決定したあるひとつのキーである．なお DBPowder では，主キーの記述を省略可能とする．省略した場合は，DBPowder が代理キーを主キーとして割り当てる．

DBPowder で扱う EER モデルでは，汎化階層を扱うことができる．ある実体 A の属性をすべて継承した実体 B について，B における要素の存在が同時に A における要素の存在を意味するとき，A と B は汎化階層の関係にあるという．このとき，B の主キーは A によって定義される．

図 3.3 に示した例では，(p) 実体 `user` は，(q) 属性 `user_name` と，実体 `register` への (r) 関連をもち，この (r) 連結度は `1:n` である．実体 `user`，`adm_user`，および `guest_user` は，(s) 汎化階層の関係にある．

第4章　DBPowder：概念モデリングに基づく O/R マッピング手法

4.1　はじめに

DB アプリケーションの開発について，工期を短縮した迅速な開発が強く求められる一方で，仕様を固めるのに時間を要するケースが増えている．ここで単純な対応関係を簡易に実現できる ORM フレームワークを用いると，複雑な対応関係をうまく扱えない．その一方で，複雑な対応関係を実現できる ORM フレームワークでは，詳細な設計を得にくい開発初期の段階から永続化クラスとテーブルの詳細な対応関係を記述する必要が生じる上に，機能拡張の際にこれを修正する必要が頻繁に出てくる．このため，迅速な開発が強く求められる開発初期の段階から開発の負担が大きくなる．

第 4 章ではこの問題の克服のために，概念モデリングに基づく O/R マッピング（ORM）手法，DBPowder を提案する．DBPowder では，以下の三点を提供することで問題を克服する．

- Extended Entity-Relationship (EER) モデル [94] を用いた，単純な対応関係を簡易に実現できる ORM 手法

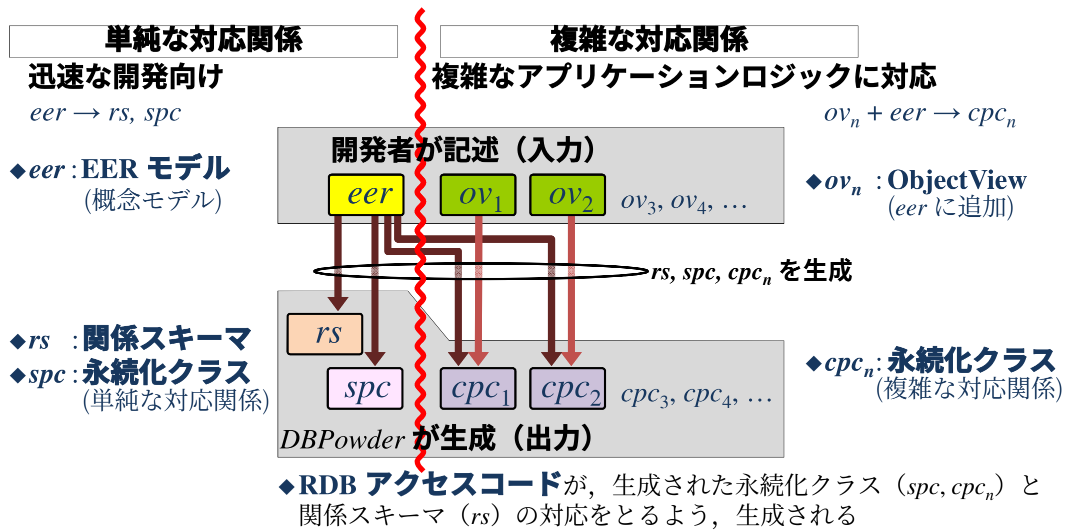


図 4.1: DBPowder による O/R マッピング（ORM）の概要

- EER モデル上の実体の走査経路と利用方法を有向グラフベースで指定した *ObjectView* の併用による，複雑な対応関係を実現できる ORM 手法
- EER モデルと ObjectView の組み合わせで追加的または修正的に複雑な対応関係をもつ永続化クラスを生成する手法による，単純な対応関係から複雑な対応関係に移行する開発の効率的なサポート

DBPowder による ORM の概要を図 4.1 に示す．開発者は初め，EER モデル (*eer*) のみを用いて実体，属性，関連，および連結度を記述する．DBPowder は，この *eer* を入力として永続化クラス (*spc*) と関係スキーマ (*rs*) を生成する．これにより，*spc* と *rs* との単純な対応関係による ORM が成立する．開発が進み複雑な対応関係をもつ ORM が必要になった際には，開発者は ObjectView(ov_n) を追加で記述する．DBPowder は，この ov_n を入力として永続化クラス (cpc_n) を生成する．これにより，cpc_n と生成済の *rs* との複雑な対応関係による ORM が成立する．なお，それぞれの ORM について DBPowder は，生成される永続

化クラスと関係スキーマの対応をとるよう，RDB アクセスコード[1]を生成する．

EER モデルは，詳細な設計を得にくい開発初期にデータの性質を捉えた概念モデルの開発を行うために，開発者が利用する．DBPowder はこれに基づいて単純な対応関係をもつ永続化クラスと関係スキーマを生成する．EER モデルによる枠組は開発初期には威力を発揮するが，開発が進み複雑な対応関係を設計開発する必要が生じると，EER モデルのみでは十分に対応できなくなる．そこで，アプリケーションロジックの各場面に対応した複雑な対応関係を開発者が記述できるよう，ObjectView を導入する．開発者が ObjectView を追加すると，DBPowder はその内容に基づいて永続化クラスの追加や修正を実施する．なお関係スキーマは ObjectView から追加や修正を受けることはなく，EER モデルから生成されたものがそのまま用いられる．

本手法の特徴は，単純な対応関係の簡易な実現と複雑な対応関係の実現についてサポートを両立できる手法を提案した点と，この手法が単純な対応関係から複雑な対応関係に移行する開発を効率的にサポートできる点にある．EER モデルと連携させて ObjectView を併用することで，DBPowder は単純な対応関係の迅速な開発をサポートするとともに，複雑な対応関係による永続化クラスを簡易に追加できる．また，単純な対応関係から複雑な対応関係に移行する開発を効率的にサポートできる．

また DBPowder では，EER モデルと ObjectView をコンパクトに記述できる言語 DBPowder-mdl を提供する．DBPowder-mdl は，階層構造および設定より規約（CoC；2.2.1 節）[33, 22] に基づく簡潔な記法を提供しつつ，グラフを記述するための記法や CoC から逸脱したスキーマの設計も柔軟にサポートする．

[1]RDB アクセスコードは，永続化クラスから RDB へクエリし応答を受けとるコード（第 1 章）．

本章の提案の有効性を検証するために，記述力の評価と開発に要する負担の評価を実施する．

記述力の評価では，単純な対応関係を簡易に実現できる Ruby on Rails（RoR）と複雑な対応関係を実現できる Hibernate を比較対象とする．DBPowder-mdl の言語定義と Hibernate の ORM 記述を行う .hbm XML ファイルの文書型定義（DTD）を比較し，その結果を RoR の機能と照合することで，DBPowder の記述力を比較検証する．

開発に要する負担の評価では，二種類の比較評価を実施する．ひとつは単語数を用いた記述量の評価である．もうひとつは，複雑な対応関係に移行する開発で要した，編集の回数および編集した単語数の比較である．記述量の評価では，DBPowder，Ruby on Rails（RoR），および Hibernate について，各々のフレームワークが ORM を実現する際に必要とするコードを用意し，各々のコードに含まれる単語数を数えることで，開発で要求される記述量を比較する．これを，単純な対応関係や複雑な対応関係について行う．編集の回数および編集した単語数の比較では，DBPowder と Hibernate について，単純な対応関係のみで構成される ORM に対して複雑な対応関係を追加する際に，必要とされる編集回数や単語数を数え，比較する．

本章の構成を以下に示す．4.2 節で EER モデルを用いた単純な対応関係による ORM について述べ，4.3 節で ObjectView を加えた複雑な対応関係による ORM を述べる．4.4 節では eer と ov_n を記述するための言語 DBPowder-mdl を示す．なお DBPowder-mdl については，4.2.2 節 と 4.3.3 節 で例示ベースの導入を行う．提案手法 DBPowder の評価として，4.5 節で 記述力を評価する．続いて，開発に要する負担の評価として，4.6 節では単語数を用いた記述量の比較を行い，4.7 節では複雑な対応関係に移行する開発における，編集の回数と単語数の比較を行う．最後に 4.8 節で本章をまとめる．

4.2 EER モデルによる単純な対応関係を簡易に実現できる ORM 手法

4.2.1 EER モデルを用いた ORM の手順

4.1 節 で述べたように，迅速な開発が求められる開発初期フェーズでは，開発者は EER モデル *eer* のみを記述する．DBPowder はこの記述内容を入力として解釈し，単純な対応関係をもつ関係スキーマ *rs* と永続化クラス *spc* を生成する．また，この *rs* と *spc* の対応をとるよう，RDB アクセスコードを生成する．

この手順を，以下に ORM 手順 1 – 4 で述べる．ORM 手順 1 では，開発者が記述した *eer* 上の実体に代理キーが追加される．ORM 手順 2 では，*eer* に基づき *rs* が生成される．ORM 手順 3 では，*eer* に基づき *spc* と RDB アクセスコードが生成される．なお汎化階層の扱いについては，ORM 手順 4 で別途述べる．

ORM 手順 1 – 4 で用いられる *eer*，*rs*，*spc* の各要素の対応関係を 表 4.1 に示す．なお表中の ov_n と cpc_n については，4.3 節 で述べる．また，例を 図 4.2 に示し，4.2.3 節 で説明する．

ORM 手順 1 DBPowder は，*eer* 上の主キーが省略された実体に対して，代理キーを追加する．

ORM 手順 2 DBPowder は，Teorey らが示した手法 [94] に基づき，*eer* から *rs* を生成する．はじめに *eer* の実体 E に対応して *rs* のテーブル T を生成する．次に E のもつ全ての属性を T の属性とし，E の主キーを T の主

キーとする．

eer の関連 R に対しては，連結度が 1 対多や多対 1 の場合は，連結度が 1 側の実体に対応したテーブル T_1 の主キーを，連結度が多側の実体に対応したテーブル T_2 に加え，この加えたキーを外部キーとする．R が属性をもつ場合は，R がもつ属性を T_2 に加える．

連結度が 1 対 1 の場合は，二通りの対応が取れる．一つ目は，いずれかの 1 を多に読み替え，1 対多や多対 1 の扱いとする．二つ目は，片方のテーブル T_1 上の主キーを他方のテーブル T_2 の主キーへの外部キーとする．この場合，R が属性をもつ場合は R がもつ属性を T_1 に加える．

連結度が多対多の場合は，2.3.1 節 で示した結合テーブルによる単純な対応関係を EER モデルの扱いに読み替える．即ち，R に対応する結合テーブル T' を生成し，R が結びつける二つのテーブルの主キーの両方を T' に加え外部キーとし，この外部キーの組を T' の複合主キーとする．なお R が属性をもつ場合は，この属性を全て T' の属性とする．

表 4.1: *eer*, ov_n, *rs*, *spc*, cpc_n の主な対応関係

データモデル	用いられる対応関係	対応する要素			
eer : EERモデル	単純 / 複雑	実体	主キー	属性	関連
ov_n : ObjectView (in complex corr.)	複雑	節点	(*eer* 内の主キー)	(*eer* 内の属性)	枝
rs : 関係スキーマ	単純 / 複雑	テーブル	主キー	属性	外部キー，結合テーブル(*1)
spc : 永続化クラス (in simple corr.)	単純 / 複雑	クラス	識別子	属性	関連
cpc_n : 永続化クラス (in complex corr.)	複雑				

(*1) 結合テーブルの使用は，多対多関連のみ

ORM 手順 3 DBPowder は，ORM 手順 2 と類似の方法により，*eer* から *spc* を生成する．クラスと属性については，ORM 手順 2 に示した *rs* のテーブルと属性の生成方法と同様にして，生成する．関連については，ORM 手順 2 に示した *rs* の外部キーと同様の方法により片方向関連を生成し，同時に逆方向の関連を生成することにより，生成する．ここで *spc* 上の関連は，*eer* 上の定義に沿って双方向関連となる．関連の多重度は，*eer* で記述したものを採用する．

なお *eer* 上での連結度が多対多の場合は，ORM 手順 2 と同様にして関連 R に対応したクラス c' を生成し，関連が結びつける二つの実体に対応するクラス c_1 と c_2 から，それぞれ c' への一対多の双方向関連を生成する．これとともに，c_1 上に c' 経由で c_2 にアクセスする getter/setter メソッドと，c_2 上にその逆方向の getter/setter メソッドを生成することで，*eer* 上の多対多を *spc* 上で表現する[2]．なお R が属性をもつ場合は，この属性を全て c' の属性とする．R が属性をもたない場合は，開発者はクラス c' を直接には必要としないが，R が属性をもつ場合との互換性を考慮して開発者に開示する．

また DBPowder は，生成された *rs* と *spc* の対応をとるよう，RDB アクセスコードを生成する．

EER モデルの汎化階層は，永続化クラスでは継承を用いて直接扱うことができるが，関係スキーマ上で直接扱うことはできない．DBPowder では，2.3.1 節 で示した継承による単純な対応関係を EER モデルの扱いに読み替えることで，これに対応する．

ORM 手順 4 DBPowder では，文献 [36, 3, 8, 38, 14] に示された単一テーブル継承（SR），クラステーブル継承（CR），具象テーブル継承（CCR）の

[2] 3.1 節で示したように，DBPowder のオブジェクトモデルでは getter/setter メソッドの組によって関連を表現し，開発者は参照や変更がどのように実現されるかを意識しない．

いずれかの方法で，*eer* の汎化階層を *rs* で扱う．ここで，

- SR: 汎化階層に属する全ての実体を，一つのテーブルにまとめて対応させる．
- CR: 汎化階層上の各々の実体を，一対一でテーブルに対応させる．
- CCR: 汎化階層上の各々の葉に属する実体を，一対一でテーブルと対応させる．

SR，CR，CCR ともに，各々の実体がもつ属性を，対応したテーブルの属性とする．なお CCR においては各々の葉に属する実体について，その親がもつ属性も葉が対応するテーブルの属性とする．

また，*eer* の汎化階層をそのままクラス階層に読み替えて，*eer* から *spc* を生成する．

4.2.2 DBPowder-mdl による単純な対応関係の記述（導入）

図 4.2-1 と図 4.2-2 に対応する DBPowder-mdl の記述例を，図 4.3 に示す．本節では，DBPowder-mdl による単純な対応関係の記述の導入を行う．なお DBPowder-mdl の詳細な説明は 4.4 節 で行う．

図 4.3 のうち指し示されている記述例は，実体 `user`，属性 `user_name`，実体 `user` と `register` の関連，および汎化階層 `adm_user` である．DBPowder-mdl では以下のように記述する．

- 実体を，［ ］囲みで記述する．
- 属性を，実体からインデントを一つ下げて記述する．

- 実体 E_1 からインデントを一つ下げて記述する実体 E_2 は，同時に E_1 と E_2 の関連を表す．これを関連実体とよぶ．このとき，連結度を < > 囲みで記述する．

- 汎化階層に属する実体は，汎化階層の親となる実体からインデントを一つ下げて，inherit-sr，inherit-cr，inherit-ccr のいずれかを宣言することで記述する．

図 4.3 の例では，実体は user，register，host，adm_user，guest_user であり，属性は user_name，reg_date，ip_addr，host_name，adm_role，affiliation である．register は user と関連をもち，user:register の連結度は 1:n である．adm_user は user 配下の汎化階層に属する実体であり，inherit-sr が指定されているため，単一テーブル継承（SR）が採用される．

4.2.3 単純な対応関係の実現例（user – register – host スキーマ）

単純な対応関係の実現例（user – register – host スキーマ）を，図 4.2 に示す．ここで 図 4.2-1 は，開発者が設計した EER モデル *eer* であり，図 4.2-2 は EER モデルから生成された関係スキーマ *rs* および，単純な対応関係をもつ永続化クラス *spc* である．また 図 4.3 は，4.2.2 節 で導入した DBPowder-mdl での記述である．

rs と *spc* 内の user_id，register_id，host_id は，ORM 手順 1 で追加された代理キーに対応している．開発者が記述した *eer* 上の実体 user，register，host をもとに，ORM 手順 2 で *rs* 上のテーブル user，register，host が生成され，ORM 手順 3 で *spc* 上の永続化クラス User，Register，Host が生成される．

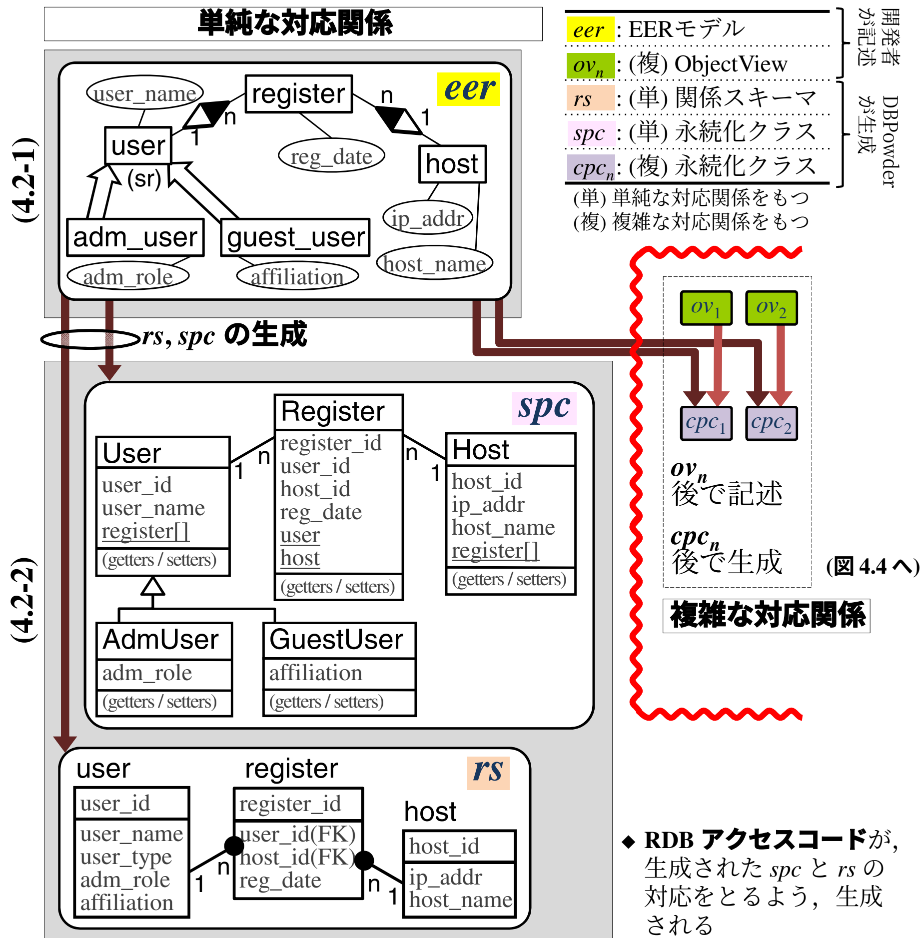


図 4.2: DBPowder における単純な対応関係の実現例（user – register – host スキーマ）．(4.2-1) *eer*, (4.2-2) *rs*, *spc*． *eer* を記述 → *rs*, *spc* が生成される．

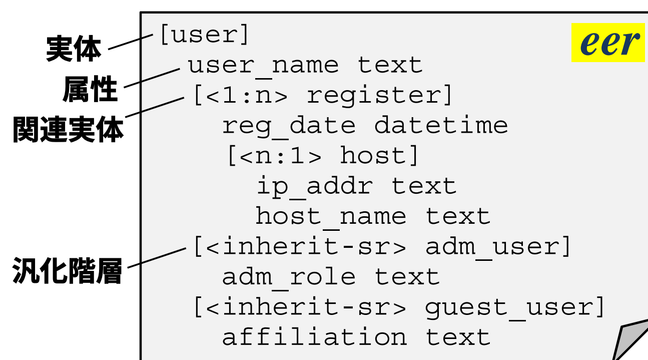


図 4.3: DBPowder-mdl の記述例：図 4.2 の EER モデル *eer*

eer で記述された関連に基づいて，*rs* 上のテーブル `register` に外部キー (FK) が二つ追加され，*spc* 上の永続化クラス `Register` にも対応する属性が追加される．また，永続化クラス `User – Register` 間や，`Register – Host` 間で，双方向関連が生成される．永続化クラス `User` と `Host` はそれぞれ，`Register` への多の多重度をもつため，関連 `register` はコレクションとなる．

ここで示した汎化階層の `user`, `adm_user`, `guest_user` に関係する ORM は，単一テーブル継承（SR）の例であり，*eer* と *spc* の各要素が一対一に対応する一方で，*rs* 上ではテーブル `user` に属性がまとめられている．汎化階層に関係する箇所以外は，属性や関連は各々，定義に基づいて一対一に対応している．

以上に示す EER 概念モデル，概念モデルに基づいた関係スキーマと永続化クラスの生成，および生成された二者の単純な対応関係を，DBPowder-mdl（図 4.3）ではコンパクトに記述できている．

4.3 ObjectView による複雑な対応関係を実現できる ORM 手法

4.3.1 ObjectView の導入

複雑な対応関係を実現するために，DBPowder では *ObjectView* を導入する．ObjectView は，EER モデル（3.3 節）に対する有向グラフベースのオブジェクト記述形式である．開発者は，記述済の EER モデル（*eer*）に連携させて ObjectView（ov_n）を記述する．DBPowder はこの ov_n を入力として解釈し，*eer* の入力から生成済の関係スキーマ（*rs*）に対して複雑な対応関係をもつ永続化クラス（cpc_n）を生成する．また，この *rs* と cpc_n の対応をとるよう，RDB ア

クセスコードを生成する．このように開発者は，ov_n を記述することで複雑な対応関係をもつ永続化クラスを簡易に生成できる．なお ObjectView は永続化クラスの生成を目的としており，関係スキーマの構築や修正の機能は有しない．

ObjectView では開発者は，アプリケーションロジックが必要とする実体と関連について，走査経路と利用方法を指定する．これにより，複雑な対応関係をもつ永続化クラスが追加的または修正的に生成される．ObjectView がサポートする rs と cpc_n の複雑な対応関係は，以下に示す三種類である．(a）あるテーブルを基準にして多対 1 や 1 対 1 で結合される複数のテーブルを，一つの永続化クラスに対応づける対応関係，(b）テーブル内にある属性のうち一部を，永続化クラス内でまとめて一つの構造化リテラル[3]として対応づける対応関係，(c）複数のテーブルに分散する各々の属性の集合を，一種類のインタフェース[3]として対応づける対応関係．

4.3.2　ObjectView の定義と ORM の内容

ObjectView は，以下に示す ov 記述 1, 2, 3 から構成される，有向グラフ G と G の要素へのグループ化指定によって定義される．

- 【ov 記述 1】 アプリケーションロジックが必要とする実体と関連の，走査経路の指定による，有向グラフ G
- 【ov 記述 2】 G を単純化するための，G の節点のグループ化指定
- 【ov 記述 3】 グループ化された G の部品化（コンポジション）と多態性（ポリモーフィズム）を実現するための，構造化リテラルとインタフェースによる属性のグループ化指定

[3]構造化リテラルとインタフェースの定義は，3.1 節 による．

開発者は *ov* 記述 1 で，*eer* 上の実体 E と関連 R の走査経路の指定によって複雑な対応関係を実現する範囲を指定することで，G のグラフ構造を記述する．*ov* 記述 2 と *ov* 記述 3 ではアプリケーションロジックの必要性に応じて G の要素へのグループ化指定を行う．*ov* 記述 2 では G を単純化するための節点のグループ化を実施し，*ov* 記述 3 ではグループ化された G の部品化と多態性を実現する．このように構成した，グループ化指定を伴う有向グラフ G は，*eer* におけるアプリケーションロジックに対応した実体の走査経路と利用方法を指定したグラフとなる．

以下，*ov* 記述 1, 2, 3 の記述内容を述べ，その後に G により実現される ORM について述べる．

***ov* 記述 1** アプリケーションロジックが必要とする実体と関連の，走査経路の指定による，有向グラフ G

ObjectView のグラフ構造は，始点をもつ有向グラフ $G = (s, \{E_v\}, \{R_v\})$ により定義される．ここで s は始点，$\{E_v\}$ は節点の集合，$\{R_v\}$ は枝の集合である．節点 s と E_v には *eer* 上に対応する実体 E が存在し，枝 R_v には *eer* 上に対応する関連 R が存在する．

開発者は ObjectView で扱おうとする *eer* において，アプリケーションロジックが中心に扱う実体 E_s を一つ決め，これを始点 s と対応づける．この E_s を中心実体とよぶ．続いて開発者は，*eer* 上で E_s を始点としてアプリケーションロジックで使用される実体 E と関連 R を走査し，走査した内容を節点 E_v や枝 R_v に対応づける．このようにして有向グラフ G を決定する．

このように決定された G において，それぞれの枝 R_v は終点側にもとの R に対応した連結度をもち，EER モデルで指定した連結度を引き継ぐ．汎化階層上の E や R に対応づけられた節点 E_v や枝 R_v については，汎化階層の親から

子にたどる各々の経路に対応して，定義される．

また実体 E のもつ全ての属性は，E に対応する節点 E_v の属性となる．R が属性をもつならば，4.2.1 節 の ORM 手順 3 で関連がもつ属性を永続化クラスの属性にしたのと同様の方法で，R 上の全ての属性が E_v に加えられる．

なお，関連 R の連結度が多対多の場合は，R は節点として扱われる．この場合，R をアプリケーションロジックの中心として扱うことも可能である．このような R を中心関連とよぶこととする．以後，中心実体について論じる場合，特に断りがない場合は中心関連を含むものとして扱う．

ある E_v について始点から E_v に至る経路が複数本あるとき，E_v はこの経路上の枝に対応する関連が示す制約条件を，全て満たす必要がある．複数本の経路について論理積がとられることを意図しない場合は，E_v とは別の節点 E_v' を E に割り当て，E_v への経路のうち一つについて E_v' へ経路を付け替える．これにより，それぞれの経路が示す条件を E_v と E_v' に振り分けることができる．

> ***ov* 記述 2** G を単純化するための，G の節点のグループ化指定

ov 記述 2 では，始点から辿って連結度が 1 の枝とそれに連結する節点からなる部分グラフをまとめて，グループ化できる．*ov* 記述 1 で指定した ObjectView の中心実体から連なる実体と関連の構造を，対象のアプリケーションロジックを踏まえて単純化できる．

> ***ov* 記述 3** グループ化された G の部品化（コンポジション）と多態性（ポリモーフィズム）を実現するための，構造化リテラル[4] とインタフェース[5] による属性のグループ化指定

ObjectView では，構造化リテラル[4]を開発者が定義するリテラルとして，グループ化された G の部品化に利用できる．例えば (金額，通貨名) というリテラルの組を,「国際通貨」という新たな構造化リテラルとして定義し，利用できる．

ObjectView では，インタフェース[5]を有向グラフ G とは独立に定義し，*ov* 記述 2 でグループ化指定された節点に適用できる．インタフェースは，永続化クラスに複数種類のふるまいを規定する多態性を実現する．インタフェースは，以下の二つの目的に利用できる．

- 複数の永続化クラスがもつ同一のふるまいに名称をつけ，同じ型として扱う．
- 永続化クラスの getter や setter の使用に制限をつける．

なおグループ化指定された節点は，複数のインタフェースをもつことができる．

以上，*ov* 記述 1，2，3 に示した ObjectView を入力として，DBPowder では複雑な対応関係を実現した永続化クラス cpc_n と RDB アクセスコードを生成する．

ORM 手順 5 *ov* 記述 1，2，3 に示した ObjectView により，DBPowder は複雑な対応関係を実現した永続化クラス cpc_n を生成する．この cpc_n と ORM 手順 2 で生成済の rs により，複雑な対応関係による ORM が成立する．

ObjectView の有向グラフ G において，*ov* 記述 2 でグループ化された節点 N ごとに，*ov* 記述 3 で指定された構造化リテラルとインタフェースを反映させ，DBPowder は永続化クラス cpc_n を生成する．各々の生成されるクラ

[4] リテラルの集合を新たに一つのリテラルとして定義したリテラル（3.1 節）．各々の要素リテラルは要素名をもつ．

[5] オブジェクト型のふるまいのみを定義した仕様（3.1 節）．

ス c において，c に対応するグループ化された節点がもつ全ての実体の属性がまとめられ，c の属性となる．但し構造化リテラル sl が指定された場合は，sl に対応する属性は sl のメンバ属性となる．なお sl が指定されてかつ，sl に対応する属性が c 上に見つからない場合は，エラーとなり ORM は成立しない．

また DBPowder は，生成済の rs とここで生成された cpc_n の対応をとるよう，RDB アクセスコードを生成する．

グループ化された節点 N に対応する永続化クラスの識別子 ID には，グループを構成する部分グラフの根 n_r に対応する永続化クラスの識別子が採用される．節点 N 内の対応する実体はすべて，n_r からみて連結度が 1 であるため，識別子 ID は，グループ内のすべての属性を識別できる．

ORM 手順 5 で実現される ORM において，ov 記述 2 により，(a) あるテーブルを基準にして多対 1 や 1 対 1 で結合される複数のテーブルを一つの永続化クラスに対応づける対応関係をサポートできる．ov 記述 3 で永続化クラスが部品化されることにより，(b) テーブル内にある属性のうち一部を永続化クラス内でまとめて一つの構造化リテラルとして対応づける対応関係をサポートできる．また ov 記述 3 で永続化クラスが多態性をもつことにより，(c) 複数のテーブルに分散する各々の属性の集合を一種類のインタフェースとして対応づける対応関係をサポートできる．

4.3.3 DBPowder-mdl による複雑な対応関係の実現（導入）

図 4.4-1 と図 4.4-2 に対応する DBPowder-mdl の記述例を，図 4.5 に示す．本節では，DBPowder-mdl による複雑な対応関係の実現について，導入を行う．なお DBPowder-mdl の詳細な説明は 4.4 節 で行う．

図 4.5 のうち指し示されている記述例は，始点 user，節点 user,register，host，グループ化される節点 host，およびそれらを接続する枝である．DBPowder-mdl では以下のように記述する．

- 節点を，［ ］囲みで記述する．
- 枝を，節点間のインデント関係により表す．
- インデントの最上層の節点は，始点となる．
- グループ化される節点を，｛ ｝囲みで記述する．
- 節点名で @ をつけない場合，*eer* で生成済の永続化クラスの定義に追加される形で ov_n の定義が反映される．
- 節点名で @ をつける場合，@ の前にクラス名，@ の後に実体名を指定する．このときに指定したクラスは，新たな永続化クラスとして定義される．

図 4.5 の例では，始点の user の指定では @ を用いておらず，クラス名を指定していないので，*eer* で生成済の User クラスに ObjectView の定義が追加される．節点の register ではクラス名 RegHost を指定しているので，新たに RegHost クラスが生成される．節点 host では上の階層の節点とのグループ化を指定しているので，生成される RegHost クラスは，実体 register と host が結合したクラスとして定義される．

4.3.4 複雑な対応関係の実現例（user – register – host スキーマ）

複雑な対応関係の実現例（user – register – host スキーマ）を 図 4.4 に示す．ここで 図 4.4-1 は，開発者が設計した ObjectView ov_1 と ov_2 であり，図 4.4-2

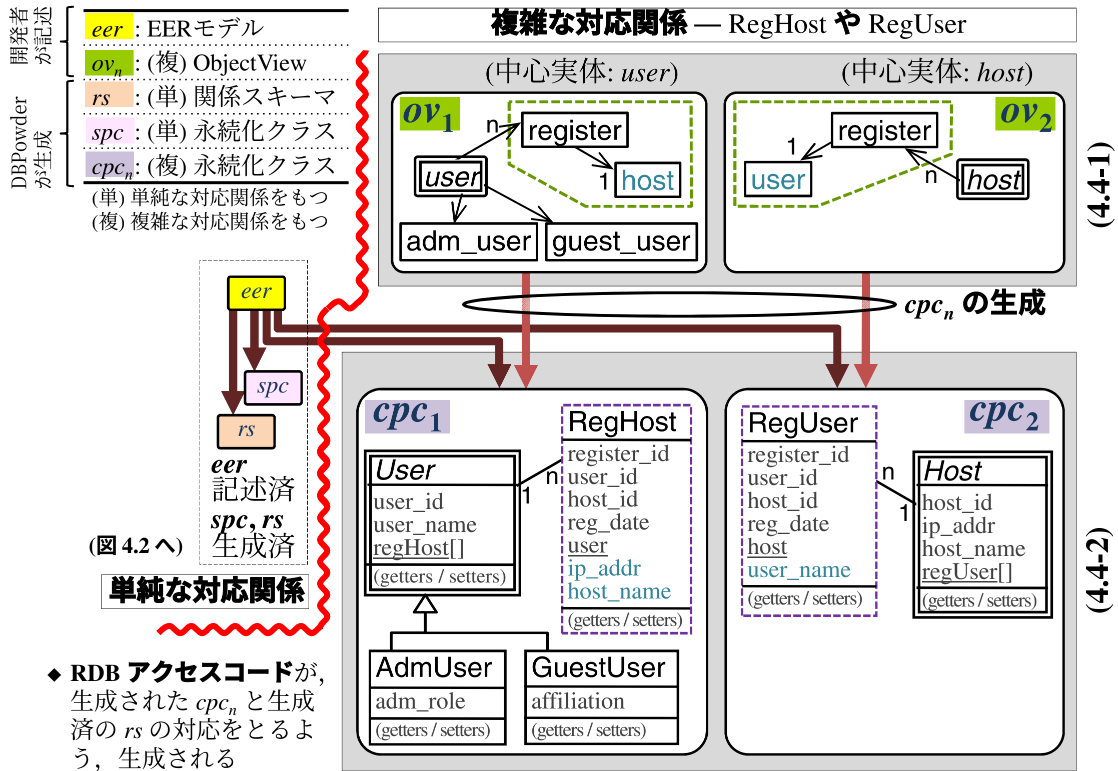


図 4.4: DBPowder における複雑な対応関係の実現例（user – register – host スキーマ）．(4.4-1) ov_n, (4.4-2) cpc_n．eer は図 4.2 で記述済，rs と spc は図 4.2 で生成済．ov_1を記述 + (記述済 eer) → cpc_1が生成される．ov_2を記述 + (記述済 eer) → cpc_2が生成される．ov_n 内の緑色で囲まれた領域は，グループ化の対象であることを示す．

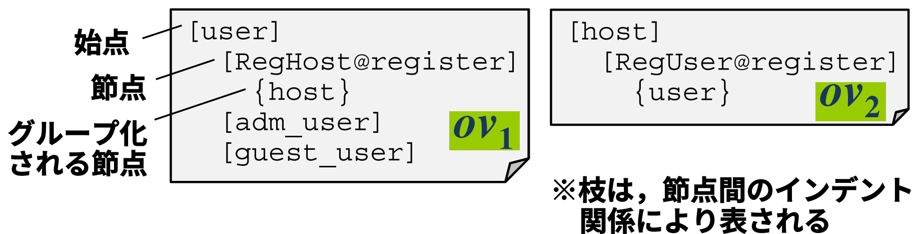


図 4.5: DBPowder-mdl の記述例：図 4.4 の ObjectView ov_1 と ov_2

は ObjectView から生成された複雑な対応関係をもつ永続化クラス cpc_1 と cpc_2 である．また 図 4.5 は，4.3.3 節 で導入した DBPowder-mdl での記述である．ここでは，*eer* は図 4.2-1 で記述済みであり，*rs* と *spc* は図 4.2-2 で生成済みであるとする．よって，*rs* と *spc* は単純な対応関係をもつ．

図 4.4-1 で，ov_1 では `user` を中心実体として指定している．節点 `register` と `host` は，枝の方向を考えると連結度が 1 なのでグループ化できる．こうして構成される ObjectView ov_1 は，管理者 `user` を中心としたアプリケーションロジックを開発者に提供する．図 4.4-1 上の ov_2 は，`host` を中心実体とした別の例である．この例では，節点 `register` と `user` をグループ化し，汎化階層の子に属する実体 `adm_user` と `guest_user` を必要としない．こうして構成される ObjectView ov_2 は，ホスト `host` を中心としたアプリケーションロジックを開発者に提供する．

つぎに，構成された ObjectView ov_1，ov_2 が生成する永続化クラスをみる．図 4.4-2 では，開発者が記述した ov_1 を用いて DBPowder が `User` と `RegHost` から構成される永続化クラス cpc_1 を生成した例と，同様に ov_2 を用いて `Host` と `RegUser` から構成される永続化クラス cpc_2 を生成した例を示す．

永続化クラス cpc_1 は，ov_1 の記述に従って `User` を中心としたクラス構成となり，ov_1 上の節点 `register` と `host` はクラス `RegHost` にまとめられる．もとの *eer* の属性はそれぞれ，ov_1 で構成された節点とグループに基づいて cpc_1 上の属性となる．たとえば永続化クラス `RegHost` 上の属性は，実体 `register` と `host` の属性の両方をもつ．同様にして cpc_2 は，`Host` を中心としたクラス構成となり，ov_2 上の節点 `register` と `user` はクラス `RegUser` にまとめられる．属性の構成方法も cpc_1 と同様である．クラス `RegHost` や `RegUser` のようにグループ化された節点から生成される永続化クラスは，生成済の *rs* 上に対応するテーブルを複数個もつため，*rs* とは複雑な対応関係をもつ．

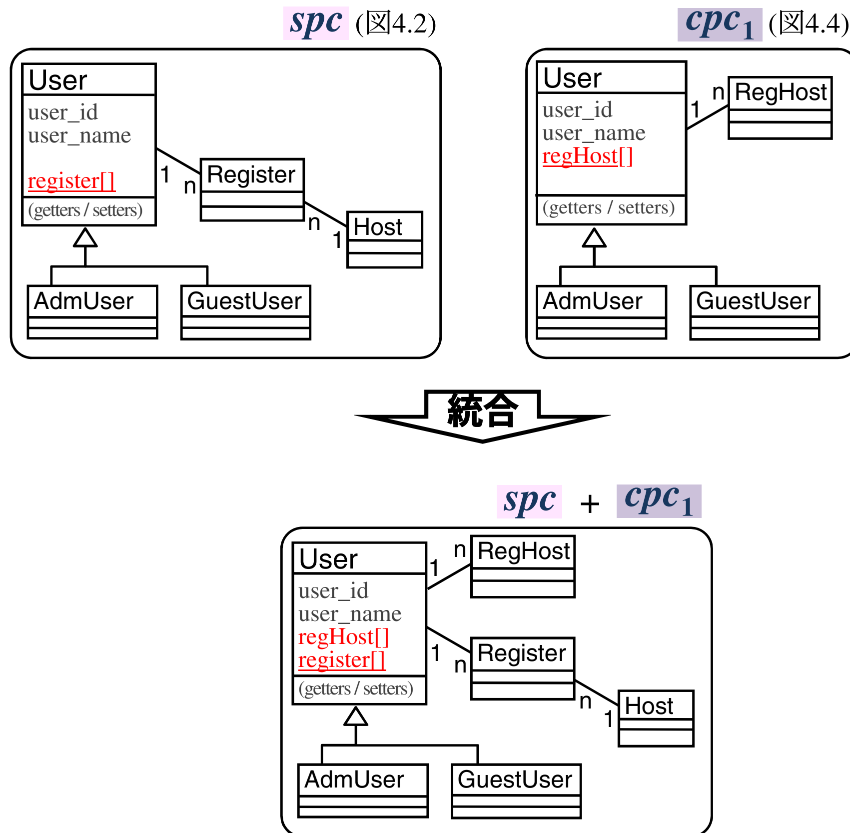


図 4.6: 永続化クラス User の統合例

生成された永続化クラス cpc_1 は，管理者 User および，これと 1:n で関連をもつ登録されたホスト RegHost から構成される．RegHost はホスト情報のほか，管理者がそのホストを登録した日付 reg_date を属性としてもつ．このように cpc_1 は，管理者 User を中心としたアプリケーションロジックを開発者に提供している．同様にして cpc_2 は，ホスト Host および，これと 1:n で関連をもつ登録された管理者 RegUser から構成される．RegUser は管理者情報のほか，ホストの管理者となった日付 reg_date を属性としてもつ．このように cpc_2 は，ホスト Host を中心としたアプリケーションロジックを開発者に提供

している．

ここで生成された cpc_1 上の永続化クラス User や cpc_2 上の永続化クラス Host は，spc にも存在する．このような永続化クラスは，定義がまとめられて統合される．図 4.6 に永続化クラス User の統合例を示す．ここでは，図 4.2 で示した spc におけるクラス User と図 4.4 で示した cpc_1 におけるクラス User が，両方の定義を満たす新たなクラス User に統合される．図 4.4 では，永続化クラス Host でも同様の統合が発生する．

以上に示した，アプリケーションロジックの有向グラフによる記述や，記述した有向グラフに対応する永続化クラスの生成や，永続化クラスともとの関係スキーマとの複雑な対応関係を，DBPowder-mdl ではコンパクトに記述できる．図 4.5 にその内容を示す．

4.3.5 複雑な対応関係の実現例（6.1 節 KEKapp の一部）

図 4.7 に，6.1 節 で紹介する KEKapp からスキーマの一部を抜粋したものを示す．図 4.8 は，これの DBPowder-mdl による記述例である．

KEKapp アプリケーションでは，各種の管理タスクが生じる．その一例として，管理者 user ごとのレポート管理がある．管理者 user は，自らが管理するホスト host ごとにセキュリティレポート sec_report を提出する．このホスト host は，複数の管理者 user によって管理されることがあり，その全員がレポートを提出するわけではない[6]．管理タスクはその内容に応じて，管理者が属する組織 division や大組織 division_category ごとにまとめられる．

[6] ここで，アプリケーションロジックが扱っている管理者と，同じホストを管理する別の管理者があらわれる．ov_1 や ov_2 では，節点 user に加えて user' をもうけることで，この状況に対応している．

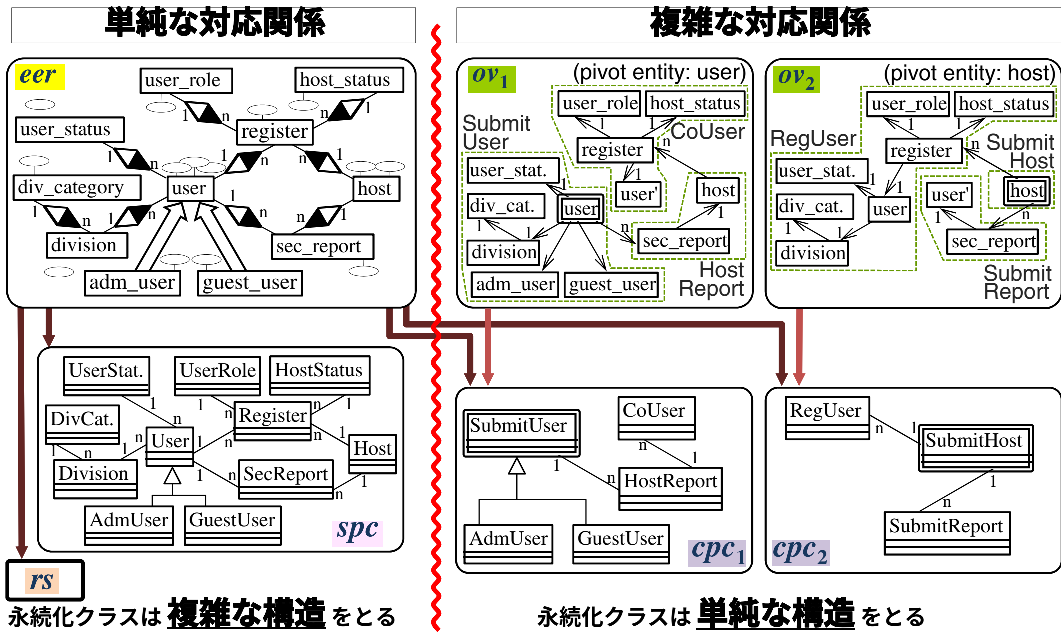
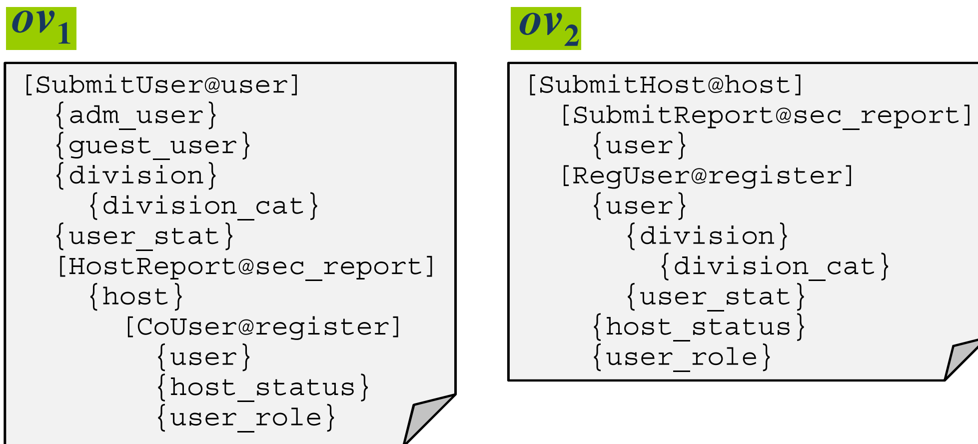


図 4.7: DBPowder における複雑な対応関係の実現例（6.1 節 KEKapp の一部）; ov_n 内の緑色の点線で囲まれた領域は，グループ化の対象であることを示す．



ov_1

```
[SubmitUser@user]
  {adm_user}
  {guest_user}
  {division}
    {division_cat}
  {user_stat}
  [HostReport@sec_report]
    {host}
      [CoUser@register]
        {user}
        {host_status}
        {user_role}
```

ov_2

```
[SubmitHost@host]
  [SubmitReport@sec_report]
    {user}
  [RegUser@register]
    {user}
      {division}
        {division_cat}
      {user_stat}
    {host_status}
    {user_role}
```

図 4.8: DBPowder-mdl の記述例：図 4.7 の ObjectView ov_1 と ov_2

このような状況下で発生する管理タスクは，しばしば複雑なものとなる．

開発者はこのようなデータ構造を *eer* で捉えたのちに，DBPowder をもちいて関係スキーマ *rs* と単純な対応関係に基づく永続化クラス *spc* を生成する．この永続化クラス *spc* は様々なアプリケーションロジック上で利用できるが，11 つのクラスから構成されており，その構造は多くのアプリケーションロジックにとって複雑すぎる．たとえば，組織や大組織を主題とするケース以外では，`Division` や `DivisionCategory` は管理者 `User` の属性として扱われれば十分な場合が多い．しかし *spc* を使用した場合には，組織や大組織が主題ではないケースでも，クラス `User` → `Division` → `DivisionCategory` という走査が必要になる．よって，このデータ構造を常に意識する必要があり，開発の負担が増大する．

ObjectView は，場面ごとのアプリケーションロジックに着目して，それぞれの場面に適した永続化クラスを簡易に生成できる．たとえば中心実体を `user` として ObjectView ov_1 を記述することで生成される永続化クラス cpc_1 は，管理者を主題としたアプリケーションロジックを扱うのに適した永続化クラスであり，これを構成するクラスは五つである．また，中心実体を `host` として ObjectView ov_2 を記述することで生成される永続化クラス cpc_2 は，ホストを主題としたアプリケーションロジックを扱うのに適した永続化クラスであり，これを構成するクラスは三つである．いずれも，11 つのクラスから構成される *spc* と比べて十分な簡略化ができている．たとえば ov_1 や ov_2 では組織や大組織は主題ではないため，これに関係する属性は管理者 `User` の属性として扱われる．したがって，`user` → `division` → `division_category` というデータ構造を意識する必要がない．

以上に示したアプリケーションロジックの有向グラフによる記述や，記述した有向グラフに対応する永続化クラスの生成や，永続化クラスともとの関係ス

キーマとの複雑な対応関係を，図 4.8 に示す DBPowder-mdl ではコンパクトに記述できている．

4.4 記述言語 DBPowder-mdl の定義

DBPowder は記述言語 DBPowder-mdl をもつ．DBPowder-mdl については 4.2.2 節（図 4.3）と 4.3.3 節（図 4.5）で例示による導入を行った．本節では DBPowder-mdl の定義を与える．4.4.1 節 では DBPowder-mdl の言語としての形式的な定義を与え，4.4.2 節 では 4.4.3 節 では 4.4.4 節 では 4.4.5 節 では 4.4.6 節 では DBPowder-mdl の実体名表記によるクラス名やテーブル名などの名称の決定方法を示し，4.4.7 節 では DBPowder-mdl における実体パラメータの決定方法を示す.

4.4.1 形式的な定義

DBPowder-mdl は，DBPowder における EER モデルや ObjectView を記述するための言語である．DBPowder-mdl の主要要素は，EER モデルの記述で使用する実体 E，属性 A，関連実体 L，および，ObjectView の記述で使用する始点 PN，節点 N，グループ化される節点 MN であり，階層構造を用いてこれらの関係を記述する．

EER モデルの記述では，階層の最上位に実体 E をとり，E が保持する属性 A を E の一段下に記述する．E と関連 R をもつ実体 E' を E の一段下に記述し，E と E' にある階層の上下関係をもって，関連 R を表現する．連結度などの R のパラメータは E' 上に記述する．このような実体 E' と関連 R の組み合

わせを，DBPowder-mdl では関連実体 L とよぶ．なおここでは，E と E' の上下関係における順序は意味をもたない．すなわち，E' と E の上下関係を交換し，関連 R の連結度を考慮して交換した E' と E につなぎ直し，E' を実体，E を関連実体とした記述は，もとの記述と等価である．なお関連実体 L は，自らまたは他の関連実体を階層の下位にもつことが可能である．

ObjectView の記述では，階層の最上位に始点 PN をとる．PN と枝 Ed で結ばれた節点 N については，PN の一段下に N を記述する．PN と N にある階層の上下関係をもって枝 Ed を表現する．なおこの Ed に対応する EER モデルの関連が，別途記述されている必要がある．EER モデルの記述と異なり，PN と N の上下関係における順序は，枝の方向を表す．なお節点 N は，自らまたは他の節点を階層の下位にもつことが可能である．枝 Ed の先の連結度が 1 であって，枝の先に連結された節点をグループ化する場合には，節点 N のかわりにグループ化節点 MN を用いる．

階層内において，Ed，L，PN，N，MN については同じ定義を複数回登場させることができる．この場合，各々の定義は統合され，一つの定義として認識される．これにより DBPowder-mdl ではグラフ表現を扱うことができる．

以上の定義による DBPowder-mdl は，3.3 節 と 4.2 節で示した EER モデルによる単純な対応関係および，4.3 節で示した ObjectView による複雑な対応関係を実現できる．

4.4.2 EER 宣言部の構文

DBPowder-mdl の構文は，EER モデルを記述する EER 宣言部と ObjectView を記述する ObjectView 宣言部を備える．本節では EER 宣言部について述べる．

```
 1: <name> ::= {[A-Za-z][A-Za-z0-9_]*}
 2: <indentA> ::= {[ ]+}    # 属性の，所属する実体を示すインデント
 3: <indentL> ::= {[ ]*}    # 関連実体の，リンク先実体を示すインデント
==============================================================================
# 実体宣言(E)
 4: <EntityDef> ::= [<entityDesc> [<entityParam>] ]
# 関連実体宣言(L)
 5: <RelshipEntityDef>
      ::= <indentL>[<cardinality>|<inheritType> <entityDesc> [<entityParam>]]
 6:   <cardinality> ::= <1:1> | <1:n> | <n:1> | <n:n>              # 連結度表記
 7:   <inheritType> ::= inherit-sr | inherit-cr | inherit-ccr      # 汎化階層表記
  # 実体名表記
 8:   <entityDesc> ::= <commonEntityName>[<interfaceDesc>]
                     | <className>[<interfaceDesc>]@<tableName>
 9:     <commonEntityName>,<className>,<tableName> ::= <name>
10:     <interfaceDesc> ::= !<interfaceName>[<interfaceDesc>]
11:       <interfaceName> ::= <name>
  # 実体パラメータ表記
12:   <entityParam> ::= [ importDB ][<pKeysDef>][<joinOnDef>][<relshipNameDef>]
                        [<defaultPkeyTypeDef>][<throughJoinOnDef>][<throughDef>]
13:     <pKeysDef> ::= pkeys="<pKeysList>"
14:       <pKeysList> ::= <pKeysItem>[ ,<pKeysList> ]
15:     <joinOnDef> ::= joinOn="<joinOnList>[ = <joinOnList>]"
16:       <joinOnList> ::= <joinOnItem>[ , <joinOnList>]
17:     <pKeysItem>, <joinOnItem> ::= <name>
18:     <relshipNameDef> ::= $<name>[/<name>]  # 一つ目は順方向関連，二つ目は逆方向関連
19:     <defaultPkeyTypeDef> ::= defaultPkeyType="<typeName>"
20:     <throughJoinOnDef> ::= throughJoinOn="<joinOnList>[ = <joinOnList>]"
21:     <throughDef> ::= through="<tableName>"
==============================================================================
# 属性宣言(A)
22: <AttributeDef> ::= <indentA><attrDesc> <typeName> [<extDef>]
23:   <attrDesc> ::= <commonAttrName> | <classAttrName>@<columnName> # 属性名表記
24:     <commonAttrName>,<classAttrName>,<columnName> ::= <name>
25:   <typeName> ::= int | text{[0-9]+} | number{[0-9]+} | datetime
                                                   ( <typeName> は，今後追加予定)
26: <extDef> ::= [<extDef>] [ not null | readOnly | unique | default "{.+}" ]
27: <AttributeDefs> ::= <AttributeDef>[,<AttributeDefs>]
```

凡例(BNF**非標準記法**)　　{RE}　RE **は正規表現**　　L　L**は文字列**

図 4.9: DBPowder-mdl の BNF 記法による文法（EER 宣言部）

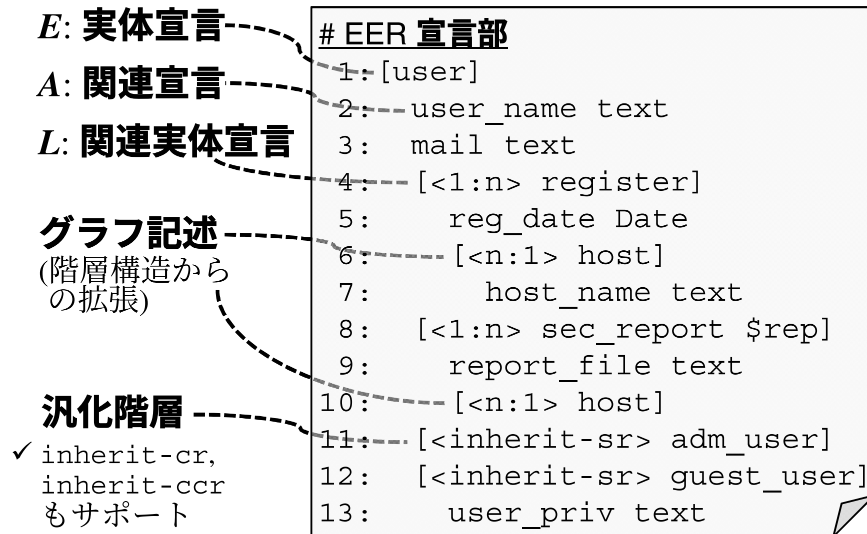


図 4.10: DBPowder-mdl の記述例：EER 宣言部

BNF 記法による文法を 図 4.9 に示し，図 4.10 に EER モデルを記述する例を示す．

EER 宣言部は，実体宣言<EntityDef>，関連実体宣言<RelsihpEntityDef>，および属性宣言<AttributeDef>から構成される．実体宣言を階層の最上位に記述する．関連実体 L と実体 E が関連をもつとき，関連実体宣言を実体宣言の一段下に記述する．関連実体 L' と関連実体 L が関連をもつとき，関連実体 L' の宣言を L の宣言の一段下に記述する．実体 E や関連実体 L が属性 A をもつとき，属性 A の宣言を E や L の一段下に記述する．

実体宣言<EntityDef> は，実体名表記<entityDesc> と実体パラメータ表記 <entityParam>から構成される．実体名表記や実体パラメータでは設定より規約（CoC[7]）が定められており，DBPowder の命名規約に従うならば実体名を記述するのみで良い一方で，詳細な指定も可能である．たとえば関連名表

[7] 2.2.1 節 および文献 [33, 22] 参照

記<relshipNameDef>について，順方向の関連名と逆方向の関連名を明示的に指定できる．詳細な指定については 4.4.6 節と 4.4.7 節で述べる．関連実体宣言<RelshipEntityDef> は実体宣言に加え，連結度表記<cardinality> または汎化階層表記<inheritType>のいずれか片方をもつ．連結度表記や汎化階層表記では，一段上の階層にある実体または関連実体に対する連結度や汎化階層の *rs* における扱い（ORM 手順 4 – 4.2.1 節）を記述する．連結度表記は<1:1>, <1:n>, <n:1>, <n:n> のいずれかであり，汎化階層表記は inherit-sr, inherit-cr, inherit-ccr のいずれかである．

属性宣言<AttributeDef> は，属性名表記<attrDesc> ，属性型表記<typeName>，およびオプション表記<extDef>から構成される．属性名表記もまた CoC が定められており，DBPowder の命名規約に従うならば属性名を記述するのみで良い一方で，詳細な指定も可能である．属性名表記詳細な指定については 4.4.6 節で述べる．属性型表記では属性の型を記述する．オプション表記では属性につけるオプションを記述する．ここでは複数のオプションをつけることができる．

実体宣言<EntityDef>において階層構造では不足する箇所では，実体名表記<entityDesc> にて同一名称を複数回記述することで，グラフ構造を記述できる．

4.4.3 EER 宣言部の例

図4.10 はEER 宣言部の例であり，図4.7 の *eer* をもとに，実体user, register, host, sec_report, host, adm_user, guest_user と，これらの実体がもつ属性や関連を記述したものである．1 行目が実体宣言の例，4 行目や 6 行目などが関連実体宣言の例，2 行目や 5 行目などが属性宣言の例である．実体 user

はuser_nameやmailを属性として保持し，registerなどを関連実体として保持する．関連実体registerはreg_dateを属性として保持し，hostを関連実体として保持する．また8行目では，実体userとsec_reportの関連名に明示的にrepと指定している．明示的に指定することで，自己参照や同じ実体への複数種類の関連などを柔軟に扱うことができる．なお明示的に指定しなかった場合は4.4.7節 に示されるデフォルト値が採用され，この例では関連名はsecReportとなる．

EER宣言部では，関連実体を導入した上で基本要素を階層構造で記述することにより，EERモデルの記述の際に同じ名称を指し示す回数を減らしており，結果として簡潔な記述を可能としている．但しデータ表現能力が実体の階層構造では不足する箇所では，図4.10の6行目と10行目に示すように，実体Eや関連実体Lの同一要素名を複数回記述することで，グラフ構造を記述できる．また11行目と12行目に示すように，汎化階層の関係スキーマでの扱いをサポートする．例はSRであり，左側の注釈の通りCRとCCRもサポートする．

4.4.4 ObjectView宣言部の構文

本節では4.4.2節 のEER宣言部に引き続き，ObjectView宣言部について述べる．BNF記法による文法を 図4.11に示し，ObjectViewを記述する例を 図4.12に示す.

ObjectView宣言部は，始点宣言<PivotNodeDef>，節点宣言<NodeDef>，グループ化節点宣言 <GroupedNodeDef>，構造化リテラル宣言 <StructuredLiteralDef>，およびインタフェース宣言<InterfaceDef>から構成される．始点宣言を階層の最上位に記述する．始点宣言からインデントを

```
 1: <name> ::= {[A-Za-z][A-Za-z0-9_]*}
 2: <indentA> ::= {[ ]+}    # 属性の，所属する構造化リテラルやグループ化節点を示すインデント
 3: <indentL> ::= {[ ]*}    # 節点やグループ化節点の，リンク先の節点を示すインデント
==============================================================================
# 始点宣言(PN)              – 中心実体に対応
 4: <PivotNodeDef> ::= [<nodeDesc> [<nodeParam>] ]
# 節点宣言(N)               – 実体に対応
 5: <NodeDef> ::= <indentL>[<nodeDesc> [<nodeParam>] ]
# グループ化節点宣言(GN)    – 実体に対応
 6: <GroupedNodeDef> ::= <indentL>{<nodeDesc> [<nodeParam>] }
  # 節点名表記
 7:   <nodeDesc> ::= <entityDesc>  # 図4.9 EER宣言部を参照
  # 節点パラメータ表記
 8:   <nodeParam> ::= [<relshipNameDefOV>][<throughJoinOnDef>][<throughDef>]
 9:     <relshipNameDefOV> ::= $<name>   # 順方向関連のみ
 ##     <throughJoinOnDef>, <throughDef> は，図4.9 EER宣言部を参照
==============================================================================
# 構造化リテラル宣言(SL)
10: <StructuredLiteralDef> ::= [<indentL>]( <slName> < relshipNameDefOV > )
11:   <slName> ::= <name>       # 構造化リテラル名
  # 構造化リテラルの属性宣言
12:   <StructuredLiteralAttributeDef>
          ::= <indentA><classAttrName>[@<columnName>] [<typeName>]
 ##       <classAttrName>, <columnName>, <typeName> は，図4.9 EER宣言部を参照
# インタフェース宣言(IF)
13: <InterfaceDef> ::= [[ <interfaceName> ]]
  # インタフェースの属性宣言
14:   <InterfaceAttributeDef> ::= <indentA><classAttrName> <typeName>
```

凡例(BNF非標準記法)　　{RE} RE は正規表現　　L Lは文字列

図 4.11: DBPowder-mdl の BNF 記法による文法（ObjectView 宣言部）

下げて，節点宣言やグループ化節点宣言を記述する．インデントの上下関係は，始点から節点やグループ化節点を走査する経路を表す．

構造化リテラル宣言は，始点宣言，節点宣言，グループ化節点宣言（以下，この三つをあわせて節点類宣言とよぶ）によって利用され，利用する宣言からインデントを一段下げて記述する．インタフェース宣言は階層の最上位に記述

PN : **始点宣言**（中心実体に対応）
class-assigned-name@EER-assigned-name
N: **節点宣言**（実体に対応）
GN: **グループ化節点宣言**

```
# ObjectView 宣言部
1:[SubmitUser@user]
2:  [HostReport@sec_report $rep]
3:    {host}
4:      [CoUser@register]
5:        {user}
6:  {adm_user}
7:  {guest_user}
```

図 4.12: DBPowder-mdl の記述例：ObjectView 宣言部

する．記述済のインタフェース宣言は，始点やグループ化節点を含む節点や，他の実体（EER 宣言部）によって利用される．構造化リテラル宣言とインタフェース宣言は，それぞれの属性宣言 <StructuredLiteralAttributeDef> と <InterfaceAttributeDef> をもつ．

ObjectView の始点宣言は EER の実体宣言と類似しており，節点宣言とグループ化節点は関連実体宣言と類似しているが，ObjectView 宣言部は EER 宣言部で宣言済の実体や関連を参照することを目的とする点が異なる．また，構造化リテラル宣言とインタフェース宣言は ObjectView 宣言部のみが備える．

始点宣言<PivotNodeDef>，節点宣言<NodeDef>，およびグループ化節点宣言<GroupedNodeDef>は，節点名表記<nodeDesc> と節点パラメータ表記<nodeParam>から構成される．節点名表記や節点パラメータ表記では，EER 宣言と同様に CoC が定められており，DBPowder の命名規約に従うならば節点名を記述するのみで良い一方で，詳細な指定も可能である．詳細な指定につい

ては 4.4.6 節と 4.4.7 節で述べる．

構造化リテラル宣言<StructuredLiteralDef>は構造化リテラルの属性宣言<StructuredLiteralAttributeDef>をもち，構造化リテラルがまとめる対象の属性を指定する．構造化リテラル *sl* の属性宣言では，*sl* がまとめるオブジェクトモデルの属性名<classAttrName>と，<classAttrName>が扱う EER モデル上の属性名<columnName>を対にして指定する．構造化リテラル宣言を単独で階層の最上位に記述する場合は，宣言のみが行われる．他の節点類宣言から一段下げて記述する場合は，対象の節点類宣言が実体を通じて与えられる属性をまとめる役割を果たす．

インタフェース宣言<InterfaceDef>はインタフェースの属性宣言<InterfaceAttributeDef>をもち，<InterfaceDef>において多態性を扱う対象の属性を指定する．インタフェースは，節点類や EER 宣言部の実体や関連実体によって用いられる．

4.4.5 ObjectView 宣言部の例

図 4.12 は ObjectView 宣言部の例であり，図 4.7 の ov_1 をもとに，中心実体に対応する始点宣言 `user` と，実体に対応する節点宣言 `sec_report` や `register` と，実体に対応するグループ化節点宣言 `host` および，これらの節点類がもつ枝を記述したものである．1 行目が始点宣言，2 行目や 4 行目などが節点宣言，3 行目や 5 行目などがグループ化節点宣言である．また 6 行目と 7 行目は，汎化階層に属する節点の宣言となる．

この例では，`SubmitUser`，`HostReport`，`CoUser` という三つの永続化クラスのグループが宣言されている．グループ化節点宣言にあたる `host`，`user`，

adm_user，guest_user という四つの節点に対応する実体は，三つの永続化クラスのグループにまとめられる．

ObjectView 宣言部では，参照する EER 宣言部での実体や属性の設定を引き継ぐため，EER 宣言部における連結度表記や汎化階層表記にあたる記述を必要としない．但しスキーマの構造が複雑になる場合などは，関連名を明示的に指定する必要が生じることがある．2 行目では，図 4.10 の EER 宣言部の例と同様，関連名を明示的に rep と指定している．明示的に指定することで，自己参照や同じ実体への複数種類の関連などを柔軟に扱うことができる．

4.4.6　CoC の活用：実体名表記による名称の決定方法

DBPowder-mdl における，クラス名，テーブル名，プロパティ名，属性名，getter/setter 名には，2.2.1 節 に示した設定より規約（CoC）の考えに基づく命名規則が定められている．表4.2 に DBPowder-mdl の CoC による名称の決定方法を示す．クラス名，テーブル名，およびクラスやテーブル内の属性名の各々についてこの CoC に従える場合は，単語を一つ指定すれば名称は自動的に決定される．

CoC に従えない箇所については，開発者が明示的に指定することで名称を決定する．DBPowder-mdl における（クラス名，インタフェース名，テーブル名）の明示的な指定は，EER モデルの実体や ObjectView の節点の記述にて，下記に示す書式により行う．但し，インタフェースを利用しない場合はインタフェース名を指定しない．

```
（クラス名，インタフェース名，テーブル名）の明示的な指定
ClassName!InterfaceName@table_name
```

内容	例
DBPowder の実体名や属性名において，_（アンダースコア）または小文字大文字と続く文字がある場合，DBPowder では複数の単語をつなげた名前を指定したと認識される．	user_name や userName は，user と nameという 2 つの単語をつなげた名前と認識される．
クラス名は，DBPowder で認識された各々の単語について，頭文字を大文字に変換し，それをつなげた名前とする（CamelStyle）	guest_user や GuestUser という実体の，クラス名は GuestUser.
テーブル名，プロパティ名，属性名は，DBPowder で認識された各々の単語を _ でつなげ，頭文字を小文字に変換した名前とする（under_score_style）	guest_user や GuestUser という実体の，テーブル名は guest_user.
getter/setter 名は，CamelStyle 名に get や set という接頭語をつなげた名前となる．	user_name の setter 名は setUserName.

表 4.2: DBPowder-mdl における，設定より規約（CoC）による名称の決定方法

パラメータ名	デフォルト値	内容
importDB	false	指定されたテーブル定義がもつ属性を読み込み，自らの実体の属性とする．
pkeys	実体名_id	主キー名を指定．カンマ区切りにすると複合主キーとなる．属性定義に指定した主キー名が無い場合は，DBPowder が代理キーを補う．
defaultPkeyType	int	主キーの型を指定する．但し，属性定義で別途指定される場合は，この設定値は用いられない．
joinOn	1:n, n:1 のときは1 側の主キー名．1:1, n:m については別途	結合時に用いるキーを指定する．"一階層上のキーリスト＝自階層のキーリスト" と指定．名称が同じ場合は，一階層上のキーリストを省略できる．多対多の時は，結合テーブルと自階層について指定する
throughJoinOn	一階層上の実体の主キー名	多対多指定したときに，一階層上の実体と結合テーブルについて，結合時に用いるキーを指定する．
through	一階層上のテーブル名_自階層のテーブル名	多対多指定したときの，結合テーブルの名前を指定．
relshipName	クラス名の頭文字小文字	関連名を指定．なお，relshipName=“relname” という指定を，$relname と短縮記述できる．

表 4.3: DBPowder-mdl の実体で指定できるパラメータと，そのデフォルト値

また，DBPowder-mdl において属性名の明示的な指定は,（クラス内の属性名，テーブル内の属性名）の組による以下の書式をとる.

（クラス内の属性名，テーブル内の属性名）の明示的な指定
`classAttributeName@table_attribute_name`

4.4.7 CoC の活用：実体パラメータの決定方法

DBPowder-mdl の実体で指定できるパラメータと，そのデフォルト値を，表 4.3 に示す．ここに示した全てのパラメータについてデフォルト値が定められており，指定は必須ではない．開発者がこれらのパラメータを指定すれば，様々な永続化クラスや関係スキーマを記述でき，また二者の様々な対応関係を記述できる.

なお 表 4.3 のうち，`importDB = "true"` を指定すると，DBPowder では指定されたテーブルがもつ属性定義を，構築済の関係スキーマから読み込み，自らの実体の属性定義にできる．この機能は，ORM 手法の bottom-up マッピングアプローチ（2.2.1 節）をサポートする.

4.4.8 オブジェクトモデルの継承を関係モデルで扱う記述例と他の ORM フレームワークでの記述例

本節では DBPowder-mdl の例として，オブジェクトモデルの継承を関係モデルで扱う記述例を示す．また，同等の記述を Hibernate や Ruby on Rails (RoR) で実施する例も示す.

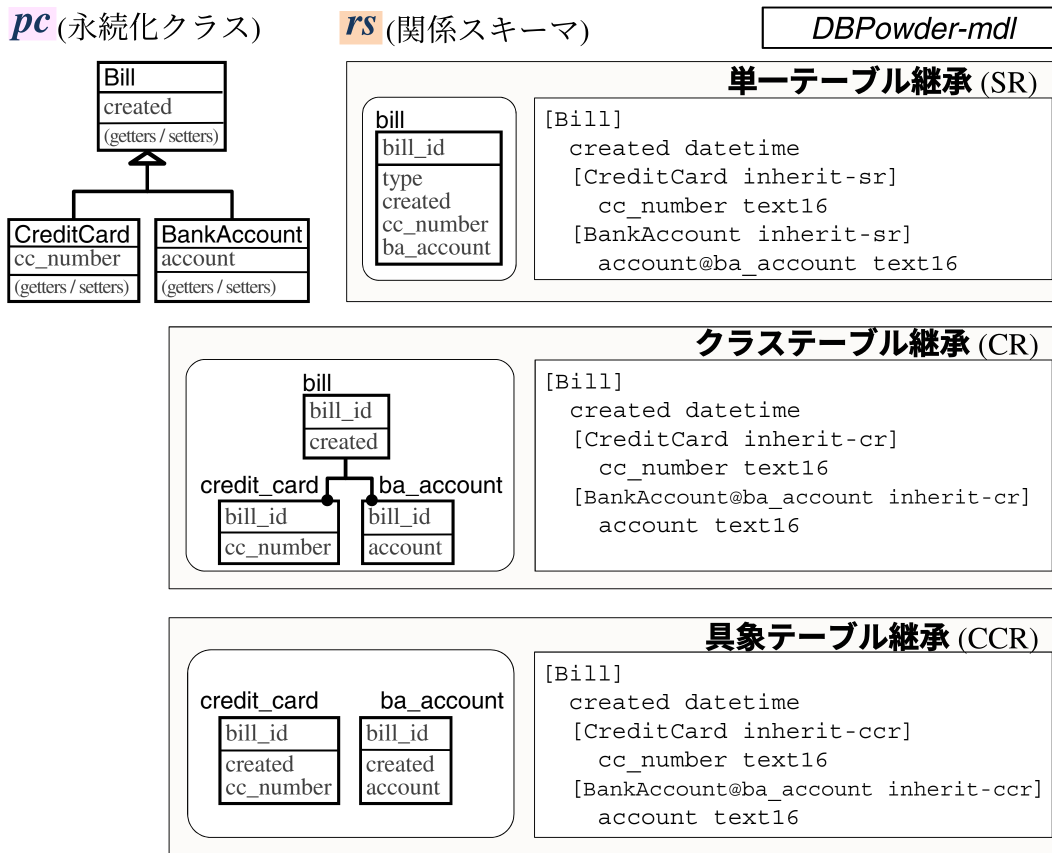


図 4.13: DBPowder-mdl の記述例：オブジェクトモデルの継承を関係モデルで扱う例

図4.13 に，DBPowder でオブジェクトモデルの継承を関係モデルで扱う例を示す．図の左側が永続化クラス，図の中央が関係スキーマ，図の右側が DBPowder-mdl による ORM の記述である．DBPowder では `inherit-sr`，`inherit-cr`，`inherit-ccr` と指定することで，単一テーブル継承（SR），クラステーブル継承（CR），および具象テーブル継承（CCR）を実現[8]できる．

[8]単一テーブル継承，クラステーブル継承，および具象テーブル継承の詳細については，2.3.1 節を参照．

※ 永続化クラス（***pc***）と関係スキーマ（***rs***）は図 4.13 参照

単一テーブル継承 (SR)　　*Hibernate*

```
<class name="Bill" table="bill">
  <id name="bill_id" column="bill_id" type="long">
    <generator class="native"/>
  </id>
  <discriminator column="type" type="string"/>
  <property name="created" column="created"
            type="timestamp"/>
  <subclass name="CreditCard"
            discriminator-value="CC">
    <property name="cc_number" column="cc_number"
              type="string"/>
  </subclass>
  <subclass name="BankAccount"
            discriminator-value="BA">
    <property name="account" column="ba_account"
              type="string"/>
  </subclass>
</class>
```

クラステーブル継承 (CR)

```
<class name="Bill" table="bill">
  <id name="bill_id" column="bill_id" type="long">
    <generator class="native"/>
  </id>
  <property name="created" column="created"
            type="timestamp"/>
  <joined-subclass name="CreditCard"
                   table="credit_card">
    <key column="bill_id"/>
    <property name="cc_number" column="cc_number"
              type="string"/>
  </joined-subclass>
  <joined-subclass name="BankAccount"
                   table="ba_account">
    <key column="bill_id"/>
    <property name="account" column="account"
              type="string"/>
  </joined-subclass>
</class>
```

具象テーブル継承 (CCR)

```
<class name="Bill" abstract="true">
  <id name="bill_id" column="bill_id" type="long">
    <generator class="native"/>
  </id>
  <property name="created" column="created"
            type="timestamp"/>
  <union-subclass name="CreditCard"
                  table="credit_card">
    <property name="cc_number" column="cc_number"
              type="string"/>
  </union-subclass>
  <union-subclass name="BankAccount"
                  table="ba_account">
    <property name="account" column="account"
              type="string"/>
  </union-subclass>
</class>
```

Ruby on Rails (*RoR*)

```
# 関係スキーマ作成
rails g scaffold bill ¥
       type:string ¥
       created:datetime ¥
       cc_number:string ¥
       ba_account:string

# 永続化クラス作成
class CreditCard < Bill
end
class BankAccount < Bill
end

# 参考:Billクラスは, scaffold
#      実行時に自動生成される
class Bill < ActiveRecord::Base
end
```

※RoRは，CR と CCR をサポートしない．

※上記で生成されるテーブル名や主キー名は，図 4.13 の左側で示した ***pc*** や ***rs*** と一致しない．一致させるためには追加の記述が必要である．その例は以下の通り．

```
# 関係スキーマ側の処置
# SQL の ALTER TABLE 文などで，
# 以下を変更
#   テーブル名: bills -> bill
#   カラム名  : id    -> bill_id


# 永続化クラス側の処置
# 下線を引いた行を追加する
class Bill < ActiveRecord::Base
  set_table_name "bill"
  set_primary_key "bill_id"
end
```

図 4.14: 図 4.13 の Hibernate や Ruby on Rails での扱い

図 4.14 は，同様の記述を Hibernate や Ruby on Rails (RoR) で行った例である．図の左側が Hibernate の例，図の右側が RoR の例である．なお，ここで用いた永続化クラスと関係スキーマは 図 4.13 に示したものと同様である．

Hibernate と RoR のいずれも，DBPowder-mdl と比べて明らかに多くの記述を要している．Hibernate（図 4.14 左側）では ORM の記述に XML を用いているため，`property` や `name` などの，XML タグ名や XML 属性名の記述が多くなっている．また，クラス名，テーブル名，属性名，主キー名など，多くの指定を記述する必要がある．RoR（図 4.14 右側）では，Scaffold により簡易な記述を実現できているが，右上の例から生成される関係スキーマは，テーブル名や主キー名が 図 4.13 に示したものと異なる．これは，RoR の設定より規約（CoC）に従って記述したことが原因である．テーブル名や主キー名を 図 4.13 に示したものにするためには，図 4.14 右下に示す追加記述が必要である．また RoR は，クラステーブル継承（CR）や具象テーブル継承（CCR）をサポートしない．

4.5 記述力の評価

4.5.1 評価の内容と方法

本節では DBPowder の記述力の評価を実施した．比較対象は，単純な対応関係を簡易に実現できる Ruby on Rails（RoR）と複雑な対応関係を実現できる Hibernate である．なお RoR はバージョン 3.2.12，Hibernate はバージョン 4.1.10 を対象にした．

Hibernate の ORM 記述を行う .hbm XML ファイルについて，文書構造を定

める文書型定義（DTD）が提供されている．この DTD の要素型宣言や属性リスト宣言を上から走査し，それぞれ DBPowder-mdl でサポートするかを検証する．サポートする場合には続けてその要素型宣言の構成要素を走査対象とし，サポートしない場合にはその要素型宣言配下の構成要素を走査しない．これを，全ての走査が終えるまで繰り返す．次に，こうして検証した機能のそれぞれについて RoR の機能と照合し，サポートするかどうかを検証した．こうして得られた検証結果リストについて，同種の機能をまとめた上でカテゴライズし，最終結果とした．

Hibernate は複雑な対応関係を実現できる ORM フレームワークで世に広く使われている．本評価では，これと比較した DBPowder や RoR の機能不足を検証することができる．

4.5.2 評価結果

評価結果を表 4.4 と表 4.5 に示す．このうち表 4.4 は単純な対応関係と複雑な対応関係についてまとめたものであり，単純な対応関係を (s)，複雑な対応関係を (c) で番号付けした．表 4.5 はその他をまとめたものであり，制約 (oc)，内部的な動作の制御 (oi)，対象データの扱い (ot)，その他 (oe) の四種類に分類した．

まず表 4.4 をみる．DBPowder や Hibernate がサポートする機能のうち，RoR では単純な対応関係のうち (s5) および (s7) を，複雑な対応関係のうち (c3) および (c4) をサポートしない．特に (s7) のクラス継承（CR, CCR），(c3) 複数テーブルを結合して一つのクラスで扱う (c4) 複合キーによる主キーの三種類については，無理にサポートを試みると，RoR の特徴である文法ベースの CoC により開発の負担を軽減しようとするアプローチの利点が損なわれるため，サ

ポートを見送ったと考えられる．また，Hibernate がサポートする機能のうち，DBPowder と RoR では複雑な対応関係のうち (c6), (c7), (c8) をサポートしない．このうち (c6) についてはクラス継承の SR で代用可能であり，(c7) については SQL 文を直接永続化クラスや属性に対応づける点で ORM の別種のアプ

表 4.4: 記述力の評価結果 (1)：単純な対応関係と複雑な対応関係

○：サポートする　　×：サポートしない

		DBPowder	RoR	Hibernate
単純な対応関係	(s1) 永続化クラスとテーブルを一対一で対応づける	○	○	○
	(s2) 永続化クラスとテーブルで，属性を一対一で対応づける	○	○	○
	(s3) 永続化クラスに 1:1, 1:n, n:1, m:n の多重度をもつ別の永続化クラスを関連づける	○	○	○
	(s4) 多の多重度をもつ関連プロパティのコレクションの型を List とする	○	○	○
	(s5) 多の多重度をもつ関連プロパティのコレクションの型を Set や Map にする	○	×	○
	(s6) クラス継承（SR）	○	○	○
	(s7) クラス継承（CR, CCR）	○	×	○
	(s8) 属性値の制約条件を指定（値の指定 / not null）[1]	○	○	○
	(s9) 関連する永続化クラスを取得する際の制約条件を指定 [1]	○	○	○
複雑な対応関係	(c1) 属性をグループ化して一つの型として扱う	○	○	○
	(c2) 多態性のサポート	○	○	○
	(c3) 複数テーブルを結合して一つの永続化クラスで扱う	○	×	○
	(c4) 複合キーによる主キー	○	×	○
	(c5) 一意キー	○	○	○
	(c6) テーブルのカラム値に応じて異なる永続化クラスを対応づける（＝リンクなし）[2]	×	×	○
	(c7) SQL 文を永続化クラスや属性に対応づける（INSERT, UPDATE, DELETE, SELECT, SELECT 文中の列定義）[3]	×	×	○
	(c8) コレクションクラスから属性を取り出し，配列として扱う	×	×	○

1. DBPowder では，実行時に検索メソッドの条件として指定する
2. クラス継承の SR で代用可能
3. 検索の場合，オブジェクトを新たに永続化する機能をもたない

表 4.5: 記述力の評価結果 (2)：表 4.4 以外の結果

○：サポートする　　×：サポートしない

			DBPowder	RoR	Hibernate
その他（単純な対応関係や複雑な対応関係以外）	制約	(oc1) 属性値の変更可否を指定	○	○	○
		(oc2) テーブルへのタプル挿入可否の指定	×	○	○
		(oc3) テーブルのチェック制約	×	○	○
	内部的な動作の制御	(oi1) 関連の読込時の戦略（遅延読込 / イーガーフェッチ）[1]	○	○	○
		(oi2) 属性値の遅延読込	×	×	○
		(oi3) 識別子の値を自動生成する方法のカスタマイズ	×	×	○
		(oi4) 一括読込 / 書込（バッチ処理）時の，1回あたりの処理件数を指定	×	○	○
		(oi5) オブジェクトのメモリへの読込方法や永続化方法をカスタマイズ	×	○	○
		(oi6) メモリに読み込んだオブジェクトの，キャッシュ利用有無を指定	×	○	○
		(oi7) Insert/update で実行時にSQLをつくり，値の変更があったカラムのみ更新	×	○	○
		(oi8) キー制約名，外部キー制約名，インデックス名の明示的な指定	×	○	○
	対象データの扱い	(ot1) 連鎖参照整合性制約（update cascade, delete cascade）	○	○	○
		(ot2) 属性のデフォルト値の指定	○	○	○
		(ot3) コレクションのソート [2]	○	○	○
		(ot4) テーブルの悲観的ロック [3]	○	○	○
		(ot5) テーブルの楽観的ロック	×	○	○
		(ot6) オブジェクトが見つからない場合に，エラーにするか null を返すかを指定	×	○	○
		(ot7) テーブルのタプル値を Map [4] で扱う	×	○	○
		(ot8) List での添字用属性の指定，Map でのキー属性の指定	×	×	○
		(ot9) 多の多重度をもつ関連プロパティのコレクションの型を配列にする (Java) [5]	×	-	○
		(oe1) 開発者が記述したコードを ORM に組み込む [6]	○	○	○

1. DBPowder では，一度に読み込む範囲を検索実行時に指定する．Hibernate では，ORM 定義のなかでデフォルトの操作を指定できる
2. DBPowder では，実行時に検索メソッドの条件として指定する
3. DBPowder では，JDBC ドライバによりサポートする
4. RoR や Ruby 言語では連想配列とよばれる
5. Java の配列は List で代用可能である．また Java では，List から配列を取り出す toArray メソッドが提供される．
6. DBPowder では，Generation gap パターンにより永続化クラスはラップされており，ラッパークラスにコーディングできる．Hibernate は，開発者の独自クラスの getter/setter を .hbm ファイルに記述し，.hbm ファイルで使う型にできる

ローチと言える．また (c8) については，頻繁に使われる機能ではないとする意見がある [97, 83].

つぎに表 4.5 をみる．これらは，永続化クラスと関係スキーマの対応関係とは独立に設計できる部分である．特に DBPowder のサポートが弱い (oi1) – (oi8) は，ORM を実施する際の内部的な動作の制御に関わる部分である．

上記の通り，永続化クラスと関係スキーマの対応関係について DBPowder は，Hibernate がサポートする対応関係のうち，表 4.4 に示された三つ以外はサポートできることが確かめられた．

- (c6) テーブルのカラム値に応じて，関連や継承関係をもたない複数のクラスを対応づける
- (c7) SQL 文を永続化クラスや属性に対応づける
- (c8) コレクションを構成するクラスから一属性を選んで配列にして，コレクションを保持するクラスの属性として対応づける

4.6 開発に要する負担の評価：単語数を用いた記述量の比較

4.6.1 評価の内容と方法

本節と 4.7 節 では，単純な対応関係，複雑な対応関係，および単純な対応関係から複雑な対応関係への移行を扱う際の，開発に要する負担を比較評価する．本評価の対象は，DBPowder，Ruby on Rails（RoR），および Hibernate で

ある．これらの ORM フレームワークを用いて永続化クラス，関係スキーマ，および二者の対応をとる RDB アクセスコードを得る場合を想定し，単語数を用いた記述量の比較を実施する．

本評価は，ORM フレームワークを用いて新規に永続化クラス *pc*，関係スキーマ *rs*，および二者の対応をとる RDB アクセスコード *map* を開発する際に要する負担を比較評価するために実施する．対象としたアプリケーションは，*Redmine* [50]，*Magento* [61]，*KEKapp*（第 6 章），*CaveatEmptor*（ [8, 24]，図 4.15, 4.16），および OO7 ベンチマーク（ [17]，図 4.17，5.5.1 節）の五種類であり，単純な対応関係や複雑な対応関係の複数のパターンを対象とする．各々のアプリケーションの内容を表 4.6 に示す．各々のアプリケーションについて ORM を実現する際に開発者が記述するスクリプトを，DBPowder，Ruby on Rails（RoR），Hibernate について用意し，そのスクリプトの単語数を比較する．このスクリプトは，DBPowder では DBPowder-mdl，RoR では Scaffold ツール[9]に渡すパラメータ群と ActiveRecord クラス内での関連の宣言，Hibernate で

表 4.6: 開発に要する負担の評価：評価対象としたアプリケーション

アプリケーション名	内容
Redmine [50]	RoR で開発されたウェブベースのプロジェクト管理ツール
Magento [61]	e-Commerce サイト構築ツール
KEKapp（第 6 章）	KEK で運用しているセキュリティ管理サイト
CaveatEmptor（ [8, 24]，図 4.15, 4.16）	例示用ネットオークションアプリケーション．Hibernate の解説本 [8] で，Hibernate の複雑な対応関係への記述力を示すために用いられた
OO7 ベンチマーク（ [17]，図 4.17，5.5.1 節）	OODB 用に設計されたベンチマーク

[9]Scaffold ツールについては，2.2.1 節 を参照．

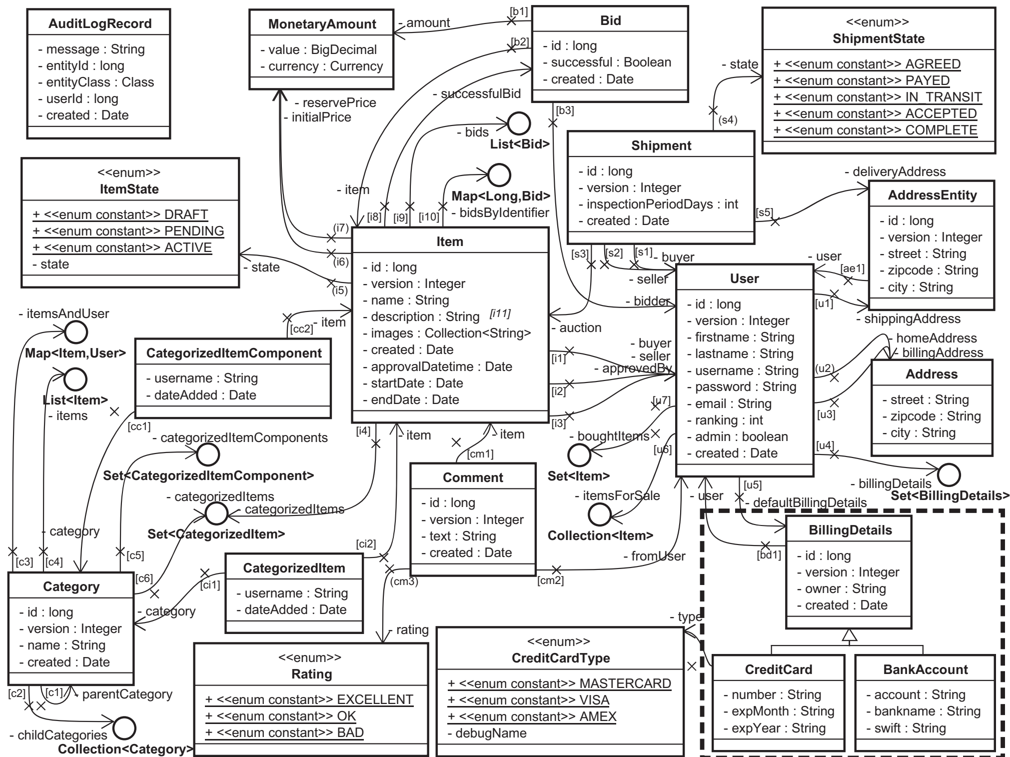


図 4.15: CaveatEmptor [8, 24]（オブジェクトモデル：クラス図）

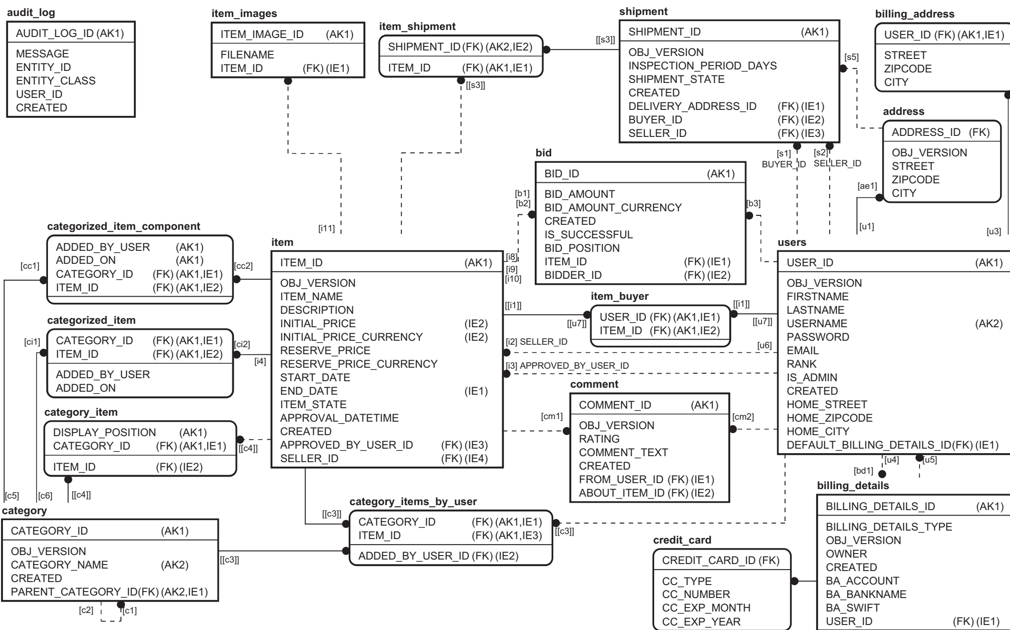


図 4.16: CaveatEmptor [8, 24]（関係モデル：IDEF1x 表記）

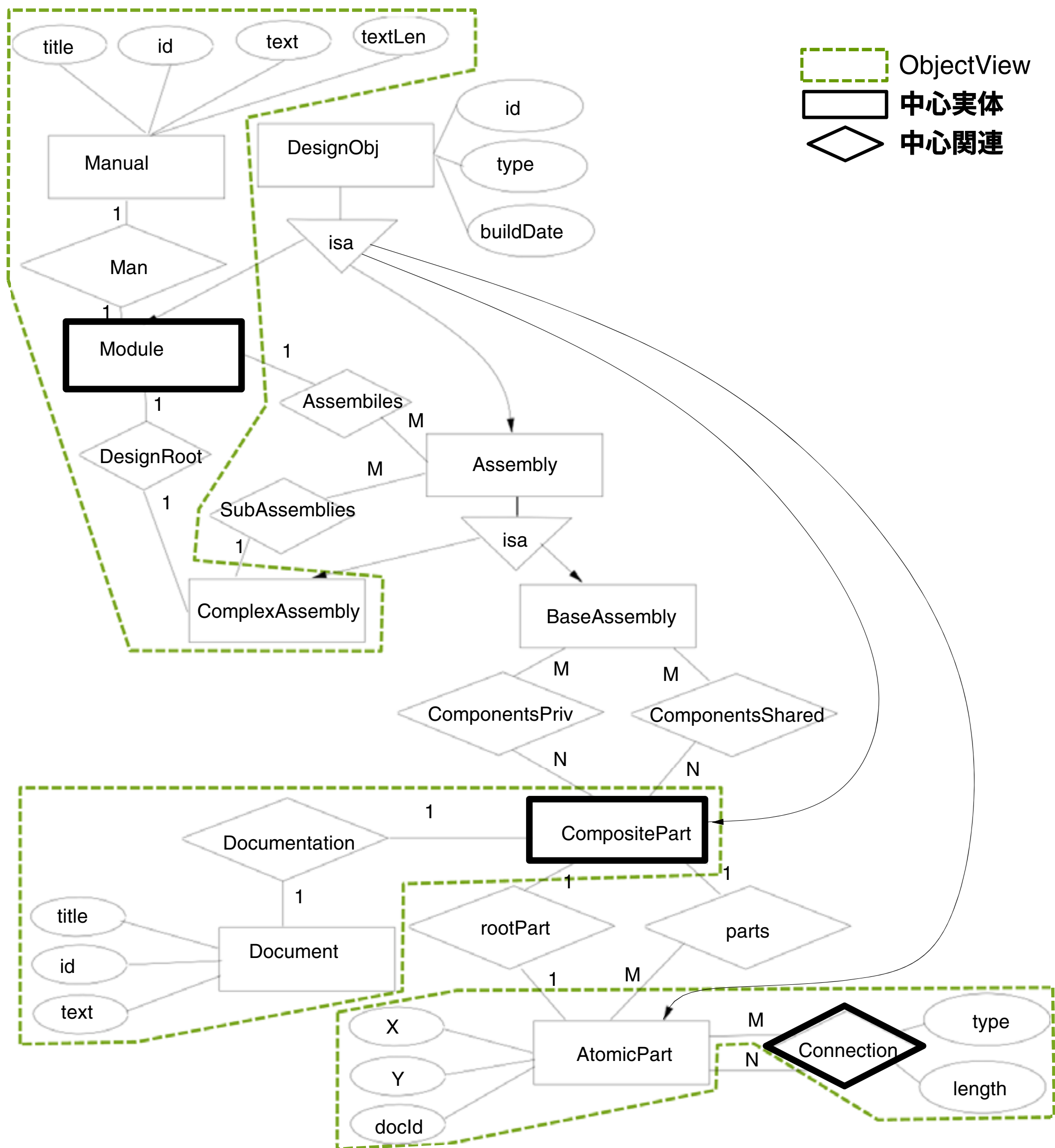


図 4.17: OO7 ベンチマーク [18, 17]（ER モデル）と，ObjectView の適用

```
[ModuleInfo@Module]
  {Manual@Manual}
[CompositePartInfo@CompositePart]
  {document}
  {AtomicPart@AtomicPart $rootPart}
[ConnectionInfo@Connection]
  {AtomicPart@AtomicPart $to}
```

図 4.18: DBPowder-mdl の記述例：図 4.17 の ObjectView

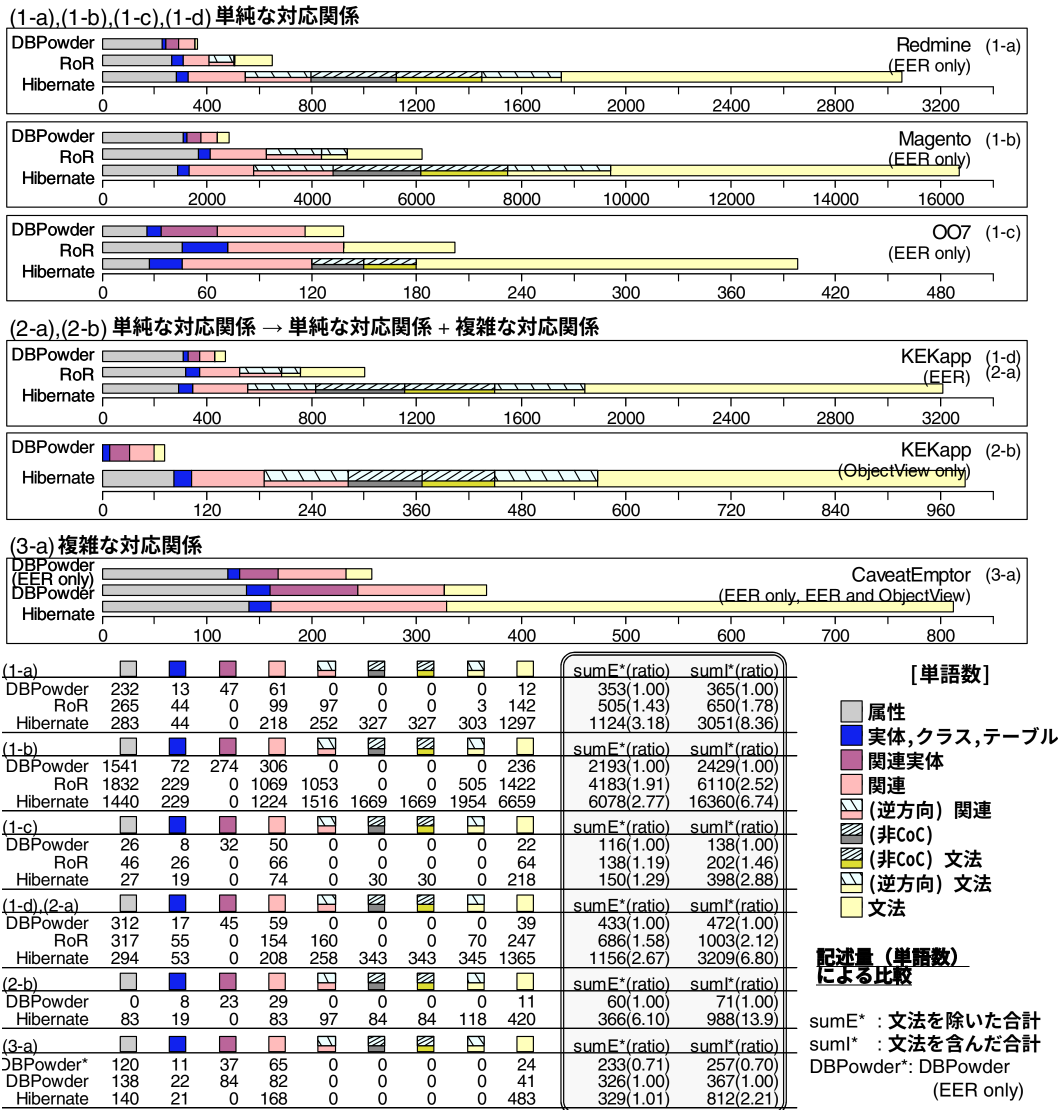
は .hbm XML ファイルである．

なお RoR と Hibernate で記述するスクリプトについては，DBPowder-mdl のスクリプトを RoR や Hibernate のスクリプトに変換するプログラムを作成した上で，その出力結果を用いる．この変換プログラムが生成するスクリプトは，フレームワークごとに単語数を小さくできるようにしている．また，変換結果である RoR や Hibernate のスクリプトが DBPowder-mdl のものと同等の内容を示すことを，OO7 ベンチマークを用いた動作内容の比較（5.5.2 節）により検証済である．

4.6.2　評価結果

図 4.19 に，DBPowder，RoR，Hibernate について，ORM 実施のための開発に要する負担を，単語数による記述量で比較した結果を示す．はじめに各々の比較結果の合計値について，全体的な傾向をみる．合計値は，図 4.19 の下側中央にある四列であり，灰色の囲みの中の記述である．(3-a) の syntax 込みの結果を除いた全てについて，DBPowder では必要な記述量を大幅に削減できることが示された．続いて syntax について見ると，Scaffold ツールへのパラメータ群と Ruby 言語で記述される RoR は，XML ベースの Hibernate と比べて必要な記述量が非常に小さいことが読み取れる．その RoR と比べても，DBPowder で必要とされる記述量は非常に小さい．この syntax の影響を除いても，DBPowder で必要な記述量は，関連において非常に小さい．

このような結果になった理由として，二つが考えられる．一つ目は，DBPowder-mdl では階層構造で実体間の関連を表しているため，関連内で改めて実体名を指定する必要がないことが挙げられる．二つ目は，DBPowder-mdl の関連実体の記述によって，実体名やクラス名やテーブル名を何度も記述する必要がない



	属性	実体,クラス,テーブル	関連実体	関連	(逆方向) 関連	(非CoC)	(非CoC) 文法	(逆方向) 文法	文法	sumE*(ratio)	sumI*(ratio)
(1-a)											
DBPowder	232	13	47	61	0	0	0	0	12	353(1.00)	365(1.00)
RoR	265	44	0	99	97	0	0	3	142	505(1.43)	650(1.78)
Hibernate	283	44	0	218	252	327	327	303	1297	1124(3.18)	3051(8.36)
(1-b)											
DBPowder	1541	72	274	306	0	0	0	0	236	2193(1.00)	2429(1.00)
RoR	1832	229	0	1069	1053	0	0	505	1422	4183(1.91)	6110(2.52)
Hibernate	1440	229	0	1224	1516	1669	1669	1954	6659	6078(2.77)	16360(6.74)
(1-c)											
DBPowder	26	8	32	50	0	0	0	0	22	116(1.00)	138(1.00)
RoR	46	26	0	66	0	0	0	0	64	138(1.19)	202(1.46)
Hibernate	27	19	0	74	0	30	30	0	218	150(1.29)	398(2.88)
(1-d),(2-a)											
DBPowder	312	17	45	59	0	0	0	0	39	433(1.00)	472(1.00)
RoR	317	55	0	154	160	0	0	70	247	686(1.58)	1003(2.12)
Hibernate	294	53	0	208	258	343	343	345	1365	1156(2.67)	3209(6.80)
(2-b)											
DBPowder	0	8	23	29	0	0	0	0	11	60(1.00)	71(1.00)
Hibernate	83	19	0	83	97	84	84	118	420	366(6.10)	988(13.9)
(3-a)											
DBPowder*	120	11	37	65	0	0	0	0	24	233(0.71)	257(0.70)
DBPowder	138	22	84	82	0	0	0	0	41	326(1.00)	367(1.00)
Hibernate	140	21	0	168	0	0	0	0	483	329(1.01)	812(2.21)

図 4.19: 開発に要する負担の評価結果：単語数を用いた記述量の比較

ことが挙げられる．このように全体的な比較結果としては，DBPowder では必要な記述量を大幅に軽減できており，単純な対応関係を簡易に実現でき，かつ複雑な対応関係を実現できることが示された．

次に，単純な対応関係に関する結果を見る．図 4.19 (1-a) の Redmine は RoR で開発されたアプリケーションであり，構築される ORM のモジュールは，RoR の設定より規約（CoC）に基づいた構成となる．したがって，RoR が大幅に優位な結果になると予想される．実際に RoR と Hibernate の比較では RoR が大幅に優位な結果となっており，(1-a) 以外と比べてその差は顕著となっている．DBPowder は，(1-a) のような RoR に有利なアプリケーションで syntax を除外して比較しても，RoR よりも優位な結果となっている．

一方，図 4.19 (1-b) の Magento では RoR の優位性は損なわれており，syntax を除けば RoR と Hibernate の差異は小さくなっている．この原因は，Magento のスキーマが RoR の CoC に従わない一方で関係スキーマの構造が複雑であるために，文法ベースで必要な記述量を下げようとするアプローチの RoR が有効に機能しなかったためと考えられる．そのようなスキーマにおいても，DBPowder では大幅に優位な結果をもたらしている．この原因は，DBPowder が EER モデルでデータの構造をとらえつつ階層構造と実体関連でスキーマの大部分を簡易に記述する手法とったことにあると考えられる．

属性に関しては，syntax を除けば DBPowder，RoR，および Hibernate の結果はほぼ同一となっている．この理由としては，非キー属性のみを見ると構造をもたず，EER モデルや CoC などの工夫による冗長度を下げる余地が少ないことが挙げられる．

次に，複雑な対応関係に関する結果を見る．図 4.19 (3-a) の *CaveatEmptor* では，syntax を除けば Hibernate とほぼ同等の結果となり，優位な結果を得られなかった．この原因としては，DBPowder が単純な対応関係と複雑な対応関係の ORM を両方提供しているのに対し，Hibernate ではアプリケーションに特化した複雑な対応関係のみを提供している点があると考えられる．アプリケーションに特化した複雑な対応関係のみでは永続化データへの操作に制限が出

る可能性があることを考えると，DBPowder が提供する単純な対応関係に基づく ORM では，様々なアプリケーションに通用する一般的なモデルは，アプリケーションに特化した開発を行っているときでも有用である．しかし，データモデルの設計が十分に妥当であれば，単純な対応関係に基づく ORM は冗長となる可能性もある．別の原因としては,

なおこれまで syntax を除いた記述量についても述べてきたが，実際の開発ではスクリプトから syntax に属する記述を排除できない．したがって，本来の開発の負担の意味合いからは syntax は含まれるべきものである．Syntax を含む場合は全てのケースで DBPowder の優位性が高まる結果となる．

4.7 開発に要する負担の評価：複雑な対応関係に移行する開発で要した編集の回数や単語数の比較

4.7.1 評価の内容と方法

本節では 4.6 節 に引き続き，単純な対応関係，複雑な対応関係，および単純な対応関係から複雑な対応関係への移行を扱う際の，開発に要する負担を比較評価する．本評価の対象は，DBPowder と Hibernate であり，これらの ORM フレームワークを用いて永続化クラス，関係スキーマ，および二者の対応をとる RDB アクセスコードを得る場合を想定し，単純な対応関係から複雑な対応関係に移行する開発で要した編集の回数および編集した単語数の比較を実施する．

本評価は，既に単純な対応関係に基づく ORM が成立していて，そこに複雑な対応関係による ORM を追加する際に生じる，開発に要する負担を比較評価

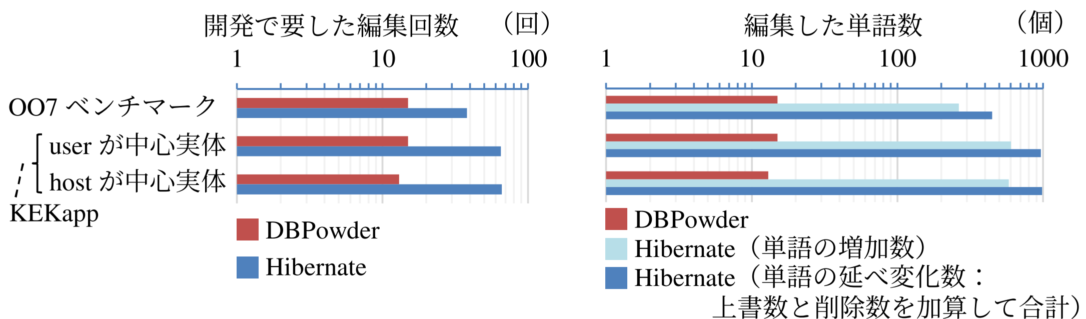
するために実施する．対象としたアプリケーションはOO7ベンチマークおよびKEKappであり，複雑な対応関係に基づくORMを追加するのに必要なスクリプトをDBPowderとHibernateで記述し，その開発で要した編集の回数および編集した単語数を比較する．追加した複雑な対応関係は，図4.17（OO7ベンチマーク）や図4.7（KEKapp）の緑囲みで示したDBPowderのObjectViewと，それに相当するHibernateの記述である．記述したDBPowder-mdlのObjectViewを，図4.18（OO7ベンチマーク）および図4.8（KEKapp）に示す．

4.7.2　評価結果

単純な対応関係のみから構成されるORMから複雑な対応関係に移行する開発で要した，編集の回数および編集した単語数を比較した結果を，表4.7に示す．上部に合計数をまとめたグラフを示し，下部にその内訳を示す．内訳は，上段がOO7ベンチマーク（図4.17），中段がKEKappにて`user`を中心実体とした場合（図4.7のov_1とcpc_1），下段がKEKappにて`host`を中心実体とした場合（図4.7のov_2とcpc_2）である．なお各々で用いたObjectViewについて，OO7ベンチマークでのDBPowder-mdlによる記述は図4.18に示され，KEKappでのDBPowder-mdlによる記述は図4.8に示されている．

各々，結合対象とした実体名またはテーブル名を表の左部に示し，DBPowderとHibernateで対応するORMを追加する際に必要とした編集回数を，コピー＆ペースト，上書，追加，削除に分類して表の中央部に示し，同じく編集した単語数を表の右部に示した．いずれの結果においても，DBPowderでは対象の実体名をObjectViewとして追加したのみだが，Hibernateでは対象のクラス定義の全体をコピー＆ペーストしたのちに，単語数の増減を伴う複数回の編集が発生した．そこでHibernateにおける編集した単語数の合計では，編集した単

表 4.7: 開発に要する負担の評価結果：複雑な対応関係に移行する開発で要した，編集の回数および編集した単語数の比較．移行内容は，図 4.17（OO7 ベンチマーク）や図 4.7（KEKapp）に示した ObjectView を参照．



結合対象とした実体名またはテーブル名	開発で要した編集回数 (回)						編集した単語数 (個)						
	DBPowder	Hibernate					DBPowder	Hibernate					
		合計	内訳					合計	合計	内訳			
	追加のみ発生		コピー&ペースト	上書	追加	削除	追加のみ発生	増加数	延べ変化数	コピー&ペースト	上書	追加	削除
OO7 ベンチマーク													
Module, Manual	4	10	2	4	1	3	4	63	109	81	6	2	20
CompositePart, Document, AtomicPart	6	18	3	7	2	6	6	116	207	152	11	4	40
Connection, AtomicPart	5	10	2	4	1	3	5	85	131	103	6	2	20
合計	15	38					15	264	447				
合計の DBPowder 比	*1*	*2.53*					*1*	*17.6*	*29.8*				
KEKapp: user が中心実体													
user, adm_user, guest_user, division, division_cat, user_stat	7	27	4	10	4	9	7	195	331	239	16	16	60
sec_report, host	3	11	2	4	2	3	3	198	244	206	6	12	20
register, user, host_status, user_role	5	27	4	10	3	10	5	209	389	285	16	6	82
合計	15	65					15	602	964				
合計の DBPowder 比	*1*	*4.33*					*1*	*40.1*	*64.3*				
KEKapp: host が中心実体													
host	2	4	1	1	2	0	2	84	85	64	1	20	0
sec_report, user	3	11	2	4	1	4	3	259	349	299	6	2	42
register, user, division, division_cat, user_stat, host_status, user_role	8	51	7	19	6	19	8	237	552	367	31	12	142
合計	13	66					13	580	986				
合計の DBPowder 比	*1*	*5.08*					*1*	*44.6*	*75.8*				

語の最終的な増加数を示すとともに，編集過程で発生した上書数や削除数を加算して合計した単語の延べ変化数も合わせて集計した．

結果を比較したところ，編集回数，編集した単語の増加数，編集した単語の延べ変化数の全てにおいて，Hibernate に比べて DBPowder で開発に要する負担が大幅に少ない結果が得られた．OO7 の比較結果（上段）では，DBPowder に比べて Hibernate が，編集回数で 2.53 倍，単語の増加数で 17.6 倍，単語の延べ変化数で 29.8 倍，多く要する結果となった．KEKapp：`user` が中心実体の比較結果（中段）では，DBPowder に比べて Hibernate が，編集回数で 4.33 倍，単語の増加数で 40.1 倍，単語の延べ変化数で 64.3 倍，多く要する結果となり，差異は OO7 よりもさらに大きくなった．また KEKapp：`host` が中心実体の比較結果（下段）では，DBPowder に比べて Hibernate が，編集回数で 5.08 倍，単語の増加数で 44.6 倍，単語の延べ変化数で 75.8 倍，多く要する結果となり，同じく差異は OO7 よりもさらに大きくなった．

複雑な対応関係に移行する開発において，DBPowder では，EER 概念モデルを基準にして，アプリケーションロジックで利用するデータ構造を ObjectView として指定する．今回の三つの例においても，設計元となった EER 概念モデルを検討し，アプリケーションロジックで利用する実体を ObjectView で指定して追加するのみで，所望の永続化クラスを得ることができた．一方 Hibernate では，ORM を実現する各々の永続化クラスについて，必要な定義を全て記述する必要がある．したがって，Hibernate で個別のアプリケーションロジックに適した永続化クラスを追加する際に効率の良い開発方法は，開発の開始時に，既存の .hbm 定義ファイルで記述済の ORM 定義のうち，追加する永続化クラスに関係する部分をはじめにコピー＆ペーストすることである．表 4.7 のコピー＆ペーストの項目で示されるように，ここでの編集回数は少ないが，扱われる単語数は多い．この単語数が膨らんだコピー＆ペーストの結果から実際

に必要な永続化クラスと ORM を得るためには，様々な箇所を修正する必要が生じた．結果として，編集を実施するべき回数や単語数が膨らんだ．

次に，KEKapp と OO7 にあらわれた結果の違いについて考察する．KEKapp は OO7 とくらべて，一実体や一クラスあたりの属性数が多くなっており，それが編集した単語数の更なる差異となって反映されている．また，図 4.18 に示すように，OO7 のテーブル名は一文字目が大文字であるために，設定より規約（CoC，表 4.2）を適用できず，クラス名とテーブル名の両方を明示的に指定する必要が生じた．これにより OO7 における DBPowder-mdl の記述は，KEKapp と比べて一実体あたりに必要とされる単語数が多くなった．よって，OO7 における開発に要する負担の軽減効果は，KEKapp よりは弱くなった．とはいえ OO7 においても，Hibernate との比較では開発に要する負担が大幅に少ない結果が得られた．

以上の評価結果により，DBPowder は単純な対応関係から複雑な対応関係に移行する開発を効率的にサポートできることが示された．

4.8 まとめ

単純な対応関係を簡易に実現できる ORM フレームワークでは，複雑な対応関係をうまく扱えない．一方で複雑な対応関係を実現できる ORM フレームワークでは，開発初期の段階から永続化クラスとテーブルの詳細な対応関係を記述する必要が生じる上に，機能拡張の際にこれを修正する必要が頻繁に出てくるため，迅速な開発が強く求められる開発初期の段階から，開発の負担が大きくなる．第 4 章 ではこの問題の克服のために，概念モデリングに基づく ORM 手法，DBPowder を提案した．DBPowder では問題の克服のために，EER モデルを用いた概念モデリングと，有向グラフベースで EER モデル上の実体

の走査経路と利用方法を指定する ObjectView の，連携による併用方式を提案した．EER モデルにより，単純な対応関係をもつ永続化クラスと関係スキーマおよび，これの対応をとる RDB アクセスコードを簡易に生成できる．EER モデルに ObjectView を加えることで，追加的または修正的に複雑な対応関係をもつ永続化クラスを生成できる．また，EER モデルと ObjectView をコンパクトに記述できる言語 DBPowder-mdl を提案した．

第 4 章 の提案の有効性を検証するために，記述力の評価と開発に要する負担の評価を実施した．

記述力の評価では，単純な対応関係を簡易に実現できる Ruby on Rails（RoR）と複雑な対応関係を実現できる Hibernate を比較対象とし，DBPowder による ORM を記述するための言語 DBPowder-mdl の言語定義と Hibernate の ORM 記述を行う .hbm XML ファイルの文書型定義（DTD）を比較し，DBPowder の記述力を比較検証した．その結果，永続化クラスと関係スキーマの対応関係について DBPowder は，以下の三種類を除いて Hibernate と同等のサポートを得られることが確かめられた．1) 関連や継承関係をもたない複数のクラスを対応づける，2) SQL 文を永続化クラスや属性に対応づける，3) コレクションを構成するクラスから一属性を選んで配列にして，コレクションを保持するクラスの属性として対応づける．

開発に要する負担の評価では，単語数を用いた記述量の評価と，複雑な対応関係に移行する開発で要した編集の回数および編集した単語数の比較の二種類の評価を実施した．

単語数を用いた記述量の評価では，Hibernate では複雑な対応関係を実現できるが必要な記述量は膨大になり，Hibernate よりも DBPowder が大幅にコンパクトに ORM を記述できることが明らかとなった．この理由として，DBPower

ではEERモデルの導入，階層構造を基本にした記述形式，およびCoC導入の効果が大きく，同じ要素を何度も指定する必要がなく，また，文法により必要とされる記述要素をコンパクトに抑えたことが挙げられる．次に，単純な対応関係におけるRoRとDBPowderの比較では，文法ベースのCoCを満たさない場合に損なわれる簡易さの程度が，RoRと比べてDBPowderが限定的であることが明らかとなった．この理由として，RoRにおける文法ベースで必要な記述量を下げようとするアプローチが，関係スキーマの構造が複雑な場合に有効に作用しなくなる一方で，EERモデルに基づいたデータ構造をとらえて単純な対応関係を簡易に実現するDBPowderにおいて，複雑な関係スキーマにおける文法ベースのCoCの逸脱による影響がRoRと比べて限定的に抑えられた点が挙げられる．

複雑な対応関係に移行する開発で要した，編集の回数および編集した単語数の比較は，DBPowderとHibernateで実施した．Hibernateにおいて複雑な対応関係によるORMを追加するためには，記述済のORM定義のコピー＆ペーストが必要となる．このコピー＆ペーストの結果を編集すると目的のORMを得ることができるが，その開発の負担は大きいことが明らかになった．それと比べてDBPowderでは，個別のアプリケーションロジックのみをObjectViewで記述して，永続化クラスを効率的に追加できることが定量的に示された．この結果は，DBPowderのObjectViewが，既存のデータスキーマを用いて個別のアプリケーションロジックに適した永続化クラスを効率的に追加する枠組を提供することによりもたらされた．

上記に示した評価により，DBPowderが単純な対応関係の簡易な実現と複雑な対応関係の実現についてサポートを両立でき，かつ，単純な対応関係から複雑な対応関係に移行する開発を効率的にサポートできることが示された．

第5章　DBPowder ORM システム

5.1　はじめに

第4章では，概念モデリングに基づくO/Rマッピング（ORM）手法としてDBPowderを提案するとともに，その記述言語DBPowder-mdlを示し，評価を実施した．その結果，提案手法DBPowderが，単純な対応関係の簡易な実現と複雑な対応関係の実現についてサポートを両立できることと，単純な対応関係から複雑な対応関係に移行する開発を効率的にサポートできることが示された．

実用上は，この提案がシステムとして動作し，妥当な性能を示すことが重要である．たとえばDBPowderでは，ObjectViewによりアプリケーションロジックに対応した複数種類の永続化クラスを柔軟に生成できる．これにより，ひとつの永続化されるプロパティのインスタンスを，ObjectViewで柔軟に生成された複数種類の永続化クラスが管理する事態が生じる．これは，ひとつのセッション内でひとつの永続化されるインスタンスが複数のオブジェクトに複製されうることを意味しており，これを適切に管理しないと，属性値に矛盾が発生しうる．セッション処理においてこの事態を適切に処理することは重要である．

本章で示すDBPowder ORMシステムは，アプリケーションプログラム上でORM機能を実現するためのコードを，開発者（応用開発者）に提供するため

のシステムである．なお本章では開発者を，データスキーマを設計する設計開発者とアプリケーションを開発する応用開発者に区別する[1]．設計開発者はEER モデルと ObjectView を DBPowder-mdl で記述する．DBPowder ORM システムはこれらを入力として解釈し，永続化クラス，関係スキーマ，および二者の対応をとる RDB アクセスコードの三点を，ORM 機能を実現するコードとして応用開発者に提供する．

DBPowder ORM システムの評価として，OO7 ベンチマークによる性能評価を，DBPowder，Hibernate，および RoR について実施する．

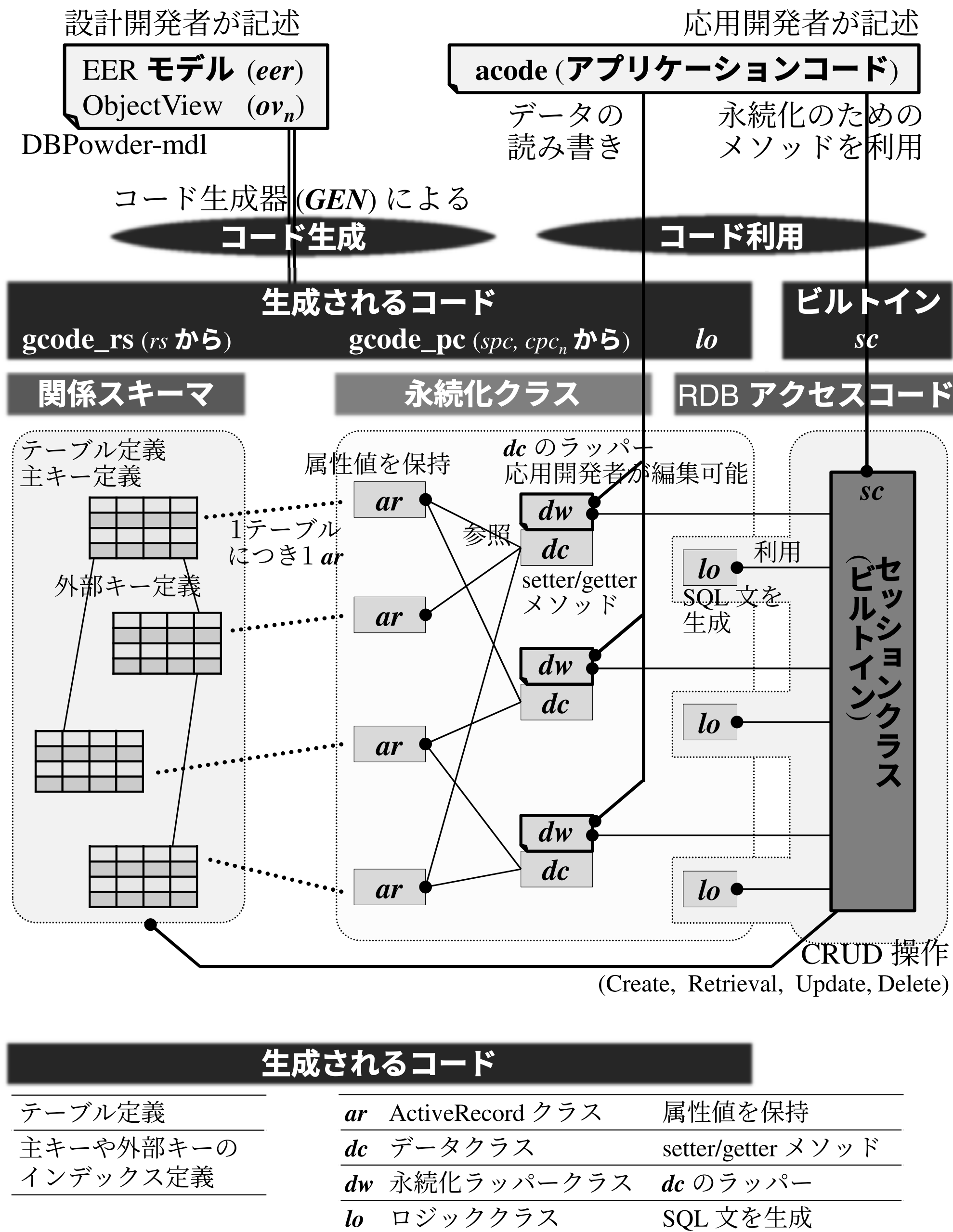
図 5.1 に，DBPowder ORM システムの全体構成を示す．DBPowder ORM システムは，大きく二つの機能から構成される．一つ目は ORM 機能をもつコードの生成機能であり，これを 5.2 節 で示す．二つ目は生成されたコードにより実現される ORM 機能であり，これを 5.3 節 で示す．続いて 5.4 節 で DBPowder ORM システムのプロトタイプ実装を示し，5.5 節 で OO7 ベンチマークによる性能評価を示す．最後に 5.6 節 で本章をまとめる．

5.2　ORM 機能をもつコードの生成機能

図 5.1 に，DBPowder ORM システムの全体構成を示す．DBPowder ORM システムは ORM 機能をもつコードを生成し，生成されたコードは ORM モジュールを構成する．以下に，DBPowder ORM システムが関係スキーマ，永続化クラス，および二者の対応をとる RDB アクセスコードを生成し，応用開発者がこれらのコードを利用するまでの過程を示す．

(1) 設計開発者は，記述言語 DBPowder-mdl（4.4 節）で EER モデル（*eer*）

[1]設計開発者と応用開発者は同一人物であっても差し支えない．



テーブル定義
主キーや外部キーの インデックス定義

ar	ActiveRecord クラス	属性値を保持
dc	データクラス	setter/getter メソッド
dw	永続化ラッパークラス	*dc* のラッパー
lo	ロジッククラス	SQL 文を生成

図 5.1: DBPowder ORM システムの全体構成：ORM 機能をもつコードの生成と，生成されたコードが構成する ORM モジュール

と ObjectView（ov_n）を記述する．

(2) コード生成器（*GEN*）は，記述された *eer* と ov_n を解釈し，4.2 節 および 4.3 節に示した方法で *rs*, *spc*, cpc_n を生成する．

(3) 続いて *GEN* は，*rs* の定義に従って関係スキーマのコード gcode_rs を生成し，*spc* と cpc_n の定義に従って永続化クラスのコード gcode_pc を生成する．また *GEN* は，ビルトインのセッションクラス *sc*（後述）が SQL 文を生成する際に利用するロジッククラス *lo* を生成する．永続化クラスと関係スキーマの対応をとる RDB アクセスコードは，*sc* と *lo* により構成される．

(4) 応用開発者は，生成された gcode_pc とビルトインの *sc* を利用して，目的のアプリケーションコード（acode）を記述する．

コード生成器 *GEN* が生成するコードは，関係スキーマのコード gcode_rs, 永続化クラスのコード gcode_pc，およびロジッククラス（*lo*）である．gcode_rs は，テーブル定義および，主キーや外部キーへのインデックス定義から構成される．gcode_pc は図 5.1 に示す通り，永続化ラッパークラス（*dw*），データクラス（*dc*），ActiveRecord[2]（*ar*）から構成される．セッションクラス（*sc*）と *lo* により，RDB アクセスコードが構成される．応用開発者がアプリケーションコード（acode）から直接扱うクラスは *dw* と *sc* であり，他のクラスは全て DBPowder 内部で使用される．

永続化ラッパークラス *dw* は応用開発者に，getter/setter メソッドから構成される永続化クラスのインタフェースを提供する．セッションクラス *sc* は，応用開発者が *dw* を利用して実際に永続化に関係する操作を実施するために利用する．応用開発者が使用する *dw* と *sc* の詳細は，5.3 節 で述べる．

[2] 2.2.1 節に示した ActiveRecord クラスを基本とする．

各々のテーブルと永続化クラスの複雑な対応関係は，*ar* と *dc* の対応関係によって実現される．ActiveRecord クラス（*ar*）は，`gcode_rs` 上のテーブルと一対一に対応しており，永続化対象のデータについて DBPowder では，属性値のインスタンスは *ar* にのみ保持される．これにより，永続化クラスとテーブルが複雑な対応関係をもつ場合でも，それぞれの属性値がひとつのセッション内で重複して生成されることはなく，データ一貫性の把握が簡潔になる．これについては 5.3.2 節で論じる．

永続化ラッパークラス *dw* の実装は，*dc* の継承のみで構成される．応用開発者は *dw* 上に追加コードを実装することで，永続化クラスのふるまいを追加できる．設計開発者が DBPowder-mdl に変更を加えた際には *GEN* が既存のコードを上書きするが，コード *dw* は *dc* の継承のみで構成されるため，DBPowder-mdl の変更は *dw* には波及しない．したがって，応用開発者が *dw* に加えた変更は DBPowder-mdl の変更後も上書きされず保持され，応用開発者は *GEN* の挙動を考慮せずに *dw* に追加開発できる．なおこのような *dw* の使われ方は，generation gap [98] デザインパターンとよばれる．

ロジッククラス *lo* は *sc* から利用される．*lo* は，応用開発者が *sc* に要求した永続化に関係する操作の要求を受けとり，これに基づいて RDB への実際のクエリや応答受けとりを実施するための SQL を生成する．RDB への検索が *sc* から要求された際には，*lo* は要求を SQL 文に変換して RDB にクエリを実施する．RDB からの応答結果を *ar* に格納し，*dc* や *dw* を構成した上で，*dw* を *sc* に返す．RDB へのデータ格納が *sc* から要求された際には，*lo* は格納するデータを *dw* から受けとり，*dw* を構成する *ar* の内容を SQL 文に変換して，RDB にデータ操作を要求する．続けて *lo* は，RDB から要求したデータ操作の成功/失敗を受けとり，その内容を *sc* に返す．

```
 1: // セッション so を予め取得しておく
// newHost の永続化（保存）
 2: Host newHost = new Host();
 3: newHost.setIp("192.168.x.x");
 4: so.insert(newHost);
// 自動生成された主キー値の表示
 5: print(newHost.hostId());
// like 検索（ロード）
 6: List<Host> currHostList =
      so.find(Host.class,"name like 'te%'");
 7: // (currHostList に関する処理)
// 処理後，currHost1 の値を更新
 8: currHost1.setHostName("test-pc");
 9: so.update(currHost1);
// 処理後，currHost2 を削除
10: so.delete(currHost2);
// コミット /* ロールバックは so.rollback() */
11: so.commit();
// セッションを閉じる
12: so.close();
```

図 5.2: アプリケーションコード（acode）の例：応用開発者による永続化クラスのコード（gcode_pc）の利用

5.3 生成されたコードにより実現される ORM 機能

5.3.1 永続化に関する基本的な操作

図 5.2 に，永続化クラスのコード gcode_pc を使用したアプリケーションコード acode の記述例を示す．セッションクラス *sc* は永続化を実行するためのメソッド保持しており，永続化ラッパークラス *dw* を用いて永続化を実行す

る．*sc* がもつ永続化に関するメソッドは，図中に太字で示した `insert`，`find`，`update`，`delete`，`commit`，`rollback`，および `close` である．3.1 節 で述べたとおり，永続化に関する操作には，保存，ロード，更新，削除の四種類があり，`insert`，`find`，`update`，`delete` メソッドはこのそれぞれに対応している．永続化に関係する操作は *sc* をインスタンス化したセッションオブジェクト so 単位で行われる．DBPowder では，このセッション単位でトランザクション処理を行うことが可能であり，トランザクション内では ACID 特性 [103] が保証される．

永続化メソッド `insert`（図 5.2 の例：2 行目 - 5 行目）は，永続化ラッパークラス *dw* のオブジェクト（以下，永続化オブジェクト）について，保存を指示する．応用開発者が永続化ラッパークラス *dw* の setter メソッドによりオブジェクトに値をセットした後に so オブジェクトの `insert` メソッドを発行することで，値の保存が指示される．但し，指示した段階では実際には関係データベースに値は格納されない．実際に格納するためには，後に説明する `commit` メソッドの発行が必要である．主キー値は DBPowder ORM システムが RDBMS などの機能を使って自動生成するため，応用開発者が主キー値を意識する必要はない．

永続化メソッド `find`（図 5.2 の例：6 行目）は，RDB への保存が既に完了している値を RDB からロードし，その値を用いて永続化ラッパークラス *dw* をインスタンス化し，返値とする．値の取得条件 は，SQL の WHERE 句にあたる検索文により指定できる．主キー値をキーにした検索では返値の件数は 1 件または 0 件が保証されるが，通常の検索では件数が複数件になる場合があるので，`find` メソッドの返値の型はコレクションとなる．なお，取得したオブジェクトのプロパティへのアクセスは，getter メソッドにより行う．

永続化メソッド `update`（図 5.2 の例：8 行目 - 9 行目）は，保存済のオブジェ

クトの値を更新する．応用開発者が永続化ラッパークラス *dw* の setter メソッドによりオブジェクトに値をセットした後に `so` オブジェクトの `update` メソッドを発行することで，保存済の値の更新が指示される．`insert` 同様，実際に更新するためには，後に説明する `commit` メソッドの発行が必要である．

永続化メソッド `delete`（図 5.2 の例：10 行目）は，保存済のオブジェクトについて，永続化を解除する．応用開発者が `so` オブジェクトの `delete` メソッドを発行することで，保存済の値の永続化解除が指示される．`insert` 同様，実際に解除するためには，`commit` メソッドの発行が必要である．

メソッド `commit`（図 5.2 の例：11 行目）は，`insert`，`update`，および `delete` メソッドで指示された永続化に関する命令を RDBMS に発行し，操作を完了させる．永続化に関する命令を取り消したいときは，`rollback` メソッドを発行する．

メソッド `close`（図 5.2 の例：12 行目）は，セッションを終了する．

5.3.2 永続化に関する内部的な処理

DBPowder と ObjectView を用いた開発では，設計開発者が個々のアプリケーションロジックに応じて永続化クラスを設計し開発するため，ひとつのテーブルやテーブル内の属性を複数の永続化クラスが利用する状況が発生する．図 5.3 に例を示す．この例において，三つのテーブルから構成される関係スキーマ *rs* は，アプリケーションロジックを考慮すると様々な永続化クラスの設計があり得る（図 4.2，図 4.4 参照）．ここで永続化クラス *spc*，cpc_1，cpc_2 を設計したケースを考える．

通常にとられる，永続化クラスとテーブルを各々に対応づける手法では，図

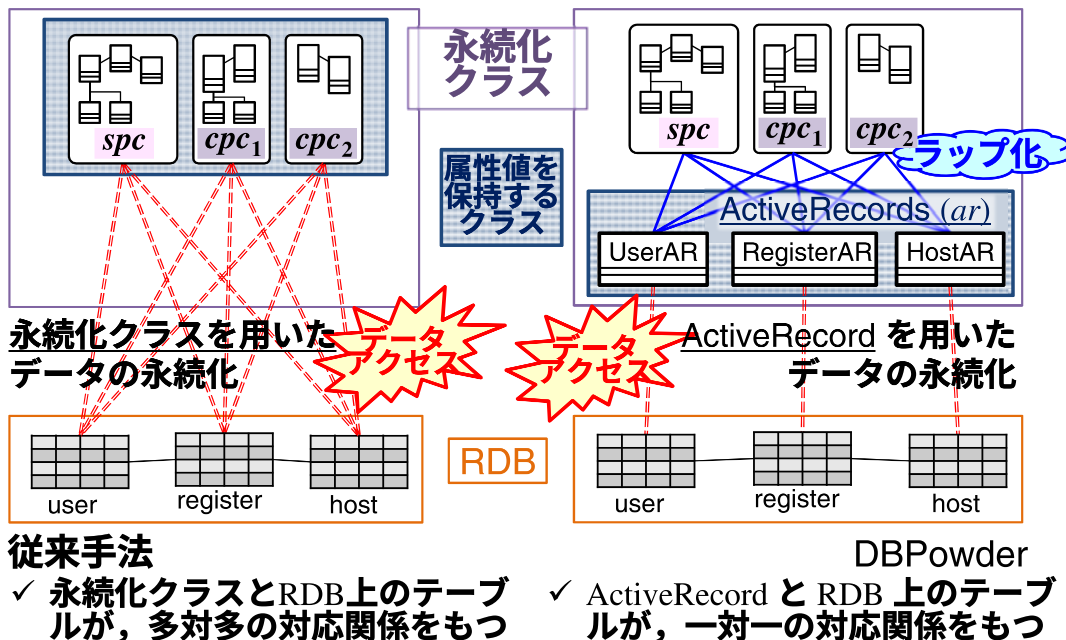


図 5.3: ActiveRecord クラスの導入による，永続化対象のオブジェクトと RDB 上のタプル値の一貫性管理

4.2 の左側に示すように，ひとつのテーブルが複数種類の永続化クラスによって利用される状況が生じる．ここで，単一のセッション内で永続化クラス *spc*，cpc_1，cpc_2 のうち複数種類のクラスが使用されるケースを考える．複数種類のクラスの別々のオブジェクトが同じタプル内の属性値を保持していて，その片方のオブジェクトで値の変更が起きた場合に，別のクラスのオブジェクトは，この値の変更を検知できない．このように同一セッション内において，複数のオブジェクトが参照する同じタプル内の属性値について，変更が相互に管理されないと，セッション内の状態に矛盾が生じる．この場合，矛盾した属性値について整合性を保つのが困難となる．

DBPowder では，*ActiveRecord* クラスを活用したアーキテクチャをとることにより，この問題を解決する．以下，まずは DBPowder がとるアーキテクチャ

について述べ，そののちに ActiveRecord がこの問題を解決する様子を示す．

DBPowder において永続化ラッパークラス（図 5.1 における *dw*）は，データクラス（同じく *dc*）を継承することで getter/setter メソッドを保持する．また，*dc* は *ActiveRecord* [36] クラス（同じく *ar*）を保持する．DBPowder における ActiveRecord クラスは 2.3.2 節 で示した通常の ActiveRecord クラスと同様である．ActiveRecord クラスは `gcode_rs` におけるテーブルごとに生成され，テーブルがもつ属性と一対一に対応して ActiveRecord クラスの属性が生成される．

DBPowder では永続化対象のデータについて，属性のインスタンスは *ar* にのみ保持される．データクラス *dc* は，*ar* の getter/setter メソッドを用いて属性にアクセスし，永続化ラッパークラス *dw* は，*dc* の getter/setter メソッドを用いて属性にアクセスする．結果として永続化ラッパークラス *dw* は，getter/setter メソッドを用いて *ar* が保持する属性を用いている．

図 5.3 の右側に，その様子を示す．永続化ラッパークラス *dw* は ActiveRecord クラス *ar* を介してテーブル内の属性との対応関係がとられる．テーブル内の属性値は，永続化ラッパークラス *dw* 内において直接には *ar* が保持しており，*ar* に対する値の変更は直接に *dw* に反映される．よって，複数の永続化クラスのオブジェクトが同じタプル内の属性値を保持していて，その片方のオブジェクトで値の変更が起きた場合に，全てのオブジェクトがその変更を直接に検知できる．よって，同一セッション内で属性値が矛盾する問題を回避できる．

5.4 DBPowder ORM プロトタイプシステムの実装

提案手法の評価のために，DBPowder ORM プロトタイプシステムを実装した．実装の対象とした環境は Java 1.7.0_25 [77] および MySQL 5.5.32 [70] であるが，MySQL 5 以上，Java 1.6 以上での動作を確認している．また，PostgreSQL や Oracle Database 上でも，多くの機能が動作する．現在のプロトタイプシステムは，以下の内容を提供する．

- 設計開発者が EER モデルと ObjectView を記述するための，記述言語 DBPowder-mdl
- DBPowder-mdl を解釈し，永続化クラス，関係スキーマ，および二者の対応をとる RDB アクセスコードの三点を生成するコード生成器
- コード生成器が生成したコードと一連のライブラリ群による，ORM 機能を実現するコード

DBPowder-mdl に変更が加えられた際には，それに合わせて既存のコードは上書きされる．但し以下の例外がある．

- 永続化ラッパークラスのコード *dw* は generation gap デザインパターンにより保持される．DBPowder は *dw* を生成するが変更を加えない．したがって応用開発者は，コード生成器の動作を気にせずに *dw* にコードを追記できる．
- 関係スキーマや DBPowder-mdl の変更により，DBPowder-mdl が生成する *rs* の内容と現在の関係スキーマの内容が一致しなくなった場合には，コードを再生成すると該当するテーブルも再生成され，既存のデータは

消去される．但しデータの消去を伴う変更の際には，実行可否の確認が事前にとられる．

- データの消去を伴う関係スキーマの変更が不可の場合に，変更後のテーブル定義をダンプする機能がある．設計開発者が変更前後のテーブル定義を比較して定義の同期をとることで，DBPowder によるテーブルの再生成や既存のデータの消去を回避できる．
- テーブル定義が DBPowder-mdl から消えた際のテーブルの削除は，DBPowder では行わない．削除が必要な場合には設計開発者が手動で実行する．

当システムの付属機能として，データの CRUD（生成，検索，更新，および削除）機能をもつウェブアプリケーションを構築する機能を備える．この機能を 第 6 章 の応用事例で用いた．この機能を構築するために，Tomcat 6.0.20 [5] と Struts 1.3.5 [4] を使用した．

5.5　性能評価

5.5.1　OO7 ベンチマークの導入

OO7 ベンチマーク [17]（以下，OO7）を用いて DBPowder ORM プロトタイプシステムの性能評価を実施した．OO7 は OODB 用に設計されたベンチマークであり，CAD/CAM/CASE アプリケーションを連想させるデータスキーマと，それに対するクエリやデータ操作の実行シナリオを保持している．本評価では OO7 を，ORM で用いる永続化クラスに適用した．OO7 ベンチマークのスキーマとデータの件数を 図 5.4 に示す．データセットは Large，Medium，Small の三種類があり，それに応じてデータ件数も三種類となる．最大件数が格納され

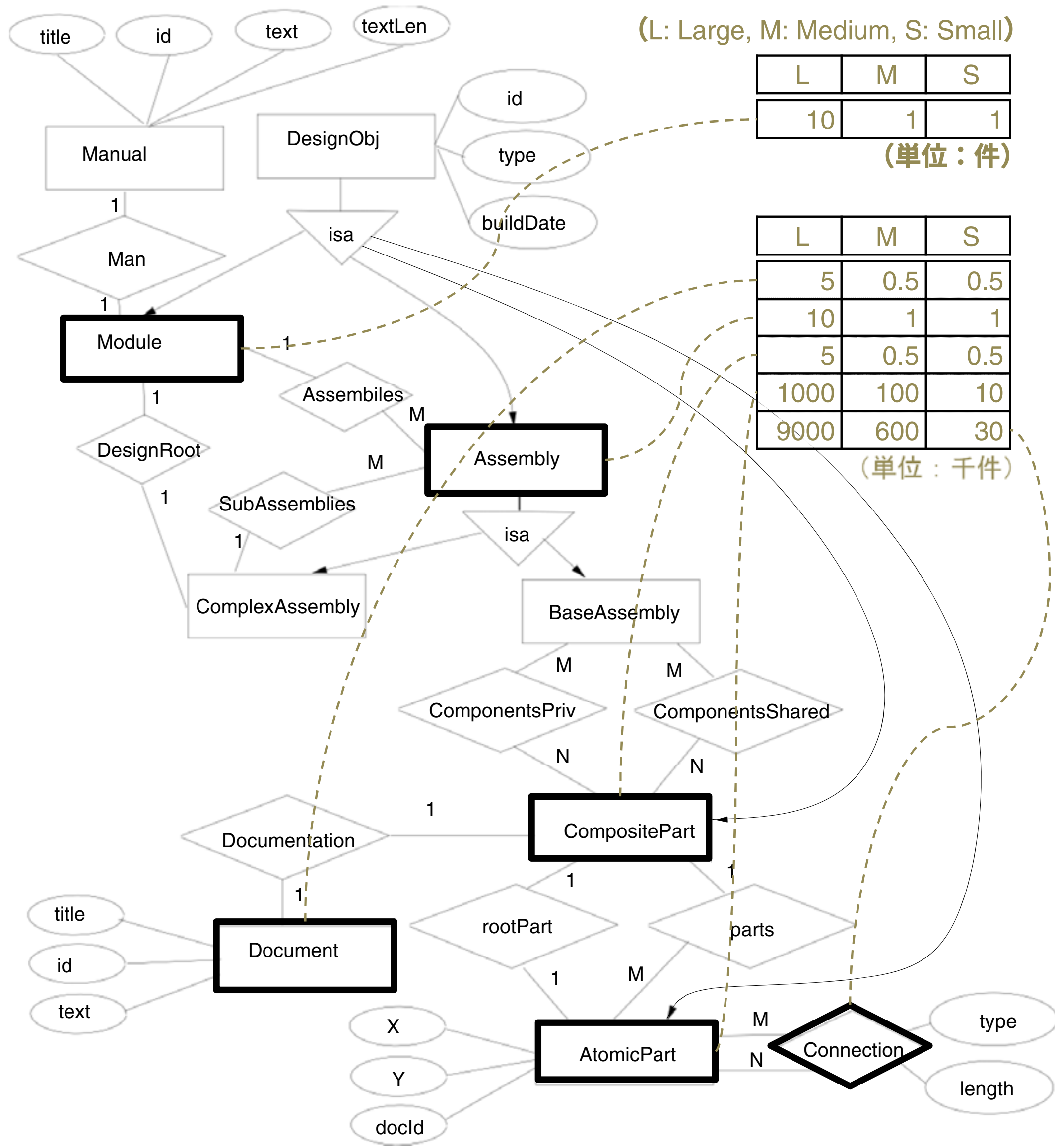


図 5.4: OO7 ベンチマーク [18, 17] のデータ件数

ているのは関連 Connection に対応するテーブルやクラスであり，Large 900 万件，Medium 60 万件，Small 3 万件となっている．

クエリやデータ操作の実行として八種類の Query シナリオおよび 11 種類の Traversal シナリオが用意されており，Q1，Q2，Q3，Q4，Q5，Q6，Q7，Q8 および T1，T2a，T2b，T2c，T3，T4，T5，T6，T7，T8，T9 と名付けられている．

AtomicPart を検索するシナリオとして，Q1，Q2，Q3，Q7，Q8 の五種類がある．Q1 は，存在する ID を 10 件ランダムに指定し，その ID で AtomicPart を検索する．Q2 は，直近から 1%の日付の範囲を指定して，AtomicPart を検索する．Q3 は，直近から 10%の日付の範囲を指定して，AtomicPart を検索する．Q7 は，全件の AtomicPart を検索する．Q8 は，AtomicPart.docId に合致する全件の Document を検索する，直接の関連をもたない AtomicPart と Document を対象としたアドホッククエリを実施する．

BaseAssembly を検索するシナリオとして，Q4，Q5 の二種類がある．Q4 は，Document の題名を 100 件生成し，それと関連をもつ BaseAssembly を検索する．Q5 は，BaseAssembly と それにひもづく CompositePart のビルド日時を各々比較し，CompositePart が後であるものを検索する．

Assembly クラスを始点とした走査のシナリオとして，T1，T2a，T2b，T2c，T6，T3，T4，T5 の八種類がある．T1 は，Assembly を始点として BaseAssembly が現れるまで ComplexAssembly を何度も走査する．その後，BaseAssembly から AtomicPart を訪問し，AtomicPart 内を走査する．そして，全ての AtomicPart を訪問するまで走査を続ける．T2a，T2b，T2c は，AtomicPart クラスの属性値（x，y）を交換する更新を実施しつつ，T1 と同じ走査を実施する．このうち T2a ではクラス取得時のみ更新を行い，T2b ではクラス取得時と走査時に更新を行い，T2c ではクラス取得時と走査時にそれぞれ 4 回更新を行う．T6 は T1 の一部のみを実行し，AtomicPart に到達したら走査を終了する．T3 は，Module, Assembly, CompositePart, AtomicPart クラスの buildDate 属性の値につ

いて加減算を繰り返しつつ，T1 と同様の走査を実施する．T4 は，Assembly クラスを始点として CompositePart がもつ Document までオブジェクトを走査し，辿り着いた Document 上から ‘I’ の文字を探す．T5 は，Document 上のテキストを更新しつつ T4 と同様の走査を実施する．

Manual を走査するシナリオとして T8，T9 の二種類がある．T8 は，マニュアル内文書から，’i’ の出現回数を検査する．T9 は，マニュアル内文書から，最初と最後の文字が等しいかどうかを検査する．

その他のシナリオとして Q6, T7 がある．Q6 では，全ての Module を検索し，各々にひもづいた Assembly を全て訪問する．訪問の際に，Assembly よりも新しい日付のついた CompositePart があるかをチェックする．チェック結果はキャッシュしておき，同じチェックが何度も走らないようにする．Assembly 内を走査し，Assembly の訪問と CompositePart との日付チェックを繰り返す．このような訪問をし尽くしたときに Q6 は終了する．T7 は T1 の逆方向の走査を行うシナリオである．T7 では，AtomicPart を一つランダムに抽出し，それにひもづく CompositePart P から走査を開始する．P から，P を参照するすべての BaseAssembly を走査し，そのそれぞれの BaseAssembly に対して，DesignRoot に到達するまで Assembly を上方向に走査する．

5.5.2　評価の内容と方法

本評価では，OO7 ベンチマークのシナリオのうち，Q1, Q2, Q3, Q4, Q5, Q7 と，T1, T2a, T2b, T2c, T6, T8, T9 を実施した．Q6, Q8, T3, T4, T5, T7 の実施を見送った理由は以下の通りである．

- DBPowder はアドホッククエリをサポートしないため，Q8 を実施しない．

- 本評価はデータベースの性能評価が目的ではないので，インデックスの更新に関する性能評価を目的とした T3 を実施しない．

- シナリオから考察して T1 と性質が類似している T4 と，T1 の逆方向走査である T7 を実施しない．また T2 と類似している T5 を実施しない．なお T4，T5，T7 については，OO7 の論文 [17] 中に，特筆すべき結果は得られなかったと報告されている．

- T1 と類似した動きをする Q6 を実施しない．

本評価の適用対象は DBPowder，Hibernate，および RoR である．OO7 の Hibernate による実装が公開されている [46]．本評価ではこのプログラムを DBPowder と RoR に移植し，DBPowder，Hibernate，RoR でシナリオごとに処理時間を比較した．また，各々の実行についてオブジェクトの取得件数や更新回数を比較し，また 試験用データの Tiny における T2a シナリオを実行してオブジェクトの訪問履歴を記録し比較することで，DBPowder，Hibernate，RoR のそれぞれがふるまいとして同じかどうかを確認した．但し Q4 は，生成した題名により取得件数が変わるシナリオであるため，取得件数の比較対象から除外した．また訪問履歴の比較では，オブジェクトのコレクションを取得した時には，走査前に予め ID 順でソートを実施してから，履歴を取ることとした[3]．

各々のシナリオについて 5 回試行し，初めの 1 回を cold run, 2 回目から 4 回目について平均処理時間を hot run として集計した．なお OO7 の論文 [17] の指示通り，5 回目の試行については集計に加えないこととした．使用した計算機環境は，Opteron 6128 8 コア 2CPU，48GB メモリ，Ubuntu 12.04.2 LTS である．

[3]関係データベースでは，集合の取得時にその順序性は保証されない．したがって訪問履歴を比較するためには，別の方法で順序性を保証する必要がある．

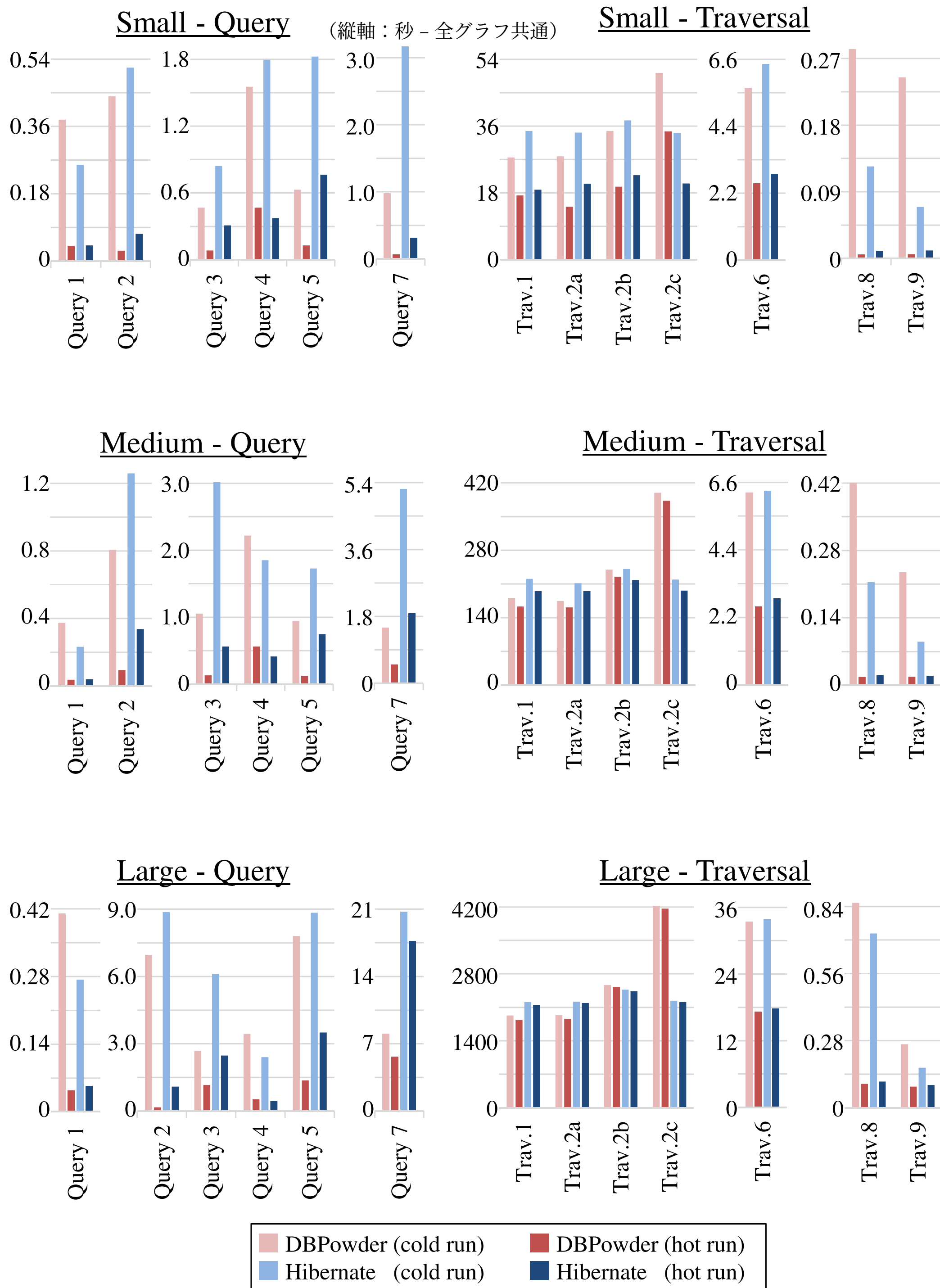


図 5.5: 性能評価の結果：OO7 ベンチマークで計測した，DBPowder と Hibernate の処理時間の比較

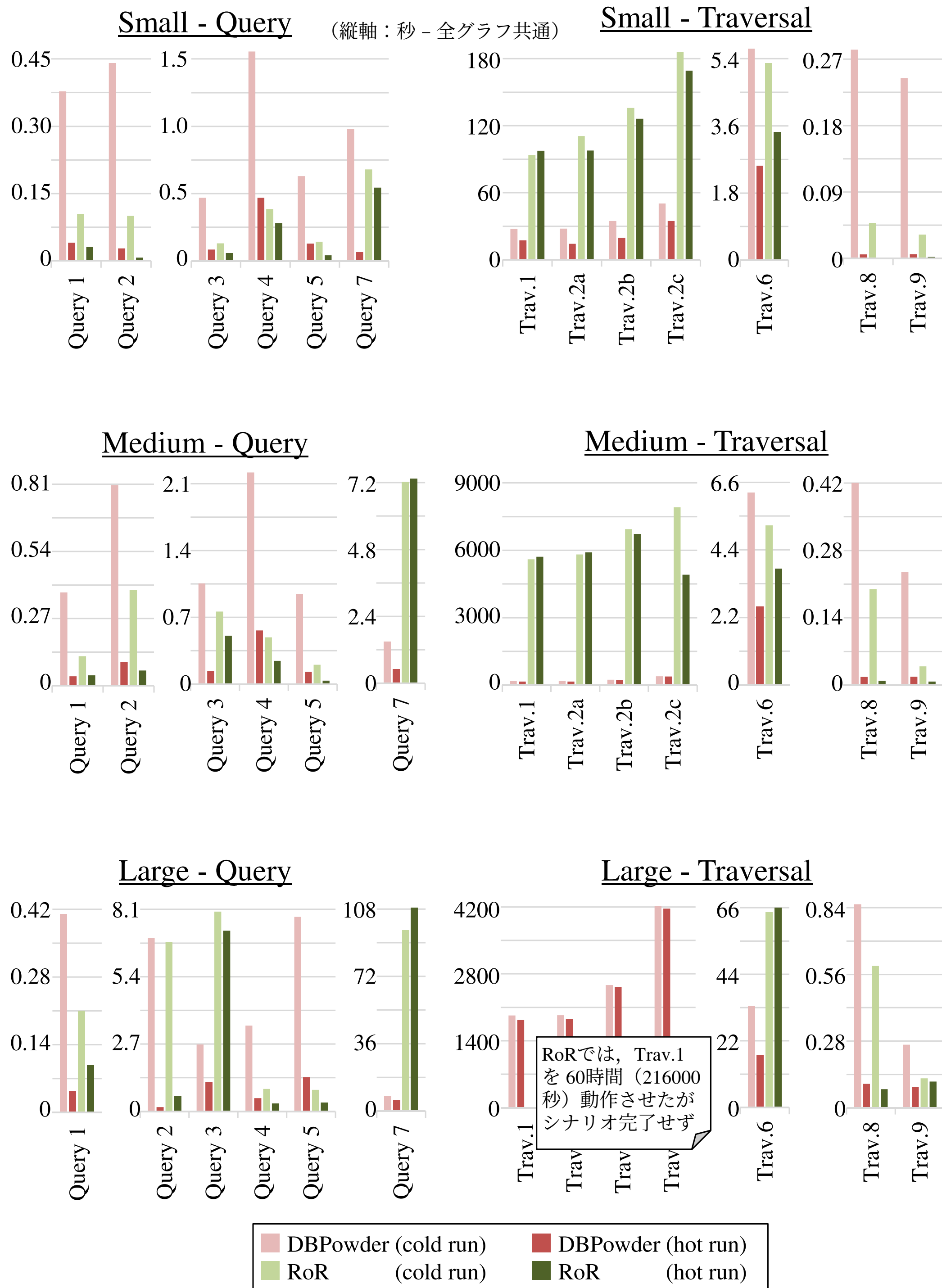


図 5.6: 性能評価の結果：OO7 ベンチマークで計測した，DBPowder と Ruby on Rails (RoR) の処理時間の比較

5.5.3 評価結果

5.5.2 節で示した評価のうち取得件数，更新回数，および Tiny-T2a シナリオの訪問履歴について，各々のシナリオで DBPowder，Hibernate，RoR を比較したところ，結果が一致することを確認できた．以下，処理時間の比較結果について示す．

図 5.5 と図 5.6 に，処理時間を比較した結果を示す．図 5.5 は DBPowder と Hibernate の比較結果，図 5.6 は DBPowder と RoR の比較結果である．図中，Small，Medium，Large は図 5.4 で示した件数のカテゴリであり，各々のグラフについて縦軸は処理時間（秒），横軸はシナリオ名である．棒グラフの赤色は DBPowder，青色は Hibernate，緑色は RoR，淡色は cold run，濃色は hot run である．

DBPowder と Hibernate の比較（図 5.5）では，多くのシナリオについて DBPowder のほうが処理時間が短い結果となった．Hibernate のほうが処理時間が短かったシナリオは，Q1（cold のみ），Q4，T2c，T8，T9 であった．

Hibernate のほうが処理時間が短いシナリオのうち Q1，Q4，T8，T9 は，RDB へ発行される 1 回のクエリあたりの取得件数が少ないシナリオである．DBPowder では SQL 文を生成するロジックで非効率な箇所が残っており，1 回のクエリあたりの取得件数が少ないシナリオでは，この箇所がボトルネックになったと考えられる．残りの T2c について，T1, T2a, T2b, T2c では走査シナリオは同一だが，T1 では値の更新無し，T2abc では値の更新があり a-b-c の順に更新回数が増えるシナリオとなっている．DBPowder では値の更新に関係するメソッドを発行する度に RDB への SQL 発行が発生するが，Hibernate では複数回の更新 SQL をまとめて発行するようになっている．T2c において Hibernate のほうが処理時間が短くなったのは，このように T2c が更新回数の多いシナリオで

あったことが原因と考えられる．残りの Q1（Hot），Q2，Q3，Q5，Q7，T1，T2a，T2b では DBPowder のほうが処理時間が短い結果を示した．

これらの結果を総合すると，DBPowder と Hibernate の処理時間の比較では，多くの場合について DBPowder のほうが処理時間が短く，またほぼ全てについて，DBPowder の性能が著しく劣ることはないことを示せた．

次に DBPowder と RoR の比較（図 5.6）では，Q1，Q4，Q5，T8，T9，および Small の Q2 と Q3 で，RoR のほうが処理時間が短い結果となった．Q2 と Q3 では，Medium-Large と件数が増えるにつれ DBPowder の処理時間が比較して短くなり，Large では DBPowder のほうが短くなった．残りの Q7，T1，T2a，T2b，T2c，T6 では，DBPowder のほうが処理時間が短い結果となり，その差は Medium-Large と件数が増えるにつれ拡がった[4]．

RoR では，`SELECT *` のようなテーブル上の属性名を指定しない単純な SQL をテーブル毎に発行するのを基本方針としている．一方で DBPowder や Hibernate では，SQL 毎に必要な属性名を指定する．RoR では永続化クラスとテーブルを単純な対応関係に限定しているため，SQL を発行するロジックも単純にできると考えられる．また DBPowder や Hibernate の SQL はパラメータを変数化して SQL を準備（Prepare Statement）したのちに変数を送り込むため，1 クエリ毎に RDB との通信が 2 回発生しているのに対し，RoR ではパラメータを SQL に直接埋め込んでいる．この 2 点が，Q1，Q4，T8，T9 のような RDB へ発行される 1 回のクエリあたりの取得件数が少ないシナリオにおいて，RoR

[4]なお Large の T1 について，約 60 時間シナリオを動作させたが完了できず，DBPowder や Hibernate の処理時間と比べて少なくとも 100 倍以上の差が生じた．Large の T2a，T2b，T2c は T1 に更新処理を加えたシナリオであることから，RoR が DBPowder や Hibernate と比べて著しく処理時間を要すると予想される．従って，RoR について Large の T2a，T2b，T2c シナリオは実施していない．また Medium の T2c cold run では，T2a や T2b に比べて更新処理が多く実施されているにも拘わらず T2a や T2b よりも処理時間が短い結果となった．Medium の T2c cold run における RoR/DBPowder の処理時間は 12.8 倍 RoR のほうが長いため，当節における議論に影響は及ぼさないものの，原因は不明である．

の処理時間が短くなった原因と考えられる．一方で Q7，T1，T2a，T2b，T2c，T6 や Small 以外の Q2 と Q3 に見るような処理件数が 1000 件を超えるような件数の多い処理について，RoR は不得意であると考えられる．また T1，T2a，T2b，T2c，T6 では走査時にコレクションを取得する処理が発生している．n 件のコレクションを取得する際，DBPowder や Hibernate では走査元のテーブルを結合したクエリを 1 回発行するのに対し，RoR ではコレクションの主キー値のリストを取得した後にそれぞれの主キー毎にクエリを発行するため，$n+1$ 回のクエリを発行する．これが，T1，T2a，T2b，T2c，T6 において RoR の処理時間が圧倒的に長い結果をもたらしたと考えられる．特に Large の T1 については，100 倍以上の差が生じた．

これらの結果を総合すると，DBPowder と RoR の処理時間の比較では，SQL が単純かつ 1 クエリあたりの応答件数が小さい処理では，RoR のほうが処理時間が短い結果となった．これは，RoR では単純な対応関係に特化した処理を実施しているのが原因と考えられる．概ね 1000 件程度以上となる 1 クエリあたりの応答件数が大きい処理や，処理回数が多くなる走査処理では，DBPowder のほうが処理時間が短い結果となった．

5.6 まとめ

提案手法 DBPowder では，ObjectView によりアプリケーションロジックに対応した複数種類の永続化クラスを柔軟に生成できる．実用上は，この提案がシステムとして動作し，妥当な性能を示すことが重要である．第 5 章 では，本論文で提案した ORM 手法のシステムとしての実現性を明らかにし，性能評価を実施した．第 5 章 で提案した DBPowder ORM システムでは，ObjectView が複数種類の永続化クラスを柔軟に生成した際に発生する，ひとつの永続化さ

れるインスタンスが複数のオブジェクトに複製され，同一セッション内で属性値が矛盾しうる状況に対して，ActiveRecord クラスを用いたシステム構成を示し，同一セッション内において属性値の矛盾を回避できることを示した．また性能評価においては，OO7 ベンチマークを導入し，Hibernate や RoR との比較を実施した．その結果，Hibernate との比較ではベンチマークのシナリオの多くで DBPowder のほうが性能が上回り，少なくとも DBPowder の性能が著しく劣ることはないことを示せた．RoR との比較では，SQL が単純かつ 1 クエリあたりの応答件数が小さい処理では RoR のほうが処理時間が短くなったが，概ね 1000 件程度以上となる 1 クエリあたりの応答件数が大きい処理や，処理回数が多くなる走査処理では，DBPowder のほうが処理時間が短い結果となった．

第6章　応用事例

6.1　ウェブサイトの構築：サーバのセキュリティ管理ポータル

本研究で提案した DBPowder の応用事例として，DBPowder ORM プロトタイプシステム（5.4 節）を用いたウェブアプリケーション（KEKapp）を 2006 年に構築し，以来改良を重ねている．高エネルギー加速器研究機構（KEK）ではこれを，DMZ User's Portal [1] とよんでいる．DMZ User's Portal は，2006 年より KEK でのセキュリティ管理業務に利用されており [69, 100, 1]，2011 年に連携組織（J-PARC）に展開された [101]．利用者数は 2013 年 12 月現在，およそ 130 名である．

図 6.1 に DMZ User's Portal のシステム全体図を示す．DMZ User's Portal は，KEK における公開サーバ（DMZ ホスト）のセキュリティを管理するためのウェブサイトである．DMZ User's Portal の主なリソースは，ホスト管理者，DMZ ホスト，およびセキュリティレポートである．各々のホスト管理者は DMZ User's Portal 上にアカウントを保持している．ホスト管理者は DMZ User's Portal にログインすることで，登録済の DMZ ホストのセキュリティ管理を実行できる．内容は主に下記の通りである．

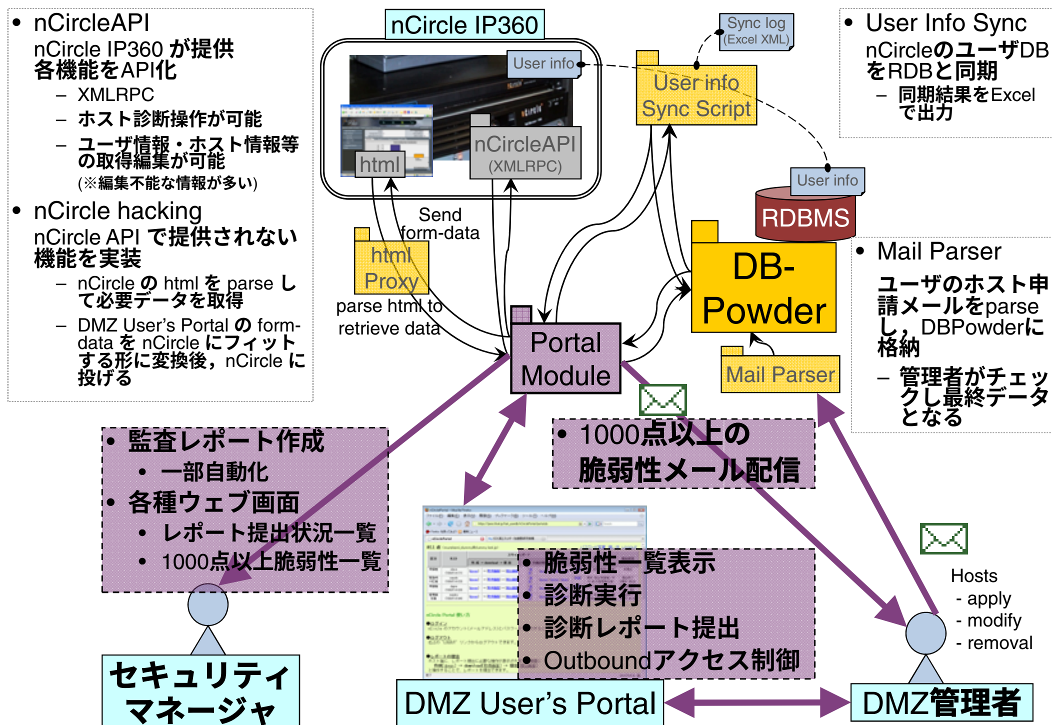


図 6.1: DMZ User's Portal（KEKapp）[1] のシステム全体図

- nCircle IP360 というセキュリティ診断装置を使って，DMZ ホストの脆弱性を診断できる．
- 診断済の DMZ ホストの脆弱性を閲覧できる．また，nCircle IP360 を使用してセキュリティレポートを作成できる．
- DMZ ホストの脆弱性情報に関するメールを受けとることができる．
- 年 1 度のセキュリティ総点検で，セキュリティレポートを提出できる．この提出されたセキュリティレポートは，セキュリティ管理者による部会で審査される．DMZ User's Portal は，セキュリティ管理者向けの，セキュリティレポートの閲覧機能を併せて提供する．

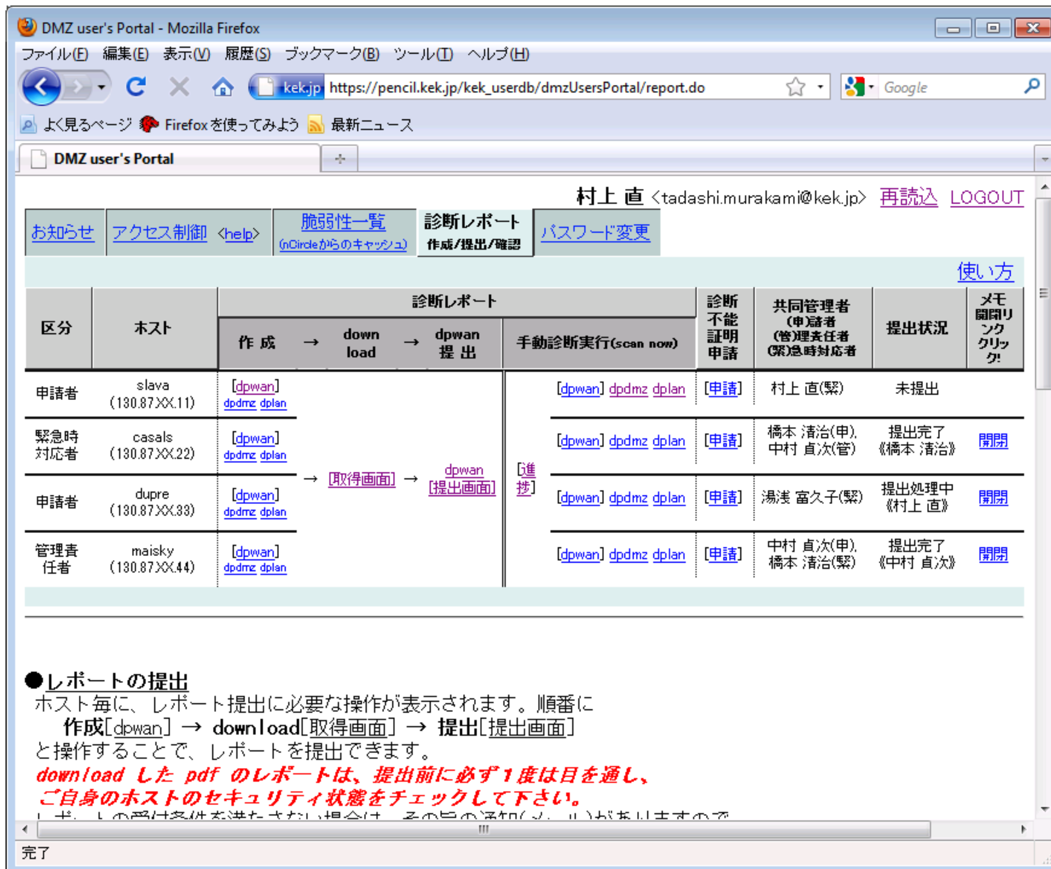


図 6.2: DMZ User's Portal のページ（一部） [102]

このように，サーバ運用の専門家向けの診断装置 nCircle IP360 を，セキュリティ総点検に必要な機能と統合しつつ，KEK での運用に合わせてホスト管理者にわかりやすい形で提供することで，セキュリティレベルの向上に役立てている [100].

DMZ User's Portal では，図 6.1 の中央部に示された Portal Module が全体を司り，主にホスト管理者ごとの DMZ ホスト保有情報の管理，ウェブユーザインタフェース（図 6.2, 6.3）による nCircle IP360 の操作や脆弱性情報などの提

図 6.3: CRUD 機能を実現するウェブページ [102]

示，DMZ ホストごとの脆弱性情報の蓄積と管理，ホスト管理者への脆弱性情報のメール送信，年 1 度のセキュリティ総点検の際のセキュリティレポートの受付，セキュリティ管理者への診断サマリ出力などを行う．

このようなサイトを継続的に運用するためには，各種データの適切な管理が必要であるほか，サイト構築後も継続的な改良が必要であり，ホスト管理者やセキュリティ管理者からのフィードバックに柔軟に対応し，サイトを進化させ

る必要がある．本事例では，DBPowder を当構築の中心モジュールとして利用することで，このような要求に応えることができた．DBPowder の単純な対応関係の簡易な実現によって，工期を短縮した迅速な開発を実施できた．その一方で，単純な対応関係をもつスキーマから複雑な対応関係をもつスキーマへ移行する開発が容易にできることから，長期運用を念頭に置いた仕様固めや改良を，サイトを運用しつつ継続的に実施できている．その効果の一部は，4.6.2 節 や 4.7.2 節 の評価結果（図 4.19，表 4.7）に示された開発に要する負担の軽減や，4.3.5 節（図 4.7，図 4.8）の例に示された単純な構造をとる複雑な対応関係を簡易に実装できる点に見いだすことができた．このような KEK での運用実績が評価され，連携組織 J-PARC への DMZ User's Portal の展開が決まった [101]．2013 年現在では，KEK と J-PARC の 2 サイトで DMZ User's Portal を運用している．

6.2　構築済みウェブサイトへの追加開発

DMZ User's Portal では，サイト構築後に継続的な改良を実施している．その際，本研究により実現した，DBPowder ORM システムが単純な対応関係の簡易な実現と複雑な対応関係の実現についてサポートを両立できる点と，単純な対応関係から複雑な対応関係に移行する開発を効率的にサポートできる点が活用されている．

たとえば 4.3.5 節（図 4.7，図 4.8）に示した DBPowder の利用例は，DMZ User's Portal のデータスキーマの一部である．ここでは，ひとつのデータスキーマに対するアプリケーションからの様々な要求に対して，DBPowder の ObjectView を用いると各々の要求に応じた様々なパターンの永続化クラスを容易に生成できることを示した．

また，4.6.2 節（図 4.19）に示されるように，単純な対応関係や複雑な対応関係をもつ新しいデータスキーマを追加する開発が容易であることは，新規開発へのハードルを下げている．また，4.7.2 節（表 4.7）に示されるように，既存のデータスキーマを利用した，複雑な対応関係による新しいアプリケーションロジックを追加した開発が容易であることも同様である．

このように DMZ User’s Portal では，追加開発を幾度も行い利用者に提供しており，セキュリティ管理や，部署ごと年度ごとに実施するセキュリティ点検における，業務上の要望に継続的に応えている．

第7章　結論

本論文では，概念モデリングに基づくO/Rマッピング（ORM）手法に関する研究として，ORMフレームワークDBPowderを提案した．

DBアプリケーションの開発について，工期を短縮した迅速な開発が強く求められる一方で，仕様を固めるのに時間を要するケースが増えている．ここで単純な対応関係を簡易に実現できるORMフレームワークを用いると，複雑な対応関係をうまく扱えない．その一方で，複雑な対応関係を実現できるORMフレームワークでは，詳細な設計を得にくい開発初期の段階から永続化クラスとテーブルの詳細な対応関係を記述する必要が生じる上に，機能拡張の際にこれを修正する必要が頻繁に出てくる．このため，迅速な開発が強く求められる開発初期の段階から開発の負担が大きくなる．

第4章ではこの問題の克服のために，概念モデリングに基づくO/Rマッピング（ORM）手法，DBPowderを提案した．DBPowderでは問題の克服のために，Extended Entity-Relationship (EER) モデルを用いた概念モデリングと，有向グラフベースでEERモデル上の実体の走査経路と利用方法を指定するObjectViewの，連携による併用方式を提案した．EERモデルにより，単純な対応関係をもつ永続化クラスと関係スキーマおよび，これの対応をとるRDBアクセスコードを簡易に生成できる．EERモデルにObjectViewを加えることで，追加的または修正的に複雑な対応関係をもつ永続化クラスを生成できる．また，EERモデルとObjectViewをコンパクトに記述できる言語DBPowder-mdl

を提案した．

第 4 章 の提案の有効性を検証するために，記述力の評価と開発に要する負担の評価を実施した．

記述力の評価では，単純な対応関係を簡易に実現できる Ruby on Rails（RoR）と複雑な対応関係を実現できる Hibernate を比較対象とし，DBPowder による ORM を記述するための言語 DBPowder-mdl の言語定義と Hibernate の ORM 記述を行う .hbm XML ファイルの文書型定義（DTD）を比較し，DBPowder の記述力を比較検証した．その結果，永続化クラスと関係スキーマの対応関係について DBPowder は，以下の三種類を除いては Hibernate と同等のサポートを得られることが確かめられた．1) 関連や継承関係をもたない複数のクラスを対応づける，2) SQL 文を永続化クラスや属性に対応づける，3) コレクションを構成するクラスから一属性を選んで配列にして，コレクションを保持するクラスの属性として対応づける．

開発に要する負担の評価では，二種類の評価を実施した．ひとつは単語数を用いた記述量の評価である．もうひとつは，複雑な対応関係に移行する開発で要した，編集の回数および編集した単語数の比較である．

単語数を用いた記述量の評価では，Hibernate では複雑な対応関係を実現できるが必要な記述量は膨大になり，Hibernate よりも DBPowder が大幅にコンパクトに ORM を記述できることが明らかとなった．この理由として，DBPower では EER モデルの導入，階層構造を基本にした記述形式，および CoC 導入の効果が大きく，同じ要素を何度も指定する必要がなく，また，文法により必要とされる記述要素をコンパクトに抑えたことが挙げられる．次に，単純な対応関係における RoR と DBPowder の比較では，文法ベースの CoC を満たさない場合に損なわれる簡易さの程度が，RoR と比べて DBPowder が限定的であることが明らかとなった．この理由として，RoR における文法ベースで必要な

記述量を下げようとするアプローチが，関係スキーマの構造が複雑な場合に有効に作用しなくなる一方で，EER モデルに基づいたデータ構造をとらえて単純な対応関係を簡易に実現する DBPowder において，複雑な関係スキーマにおける文法ベースの CoC の逸脱による影響が RoR と比べて限定的に抑えられた点が挙げられる．

複雑な対応関係に移行する開発で要した，編集の回数および編集した単語数の比較は，DBPowder と Hibernate で実施した．Hibernate において複雑な対応関係による ORM を追加するためには，記述済の ORM 定義のコピー＆ペーストが必要となる．このコピー＆ペーストの結果を編集すると目的の ORM を得ることができるが，その開発の負担は大きいことが明らかになった．それと比べて DBPowder では，個別のアプリケーションロジックのみを ObjectView で記述して，永続化クラスを効率的に追加できることが定量的に示された．この結果は，DBPowder の ObjectView が，既存のデータスキーマを用いて個別のアプリケーションロジックに適した永続化クラスを効率的に追加する枠組を提供することによりもたらされた．

上記に示した評価により，DBPowder が単純な対応関係の簡易な実現と複雑な対応関係の実現についてサポートを両立でき，かつ，単純な対応関係から複雑な対応関係に移行する開発を効率的にサポートできることが示された．

提案手法 DBPowder では，ObjectView によりアプリケーションロジックに対応した複数種類の永続化クラスを柔軟に生成できる．実用上は，この提案がシステムとして動作し，妥当な性能を示すことが重要である．第 5 章 では，本論文で提案した ORM 手法のシステムとしての実現性を明らかにし，性能評価を実施した．第 5 章 で提案した DBPowder ORM システムでは，ObjectView が複数種類の永続化クラスを柔軟に生成した際に発生する，ひとつの永続化されるインスタンスが複数のオブジェクトに複製され，同一セッション内で属性

値が矛盾しうる状況に対して，ActiveRecord クラスを用いたシステム構成を示し，同一セッション内において属性値の矛盾を回避できることを示した．また性能評価においては，OO7 ベンチマークを導入し，Hibernate や RoR との比較を実施した．その結果，Hibernate との比較ではベンチマークのシナリオの多くで DBPowder のほうが性能が上回り，少なくとも DBPowder の性能が著しく劣ることはないことを示せた．RoR との比較では，SQL が単純かつ 1 クエリあたりの応答件数が小さい処理では RoR のほうが処理時間が短くなったものの，概ね 1000 件程度以上となる 1 クエリあたりの応答件数が大きい処理や，処理回数が多くなる走査処理では，DBPowder のほうが処理時間が短い結果となった．

今後の課題としては四点が挙げられる．一点目はデータ更新処理の効率化である．現在 DBPowder では，insert/update/delete メソッドの発行毎に，RDB に SQL 文が送信される方式となっている．DBPowder のセッション管理において，トランザクションを考慮しつつ SQL をまとめて発行するなどの効率化方式を検討することで，処理速度の向上が期待できる．二点目は大規模データへの対応である．これの実現のためには，永続化クラスのインスタンス管理方式について検討が必要である．三点目はリバースエンジニアリングが挙げられる．DB アプリケーションの開発において，既に存在するデータスキーマを使用するケースは多いと考えられる．これに対応するための方式を提案し，DBPowder の一部として提供できれば，有用性の向上が期待できる．四点目は EER モデルと ObjectView の連携方式の深化が挙げられる．たとえば，RDB の特性を踏まえた物理スキーマを設計すると，処理性能の向上が期待できる．しかし DBPowder では，EER モデルのみに基づいて関係スキーマを生成するため，物理スキーマの設計が概念モデルに影響を及ぼしてしまう問題がある．この問題に対処する場合，ObjectView との連携方式についても何らかの改良が必要になると考えている．

参考文献

[1] Activity report 2012 computing research center KEK. *KEK Progress Report 2013-2*, Dec. 2013.

[2] Atul Adya, José A. Blakeley, Sergey Melnik, and S. Muralidhar. Anatomy of the ADO.NET entity framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pp. 877–888. ACM, 2007.

[3] Scott W. Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[4] Apache Software Foundation. Apache Struts Project. http://struts.apache.org/. (accessed in Dec. 2013).

[5] Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/. (accessed in Dec. 2013).

[6] Apple Computer, Inc. EOModeler user guide. https://developer.apple.com/legacy/library/documentation/WebObjects/UsingEOModeler/UsingEOModeler.pdf, May 2006. (accessed in Dec. 2013).

[7] Roland Barcia, Geoffrey Hambrick, Kyle Brown, Robert Peterson, and Kulvir Singh Bhogal. *Persistence in the enterprise: a guide to persistence technologies*. Addison-Wesley Professional, 2008.

[8] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, CT, USA, 2006.

[9] Philip A. Bernstein and Sergey Melnik. Model management 2.0: manipulating richer mappings. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pp. 1–12. ACM, 2007.

[10] Michael Blaha, William Premerlani, and Hwa Shen. Converting OO models into RDBMS schema. *IEEE Software*, Vol. 11, No. 3, pp. 28–39, 1994.

[11] José A. Blakeley, David Campbell, S. Muralidhar, and Anil Nori. The ADO.NET entity framework: making the conceptual level real. *SIGMOD Record*, Vol. 35, No. 4, pp. 32–39, 2006.

[12] Marco Brambilla, Sara Comai, Piero Fraternali, and Maristella Matera. *Designing Web Applications with WebML and WebRatio*, chapter 9, pp. 221–261. Human-Computer Interaction Series. Springer London, London, 2008.

[13] Luca Cabibbo. Objects meet relations: On the transparent management of persistent objects. In *CAiSE*, Vol. 3084 of *Lecture Notes in Computer Science*, pp. 429–445. Springer, 2004.

[14] Luca Cabibbo and Antonio Carosi. Managing inheritance hierarchies in object/relational mapping tools. In *CAiSE*, Vol. 3520 of *Lecture Notes in Computer Science*, pp. 135–150. Springer, 2005.

[15] Luca Cabibbo and Roberto Porcelli. M^{2}ORM^2 a model for the transparent management of relationally persistent objects. In *Database Programming Languages, 9th International Workshop, DBPL 2003, Potsdam, Germany, September 6-8, 2003, Revised Papers*, Vol. 2921 of *Lecture Notes in Computer Science*, pp. 166–178. Springer, 2003.

[16] Michael J. Carey and David J. DeWitt. Of objects and databases: A decade of turmoil. In *VLDB 96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pp. 3–14. Morgan Kaufmann, 1996. invited paper, author of best impact paper of VLDB 86.

[17] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pp. 12–21, New York, NY, USA, 1993. ACM.

[18] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark (version of january 21, 1994). http://pages.cs.wisc.edu/~dewitt/includes/benchmarking/oo7.pdf, 1994. (accessed in Dec. 2013).

[19] R. G. G. Cattell and Douglas K. Barry. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[20] Stefano Ceri, Marco Brambilla, and Piero Fraternali. The history of WebML lessons learned from 10 years of model-driven development of web applications. In *Conceptual Modeling: Foundations and Applications - Essays in Honor of John Mylopoulos*, Vol. 5600 of *Lecture Notes in Computer Science*, pp. 273–292. Springer, 2009.

[21] Stefano Ceri, Florian Daniel, Maristella Matera, and Federico Michele Facca. Model-driven development of context-aware web applications. *ACM Trans. Internet Techn.*, Vol. 7, No. 1, 2007.

[22] Nicholas Chen. Convention over Configuration. http://softwareengineering.vazexqi.com/files/pattern.html, Nov 2006. (accessed in Dec. 2013).

[23] Peter Pin-Shan Chen. The entity-relationship modeltoward a unified view of data. *ACM Trans. Database Syst.*, Vol. 1, No. 1, pp. 9–36, 1976.

[24] Christian Bauer. Java persistence with hibernate, browse & download examples. Available from http://jpwh.org/examples/. (accessed in Dec. 2013).

[25] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, Vol. 13, No. 6, pp. 377–387, June 1970.

[26] William R Cook, Robert Greene, Patrick Linskey, Erik Meijer, Ken Rugg, Craig Russell, Bob Walker, and Christof Wittig. Objects and databases: State of the union in 2006. In *Companion to the 21st ACM SIGPLAN symposium*

on Object-oriented programming systems, languages, and applications, pp. 926–928. ACM, 2006.

[27] William R. Cook and Ali H. Ibarahim. Integrating programming language & databases: What's the problem? *ODBMS.ORG, Expert Article*, September 2006.

[28] George Copeland and David Maier. Making smalltalk a database system. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pp. 316–325, New York, NY, USA, 1984. ACM.

[29] Microsoft Corporation. ADO.NET entity data model tools. http://msdn.microsoft.com/en-US/library/vstudio/bb399249(v=vs.100).aspx. (accessed in Dec. 2013).

[30] Microsoft Corporation. C# language specification 5.0. http://www.microsoft.com/en-us/download/details.aspx?id=7029. (accessed in Dec. 2013).

[31] Microsoft Corporation. Visual Basic language specification 11.0. http://www.microsoft.com/en-us/download/details.aspx?id=15039. (accessed in Dec. 2013).

[32] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, Vol. 39, No. 2, pp. 25–31, Feb 2006.

[33] Edd Dumbill. Ruby on Rails: An interview with David Heinemeier Hansson. http://www.oreillynet.com/pub/a/network/2005/08/30/ruby%2Drails%2Ddavid%2Dheinemeier%2Dhansson.html, Aug 2005. (accessed in Dec. 2013).

[34] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 2nd Edition*. Benjamin/Cummings, 1994.

[35] Christian Fahrner and Gottfried Vossen. A survey of database design transformations based on the entity-relationship model. *Data & Knowledge Engineering*, Vol. 15, No. 3, pp. 213–250, 1995.

[36] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, Boston, MA, USA, November 2002.

[37] Piero Fraternali. Tools and approaches for developing data-intensive web applications: A survey. *ACM Comput. Surv.*, Vol. 31, No. 3, pp. 227–263, September 1999.

[38] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[39] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification. Java SE 7 Edition*. Addison-Wesley Professional, Boston, MA, USA, 2013.

[40] David Heinemeier and et al. Ruby on Rails. http://www.rubyonrails.org/. (accessed in Dec. 2013).

[41] Joseph M. Hellerstein and Michael Stonebraker. *Readings in Database Systems, 4th Edition*. MIT Press, 2005.

[42] Jean-Marc Hick and Jean-Luc Hainaut. Strategy for database application evolution: The DB-MAIN approach. In *Conceptual Modeling - ER 2003, 22nd International Conference on Conceptual Modeling, Chicago, IL, USA, October 13-16, 2003, Proceedings*, Vol. 2813 of *Lecture Notes in Computer Science*, pp. 291–306. Springer, 2003.

[43] Richard Hull and Masatoshi Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pp. 455–468, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[44] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, Boston, MA, USA, October 1999.

[45] IBM Corporation. DB2 database software. http://www.ibm.com/software/data/db2/. (accessed in Dec. 2013).

[46] Ibrahim, Ali and Cook, William R. OO7 object/database benchmark. http://sourceforge.net/projects/oo7/. (accessed in Dec. 2013).

[47] ISO/IEC. ISO/IEC 9075-*:1999 Information technology – Database languages – SQL – Part 1,2,3,4,5.

[48] ISO/IEC. ISO/IEC 9075:1992 Information technology – Database languages – SQL.

[49] JBoss Inc. Hibernate. http://www.hibernate.org/. (accessed in Dec. 2013).

[50] Jean-Philippe Lang and et al. Redmine. http://www.redmine.org/. (accessed in Dec. 2013).

[51] Trevor H. Jones and Il-Yeol Song. Binary equivalents of ternary relationships in entity-relationship modeling: A logical decomposition approach. *Journal of Database Management*, Vol. 11, No. 2, pp. 12–19, 2000.

[52] J.Rode, Y.Bhardwaj, M.A.Pérez-Qui nones, M.B.Rosson, and J.Howarth. As easy as "Click": End-user web engineering. *Lecture notes in computer science, D.Lowe and M.Gaedke(Eds.): ICWE2005, LNCS 3579*, pp. 478–488, July 2005.

[53] Alan C. Kay. History of programming languages—ii. chapter The Early History of Smalltalk, pp. 511–598. ACM, New York, NY, USA, 1996.

[54] David Kensche, Christoph Quix, Mohamed Amine Chatti, and Matthias Jarke. GeRoMe: A generic role based metamodel for model management. *J. Data Semantics*, Vol. 8, pp. 82–117, 2007.

[55] David Kensche, Christoph Quix, Xiang Li, and Sandra Geisler. Solving ORM by MAGIC: Mapping generation and composition. In *Objects and Databases - Third International Conference, ICOODB 2010, Frankfurt/Main, Germany, September 28-30, 2010. Proceedings*, Vol. 6348 of *Lecture Notes in Computer Science*, pp. 118–132. Springer, 2010.

[56] Charly Kleissner. Enterprise objects framework, a second generation object-relational enabler. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pp. 455–459. ACM Press, 1995.

[57] Thomas Kowark, Robert Hirschfeld, and Michael Haupt. Object-relational mapping with SqueakSave. In *Proceedings of the International Workshop on Smalltalk Technologies*, pp. 87–100. ACM, 2009.

[58] Neal Leavitt. Whatever happened to object-oriented databases? *IEEE Computer*, Vol. 33, No. 8, pp. 16–19, 2000.

[59] Fakhar Lodhi and Muhammad Ahmad Ghazali. Design of a simple and effective object-to-relational mapping technique. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11-15, 2007*, pp. 1445–1449. ACM, 2007.

[60] Rick Long, Robert Hain, and Mark Harrington. *IMS Primer*. IBM redbooks. 2000.

[61] Magento Inc. Magento. http://www.magentocommerce.com/. (accessed in Dec. 2013).

[62] David Maier. Representing database programs as objects. In François Bancilhon and Peter Buneman, editors, *Advances in Database Programming Languages, Papers from DBPL-1, September 1987, Roscoff, France*, pp. 377–386. ACM Press / Addison-Wesley, 1987.

[63] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[64] Sergey Melnik, Atul Adya, and Philip A. Bernstein. Compiling mappings to bridge applications and databases. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 461–472, New York, NY, USA, 2007. ACM.

[65] Sergey Melnik, Atul Adya, and Philip A. Bernstein. Compiling mappings to bridge applications and databases. *ACM Trans. Database Syst.*, Vol. 33, No. 4, 2008.

[66] Paolo Merialdo, Paolo Atzeni, and Giansalvatore Mecca. Design and development of data-intensive web sites: The Araneus approach. *ACM Trans. Inter. Tech.*, Vol. 3, No. 1, pp. 49–92, 2003.

[67] Microsoft Corporation. Microsoft SQL Server. http://www.microsoft.com/sqlserver/. (accessed in Dec. 2013).

[68] Stefano Mostarda, Marco De Sanctis, and Daniele Bochicchio. *Entity Framework 4 in Action*. Manning Publications Co., 2011.

[69] Tadashi Murakami, Fukuko Yuasa, and Toshiaki Kaneko. Vulnerability management by the integration of security resources and devices with DBPowder. *Grid Camp and HEPiX Fall 2008, Taipei, Taiwan*, 2008.

[70] MySQL AB. MySQL Documentation. http://dev.mysql.com/doc/. (accessed in Dec. 2013).

[71] Badri Narasimhan, Shamkant B. Navathe, and Sundaresan Jayaraman. On mapping ER models into OO schemas. In Ramez Elmasri, Vram Kouramajian, and Bernhard Thalheim, editors, *Entity-Relationship Approach - ER'93, 12th International Conference on the Entity-Relationship Approach, Arlington, Texas, USA, December 15-17, 1993, Proceedings*, Vol. 823 of *Lecture Notes in Computer Science*, pp. 402–413. Springer, 1993.

[72] Object Management Group. Unified modeling language (UML). http://www.uml.org/. (accessed in Dec. 2013).

[73] Object Management Group. MDA specifications. http://www.omg.org/mda/specs.htm, Oct 2013. (accessed in Dec. 2013).

[74] T William Olle. *The CODASYL Approach to Data Base Management*. John Wiley & Sons, Inc., 1978.

[75] Oracle Corporation. Enterprise JavaBeans technology. http://www.oracle.com/technetwork/java/index-jsp-140203.html. (accessed in Dec. 2013).

[76] Oracle Corporation. Java persistence API. http://www.oracle.com/technetwork/jp/java/javaee/tech/persistence-jsp-140049.html. (accessed in Dec. 2013).

[77] Oracle Corporation. Java SE. http://www.oracle.com/technetwork/java/javase/. (accessed in Dec. 2013).

[78] Oracle Corporation. Oracle Database. http://www.oracle.com/. (accessed in Dec. 2013).

[79] P.Fratermali and P.Paolini. Model-driven development of web applications: The Autoweb system. *ACM Transactions on Information Systems*, Vol. 28, No. 4, pp. 323–382, Oct 2000.

[80] Stephan Philippi. Model driven generation and testing of object-relational mappings. *Journal of Systems and Software*, Vol. 77, No. 2, pp. 193–207, 2005.

[81] PostgreSQL. http://www.postgresql.org/. (accessed in Dec. 2013).

[82] William J. Premerlani and Michael R. Blaha. An approach for reverse engineering of relational databases. In *WCRE*, pp. 151–160, 1993.

[83] Red Hat, Inc. Hibernate community index page. https://forum.hibernate.org/viewtopic.php?f=1&t=939610, 2005. (accessed in Dec. 2013).

[84] W.W. Royce. Managing the development of large software systems. In *proceedings of IEEE WESCON*, Vol. 26. Los Angeles, 1970.

[85] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[86] Bernhard Rumpe. Executable modeling with UML. a vision or a nightmare? *Issues & Trends of Information Technology Management in Contemporary Associations*, pp. 697–701, 2002.

[87] Salesforce.com, inc. Salesforce.com completes acquisition of heroku. http://www.salesforce.com/company/news-press/press-releases/2011/01/110104.jsp, 2011. (accessed in Dec. 2013).

[88] Scott W. Ambler. Are you ready for the MDA? http://www.agilemodeling.com/essays/readyForMDA.htm, 2004. (accessed in Dec. 2013).

[89] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, pp. 19–25, Sep/Oct 2003.

[90] S.Mellor, M.Balcer, and M.J.Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, Boston, MA, USA, 2002.

[91] Sam Smoot and et al. DataMapper. http://datamapper.org/. (accessed in Dec. 2013).

[92] sourceforge.net. phpClick. http://phpclick.sourceforge.net/. (accessed in Dec. 2013).

[93] Squeak Foundation. Squeak Smalltalk. http://www.squeak.org/. (accessed in Dec. 2013).

[94] Toby J. Teorey, Dongqing Yang, and James P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, Vol. 18, No. 2, pp. 197–222, 1986.

[95] Philippe Thiran, Jean-Luc Hainaut, and Geert-Jan Houben. Database wrappers development: Towards automatic generation. In *9th European Conference on Software Maintenance and Reengineering (CSMR 2005), 21-23 March 2005, Manchester, UK, Proceedings*, pp. 207–216. IEEE Computer Society, 2005.

[96] Philippe Thiran, Jean-Luc Hainaut, Geert-Jan Houben, and Djamal Benslimane. Wrapper-based evolution of legacy information systems. *ACM Transactions on Software Engineering Methodology*, Vol. 15, No. 4, pp. 329–359, 2006.

[97] tutorialspoint. Hibernate O/R mappings. http://www.tutorialspoint.com/hibernate/hibernate_or_mappings.htm. (accessed in Dec. 2013).

[98] John Vlissides. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[99] Web Models. WebRatio. http://www.webratio.com/. (accessed in Dec. 2013).

[100] 京セラコミュニケーションシステム株式会社. 脆弱性診断・管理システム 導入事例 大学共同利用機関法人高エネルギー加速器研究機構. http://www.kccs.co.jp/products/ncircle/case/case01.html, Jul 2010. (accessed in Dec. 2013).

[101] 石川弘之, 舘明宏, 村上直. J-PARC における情報セキュリティ脆弱性リスク管理システムの開発. *JAEA-Technology 2011-030*, pp. 1–62, Feb. 2012.

[102] 村上直. DBPowder-mdl: EoD と記述力を兼備した O/R マッピング言語. 情報処理学会論文誌データベース (TOD), Vol. 3, No. 3, pp. 46–67, Sep. 2010.

[103] 北川博之. データベースシステム. 昭晃堂, 東京, 1996.

研究業績

博士論文に関する研究

査読付き学術雑誌論文

- 村上 直．DBPowder-mdl: EoD と記述力を兼備した O/R マッピング言語．情報処理学会論文誌：データベース，Vol.3，No.3 (TOD47)，pp.46–67，Sep. 2010.

査読付き国際会議論文

- Tadashi Murakami, Toshiyuki Amagasa, Hiroyuki Kitagawa. DBPowder: A Flexible Object-Relational Mapping Framework based on a Conceptual Model. *Proceedings of the 37th Annual International Computer Software & Applications Conference (COMPSAC 2013)*, Kyoto, Japan, pp.589–598, Jul. 2013.
- Tadashi Murakami. SQL-based Object-Oriented Development with DBPowder. *Proceedings of the 3rd International Conference on Complex, Intelligent and Software Intensive Systems (CISIS-2009)*, Fukuoka, Japan, pp.112–119, Mar. 2009.
- Tadashi Murakami. DBPowder-mdl: Mapping Description Language between Applications and Databases. *Proceedings of the 7th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2008)*, California, USA, pp.127–132, May. 2008.

査読付き国内会議論文

- 村上 直．DBPowder-web: RDBMS を用いたウェブアプリケーションの構築を支援するフレームワーク．第 *17* 回電子情報通信学会データ工学ワークショップ *(DEWS2006)* 論文集，pp.4A–o4，Mar. 2006.

研究会発表，諸報告

- 村上 直，天笠 俊之，北川 博之．概念モデルに基づく柔軟な O/R マッピングを実現するフレームワーク DBPowder の提案．研究報告データベースシステム（*DBS*），2012-DBS-154(13)，pp.1–8，Aug. 2012.
- 石川 弘之，舘 明宏，村上 直．J-PARC における情報セキュリティ脆弱性リスク管理システムの開発．*JAEA-Technology 2011-030*，62 頁，Feb. 2012.
- Tadashi Murakami, Fukuko Yuasa, Toshiaki Kaneko. Vulnerability Management by the Integration of Security Resources and Devices with DBPowder. *Grid Camp and HEPiX Fall 2008*, Oct. 2008.
- 村上 直．DBPowder: RDBMS 活用フレームワーク 並木・齋藤・高田・黒川４ PM 合同成果報告会．*IPA* 未踏ソフトウェア創造事業，2006 年度下期並木 PM，Aug. 2007.
- 村上 直．DBPowder: RDBMS 活用フレームワーク 黒川・千葉・並木・斎藤４ PM 合同キックオフ．*IPA* 未踏ソフトウェア創造事業，2006 年度下期並木 PM，Dec. 2006.
- 村上 直．リレーショナルスキーマを起点としたアプリケーション開発の一手法．信学技報，DE2006-128，DC2006-35，pp.13–18，Oct. 2006.

その他の論文

査読付き国内会議論文

- 柴田 章博，村上 直．インターネット通信の DFA に基づく共分散分析．第 *16* 回交通流のシミュレーションシンポジウム，pp.17–20，Dec. 2010.
- 柴田 章博，村上 直．インターネット通信のサービス別揺ぎの DFA 解析．第 *15* 回交通流のシミュレーションシンポジウム，pp.65–68，Dec. 2009.
- 柴田 章博，村上 直．インターネット接続の揺らぎの DFA 解析．第 *14* 回交通流のシミュレーションシンポジウム，pp.77–80，Nov. 2008.

研究会発表

- 村上 直，倉光 君郎．SQL などをマッピング定義として用いた軽量な ORM 機構．第 *19* 回電子情報通信学会データ工学ワークショップ *(DEWS2008)* 論文集，pp.C10-2，Mar. 2008.