情報処理学会論文誌　データベース　Vol. 3　No. 3　46–67 (Sep. 2010)

DBPowder-mdl：EoDと記述力を兼備したO/Rマッピング言語

村　上　　直[†1]

O/R マッピングフレームワークを用いることで，リレーショナルデータベース（RDB）を用いたアプリケーション開発の困難さを軽減できる．O/R マッピングでは，開発の容易さ（Ease of Development: EoD）とデータモデルの記述力が重要である．既存の O/R マッピングフレームワークでは，EoD と記述力がトレードオフの関係となっており，両方を兼備するのは困難である．すなわち，EoD を重視した場合はデータモデルの記述力を犠牲にせざるをえず，記述力を重視した場合は EoD を犠牲にせざるをえない．本研究では，EoD と記述力を兼備した O/R マッピング言語 DBPowder-mdl を提案する．DBPowder-mdl は，リレーショナルモデル（RM）の記法とオブジェクトモデル（OM）の記法を提供する．設定より規約（Convention over Configuration: CoC）を推し進めることで設計記述量を減らすことができる一方，設計内容を明示的に記述することで高い記述力を得ることもできる．RM または OM のうち，その一部分を設計すれば CoC により多くが補われる．いっぽう，RM と OM の双方を制御する記法も提供しており，柔軟な O/R マッピングも実現可能である．DBPowder-mdl は，EoD と記述力を兼備することに成功した．これにより，開発のフェーズやスキーマの部分に応じて EoD と記述力から受ける恩恵を使い分けることを可能とした．

DBPowder-mdl: EoD Featured and Much Descriptive Domain Specific Language for O/R Mapping

Tadashi Murakami[†1]

O/R mapping frameworks reduce difficulties to develop applications with relational databases (RDB). While they are expected to take advantages of the ease of development (EoD) and enough descriptive power of the data model, it is difficult to take both of them since there are trade-offs between them. In this paper, we propose DBPowder-mdl: EoD featured and much descriptive domain specific language for O/R mapping. DBPowder-mdl describes a relational model (RM) and an object model (OM). DBPowder-mdl has the feature of *Convention over Configuration (CoC)* that reduces the amount of design and description. In contrast, DBPowder-mdl supports the style of explicit description which brings enough descriptive power. The result of O/R mappings can be derived from either of RM or OM since DBPowder-mdl complements the omissions. DBPowder-mdl also offers the flexible ways to describe with no omissions of RM and OM. In conclusion, DBPowder-mdl succeeded to take both advantages of EoD and enough descriptive power of the data model as the situations demand.

1. はじめに

永続データの管理を目的として，リレーショナルデータベース（RDB）は広く用いられている．RDB は複雑大量のデータを効果的に管理できる一方で，データモデルとして採用されるリレーショナルモデル（以下，RM と略記）とプログラミング言語との乖離がしばしば問題となる（インピーダンスミスマッチ）．プログラミング言語で広く用いられるオブジェクト指向の手法でそのままデータモデルを設計する場合も，やはり RM に対するインピーダンスミスマッチを解決する必要がある．

O/R マッピングフレームワークを用いることで，このインピーダンスミスマッチを軽減することができ，開発の困難さを軽減できる．既存の O/R マッピングフレームワークには様々な特徴がある．

(1) RM の設計を開発の出発点とするもの
(2) オブジェクトモデル（以下，OM と略記）の設計を開発の出発点とするもの
(3) RM，OM，および 2 者のマッピング（以下，RM と OM のマッピングを mOR と略記）を最初からすべて設計記述するもの
(4) 設定より規約（Convention over Configuration: CoC）[9),22)] を推し進めることで設計記述量を減らそうとするもの
(5) 設計内容を明示的に記述することで高い記述力を得ようとするもの

これらの複数項目を同時に満たすのは困難である．たとえば，RM と OM は多くの箇所で密接に関わりを持つにもかかわらず，RM の設計を開発の出発点とするフレームワークは，OM の設計をフレームワークにフィードバックできないものが多い．また，CoC により必要な設計記述量が減れば開発の容易さ（Ease of Development: EoD）におおいに貢献

†1 高エネルギー加速器研究機構計算科学センター
Computing Research Center, High Energy Accelerator Research Organization (KEK)

するが，この CoC は一方で記述力にとっては足かせとなるケースが多い．

本研究では，EoD と記述力を兼備した O/R マッピング言語 DBPowder-mdl とその関連フレームワークを提案する．DBPowder-mdl は，1 つの言語で (1)，(2)，(3)，(4)，(5) の特徴を使い分けた設計と開発を可能とする．また，(1)，(2)，(3) の要素を DBPowder-mdl という 1 つの言語に集約することで，RM と OM の柔軟なラウンドトリップ開発を可能とする．CoC を推し進めることで設計記述量を減らすことができる一方で，設計内容を明示的に記述することで高い記述力を得ることもできる．

DBPowder-mdl は，プログラマに以下のようなメリットを与える．

- はじめに RM と OM の片方のみを設計記述し他方を導出に頼ることで EoD を享受し，のちに導出部分を詳細設計することで記述力の恩恵を受けることができる．
- RM を設計後，その資産を活かして OM を設計できる．その逆（OM → RM）も可能である．
- スキーマが複雑であると予想される箇所ははじめから RM，OM，mOR のすべてを設計記述するようにし，それ以外の箇所は RM と OM の片方のみを設計記述することができる．

本論文の構成は以下のとおりである．2 章では O/R マッピングを CoC，EoD および記述力の観点から議論し，アプローチを分類する．3 章で DBPowder-mdl を提案する．4 章で DBPowder-mdl の評価を行い，5 章で総括する．

2. CoC，EoD，記述力と O/R マッピングフレームワーク

2.1 設定より規約（CoC）と開発の容易さ（EoD）

アプリケーションフレームワーク（以下，フレームワーク）の目的の 1 つは，複雑な開発対象の開発を容易にすることである．それを実現するためのアプローチとして，設定より規約（Convention over Configuration: CoC）[9),22)] という考え方が現在注目を集めている．

CoC とは，フレームワークが規約（Convention）を定めプログラマがそれを守ることで，プログラマが行う設計記述（Configuration）の一部をフレームワークが導出するようにして，プログラマの負担を下げようという考え方である．この規約と導出による仕掛けが有効に作用すれば，プログラマは開発の容易さ（Ease of Development: EoD）を享受できる．

CoC の考え方を強力に推し進めたフレームワークの 1 つとして，Ruby on Rails（RoR）[11)] があげられる．RoR の特徴の 1 つとして強力な O/R マッピング機能がある．以下に規約の一部を示す．

(i) テーブル名はクラス名単語の複数形とし，主キー名は `id` とする[⋆1]．

(ii) 外部キーは，対象テーブル名の単数形に `_id` を付与したものとする[⋆2]．

プログラマがこのような規約を受け入れるならば，テーブル名，主キー名，外部キー名をフレームワークに改めて指示する必要がなくなり，設計記述量を大幅に削減できるため，RoR の CoC は EoD に寄与する．

一方で，CoC における規約を受け入れられない箇所が多い場合は，CoC に基づくフレームワークの導入メリットは小さくなる．例を以下に列挙する．

(1) 規約が難解だと，規約の習得そのものが困難となる．

(2) 規約がフレームワークの記述力を制約する場合がある．

(3) 既存システムが規約に適合しない場合がある．

(4) 規約違反の連鎖が発生する場合がある．これについては 4.4.3 項に例示する．

(5) 規約違反の箇所が多いと，プログラマが記述すべき内容が複雑になる場合がある．

RoR における (3) の例を示す．RoR の命名規約はテーブル，主キー，外部キーの名称決定に (i) や (ii) のような強い制約を課するため，RoR によらない既存システムは，RoR の規約に合致しない箇所が多く発生すると考えられる[⋆3]．このようなシステムに RoR を導入するためには，RoR 内でテーブル名などを明示的に指定して規約に違反させるか，あるいは規約に合致させるためにテーブル名などを変更する必要がある[⋆4]．(4) および (5) と既存システムのテーブル名変更は困難をともなう場合が多いことをふまえると，既存システムが保持するテーブルの名称が RoR の規約と整合性が悪い場合，EoD の観点からは無理に RoR を導入しない方がよいケースも多いと考えられる．

2.2 O/R マッピングフレームワークの分類

本論文では RDB を用いたオブジェクト指向開発の手法を，O/R マッピングの観点から 4 つに分類する（**表 1**）．

(a) リレーション中心アプローチ[11),17),18),23)]

(b) オブジェクト中心アプローチ[5),7),15),25)]

(c) 全定義アプローチ[6),13),24),26)]

⋆1 RoR の規約に従うと，person クラスに複数の address クラスを持たせたい場合，RM のテーブル名はそれぞれ `people`，`addresses` となり，主キー名はいずれも `id` である．

⋆2 同じく，`addresses` テーブルは `person_id` という外部キーを持つ．

⋆3 たとえば，`person_tbl`，`addr_tbl` というテーブル名は単語の複数形ではないので，RoR の規約に合致しない．

⋆4 たとえばテーブル名を `person_tbls` と `addr_tbls` に変更する．

表 1　O/R マッピングの分類：RM，OM，mOR
Table 1　Classification of O/R maping: RM, OM and mOR.

アプローチ	RM	OM	mOR	例	CoC
(a) リレーション中心	■	-	-	ADO.NET, Oracle APEX	多用される傾向
(a') Ruby on Rails	■	□	-	*(Ruby on Rails (RoR))*	多用
(b) オブジェクト中心	-	■	-	WebML, phpClick	多用される傾向
(c) 全定義	■	■	■	Hibernate, JPA, Toplink, M²ORM²+HIE	一部取り入れ
(d) O/R マッピング不使用	■	■	■	*(プログラマが設計及び実装)*	

RM	リレーショナルモデル	■	記述の必要あり
OM	オブジェクトモデル	□	一部記述の必要あり
mOR	RMとOMのマッピング	-	記述の必要なし (導出される)

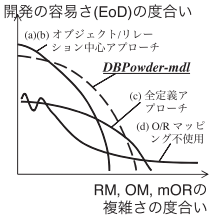


図 1　RM，OM，mOR の複雑さと EoD の関係
Fig. 1　Relationship between the degree of complexity and EoD: RM, OM and mOR.

(d) O/R マッピングフレームワーク不使用アプローチ

(a) は，ER モデル[8] などを用いてプログラマが RM（リレーショナルモデル）を設計し，RM に対応する OM（オブジェクトモデル）と mOR（RM と OM のマッピング）を O/R マッピングフレームワークで導出するアプローチである[11),17),18),23)]．フレームワークが導出する OM と mOR はプログラム言語で記述され，プログラマはこれらを直接利用できることから，RM と OM のインピーダンスミスマッチは軽減される．

(b) は，プログラマが OM を設計し，OM に対応する RM と mOR を O/R マッピングフレームワークで導出するアプローチである[5),7),15),25)]．プログラマが設計する OM はオブジェクト指向のプログラム言語でクラスとして直接表現できることと，フレームワークが RM と mOR を導出することから，RM と OM のインピーダンスミスマッチは軽減される．

(c) は，RM，OM，mOR のすべてをプログラマが自前で設計するアプローチである[6),13),24),26)]．(c) では，プログラマが記述した RM，OM，mOR の設計に基づいて，O/R マッピングフレームワークがこれらの実装を生成する．

(d) は，O/R マッピングフレームワークを使用しないアプローチである．RM，OM，mOR をすべて，プログラマが自前で設計を行い実装も行う必要がある．フレームワークによる導出や生成を利用しないアプローチである．

(a) と (b) は，CoC が多用される傾向が強い．RoR はその典型例である．(c) は，モデルの記述力が重要視されるため，CoC の活用は一部にとどまるケースが多い．

2.3　O/R マッピングフレームワークにおける EoD と記述力のトレードオフ

図 1 は，表 1 に示す 4 つのアプローチ (a)，(b)，(c)，(d) および DBPowder-mdl について，開発対象の RM，OM，mOR の複雑さに対する開発の容易さ（EoD）の度合いの関係を定性的に図示化したものである．RM，OM，mOR の複雑さは，この 3 要素のいずれかの複雑さが増すことで増加する．また，RM と OM の乖離度が増すと mOR は自ずと複雑になる．

開発対象の複雑さと EoD は，アプローチの手法によらず明らかにトレードオフの関係にある．なお，本論文で提案する DBPowder-mdl の目標を図 1 内の太点線で示した[⋆1]．以下に比較ポイントを示す．

(1) (a)，(b) と (c) の比較
RM，OM，および mOR について，類似性のある箇所は多いと考えられる．したがって多くの箇所で，(a) による OM と mOR の導出や (b) による RM と mOR の導出は効果を発揮する．このような箇所で (c) を用いると，重複した設計記述が大量に発生するため，生産性や保守性の観点からは好ましくない[12)]．RM と OM が乖離した場合は，(a) や (b) はこれをある程度記述できるが，乖離度が限界を超えると (c) のように RM，OM，mOR のすべてを記述した方がかえって開発が容易になると考えられる．

(2) (c) と (d) の比較
(d) は，RM，OM，mOR のすべてを設計だけでなく実装も行う必要がある．(c) は，RM，OM，mOR のすべてを設計すれば O/R マッピングフレームワークが実装を生成する．3 者のいずれかが複雑な場合は実装も複雑になるため，(c) を用いる効果は高い．ただし，記述対象が (c) の想定外の場合に O/R マッピングフレームワークを無理に適用しようとすると，かえって開発が困難になる場合がある．また (c) は RM，OM，mOR のすべてを明示的に記述する必要があるため，開発内容自体が非常に単純な場合は (c) よりも (d) のほうがかえって開発が容易になるケースもあると考えられる．

(a)，(b)，(c) を比較すると，複雑な開発対象のサポートを重視するほど EoD の度合いを下げざるをえず，EoD を重視するほど複雑な開発対象のサポートに限界が生じる．すなわち，EoD とフレームワークとしての記述力にはトレードオフが生じる．なお (a)，(b)，(c) の適材適所での併用も考えられるが，複数の異なる設計思想を持つフレームワークが生成したスキーマどうしを連携させるのは困難がともなう．特に (a) や (b) が導出する側のモデルに関する深い理解が必要となる．

⋆1 非常に単純なスキーマについて，DBPowder-mdl と (a)，(b) との EoD の差異は不明確である．また，非常に複雑なスキーマの記述力について (c) との比較評価を，本論文の範囲で十分に実施できたとはいえない．したがって図 1 のグラフでは (a)，(b) や (c) との交点が存在することとした．

3. 提案：DBPowder-mdl

3.1 概　　要

本研究では，EoD（開発の容易さ：2.1 節参照）と記述力を兼備した O/R マッピング言語，DBPowder-mdl を提案する．DBPowder-mdl は，プログラマが記述すべき必須項目を絞り込み簡潔な記法を実現するために，CoC（設定より規約：2.1 節参照）とインデントを言語設計の中核に据える．

DBPowder-mdl では，RM（リレーショナルモデル）と OM（オブジェクトモデル）の名称記述を CoC を用いて一部省略できる一方で，必要な場合は柔軟な名称記述や追記も可能である（3.3.1 項）．主キーや外部キーの設計記述を CoC を用いて省略できる一方で，規約違反を想定した言語設計により柔軟な設計記述も可能である（3.3.3 項）．OM では，関連（3.3.2 項）のほかに，委譲（3.3.2 項），多態性（3.5.3 項），継承（3.5.5 項）を実現できる．この際の RM と mOR（RM と OM のマッピング）は，CoC に基づく導出と明示的設計の両方が可能である．これらの要素を DBPowder-mdl という 1 つの言語に盛り込むことで，ラウンドトリップ型開発に基づく RM，OM，mOR の多彩な開発を可能とする（3.5.1 項）．DBPowder-mdl の特徴は，**図 2** ⋆1に示すような簡潔な記法を起点とした EoD を CoC に基づいて実現しつつ，ここにあげた多彩な記述力を兼備することにある．

記述力の高いフレームワークは必要な記述量が多くなり複雑かつ難解になりがちだが，DBPowder-mdl では，これを CoC に隠蔽することで簡潔な記法を実現した．**図 4**（3.4 節）は，`faculty` と `course` から構成されるスキーマを DBPowder-mdl で 3 種類記述しており，(a) は RM の記述，(b1) と (b2) は OM の記述である．(a) をプログラマが設計記述すれば，DBPowder-mdl の規約が OM と mOR を導出する．(b1) または (b2) をプログラマが設計記述すれば，DBPowder-mdl の規約が RM と mOR を導出する．

たとえば図 4 の (b1)（または図 2）では，プログラマはクラス `[Faculty]` と `[Course]` を記述し，それぞれのインデントの内側に属性を書き下す．そのうえでクラス `[Faculty]` と `[Course]` の関連をインデントで記述し，多重度 `1:n` を `[Course]` 内に宣言すれば，図 4 (b1) のクラス図に示した OM の記述は完成である．(b2) も同様である．なお (b1) または (b2) から規約により RM と mOR を導出すると (a) になる．このように，図 2 や図 4 に示した簡潔な記法のみを用いても，多彩な RM，OM，mOR の設計記述が可能である．なお，ここで例示した（クラス，属性，関連，多重度）は，3.2 節で述べる OM の主要 4 要素を構成する．

⋆1 本節での説明のため，図 4（3.4 節）の (b1) から DBPowder-mdl のみ抜粋したものが図 2 である．

```
[Faculty]
  fac_name text128
  [Course 1:n]
    co_name text128
    credits int
    requisite bool
```

図 2　学科–科目スキーマ（図 4）の (b1) 抜粋
Fig. 2　Extraction of (b1) from faculty–course schema (Fig. 4).

表 2　DBPowder-mdl 主要 4 要素 (E, A, L, C)
Table 2　DBPowder-mdl: principal 4 elements of (E, A, L, C).

DBPowder-mdl	対応する RM, OM の要素	
	RM	OM
E: エンティティ	E_R: リレーション(テーブル)	E_O: クラス
A: 属性	A_R: 属性(カラム)	A_O: 属性
L: リンク	L_R: リレーションシップ	L_O: 関連
C: カーディナリティ	C_R: カーディナリティ	C_O: 多重度

3.2 主要 4 要素：(E, A, L, C) と (E_R, A_R, L_R, C_R)，(E_O, A_O, L_O, C_O)

DBPowder-mdl では，エンティティ (E)，属性 (A)，リンク (L)，カーディナリティ (C) を主要 4 要素 (E, A, L, C) とする．そのうえで，(E, A, L, C) に対応する RM と OM の主要 4 要素を定め，それぞれ (E_R, A_R, L_R, C_R) および (E_O, A_O, L_O, C_O) とする（**表 2**）．

3.1 節で例示したように，プログラマが OM のクラス（E_O），属性（A_O），関連（L_O）および多重度（C_O）を記述すると，DBPowder-mdl は対応する RM と mOR を規約を用いて導出する．また，RM に関しても同様に，テーブル（E_R），カラム（A_R），リレーションシップ（L_R）およびカーディナリティ（C_R）を記述すると，DBPowder-mdl は RM の主キーと外部キーを補完し，対応する OM と mOR を規約を用いて導出する．

ここに述べた RM，OM，mOR やキー属性の導出メカニズムは，3.3 節で明らかにする．

3.3 DBPowder-mdl の記法

本節では DBPowder-mdl の記法を述べる．なお文法を付録 A.1 に記し，BNF 記法を図 15 に示した．

3.3.1 CoC によるエンティティ E と属性 A の名称記法：省略記法と無省略記法

(E_R, A_R, L_R, C_R) と (E_O, A_O, L_O, C_O) の混在記述や導出を実現するために，DBPowder-mdl はエンティティ E と属性 A の名称記述に関して，省略記法と無省略記法を備える．それぞれの記法を以下に示す．

- 省略記法：　`commonName`
- 無省略記法：　`objectName@relationName`

プログラマが省略記法で記述すると，DBPowder-mdl は規約に従い `commonName` から

RM と OM の名称を導出する[⋆1]．プログラマが無省略記法で記述すると，DBPowder-mdl の名称に関する規約や導出は用いられず，記述内容がそのまま RM と OM の名称になる．`relationName` が RM に，`objectName` が OM に対応する．省略記法は CoC を用いた EoD に寄与し，無省略記法はプログラマの詳細な設計開発に寄与する．

今までに述べた図 2 や図 4 (a)，(b1)，(b2) の例では，E と A はすべて省略記法である．**図 5**[⋆2]の上段（無省略記述）では，図 4 で記述した E と A の省略名称すべてが無省略記法に変換されている．たとえば図 4 (b1) の省略記法 `Faculty` は，規約により図 5 では無省略記法 `Faculty@faculty` に変換されている．すなわち，RM での名称は `faculty`，OM での名称は `Faculty` となる．

3.3.2　インデントを用いた (E, A, L, C) の記法

DBPowder-mdl では，エンティティEが持つ属性Aや，2 つのエンティティ(E_{outer}, E_{inner})のリンク L を，要素間のインデントを用いて記述する．ここで，E_{outer} をインデント外側のエンティティ，E_{inner} をインデント内側のエンティティとする．カーディナリティC は E_{inner} 内に記述する．たとえば図 4 の (b1) で E としての `Faculty` は，A として `fac_name` を持ち L の先が `Course` である．`Faculty` と `Course` の C は `1:n` である．

属性 A へのデータアクセスは，DBPowder-mdl が OM に生成する getter および setter メソッドを介して行う．たとえば属性 `fac_name` へのアクセスは，DBPowder-mdl が生成する `getFacName` メソッドや `setFacName` メソッドを介して行う．

プログラマは，E_R や E_O の名称を複数の E 内で重複して記述できる．重複した場合，各々が統合され 1 つのテーブル E_R やクラス E_O として扱われる．また，3.3.1 項の無省略記法により 1 つのテーブル E_R に複数のクラス $(E_{O_0}, E_{O_1}, ...)$ を対応付けできる．これにより，リンク L を任意のエンティティE 間に張ることができる[⋆3]．L_R と E_R についても同様である．ただし，この記法で 1 つのクラス E_O に複数のテーブル $(E_{R_0}, E_{R_1}, ...)$ を対応付けることは不可としている．対応付ける場合は本項の後に述べる `dele` 記法を用いる．

プログラマが `rel` または `bidir`[⋆4]を E_{inner} 内で指定すると，DBPowder-mdl はリレーションシップ L_R および関連 L_O について，(E_{outer}, E_{inner}) を双方向に関連づけるものと解釈する．`rel` や `bidir` を指定しない場合は，DBPowder-mdl は L_R を双方向リレーションシップ，L_O を片方向関連と解釈する．L_O が片方向関連の場合はインデントの上下関係が関連の方向になる一方で，L_R が片方向になることはできず必ず双方向になる[⋆5]．プログラマは，RM 設計時に `rel` と記述しておき，後で OM 的な要素を検討して `rel` 記述を除去するといった使い方ができる．なお，L_R や双方向を指定した L_O については，プログラマが E のインデント上下関係と C の左右を交換しても L_R や L_O は等価に保たれる[⋆6]．

プログラマが `dele` を E_{inner} 内で指定すると，E_{outer} のクラス E_O は E_{inner} の getter および setter の委譲メソッドを持つ．これによりプログラマは，関連 L_O をたどらずに E_{outer} から E_{inner} の属性 A_{inner} に直接アクセスできる．`dele` 指定時に C が `1:n` または `n:n` の場合は，関連 L_O はリストとなるため，委譲メソッドはリストの最初の要素へのアクセスのみを提供する．

プログラマが `inherit` を E_{inner} 内で指定すると，E_{inner} は E_{outer} を継承する．詳しくは 3.5.5 項で述べる．

3.3.3　CoC による (E, A, L, C) の補完規約

DBPowder-mdl では，プログラマは主キーや OID といった識別キーや，外部キーやリファレンスといった参照キーを明示的に記述する必要はない．これらのキーは，本項で述べる補完規約を用いて DBPowder-mdl が必要なものを挿入する．なお，必要な場合はプログラマが明示的に記述可能である．

プログラマがエンティティE を記述すると，DBPowder-mdl は規約により RM に主キー[⋆7]を補完し，OM に補完内容を反映させる．プログラマがリンク L やカーディナリティC を記述すると，DBPowder-mdl は規約により RM に外部キーや結合テーブルを補完し，OM に補完内容を反映させる．プログラマは主キー，外部キーや結合テーブルを記述する必要はないが，下記に列挙する記法を用いて規約に頼らず明示的に記述することもできる．

- `pkeys="..."`　主キー
- `joinOn="..."`　外部キー，結合テーブル

図 3 に，L や C に関する RM への補完規約を図示する．DBPowder-mdl は，必要に応じて外部キーや結合テーブルを補完する．`1:1` や `1:n` の場合は，DBPowder-mdl は外部キーをインデント内側のエンティティ（E_{inner}）に挿入する．`n:1` の場合は，DBPowder-mdl は外部

⋆1 命名規約を付録 A.1.1 に記載した．なお命名規約は，3.5.2 項に述べるカスタマイズが可能である．
⋆2 図 5 の詳細は 3.4 節で述べる．
⋆3 付録 A.2 に証明を示した．
⋆4 `rel` はリレーションシップ（RELationship），`bidir` は双方向（BI-DIRection）関連を意味する．`rel` と `bidir` は DBPowder-mdl の解釈としては同義語であり，動作としては区別はない．
⋆5 RM の一般的なリレーションシップ定義に拠る．
⋆6 C が $m:n$ であった場合に E のインデント上下関係を交換すると，C は $n:m$ になる．
⋆7 自動採番の整数値カラムである．

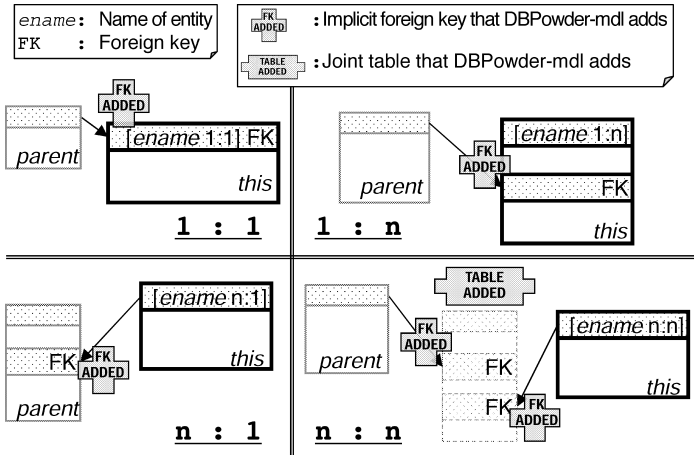


図 3　DBPowder-mdl による外部キーと結合テーブルの挿入
Fig. 3　Foreign keys and joint tables inserted by DBPowder-mdl on the relational model.

キーをインデント外側のエンティティ（E_{outer}）に挿入する．n:n の場合は，DBPowder-mdl は結合テーブルを挿入する．OM については，RM の補完内容が反映されたうえで，C に応じて関連 L_O がリファレンスまたはリストと解釈される[⋆1]．なお，RM と OM ともに，C として指定できるのは図 3 に示した 4 種類である．

3.4　記述例 1：学科–科目スキーマの DBPowder-mdl による記述

DBPowder-mdl の記述例とそれがもたらす効果を見るために，本節では 3.1 節に引き続いて学科–科目スキーマ（図 4，5）の例を取り上げる．学科–科目スキーマは学科（faculty）で受講可能な科目（course）を管理するスキーマである．学科は，名前（fac_name）を属性として持つ．科目は，名前（co_name）のほかに単位数（credits）と必須科目かどうか（requisite）という属性を持つ．ある学科で受講可能な科目は複数あるため，faculty と course のカーディナリティは 1 : n である．

図 4，5 は，学科–科目スキーマを以下の 3 パターンで記述したものである．(a) RM，(b1) 学科を起点とした OM，(b2) 科目を起点とした OM．プログラマが図 4 の (a)，(b1)，(b2) のいずれかを CoC を活用して設計記述すれば，DBPowder-mdl は記述していない方のモデル（RM または OM）を 3.3 節に示した規約により導出し（図 5 上段），図 5 下段に示す実行コードを生成する．なお図 5 上段で，規約が補完した内容を下線太字で示した．

⋆1 参照の実現方法はプログラム言語によって異なるが，通常，参照対象クラスのリファレンスを保持することで実現する．参照先が多の多重度を持つ場合は，リファレンスの配列，セット，リストなどが用いられる．なお本論文では，C++言語などで使われるポインタなども広義の意味でリファレンスと見なす．

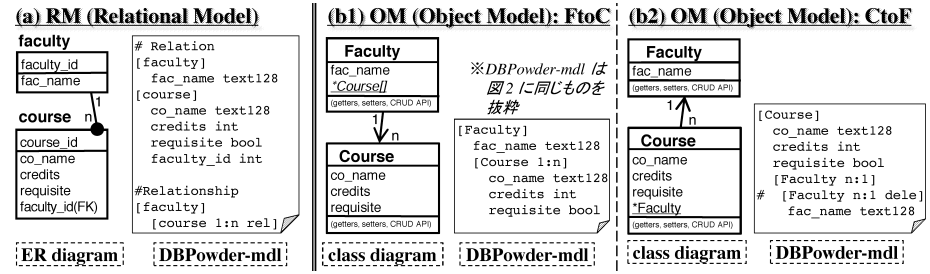


図 4　DBPowder-mdl 記述例 1：学科–科目スキーマと 3 種類の記述方法（RM 1 種類，OM 2 種類）
Fig. 4　Example of DBPowder-mdl (1): faculty–course schema and three kinds of description.

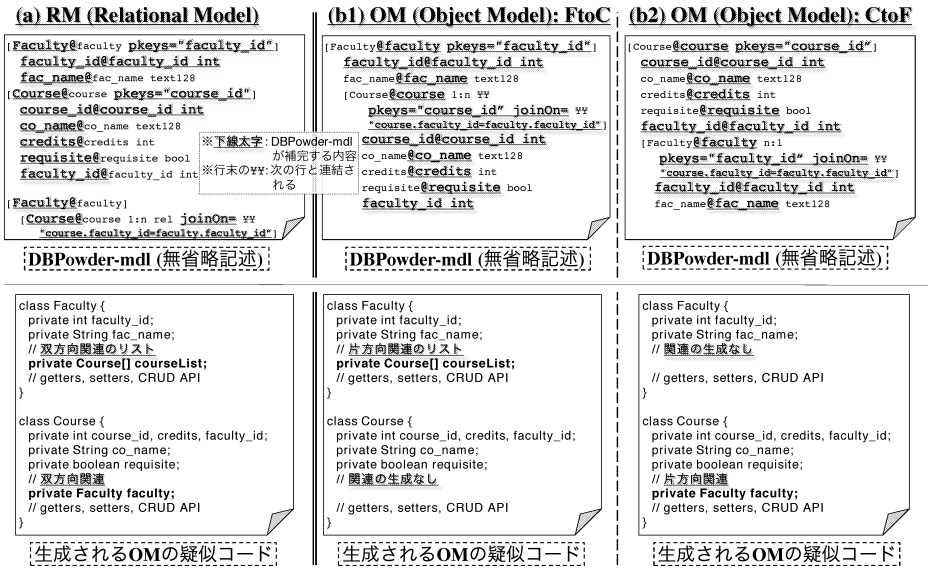


図 5　図 4 の 3 種類それぞれの無省略記述と，DBPowder-mdl が生成する疑似コード
Fig. 5　Full descriptions for the three kinds in Fig. 4, and pseudo codes generated by DBPowder-mdl.

3.4.1　(a)：RM（リレーショナルモデル）の記述

本節では図 4，5 の (a) について述べる．(a) は，プログラマが DBPowder-mdl で設計記述した RM（図 4）から，実行コード（図 5 下段）を得る過程である．OM は DBPowder-mdl がすべて導出し生成するので，プログラマが意識する必要はない．図 5 下段の疑似コードに示すように，DBPowder-mdl は (a) で，属性，getter，setter，CRUD API のほかに `Faculty` と `Course` 間の双方向関連を導出し生成した．

図 4 でプログラマは，2 つのテーブル `faculty` と `course` をエンティティ表現 E で記述し，リンク表現 L でこれらのリレーションシップ L_R を記述した．2 テーブルの主キーと L_R で用いる外部キー制約をプログラマが設計記述する必要はない．図 4 と図 5 上段を見比べると，DBPowder-mdl が補完によって，以下の内容を挿入したことが分かる．

- 主キー `faculty_id, course_id`
- 外部キー制約 `faculty_id`（`joinOn` 記述による）
- E_R および A_R に対応する OM の名称

なお，DBPowder-mdl で RM を設計記述する際は，リンク表現 L_R で `rel` と記述しておく．これにより，DBPowder-mdl は関連 L_O を双方向関連として導出する．`rel` を記述しない場合，L_R はリレーションシップなので双方向のままだが，L_O の関連が片方向になるため，RM と OM の内容が乖離する．したがって `rel` と記述しない場合は，OM への配慮が必要である．

3.4.2　(b1)，(b2)：OM（オブジェクトモデル）の記述

本項では図 4，5 の (b1) と (b2) について述べる．(b1) や (b2) は，プログラマが DBPowder-mdl で設計記述した OM（図 4）から，実行コード（図 5 下段）を得る過程である．RM は DBPowder-mdl がすべて導出し生成するので，プログラマが意識する必要はない．

(b1) でプログラマは，`Faculty` クラスのインデント内側に `Course` クラスを置くことで，`Faculty` から `Course` への片方向関連 L_O を記述している．この関連 L_O は `[Course 1:n]` と記述される．`Faculty` から `Course` への多重度 C_O が多であるので，`Faculty` は `Course` のリストを持つ．図 4 と図 5 上段を見比べると，DBPowder-mdl が補完によって，以下の内容を挿入していることが分かる．

- E_O および A_O に対応する RM の名称
- RM の対応する主キー `faculty_id, course_id`
- RM の対応する外部キー `course.faculty_id`
- RM の対応する外部キー制約 `faculty_id`（`joinOn` 記述による）
- DBPowder-mdl が挿入した RM の各要素を OM に反映

図 5 下段の疑似コードに示すように，DBPowder-mdl は (b1) で，属性，getter，setter，CRUD API のほかに `Faculty` から `Course` への片方向関連を導出し生成した．

(b2) では，片方向関連 L_O の方向が (b1) とは逆方向であることに注意が必要である．これにより，多重度 C_O も対応が逆転して `n:1` となる．関連 L_O は，`Course` が `Faculty` へのリファレンスを持つことで実現される．なお，インデント内側のクラス表現 E_O で `dele` と宣言することで，インデント外側のクラスは getter および setter の委譲メソッドを持つ．たとえば `[Faculty n:1]` についてコメントアウトしたほうの宣言を有効にすると，`Course` クラスは `getFacName` および `setFacName` メソッドを持つようになる．図 5 下段の疑似コードに示すように，DBPowder-mdl は (b2) で，属性，getter，setter，CRUD API のほかに `Course` から `Faculty` への片方向関連を導出し生成した．

3.4.3　(b1)，(b2) と (a) の関係

図 4，5 の (b1) や (b2) では，プログラマがそれぞれ別のセマンティクスに着目して OM のスキーマを設計し，それに基づき DBPowder-mdl スクリプトを記述している．着目するセマンティクスが異なるため，DBPowder-mdl が生成する OM のコードの動作は異なる．一方で DBPowder-mdl が (b1) や (b2) から導出する RM は，いずれも (a) のものと等しくなる．これは，DBPowder-mdl が補完により属性などを挿入した結果である．

3.5　そ　の　他

3.5.1　ラウンドトリップ型開発：RM と OM の混在

DBPowder-mdl は，(E_R, A_R, L_R, C_R) と (E_O, A_O, L_O, C_O) を混在させつつ，RM と OM の双方について CoC による導出と明示的な設計記述の同時利用が可能である．この特徴により，プログラマは DBPowder-mdl を用いることで，RM と OM のラウンドトリップ型開発やそれにともなう様々な開発のスタイルが可能となる．その例を以下に列挙する．

- はじめは RM と OM の片方のみを設計記述し他方を導出に頼ることで EoD を享受し，のちに設計記述しなかった部分を詳細設計することで記述力を享受できる．
- RM の得意なプログラマと OM の得意なプログラマが 1 つのシステムを開発する場合，お互いに得意なモデルで開発を開始し，適宜マージしながら進めることができる．
- RM を設計開発後，その資産を活かして OM を設計開発できる．その逆（OM → RM）も可能である．
- スキーマが複雑であると予想される箇所ははじめから RM，OM，mOR のすべてを設計記述し，それ以外の箇所は RM と OM の片方のみ設計記述して開発を進めることが

できる．

3.5.2　規約のカスタマイズ

DBPowder-mdl では，CoC による命名規約を実現する名称変換モジュールについて，カスタマイズできる．たとえば以下に列挙する名称変換モジュールをカスタマイズできる．

- エンティティ名を変換し，テーブル名を得る．
- エンティティ名を変換し，クラス名を得る．
- テーブル名を変換し，主キー名を得る．
- リンク実現の際，リレーションシップの種類と関係するテーブルの主キー名を変換し，外部キーの名称を得る．

これにより，開発プロジェクトごとに独自の CoC を導入できる．たとえば RoR（Ruby on Rails）風の CoC を導入できる．ただし 2.1 節の議論により，命名規約が複雑になったり不自然になったりするのは望ましくないため，使用には注意が必要である．

付録 A.3 に規約のカスタマイズ例を示した．

3.5.3　仮想エンティティ

DBPowder-mdl は，OM の多態性と継承を実現するために，仮想エンティティに関する記法を持つ．DBPowder-mdl の仮想エンティティは，他のエンティティから利用され自身では RM を処理しないエンティティである．利用方法を本項の後に (1)，(2)，(3) として列挙する．なお本項では，仮想エンティティと対比して通常のエンティティ E を実エンティティと呼ぶ．

仮想エンティティを記述するためには，エンティティ E の記述で括弧 `[]` の代わりに `()` を用いる．例として，`Bill` という仮想エンティティが `owner`，`created` という属性を持つ場合をあげる．

```
# DBPowder-mdl
(Bill)
  owner text64
  created datetime
```

以下に 3 種類の利用方法を列挙する．

(1) 実エンティティの実装インタフェースとして宣言できる．宣言した実エンティティは仮想エンティティに記述された属性を持つようになる．たとえば

`[BillingDetails!Bill]`

とプログラマが記述した場合，実エンティティ`BillingDetails` は，仮想エンティティ `Bill` に記述された属性 `owner` と `created` を持つ．DBPowder-mdl の実装言語がサポートする場合は，`Bill` という型を持つオブジェクトとして扱うことができる．たとえば Java の場合，DBPowder-mdl が生成する BillingDetails クラスは Bill interface を保持する．

```
// Java
Bill bill = new BillingDetails();
```

(2) インデント内側に継承エンティティ（3.5.5 項）を持つことができる．この場合，継承の一形態として Concrete Class Relation Inheritance（CCR）を実現できる．3.5.5 項で述べる．

(3) 他の実エンティティ E のインデント内側に入り，実エンティティに類似した形で利用できる．C の種類は `1:1` と `n:1` のみ可能である．たとえば `(Bill)` が別エンティティ E のインデント内側に入った場合，E は E_R の属性 A_R として `owner`，`created` を持つようになる．E_O としては，`(Bill)` を関連 L_O として保持するのみであり，`(Bill)` を実エンティティとした場合と変わらない．

3.5.4　CommonDef と importDB

DBPowder-mdl は，プログラマが複数のエンティティ E について同じ仮想エンティティやテーブルを割り当てたい場合のために，CommonDef 記法を備える．付録 A.1.5 に文法を示す．3.5.5 項の（SR）に記述例を示す．

プログラマがエンティティ E の記述で `importDB` と宣言すると，E の対象テーブル E_R から属性一覧が読み込まれ，エンティティ記述に統合される．この場合，E_R が存在しなければエラーとなる．

3.5.5　OM 継承と RM での取扱い

RM には OM の継承に対応する概念がないため，OM の継承を RM 上で扱う方法が複数存在する[1),4),6),10)]．文献 1)，4)，6)，10) では，継承を RM 上で実現するアプローチは 3 種類あるとしており，(1) Single Relation Inheritance（SR），(2) Class Relation Inheritance（CR），(3) Concrete Class Relation Inheritance（CCR）である．

DBPowder-mdl では，エンティティ E の記述で `inherit` と宣言することで，(SR)，(CR)，(CCR) の継承を実現する．**図 6** に，継承と RM での取扱いを DBPowder-mdl で記述した例を示す．`inherit` を宣言すると，キー属性の扱いはカーディナリティ C で `1:1` 宣言したものに準じる（図 3）．すなわち，(SR)，(CR)，(CCR) で用いられるテーブルの個数は

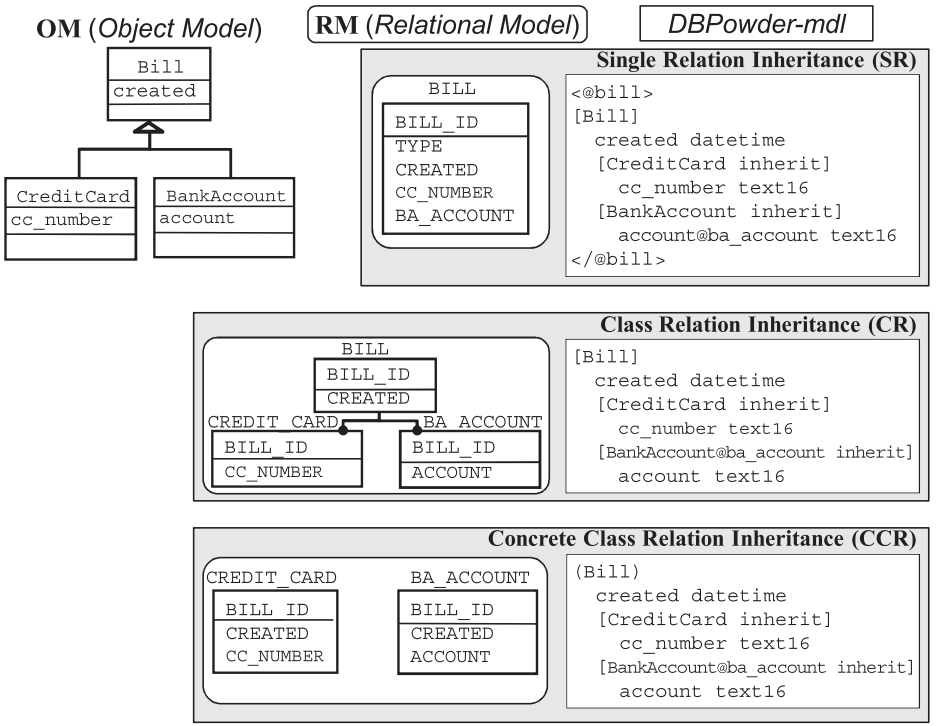


図 6　DBPowder-mdl 記述例 2：OM 継承と RM での取扱い
Fig. 6　Example of DBPowder-mdl(2): Inheritance.

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>
```

```
<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>
```

```
<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)

```
# RM 作成
ruby script/generate scaffold bill ¥
       type:string ¥
       created:datetime ¥
       cc_number:string ¥
       ba_account:string


# OM 作成
class CreditCard < Bill
end
class BankAccount < Bill
end

# 参考:Billクラスは，RM作成時に自動生成済
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は，本図の左側で示した*OM*や*RM*と一致しない．一致させるためには以下の追加処理が必要．

```
# RM 側の処置
# SQL の ALTER TABLE 文などで，
# 以下を実施
#   テーブル名: BILLS -> BILL
#   カラム名1  : ID    -> BILL_ID

# OM 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 7　OM 継承と RM での取扱い（Hibernate，RoR（Ruby on Rails））
Fig. 7　Inheritance (Hibernate, RoR (Ruby on Rails)).

異なるが，主キーはすべて `bill_id` である．また，（SR）と（CR）の DBPowder-mdl の記述は，OM の視点からは等しいものである．

図 6 の（SR）は CommonDef 記述（3.5.4 項）を用いて実現する．タグ内の 3 エンティティ `Bill`，`CreditCard`，`BankAccount` について，対象テーブルがすべて `bill` であると宣言しているため，3 エンティティの記述内容がテーブル E_R についてはすべて `bill` テーブルに集結する．

図 6 の（CR）は DBPowder-mdl における継承取扱いのデフォルトである．E_R と E_O が 1 対 1 にマッピングされる．

図 6 の（CCR）は `(Bill)` を仮想エンティティ（3.5.3 項）として扱うことで実現する．仮想エンティティは自身では RM を処理しないため，`(Bill)` エンティティの RM は，インデント内側の継承エンティティ（E_R は `credit_card` と `ba_account`）で処理される．すなわち，`(Bill)` で宣言された `created` 属性は，RM のカラム A_R としては `credit_card` と `ba_account` に移動する．

図 7 は，同様の内容を Hibernate[13] と RoR で記述した例である．4.2 節で比較評価を実施する．

3.6　DBPowder-mdl が生成するコード

3.6.1　DBPowder-mdl のコードジェネレータと，生成されるコンポーネント

DBPowder-mdl から RM と OM の実行コードを生成する様子を，**図 8** に示す．コード

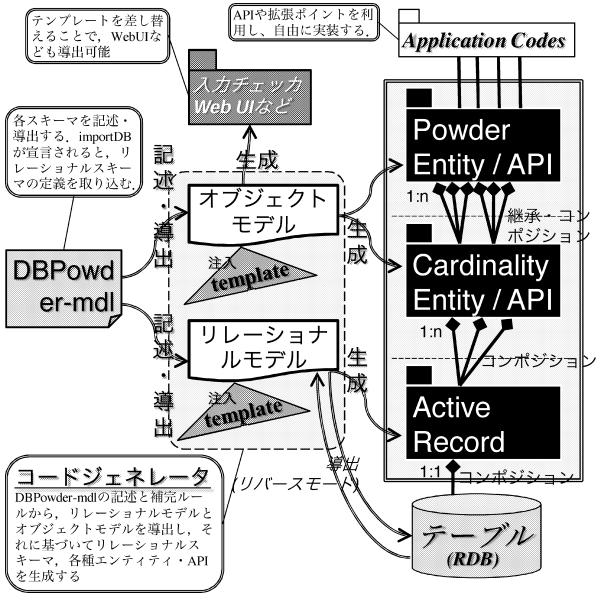


図 8　DBPowder-mdl から生成されるコンポーネント
Fig. 8　Components generated from DBPowder-mdl.

ジェネレータは以下の 2 つの役割を果たす．

- DBPowder-mdl スクリプトから RM，OM，mOR を導出する．
- 導出された各モデルから RM と OM の実行コードを生成する．内蔵するテンプレートを変更することで，WebUI などの別の実行コードを同時に生成することもできる．

コードジェネレータが生成する RM と OM の実行コードは，以下に列挙する要素から構成される．

- 各々の RDB テーブルに対応した Active Record
- Powder Entity，Cardinality Entity，RDB テーブル
- Powder Entity，Cardinality Entity，Active Record のそれぞれについて，対応するテーブル群への CRUD（Create，Retrieval，Update，Delete）機能を実現する API

Active Record [10] は，RDB のテーブル E_R をそのまま OM にマッピングしたものである．RoR などでも採用されている．テーブル名はクラス名に，カラム名はメンバ変数名に，主キーはオブジェクト識別子にマッピングされる．Active Record のクラスは RDB におけるテーブル E_R と 1 対 1 に対応する．Active Record の各クラスは，対応するテーブル E_R に対して CRUD 機能を実現する API を有する．

Cardinality Entity は，1 つ以上の Active Record と他 Cardinality Entity への関連 L_O から構成され，クラス E_O を構成する．Active Record 同様，それぞれの Cardinality Entity は CRUD 機能を実現する API を有する．この API は，関係する複数の Active Record 群や Cardinality Entity 群と連携して処理を実現する．

Powder Entity は Cardinality Entity のラッパクラスであり，プログラマが直接的に利用する．Powder Entity はプログラマに，Cardinality Entity やその API の機能を拡張するための拡張ポイントを提供する．コードジェネレータは Powder Entity を生成するが，変更は行わない．DBPowder-mdl の記述内容に変更があった場合，変更の波及範囲を Cardinality Entity までにとどめられるよう，生成される実行コードは設計されている．これにより，プログラマが Powder Entity に変更を加えた後に DBPowder-mdl を変更しても，プログラマの変更内容が消去されることはない．

Powder Entity は，型が同一の Cardinality Entity または Active Record を持つ他の Powder Entity と，データを通信する機能を持つ．これにより，異なる利用場面を持つエンティティ間でのデータ交換が可能となる．

DBPowder-mdl が直接的に働きかける RDB のスキーマカタログは，テーブル，カラム，主キー制約，外部キー制約の 4 つである．このうちテーブル，カラム，主キー制約は，前述のとおり Active Record に 1 対 1 に対応する．外部キー制約は Cardinality Entity における関連 L_O に対応しており，RM に実際に外部キー制約をかけるか否かを選択できる．外部キー制約をかけない場合は，RM には制約をともなわないリレーションシップ L_R が構築される．なお，コードジェネレータがテーブル E_R を作成する際に既存のテーブル定義の変更がともなう場合には，その旨を警告する．

3.6.2　DBPowder-mdl における CRUD 機能の実現

DBPowder-mdl が生成する Active Record，Cardinality Entity，Powder Entity は CRUD API を有し（図 8），find，insert，update，delete メソッドを持つ．これらはそれぞれ，対応するテーブル群への SQL SELECT，INSERT，UPDATE，DELETE 機能に相当する．find，update，delete では，エンティティが持つ属性ごとに等号，不等号，`LIKE`，`IN`，`BETWEEN` といった条件句をセットすることで条件の絞り込みを行う，QBE（query by example）機能を持つ．また，SQL の `WHERE`，`GROUP BY`，`HAVING`，`ORDER BY` 句に相当す

る箇所に SQL を直接埋め込むこともできる．

処理の対象となる Active Record の一覧を指定することで，必要のないアクセスを抑制できる．インデント内側のエンティティすべてを指定する場合と，自 Active Record のみ指定する場合は，一覧を指定する必要がない．また，内側にあるエンティティであるノードを指定から外すと木の刈り込みが行われ，そのノード配下のエンティティはアクセス対象から外される．

4. 評　　価

4.1 DBPowder-mdl プロトタイプシステムと，実運用への適用

本研究では，3 章の提案を実現するプロトタイプシステムを構築した．この DBPowder-mdl プロトタイプシステムは，MySQL 5.0.22 [21]，Java SE 1.6.0_03 [27]，Tomcat 6.0.14 [3]，Struts 1.3.5 [2] を用いて構築した．

このプロトタイプシステムは，パーザとコードジェネレータから構成される．パーザは DBPowder-mdl で記述したスクリプトを解釈し，コードジェネレータは解釈結果に基づき RM（リレーショナルモデル），OM（オブジェクトモデル），mOR（RM と OM のマッピング）を実現するコードを生成する．生成される RM は SQL の `CREATE TABLE` 文で，OM と mOR は Java コードで，それぞれ記述される．RDB へのアクセスには JDBC を利用する．

コードジェネレータは，OM 内に拡張ポイントを生成する．拡張ポイントを起点として，プログラマは生成されたコードに機能を追加できる．

このプロトタイプシステムは O/R マッピングに関するコードに加えて，CRUD 機能を持つウェブページを同時に生成できる[28]．ウェブページ生成機能は図 8 におけるテンプレートを追加実装することで実現した．生成されたページの一例を付録 A.5 の図 16 に示す．プログラマは生成されたウェブページを起点として，必要な要件を満たすウェブページへと発展させることができる．

このプロトタイプシステムを用いて，実運用に供するアプリケーション群（KEKapp）を開発した[19),20]．付録 A.5 の図 17 に設計した ER 図の一部を示し，図 18 に開発したアプリケーションの一部を示す．KEKapp は高エネルギー加速器研究機構（KEK）で実稼働しており，機構内で動作している DMZ クラスタ（DMZ）[⋆1] の各ホスト環境を管理するのに用いている．

KEK では，DMZ 内のすべてのホスト管理者は，年に 1 度セキュリティレポートを提出する義務を持つ．このレポート作成のために，KEK が保持するセキュリティ診断装置を使用することができる．この診断装置は，装置に特化する形で各ホストやホスト管理者の環境を管理する．この管理情報はバッチジョブや GUI を通じた閲覧や操作が可能だが，管理情報をカスタマイズする機能がないため，我々が所望する形でのホスト環境管理にそのまま利用することはできない．

そこで我々は，我々が所望する形でホスト環境管理を実現するために，DBPowder-mdl プロトタイプシステムを用いて KEKapp を開発した．KEKapp を構成するアプリケーション群はいずれも我々の管理業務軽減に大きく寄与しているほか，100 名強のユーザに DMZ User's Portal というセキュリティ統合管理サイトを提供，運用している．

4.2 CaveatEmptor による記述力の評価

本節以降，O/R マッピングの実用的フレームワークとして評価の高い RoR（Ruby on Rails）および Hibernate と DBPowder-mdl の比較を主としながら，DBPowder-mdl の評価を行う．

図 9 と**図 10** に，CaveatEmptor [4] の ER 図とクラス図を示す．CaveatEmptor は，Java Persistence with Hibernate [4),⋆2] 全体を通して，Hibernate の豊富な表現力を解説するための例として用いられるネットオークションシステムである．

クラス図（図 9）の関連を示す矢印それぞれに番号を振り，矢印の根に [ ] 囲みで示した．この番号それぞれについて，ER 図（図 10）の対応するリレーションシップに同じ番号を振った．CaveatEmptor は，RM と OM の対応関係が複雑であるといえる．

この CaveatEmptor について Hibernate と DBPowder-mdl で，図 9 と図 10 を満たすスクリプトを記述した．コメントや空白行を除いて，Hibernate では 292 行，DBPowder-mdl では 160 行であった．

図 9 の右下部分（`BillingDetails`，`CreditCard`，`BankAccount`）についてそれぞれ属性 1 つずつを取り出した OM を作成し，DBPowder-mdl，Hibernate，および RoR で記述した例を，図 6 と図 7 に示した．3.5.5 項で述べたとおり，継承を RM 上に実現するための方法として，典型的には SR（Single Relation Inheritance），CR（Class Relation Inheritance），CCR（Concrete Class Relation Inheritance）の 3 種があるとされる[1),4),6),10)]．

⋆1 インターネットからのアクセス要求を受け付けるネットワークセグメントである．外部からのセキュリティ脅威につねにさらされているため，セキュリティの維持管理が重要である．

⋆2 Hibernate の解説書として名高い．

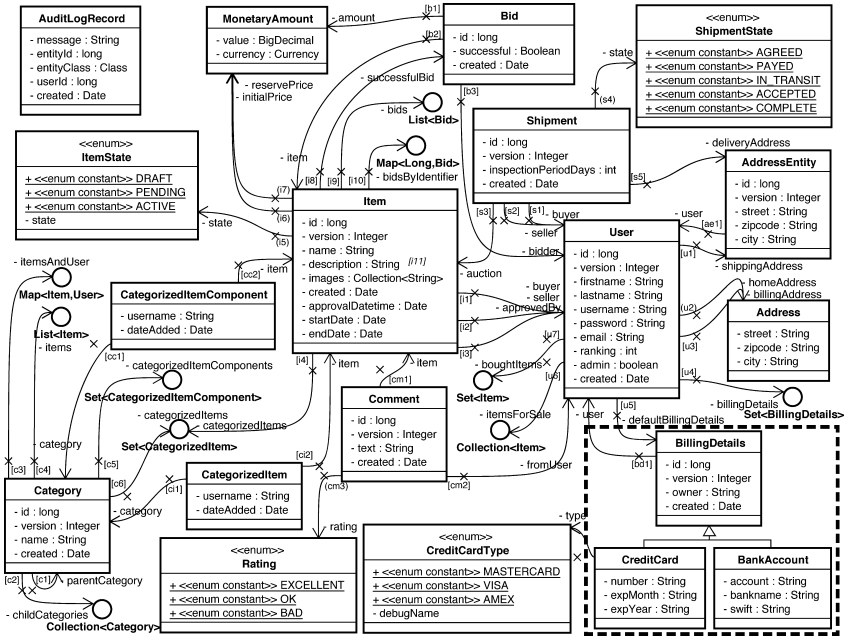


図 9　記述例：CaveatEmptor[4]（クラス図）
Fig. 9　CaveatEmptor (class diagram).

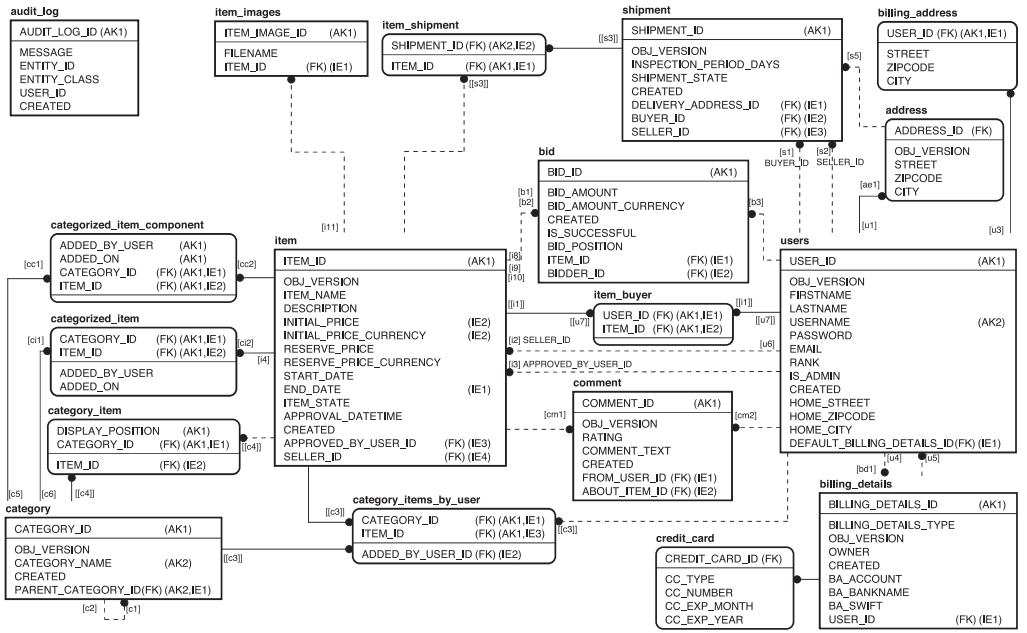


図 10　記述例：CaveatEmptor[4]（ER 図）
Fig. 10　CaveatEmptor (ER diagram).

図 6（DBPowder-mdl）については 3.5.5 項で説明したので，ここでは図 7（Hibernate, RoR）について述べる．

Hibernate を用いた開発では，プログラマは原則として RM と OM の要素をすべて記述する必要があるため，記述力は高いが記述量も多くなる．SR は `<subclass>` タグを用いて，CR は `<joined-subclass>` タグを用いて，CCR は `<union-subclass>` タグを用いて実現する．

RoR を用いた開発では，プログラマは scaffold コマンドを実行して RM のテーブルを作成した後に，同時に生成される `ActiveRecord` クラスを継承したクラスにリレーションシップなどを記述することで，RM と OM を実装する．RoR は SR をサポートするが，CR と CCR をサポートしない．SR の実現に関しては，scaffold で生成された `Bill` クラスを継承した `CreditCard`，`BankAccount` クラスを作成すればよいため，簡潔に記述できる．ただし，RoR の CoC に従ったテーブルとクラスが生成されるため，図 6 に指定した RM および OM とは，名称が異なっている．名称をそろえるためには，RM のテーブル名などを変更したうえで，Ruby のクラスにその変更を記述する必要がある（図 7 下部）[⋆1]．このような変更が増えてくると，RM や OM の記述が散逸し，CoC のメリットを享受できなくなる．

4.3　記述量による評価

表 3 に，3 種類のアプリケーションで用いられるスキーマを RoR，Hibernate，DBPowder-mdl で記述する際に必要となるコード量を，行数と項目数で示した．3 フレームワークとも，テーブル（E_R）やクラス（E_O）が持つ属性（A_R，A_O）について複数個を 1 行にまとめて記述できるが，ここでは 1 属性 1 行として数えた．対象としたアプリケーションは，KEKapp,

⋆1 このような名称変更が必要になるケースは様々に考えられるが，たとえば 2.1 節の (3) をあげることができる．

表 3　記述量の比較
Table 3　Quantitative comparison.

行数比較

	Ruby on Rails (RoR)	Hibernate	DBPowder-mdl
KEKapp	433 (367)	1,380 (979)	357 (333)
Redmine	(298)	(828)	(274)
magento	(2,103)	(4,471)	(1,877)

(単位:行)

項目数比較

	Ruby on Rails (RoR)	Hibernate	DBPowder-mdl
KEKapp	774 (682)	2,009 (1,490)	833 (739)
Redmine	(552)	(1,289)	(579)
magento	(3,977)	(7,325)	(3,922)

(単位:個)

名称	説明
KEKapp	KEKのDMZ User's Portal (4.1節)
Redmine	ウェブベースのプロジェクト管理ツール．作業チケット管理・ワークフロー管理・レポジトリ (CVS, SVN, ...)連携などの機能を有する
magento	ウェブベースのショッピングサイト構築アプリケーション

Redmine [14]，magento [16] である．対象となるテーブル（E_R）の個数は KEKapp 52 個，Redmine 44 個，magento 229 個である．

RM の複数テーブルを利用するための OM のエンティティ表現は，実行時のセマンティクスに応じて複数種類が考えられる．KEKapp ではこれによる複数種類エンティティが含まれている．上段に示した数値は複数のセマンティクスを含んだ場合であり，下段に示した括弧付きの数値はセマンティクスを単一のみに絞った場合である．Redmine，magento では，実行時のセマンティクスは単一のみで評価したため，数値はすべて括弧付きとなっている．

RoR では，CoC により RM のテーブル命名などに規約を課すことで，設計実装の簡略化を図っている．Redmine は RoR を用いたアプリケーションであるために，RM は規約に従っている．一方で KEKapp と magento は規約に従っていない．KEKapp の RoR での評価については，規約に極力従ったスキーマに変換したものを対象とした．magento については規約との乖離が著しかったため，スキーマ変換を行わず，`set_table_name` や `set_primary_key` などによりテーブル名や主キー名などを明示的に指定することとした[⋆1]．

RoR と DBPowder-mdl の比較では，行数と項目数の両方で，数値差は比較的小さいものとなった．このことから，ほぼ同一程度の簡潔さでスキーマを表現することに成功したといえる．DBPowder-mdl のキー挿入機能（図 3）などがこの簡潔さに寄与したと考えられる．Hibernate と DBPowder-mdl の比較では，行数で 2.4 倍〜3.9 倍，項目数で 1.9 倍〜2.4 倍の違いが出た．特に KEKapp の数値に表れるように，実行時のセマンティクスを複数持った場合に大きな差が開いた．これについては 4.4.3 項で議論する．

4.4　記述例 3：学科–生徒–履修–科目スキーマ（図 11）を用いた記述力評価

本節では，学科–生徒–履修–科目スキーマ（図 11）を用いて DBPowder-mdl の記述力を評価する．学科–生徒–履修–科目スキーマは，学科–科目スキーマ（図 4，図 5）を拡張したものであり，以下に列挙する 3 テーブルおよびそれに対応するクラスを追加した．

- `student`：学科に所属する生徒
- `register`：生徒が受講する履修
- `cert_course`：複数学科に共通する科目を登録可能にするための結合テーブル

図 11 の a) に RM，b) に OM，c) に実行時のセマンティクスに着目したクラス構成を示す．c') は c) から `faculty` と `cert_course` のクラスと属性を省いたものであり，4.4.3 項で使用する．d) は c) をそのまま RM にマッピングしたものである．

図 11 の a) をそのまま OM にマッピングすると b) になる．関連 L_O は双方向であるため，5 つあるエンティティの様々な関係を扱える一方で，属性 A_O へのアクセスのために関

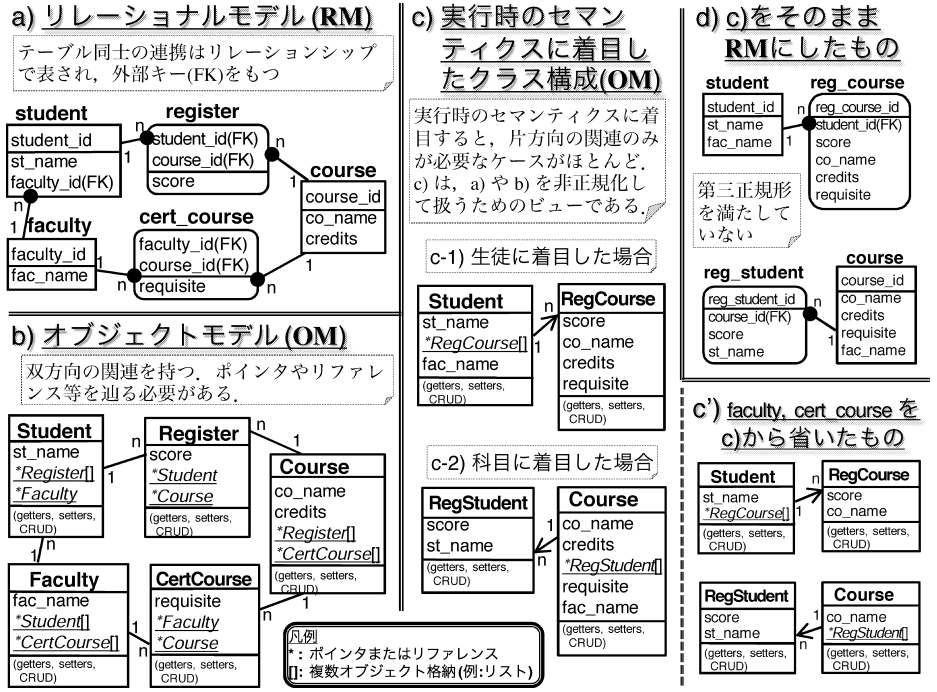


図 11　学科–生徒–履修–科目スキーマ（ER 図とクラス図）
Fig. 11　Faculty–student–register–course schema (ER diagram and class diagrams).

⋆1 magento についてスキーマ変換を行わなかった経緯を，付録 A.4 に示した．

連 L_O を頻繁にたどる必要が生じる．ここで，実行時のセマンティクスに着目し特化して設計すると，c-1) や c-2) のように明らかに簡潔なスキーマを得ることができる．c-1) は生徒（`student`）が受講する科目という観点でクラス設計をしており，c-2) はそれぞれの科目（`course`）を受講する生徒という観点でクラス設計をしている．

図 11 の c-1) や c-2) はプログラム言語にとって使いやすいクラス構成だが，これをそのままデータ永続化のためのスキーマとして用いるのはふさわしくない．d) は c) をそのまま RM にマッピングしたものだが，これは学科–生徒–履修–科目スキーマとしては RDB の第 3 正規形を満たしていない[⋆1]．データ永続化のスキーマとして c) や d) を設計した場合には，データベース正規化を実施して a) のような正規形を得る必要がある[⋆2]．RM に a) を採用し OM のビューとして c-1) と c-2) を併用すれば，生徒と科目に着目した場合について RM と OM の双方に適したスキーマを得ることができる．ただしこの場合は，RM と OM のマッピング（mOR）は複雑なものとなる．

本節を通じて，図 11 の c) や d) のスキーマを出発点として a) や b) を得ることを正規化と呼び，その逆操作を非正規化と呼ぶ．以下，4.4.1 項では，DBPowder-mdl における RM 先行型設計と OM 先行型設計を比較する（**図 12**）．4.4.2 項では，c-1) と c-2) のスキーマを正規化する例を示す（**図 13**）．4.4.3 項では，4.4.2 項に示す正規化から一部分を取り出したものについて，DBPowder-mdl，Hibernate，RoR を比較する（**図 14**）．

4.4.1　RM 先行型設計と OM 先行型設計の比較

図 12 は，図 11 の a)，b) をそれぞれ DBPowder-mdl で記述したものである．上段の a)，a’)，b) は DBPowder-mdl の CoC を活用した記述であり，下段の A) と B) は DBPowder-mdl が補完する内容を下線太字で追記した無省略記述である．

a) はプログラマが，テーブル E_R ごとに属性 A_R を記述し，リレーションシップ L_R を別途記述した．L_R で `rel` と宣言することで，DBPowder-mdl は OM 側の関連 L_O を双方向関連として導出する．したがって，プログラマが RM のみに着目して DBPowder-mdl を記述する場合は，導出される L_O の観点からみて複数の等価な記述をとりうる．その 1 つの例を a’) として示した．a) と a’) の L に関する記述は等価なので，a’) に a) のリレーション記述を加えたものは a) と等しい．

b) はプログラマが L_O で `bidir` と宣言することで，やはり DBPowder-mdl は L_O を双方向関連として導出する．`bidir` と `rel` は，いずれも OM に対する双方向リンクを意味するため，相互に書き換え可能である．

a) と b) は明らかに異なる記述だが，この 2 つが RM と OM の双方で同じスキーマを導出することを，以下に示す：

図 12　DBPowder-mdl 記述例 3：学科–生徒–履修–科目スキーマ
Fig. 12　Example of DBPowder-mdl(3): faculty–student–register–course schema.

⋆1 キー属性の選び方によっては第 2 正規形も満たさなくなる．
⋆2 OM 中心の開発アプローチをとっている場合でも，c) のスキーマは文献 29) に指摘される更新時異状などの問題をかかえるため，データ永続化層としては a) や b) のような正規化されたスキーマを得る必要がある．

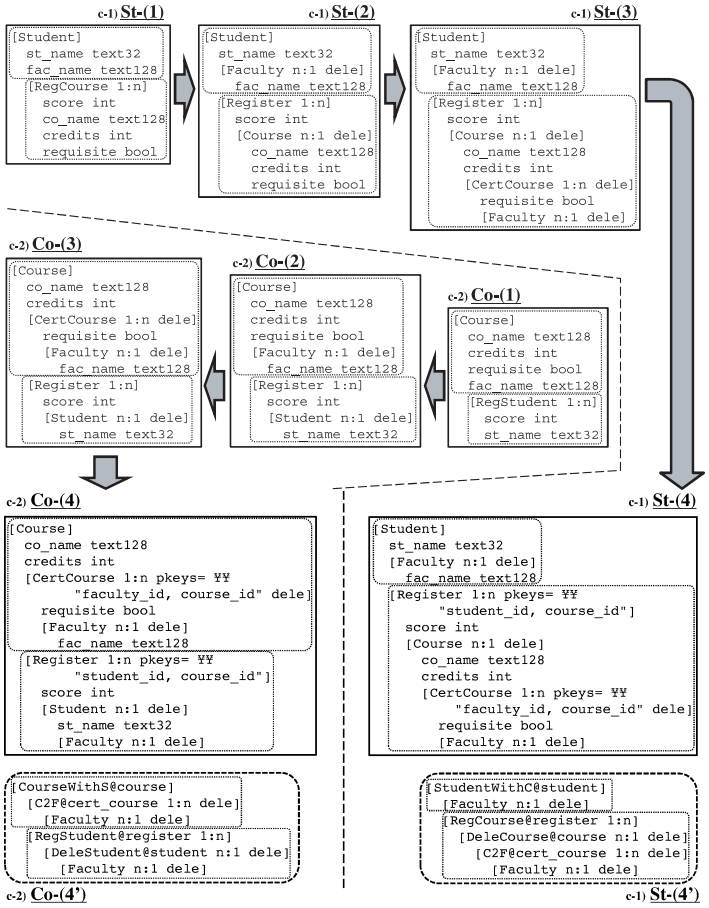


図 13　DBPowder-mdl 記述例 3：図 11 の c) を出発点とした正規化
Fig. 13　Example of DBPowder-mdl(3): Normalization.

(1) a) と b) の無省略記述が A) と B) なので，a) と A) は DBPowder-mdl として同じ意味を保ちながら相互に変換可能である．b) と B) についても同様である．

(2) B) からカーディナリティ記述と `bidir` を除去し，エンティティどうしのインデントを外すと，A) のリレーション表現と等しくなる．また，B) からエンティティ表現を抽出し，`pkeys` 宣言を除去すると，A) のリレーションシップ表現と等しくなる．したがって，B) を A) のリレーション表現とリレーションシップ表現に分解することができる．

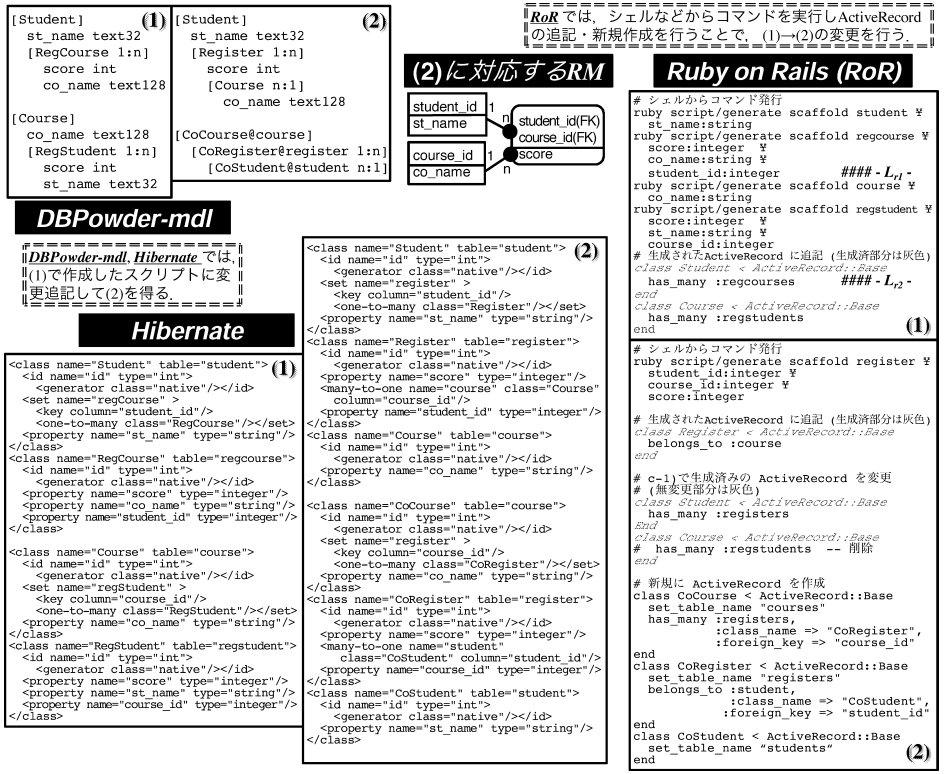


図 14　正規化過程のフレームワークごとの比較
Fig. 14　Comparison of normalization.

(3) A) のリレーション表現（5 つ）は，それぞれ独立している．この 5 つをつなげているのはリレーションシップ表現である．リレーションシップ表現にリレーション表現を重ね合わせると，B) と等しい表現を得ることができる．したがって，A) のリレーション表現とリレーションシップ表現を合成すると B) を得ることができる．

(4) 上記 (2) と (3) により，A) と B) は DBPowder-mdl として同じ意味を保ちながら相

互に変換可能である．

(5) (1) と (4) により，a) と b) は DBPowder-mdl として同じ意味を保ちながら相互に変換可能である．したがって a) と b) は，RM と OM の双方で同じスキーマを導出する．（完）

4.4.2　c-1) と c-2)（図 11）の正規化

本項では，DBPowder-mdl により図 11 の c-1) と c-2) を設計済みであるプログラマが，正規化により a) を得る過程を題材として議論する．正規化開始時点では，図 11 において c-1) と c-2) のみを既知とする．

図 13 に正規化の過程を示す．図 13 に示した DBPowder-mdl スクリプトはすべて，プログラマが明示的に記述したものである．c-1) と c-2) はいずれも，4 段階を経て RM については図 11 の a) と等価なスキーマを得る一方，OM については図 11 の c) の簡潔なインタフェースを維持している．c-1) を出発点としてプログラマが実施する正規化手順を以下に示す．

(1) 図 11 の c-1) を記述したものが図 13 の St-(1) である．

(2) St-(1) → St-(2) では，`fac_name` 属性を `Faculty` クラスに移動し，`co_name`，`credits`，`requisite` 属性を `Course` クラスに移動した．これにより，`RegCourse` クラスに `Course` クラスに関わる属性がなくなったので，`RegCourse` を `Register` に改名した．なお `dele` 宣言を記述していることから，DBPowder-mdl が生成する `Student` クラスと `Register` クラスの実行コードには，それぞれ `Faculty` クラスと `Course` クラスの属性に直接アクセスするための getter，setter メソッドが付与される．これにより St-(2) は，St-(1) と互換性のあるインタフェースを保っている．なお getter，setter メソッドは委譲によって実現される．

(3) St-(2) → St-(3) では，`requisite` 属性を `CertCourse` クラスに移動したうえで，`CertCourse` → `Faculty` への関連 L_O を追加した．St-(1) → St-(2) と同様，`dele` 宣言により getter，setter の委譲を行うことで St-(1) と互換性のあるインタフェースを保っている．

(4) St-(3) → St-(4) では，`Register` クラスと `CertCourse` クラスについて主キーを複合キーに変更した．

(5) これよにり St-(4) は，OM のクラスとしては St-(1) と同等の簡潔な非正規化インタフェースを保ちつつ，RM としては図 11 の a) と同等な正規化スキーマを実現した．

図 13 の c-2) についても c-1) と同様に，Co-(4) で OM のクラスとしては Co-(1) と同等の簡潔な非正規化インタフェースを保ちつつ，RM としては図 11 の a) と同等な正規化スキーマを実現した．なお Co-(4') は，Co-(4) からエンティティ記述 E を抜き出し，RM を同等に保ったままクラス名 E_O に変更を加えたものである．これを実現するために`@`記述を加えた．St-(4) と Co-(4') を同時に記述することで，図 11 の c-1) と c-2) のそれぞれのセマンティクスに対応するクラスを，RM における図 11 の a) を共有する形で得ることができる．St-(4') と Co-(4) の組合せでも同様のことがいえる．

4.4.3　正規化過程のフレームワークごとの比較

本項では，図 13 の St-(1) → St-(2) と Co-(1) → Co-(2) の過程を DBPowder-mdl，RoR，Hibernate で実施し比較する．ただし題材の簡素化のため，St-(1) および Co-(1) 時点でのスキーマは図 11 の c) ではなく c') とする．正規化の開始時点では，図 11 の c') のみを設計済みであるとして考える．

図 14 に，正規化過程を比較したものを示す．ここに示すスクリプトは RoR の斜字灰色部分を除いて，すべてプログラマが明示的に記述したものである．それぞれのフレームワークについて図 14 の (1) は，図 11 の c') を記述したものであり，生徒に着目した場合（`student` → `register` → `course`）と科目に着目した場合（`course` → `register` → `student`）の両方のセマンティクスに対応できている．しかし，これを 1 対 1 に RM にマッピングすると，生徒および科目を RDB 上で 2 重に管理することになるため，正規化することが望ましい．図 14 の (2) は正規化を実施した結果である．命名規約はそれぞれのフレームワークに親和性の高いものを採用しており，RM のテーブル名やカラム名についてはズレが生じている[*1]．いずれのフレームワークも，正規化の過程で RM と OM の双方を意識する必要がある点と，それぞれのフレームワークについてある程度の理解を要する点では変わらない．

Hibernate は，RM，OM，mOR のすべてを明示的に設計記述する必要があるため，RoR や DBPowder-mdl と比べて必要な記述量が大変多い．特に Hibernate は，RM の記述内容を OM の記述として再利用する機能を持たないため，(2) で属性 A_R と A_O について，`student` → `register` → `course` 方向と `course` → `register` → `student` 方向で重複して記述する必要が生じている．Hibernate は，RM，OM，mOR のすべてを記述することが原則となっている．

RoR では，CoC により Hibernate と比べて記述量の大幅削減に成功した．RoR による開発では，シェルから scaffold コマンドを実行することでクラスやテーブルなどが生成される．

⋆1 たとえば RoR のみ，生徒テーブル名が `students` となる．

関連やリレーションシップなどは，生成された ActiveRecord 上に追記する．ActiveRecord は，対象テーブルからカラムを動的に読み取ってクラスの属性とするため，(2) で属性の重複記述が発生していない．一方で，(1) で `student` → `regcourse` にリンクを張るために，離れた 2 カ所に記述する必要が生じている．該当箇所を図中 -L_{r1}-，-L_{r2}- としてマークした．また，(2) では 1 つのテーブルを 2 つ以上のクラスが参照する必要があるため，RoR の規約に違反せざるをえない[*1]．`set_table_name` によりテーブル命名規約に違反したため，クラス間のリンク `has_many`，`belongs_to` でも明示的にリンク先クラス名を指定する必要が生じている．このように命名規約に違反することで違反の連鎖が起きている．

DBPowder-mdl では，RoR と比べても大幅な記述量削減に成功した．(2) では `student` → `register` → `course` 方向の設計で属性をすべて記述し終えているので，`course` → `register` → `student` 方向では属性を改めて記述する必要がない．たとえば `[CoStudent@student]` はクラス名が `CoStudent` でテーブル名が `student` である．`student` の記述は (2) の 1 行目ですでに終えており[*2]，`st_name` という属性を持っているため，`[CoStudent@student]` も属性 `st_name` を持つ．DBPowder-mdl では，リンク L はエンティティ E どうしのインデントで表現されるため，テーブル命名規約に違反する場合でも，違反の連鎖は起きない．また，外部キー A_R は DBPowder-mdl の規約によって補完されるため，明示的な記述を必要としない．

4.5　設計手法サポート度合いの比較

表 4 に表 1 の (a)，(b)，(c)，(d) それぞれのアプローチの実用例をあげ，DBPowder-mdl とともに設計手法のサポート度合いの比較を行った．それぞれの例は以下のとおりである：(a) RoR，(b) WebML [5),7)]，(c) Hibernate．R → (O)，O → (R)，R & O，{RT} の意味は，表 4 内に説明を記載した．(d) DSL（Domain Specific Language）や専用ライブラリなどを使って SQL を直接プログラムコード内に埋め込む方式も比較対象とした．

SQL を直接コード内に埋め込む方式では，R → (O) や O → (R) の導出をサポートしない．プログラム言語や対象 DBMS そのものの記述能力の限界まで RM，OM，mOR について自由に設計開発できる（R & O）一方で，あらゆる O/R マッピングフレームワークが備えている，RDB アクセス API 自動生成機能の恩恵を受けられない（R & O）．つねに RM，OM，mOR のすべてを開発する必要があるため，設計開発スタイルとして {RT} や {impR} に至ることはできない．

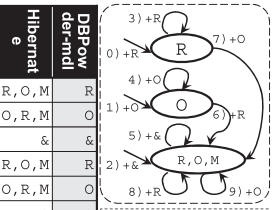
表 4　設計手法のサポート度合い比較
Table 4　Comparison among frameworks on RM, OM and mOR developments.

表1の分類	(d) SQL埋め込み	(a) Ruby on Rails (RoR)	(b) WebML	(c) Hibernate	(提案手法) DBPowder-mdl
R→(O)	×	○	△	◎	◎
O→(R)	×	×	○	△	◎
R & O	(△)	△	△	○	○
{RT} Round Trip	×	×	△	△	○
{impR}	×	×	○	○	○

R→(O)　：RMを設計，OMはRMに依存 (自動的な導出)
O→(R)　：OMを設計，RMはOMに依存 (自動的な導出)
R & O　：OMとRMの両方を設計
{RT}　：OMとRMを交互に行き来しながらの設計・開発 (Round Trip)
{impR}　：構築済みの任意のRMを取り込み，OMを生成

表 5　場面ごとの要設計モデル比較
Table 5　Comparison among frameworks.

	Ruby on Rails (RoR)	WebML	Hibernate	DBPowder-mdl
0) +R	R	R_S	R,O,M	R
1) +O	R,O_R,O_S	O	O,R,M	O
2) +&	R,O_R,O_S	R_S,O	&	&
3) R+R	R	R_S	R,O,M	R
4) O+O	R,O_R,O_S	O	O,R,M	O
5) &+&	R,O_R,O_S	R_S,O	&	&
6) O+R	R	R_S,O	R,O,M	R_O,R
7) R+O	R,O_R,O_S	O	O,R,M	O_R,O
8) &+R	R	R_S,O	R,O,M	R
9) &+O	R,O_R,O_S	O	O,R,M	O



R:RM，O:OM，M:MOR
&:RM,OM,MOR全て
O_R:RMをOMに読み替える
R_O:OMをRMに読み替える
R_S:RMのソースを直接編集
O_S:OMのソースを直接編集
R+O:設計済のRMにOMを追加 (R+R, &+Oなど，同様に読替)

RoR では，RM をプログラマが設計すれば OM は自動的に導出される（R → (O)）．設計する RM について，主キー名は `id` のみ許されるなどの強い規約がある．これに従うことで EoD が向上する一方で，構築可能なスキーマの自由度は制約を受ける（R → (O)）．導出された OM に直接コーディングすることで OM を拡張することはできるが（R & O），拡張した OM を RM にフィードバックすることはできない（O → (R)）．O → (R) をサポートしないことから，{RT} もサポートしない．RM の強い設計規約があるため，{impR} を実現することはできない．

WebML では，OM をプログラマが設計すれば RM は自動的に導出される（O → (R)）．テーブル E_R とクラス E_O およびそれぞれが保持する属性 A_E，A_O について，OM と RM の対応関係が 1 対 1 で固定される必要がある（R & O）が，OM と RM で別の名称をつけることはできる（R & O）．WebML で設計した OM からの継承や OM への委譲を用いた直接コーディングによって OM を拡張することができる（R & O）．構築済みのリレーショナルスキーマを OM に反映させる機能を持つ（R → (O)）．WebML は，OM と RM の同期がとれていれば OM と RM の完全なスキーマがつねに保たれるため，{RT} をサポートするといえるが，OM と RM の (E_O, A_O) と (E_R, A_R) の対応関係が 1 対 1 で固定される必要があるために，OM と RM の設計開発の自由度は制限される．{impR} のサポート能力は高いといえる．

Hibernate では RM，OM，mOR のすべてを盛り込んだ `.hbm` 設計ファイルを中心に開

⋆1 RoR の規約では，クラス名名詞を複数形にしたものをテーブル名とするように定めている．したがって，1 つのテーブルに対応するクラスで RoR の規約に従ったものは 1 つのみ作成できる．
⋆2 `[Student]` は `[Student@student]` と同義である．

発を進める．基本的には全要素を明示的に設計開発する必要がある一方で，開発の自由度は大きい（R & O）．*reveng* という仕掛けを使って RM から .hbm ファイルを自動生成する機能がある（R → (O)）．*reveng* による変換は，全テーブルを対象とした自動抽出をサポートする一方で条件を指定した細かい制御も可能であり，開発の容易さと拡張性を兼ね備えている．また，OM のソースコードにアノテーションをつけることで RM を記述する手法をサポートする（O → (R)）が，既存の OM の要素それぞれに対応する RM をすべて記述する必要がある．{RT} に関しては，.hbm ファイルの編集結果を *reveng* に反映させる機能を持たないことから，サポート能力が高いとはいえない．*reveng* が利用できることから {impR} のサポート能力は高い．

DBPowder-mdl は，表 1 の (a)，(b)，(c) を同時にサポートすることを特徴とする．設計量や記述量の少なさが重視される箇所やフェーズでは (a) または (b) の記法を利用し（R → (O)，O → (R)），設計自由度が重視される箇所やフェーズでは (c) の記法を利用することができる（R & O）．キー属性自動補完機能などを援用して EoD を享受できる一方で，それらを明示的に指定することで高い記述力を得ることもできる．(a) や (b) で設計済みの箇所に記述を追加して (c) に移行することも可能である．RM，OM，mOR のすべての要素が DBPowder-mdl スクリプトに集約されるため，{RT} に関しても高いサポート能力を持つ．`importDB` 記法により既存のリレーショナルスキーマの内容を DBPowder-mdl に取り込む機能を持つことから，{impR} のサポート能力は高い．

4.6　場面ごとの要設計モデル比較

本節では，表 4 で比較した 4 つのフレームワークについて，設計が必要となるモデルの比較を**表 5** に沿って行う．現在の設計開発状態から RM，OM や mOR を加える場合の数は 0)〜9) の 10 通りある．各状態の遷移図を表の右上に記した．

RoR では RM が OM に単純にマッピングされる．また，OM の属性 A_O は，実行時に RM の属性 A_R を動的に読み取ることでつど導出される．このことはフレームワークにシンプルさをもたらす一方で，OM の設計実装時につねに RM への配慮が必要であることを意味する．実際，RM のみの開発や追加の際は RM のみを配慮すればよい一方で，OM を含む開発や追加の際は以下の 3 手順を踏む必要があり，煩雑となる．

(1) OM へのマッピングを配慮に入れた RM の開発や追加
(2) RM を OM にマッピング
(3) OM のソースコードを直接編集

WebML はオブジェクト中心アプローチだが，実装済みの RM を OM に取り込む機能を持つ．これにより，OM の開発や追加だけでなく，RM の開発や追加もシンプルになる．ただし既述のとおり，OM と RM の対応関係は 1 対 1 で固定される必要があるので，取り込んだ場合の OM への副作用について配慮が必要である．

Hibernate は，RM，OM，mOR のすべての設計開発を必要とするフレームワークである．0)〜9) のすべてについて，3 者すべての設計開発が必要となる一方で，RM と OM の各要素間の対応関係を自由にできる．

DBPowder-mdl は，1 つの言語内で RM と OM 両方の記法をサポートする．これにより，RM と OM の各要素間の対応関係を自由にできるうえに，0)〜9) すべての状態遷移において，開発や追加しない側のモデルを考慮する必要性を大幅に削減した．ただし 6)，7) については，既存内容と追加内容が共有できるモデル（RM または OM）がないために，既存内容の読み替えが必要となる．この読み替えのコスト分だけ，6) は RM に特化した RoR の方が，7) は OM に特化した WebML の方がシンプルである．このケースで読み替えのコストを軽減するためには，DBPowder-mdl への熟練を必要とする．

5. 結　　論

本論文では EoD と記述力を兼備した O/R マッピング言語 DBPowder-mdl を提案した．

DBPowder-mdl は，1 つの言語で (a) リレーション中心アプローチ，(b) オブジェクト中心アプローチ，(c) 全定義アプローチを同時に実現することを特徴とする．設計量や記述量の少なさが重視される箇所やフェーズでは (a) または (b) の記法を利用し，設計自由度が重視される箇所やフェーズでは (c) の記法を利用できる．キー属性自動補完機能などの設定より規約（CoC）を推し進めることで開発の容易さ（EoD）を享受できる一方，それらを明示的に指定することで高い記述力を得ることもできる．(a) や (b) で設計済みの箇所に記述を追加して (c) に移行することも可能である．(a)，(b)，(c) のすべての要素が DBPowder-mdl スクリプトに集約され，またリレーショナルモデルとオブジェクトモデルの混在が可能であることから，柔軟なラウンドトリップ開発を可能とする．また，オブジェクトモデルの特徴である関連，継承，委譲，多態性を，パラダイムの異なるリレーショナルモデルにマッピングすることができる．実行時のセマンティクスも考慮に入れたリレーショナルモデルとオブジェクトモデルの多彩な関係を，DBPowder-mdl を用いると簡潔に記述できる．

4 章の評価を通して，DBPowder-mdl が EoD を実現する一方で複雑なリレーショナルモデル，オブジェクトモデル，およびそのマッピングを記述できることを，具体例をふまえつつ示した．

謝辞　様々な形で助言や支援を賜りました，東京農工大学の並木美太郎教授，横浜国立大学の倉光君郎准教授，KEK の皆様（川端節彌氏，金子敏明氏，湯浅富久子氏，柴田章博氏）に深謝いたします．本論文に対して貴重な示唆を賜りました査読者および編集委員の皆様に謝意を表します．本研究の一部は，情報処理推進機構（IPA）2006 年度下期「未踏ソフトウェア創造事業」の支援を受けました．

参　考　文　献

1) Ambler, S.W.: *Agile Database Techniques: Effective Strategies for the Agile Software Developer*, John Wiley & Sons, Inc., New York, NY, USA (2003).
2) Apache Software Foundation: Apache Struts Project (2010). http://struts.apache.org/
3) Apache Software Foundation: Apache Tomcat (2010). http://tomcat.apache.org/
4) Bauer, C. and King, G.: *Java Persistence with Hibernate*, Manning Publications Co., Greenwich, CT, USA (2006).
5) Brambilla, M., Comai, S., Fraternali, P. and Matera, M.: Designing web applications with WebML and WebRatio, *Web Engineering: Modelling and Implementing Web Applications*, pp.221–260 (2007).
6) Cabibbo, L. and Carosi, A.: Managing Inheritance Hierarchies in Object/Relational Mapping Tools, *CAiSE*, Pastor, O. and e Cunha, J.F. (Eds.), Lecture Notes in Computer Science, Vol.3520, pp.135–150, Springer (2005).
7) Ceri, S., Fraternali, P. and Bongio, A.: Web Modeling Language (WebML): a modeling language for designing Web sites, *Computer Networks*, Amsterdam, Netherlands: 1999, Vol.33, No.1–6, pp.137–157 (2000).
8) Chen, P.P.: The entity-relationship model—toward a unified view of data, *ACM Trans. Database Syst.*, Vol.1, No.1, pp.9–36 (1976).
9) Edd Dumbill: Ruby on Rails: An Interview with David Heinemeier Hansson (2005). http://www.oreillynet.com/pub/a/network/2005/08/30/ruby%2Drails%2Ddavid%2Dheinemeier%2Dhansson.html
10) Fowler, M.: *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional (2002).
11) Heinemeier, D. et al.: Ruby on Rails (2010). http://www.rubyonrails.org/
12) Hunt, A. and Thomas, D.: *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley Professional (1999).
13) JBoss Inc.: Hibernate (2007). http://www.hibernate.org/
14) Lang, J., et al.: Redmine (2010). http://www.redmine.org/
15) Rode, J., Bhardwaj, Y., Pérez-Quiñones, M.A., Rosson, M.B. and Howarth, J.: As Easy as "Click": End-User Web Engineering, *ICWE2005*, Lowe, D. and Gaedke, M. (Eds.), LNCS 3579, pp.478–488 (2005).
16) Magento Inc.: Magento (2010). http://www.magentocommerce.com/
17) Meijer, E., Beckman, B. and Bierman, G.: LINQ: reconciling object, relations and XML in the .NET framework, *SIGMOD '06: Proc. 2006 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, pp.706–706, ACM (2006).
18) Melnik, S., Adya, A. and Bernstein, P.A.: Compiling mappings to bridge applications and databases, *SIGMOD '07: Proc. 2007 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, pp.461–472, ACM (2007).
19) Murakami, T.: DBPowder-mdl: Mapping Description Language Between Applications and Databases, *7th Annual IEEE/ACIS International Conference on Computer and Information Science* (*ICIS 2008*), 14-16 May 2008, Portland, America, pp.127–132, IEEE Computer Society (2008).
20) Murakami, T., Yuasa, F. and Kaneko, T.: Vulnerability Management by the Integration of Security Resources and Devices with DBPowder, *Grid Camp and HEPiX Fall 2008*, Taipei, Taiwan (2008).
21) MySQL AB: MySQL Documentation (2010). http://dev.mysql.com/doc/
22) Chen, N.: Convention over Configuration (2006). http://softwareengineering.vazexqi.com/files/pattern.html
23) Oracle Corporation: Oracle Application Express (2010). http://www.oracle.com/technology/products/database/application_express/
24) Oracle Corporation: TopLink (2010). http://otn.oracle.co.jp/products/ias/toplink/
25) sourceforge.net: phpClick (2005). http://phpclick.sourceforge.net/
26) Sun Microsystems: Java Persistence API (2010). http://java.sun.com/javaee/technologies/persistence.jsp
27) SunMicrosystems: Java Platform, Standard Edition 6 Release (2010). http://java.sun.com/javase/6/
28) 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 (2006).
29) 増永良文：リレーショナルデータベース入門，サイエンス社，東京 (2003).

付　　録

A.1　DBPowder-mdl の文法

DBPowder-mdl の文法を BNF 記法で記したものを**図 15** に示す．(E, A, L, C)（表 2）のうち 3 つは，以下に対応する．

```
 1: <name> ::= {[A-Za-z][A-Za-z0-9_]*}
 2: <indentA> ::= {[ ]+}        # 所属するエンティティを示すインデント
 3: <indentL> ::= {[ ]*}        # リンク先エンティティを示すインデント．最上層ではインデントなし
    # エンティティ(E)
 4: <EntityDef> ::= <indentL>'['<EntityDesc> <Cardinality> <linkType> [<extDef>] ']'
 5:   <EntityDesc> ::= <commonEntityName>[<virtualEntityDesc>]
                     | <className>[<virtualEntityDesc>]'@'<tableName>
 6:     <commonEntityName>,<className>,<tableName> ::= <name>
 7:     <virtualEntityDesc> ::= '!'<virtualEntityName>[<virtualEntityDesc>]
 8:       <virtualEntityName> ::= <name>
      # カーディナリティ(C)
 9:   <Cardinality> ::= '1:1' | '1:n' | 'n:1' | 'n:n'    # 最上層では指定しない
10:   <linkType> ::= '' | 'rel' | 'bidir' | 'dele' | 'inherit'
11:   <extDef> ::= [ <pKeysDef>][ <joinOnDef>][ 'importDB']
12:     <pKeysDef> ::= 'pkeys="'<pKeysList>'"'
13:       <pKeysList> ::= <pKeysItem>[','<pKeysList>]
14:         <pKeysItem> ::= <name>
15:     <joinOnDef> ::= 'joinOn="'<joinOnList>'"'
16:       <joinOnList> ::= <joinOnItem>[','<joinOnList>]
17:         <joinOnItem> ::= <name>['='<name>]
    # 属性(A)
18: <AttributeDef> ::= <indentA><attrDesc> <typeName>
19:   <attrDesc> ::= <commonAttrName> | <classAttrName>'@'<columnName>
20:     <commonAttrName>,<classAttrName>,<columnName> ::= <name>
21:   <typeName> ::= 'text'{[0-9]+} | 'number'{[0-9]+} | 'datetime'
                                          ( <typeName> は，今後追加予定)
22: <AttributeDefs> ::= <AttributeDef>[','<AttributeDefs>]
23: <CommonDefStartTag> ::= '<'[<virtualEntityDesc>]['@'<tableName>]'>'
24: <CommonDefEndTag> ::= '</'[<virtualEntityDesc>]['@'<tableName>]'>'
```

凡例(BNF非標準記法)　{RE} RE は正規表現　'L' L はリテラル

図 15　DBPowder-mdl の BNF 記法による文法（中心部分）
Fig. 15　BNF of DBPowder-mdl (main part).

- エンティティE　図 15 L4（<EntityDef>）
- 属性 A　L18（<AttributeDef>）
- カーディナリティC　L9（<Cardinality>）

リンク L は，2 つのエンティティ（E_{outer}，E_{inner}）のインデント上下関係により記述する．

以下，付録 A.1.1 では CoC による E と A の命名規約について述べる．付録 A.1.2 以降では図 15 の BNF を参照しつつ，主要 4 要素（E，A，L，C）について述べる．

なお付録 A.1 では，それぞれの節の内容に対応する BNF の行番号と左辺の主なシンボルを，タイトル内に示す．

A.1.1　図 15 L5，19（<EntityDesc>,<attrDesc>）：CoC によるエンティティE と属性 A の命名規約

本項は省略記法 commonName および無省略記法 objectName@relationName（3.3.1 項）の命名規約について述べる．

DBPowder-mdl の CoC による命名規約として，テーブル E_R，カラム A_R，属性 A_O は under_score_style [⋆1]，クラス E_O は CamelStyle [⋆2]，メソッド名は camelStyle [⋆3]とする．3.3.1 項の省略記法を用いると，ここに述べた命名規則により無省略記法の（objectName, relationName）が導出される．プログラマが明示的に無省略記法で記述した場合は，本項の命名規約は用いられない．

3.3.1　項の省略記法に対応する E や A の BNF は，L5（<EntityDesc>）の <commonEntityName>... や L19（<attrDesc>）の <commonAttrName> である．無省略記法に対応するのは，L5（<className>...）や L19（<classAttrName>...）である．

この規約はカスタマイズできる（3.5.2 項）．

A.1.2　図 15 L18–L22（<AttributeDef>）：属性 A

DBPowder-mdl では，属性 A を名称とデータ型の組で表し，所属するエンティティE から 1 つ内側のインデントに記述する．図 4 の例では fac_name text128 などが該当する．

プログラマは L19（<attrDesc>）により A_O と A_R の名称を記述する．記述方法は付録 A.1.1 項で述べたとおりである．なお L22（<AttributeDefs>）により <AttributeDef> をカンマ区切りで並べて記述すると，複数の属性を 1 行にまとめることができる．

A.1.3　図 15 L4–8，11（<EntityDef>）：エンティティE

DBPowder-mdl では，エンティティE を BNF の L4（<EntityDef>）で記述する．

- L5（<EntityDesc>）　E_O と E_R の名称を記述．
- L9，10（<Cardinality>，<linkType>）　C と L を記述．付録 A.1.4 項で述べる．
- L11（<extDef>）　省略可能な記述．

図 4 の例では [faculty] や [course] などが該当する．

プログラマは L5（<EntityDesc>）により E_O と E_R の名称を記述する．名称の記述方法は付録 A.1.1 項で述べたとおりである．<className> や <tableName> を重複して記述できる．この場合の挙動は 3.3.2 項に示したとおりである．なお <virtualEntityDesc> に

⋆1 under_score_style: 小文字単語をアンダースコア（_）で連結したもの．
⋆2 CamelStyle: 単語の頭文字を大文字にしてそのまま連結したもの．
⋆3 camelStyle: CamelStyle に変換した名称の先頭文字のみを小文字としたもの．

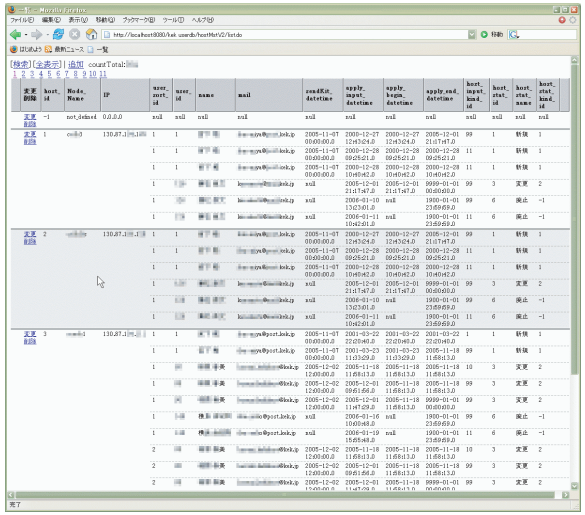


図 16　追加実装テンプレートを用いて生成した，CRUD 機能実現ウェブページ[28)]

Fig. 16　Web page generated by DBPowder-mdl with CRUD functions.

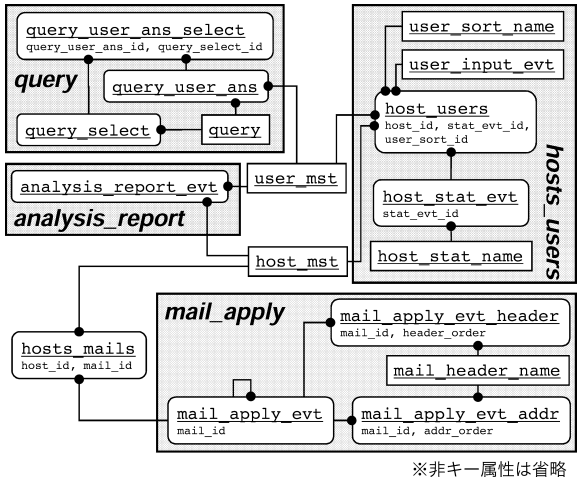


図 17　DBPowder-mdl プロトタイプシステムを用いて開発したアプリケーションの ER 図（一部）

Fig. 17　ER diagram of an application developed by the DBPowder-mdl prototype system (portion).

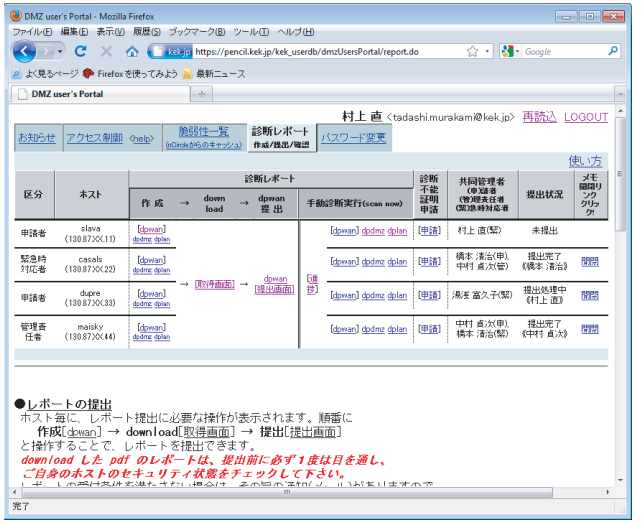


図 18　DMZ User's Portal

Fig. 18　DMZ User's Portal.

より実装インタフェース（3.5.3 項）を指定できる．

プログラマは L11（<extDef>）により，L12（<pKeysDef>）や L15（<joinOnDef>）記述を用いて，E_R の主キーと外部キーを明示的に指定できる．3.3.3 項で述べた規約に従うならば，<pKeysDef> や <joinOnDef> を記述する必要はない．

プログラマが L11（<extDef>）で importDB を記述すると，RDB に現在設定されている定義が E に統合される（3.5.4 節）．

A.1.4　図 15 L9，10（<Cardinality>,<linkType>）：カーディナリティ C とリンク L

本項で述べる C と L は，2 つのエンティティ（E_{outer}，E_{inner}）のインデント上下関係に対して作用するものであり，E_{inner} に記述する．E_{inner} の 1 つ外側のインデントにあるエンティティを E_{outer} とする．インデントの最上層では自らよりも外側のエンティティを記述することができないため，C と L も記述できない．

プログラマは E_{inner} 内で L9（<Cardinality>）によりカーディナリティ C を記述する．図 4 の例では，(b1) の [course 1:n] などが該当する．1:1，1:n，n:1，n:n をサポートする．なお，1::1，1::n，n::1，n::n とコロンを重ねると，リレーショナルスキーマに外部キー制約をかけることができる．

プログラマは E_{inner} 内で L10（<linkType>）によりリンクの種類を記述する．<linkType> は，無記述，rel，bidir，dele，inherit の 5 種類があるが，いずれも OM の関連 L_O のみに作用し，RM のリレーションシップ L_R には作用しない．無記述の場合と，rel，bidir，dele を指定したときの動作は，3.3.2 項に示したとおりであり，残りの inherit については 3.5.5 項で述べたとおりである．

A.1.5　図 15 L23，24（<CommonDefStartTag>，<CommonDefEndTag>）：CommonDef 記法（3.5.4 項）による記述

プログラマは L23（<CommonDefStartTag>）と L24（<CommonDefEndTag>）の CommonDef 記述により，タグ内に囲まれたエンティティ E のすべてにデフォルト値として，指定した仮想エンティティ（!<virtualEntityDesc>）やテーブル（@<tableName>）を割り当てることができる．3.5.5 項の（SR）で，記述例を示した．

A.2　リンク L とエンティティ E

DBPowder-mdl では，リンク L を任意のエンティティ E 間に張ることができる．このことを以下に示す（3.3.2 項の補足）．

エンティティ E_0，$\{E_f\}$，$\{E_t\}$ があり，

$\{E_f\} = \{E_{f_1}, ..., E_{f_m}\}$

$\{E_t\} = \{E_{t_1}, ..., E_{t_n}\}$

とする（m，n：任意の自然数）．なお E_0，$\{E_f\}$，$\{E_t\}$ には，同じエンティティが複数含まれてもよいものとする．ここで，リンク $\{L_f\}$，$\{L_t\}$ を以下のように定める．

$\{L_f\} = \{L_{f_1}, ..., L_{f_m} | L_{f_i} : E_{f_i}$から E_0へのリンク $\}$

$\{L_t\} = \{L_{t_1}, ..., L_{t_n} | L_{t_i} : E_0$から E_{t_i}へのリンク $\}$

DBPowder-mdl ではエンティティの重複記述により，以下のように記述できる．

- $\{E_f\}$ それぞれの内側インデントに，E_0 をそれぞれ記述する．
- 上記とは別に E_0 を記述し，その内側インデントに $\{E_t\}$ をすべて記述する．

E_0，$\{E_f\}$，$\{E_t\}$ には制約条件がないことから，$(E_0, \{E_f\}, \{E_t\}, \{L_f\}, \{L_t\})$ に対してこれを再帰的に適用することで，DBPowder-mdl の記述から E，L による任意の有向グラフを構成できる．したがって DBPowder-mdl では，リンク L を任意のエンティティ E 間に張ることができる．（完）

A.3　規約のカスタマイズ例

本節では，3.5.2 項で述べた規約のカスタマイズを 4.1 節のプロトタイプシステムで開発した例を示す．4.1 節のプロトタイプシステムでは，Java のポリモーフィズムを活用し，同一の interface をもった独自実装クラスを DBPowder-mdl の起動時に指定することで，カスタマイズを実現した．

実装した命名規約は 2 つあり，RM のリソース名をすべて大文字または小文字に統一するものと，テーブル名の末尾に `_mst`，`_trn`，`_name` のいずれかがつく場合は DBPowder-mdl のリソース名として取り除くものである．後者の命名規約では，たとえば `user_mst` というマスタテーブルの主キーは `user_mst_id` ではなく `user_id` であり，これに対応するエンティティ名は `User` となる．

A.4　magento（4.3 節）を RoR の規約に合致させるための作業見積

4.3 節で，magento を RoR（Ruby on Rails）の規約に合致させることは，評価を実施する前作業としては適切ではないと判断した．本節では，そのように判断した経緯を述べる．

magento では，RoR の規約に従ったテーブルは 0 個である．よって，規約に従ったスキーマに変換するためには少なくとも以下の作業が必要である．

- 239 個の全テーブルおよび主キーと，319 個（grep による概算見積り）のすべての外部キーについて，RoR の規約に合致するように名称変更
- 上記にともない，319 個（grep による概算見積り）のすべての外部キー制約を再構築

比較題材のリレーショナルスキーマについて，RoR のみ上記のような大幅な変換を施すのは，比較実験として妥当かどうか疑問の余地があった．そこで magento の比較では，RoR，Hibernate，DBPowder-mdl について比較題材のスキーマに変換を施さないこととした．

なお 2.1 節で述べたように，RoR の導入対象や使用方法として magento が適切かどうかは議論の余地があるが，今回はあくまで記述量の評価を目的とした．

A.5　DBPowder-mdl プロトタイプシステムの参考資料

DBPowder-mdl プロトタイプシステムと，これを用いて開発したアプリケーション群（KEKapp）に関する参考資料を**図 16**，**図 17**，**図 18** に示す．資料はいずれも，4.1 節に関するものである．

(平成 22 年 3 月 20 日受付)
(平成 22 年 7 月 7 日採録)

（担当編集委員　富井 尚志）

村上　　直（正会員）

平成 11 年京都大学工学部情報学科卒業，平成 13 年東京大学大学院理学系研究科情報科学専攻修了．SE 職を経て，現在高エネルギー加速器研究機構（KEK）計算科学センター助教．データ工学の研究に従事．電子情報通信学会，日本データベース学会，IEEE-CS，ACM 各会員．