Vol.2017-IS-140 No.1
2017/6/3

オブジェクトの正規性を重視して全面的に存在従属関連を用いたドメインモデリング手法の提案

金田 重郎[1,a)]　井田 明男[1]

概要：オブジェクト指向モデリングでは，1) 属性値のライフタイムはインスタンスのライフライムと一致し，2) インスタンスのすべての属性値が，当該インスタンスのオブジェクト ID(プライマリーキー) に非推移的に関数従属することが望まれる．しかし，現実の設計でには，関連した他インスタンスの属性値をコピーして保存し，以後の処理に利用する事が多い．ビューとでも言うべきこの追加属性は，クラスの正規性を破壊し，AP のモジュラリティを低下させる．この問題を回避するため，本稿では，(1) 存在従属関連のみで永続化すべきクラスを記述し，(2) 生成されたインスタンスは消さない，アプローチを提案する．ただし，実際には，インスタンスはアップデートされ，あるいは消去される．その情報を表現するため，インスタンスには，当該バージョンの有効期間を示すタイムスタンプ属性を設定し，更に複数のバージョンのインスタンスを設けて，データ状態変化のログを記録する．これにより，各インスタンスは，時点さえ与えられば，存在従属関連を上流方向に辿り，タイムスタンプを参照しながら，当該時点において適切な属性値を選択・利用できる．結果として，任意の時点で，多重度 1 でポイントされる範囲内にある他インスタンスの属性値を辿ることを保証しつつ，クラスの正規性を担保できる．ただし，本提案のメカニズムにおける，(1) 実規模問題への適用性の検証，(2) 提案されたメカニズムを実現するプラットフォームの検討・実現等，は今後の課題である．

キーワード：存在従属関連，クラスの正規化，ライフタイム，タイムスタンプ，クラス図

1. はじめに

アプリケーション・ソフトウェアが内部に持つオブジェクトを永続化するため，関係データベース (以下 RDB) を用いることは，極めて一般的である．RDB では正規化は必須のものとされている．しかし，そこに写し取られるべき，オブジェクトの正規化については，必ずしも必須のものとしては見られていない様に思われる．具体的には，あるクラスのインスタンスを生成する際に，インスタンスの正規性が壊れることを承知した上で，新設するインスタンスに関連した他インスタンスから属性値コピーして，属性値とすることはしばしば行われる．しかし，この様なビュー的な属性値をクラスに取り込むと，そのクラスの構造が，第 3 正規形から逸脱する恐れがある．それは，望ましくない．

本稿では，永続させるべきクラスは，2 つの点から正規性を担保されねばならないと考える．第 1 に，インスタンスの属性値のライフタイムが，インスタンスのライフタイムと一致していることである [1][2] [*1]．第 2 に，インスタンスの属性値は，当該インスタンスのオブジェクト ID から「非推移的に」関数従属していることが望ましい[*2]．この 2 点を満足するクラスを「正規化されたクラス」と本稿では呼ぶ．

すべてのクラスを正規化するために，本稿では，井田による存在従属クラス図を用いる [3]．井田の存在従属関連では，「1 対多」関連の「1」側の多重度は「1 それに限る」である．言い換えると，下流側のインスタンスからは，存在従属の上流側にあるインスタンスを，辿れることになる．クラスの正規化を担保しつつ，関連した属性値が必要になった時点で，存在従属関連を上流方向に辿り，必要な属性値を取り出せば良い，と言うのが本稿の基本的スタンスである[*3]．

[1] 同志社大学理工学部
〒 610–0321，京都府京田辺市多々羅都谷 1-3
[a)] skaneda@mail.doshisha.ac.jp

[*1] 税金の例でいえば，納税義務者の氏名や住所は，税自体のライフライムとは一致しない．税オブジェクトに，納税義務者のデータをコピーすると，同一情報が複数個所に存在し，保守性に問題を生じる．

[*2] 例えば，税オブジェクトには納税義務者の氏名，住所を属性として持たせたい．しかし，税賦課の直接的な原因は，（特定市民による特定固定資産の）所有であって，税と所有者の間には，直接的な因果関係，すなわち，非推移的関数従属性は存在しない．

[*3] 正確には，着目している下流側インスタンス自身，及び，当該イ

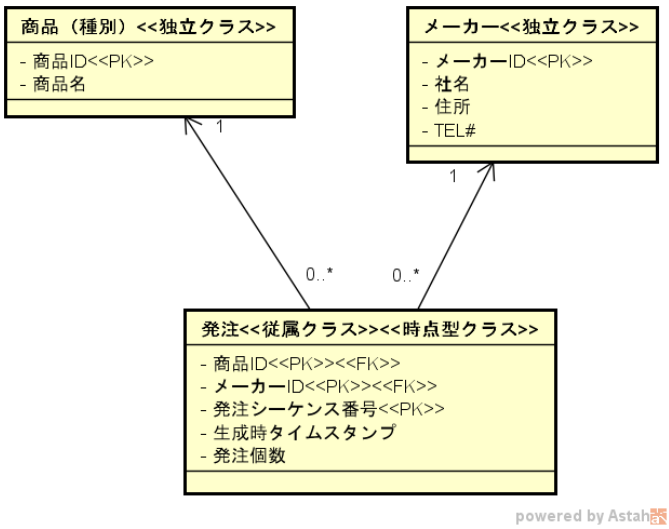

図 1 存在従属クラス図の簡単な例（多重度付）

ただし，生成された存在従属関連は，そのまま永久に繋がっている訳ではない．多重度が「1 それに限る」とは，「下流側イスタンンス」と「関連」が同時に生成されたり，削除されたりするとの意味である．ポインタである関連が消えては，逆に辿ることはできない．

そこで，本稿では，いつでも，存在従属関連を辿れる様に，一度生成した存在従属関連，及びインスタンスも消さないで永続化する．ただし，実際のソフトウェア上の意味では消去されたり，アップデートされているので，これを表現するために，タイムスタンプを導入し，インスタンスをバージョン管理して，後から，好きな時点で，存在従属関連を辿ることを可能とする．

以下，第 2 章では，存在従属クラス図について説明する．そして，対象業務を存在従属クラス図のみで記述できることを示す．第 3 章では，2 つのインスタンスのライフサイクルタイムの分析から，インスタンスに，「時点型クラス（タイムスタンプ 1 個）」及び「期間型クラス（タイムスタンプ 2 個）」を設けるべきことを提案する．次に，第 4 章では，タイムスタンプによりバージョン管理されたインスタンスを用いることにより，任意の時点における，存在従属関連を状両方向へ辿った処理が実現できることを示す．第 5 章は，上記提案手法の適用の簡単な例をしめす．第 6 章はまとめである．

2. 存在従属クラス図

2.1 存在従属クラスと多重度

図 1 は，商品をメーカーに発注できることを示す存在従属クラス図の例である [3]．ただし，井田の記法とは異なり，多重度を描いている．存在従属クラス図のクラスには 2 種類あり，「独立クラス」は，インスタンスを他のインスタンスの状態に依存せずに生成できるクラスである．図 1 の商品クラスとメーカークラスが独立クラスである．これに対して，「従属クラス」のインスタンスは，他の特定クラスのインスタンスを指定しないと生成できない．図 1 では，発注クラスがこれにあたる[*4]．

結果的に，発注クラスは，上流側の 2 つのクラス（商品クラスとメーカークラス）の存在に依存[*5]しており，→ でその依存関係を表現している．ここで，→ の先端側クラスを「上流側クラス」，反対側のクラスを「下流側クラス」という．存在従属クラス図は，全体として DAG(サイクルを持たな有向グラフ) を構成するので，あるクラスから，存在従属関連（→）を，順番に追ってゆくことができる．これは，一種の関数従属性であり，推移的な関数従属ではなく，非推移的な，直接的な材料（クラス）を，直近の上流側に持ってくるべきことを意味している．本稿では，この「多重度＝ 1 それに限る」の関係を関数従属性と呼ぶ．

存在従属クラス図では，下流側のクラスは，上流側の PK（Primary Key）をそのまま，FK(Foreign Key) として，プライマリーキー集合に取り込んでいる．下流側の多重度は，通常は，0..*である．このため，複数のインスタンスを生成可能とするため，図 1 では，発注クラスのシーケンス番号（発注シーケンス番号）を PK として含めている．ただし，この枠組みがあったからと言って，発注インスタンスは，同一の商品とメーカーの組み合わせに対して一つとアプリ側で規制することは何ら問題ない．そのようなビジネスモデルなら，図 1 の構成を変えずに，下流側は 0..1 とできる．

ンスタンスから辿れる存在従属の上流側インスタンスだけではなく，これらのインスタンスを上流側として，下流側の多重度が「0..1」及び「1」でリンクしているインスタンスも辿ることができる．

[*4] 椿によるクラスの種別 [4] と比較すると，独立クラスは椿のリソースクラス，従属クラスは椿のイベントクラスに近い．
[*5] AND 条件．

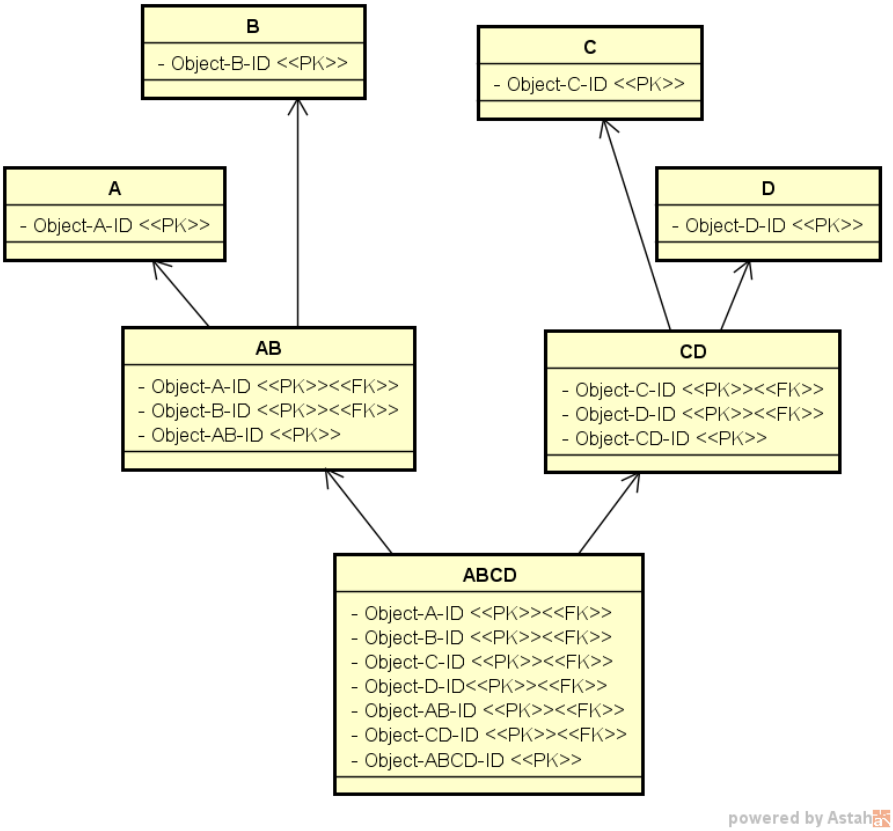

図 2　存在従属クラス図における PK の継承

井田の存在従属クラス図では，下流側のクラスの PK は，上流側のクラスの PK の和集合である．したがって，着目しているクラスの上流側クラスがさらに上流側クラスから PK を継承する．ただし，独立クラス，従属クラスを問わず，各クラス独自の PK(オブジェクト ID) は追加されるものとする．その様子を図 2 に示す．上流側のクラスのオブジェクト ID がつぎつぎと継承されている．存在従属関連の上流側の多重度は 1 であるから，下流側クラスのインスタンスが決まれば，上流側のインスタンスは唯一決まる．これは，関数従属性に類似であり，非推移的であることが期待される．

2.2　存在従属クラス図の記述性

存在従属クラス図では，1 対多の「1 側」の多重度が「1 それに限る」に限定している．但し，井田は，すべての関連を存在従属にしろとは言っていない．しかし，これにより表現能力が限定されるのではないかとの懸念はあり得る．

しかし，現実を見ると，大半の関連は，存在従属関連と同じ多重度「1 それに限る」にできる様である．例えば，よく知られた ER 図設計書である渡辺幸三 [5]，佐藤正美 [6] らの著作を見ても，片方の多重度が「1 それに限る」となっているものが多い．その理由を理論的に論じることは難しいが，以下の定性的な分析は可能である．

図 3 は，2 つのインスタンスの時間的関係を列挙したものである．2 つのインスタンス間の時間的関係は，本質的に 4 通りしか存在しない．「同一」ではライフタイムは一致しており，2 つのインスタンスに分けておく必然性がない．「乖離」では関連を引くことは不可能である．「すれ違

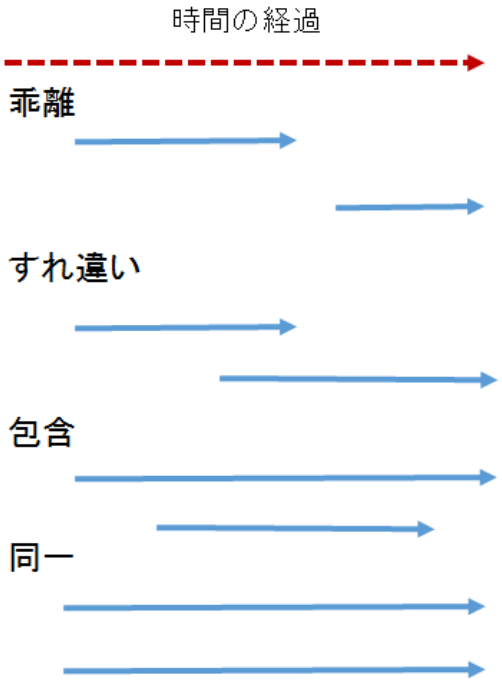

図 3　2 つのインスタンスの時間的関係

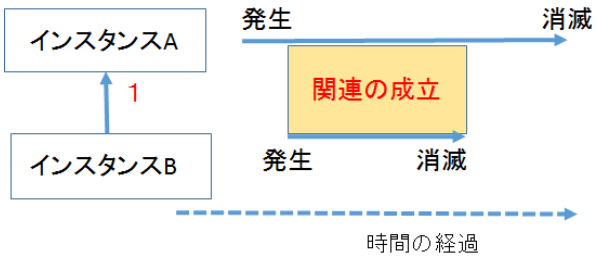

図 4　存在従属関連

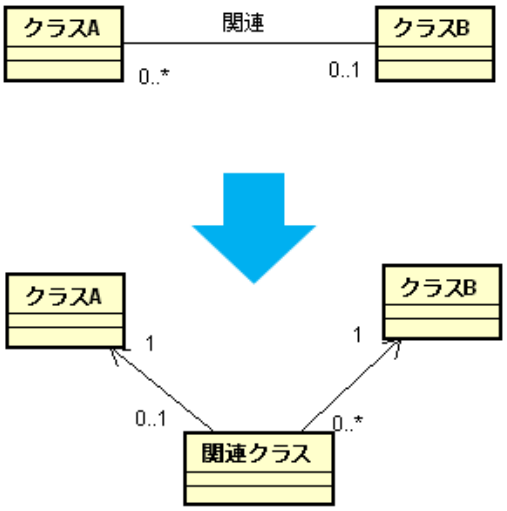

図 5　関連クラスによる完全存在従属化の例

い」は時間的にオーバラップするが，包含にはならない．「包含」では，一方のライフタイムに他方のライフタイムが包含される．包含の場合，ライムタイムの短い側のインスタンスのライフタイムと，関連のライフタイムが一致すると，井田の存在従属関連 [3] となる．

渡辺や佐藤らの文献でも，片方の多重度が 1 の ER 図が多いことは，多くのエンティティ間の関係は包含型であり，図 4 の様に，ライムタイムが短い側のインスタンスのライフタイムと，関連が引かれている期間（ライフタイム）が一致することが多いという意味であろう．

但し，図 3 の「すれ違い」のケースは存在従属では表現できない．多重度がどうしても「0..1」となるからである．この場合，佐藤のアプローチ [6] に近いが，図 5 の様に，関連クラスを設けてやれば存在従属化できる．しかし，もともと，すれ違うことは，インスタンス間に独立性が強く，結果的に意味的関連が弱く，関連を引かれることも少ないと想定される．

3.　提案手法

存在従属関連では，下流側のインスタンスがひとつ選ばれると，その上流に存在する他インスタンスが一意に決まる．本稿では，この性質を利用して，(1) アプリケーション側におけるクラスをあくまでも正規化して，将来のメンテナンス容易化を狙いとする．そして，(2)「着目しているインスタンスから一意に決まる（他インスタンス上にあり，しばしば業務に必要となるであろう）関連データ項目」は，必要になった段階で，存在従属関連を用いて取り出す，ことである．但し，それを可能とするためには，存在従属関連の扱いに，若干の変更を加える必要がある．以下に順に説明する．

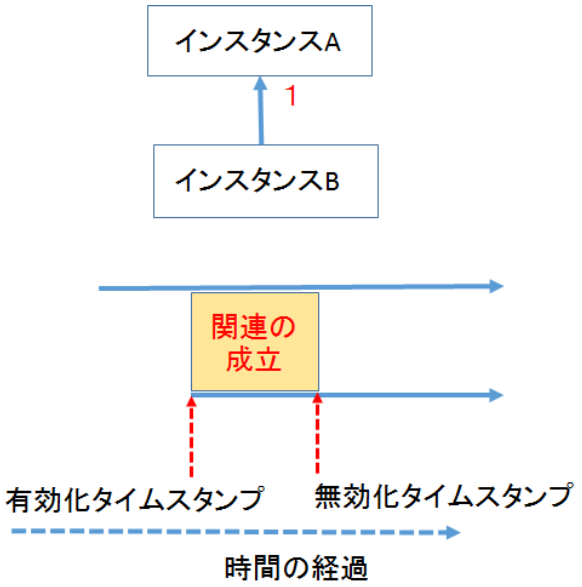

図 6　存在従属関連の表現方法

3.1　インスタンスの永続化

本提案手法では，第一に，一度生成したインスタンスは消さない．存在従属関連の上流側の多重度は，「1 及びそれに限る」である．このことは，2 つのインスタンスが常に繋がっていることを意味するものではない．図 4 に示した様に，存在従属関連は一度つながっても，下流側のインスタンスとともに消えることがある．これでは，後からの関連インスタンスからのデータ取得のポインタとはならない．そこで，本稿では，存在従属関連にポインタとしての永続性を担保するため，一度生成したインスタンスはすべて消さないこととする．

ただし，アプリケーションから見て，消えているべきインスタンスが残っていることは許されない．このため，図 6 の様に，有効化タイムスタンプと無効化タイムスタンプを導入し，ソフトウェアから見た（ある特定のタイミングにおける）インスタンスの実在は，タイムスタンプにより判定する．この様に，有効化タイムスタンプと無効化タイムスタンプを持ち，ある期間のみ成立するインスタンスを，本稿では，「期間型インスタンス」と呼ぶ．

尚，タイムスタンプが 1 個でよいケースがある．椿らの言う「イベント型インスタンス」である．ある一瞬の出来事の記録がインスタンスとなる．本稿では，これを「時点型インスタンス」と呼ぶ．時点型インスタンスでは，本来，

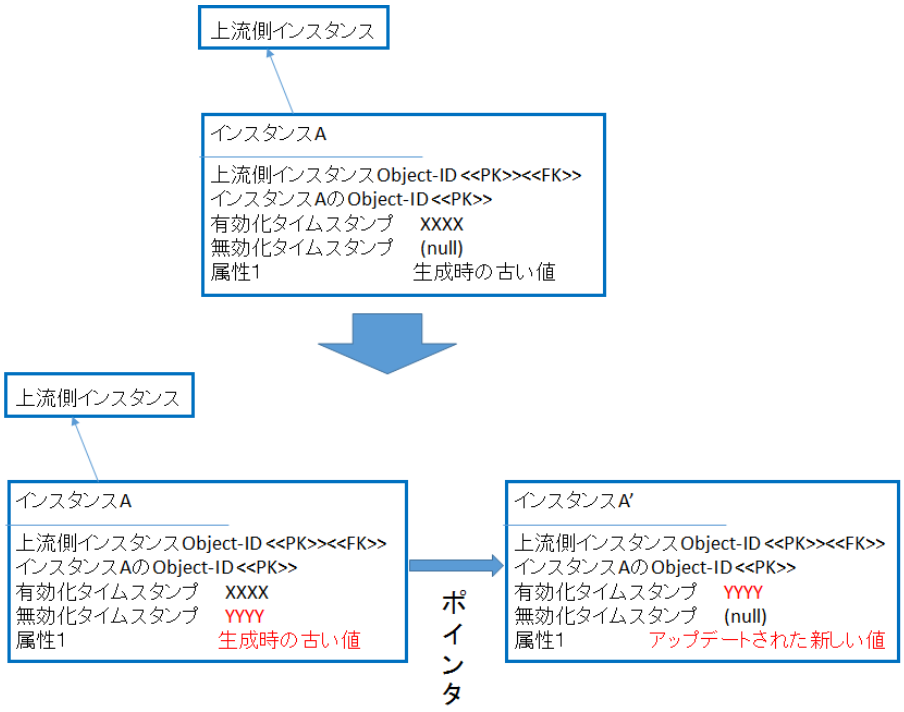

図 7　生成されるインスタンスの LOG

属性値を修正・追記することは望ましくない．後から追記する場合には，時点型クラスのインスタンスを上流側として，下流側に新規にデータ項目を記載したインスタンスを存在従属させるべきと考える．

一方，期間型クラスでは，インスタンスが実在している期間内に，属性値がアップデートされることがある．後から特定の時点（タイミング）で，データにアクセスする以上，インスタンスの変化は，ログとして残しておく必要がある．図 7 は，このインスタンスの履歴管理のイメージである．最初，インスタンスは，2 つのタイムスタンプの中で，有効化タイムスタンプしか持たない．しかし，属性値 1 の値がある時点で書き換えられると，書き換える前までの期間を無効化タイムスタンプで表現する，最初のバージョンと，書き換えられた時点を有効化タイムスタンプとする新しいバージョンのインスタンスが出現する．追加されたバージョンへは，最初のバージョンから辿ってゆける．

上記の理由で，本手法が対象とするクラスは，すべて，時点型か期間型に分ける必要がある．その上で，期間型に対する更新と削除の概念を以下の様に変更する．

- 更新とは，更新前の期間型のインスタンスに無効化タイムスタンプを設定して，新しい期間型のインスタンスをバージョンとして追加すること
- 削除とは，削除前の期間型のインスタンスに無効化タイムスタンプを設定すること

3.2　関連データアクセス

以上の構成により，着目しているインスタンスから上流側に存在するインスタンスの属性値はいつでも，アクセス可能である．各インスタンスの上流側にあるインスタンスは，消されずに残っている存在従属関連を辿ることで行う．従って，各インスタンスは，自分の上流側に存在するインスタンス名とどの FK でアクセスするべきかはわかっている必要があり，インスタンスが持つ操作として，個別の上流側インスタンスへのアクセス法が提供されるべきであろう．

ただし，正確には，辿れるのは，着目しているインスタンスから上流側に存在するインスタンスのみではない．着目しているインスタンスを含めて，存在従属の上流方向に辿れるインスタンスから 1 対 1 の多重度でリンクしている（存在従属の下流側に辿ることになる）インスタンスも一意に決定可能である．

また，特定のインスタンスが持つ属性値へのアクセスを実行する操作は，最新の属性値を取ってくるのか，過去のある時点の属性値がほしいのかは指定可能とするべきである．例えば，図 7 のような構造をターゲットのインスタンスが持つ場合，最新の属性値がほしいなら，最初に生成され，すでに無効化されているインスタンス A から出発して，リンクされた最新のバージョンが利用される．

上記の枠組みには，ひとつの検討事項がある．「着目しているインスタンスが決まれば，そのインスタンスが，から関数従属的に 1 個だけユニークにインスタンスが指定されるクラス」が上記の枠組みで網羅されているのかという疑問である．つまり，着目しているインスタンンスから関数従属的に指定される（あるクラスの）インスタンスで，上

流側クラスに属さない場合があるのか？との疑問である．

この点については，モデルが推移性を伴わないように，非推移的な関数従属に相当するリンクで構成されていれば，アプリケーションが持っていた関数従属性はモデルに現れていると信じたい．しかし，この判断が正しいか否かは検証を必要とする．

以上の議論から，関連データへのアクセスは以下のステップによる．

(1) 着目しているインスタンスは，一意にそのインスタンスを指定できるクラスのリストを有し，そのクラス中の特定の 1 個のインスタンスを指定する FK は，もともと，そのインスタンスの PK の一部として持っている．
(2) 上記リストから，（プログラムで指定された）必要なクラス名を取り出し，FK に従って，ターゲットのインスタンスを検索する．これによって，上流側のインスタンスをひとつ特定する．
(3) 上記特定されたインスタンス，または，バージョン管理されているインスタンス群の中から，現時点のタイミングに合致するもののみを取り出して，その属性値を利用する．
(4) なお，上記で特定された上流側インスタンスから，（着目しているインスタンスの上流側には属さない）多重度が「0..1」または「1」でリンクされたインスタンスがある場合には，リンクを辿りそのインスタンスを特定し，リンクされたインスタンスの属性が利用可能となる様に考える必要がある[*6]．

3.3 存在従属クラス図を用いる利点

本提案方式では，クラスを正規化している．従って，保守上のメリットが得られるはずである．それは，提案方式の大きな目的の一つである．しかし，そのことは，存在従属クラス図を用いなければならない理由ではあり得ない．ただし，存在従属クラス図を利用すると，存在従属クラス図では，PK に多くの FK を用いているので，下流側にインスタンスが存在する状態で，上流のインスタンスを消去する（バグである）と，DBMS による警告を受けるメリットはある．通常のオブジェクト指向の考え方を採用し，各クラスに PK として，オブジェクト ID を与えて，正規化されたクラス群を実装したのでは，このようなインテグリティ・チェック機能は期待できない．

しかし，各クラスに PK として，オブジェクト ID を与える通常の実装方式が採用されているとしても，（提案方式と同様に）クラスを正規化し，必要になった時点で，着目しているオブジェクト（インスタンス）に関連した業務データを他のインスタンスから取得することはあり得ることである．クラスの識別にオブジェクト ID を使っていようと（従来方式），存在従属クラス図を使っていようと，プログラマは，データの存在するインスタンスを特定する情報と，データ取り出しの必要な属性名を知らずして，プログラミングはできないであろう．しかし，存在従属クラス図を利用した場合と，一般のオブジェクト ID を利用した場合で，以下の差異が生じると考えている．

(1) 存在従属クラス図では，→ が関連に付属しているので，着目しているインスタンスから，どの範囲のクラスのインスタンスを一意に特定できるかが直感的に把握できる．
(2) 存在従属クラスのインスタンスでは，(a) 自分の上流側に存在する一意に特定できるインスタンスが付属するクラス名が分かっている，(b) ターゲットとなっているクラスに属している，ターゲットインスタンスを特定する PK を，自分の FK として持っている．(c) このため，必要なクラス名，属性名さえ，着目しているインスタンスにメッセージパッシングすれば，ターゲットとなったインスタンスから属性値を取得できる．
(3) これに対して，オブジェクト ID を用いた従来方式では，ターゲットインスタンスに関して，必要なクラス名，属性名をもらっても，ターゲットクラスのどのインスタンスかは特定できない．このため，何か別の属性値を自己のインスタンスから取り出して，ターゲットインスタンスを探すようなルーチンを書かなくてはならない．この「別の属性」は，おそらく，クラスやアプリケーションによってさまざまであろう．これが面倒なので，インスタンスを生成する時に，あらかじめ，関係したビュー的なデータを，インスタンスにコピーしているとも理解できる．

上記の項目 (2)(3) を比較すれば，すべての上流側クラスの PK を継承している存在従属クラス図のメリットは明かと思われる．ビュー的なデータ項目をアクセスするためのメソッドを自動生成するのに，どちらが有利かは自明である．上流側インスタンスから，下流側の多重度「1」，あるいは「0..1」で，ユニークにつながっているインスタンスについても，上流側インスタンスのどれに繋がっているかで，PK の成分は判明する．同じく，アクセス関数は，存在従属クラス図から自動生成可能と思われる．

4. 具体的な例と考察

図 8 は，より具体的な例として，固定資産税のモデルを作成した例である．ただし，実際の地方税法の固定資産税をカバーするものではなく，説明のための簡単化されたモデルである．固定資産は，共有があり得るが，その際の固定資産税の支払義務は，共有の所有者が持ち分に比例して追う．しかし，実務では，登録上の筆頭者のみに請求したり，各人に請求したりとケースバイケースである．

[*6] 実際，図 8 における左下の税金を引き落とす口座番号のクラスは，これに該当する．

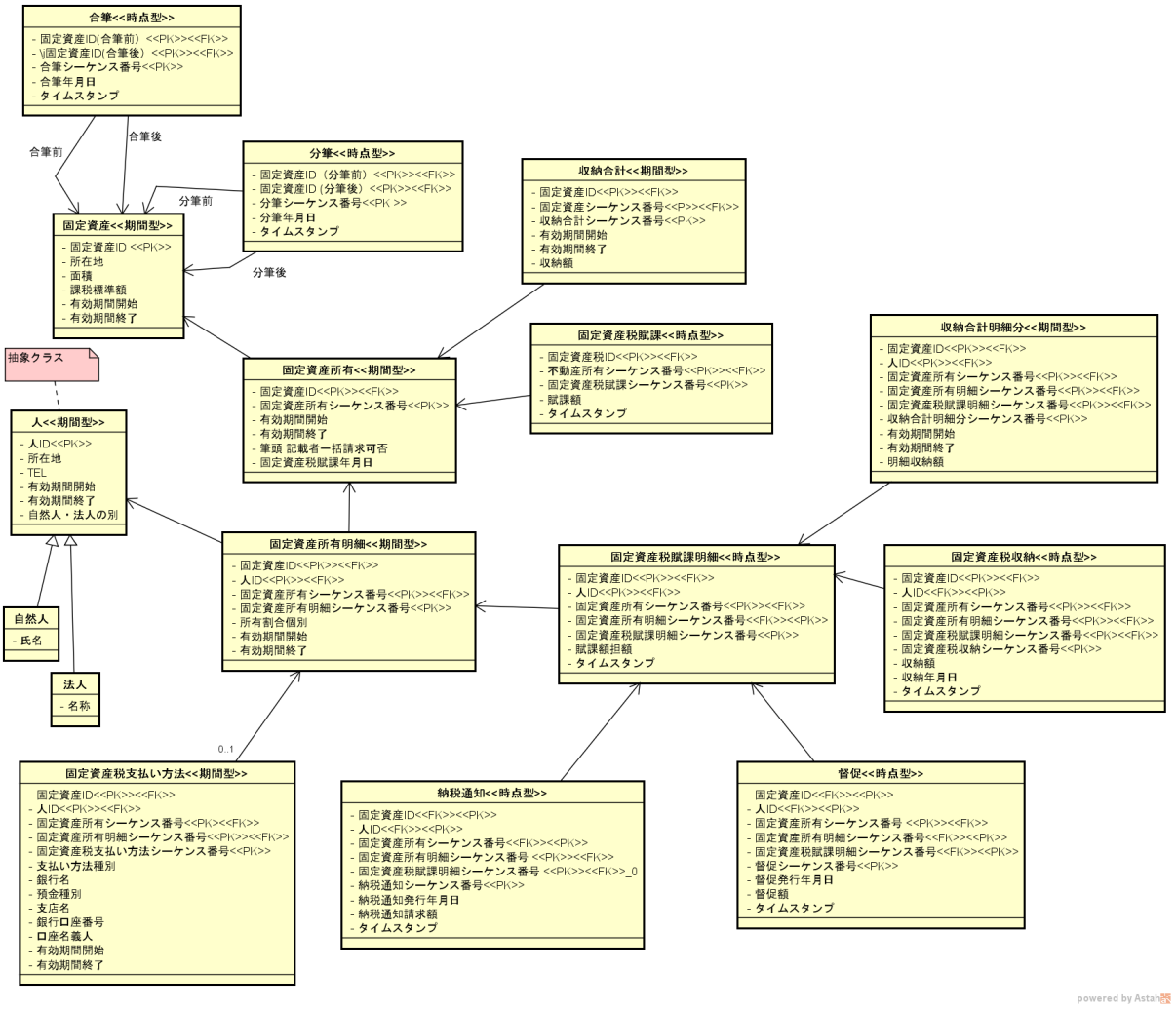

図 8 固定資産税に対する部分的な存在従属クラス図

図 8 の例でいえば，固定資産税の賦課の直接的原因は，「不動産の所有」にあるとし，「不動産所有」の原因として，所有者である人（法人，自然人を問わない）と不動産自体があるものとしている．このため，クラス「固定資産税賦課」には，納税義務者の名前や連絡先住所の情報はない．納税通知書の印刷に際しては，関数従属関係をたどり，所有者の氏名，住所等を取り出す必要がある．

上記図 8 では，もし，所有者の住所が変わると，自然人のクラスのインスタンスに対して，図 7 の様にして履歴を作る．そうすると，その後に納税通知が作成される際にでも，最新の当該自然人（所有者）の情報が活用できる．図 8 で，所有関係が変わっても，つまり，不動産が課税後に所有権移転の対象となっても問題はない．固定資産所有クラス，固定資産所有明細クラスは，無効化タイムスタンプにより，論理的には存在しなくなる．しかし，もともと，納税通知の PK には，人 ID が記載されており，支払い義務者（正確には，共有している所有者のひとり）をすぐに探すことができる．そして，図 7 の様にして，最新のデータが管理されているので，古い住所で，納税通知を送る恐れもない．

ただし，図 8 には，支払い方法に関するクラスがあるが，これは，存在従属関連のリンクでは辿ることができない．しかし，支払い方法のクラスは，ここでは，固定資産の所有関係クラスの下流クラスとなっており，しかも，下流のバンクアカウント情報側の多重度が「0..1」あるいは「1」である．従って，固定資産の所有関係クラスから一意にリンクを辿って行ける．言い換えると，例えば，納税通知書に，引き落としの口座番号を記載することができる．

5. 残された課題

以上，提案方法について述べた．提案手法は，幾つかの課題を持っている．第一に，上記のメカニズムがうまく動くことの確認である．この複雑な処理を，スクラッチからプログラミングする形で，アプリケーションごとに繰り返すのは得策ではない．第一にプラットフォームの設計・実装を必要とする．バージョン管理を含めて，プラットフォーム化が必要である．

第二に，存在従属関連で一意に指定できるインスタンス

が持つ情報のみで，本当に業務ができるのか？との問いがある．この点は，実務レベルで実際にクラス図の設計を行い，確認する以外，手がなさそうである．ただし，「このインスタンスが決まると，どのインスタンス（複数）がユニークに指定可能なのか？」は，対象ドメインの問題であり，対象ドメインがうまくクラス図化されていれば，実務的にも，存在従属関連でアクセスできる範囲に，必要なデータ項目はすべて含まれていることを期待している．RDB の例でいえば，すべてのテーブルが第 3 正規形になっており，隠れた関数従属性が無い状態である．

6. 終わりに

存在従属クラス図の適用により，モデル中のクラスの正則性（クラスが第 3 正規形であること，及び，インスタンスのライフタイムと属性値のライフタイムが一致していること）を担保できる手法を提案した．ただし，インスタンスは一度生成されると消去されることはなく，タイムスタンプ付でバージョン管理されており，あるインスタンスに関連した項目のデータ値は，そのインスタンスの（存在従属クラス図でいう）上流側のインスタンスから，いつでも取り出すことができる．

本提案手法により，アプリケーションの構造は，極めてシンプルとなり，クラスの正規性を担保しているので，保守も容易化するものと期待している．しかし，反面，提案手法を実現するためには，共通機能をプラットフォーム化する必要がある．その実装と，存在従属関連により下流側から上流側にたどることのできる範囲で，本当にアプリケーションが描けるのかという点も実用規模での検証が望まれる．

本提案手法は，オブジェクトの履歴をバージョンとして管理する．このために，メモリ消費が大幅に増えるものと想像される．また，ビューに相当するデータ項目を，必要なタイミングで，取りに行っている．この取りに行くプロセスの負荷も大きくなる恐れがある．しかし，CPU の性能向上は著しく，DRAM の大容量化は，2G バイト/chip のレベルに達している [7]．また，ReRAM[8] の様に，主記憶を不揮発化できるメモリの進展も著しい．近い将来，アプリケーションの全オブジェクトをメインメモリに展開することは不自然とは言えないのではないだろうか．

参考文献

[1] 大木幹雄，「ライムタイム分析に基づくクラス構造抽出の定式化と構造に関するデザインパターンの抽出実験」，情報処理学会論文誌，Vol.45, No.6, pp.1554-1568，2004 年 6 月
[2] 金田重郎，井田明男，酒井孝真，熊谷聡志，「日本語仕様文からの概念モデリングガイドライン－行為文と関数従属性に基づくクラス図の作成－」，電子情報通信学会論文誌 D，Vol.J98-D，No.7，pp.1068-1082，2015 年 7 月
[3] 井田明男，金田重郎，熊谷聡志，藤本明莉，「存在従属性に着目した論理要件ロバストなドメインモデルの作成- ドメインクラス図をユビキタス言語として用いるために -」，情報処理学会論文誌，Vol.56, No.5, pp.1340-1350，2015 年 5 月
[4] 椿正明，「名人椿正明が教えるデータモデリングの技」翔泳社，2005 年 11 月
[5] 渡辺幸三，「データモデリング入門」日本実業出版社，2001 年 7 月
[6] 佐藤正美，「データベース設計論―T 字型 ER」ソフト・リサーチ・センター，2005 年 9 月
[7] マイクロン社 Web サイト
https://www.micron.com/products/dram/ddr3-sdram
[8] 富士通セミコンダクタ，Web サイト，ReRAM 量産製品として世界最大容量の 4M ビット品の提供を開始
http://www.fujitsu.com/jp/group/fsl/resources/news/press-releases/2016/1026.html