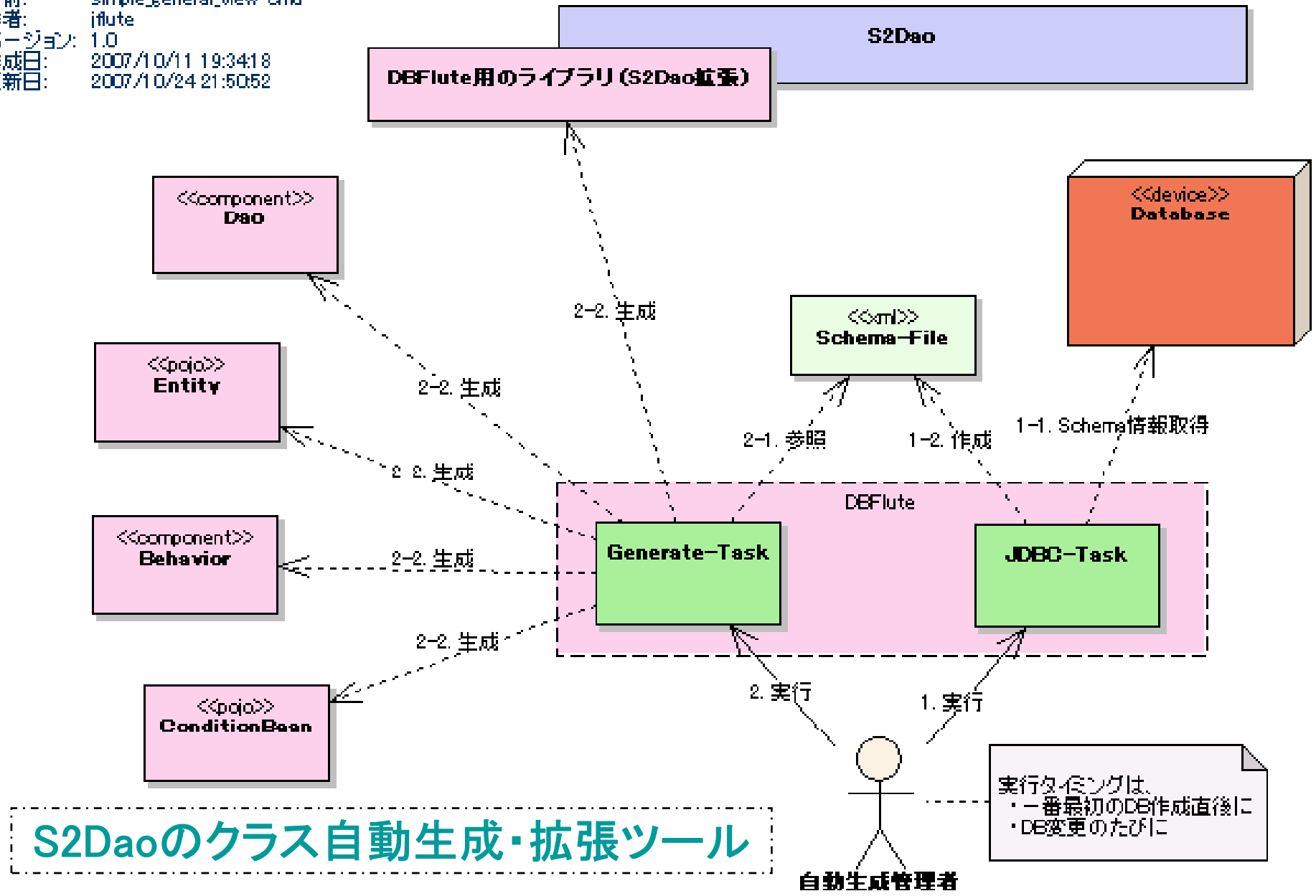
現場ソリューション DBFlute

株式会社ビルドシステム
久保　雅彦

DBFluteとは？

S2Daoのクラス自動生成＆拡張ツール

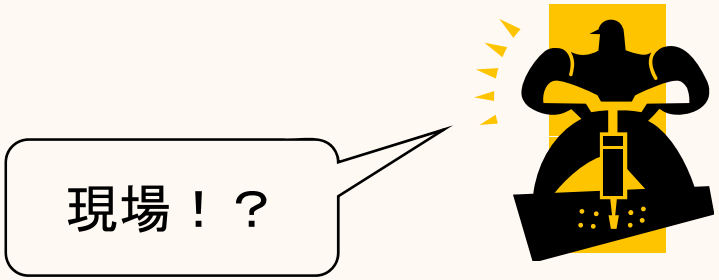
名前: simple_general_view-cmu
作者: jflute
バージョン: 1.0
作成日: 2007/10/11 19:34:18
更新日: 2007/10/24 21:50:52



S2Daoのクラス自動生成・拡張ツール

今回のテーマは？

現場ソリューション
(Genba Solution)

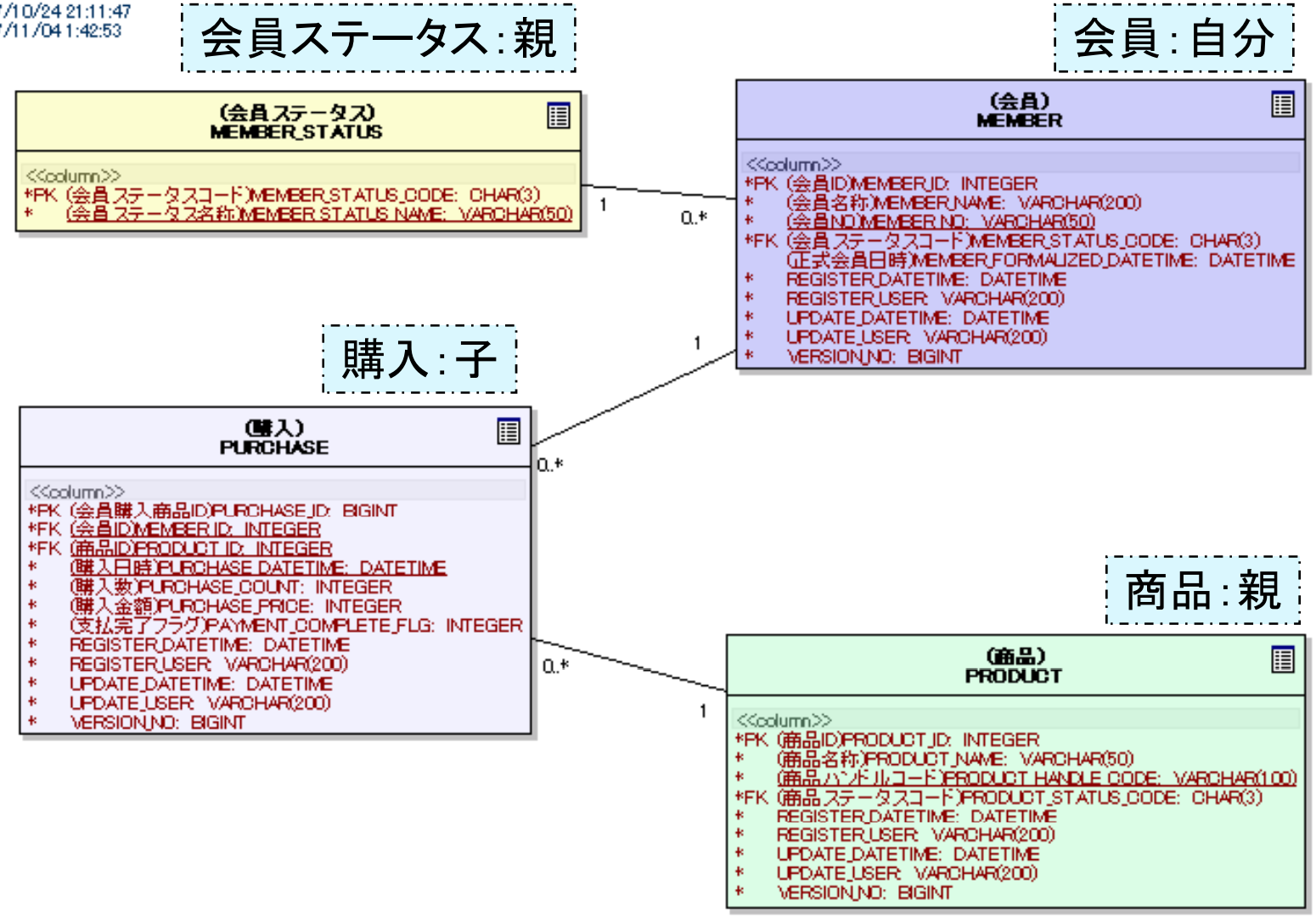


今回の現場ソリューション

1. DB変更でいつもデグレってしまう
2. 区分値の条件付与とか判定処理とかのProject内の統一に悩む
3. 登録日時とか更新者とかの値の設定はどこかで共通的にやりたい
4. 検索一覧画面のページナビゲーションの計算処理がいつも面倒
5. 日付範囲検索で終了日のデータがいつもHITしない
6. Where句条件の再利用ってできないものだろうか
7. やっぱりone-to-many(-to-mamy)でEntityが欲しい
8. 削除されたTableがSchemaに残り続けて時々いやな思いをする
9. 大量件数を検索するとOutOfMemoryになる

例で利用する前提テーブル構造

名前: exampledb_genba_view-cls
作者: jflute
バージョン: 1.0
作成日: 2007/10/24 21:11:47
更新日: 2007/11/04 1:42:53



【悩み】

「要件変更によるDB変更」・「列名のスペル間違い」・「列名の命名規約違反」があったのでDB変更したいのですが、DB変更の影響範囲が網羅できないため、変更するとプログラマが暴動を起こしかねません。

[1]DB変更：全部単体テストは？

全部単体テストを書いたらどうですか？

いやぁ、納期も短く
、逐一全てのSQL
に
単体テストは書けて
ません…

DBFluteなら…

[1]DB変更：DBFlute的解決

- 9割のSQL (ConditionBean)
  - タイプセーフなのでコンパイルエラーでわかる
    - → DB設計者でも直せる場合も
- 1割のSQL (外だしSQL)
  - 1割なら量的に単体テストも書きやすい。
  - 外だしSQLの2Way-SQLを利用した一括実行テスト
    - → outside-sql-test.bat を実行

[1]DB変更：Torqueゆずり

Apache Torqueからヒントを得る

http://db.apache.org/torque/

torque

そもそもDBFluteはApache TorqueをS2Daoに適用したもの

[2]区分値:悩み

【悩み】

区分値の条件付与とか判定処理とか、Project内の統一にいつも悩みます。統一できなくて可読性は悪いし、区分値変更時に修正漏れたり....

例えば「区分値の条件設定」

```
// 正式会員のものを検索
cb.query().setMemberStatusCode_Equal(“FML”);
```

→ ハードコードはつらい。

→ せめて定数(static final)とかENUMとかにしたい。
(定義の規約とか命名方針とか考えたり…)

例えば「区分値の判定」

```
// 正式会員の場合
if (“FML”.equals(member.getMemberStatusCode())) {
```

→ ハードコードはつらい。

→ せめて定数(static final)とかENUMとかにしたい。
(定義の規約とか命名方針とか考えたり…)

例えば「区分値の設定」

```
// 正式会員を設定
memerb.setMemberStatusCode(“FML”);
```

→ ハードコードはつらい。

→ せめて定数(static final)とかENUMとかにしたい。
(定義の規約とか命名方針とか考えたり…)

DBFluteなら…

[2]区分値：DBFlute的解決-条件設定

ConditionBeanに区分値の条件付与メソッド

cb.query().setMemberStatusCode_Equal_Formalized();

(cb.query().setMemberStatusCode_Equal(“FML”);)

↓※SQL上は

where member.MEMBER_STATUS_CODE = ‘FML’

[2]区分値：DBFlute的解決-判定

Entityに区分値の判定メソッド

```
// 正式会員の場合
if (member.isMemberStatusCodeFormalized()) {
(if (“FML”.equals(member.getMemberStatusCode()) {)
    …
}
```

[2]区分値：DBFlute的解決-設定

Entityに区分値の設定メソッド

```
// 正式会員を設定
member.classifyMemberStatusCodeFormalized();
(member.setMemberStatusCode(“FML”);)
```

[2]区分値：設定方法-区分値

classificationDefinitionMap.dfprop

```
map:{
    ; Flg = list:{
        ; map:{topComment=フラグを示す}
        ; map:{code=1;name=True ;comment=有効を示す}
        ; map:{code=0;name=False;comment=無効を示す}
    }
    ; MemberStatus = list:{
        ; map:{topComment=会員の状態を示す}
        ; map:{code=FML;name=Formalized;comment=正式会員を示す}
        ; map:{code=WDL;name=Withdrawal;comment=退会会員を示す}
        ; map:{code=PRV;name=Provisional;comment=仮会員を示す}
    }
}
```

classificationDeploymentMap.dfprop

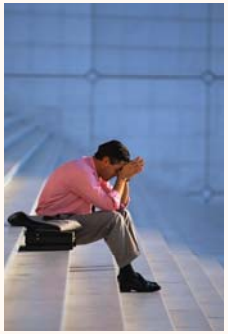
```
map:{
    ; $$ALL$$ = map:{suffix:_FLG=Flg}
    ; MEMBER = map:{MEMBER_STATUS_CODE=MemberStatus}
}
```

※ $$ALL$$は全てのテーブルを表す

【悩み】

登録日時とか更新者とかの全てのテーブルに定義される共通列へのInsert/Update時の値の設定を自動でやりたいのです。

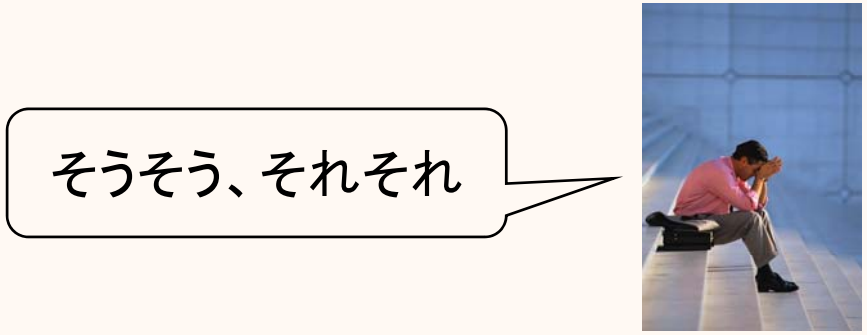
(結構みんな忘れてて実行時の例外で気付く)



共通列とは？

例えば

- **登録日時**:Insertした日時。その後更新されない。
- **登録ユーザ**:Insertしたログインユーザ。その後更新されない。
- **更新日時**:Updateした日時。
- **更新ユーザ**:Updateしたログインユーザ。



DBFluteなら…

BehaviorのFilterにて共通列の値を自動設定

```
Member member = new Member();
member.setMemberName(“Billy Joel”);

// 共通列(更新日時や更新者)は設定しなくてよい
memberBhv.update(member);
```

メソッド内部にて設定ファイルにて定義された値が自動設定される

build-xxx.propertiesにおいて共通列の設定

```
torque.commonColumnMap = map:{ ¥
    ; REGISTER_DATETIME=TIMESTAMP; REGISTER_USER=VARCHAR ¥
    ; UPDATE_DATETIME=TIMESTAMP   ; UPDATE_USER=VARCHAR ¥
}
torque.commonColumnSetupBeforeInsertInterceptorLogicMap = map:{ ¥
    ; REGISTER_DATETIME  = new java.sql.Timestamp(System.cur…) ¥
    ; REGISTER_USER  = “RegistUser” ¥
    ; UPDATE_DATETIME = entity.getRegisterDatetime() ¥
    ; UPDATE_USER = entity.getRegisterUser() ¥
}
torque.commonColumnSetupBeforeUpdateInterceptorLogicMap = map:{ ¥
    ; UPDATE_DATETIME = new java.sql.Timestamp(System.cur…) ¥
    ; UPDATE_USER = “UpdateUser” ¥
}
```

個々の設定を「dfpropファイル」で設定することも可能
ex) ./dfprop/commonColumnMap.dfprop

AccessContext(ThreadLocal)を利用

```
torque.commonColumnMap = map:{ ¥
    ; REGISTER_DATETIME=TIMESTAMP; REGISTER_USER=VARCHAR ¥
    ; UPDATE_DATETIME=TIMESTAMP    ; UPDATE_USER=VARCHAR ¥
}
torque.commonColumnSetupBeforeInsertInterceptorLogicMap = map:{ ¥
    ; REGISTER_DATETIME = $$AccessContext$$.getAccessTimestampOn...() ¥
    ; REGISTER_USER = $$AccessContext$$.getAccessUserOnThread() ¥
    ; UPDATE_DATETIME    = entity.getRegisterDatetime() ¥
    ; UPDATE_USER    = entity.getRegisterUser() ¥
}
torque.commonColumnSetupBeforeUpdateInterceptorLogicMap = map:{ ¥
    ; UPDATE_DATETIME = $$AccessContext$$.getAccessTimestampOn...() ¥
    ; UPDATE_USER = $$AccessContext$$.getAccessUserOnThread() ¥
}
```

$$AccessContext$$はDBFluteが内部でxxx.allcommon.AccessContextに置き換える

AccessContext(ThreadLocal)に値を設定

ex) Pageクラス(画面イベントのクラス)でのInterceptorにて

```
public Object invoke(MethodInvocation invocation) {
    try {
        AccessContext context = new AccessContext();
        context.setAccessTimestamp([現在日時取得Interfaceから取得]);
        context.setAccessUser([セッションからログインユーザの取得]);
        AccessContext.setAccessContextOnThread(context);
        return invocation.proceed();
    } finally {
        AccessContext.clearAccessContextOnThread(); // 最後は必ずクリア
    }
}
```

ServletFilterでもOK

【悩み】

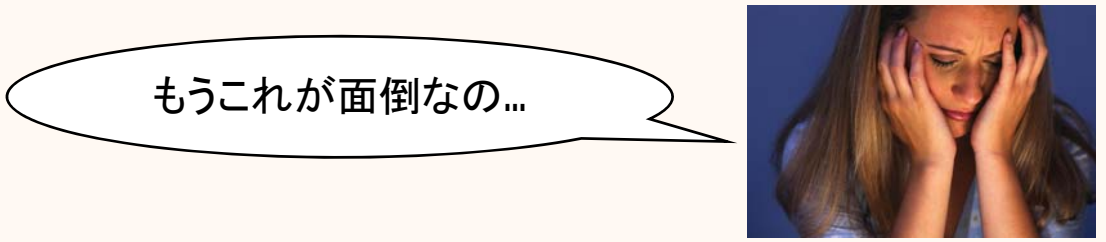
ページングナビゲーションの判定処理などのやり方が散在してしまっています。判定処理にバグも多くみられ、統一できないものかと悩んでいます。

【ページング処理の基本】

1. Paging条件無しの時の全件Record数の検索
   → select count(*) from xxx

2. Paging条件ありでの検索
   → select ... from xxx limit 20 offset 40

3. {1}と{2}の結果から総ページ数や次ページ存在判定などの計算



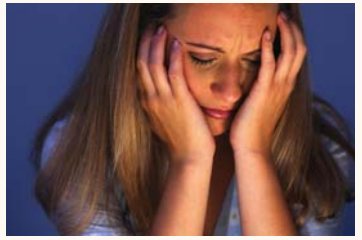
【ページングナビゲーションの基本(例)】

7 / 43 (863)

前へ 2 3 4 5 6 7 8 9 10 11 12 次へ

※前後5ページ分のリンクを表示
（DBFluteではこのパターンをPageRangeと呼ぶ）

もう全件表示すればいいじゃないの！



[4]ページング：DBFluteなら…

DBFluteなら…

[4]ページング：DBFlute的解決(検索)

ex) 会員の一覧をページング検索

```
MemberBhv memberBhv = [...BehaviorのInstanceを取得]
MemberCB cb = new MemberCB();
[...ConditionBean(条件)の設定]
cb.fetchFirst(20); // 1ページのSizeは20
cb.fetchPage(3); // 3ページ目を検索したい

// {1}-{2}-{3}の処理を実行
PagingResultBean<Member> memberPage
                                = memberBhv.selectPage(cb);
```

[4]ページング：DBFlute的解決 (ナビゲーション-1)

(続き)

```
// java.util.Listの実装クラスなのでそのまま扱うことが可能
for (Member member : memberPage)  …

// 総レコード数と現在ページ番号と総ページ数
int allRecordCount = memberPage.getAllRecordCount();
int currentPageNumber = memberPage.getCurrentPageNumber();
Int allPageCount = memberPage.getAllPageCount();
```

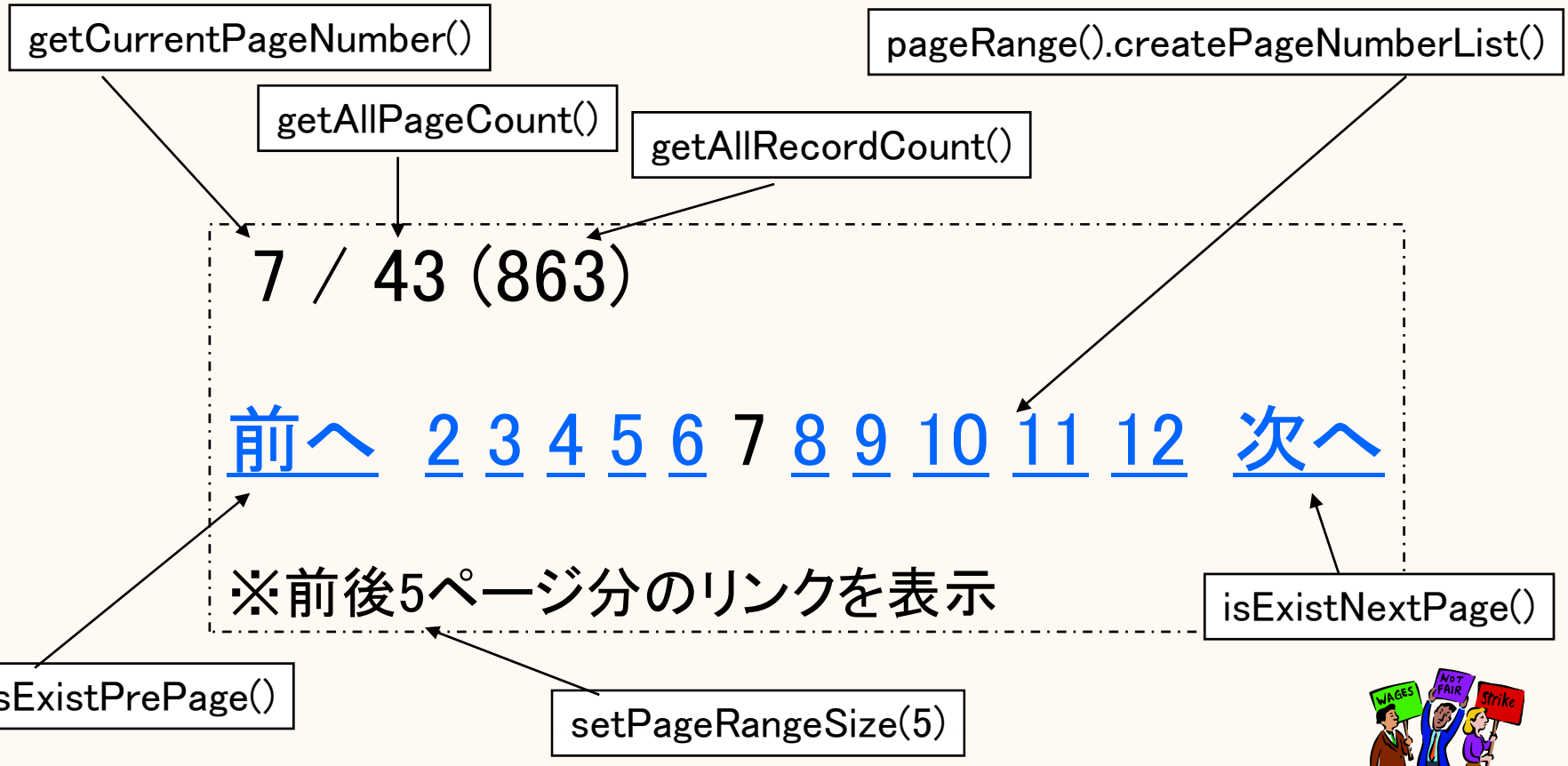
[4]ページング：DBFlute的解決 (ナビゲーション-2)

(続き)

```
// 前・次のページが存在するか
boolean isExistPrePage = memberPage.isExistPrePage();
boolean isExistNextPage = memberPage.isExistNextPage();

// ナビゲーションのページリンク候補一覧
memberPage.setPageRangeSize(5);
List<Integer> pageNumberList =
        memberPage.pageRange().createPageNumberList();
```

先ほどの例に当てはめると



他のパターンにも対応 (PageGroup)

1 2 3 4 5 6 7 8 9 10 次へ

前へ 11 12 13 14 15 16 17 18 19 20 次へ

※10ページを1つのグループとして表示
※この場合の「次へ」は「次のグループの先頭ページへ」

[4]ページング：他のパターンにも対応 (PageRange + 固定数表示)

他のパターンにも対応 (PageRange + 固定数表示)

前へ 2 3 4 5 6 7 8 9 10 11 12 次へ ※7ページ

1 2 3 4 5 6 7 8 9 10 11 次へ ※3ページ

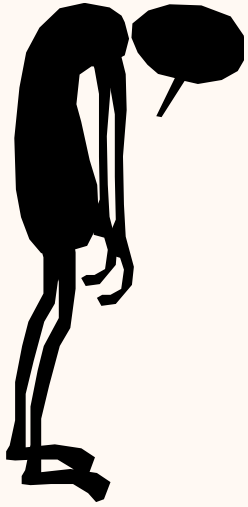
※前後5ページを表示 + 常に10ページ分を表示

[5]日付範囲検索:悩み

【悩み】

画面入力の日付範囲検索で、いつも終了日のデータがHITしないのです。

終了日に 11月20日 って入力したのに
11月20日 10:45:00 のデータがHITしないよぉ…



[5]日付範囲検索:原因

人間の認識とコンピュータの認識に
差があることによって発生する問題

画面

開始日: 2007/11/11

終了日: 2007/11/20

プログラム

2007/11/11 00:00:00

2007/11/20 00:00:00

2007/11/20 10:45:00 がHITしない！

そのまま実装すると…

```
where DATETIME >= '2007/11/11 00:00:00'
  and DATETIME <= '2007/11/20 00:00:00'
```

[5]日付範囲検索:2つの解決方法

1. 終了日の時分秒を埋める。

→ △ 処理が煩雑 (日付操作って意外に危険)

→ × DBによってミリ秒精度にクセがある

2. 終了日を一日進めて'<='を'<'にする。

→ △ 処理が煩雑 (日付操作って意外に危険)

→ ○ DBの仕様に依存しない

「2」が良いのだが、いずれにせよ処理が煩雑であることには変わりはない。
各プログラマがバラバラに実装すると最悪。

[5]日付範囲検索:DBFluteなら…

DBFluteなら…

[5]日付範囲検索:DBFlute的解決

ex) 検索画面にて開始日と終了日を入力した場合

```
Date fromDate = [画面入力値そのまま(2007/11/11)]
Date toDate = [画面入力値そのまま(2007/11/20)]
MemberCB cb = new MemberCB();
cb.query().setMemberFormalizedDatetime_DateFromTo(fromDate, toDate);
```

↓ DBFluteは「解決方法:2」を採用し、定番化している

```
where MEMBER_FORMALIZED_DATETIME >= '2007/11/11 00:00:00'
  and MEMBER_FORMALIZED_DATETIME < '2007/11/21 00:00:00'
```

toDateは内部的に一日進めて時分秒を切り取っている

[6]Where句再利用:悩み

【悩み】

Where句を再利用したいです…

[6]Where句再利用:DBFluteなら…

DBFluteなら…

[6]Where句再利用：DBFlute的解決

ConditionQueryのGapクラスに
業務的に定番な条件設定メソッドを作成

再利用メソッドの定義(例)

```
public class MemberCQ extends BsMemberCQ {
    /**
     * 正式会員で未払い購入がある会員
     */
    public void arrangeFormalizedUnpaidPurchase() {
        setMemberStatusCode_Equal_Formalized();

        // 会員の子テーブル「購入(Purchase)テーブル」の条件を設定
        PurchaseCB purchaseCB = new PurchaseCB();
        purchaseCB.query().setPaymentCompleteFlgFalse();
        setMemberId_ExistsSubQuery(purchaseCB.query());
    }
}
```

再利用メソッドの呼び出し(例)

```
MemberCB cb = new MemberCB();
cb.query().setMemberName_Equal("Billy Joel")
cb.query().arrangeFormalizedUnpaidPurchase();
```

↓

```
  from MEMBER member
where member.MEMBER_NAME = 'Billy Joel'
    and member.MEMBER_STATUS_CODE = 'FML'
    and exists (select
purchase.MEMBER_PURCHASE_PRODUCT_ID
                  from PURCHASE purchase
                 where purchase.MEMBER_ID = member.MEMBER_ID
                   and purchase.PAYMENT_COMPLETE_FLG = 0)
```

[6]Where句再利用：外だしSQLでは

2WAY-SQLを優先すると無理

(1割のSQLは許容)

↓

IFコメントで極力SQL自体を再利用

【悩み】

1:n…

DBFluteなら…

[7]one-to-many：DBFlute的解決

ex) 会員(Member)に紐付く購入(Purchase)を取得
(但し、購入数が2つ以上の購入日時降順)

```
List<Member> memberList = memberBhv.selectList(cb);
ConditionBeanSetupper<PurchaseCB> setupper
                            = new ConditionBeanSetupper<PurchaseCB>() {
    public void setup(PurchaseCB cb) {
        cb.query().setPurchaseCount_GreaterEqual(2);
        cb.query().addOrderBy_PurchaseDate_Desc();
    }
};
memberBhv.loadPurchaseList(memberList, setupper);
for (Member member : memberList) {
    // 検索された(Loadされた)購入リストを取得
    List<Purchase> purchaseList = member.getPurchaseList();
}
```

匿名内部クラスとして生成。
購入(PURCHASE)側の絞り込み条件を指定している。

[7]one-to-many：実現方法

一括Loadで実現している

1. 指定された会員一覧に関連する購入を全て検索(SQL一発)

   → select … from PURCHASE where member_id in (?, ?...) and …

2.指定された会員一覧に取得した購入一覧を関連付ける

↓

HibernateのSubSelectフェッチと同じやり方。

N+1問題は発生しない

[8]ReplaceSchema ：悩み

【悩み】

新しいDDL文をツールで自動生成するのですが、削除されたTableの分のDROP文は無いので、FK違反とかでそのまま素直に実行できなかったりするのです。

A.「Schemaの作り直しは面倒だ…」
B.「ERWinは高いしな…」

[8]ReplaceSchema：DBFluteなら…

DBFluteなら…

[8]ReplaceSchema：DBFlute的解決

ReplaceSchemaを使う

以下を行うDBFluteのタスク

1. DB情報からFKとTableを全てDROP
2. 指定されたDDL文を実行(Drop文は不要)

[8]ReplaceSchema：利用方法

【利用方法】

1.ツールで自動生成したDDL文を以下に保存

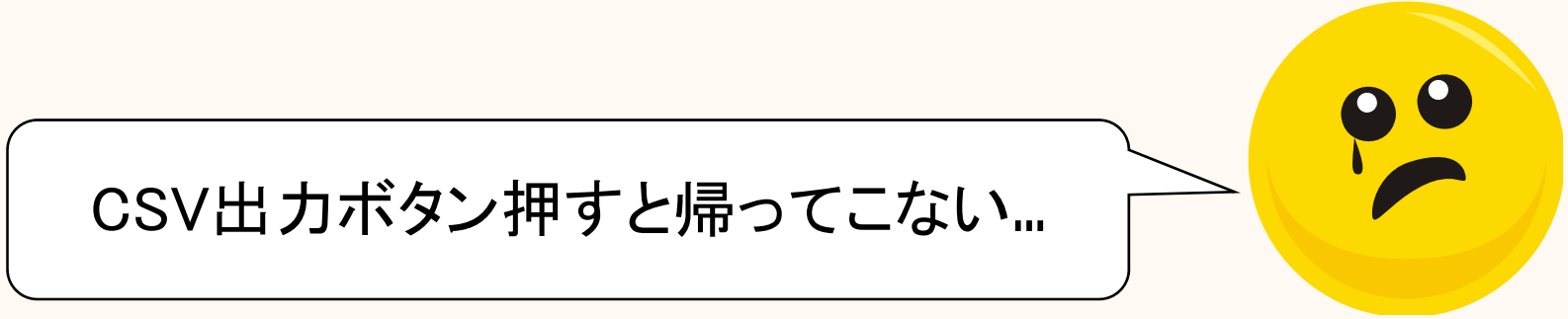
./playsql/replace-schema.sql

2. replace-schema.batを実行

o DDL文にDROP文は不要
o DB接続情報は「./dfprop/databaseInfoMap.dfprop」

【悩み】

大抵のO/Rマッパは検索後に対象レコードを全てメモリに展開するので(それはそれで便利なのですが)、どうしても大量件数を扱うときにOutOfMemoryになって困るのです。



[9]大量件数：JDBC直接使ったら？

JDBC直接使ったらどうですか？

なんのためのO/Rマッパだい！
OutOfMemoryになるのに気付くのは本番想定の大量データとか入れたときで、やり方を大きく変えるのがつらい時期だったりすることが多い。
外だしSQLそのままでResultSetを直接扱えればいいのに！

もちろん該当処理が大量データなのかどうかは
設計時点で明確にすべきですが…

[9]大量件数：DBFluteなら…

DBFluteなら…

[9]大量件数：DBFlute的解決

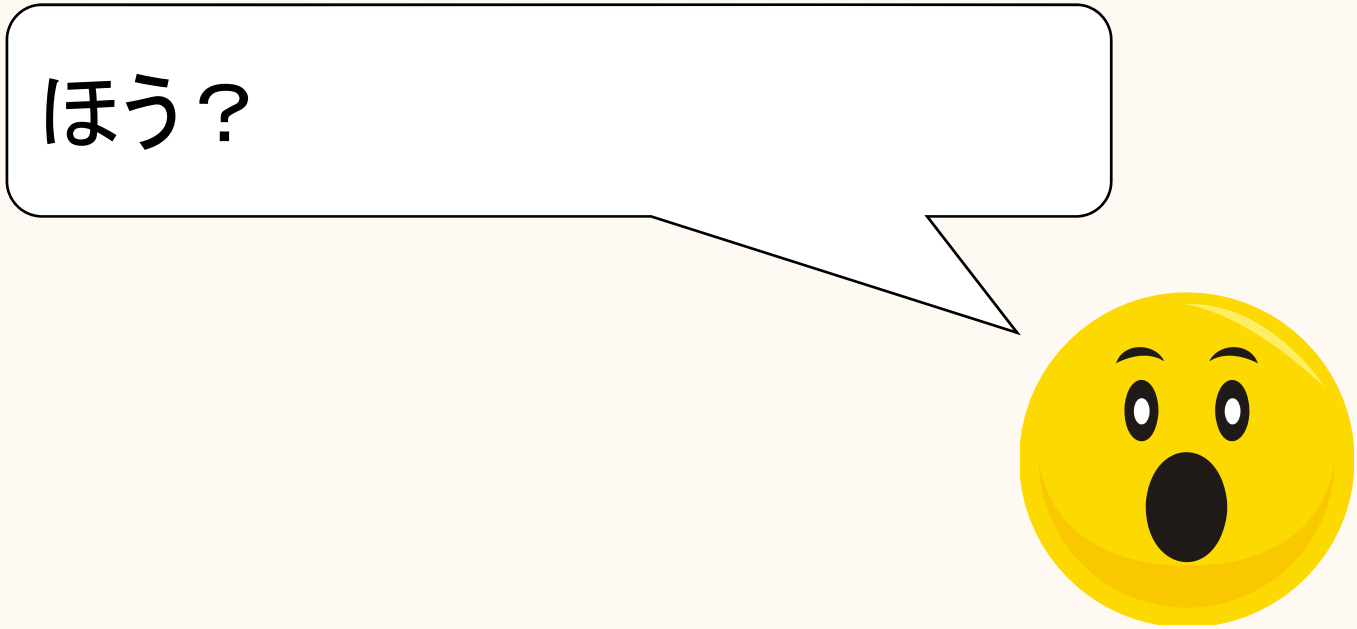
ex) 外だしSQLの結果からCSV作成(BehaviorのGapクラスに実装)

```
public void makeCsvSummaryMember(SummaryMemberPmb pmb) {
    CursorHandler handler = new CursorHandler() {
        public Object handle(ResultSet rs) throws SQLException {
            while (rs.next()) {
                rs.getString("MEMBER_NAME");
                ...[CSV出力処理]
            } // ResultSetのCloseはFrameworkが行うので必要なし
            return null; // ここで処理が完結してるので戻り値は不要
        }
    };
    // 外だしSQL実行 {pmbはBindパラメータのための引数DTO}
    // 最後の引数にCursorHandlerを指定
    outsideSql().cursorHandling().selectCursor("xxx.sql", pmb, handler);
}
```

[9]大量件数：ResultSet直接触るの？

ResultSet直接扱うのかぁ…

いえ、そんなことはありません



[9]大量件数：タイプセーフCursor作成

ex) 外だしSQL(xxx.sql)にてタイプセーフCursorの設定
(Sql2Entityにて自動生成)

```
-- #SummaryMember#
-- +cursor+

select member.MEMBER_ID, member.MEMBER_NAME
     , (select sum(purchase.PURCHASE_COUNT)
          from PURCHASE purchase
         where purchase.MEMBER_ID = member.MEMBER_ID
       ) as PURCHASE_SUMMARY
  from MEMBER member
 where member...
```

[9]大量件数：タイプセーフCursor利用

ex) 外だしSQLの結果からCSV作成 (BehaviorのGapクラスに実装)

```
public void makeCsvSummaryMember(SummaryMemberPmb pmb) {
    SummaryMemberCursorHandler handler
                                        = new SummaryMemberCursorHandler() {
        public Object fetchCursor(SummaryMemberCursor cursor)
                                                    throws SQLException {
            while (cursor.next()) {
                Integer memberId = cursor.getMemberId();
                String memberName = cursor.getMemberName();
                Integer purchaseSummary = cursor.getPurchaseSummary();
                ...[CSV出力処理]
            } // ResultSetのCloseはFrameworkが行うので必要なし
            return null; // ここで処理が完結してるので戻り値は不要
        }
    };
    ...
}
```

現場ドリブンでの機能整備を
重要視している。

↓

現場指向ポリシー

現場指向ポリシーについて

- 現場指向ポリシーとは？
- 実装バランスとは？
- 実装バランス
- 実装バランス(補足１)
- 実装バランス(補足２)
- 結合の取捨選択
- 結合の取捨選択(補足１:他のやり方との比較)
- 結合の取捨選択(補足２:アーキテクチャとの絡み)
- 結合の取捨選択(補足３:アーキテクチャとの絡み)
- 結合の取捨選択(補足４:アーキテクチャとの絡み)

現場指向ポリシーとは？

現場にフィットすることを最重要視した
O/Rマッパとしてのポリシー

※今回は、以下の2つに着目

「DBアクセス実装のバランス」

「結合の取捨選択」

「理想」とか「カッコいい」とかよりも、
現場の開発者/メンテナンス者が「助かるだろう」
を一番に考えるポリシー

現場指向ポリシー：実装バランスとは？

- Hibernate
  - Criteriaをどれだけ使う？
  - HQLをどれだけ使う？
  - NativeSQLをどれだけ使う？
  - どれか1つしか使わない？
- S2Dao
  - DTOによるSQL自動生成をどれだけ使う？
  - QUERYアノテーションをどれだけ使う？
  - SQLファイルをどれだけ使う？
  - どれか1つしか使わない？
- Torque
  - Criteriaをどれだけ使う？
  - というかCriteriaしかない？

O/Rマッパの提供する機能を現場で
どういった役割・配分で利用するかのバランス

現場指向ポリシー：実装バランス

【DBアクセス実装のバランス】

- 簡単なSQL + 定型的なSQL（約9割）
  - ConditionBeanにてタイプセーフ実装
- 複雑で独自的なSQL（約1割）
  - 外だしSQLにて実装

どちらか一方ではなく、このバランスが重要

現場指向ポリシー: 実装バランス (補足1)

どちらか一方に寄せるべきでない

- 全てをConditionBeanにて実装
  - → ×技術的に無理(クリティカルな理由)
  - → ×複雑で独自的なSQLを無理やり実現すると可読性が悪い
  - → ×複雑で独自的なSQLはチューニングなど厳密にしたい
- 全てを外だしSQLにて実装
  - → ×DB変更が恐すぎる
  - → ×SQLのしょうもないミスが多発
  - → ×SQLプログラマの確保は難しい(リリース後も含めて)

現場指向ポリシー: 実装バランス (補足2)

もちろん選択肢が2つになる弊害はある

ConditionBeanと外だしSQLの判断に迷うことがある

- パターン解説のドキュメントの充実
- DBFlute-Exampleの充実
- 経験者の増加を促進 (→ 利用者を増やす努力をする)
- Seasar-user MLでのサポート

今後の課題！

現場指向ポリシー:結合の取捨選択

【必要なテーブルを選択】

– どのテーブルを結合してインスタンス化したいかを選択
(全部取得でもなく、LazyLoadでもない)

ex) 会員ステータス(MemberStatus)を結合して会員(Member)を検索

```
MemberCB cb = new MemberCB();
cb.setupSelect_MemberStatus(); // 「MemberStatus」を選択(結合)
cb.query().setMemberId_Equal(3);
Member member = memberBhv.selectEntityWithDeletedCheck(cb);
    ↓
select member.MEMBER_ID..., memberStatus.MEMBER_STATUS_NAME, ...
  from MEMBER member
    left outer join MEMBER_STATUS memberStatus
      on ...
```

現場指向ポリシー：結合の取捨選択 (補足１：他のやり方との比較)

- 全てを取得 (S2DaoのSQL自動生成)
  - → ×余計なテーブルの取得の分遅い
- LazyLoad (Hibernateの一部機能)
  - → ×知らないうちにとんでもない数のSQLが発行される
  - → △Session in Viewが発生する

※LazyLoadはオンメモリDBにならない限り、
現場利用はつらいことが多いのではないかと。

現状では「何が欲しい？」を明示的に指定するのがGood

現場指向ポリシー：結合の取捨選択 (補足２：アーキテクチャとの絡み)

但し、DBアクセス隠蔽アーキテクチャと相性悪い

（DBアクセスには条件組み立てやSQLそのものを含む）

そもそもDBFluteはDBアクセスを隠蔽するという考えはない。

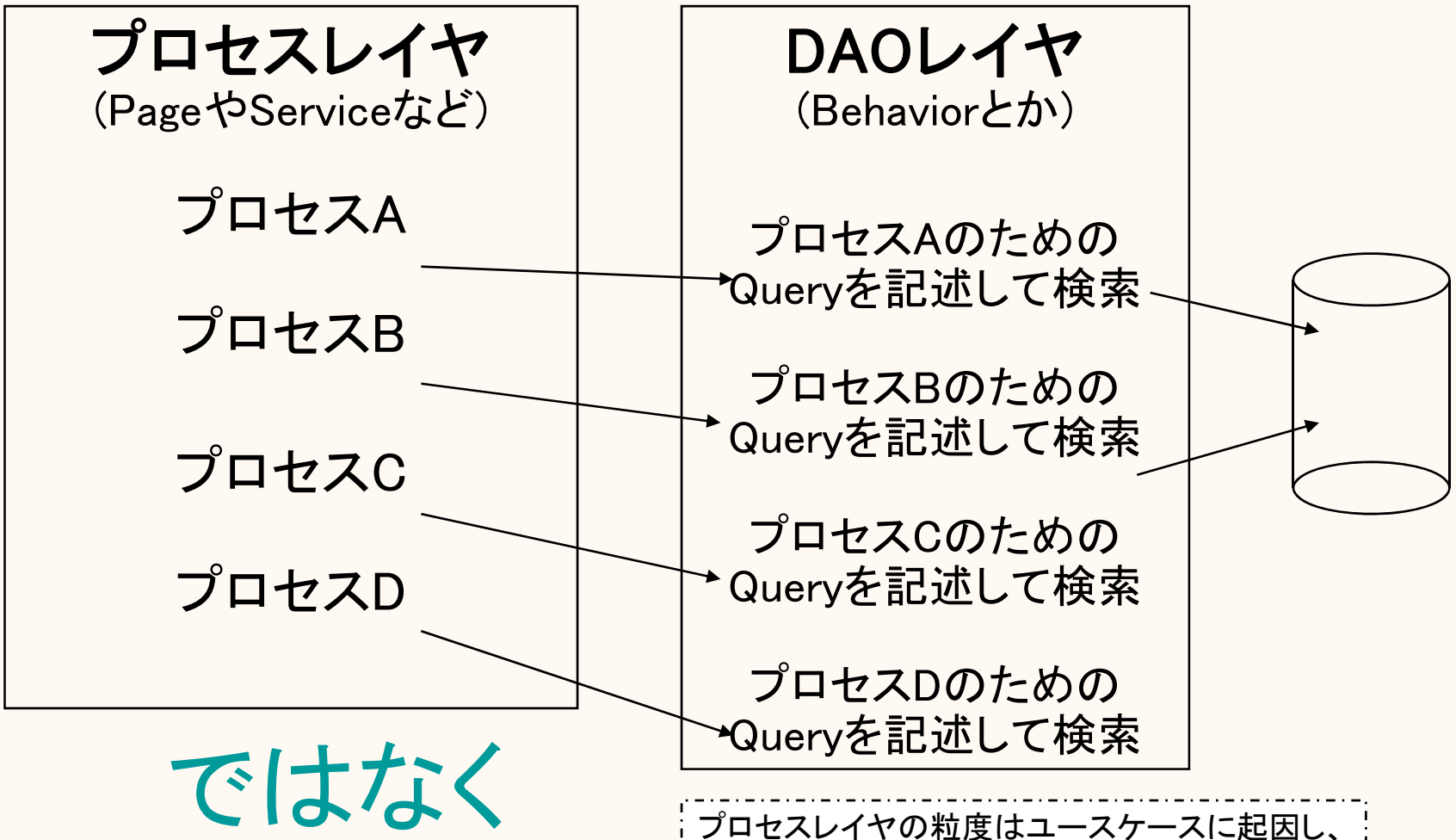
条件組み立てをDaoレイヤに全て記述すると：

- プロセス依存のメソッドとなり、メソッド名の命名に困る。
- プロセス依存のメソッドとなり、再利用できない。

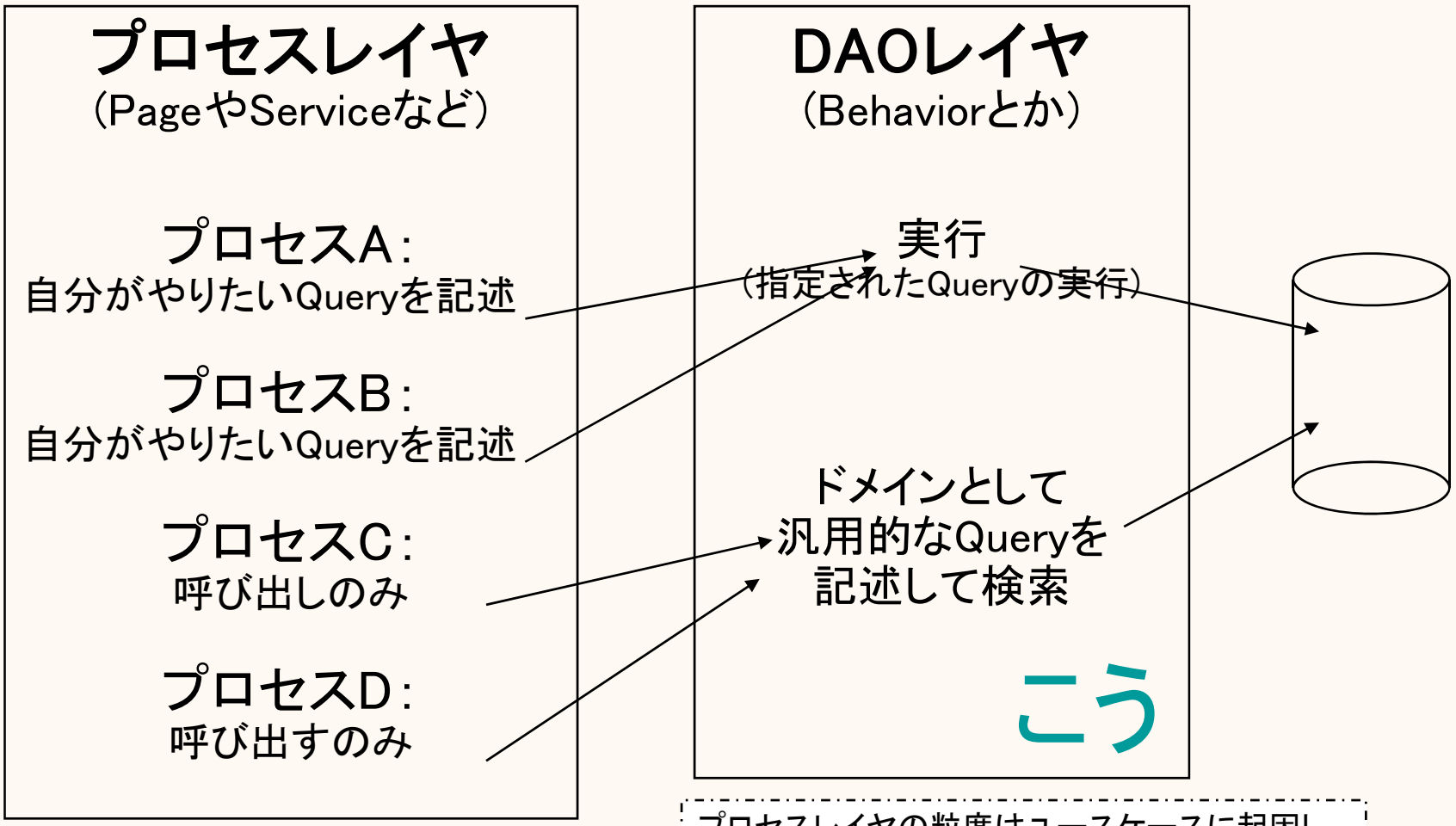
→「何が欲しい？」はプロセス依存であるため

但し、Daoレイヤに全てのQueryが集められて、管理しやすいというメリットは残る。

現場指向ポリシー:結合の取捨選択 (補足3:アーキテクチャとの絡み)



現場指向ポリシー:結合の取捨選択 (補足4:アーキテクチャとの絡み)



プロセスレイヤの粒度はユースケースに起因し、
DAOレイヤの粒度はテーブルに起因する。

最後に…:DBFluteの適用プロジェクト (実装1～2人程度のプロジェクト)

実装1～2人でやるようなプロジェクト
テーブル数が極端に少ないプロジェクト

↓

どのO/Rマッパでも大きな失敗はないため
特別DBFluteをお奨めすることはない。

もちろん、使えば便利であることには変わりはない。

最後に…:DBFluteの適用プロジェクト (実装3～9人程度のプロジェクト)

実装3～9人でやるようなプロジェクト
テーブル数もそれなりプロジェクト

↓

全員が(技術的に)ハイスキラであることは稀なため、
今回の現場ソリューションのような機能が大きく効果を発揮。
プロジェクト成功のためにDBFluteをお奨め。

DBFluteの実装環境を整備する人とそれを使って業務ロジックを実装する人は分業。
業務仕様に長けた人を技術的なところで時間を使わせたくない。

最後に…:DBFluteの適用プロジェクト (実装10人以上程度のプロジェクト)

実装10人以上でやるようなプロジェクト
テーブル数もかなりあるプロジェクト

場合によっては200を超えるテーブルとか

↓

今回の現場ソリューションのような機能が必須であり、
独自で対応もするだろうがその時間は節約したい。
プロジェクト成功のためにDBFluteをお奨め。

DBFluteの実装環境を整備する人とそれを使って業務ロジックを実装する人は分業。
業務仕様に長けた人を技術的なところで時間を使わせたくない。

最後に…:リリース後のメンテも考慮して

リリース後も悩みは発生

開発とメンテする人が違う場合も多々ある。
リリース後は開発時のような大人数はありえない。

↓

プログラムはできるだけ安全な方法で
実装(統一)されていて欲しい。
プロジェクト内・プロジェクト間でも

最後に…:今後の展望

EclipseのPluginによるサポート (Emechaの充実)

ドキュメントの充実 (DBFluteサイトの再構築)

全体の品質の向上 (都度サポート・メンテナンス)

↓

SandBox卒業！

外部情報

【DBFluteトップページ】
http://dbflute.sandbox.seasar.org/ja/index.html

【メイン開発者&出羽さんのブログ】
http://d.hatena.ne.jp/jflute
http://d.hatena.ne.jp/dewa

【今回のデモで利用したExample(SVNリポジトリ)】
https://www.seasar.org/svn/sandbox/dbflute/trunk/dbflute-basic-example
https://www.seasar.org/svn/sandbox/dbflute/trunk/dbflute-teeda-example

終わり

ご清聴ありがとうございました。

以後、補足資料

今回のセッションの前提バージョン

- DBFlute
  - バージョン：0.5.7
  - ステータス：SandBox
- S2Dao
  - バージョン：1.0.47
  - ステータス：Product

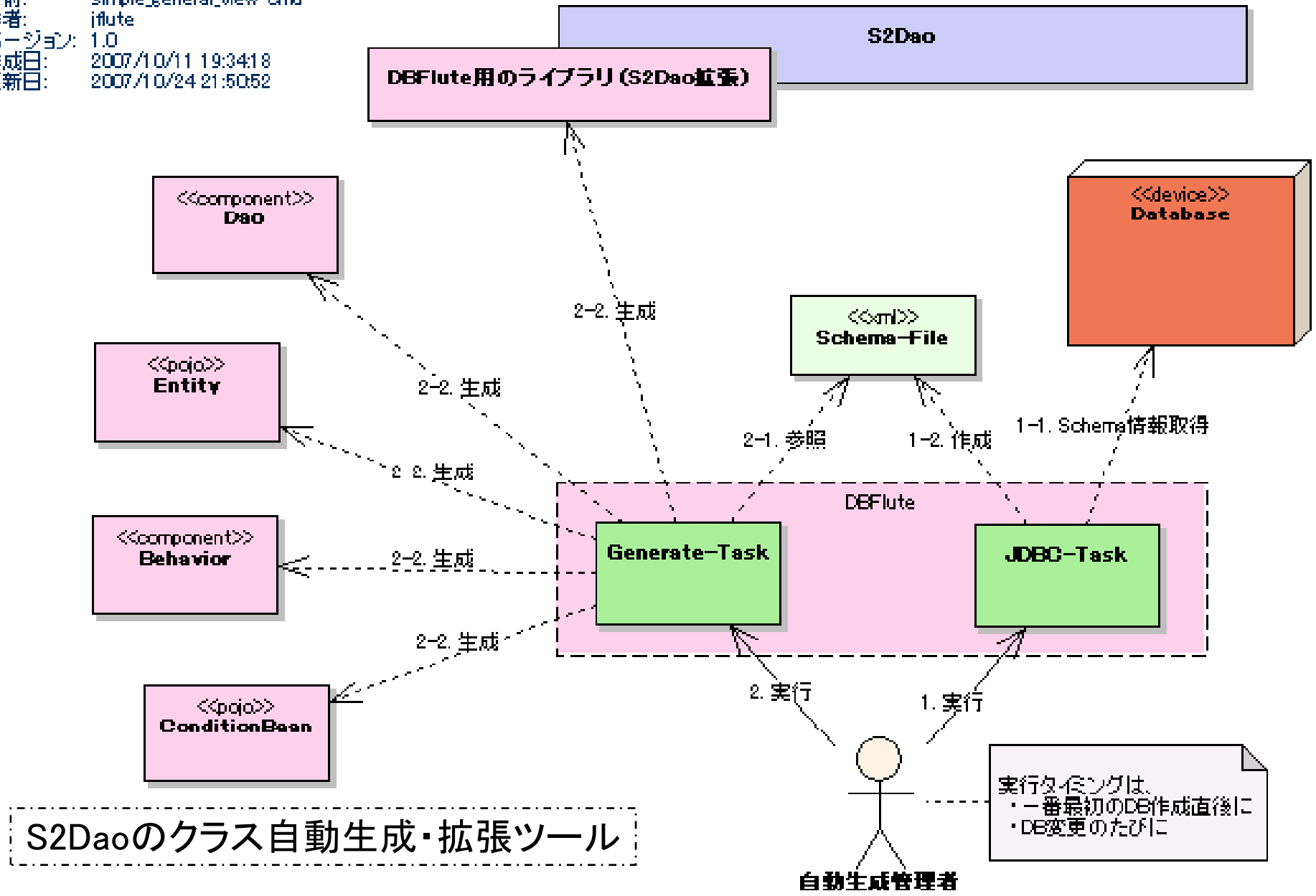
※執筆時点の最新

Agenda

–【DBFlute概要】

- DBFluteとは？
- DBFluteのタスク一覧
- 検索プログラム例
- 検索フロー
- 更新プログラム例
- 更新フロー
- 生成クラスカテゴリー覧
- Behaviorとは？
- Behaviorの参照系メソッド
- Behaviorの参照系メソッド（例外補足：一件検索の比較-結果0件）
- Behaviorの更新系メソッド（例外補足：一件検索の比較-結果2件以上）
- DBFlute概念図(詳細)

名前: simple_general_view-cmu
作者: jflute
バージョン: 1.0
作成日: 2007/10/11 19:34:18
更新日: 2007/10/24 21:50:52



S2Daoのクラス自動生成・拡張ツール

DBFluteを知る上での最低限のS2Daoの知識

- SQL文を外だしのファイル(SQLファイル)に記載して実行が可能
  (DBFluteでは「外だしSQL」と呼ぶ)

- 外だしSQLにて分岐を設定してアプリ呼び出し時に動的にSQLを変化させることが可能 (画面の検索条件の組み立てなどに有効)

- 2WAY-SQLである。
  (アプリケーションが使うSQLとSQLツールで実行するSQLを
  同じものとして管理することができる)

DBFlute概要：2WAY-SQLとは？

ex) 会員IDと会員名が指定されていたらその値で絞り込む。
　　会員ステータスの取得を指定されていたら、会員ステータスを結合して取得する。

```
select member.MEMBER_ID, member.MEMBER_NAME
        /*IF isIncludeStatus*/, memberStatus. MEMBER_STATUS_NAME/*END*/
  from MEMBER member
    /*IF isIncludeStatus*/
    left outer join MEMBER_STATUS memberStatus
      on member.MEMBER_STATUS_CODE=memberStatus.MEMBER_STATUS_CODE
    /*END*/
/*BEGIN*/
 where /*IF memberId != null*/member.MEMBER_ID = /*memberId*/3/*END*/
  /*IF memberName != null*/
     and member.MEMBER_NAME = /*memberName*/'Billy' || '%'
  /*END*/
/*END*/
```

このSQLは、アプリからの実行でもSQLツールでの実行でも動作する。
開発者はアプリを動かさなくてもSQLツールでSQLを簡単に確認できる。

アプリ実行時にこのandが必要ない場合は
自動的に除去される

アプリ実行時はメソッドの引数で指定された値を
バインド変数として扱い、テスト値は無視する。

DBFlute概要：DBFluteのタスク一覧

タスク	概要	アウトプット	DB接続有無
jdbc	JDBC経由でDBのスキーマ情報を取得する。	./schema/project-schema-xxx.xml	○
doc	スキーマ情報からテーブル一覧表を生成する。	./output/doc/project-schema-xxx.html	
generate	スキーマ情報からJava/C#のクラスを生成する。	クラスのソースファイル(Dao/Entityなど)	
sql2entity	外だしのSQL文から戻り値Entityを生成する。	クラスのソースファイル(Entity)	○
replace-schema	既存FKとテーブルを削除し、DDL文を実行する。	(DBにDDL実行結果が反映)	○
outside-sql-test	外だしのSQL文をテスト実行する。	(コンソールで文法エラーのSQLを確認)	○

DBFlute概要：検索プログラム例

ex)「Billy」で始まる名前の会員(Member)リストを検索

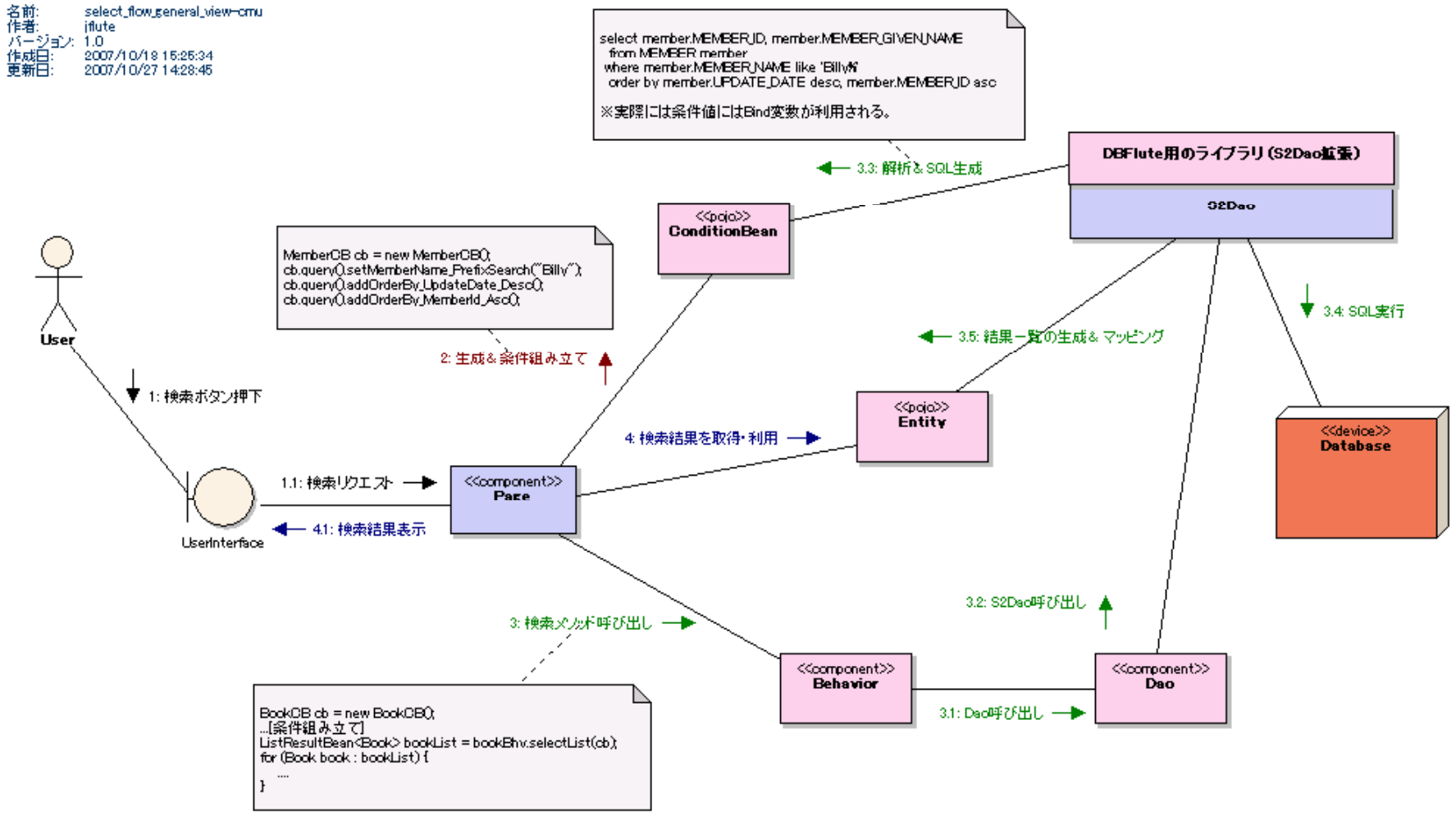
```
MemberCB cb = new MemberCB();
cb.query().setMemberName_PrefixSearch("Billy");
List<Member> memberList = memberBhv.selectList(cb);
```

↓

```
select MEMBER_ID, ...
  from MEMBER
where MEMBER_NAME like 'Billy%'
```

memberBhv はDIコンテナにインジェクションしてもらう

DBFlute概要：検索フロー



DBFlute概要：更新プログラム例

ex) 会員ID「3」の会員(Member)の名称を「Billy Joel」に更新

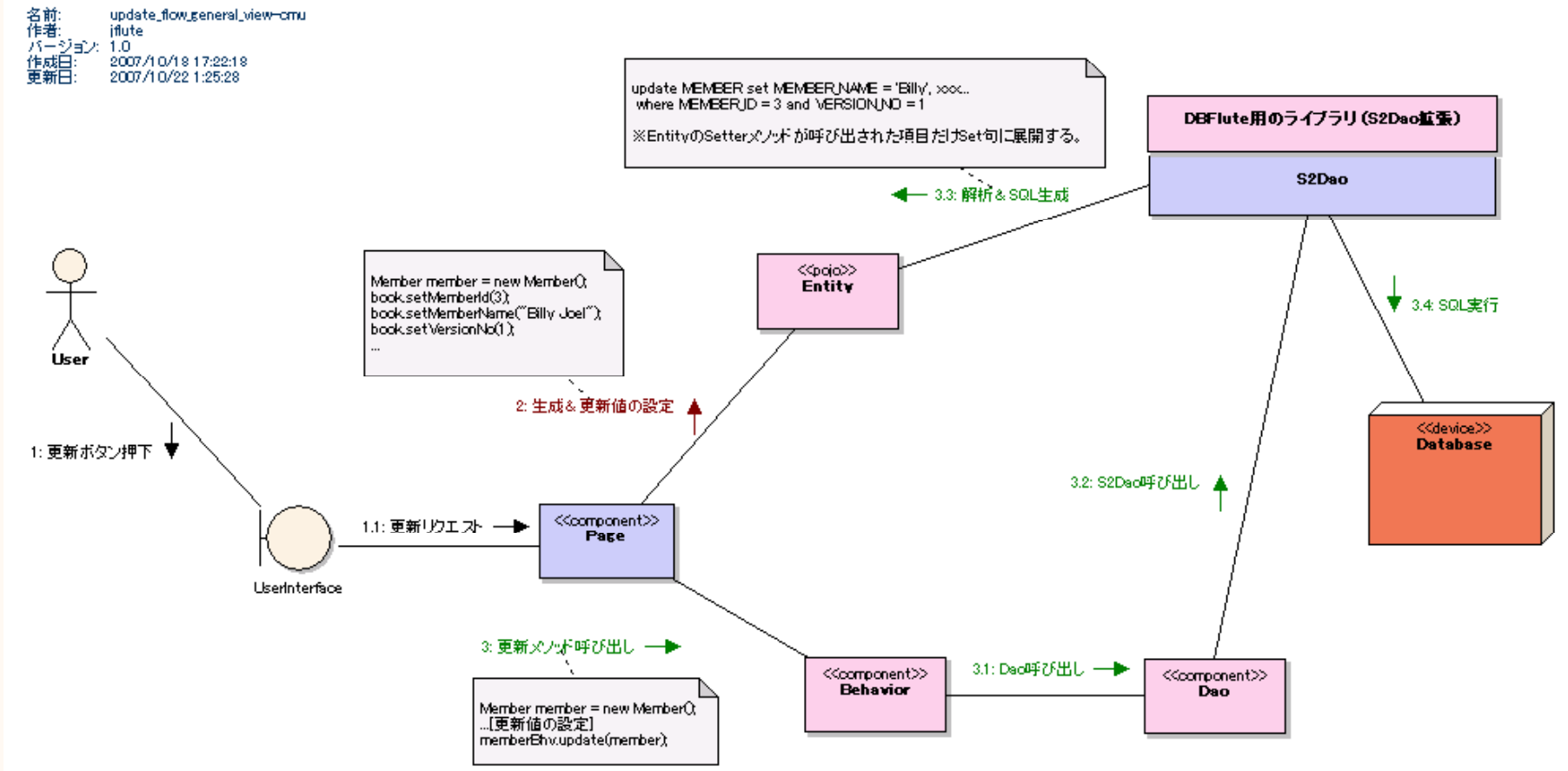
```
Member member = new Member();
member.setMemberId(3); // PrimaryKey
member.setMemberName("Billy Joel");
memberBhv.update(member);
```

↓

```
update MEMBER
    set MEMBER_NAME = 'Billy Joel'
 where MEMBER_ID = 3
```

memberBhv はDIコンテナにインジェクションしてもらう

DBFlute概要:更新フロー



DBFlute概要：生成クラスカテゴリー覧

クラスカテゴリ	概要	ジェネレーション ギャップ	利用頻度 ※1
Behavior	DaoとEntityの定番処理を行うObject。	○	○
DAO	S2DaoのDAOインターフェース。	○	△
Entity	テーブルに対応するドメインエンティティ。	○	○
ConditionBean	SQL組み立てObject。	○	○

※1 プログラマが直接利用することが多いものは○、全く利用しないものは×、その間は△

DaoとEntityの定番処理を行うObject

名前の由来は？ → **DAOの振舞い**
(The behavior of DAO)

その役割は？

JDBC → 低水準

DAO + Entity → 中位の水準

Behavior + ConditionBean → 高水準

厳密な意味がどうでもいい人は、
「DBFluteではBehaviorを使って検索したり更新したりする」
と覚えてもらうだけで構いません。

DBFlute概要：Behaviorの参照系メソッド

検索タイプ	メソッド	存在チェック ※1	重複チェック ※2
件数	int selectCount(MemberCB cb)	-	-
1件	Member selectEntity(MemberCB cb)	×	○
	Member selectEntityWithDeletedCheck(MemberCB cb)	○	○
	Member selectByPKValueWithDeletedCheck(Integer memberId)	○	○
n件	ListResultBean<Member> selectList(MemberCB cb)	-	-
ページング	PagingResultBean<Member> selectPage(MemberCB cb)	-	-

※1　結果が0件の場合は EntityAlreadyDeletedException が発生します。
※2　結果が2件以上の場合は EntityDuplicatedException が発生します。

一件検索の比較 (S2Daoとの比較)

検索結果が0件の場合：

→ 条件が間違っているかもしれない

→ データが間違っているかもしれない

→ 他のユーザが削除したかもしれない(すれ違い)

例外であって欲しい

selectEntityWithDeletedCheck() → 例外(EntityAlreadyDeletedException)

S2DaoのselectEntity() → null

もし、0件も想定する検索の場合は、BehaviorのselectEntity()で戻り値null判定をすればよい。
EntityAlreadyDeletedExceptionのtry-catchでもOK。

一件検索の比較 (S2Daoとの比較)

検索結果が2件以上の場合：

→ 条件が間違っているかもしれない

→ データが間違っているかもしれない

→ DBの制約が抜けているかもしれない

例外であって欲しい

selectEntity()/selectEntityWithDeletedCheck() → 例外(EntityDuplicatedException)

S2DaoのselectEntity() → 先頭の1件

もし、先頭の1件が欲しい場合は、ConditionBeanのfetchFirst(1)を利用すれば
limit/offset検索でSQL自体の結果が1件になるのでその方がよい。
→ ConditionBeanには、ROWNUMやTOPやLimit/OffsetなどDBの構文を利用して取得件数を絞り込む機能が備わっている。

DBFlute概要：Behaviorの更新系メソッド

更新件数	更新タイプ	メソッド		Modified Only ※3
		排他制御あり ※1	排他制御なし(Nonstrict) ※2	
1件	INSERT	void insert(Member member)		－
	UPDATE	void update(Member member)	void updateNonstrict(Member member)	○
	DELETE	void delete(Member member)	void deleteNonstrict(Member member)	－
			void deleteNonstrictIgnoreDeleted(Member member)	－
	INSERT or UPDATE	void insertOrUpdate(Member member)	void insertOrUpdateNonstrict(Member member)	○
n件	INSERT	int[] batchInsert(List<Member> memberList)		－
	UPDATE	int[] batchUpdate(List<Member> memberList)	int[] batchUpdateNonstrict(List<Member> memberList)	×
	DELETE	int[] batchDelete(List<Member> memberList)	int[] batchDeleteNonstrict(List<Member> memberList)	－

※1　排他制御の結果として対象のRecordが存在しない場合に EntityAlreadyUpdatedException が発生します。
　　この例外はS2Daoの排他例外である org.seasar.dao.NotSingleRowUpdateRuntimeException を継承しています。
※2　updateNonstrict()とdeleteNonstrict()は、対象のEntityが存在しない場合に EntityAlreadyDeletedException が発生します。
※3　Setterが呼び出された項目だけUpdate文のSet句に列挙して更新します。

DBFlute概要：ConditionBeanとは？

定番Queryのタイプセーフ実装を提供するObject

【主な特徴】

- o IDE補完によるメソッド選択型
- o 完全タイプセーフ{テーブル名・列名・演算子等}
- o 可読性考慮(GroupByや定番でない複雑なQueryはノンサポート)
- o in (select …)やexists (select …)など定番で、かつ、間違いやすいQueryのサポート

DBFlute概要：ConditionBeanの機能概要(基本)

```
Member cb = new Member();
// 結合してSelect句に設定(2階層親までOK → setupSelect_Xxx().withYyy();)
cb.setupSelect_MemberStatus();

// 基本条件
cb.query().setMemeberId_Equal(3); // MEMBER_ID = 3
cb.query().setMemberId_NotEqual(3); // MEMBER_ID != 3
cb.query().setUpdateDate_GreaterEqual('2007/11/11'); // UPDATE_DATE >= '2007/11/11'
cb.query().setUpdateDate_GreaterThan('2007/11/11'); // UPDATE_DATE > '2007/11/11'
cb.query().setUpdateDate_LessEqual('2007/11/11'); // UPDATE_DATE <= '2007/11/11'
cb.query().setUpdateDate_LessThan('2007/11/11'); // UPDATE_DATE < '2007/11/11'
cb.query().setMemberName_PrefixSearch('Billy'); // MemberName like 'Billy%'
cb.query().setMemberId_InScope(Arrays.asList(Integer[]{1,2,3})); // MEMBER_ID in (1,2,3)
cb.query().setMemberId_NotInScope(Arrays.asList(Integer[]{1,2,3})); // MEMBER_ID not in (1,2,3)

// 親テーブル列への条件(無限階層親までOK → query().queryXxx().queryYyy().queryZzz()…)
cb.query().queryMemberStatus().setMemberStatusName_Equal("正式会員");
cb.query.addOrderBy_UpdateDate_Desc(); // order by UPDATE_DATE desc
cb.query().addOrderBy_MemberId_Asc(); // order by UPDATE_DATE desc, MEMBER_ID asc
```

DBFlute概要：ConditionBeanの機能概要(応用)

一部紹介し切れない機能あり

```
Member cb = new Member();

// InScopeSubQuery → where MEMBER_ID in (select MEMBER_ID from PURCHASE where PURCHASE_COUNT >= 2)
PurchaiseCB purchaseCB = new PurchaseCB();
purchaseCB.query().setPurchaseCount_GreaterEqual(2);
cb.query().setMemberId_InScopeSubQuery(purchaseCB.query());

// ExistsSubQuery → where exists (select MEMBER_ID from PURCHASE where MEMBER_ID = MEMBER.MEMBER_ID and …)
PurchaiseCB purchaseCB = new PurchaseCB();
purchaseCB.query().setPurchaseCount_GreaterEqual(2);
cb.query().setMemberId_ExistsSubQuery(purchaseCB.query());

// 自テーブル同士のUNION (orの代替 → ConditionBeanではor句はできない(しない))
Member member = new Member()
member.query().setMemberStatusCode_Equal_Formalized();
cb.union(cb.query()); // UNION ALLの場合は、cb.unionAll(cb.query())

// ページング
cb.fetchFirst(20); // 最初の20件を検索
cb.fetchScope(40, 20); // 40件飛ばしてそこから20件を検索(41-60)
cb.fetchPage(3); // fetchFirst()と組み合わせて利用。1ページ20件として3ページ目(41-60)を検索
// 更新ロック
cb.lockForUpdate(); // Oracle「for update of …」、SQLServer「WITH(UPDLOCK)」、Other「for update」
```

最近のDBではorを解析時にUNIONするようになってきたが、まだまだ全てがそうではない

DBFlute概要：DBFlute概念図(詳細)