ビジネスロジック記述用
宣言的・What記述指向DSL

SPECRIPT

/* the first study */

2008.5.27

Specript Development Team

BACKGROUND

- 企業システム開発現場における問題
  - 新規開発要求の量と仕様変更要求の頻度の増大、そして短工期化
  - コード（動いている仕様）とドキュメント（文書化されている仕様）が乖離していく
  - 開発者の入れ替わりに伴い、システム改修時のエンバグが多発
- 根本原因
  - 業務記述の観点からは、プログラミング言語が‘低級’すぎる
  - 実装コードから仕様が読めない
- 最終回答
  - 業務記述にとって十分な記述レベルをもったWhat記述指向言語

    ”SPECRIPT” → ビジネスロジックのみを記述

OBJECTIVE

- コードからの仕様読解性を高める
  - 業務仕様記述と言えるレベルまでに記述の抽象度を高める
  - HOW記述を徹底排除→WHAT化
- 意図せぬバグ混入抑止を支援する機能を言語に持たせる
  - 以下3ポイントに着目：
    - ・データのタイプだけでなく、値内容に関わる規定を、プログラムコード　上で表す
    - ・分岐において、「そうでなかった場合」を必ず意識せざるを得ない仕　　組みを検討
    - ・プログラムモジュール間の依存・影響関係を意識できるような仕組　　みを検討

STRATEGY

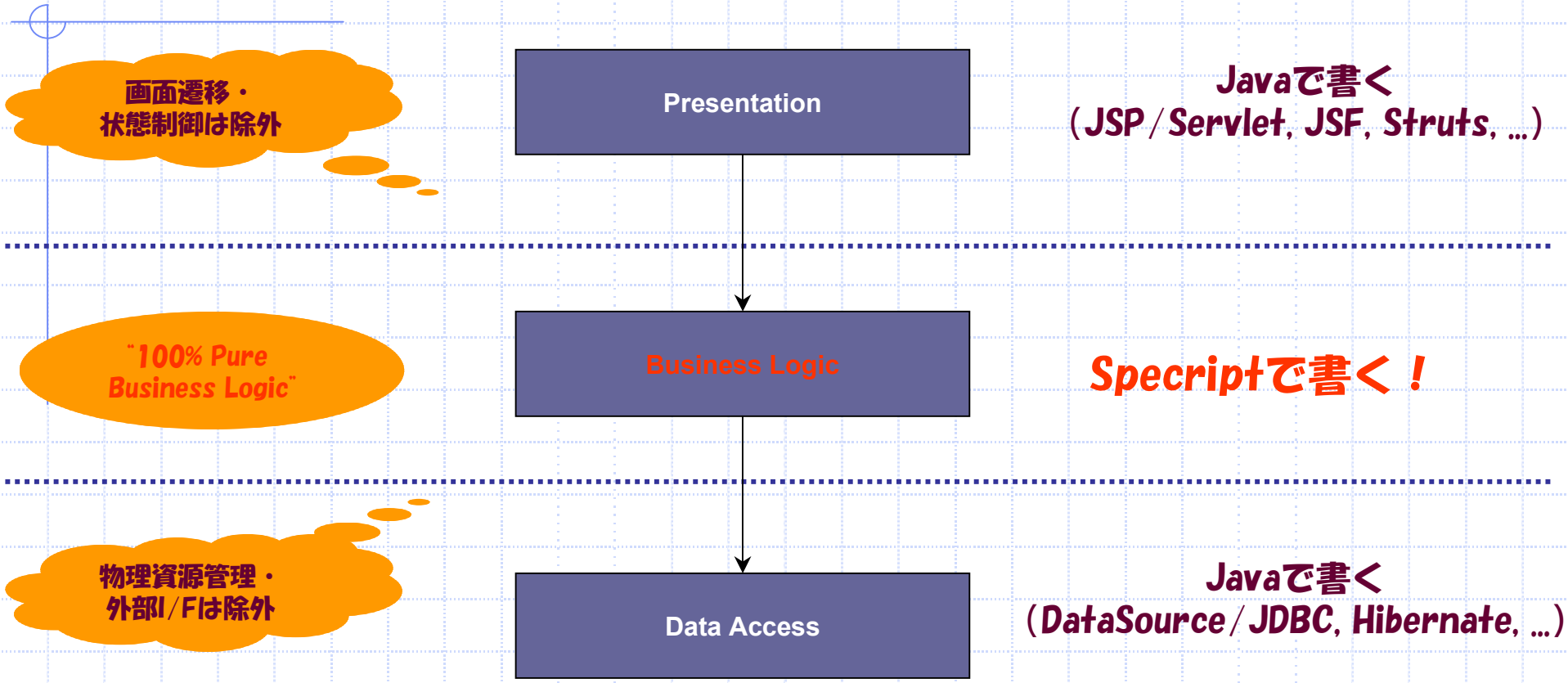
- WHAT化、STEP 1：物理環境に関わる事項、制御や状態管理に関わる事項は言語から排除
  - 外部リソース管理やデプロイ環境に関する諸事項を言語仕様から分離
  - アプリケーションの実行制御に関わる諸事項も分離
- WHAT化、STEP 2：「手続き的」→「宣言的」
  - データの値内容をも規定できる、高度な構造記述を考案

    キャッチフレーズ＝「ほとんど定義ファイルのようなソースコード」
- WHAT化、STEP 3：「どうしても残る手続き的要素」対策

  ~~ループ　条件分岐　一時変数~~
  - listに対する集合処理関数（select, transform, satisfiesAll, ...）
  - 繰り返し処理を抽象化した特別な"cumulate"関数
  - 拡張多分岐演算子
  - 代入処理（変数）の廃止

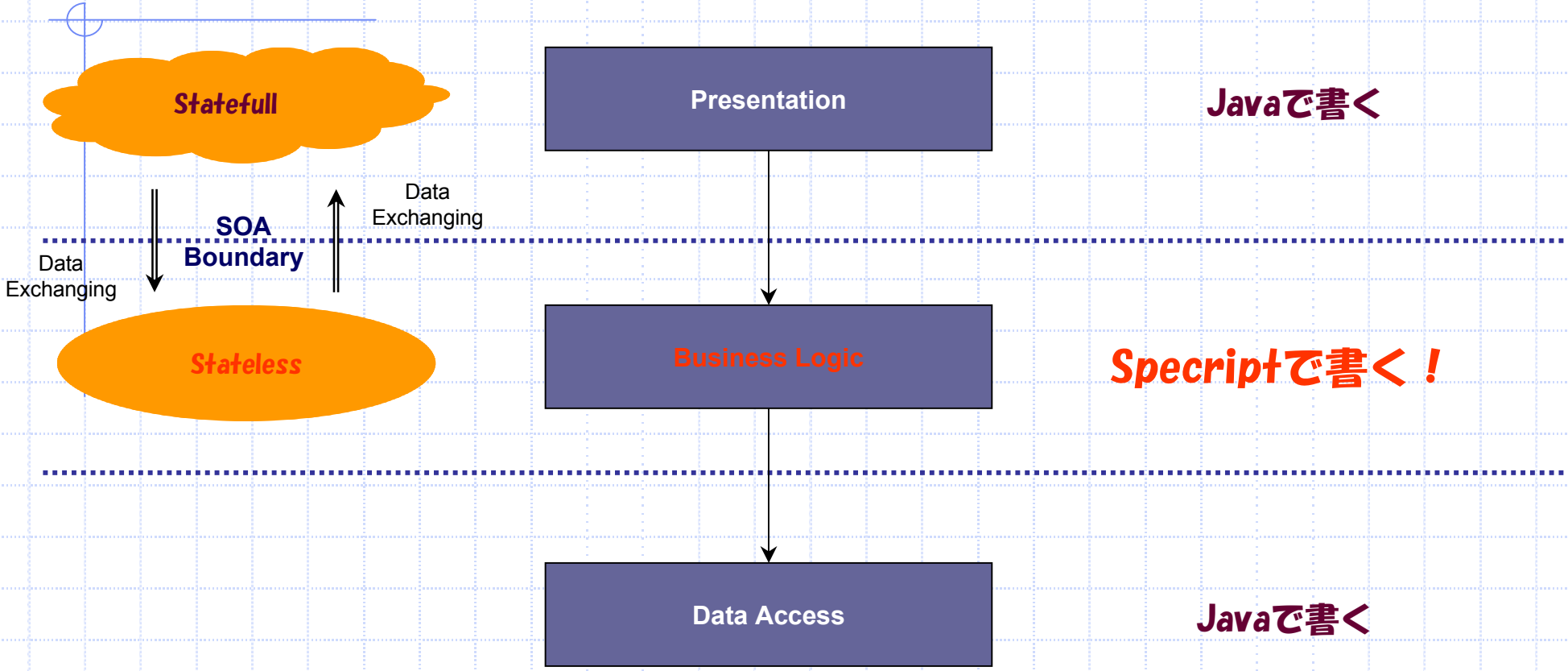
POSITION

●Business Logic記述に特化（Javaと連携することを前提）



ARCHITECTURE

●Service指向が似合う



LANGUAGE BASIC - OVERVIEW

- property
  - ✓データの定義
- function
  - ✓機能の定義
- spec
  - ✓データ仕様の定義
- namespace
  - ✓property/function/specの所属する業務領域の宣言
- 式
  - ✓全ての機能、処理は、「式」として表記する

```
// 業務領域の表明
namespace 商品::在庫;

// データの定義
property 在庫区分一覧 = [
    在庫区分コード.常備在庫 => {名称 = "常備在庫"},
    在庫区分コード.工場在庫 => {名称 = "工場在庫"},
    在庫区分コード.受注生産 => {名称 = "受注生産品"}
];

// 機能の定義
function 在庫ステータス(商品コード:商品コード仕様):在庫ステータス仕様 =
    在庫区分 = 商品(商品コード).在庫区分,
    実在庫数 = 現在の実在庫数(商品コード),
    在庫区分 ==?
        在庫区分コード.常備在庫 ?
            (実在庫数 >=?
                1000 ? 在庫ステータスコード.在庫有り:
                 300 ? 在庫ステータスコード.在庫僅少:
                 100 ? 在庫ステータスコード.在庫無し:
                   0 ? 在庫ステータスコード.出庫制限中
            ) :
        在庫区分コード.工場在庫 ?
            (実在庫数 >=?
                5000 ? 在庫ステータスコード.在庫あり:
                1500 ? 在庫ステータスコード.在庫僅少:
                 300 ? 在庫ステータスコード.在庫無し:
                 100 ? 在庫ステータスコード.出庫制限中
            ) :
        在庫区分コード.受注生産 ?
             在庫ステータスコード.直販品;

// データ仕様の定義
spec 商品在庫仕様 : {
    property 商品コード:      not null 商品コード仕様;
    property 受注可能在庫数: not null 受注数仕様;
    property 実在庫数:        not null 受注数仕様;
}

// データ仕様の定義
spec 在庫ステータス仕様 : string {
    function 名称 = 在庫ステータス一覧[this].名称;
    constraint function 在庫ステータス一覧に存在する =
        在庫ステータス一覧.contains(this) # "不明な在庫ステータスコードです。";
}
```

✧property

✓データの定義

property <識別子> : <spec> = <式> ;

✓右辺にrecord、list、mapの記述も可能

✓右辺に「式」の記述も可能(※定義時に一度だけ評価される)

```
// decimalタイプのproperty
property 消費税率:decimal = 0.07;
```

```
// 式の例
property 会計区分コード:会計区分コード仕様 = 商品(商品シリアルID).会計区分コード;
```

```
// list、mapの例
property 入力区分に対して指定可能な取引条件表:map<入力区分仕様, list<取引条件仕様>> =
[
    入力区分コード.一般 => [取引条件コード.特約店取引,
                            取引条件コード.業者間取引,
                            取引条件コード.販促品取引,
                            取引条件コード.直販取引
                           ],
    入力区分コード.TEL  => [取引条件コード.特約店取引,
                            取引条件コード.業者間取引
                           ],
    入力区分コード.FAX  => [取引条件コード.特約店取引,
                            取引条件コード.業者間取引
                           ],
    入力区分コード.営業 => [取引条件コード.特約店取引,
                            取引条件コード.業者間取引,
                            取引条件コード.直販取引
                           ]
];
```

✧function

✓機能の定義

function <識別子> (<引数>) : <返り値のspec> = <式> ; // 引数あり関数

function <識別子> : <返り値のspec> = <式> ; // 引数なし関数

```
// 単純な例
function 肥満度(身長:real, 体重:real):real = 体重 / (身長 * 身長);
```

```
// 条件分岐とrecord式の例
function 月次期間(日付:date):月次期間仕様 =
    日付.day <= 20 ? {年度    = 日付.year,
                      月度    = 日付.month,
                      開始日 = 日付.dateOfDay(21).decreaseMonth(1),
                      終了日 = 日付.dateOfDay(20)
                     }
                   :
                     {年度    = 日付.increaseMonth(1).year,
                      月度    = 日付.increaseMonth(1).month,
                      開始日 = 日付.dateOfDay(21),
                      終了日 = 日付.dateOfDay(20).increaseMonth(1)
                     };
```

次の「spec」で説明

✓propertyと引数なしfunctionの違い:

property → 宣言時に一度だけ評価／Java側と交換される

引数なしfunction → 呼び出しの都度評価／Java側と交換されない

✧spec

✓データ仕様の定義

spec <識別子> : <ベースspec> { <要素定義> <制約定義> };

✓Specriptにおけるデータタイプ（※構造体やクラスのような位置付け）

✓プリミティブタイプに対してもfunctionの定義ができる

```
// recordをベースとしたspec定義の例
spec 商品仕様 : record {
    property 商品コード      : not null 商品コード仕様;
    property 商品名称        : string;
    property 会計区分コード  : not null 会計区分コード仕様;
    //property 会計区分名称    : string  // propertyにすればJava側に渡る
    function 会計区分名称    : string
                            = 会計区分(会計区分コード).会計区分名称;
    property 在庫区分        : not null 在庫区分仕様;
    property 卸価格          : 価格仕様;
    property 標準小売価格    : 価格仕様;
    property 出荷開始日      : not null date;
    property 出荷停止日      : date;
}

// "record"は省略可
spec 商品仕様 : {
    ・・・
}
```

```
// stingをベースとしたspec定義の例
spec 在庫ステータス仕様 : string {
    function 名称 = 在庫ステータス一覧[this].名称;
}

property 在庫ステータス一覧 = [
    "1" => {名称 = "在庫有り"},
    "2" => {名称 = "在庫僅少"},
    "3" => {名称 = "在庫無し"},
    "4" => {名称 = "出庫制限中"},
    "5" => {名称 = "直販品"}
];

// その利用場面
property とある在庫ステータス:在庫ステータス仕様 = "2";
property とある在庫ステータスの名称 = とある在庫ステータス.名称;
```

✧spec、その２:制約

✓constraint(制約)をspecのメンバーとして定義可能

constraint function <識別子> = <式> ; // 制約関数

constraint property <識別子> = <式> ; // 制約属性

✓constraintは、property定義時、引数引渡し時などに評価される(実行時)

```
// 偶数
spec 偶数 : integer {
    constraint function 偶数のみ = (this % 2 == 0);
}

// その利用場面
property ある値:偶数 = 3;  // spec違反！

// 長さ６文字の文字列
spec 長さ６文字の文字列 : string {
    constraint function 長さ検査 = this.length == 6;
}

// その利用場面
property ある文字列:長さ６文字の文字列 = "ABCD";  // spec違反！
```

```
// stringをベースとしたspecにおける制約の例
spec 商品コード仕様 : string {
    constraint function ZZZ9999999 = this =~ "[A-Z]{3}[0-9]{7}";
}
```

```
// integerをベースとしたspecにおける制約の例
spec 価格仕様 : integer {
    constraint property min_value = -9999999;
    constraint property max_value = 9999999;
}
```

```
// recordをベースとしたspecの場合の例
spec 期間仕様 : {
    property 開始日:not null date;
    property 終了日:not null date;
    constraint function 開始日は終了日より前 = 開始日 < 終了日;
}
```

✧spec、その3:制約違反と制約違反情報

✓制約違反が発生するとJava側にSpecViolationExceptionをスロー

✓そのときSpecViolationExceptionに詰められる情報

constraint function <識別子> = <式> # <違反時情報の式> ;

constraint property <識別子> = <式> # <違反時情報の式> ;

```
namespace 商品;

// 制約違反情報の定義
spec 商品コード仕様 : string {
    constraint function ZZZ9999999 = this =~ "[A-Z]{3}[0-9]{7}" # "商品コードの形式が異なります。";
}

// 「商品」を取得するfunction
featured function 商品(商品コード:商品コード仕様):商品仕様 = ・・・;  // 実行時、引数「商品コード」のspecがチェックされる
```

```
// Java側

SpecriptRuntime sr = ・・・;
Object res = null;
try {
    res = sr.perform("商品::商品", new Object[] { "ABCDE" });
} catch(SpecViolationException ex) {
    List<Object> svi = ex.getSpecViolationInformation();
    System.out.println(svi.get(0));  // "商品コードの形式が異なります。"と出力される
}

// TODO resを利用するコード
```

✧namespace

✓property/function/specの所属する業務領域の宣言

namespace <識別子> :: <識別子> :: ・・・ ;

✓1ファイルに複数namespace宣言が可能

```
namespace マスター管理::代理店;
namespace 商品::会計::売掛;
namespace 商品::在庫::販売管理;

spec 代理店仕様 : {
    ・・・
}

spec 取引条件仕様 : string {
    ・・・
}
```

```
// 利用側

property とある代理店:商品::会計::売掛::代理店仕様 = ・・・;

property とある代理店:商品::在庫::販売管理::代理店仕様 = ・・・;
```

✓スコープ:

default → 自namespaceと、系列下位のnamespaceから参照可

private → 自ファイル内からのみ参照可

public → 他系列のnamespaceからも参照可

✧式

- ✓全ての機能、処理は「式」として表記する
- ✓条件分岐も「式」
- ✓ループの多くは集合演算関数で記述可能

```
// 多分岐演算子の例
function 在庫ステータス(商品コード:商品コード仕様):在庫ステータス仕様 =
    在庫区分 = 商品(商品コード).在庫区分,
    実在庫数 = 現在の実在庫数(商品コード),
    在庫区分 ==?
        在庫区分コード.常備在庫 ?
            (実在庫数 >=?
                1000 ? 在庫ステータスコード.在庫有り    :
                 300 ? 在庫ステータスコード.在庫僅少    :
                 100 ? 在庫ステータスコード.在庫無し    :
                       在庫ステータスコード.出庫制限中
            ) :
        在庫区分コード.工場在庫 ?
            (実在庫数 >=?
                5000 ? 在庫ステータスコード.在庫あり   :
                1500 ? 在庫ステータスコード.在庫僅少   :
                 300 ? 在庫ステータスコード.在庫無し   :
                       在庫ステータスコード.出庫制限中
            ) :
        在庫区分コード.受注生産 ?
             在庫ステータスコード.直販品;
```

```
// recordメンバーの初期化式
spec 登録時受注仕様 : 受注仕様 {
    ・・・

    // property 代理店コード:not null 代理店コード仕様;

    property 受注明細リスト:list<受注明細仕様 {

        // property 商品コード:not null 商品コード仕様;

        property 卸価格:not null 価格仕様 = 商品(商品コード).卸価格;
        property 請求区分コード:請求区分コード仕様
            = 請求区分コード取得(
                代理店コード = 代理店コード,
                会計区分コード = 商品(商品コード).会計区分コード);
    }>;

    ・・・
}
```

```
// 集合演算の例（※制約関数の場合の例）
constraint function 業者間取引のとき、一商品あたりの受注数は１グロス以上でなければならない =
    取引条件 == 取引条件コード.業者間取引 ? 受注明細リスト.satisfiesAll(|it.受注数 >= 144|) : true;
```

✧式、その２:"cumulate"関数

✓ループを抽象化した関数

```
// cumulate関数の定義
intern function <E, R> cumulate(
    for:list<E>,                          // ループのベースとなるリスト
    res:R,                                // 前回値蓄積用プロパティの初期値
    cont(res:R, it:E):boolean = |true|,   // 終了条件を表すClosure、省略可
    proc(res:R, it:E):R                   // 処理本体を表すClosure
):R;
```

```
// 合計
function 合計(alist:list<decimal>):decimal =
    cumulate(for = alist, res = 0, proc = | res + it |);

// select（抽出）関数の実装
function <E> select(alist:list<E>, cond(it:E):boolean):list<E> =
    cumulate(for = alist, res = [], proc = | res + (cond(it) ? [it] : []) |);
```

✧extern property／extern function

✓Javaで実装された外部コードの呼び出し

```
// extern functionの定義
public extern function query(sql:string, params:record):list<record>;
```

```
// Java側実装

import org.specript.runtime.extern.ExternFunction;

public class QueryFunction implements ExternFunction
{
    public Object evaluate(Object[] args) throws ApplicationException, InfrastructureException
    {
        ・・・
    }
}
```

✓Specript上での識別子と実装クラスの関連付けはSpecriptConfigクラスに設定

PRACTICE - OVERVIEW

- ソースコードからの仕様の可読性がよい
- 業務ルールをデータに関連付けつつ取り纏めることができる
- 想定外のデータの発生を確実に検知できる
- ケース漏れが起きない（※少なくとも起きにくい）
- 業務の依存関係あるいは‘文脈’を明示できる

◆ソースコードからの仕様の可読性がよい

・コンパクトな構文、制約やnot nullなどの厳密なデータ定義、処理のまとまりにはfunctionとして名前付けをしなければならないこと、これらがあいあまって、ソースコードからの仕様の可読性が高いと言える

【デモコードについて】

・代理店から商品の受注をする、というシナリオ

・取り扱いデータ種:

受注／受注明細
商品
在庫
代理店

・取り扱い属性データ:

会計区分 … 商品の属性で代理店との取引の有無に関わる
在庫区分 … 商品の属性で、取引条件との組み合わせなどで受注可否が決まる
在庫数 … 他の属性との組み合わせで受注可否がきまる
取引条件 … 受注の属性で、受注の内容を規定する
入力区分 … 受注の属性で、受注の内容を規定する

商品.spec

在庫ステータス.spec

受注.spec

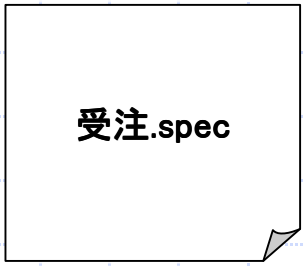
PRACTICE #2

◆業務ルールをデータに関連付けつつ取り纏めることができる

・spec定義に、そのデータの取り扱いに関わる業務ルールを制約として記述することができる

・業務ルールに違反するデータがあれば確実に検知される

・データの処理手順（※C/R/U/Dなど）から業務ルールを確実に分離できる



PRACTICE #3

◆想定外のデータの発生を確実に検知できる

- spec定義に、値内容に踏み込んだデータ項目の規定(＝仕様)を盛り込むことができる
- 制約違反があれば確実に検知される

```
spec 商品コード仕様 : string {
    constraint property min_length = 10;
    constraint property max_length = 10;
    constraint function ZZZ9999999 = this =~ "[A-Z]{3}[0-9]{7}";
}

spec 商品仕様 : {
     property 商品コード:not null 商品コード仕様;  // 書式に合わないデータがセットされることがあり得ない
     property 商品名称:string {  // 下記制約を越えた値がセットされることがあり得ない
         constraint property max_length = 40;
         constraint function バイト数の制限 = this.size("UTF8") <= 120;
     };
     property 会計区分コード:not null 会計区分コード仕様;
     ・・・
```

PRACTICE #4

◆ケース漏れがおきない(※少なくとも起きにくい)

- 条件演算子 "?" ～ ":" ～ の場合、「else」の記述が必須
- 多分岐演算子"==?"や"in?"の場合、「default」の記述が不可

```
// 条件分岐演算子の例
function 受注データステータス(受注シリアル番号:integer, 更新バージョン:integer, ロックする:boolean):データステータス仕様 =
    res = _受注トラン情報取得実行(受注シリアル番号 = 受注シリアル番号, ロックする = ロックする),
    (
        res == null                                          ? データステータスコード.NOT_EXISTING :
        res.DTS != null                                      ? データステータスコード.DELETED :
        更新バージョン != null && res.UVR != 更新バージョン ? データステータスコード.UPDATED :
                                                               データステータスコード.AVAILABLE
    );
```

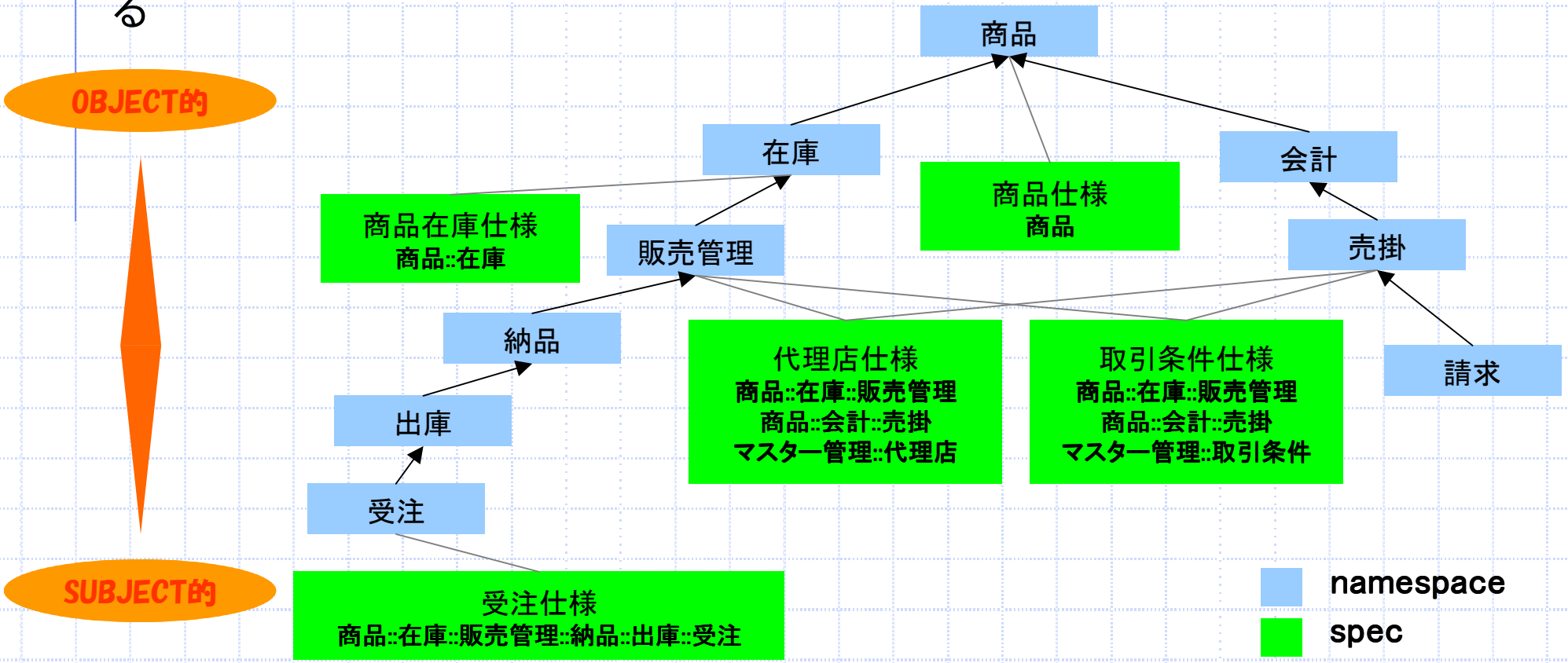
```
// 多分岐演算子"==?"と">=?"の例
function 在庫ステータス(商品コード:商品コード仕様):在庫ステータス仕様 =
    在庫区分 = 商品(商品コード).在庫区分,
    実在庫数 = 現在の実在庫数(商品コード),
    在庫区分 ==?
        在庫区分コード.常備在庫 ?
            (実在庫数 >=?
                1000 ? 在庫ステータスコード.在庫有り    :
                 300 ? 在庫ステータスコード.在庫僅少    :
                 100 ? 在庫ステータスコード.在庫無し    :
                       在庫ステータスコード.出庫制限中
            ) :
        在庫区分コード.工場在庫 ?
            (実在庫数 >=?
                5000 ? 在庫ステータスコード.在庫あり   :
                1500 ? 在庫ステータスコード.在庫僅少   :
                 300 ? 在庫ステータスコード.在庫無し   :
                       在庫ステータスコード.出庫制限中
            ) :
        在庫区分コード.受注生産 ?
             在庫ステータスコード.直販品;
```

```
// 多分岐演算子"in?"の例
constraint function 入力区分に対する受注日の制約 =
    入力区分 in?
        [入力区分コード.TEL, 入力区分コード.FAX]   ?
            Today <= 受注日 :
        [入力区分コード.一般, 入力区分コード.営業] ?
            (月次期間(Today).開始日 <= 受注日 &&
                受注日 <= 月次期間(Today).終了日);
```

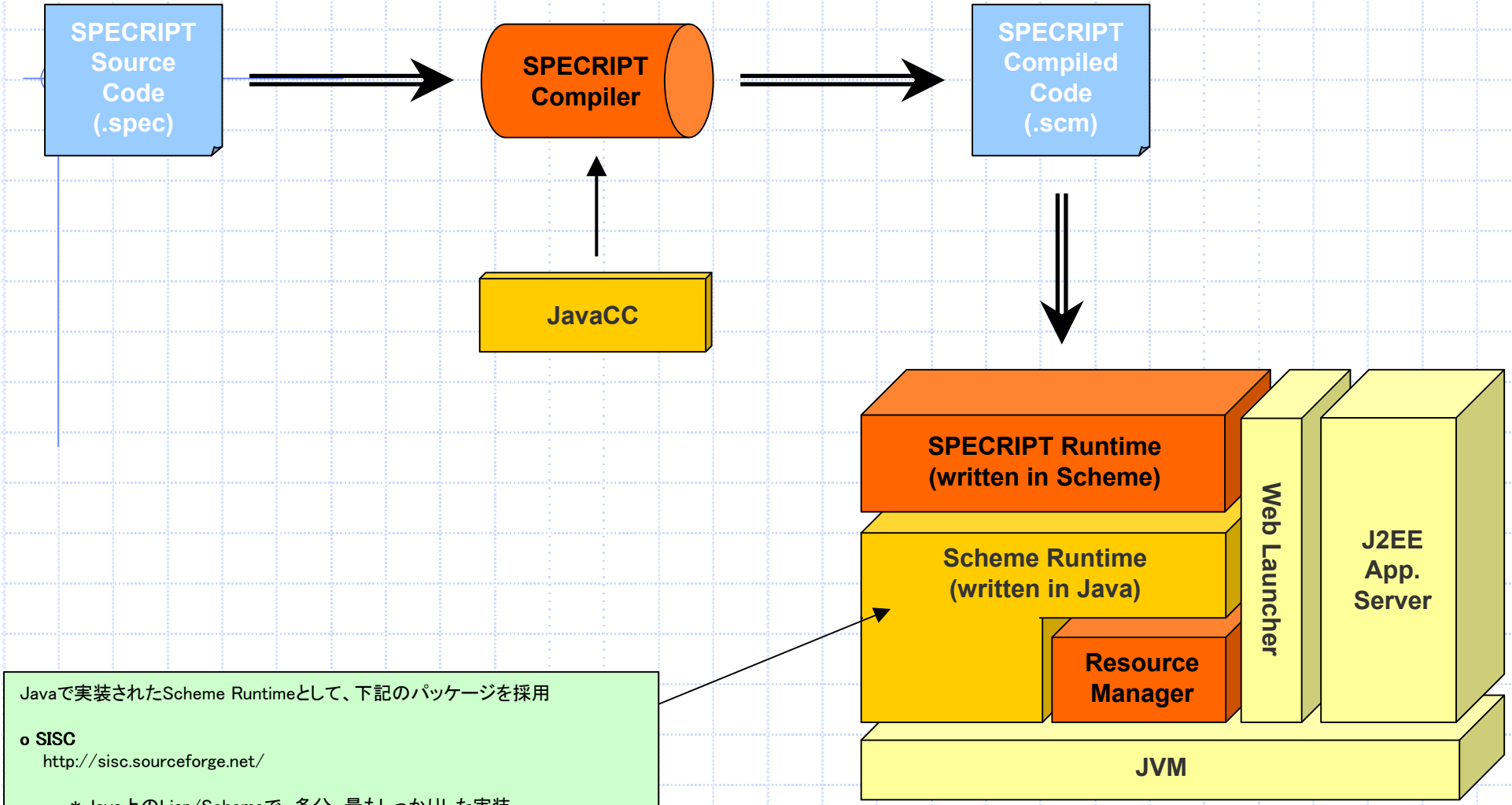
◆業務の依存関係（あるいは業務のコンテキスト）を明示できる

・業務構造を表すようにnamespace階層を構築することにより、業務の依存関係、もしくは業務の‘文脈’がソースコード上で明示的となる

・複数の依存関係があった場合も、複数のnamespaceを付与することで対応できる



IMPLEMENTATION



Javaで実装されたScheme Runtimeとして、下記のパッケージを採用

o SISC
http://sisc.sourceforge.net/

＊Java上のLisp/Schemeで、多分、最もしっかりした実装

JAVA API

- org.specript.runtime.SpecriptRuntime
  - *Specript実行環境の本体
  - *performメソッドでSpecript functionを呼び出す
- org.specript.common.SpecViolationException
  - *制約違反時にraiseされる例外
  - *チェック例外
- org.specript.common.ApplicationException
  - *Specriptプログラムとしてのアプリケーションエラーの発生
  - *RuntimeException
- org.specript.common.InfrastructureException
  - *何らかの環境的、外部資源的エラーの発生
  - *RuntimeException

CONCLUSION

一. 業務仕様記述と言えるレベルの記述性（＝What指向）を持った言語の開発を計画した

二. 結果的に、ビジネスロジック記述特化言語（DSL）とすることで、目的を一定レベル達成することができた

三. 開発された言語「Specript」固有の特徴、およびその効果として以下をあげられる：

- データのタイプに値内容に関する制約情報を盛り込めるようにしたことで、予期せぬデータの発生を確実に検知できるようにしたこと
- 条件分岐において、ifではelse必須、switch caseではdefaultが無い、という文法により、常に全てのケースを意識せざるを得ないようにしたこと
- namespaceの参照可能スコープの工夫と多重宣言を可能としたことで、プログラムモジュール間の影響関係を、業務のコンテキストを意識しつつ認識できるようになったこと

specdoc

・Javadocが「実装」を記述するJavaプログラムコードに「仕様」をコメントし、それをドキュメント化するものであるとしたら、Specriptにおけるコメントとは「要件」であるといる

・そこで、下記コメントタグを用意し、ドキュメント化するツールを開発する

@what これは何であるかの一般的説明

@why なぜこのような仕様であるかの背景説明

@how これを用いるときの適用方法や注意事項などの補足

・例えば、specdocで生成されるSpecriptソースコードから拾われた識別子のインデックスは、業務用語集やデータ種一覧となる可能性がある