社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
SS2007-22 (2007-08)

JMLを用いた在庫管理プログラムの設計とESC/Java2を用いた検証

尾鷲　方志†　岡野　浩三†　楠本　真二†

† 大阪大学大学院情報科学研究科　〒560-8531 豊中市待兼山 1-3

あらまし　近年，契約に基づく設計による開発手法が脚光を浴びている．これを実現する言語の一つとしてJMLが挙げられる．本研究では、プログラム設計の共通問題である酒屋の在庫管理問題に対してJMLを用いて仕様作成を行いプログラムを記述し，ESC/Java2を用いて実際に検証を行った．本報告では，この過程で得られた知見(作業量，現バージョンのESC/Java2の記法の問題点，解決法など)について報告する．また，ソフトウェア開発にJMLを用いる上での課題について考察する．

キーワード　ESC/Java2，JML，在庫管理問題，設計検証，契約による設計

A Case study — Design of Warehouse Management Program in JML and Verification with ESC/Java2 —

Masayuki OWASHI†, Kozo OKANO†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University
Matikaneyama 1-3, Toyonaka-shi, 560-8531 Japan

Abstract　In the software development, the notion "Design by Contract(DbC)" has been recently obtained a lot of attentions. In the DbC, contracts are represented as assertions. JML(Java Modeling Language) is one of the languages able to represent such assertions. In this research, we designed and programmed the Warehouse management program in Java and JML as a case study. This design is verified with "ESC/Java2". The paper reports the knowledge about what we learned in this process. Discussion about the issue on using JML in software development is also presented.

Key words　ESC/Java2, JML, Warehouse Manaagement, Design and verification, Design by Contract

1. まえがき

ソフトウェアの開発手法において，従来から用いられている手法としてV字型モデルによる設計法[8]がある．V字型モデルによる設計法とは，開発における各フェーズを「設計プロセス」と「検証プロセス」に分割し，それぞれを検証内容によって対応させたものである．

V字型モデルによる設計法において，近年，Design by Contract(契約に基づく設計，以下DbCとする)[1]による考え方が脚光を浴びている．DbCに基づいた開発では，アサーションにより契約を示すことで，詳述段階におけるデバッグを効率化され，品質が向上する．この考え方はV字型モデルによる設計法に対して非常に有用である．

DbCを実現するための言語としてJML[2]が挙げられる．Javaプログラムに対し，JMLは，Javaコード実体と結びついた契約を記述することができ，ESC/Java(2)[4][5]を用いて，コードが契約を侵す恐れがないかテストができるため，抽象度の低い段階での設計，実装に役立つ．

本事例研究では，プログラム設計の共通問題である在庫管理問題[10][11][12]についてモデル化し，JMLを用いて仕様を記述した．また，実際にJavaを用いてプログラムを記述した後，ESC/Java2を用いて検証を行い，設計と実装がともにESC/Java2において妥当であるということを示すことができた．さらに，例外処理に関する仕様，実装の正当性に関しても十分な記述をし，ESC/Java2において妥当であるとの出力を得ることができた．

最後に，今回の事例研究を通じて得られた知見に基づいて，今後の研究として他の抽象レベルの高い設計検証法などを組み合わせた応用例などについても述べる．

2. 在庫管理プログラムの仕様

2.1 概　　要

酒屋の在庫管理問題に対するクラス図を図1に表す．受付業務と倉庫業務を中心とした6のクラスで表される．それぞれの

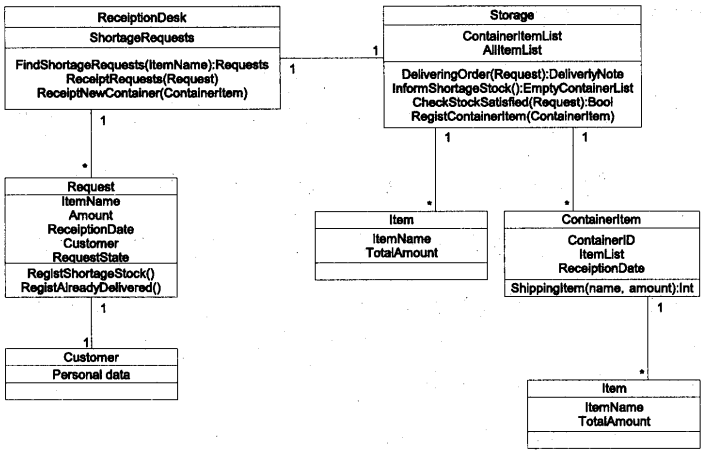

図 1　在庫管理プログラムのクラス構成

仕様について以下にまとめる．

a）受　付　係

ReceiptionDesk クラスで表され，外部からの要求を受け取り，倉庫に対して在庫が要求を満たすか確認させ，可能ならばその要求に含まれる商品の出庫を倉庫係に依頼し，不可能であれば未処理要求リストとして保存する．同様に，外部からコンテナを受け付け，受け付けたコンテナを倉庫係に搬入させる．このとき，未処理の要求があればその要求の処理をする．外部から渡されたデータを直接処理できるのは受付係のみである．

b）倉　庫　係

Storage クラスで表され，搬入したコンテナのリストをもっている．受付から送られてきた要求に対し，在庫が要求を満たしているかどうかをかえす．また，出庫指示に対して，どのコンテナからどれだけ出庫したかを示す出庫指示書を受付に返す．全品目の集合の各要素は一意であり，また，コンテナの集合に対し，各コンテナのコンテナ番号は一意である．品物の搬入はコンテナ単位で行われ，品目単位での出庫が行われる．

c）注　　文

Request クラスで表され，要素として，要求品名と要求数量，要求者 (顧客)，さらに要求が不足している，発送済みである等の状態を示す値を持つ．また，一度の要求に対して要求できる品目は一種類のみである．

d）顧　　客

Customer クラスで表され，顧客の住所，氏名などの個人情報を持つ．

e）品　　目

Item クラスで表され，要素として，品名，数量の情報を持つ．

f）コンテナ

ContainerItem クラスで表され，内蔵品の集合を持つ．各内蔵品は品目として表され，内蔵品の集合内の品目は一意である．また，倉庫からの出庫指示に対し，内蔵品の集合の中の品目を指定量取り出し，発送する．

2.2　実装上の工夫

倉庫に対しての商品の搬入はコンテナ単位で行われるが，顧客からの注文は品目単位で行われるため，コンテナの集合だけでなく，その中身をまとめた全品目の集合を要素として倉庫係に保持させた [11]．全品目の集合とコンテナの集合中の内蔵品の合計は互いに矛盾しない．

3.　JML 記述

3.1　JML

JML とは，“Java Modeling Language” の略で，仕様を記述するための言語である [2]．JML は Java プログラムの特殊なコメント中に注釈 (アノテーション) として，Java とよく似た文法で記述することができる．記述言語は基本的に Java であり，いくつかの拡張がなされている．拡張としては、 \old, \result, \forall, \exists などがあり，それぞれメソッド実行前のフィールドの値，そのメソッドの戻り値，ある条件を満たす任意のフィールドの値，ある条件を満たすフィールドの値の存在を意味する．

また，JML は，DbC に基づき，事前条件，事後条件などの契約により仕様を記述する [9]．JML による DbC の実現は主に以下の節を用いて実現される．

invariant　クラス不変式を記述する．このクラスのオブジェクトが生存中に各インスタンス変数が満たすべき契約を表す．

requires　メソッドの事前条件を記述する．これが記述されたメソッドが実行されるまでに満たしているべき契約を表す．

ensures　メソッドの事後条件を記述する．これが記述されたメソッドが実行された後に満たしているべき契約を表す．

JML は他にも，例外が起こる条件を示す “signals” 節や，無限ループが起こる条件を示す “diverges” などがある．

リファレンスは [3] にて公開されている．

3.2　記述概要

各クラスに対して invariant 節を用いた不変条件，また，各メソッドに対し requires 節，ensures 節を用いた事前，事後条件および，signals 節を用いて例外が起こる条件，また確実に例外が起こり得ないことを明示した．各クラスに対しての詳細な記述は，文献 [14] を参照のこと．記述と検証作業にかかった日数は 2ヶ月ほどで，この記述を得るのに全部で 6 回のバージョン変更を行っている．

```
/*@ public behavior
        requires c != null;
        assignable storage;
        ensures (\exists ContainerItem ci;
                storage.getContainerItemList().contains(ci);
                ci.getContainerID()==c.getContainerID());
        signals (Exception) false;
  @*/
public void receiptNewContainerItem(ContainerItem c)
{
        storage.registContainerItem(c);
        deliveringOrder();// 未発送リクエストの発送チェック
}
```

図 2　例外処理を含まないメソッドの仕様記述例

```
private /* spec_public non_null */ Date carryingDate;
private int containerID;
private /* spec_public non_null */ LinkedList itemlist;
/* public invariant containerID >= 0;
        public invariant \typeof(itemlist) == \type(Item);
        public invariant (\forall Item i,j;
                itemlist.contains(i) && itemlist.contains(j)
                && i != j;i.getName() !=j.getName());
  */
```

図 3　コンテナの不変条件

また，これらの JML 記述が，どのような規模の Java プログラムに対しどれだけの記述がなされたかについては表 1 にまとめた．

3.3 記　述　例

図 2 に例外処理を含まないメソッドの仕様記述例を示す．このメソッドは，酒屋の在庫管理問題をモデル化するにあたり定義した ReceiptionDesk クラスのメソッドで，新規コンテナの受付処理をするメソッドである．

このメソッドが呼び出し側に要求する事前の条件として，引数 c に有効な値が含まれていることを要求する．今回定義したクラスについて，null でなければ適当な値が入っている，もしくはそのオブジェクトが満たすべき本質的な性質が満たされていることが保証されている．例えば図 3 で示した今回引数とされるコンテナクラスの不変条件は，クラス内の各フィールドが null でなく，品名は一意であり，内蔵品リストの中身は品目である Item クラスであるということが保証されている．したがって，ここで記述すべき事前条件は c に有効な値が入っていることであるから，$c! = null$ とだけ記述する．

受け付けたコンテナは Storage.registContainerItem メソッドにより倉庫に格納されるので，倉庫の中に存在するはずであるから，\exists を用いて storage 中の containerlist に引数で渡された c と同じ ci が存在するはずであるということを事後条件として保証した．

また，このメソッドにより例外は起こらないので，そのように明示した．

```
/*@ also
        public normal_behavior
                .
                .
        also
        public exceptional_behavior
        requires o == null;
        signals_only java.lang.NullPointerException;
        also
        public exceptional_behavior
        requires !(o instanceof Request);
        signals_only java.lang.ClassCastException,
                java.lang.NullPointerException;
        signals (java.lang.ClassCastException);
  @*/
public /*@ pure @*/ int compareTo(Object o)
```

図 4　例外処理を含むメソッドの仕様記述例

3.4 例外処理記述例

図 4 に例外処理を含むメソッドの仕様記述例を示す．このメソッドは，酒屋の在庫管理問題をモデル化するにあたり定義した Request クラスのメソッドである．インターフェース Compareble のメソッド compareTo をオーバーライドしたものであるから，also 以下に仕様記述を始める．例外の振る舞いは，exceptional_behavior 以下に記述する．

まず，引数が null であった場合の例外としては NullPointerException を発生させるので，この場合の事前条件は，$o == null$ となる．また，このときは例外が NullPointerException 以外が発生しないため，signals_only 節に NullPointerException のみを記述した．

また，引数が Request 以外の型のものである場合もキャスト処理が定義されていないので例外として扱う必要があるから，事前条件に，!(o instanceof Request) と記述し，この場合は ClassCastException と NullPointerException が発生する可能性があるため signals_only 節にそのように記述した．

3.5 記述の上で遭遇した問題点

JML 記述において，メソッド呼び出しを用いることにより効率的に仕様を記述することができる．しかし，pure であるメソッドしか用いることができない．pure であるメソッドとは，メソッド実行の前後でオブジェクトの内部状態を変化させないメソッドのことである．メソッドの事後条件を記述する場合において，あるメソッドの戻り値を用いたい場合，そのメソッドが pure でなければ仕様記述に使うことができない．本研究では，受付の出庫処理の仕様記述において，この問題に直面した．これを解決する方法として考えられる方法は以下の二つが考えられる．

- 期待した戻り値を出力する pure なメソッドを別に作る
- ghost field の利用

前者の方法を用いた場合，新たなメソッドを定義することにより問題なく仕様記述を行うことができるが，実プログラム中で利用しないメソッドを仕様記述のためだけに実プログラムを用

```
/*@ public ghost List tmplist;
  @*/
/*@ public behavior
        assignable RequestList,storage;
        ensures (\forall Request r; RequestList.contains(r)
                && (\sum ContainerItem ci;tmplist.contains(ci)
                        && ((Item)ci.getContainedItem().get(0))
                                .getName().equals(r.getName());
                        ((Item)ci.getContainedItem().get(0))
                                .getAmount())==r.getAmount();
                r.getRequestState() == StockState.DELIVERED);
                ..
        signals (Exception) false;
  @*/
public List deliveringOrder()
{
        ..
        while(i.hasNext()){
                ..
                        deliveredlist.add(
                                storage.deliveringOrder(request));
                        //@ set tmplist = deliveredlist;
```

図 5　ghost field 使用例

いて定義するのは多くの状況において良い解決法とはならないことが多い.

ここではghost fieldを用いた. ghost fieldとは, JML記述中のみで扱われる特殊な変数で, 任意の式の値を代入しておき, 仕様記述に用いることができる. 図5に使用例を示した. ここでは, pureではないメソッド "Storage.deliveringOrder(request r)" の戻り値を用いたい場合に, ghostとしてJML記述中に宣言されたtmplistに上記の戻り値を含んだ変数を, 図5の最後のコメント文のようにsetを用いて保持している. このことにより, 事後条件の記述にメソッド中で "Storage.deliveringOrder(request r)" の戻り値を保持したtmplistを用いることができるため, pureでないメソッドの戻り値を事後条件の評価に利用することができた.

一方ghost fieldを用いる場合の欠点は, メソッドをそのまま用いる場合に比べ, 一見した場合に意味がわかりづらいことである. どのメソッドからどういう値を期待しているのかということを即座に理解するためにはできるだけメソッド呼び出しを用いたほうがよいと思われる.

また, ghost fieldのように, JML記述中のみで有効な特殊なメソッドを定義することもできるが, 事前, 事後条件のみの定義で期待する値を表現することができない場合もあるので, ここでは除外した.

4. ESC/Java2による検証

4.1 ESC/Java2

ESC/Java2におけるESCは "Extended Static Checker" の略であり, ESC/Java2とは, JMLつきのJavaプログラムのサブクラスを論理式に変換し, 記述されたプログラムが注釈付けられた仕様に対し妥当であるかを検証するツールである.

表 1　JML 設計の規模 (仮)

	LOC	JML の LOC
ReceiptionDesk	141	55
Storage	242	97
ContainerItem	171	69
Request	185	87
Item	78	37
Customer	122	55
総計	939	380

表 2　仕様記述に対する検証時間

Class Name	with Exception(s)	without Exception(s)
ReceiptionDesk	7.265	7.172
Storage	9.094	9.328
ContainerItem	18.079	18.125
Request	4.797	4.297
Item	9.328	9.328
Customer	6.375	
Total	54.938	54.575

ESC/Java2は, 定理証明器として一階述語論理の証明器である "Simplify" [7] を用いている. 前身のプロジェクトであるESC/Javaでは不十分であった, 仕様記述におけるメソッド呼び出しの利用の実装等が強化された.

ESC/Java2の検証は完全でも健全でもないが [6], 比較的簡単なロジックや, 型検査, 例外の検査などが得意であり, デバッガとしては優秀であるとの評価がある.

4.2 検証結果

文献 [14] のJML記述に対し, ESC/Java2を用いて一部警告はあるものの, ほぼ全てのクラスのメソッドに対しValidであるとの結果を得ることができた. 各クラスの検証にかかった時間は表2のようになった.

5. 考　　察

上記の検証結果より, 例外を明示することにより検証時間が目立って増加しているわけではないことがわかる. 一部のクラスの検証に関して言うならば, 検証時間が短くなっているものも見られる. このことから, [6] において言われていた, 例外が起こらない場合についてもそのことを明示すべきだ, という主張が数値的にも妥当である例を確認することができ, 実際に明示することによるデメリットはないと言うことができるだろう.

また, 仕様記述において,

\forall Object o; List.contains(i);...

のような形で各オブジェクトを参照するよりも,

\forall int i; 0 <= i && i < List.size();List.get(i)...

のように, 配列を扱うようにオブジェクトを参照した方がESC/Java2を通した検証にかかる時間が少なかった. 実際にContainerItemクラスの不変条件において適用した結果を表3に示す. 他のクラスについて同様の傾向があった.

Simplifyの入力となる論理式では, オブジェクトのフィールドを配列として表現するため [6], 配列形式を直接扱った方が定理証明にかかる時間が少なくすむと考えられる. 一方, 仕様記

表 3　配列形式の仕様記述にした場合の検証時間比較

ContainerItem	18.125
ContainerItem(Array)	16.891

述の観点からは前者の記述方のほうが好ましいと考えられる.

6. 今後の展望

JML が，Java のコードに結びついて，かなり詳細な設計を記述するための言語であることから，実際のプログラムコードに縛られない，より抽象的な設計を記述できる言語を用いた仕様記述を JML に変換する方針で研究を進めていきたいと考えている．抽象的な設計を記述できる言語の例として Alloy や VDM，Z 等が挙げられる．特に，Alloy は AlloyAnalyzer とセットになっていて，記述した仕様，またその仕様の反例を図示できるようになっていて，自動検証に強い [13]．このようなツールを用いて抽象段階での仕様の安全性が保証された設計から JML を自動生成できるようになると，手動で設計を記述する場合に比べ人為的なミスが減り，仕様，設計の信頼性の向上に繋がる.

7. ま と め

本研究において，酒屋の在庫管理問題についてモデル化し，JML を用いて仕様を記述することができた．また，実際に Java を用いてプログラムを記述し，ESC/Java2 を用いて検証を行い，設計と実装，例外処理がともに ESC/Java2 において妥当であるということができた.

今後は 6 で述べた内容について取り組みたい.

文　献

[1] Bertrand Meyer, Object-Oriented Software Construction, Second Edition, Prentice Hall, 2000.

[2] The Java Modeling Language, http://www.cs.iastate.edu/ leavens/JML/

[3] JML Reference Manual, http://www.cs.iastate.edu/ leavens/JML/jmlrefman /jmlrefman.html

[4] Extended Static Checking for Java (from Internet archive), http://web.archive.org/ http://research.compaq.com/SRC/esc/

[5] Kind Software, ESC/Java2, http://secure.ucd.ie/products/opensource/ESCJava2/

[6] David R. Cok, Reasoning with specifications containing method calls in JML and firstorder provers, ECOOP Workshop FTfJP'2004 Formal Techniques for Java-like Programs, Oslo, Norway, June 15, 2004. ICIS report NIII-R0426, pp 41-48, 2004

[7] David Detlefs, Greg Nelson, James B. Saxe, Simplify: a theorem prover for program checking, Journal of the ACM, Vol. 52 , Issue 3, pp. 365 - 473, 2005.

[8] 開発プロセスの基本を学ぶ, http://itpro.nikkeibp.co.jp/article/lecture/ 20061130/255501/?ST=develop

[9] Gary T.Leavens, Yoonsik Cheon Design by Contract with JML, September 28, 2006.

[10] 大木優, 竹内彰一, 宮崎敏彦, 古川康一, 二村良彦, Concurrent Prolog によるオンライン在庫管理システムの記述, TM0082, 1984.

[11] 岡野浩三, 北道淳司, 東野輝夫, 谷口健一, 順序機械型プログラムの階層的設計法と在庫管理プログラムの開発例, 電子情報通信学会論文誌, Vol.J76-DI, No.7, pp.354-363, 1993.

[12] 二村良彦, 雨宮真人, 山崎利治, 淵一博, 新しいプログラミング・パラダイムによる共通問題の設計, 情報処理学会会誌 情報処理, Vol.25, No.5, 1985.

[13] Daniel Jackson, Alloy: A Lightweight Object Modelling Notation, ACM Transactions on Software Engineering and Methodology, Vol.11, Issue 2, pp.256-290, April, 2002.

[14] 尾鷲方志, 岡野浩三, 楠本真二, JML を用いた在庫管理プログラムの設計と ESC/Java2 を用いた検証の詳細例, http://www-tani.ist.osaka-u.ac.jp/ m-owasi/esc_warehouse_full.pdf

付　録

1. 主要なクラスの JML 記述

1.1 ReceiptionDesk.java

```
package WarehouseManagement;
import java.util.*;

public class ReceiptionDesk {
        private /*@ spec_public non_null @*/
                List RequestList;
        private /*@ spec_public non_null @*/
                Storage storage;

        /*@ public invariant \typeof(RequestList)                    10
                        == \type(Request);
                public ghost List tmplist;
          @*/

        /*@ public behavior
                requires r != null;
                requires \typeof(storage.allitemlist)
                        == \type(Item);
                requires \typeof(storage.containerlist)
                        == \type(ContainerItem);                    20
                assignable storage, RequestList;
                ensures \result
                        == storage.checkStockSatisfied(r);
                signals (Exception) false;
          @*/
        public boolean receiptRequest(Request r){
                if(storage.checkStockSatisfied(r)){
                        storage.deliveringOrder(r);
                        return true;
                }                                                   30
                if(RequestList.add(r)){
                        int tmpsize = RequestList.size();
                        Request tmplist[] = new Request[tmpsize];
                        Iterator i = RequestList.iterator();
                        for(int j=0;i.hasNext();j++){
                                tmplist[j] = (Request)i.next();
                        }
                        Arrays.sort(tmplist);
                        RequestList.clear();
                        for(int j=0;j<tmpsize;j++){                 40
                                RequestList.add(tmplist[j]);
                        }
                        return false;
                }
                else return false;
        }
```

```
/*@ public behavior
        assignable RequestList,storage;
        ensures (\forall Request r;RequestList.contains(r)
                && (\sum ContainerItem ci;
                        tmplist.contains(ci) && ((Item)ci
                        .getContainedItem().get(0))
                        .getName().equals(r.getName());
                        ((Item)ci.getContainedItem().get(0))
                        .getAmount())==r.getAmount();
                r.getRequestState()
                        == StockState.DELIVERED);
        ensures (\forall Request r;
                RequestList.contains(r)
                && r.getRequestState()
                == StockState.SHORTAGE;
                \old(storage).checkStockSatisfied(r)
                == false);
        ensures (\forall Request r;RequestList.contains(r)
                && r.getRequestState()
                        == StockState.DELIVERED
                && \old(r.getRequestState())
                        == StockState.SHORTAGE;
                \result.contains(\old(storage)
                        .deliveringOrder(r)));
        signals (Exception) false;
  @*/
public List deliveringOrder()
{
        Iterator i = RequestList.iterator();
        List deliveredlist = new LinkedList();
        while(i.hasNext()){
                Request request = (Request)i.next();
                if(request.getRequestState()
                        != StockState.DELIVERED){
                        if(storage.checkStockSatisfied(
                                request)){
                                deliveredlist.add(
                                        storage
                                        .deliveringOrder(request));
                                //@ set tmplist
                                        = deliveredlist;
                                request
                                .registAlreadyDelivered();
                        } else {
                                request
                                .registShortageStock();
                                request
                                .getCustomer()
                                .informShortageStock(request);
                        }
                }
        }
        return deliveredlist;
}

/*@ public behavior
        assignable \nothing;
        ensures \typeof(\result) == \type(Request);
        signals (Exception) false;
  @*/
public /*@ pure @*/ List getRequestList()
{
        return RequestList;
}

/*@ public behavior
        ensures \result == storage;
        assignable \nothing;
        signals (Exception) false;
  @*/
public /*@ pure @*/ Storage getStorage()
{
        return storage;
}

/*@ public behavior
        requires \typeof(storage.allitemlist) == \type(Item);
        assignable \nothing;
        ensures \result == storage.getItemList();
        signals (Exception) false;
  @*/
public /*@ pure @*/ List getItemList()
{
        return storage.getItemList();
}

/*@ public behavior
        requires \typeof(storage.containerlist)
                == \type(ContainerItem);
        assignable \nothing;
        ensures \result
                == storage.getContainerItemList();
        signals (Exception) false;
  @*/
public /*@ pure @*/ List getContainerItemList()
{
        return storage.getContainerItemList();
}

/*@ public behavior
        requires c != null;
        assignable storage;
        ensures (\exists ContainerItem ci;
                storage.getContainerItemList().contains(ci);
                ci.getContainerID()==c.getContainerID());
        signals (Exception) false;
  @*/
public void receiptNewContainerItem(ContainerItem c)
{
        storage.registContainerItem(c);
        deliveringOrder();// 未発送リクエストの発送チェック
}

/*@ public behavior
        assignable RequestList, storage;
        signals (Exception) false;
  @*/
public ReceiptionDesk()
{
        RequestList = new LinkedList();
        storage = new Storage();
}
}
```