# BoxPacker Documentation

*Release version 3*

**Doug Wright**

**Dec 22, 2019**

# Contents

BoxPacker is an implementation of the "4D" bin packing/knapsack problem i.e. given a list of items, how many boxes do you need to fit them all in.

Especially useful for e.g. e-commerce contexts when you need to know box size/weight to calculate shipping costs, or even just want to know the right number of labels to print.

CHAPTER 1

# License

BoxPacker is licensed under the MIT license.

## 1.1 Installation

The recommended way to install BoxPacker is to use Composer. From the command line simply execute the following to add `dvdoug/boxpacker` to your project's `composer.json` file. Composer will automatically take care of downloading the source and configuring an autoloader:

```
composer require dvdoug/boxpacker
```

If you don't want to use Composer, the code is available to download from GitHub

### 1.1.1 Requirements

BoxPacker v3 is compatible with PHP 7.1+

**Note:** Still running an older version of PHP? No problem! BoxPacker v2 is compatible with PHP 5.4 and up, is just as production-ready, and is actively maintained.

### 1.1.2 Versioning

BoxPacker follows Semantic Versioning. For details about differences between releases please see What's new

## 1.2 Principles of operation

Bin packing is an NP-hard problem and there is no way to always achieve an optimum solution without running through every single permutation. But that's OK because this implementation is designed to simulate a naive human approach to the problem rather than search for the "perfect" solution.

This is for 2 reasons:

1. It's quicker
2. It doesn't require the person actually packing the box to be given a 3D diagram explaining just how the items are supposed to fit.

At a high level, the algorithm works like this:

- Pack largest (by volume) items first
- Pack vertically up the side of the box
- Pack side-by-side where item under consideration fits alongside the previous item
- If more than 1 box is needed to accommodate all of the items, then aim for boxes of roughly equal weight (e.g. 3 medium size/weight boxes are better than 1 small light box and 2 that are large and heavy)

## 1.3 Getting started

BoxPacker is designed to integrate as seamlessly as possible into your existing systems, and therefore makes strong use of PHP interfaces. Applications wanting to use this library will typically already have PHP domain objects/entities representing the items needing packing, so BoxPacker attempts to take advantage of these as much as possible by allowing you to pass them directly into the Packer rather than needing you to construct library-specific datastructures first. This also makes it much easier to work with the output of the Packer - the returned list of packed items in each box will contain your own objects, not simply references to them so if you want to calculate value for insurance purposes or anything else this is easy to do.

Similarly, although it's much more uncommon to already have 'Box' objects before implementing this library, you'll typically want to implement them in an application-specific way to allow for storage/retrieval from a database. The Packer also allows you to pass in these objects directly too.

To accommodate the wide variety of possible object types, the library defines two interfaces `BoxPacker\Item` and `BoxPacker\Box` which define methods for retrieving the required dimensional data - e.g. `getWidth()`. There's a good chance you may already have at least some of these defined.

If you do happen to have methods defined with those names already, **and they are incompatible with the interface expectations**, then this will be only case where some kind of wrapper object would be needed.

### 1.3.1 Examples

#### Packing a set of items into a given set of box types

```
<?php
    use DVDoug\BoxPacker\Packer;
    use DVDoug\BoxPacker\Test\TestBox;  // use your own object
    use DVDoug\BoxPacker\Test\TestItem; // use your own object

    $packer = new Packer();
```

(continues on next page)

(continued from previous page)

```
    /*
     * Add choices of box type - in this example the dimensions are passed in␣
↪directly via constructor,
     * but for real code you would probably pass in objects retrieved from a database␣
↪instead
     */
    $packer->addBox(new TestBox('Le petite box', 300, 300, 10, 10, 296, 296, 8,␣
↪1000));
    $packer->addBox(new TestBox('Le grande box', 3000, 3000, 100, 100, 2960, 2960, 80,
↪ 10000));

    /*
     * Add items to be packed - e.g. from shopping cart stored in user session. Again,
↪ the dimensional information
     * (and keep-flat requirement) would normally come from a DB
     */
    $packer->addItem(new TestItem('Item 1', 250, 250, 12, 200, true));
    $packer->addItem(new TestItem('Item 2', 250, 250, 12, 200, true));
    $packer->addItem(new TestItem('Item 3', 250, 250, 24, 200, false));

    $packedBoxes = $packer->pack();

    echo "These items fitted into " . count($packedBoxes) . " box(es)" . PHP_EOL;
    foreach ($packedBoxes as $packedBox) {
        $boxType = $packedBox->getBox(); // your own box object, in this case TestBox
        echo "This box is a {$boxType->getReference()}, it is {$boxType->
↪getOuterWidth()}mm wide, {$boxType->getOuterLength()}mm long and {$boxType->
↪getOuterDepth()}mm high" . PHP_EOL;
        echo "The combined weight of this box and the items inside it is {$packedBox->
↪getWeight()}g" . PHP_EOL;

        echo "The items in this box are:" . PHP_EOL;
        $packedItems = $packedBox->getItems();
        foreach ($packedItems as $packedItem) { // $packedItem->getItem() is your own␣
↪item object, in this case TestItem
            echo $packedItem->getItem()->getDescription() . PHP_EOL;
        }
    }
```

### Does a set of items fit into a particular box

```
<?php
    /*
     * To just see if a selection of items will fit into one specific box
     */
    $box = new TestBox('Le box', 300, 300, 10, 10, 296, 296, 8, 1000);

    $items = new ItemList();
    $items->insert(new TestItem('Item 1', 297, 296, 2, 200, false));
    $items->insert(new TestItem('Item 2', 297, 296, 2, 500, false));
    $items->insert(new TestItem('Item 3', 296, 296, 4, 290, false));

    $volumePacker = new VolumePacker($box, $items);
    $packedBox = $volumePacker->pack(); //$packedBox->getItems() contains the items␣
↪that fit
```

(continues on next page)

(continued from previous page)

```php
<?php
    $box = new TestBox('Le box', 300, 300, 10, 10, 296, 296, 8, 1000);

    /*
     *  You can also supply an (optionally pre-sorted) array of items. By default the library will sort the items
     *  passed to it via a heuristic to achieve optimal packing density. If you need to control the order of items,
     *  or have application-specific knowledge that sorting will not help (e.g. all items have the same dimensions)
     *  you can tell the library to skip this step.
     */

    $itemList = ItemList::fromArray($anArrayOfItems, true); // set the optional 2nd param to true if presorted

    $volumePacker = new VolumePacker($box, $itemList);
    $packedBox = $volumePacker->pack(); //$packedBox->getItems() contains the items that fit
```

## 1.4 Weight distribution

If you are shipping a large number of items to a single customer as many businesses do, it might be that more than one box is required to accommodate all of the items. A common scenario which you'll have probably encountered when receiving your own deliveries is that the first box(es) will be absolutely full as the warehouse operative will have tried to fit in as much as possible. The last box by comparison will be virtually empty and mostly filled with protective inner packing.

There's nothing intrinsically wrong with this, but it can be a bit annoying for e.g. couriers and customers to receive e.g. a 20kg box which requires heavy lifting alongside a similarly sized box that weighs hardly anything at all. If you have to send two boxes anyway, it would be much better in such a situation to have e.g. an 11kg box and a 10kg box instead.

Happily, this smoothing out of weight is handled automatically for you by BoxPacker - once the initial dimension-only packing is completed, a second pass is made that reallocates items from heavier boxes into any lighter ones that have space.

For most use-cases the benefits are worth the extra computation time - however if a single "packing" for your scenarios involves a very large number of permutations e.g. thousands of items, you may wish to tune this behaviour.

By default, the weight distribution pass is made whenever the items fit into 12 boxes or less. To reduce (or increase) the threshold, call `setMaxBoxesToBalanceWeight()`

```php
<?php
    use DVDoug\BoxPacker\Packer;

    $packer = new Packer();
    $packer->setMaxBoxesToBalanceWeight(3);
```

**Note:** A threshold value of either 0 or 1 will disable the weight distribution pass completely

## 1.5 Too-large items

As a library, by default BoxPacker makes the design choice that any errors or exceptions thrown during operation are best handled by you and your own code as the appropriate way to to handle a failure will vary from application to application. There is no attempt made to handle/recover from them internally.

This includes the case where there are no boxes large enough to pack a particular item. The normal operation of the Packer class is to throw an `ItemTooLargeException`. If your application has well-defined logging and monitoring it may be sufficient to simply allow the exception to bubble up to your generic handling layer and handle like any other runtime failure. Alternatively, you may wish to catch it explicitly and have domain-specific handling logic e.g.

```
<?php
    use DVDoug\BoxPacker\ItemTooLargeException;
    use DVDoug\BoxPacker\Packer;
    use DVDoug\BoxPacker\Test\TestBox;  // use your own object
    use DVDoug\BoxPacker\Test\TestItem; // use your own object

    try {
        $packer = new Packer();

        $packer->addBox(new TestBox('Le petite box', 300, 300, 10, 10, 296, 296, 8,␣
↪1000));
        $packer->addBox(new TestBox('Le grande box', 3000, 3000, 100, 100, 2960, 2960,
↪ 80, 10000));

        $packer->addItem(new TestItem('Item 1', 2500, 2500, 20, 2000, true));
        $packer->addItem(new TestItem('Item 2', 25000, 2500, 20, 2000, true));
        $packer->addItem(new TestItem('Item 3', 2500, 2500, 20, 2000, true));

        $packedBoxes = $packer->pack();
    } catch (ItemTooLargeException $e) {
        $problemItem = $e->getItem(); //the custom exception allows you to retrieve␣
↪the affected item
        // pause dispatch, email someone or any other handling of your choosing
    }
```

For some applications the ability/requirement to do their own handling of this case may not be wanted or may even be problematic, e.g. if some items being too large and requiring special handling is a normal situation for that particular business.

BoxPacker also supports this workflow with the `InfalliblePacker`. This class extends the base `Packer` and automatically handles any `ItemTooLargeExceptions`. It can be used like this:

```
<?php
    use DVDoug\BoxPacker\InfalliblePacker;
    use DVDoug\BoxPacker\Test\TestBox;  // use your own object
    use DVDoug\BoxPacker\Test\TestItem; // use your own object

    $packer = new InfalliblePacker();

    $packer->addBox(new TestBox('Le petite box', 300, 300, 10, 10, 296, 296, 8,␣
↪1000));
    $packer->addBox(new TestBox('Le grande box', 3000, 3000, 100, 100, 2960, 2960, 80,
↪ 10000));

    $packer->addItem(new TestItem('Item 1', 2500, 2500, 20, 2000, true));
```

(continues on next page)

(continued from previous page)

```
    $packer->addItem(new TestItem('Item 2', 25000, 2500, 20, 2000, true));
    $packer->addItem(new TestItem('Item 3', 2500, 2500, 20, 2000, true));

    $packedBoxes = $packer->pack(); //same as regular Packer
    $unpackedItems = $packer->getUnpackedItems();
```

## 1.6 Advanced usage

### 1.6.1 Used / remaining space

After packing it is possible to see how much physical space in each `PackedBox` is taken up with items, and how much space was unused (air). This information might be useful to determine whether it would be useful to source alternative/additional sizes of box.

At a high level, the `getVolumeUtilisation()` method exists which calculates how full the box is as a percentage of volume.

Lower-level methods are also available for examining this data in detail either using `getUsed[Width|Length|Depth()]` (a hypothetical box placed around the items) or `getRemaining[Width|Length|Depth()]` (the difference between the dimensions of the actual box and the hypothetical box).

**Note:** BoxPacker will always try to pack items into the smallest box available

#### Example - warning on a massively oversized box

```
<?php

    // assuming packing already took place
    foreach ($packedBoxes as $packedBox) {
        if ($packedBox->getVolumeUtilisation() < 20) {
            // box is 80% air, log a warning
        }
    }
```

### 1.6.2 Positional information

It is also possible to see the precise positional and dimensional information of each item as packed. This is exposed as x,y,z co-ordinates from origin, alongside length/width/depth in the packed orientation.

#### Example

```
<?php

    // assuming packing already took place
    foreach ($packedBoxes as $packedBox) {
        $packedItems = $packedBox->getItems();
```

(continues on next page)

(continued from previous page)

```
        foreach ($packedItems as $packedItem) { // $packedItem->getItem() is your own␣
↪item object
            echo $packedItem->getItem()->getDescription() .  ' was packed at co-
↪ordinate ' ;
            echo '(' . $packedItem->getX() . ', ' . $packedItem->getY() . ', ' .
↪$packedItem->getZ() . ') with ';
            echo 'l' . $packedItem->getLength() . ', w' . $packedItem->getWidth() . ',
↪ d' . $packedItem->getDepth();
            echo PHP_EOL;
        }
    }
```

### 1.6.3 Custom Constraints

For more advanced use cases where greater control over the contents of each box is required (e.g. legal limits on the number of hazardous items per box, or perhaps fragile items requiring an extra-strong outer box) you may implement the `BoxPacker\ConstrainedPlacementItem` interface which contains an additional callback method allowing you to decide whether to allow an item may be packed into a box or not.

As with all other library methods, the objects passed into this callback are your own - you have access to their full range of properties and methods to use when evaluating a constraint, not only those defined by the standard `BoxPacker\Item` interface.

#### Example - only allow 2 batteries per box

```
<?php
    use DVDoug\BoxPacker\Box;
    use DVDoug\BoxPacker\Item;
    use DVDoug\BoxPacker\ItemList;

    class LithiumBattery implements ConstrainedPlacementItem
    {

        /**
         * Max 2 batteries per box.
         *
         * @param  Box            $box
         * @param  PackedItemList $alreadyPackedItems
         * @param  int            $proposedX
         * @param  int            $proposedY
         * @param  int            $proposedZ
         * @param  int            $width
         * @param  int            $length
         * @param  int            $depth
         * @return bool
         */
        public function canBePacked(
            Box $box,
            PackedItemList $alreadyPackedItems,
            int $proposedX,
            int $proposedY,
            int $proposedZ,
            int $width,
```

(continues on next page)

(continued from previous page)

```
            int $length,
            int $depth
        ) {
            $batteriesPacked = 0;
            foreach ($alreadyPackedItems as $packedItem) {
              if ($packedItem->getItem() instanceof LithiumBattery) {
                  $batteriesPacked++;
              }
            }

            if ($batteriesPacked < 2) {
                return true;  // allowed to pack
            } else {
                return false; // 2 batteries already packed, no more allowed in this␣
↪box
            }
        }
    }
```

### Example - don't allow batteries to be stacked

```
<?php
    use DVDoug\BoxPacker\Box;
    use DVDoug\BoxPacker\Item;
    use DVDoug\BoxPacker\ItemList;

    class LithiumBattery implements ConstrainedPlacementItem
    {

        /**
         * Batteries cannot be stacked on top of each other.
         *
         * @param  Box            $box
         * @param  PackedItemList $alreadyPackedItems
         * @param  int            $proposedX
         * @param  int            $proposedY
         * @param  int            $proposedZ
         * @param  int            $width
         * @param  int            $length
         * @param  int            $depth
         * @return bool
         */
        public function canBePacked(
            Box $box,
            PackedItemList $alreadyPackedItems,
            int $proposedX,
            int $proposedY,
            int $proposedZ,
            int $width,
            int $length,
            int $depth
        ) {
            $alreadyPackedType = array_filter(
                iterator_to_array($alreadyPackedItems, false),
                function (PackedItem $item) {
```

(continues on next page)

(continued from previous page)

```
                    return $item->getItem()->getDescription() === 'Battery';
                }
            );

            /** @var PackedItem $alreadyPacked */
            foreach ($alreadyPackedType as $alreadyPacked) {
                if (
                    $alreadyPacked->getZ() + $alreadyPacked->getDepth() ===
↪$proposedZ &&
                    $proposedX >= $alreadyPacked->getX() && $proposedX <= (
↪$alreadyPacked->getX() + $alreadyPacked->getWidth()) &&
                    $proposedY >= $alreadyPacked->getY() && $proposedY <= (
↪$alreadyPacked->getY() + $alreadyPacked->getLength())) {
                    return false;
                }
            }

            return true;
        }
    }
```

## 1.7 What's new / Upgrading

**Note:** Below is summary of key changes between versions that you should be aware of. A full changelog, including changes in minor versions is available from https://github.com/dvdoug/BoxPacker/blob/master/CHANGELOG.md

### 1.7.1 Version 3

#### Positional information on packed items

Version 3 allows you to see the positional and dimensional information of each item as packed. Exposing this additional data unfortunately means an API change - specifically `PackedBox->getItems` now returns a set of `PackedItem`s rather than `Item`s. A `PackedItem` is a wrapper around around an *Item* with positional and dimensional information (x/y/z co-ordinates of corner closest to origin, width/length/depth as packed). Adapting existing v2 code to v3 is simple:

Before

```
<?php
    $itemsInTheBox = $packedBox->getItems();
    foreach ($itemsInTheBox as $item) { // your own item object
        echo $item->getDescription() . PHP_EOL;
    }
```

After

```
<?php
    $packedItems = $packedBox->getItems();
    foreach ($packedItems as $packedItem) { // $packedItem->getItem() is your own␣
↪item object
```

(continues on next page)

(continued from previous page)

```
        echo $packedItem->getItem()->getDescription() . PHP_EOL;
    }
```

If you use `BoxPacker\ConstrainedItem`, you'll need to make the same change there too.

#### PHP 7 type declarations

Version 3 also takes advantage of the API break opportunity introduced by the additional positional information and is the first version of BoxPacker to take advantage of PHP7's type declaration system. The core `BoxPacker\Item` and `BoxPacker\Box` interfaces definitions have been supplemented with code-level type information to enforce expectations. This is a technical break only, no implementation requires changing - only the correct type information added, e.g.

Before

```
<?php
    /**
     * @return string
     */
    public function getDescription()
    {
        return $this->description;
    }
```

After

```
<?php
    /**
     * @return string
     */
    public function getDescription(): string
    {
        return $this->description;
    }
```

### 1.7.2 Version 2

#### 3D rotation when packing

Version 2 of BoxPacker introduces a key feature for many use-cases, which is support for full 3D rotations of items. Version 1 was limited to rotating items in 2D only - effectively treating every item as "keep flat" or "ship this way up". Version 2 adds an extra method onto the `BoxPacker\Item` interface to control on a per-item level whether the item can be turned onto it's side or not.

#### Removal of deprecated methods

The `packIntoBox`, `packBox` and `redistributeWeight` methods were removed from the `Packer` class. If you were previously using these v1 methods, please see their implementations in https://github.com/dvdoug/BoxPacker/blob/1.x-dev/Packer.php for a guide on how to achieve the same results with v2.