

---

# **BoxPacker Documentation**

*Release version 1*

**Doug Wright**

**Nov 20, 2018**



---

## Contents

---

**1 License**

**3**



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.



BoxPacker is licensed under the [MIT license](#).

**Warning:** You are reading the documentation for BoxPacker v1. Although still supported with occasional back-ports from newer versions, all users are recommended to upgrade to v2/v3 which removes the “always keep flat” limitation on items.

## 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 is compatible with all versions of PHP 5.4+

### 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)

### 1.2.1 Limitations

- Items are assumed to be shipped flat (e.g. books, fragile items). The algorithm as implemented therefore considers simple 2D rotation of items when determining fit but will not try turning items on their side
- The algorithm does consider spatial constraints in all 3 dimensions, plus consideration of weight

## 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();

/*
 * 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, 1_
↳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
 */
$packer->addItem(new TestItem('Item 1', 250, 250, 12, 200));
$packer->addItem(new TestItem('Item 2', 250, 250, 12, 200));
$packer->addItem(new TestItem('Item 3', 250, 250, 24, 200));

$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;
    $itemsInTheBox = $packedBox->getItems();
    foreach ($itemsInTheBox as $item) { // your own item object, in this case_
↳TestItem
        echo $item->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);

```

(continues on next page)

(continued from previous page)

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

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

## 1.4 Advanced usage

### 1.4.1 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
    }
}
```

### 1.4.2 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\ConstrainedItem` 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 ConstrainedItem
{
    /**
     * @param ItemList $alreadyPackedItems
     * @param TestBox $box
     *
     * @return bool
     */
    public function canBePackedInBox(ItemList $alreadyPackedItems, Box $box)
    {
        $batteriesPacked = 0;
        foreach ($alreadyPackedItems as $packedItem) {
            if ($packedItem instanceof LithiumBattery) {
                $batteriesPacked++;
            }
        }

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

```

## 1.5 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
    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

---

## 1.6 What's new / Upgrading

---

**Note:** A full changelog, including changes in versions not yet released is available from <https://github.com/dvdoug/BoxPacker/blob/master/CHANGELOG.md>

---