# OSE Workbench Platform

***Release 0.1.4***

**G Roques**

**Dec 27, 2020**

# WORKBENCH PLANNING

A platform for workbench development by Open Source Ecology.

OSE defines a "workbench" as a set of tools in CAD software to design and make a particular machine.

The below **Workbench Planning** pages cover planning for workbench development. Note, anyone can contribute to the workbench planning process, and you don't need to be a programmer. Indeed, OSE workbench development teams benefit from a diversity of people with different backgrounds and skill-sets.

The below **Developer Onboarding** pages contain guides for getting setup as an OSE workbench developer.

The below **Development Process** pages describe various processes related to development such as breaking down development work, branching, and versioning.

The below **Pattern Catalog** pages describe structure and patterns for solving common problems in workbenches using the FreeCAD platform.

Every workbench should follow the above standards and guidelines to make working between various workbenches easier, and increase collaboration.

CHAPTER

ONE

# DECIDING ON A MACHINE

OSE Workbenches generally center around **one** machine in the Global Village Construction Set (GVCS).

Examples of machines in the GVCS include:

- Tractors
- Compressed Earth Brick (CEB) Presses
- Power Cubes
- and 3D Printers

The first step in workbench planning is deciding on a machine to focus on.

The goal of an OSE workbench should be to stream-line the design of machine variants.

## 1.1 Amount of Allowed Variation

The balance between how much variation the workbench allows in the design of a machine is a delicate one.

On the one hand, the value of a workbench is it's *strictness* in variation based on what OSE has tested and discovered works best in practice.

On the other hand, the value of a workbench is allowing people to deviate from the "standard design" to create a design suited to their unique needs.

## 1.2 Related Machines

Depending on which machine the workbench focuses on, there may be *related* machines.

For example, the Tractor includes a Power Cube as it's primary energy source.

These relationships need to be considered carefully in the initial workbench development planning phase.

In the above example, it would be wise to create a **Power Cube Library** that *both* the Tractor and Power Cube workbenches rely on.

## 1.3 Next Step

Once the machine is decided, you must break down the machine into individual parts.

CHAPTER

TWO

# BREAKING DOWN A MACHINE INTO PARTS

Every machine can be broken down into individual parts.

For example, a simplified part breakdown of a 3D printer might be:

- Frame
- X, Y, and Z Axes
- Extruder
- Heated Bed

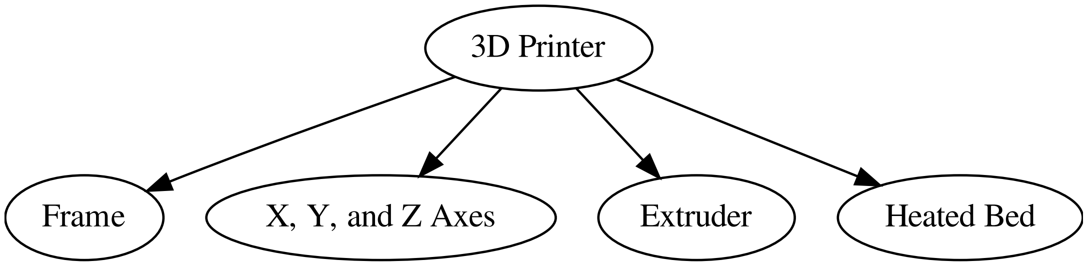


Fig. 1: Simplified part breakdown of a 3D printer

Each of these parts usually correspond to buttons on the **main toolbar** of a workbench, and need corresponding icons.

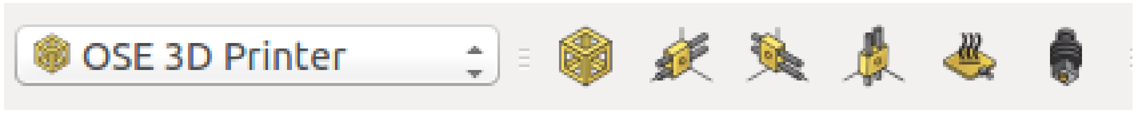


Fig. 2: 3D Printer Workbench: Main Toolbar Buttons

Clicking one of these buttons adds the corresponding part to the user's active document in FreeCAD.

## 2.1 Level of Breakdown

Do you need to include *every* part of a machine in the initial breakdown?

No.

For example, we excluded *less-critical* parts of the 3D Printer such as the controller, power supply, wiring, and spool holder.

Is our above simplified part breakdown still useful? Yes.

Thus, a workbench *only* including a simplified minimal set of **core parts** is useful.

In fact, to begin workbench development, definining the *minimal set of core parts* for the machine is recommended.

## 2.2 Define Core Parts

How do you decide on which parts to include in the minimal core set?

It's helpful to identify a subject matter expert (SME) or Product Owner to assist in this decision.

Someone who's knowledgeable about the machine, and what would be most useful to users of the workbench.

Start with the Minimum Viable Product (MVP), iterate, and come back to the other parts you left out in a later phase.

Deciding on the parts of a machine is not performed once and unable to change.

It's an *iterative* process that occurs over the lifetime of a workbnech.

## 2.3 Next Steps

The next two steps in workbench planning can be performed in parallel:

1. Breaking down parts into sub-parts
2. Designing icons

CHAPTER

THREE

# DESIGNING ICONS

Once the machine is broken down into parts, icons representing each part can be designed.

See FreeCAD Wiki: Artwork Guidelines for recommendations on how to design the icons.

Once designed, add the icons to the OSE FreeCAD Icons page for documentation purposes.

If you lack a designer, or desire to create icons, then you may *temporarily* use existing FreeCAD icons that adhere to FreeCAD Artwork Guidelines.

However, this approach is not recommended for a long-term solution, and care must be taken to design appropriate icons for each part that are distinct from other icons in the FreeCAD UI.

CHAPTER

FOUR

# BREAKING DOWN PARTS INTO SUB-PARTS

Once the machine is broken down into individual parts, then those parts can be further broken down into **sub-parts**.

Going back to the simplified part breakdown of a 3D printer as an example:

- Frame
- X, Y, and Z Axes
- Extruder
- Heated Bed

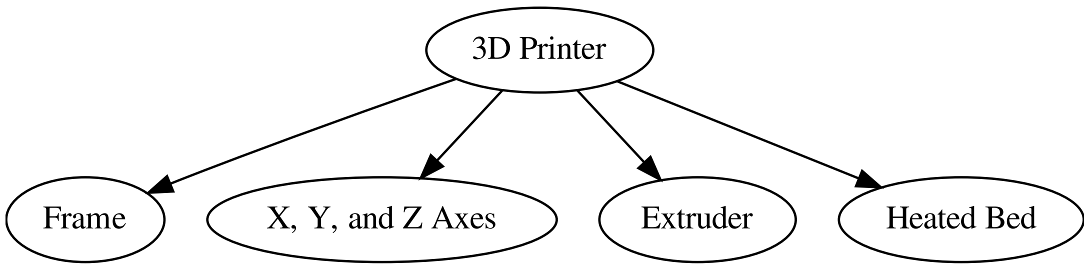


Fig. 1: Simplified part breakdown of a 3D printer

We may breakdown the Frame into:

- Angled Bars
- Angled Bar Connectors

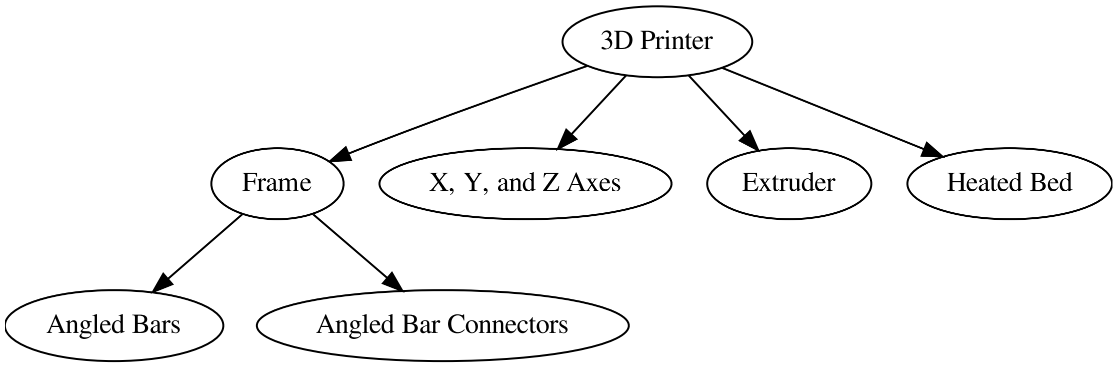


Fig. 2: Further part breakdown of a 3D printer with Frame breakdown

## 4.1 Terminology

The amalgamation of parts and sub-parts is called an **assembly**.

Also note, there may not be a difference between a **part** and a **sub-part**.

The "part" and "sub-part" terms are contextual.

For example, the Frame is both a part in it's own right, and a sub-part of the 3D printer.

## 4.2 Level of Breakdown

Similarly, we could breakdown the axes, extruder, and heated bed into sub-parts.

Then, we could continue breaking down those sub-parts into sub-parts until we get to the most basic parts of the machine.

There's no real limit to how far you can breakdown a machine. It's recommneded to continue breaking down a machine for as long as it's useful and practical.

**Tip:** See Depth of Modularity for more information.

Similar guidance as specified in breaking down a machine applies.

For the first iteration of a workbench, it's easier to include less detail in the breakdown of parts.

File size, memory consumption, and performance must also be considered when designing a workbench.

For example, parts that include more details will take up more space on disk, take longer to render, and potentially slow down FreeCAD.

Due to these limitations, starting with simplified parts is recommneded.

## 4.3 3D Printing Considerations

Does the part need to be 3D printed?

If so, then you'll need to include all details in the part.

In cases like this, it's helpful to allow the user to create a simplified version of the part for modeling purposes, and the full-detailed part separately for exporting to STL or OBJ for printing.

## 4.4 Shared Sub-Parts

The process of breaking down parts into sub-parts can reveal **shared sub-parts**.

For example, the axis and extruder might both contain a motor, or the same fasteners like nuts, screws, and bolts.

This information is useful to programmers as they can abstract the modeling for these parts into a common place for re-use.

## 4.5 Next Step

Once the top-level parts are broken down into sub-parts, those parts can be designed in FreeCAD.

CHAPTER

FIVE

# DESIGNING PARTS

Once the machine is broken down into individual parts, and those parts are broken down into sub-parts, then someone can design those parts in FreeCAD.

The generated FreeCAD asset files for each part can be documented on the OSE Wiki as a **Part Library**.

The **Part Library** serves as a helpful guide for developers who need to replicate that geometry programatically in Python.

## 5.1 Design All the Variations

Does the part have different states or variations?

For example, the Angle Frame Connector, or part that connects the angled bars together for the 3D Printer Frame, can include extra geometry for holding the angled bars in place with a set screw.

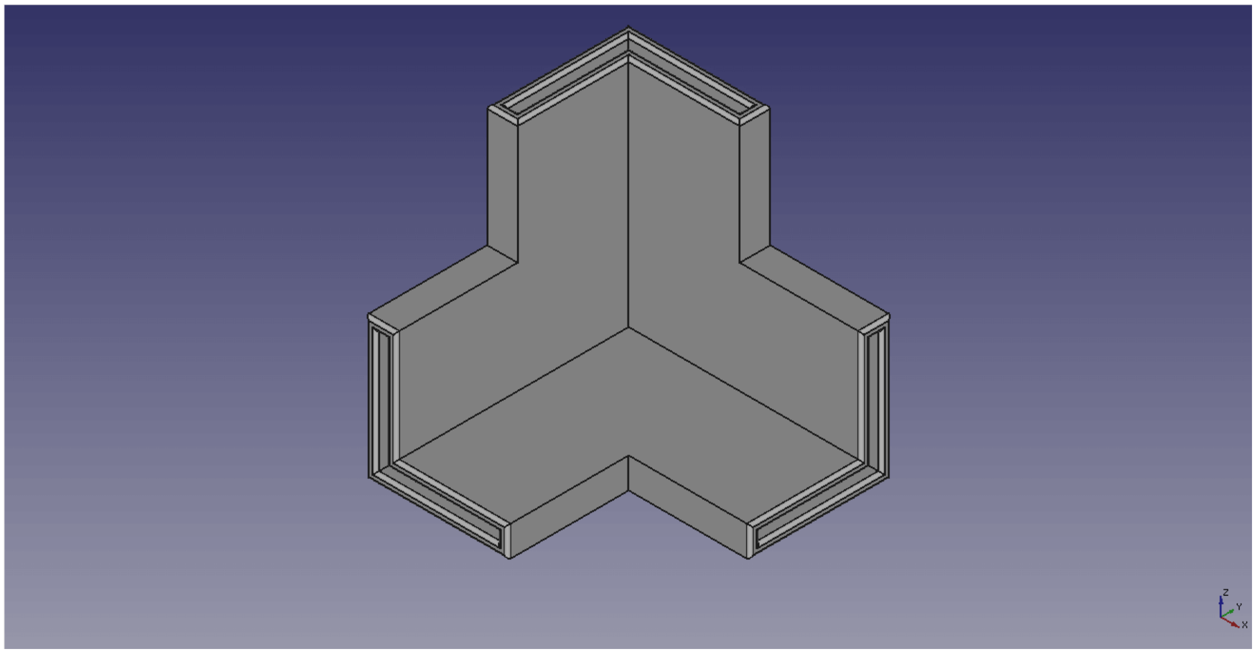


Fig. 1: Angle Frame Connector

Note, the basic geometry in both designs are very similar.

The design with a set screw is a more complicated variation of the initial design.

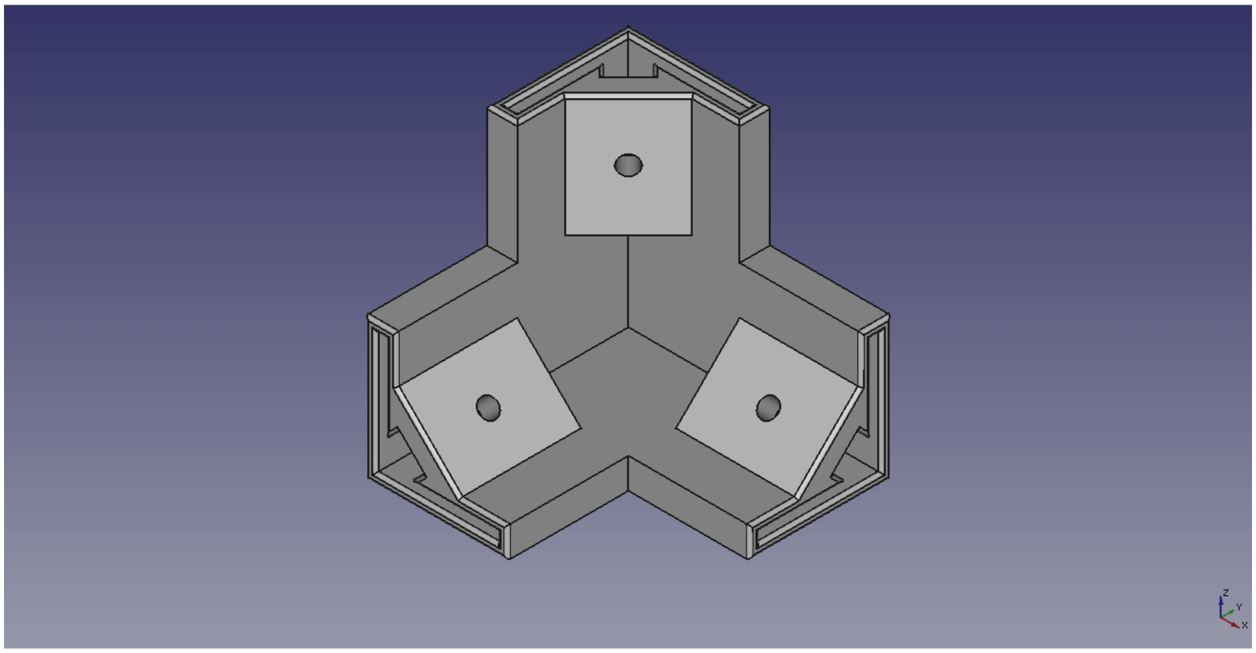


Fig. 2: Angle Frame Connector with Set Screw

A developer can take advantage of this similarity to reduce and share code when developing the part in the workbench.

It's easier to identifiy variations up-front in the design phase before writing the code.

## 5.2 Define the Parameteric Properties

What are the parameteric properties?

What attributes of the part do you want to parameterize, or allow the user to change and input values for?

For example, the Angle Frame Connector can have different slot widths and thicknesses to support smaller or larger 3D Printer Frames.

Below we see two angle frame connectors with different values for these parameters.

| Angle Frame Connector | Slot Width | Slot Thickness |
|---|---|---|
| Small | 1.5 inches | 0.125 inches |
| Large | 3 inches | 0.25 inches |

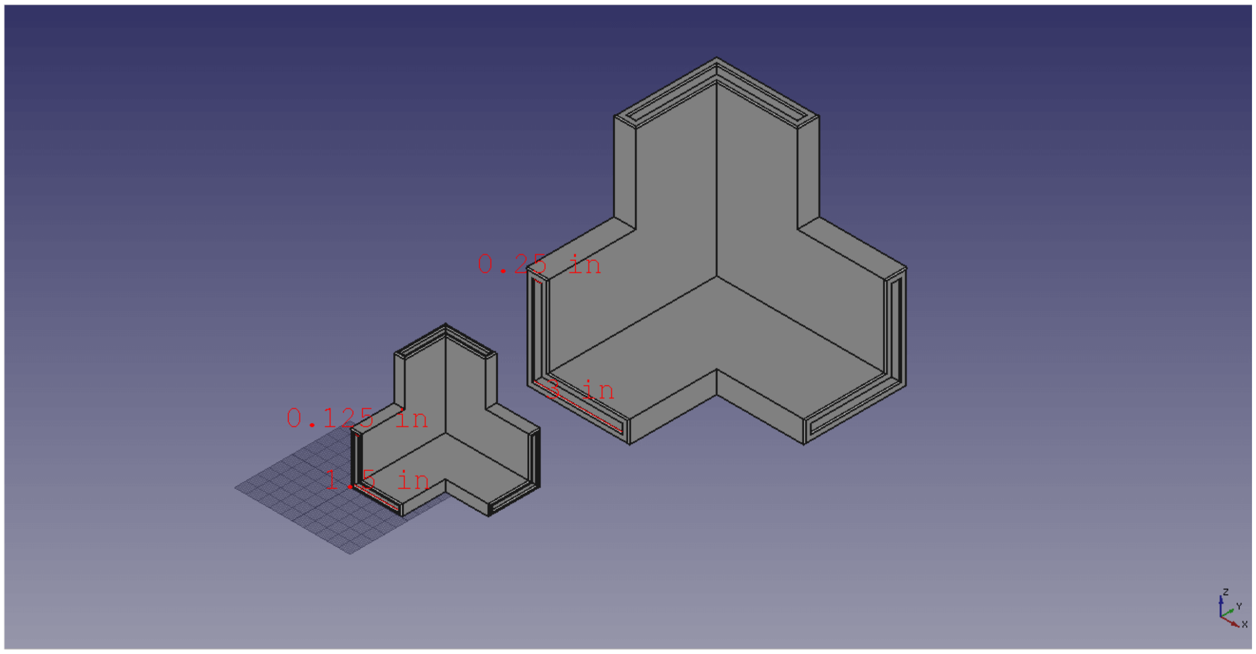


Fig. 3: Angle Frame Connector with Different Sizes

## 5.3 Next Step

After the parts are designed, you can define the relationship between parts.

CHAPTER

SIX

# DEFINING RELATIONSHIPS BETWEEN PARTS

Once the parts are designed, relationships between parts can be defined.

## 6.1 Attachment

A common way to relate parts together is through **attachment**.

For example, in a 3D Printer, axes can be attached to the frame.

If the user moves the frame, then the axes should move accordingly.

Note, these relationships can be hierarchical.

For example, you can attach an X axis to the top of the frame.

Then, you can attach an extruder to the X axis.

The extruder isn't directly attached to the frame, but if the frame moves, then the X axis should move, and thus the extruder should also move.

## 6.2 Parent-Child Relationships

It's also possible that values for properties trickle down from parts to sub-parts through parent-child relationships.

For example, the 3D Printer frame may a thickness property to adjust how thick the metal is.

Sub-parts like the angled bars and connectors inherit the value for the thickness property of the frame to ensure they always match.

Identifying properties that flow from parent to child parts can be helpful prior to development.

**Tip:** Tree diagrams can be made in the part breakdown phase to visualize parent-child part relationships

CHAPTER

# SEVEN

# EDITOR

The following page outlines guides for recommended editors and integrated development environments (IDEs) for OSE workbench development.

## 7.1 Visual Studio Code

Visual Studio Code is a **free**, cross-platform, extensible editor built on open-source.

### 7.1.1 Recommended Extensions

| Extension | Description |
|---|---|
| Python | Python language support |
| Python Test Explorer for Visual Studio Code | Run your Python Unittest or Pytest tests with the Test Explorer UI. |
| Python Docstring Generator | Quickly generate docstrings for python functions. |

The above extensions should be listed in `.vscode/extensions.json` within each workbench repository to prompt the user to install them. See Workspace recommended extensions for additional information.

### 7.1.2 Recommended Extensions Configuration

OSE Workbench Platform includes an `editor-config` command configuring the above VS Code extensions with the recommended configuration.

See the editor-config command documentation in the README for additional information.

~~~
OSE Workbench Platform includes an `editor-config` command for outputting recommended␣
↪VS Code configuration.

```
$ osewb editor-config -h
usage: osewb editor-config

optional arguments:
  -h, --help            show this help message and exit
  -m, --merge-workspace-settings
                        Merge VS Code workspace settings.
  -o, --overwrite-workspace-settings
                        Overwrite VS Code workspace settings.
~~~

(continues on next page)

(continued from previous page)

~~~
```

Simply running `osewb editor-config` will output the recommended VS Code␣
↪configuration settings which user's can copy-paste into their VS Code user settings,
↪ or workspace settings, `settings.json` file(s). See [User and Workspace␣
↪Settings](https://code.visualstudio.com/docs/getstarted/settings) for additional␣
↪information.

The `-m` or `--merge-workspace-settings` flag will merge the current VS workspace␣
↪settings into the platform's recommended settings. The platform's settings will win␣
↪any collisions or merge conflicts.

The `-o` or `--overwrite-workspace-settings` flag will overwrite the current VS Code␣
↪workspace settings with either the minimal-set of recommended configuration or␣
↪merged settings depending upon the presence of the `-m` flag. Before overwriting,␣
↪users will see a preview of the settings and must confirm overwriting in a yes or␣
↪no CLI prompt.
~~~

### 7.1.3 Python Docstring Generator Configuration

A **custom docstring template** for OSE workbenches has been included in `ose-workbench-platform/osewb/.mustache`.

```
{{! Sphinx Docstring Template }}
{{! For VSCode Python Docstring Generator Extension }}
{{! https://marketplace.visualstudio.com/items?itemName=njpwerner.autodocstring }}
{{! Modified to remove typePlaceholder }}
{{summaryPlaceholder}}

{{extendedSummaryPlaceholder}}

{{#args}}
:param {{var}}: {{descriptionPlaceholder}}
{{/args}}
{{#kwargs}}
:param {{var}}: {{descriptionPlaceholder}}
{{/kwargs}}
{{#exceptions}}
:raises {{type}}: {{descriptionPlaceholder}}
{{/exceptions}}
{{#returns}}
:return: {{descriptionPlaceholder}}
{{/returns}}
{{#yields}}
:yield: {{descriptionPlaceholder}}
{{/yields}}
```

You should configure the extension to use this template in order to avoid adding **types** to your docstrings.

Types in docstrings are redundant with Type Hints — the preferred way to document the types in Python.

**Is VS Code open-source?**

Explained by a VS Code developer:

> When we set out to open source our code base, we looked for common practices to emulate for our scenario. We wanted to deliver a Microsoft branded product, built on top of an open source code base that the community could explore and contribute to.
>
> We observed a number of branded products being released under a custom product license, while making the underlying source code available to the community under an open source license. For example, Chrome is built on Chromium, the Oracle JDK is built from OpenJDK [. . . ] Those branded products come with their own custom license terms, but are built on top of a code base that's been open sourced.
>
> We then follow a similar model for Visual Studio Code. We build on top of the vscode code base we just open sourced and we release it under a standard, pre-release Microsoft license.

## 7.2 VSCodium

For open-source purists, you may be interested in the MIT-licensed VSCodium as a VS Code alternative.

## 7.3 Don't See Your Preferred Editor?

We need **you** to help write the guide!

# CHAPTER EIGHT

# BREAKDOWN STRATEGY

This page defines a process for breaking down the development work of a workbench.

1. Initialize Workbench
2. Make Parts
3. Parameterize Parts
4. Attachment
5. Cut List Generation
6. CAM File Generation

If you have multiple developers, then work on separate parts in parallel:

**Important:** Each step in the above process may not apply to all parts depending upon requirements.

## 8.1 1. Make a New Workbench

1. Use the `osewb make workbench` command to make a new workbench.
2. Create a git repository and host it on a centralized platform like GitHub.

## 8.2 2. Make Parts

1. Add packages for each part in the part package and corresponding part classes.
2. Add icons for each part in the icon package.
3. Add packages for each part in the command package and corresponding command classes that call the part classes.
4. Register that command in the command registry module and associate it to the **main toolbar**.

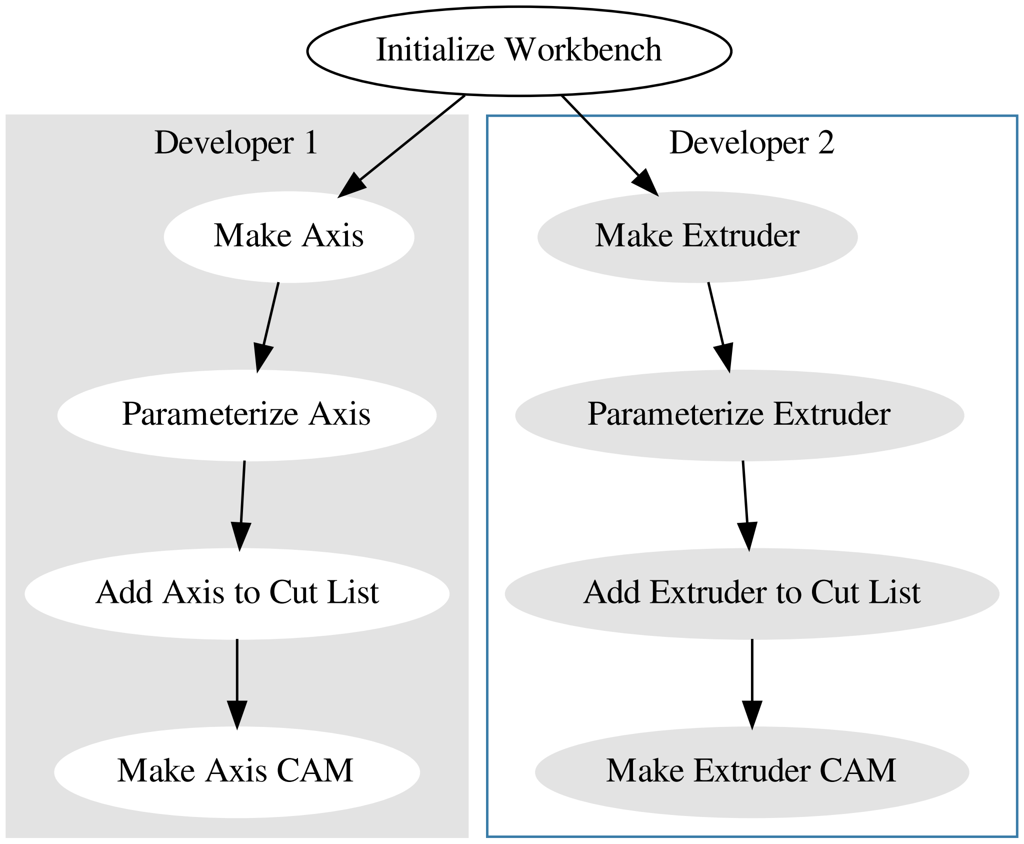


Fig. 1: Breakdown Strategy for Multiple Developers

## 8.3 3. Parameterize Parts

1. Add packages for each part in the model package and corresponding model classes.
2. Add packages for each part in the part feature package and corresponding part feature creation functions.
3. Refactor the corresponding command class to call the newly created part feature creation function instead of the part classes.

## 8.4 4. Attachment

1. Add packages for each attachment relationship in the attachment package and corresponding attachment functions.
2. Refactor the corresponding command class to call the attachment function, and refactor part feature creation function, model, and part classes as needed.

## 8.5 5. Cut List Generation

1. Add commands for generating a cut list.
2. Modify the `build_cut_list` function as needed for each part.

## 8.6 6. CAM File Generation

1. Modify part classes with as much detail as needed for CAM file generation. If a lot of detail is needed, then refactor the part class to support making a simplified or detailed version.
2. If needed, created a new command for exposing the detailed version of the part and expose that to the user through the **main menu** while the **main toolbar** exposes the simplified version for modeling purposes.

CHAPTER

NINE

# BRANCHING STRATEGY

OSE Workbenches should follow the Feature Branch Workflow:

> The core idea behind the **Feature Branch Workflow** is that all feature development should take place in a dedicated branch instead of the `master` branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the `master` branch will never contain broken code, which is a huge advantage for continuous integration environments.
>
> Encapsulating feature development also makes it possible to leverage pull requests, which are a way to initiate discussions around a branch. They give other developers the opportunity to sign off on a feature before it gets integrated into the official project. Or, if you get stuck in the middle of a feature, you can open a pull request asking for suggestions from your colleagues. The point is, pull requests make it incredibly easy for your team to comment on each other's work.
>
> —Atlassian, Git Feature Branch Workflow

# CHAPTER TEN

# VERSIONING STRATEGY

OSE workbenches should use Semantic Versioning strategy:

> Given a version number MAJOR.MINOR.PATCH, increment the:
>
> - **MAJOR** version when you make incompatible API changes,
> - **MINOR** version when you add functionality in a backwards compatible manner, and
> - **PATCH** version when you make backwards compatible bug fixes.
>
> Additional labels for pre-release are available as extensions to the MAJOR.MINOR.PATCH format.
>
> —https://semver.org/

CHAPTER

ELEVEN

# THIRD PARTY SERVICES

OSE workbenches rely on a number of wonderful and free third party services.

The page aims to document which external third party services we depend on and why.

| Name | Description |
|---|---|
| | Git repository hosting and integration with Travis CI & Read the Docs. |
| | For continuous integration. |
| | Documentation hosting and themeing. |
| ANACONDA CLOUD | Hosting ose-workbench-platform conda package. |
| | Hosting ose-workbench-platform python package. |

CHAPTER

TWELVE

# REPOSITORY SCOPE AND NAMING

**Motivation**

Provide a uniform catalog of OSE workbenches.

As dicussed in Workbench Planning: Deciding on a Machine, workbenches should generally be centered around **one** OSE machine in the Global Village Construction Set (GVCS).

Each workbench should have a single Git repository hosted on a centralized, publicly available, free platform like GitHub.

## 12.1 Repository Naming Convention

OSE workbench repository names should be in **all lower-case letters** with dashes - delimiting spaces, following the pattern `ose-<machine>-workbench`.

For example, the OSE workbench repository for power cubes should be named `ose-power-cube-workbench`.

Note, the machine name is in **singular** form.

CHAPTER

THIRTEEN

# ROOT REPOSITORY CONTENTS

**Motivation**

Ensure workbenches contain the same core elements.

The following page describes the directories and files included in the root of the repository.

```
$ tree -a --sort=name -L 1 -F --dirsfirst
.
├── docs/
├── freecad/<package name>/
├── <package name>/
├── tests/
├── CONTRIBUTING.md
├── environment.yml
├── .gitignore
├── LICENSE
├── MANIFEST.in
├── README.md
├── .readthedocs.yml
└── setup.py
```

## 13.1 README File

Every workbench should have a README file, named `README.md`, containing basic information about the project.

## 13.2 License File

Each workbench should include a software license in a file named `LICENSE`.

We recommend the GNU Lesser General Public License, version 2.1, as it's same license as FreeCAD to ensure workbenches could potentially be incorporated into future FreeCAD modules or FreeCAD source itself.

## 13.3 Contributing Guidlines

OSE workbenches should include contributing guidelines describing how people can contribute to the project inside a file named `CONTRIBUTING.md`.

## 13.4 Library & Workbench Package

Workbenches should organize source code into two main packages:

1. A library package
2. and a workbench package

The library package should be named `ose<machine>`, where `<machine>` is the name of the machine in all lower-case letters without spaces, hypens, or underscores.

The workbench package should be named the same as the library package, but located inside a directory named `freecad/`.

For example, the library package of the `ose-power-cube-workbench` should be named `osepowercube`.

```
.
├── osepowercube/           # Library Package
└── freecad/osepowercube/   # Workbench Package
```

This naming convention follows PEP 8's guidance on package naming:

> Python packages should . . . have short, all-lowercase names, . . . the use of underscores is discouraged.
>
> —PEP 8

For additional information, see App Gui Architecture.

## 13.5 Documentation

Documentation for workbenches should be located in the `docs/` directory.

Hosting of documentation should be performed by Read the Docs with configuration located in `.readthedocs.yml`.

## 13.6 Tests

Tests for workbenches should be located in the `test/` directory.

## 13.7 Continuous Integration

Workbenches should use Travis CI for Continuous Integration (CI).

Following the Feature Branch Workflow, each feature branch will be tested to ensure it doesn't break existing code before that branch is merged into the `master` branch.

Configuration for Travis CI is located within a file named `.travis.yml`.

## 13.8 Setup Module

Workbenches should include a `setup.py` module for describing how to package and distribute the workbench as a Python package.

## 13.9 MANIFEST.in

The `MANIFEST.in` file describes additional files to include in the Python package distribution.

For more information, see Including files in source distributions with MANIFEST.in.

## 13.10 environment.yml

The `environment.yml` file describes how to create a conda environment for local workbench development.

## 13.11 .gitignore

A .gitignore file should be included to specify any directories and files that shouldn't be checked into version control.

# CHAPTER FOURTEEN

# APP GUI ARCHITECTURE

**Motivation**

Encapsulate source code and separate the geometry of parts from their graphical representation.

FreeCAD is made from the beginning to work as a command-line application without its user interface. Therefore, almost everything is separated between a "geometry" component and a "visual" component. When you execute FreeCAD in command-line mode, the geometry part is present, but the visual part is absent.

For more information, see "Python scripting tutorial - App and Gui", on the FreeCAD Wiki.

OSE workbenches mirror this structure, and separate code into two main sub-packages:

1. A library package containing `App` functionality
2. A workbench package containing `Gui` functionality

In doing so, workbenches gain the following advantages:

- Provide the ability to run the library package from a command-line context, similar to FreeCAD
- Encapsulate logic in the library package, and keep the workbench package "dumb"
- Make the library package easy to write unit tests for

At a high-level, the library package contains code related to the geometry of parts, and how those parts relate to each other.

While the workbench package contains code related to the graphical user interface of FreeCAD, such as what happens when users interact with the workbench (e.g. a user clicks a button on a toolbar), or various components the user may interact with such as dialogs or panels.

Code in the workbench package may reference code in the library package, while **the reverse is not true**.

The main goal of this rule is to decouple machine-specific knowledge, such as the geometry of parts, from it's graphical representation.

In doing so, theoretically, other frontends besides FreeCAD's GUI can be used to display and interact with OSE's machines. For example, imagine other desktop, web, or mobile applications.

See Library Package and Workbench Package for additional information.

CHAPTER

FIFTEEN

# LIBRARY PACKAGE

**Motivation**

Organize code related to the geometry parts, and allow parts to be made from a command-line context.

The library package, located within the root level of the repository, contains code for the geometry of parts, and how those parts relate to each other.

The "geometry of parts" is defined as:

- Geometric primitives that make up parts such as vertexes, edges, and faces
- Basic shapes such as boxes, circles, cones, and cylinders
- and operations on, or between these primitives and basic shapes such as extrusion, chamfer, union, difference, or intersection.

For a formal introduction to these concepts, see Solid modeling, Constructive solid geometry, and Boundary representation.

FreeCAD exposes the ability to define and manipulate the geometry of parts through it's Part module.

See the FreeCAD Wiki on Creating and manipulating geometry, and Topological data scripting for additional details.

## 15.1 Sub-packages

The following are typical sub-packages the library package may contain:

```
<library package>/
├── part/
├── model/
├── attachment/
└── __init__.py
```

**Note:** The library package typically only contains sub-packages without any direct modules.

## 15.2 Part Sub-package

The part sub-package exposes Part Classes encapsulating the geometry for parts, and is made up of further **private** sub-packages for each part.

For example, the part package in the ose-3d-printer-workbench contains the following:

```
<library package>/part/
├── _axis/
├── _extruder/
├── _frame/
├── _heated_bed/
└── __init__.py
```

The _axis/ package exposes an Axis class for "making" the geometry of an axis.

Similarly, the _extruder/ package exposes an Extruder class, _heated_bed/ exposes a HeatedBed class, and _frame/ exposes multiple classes related to a frame.

All the exposed part classes are imported within the __init__.py file, and declared **public** using __all__:

```
"""Parts for a 3D Printer."""
from ._axis import Axis
from ._extruder import Extruder
from ._frame import AngledBarFrame, AngleFrameConnector, CNCCutFrame
from ._heated_bed import HeatedBed

__all__ = [
    'AngleFrameConnector',
    'AngledBarFrame',
    'Axis',
    'CNCCutFrame',
    'Extruder',
    'HeatedBed'
]
```

**Tip:** It's best-practice to include docstring for all public packages.

For more information on part classes themselves, see Part Classes.

## 15.3 Model Sub-package

The model sub-package exposes Model Classes for making the *static* geometry of part classes **dynamic**.

For example, the model package in the ose-3d-printer-workbench contains the following:

```
<library package>/model
├── _axis/
├── _extruder/
├── _frame/
├── _heated_bed/
└── __init__.py
```

The _axis/ package exposes an AxisModel class for "making" the geometry of the Axis part class dynamic.

Similarly, the `_extruder/` package exposes an `ExtruderModel` class, `_heated_bed/` exposes a `HeatedBedModel` class, and `_frame/` exposes a `FrameModel` class.

All the exposed model classes are imported within the `__init__.py` file, and declared **public** using `__all__`:

```
"""Models for 3D Printer parts."""
from ._axis import AxisModel
from ._extruder import ExtruderModel
from ._frame import FrameModel
from ._heated_bed import HeatedBedModel

__all__ = [
    'AxisModel',
    'ExtruderModel',
    'FrameModel',
    'HeatedBedModel'
]
```

For more information on model classes themselves, see Model Classes.

## 15.4 Attachment Sub-package

The `attachment` sub-package exposes Attachment Functions that return keyword arguments to make one part appear "attached to" another.

For example, the `attachment` package in the `ose-3d-printer-workbench` contains the following:

```
<library package>/attachment
├── _get_axis_frame_attachment_kwargs/
├── _get_extruder_axis_attachment_kwargs/
├── _get_heated_bed_frame_axis_attachment_kwargs/
└── __init__.py
```

The `_get_axis_frame_attachment_kwargs/` package exposes an `_get_axis_frame_attachment_kwargs` function for "attaching" the axis to the frame.

Similarly, the `_get_extruder_axis_attachment_kwargs/` package exposes a `get_extruder_axis_attachment_kwargs` function, and `_get_heated_bed_frame_axis_attachment_kwargs/` exposes a `get_heated_bed_frame_axis_attachment_kwargs` function.

All the exposed attachment functions are imported within the `__init__.py` file, and declared **public** using `__all__`:

```
"""Attachment functions to make 3D Printer parts appear attached to each other."""
from ._get_axis_frame_attachment_kwargs import (
    get_axis_frame_attachment_kwargs, get_default_axis_creation_kwargs)
from ._get_extruder_axis_attachment_kwargs import \
    get_extruder_axis_attachment_kwargs
from ._get_heated_bed_frame_axis_attachment_kwargs import \
    get_heated_bed_frame_axis_attachment_kwargs

__all__ = [
    'get_axis_frame_attachment_kwargs',
    'get_default_axis_creation_kwargs',
    'get_extruder_axis_attachment_kwargs',
    'get_heated_bed_frame_axis_attachment_kwargs'
]
```

For more information on attachment functions themselves, see Attachment Functions.

CHAPTER

SIXTEEN

# PART CLASSES

**Motivation**

Encapsulate how the geometry of a part is made.

Parts are often thought about as real world objects, and therefore fit nicely into the paradigm of Object Oriented Programming (OOP) as **classes**.

Each part class has the single-responsibility to "make" the geometry for a given part.

For example, you might have a `Box` class with a `make` method that encapsulates and exposes how to create the geometry of a box.

```python
import Part


class Box:

    @staticmethod
    def make():
        box = Part.makeBox(10, 10, 10)
        return box
```

**Note:** Naming the method `make` is a convention inspired by FreeCAD's `make*` Part API.

While in this trivial example the `Box` class and `make` method don't provide much value, this abstraction offers a simple interface for "making" more complex and custom geometry.

For example, you may pass in the `length` and `width` into the `make` method as parameters for creating boxes of different sizes.

```python
class Box:

    @staticmethod
    def make(length, width):
        height = 10
        box = Part.makeBox(length, width, height)
        return box
```

We could have defined a `make_box` **function** instead, but why is the `class` approach preferable?

Imagine the box is a **sub-part** of a more complex part, and that *parent* part needs to know about the static `height` of `10` for the box.

With a quick refactor, the parent part can now access the `height` of the `Box` as a static property, and that information stays close to the construction of the box geometry, as opposed to being defined somewhere else in the program via constants or some other approach.

```python
class Box:

    height = 10

    @classmethod
    def make(cls, length, width):
        box = Part.makeBox(length, width, cls.height)
        return box
```

CHAPTER

SEVENTEEN

# MODEL CLASSES

**Motivation**

Encapsulate values and persistence of user-configurable properties for parts.

Model classes act as extensions to Part Classes, for when you want dynamic geometry, or parameteric properties the user can manipulate in FreeCAD's GUI within the Property Editor after the part is made.

For example, extending our `Box` part class to make the length and width editable by the user:

```
from example.app.part import Box


class BoxModel:

    def __init__(self, obj):
        self.Type = 'Box'

        obj.Proxy = self

        obj.addProperty('App::PropertyLength', 'Length',
                        'Dimensions', 'Box length').Length = 10.0
        obj.addProperty('App::PropertyLength', 'Width',
                        'Dimensions', 'Box width').Width = 10.0

    def execute(self, obj):
        obj.Shape = Box.make(obj.Length, obj.Width)
```

The constructor or `__init__` method initializes the parameteric properties, and the `execute` method handles the construction of the geometry.

Colloquially known in the FreeCAD community as FeaturePython Objects or Scripted Objects, we choose the name "model" as we believe the terms "feature python object" or "scripted object" are not accurate enough and are potentially misleading.

Additionally, model classes handle **serialization**, or saving and restoring data through `App::Property` objects. This is a similar role to what some frameworks call a **Data Transfer Object (DTO)**.

Model objects are saved in FreeCAD .FcStd files with Python's `json` module.

The `json` module transforms the model object into **JSON** (a string with a special format) for persisting the object to disk. Upon loading FreeCAD, the `json` module uses that string to recreate the original object, **provided it has access to the source code that created the object**.

**Important:** This means users need to have the workbench installed in order to open any FCStd files with model classes saved in them.

Another motivation for the separation of part classes from model classes is to keep the "shape" separate from the persistence of dynamic properties to disk.

Sometimes, you don't want to force users to install a workbench in order to open FCStd files with particular parts in them.

Therfore, converting a model to a "shape" using a part class can be useful.

**Hint:** The platform may standardize a `to_shape()` method on model classes for this purpose in the future.

For additional information, see the FreeCAD Wiki on FeaturePython Objects and Scripted Objects.

CHAPTER

# EIGHTEEN

# ATTACHMENT FUNCTIONS

**Motivation**

Make parts appear attached to each other.

Attachment functions return keyword arguments to make one part to appear "attached to" another.

Each attachment function is named following the pattern `get_<attachee part>_<attached to part>_attachment_kwargs` where:

- `<attachee part>` is the part being attached to another part
- and `<attached to part>` is the part getting attached to

Attachment functions returns a dictionary `{}`, or set of keyword arguments (a.k.a "kwargs") for the *attachee part*.

The keyword arguments typically describe parameters like placement and orientation the attachee part must be in to appear "attached to" the desired part.

## 18.1 An Example

Let's take a concrete example from the ose-3d-printer-workbench.

Consider attaching an axis to the frame in a D3D printer, and the get_axis_frame_attachment_kwargs attachment function.

```
def get_axis_frame_attachment_kwargs(frame,
                                     selected_frame_face,
                                     axis_orientation):
...
```

First, let's deconstruct the name.

The *attachee part* is the **axis**, and the part getting *attached-to* is the **frame**.

The first argument to the attachment function **is always** the *attached-to* part. In this case, the `frame`.

Other arguments will vary from attachment function to attachment function depending upon requirements, but might include selected faces, or other parts.

In this case, the second and third arguments are the face the user selected (`selected_frame_face`), and the selected orientation of the axis (`axis_orienation`).

Consider the dictionary, or axis kwargs, `get_axis_frame_attachment_kwargs` returns when attaching the axis to the **front face** of the frame:

```
{
    "carriage_position": 90,
    "placement": Placement [Pos=(152.4,0,0), Yaw-Pitch-Roll=(0,0,0)],
    "orientation": "z",
    "origin_translation_offset": Vector (-0.5, -1.0, 0.0),
    "length": "304.8 mm",
    "side": "front"
}
```

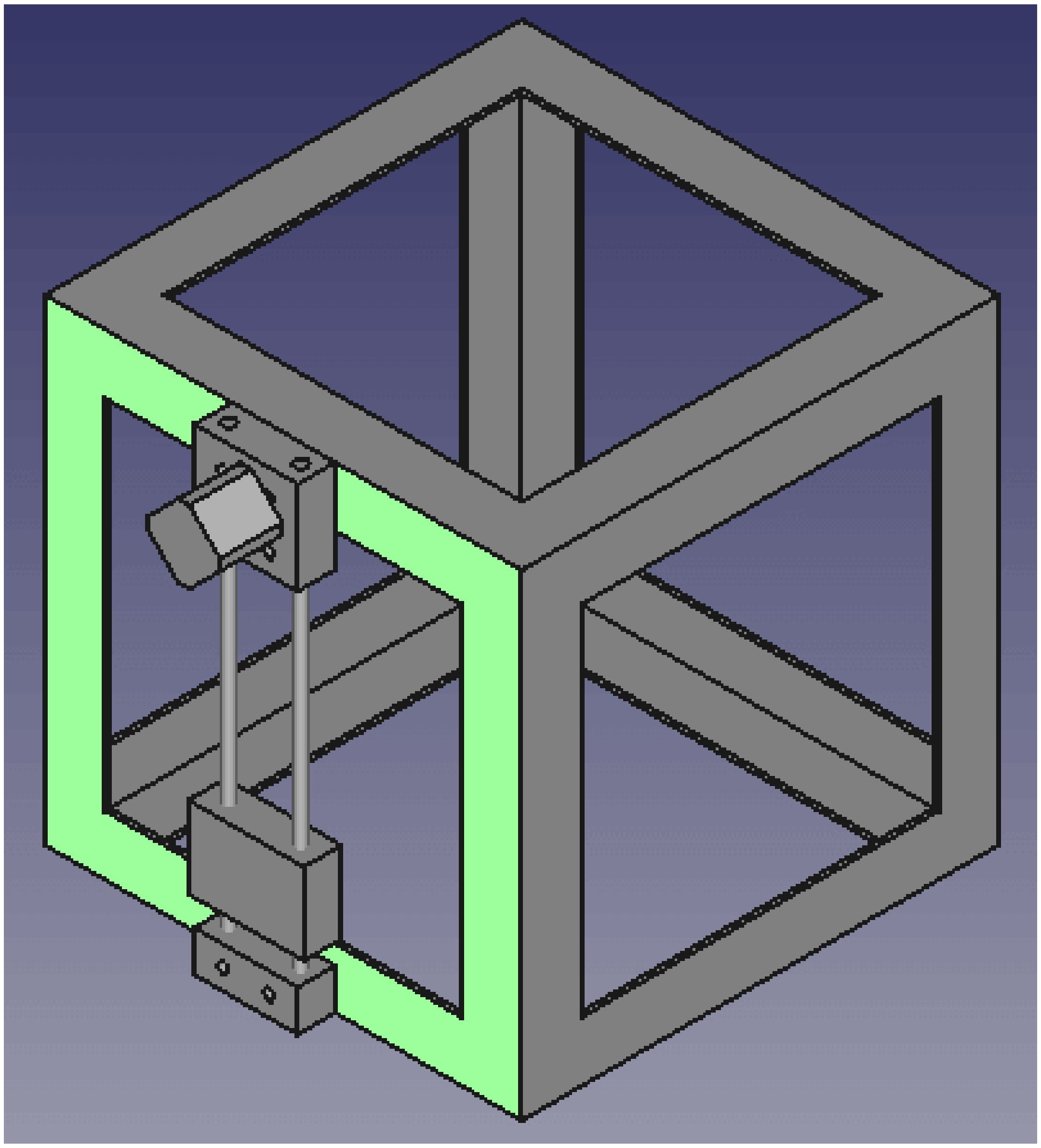


Fig. 1: Attaching axis to front face of frame

These keyword arguments describe how to make the axis geometry appear attached to the desired position on the frame.

CHAPTER

# NINETEEN

# WORKBENCH PACKAGE

**Motivation**

Organize code related to the graphical representation of parts and a workbench.

The workbench package, located within the `freecad/` directory, contains code related to the graphical user interface of FreeCAD, such as what happens when users interact with the workbench (e.g. a user clicks a button on a toolbar), or various components the user may interact with such as dialogs or panels.

```
freecad/<workbench package>/
├── command/
├── icon/
├── part_feature/
├── __init__.py
├── init_gui.py
└── OSE_<Machine_Name>.py
```

## 19.1 init_gui.py

Every workbench will have a `init_gui.py` module within the workbench package.

The `init_gui.py` module contains a *single* **workbench class** that extends `Gui.Workbench` following the pattern `<Machine>Workbench`, where `<Mahcine>` is the name of the machine in **pascal** or **UpperCamelCase**.

For example, the **workbench class** for OSE's Tractor Workbench will be located inside the `init_gui.py` module and named `TractorWorkbench`:

```python
import FreeCAD as App
import FreeCADGui as Gui

from .icon import get_icon_path
from .OSE_Tractor import register_commands


class TractorWorkbench(Gui.Workbench):
    """
    Tractor Workbench
    """
    MenuText = 'OSE Tractor'
    ToolTip = \
        'A workbench for designing Tractor machines by Open Source Ecology'
```

(continues on next page)

(continued from previous page)

```
    Icon = get_icon_path('Tractor.svg')

    def Initialize(self):
        """
        Executed when FreeCAD starts
        """
        main_toolbar, main_menu = register_commands()

        self.appendToolbar('OSE Tractor', main_toolbar)
        self.appendMenu('OSE Tractor', main_menu)

    def Activated(self):
        """
        Executed when workbench is activated.
        """
        if not(App.ActiveDocument):
            App.newDocument()

    def Deactivated(self):
        """
        Executed when workbench is deactivated.
        """
        pass

    def GetClassName(self):
        return 'Gui::PythonWorkbench'


Gui.addWorkbench(TractorWorkbench())
```

**Important:** FreeCAD imports this module when it initializes it's GUI. The last statement in `init_gui.py` instantiates the **workbench class** and adds it to FreeCAD via `Gui.addWorkbench`.

For a complete reference of the `Gui.Workbench` class, see Gui::PythonWorkbench Class Reference.

## 19.2 Command Sub-package

The `command` sub-package exposes Command Classes that are executed when users perform various actions in the workbench such as clicking a button in a toolbar or selecting an option in a menu.

For example, the `command` package in the `ose-3d-printer-workbench` contains the following:

```
freecad/<workbench package>/command
├── _add_axis/
├── _add_extruder/
├── _add_frame/
├── _add_heated_bed/
└── __init__.py
```

The `_add_axis/` package exposes an `AddAxisCommand` that's executed when the user wants to add an axis to the document.

Similarly, the `_add_extruder/` package exposes an `AddExtruderCommand` class, `_add_frame/` exposes `AddFrameCommand`, and `_heated_bed/` exposes `AddHeatedBed`.

For more information on command classes themselves, see Command Classes.

## 19.3 Command Registry Module

Every workbench contains a **command registry module** within the workbench package.

The command registry module is where all commands are imported, registered via `Gui.addCommand`, and associated together into lists for adding to toolbars or menus.

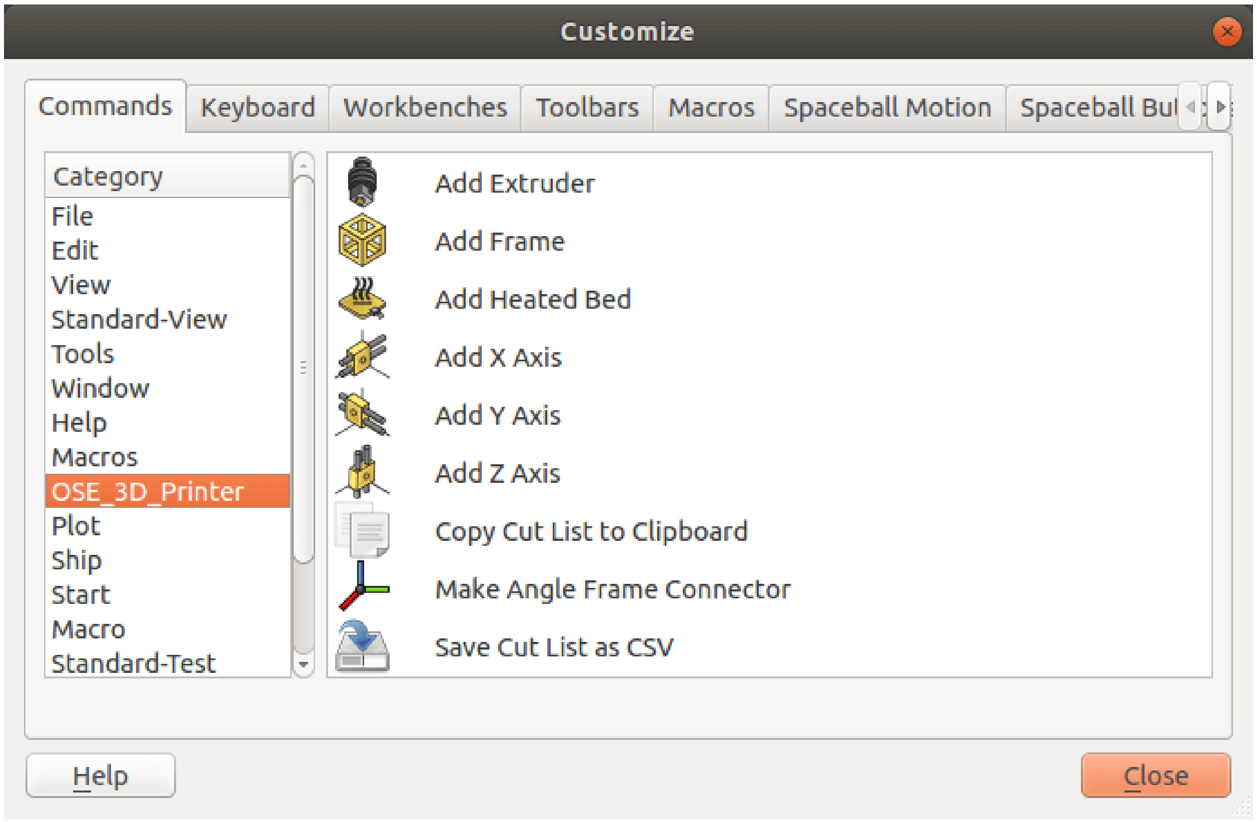
The command registry module name follows the pattern `OSE_<Machine>.py`, where `<Machine>` is the name of the machine, with spaces delimited by underscores _.

For example, the command registry module name for the 3D Printer workbench is named `OSE_3D_Printer.py`.

Normally python modules use all lower-case letters, so why the deviation?

FreeCAD derives a "Category" to organize commands from the name of the Python module where `Gui.addCommand` is called.

Since all commands in the workbench are registered with `Gui.addCommand` in a Python module called `OSE_3D_Printer.py`, the derived "Category" for grouping these commands is "OSE_3D_Printer".



When you register custom commands for an external workbench via `Gui.addCommand(commandName, commandObject)`, FreeCAD adds the command to it's global command registry.

To avoid name collisions and ensure uniqueness, a command name is typically prefixed with the name of the module and underscore. For example, "Part_Cylinder" or "OSE3DP_AddFrame".

The command registry module handles prefixing a unique namespace to the name of your command for you.

In this way, if in the future we need to change the name of our command namespace (e.g. “OSE3DP”) because it collides with another external workbench, then the change is easy.

You can see a simple and relatively complete command registry module example based on the `ose-3d-printer-workbench` below:

```python
import FreeCADGui as Gui

from .command import AddExtruderCommand, AddFrameCommand, AddHeatedBedCommand

#: Command Namespace: Must be unique to all FreeCAD workbenches.
command_namespace = 'OSE3DP'


def register_commands():
    """
    Register all workbench commands,
    and associate them to toolbars, menus, sub-menus, and context menu.
    """
    add_frame_key = _register(AddFrameCommand.NAME, AddFrameCommand())
    add_heated_bed_key = _register(
        AddHeatedBedCommand.NAME, AddHeatedBedCommand())
    add_extruder_key = _register(AddExtruderCommand.NAME, AddExtruderCommand())

    #: Main Toolbar Commands
    main_toolbar_commands = [
        add_frame_key,
        add_heated_bed_key,
        add_extruder_key
    ]
    return main_toolbar_commands


def _register(name, command):
    key = '{}_{}'.format(command_namespace, name)
    Gui.addCommand(key, command)
    return key

__all__ = ['register_commands']
```

## 19.4 Icon Sub-package

The `icon` sub-package contains icons for the workbench (typically in `.svg` format) and exposes a `get_icon_path` function that takes the name of an icon file and returns the absolute path to the icon.

```python
from .icon import get_icon_path

get_icon_path('MyIcon.svg') # => /home/user/.FreeCAD/Mod/my-workbench/myworkbench/gui/
↪icon/MyIcon.svg
```

## 19.5 Part Feature Sub-package

The `part_feature` sub-package exposes functions to create Part Feature objects.

For example, the `part_feature` package in the `ose-3d-printer-workbench` contains the following:

```
freecad/<workbench package>/part_feature
├── _axis/
├── _extruder/
├── _frame/
├── _heated_bed/
└── __init__.py
```

The `_axis/` package exposes a `create_axis` function that creates and adds an axis part feature object to a specified document.

Similarly, the `_extruder/` package exposes a `create_extruder` function, `_frame/` exposes `create_frame`, and `_heated_bed/` exposes `create_heated_bed`.

A simple example of a part feature creation function looks like:

```python
from ose3dprinter.app.model import AxisModel


def create_axis(document, name):
    """
    Creates a axis object with the given name,
    and adds it to a document.
    """
    obj = document.addObject('Part::FeaturePython', name)
    AxisModel(obj)
    obj.ViewObject.Proxy = 0  # Mandatory unless ViewProvider is coded
    return obj
```

The single responsibility of a part feature creation function is to add a `Part::FeaturePython` to a document, and decorate it with a model class, and *optionally* a view provider.

CHAPTER

TWENTY

# COMMAND CLASSES

**Motivation**

Encapsulate action users can perform when interacting with FreeCAD's UI.

Command Classes are executed when users perform various actions in the workbench such as clicking a button in a toolbar or selecting an option in a menu.

OSE Workbench Command Classes are an opinionated extension to FreeCAD Command Classes with the following observed conventions:

1. Names sound like actions, typically begin with verbs, and always end with a "command" suffix
    - For example, a command class to add a frame to the document might be named `AddFrameCommand`
    - The command should be located in a module named after the command (e.g. `add_frame_command.py`)
2. Have a static `NAME` **string** constant
    - Typically the same name as the command (e.g. `'AddFrameCommand'`)

**Important:** `NAME` must be unique for all commands within the scope of the workbench

3. Located and exposed by the `command` sub-package of the workbench package.

```
freecad/ose3dprinter/command
├── _add_frame/
│   ├── add_frame_command.py
│   └── __init__.py
└── __init__.py
```

Within `_add_frame/__init__.py`:

```
from .add_frame_command import AddFrameCommand

__all__ = ['AddFrameCommand']
```

Within `freecad/ose3dprinter/command/__init__.py`:

```
"""Commands users can perform in FreeCAD's GUI."""
from ._add_frame import AddFrameCommand

__all__ = ['AddFrameCommand',]
```

The following is a complete example taken from the ose-3d-printer-workbench:

```python
import FreeCAD as App

from freecad.ose3dprinter.icon import get_icon_path
from freecad.ose3dprinter.part_feature import create_frame


class AddFrameCommand:
    """
    Command to add Frame object.
    """

    NAME = 'AddFrame'

    def Activated(self):
        document = App.ActiveDocument
        if not document:
            document = App.newDocument()
        create_frame(document, 'Frame')
        document.recompute()

    def IsActive(self):
        return True

    def GetResources(self):
        return {
            'Pixmap': get_icon_path('Frame.svg'),
            'MenuText': 'Add Frame',
            'ToolTip': 'Add Frame'
        }
```

For additional information, see Command on the FreeCAD Wiki.

# CHAPTER TWENTYONE

# OSE WORKBENCH ECOSYSTEM

This document aims to catalog the **OSE Workbench Ecosystem** – or collection of projects relating to OSE Workbenches.

In general, there are OSE workbenches, and OSE workbench libraries.

OSE workbenches have a FreeCAD frontend, while OSE workbench libraries are a collection of code to be leveraged by OSE workbenches.

## 21.1 OSE Workbenches

| Logo | Name | Documentation |
|---|---|---|
| | 3D Printer | https://ose-3d-printer-workbench.readthedocs.io/en/latest/ |

## 21.2 OSE Workbench Libraries

| Logo | Name | Documentation |
|---|---|---|
| | Workbench Core | https://ose-workbench-core.readthedocs.io/en/latest/ |

## 21.3 UML Diagram

The below UML diagram shows the relationships between different components in the OSE Workbench Ecosystem.

The **dashed** lines represent dependencies.

# CHAPTER TWENTYTWO

# OSEWB

## 22.1 osewb.docs

### 22.1.1 osewb.docs.ext

Custom Sphinx Extensions for OSE Workbenches.

These extensions designed to be externalized into separate projects if others are interested in using them.

#### all_summary_table

| Name | Description |
|---|---|
| *setup* | Setup extension. |

---

**setup** (*app: sphinx.application.Sphinx*) → None
Setup extension.

**Parameters** **app** – application object controlling high-level functionality, such as the setup of extensions, event dispatching, and logging. See Also: https://www.sphinx-doc.org/en/master/extdev/appapi.html#sphinx.application.Sphinx

#### freecad_custom_property_table

Adds a `.. fc-custom-property-table::` directive to create a table documenting the properties of custom FreeCAD objects.

Must add the `.. fc-custom-property-table::` directive to the docstring of a scripted object class:

```
class BoxModel:
    """
    .. fc-custom-property-table::
    """
```

Or pass the path to the class as an **optional** argument:

```
.. fc-custom-property-table:: examples.box_model.BoxModel
```

Supports a `remove_app_property_prefix_from_type` configuration value to remove the `App::Property` prefix from the **Type**. Defaults to `False`.

These objects are also known as "FeaturePython Objects" or "Scripted Objects" in the FreeCAD community.

See BoxModel for an example.

**class FreeCADCustomPropertyTable**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Bases: `sphinx.util.docutils.SphinxDirective`

**has_content = False**

**optional_arguments = 1**

**run**()

**create_table_row**(*row_cells*)

**setup**(*app: <module 'FreeCAD' from '/home/docs/checkouts/readthedocs.org/user_builds/ose-workbench-platform/conda/latest/lib/FreeCAD.so'>*) → None

Setup extension.

**Parameters** **app** – application object controlling high-level functionality, such as the setup of extensions, event dispatching, and logging. See Also: https://www.sphinx-doc.org/en/master/extdev/appapi.html#sphinx.application.Sphinx

## freecad_icon

**fcicon_role**(*name: str*, *rawtext: str*, *text: str*, *lineno: int*, *inliner: docutils.parsers.rst.states.Inliner*, *options: dict = {}*, *content: List[str] = []*)

FreeCAD Icon role function.

Returns 2 part tuple containing list of nodes to insert into the document and a list of system messages. Both are allowed to be empty.

**For additional information on role functions, see:**

- https://docutils.readthedocs.io/en/sphinx-docs/howto/rst-roles.html
- https://doughellmann.com/blog/2010/05/09/defining-custom-roles-in-sphinx/

**Parameters**

- **name** – The role name used in the document.
- **rawtext** – The entire markup snippet, with role.
- **text** – The text marked with the role.
- **lineno** – The line number where rawtext appears in the input.
- **inliner** – The inliner instance that called us.
- **options** – Directive options for customization.
- **content** – The directive content for customization.

**make_image_node**(*freecad_icon_directory: str*, *alt: str*, *size: str*, *filename: str*) → docutils.nodes.image

Make image node for icon.

**Parameters**

- **freecad_icon_directory** – Directory to FreeCAD Icons.

- **`alt`** – Alt text of icon.
- **`size`** – Must be one of “sm” (small), “md”, (medium), or “lg” (large).
- **`filename`** – Filename of icon.

**`setup`**(*app: sphinx.application.Sphinx*) → None
Setup extension.

**Parameters** **`app`** – application object controlling high-level functionality, such as the setup of extensions, event dispatching, and logging. See Also: https://www.sphinx-doc.org/en/master/extdev/appapi.html#sphinx.application.Sphinx

#### osewb_docstring_process

| Name | Description |
|---|---|
| `setup` | Setup extension. |

**`setup`**(*app: sphinx.application.Sphinx*) → None
Setup extension.

**Parameters** **`app`** – application object controlling high-level functionality, such as the setup of extensions, event dispatching, and logging. See Also: https://www.sphinx-doc.org/en/master/extdev/appapi.html#sphinx.application.Sphinx

### 22.1.2 conf

Shared base configuration for OSE workbench documentation.

## 22.2 check_for_executable_in_path

**`check_for_executable_in_path`**(*executable_name*) → None

## 22.3 execute_command

**`chain_commands`**(**commands: str, env: dict = {}, exit_on_non_zero_code: bool = True*) → List[int]

**`execute_command`**(*command: str, env: dict = {}, exit_on_non_zero_code: bool = True*) → int

## 22.4 find_base_package

**`exec_git_command`**(*git_command: str*) → Optional[str]
Find the root of the current git repository.

Returns None if there’s an error, or not in a git repository.

**Parameters** **`git_command`** – git command string

**Returns** path to root of git repository

**find_base_package**() → Optional[str]
Find the base package in a workbench repository.

Return None if not in a git repository, or no directory in the root of the repository starts with "ose".

If multiple directories in the root of the repository start with "ose", then we return the first match.

**Returns** Base package of workbench repository.

**find_git_user_name**() → Optional[str]
Find the user name defined by git config.

**Returns** Git user name

**find_root_of_git_repository**() → Optional[str]
Find the root of the current git repository.

Returns None if there's an error, or not in a git repository.

**Returns** path to root of git repository

## 22.5 handle_browse_command

**handle_browse_command**(*root_of_git_repository: str*, *browse_subcommand: str*) → None

## 22.6 handle_build_command

**handle_build_command**() → None

## 22.7 handle_docs_command

**handle_docs_command**(*base_package*, *root_of_git_repository*) → None

**remove_existing_directory_recursively**(*directory: str*) → None

## 22.8 handle_editor_config_command

**get_editor_config**() → dict

**get_vs_code_workspace_settings**(*root_of_git_repository: str*) → Optional[dict]

**get_vs_code_workspace_settings_path**(*root_of_git_repository: str*) → pathlib.Path
Get the path to VS Code workspace settings.

**See Also:** https://code.visualstudio.com/docs/getstarted/settings#_settings-file-locations

**handle_editor_config_command**(*root_of_git_repository: str*, *merge_workspace_settings: bool = False*, *overwrite_workspace_settings: bool = False*) → None

**path**(*filename: str*) → str

**query_yes_no**(*question*, *additional_pre_information: Optional[str] = None*, *default: str = 'yes'*)
Ask a yes/no question via raw_input() and return their answer.

"question" is a string that is presented to the user. "default" is the presumed answer if the user just hits <Enter>. It must be "yes" (the default), "no" or None (meaning an answer is required of the user).

The "answer" return value is True for "yes" or False for "no".

## 22.9 handle_lint_command

**`handle_lint_command`**(*root_of_git_repository: str*, *should_fix: bool = False*) → None

## 22.10 handle_make_component_command

## 22.11 handle_make_workbench_command

## 22.12 handle_test_command

**`find_coverage_package`**(*root_of_git_repository: str*, *base_package: str*) → str

**`handle_test_command`**(*base_package: str*, *root_of_git_repository: str*, *with_coverage: bool = False*) → None

**`is_workbench`**(*root_of_git_repository: str*) → bool

## 22.13 osewb

## 22.14 part_screenshot

Utility script to automatically create thumbnail screenshots of parts.

Run with freecad -c part_screenshot.py when conda environment is activated.

**`format_list`**(*l*)

**`get_required_arguments`**(*make_method*) → List[str]

**`main`**()

CHAPTER

# TWENTYTHREE

# EXAMPLES

This example package is meant to test and exemplify OSE Workbench Platform Sphinx extensions found in *`osewb.docs.ext`*.

## 23.1 examples.model

Example model package.

| Name | Description |
|---|---|
| *`BoxModel`* | Encapsulates the data (i.e. topography and shape) for a Box, |

---

**`class BoxModel`**(*obj*)

Bases: `object`

Encapsulates the data (i.e. topography and shape) for a Box, and is separate from the "view" or GUI representation.

| Name | Type | Default Value | Description |
|---|---|---|---|
| **Height** | `App::PropertyLength` | 10.0 mm | Box height |
| **Length** | `App::PropertyLength` | 10.0 mm | Box length |
| **Width** | `App::PropertyLength` | 10.0 mm | Box width |

**`execute`**(*obj*)

## 23.2 examples.part

Example part package.

| Name | Description |
|---|---|
| *`Box`* | Represents a box. |

---

**`class Box`**

Bases: `object`

Represents a box.

**static make**() → Part.Shape

Make a box.

**Returns** The shape of a box.

**Return type** Part.Shape

# CHAPTER TWENTYFOUR

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

# INDEX