PBK

Author: Eugene A. Fedotov, Matthew Douglass, Jovaughn Chin
Version: 0.1

The PBK package aims to be a Portable Kit for open-source Books. It is meant to provide a diverse selection of common graphical diagrams in select subjects. Currently, the focus is on compilers and programming in Java.

Dependencies

The following libraries are required to run PBK:

- PGF 3.0
- PGF-TiKz
- DraTeX, AlDraTeX

Libraries

Agenda **(subject to change)**

0.1 Sprint 1 : 3/9 - 3/13

1. Decide on a set of diagrams for each person.
2. Update design doc with a diagram example from the set that illustrates the output look.
3. Consider what possible options can be implemented.
4. Create a function, customized by user input, to generate diagram(s).

0.2 Sprint 2 : 3/16 - 3/20

1. Repeat Sprint 1, with next set of diagrams(CS discipline).
2. Update doc

0.3 Sprint 3 : 3/23 - 3/27

1. Repeat Sprint 1, with next set of diagrams from other disciplines(bio,physics,math..)
2. Update doc

0.4 Sprint 3 : 3/30 - 4/3

1. Work on special features mentioned in spec.
2. define function for color interchangeability
3. define function for pdf content links(bookmarking)

0.5 Sprint 4 : 4/6 - 4/9

1. Test, upload the functions to the package, then CTAN.
2. Work on demo
3. Work on Symposium poster

Diagrams **(subject to change)**

Class Diagram

Free Body Diagrams

Collections: Linked Lists, Array, and Maps/Sets

1 User Input

Users will be able to create diagrams by editing a section near the top of the latex code. This section will be clearly defined, easily editable and understandable. Below is a work in progress of some latex code that the user will have to edit to create a diagram that fits their specs. The example below is specifically used when making array diagrams and is far from final.

Declare Class Diagram:

```
\classdiagram{ {List of key:value sets } }
```

Parameter Format:

```
class:g=class,class,class,...,class:a=class,class,class,..., ...
```

"class:g" defines a class with a generalization relationship(:g) to one or more other classes(=class,class,class,...).

Relationships:

:g - Generalization: A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, the specific classifier inherits the features of the more general classifier.

:c - Composition(Composite Association): An association may represent a composite aggregation (i.e., a whole/part relationship). Only binary associations can be aggregations. Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted with it. Note that a part can (where allowed) be removed from a composite before the composite is deleted, and thus not be deleted as part of the composite. Compositions may be linked in a directed acyclic graph with transitive deletion characteristics; that is, deleting an element in one part of the graph will also result in the deletion of all elements of the subgraph below that element. Composition is represented by the isComposite attribute on the part end of the association being set to true.

:d - Dependency: A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation. This means that the complete semantics of the depending elements is either semantically or structurally dependent on the definition of the supplier element(s).

:di - Instantiation: A usage dependency among classifiers indicating that operations on the client create instances of the supplier.

:da - Abstraction: An abstraction is a relationship that relates two elements or sets of elements that represent the same concept at different levels of abstraction or from different viewpoints. In the metamodel, an Abstraction is a Dependency in which there is a mapping between the supplier and the client.

:dp - Permission: Permission is a kind of dependency. It grants a model element permission to access elements in another namespace.

:r - Realization: Realization is a specialized abstraction relationship between two sets of model elements, one representing a specification (the supplier) and the other represents an implementation of the latter (the client). Realization can be used to model stepwise refinement, optimizations, transformations, templates, model synthesis, framework composition, etc.

:dr - Refine: Specifies a refinement relationship between model elements at different semantic levels, such as analysis and design. The mapping specifies the relationship between the two elements or sets of elements. The mapping may or may not be computable, and it may be unidirectional or bidirectional. Refinement can be used to model transformations from analysis to design and other such changes.

:s - Substition: A substitution is a relationship between two classifiers which signifies that the substitutingClassifier complies with the contract specified by the contract classifier. This implies that instances of the substitutingClassifier are runtime substitutable where instances of the contract classifier are expected.

:t - Trace: Specifies a trace relationship between model elements or sets of model elements that represent the same concept in different models. Traces are mainly used for tracking requirements and changes across models. Since model changes can occur in both directions, the directionality of the dependency can often be ignored. The mapping specifies the relationship between the two, but it is rarely computable and is usually informal.

:u - Usage: A usage is a relationship in which one element requires another element (or set of elements) for its full implementation or operation. In the metamodel, a Usage is a Dependency in which the client requires the presence of the supplier.

:m - Merge: A package merge is a directed relationship between two packages that indicates that the contents of the two packages are to be combined. It is very similar to Generalization in the sense that the source element conceptually adds the characteristics of the target element to its own characteristics resulting in an element that combines the characteristics of both.
This mechanism should be used when elements defined in different packages have the same name and are intended to represent the same concept. Most often it is used to provide different definitions of a given concept for different purposes, starting from a common base definition. A given base concept is extended in increments, with each increment defined in a separate merged package. By selecting which increments to merge, it is possible to obtain a custom definition of a concept for a specific end. Package merge is particularly useful in meta-modeling and is extensively used in the definition of the UML metamodel.
Conceptually, a package merge can be viewed as an operation that takes the contents of two packages and produces a new package that combines the contents of the packages involved in the merge. In terms of model semantics, there is no difference between a model with explicit package merges, and a model in which all the merges have been performed.

:i - Import: A package import is defined as a directed relationship that identifies a package whose members are to be imported by a namespace.

:dd - Derive: Specifies a derivation relationship among model elements that are usually, but not necessarily, of the same type. A derived dependency specifies that the client may be computed from the supplier. The mapping specifies the computation. The client may be implemented for design reasons, such as efficiency, even though it is logically redundant.

:b - Binding: TemplateBinding is a directed relationship from a bound templateable element to the template signature of the target template. A TemplateBinding owns a set of template parameter substitutions.

:a - Aggregation: A kind of association that has one of its end marked shared as kind of aggregation, meaning that it has a shared aggregation.

Declare a Class Object:

```
\oclass[content]{Name}
```

Declare an Array:

```
\array{List of values}
```

List of Values: Value of each index of the array separated by commas.(Value can reference to objects)

Format of parameter:

```
{adam, jack, joe} or {\oclass[content]{Name},\oclass[content]{Name},
.....}.
```

If List of values is empty,a minimal Array with default non-values will be produced.

Declare a Singly LinkedList:

```
\slinklist{List of values}
```

List of Values: Value of each index of the Linked list separated by commas. (Value can reference to objects).

Format of parameter:

```
{adam, jack, joe} or {\oclass[content]{Name},\oclass[content]{Name},
.....}.
```

If List of values is empty, a minimal Linked List with default non-values will be produced.

Declare a Doubly LinkedList:

```
\dlinklist{List of values}
```

List of Values: Value of each index of the Linked list separated by commas. (Value can reference to objects)

Format of parameter:

```
{adam, jack, joe} or {\oclass[content]{Name},\oclass[content]{Name},
.....}.
```

If List of values is empty, a minimal Linked List with default non-values will be produced.

Declare a Map/Set:

```
\mapset{key}{value}
```

Key: key of object
Value: value of object

Declare a Hashmap/Hashset:

```
\hashmapset{list of key:value pairs}
```

Parameter Format:

```
key:value,key:value,....
```

Key: key of object
Value: value of object

Change History