Chapter 3

Planning the project

3.1 Implementing PT as a TS library

In order to implement PT we need to be able to handle the following:

- Defining templates
- Renaming classes and class attributes
- Instantiating templates
- Merging classes

3.1.1 Defining templates

For defining templates we would like a construct that can wrap our template classes in a scope. We will also need to be able to reference the template. ECMAScript has three options for this, an array, an object or a class. It should however also be possible to inherit from classes in your own template, which pretty much rules out both arrays and objects, as there is no way of referencing other members during definition of the array/object. Templates could there be defined as ECMAScript classes, where each member of the template is a static attribute of the template. In listing 1 on the following page we see an example of how this could be done.

We are making the templates classes static in order to be able to rename them, see section 3.1.2.

3.1.2 Renaming classes and class attributes

Renaming of classes is possible to an extent. Since we made the classes static attributes of the template class we could easily just create a new static field on the template class (with decleration merging to get the correct type for the new name) and `delete`[3] the old field. We can see an example of this in listing 2 on the following page.

Even though we were able to give the class a "new name", this would still not actually rename the class. Any reference to the old names would be left unchanged, and thus we are not able to achieve renaming in TypeScript.

```
class T1 {
  static A = class {
    i = 1;
  };

  static B = class extends T1.A {
    b = 2;
  };
}
```

Listing 1: Example of defining a template

```
class T1 {
  static A = class {
    i = 0;
  };
  static B: any;
}

interface T1 {
  B: typeof T1.A;
}

const classRef = T1.A;
delete T1.A;
T1.B = classRef;
```

Listing 2: Example of renaming a template class

```
class T2 {
  static A = class {};
}

const P = class {};
for (let attr of Object.keys(T2)) {
  P[attr] = T2[attr];
}
```

Listing 3: Example of instantiating a template

3.1.3 Instantiating templates

As with renaming, we are also able to instantiate templates to an extent. We are able to iterate over the attributes of the template class, and populate a package/template with references to the template. An example of this can be seen in listing 3.

The instantiation will only contain references to the instantiated templates classes, while PT instantiations make textual copies of the templates content. Only having references to the original template could mean that if a template that has been instantiated is later renamed, then the instantiated template might lose some of its references.

3.1.4 Merging classes

For merging of types you would use the built-in decleration merging[7]. Implementation merging is also possible because ECMAScript has open classes. For implementation merging you would create an empty class which has the type of the merged declerations, and then assign the fields and methods from the merging classes to this class. There are several libraries that supports class merging, such as mixin-js[5].

3.1.5 Conclusion

Since we are not able to support renaming and instantiations we can conclude that Package Templates can not be implemented as a library in TypeScript, we will need to implement this as part of a compiler.

3.2 What Do We Need?

- The ability to add custom syntax (access to the tokenizer / parser)
- Some semantic analysis.

3.3 Syntax

For the implementation of PT we need syntax for the following: