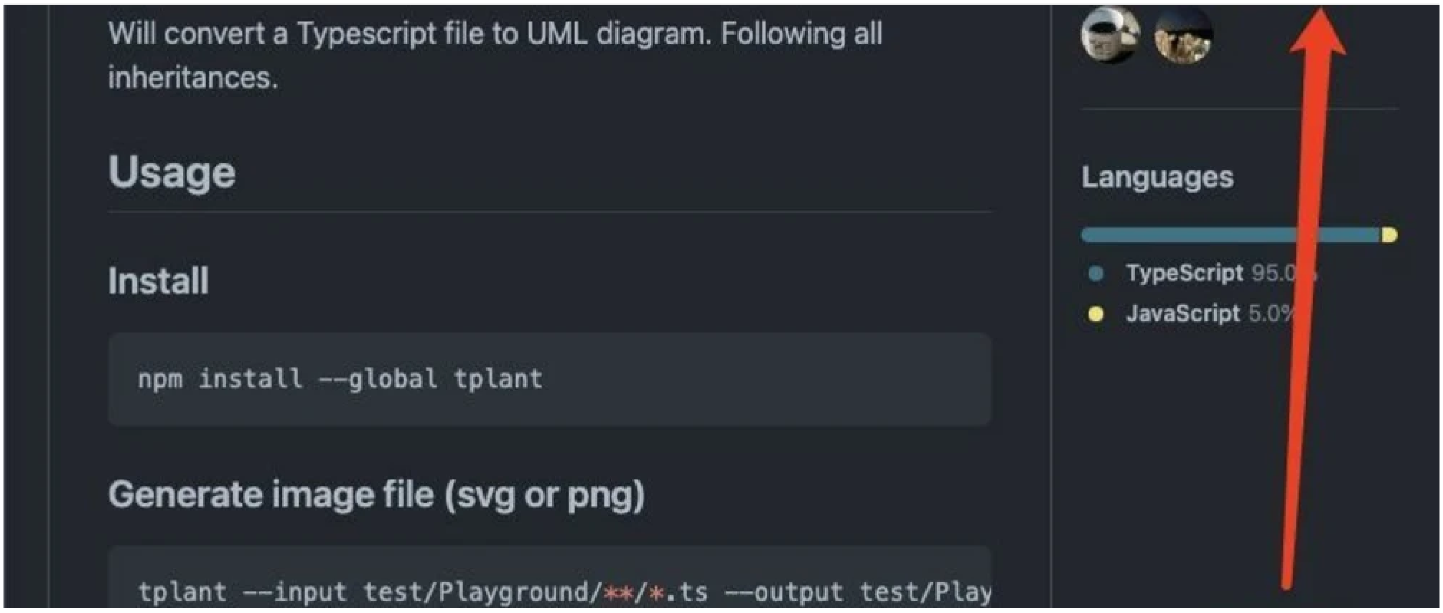

Will convert a Typescript file to UML diagram. Following all inheritances.

Usage

Install

```
npm install --global tplant
```

Generate image file (svg or png)

```
tplant --input test/Playground/**/*.ts --output test/Play
```

Jeff
Posted on 22 de jul. de 2021

Use TypeScript to make DDD come true

#typescript #tplant #ddd #uml

DDD

Domain Driven Design is popular and even became a de facto standard for enterprise technology team, however, it's only showcases, not the real implementation. How many times that you see the fancy design diagrams and at them same the poor code?

Many people prefers design first, what implies that code is the implementation, and only the slide shows are design. But actually, code is design.

Code = Design = Model = Documents

If not the case, then everytime you change the code you need to update the UML class diagrams and database E-R design documents. And eventually these kind of documents becomes unreliable gradually because you can't make sure that they are updated in time.

In fact, code is more suitable for expressing the designs, and source code is a document indeed, can be used to describe the current product's design decisions perfectly.

If developer created a domain model by code which is consistent with what in a domain expert's brain, then the source code is the most efficient, realtime model no doubtedly.

The limit of the equation of `Code = Deisgn = Model = Documents` is whether the domain expert can read the code. So the easy to learn, expressive and intuitive programming language will have strong benefits in the process of creating domain models.

Domain Modelling

Domain modelling is the most important part in DDD for developers as it requires developers have good abstraction, and it differs from traditional database modelling, and need developers to map the domain knowledge into the code models through the most efficient programming technic.

In the long run, the object oriented language is the first choice of domain modelling, some OO skills can be used to do the domain model abstraction. In the contrary, the functional programming langauges are typically thought to be suitable only for data processing, scientific computing, etc.

TypeScript

But this article shows that TypeScript, which has many functional programming features (
TypeScript's type system fully meets the functional programming requirements), can be used to do the domain modelling, and thanks to its type system and related tool chains, TypeScript should be considered as the best language to make DDD landing.

TypeScript's type system

Comparing to OO, you only need to know a few grammer and it's enough to start domain modelling, so in terms of simplicity, algebraic data type is more suitable for domain modelling so as to make the domain models be documents.

Type

All kinds of programming languages provide primitive types by design, such as `string`, `bool`, `number`, etc.

In TypeScript, you can use the keyword `type` to compose larger types:

```
type Name = {
  firstName: string
  middleName: string
  lastName: string
}
```

The usage of the above is obviouse, and besides this kind of usage, the keyword `type` has other usages, which is not a trivial feature. It can help you record the domain knowledge into domain models, for example:

```
const timeToFly = 10
```

you can not guess the domain knowledge at first glance at the above code. How to make sure what the 10 means? Look up in a document? No, you need to tell yourself that code is document, so you improve your code as follows:

```
type Second = number
const timeToFly: Second = 10
```

Type Or

In TypeScript, they are called as Union Types, which can be built by the symbol `|`:

```
type Pet = Fish | Bird
```

So `Pet` is in type `Fish` or `Bird`. In general, functional programming languages have strong pattern match capability to process this kind of type. But the sad thing is TypeScript has limited pattern match capability so you can often see some string literals present in the types to distinguish different types.

Type And

In TypeScript, they are called as Intersection Types, which can be built by the symbol `&`:

```
type ABC = A & B & C
```

The above code tries to say that type ABC contains all A, B and C's properties.

Define Function Types

In TypeScript, there were no differences between function and other types, so you can define functions by using the keyword `type`:

```
type Add = (a: number) => (b: number) => number
```

The above code shows that `Add` is a function who accepts 2 numbers as arguments and returns a number.

Using code to share domain knowledge

```
type CreditCard = {
  cardNo: string
  firstName: string
  middleName: string
  lastName: string
  contactEmail: Email
  contactPhone: Phone
}
```

Notice that we can easily write the above code by just having the knowledge showed previously, to describe the `CreditCard` payment method. Also please notice we don't use `class` here.

But is it a reliable domain model? If not, where is the problem?

The most serious problem of the above code is that it didn't record the domain knowledge which should be owned by it inside of it. Let me ask some questions:

Question: can `middle name` be empty?
Answer 1: Not sure, need to check document.
Answer 2: Maybe? `middle name` can be null.

Modelling for the nullable type

In functional programming languages, the nullable types can be defined as Optional. Although null is valid in TypeScript (Note: we can enable `strictNullChecks` to enforece the null check), but in functional programming, you can only use Optional type to express nullable type.

If the domain expert tells you that `middle name` can exists, or be empty. Plese notice the word "or", indicate that we can use Union Type to model for the nullable type:

```
type Optional<T> = T | null
```

A simple Optional is just a Type Or. The improved code looks as follows:

```
type CreditCard = {
  cardNo: string
  firstName: string
  middleName: Option<string>
  lastName: string
  contactEmail: Email
  contactPhone: Phone
}
```

Avoid Primitive Obsession

Question: Can we express `cardNo` with `string`? If so can it be any string? Is `firstName` an arbitrary length string? Obviously you can't answer these questions as this model doesn't contain relative domain knowledge.

You may use `string` type for `cardNo` during programming, but in domain model, `string` can't express the domain knowledge of `cardNo`.

`cardNo` is a 19-length string starts with `200`, `name` is a string whose length is less than or equal to 50. As such, the domain information can be implemented by `type alias`:

```
type CardNo = string
type Name50 = string
```

With the above types, you now have chance to include the `cardNo` business rules inside domain models by defining function.

```
type GetCardNo = (cardNo: string) => CardNo
```

If a user typed in a string with 20 length, then what will the function `GetCardNo` return? null? or exception thrown? Actually functionaly programming has more elegant way such as Either Monad or Railway oriented programming to handle errors. At least we can present the function's signature by Optional:

```
type GetCardNo = (cardNo: string) => Optional<CardNo>
```

The function expresses the validation process clearly, if you user typed in a string, then returns a CardNo type or empty.

```
type CreditCard = {
  cardNo: Optional<CardNo>
  firstName: Name50
  middleName: Optional<string>
  lastName: Name50
  contactEmail: Email
  contactPhone: Phone
}
```

So now the code is full of domain knowledge, and these types can be used as unit tests as well. For example, you'll never assign an email to contactPhone, as they are not string, so in turn they represent different domain knowledge.

The Atomicity and Composibility of the domain models

There were 3 names in the above domain model, can they be changed separately? for example, change `middle name` only? If not how can we encapsulate the knowledge of atomicity change into the domain model?

In fact we can easily extract `Name` and `Contact` types and compose them:

```
type Name = {
  firstName: Name50
  middleName: Option<string>
  lastName: Name50
}
type Contact = {
  contactEmail: Email
  contactPhone: Phone
}
type CreditCard3 = {
  cardNo: Optional<CardNo>
  name: Name
  contact: Contact
}
```

Make the error state can't be present

There is an important principle in domain modelling, which can be understood as: The domain models you built should have as many static validations and constraints as possible to make error occurs in compilation time instead of run time, so as to avoid the chance for mistakes. In fact all the domain modellings are following this principle, for example, the Email type and Phone type in the above code. Why not use string? Because string is lakcing of domain knowledges, which gives developers chances to make mistakes.

Let's see another example. The above domain model has a contact type, which contains an Email and Phone properties. After payment done, system can utilize these 2 properties to send notification to user, so there is a rule generated: User must fill in

Email or Phone to receive payment messages.

First of all, the above domain models are not matching this business rule, because both Email and Phone are non-nullable type, which means these 2 properties are both required.

Can we change both of them to be Optional?

```
type Contact = {
  contactEmail: Option<Email>
  contactPhone: Option<Phone>
}
```

Obviousely we can't do this, as it violets the principle of Make illegal state unrepresentable, so gives chances for coding mistakes. Your domain model represents an illegal state, that both Email and Phone can be empty. You may argue that my xxService will do the validation, to make sure they'll never be both empty. Sorry, we hope our domain model can encapsulate this domain knowledge. For xxService, it's unrelated to domain model. So can we express this rule in the modle model or not? The answer is yes! Because there is a "or" in the rule, so it implies that we can use the type Or (union type) to express this relationship:

```
type OnlyContactEmail = Email
type OnlyContactPhone = Phone
type BothContactEmailAndPhone = Email & Phone
type Contact =
  | OnlyContactEmail
  | OnlyContactPhone
  | BothContactEmailAndPhone
```

Conclusion

By using TypeScript to guide the domain modelling, we can avoid classes and sub classes, let alone the keywords `abstract` and `bean`, etc.

To measure how good or how bad a domain model is, we need to judge

- is the domain model contains as many domain knowledges as possible, can it map the domain models inside domain experts' brains?
- Can domain model itself be the documents, so everyone can share and communicate with it?

At the mean time, a framework should have as less jargons as possible. For example if you created a domain model class named `AbstractContactBase`, you increased the

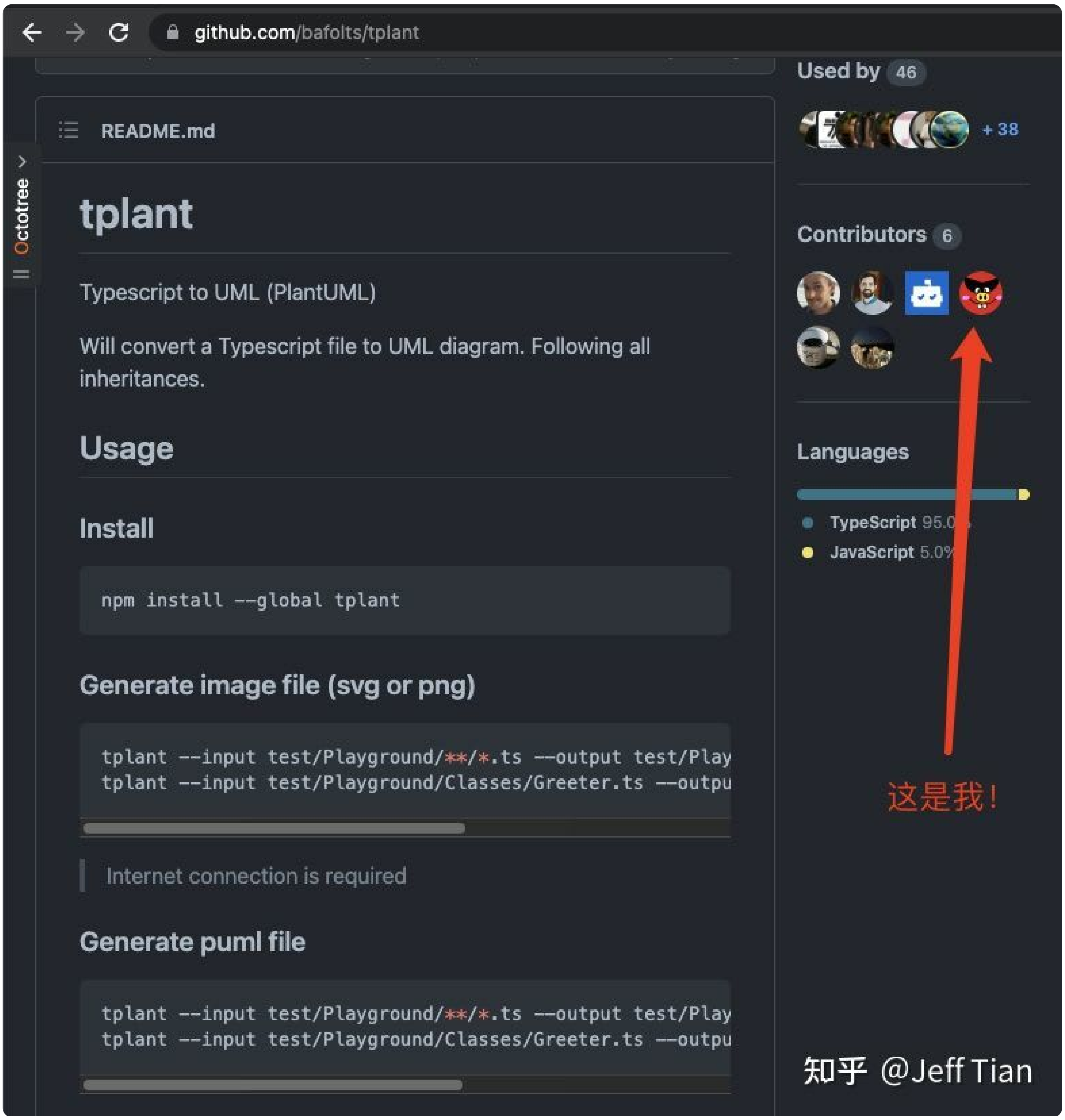
complexity of the system without any help on domain models sharing.

Bonus

If you use TypeScript, not only you can build a rich model models, but also you can utilize some tools to generate UMLs from the code. So you can birdview the project's domain models very quickly!

tplant

tplant is that kind of tool which I participated in it.



tplant

Typescript to UML (PlantUML)

Will convert a Typescript file to UML diagram. Following all inheritances.

Usage

Install

```
npm install --global tplant
```

Generate image file (svg or png)

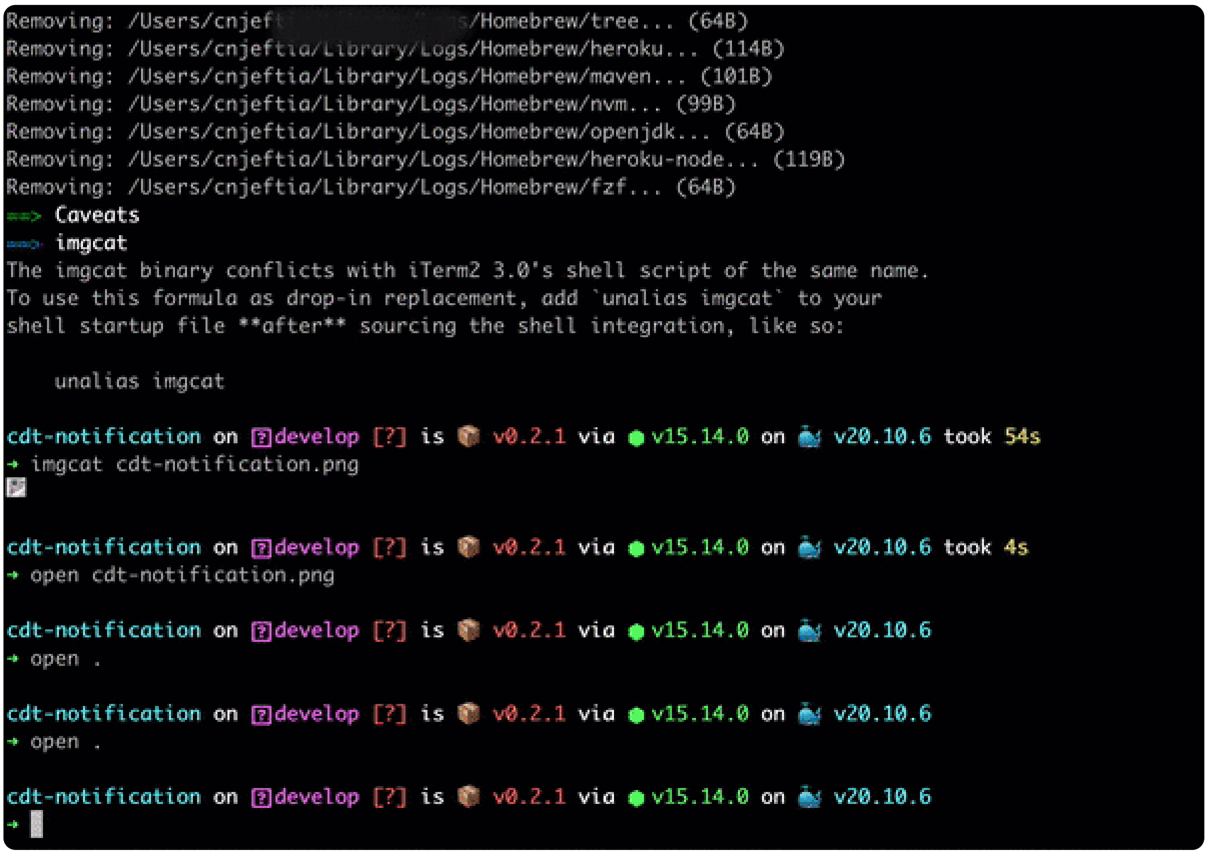
```
tplant --input test/Playground/**/*.ts --output test/Play
tplant --input test/Playground/Classes/Greeter.ts --outpu
```

Internet connection is required

Generate puml file

```
tplant --input test/Playground/**/*.ts --output test/Play
tplant --input test/Playground/Classes/Greeter.ts --outpu
```

Screen recording of usage



```
Removing: /Users/cnjef[redacted]s/Homebrew/tree... (64B)
Removing: /Users/cnjeftia/Library/Logs/Homebrew/heroku... (114B)
Removing: /Users/cnjeftia/Library/Logs/Homebrew/maven... (101B)
Removing: /Users/cnjeftia/Library/Logs/Homebrew/nvm... (99B)
Removing: /Users/cnjeftia/Library/Logs/Homebrew/openjdk... (64B)
Removing: /Users/cnjeftia/Library/Logs/Homebrew/heroku-node... (119B)
Removing: /Users/cnjeftia/Library/Logs/Homebrew/fzf... (64B)
==> Caveats
==> imgcat
The imgcat binary conflicts with iTerm2 3.0's shell script of the same name.
To use this formula as drop-in replacement, add `unalias imgcat` to your
shell startup file **after** sourcing the shell integration, like so:

    unalias imgcat

cdt-notification on develop [?] is 📦 v0.2.1 via ● v15.14.0 on 🐳 v20.10.6 took 54s
→ imgcat cdt-notification.png

cdt-notification on develop [?] is 📦 v0.2.1 via ● v15.14.0 on 🐳 v20.10.6 took 4s
→ open cdt-notification.png

cdt-notification on develop [?] is 📦 v0.2.1 via ● v15.14.0 on 🐳 v20.10.6
→ open .

cdt-notification on develop [?] is 📦 v0.2.1 via ● v15.14.0 on 🐳 v20.10.6
→ open .

cdt-notification on develop [?] is 📦 v0.2.1 via ● v15.14.0 on 🐳 v20.10.6
→
```

Text version of usage

```
npm install --global tplant
cd your-typescript-project-folder
tplant --input src/**/*.ts --output output.svg
open output.svg
```

Top comments (2)

sadtomatoofjoy • 3 de set.

> For example, you'll never assign an email to contactPhone, as they are not string, so in turn they represent different domain knowledge.

Typescript's type compatibility will actually allow this (if they are both string). In this example, you can also assign a Name50 to a CardNo beacause they are both string and typescript won't say anything if you mistakenly use one instead of the other

ZairaMoore • 21 de mai.

I got some helpful information in your post!! Spell to make him want you

Code of Conduct • Report abuse

Friends don't let friends browse without dark mode.

Sorry, it's true.

Jeff

A wild full stack developer.

LOCATION
Shanghai

EDUCATION
Fudan University

WORK
Full stack developer

JOINED
16 de jan. de 2020

More from Jeff

The Implementation of Inversion of Control based on TypeScript
#typescript #ioc #di