Abstract

Software Engineering is less about coding and more about communication, between stakeholders, team members and programs themselves. The true job of a Software Engineer is to establish a control- and message-flow mechanisms that are easily maintained, understood by other developers and can scale up well. This is especially true with Web Computing. However, with the rise in the number of readily available “frameworks” that claim to solve it in a way, the traditional values behind building software systems are gradually lost.

The author was very proud to study Computer Science at the University of Sussex, where he was taught the basics of web computing, with such concepts as XSLT, but he also saw that a lot of other people would prefer to learn a trendy framework instead. Having passed his exam with 97% and now founded a Software Engineering company, he can’t emphasise enough the importance of the computing tradition as well as culture, which must be preserved at all cost.

In trying optimise the internal development process, we have come up with a discovery called type.engineering, which by looking at the waterfall model, decides to take a step back and focus on the design of systems before implementation. The paper contributes a novel way to write programs in XML notation, which can then be converted into any language of choice via abstract classes and code generation. We explore the topic of developer satisfaction to understand the significance of auto-completion hints and demonstrate how multiple inheritance is arguably one of the best features of a programming language, which sadly is missing from most technology stacks.
Background

We’ve been developing JavaScript software since 2014. Following the scientific method, we created a context-testing framework as well as a documentation utility that together would, in the process of making and publishing packages, allow us to conduct experiments and document the findings. Having undergone training in Computer Science and AI, we’ve seen that science is about understanding the problem, coming up with hypotheses, designing and executing experiments, evaluating results critically and then drawing conclusions. Software Engineering then is just applied science, and this is why, the tagline for our company in London, is “Backed by the British Scientific tradition”.

This extensive research into optimisation of developer workflows has led us to the invention of type.engineering: the core idea is to extract types buried in the language-specific implementations into the design stage, and provide tools for working with them. But to understand how we came about this solution, we need to look at the software stack that was built in the last couple of years.

A. Documentation

**Documentary**

*Documentary is a command-line tool to manage documentation of Node.js packages of any size. Due to the fact that there is usually a lot of manual labour involved in creating and keeping up-to-date a README document, such as copying examples and output they produce, there is a need for software that can help automate the process and focus on what is really important, i.e., documenting features. *Documentary* serves as a preprocessor of documentation and enhances every area of the task of making available high-quality docs for Node.js (and other languages) packages for fellow developers.*

```
yarn add -D documentary
npm i documentary --save-dev
```

Documentary is a documentation processor that allows to put together a single readme from multiple files for better organisation.
It can embed examples and their output automatically which improves productivity and provides extra quality assurance.
It presents information about types, with links between them. Documentary can generate Wikis and tables of content.
B. Testing

Secondly, zoroaster is a context-testing framework which isolates tests from the scope to make them pure functions.

Tests now receive testing APIs via a context with various utility methods which are documented for better QA experience.
Moreover, contexts can be published separately which allows to test them, providing double quality assurance.
Zoroaster also supports mask testing where inputs are mapped to outputs in non-js files which enables syntax highlighting and improves developer satisfaction.
C. Compilation

Thirdly, we’ve adapted Google Closure Compiler to compile Node.JS packages which wasn’t possible before: it worked on JavaScript, but not on Node.JS which is server JavaScript.

The compiler provides static-type checking which is very useful.
It also obfuscates source code to protect intellectual property.

```javascript
/**
 * @extends {typal.IMember}
 * @extends {typal.ICallable}
 * @interface
 */
_typal.IMethod = function() {}

/**
 * @type {boolean}
 */
_typal.IMethod.prototype.isConstructor

/**
 * @param {function(!_typedefsParser.Type): string} getLinks
 * @return {string}
 */
_typal.IMethod.prototype.toTypeScriptFunction = function(getLinks) {}
```

The compiler requires the presence of externs, i.e., header files with type information to enable static type-checking. Maintaining externs violates DRY principle and there’s no supporting infrastructure.
D. Type Management

In trying to prevent the violation of the don’t repeat yourself principle, I invented a new software development methodology, called Type.Engineering. It models systems by writing types in XML files, independently from any programming language.

- Type Engineering is a modelling language based on XML where types are defined in separate files.
- Disentangles design from implementation and documentation.
- Allows to project onto any programming language.
Types are used all over the world in OOP, but there are no tools to work with them. Once they are maintained separately, abstract classes can be generated and implementation added on them so the developer experience is just incredible!

- Interface-driven development for best-in-class OOP.
- From types, abstract classes are generated.
- Implementation is added against auto-generated abstract classes.
- The developer experience is better than anything ever seen before!
Additionally, I’ve turned on multiple inheritance which JavaScript doesn’t nominally support. This unlocks many possibilities for code reuse impossible with traditional inheritance hierarchy. I’ve introduced 3 new design patterns.

- Multiple inheritance support: almost a new programming language!
- Incredible opportunities for code reuse and modularity.
- 3 new design patterns: initialiser, implementer and bound destructuring.
Most of all, Type Engineering is a communication tool that allows different components like UI and back-end to be treated as a whole. Being a design language, it involves everyone in the software development process and lets them see greater picture.

"But the business of software building isn't really high-tech at all. It's most of all a business of talking to each other and writing things down. Those who were making major contributions to the field were more likely to be its best communicators than its best technicians."

Tom DeMarco [Why does software cost so much?]

Software engineering is about meeting business needs and establishing communication channels in a robust, sustainable and maintainable ways. This is exactly what Type Engineering is for.

Summary

There are 4 main iterative stages in programming: design/implementation, documentation, testing, and building. We put design and implementation together because here we’re talking about actual programming experience rather than requirements analysis etc, so the design means field-work of defining interfaces and types in the system.

The standard web-developer would involve writing JavaScript code and annotating it with some JSDoc documentation above functions, classes, methods and fields. A documentation processor would then create a web-page from these types. However, this is inefficient as:

- There’s little control of how the documentation is outputted and it always looks like a machine-put together bunch of class definitions instead of user-friendly manual with step-by-step instructions enhanced with examples and discussion.
- During the compilation, the JSDoc is lost and the in-IDE documentation is not given to the user which is a major drawback as it’s an absolute must for developer productivity and satisfaction as discussed later.
- The presence of JSDoc comments in source makes navigating the code base hard and obscure.
- There are bugs in both the IDE and the most commonly used JS transpiler (Babel) that prevent the correct workings of JSDoc parsing, making millions of developers switch to TypeScript.
- A lot of organisations would like to work with pure JS as it’s more traditional however because of the point above they are literally forced into adopting TypeScript.
- No company in the world, except for ours, has understood the real needs of the developer by looking at the root of the problem of programming experience and providing the right tool to solve it.

By having a single repository of types, defined in XML files (but in future, any front-end to such system is possible, including a user-interface), we achieve three-fold optimisations:
1. Documentation

As Documentary can put together multiple files, we just need to create different sections of documentation in different files, and talk about each type and function on its own page. The processor will ensure that any types referenced in the process, are linked to the correct anchor in the README file, or across URLs if we’re generating Wikis. Using the `<protype name="ns.Type"/>` component, the table for a type will be generated, and then we can use the `<example src="doc/example/example.js" />` component and `<fork src="doc/example/example.js" />` to add the example listing as well as its output. This process not only ensures that the examples actually work, but also saves so many man-hours every year that it should be made compulsory for introduction in every organisation to help the economy.

2. IDE experience

Here, things are two-fold. We’re talking about the experience of both the developer-producer as well as developer-consumer. The producer of source code doesn’t need to have any JSDoc in his source, as the types are defined in separate files, known as typedefs. This is analogous to C .h header files that provide the type information for the compiler, but typedefs also provide natural language descriptions of functions, methods and fields. The source code can simple add `/*/ ** @type {ns.Type} */` above a function and the IDE will recognise the types of arguments so that it’s easy to work with them.
As the source code is clear of JSDoc, and the compiled program doesn’t have any either, the type information is stored in typedefs. But for the best experience of those who install the package, the functions need to be documented in the entry to the software component. This is done automatically by the tool that we’ve developed.

```
/**
 * Parses the supplied HTML and CSS and removes unused selectors. Also removes empty CSS rules.
 *
 * @param {type} config — Options for the program.
 * @param {type} config.html — The input HTML.
 * @param {type} config.css — The CSS to drop selectors from.
 * @param {type} config.keepAlternate — Whether to keep the @alternate comment for Closure Stylesheets. Default false.
 * @return {type} Whether TrapCSS should remove this selector. The shouldDrop hook is called for
 */

function trapcss(config) {
    return _trapcss(config)
}
```

3. Compiler Externs

The compiler needs to know the type information to be able to statically type-check programs. Such information is provided via externs. From XML files, we also generate externs at the same time as typedefs. In other words, using the single source of truth, we’ve killed 3 birds with one stone: readme documentation, IDE documentation for developer and consumer, and compiler infrastructure. It’s absolutely not possible to have established this highly-efficient strategy without treating types as first-class citizen in a design language like type.engineering.
Things Start To Get More Interesting

We’ve talked about IDE documentation of functions, however there’s another important discovery that has been made in summer 2020. That is, of working with classes in JavaScript. The main problem that we were solving, was to provide good IDE experience for the developer when she’s writing classes that are supposed to conform to the interfaces that she’s defined in types.xml. The issue is that the IDE doesn’t understand the `@implements` tag, and when extending classes, the type info on arguments is lost.

In the example above, the Child class extends the Example class. Although the example has defined the method hello with an argument s as string, the child doesn’t pick up this type information which is very inconvenient. Moreover, our design language allows for a Protype (class) to implement multiple interfaces. Using current language capabilities, it is impossible which required us to build a “programming framework” that would solve these 2 problems.

Requirements:

1. Access documentation of interfaces with fields and methods and auto-completions on arguments to such methods.
2. To let implementations subclass from more than 1 interface, i.e., multiple inheritance.
Just as before, we’ll use the type information to generate abstract classes, which are then to be extended in code with the implementations. We’ll use an example of a TOM (type object model) node to illustrate the solution.

We have an INode interface that implements IParental and IAttributable.

```xml
<types ns="engineering.type.tom">
  <INode implements="IParental, IAttributable">
    <string name="content" init>
      The text content of the node without tags.
    </string>
    A node that underlies types and their parts.
  </INode>
  <IAttributable>
    <field type="!(Object<string, *)" name="attributes" init>
      Some attributes on the node.
    </field>
    <fn name="hasAttribute">
      <arg string name="attr">The name of the attribute.</arg>
      <return boolean>True if the attribute is present and false otherwise.</return>
      Checks whether the node has an attribute with the given name.
    </fn>
    An item with properties known as attributes.
  </IAttributable>
  <IParental>
    <field type="!(ArrayOf!INode)" name="children" init>
      An array with children.
    </field>
    <fn name="addChild">
      <arg !INode name="child">The child to add to the node.</arg>
      <return !IParental>"The chainable instance."/return>
      Adds a child to the list of children.
    </fn>
    An item that can act as a parent and be responsible for its children.
  </IParental>
</types>
```

Such design allows for clear separation of concerns: a node on its own is tasked with storing content, while the attributes handling as well as children handling are delegated to the appropriate interfaces.
Unfortunately, traditional OOP model seldom supports multiple inheritance which would mean that a Node implementation cannot inherit from both Attributal and Parental, as they’re not variant (not related to each other). Fortunately, JavaScript being a prototype-based language, we can fix this shortcoming by manipulating the prototype of any class.

```javascript
import { SUBTYPE_OPTS, time, getSuper, bindPrototype, initAbstract, assignSupers, subtype, implementEngineer } from '@type.engineering/type-engineer'
import './*

/** @extends {AbstractNode} */
class _AbstractNode {
  /** @param {string} name */
  constructor(name) {
    /** @type {string} */ this.content = ''
    /** @type {string} */ this.raw = ''
    /** @type {!Array<!engineering.type.tom.INode>} */ this.children = []
    /** @type {!Object<string, *}> */ this.attributes = {}
  }

  get asINode() {
    return /** @type {!engineering.type.tom.BoundINode} */ (this)
  }

  get superINode() {
    return /** @type {!engineering.type.tom.BoundINode} */ (bindPrototype(_AbstractNode, this))
  }

  get asIParental() {
    return /** @type {!engineering.type.tom.BoundIParental} */ (this)
  }

  get superIParental() {
    return /** @type {!engineering.type.tom.BoundIParental} */ (getSuper('IParental', this))
  }

  get asIAttributable() {
    return /** @type {!engineering.type.tom.BoundIAttributable} */ (this)
  }

  get superIAttributable() {
    return /** @type {!engineering.type.tom.BoundIAttributable} */ (getSuper('IAttributable', this))
  }

  get superNode() {
    return /** @type {!engineering.type.tom.BoundNode} */ (bindPrototype(_AbstractNode, this))
  }
}

initAbstract(_AbstractNode, 'INode')
```
A node that underlies types and their parts.

```javascript
export default class AbstractNode extends _AbstractNode {

  constructor(nodeName) {
```

First, we’ll generate an abstract class for the Node and other interfaces. They’ll include default initialisers as well as provide the `__implement` static method that can be used to copy methods and accessors from provided implementations onto the prototype of the abstract class.

Second, we extend the abstract class in code, and call the `__implement` method with all implementations that need to be present on the class.

```javascript
import AbstractNode from './types/lux'
import Attributable from './Attributable'
import Parental from './Parental'

export default class Node extends AbstractNode {
  constructor(nodeName) {
```
super(nodeName)
if(!nodeName) throw new Error('Node name is required.')

const { asINode: { setNameName },
} = this
setNameName(nodeName)

*** @constructor @extends {Node} @suppress {checkTypes} */ function _Node() {}
AbstractNode._implement(Node, Parental, Attributable,
_Node.prototype = /** @type {!Node} */ {
    initialiser(props, callback) {
        const { content = '' } = props
        const {
            asINode: { setContent },
            superParental: { initialise: parentalInitialiser },
            superAttributable: { initialise: attributalInitialiser },
        } = this
        parentalInitialiser(props)
        attributalInitialiser(props)

        setContent(content)
    },
    setNameName(nodeName) {
        const { asINode } = this
        asINode.nodeName = nodeName
        return this
    },
    setContent(content) {
        const { asINode } = this
        asINode.content = content
        return this
    },
    is(type) {
        const { asINode: { nodeName } } = this
        return nodeName === type
    },
    clone(deep = false) {
        const {
            asINode: { nodeName, content },
            asIAttributable: { attributes },
        } = this
        return this
    }
}
const n = new Node(nodeName)
n.initialiser({
  attributes,
  children,
  content,
})
n.raw = this.raw
return n
}
})

The interesting thing to see here is that methods are not defined within the class body itself, but in object literals that are passed to the implementor. This little trick allows to annotate such object literal with the type of the prototype, and hence enable the IDE hints for the developer.

```javascript
/*@constructor @extends {Node} @suppress {checkTypes} */
function _Node() {
  AbstractNode._Implements(Node, Parental, Attributable,
  _Node.prototype =
  /** @type {{#Node} */
  function initialiser(props, callback) {
    const { content = '' } = props
    const {
      asINode: { setContent },
      superParental: { initialiser: parentalInitialiser },
      superAttributable: { initialiser: attributalInitialiser },
    } = this
    parentalInitialiser(props)
    attributalInitialiser(props)
  }

  @method engineering.type.tom.INode.setContent(content: string): INode
  Sets the node name.
  @param content — The content.
  @return — The chainable instance.

  setContent(content) {
    const { asINode } = this
    asINode.content = content
    return this
  }]
```
As you can see above, the “content” argument to the `setContent` method is known to be of type string, and the IDE provides the developer with necessary documentation about the method itself, shown by hovering over the method.

Thirdly, we import the Parental and Attributable implementations (which we programmed using the similar method, but in separate files) into the JS module, and pass them to the `__implement` method. Now Node will subtype both of these implementations and thus enjoy the multiple inheritance property which is otherwise impossible in JavaScript.

We believe that this invention is ground-breaking as it not only facilitates an absolutely seamless developer experience with type inference and autocompletions, but also takes the language to the next level by allowing to subclass any number of classes. Having discovered this process, it’s very hard for us to imagine how not to use multiple inheritance, and we’d like to share our contribution to the broader developer community who might not have had the chance to try it in real life, as other languages such as Java or Go don’t support it either.

Dynamic Binding

We’ll come back to the example as there are additional points to discuss, however let’s quickly refer to Brad J. Cox Ph.D, the inventor of ObjC.

Revolutions happen so slowly, and often displace one group by another, because of value rigidity, the inability to relax the pursuit of an older good to gain a newer one. Examples of potential value rigidity traps abound:

[…] Seamless panaceas are better than kits of diverse tools, and Software is a closed universe in which all potential interactions between the parts of that universe can be declared when these parts are created by their compiler.

The closed-universe model surfaces as the belief that compile-time type checking is universally 'better'. The preference for panaceas instead of tools surfaces as the belief that early and late binding are mutually exclusive panaceas, that one should be chosen at the expense of the other by the language designer rather than providing many binding technologies that the user can choose from according to the job at hand.

He then goes on to give an example where a dynamic binding would be more suitable that static type-checking. We believe that our solution also benefits from the both worlds: the static type-checking can be performed on the written program to highlight any inaccuracies, but since JavaScript is a dynamic programming language, we can extend prototypes dynamically at runtime. The `type-engineer` package also provides the `__$extend` method that would add implementations to existing instances, which is useful to compose complex systems.
New Patterns

On top of that, to establish a super developer-friendly method of working with MI in JavaScript, Typal has a feature that automatically generates something called super-getters. As node needs to be initialised as a parent, and as an attributal, we need to be able to call both initialisers from these implementations in the body of the node intialiser. Due to the diamond problem, it would be impossible to simply call .super.initilse as there will be 2 superclasses. Therefore, our abstract class generator also provided 2 getters: superAttributal and superParental. These point to methods on the prototype of these 2 implementations and allow to call the relevant one easily in the order required by the programmer.

```javascript
initialiser(props, callback) {
    const { content = '' } = props
    const {
        asINode: { setContent },
        superParental: { initialiser: parentalInitialiser },
        superAttributal: { initialiser: attributalInitialiser },
    } = this
    parentalInitialiser(props)
    attributalInitialiser(props)
    setContent(content)
}
```

We call this the “intialiser” pattern. Moreover, there’s an additional “bound destructuring” pattern that allows to access methods without them loosing the “this” context. It’s made possible as once the super-getter is accessed, the low-overhead runtime binds all methods to the prototype so that the code can be kept clean, and a neat destructuring tree drawn, before using all such methods in the method body.

It works not only with super-classes, but with standard interfaces as well. For example, consider the “clone” method:
We’ve accessed the `children` property not on the instance itself, but via casting it into the IParental first via the `asIParental` getter also generated in the abstract class. It’s super-userfull to have this casters for system with a large number of interfaces and multiple inheritance enabled. It greatly improves the developer understanding and thought flow, as her attention span allows to keep references to interfaces she’s currently using and methods that belong to them, instead of referring to a single “this” all the time.

**Autocompletions = Dopamine**

In our research and experience, it’s become clear that the single most satisfying thing in being a developer, is the ability to receive autocompletion hints. If the hint is shown by the IDE, it proves that what we’re doing is right. If it’s not shown, we can immediately sense that something is wrong, either with our design or implementation. Such hints are provided during destructuring:
We can then select the correct property or method that we want to use in code. I strongly believe that it’s the single most important thing in the whole process of modern software development. I can keep on writing my programs without running the compiler on them, by simply using the power of autocompletions-checks. Every time I get access to the list of methods and fields, I know that my types are right. Also I don’t have to look up the documentation online or guess whether a property has American English or British English spelling, say. I type a single letter and from the list select what I need.

It’s extremely satisfying and it’s like completing a puzzle. So first we design our types and create the actual jigsaw pieces, and then put them together during implementation. Since multiple inheritance is now enabled, we can operate on almost 3-dimensional abstraction level which isn’t possible with traditional vertical inheritance. The design language that we’re proposing is called type.engineering precisely because we need to have those pieces, known as types, first, and then engineer them into software. The current approach of simply using JSDoc is schoolboy and simply inadequate. Our professional software development company seeks to promote this method to the industry members as it will transform the way we build software.

Piping

Finally, let’s quickly mention other applications of type engineering. In the paper, we’ve shown how to write JavaScript programs while enjoying great devX. But it doesn’t end here. More often than not, software teams are multilingual these days. It means that front-end developers have to interact with back-end coders, which is usually done via REST interfaces. However, we believe that REST is a polyfill that is used until the real solution has been found. And we claim to have found such a solution. It’s actually been known for a long time but with the demise of web computing, has been largely abandoned. It’s called RPC - remote procedure calls.

Instead of manually drafting POST/GET/PUT routes, we need to focus on building our models and services. A model will then have all the required methods that can be called on it. To call them from the front-end, we again generate the abstract classes for JavaScript, as well as the server, e.g., Java, and let our tool do the work. Using protocol buffers binary encoding, we also achieve 2 things:

- Smaller payloads as compared to general JSON usage
- Hiding the data structures from peeking eyes in the console.

As types are now treated on their own, using TYNG (a sophisticated piping tool for Type Engineering), we create A) .proto files B) the .java files to handle encoded messages and C) the .js files that both encode and decode data from the server.
**Conclusion**

Type Engineering is a radical innovation that will change the way we approach Software Engineering. Quite often, we need to take a break from our current routines and analyse where optimisations can be performed. Our young company, Art Deco™, in the process of establishing its own stack, rather than using existing frameworks, has researched and fully understood the requirements that modern programmers have.

Being inspired by traditional OOP school represented by people like Brad Cox we’ve been able to extend the classical concepts to the current day trends and reimagine how to approach crafting software with the most productive and HOLISTIC methodology.

---

**Brendan Eich**

**JANUARY 25, 2011 AT 4:32 PM**

It would look like what you sketch, aka “classes as sugar”:

```java
    class Derived extends Base {
        ...
    }
```

The [this]-free private member access is a win. Private by default was agreed to in 2008 fall. What went wrong?

Nothing, really. This plan may make a come-back, but with some traits or mix-in support. The constructor object initialiser extension should merge or subsume, somehow. The guard idea depends on the class/trait/constructor proposal, and should not be rushed, so it’ll come after.

I expect we’ll hear more on classes and traits soon on es-discuss.

/be

P.S., it’s interesting to see how Brendan Eich, the author of JS, talked about “constructor object initialiser” back in 2011. We didn’t find that comment prior to coining the “initialiser” pattern which is a great coincidence which indicates that we’re on the right path.
Art Deco™ is a professional Software Engineering company in London that applies Scientific research methodology to building robust, stable and maintainable software systems. With introduction of its own inventions such as context-testing framework and mask-testing, it was able to achieve enormous productivity that it wills to share with Software Engineers in the industry, to build happier tomorrow and lead the way to ethical, cultural and rational future of the profession.

Anton studied Computer Science & AI at the University of Sussex, England where he was introduced to the scientific method. He really enjoyed the scientific process and fully understood its purpose and tactics. Having developed the skill to analyse problems through research and making of hypnoses and to tackle them critically through experimentation and evaluation, he gained the ability to approach any problem-solving exercise with a mindset cultured through traditional British science.

PPS. All the apologies for any grammatical/stylistic mistakes in the paper. The Call for Papers was literally seen 4 hours before the deadline. We’re looking forward to working with the panel on improving upon the initial write-up.