CIMAT
Centro de Investigación en Matemáticas, A.C.

IMPLEMENTACIÓN DE DBC Y REFINEMENT EN MANTIS

REPORTE TÉCNICO

Que para obtener el grado de
Maestro en Ingeniería de Software

Presenta
Miguel Ángel Iñiguez González

Director de Reporte Técnico:
Mtro. Alejandro García Fernández

____
Autorización de la versión final

Zacatecas, Zac., 26 de Octubre de 2018

Agradecimientos

A Dios por la oportunidad de cada día aprender más. A mis compañeros y amigos, por compartir sus conocimientos, esfuerzo y trabajo. A los docentes del Centro de Investigación en Matemáticas Unidad Zacatecas, por su gran profesionalismo al impartir y compartir sus conocimientos.

A mi director de reporte técnico el Mtro. Alejandro García Fernández, por compartir en todo momento sus conocimientos, por su paciencia y ayuda durante el desarrollo de este reporte técnico.

Finalmente, al Concejo Nacional de Ciencia y Tecnología (CONACYT) por el financiamiento mediante la beca de posgrado para la realización de mis estudios.

Resumen

Contexto: Desde hace 9 años la cadena de bloques (blockchain) ha surgido como una tecnología y una revolución financiera iniciada por Bitcoin, desde entonces han aparecido muchas otras criptomonedas como por ejemplo Ethereum(ETH), Litecoin, Ripple.

Originalmente ETH fue creada como una mejora a Bitcoin y junto con ella nació un nuevo modelo empresarial llamado Organización Autónoma Descentralizada (DAO). La DAO fue creada como un Smart Contract (*contrato inteligente*) dentro de la cadena de bloques de ETH y en principio no contaba con una estructura administrativa convencional ya que esta estructura estaba formada por los inversionistas de la DAO. Uno de los métodos del contrato inteligente de la DAO tenía una vulnerabilidad de seguridad (splitDAOfunction), el resultado de este fallo se traduce en el robo de 3,642,408.53 de Ethers y el nacimiento de una nueva moneda criptográfica llamada Ethereum Classic (ETC). Hoy en día ETC se ha afianzado dentro de las criptomonedas más usadas, tanto es así que a la fecha (abril de 2018) la capitalización de mercado de esta moneda se calcula en $2,240,739,489 USD (datos tomados de coinmarketcap.com).

Todas las criptomonedas antes mencionadas tienen clientes que proporcionan interfaces gráficas o de consola que nos permiten interactuar con la blockchain, en las cuales es posible realizar bloques o transacciones. En ETC podemos encontrar al proyecto Mantis, el cual es un desarrollo realizado especialmente para esta moneda por la empresa IOHK. El cliente Mantis tienen un code coverage (*cobertura de código*) del 83% y ha sido sujeto a auditorías externas de seguridad. Con el objetivo de mejorar la calidad del software, los desarrolladores de IOHK se preguntaron si era posible utilizar precondiciones y postcondiciones en Mantis, esta es la pregunta que motiva este reporte técnico.

Objetivo: Hacer refactoring (*refactorización*) del cliente de Mantis utilizando Design By Contract (DBC) e identificar si esta técnica realmente mejora la calidad de Mantis o si con el desarrollo que se tiene es suficiente y no vale la pena utilizar DBC.

Método: Analizar el código de Mantis con grafos para identificar el módulo que más se usa. El módulo identificado se refactoriza (con DBC), se prueba (usando las pruebas estandarizadas del proyecto) y se analizan los resultados.

Resultados: Se realizaron muy pocas modificaciones de código de la clase más usada (BlockHeader), los cambios se realizaron con base en el Yellow Paper (documentación fundamental de Ethereum). Se descubrió que, aunque en la práctica nunca se ha presentado la creación un de un bloque inválido, si es posible en teoría crear uno, por que las pruebas automatizadas no siguen las reglas del Yellow Paper para la creación de bloques. Se eliminó la posibilidad de que alguna persona maliciosa pueda atacar Mantis a través de este mecanismo.

Conclusiones: DBC trae beneficios a un código de alta calidad *per se*, que ya ha sido auditado y que aún así cubre partes que simplemente testing (*pruebas*) y la inspección de código no pueden descubrir. Es recomendable usar DBC e incluso aplicarlo antes de hacer la codificación de las pruebas unitarias. Las pruebas sólo pueden demostrar la presencia de errores y no su ausencia, mientras que con DBC si se puede demostrar su inexistencia.

Palabras clave: Design by Contract, Refinement types, Call graph.

Índice general

ÍNDICE DE FIGURAS......IX

ÍNDICE DE ALGORITMOS......XI

ÍNDICE DE ABREVIATURAS......XIII

1 INTRODUCCIÓN **15**

1.1 OBJETIVO GENERAL 15

1.2 JUSTIFICACIÓN 16

1.3 ORGANIZACIÓN DEL REPORTE TÉCNICO 17

2 ANTECEDENTES **18**

2.1 CADENA DE BLOQUES 18

2.1.1 Blockchain......18

2.1.2 Smart Contracts......20

2.1.3 Bitcoin......21

2.1.4 Ethereum......22

2.1.5 DAO......23

2.1.6 Ethereum Classic......24

2.2 SOFTWARE EN LA CADENA DE BLOQUES 25

2.2.1 Mineros......25

2.2.2 Nodos......26

2.2.3 Wallets......26

2.2.4 Mantis......26

2.2.5 Scala......27

2.3 ANÁLISIS DE DEPENDENCIAS 28

2.3.1 Análisis Estático y Dinámico......28

2.3.2 Gráfico de llamadas (Call Graph)......29

2.3.3 Java-callgraph......30

2.4 TÉCNICAS DE DESARROLLO DE CALIDAD DE SOFTWARE 30

2.4.1 Pruebas (Testing) *31*

2.4.2 Prueba Unitaria (Unit Test) *31*

2.4.3 Property-based Testing *32*

2.4.4 Integración continua (Continuous Integration) *32*

2.4.5 Tipos refinados (Refinement types) *33*

2.4.6 Diseño por contrato (Design By Contract) *34*

2.4.7 Liskov Substitution Principle (LSP) *35*

2.4.8 Métodos formales *36*

3 EXPERIMENTO I: ANÁLISIS DE DEPENDENCIAS 37

3.1 DESCRIPCIÓN DEL PROBLEMA 37

3.2 MÉTODO IMPLEMENTADO 38

3.3 RESULTADOS 43

3.4 CONCLUSIONES 44

4 EXPERIMENTO II: APLICACIÓN DE TÉCNICAS DE CALIDAD EN EL SOFTWARE 45

4.1 DESCRIPCIÓN DEL PROBLEMA 45

4.2 MÉTODO IMPLEMENTADO 47

4.2.1 Refinement types *47*

4.2.2 Design By Contract *48*

4.3 RESULTADOS 51

4.4 CONCLUSIONES 52

5 DISCUSIÓN DE LOS RESULTADOS 54

6 CONCLUSIONES 55

6.1 OBJETIVO GENERAL 55

6.2 LECCIONES APRENDIDAS 56

6.3 TRABAJO FUTURO 57

7 REFERENCIAS 58

Índice de figuras

FIGURA 2.1EJEMPLO DE CADENA DE BLOQUES. (ZHENG ET AL 2017) 18
FIGURA 2.2 SISTEMA DE SMART CONTRACT (DELMOLINO ET AL 2016)........ 20
FIGURA 3.1 TOP 10 DE CLASES CON MAYOR GRADO DE CENTRALIDAD 43
FIGURA 3.2 ZOOM DE CALL GRAPH MOSTRANDO LAS CONEXIONES A LA CLASE BLOCKHEADER 44
FIGURA 4.1 RESULTADOS DE PRUEBAS AL APLICAR DBC Y REFINEMENT TYPES 52

Índice de algoritmos

ALGORITMO 3.1EMPAQUETADO DE MANTIS 38
ALGORITMO 3.2 INSTALACIÓN DE JAVA-CALLGRAPH 39
ALGORITMO 3.3 EJECUCIÓN DE ANÁLISIS ESTÁTICO SOBRE EMPAQUETADO DE MANTIS 39
ALGORITMO 3.4 GUARDADO DE INVOCACIONES DE CLASES 39
ALGORITMO 3.5 ESTRUCTURA DE BASE DE DATOS 40
ALGORITMO 3.6 INSERCIÓN DE BASE DE DATOS 41
ALGORITMO 3.7 GRÁFICO DE CLASES CON MAYOR GRADO DE CENTRALIDAD 42
ALGORITMO 4.1 CONSTRUCTOR DE BLOCKHEADER 46
ALGORITMO 4.2 TIPOS REFINADOS DE HASH 256, 160 Y 64 BIT 47
ALGORITMO 4.3 FUNCIÓN PARA VALIDAD TIPOS REFINADOS EN TIEMPO DE EJECUCIÓN 48
ALGORITMO 4.4 FUNCIÓN VALREF 48
ALGORITMO 4.5 FUNCIÓN BOOLEANTOMAP 49
ALGORITMO 4.6 FUNCIÓN VALIDATECONSTRUCTOR 50
ALGORITMO 4.7 ASERCIÓN DE PRECONDICIÓN 51

Índice de abreviaturas

ABREVIATURA	SIGNIFICADO
CI	Integración continua
DAO	Organización Autónoma Descentralizada
DBC	Design By Contract
ETC	Ethereum Classic
ETH	Ethereum
EVM	Ethereum Virtual Machine
IOT	Internet de las cosas
JVM	Java Virtual Machine
LSP	Liskov Substitution Principle
PBT	Property-based testing
USN	Universal Sharing Network

1 Introducción

La cadena de bloques ha surgido como una tecnología revolucionaria, impactando en todas las áreas donde se utilizan bases de datos, aún cuando su aplicación se centra en las finanzas por medio de las criptomonedas.

Lo que hace posible la existencia de la cadena de bloques y las criptomonedas son una serie de programas que corresponden genéricamente a la categoría de: Mineros, Nodos y Wallets.

En el presente reporte técnico, se expone el proceso de refactoring que se hizo al nodo Mantis. El software Mantis es un nodo que se conecta a la cadena de bloques Ethereum Classic (ETC) y actualmente (abril del 2018) los ETCs en circulación tienen un valor de mercado de: 2.2 Billones de dólares. Por ello, es de suma importancia que el software relacionado con la cadena de bloques sea programado con los más altos estándares de calidad. En el presente trabajo nos planteamos los siguientes Objetivos.

1.1 Objetivo General

Incrementar la calidad del nodo Mantis y evaluar si la intervención tuvo resultados positivos a través de pruebas automatizadas.

1.1.1 Objetivos Específicos

1. Identificar los módulos más críticos de Mantis
2. Proponer cuales son las técnicas de aseguramiento de la calidad que dan la mejor rentabilidad sobre el tiempo invertido
3. Medir cual es el impacto de aplicar los paradigmas de diseño por contrato y refinamiento de tipos, en el código de Mantis

1.2 Justificación

El mercado total de criptomonedas está valuado por encima de los 700 billones de dólares. A la fecha de este reporte (abril del 2018) ETC está valuado en 2.2 Billones de dólares. Este mercado global depende 100% de software.

Sabemos de muchos casos famosos donde una vulnerabilidad del software, causó pérdidas por cientos de millones de dólares. Por el gran valor de mercado y la fragilidad del software, es necesario que todos los programas que colaboran en la gestión de una criptomoneda, sean desarrollados con los más altos estándares de calidad.

En este reporte técnico tomamos el Nodo Mantis, el cual ya ha sido desarrollado con múltiples técnicas de aseguramiento de la calidad (pruebas unitarias, property-based testing, continuous integration, inspecciones y auditorías externas) (Ethereum Classic Client (Mantis) Security Audit 2018) e investigamos si era posible incrementar su calidad un poco más. Se aplicaron técnicas reconocidas en la literatura, pero poco utilizadas en particular, tales como Design by Contract (Meyer 1997, Mitchell et al 2002) y Refinement types (Davies 1997).

El uso del paradigma Design by Contract puede reducir errores y en consecuencia mejorar la confiabilidad del software (Meyer 1992), la técnica Refinement types agrega nuevas

características que pueden ser expresadas y verificadas a tipos primarios y así, mejorar la calidad del software (Davies 1997).

1.3 Organización del Reporte Técnico

El reporte está organizado de la siguiente manera:

Capítulo 2 - Antecedentes: Engloba la revisión literaria acerca de conceptos fundamentales sobre la cadena de bloques y técnicas de desarrollo de calidad de software.

Capítulo 3 - Experimento I: Análisis de dependencias: Presenta el trabajo realizado de análisis de dependencias sobre el nodo Mantis.

Capítulo 4 - Experimento II: Aplicación de técnicas de calidad en el software: Presenta el trabajo realizado aplicando Refinement types y DBC sobre el nodo Mantis.

Capítulo 5 - Discusión de resultados: Describe el por qué de los resultados del refactoring aplicado a mantis.

Capítulo 6 - Conclusiones y trabajo futuro: Presenta las conclusiones obtenidas de este trabajo y las mejoras que podrían ser añadidas.

2 Antecedentes

En esta sección se describen los conceptos relacionados con la cadena de bloques, software que interviene en la blockchain, análisis de dependencias y técnicas de desarrollo de calidad de software.

2.1 Cadena de bloques

A continuación, se describen algunos conceptos básicos sobre la cadena de bloques, así como la historia que generó la criptomoneda Ethereum classic.

2.1.1 Blockchain

Originalmente blockchain fue introducido por Bitcoin (un sistema de pago peer-to-peer), pero luego evolucionó para ser utilizado en el desarrollo de una amplia gama de aplicaciones descentralizadas (Alharby and Moorsel 2017). Técnicamente, blockchain consiste en un conjunto de datos conformados por cadenas de paquetes (bloques), donde un bloque comprende múltiples transacciones. En la Fig. 2.1 se muestra como el blockchain es extendido por cada bloque agregado, formando así un ledger (*base de datos distribuida*).

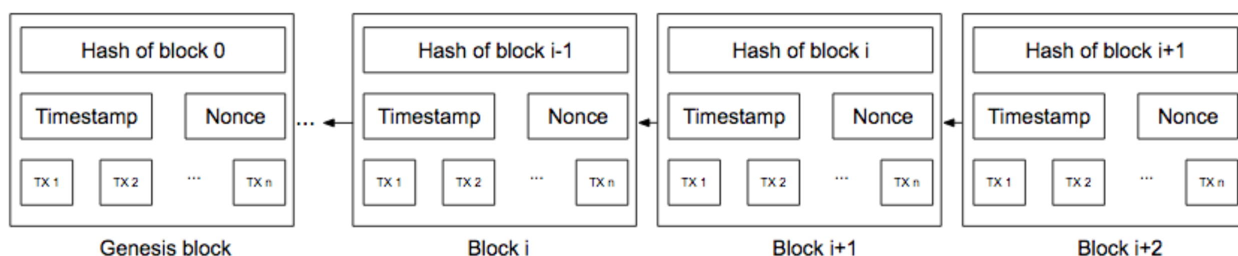

Figura 2.1Ejemplo de cadena de bloques. (Zheng et al 2017)

Cada bloque contiene un timestamp (*sello de tiempo*), un valor hash del bloque anterior (parent) y un nonce *(número aleatorio)* que sirve como verificador del nuevo hash (operación criptográfica sobre timestamp, nonce, parent y el nuevo valor). Debido a que cada nuevo bloque de transacciones debe de cumplir con un mecanismo de consenso, el cual según Swan 2015, "Es el proceso en el cual la mayoría (o en algunos casos todos), los integrantes de una red de validadores están de acuerdo en el estado del ledger". Gracias a esto las nuevas transacciones no son agregadas al ledger automáticamente, si no que ocurre un cierto tiempo hasta que se realiza el mecanismo de consenso y al ser validadas son guardadas en el blockchain.

Hasta nuestros días podemos identificar tres etapas o fases de evolución de blockchain (Burgess and Colangelo 2015, Swan 2015, Zhao et al 2016):

1. moneda digital
2. economía digital
3. sociedad digital

Moneda digital se refiere a la tecnología fundamental de la plataforma como lo es minería, hashing *(algoritmos)*, el ledger público, y el protocolo de software que habilita las transacciones y la moneda digital en sí (Burgess and Colangelo 2015).

Desde hace más de 20 años fue propuesto el concepto de economía digital (Tapscott 1997) y hasta nuestros días, es que se cuenta con una plataforma adecuada para su funcionamiento. La economía digital se refiere al amplio rango de economías y aplicaciones financieras que van más allá de simples pagos, transferencias y transacciones. Tales aplicaciones incluyen instrumentos bancarios tradicionales como los préstamos y las hipotecas, instrumentos financieros complejos como las acciones y bonos, instrumentos legales como los títulos de propiedad, contratos y propiedades o bienes que pueden ser monetizados (Burgess and Colangelo 2015).

El concepto de sociedad digital se refiere al gran número de aplicaciones que comprenden los términos de dinero, comercio, mercados financieros, u otras actividades que incluyen conceptos de salud, ciencia, gobierno, educación, entre otros aspectos de cultura y comunicación (Burgess and Colangelo 2015).

2.1.2 Smart Contracts

Los Smart Contracts, son códigos que corren sobre la blockchain para facilitar, ejecutar y asegurar el cumplimiento de términos de un acuerdo. El principal objetivo de un Smart Contract es correr automáticamente las condiciones especificadas de un convenio. Debido a esto, las tarifas de transacciones son bajas ya que, en comparación con los sistemas tradicionales, los Smart Contracts no dependen de terceros para garantizar el cumplimiento de un contrato. Un Smart Contract puede considerarse como un sistema que libera activos digitales para todos o algunos de los involucrados, una vez que las reglas predefinidas hayan sido cumplidas.

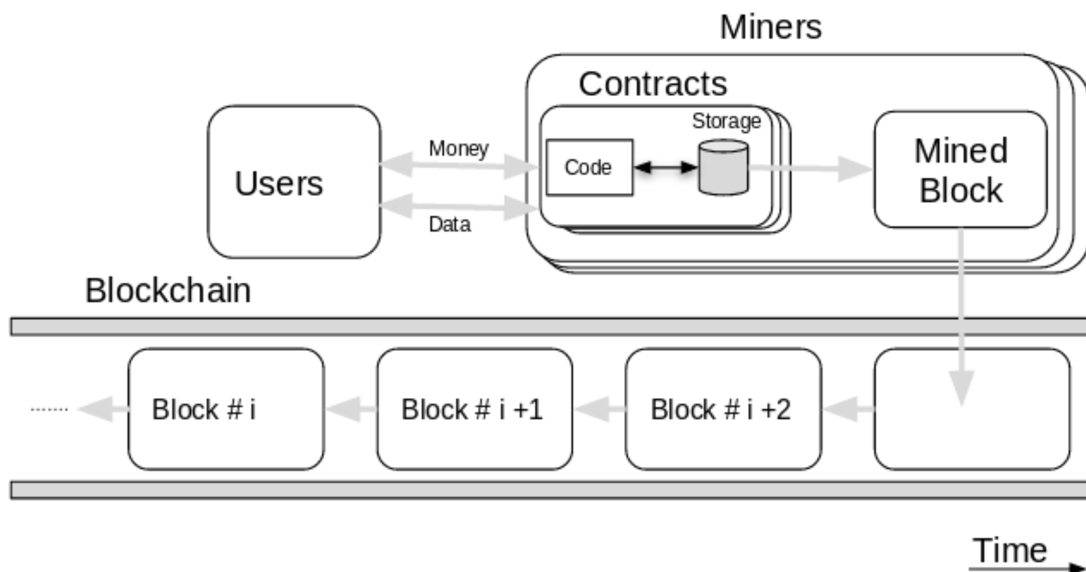

Figura 2.2 Sistema de Smart Contract (Delmolino et al 2016)

En la Fig. 2.2 se muestra un Smart Contract sobre blockchain. Cada contrato será asignado a una dirección única. Una vez desplegado el contrato en la blockchain ese no podrá ser modificado. Para correr un contrato, los usuarios únicamente tendrán que enviar una transacción a la dirección del contrato. Esta transacción será ejecutada por el mecanismo de consenso (mineros) y después de esto será actualizado el estado del contrato. El contrato puede, en función de la transacción que recibe, leer o escribir en su almacenamiento privado, guardar dinero en su saldo de cuenta, enviar o recibir mensajes, o dinero de usuarios u otros contratos e incluso crear nuevos contratos (Alharby and Moorsel 2017).

2.1.3 Bitcoin

Bitcoin es un sistema de pago electrónico que permite que dos partes realicen transacciones con dinero digital entre sí, de una manera segura sin pasar por un intermediario (por ejemplo, un banco). Las transacciones ocurridas son validadas por un mecanismo de consenso (nodos especiales llamados mineros) (Alharby and Moorsel 2017).

Bitcoin es un raro caso en donde la práctica parece estar primero que la teoría (Narayanan et al 2016). Las principales ventajas de esta criptomoneda son:

- Ofrece la posibilidad de reducir en gran medida los cobros por transacciones en compras on-line.
- Proporciona mejor anonimato que las tarjetas de crédito. Las cuentas son pseudónimos y el protocolo fomenta el uso de nuevos números de cuentas para cada transacción (Moore 2013).
- El diseño descentralizado protege de inflaciones a Bitcoin y a otras monedas. Las monedas tradicionales confían la regulación del suministro de dinero en bancos centrales. En contraste, Bitcoin usa criptografía para garantizar un

suministro de dinero relativamente fijo, al cual se le permite crecer en intervalos regulares (Moore 2013).

Como ya se ha mencionado, Bitcoin es una plataforma pública blockchain que se puede usar para procesar transacciones de criptomonedas, pero con una capacidad de cómputo muy limitada ya que usa un lenguaje scripting basado en pilas (Lewis 2016).

Es posible realizar validaciones de transacciones que requieren lógicas simples. Sin embargo, escribir contratos con lógicas complejas no es posible debido a las limitaciones del lenguaje de scripting, como por ejemplo no es posible escribir ciclos, la única forma de realizar esta tarea es repetir el código muchas veces, lo cual es ineficiente (Buterin 2018).

2.1.4 Ethereum

Es una plataforma opensource descentralizada basada en blockchain la cual maneja Smart Contracts, originalmente fue propuesta por Buterin (Buterin 2018). Ethereum puede ser comprendido como una extensión de Bitcoin, pero con un alcance más amplio de aplicaciones (Nofer et al 2017). Esta plataforma tiene su propia moneda llamada Ether (definida como ETH) la cual juega un papel secundario. El rol primario de ETH es, proporcionar una plataforma y un lenguaje que cuente con la suficiente flexibilidad expresiva para describir cualquier contrato ejecutable criptográficamente (Koblitz and Menezes 2015).

ETH es adecuado para construir sistemas económicos en software puro. Es decir, software para lógica de negocios, donde las personas (usuarios) pueden mover dinero (datos que representan valor) con la velocidad y a escala que normalmente obtenemos con los datos. Estas transacciones corren sobre una máquina virtual de ETH (EVM del inglés Ethereum Virtual Machine).

EVM es el entorno de ejecución para Smart Contracts en ETH. Los contratos son escritos en un lenguaje de programación de alto nivel llamado Solidity, y estos son compilados

en bytecode usando como intérprete a la EVM. Después el bytecode ejecutable es cargado en la Blockchain usando un cliente de ETH. EVM está diseñado para estar completamente aislado del entorno y resto de la red. El código que se ejecuta dentro de EVM no interviene en la red ni a ningún otro proceso; sólo después de compilarse en bytecode, los contratos tienen acceso al mundo externo y a otros contratos (Dhillon et al 2017).

2.1.5 DAO

Se puede decir que Bitcoin fue la primera organización autónoma descentralizada (DAO). La cual, creó un ecosistema de participantes en red que contribuyó con el poder computacional hacia un objetivo en común. El protocolo distribuido que proporcionaba un servicio financiero y recompensaba a los mineros, se convirtió en una organización descentralizada rudimentaria (Dhillon et al 2017).

Según Ralph Merkle (2016) la DAO es una entidad que posee propiedades internas de valor, las cuales pueden actualizar su estado, responder a los miembros y ejecutar Smart Contracts, siendo estas algunas de las funciones más básicas que cualquier DAO debería poder realizar.

La primera implementación de DAO dentro de la blockchain fue escrita como un contrato inteligente en el lenguaje de programación Solidity, el cual corre sobre ETH. El código puede ser utilizado por personas que trabajan de forma colaborativa fuera de un formato corporativo tradicional, también puede ser utilizado por una entidad registrada para automatizar reglas formales gubernamentales contenidas en los estatutos corporativos o impuestas por la ley (Jentzsch 2016).

El desarrollo de la primer DAO dentro de ETH fue guiado por los líderes (Christoph Jentzsch, CEO, y Stephan Tual, COO) de Slock.it, la cual es una empresa alemana dedicada a la innovación de la economía de intercambio por medio de una tecnología llamada Universal

Sharing Network (USN). En la construcción de la USN, Slock.it realizó un papel central en la adopción generalizada de la tecnología del internet de las cosas (IoT) dentro de la Blockchain. Al proporcionar un medio de interacción con dispositivos en la red desde cualquier lugar del mundo, la USN se convertiría en la columna vertebral de un mundo hiperconectado, donde cualquiera pudiera alquilar a otras personas sus propiedades o bienes sin la necesidad de compañías centralizadas como Uber o Airbnb.

En abril de 2016 fue lanzada una convocatoria para financiar el proyecto USN. La colecta se realizó por medio de un Smart Contract desarrollado en Solidity por Christoph Jentzsch. El retorno de inversión que se ofreció fue la toma de decisiones y el poder de votación en el rumbo de la plataforma por medio de tokens DAO, por cada Ether el inversionista recibió 100 tokens. A los 28 días del lanzamiento de la convocatoria se lograron recabar más de 150 millones de dólares esto a través de más de 11 mil inversores. En junio de 2016 Christian Reitwiessner (uno de los desarrolladores de Solidity), descubrió un fallo a través del cual los Smart Contracts podrían ser hackeados. Por otra parte, en el mismo mes Stephan Tual anunció que la DAO (siendo un contrato) no estaba en riesgo. Días más tarde la DAO fue atacada y sus fondos comenzaron a ser drenados (Dhillon et al 2017).

2.1.6 Ethereum Classic

El resultado del ataque que sufrió la DAO de Slock.it se traduce en el robo de 50 millones de dólares, esto es casi el 5% de Ethers que circulaban en ese momento. Esto contrajo implicaciones para todo el ecosistema de ETH y dio lugar a uno de los temas más debatidos de la Blockchain.

Por un lado del debate estaban aquellos que buscaban proteger el ecosistema de ETH, no les preocupaba la desintegración de la DAO, si no más bien asegurar que ETH sobreviviera como una plataforma de Blockchain confiable. Y por el otro lado, aquellos idealistas en la

descentralización e inmutabilidad, que creían que sobrescribir la cadena para revertir el ataque comprometería la integridad de la Blockchain.

En julio del 2016 el 90% de la comunidad de ETH voto por que los fondos de DAO se devolvieron a los inversores como si la organización nunca hubiera existido y por la creación de un fork de ETH lo que condujo a la aparición de Ethereum Classic (ETC). A pesar de que ETC a tenido una apreciación menos rápida que ETH o Bitcoin, los desarrolladores de ETC han acelerado su partida de ETH como plataforma con el lanzamiento de Mantis, el primer cliente creado desde cero para ETC a diferencia de Mist, Parity y otros clientes para ETH (Dhillon et al 2017).

2.2 Software en la cadena de bloques

Existen 3 partes fundamentales para el funcionamiento de una cadena de bloques: Mineros, Nodos y Wallets, a continuación, se describe cada una de ellas:

2.2.1 Mineros

La blockchain registra las transacciones realizadas en bloques. Un bloque válido está compuesto por operaciones criptográficas que contiene el hash del bloque que lo precede, un hash del bloque actual y la dirección donde se depositará una recompensa por resolver la operación hash. Este proceso es conocido como minería.

Un minero es un ente que puede agregar un bloque válido en la blockchain después de ser validado por un consenso (al menos el 51% de la red de mineros), y recibir una retribución en criptomonedas por haber resuelto la operación criptográfica (Swan 2015).

2.2.2 Nodos

La estructura de la blockchain está basada sobre una arquitectura peer-to-peer (*red entre pares*). Esta arquitectura se basa en el principio de que cada integrante (equipo de cómputo) de la red es igual entre sí. En la blockchain cada ente que forma parte de la red se le conoce como nodo.

Los nodos mantienen constantemente copias actualizadas del ledger completo de la criptomoneda. La interconexión de los nodos forma una red plana que no tiene servidores centrales ni jerarquías dentro de la red (Swan 2015).

2.2.3 Wallets

Un wallet es una infraestructura esencial para las criptomonedas, ya que es el mecanismo responsable de la tenencia segura, de la transferencia de criptomonedas y de cualquier activo criptográfico.

El software wallet también puede mantener una copia de la cadena de bloques conteniendo el histórico de todas las transacciones que se han producido en la criptomoneda (Swan 2015).

2.2.4 Mantis

Es un cliente terminal construido desde cero por un grupo de desarrolladores de ETC; por medio del proyecto mantis esta criptomoneda ha querido diferenciarse y ser más que un "cut-and-paste" de ETH. Desde que ETC surgió como el resultado del ataque a la DAO de Slock.it, las dos plataformas de contratos han compartido el código base y las herramientas para la construcción de contratos.

En enero del 2017 Alan McSherry, desarrollador y líder del proyecto Mantis anunció la versión beta del nuevo cliente para ETC. Este software pretende colocar a ETC como una alternativa viable sobre ETH. Mantis fue desarrollado en el lenguaje de programación funcional Scala. Este cliente pretende dar el reconocimiento a la comunidad de ETC por tener su propio equipo de desarrollo al igual que ETH lo tiene y establecer bases para posteriores innovaciones en ETC (Castor 2017).

El primer lanzamiento de mantis comprende las siguientes características e hitos funcionales (Input Output. (n.d.). Mantis – Ethereum Classic Beta Release):

- Descarga de la cadena de bloques
- Ejecución de transacciones
- Interfaz para comandos y consultas
- Minería
- Conexión con la interfaz Mist
- Multiplataforma
- Testnet y soporte a cadenas privadas
- Documentación de configuración

2.2.5 Scala

Scala es un lenguaje estáticamente tipado que se ejecuta sobre la máquina virtual de Java. Este lenguaje tiene una interoperabilidad perfecta con Java lo que le permite el uso de la mayoría de sus librerías. Sintácticamente, Scala es lo suficientemente conciso para construir APIs con menos código que Java, esto debido a su modelo de programación que ofrece un mayor nivel de abstracción. Scala es considerado como un lenguaje objeto-funcional, que ofrece el uso de la programación orientada a objetos con características funcionales, permitiendo a los desarrolladores el uso de constructores, objetos y traits.

Con el aumento en el uso de multiprocesadores, las aplicaciones generalmente se desempeñan mejor en lenguajes que cuentan con cierto grado de paralelismo. Los mensajes asíncronos, son una manera efectiva usada por los desarrolladores para solventar el cómputo distribuido. Java cuenta con un gran número de librerías basadas en concurrencia. Scala hace uso de estas bibliotecas y además implementa un modelo de paralelismo basado en Erlang (Ghosh et al 2011).

Dicho por Alan McSherry, Scala fue elegido para el desarrollo de Mantis por la facilidad de realizar pruebas y la predictibilidad del lenguaje, características que permiten a los desarrolladores auditar el código en busca de errores y fallas de seguridad más fácilmente que en otros idiomas. "Si tiene un código más predecible, se aprovechará de la calidad general del producto” (Castor 2017).

2.3 Análisis de dependencias

Los objetivos del análisis de dependencias son: reflejar la estructura del sistema, resaltar patrones y relaciones de código a través de diversas áreas de la aplicación (Arias et al 2011).

2.3.1 Análisis Estático y Dinámico

La información del análisis estático es recolectada de las importaciones y llamadas de clases o métodos del código fuente del programa, cuando este no se encuentra ejecutándose. Ahora bien, el análisis dinámico utiliza datos basados en código fuente en forma de rastros de ejecución; un rastro de ejecución puede ser identificado como una función, procedimiento o método que ha sido llamado, la información de este tipo de análisis es recaba mediante

técnicas de instrumentación del código fuente, estudios de la plataformas y análisis en el compilador (Arias et al 2011).

2.3.2 Gráfico de llamadas (Call Graph)

Una dependencia de software es una relación directa entre dos piezas de código. En general estas dependencias se separan en dos tipos: de datos entre la definición y uso de valores, y de llamadas entre la declaración de una función y las partes en las que es llamada.

Uno de los conceptos más usados en el análisis de dependencias es la medición de centralidad. Esta medida es usada para identificar los métodos o clases que son el objetivo de muchas dependencias. Existen diferentes tipos de centralidad:

- Grado de centralidad: Mide el número de dependencias de un método o clase.
- Grado de proximidad: Toma las distancias de todas las conexiones de dependencias.
- Grado de intermediación: Mide el número de veces que un nodo actúa como un puente entre el camino más corto entre otros dos nodos.

Las piezas de código con más dependencias son más propensas a contener defectos ya que los errores son propagados (Zimmermann and Nagappan 2018).

Ahora bien, Call Graph es un gráfico que representa las relaciones de llamadas entre los procedimientos de un programa (Grove et al 1997). Esta técnica es fundamental para el análisis en términos de pruebas (*testing*), optimización de código y paralelización de procedimientos utilizados en compiladores, herramientas de verificaciones y en general en la comprensión de programas (Ali and Lhoták. 2012).

La construcción de este tipo de gráficos ha sido principalmente estudiada en lenguajes orientados a objetos tales como Java, C++ y Self, en funcionales como Scheme y de tipo scripting como JavaScript. Sin embargo, hasta la fecha no se han evaluado o propuesto

algoritmos para la construcción de Call Graphs para Scala. Es posible construir dichos gráficos a partir del bytecode generado por el compilador de la Java Virtual Machine (JVM) y después con la ayuda de algún framework de análisis tal como WALA o SOOT construir el Call Graph (Ali et al 2014).

2.3.3 Java-callgraph

Es una suite de programas para la generación estática o dinámica de Call Graphs para Java. El análisis estático de esta herramienta se ejecuta sobre un empaquetado jar (*archivo que contiene el bytecode de las clases compiladas),* generando una tabla de relaciones entre métodos, interfaces y clases. Por otro lado, el análisis dinámico corre como un agente de Java sobre clases preestablecidas para obtener sus invocaciones y la ejecución genera las relaciones de llamadas entre métodos y el número de llamadas de los mismos (Gousiosg/java-callgraph 2018).

2.4 Técnicas de desarrollo de calidad de Software

El software Mantis, fue desarrollado con múltiples técnicas de aseguramiento de la calidad: Unit test (*pruebas unitarias*), Property-based testing (*pruebas de escenarios generados automáticamente*), Continuous integration (*integración continua*) y Auditorías de Seguridad Externas. A continuación, se describen las técnicas utilizadas en la construcción del nodo Mantis, así como las propuestas en este reporte técnico (Refinement types y Design by contract).

2.4.1 Pruebas (Testing)

Las pruebas (*testing*) son una práctica importante del desarrollo de software para asegurar el correcto funcionamiento del código. La escritura de pruebas varía enormemente dependiendo del ámbito en el que se aplican, desde los rigurosos estándares de la industria aeroespacial, hasta sistemas en los que se descuidan las comprobaciones y se deja que el usuario final descubra los errores.

El mayor reto de testing es mantener una calidad aceptable del software sin afectar el costo del proyecto, ya que el codificar las pruebas generalmente equivale al 50% del costo. La manera más costosa de desarrollar comprobaciones de código es realizarlas manualmente, en la práctica, los desarrolladores automatizan sus pruebas para escribirlas una vez y luego ejecutarlas muchas veces. Existen herramientas (Unit test) y prácticas (Test driven development) que ayudan con la automatización de esta tarea, aún así, escribir un caso de prueba para cada escenario posible no es económico (Hughes 2010).

2.4.2 Prueba Unitaria (Unit Test)

Una prueba de unidad ó prueba unitaria, es un programa que verifica el comportamiento de una porción de código. Una unidad es la parte más pequeña comprobable de un sistema. Las pruebas unitarias están formadas por tres secciones: datos de prueba, secuencias de métodos y aserciones.

Las pruebas unitarias son usadas para documentar requerimientos del cliente, decisiones de diseño, verificación de errores por cambios, y lo más importante, son parte del proceso de pruebas (testing) que busca producir una alta cobertura de código que dé certidumbre a las acciones que debe realizar el software (Tillmann et al 2010).

Unit test es el nivel más básico para probar un proyecto, este tipo de comprobaciones ejecutan el código base mandando parámetros a las funciones, para recorrer cada una de las posibles ramas de las mismas. De esta manera, las pruebas unitarias cubren un cierto porcentaje del código. A esto se le conoce como Code Coverage (Li 2017).

2.4.3 Property-based Testing

Property-based testing (PBT) es una técnica para realizar pruebas de código, que generan entradas aleatorias a una propiedad dada, verificando el comportamiento de la misma. Las propiedades pueden variar desde ecuaciones algebraicas simples, hasta modelos complejos de máquinas de estados (Aichernig and Schumi 2017).

Esta técnica es una alternativa al desarrollo manual de pruebas, o para describir un modelo formal de un sistema o componente. PBT usa declaraciones para establecer propiedades que el software debe de realizar de acuerdo con su especificación. Desde este planteamiento varios casos de prueba son generados automáticamente para cada propiedad descrita, sin la necesidad de realizarlos manualmente (Hughes 2010).

2.4.4 Integración continua (Continuous Integration)

Su nombre por sí mismo describe el propósito de este principio de ingeniería de software. Durante el ciclo de vida de un proyecto, continuamente se integran partes de código que deben ser probadas para asegurar que las nuevas piezas, no rompan el funcionamiento de todo el proyecto (Pouclet 2014).

Dentro de las principales ventajas de usar integración continua se encuentran:

- La retroalimentación, esto es si la porción de código a integrar rompe el conjunto de pruebas, es relativamente sencillo identificar los problemas que originaron el fallo y con esto se ahorra tiempo de depuración.
- Diagnosticar fallos en etapas tempranas.
- Asegurar el funcionamiento del proyecto al agregar nuevas partes de código.

El nodo Mantis utiliza la herramienta de CircleCI como CI, dicho por el Senior Engineering Manager de este software Jhonny Sheeley: “CircleCI triplicó la eficiencia de nuestro equipo. Con CircleCI, la gente puede probar diferentes idiomas, trabajar con el código de otra persona, establecer tantas construcciones como necesite para usted y mucho más. CircleCI hace a nuestro equipo mucho más feliz sobre lo que están construyendo” (https://circleci.com/).

2.4.5 Tipos refinados (Refinement types)

El uso de lenguajes de programación estáticamente tipados, mejora a la productividad de los programadores. Una razón obvia es que reduce dramáticamente la cantidad de tiempo gastado en la depuración de errores, que comúnmente son detectados en tiempo de compilación. Una razón aún más fundamental, es que los programadores pueden usar tipos para guiar el entendimiento de la estructuración de una pieza de código, durante el desarrollo y mantenimiento del mismo. Una propuesta para incrementar los beneficios del tipado estático es, ampliar un lenguaje existente para que cada tipo ordinario sea extendido por tipos refinados, los cuales permitirán que propiedades de programas comunes sean expresadas y revisadas (Davies 1997).

Los lenguajes estáticamente tipados son muy efectivos para capturar la estructura básica de una parte de código, pero generalmente los programas involucran muchas propiedades importantes que no son captadas por los tipos comunes. Por ejemplo, una función en particular necesita parámetros que sean mayores a cero o que el valor de retorno no sea cero, para este caso, los lenguajes de programación no proveen un tipo de dato numérico diferente a cero. Tales invariantes generalmente críticas para el funcionamiento de un programa están únicamente en comentarios de manera informal.

El concepto de tipos refinados fue implementado por primera vez por Freeman and Pfenning en el lenguaje de programación funcional ML. El uso de este paradigma no requiere la alteración de los tipos primarios de un lenguaje, si no en agregar nuevas características que son apropiadas para expresar y verificar propiedades, las cuales heredan las estructuras de los tipos de los cuales derivan (Davies 2005).

2.4.6 Diseño por contrato (Design By Contract)

Design By Contract (DBC) es un enfoque de diseño de software en el cual la relación entre proveedor y cliente, es vista como un contrato formal en el que cada parte tiene derechos y obligaciones (Meyer 1997).

DBC fue popularizado por Bertrand Meyer e incluido en su lenguaje de programación Eiffel. En las últimas décadas DBC comenzó a usarse en más lenguajes de programación, ya sea a través de soporte nativo o con terceros.

Diseñar contratos puede reducir errores y por ende mejorar la confiabilidad del software (Meyer 1992). Gracias a las especificaciones (Precondiciones, Postcondiciones e Invariantes) dadas en el propio código, los compiladores pueden realizar aserciones. Otro beneficio del uso de este paradigma es que los programadores tienen un mejor entendimiento de lo que una porción de código realiza o debe de realizar (Feldman 2003).

Las aserciones claves en DBC se definen como precondiciones, postcondiciones e invariantes.

- Una precondición es un requisito o requisitos de entrada que deberán ser verdaderos para poder invocar un método.
- Una postcondición es una aserción que deberá de garantizar ciertas condiciones de salida.
- Una invariante se define como la inmutabilidad en las propiedades de una clase a pesar de la ejecución de un método.

Según Mitchell et al (2002), la técnica de DBC cuenta con seis principios:

1. Separar consultas de comandos.
2. Separar consultas básicas de consultas derivadas.
3. Por cada consulta derivada, escribir una postcondición que especifique que resultado será devuelto en términos de una o más consultas básicas.
4. Por cada comando, escribir una postcondición que especifique el valor afectado por medio de una consulta básica.
5. Establecer una precondición adecuada para cada consulta y comando.
6. Escribir invariantes para definir propiedades de objetos inmutables.

2.4.7 Liskov Substitution Principle (LSP)

LSP es un principio enfocado a la programación orientada a objetos y puede ser descrito como: "Un tipo debe ser sustituible por sus subtipos sin alterar el correcto funcionamiento de la aplicación" (Joshi 2016). Este principio es semejante al enfoque de DBC de Bertrand Meyer, ya que los subtipos siguen reglas de precondiciones, poscondiciones e invariantes (Roth 2017):

1. Las precondiciones de un tipo no pueden ser mas fuertes que las de un subtipo
2. Las postcondiciones de un tipo no pueden ser mas débiles que las de un subtipo
3. Todas las invariantes de un tipo deben de permanecer sin cambios en un subtipo

2.4.8 Métodos formales

Son procesos aplicados en el desarrollo de sistemas de software, sustentados por notaciones y pruebas matemáticas (Kuhn et al 2002). En ingeniería de software los métodos formales son utilizados en:

- Políticas de requerimientos. Pueden ser consideradas como las principales propiedades de seguridad que un sistema debe de preservar.
- Especificaciones. Descripciones formales (tablas de estado, o lógica matemática) del comportamiento de un sistema.
- Pruebas de correspondencia entre especificaciones y requerimientos. El sistema debe de mostrarse tal y como se describen las especificaciones, y deberá preservar las propiedades descritas en los requerimientos.
- Pruebas de correspondencia entre el código fuente y las especificaciones. Esto rara vez se realiza debido a costos y tiempo, pero puede realizarse para probar partes críticas del sistema.
- Pruebas de correspondencia entre código máquina y código fuente. Raramente se aplica esta técnica debido al costo y a la alta confiabilidad de algunos compiladores.

3 Experimento I: Análisis de dependencias

En casos en los que se necesita realizar optimizaciones y por consecuencia entender el funcionamiento del código de un proyecto, es de gran ayuda el uso de técnicas y/o herramientas de análisis de dependencias (Ali and Lhoták. 2012).

En el caso particular del cliente Mantis, a la fecha (abril del 2018) está compuesto por más de 50 mil líneas de código (datos obtenidos de github.com/input-output-hk/mantis), por lo que el uso de Call Graph se vuelve un factor fundamental para poder realizar una refactorización. Sin esta técnica sería muy difícil realizar cualquier optimización debido al tamaño del proyecto Mantis.

3.1 Descripción del problema

Debido a que las piezas de código con más dependencias son más propensas a contener defectos (Zimmermann and Nagappan 2018), nuestro problema es representar las relaciones de llamadas entre los procedimientos de Mantis y así poder obtener las clases o métodos con el mayor grado de centralidad, para después aplicar en ellas algunas técnicas de calidad de desarrollo de software.

3.2 Método implementado

El método que se uso para detectar cuales son los módulos de Mantis más críticos fue Call Graph, el cual es un gráfico que representa las relaciones de llamadas entre las clases o métodos de un programa (Grove et al 1997).

Como ya se ha mencionado, Mantis fue desarrollado en el lenguaje de programación funcional Scala. En la actualidad no existen herramientas para la construcción de gráficos de llamadas sobre este lenguaje, pero gracias a que Scala se ejecuta sobre la máquina virtual de Java (Ghosh et al 2011), fue posible construir un Call Graph a partir del bytecode generado por el compilador de la Java Virtual Machine (Ali et al 2014).

A continuación, se describe el procedimiento realizado en Scala para la generación del Call Graph de Mantis, utilizando el software java-callgraph con un análisis estático (Gousiosg/java-callgraph 2018):

1. Empaquetar el proyecto Mantis en un archivo jar

```
1 import java.io._
2 import sys.process._
3
4 "git clone https://github.com/input-output-hk/mantis.git".!!
5
6 val fw = new FileWriter("mantis/project/plugins.sbt", true)
7 try {
8   fw.write( "addSbtPlugin(\"com.eed3si9n\" % \"sbt-assembly\" %
        \"0.14.5\")")
9 }
10 finally fw.close()
11 val currentDirectory = new java.io.File(".").getCanonicalPath
12 val output = Process("sbt assembly", new File(currentDirectory+"/mantis
       ")).!!
13 s"mv $currentDirectory/mantis/target/scala-2.12/mantis-assembly-0.3-cli
       -beta.jar .".!!
```

Algoritmo 3.1Empaquetado de Mantis

2. Instalar el software java-callgraph

```
1 "git clone https://github.com/gousiosg/java-callgraph.git".!!
2 s"mvn install -f $currentDirectory/java-callgraph".!!
```

Algoritmo 3.2 Instalación de java-callgraph

3. Ejecutar el análisis y guardar los resultados en un archivo de texto

```
1 val pw = new PrintWriter(new File("call_graph_mantis.txt" ))
2 val result = s"java -jar $currentDirectory/java-callgraph/target/javacg
      -0.1-SNAPSHOT-static.jar mantis-assembly-0.3-cli-beta.jar".!!
3 pw.write(result)
4 pw.close
```

Algoritmo 3.3 Ejecución de análisis estático sobre empaquetado de Mantis

4. Filtrar los resultados y solo incluir llamadas a clases (c) que pertenezcan al proyecto Mantis

```
1 import scala.io.Source
2
3 val lines = Source.fromFile("call_graph_mantis.txt")
4 .getLines.toList
5 .filter(_.startsWith("C"))
6 .filter(_.contains("io.iohk.ethereum"))
7 .filter(!_.contains("$"))
8 val finvoked = new PrintWriter(new File("invokedclasses.csv" ))
9
10 for (x <- lines) {
11   var line = x.split(" ")
12   var left = line(0).substring(2)
13   var right = line(1)
14   if (!right.contains(left)) {
15       finvoked.write(left +","+ right + "\n")
16   }
17 }
18 finvoked.close
```

Algoritmo 3.4 Guardado de invocaciones de clases

5. Crear una base de datos e importar las llamadas de clases

```
1 val sql = """
2     create schema if not exists dbMantis;
3     set schema dbMantis;
4     create table if not exists callgraph (
5       pathClassSource varchar(70) not null,
6       pathClassTarget varchar(70) not null)
7       as select "source","target" from CSVREAD('./invokedclasses.csv','
            source,target', NULL);
8     create table if not exists classes (
9       id int auto_increment primary key,
10      nameSource varchar(40) not null,
11      pathClassSource varchar(70) not null)
12      as SELECT null,
13      SUBSTR(t. PATHCLASSSOURCE, LOCATE( '.', t. PATHCLASSSOURCE, -1) +
             1),
14      t.PATHCLASSSOURCE
15      from (SELECT PATHCLASSSOURCE FROM DBMANTIS.CALLGRAPH group by
            PATHCLASSSOURCE)t;
16    create table if not exists calls (
17      id int auto_increment primary key,
18      nameTarget varchar(40) not null,
19      nameSource varchar(40) not null,
20      pathClassTarget varchar(70) not null,
21      pathClassSource varchar(70) not null)
22      as SELECT null,
23      SUBSTR(PATHCLASSTARGET, LOCATE( '.', PATHCLASSTARGET, -1) + 1) ,
24      SUBSTR(PATHCLASSSOURCE, LOCATE( '.', PATHCLASSSOURCE, -1) + 1) ,
25      PATHCLASSTARGET,
26      PATHCLASSSOURCE
27      FROM
28      DBMANTIS.CALLGRAPH where PATHCLASSTARGET like '%ethereum%';
29  """
```

Algoritmo 3.5 Estructura de base de datos

```scala
import java.sql.{Connection, DriverManager}
def createDbStructure(conn: Connection, sql: String): Unit = {
  val stmt = conn.createStatement()
  try {
    stmt.execute(sql)
  }
  finally {
    stmt.close
  }
}
Class.forName("org.h2.Driver")
val conn: Connection = DriverManager.getConnection( "jdbc:h2:./mantis",
      "sa", "" )
try {
  createDbStructure(conn, sql)
}
finally {
  conn.close()
}
```

Algoritmo 3.6 Inserción de base de datos

6. Obtener las clases con mayor centralidad

```
1 import $ivy.`org.plotly-scala::plotly-jupyter-scala:0.3.1`
2 import plotly._
3 import plotly.element._
4 import plotly.layout._
5 import plotly.JupyterScala._
6 import scala.collection.mutable.ListBuffer
7
8 val conn: Connection = DriverManager.getConnection( "jdbc:h2:./mantis",
      "sa", "" )
9 val sqlIns = """SELECT
10   SUBSTR(PATHCLASSTARGET, LOCATE( '.', PATHCLASSTARGET, -1) + 1) AS
        target,
11   COUNT(ID) AS total
12   FROM DBMANTIS.CALLS
13   GROUP BY PATHCLASSTARGET ORDER BY COUNT(ID) DESC, PATHCLASSTARGET"""
14 val stmTop = conn.prepareStatement(sqlIns)
15
16 var x = new ListBuffer[String]()
17 var y = new ListBuffer[String]()
18
19 try {
20   val result = stmTop.executeQuery()
21   while (result.next()) {
22       x+=result.getString("target")
23       y+=result.getString("total")
24   }
25 }
26 finally {
27   stmTop.close()
28   conn.close()
29 }
30
31 plotly.JupyterScala.init()
32 Bar(x, y).plot()
```

Algoritmo 3.7 Gráfico de clases con mayor grado de centralidad

3.3 Resultados

Aplicando el procedimiento descrito obtuvimos el grafo completo del proyecto Mantis. En la Fig. 3.1 se muestran las 10 clases con mayor grado de centralidad, siendo BlockHeader y PV62 las clases con mayor número de llamadas (21).

La Fig. 3.2 nos muestran un grafo de conexiones de primer y segundo nivel que guardan las clases con BlockHeader.

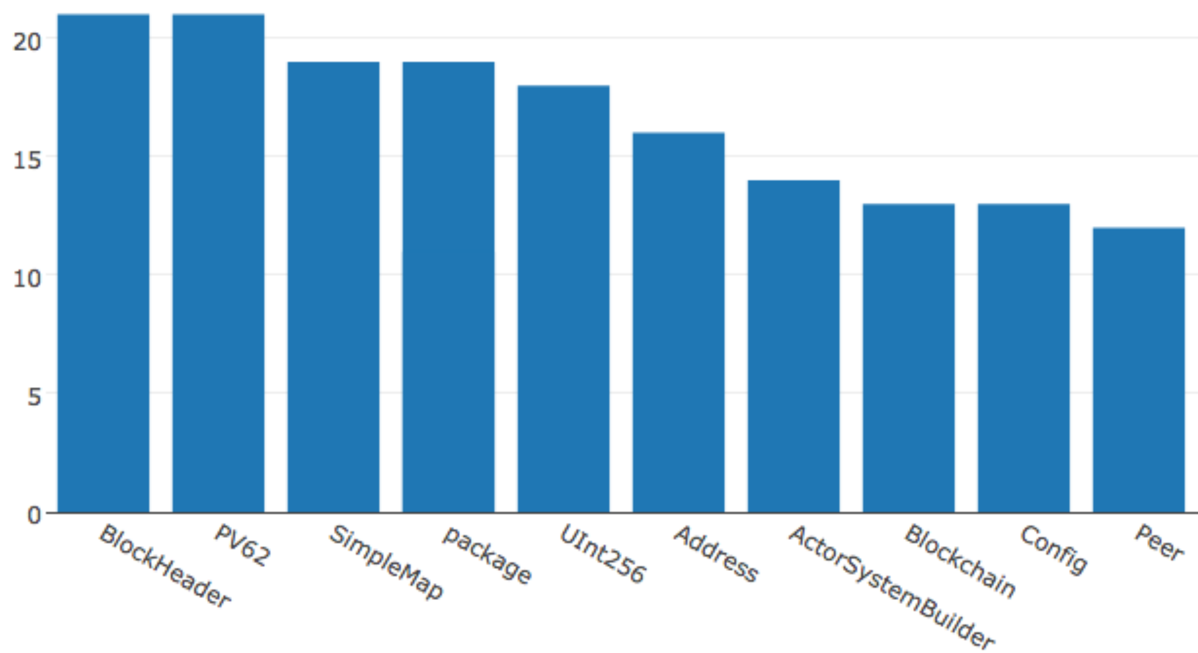

Figura 3.1 Top 10 de clases con mayor grado de centralidad

Figura 3.2 Zoom de Call Graph mostrando las conexiones a la clase BlockHeader

3.4 Conclusiones

El procedimiento aplicado fue diseñado para poder ser reproducido sin ninguna alteración para futuras actualizaciones del proyecto. Los resultados obtenidos nos dan una clara idea sobre las clases candidatas a aplicar refactorización, todo esto con el objetivo de aumentar la calidad del cliente Mantis.

4 Experimento II: Aplicación de técnicas de calidad en el software

El desarrollar soluciones de software de alto rendimiento, confiables y seguras es un gran problema en la industria del software (Davies et al 2004).

La programación de pruebas es una práctica importante para asegurar la calidad del software. Dicho por Hughes (2010), generalmente el costo de las codificaciones de pruebas es de un 50% del total del proyecto, debido a la codificación manual y a la generación de pruebas de diversos escenarios.

A la fecha el proyecto de Mantis cuenta con un code coverage del 83%, este porcentaje se logra al aplicar técnicas como PBT, Unit Test, CI y auditorías externas de seguridad.

4.1 Descripción del problema

Con el propósito de mejorar la calidad del software Mantis es que proponemos el uso de los paradigmas DBC y Refinement types. Por una parte, Design by Contract puede reducir errores y por ende mejorar la confiabilidad del software (Meyer 1992), y por otro lado la aplicación de la técnica Refinement types para agregar nuevas características que son apropiadas para expresar y verificar propiedades sobre tipos primarios y así, mejorar la productividad de los programadores (Davies 1997).

Existen otras metodologías o técnicas que nos podrían ayudar a incrementar la calidad del software, como, por ejemplo: el principio de sustitución de Liskov (LSP) o métodos formales. En este reporte técnico descartamos el uso de métodos formales debido a que utilizarlos podría resultar extremadamente costoso (Kuhn et al 2002). Y decidimos no utilizar

LSP debido a que la clase BlockHeader no hereda de una clase padre, por lo que el principio de sustitución no puede ser aplicado (Roth 2017).

Las instancias creadas de la clase BlockHeader son construidas por medio de un constructor inmutable (case class), el cual tiene una estructura basada en el Yellow Paper de ETH.

```
1 case class BlockHeader(
2   parentHash: ByteString,
3   ommersHash: ByteString,
4   beneficiary: ByteString,
5   stateRoot: ByteString,
6   transactionsRoot: ByteString,
7   receiptsRoot: ByteString,
8   logsBloom: ByteString,
9   difficulty: BigInt,
10  number: BigInt,
11  gasLimit: BigInt,
12  gasUsed: BigInt,
13  unixTimestamp: Long,
14  extraData: ByteString,
15  mixHash: ByteString,
16  nonce: ByteString) {
```

Algoritmo 4.1 Constructor de BlockHeader

Algunos atributos de la clase como parentHash, ommerHash, beneficiary entre otros mostrados en el algoritmo 4.1 son de tipo ByteString, esta clase es una secuencia inmutable de bytes (https://doc.akka.io/japi/akka/2.4.0/akka/protobuf/ByteString.html).

El tipo de dato ByteString puede contener cualquier secuencia de dato de tipo string, por lo que la generación de nuevas instancias no es validada según las reglas descritas en el Yellow Paper de ETH.

4.2 Método implementado

La aplicación de las dos técnicas se realizó con base a las reglas descritas en el Yellow Paper de Ethereum. A continuación, se describe el trabajo realizado sobre la clase de mayor centralidad.

4.2.1 Refinement types

Al conocer la naturaleza del tipo ByteString proponemos la creación de tipos refinados, los cuales contengan las definiciones formales del protocolo ETH. En esta etapa solo se actualizan los atributos de tipo keccak hash de 256, 160 y 64 bit para que estos sean formados por cadenas hexadecimales de 64, 40 y 16 caracteres respectivamente.

```
1 //Refinement
2 type Hash256 = MatchesRegex[W.‘"[0-9a-fA-F]{64}"‘.T]
3 type Hash160 = MatchesRegex[W.‘"[0-9a-fA-F]{40}"‘.T]
4 type Hash64 = MatchesRegex[W.‘"[0-9a-fA-F]{16}"‘.T]
```

Algoritmo 4.2 Tipos refinados de hash 256, 160 y 64 bit

La clase BlockHeader tiene un constructor implícito (algoritmo 4.1) el cual no es posible modificar para aplicar los nuevos tipos generados en lugar de ByteString. Este factor limitó la validación de tipos por parte del compilador.

Como se mencionó en el apartado de 2.4.5 los errores generados por tipos refinados son detectados en tiempo de compilación (Davies 1997). Debido a esto, es que optamos por utilizar la función de *“refineV”,* dicho método provee la funcionalidad de revisar tipos refinados en tiempo de ejecución.

```
1 refineV[Hash256](Hex.toHexString(attr.toArray[Byte]))
```

Algoritmo 4.3 Función para validad tipos refinados en tiempo de ejecución

Creadas las extensiones de tipos mostradas en el algoritmo 4.2, se creó una función *"valref"* (algoritmo 4.4) para validar los atributos ingresados en el constructor de BlockHeader.

4.2.2 Design By Contract

La técnica DBC está compuesta por las aserciones de precondiciones, postcondiciones e invariantes (Feldman 2003). El enfoque propuesto en esta etapa es agregar precondiciones dentro del constructor de la clase de mayor grado de centralidad.

Siguiendo el quinto principio de DBC "Establecer una precondición adecuada para cada consulta y comando" (Mitchell et al 2002), realizamos los siguientes pasos:

- A partir de la generación de tipos refinados, se agrega una función para validar los diferentes tipos de hash definidos en el Yellow Paper de ETH.

```
1 def valref(attr: ByteString, tkeccak: Int) : Boolean = {
2   tkeccak match {
3     case 256 => refineV[Hash256](Hex.toHexString(attr.toArray[Byte])).
          isRight
4     case 160 => refineV[Hash160](Hex.toHexString(attr.toArray[Byte])).
          isRight
5     case 64 => refineV[Hash64](Hex.toHexString(attr.toArray[Byte])).
          isRight
6     case _ => throw new NoSuchMethodException
7   }
8 }
```

Algoritmo 4.4 Función valref

- Para simplificar la aserción de cada atributo se crea un método que evalúa de manera dinámica dichos resultados.

```
1 def booleanToMap(eval: Boolean): Either[BHInvalid, BHValid] = {
2   if (eval == true)
3     Right(BHValid)
4   else
5     Left(BHInvalid)
6 }
```

Algoritmo 4.5 Función booleanToMap

- Se agrega una función para validar todos los atributos incluidos en la creación de BlockHeader y se descartan logsBloom y timeStampLinux, ya que estos últimos no cuentan con una validación descrita.

```
1 def validateConstructor(): Either[BHInvalid, BHValid] = {
2   for {
3     //Based on stated in section 4.4 of http://paper.gavwood.com/
4     _ <- booleanToMap(valref(parentHash, 256))
5     //Based on stated in section 4.4 of http://paper.gavwood.com/
6     _ <- booleanToMap(valref(ommersHash, 256))
7     //Based on stated in section 4.4 of http://paper.gavwood.com/
8     _ <- booleanToMap(valref(beneficiary, 160))
9     //Based on stated in section 4.4 of http://paper.gavwood.com/
10    _ <- booleanToMap(valref(stateRoot, 256))
11    //Based on stated in section 4.4 of http://paper.gavwood.com/
12    _ <- booleanToMap(valref(transactionsRoot, 256))
13    //Based on stated in section 4.4 of http://paper.gavwood.com/
14    _ <- booleanToMap(valref(receiptsRoot, 256))
15    //Based on validation stated in section 4.4.2 of http://paper.
          gavwood.com/
16    _ <- booleanToMap(difficulty >=0)
17    //Based on validation stated in section 4.4.2 of http://paper.
          gavwood.com/
18    _ <- booleanToMap(number >=0)
19    //Based on validation stated in section 4.4.2 of http://paper.
          gavwood.com/
20    _ <- booleanToMap(gasLimit >= minGasLimit && gasLimit <=
         maxGasLimit)
21    //Based on validation stated in section 4.4.2 of http://paper.
          gavwood.com/
22    _ <- booleanToMap(gasUsed >=0 && gasUsed <= gasLimit)
23    //Based on validation stated in section 4.4.2 of http://paper.
          gavwood.com/
24    _ <- booleanToMap(extraData.length <= MaxExtraDataSize)
25    //Based on stated in section 4.4 of http://paper.gavwood.com/
26    _ <- booleanToMap(valref(mixHash, 256))
27    //Based on stated in section 4.4 of http://paper.gavwood.com/
28    _ <- booleanToMap(valref(nonce, 64))
29  } yield BHValid
30 }
```

Algoritmo 4.6 Función validateConstructor

- Por último, agregamos la precondición por medio del método *"require"*, la cual evaluará que todas las aserciones sobre los atributos sean cumplidas al momento de crear una nueva instancia de la clase BlockHeader.

```
1 require(validateConstructor == Right(BHValid))
```

Algoritmo 4.7 Aserción de precondición

Solo se sigue el principio 5 de DBC (Mitchell et al 2002), ya que en la clase no existen comandos, y no es necesario la validación de invariantes debido al uso de variables inmutables en la clase BlockHeader.

4.3 Resultados

La aplicación de las técnicas Refinement types y DBC sobre Mantis, por medio de las funciones mostradas anteriormente, generaron errores en las pruebas desarrolladas del proyecto, esto debido a que no existía una validación para la conformación del constructor de la clase BlockHeader.

Los errores generados por la refactorización realizada, fueron corregidos según las reglas del Yellow Paper de ETH para la formación del BlockHeader, dejando el mismo porcentaje de code coverage (83%).

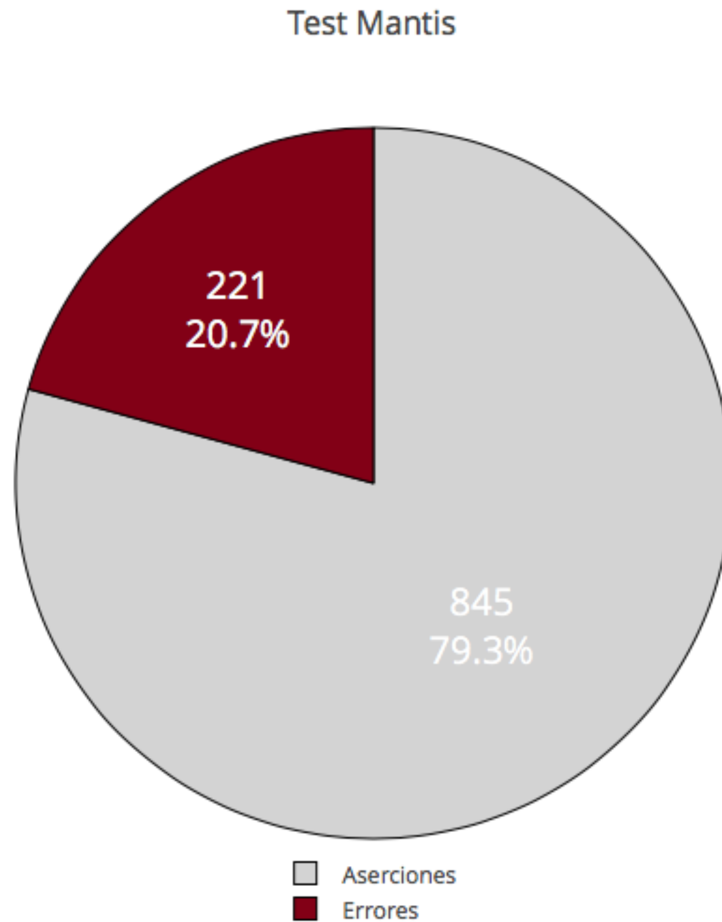

Figura 4.1 Resultados de pruebas al aplicar DBC y Refinement types

4.4 Conclusiones

A pesar de que el proyecto Mantis cuenta con una serie de técnicas de aseguramiento de calidad como los son: property-based testing, unit test y continuous integration; la implementación de DBC y Refinement types descritas en esta etapa, arrojaron resultados positivos en la búsqueda de mejorar la calidad del cliente Mantis. Ya que las pruebas existentes hasta antes de la refactorización realizada no aseguraban que el contenido de los atributos de la clase fueran formados según las especificaciones del Yellow Paper de ETH.

Con la aplicación de las técnicas propuestas se logra validar las nuevas instancias de BlockHeader que seguirán las reglas descritas en su documentación formal.

5 Discusión de los resultados

Como resultado de la aplicación de call graph sobre Mantis se obtuvo la clase con mayor grado de centralidad BlockHeader, la cual fue candidata a refactorizar (Fig. 3.1). Otras clases candidatas como PV62 o SimpleMap, fueron descartadas debido a que éstas invocan a la clase BlockHeader (en tiempo de ejecución) y no cuentan con un constructor o clase donde se pudiera aplicar DBC o Refinement types.

Además del análisis estático aplicado por la suite java-callgraph, se probó un análisis dinámico con la misma herramienta y se usaron las aplicaciones WALA y SOOT, pero sin tener resultados positivos debido a la incompatibilidad de las herramientas con el bytecode de Mantis.

El nodo Mantis tiene un total de 785 conexiones entre sus clases. Dando un zoom sobre el gráfico de nodos se extrae la Fig. 3.2, la cual nos muestra las relaciones de primer y segundo nivel que guardan las clases con BlockHeader.

Como se muestra en la Fig. 4.1, a la fecha (abril del 2018) el cliente terminal Mantis cuenta con un total de 1,066 pruebas comprendidas por las técnicas Unit test y Property-based testing. Al aplicar los paradigmas de Refinement types y DBC se obtuvo un total de 221 fallos y 845 pruebas acertadas. Las pruebas que no cumplieron las aserciones fueron editadas según las reglas aplicadas en el constructor BlockHeader. En total se modificaron 10 archivos de pruebas para continuar con el 83% de code coverage con el que contaba el nodo antes de la intervención de las técnicas aplicadas.

6 Conclusiones

Producto de los resultados obtenidos en este reporte técnico se concluye lo siguiente:

6.1 Objetivo general

Por medio de la implementación de las técnicas de diseño Refinement types y DBC, se logró aumentar la calidad del nodo Mantis validando el constructor de la clase BlockHeader. Agregando unas pocas líneas de código que implementa las técnicas descritas, aseguramos que la construcción de la cabecera de bloque siga las reglas del Yellow Paper de ETH.

6.1.1 Objetivos específicos

Con respecto a los objetivos específicos:

- Objetivo 1: Las clases de mayor grado de centralidad en el nodo Mantis son: BlockHeader, PV62 y SimpleMap, este resultado se obtuvo al aplicar la técnica de call graph por medio de un análisis estático con la herramienta java-callgraph.
- Objetivo 2: Los paradigmas de diseño DBC y Refinement types aplicados en este proyecto, aumentaron la calidad *per se* del nodo Mantis validando la construcción de la cabecera de los bloques de ETC. A pesar de contar con las técnicas de aseguramiento de calidad Unit test, Property-based testing, Continuous integration, no se llevaba a cabo la validación de la creación de nodos según el Yellow paper de ETH.
- Objetivo 3: Al refactorizar la clase BlockHeader se agregaron tan solo 30 líneas de código y se editaron 10 módulos de pruebas para continuar con el code coverage del 83 %. Con esta actualización se logró garantizar la generación de cabeceras de nodos de ETC.

6.2 Lecciones aprendidas

El nodo Mantis cuenta con más de 50 mil líneas de código, el poder entender y conocer el proyecto completo para así poder aplicar las técnicas de diseño de DBC y Refinement types, hubiera consumido mucho tiempo.

Gracias a la aplicación de call graph fue posible identificar las clases de mayor grado de centralidad, y a partir de este conocimiento refactorizar la clase BlockHeader. El agregar unas cuantas líneas de código, se dio certeza a la construcción de la cabecera del bloque siguiendo las reglas del Yellow paper de ETH.

Como el nombre por sí mismo describe a las técnicas de diseño DBC y Refinement types, estos paradigmas deberían de ser aplicados en el diseño de las clases del proyecto. La mayoría de los atributos del constructor de BlockHeader son de tipo ByteString, dicho por Ken Scambler "¿Debo de usar un String como un parámetro en un método? ¿Es la edición completa en mandarín del trabajo de Shakespeare un parámetro válido?", descrito así, cualquier entrada de tipo String es válida.

6.3 Trabajo futuro

A continuación, se listan las posibles mejoras que pueden ser implementadas:

1. **Aplicación de análisis dinámico para generar call graph.** Comprobar si la aplicación de un análisis dinámico arroja mejores resultados para encontrar la clase con mayor grado de centralidad en el proyecto.
2. **Modificar los tipos del constructor BlockHeader.** Crear un tipo refinado para validar los parámetros hash y sustituirlos por los que usan ByteString.
3. **Modificar las instancias de la clase BlockHeader.** Modificar todas las instancias a la clase BlockHeader para solventar los errores producidos por el cambio de tipos en los atributos de la clase.
4. **Implementar el paradigma Smart constructor.** Utilizar el paradigma Smart constructor en la clase BlockHeader y analizar los resultados de utilizar esta técnica.

7 Referencias

(2015, September 30). Retrieved from https://doc.akka.io/japi/akka/2.4.0/akka/protobuf/ByteString.html

(2018). Ethereum Classic Client (Mantis) Security Audit. KUDELSKI SECURITY. Retrieved from https://cybermashup.files.wordpress.com/2018/01/iohk-audit.pdf

Aichernig, B. K., & Schumi, R. (2017, 12). Property-based testing of web services by deriving properties from business-rule models. *Software & Systems Modeling*. doi:10.1007/s10270-017-0647-0

Alharby, M., & Moorsel, A. V. (2017, 08). Blockchain Based Smart Contracts : A Systematic Mapping Study. *Computer Science & Information Technology (CS & IT)*. doi:10.5121/csit.2017.71011

Ali, K., & Lhoták, O. (2012). Application-Only Call Graph Construction. *ECOOP 2012 – Object-Oriented Programming Lecture Notes in Computer Science,* 688-712. doi:10.1007/978-3-642-31057-7_30

Ali, K., Rapoport, M., Lhoták, O., Dolby, J., & Tip, F. (2014). Constructing Call Graphs of Scala Programs. *ECOOP 2014 – Object-Oriented Programming Lecture Notes in Computer Science,* 54-79. doi:10.1007/978-3-662-44202-9_3

Arias, T. B., Spek, P. V., & Avgeriou, P. (2011, 03). A practice-driven systematic review of dependency analysis solutions. *Empirical Software Engineering, 16*(5), 544-586. doi:10.1007/s10664-011-9158-8

Burgess, K., & Colangelo, J.(2015). The Promise of Bitcoin and the Blockchain. Consumers' Research.

Buterin, V., E. (2018). Ethereum/wiki. Retrieved from

https://github.com/ethereum/wiki/wiki/White-Paper

Castor, A. (2017, August 11). Ethereum Classic Forges Its Own Identity With New Mantis Client. Retrieved from https://bitcoinmagazine.com/articles/ethereum-classic-forges-its-own-identity-new-mantis-client/

Continuous Integration and Delivery. (n.d.). Retrieved from https://circleci.com/

Davies, J., Schulte, W., & Barnett, M. (2004). *Formal methods and software engineering: 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004: Proceedings*. Springer.

Davies, R. (2005). *Practical refinement-type checking*.

Davies, R. (1997). A refinement-type checker for standard ML. *Algebraic Methodology and Software Technology Lecture Notes in Computer Science,* 565-566. doi:10.1007/bfb0000499

Delmolino, K., Arnett, M., Kosba, A., Miller, A., & Shi, E. (2016). Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab. *Financial Cryptography and Data Security Lecture Notes in Computer Science,* 79-94. doi:10.1007/978-3-662-53357-4_6

Dhillon, V., Metcalf, D., & Hooper, M. (2017). *Blockchain Enabled Applications Understand the Blockchain Ecosystem and How to Make it Work for You*. Apress.

Feldman, Y. A. (2003). Extreme Design by Contract. *Extreme Programming and Agile Processes in Software Engineering Lecture Notes in Computer Science,* 261-270. doi:10.1007/3-540-44870-5_32

Ghosh, D., Sheehy, J., Thorup, K. K., & Vinoski, S. (2011, 11). Programming language impact on the development of distributed systems. *Journal of Internet Services and Applications, 3*(1), 23-30. doi:10.1007/s13174-011-0042-y

Gousiosg/java-callgraph. (2018). [online] Available at: https://github.com/gousiosg/java-callgraph.git.

Grove, D., Defouw, G., Dean, J., & Chambers, C. (1997). Call graph construction in object-oriented languages. *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications - OOPSLA '97*. doi:10.1145/263698.264352

Hughes, J. (2010). Software Testing with QuickCheck. *Central European Functional Programming School Lecture Notes in Computer Science,* 183-223. doi:10.1007/978-3-642-17685-2_6

Input Output. (n.d.). Mantis – Ethereum Classic Beta Release. Retrieved from https://iohk.io/blog/mantis-ethereum-classic-beta-release/

Joshi, B. (2016). SOLID Principles. Beginning SOLID Principles and Design Patterns for ASP.NET Developers, 45-85. doi:10.1007/978-1-4842-1848-8_2

Jentzsch, C. (2016). Decentralized Autonomous Organization to Automate Governance. *SlockIt*, 1–30. Retrieved from slock.it/dao.htm

Koblitz, N., & Menezes, A. J. (2015, 11). Cryptocash, cryptocurrencies, and cryptocontracts. *Designs, Codes and Cryptography, 78*(1), 87-102. doi:10.1007/s10623-015-0148-5

Lewis, A. (2016, November 24). A gentle introduction to smart contracts. Retrieved from https://bitsonblocks.net/2016/02/01/a-gentle-introduction-to-smart-contracts/

Kuhn, D. R., Chandramouli T. & Butler R. W. (2002). “Cost effective use of formal methods in verification and validation”. Foundations '02, a Workshop on Modeling and Simulation Verification and Validation for the 21st Century, Laurel, Maryland, USA, 106-144

Li, T. H. (2017). Deployments. *Advanced Microservices,* 55-72. doi:10.1007/978-1-4842-2887-6_3

Main Page. (n.d.). Retrieved from http://wala.sourceforge.net/

Merkle, R. (2016) DAOs, Democracy and Governance. Cryonics Magazine, July- August, Vol 37:4, pp 28-40; Alcor, www.alcor.org

Meyer, B. (1992, 10). Applying 'design by contract'. *Computer, 25*(10), 40-51. doi:10.1109/2.161279

Meyer, B. (1997). *Object-oriented software construction*. Prentice Hall.

Mitchell, R., & McKim, J. (2002). *Design by contract, by example*. Addison Wesley.

Moore, T. (2013, 12). The promise and perils of digital currencies. *International Journal of Critical Infrastructure Protection, 6*(3-4), 147-149. doi:10.1016/j.ijcip.2013.08.002

Narayanan, A., Bonneau, J., Felten, E., Miller, A., Goldfeder, S.(2016, 08). Bitcoin and Cryptocurrency Technologies. *Network Security.* doi:10.1016/s1353-4858(16)30074-5

Nofer, M., Gomber, P., Hinz, O., & Schiereck, D. (2017, 03). Blockchain. *Business & Information Systems Engineering, 59*(3), 183-187. doi:10.1007/s12599-017-0467-3

Pouclet, R. (2014). Pro iOS Continuous Integration. Apress.

Roth, S. (2017). Object Orientation. Clean C, 133-166. doi:10.1007/978-1-4842-2793-0_6

Swan, M. (2015). BLOCKCHAIN: Blueprint for a new economy. SHROFF & DISTR.

Tapscott, D. (1997). The digital economy: Promise and peril in the age of networked *intelligence*. McGraw-Hill.

Tillmann, N., Halleux, J. D., & Schulte, W. (2010). Parameterized Unit Testing with Pex: Tutorial. *Lecture Notes in Computer Science Testing Techniques in Software Engineering,* 141-202. doi:10.1007/978-3-642-14335-9_5

Zhao, J. L., Fan, S., & Yan, J. (2016, 12). Overview of business innovations and research opportunities in blockchain and introduction to the special issue. *Financial Innovation, 2*(1). doi:10.1186/s40854-016-0049-2

Zheng, Z., Xie, S., Dai H., Chen, X., Wang, H. (2017). Blockchain Challenges and Opportunities: A Survey.

Zimmermann, T., & Nagappan, N. (2008). Predicting defects using network analysis on dependency graphs. *Proceedings of the 13th International Conference on Software Engineering - ICSE '08*. doi:10.1145/1368088.1368161