



UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES
DEPARTAMENTO DE COMPUTACIÓN

Test Lint

Evaluando Calidad de los Tests

Tesis de Licenciatura en Ciencias de la Computación

Facundo Linari

Director: Lic. Hernán Wilkinson

Buenos Aires, 2023

Resumen

El Testing es una actividad importante en el desarrollo de proyectos de software. Los tests automáticos aseguran que el sistema se comporta como lo esperado y sirven como documentación para entender código de terceros.

Sin embargo, mientras el sistema crece y evoluciona, los tests necesitan adaptarse también para mantenerse a la par de las necesidades del sistema. Mientras el conjunto de tests crece, el esfuerzo invertido para mantenerlos se convierte en una actividad que involucra tiempo y esfuerzo, lo que impacta directamente en los objetivos finales del desarrollo. En este contexto, la calidad de los tests es un problema importante, ya que, los desarrolladores necesitan valorar y entender los tests para que estos cumplan con los requerimientos actuales.

Mientras que el testing ha pasado a ser una forma popular y soportada por los IDEs actuales para la comprobación del correcto funcionamiento del código, las metodologías y herramientas que intentan evaluar la calidad de los tests, son escasas o para nada integradas al proceso de testing, yendo más allá, casi no hubo intentos concretos de medir la calidad de un test detectando errores de diseño en ellos. Estos errores se los llaman Test Smells, en honor a los *Coding Smells* que son los mismos errores pero en el código fuente. Un Lint es una herramienta que detecta estos errores de diseño en el código de forma estática. De forma similar, existen los Test Lints que analizan los errores de diseño del código de test.

El presente trabajo contribuye a la investigación de metodologías de testing al medir la calidad de los test. En particular se analizan Test Smells y define un conjunto de criterios para determinar la calidad de los tests. También se presenta la herramienta *S-TestLint*, un analizador de tests que encuentre Test Smells cometidos y que propone soluciones de forma automática. Esta herramienta fue implementada e integradas en *CwisUniversity* y comenzaron a ser usadas por los estudiantes de la materia de Ingeniería de Software 1 de la FCEN, para promover una mejor calidad de los tests en su etapa de aprendizaje.

Palabras clave: Software Testing, Test Smells, Static Analysis.

Agradecimientos

Para empezar quiero agradecer a aquellas personas que hicieron posible esta tesis. A Hernán, ya que este trabajo es tanto mío como suyo. Gracias por su mentoría y paciencia. Disfruté mucho nuestros encuentros, de los cuales siempre me llevo algo valioso. A mi hermano Nacho y mi primo Mati cuyo apoyo fue crucial en algunos de los tantos desencuentros que tuve con este trabajo. A mis líderes Tonny y Clau que me dieron todo su apoyo y consideración para que pueda llevar a cabo este trabajo a la par del trabajo. A mis viejos, que me permitieron estudiar esta carrera. A los excelentes profesores que ayudaron a formarme. Y finalmente a mis amigos, los cuales siempre me acompañaron a la vera de este camino.

Índice

Resumen	I
Índice	III
1. Introducción	1
1.1. Objetivos	3
1.1.1. Implementación de S-TestLint	3
1.1.2. Aplicación de la herramienta sobre casos de estudio	3
2. Marco teórico	5
2.1. <i>TDD</i>	6
3. Implementación	8
3.1. Objetivo de S-TestLint	8
3.2. Desarrollo	8
3.3. Arquitectura de TestLint	9
3.4. Flujo completo de ejecución	11
3.5. Test Smells	15
3.5.1. Anonymous Test	15
3.5.2. Assertionless Test	16
3.5.3. Guarded Test	16
3.5.4. Long Test	17
3.5.5. Magic Literals Test	17
3.5.6. Proper Organization Test	18
3.5.7. Wrong Assert Usage Test	19
3.5.8. Unclassified Method Category	20
3.5.9. Test With Return	20
3.5.10. Transcribing Test	21
3.5.11. Returning Assertion	22
3.5.12. Unusual Test Order	22
3.5.13. Mixed Selectors	23
3.5.14. Test Method Category Name	23
3.5.15. Empty Method Category	24
3.5.16. Unused Shared-Fixture Variables	24
3.5.17. Max Instance Variables	25
3.5.18. Teardown Only Test	26
3.5.19. Abnormal UTF-Use	26
4. Caso de estudio	27
4.1. Metodología	27
4.2. Resultados y conclusiones	28
4.2.1. 2do Parcial - 1er Cuatrimestre 2022	28

4.2.2.	2do Parcial - 2do Cuatrimestre 2022	32
4.2.3.	Paquetes de Cuis University	35
5.	Conclusiones	39
5.1.	Conclusiones obtenidas a partir del desarrollo de la herramienta	39
5.2.	Conclusiones sobre los resultados estadísticos	39
5.3.	Trabajo futuro	39
5.3.1.	UI	39
5.3.2.	<i>Test Smells</i> dinámicos	39
6.	Apéndices	40

Introducción

El testing es una disciplina en la ingeniería de software que permite encontrar defectos surgidos durante el desarrollo. En cada test se define un comportamiento esperado y se valida que el resultado real del programa al ser ejecutado coincida con este comportamiento deseado. El testing se puede hacer de forma automática (tests en los que la ejecución y comparación del resultado con el valor esperado se realiza de forma automática) o manual (ejecución manual del programa en la que se ve si el resultado).

En el contexto de este trabajo se refiere a tests unitarios como tests automáticos que corren rápido (tienen una duración menor a 10 milisegundos) sin importar el alcance que tienen, es decir, no están necesariamente acotados a un método o clase. Los tests unitarios sirven como pruebas de regresión. Ayudan al desarrollador a asegurar la calidad del código y detectar errores en la aplicación a medida que se van realizando cambios en el código. Además, los tests sirven como una documentación viva que puede ser utilizada para entender código ajeno. En los tests se tiene ejemplos de uso del código que se testea.

La ausencia de tests unitarios aumenta la posibilidad de que se introduzcan errores en el código al realizar cambios que no son detectados. En caso de realizar tests incorrectamente se tiene el mismo problema. Tests incorrectos dan una falsa seguridad de que si se introducen errores van a ser detectados cuando no lo van a ser. Los tests pueden no cubrir alguna funcionalidad o requerimiento de la aplicación, dejando ciertas características sin testear y no asegurando la estabilidad del código fuente.

Los tests unitarios son una actividad relevante que ha tomado mucha preponderancia en el desarrollo de proyectos de software de hoy en día al punto que ya todo ambiente de desarrollo tiene integrada una herramienta para realizarlos. Por ejemplo, Kent Beck et. al. introducen la herramienta de testing JUnit para el lenguaje *Java* [BG98], el lenguaje *Go* provee en su biblioteca estandar soporte para tests automáticos de paquetes de *Go* y el lenguaje *Rust* tiene integrado atributos y herramientas para dar soporte a test unitarios.

Los tests unitarios solo indican si para entradas específicas el programa se comporta como es esperado, esto significa que puede haber entradas que no se comportan como es esperado. Sumado a esto, los tests unitarios no revelan de forma directa información de la cantidad de funcionalidad o datos que se testean, ya que habría que ir por cada test analizando como procesa la entrada utilizada para saberlo. Finalmente, los tests unitarios tampoco indican qué tan bien testado está el código fuente ni si son confiables debido a que depende de como fueron realizados si logran identificar errores. Un ejemplo de esto es un test sin ninguna aserción el cual permite que el código fuente realice cualquier acción, siempre y cuando no arroje una excepción, lo que claramente permite cualquier modificación del comportamiento y alejarse del de lo esperado reduciendo la confianza que se tiene sobre el mismo.

Se ha llevado a cabo gran cantidad de investigación sobre los tests automáticos para evaluar la calidad de estos mismos con diferentes perspectivas, cada una de estas con

distintos enfoques que permiten evaluar distintos aspectos de los tests. En particular hay 3 enfoques dominantes en la literatura. A continuación se explica cada una de ellas.

Code Coverage consiste en identificar qué partes del código son ejecutadas en cada test. Existen distintos criterios para determinar la cobertura que proporciona un test. Por ejemplo la cobertura de sentencias, en donde cada sentencia se considera cubierta si es ejecutada por un test. Esto provee una medida cuantitativa de funcionalidad siendo testeada [ZHM97].

Mutation Analysis da conocimiento de la estabilidad del código [MBLT06]. Esto lo hace sobre todo un conjunto de tests evaluando si logra identificar como errores pequeñas modificaciones en el código fuente, los cuales son llamados mutantes. Se obtiene el mutation score a partir de la cantidad de mutaciones que lograron hacer que depende del conjunto de mutantes que se utilizó. Esta técnica sólo aporta información sobre todo el conjunto de tests y no sobre un test en particular, ya que es evaluado sobre todo el conjunto de mutantes.

Test Ordering muestra la interconexión de tests [PJ02]. Indica un orden parcial entre los tests para identificar el test más específico para una parte de código cubierta. Esto sirve especialmente para reparar errores, ya que tests menos específicos se están rompiendo probablemente por la misma razón y pueden agregar ruido al momento de analizar y arreglar el error. Esta técnica es principalmente teórica y no está en los frameworks de testing actuales.

Estas tres metodologías contribuyen de forma distinta a evaluar la calidad de los tests. Sin embargo, todas comparten que sus resultados son estáticos [Rei07]: Code Coverage indica las líneas cubiertas, Mutation Analysis indica el mutation score y Test Ordering indica un orden parcial entre los tests. También comparten que no tienen relación directa con el código de test debido a la dificultad para vincular el resultado con un test en particular, ya que la mayoría de estas metodologías se realizan sobre un conjunto de tests. Dando información sobre todo el conjunto de tests y no un test en particular.

Todas estas cuestiones hacen que se pierda el foco para el proceso de testing y desarrollo para estas metodologías, simplemente evalúan los tests de una forma abstracta, a un nivel general. El no tener información específica sobre cada test puede traer el problema de que se estén cometiendo errores en los tests que afecten su mantenibilidad y legibilidad. A este tipo de errores los llamamos *Test Smells* [RDBD06a]. Los *Test Smells* introducidos en la literatura describen tests unitarios que hacen asunciones inapropiadas sobre la disponibilidad de los servicios externos, tests que son largos y complejos así como tests que exponen signos de redundancia. Tales tests tienen un impacto negativo en la mantenibilidad: de un lado los tests complejos son difíciles de entender y modificar; y del otro lado, la presencia de estos *Test Smells* amenaza con la repetición, independencia y estabilidad de los tests. Por esto, resulta importante tener herramientas que analicen el código de los tests en busca de estos errores. Y es aún más importante en etapas tempranas de aprendizaje para no obtener malos hábitos de testing. Por ejemplo, tener tests anónimos o sin nombre es un *Test Smell* y si nunca se lo marca a un desarrollador puede terminar adquiriendo este hábito. Dependiendo de como se lleva a cabo el análisis por la herramienta se clasifican a los *Test Smells* como estáticos (solo analizan el código fuente del test) o dinámicos (ejecutan el test para obtener información adicional en tiempo de ejecución).

Siguiendo un proceso de desarrollo Agile, el cuerpo de los tests crece a la par del código fuente. Sin embargo, debido a refactorings y cambios en los requerimientos, el

código puede sufrir un proceso de erosión [EGK⁺01]. La misma erosión sucede para el código de los tests [RDBD06b]. Los tests se vuelven largos y complejos. Aunque estos tests siguen cumpliendo el propósito de validar la correctitud del sistema, cumpliendo con las validaciones que hacen las metodologías descritas, estos pueden fácilmente romperse cuando nuevas adaptaciones son requeridas para el código de una aplicación.

Para atacar el problema de la erosión del código existen los lints, analizadores de código que buscan patrones que tienden a llevar a la erosión del proyecto que se mencionó previamente. Como contraparte también existen los TestLint que son lints enfocados en el código de los tests. Estos al ser enfocados directamente sobre el código no sufren esa dificultad para vincular el resultado con el test específico que tiene errores, a diferencia del resto de las metodologías previamente mencionadas.

1.1. Objetivos

1.1.1. Implementación de S-TestLint

El objetivo principal del presente trabajo es desarrollar S-TestLint, un analizador de tests que encuentre *Test Smells* cometidos y que proponga soluciones.

Además, mediante la implementación de esta herramienta se busca profundizar sobre *Test Smells* conocidos y definir nuevos criterios que se adapten mejor al caso de uso de alumnos recién comenzando a adoptar técnicas de testing.

Uno de los objetivos de este trabajo es dar feedback a los alumnos y docentes sobre la calidad de los tests que se escriben, específicamente dentro del contexto de la materia de Ingeniería de Software 1 de la Facultad de Ciencias Exactas y Naturales.

En la materia Ingeniería del Software 1 se trata con la problemática del desarrollo de software a gran escala, y esto abarca tanto temas técnicos como de gestión. Esta materia funciona para brindarle a sus alumnos un panorama de los procesos y técnicas que pueden ser usadas en este tipo de proyectos. Por esto mismo, resulta de mucho valor introducir esta herramienta con el fin de reforzar el aprendizaje de buenas prácticas de testing.

Este trabajo se desarrollará en *Smalltalk* usando la tecnología de *CuisUniversity*. Este es un entorno enfocado en *Smalltalk-80* libre y Open Source. Está basado en la *OpenSmalltalk VM*, como *Squeak*, y su principal objetivo es mantener su base pequeña y simple.

CuisUniversity es un ambiente basado en Cuis creado con el objetivo de enseñar programación orientada a objetos. Actualmente, es utilizado en la Universidad de Buenos Aires, en la Universidad de Quilmes y en la Universidad Católica Argentina.

1.1.2. Aplicación de la herramienta sobre casos de estudio

Para utilizar la herramienta S-TestLint y obtener información de los *Test Smells* que más se cometen para así destacar en que cosas hay que hacer foco al enseñar a los alumnos y ver que casos no están aportando mucha información y deberían ser adaptados para disminuir falsos positivos.

Para cada caso se analizará la cantidad de veces que se comete cada *Test Smell*. También se tendrá en cuenta la cantidad de tests en cada paquete con el fin de comparar si a mayor cantidad de tests se cometen menos *Test Smells*.

Por un lado, se utilizará la herramienta sobre resoluciones de parciales pasados de la materia de Ingeniería de Software 1 de la Facultad de Ciencias Exactas y Naturales. Para

estos casos también se analizará si hay alguna correlación entre la nota del parcial y *Test Smells* cometidos.

Por otro lado, se analizarán los paquetes que integran a CuisUniversity como Aconagua y Collections.

Entre los parciales de alumnos y paquetes que integran a CuisUniversity se espera que, por el nivel de expertise diferente de quienes lo desarrollaron, los parciales contengan una mayor cantidad de *Test Smells* que los paquetes de CuisUniversity y que los parciales de los alumnos, por ser de dominios más acotados y simples, no presenten algunos *Test Smells* muy específicos que si se espera que los presenten los paquetes de CuisUniversity por tener que hacer tests más complejos en algunos casos.

Marco teórico

Dentro de la literatura, uno de los trabajos más influyentes sobre la presente tesis, fue el de Stefan Reichhart [Rei07]. En dicho trabajo se desarrolló un TestLint para Squeak, dialecto de Smalltalk, para calificar la calidad de los tests. Ahí se define un conjunto de criterios para determinar calidad de tests y evaluar su enfoque en una gran muestra de tests unitarios encontrados en proyectos Open-Source con el objetivo de aprender las características que influyen la legibilidad y mantenibilidad de los tests.

Particularmente, su caso de estudio consistió de 4834 métodos de test y clases de test 742, los cuales manejaron de la siguiente forma:

1. Su primer paso fue obtener los tests y recolectar una lista de los problemas encontrados en los tests vía inspección manual. Debido al gran número de tests no analizaron todos, pero se enfocaron en una muestra de métodos de test que era sabido que eran buenos o malos basado en los comentarios de la comunidad de Squeak.
2. En el segundo paso agrupó los problemas para identificar cosas comunes y diferencias, y aplicaron lo aprendido para automatizar la identificación y así implementar su TestLint.
3. En el tercer paso aplicó su herramienta sobre todos los tests de su caso de estudio e inspeccionaron manualmente los *Test Smells* detectados para identificar falsos positivos.

Al analizar los resultados concluyó que se generan falsos positivos porque las reglas utilizadas para evaluar si se está cometiendo un *Test Smell* no están suficientemente bien especificadas o la sensibilidad al contexto no puede ser apropiadamente tratada. También concluyó que mientras más clara sea la meta del *Test Smell*, más precisa es la regla y que los *Test Smells* no están equitativamente distribuidos. Unos son cometidos más que otros. Con sus resultados concluye que el testing, por lo menos a largo plazo, escala bien así como la calidad de los tests aumenta junto a la experiencia de escribir tests. Un test bueno cualitativamente tiene un efecto positivo en la aplicación.

En el presente trabajo se plantea una herramienta similar a la de Stefan Reichhart adaptado a la tecnología CuisUniversity. Por limitación de la tecnología no se utilizó Traits, técnica para reducir duplicidad del código evitando herencia, a diferencia del trabajo de Stefan Reichhart. Se implementaron solo un subconjunto de los *Test Smells* estáticos, a pesar de existir *Test Smells* dinámicos, por simplicidad. Además, se definen nuevos criterios para *Test Smells* que se adapten a distintas situaciones. Al igual que en el trabajo de Stefan Reichart, se utiliza la herramienta planteada para analizar paquetes Open-Source.

2.1. TDD

El *Test Driven Development* (*TDD*) es una práctica de desarrollo de software que involucra escribir primero los tests antes de implementar el código. En *TDD*, los tests se utilizan como una guía para dirigir el diseño y la implementación del código, asegurando que este cumpla con los requisitos establecidos. Al utilizar *TDD*, se busca mejorar la calidad del código y reducir la posibilidad de introducir errores, ya que cada parte del código es validada a través de los tests correspondientes.

El origen del *TDD* se remonta a las prácticas de desarrollo ágil y, en particular, a los principios de Extreme Programming (XP) [BA04], una metodología ágil popular. Fue desarrollado por Kent Beck [Bec02] como parte de la metodología XP.

La técnica de TDD propone que el desarrollo de software se realice en iteraciones de tres pasos principales:

1. Escribir un test y ejecutarlo para que falle: En esta etapa, el desarrollador define un test automático que describe un comportamiento o funcionalidad deseada y la ejecuta. Dado que aún no se ha implementado la funcionalidad requerida, se espera que el test falle en esta fase inicial. Este paso garantiza que el test sea válida y esté correctamente vinculada a la funcionalidad que se está desarrollando.
2. Escribir código de producción para que el test pase: Una vez que el test ha fallado, se procede a implementar el código necesario para que el test pase exitosamente. El objetivo es escribir la cantidad mínima de código requerido para que el test sea exitosa, sin agregar funcionalidad innecesaria.
3. Refactorizar: Después de que el test pase, se realiza la etapa de refactorización. En esta fase, se mejora la estructura, la legibilidad y la eficiencia del código sin alterar su comportamiento externo. Se eliminan duplicaciones, se mejoran los nombres de variables y funciones, se simplifican expresiones, se aplica la modularización, etc. El objetivo es mejorar la calidad del código y hacerlo más mantenible y comprensible.

Estas tres etapas se repiten en un ciclo iterativo continuo a medida que se desarrolla el software. El *TDD* promueve la escritura de tests antes de la implementación y el refactoring constante para garantizar un código limpio, bien probado y fácil de mantener.

El uso de *TDD* en el desarrollo de software proporciona varios beneficios significativos:

- Mejora la calidad del software: Al escribir tests automáticos antes de implementar el código de producción, el *TDD* ayuda a identificar y corregir errores en etapas tempranas del proceso de desarrollo. Esto permite detectar y solucionar problemas de forma proactiva, lo que resulta en un software más confiable y de mayor calidad.
- Facilita la detección temprana de errores: El enfoque *TDD* garantiza que los tests se ejecuten de forma regular y automatizada. Como resultado, cualquier regresión o problema introducido por cambios posteriores en el código se detecta rápidamente. Esto facilita la localización y corrección temprana de errores, evitando que se propaguen y se conviertan en problemas más difíciles de solucionar.
- Promueve un diseño modular y limpio: Al escribir tests antes de implementar el código de producción, se fomenta el diseño de interfaces claras y una arquitectura

modular. El *TDD* ayuda a identificar las dependencias entre componentes y a definir interfaces adecuadas, lo que conduce a un código más modular, mantenible y extensible.

- Proporciona una documentación ejecutable: Los tests en *TDD* sirven como una forma de documentación ejecutable. Al leer los tests, se obtiene una comprensión clara de cómo se supone que debe funcionar el código. Esto facilita la colaboración en equipos de desarrollo y permite una mejor comprensión del sistema en general.
- Incrementa la confianza en el código: Al tener una suite de tests automáticos que se ejecutan regularmente, los desarrolladores tienen mayor confianza en que los cambios realizados no introducen regresiones o problemas. Esto reduce el riesgo de introducir errores y brinda seguridad al realizar modificaciones o mejoras en el software existente.
- Fomenta la refactorización segura: El *TDD* permite realizar refactorizaciones con confianza. Al tener una cobertura de tests sólida, se puede modificar y mejorar el código existente sin preocuparse por romper la funcionalidad existente. Los tests actúan como salvaguardas, asegurando que el comportamiento esperado se mantenga intacto después de la refactorización.
- Ahorra tiempo a largo plazo: Aunque inicialmente puede parecer que el *TDD* requiere más tiempo y esfuerzo debido a la necesidad de escribir tests adicionales, a largo plazo, puede ahorrar tiempo significativo. Al evitar errores y regresiones, se reduce la cantidad de tiempo dedicado a la depuración y corrección de problemas. Además, al facilitar la comprensión del código y promover la modularidad, el mantenimiento y la adición de nuevas características se vuelven más eficientes.

En resumen, el uso del *TDD* proporciona beneficios como una mayor calidad del software, detección temprana de errores, diseño modular y limpio, documentación ejecutable, confianza en el código, refactorización segura y ahorro de tiempo a largo plazo. Estos beneficios contribuyen a un desarrollo de software más eficiente y confiable.

Implementación

En este capítulo se describe la implementación de la herramienta S-TestLint, incluyendo tanto su modelo como su flujo de ejecución. En primer lugar se expone el objetivo de S-TestLint. A continuación se detalla el modelo utilizado, las entidades principales empleadas y los aspectos relacionados con su desarrollo. Además, se realiza una comparación entre esta herramienta y TestLint, implementada por Stefan Reichhart [Rei07], en base a cuyo trabajo se fundamentó el presente estudio. Seguidamente, se presenta el funcionamiento completo de la herramienta, desde el momento en que se emite la orden de analizar una entidad hasta la presentación de los resultados obtenidos. Por último, se describen todos los *Test Smells* para los cuales se ha implementado un método de identificación.

3.1. Objetivo de S-TestLint

S-TestLint es una herramienta diseñada para analizar tests en busca de *Test Smells* y proporcionar soluciones para corregirlos. Esta herramienta utiliza un conjunto de reglas, una definida para cada tipo de *Test Smell* que se busca identificar, y aplica el análisis a entidades de test específicas. Estas entidades pueden ser clases, categorías de métodos o métodos de test.

Al aplicar S-TestLint, se examinan los diferentes componentes de los tests y se evalúa si cumplen con las reglas establecidas para detectar *Test Smells*. En caso de encontrar uno o más *Test Smells*, S-TestLint proporciona soluciones y recomendaciones para corregirlos. Estas soluciones pueden incluir cambios en la estructura del test, la mejora de la claridad o legibilidad del código, entre otras acciones correctivas.

El objetivo principal de S-TestLint es mejorar la calidad de los tests al identificar y ayudar a resolver los *Test Smells*. Al utilizar esta herramienta en el proceso de desarrollo de software, los desarrolladores pueden mantener tests más limpios, legibles y mantenibles, lo que contribuye a la eficacia y confiabilidad de los tests realizados en el proyecto.

3.2. Desarrollo

Para el desarrollo de S-TestLint se utilizó la técnica de *Test Driven Development* [Bec02] (*TDD*).

En este trabajo, se aplicó *TDD* para garantizar la robustez y la funcionalidad de las reglas implementadas, asegurando que los tests generados no presentaran *Test Smells* al integrar el uso de S-TestLint al ciclo de desarrollo. Se llevó a cabo esta integración de la herramienta a la técnica de *TDD* agregando a la etapa de desarrollo que se considere que el test pasa si S-TestLint no lo marca con *Test Smells*. De esta forma se llegó a que ninguno de los tests presente un *Test Smell* de los que identifica esta herramienta.

Tener en cuenta que se hizo uso de excepciones para tests que se utilizan como objetos de test. Es decir, que se crearon tests que presentan al menos un *Test Smell* para que sean utilizados por tests que buscan ejecutar partes específicas del análisis. Se optó por dejar estos objetos de test creados y no crear y eliminar dinámicamente en la ejecución de los tests para que queden como ejemplo fácilmente observable de casos que presentan determinado *Test Smell*.

3.3. Arquitectura de TestLint

La implementación del modelo está basada en las entidades regla y nodo. Reglas y nodos están organizados e implementados en una jerarquía de herencia simple y cada una provee una superclase común llamada `TestLintRule` y `TestLintNode` respectivamente.

Los nodos son entidades de Smalltalk sobre los cuales se hace el análisis. Están encapsuladas de forma polimórfica en una instancia de una subclase de `TestLintNode`. Particularmente, las entidades soportadas por la herramienta son métodos de test (por la clase `TestMethodNode`), categorías de tests (por la clase `MethodCategoryNode`) y clases de test (por la clase `TestClassNode`). Los nodos están interconectados usando una relación de ascendencia abstracta de manera que una instancia de un `TestClassNode` está vinculado con 0 o más `MethodCategoryNode` como sub-nodos y un `MethodCategoryNode` está vinculado con 0 o más `TestMethodNode` como sub-nodos. La [Figura 3.1](#) presenta los diferentes tipos de nodos disponibles en S-TestLint. Notar que otros nodos se pueden agregar sin necesidad de alterar el modelo base. Por ejemplo, para extender el modelo con nodos de paquetes o categorías de clases.

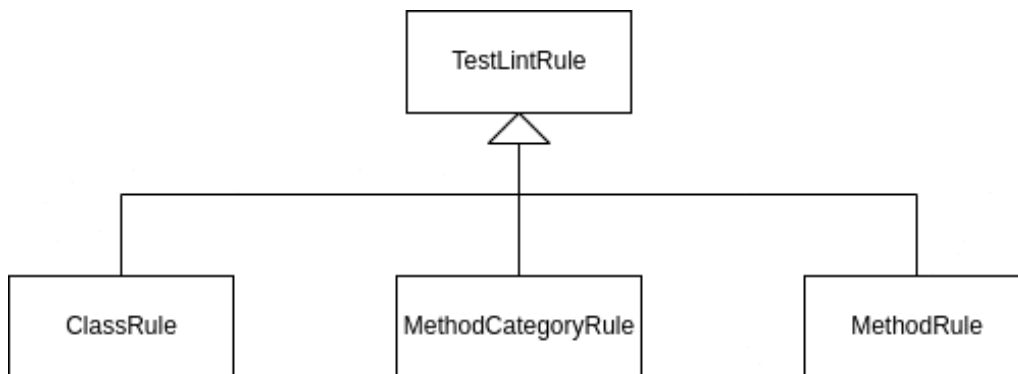


Figura 3.1. Diferentes tipos de nodos disponibles en S-TestLint.

Las reglas contienen la lógica que identifica, para determinado nodo, si contiene un test smell. Cada regla identifica un test smell distinto. Están organizadas por el tipo de nodo que aceptan como entrada. La [Figura 3.2](#) muestra la jerarquía implementada para las reglas. Todas las reglas que se utilizan en una corrida de S-TestLint son subclases de estas.

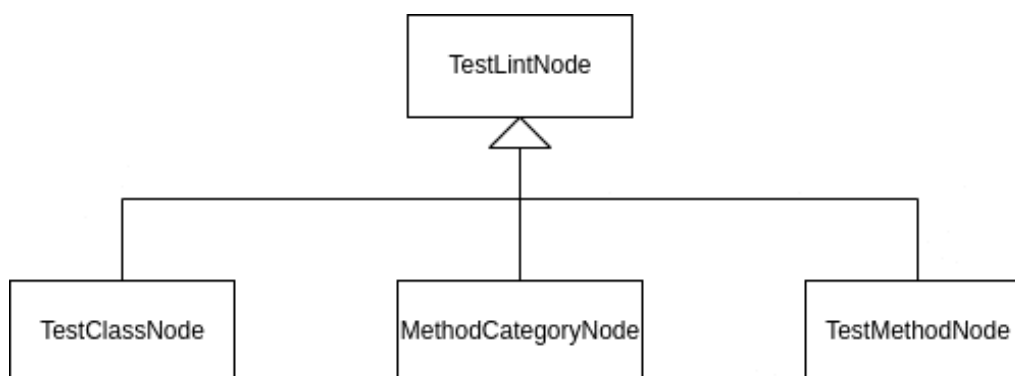


Figura 3.2. Diferentes tipos de reglas disponibles en S-TestLint.

La herramienta S-TestLint se basa en una implementación previa realizada por Stefan Reichhart [Rei07], pero con algunas diferencias y extensiones específicas. Ambos trabajos comparten el mismo modelo y manejo en cuanto a las clases principales de reglas y nodos, utilizando las mismas jerarquías y composición de estas clases principales para terminar con una técnica similar de ejecución descrita a continuación en la explicación del flujo completo de ejecución.

Sin embargo, S-TestLint se enfoca únicamente en la detección de *Test Smells* estáticos, lo cual simplifica su implementación. Por otro lado, la herramienta original implementada por Stefan Reichhart, denominada TestLint, aborda tanto *Test Smells* estáticos como los *Test Smells* dinámicos. Los *Test Smells* estáticos son identificados al analizar el código fuente de los tests, mientras que los *Test Smells* dinámicos se detectan al ejecutar los tests.

S-TestLint incluye un subconjunto de los *Test Smells* implementados en TestLint. Sin embargo, se realizaron modificaciones en algunos *Test Smells* para adaptarlos a los requisitos específicos surgidos del uso de S-TestLint en un entorno educativo. Además, se agregaron reglas para identificar *Test Smells* específicos relacionados con los alumnos.

En cuanto a la implementación, TestLint se basa en el uso de traits [DNS⁺06, SDNB02] para su funcionamiento. Los traits son una alternativa a la herencia para compartir comportamiento en la programación orientada a objetos. Tienen el objetivo de descomponer clases en bloques de construcción reutilizables. A diferencia de las clases, los traits no definen ningún tipo de estado y dependen de la composición en lugar de la herencia como mecanismo para compartir comportamiento. Los traits son componentes simples de software que proveen y requieren métodos.

Los métodos requeridos son aquellos que se utilizan en el trait pero no están implementados en él. Al utilizar traits, se puede componer una clase con múltiples traits para agregar el comportamiento deseado sin heredar de múltiples clases y evitar los problemas asociados con la herencia múltiple, como la complejidad y la ambigüedad.

En el caso de S-TestLint, se optó por utilizar herencia y composición en lugar de traits debido a que el entorno de desarrollo, CuisUniversity, no proporciona la funcionalidad de traits. Aunque los traits ofrecen ventajas en términos de reutilización y composición de comportamiento, se eligió una solución alternativa que permitiera lograr los objetivos de la herramienta dentro del entorno de desarrollo utilizado.

Estas diferencias y adaptaciones realizadas en S-TestLint permiten satisfacer los requisitos específicos del entorno educativo y garantizar un análisis efectivo de los *Test Smells*

en los tests desarrollados en CuisUniversity.

3.4. Flujo completo de ejecución

A rasgos generales S-TestLint se ejecuta sobre una entidad específica. Se arma el nodo asociado a la entidad seleccionada, se instancian las reglas a utilizar, se las evalúa para el nodo y se muestran los resultados.

En un principio se indica sobre qué entidad del sistema se ejecutará S-TestLint. Luego, se construye el nodo asociado a la entidad de prueba que se va a analizar y se obtienen las reglas para evaluarlas sobre el nodo construido. Finalmente, se procesan los resultados obtenidos de la evaluación de las reglas y se los muestra en una ventana.

En las siguientes secciones se explica de forma integral cómo funciona la herramienta S-TestLint, desde la selección del nodo a analizar hasta el procesamiento y muestra de los resultados. Cada sección aborda aspectos específicos del proceso y proporciona una visión completa del funcionamiento de la herramienta.

Selección de nodo y mandar orden de ejecución

Como primer paso se indica sobre que entidad del sistema ejecutar S-TestLint y así construir el nodo asociado a la entidad. Un nodo es la abstracción de la entidad del sistema. Puede ser un método de test, una categoría de métodos o una clase de test.

La selección de la entidad sobre la cual ejecutar S-TestLint se hace mediante los menues desplegables del *Browser del sistema*. Para dar esta funcionalidad se implementaron los métodos `#classLintMenuOptions` para ejecutar S-TestLint sobre una clase, `#messageCategoryMenuOptions` para ejecutar S-TestLint sobre una categoría de métodos y `#messageListMenuOptions` para ejecutar S-TestLint sobre un método de test. En la [Figura 3.3](#) se muestra un ejemplo en el que se manda a ejecutar S-TestLint para la categoría de métodos `guarded test`.

Cada opción que ejecuta S-TestLint termina llamando a un método distinto con el fin de identificar que tipo de nodo crear.

En todas las opciones se identifica si la entidad a analizar es de test. En caso de que no lo sea se corta la ejecución y no se provee ningún tipo de información al usuario.

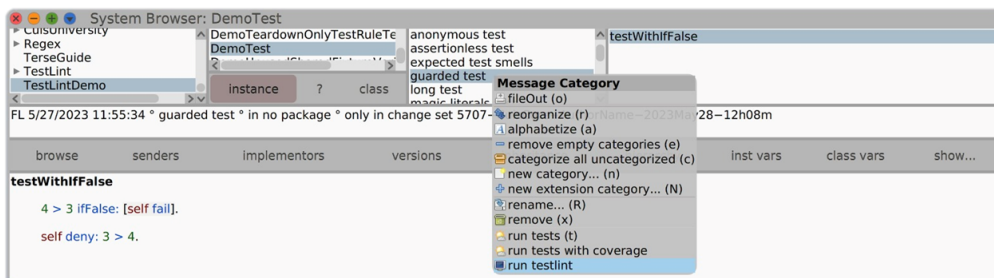


Figura 3.3. Browser del sistema con el menú desplegable de una categoría de métodos en el que se indica la opción para ejecutar S-TestLint sobre ella.

Construcción de entidades

Una vez construido el nodo asociado a la entidad de test a analizar se obtienen las reglas a utilizar. Una regla es un objeto que toma un nodo como argumento, realiza un análisis sobre él y retorna una colección de *Test Smells* violados.

Para realizar la obtención de todas las reglas existentes se obtienen todas las clases concretas de la jerarquía de **#TestLintRule**. Estas se obtienen pidiendo todas las subclases de **#MethodRule**, **#MethodCategoryRule** y **#ClassRule**.

No siempre se quieren ejecutar todas las reglas sobre un nodo particular. Por esto a la clase del nodo se le puede definir el mensaje **#exceptedTestLintRules** que retorna la lista de reglas exceptuadas. Se descartan las reglas exceptuadas de la lista de reglas obtenida.

Las reglas tienen umbrales configurables. Por ejemplo, la regla LongTest se le configura la cantidad de líneas que tiene que tener el método de test para considerar que viola dicha regla. Se define una configuración por defecto para todas las reglas. Con el fin de reducir la cantidad de falsos positivos notificados por S-TestLint se seleccionaron umbrales altos. Cada una de las reglas a utilizar se instancia con el mensaje **#newWithDefaults**.

Con el nodo y las reglas se crea la instancia **#TestLint** encargada de hacer la evaluación de las reglas sobre el nodo. La clase **#TestLint** implementa el patrón *Method Object*. Se le otorga todo lo necesario para su ejecución en la instanciación y solo está encargada de realizar la evaluación de reglas.

Antes de empezar la evaluación de las reglas se obtienen los *Test Smells* esperados, estos se definen con el mensaje **#expectedTestSmells** que debe retornar una colección de **#ExpectedTestSmellInformation**. Estas se marcan como error si se espera el fallo y no sucede.

Ejecución de reglas

Para evaluar todas las reglas seleccionadas sobre el nodo las mismas se ejecutan en serie sobre el nodo.

Para identificar que regla se aplica sobre cual nodo se utilizan *Double Dispatches* entre las entidades del modelo regla y nodo. Estos *Double Dispatches* hacen que el análisis de *Test Smells* sea completamente transparente al esconder el hecho de que las reglas están dedicadas a tipos de nodo específicos. Una instancia de **#TestClassNode** es procesada por instancias de **#ClassRule**. Debido al uso de *Double Dispatches* se asegura que se utiliza la regla correcta en el tipo de nodo correcto sin necesidad de especificar explícitamente la relación entre nodos y reglas.

Se muestra la conexión entre reglas y nodos y la funcionalidad básica de S-TestLint en el diagrama de secuencia de la [Figura 3.4](#). El diagrama muestra la interacción al aplicar una regla de tipo de categoría de tests sobre un nodo de tipo de clase.

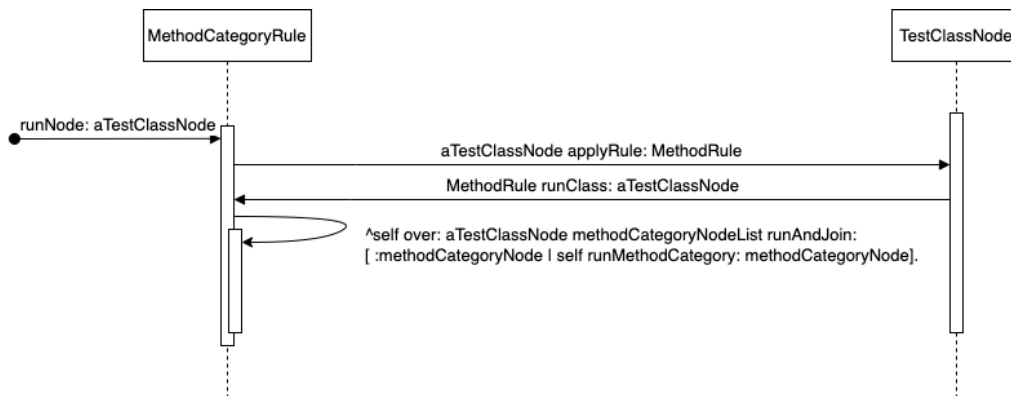


Figura 3.4. *Double Dispatch* entre nodos y reglas conformando el modelo básico de S-TestLint.

Una vez identificado el tipo de regla y nodo dependiendo de su relación se realizan distintas acciones. Si la regla no se puede aplicar sobre el nodo, por ejemplo porque la regla es para una clase y el nodo para una categoría, no se realiza ningún análisis. Si el nodo es más abarcativo que la regla, por ejemplo que el nodo es de una clase y la regla es para un método de test, se ejecuta la regla sobre los subnodos que lo conforman y se repite el proceso hasta encontrar el tipo de nodo que puede analizar la regla.

Cada ejecución de la regla sobre un nodo da como resultado una colección de **#TestSmell**. En caso de no detectar ninguna violación de la regla retorna una colección vacía y en caso de haberse ejecutado una regla para un nodo más abarcativo se conectan los resultados de evaluar la regla sobre cada subnodo.

Procesamiento de resultados

Una vez obtenida la colección de **#TestSmell** mediante la ejecución de todas las reglas para un nodo específico, se procede a verificar si estos son los *Test Smells* esperados. Al inicio de la ejecución, se recopila una colección de **#ExpectedTestSmellInformation**, definida en el método **#expectedTestSmells** de la clase analizada. Cada uno de estos elementos indica que para una entidad específica es esperado que posea un *Test Smell*. Tanto la entidad como el *Test Smell* esperado se indica en la instancia de **#ExpectedTestSmellInformation**.

Para cada **#TestSmell**, se realiza una verificación para determinar si es esperado. Se busca en la colección de **#ExpectedTestSmellInformation** si existe algún elemento vinculado al mismo nodo y a la misma regla.

Los **#TestSmell** detectados que son considerados esperados se eliminan de los resultados si existe una correspondiente **#ExpectedTestSmellInformation** asociada al mismo nodo y regla. Por otro lado, los **#TestSmell** detectados que no son esperados se mantienen en la lista si no se encuentra una **#ExpectedTestSmellInformation** relacionada con el mismo nodo y regla. Si se identifica un *Test Smell* esperado que no fue encontrado durante la ejecución, se agrega a la lista como un **#MissingTestSmell**. Tanto los **#MissingTestSmell** como los **#TestSmell** heredan de la clase abstracta **#TestLintFailure**, lo que permite tratarlos en una misma colección de resultados.

Finalmente, utilizando la colección final de **#TestLintFailure**, se genera una ventana en la que se muestran los resultados obtenidos. Una interfaz [SIK⁺82] buena y fácil de usar

es crucial para hacer testing y análisis de software eficiente. Sin embargo, desarrollar una capaz de proveer completa funcionalidad, que esconda todos los detalles del modelo y sea fácil e intuitiva de usar no es una tarea menor. Más aún, cualquier tipo de interfaz es subjetiva a diferentes expectativas y percepciones.

Como el objetivo principal era tener implementada la herramienta de análisis y detección de *Test Smells* se decidió hacer una interfaz básica sin preocuparse en la usabilidad y manejo. Es por esto que la herramienta consiste de un simple navegador de los tests con sus *Test Smells* y *Test Smells* esperados y no encontrados.

En la [Figura 3.5](#) se presenta un ejemplo visual de la ventana en la que se muestran los resultados procesados. Al comienzo hay una lista de todos los nodos con al menos un *Test Smell*. En caso de no haber ninguno queda vacío y avisa que no se detectaron *Test Smells*. Una vez seleccionado un nodo abajo aparecen 2 listas: a la izquierda las reglas no satisfacidas y a la derecha las reglas que se esperaba no satisfacer (está en la lista definida en el método `#expectedTestSmells` de la clase analizada) pero fueron satisfechas. Finalmente, una vez seleccionada el *Test Smell* del nodo aparece una descripción del mismo en la parte inferior de la ventana.

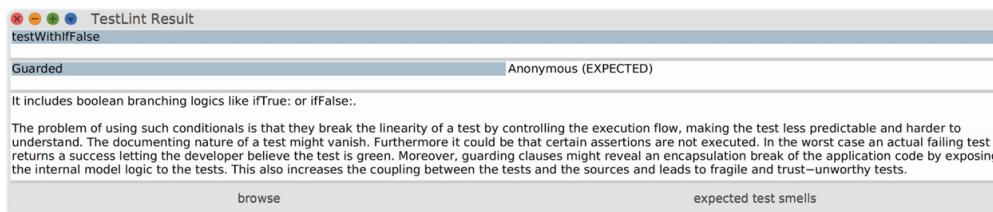


Figura 3.5. Resultado de una corrida de S-TestLint.

3.5. Test Smells

En esta sección se presenta una descripción de los *Test Smells* desarrollados para la herramienta S-TestLint. Por simplicidad todos los *Test Smells* son estáticos. Cada *Test Smell* se le otorga una sección estructurada de la siguiente forma:

- Nombre.
- Tipo de nodo al que aplica.
- Características del *Test Smell*.
- Problemas que genera.
- Implementación.

Además, para cada *Test Smell* se provee un ejemplo en donde se lo comete. Los ejemplos son en código para los *Test Smells* que son sobre métodos de test y para los que son sobre clases o categorías de métodos son en imágenes con la información que utiliza la formalización utilizada para detectarlo.

3.5.1. Anonymous Test

- Nombre: Anonymous Test.
- Tipo de Nodo: Test Method.
- Características: el método de test tiene un nombre no descriptivo.
- Problemas que trae: los tests son documentación, y el nombre es una parte importante porque resume de qué trata el test.
- Implementación: qué es un buen nombre es algo muy dependiente del contexto. Para disminuir los falsos positivos se tomó como heurística que el nombre del test no sea solo un número. También se eligió esta heurística porque en la materia IS1 se enumeran los tests como técnica para generar tests rápidamente y puede suceder que se olviden de renombrar a los tests los alumnos. Ejemplo: `#test01` se lo considera test anónimo y `#test01ejemplo` no se lo considera test anónimo.

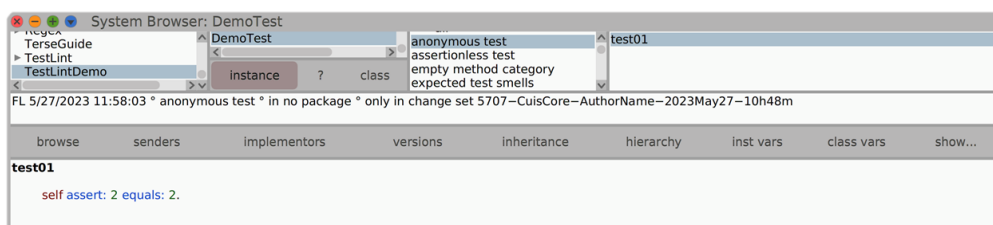


Figura 3.6. El método de test `#test01` cuenta con el smell **Anonymous Test** porque solo está enumerado y no tiene ningún nombre que aporte información.

3.5.2. Assertionless Test

- Nombre: Assertionless Test.
- Tipo de Nodo: Test Method.
- Características: el método de test no tiene ninguna aserción.
- Problemas que trae: solo ejecuta código sin mirar ningún dato, estado o funcionalidad. Además solo puede pasar el test correctamente o arrojar un error.
- Implementación: se basó en la solución realizada en [DDP+19]. Se hace la cláusula transitiva de llamados para cada mensaje del método para ver si terminan en una aserción. Se acota la búsqueda a métodos dentro de la jerarquía de la clase de test.

```
testWithoutAssert
```

```
  | aNumber |
  aNumber := 3.

  aNumber := 3 + 1.
```

Listing 1. Ejemplo de test con smell **Assertionless Test** porque no tiene ninguna aserción. Es un *Smoke Test* que solo evalúa el operador de suma.

```
testWithAssert
```

```
  | aNumber |
  aNumber := 3.

  self assert: aNumber + 1 equals: 4.
```

Listing 2. Ejemplo de test sin smell **Assertionless Test** porque tiene por lo menos una aserción.

3.5.3. Guarded Test

- Nombre: Guarded Test.
- Tipo de Nodo: Test Method.
- Características: el método de test incluye lógica de ramificación.
- Problemas que trae: rompen con la linealidad del test al controlar su flujo de ejecución, haciendo al test menos predecible y más difícil de entender. La naturaleza documentativa del test se puede perder. Además puede hacer que ciertas aserciones no se ejecuten. En el peor caso un test que debería fallar lo notifica como correcto.

- Implementación: se recorre el AST del método en busca de mensajes como `#ifTrue:` o `#ifFalse`.

Por ejemplo el siguiente test no cumple con esta regla, ya que cuenta con el mensaje `#ifFalse`:

```
testWithIfFalse
  4 > 3 ifFalse: [self fail].
  self deny: 3 > 4.
```

Listing 3. Ejemplo de test con smell **Guarded Test** porque cuenta con el mensaje `#ifFalse`.

3.5.4. Long Test

- Nombre: Long Test.
- Tipo de Nodo: Test Method.
- Características: el método de test consiste de demasiado código y sentencias.
- Problemas que trae: hace al test complejo y documenta mal el propósito del test y su aplicación en el código.
- Implementación: se cuenta la cantidad de líneas del código fuente del método compilado y se verifica si excede un umbral. El umbral seleccionado es totalmente arbitrario suficientemente alto para evitar excesivos falsos positivos.

```
testLong

  | result |
  " A lot of stuff "
  self assert: 42 equals: result.
```

Listing 4. Ejemplo de test con smell **Long Test** porque contiene una cantidad de líneas superior al umbral configurado por defecto.

3.5.5. Magic Literals Test

- Nombre: Magic Literals Test.
- Tipo de Nodo: Test Method.
- Características: el método de test tiene un uso excesivo de literales.
- Problemas que trae: demasiados literales distraen y ofuscan la funcionalidad y propósito del test complicando la lectura del test. Además, en caso de que se esté usando la misma o data similar en varios tests, consecuencia de extender o agregar tests sin diseñarlos, trae como consecuencia que sean difíciles de mantener y refactorizar.

- Implementación: se cuenta la cantidad de literales parseando el AST del método test y, al igual que en la regla Long Test, se lo compara con un umbral lo suficientemente grande para evitar muchos falsos positivos.

```
testWithALotOfLiterals
```

```
  | a |
  a := 1@2 + (3@4) + (5@6)+ (7@8)+ (9@10).

  self assert: 25@30 equals: a.
```

Listing 5. Ejemplo de test con smell **Magic Literals Test** porque contiene una cantidad de literales superior al umbral configurado por defecto.

3.5.6. Proper Organization Test

- Nombre: Proper Organization Test.
- Tipo de Nodo: Test Method.
- Características: el método de test no tiene todas las aserciones agrupadas al final del test, violando con la estructura “having-when-assert” que se recomienda que tengan los tests.
- Problemas que trae: dificultad para entender el test.
- Implementación: se analiza todo el método de test viendo si cada mensaje desemboca en una aserción con el mismo método que para la regla “Assertionless Test” verificando que las aserciones detectadas están agrupadas al final. Esta solución trajo los siguientes problemas: hay veces que se utilizan las aserciones al comienzo del test como precondition. Estos casos se pueden exceptuar del análisis definiendo el mensaje `#exceptedTestLintSelectorsForAssertionAnalysis` que retorne los mensajes exceptuados. El otro problema es que a veces para que se entiendan mejor las aserciones se utilizan variables entre las aserciones. Se decidió notificar como falsos positivos estos casos porque no encontramos caso para saber si se están evaluando más cosas o simplemente es para dar claridad. En estos casos si se hace un inline de la variable no se violaría la regla.

Ejemplo:

```
testNotConnectedAsserts
```

```
| aCollection result sizeAfterOperation |
aCollection := OrderedCollection with: 4.

result := aCollection removeFirst.

self assert: 4 equals: result.
sizeAfterOperation := aCollection size.
self assert: 0 equals: sizeAfterOperation.
```

Listing 6. Ejemplo de test con smell **Proper Organization Test** porque tiene separadas las aserciones por la asignación al colaborador **sizeAfterOperation**.

```
testConnectedAsserts
```

```
| aCollection result |
aCollection := OrderedCollection with: 4.

result := aCollection removeFirst.

self assert: 4 equals: result.
self assert: 0 equals: aCollection size.
```

Listing 7. Ejemplo de test sin smell **Proper Organization Test** porque tiene conectados al final las aserciones.

3.5.7. Wrong Assert Usage Test

- Nombre: Wrong Assert Usage Test.
- Tipo de Nodo: Test Method.
- Características: se utiliza el mensaje `#assert:` y se compara por igualdad en su argumento en vez de utilizar el mensaje `#assert:equals`.
- Problemas que trae: dificulta la lectura y sirve para dar a conocer la existencia de la alternativa `#assert:equals`: que para alumnos que comienzan a utilizar Cuis les puede ser novedoso.
- Implementación: se analiza el AST del método de test en busca de `#assert:` que se le pase como argumento “a = b”.

```
testWithWrongAssertUsage
```

```
self assert: 3 = 3.
```

Listing 8. Ejemplo de test con smell **Wrong Assert Usage Test** porque utiliza `#assert:` con el argumento “a = b”.

```
testWithCorrectAssertUsage
```

```
self assert: 3 equals: 3.
```

Listing 9. Ejemplo de test sin smell **Wrong Assert Usage Test** porque utiliza el mensaje `#assert:equals:` de forma correcta.

3.5.8. Unclassified Method Category

- Nombre: Unclassified Method Category.
- Tipo de Nodo: Test Method.
- Características: el test pertenece a la categoría de métodos default, esta es la que se utiliza cuando no se le coloca categoría al método.
- Problemas que trae: similar a tener un mal nombre de test, no tener categoría dificulta el entendimiento del objetivo del test.
- Implementación: se observa si la categoría a la que pertenece el método de test es la default.

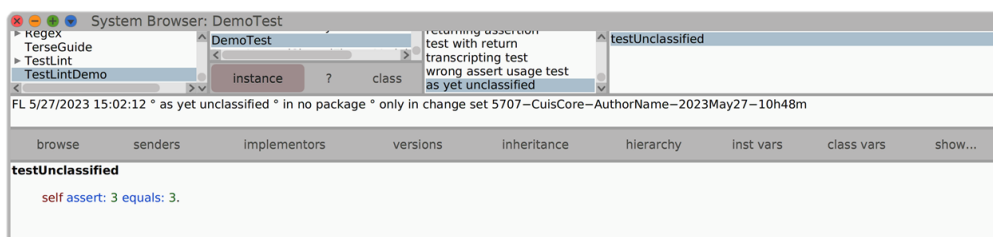


Figura 3.7. El método de test `#testUnclassified` cuenta con el smell **Unclassified Method Category** porque no pertenece a ninguna categoría de clase.

3.5.9. Test With Return

- Nombre: Test With Return.
- Tipo de Nodo: Test Method.
- Características: el test tiene un return explícito.

- Problemas que trae: al igual que en Guarded Test, rompen con la linealidad del test al controlar su flujo de ejecución, haciendo posible que haya código inaccesible.
- Implementación: se recorre el AST del método en busca de un return explícito.

```
testWithReturn
```

```
  ^self assert: 3 equals: 3
```

Listing 10. Ejemplo de test con smell **Test With Return** porque contiene un return.

```
testWithoutReturn
```

```
  self assert: 3 equals: 3
```

Listing 11. Ejemplo de test sin smell **Test With Return** porque no contiene un return.

3.5.10. Transcribing Test

- Nombre: Transcribing Test.
- Tipo de Nodo: Test Method.
- Características: el test escribe a la consola.
- Problemas que trae: el test tiene parte manual al tener que ver los datos por consola para verificar la correctitud del test. Todo lo que se escribe por consola puede agregarse a las aserciones para que el test sea completamente automático sin necesidad de verificar manualmente.
- Implementación: se recorre el AST del método en busca de mención de la clase `#Transcript`.

```
testThatUsesTranscript
```

```
  | operationResult |
  operationResult := 2 + 1.
  Transcript show: operationResult.

  self assert: operationResult > 2.
```

Listing 12. Ejemplo de test con smell **Transcribing Test** porque utiliza a la clase `#Transcript`.

3.5.11. Returning Assertion

- Nombre: Returning Assertion.
- Tipo de Nodo: Test Method.
- Características: se utiliza lo que retorna una aserción.
- Problemas que trae: las aserciones no deberían retornar nada porque deberían tener solo esa responsabilidad, arrojar una excepción si no se cumple. La otra posibilidad es que se estaría usando el return implícito. En ese caso por claridad conviene utilizar el objeto de forma directa y no desde el valor retornado para no dificultar la lectura del test.
- Implementación: se recorre el AST en busca de aserciones que utilizan su valor retornado, ya sea asignando el valor a una variable o pasando el valor como argumento de otro mensaje.

```
testWithReturningAssertion
  | assertionReturnedValue |
  assertionReturnedValue := self assert: 2 equals: 2.
  self assert: assertionReturnedValue equals: self.
```

Listing 13. Ejemplo de test con smell **Returning Assertion** porque utiliza el valor retornado por `#assert:equals:`.

3.5.12. Unusual Test Order

- Nombre: Unusual Test Order.
- Tipo de Nodo: Test Method.
- Características: el método de test llama a otro test.
- Problemas que trae: los métodos de test deberían ser solo llamados por el framework de testing. Modificaciones en el test que se llama puede generar cambios inesperados en este test perdiendo una independencia deseada entre tests.
- Implementación: se recorre el AST en busca de llamadas a tests.

```
testWithReturningAssertion

  self testThatShouldNotBeCalledHere.
```

Listing 14. Ejemplo de test con smell **Unusual Test Order** porque envía el mensaje `#testThatShouldNotBeCalledHere`.

3.5.13. Mixed Selectors

- Nombre: Mixed Selectors.
- Tipo de Nodo: Method Category.
- Características: mezclar métodos de test y otros métodos en una misma categoría de métodos.
- Problemas que trae: rompe con convenciones organizacionales de testing comunes, dificultando.
- Implementación: se identifica que todos los métodos de la categoría sean del mismo tipo.

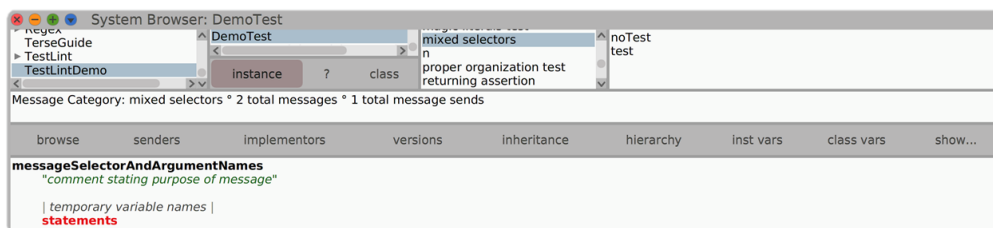


Figura 3.8. La categoría de métodos **mixed selectors** cuenta con el smell **Mixed Selectors** porque contiene el método de test **test** y el método **noTest** que no es de test.

3.5.14. Test Method Category Name

- Nombre: Test Method Category Name.
- Tipo de Nodo: Method Category.
- Características: la categoría de métodos tiene un nombre no descriptivo.
- Problemas que trae: el nombre de la categoría de métodos es documentación para describir qué métodos contiene.
- Implementación: qué es un buen nombre es algo que depende del contexto. Para disminuir los positivos se tomó como heurística que el nombre sea menor o igual a 1.

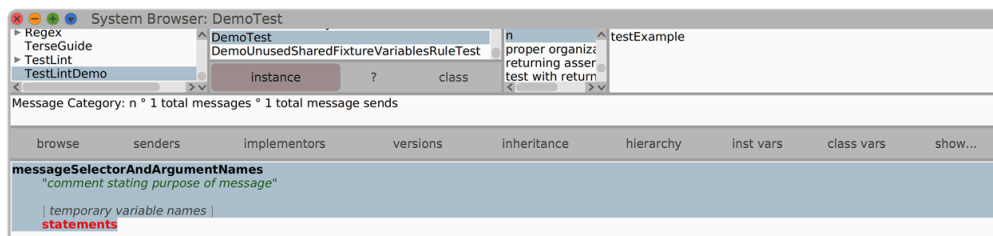


Figura 3.9. Ejemplo de resultado de la herramienta para la categoría de métodos **n** con smell **Test Method Category Name** porque la categoría tiene un mal nombre bajo la heurística que utiliza la herramienta.

3.5.15. Empty Method Category

- Nombre: Empty Method Category.
- Tipo de Nodo: Method Category.
- Características: tener una categoría de métodos sin métodos.
- Problemas que trae: dificulta el entendimiento de la clase.
- Implementación: se identifica si la colección de métodos asociados a la categoría está vacía.

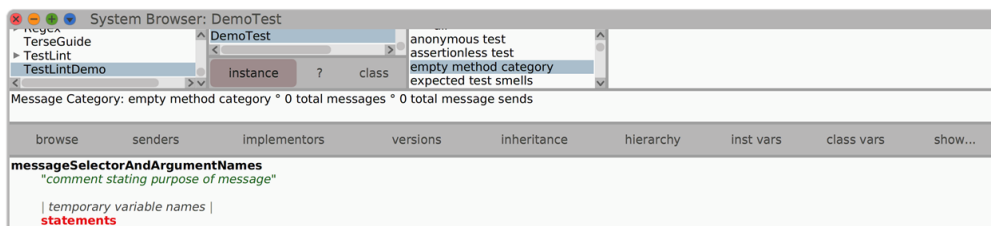


Figura 3.10. La categoría de métodos **empty method category** cuenta con el smell **Empty Method Category** porque no contiene ningún método.

3.5.16. Unused Shared-Fixture Variables

- Nombre: Unused Shared-Fixture Variable.
- Tipo de Nodo: Test Class.
- Características: una variable de instancia de la clase de test no se está utilizando en ningún test.
- Problemas que trae: tener una variable de instancia que no se utiliza en ningún test no tiene razón de ser, dificulta el entendimiento de la clase.
- Implementación: se verifica que haya un acceso desde un test para cada variable de instancia.

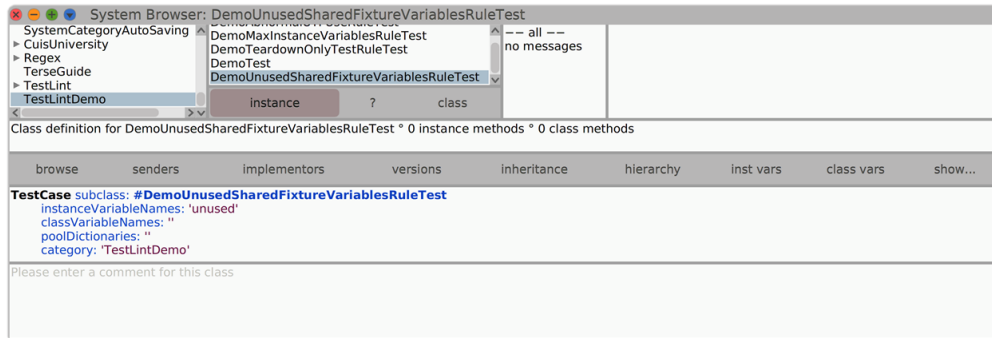


Figura 3.11. La clase `DemoUnusedSharedFixtureVariablesRuleTest` cuenta con el smell **Unused Shared-Fixture Variables** porque contiene el colaborador interno `unused` que no es utilizado en ningún método de test de la clase, ya que la clase no contiene ningún método.

3.5.17. Max Instance Variables

- Nombre: Max Instance Variables.
- Tipo de Nodo: Test Class.
- Características: la clase de test tiene demasiadas variables de instancia.
- Problemas que trae: tener tantas variables de instancia dificulta el entendimiento de la clase. Hay 2 posibilidades en las que se esté cayendo al tener muchas variables de instancia: que se usen en todos los tests todas las variables y entonces probablemente sea posible generar una abstracción que agrupe variables - que no se usen en todos los tests dificultando el entendimiento del propósito de cada variable de instancia.
- Implementación: se cuenta la cantidad de variables de instancia de la clase de test y se verifica si excede un umbral. El umbral seleccionado es totalmente arbitrario suficientemente alto para evitar excesivos falsos positivos.

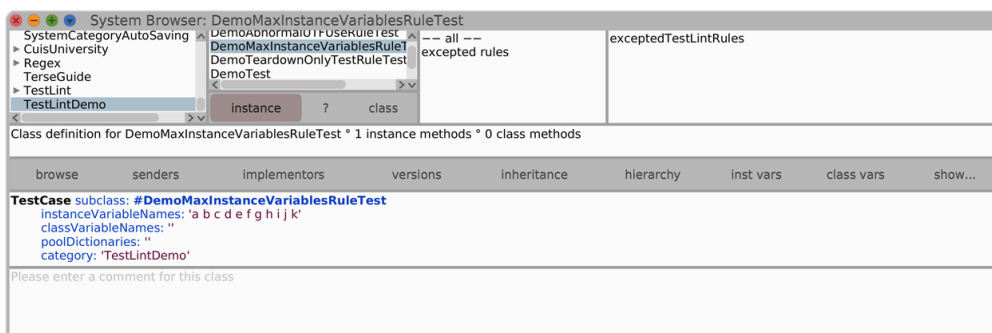


Figura 3.12. La clase `DemoMaxInstanceVariablesRuleTest` cuenta con el smell **Max Instance Variables** porque contiene 11 colaboradores internos y supera la cantidad de líneas que permite el umbral configurado por defecto.

3.5.18. Teardown Only Test

- Nombre: Teardown Only Test.
- Tipo de Nodo: Test Class.
- Características: la clase de test solo define un método de “tearDown” y no uno de “setUp”.
- Problemas que trae: es indicio que alguna estructura no es está manejando correctamente si solo se la utiliza en el método “tearDown”.
- Implementación: se valida que no suceda que la clase tenga definido el método “tearDown” y no el método “setUp”.

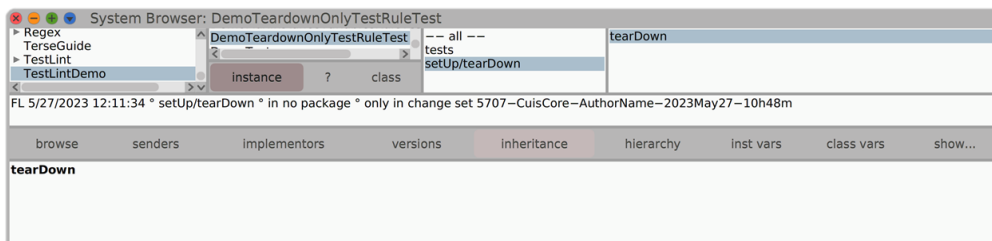


Figura 3.13. La clase `DemoTeardownOnlyTestRuleTest` con smell **Teardown Only Test** porque define el mensaje `#tearDown` y no el mensaje `#setUp`.

3.5.19. Abnormal UTF-Use

- Nombre: Abnormal UTF-Use.
- Tipo de Nodo: Test Class.
- Características: la clase de test sobrescribe código del framework de testing.
- Problemas que trae: ya no tiene asegurado funcionar correctamente el framework de testing.
- Implementación: se valida que no se redefinen métodos del framework de tests a excepción de los métodos “setUp” y “tearDown” que es esperado que sean redefinidos.

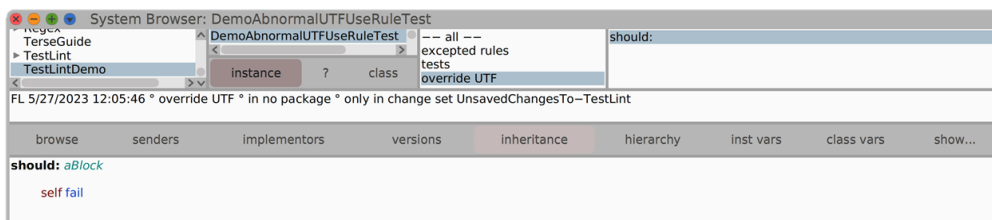


Figura 3.14. La clase `DemoAbnormalUTFUseRuleTest` con smell **Abnormal UTF-Use** porque redefine el mensaje `#should:`.

Caso de estudio

4.1. Metodología

El caso de estudio fue realizado sobre la versión 5981 de Cuis University incluyendo la última versión de S-TestLint. Este último paquete es Open-Source y puede ser descargado en [GitHub](#). Para mapear y visualizar datos utilizamos Google Sheets.

Se utilizaron los siguientes conjuntos de datos:

Exámenes de estudiantes de Ingeniería del Software II de la FCEN 2022. 2do Parcial 1er Cuatrimestre y 2do Parcial 2do Cuatrimestre. Se seleccionaron segundos parciales porque en esa materia para evaluarlos con ya conocimiento adquirido.

Paquetes de la imagen de Cuis University con el fin de analizar tests de desarrolladores con mayor nivel de experiencia.

En el [Cuadro 4.1](#) se muestra la composición de cada conjunto de datos.

Cuadro 4.1. Descripción general de la composición del caso de estudio.

Paquetes Imagen Cuis	47
Paquetes 1er Cuatrimestre	32
Paquetes 2do Cuatrimestre	48

4.2. Resultados y conclusiones

4.2.1. 2do Parcial - 1er Cuatrimestre 2022

En la [Figura 4.1](#) se muestra la nota obtenida por cada alumno y la cantidad de *Test Smells* cometidos. Se puede ver que no hay suficientes datos para concluir que haya una correlación entre la nota obtenida por los alumnos en el parcial y la cantidad de *Test Smells* cometidos. Hay un alumno que obtuvo la calificación 10 y no cometió ningún *Test Smell* pero a la vez hay varios alumnos que obtuvieron una nota rondando al 9 y con la mayor cantidad de *Test Smells*. También alumnos con una calificación baja cometieron pocos *Test Smells*. Esto se puede deber a que muchos parciales desaprobados son debidos a que no cumplieron con todo lo que se pidió desarrollar en dicho examen ni los tests que deben acompañar ese desarrollo.

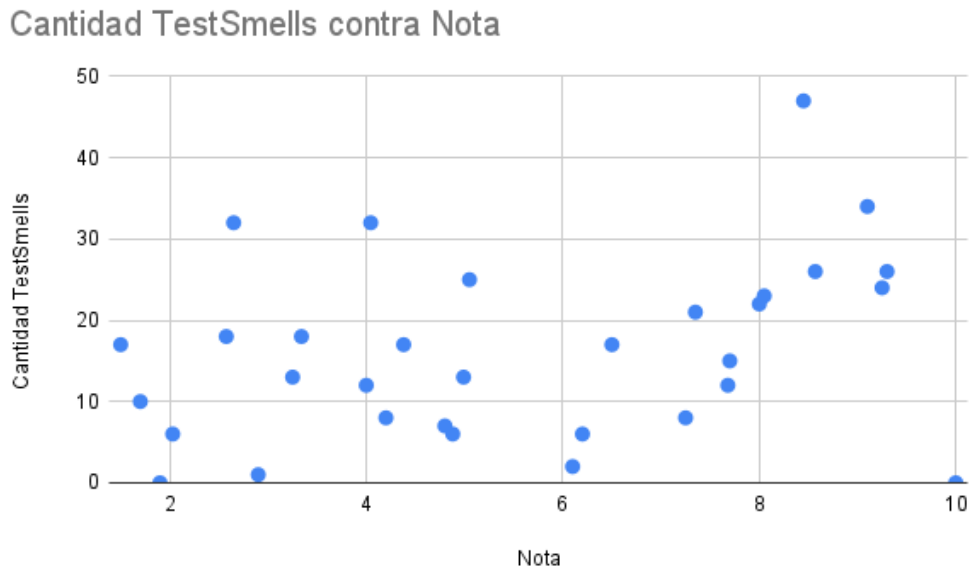


Figura 4.1. Cantidad de Test Smells contra nota para 2do Parcial - 1er Cuatrimestre 2022.

Para evitar el problema que surge por que a mayor cantidad de tests hay mayor margen para cometer *Test Smells* en la Figura 4.2 se muestra la nota obtenida por cada alumno y la cantidad de *Test Smells* cometidos sin repetir, es decir, se cuenta cada *Test Smell* cometido a lo sumo una vez sin importar en cuantos tests fue cometido. Se puede observar que no hay información suficiente para concluir que hay una correlación entre la nota obtenida y la cantidad de *Test Smells* cometidos. Esto se puede deber a que el criterio de corrección de los exámenes difieren con las *Test Smells* que reconoce S-TestLint.

Cantidad TestSmells contra Nota (Sin repetir)

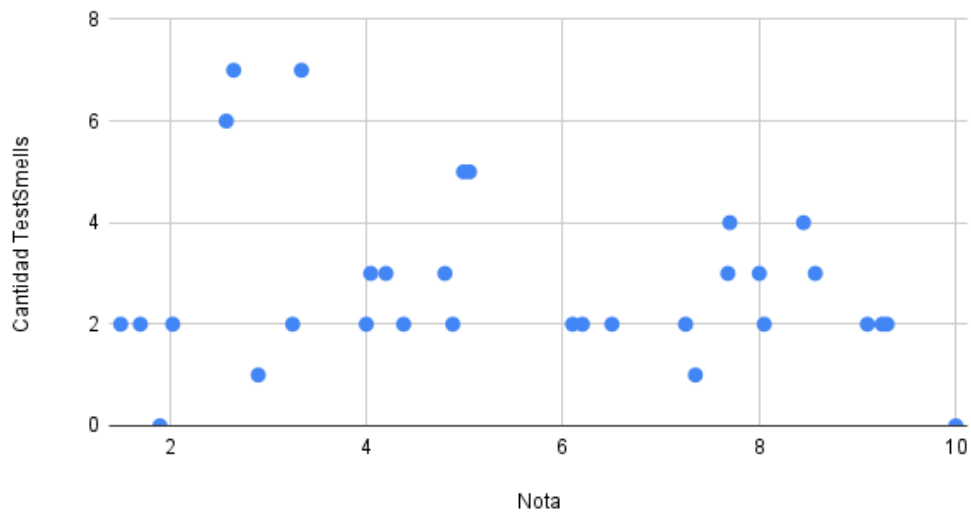


Figura 4.2. Cantidad de SmellTest contra nota sin repetir para 2do Parcial - 1er Cuatrimestre 2022.

En la [Figura 4.3](#) se muestra la cantidad de tests de cada examen ordenados por cantidad con su respectiva cantidad de *Test Smells* cometidos. Se corrobora que a mayor cantidad de tests mayor es la cantidad de *Test Smells*.

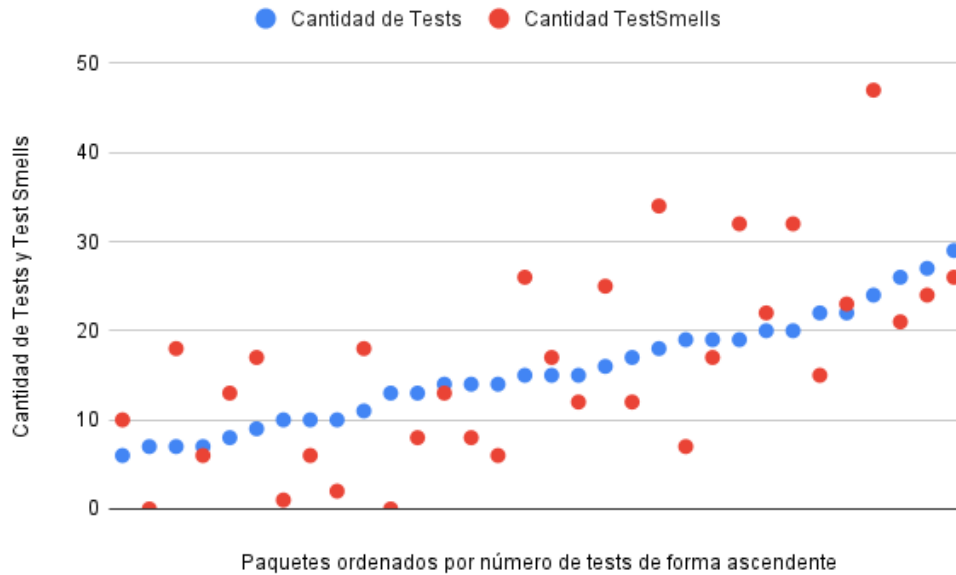


Figura 4.3. Cantidad de tests y Test Smells para cada examen ordenado por cantidad de tests.

En la [Figura 4.4](#) se muestra la cantidad de veces que se cometió cada *Test Smell* en los exámenes. Se puede destacar que son un grupo de 3 *Test Smells* los más cometidos. El más cometido fue **Magic Literals** que el gran número se puede deber a que una vez que se colocan demasiados literales para el setup de un test este se replica para el resto. Esto también puede indicar que el umbral para identificar este *Test Smell* es demasiado exigente. En segundo lugar está **Unclassified Method Category** que se puede deber a que por la exigencia de tener que realizar el examen en un tiempo reducido no se llega a colocar en categorías de métodos a todos los métodos. En tercer lugar está **Long** que se puede deber a que no llegan a extraer código repetido de los tests.

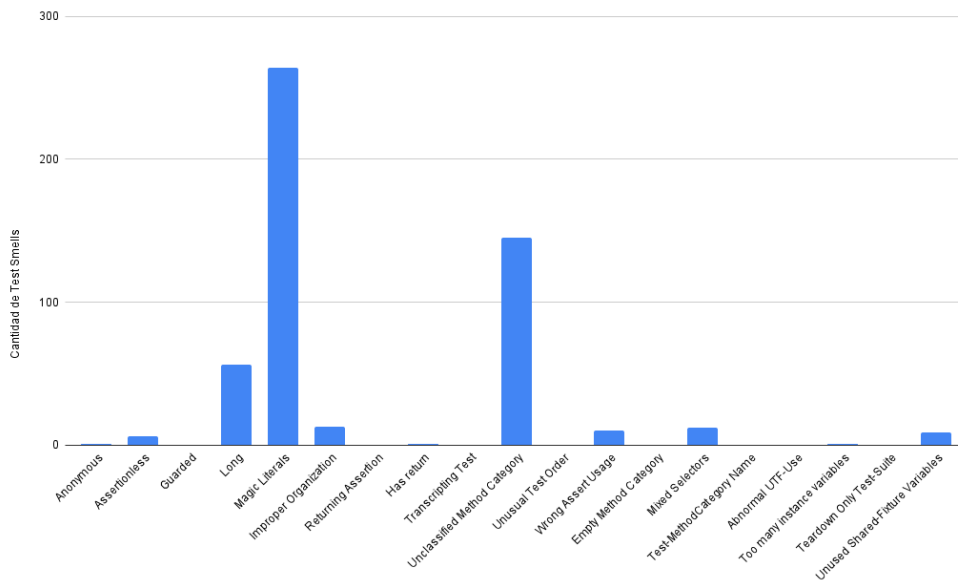


Figura 4.4. Cantidad de veces que se cometió cada tipo de Test Smell para 2do Parcial - 1er Cuatrimestre 2022.

En la [Figura 4.5](#) se muestra muestra la cantidad de veces que se cometió cada *Test Smell* en los exámenes sin repetir. Se puede observar que se mantiene el mismo grupo de 3 *Test Smells* más cometidos que en el caso que se contaban repetidos.

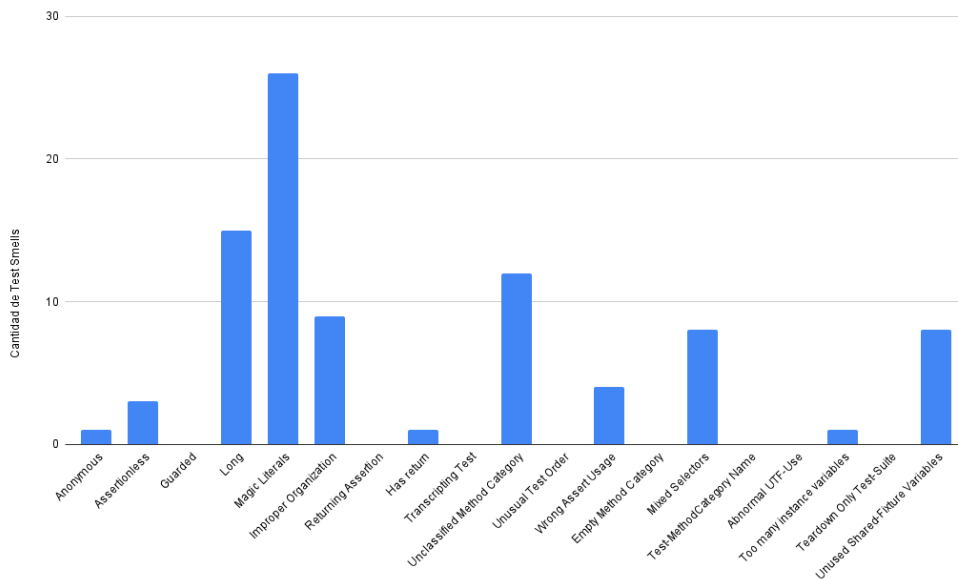


Figura 4.5. Cantidad de veces que se cometió cada tipo de Test Smell sin repetir para 2do Parcial - 1er Cuatrimestre 2022.

4.2.2. 2do Parcial - 2do Cuatrimestre 2022

En la [Figura 4.6](#) se muestra la nota obtenida por cada alumno y la cantidad de *Test Smells* cometidos. Se observa que, al igual que en cuatrimestre anterior, no hay información suficiente para determinar correlación entre la nota obtenida y la cantidad de *Test Smells* cometidos.

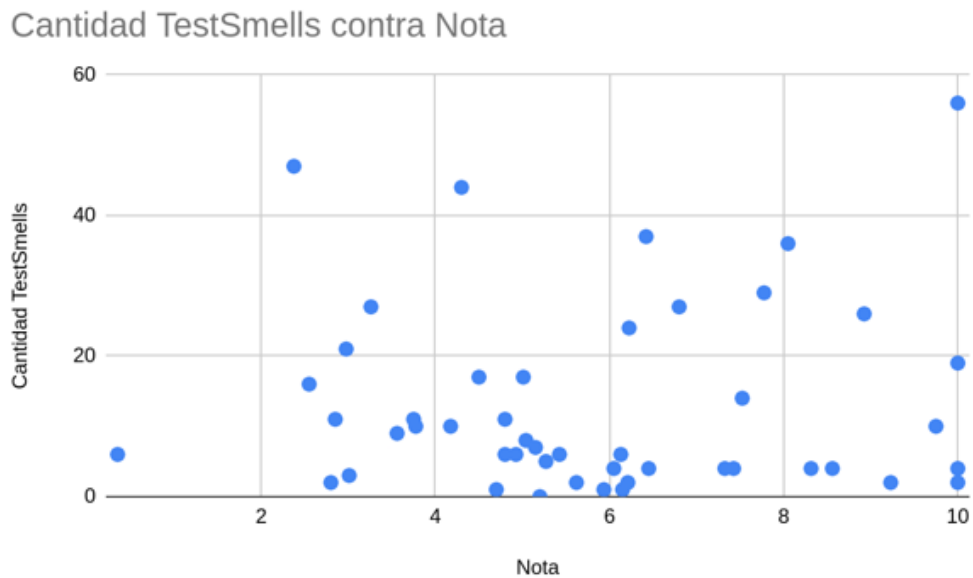


Figura 4.6. Cantidad de Test Smells contra nota para 2do Parcial - 2do Cuatrimestre 2022.

En la [Figura 4.7](#) se muestra la nota obtenida por cada alumno y la cantidad de *Test Smells* cometidos sin repetir. Se observa distribución similar al cuatrimestre anterior.

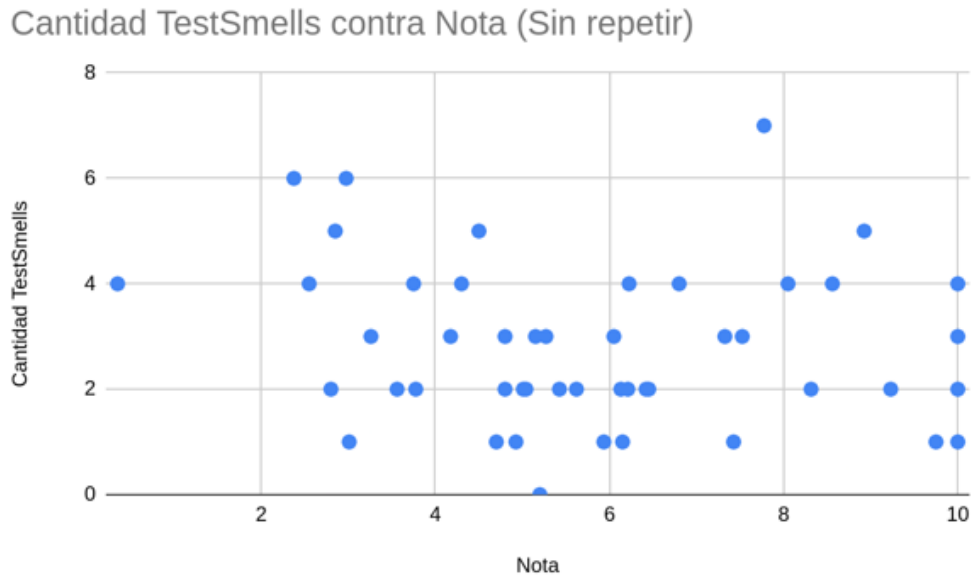


Figura 4.7. Cantidad de SmellTest contra nota sin repetir para 2do Parcial - 2do Cuatrimestre 2022.

En la [Figura 4.8](#) se muestra la cantidad de tests de cada examen ordenados por cantidad con su respectiva cantidad de *Test Smells* cometidos. En este cuatrimestre se puede destacar que gran parte de los alumnos con mayor cantidad de tests cometió una baja cantidad de *Test Smells*.

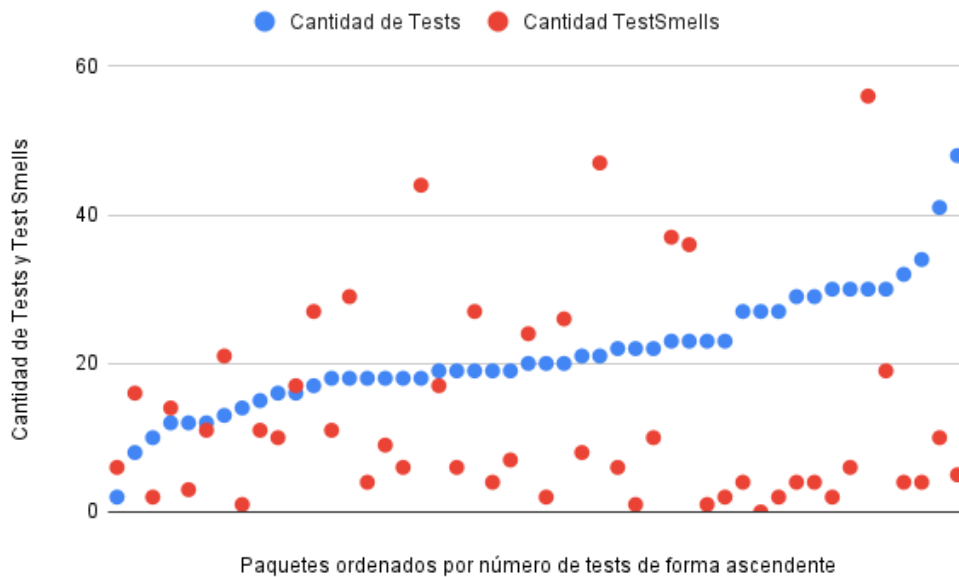


Figura 4.8. Cantidad de tests y Test Smells para cada examen ordenado por cantidad de tests.

En la [Figura 4.9](#) se muestra la cantidad de veces que se cometió cada *Test Smell* en los exámenes. Se puede destacar que son un grupo de 2 *Test Smells* los más cometidos y, al igual que en cuatrimestre anterior estos son **Magic Literals** y **Unclassified Method Category**.

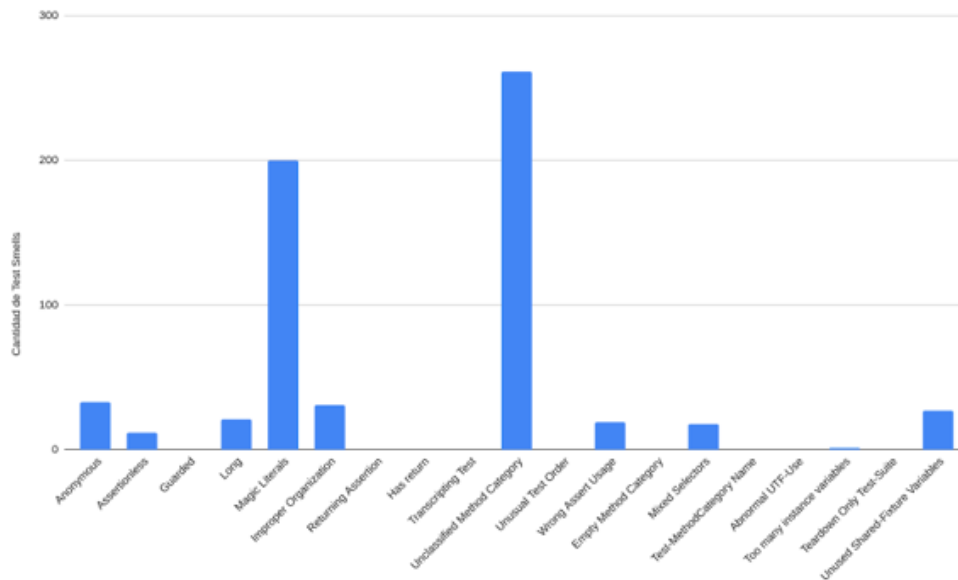


Figura 4.9. Cantidad de veces que se cometió cada tipo de Test Smell para 2do Parcial - 2do Cuatrimestre 2022.

En la [Figura 4.10](#) se muestra muestra la cantidad de veces que se cometió cada *Test Smell* en los exámenes sin repetir. Al igual que el cuatrimestre anterior **Magic Literals** se mantiene como el *Test Smell* más cometido. Sin embargo, **Unclassified Method Category** es superado por los *Test Smells* **Mixed Selectors** y **Unused Shared-Fixture Variables**. Ambos se pueden deber al mismo motivo que **Unclassified Method Category**, los alumnos no tuvieron el tiempo de quitar variables de instancia que no son utilizadas ni clasificar de forma separada a los métodos de test del resto de los métodos.

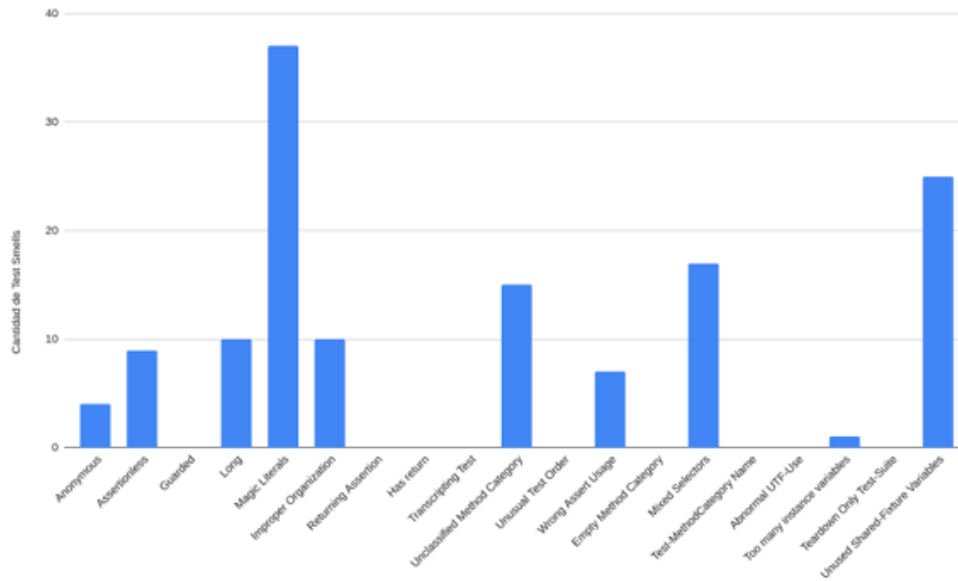


Figura 4.10. Cantidad de veces que se cometió cada tipo de Test Smell sin repetir para 2do Parcial - 2do Cuatrimestre 2022.

4.2.3. Paquetes de Cuis University

En la [Figura 4.11](#) se muestra la cantidad de *Test Smells* sobre la cantidad tests para cada paquete de Cuis University. Se puede destacar que para más de la mitad de los paquetes el valor es menor a 1 lo que quiere decir que tienen más tests que *Test Smells*.

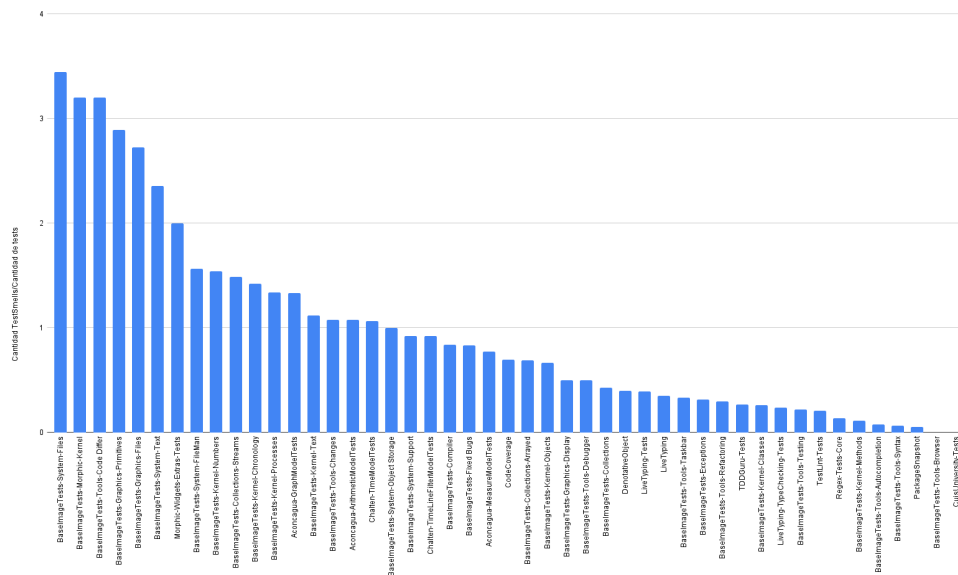


Figura 4.11. Cantidad de Test Smells sobre cantidad de tests para cada paquete de Cuis University.

En la [Figura 4.12](#) se muestra la cantidad de *Test Smells* sin repetir sobre la cantidad de tests para cada paquete de Cuis University. Como la cantidad de *Test Smells* está acotada por la cantidad que detecta la herramienta, los paquetes con gran cantidad de tests van a tener el valor bajo. Solo los paquetes con baja cantidad de tests pueden presentar un valor alto.

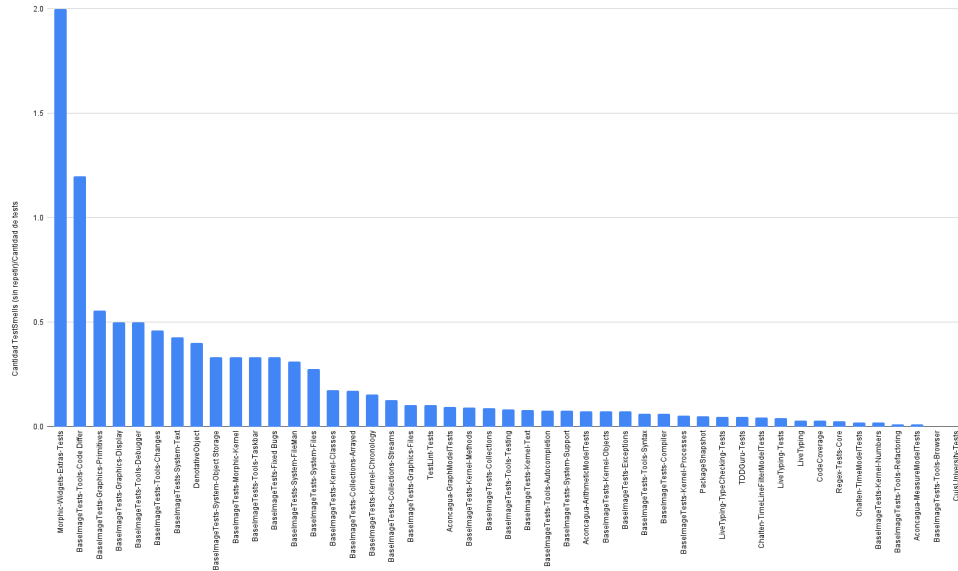


Figura 4.12. Cantidad de Test Smells sin repetir sobre cantidad de tests para cada paquete de Cuis University.

En la [Figura 4.13](#) se muestra la cantidad de tests de cada paquete ordenados por su cantidad con su respectiva cantidad de *Test Smells*. A diferencia de los parciales de alumnos, la cantidad de *Test Smells* suele ser inferior a la cantidad de tests (como se indicó en [Figura 4.11](#)). Esto se puede deber a que al ser desarrolladores más experimentados realizan menos *Test Smells* que estudiantes en formación.

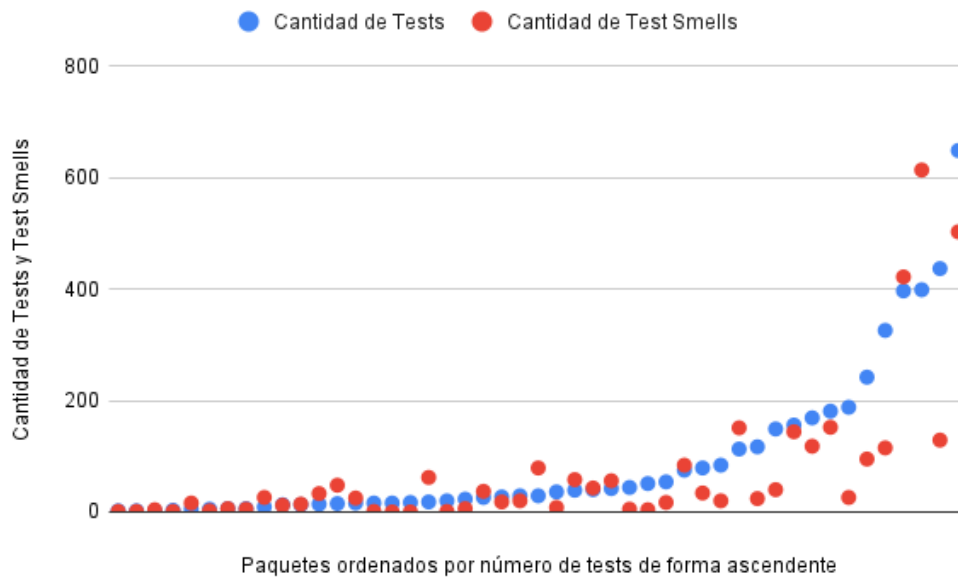


Figura 4.13. Cantidad de tests y Test Smells para cada paquete ordenado por cantidad de tests.

En la [Figura 4.14](#) se muestra la cantidad de veces que se cometió cada *Test Smell* en cada paquete de Cuis University. Cabe destacar a **Wrong Assert Usage Test** que es el más cometido. Esto se puede deber a que son paquetes implementados previos al agregado del mensaje `#assert:equals`.

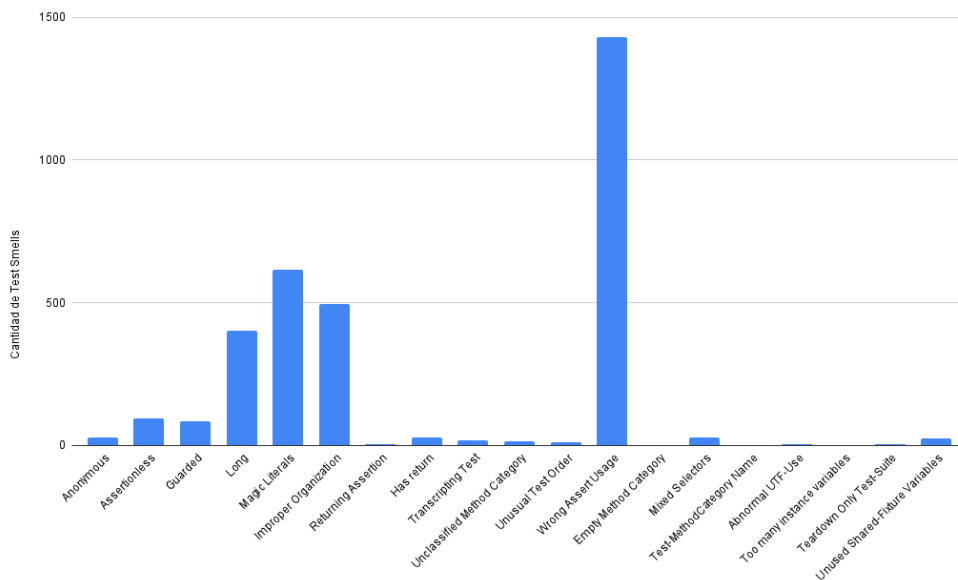


Figura 4.14. Cantidad de veces que se cometió cada tipo de Test Smell en los paquetes de Cuis University.

En la [Figura 4.15](#) se muestra la cantidad de veces que se cometió cada *Test Smell* sin repetir en cada paquete de Cuis University. Lo primero que se ve es que no hay un grupo reducido de *Test Smells* más cometidos como ocurría en los exámenes. Lo segundo es que ocurre en mayor cantidad el *Test Smell* **Improper Organization**. Esta diferencia se puede deber a que en la materia se hace énfasis en la enseñanza de la estructura que exigida para no cometer el *Test Smell*. También se puede observar que se cometen *Test Smells* que en los parciales no como **Guarded**, **Transcripting Test** y **Teardown Only Test Suite** debido a que presentan una complejidad los paquetes desarrollados a la de los parciales.

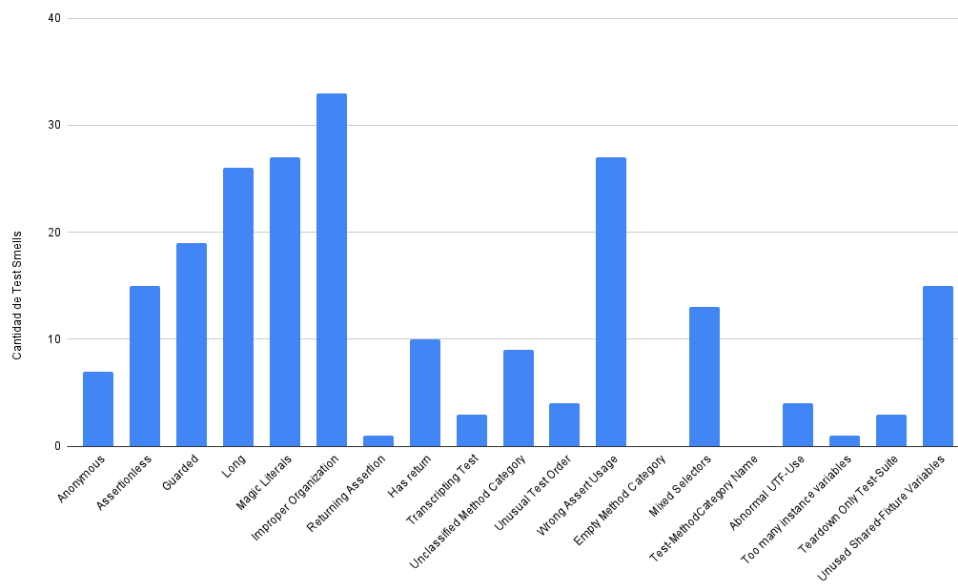


Figura 4.15. Cantidad de veces que se cometió cada tipo de Test Smell sin repetir en los paquetes de Cuis University.

Conclusiones

5.1. Conclusiones obtenidas a partir del desarrollo de la herramienta

Este trabajo produjo *S-TestLint*, una herramienta para detectar *Test Smells* en test suites, lista para su uso. Los estudiantes de la materia Ingeniería del Software I de la carrera de Licenciatura en Ciencias de la Computación ya comenzaron a utilizarla. De su implementación se concluyen varios aspectos: el gran número de *Test Smells* que los desarrolladores pueden cometer, que la UI es un desafío tan grande como la misma herramienta y es destacable el gran aprendizaje que requirió el desarrollo en cuanto a las limitaciones y las virtudes del AST en Cuis Smalltalk. Este trabajo servirá como un repositorio de ejemplos para este tipo de tareas y permitirá evaluar la dificultad de implementar otras en el futuro. Finalmente, uno de los principales objetivos de este trabajo era servir de base para el estudio posterior de su uso. Al estar siendo utilizada por los desarrolladores permitirá ver cómo se puede mejorar. Además, permitirán servir como base para la implementación de reglas de detección de *Test Smells* más complejas.

5.2. Conclusiones sobre los resultados estadísticos

5.3. Trabajo futuro

5.3.1. UI

S-TestLint es incómodo de configurar. Sería positivo no tener que definir métodos (`#expectedTestSmells` y `#exceptedTestLintRules`) para configurar la corrida. Algo como lo que ofrece Doctor Tests [GCDD19] para proveer toda la funcionalidad, esconder todos los detalles del modelo y que sea fácil e intuitiva de usar la UI.

5.3.2. *Test Smells* dinámicos

S-TestLint solo detecta *Test Smells* estáticos. El agregado de *Test Smells* dinámicos ayudaría a detectar una mayor cantidad de errores que el análisis estático no llega.

Apéndices

Cantidad de tests	Anonymous	Assertionless	Guarded	Long	Magic Literals	Improper Organization	Returning Assertion	Has return	Transcripting Test	Unclassified Method Category	Unusual Test Order	Wrong Assert Usage	Empty Method Category	Mixed Selectors	Test-MethodCategory Name	Abnormal UTF-Use	Too many instance variables	Teardown Only Test-Suite	Unused Shared-Fixture Variables	Nota	
2	2	0	0	0	0	0	0	0	0	0	2	0	0	0	1	0	0	0	0	1	0.35
8	8	0	0	0	2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	2.55
10	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	2.8
12	0	1	0	0	0	0	0	0	0	0	12	0	0	0	1	0	0	0	0	0	7.525
12	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3.01
12	0	0	0	0	2	4	4	0	0	0	0	0	0	0	1	0	0	0	0	0	3.75
13	0	4	0	0	1	0	0	0	0	0	13	0	1	0	1	0	0	0	0	1	2.975
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	4.7
15	2	0	0	0	0	3	4	0	0	0	0	0	0	0	1	0	0	0	0	1	2.85
16	0	0	0	0	0	7	2	0	0	0	0	0	0	0	0	0	0	0	0	1	4.175
16	0	0	0	0	0	1	0	0	0	0	16	0	0	0	0	0	0	0	0	0	5.01
17	0	0	0	0	0	7	0	0	0	0	17	0	3	0	0	0	0	0	0	0	3.26
18	0	0	0	0	0	0	9	0	0	0	0	0	0	0	0	0	0	0	0	1	4.8
18	0	1	0	0	1	6	1	0	0	0	18	0	0	0	1	0	0	0	0	1	7.775
18	0	1	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	1	7.325
18	0	0	0	0	0	8	0	0	0	0	0	0	0	0	1	0	0	0	0	0	3.56
18	0	0	0	0	0	4	2	0	0	0	0	0	0	0	0	0	0	0	0	0	6.13
18	0	0	0	0	7	18	0	0	0	0	18	0	0	0	0	0	0	0	0	1	4.3
19	0	0	0	0	0	1	0	0	0	0	9	0	2	0	2	0	0	0	0	3	4.5
19	0	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	4.8
19	0	0	0	0	0	3	0	0	0	0	19	0	4	0	1	0	0	0	0	0	6.8
19	0	0	0	0	0	1	3	0	0	0	0	0	0	0	0	0	0	0	0	0	6.45
19	0	0	0	0	0	1	0	0	0	0	0	0	5	0	0	0	0	0	0	1	5.15
20	0	0	0	0	0	2	0	0	0	0	20	0	0	0	1	0	0	0	0	1	6.225
20	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9.23
20	0	0	0	0	1	3	0	0	0	0	20	0	0	0	1	0	0	0	0	1	8.925
21	0	0	0	0	0	7	0	0	0	0	0	0	0	0	0	0	0	0	0	1	5.04
21	21	1	0	0	0	0	0	0	0	0	21	0	2	0	1	0	0	0	0	1	2.375
22	0	0	0	0	0	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4.925
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	6.15
22	0	0	0	0	1	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3.775
23	0	0	0	0	0	14	0	0	0	0	23	0	0	0	0	0	0	0	0	0	6.42
23	0	0	0	0	0	11	0	0	0	0	23	0	0	0	1	0	0	0	0	1	8.05
23	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5.935
23	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	5.62
27	0	1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	8.56
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5.2
27	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	6.21
29	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7.425
29	0	0	0	0	0	3	0	0	0	0	0	0	0	0	1	0	0	0	0	0	8.315
30	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	10
30	0	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	5.425
30	0	0	0	0	3	22	0	0	0	0	30	0	0	0	0	0	0	0	0	1	10
30	0	0	0	0	2	15	2	0	0	0	0	0	0	0	0	0	0	0	0	0	10
32	0	1	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10
34	0	0	0	0	0	2	1	0	0	0	0	0	0	0	1	0	0	0	0	0	6.05
41	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9.75
48	0	1	0	0	0	0	3	0	0	0	0	0	0	0	1	0	0	0	0	0	5.27

Test Smells - Paquetes de CuisUniversity

	Cantidad de tests	Anonymous	Assertionless	Guarded	Long	Magic Literals	Improper Organization	Returning Assertion	Has return	Transcribing Test	Unclassified Method Category	Unusual Test Order	Wrong Assert Usage	Empty Method Category	Mixed Selectors	Test-Method/Category Name	Abnormal UTF-Use	Too many instance variables	Teardown Only Test-Suite	Unused Shared-Fixture Variables
TDDGuru-Tests	149	0	1	0	4	30	0	0	1	0	1	0	0	0	0	0	0	0	1	2
BaseImageTests-Kernel-Objects	27	0	0	0	6	0	0	0	0	0	0	0	12	0	0	0	0	0	0	0
BaseImageTests-Kernel-Classes	23	0	1	0	0	0	0	3	0	1	0	0	0	0	0	0	0	0	0	1
BaseImageTests-Kernel-Numbers	390	0	1	19	50	189	77	0	0	0	2	0	275	0	0	0	1	0	0	0
BaseImageTests-Kernel-Text	75	0	0	1	14	22	5	0	0	0	0	0	33	0	0	0	0	0	0	0
BaseImageTests-Kernel-Chronology	26	0	0	0	4	5	7	0	0	0	0	0	21	0	4	0	0	0	0	0
BaseImageTests-Kernel-Methods	44	0	1	0	0	0	0	2	0	0	0	0	0	0	1	0	0	0	0	1
BaseImageTests-Kernel-Processes	113	0	0	5	47	7	83	0	0	0	4	0	5	0	0	0	0	0	0	0
BaseImageTests-Collections	79	1	0	1	2	5	6	0	0	0	0	0	18	0	1	0	0	0	0	0
BaseImageTests-Collections-Arrayed	29	0	0	0	4	5	1	0	0	0	1	0	9	0	0	0	0	0	0	0
BaseImageTests-Collections-Streams	39	0	0	12	7	11	8	0	0	0	0	0	20	0	0	0	0	0	0	0
BaseImageTests-Exceptions	54	5	0	0	0	0	1	0	0	3	0	0	8	0	0	0	0	0	0	0
BaseImageTests-Compiler	181	6	52	7	13	8	22	0	6	0	0	0	31	0	3	0	1	0	0	3
BaseImageTests-System-Support	13	0	0	0	0	0	0	0	0	12	0	0	0	0	0	0	0	0	0	0
BaseImageTests-System-Text	14	3	0	1	9	6	9	0	0	0	0	0	5	0	0	0	0	0	0	0
BaseImageTests-System-FileMan	16	0	0	2	2	4	14	0	0	0	0	0	3	0	0	0	0	0	0	0
BaseImageTests-System-Files	18	0	0	6	10	13	15	0	0	0	0	0	18	0	0	0	0	0	0	0
BaseImageTests-System-Object Storage	6	0	0	0	0	0	4	0	0	0	0	0	2	0	0	0	0	0	0	0
BaseImageTests-Graphics-Primitives	9	0	0	6	8	8	2	0	0	0	0	0	2	0	0	0	0	0	0	0
BaseImageTests-Graphics-Display Objects	2	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
BaseImageTests-Graphics-Files	29	0	0	0	0	21	29	0	0	0	0	0	29	0	0	0	0	0	0	0
BaseImageTests-Morphic-Kernel	15	0	0	3	9	14	8	0	0	0	0	0	14	0	0	0	0	0	0	0
BaseImageTests-Tools-Browser	17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
BaseImageTests-Tools-Changes	13	0	1	0	2	0	0	0	1	0	0	8	0	0	1	0	0	0	1	0
BaseImageTests-Tools-Syntax Highlighting	16	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
BaseImageTests-Tools-Debugger	2	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
BaseImageTests-Tools-Autocompletion	51	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0	1
BaseImageTests-Tools-Taskbar	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
BaseImageTests-Tools-Testing	36	0	4	0	0	0	3	0	1	0	0	0	0	0	0	0	0	0	0	0
BaseImageTests-Tools-Code Differ	5	0	1	3	4	1	3	0	0	0	0	0	4	0	0	0	0	0	0	0
BaseImageTests-Tools-Refactoring	437	0	0	0	80	10	35	0	0	0	0	0	0	0	2	0	0	0	0	2
BaseImageTests-Fixed Bugs	6	0	0	0	0	0	4	0	0	0	0	0	0	0	1	0	0	0	0	0
LiveTyping	326	0	9	1	48	8	33	0	8	9	1	1	0	0	5	0	0	0	0	1
LiveTyping-Tests	242	0	6	1	44	8	23	0	5	0	1	1	0	0	5	0	0	0	0	1
LiveTyping-TypeChecking-Tests	84	0	3	0	4	0	10	0	3	0	0	0	0	0	0	0	0	0	0	0
DenotativeObject	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
PackageSnapshot	20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
CodeCoverage	169	0	4	0	0	0	109	0	0	0	0	0	0	0	1	0	0	0	1	3
Morphic-Widgets-Extras-Tests	2	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
Aconocagua-ArithmeticModelTests	40	0	0	0	0	5	8	0	0	0	0	0	30	0	0	0	0	0	0	0
Aconocagua-GraphModelTests	42	0	0	0	10	15	6	0	0	0	0	0	25	0	0	0	0	0	0	0
Aconocagua-MeasureModelTests	649	0	4	0	9	17	18	0	0	0	0	0	449	0	2	0	0	0	0	4
Chalten-TimeLineFilterModelTests	156	0	2	1	2	28	10	0	1	0	0	0	100	0	0	0	0	0	0	0
Chalten-TimeModelTests	397	0	0	4	12	64	44	0	0	0	0	0	294	0	1	0	0	2	0	1
CuisUniversity-Tests	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Regex-Tests-Core	188	1	0	1	0	1	0	0	0	0	0	0	21	0	2	0	0	0	0	0
TestLint-Tests	117	3	3	5	0	0	1	4	1	1	1	1	1	0	2	0	0	0	0	1

Bibliografía

- [BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 11 2004.
- [Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 11 2002.
- [BG98] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.
- [DDP⁺19] Julien Delplanque, Stéphane Ducasse, Guillermo Polito, Andrew Black, and Anne Etien. Rotten green tests. In *ICSE 2019*, pages 500–511, 05 2019.
- [DNS⁺06] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Wuyts Roel, and Andrew Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28:331–, 03 2006.
- [EGK⁺01] Stephen Eick, Todd Graves, Alan Karr, J.S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *Software Engineering, IEEE Transactions on*, 27:1 – 12, 02 2001.
- [GCDD19] Dayne Guerra Calle, Julien Delplanque, and Stéphane Ducasse. Exposing test analysis results with drtests. *International Workshop on Smalltalk Technologies*, 2019.
- [MBLT06] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Mutation analysis testing for model transformations. *ECMDAFA*, pages 376–390, 2006.
- [PJ02] Allen Parrish and Joel Jones. Extreme unit testing: Ordering test cases to maximize early. In Michele Marchesi, Giancarlo Succi, Don Wells, and Laurie Williams, editors, *Extreme Programming Perspectives*, pages 123–140. Addison-Wesley, 08 2002.
- [RDBD06a] Bart Rompaey, Bart Du Bois, and Serge Demeyer. Characterizing the relative significance of a test smell. In *22nd IEEE International Conference on Software Maintenance*, pages 391–400, 09 2006.
- [RDBD06b] Bart Rompaey, Bart Du Bois, and Serge Demeyer. Improving test code reviews with metrics: a pilot study. *Tech. Rep., Lab On Re-Engineering, University Of Antwerp*, 2006.
- [Rei07] Stefan Reichhart. Assessing test quality — testlint. Master’s thesis, University Bern, 2007.
- [SDNB02] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. *Technical Report IAM-02-005*, 11 2002.

- [SIK⁺82] D.C.S. Smith, C. Irby, R. Kimball, B. Verplank, and E. Harlem. Designing the star user interface. *Byte*, 7(4):242–282, 04 1982.
- [ZHM97] Hong Zhu, Patrick Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys - CSUR*, 29(4):366–427, 12 1997.