Métricas de Diseño Orientado a Objetos

Aplicadas en Java.

Alumna: Lic. Gabriela Robiolo
Director: Phd. Ricardo Orozco

INTRODUCCIÓN

Existen ocasiones en las que puede ser necesario obtener en poco tiempo un conocimiento certero de un determinado producto, ya sea porque se necesita seleccionar un producto entre otros, o bien porque se quiere conocer la calidad lograda en un determinado avance reportado sobre un desarrollo en particular. Basándonos en los resultados de los testeos unitarios, de integridad, de sistemas y otros, podemos realizar un juicio sobre el comportamiento esperado de un determinado producto. Cabría preguntarnos:

- **¿el esfuerzo medido en un producto corresponde a un diseño de calidad?**
- **¿Los testeos de performance, sobrecarga, y/o de usuarios aseguran el mantenimiento, y/o la extensibilidad del producto?**
- **¿ Existe documentación suficiente para hacerse una idea cierta y general del producto, o la alternativa para conocer cómo ha sido construido el producto es examinar clase por clase?**
- **¿Tenemos algún estándar que nos permita determinar si un producto validado ha logrado pasar las pruebas de testeo con un diseño del que podríamos afirmar que es de una calidad aceptable?**

Éstas y otras preguntas llevan percibir la necesidad de encontrar un conjunto de métricas que permitan apreciar la calidad de un determinado producto.

Analizando esta premisa, se comprende la dificultad que plantea el concepto de calidad: es un concepto amplio y puede ser tratado desde diferentes puntos de vista. Como describe Davin Garbin "calidad es un concepto complejo y multifacético que puede describirse desde cinco perspectivas: visión trascendental, visión del usuario, visión de manufactura, visión del producto, visión basada en el valor". En este trabajo se centra el tema sobre la visión del producto, esto es, ver el producto hacia adentro, considerando sus características inherentes, vistas desde el diseño[1]. Evaluar la calidad midiendo las características internas de un producto es atractivo porque ofrece una visión de calidad objetiva e independiente del contexto[2].

Para comenzar, se evalúan las posibles bases sobre las que se van a tomar las mediciones y se plantean las siguientes alternativas: el modelo de requerimientos, el modelo de análisis, el modelo de diseño o el código. Se llega a la conclusión de que la forma de asegurarnos métricas objetivas, confiables y robustas es planteando métricas a partir del código. Además, se descartan las otras alternativas debido a su mayor complejidad y a que habitualmente es imposible contar con la documentación necesaria para realizar la medición. La independencia del lenguaje se logra en un planteo básico de las métricas para lenguajes orientados a objetos[3]. Se selecciona el lenguaje Java como lenguaje de los productos a ser medidos.

Por lo tanto, en el planteo inicial del presente trabajo se busca definir métricas que puedan ser útiles para analizar la calidad de diseño de un producto de software planteado con una orientación a objetos y desarrollado en el lenguaje Java.

---

[1] Phenton - Pfleeger identifican para el diseño los siguientes atributos internos: tamaño, reuso, modularidad, acoplamiento, cohesión, funcionalidad. Software Metrics, A Rigorous & Practical Approach, PWS Publishing Company, 1997

[2] Software Quality: The Elusive Target, Barbara Kitchenham, Shari Lawrence Pfleeger, IEEE, 1996

[3] Henderson-Sellers 1996, al describir las propiedades deseables de las medidas de software dice que deben ser objetivas, o sea computables en una forma precisa. Intuitiva, que tiene alguna forma de validación interna, debe proveer un feedback al que está utilizando la métrica. Confiable, robusta, independiente del lenguaje.

El diseño orientado a objetos puede ser analizado desde dos puntos de vista, el diseño de la arquitectura (alto nivel) o el diseño de clases (bajo nivel). Se descarta el punto de vista de la arquitectura, puesto que es muy dificultoso medirlo a partir del código y no existe un consenso de expertos sobre calidad de diseño de la arquitectura de un producto como para tomarlo como punto de referencia. Avanzado el planteo, se comprende que, si se mide a partir del código, es conveniente restringir el concepto de medición de "calidad de diseño" a medición "de la aplicación de buenas prácticas de diseño orientado a objetos". Se encuentra en los expertos cierta convergencia en el planteo de las buenas prácticas de diseño.

En consecuencia, para determinar si un producto tiene un planteo estructural apropiado, se plantean métricas de buenas prácticas de diseño orientado a objetos, a ser medidas sobre las clases.

Bertrand Meyer [1998], en un artículo sobre "El rol de las métricas orientadas a objetos" que ha servido de marco de referencia para el planteo de este trabajo, resalta que "Existe, de hecho una extensiva literatura de métricas de software, inclusive para desarrollos orientados a objetos, pero sorpresivamente pocas publicaciones son usadas en los proyectos actuales". Coincidiendo con la observación, se han planteado métricas que son aceptadas por la comunidad de desarrolladores orientados a objetos, cuya medición puede realizarse con la simple lectura del código. Para lograr la aceptación de los desarrolladores se busca un mapeo directo entre las métricas y los conceptos esenciales de diseño[4], o sea, un conjunto de métricas que en una lectura rápida permita hacer una validación intuitiva de éstas, con el objetivo de no dudar de su consistencia teórica desde un primer momento.

Meyer sugiere en el mismo artículo cinco reglas básicas para la evaluación cuantitativa del software. La primera regla dice "Si recogemos o calculamos números debemos tener una intención específica relacionada con la compresión, el control o el mejoramiento del software y su producción". En el presente trabajo, la intención es determinar la aplicación de las buenas prácticas de diseño en el desarrollo de un producto.

La segunda regla plantea "Las métricas internas y de producto deberían ser diseñadas para reflejar métricas externas tan exactamente como sea posible". Entiende las métricas externas como las que abarcan las propiedades visibles a los usuarios de los productos y las internas como las que abarcan las propiedades visibles sólo al grupo de los desarrolladores. El conjunto de métricas planteadas pretende describir la solidez del producto, o sea entender los fundamentos sobre los cuales se están tomando las mediciones internas y externas.

Se llega a definir un framework básico de métricas[5], apoyadas en los conceptos básicos de orientación a objetos. Se destaca que al plantear las métricas se busca que sean fácilmente comprensibles por una persona no habituada al uso de métricas.

Se quiere, a partir del análisis de los resultados de las mediciones, poder emitir un juicio sobre las características de diseño y contar con un conjunto de alarmas que faciliten y agilicen el análisis del producto.

[4] Cfr. Capítulo I
[5] Cfr. Capítulo I

La tercera regla dice que "cualquier métrica aplicada a un producto o proyecto debe ser justificada por un teoría clara sobre la propiedad que la métrica tiene la intención de ayudar a estimar". Se realiza una síntesis de las buenas prácticas de diseño ciñéndonos a los siguientes autores: Ivar Jacobson, Rebeca Wirfs-Brock, Erich Gamma, Meilir Page-Jones, Barbara Liskov. En los últimos años, con el desarrollo de la UML[6] y la publicación de experiencias de buen diseño probadas en aplicaciones exitosas por medio de los patrones de diseño[7], existe una maduración[8] de conceptos que se entienden esenciales en la orientación a objetos que facilitan el planteo teórico de las métricas. Se estudian las buenas prácticas de diseño planteadas por esta selección de autores. No es que aquí se agoten los autores, sino que se busca una convergencia en ciertos criterios que se estiman esenciales, como ser contratos, herencia, polimorfismo, tipos.

Concientes de la complejidad y tamaño de las aplicaciones actuales, es imprescindible contar con herramientas automatizadas que hagan posible realizar las mediciones sobre dichas aplicaciones. Se plantean una serie de métricas a ser medidas en forma automática a partir del código. Las mediciones se hacen sobre tres productos desarrollados en Java. Se aplican las métricas al lenguaje Java. Para la medición se usa un programa desarrollado en Java.

La cuarta regla dice "que la mayoría de las métricas son comprensibles después de la calibración y comparación con resultados tempranos". Las métricas planteadas han sido probadas en tres casos de estudio, donde se ve la utilidad de contar con información histórica de mediciones para lograr una mayor comprensión de los resultados. Se realizan las mediciones en dos casos de estudio.

En los casos de estudio se llega a percibir que el framework básico es útil para la caracterización de productos desarrollados en Java.

La quinta regla afirma "que los beneficios de un plan de métricas se apoyan en el proceso de medición tanto como en sus resultados". La definición de un plan de métricas[9] está más allá del alcance de este trabajo, pero se comparte la observación del autor.

En el Capítulo I se presenta una síntesis de las buenas prácticas de Diseño Orientado a Objetos como base teórica para el planteo de las métricas . En el Capítulo II se describen las métricas sugeridas. En el Capítulo III se realizan las consideraciones necesarias sobre la aplicación de estas métricas en Java. En el Capítulo IV se muestran los resultados de las mediciones de los dos casos de estudio. En las Conclusiones se plantea el conjunto básico de métricas finalmente planteado, su modo de uso y aplicabilidad. En los Anexos se amplía información de interés al presente trabajo.

[6] Unified Model Language
[7] Gamma 1995
[8] La UML ha logrado la unificación gráfica, conceptual y semántica de importantes autores como Ivar Jacobson, Grady Booch y James Rumbaugh.
[9] El plan de métricas define los objetivos de las mediciones, las métricas asociadas, y la descripción de las métricas a ser coleccionadas en el proyecto para monitorear su progreso.

1. SÍNTESIS DE LAS BUENAS PRÁCTICAS DE DISEÑO ORIENTADO A OBJETOS

" Si recogemos o calculamos números debemos tener una intención específica relacionada con la compresión, el control o el mejoramiento del software y su producción " Bertrand Meyer

A la luz de la experiencia recopilada en los últimos años ya se puede vislumbrar las limitaciones o potencialidad del paradigma orientado a objetos. La idea de este capítulo es recopilar sintéticamente algunas de las experiencias publicadas relativas a los aspectos esenciales del paradigma tales como Polimorfismo, Herencia, Interfase pública, Granularidad de métodos, Agregación, resaltando los aspectos críticos. Se han seleccionado estos aspectos esenciales porque establecen el paradigma y existe un sentir convergente sobre su adecuada aplicación que nos permitirá sacar conclusiones sobre "buenas prácticas" del Diseño Orientado a Objetos.

A continuación se detallan y describen cada una de estas buenas prácticas.

1.1 Granularidad de métodos

La necesidad de situar la funcionalidad para facilitar los cambios y potenciar la reutilización lleva a la definición de métodos que realizan una función bien determinada. Como claro ejemplo de factorización[10] de métodos se puede citar al patrón "Template Method" de Gamma y la sugerencia de "Rings of opertions" de Page-Jones [2000]. Page-Jones plantea la definición de operaciones ubicándolas gráficamente en anillos interiores y exteriores para lograr la encapsulación en el diseño de operaciones. Las operaciones del anillo exterior agrupan operaciones que usan otras operaciones del mismo objeto; las operaciones de anillos interiores son usadas por otras operaciones. La razón para esta distinción es ahorrar código y localizar el conocimiento de la representación de las variables.

1.2 Responsabilidades públicas.

Para realizar el diseño del producto esperado es necesario contar con un claro planteo de requerimientos. Los casos de uso son una forma de plantearlos que se ha visto adecuada. Jacobson afirma:

> **"Los casos de uso especifican el rol de su sistema cuando interactúan con sus usuarios. Los diseñadores transforman los casos de uso en responsabilidades de los objetos. Este punto de vista del diseño de un nivel más alto, que se concentra en las responsabilidades inherentes al uso de su sistema le ayuda a separarse de los detalles de implementación y concentrarse en lo que el mecanismo de software debería ser." [11]**

[10] Este término esta usado en el sentido que le da Gamma. "cuando un comportamiento común entre subclases podría ser factorizado y localizado en una clase común para evitar la duplicación de código"

[11] Palalabras del Prefacio del libro Object Design de R. Wirfs-brock et al. [2002]

Las responsabilidades de un objeto es para el objeto una obligación que implica realizar una tarea o conocer una información. La distribución de las responsabilidades entre los objetos y la definición de los respectivos contratos es un método apropiado para focalizar el diseño de los objetos desde la perspectiva de su comportamiento. Definimos el comportamiento desde el exterior del objeto: qué esperan los otros objetos que un objeto particular haga[12]. Naturalmente surge el planteo de los contratos: si un objeto tiene un contrato con otro objeto debe "pactar" como va a ser esta colaboración. Si más de dos objetos requieren la colaboración de un objeto, el objeto cumple diferentes roles[13].

Esta forma de plantear el diseño inicialmente lleva naturalmente a conservar el principio de ocultamiento de información enunciado por Parnas[14].

Los requerimientos de un sistemas se deben concretar en un conjunto responsabilidades públicas de objetos[15]. Más concretamente un caso de uso lleva a definir nuevas responsabilidades públicas de objetos nuevos o existentes o usar las responsabilidades públicas de los objetos ya existentes.

Las responsabilidades públicas son la interface del objeto. El objeto implementa la interfase, ocultando la forma en que es implementada, publicando solamente el servicio que ofrece el objeto[16].

Además, las responsabilidades públicas definen el Tipo de un objeto. Page-Jones dice que

" El mejor modo de pensar en una clase es como la implementación de un tipo, que es la visión abstracta o externa de una clase ".

Aclara que el tipo de la clase es definido por: el propósito de la clase, el "invariante de clase"[17], los atributos de la clase, las operaciones de la clase, las pre-condiciones y post-condiciones de las clases[18], definiciones y "firmas".

Define:

> **" Si S es un subtipo verdadero de T, entonces S debe ajustarse a la T. En otras palabras, un objeto de tipo S puede ser suministrado en cualquier contexto donde se espera un objeto de tipo T, y sigue siendo válido cuando se ejecuta cualquier operación de entrada[19] del objeto. "**

1.3 Colaboración-Agregación

La agregación es una alternativa para la reutilización. Es más flexible y dinámica que la herencia.

---

[12] Jacobson 1992, pag 233; Booch et al. 1999.
[13] Wirfs-Brock 2002
[14] "On the criteria to be used in decomposing systems into modules", Parnas
[15] Jacobson 1992.
[16] Booch et al. 1999
[17] El invariante de Clase C, una condición que cada objeto de C debe satifacer en cualquier momento cuando el objeto está en el equilibrio.
[18] Precondición, de una operación, es una condición que debe ser cierta para una operación al comenzar la ejecución y debe ejecutarse exitosamente. Postcondición, de una operación, una condición que debe ser cierta para una operación finaliza la ejecución.
[19] El autor se refiere a operaciones de entrada operaciones de consulta donde no se cambia el estado del sistema en su ejecución.

Gamma et al. aconseja Favorecer la composición de objetos más que la herencia de clases. Aclara:

> **" La composición de objetos es definida dinámicamente en tiempo de ejecución por medio de objetos que adquieren referencias a otros objetos. La composición requiere de los objetos respetar la interfase de cada uno, lo cual a su turno requiere el diseño cuidadoso de las interfaces de tal forma que no le impida usar un objeto con muchos otros. Como a los objetos se accede únicamente por sus interfaces, no rompemos la encapsulación. Pero hay un precio. Cualquier objeto puede ser substituido por otro en tiempo de ejecución siempre y cuando el otro sea del mismo tipo. Además, como la implementación será escrita en términos de las interfaces de los objetos, hay muchas menos dependencias de implementación".**

1.4 Polimorfismo

Jacobson (1992) dice al respecto:

> **" El polimorfismo quiere decir que el que envía un estímulo no necesita conocer la clase de la instancia receptora. La instancia receptora puede pertenecer a una clase arbitraria ... el Polimorfismo quiere decir que instancias diferentes pueden ser asociados, y que estas instancias pueden pertenecer a clases diferentes ...**
> **Un estímulo puede ser interpretado de formas diferentes, dependiendo de la clase del receptor. Es, por lo tanto, la instancia que recibe el estímulo la que determina su interpretación, y no la instancia transmisora. A menudo, se dice que el polimorfismo, significa que una operación puede ser implementada en formas diferentes en clases diferentes. Esto es en realidad sólo una consecuencia de lo dicho y no el polimorfismo en sí mismo "**

Menciona el polimorfismo limitado, para evitar errores restringiendo los receptores del estímulo. Normalmente esto se hace por medio de la jerarquía de herencia. Dice: "Esta es una herramienta sumamente sólida para permitirnos desarrollar sistemas flexibles ".

Liskov afirma que:

> **" Siempre que haya tipos relacionados en un programa existe la probabilidad de que haya polimorfismo. Este es ciertamente el caso cuando la relación está indicada por la necesidad del módulo polimórfico. Sin embargo, aún cuando la relación es identificada por adelantado, el polimorfismo es probable. En tal caso el supertipo es a menudo virtual: no tiene ningún objeto propio, porque es simplemente un lugar en la jerarquía para la familia de tipos relacionados. En este caso, cualquier módulo que use el supertipo será polimórfico. Por otra parte, si el supertipo tiene objetos propios, algunos módulos podrían usarlo solamente a él y a ninguno de sus subtipos ".**

Deja bien claro que el uso de clases abstractas en la jerarquía asegura el uso de polimorfismo, si se codifica orientado al supertipo abstracto.

1.5 Herencia

Las expectativas que se han tenido con respecto al mecanismo de herencia han sido más altas de lo que realmente se ha comprobado con la experiencia. Describimos a

continuación una serie de aspectos que resultan inesperados o contradictorios con respecto a la bibliografía optimista del paradigma de Orientación a Objetos.

a) **Uso: Harrison et al.[1999], destaca que existen reportes sobre el bajo uso de herencia.**

b) **Clases ubicadas en los niveles inferiores de la jerarquía:**
   a. **Basili et al. [1996] comprueba experimentalmente que una clase ubicada más profundamente en la jerarquía de clases es más propensa a fallas que una clase que esta ubicada más arriba en la jerarquía.**
   b. **Cartwright and Shepperd [1996], reporta que las clases que tienen el nivel de cambio más alto están en los niveles más bajos de las jerarquías.**

c) **Impacto de los niveles de Herencia en el Mantenimiento[20]:**

   Se busca determinar si es beneficioso el uso de herencia para el mantenimiento, esto es si se ha determinado en la práctica una cantidad de niveles de herencia "óptimo", de tal forma que trasgrediendo ese óptimo se dificulte el mantenimiento del sistema.
   J. Daly et al. en un experimento en [1996], sobre programas similares planteados sin herencia, con tres niveles de herencia o cinco niveles de herencia. Reporta:
   a. **Un sistema con tres niveles de herencia mostró ser más fácil de modificar que un sistema que no usa herencia.**
   b. **Un Sistema con cinco niveles de herencia, costó más modificar que un sistema sin niveles de herencia.**
   c. **Esto lleva a pensar que existe un nivel óptimo de herencia que si se sobrepasa el mantenimiento comienza a ser problemático.**

   Harrison et al. [1999] y Unger et al. [1998], replican el experimento realizado por J. Daly et al. Contradicen sus resultados. Harrison reporta:

   a. **Sistemas sin herencia son más fácil de modificar que los sistemas correspondientes que tienen tres o cinco niveles de herencia.**
   b. **Es más fácil entender un sistema sin herencia que el correspondiente versión con herencia.**
   c. **Grandes sistemas son igualmente difíciles de entender, conteniendo o no herencia.**

   Los experimentos controlados de Unger et al. [1998] tienen la particularidad de hacerlo sobre aplicaciones de un mayor tamaño[21] a los anteriores, y requerir tareas de mantenimiento más complejas. Se dispone de una amplia información de este experimento. Reportan:

   a. **Las tareas de mantenimiento realizadas con programas que usan herencia llevan más tiempo y tienen más errores, que aquellos programas que no usan herencia.**
   b. **Un programa con cinco niveles de herencia llevó más tiempo comprenderlo, pero el tiempo total del mantenimiento fue menor que un programa con tres niveles de herencia, dado que la cantidad de métodos a modificar era menor.**

[20] Mantenimiento: Etapa del ciclo de vida del sistema que comienza una vez que el producto es liberado al cliente.
[21] Tamaño: Medido en cantidad de clases.

c. **El programa con cinco niveles de herencia es el qué más errores evidenció en la modificación.**
d. **Los niveles de herencia no son buenos predictores del esfuerzo en el mantenimiento.**
e. **Se encontró una alta correlación entre tiempo requerido en el mantenimiento de en un programa y cantidad de clases del programa y cantidad de métodos que son relevantes para comprender el problema y poder realizar una modificación. Estos valores podrían usarse para estimar el tiempo de mantenimiento.**
f. **Limitaciones:**
    i. **Se localizó la comprensión, no se tuvieron en cuenta otras tareas propias del mantenimiento.**
    ii. **En las modificaciones incorporadas se trabajó con la mayoría de los métodos, cosa que en la tarea habitual de mantenimiento sólo se llega a modificar unos pocos métodos.**
    iii. **El ejemplo usa herencia para reusar, no para un buen empleo del polimorfismo.**

d) **Testing: Jacobson puntualiza que la herencia si bien implica escribir menos código hace más compleja la labor del testing, puesto que se requiere plantear una mayor cantidad de casos de prueba[22]:**

> **" Debemos testear que podamos heredar la clase y que podamos crear instancias de los descendientes. El caso de prueba debería así crear instancias de los descendientes y luego testear estas instancias. Cuando una clase hereda otra clase, las operaciones heredadas también pueden necesitar ser nuevamente testeadas en el nuevo contexto ".**

Todavía no existen suficientes estudios reportados para sacar conclusiones finales en cuanto al uso de la herencia, pero sí se puede afirmar que la tendencia es usar herencia en forma moderada, donde realmente se justifique la reutilización o se quiera buscar una mayor flexibilidad en el sistema posibilitando/facilitando la aplicación de polimorfismo.

Es recurrente en los autores la conveniencia de plantear estructuras de subtipos (Page-Jones [2002], Liskov, Gamma [1995]). A modo de ejemplo, Jacobson distingue: la herencia de subtipo, llamándola "comportamiento compatible"; la especialización, el descendiente es modificado en una forma no compatible con su padre; o la herencia conceptual, identificándola como clasificación semántica. Establece que:

> **" Según nuestra experiencia, concentrase en el protocolo de un objeto normalmente conduce a estructuras de herencia apropiadas. Esto quiere decir que concentrarse todo el tiempo en subtipos conducirá a jerarquías de herencia adecuadas que son fáciles de mantener y robustas ".**

1.6 Herencia-Polimorfismo

Liskov plantea las "Jerarquía de Tipo", compuestas por subtipos y supertipos. "La idea intuitiva de subtipo es que sus objetos proporcionan todo el comportamiento de los objetos del supertipo, más algo extra." Destaca la importancia que tiene ubicar los

[22] Casos de prueba: dada un función a testear, se define un conjunto de datos de entrada, junto con los valores esperados como salida.

requerimientos en los subtipos del programa en el diseño incremental[23], ya que se limitan los efectos de los errores de diseño y esta ubicación ayuda a organizar racionalmente el diseño, porque cada subtipo describe las decisiones de diseño tomadas en un punto particular. En cuanto a la identificación de un conjunto de subtipos aclara que cuando la relación es definida tempranamente las jerarquías son un buen modo de describirlas y probablemente se use herencia. Pero cuando los tipos relacionados ya están implementados, la herencia puede no ser la mejor forma de organizar un programa.

Page-Jones [2002] afirma:

" En un correcto diseño orientado a objetos el tipo de cada clase debería adecuarse al tipo de su superclase. En otras palabras, la jerarquía de herencia de la clase o subclase debería seguir el principio de conformidad de tipo. La razón es que para explotar el polimorfismo sin esfuerzo, debemos ser capaces de pasar los objetos de una subclase en vez de los objetos de una superclase ".

Para asegurar la compatibilidad con el tipo en la subclase, en primer lugar se necesita que el invariante de la clase sea al menos tan fuerte como el de superclase. El invariante de la subclase puede ser más fuerte que el invariante de la superclase[24]. En segundo lugar es necesario asegurarse que las siguientes restricciones en las operaciones se cumplan:

- **Cualquiera de las operaciones de la superclase tiene una operación correspondiente en la subclase con el mismo nombre y firma.**
- **Cualquier precondición de operación no es más fuerte que la correspondiente operación en la superclase. O sea que, el rango de la precondición de la operación en la subclase puede ser más amplio que la precondición de la operación de la superclase.**
- **Cualquier postcondición de operación es al menos tan fuerte como la correspondiente operación en la superclase. O sea que el rango de la postcondición de la operación en la subclase puede ser más pequeño que el rango de la postcondición de operación en la superclase.**

Se transcribe el resumen de los requerimientos para conformidad de tipo presentado por Page-Jones:

" Los principios de conformidad de tipos exigen que para que una subclase S sea un subtipo verdadero de una clase T, las seis siguientes restricciones se deban dar. (Las primeras dos se aplican a clases completas; las últimas cuatro se aplican a operaciones individuales).
1. El espacio-estado[25] de S debe tener las mismas dimensiones[26] que T. (Pero S puede tener dimensiones adicionales que se extiendan desde el espacio-estado de T).
2. En las dimensiones que S y T comparten, el espacio-estado de S debe ser igual o estar dentro del espacio-estado de T. (Otro modo de decir esto es, la clase invariante de S debe ser igual a o más fuerte que la de T).

---

[23] El autor se refiere a la incorporación de nuevos requerimientos, que definen nuevas versiones del programa.
[24] El invariante de una clase es una condición que todo objeto de la clase debe satisfacer en todos los tiempos mientras el objeto esta en equilibrio.
[25] El espacio-estado de una clase C es el conjunto de todos los estados permitidos de cualquier objeto de la clase.
[26] Las dimensiones de un espacio-estado son equivalente a los atributos definidos en la clase.

Para cada operación de T (T.op) que S sobrescribe y redefine con S.op:

3. S.op debe tener el mismo nombre que T.op.
4. La lista de argumentos de la firma de S.op debe corresponder a la lista de argumentos de la firma de T.op.
5. La precondición de S.op debe ser igual a o más débil que la precondición de T.op. En particular, cada argumento de entrada formal a S.op debe ser el supertipo de (o el mismo tipo que) el correspondiente argumento de entrada formal a T.op.
6. La postcondición de S.op debe ser igual a o más fuerte que la postcondición de T.op. En particular, cada argumento de salida formal debe ser un subtipo de (o el mismo tipo que) el correspondiente argumento de salida formal de T.op.

Jacobson puntualiza que el polimorfismo hace más fácil la incorporación de cambios. El testeo es simplificado en la medida que se use herencia de subtipo. " Si se agrega una clase hija A y se testea a A por sí misma, no se necesita testear clases clientes[27] nuevamente, en tanto y en cuanto se haya usado la herencia para crear subtipos".

Gamma et al. [1995] resalta la importancia de programar para la interface no para la implementación. Con esta expresión quiere decir que para potenciar el uso de polimorfismo y el de herencia las jerarquías deben ser de subtipos; todas las clases de una jerarquía deben derivar de una clase abstracta y el código escribirlo pensando en el tipo no en la instancia concreta.

1.7 Herencia-Sobre-escritura

Li el al. [Li 1998 a] puntualizan claramente el problema de la redifinición completa del método cuando es sobrescrito. Puntualiza que la sobre-escritura (overriding) de un método existe cuando:

- **Un método es heredado**
- **La firma del método es igual a su definición original**
- **La implementación es remplazada, extendida o provista.**

Establece que "Si el problema de la redefinición de descendientes múltiples rompe el esquema de propiedades de herencia en una jerarquía de clases, un método no debe ser completamente redefinido más de dos veces en un rama dada."
Plantea y muestra datos experimentales de la métrica Porcentaje de Métodos Redefinidos por nivel de la jerarquía. La métrica es una ayuda para la determinación de deficiencias en el diseño de las jerarquías.

Jacobson 1992 aclara que "debemos re-testear la operación sobrescrita".

1.8 Aspectos a tener en cuenta en el planteo de las métricas

Destacamos una serie de aspectos que consideramos básicos para el planteo de las métricas, como base para el planteo de las métricas presentado en el próximo capítulo.

1. **La granularidad de los métodos hace a la reutilización y a la fácil comprensión de la clase.**

[27] clases clientes de una clase A son todas las clases que usan a A

2. **Diseñar un objeto es definir sus responsabilidades públicas. Sintetizan los servicios que el objeto brinda. Se puede analizar el diseño de un sistema revisando la distribución de responsabilidades públicas entre los objetos.**
3. **Los colaboradores internos de una clase nos pueden dar una idea de la responsabilidad que ha asumido una clase. Hay que evitar diseñar clases con alta responsabilidad porque dificulta la reutilización.**
4. **El uso de polimorfismo hace más flexible el sistema.**
5. **La herencia disminuye la cantidad de código a escribir, facilita la reutilización, pero dificulta el testing. Puede ser una fuente de posibles errores innecesario si su aplicación no tienen un fundamento en la reutilización o aplicación del polimorfismo.**
6. **El planteo de estructuras de herencia de subtipo potencia el polimorfismo.**
7. **Las clases abstractas root aseguran el uso de polimorfismo, cuando se programa a la interfase de la clase root.**
8. **El polimorfismo de subtipo facilita el testing.**
9. **La sobre-escritura sucesiva de los métodos en los descendientes de la jerarquía quiebra el esquema de herencia. Un método no debe ser completamente sobrescrito más de dos veces consecutivas en un misma rama. La sobre-escritura complica el testing.**

2. DESCRIPCIÓN DE LAS MÉTRICAS. ANTECEDENTES.

" Las métricas internas y de producto deberían ser diseñadas para reflejar métricas externas tan exactamente como sea posible " Bertrand Meyer

2.1 Planteo inicial

El conjunto de métricas planteadas tiene a reflejar aspectos claves de las buenas prácticas de Diseño. Miden aspectos que se consideran críticos o que se deben tener en cuenta en un buen diseño. Responden a la siguiente pregunta:

- **¿cuáles son los aspectos críticos que tengo que tener en cuenta cuando estoy diseñando?, o**
- **si quiero tener una idea rápida de la forma en que se ha planteado el diseño, ¿qué aspectos tendría en cuenta?**

Las métricas que se plantean son pensadas en el diseño y medidas en el código. En este planteo inicial, no están orientadas a un lenguaje determinado sino planteadas con una orientación a objetos.

Henderson-Sellers [1996], afirma que las métricas pueden ser planteadas como alarmas: "Existiría una alarma cuando el valor de una métricas interna específica excediese algún umbral predeterminado." En los capítulos posteriores se muestran valores de referencia o límites, obtenidos por la observación empírica de mediciones en un determinado contexto.

Las alarmas, identifican valores que están fuera del rango considerado como probable u óptimo. Los aspectos apuntados por las alarmas se analizan con más profundidad en la información detallada de la métrica, en comparación con otras métricas, en el código, o la documentación .

Algunas métricas están planteadas para dar un marco de referencia cierto para la evaluación de otras métricas. Por ejemplo "Cantidad de jerarquías diseñadas" se usa como referencia para la evaluación de "Cantidad de mensajes polimórficos enviados".

La potencialidad del planteo realizado esta en el conjunto de las métricas. La caracterización del producto se realizará con la información que aporta el conjunto de métricas. Los valores obtenidos en el cálculo de una métrica explican los valores obtenidos en el cálculo de otras métricas.

Se busca hacer un planteo simple, basado en conceptos orientados a objetos conocidos por los líderes de proyectos o desarrolladores. Las métricas están definidas para que su percepción sea intuitiva, esto es que puedan manejarse magnitudes sin esfuerzo a la hora de comparar productos.

El cálculo de los valores es objetivo, porque esta realizado por una herramienta automatizada que analiza el código. Una persona podría realizar el mismo cálculo, pero sin duda con un gran esfuerzo.

Es posible definir nuevas métricas combinando las métricas actuales, para obtener una mayor información. El planteo esta basado en el desarrollo de métricas básicas.

Se definen varias en torno al concepto de herencia por que es el aspecto del paradigma de OO que plantea mayores dificultades[28] .

Aspecto a medir	Nombre de Métrica
Tamaño	Cantidad de clases desarrolladas
Reutilización	Cantidad de clases externas especializadas
Granularidad de métodos.	Promedio de statements por método en una clase.
Responsabilidades públicas.	Cantidad de métodos de interfase por clase.
Colaboración - Agregación.	Cantidad de colaboradores internos por clase
Reutilización - Agregación	Cantidad de colaboradores externos por clase
Polimorfismo	Cantidad de mensajes polimórficos enviados.
Herencia	Cantidad de jerarquías de clases propias diseñadas Cantidad de jerarquías extendidas de clases externas
Herencia-Especialización	Cantidad de niveles de especialización por jerarquía de clases Cantidad de niveles agregados a jerarquías donde la raíz es externa.
Herencia-Generalización	Cantidad de clases raíz no abstractas.
Herencia-Overriden	Porcentaje de Métodos Reemplazados en una Jerarquía. Porcentaje de Métodos Reemplazados en Jerarquías donde la raíz es externa
Herencia-Polimorfismo	Cantidad de jerarquías que usan herencia de subtipo.

2.2 Descripción de la métricas

2.2.1 Cantidad de clases desarrolladas

Aspecto a medir: Tamaño

Objetivo: definir una medida que permita dimensionar el producto.

Comentario: se usa como marco de referencia de otras métricas. Por ejemplo valorar la cantidad de jerarquías de herencia con respecto al total de clases.

Forma de calculo: se cuentan las clases declaradas dentro del alcance de medición[29].

[28] cfr. Capitulo anterior
[29] Conjunto de clases a ser analizadas por el programa que calcula las métricas.

- ***Clases a considerar***: **Toda clase desarrollada, extendida, usada, que es instanciada en el sistema, clases no instanciadas pero definidas en el sistema.**
- ***Clases a no considerar***: **Clases cuyos padres son externos al sistema, usada para definir una nueva clase, por extensión. Tipos primitivos de datos, como ser Integer, Carácter, etc.**

Antecedente de la métrica: Ha sido planteada por Lorenz-Kidd [1994].

2.2.2 Cantidad de clases externas especializadas

Aspecto a medir: Reutilización

Objetivo: Determinar la cantidad de clases externas al sistema que son reutilizadas por medio del mecanismo de herencia.

Comentario: se identifican las clases externas que el propio sistema está especializando, reutilizando. Es significativo mostrar la cantidad de clases reutilizadas por librería de clases.

Forma de cálculo: se cuentan las clases externas que tienen una clase hija que pertenece al alcance de medición.

- ***Clases a considerar***: **Toda clase externa al alcance de medición que es extendida en el producto.**
- ***Clases a no considerar***: **Tipos primitivos de datos, como ser Integer, Carácter, etc. Clases que pertenecen al alcance de medición y tienen subclases que pertenecen al alcance de medición. Clases que pertenecen al alcance de medición y son padres de clases externas.**

Antecedente de la métrica: Lorenz [1994] plantea "Número de veces que una clase es reusada". Mide el número de referencias a una clase, las veces que una aplicación usa una clase.

2.2.3 Promedio de statements por método de una clase

Aspecto a medir: Granularidad de métodos.

Objetivo: Determinar si existe una cantidad excesiva de statements en los métodos que componen una clase.

Comentario: Se considera que una cantidad excesiva de statements en un método pueden ser la consecuencia de una inadecuada factorización de los métodos.

Forma de calculo: Sumatoria de la cantidad de statements en todos los métodos de la clase, divido el número total de métodos.

- ***Clases a considerar***: **Toda clase desarrollada, perteneciente al sistema a evaluar.**
- ***Clases a no considerar***: **Clases externas al sistema a medir, es decir clases importadas o extendidas.**

Antecedente de la métrica: Ha sido planteada por Lorenz [1994] . El gráfico que ejemplifica la métrica presenta valores obtenido de la medición de un proyecto en Smalltalk. No brinda suficiente información para evaluar los datos que muestra sobre las mediciones realizadas.

2.2.4 Cantidad de métodos de interfase por clase

Aspecto a medir: Responsabilidad pública.

Objetivo: Medir los contratos asumidos por una clase.

Comentario: Esta métrica permite detectar clases que posean un exceso de responsabilidad pública o carencia de la misma. Una visualización rápida por medio de la cantidad de métodos públicos nos permite apreciar la distribución de responsabilidades del sistema y focalizar la atención en la clase que presenta valores llamativos.

Forma de cálculo: Se cuentan los métodos de interfase que tiene la clase.

- ***Métodos a considerar*: Se cuentan los métodos de interfase propios de la clase y métodos de interfase heredados de la clase padre.**
- ***Métodos a no considerar*: Los métodos que no pertenecen a la interfase de la clase.**

- ***Clases a considerar*: Toda clase declarada dentro del alcance de medición.**
- ***Clases a no considerar*: Clases externas al sistema a medir, clases importadas o extendidas.**

Antecedente de la métrica: Ha sido planteada por Lorenz [1994]. No presenta valores experimentales.
Es interesante el comentario de Meyer [1998] al respecto:

> **" ... en el contexto de la orientación a objetos la noción de punto función, una medida de funcionalidad ampliamente aceptada, puede ser substituida por una medida mucho más objetiva: el número de operaciones exportadas de las clases relevantes, que no requiere ninguna decisión humana, ya que pueden ser medidas trivialmente por una herramienta de análisis[30] simple."**

2.2.5 Cantidad de colaboradores por clase

Aspecto a medir: Colaboración - Agregación.

Objetivo: Medir el uso de agregación en el diseño del sistema.

Comentarios: La agregación de una clase se determina por el número de colaboradores habituales que la clase utiliza. La presencia de dichos colaboradores define una

[30] "parsing tool"

asociación de agregación o conocimiento entre la clase dueña y las clases referenciadas[31].

<u>Forma de cálculo</u>: se cuentan los objetos colaboradores declarados estáticamente en la definición de la clase[32] y que pertenecen a tipos de datos distintos a la clase considerada.

- ***Objetos a considerar*: Objetos definidos en la clase a través de una variable de instancia a la cual se le solicita colaboración, por medio del envío de un mensaje en algún método de la clase. Asimismo, se deben contabilizar los objetos declarados en la superclase, si la hubiera, e invocados en la clase corriente. Los objetos cuyas clases están declaradas dentro del alcance de medición (internos) y los objetos externos.**
- ***Objetos a no considerar*: Estos objetos pueden variar dependiendo del lenguaje de programación implementado. Sin embargo, a continuación se detallan algunos objetos que no deberían ser considerados colaboradores, porque están implícitos en la mayoría de las implementaciones. Tipos de datos primitivos, objetos que representan a tipos primitivos de datos y objetos comunes.**
  Ej. Integer, String, Byte, Char, int, char, short, flota, etc.

- ***Clases a considerar*: Toda clase desarrollada, perteneciente al sistema a evaluar.**
- ***Clases a no considerar*: Clases externas al sistema a medir, es decir clases importadas o extendidas.**

<u>Antecedente de la métrica</u>:
Lorenz [1994] plantea "Número de variables de instancia en una clase" donde cuenta la cantidad de variables de instancias públicas, privadas y protegidas. Plantea valores de referencia para "Número medio de variables de instancia por clase".
Brian [1999], cita en un extenso y profundo artículo sobre medidas de acoplamiento varias métricas para atributos de clase (DAC Data Abstraction Coupling, DAC´, IFCAIC, ACAIC, OCAIC, FCAEC, DCAEC, OCAEC), donde el atributo tiene el tipo de otra clase. Todas tienen una validación empírica y la mayoría teórica.

2.2.6 Cantidad de colaboradores externos por clase

<u>Aspecto a medir</u>: Reutilización - Agregación

<u>Objetivo</u>: determinar el uso de colaboradores externos en el sistema. Esta métrica completa el planteo de la medición de reutilización.

<u>Comentarios</u>: la agregación realizada con clases externas al sistema es otra forma posible de reutilizar clases.

<u>Forma de cálculo</u>: se cuentan los objetos colaboradores externos declarados estáticamente en la definición de la clase y que pertenecen a tipos de datos distintos a la clase considerada.

[31] "Design Patterns" pag.22-23, Erich Gamma.
[32] Una colaborador definido en forma *estática* en una clase, significa que existe una variable de instancia del tipo del colaborador.

- ***Objetos a considerar*: Los objetos cuyas clases no están declaradas dentro del alcance de medición (externos). Objetos definidos en la clase a través de una variable de instancia a la cual se le solicita colaboración, por medio del envío de un mensaje en algún método de la clase. Asimismo, se deben contabilizar los objetos declarados en la superclase, si la hubiera, e invocados en la clase corriente.**
- ***Objetos a no considerar*: Objetos internos. Estos objetos pueden variar dependiendo del lenguaje de programación implementado. Sin embargo, a continuación se detallan algunos objetos que no deberían ser considerados colaboradores, porque están implícitos en la mayoría de las implementaciones. Tipos de datos primitivos, objetos que representan a tipos primitivos de datos y objetos comunes.**
  Ej. Integer, String, Byte, Char, int, char, short, flota, etc.

- ***Clases a considerar*: Toda clase desarrollada, perteneciente al sistema a evaluar.**
- ***Clases a no considerar*: Clases externas al sistema a medir, es decir clases importadas o extendidas.**

<u>Antecedente de la métrica</u>:
Lorenz [1994] plantea "Número de variables de instancia en una clase" donde cuenta la cantidad de variables de instancias públicas, privadas y protegidas. Plantea valores de referencia para "Número promedio de variables de instancia por clase" .
Brian [1999], cita en un extenso y profundo artículo sobre medidas de acoplamiento varias métricas para atributos de clase (DAC Data Abstraction Coupling, DAC´, IFCAIC, ACAIC, OCAIC, FCAEC, DCAEC, OCAEC), donde el atributo tiene el tipo de otra clase. Todas tienen una validación empírica y la mayoría teórica.

2.2.7 Cantidad de Mensajes Polimórficos

<u>Aspecto a medir:</u> Polimorfismo

<u>Objetivo</u>: medir el uso de polimorfismo en la invocación de los métodos.

<u>Comentarios</u>: Se considera mensajes enviados a tipos internos y externos al alcance de medición.

<u>Forma de Cálculo:</u> Se cuentan los mensajes polimórficos enviados a tipos internos y externos al alcance de medición.

- ***Mensaje a considerar*: Un mensaje es contabilizado como mensaje polimórfico en el código fuente si se cumplen las siguientes condiciones:**
  - **La firma[33] del mensaje debe estar definido en la clase *raíz* de la jerarquía de clases.**
  - **El mensaje debe ser aplicado sobre un objeto cuyo tipo pertenezca a la jerarquía de clases.**
  - **La jerarquía de clases, deben tener dos o mas subclases.**

[33] signature

- **El tipo correspondiente a la declaración del objeto (que recibe el mensaje) no puede pertenecer, a ninguno de los tipos que están en el último nivel, en la jerarquía de herencia.[34]**

- ***Mensaje a no considerar:*** **si no cumple algunas de las características establecidas anteriormente.**

- ***Tipos a considerar*****: las tipos internos o externos al alcance de medición.**
- ***Tipos a no considerar*****: clases que son hojas de la jerarquía de clases a la que pertenecen, clase padre abstracto que tiene una sola hija concreta.**

<u>Antecedente de la métrica</u>:
Benlarbi y Melo (posterior al [1997]) calculan las medidas del polimorfismo de una forma estática, analizando las interfases de las clases en C++. "Definimos las medidas de dos aspectos principales de comportamiento polimorfo proporcionado por C ++: el polimorfismo basado en decisiones de unión en tiempo de compilación (por ejemplo sobrecargando funciones) y el polimorfismo basado en decisiones atadas durante el tiempo de ejecución (por ejemplo funciones virtuales) ".
Definen la siguiente tablas de métricas.

Metrics	Definition
OVO	Overloading in stand-alone clases
SPA	Static polymorphims in ancestors
SPD	Static polymorphism in descendants
DPA	Dynamic polymorphism in ancestors
DPD	Dynamic polymorphism in descendants

Llegan a la conclusión que la aplicación del polimorfismo puede incrementar la probabilidad de fallas en OO software. Se entiende que este resultado inesperado se debe a que en realidad están midiendo el mecanismo interno de la jerarquía de clases (sobrecarga y sobre-escritura) que usa el lenguaje para lograr un comportamiento polimórfico. No mide en sí mismo el polimorfismo. Están midiendo / analizando el planteo de la interfase de la clase; el uso potencial –no actual- del polimorfismo. Resultado que es esperado, y coincide con otras mediciones, si se entiende que son conclusiones aplicables a el planteo de herencia.
Es frecuente (Lorenz [1994], Briand [1999]) la medición de la invocación de los métodos para medir acoplamiento, no así la consideración de los mensajes polimórficos.

[34] Likov., también Gamma en "Design Patterns pag. 18" aconsejan que el tipo de una variable definido en el momento de la declaración, es conveniente que corresponda a un tipo abstracto y no a un tipo concreto, dado que asegura que se aplicará el polimorfismo en dicha variable en todos los casos.

Mediciones relacionadas al uso de herencia

Se describe un conjunto de métricas que permiten destacar aspectos claves en la aplicación de la herencia. Como se reflejó en el capítulo anterior el mecanismo de herencia tiene ventajas y dificultades. Se realiza una definición cuidadosa de métricas orientadas a definir alarmas sobre aspectos tales como sobre-escritura, niveles de especialización, uso de clases abstractas, uso de herencia de subtipo. Se trató de focalizar las mediciones en los aspectos críticos.

El análisis de los valores en su conjunto lleva a resaltar aspectos que deben ser analizados con más profundidad, para estar en condiciones de dar un juicio sobre el uso de herencia en el sistema estudiado.

2.2.8 Cantidad de jerarquías de clases desarrolladas

***Aspecto a medir*: Herencia**

Objetivo: medir el uso de herencia.

Comentarios: se define esta métrica como valor de referencia, útil para la valoración de los resultados obtenidos en otras métricas. Por ejemplo, valores obtenidos en la medición de cantidad de mensajes polimórficos.

Forma de Cálculo: se cuentan las jerarquías de clases propias implementadas en el sistema.

- ***Jerarquías a considerar:* jerarquías donde la raíz pertenece al alcance de medición**
- ***Jerarquías a no considerar:* jerarquías donde la raíz no pertenece al alcance de medición**

- ***Clases a considerar*: Toda clase, que pertenezca a una jerarquía de clases declarada dentro del alcance de medición.**
- ***Clases a no considerar:* Clases internas al alcance de medición, que heredan de una "superclase", que esté fuera del alcance de medición.**

Antecedente de la métrica: no se encontraron antecedentes. Se estima que la falta de mediciones que plantean marcos de referencia, como es esta métrica, pueda ser la causa de la dificultad que se percibe en la bibliografía para interpretar los valores reportados.

2.2.9 Cantidad de jerarquías extendidas de clases externas

***Aspecto a medir*: Herencia**

Objetivo: medir el uso de herencia.

Comentarios: puede suceder que algunas de las jerarquías que sean principales a la aplicación tengan una raíz externa.

Forma de Cálculo: se cuentan las jerarquías de clases, que tengan una clase padre externa, y que al menos tenga una clase hija que pertenezca al alcance de

medición, que a su vez sea padre de una clase que pertenezca al alcance de medición.

- ***Jerarquías a considerar:*** **jerarquías donde la raíz no pertenece al alcance de medición**
- ***Jerarquías a no considerar:*** **jerarquías donde la raíz pertenece al alcance de medición. Jerarquía donde la raíz no pertenece al alcance de medición y tiene un padre externo, que tiene una hija perteneciente al alcance de medición que no es padre de una clase que pertenece al alcance de medición.**

Antecedente de la métrica: no se encontraron antecedentes.

2.2.10 Cantidad de niveles de especialización por jerarquía de clases

Aspecto a medir: Herencia.

Objetivo: Determinar la especialización alcanzada por la clase "raíz" en cada jerarquía de clases.

Comentario: En el capítulo anterior se citan trabajos que reportan dificultades en los niveles bajos de especialización. Se plantea esta métrica para focalizar la atención en jerarquías complejas.

Forma de cálculo: Primero se determina la rama de subclases con mayor profundidad. La cantidad de niveles de la jerarquía es igual a la cantidad de clases de la rama de mayor profundidad[35] menos una clase.

- ***Jerarquías a considerar:*** **toda jerarquía donde la clase raíz pertenezca al alcance de medición.**
- ***Jerarquías a no considerar:*** **familias de clases externas al sistema, ramas de familias de clases externas, clases que pertenecen a la librería del lenguaje utilizado.**

Antecedente de la métrica: Henderson-Sellers [1996] planteas DIT Depth of inheritance Tree, Lorenz-Kidd [1994], Class hierarchy nesting level, similares a la planteada.

2.2.11 Cantidad de niveles agregados a jerarquías donde la raíz es externa

Aspecto a medir: Herencia.

Objetivo: Determinar la especialización alcanzada de una clase externa en el alcance de medición.

Comentario: En el capítulo anterior se citan trabajos que reportan dificultades en los niveles bajos de especialización. Se plantea esta métrica para apuntar la atención a jerarquías complejas.

[35] Es la que tiene mayor cantidad de niveles.

Forma de cálculo: A partir de la clase externa al alcance de medición, que es padre de al menos una clase que pertenece al alcance de medición, determinar la rama de subclases que pertenece al alcance de medición con mayor profundidad (la cantidad de niveles se cuentan a partir del padre externo). Para esta rama, la cantidad de niveles agregados es igual a la cantidad de clases de la rama de mayor profundidad menos una clase.

- ***Jerarquías a considerar:*** **toda jerarquía donde la clase raíz es externa, donde existe un padre externo, y en el alcance de medición la clase hija es padre de una clase que pertenece al alcance de medición.**
- ***Jerarquías a no considerar:*** **jerarquías donde existe un padre externo, y en el alcance de medición la clase hija no tiene hijos.**

Antecedente de la métrica: no se encontraron.

2.2.12 Cantidad de Clases Raíz no Abstractas

Aspecto a medir: Herencia.

Objetivo: identificar las jerarquías que no tienen una clase raíz abstracta.

Comentario: Declarar la clase raíz como clase abstracta, implica que cada subclase debe proveer una implementación completa de los métodos definidos en la clase raíz. De esta forma se asegura el uso de polimorfismo. Gamma et al. [1995] dice que facilita "programar para una interfase, y no para una implementación". Liskov asegura el uso de polimorfismo, si se programa para el tipo abstracto.

Forma de cálculo: se cuentan las clases raíz que no son abstractas.

- ***Clases a considerar*: Clases que pertenecen al alcance de medición, y son raíz de una jerarquía, que pertenece al alcance de medición.**
- ***Clases que no deben considerarse*: Clases que pertenezcan a la librería del lenguaje de programación. Cualquier clase externa que tenga subclases que pertenecen al alcance de medición.**

Antecedente de la métrica: Lorenz-Kidd [1994] plantea "Número de clases abstractas". Afirma: "el número de clases abstractas es una indicación del uso exitoso de la herencia y el esfuerzo que se ha hecho al buscar conceptos generales en el dominio de problema ". Gama [1995] afirma "Cuando la herencia es usada con cuidadosamente (algunos dirán correctamente), todas las clases derivadas de una clase abstracta compartirán su interfase ".

2.2.13 Porcentaje de Métodos Reemplazados en una Jerarquía

Aspecto a medir: Herencia.

Objetivo: determinar si la cantidad de métodos sobrescritos es excesiva.

Comentarios: el porcentaje de métodos heredados que son remplazados en cada subclase de una jerarquía de clases propia, da una idea del uso o abuso de la sobre-escritura. El problema principal de la redefinición de métodos (overriden) se

centra en aquellos casos donde la firma se mantienen, pero la implementación es remplazada completamente. Como consecuencia, se rehace el comportamiento del método y se descarta completamente el comportamiento heredado de la clase padre. La jerarquía que tiene un porcentaje alto de métodos redefinidos detiene el mecanismo de herencia, que la clase padre aporta en la subclase.

Forma de cálculo: | **PMRJ = (CMR / CMH) * 100.**

CMR = Cantidad de métodos remplazados.
CMH = Cantidad de métodos heredados.

Clases a considerar: **subclases de una jerarquía de herencia definida en el alcance de medición.**
Clases a no considerar: **Clases externas a la jerarquía de herencia. Subclases, que pertenecen a la jerarquía de herencia y que hereden de una clase definida abstracta.**

Métodos a considerar**: Métodos heredados donde la signatura es igual a la original y la implementación es totalmente re-escrita o dejada en blanco.**
Métodos a no considerar**: Métodos heredados que la signatura es igual a la original y la implementación es extendida o realizada.**

Antecedente de la métrica:
Li et al. [1998ª] plantea "Percentage of redefined methods in a class (PRMC), Percentage of redefined methods per level within a hierarchy (PRMH)". Mide los métodos redefinidos: reemplazados, extendidos o realizados. El problema se presenta en la reemplazo sucesivo de métodos.
Lorenz et al. [1994] plantea "Average number of methods overridden per class", como un cociente de "Total number overridden methods" sobre "Total number of classes". La forma en que cuenta los métodos sobrescritos es similar al presenta trabajo. Se diferencian en la forma en que esta planteado el cociente.

2.2.14 Porcentaje de Métodos Reemplazados en Jerarquías donde la raíz es externa

Aspecto a medir: Herencia.

Objetivo: determinar si se esta usando bien el mecanismo de herencia cuando se esta reusando clases.

Comentarios: no tiene sentido reutilizar una clase a la cual se le sobrescriben un alto porcentaje de métodos.

Forma de cálculo: | **PMRJ = (CMR / CMH) * 100.**

CMR = Cantidad de métodos remplazados.
CMH = Cantidad de métodos heredados.

***Clases a considerar*: subclases de una jerarquía de herencia donde la raíz es externo y tiene una clase hija en el alcance de medición que a su vez tiene una hija.**
***Clases a no considerar*: subclases de una jerarquía de herencia donde la raíz es externo y tiene una clase hija en el alcance de medición que no tiene hijos. Subclases, que pertenecen a la jerarquía de herencia de raíz externo y que hereden de una clase definida abstracta.**

***Métodos a considerar*: Métodos heredados donde la signatura es igual a la original y la implementación es totalmente re-escrita o dejada en blanco.**
***Métodos a no considerar*: Métodos heredados que la signatura es igual a la original y la implementación es extendida o realizada.**

Antecedente de la métrica:
Li et al. [1998ª] plantea "Percentage of redefined methods in a class (PRMC), percentage of redefined methods per level within a hierarchy (PRMH)". Mide los métodos redefinidos: reemplazados, extendidos o realizados. El problema se presenta en la reemplazo sucesivo de métodos.
Lorenz et al. [1994] plantea "Average number of methods overridden per class", como un cociente de "Total number overridden methods" sobre "Total number of classes". La forma en que cuenta los métodos sobrescritos es similar al presenta trabajo. Se diferencian en la forma en que esta planteado el cociente.

2.2.15 Cantidad de Jerarquías que Usan Herencia de Subtipo

Aspecto a medir: Herencia

Objetivo: Determinar si las jerarquías de clases diseñadas contemplan la herencia de subtipo.

Comentarios: las subclases deben cumplir el principio de conformidad de tipo enunciado en el capítulo anterior.

Forma de cálculo: Número de jerarquías de clases que componen el sistema y que respetan la herencia de subtipo.

- ***Jerarquías a considerar:* Toda jerarquía, que pertenezca a una jerarquía de clases propias al sistema, partiendo de una raíz propia.**
- ***Jerarquías a no considerar:* Jerarquías propias, que hereden de una "superclase", que esté fuera del alcance del propio sistema.**

Antecedente de la métrica: No se hallaron métricas similares.

3. APLICACIÓN DE LAS MÉTRICAS EN JAVA

En este capítulo se describe la forma en que se implementaron en el lenguaje Java las métricas[36] definidas en el capitulo anterior y un conjunto de métricas que se incorpora como consecuencia de las características del lenguaje.[37]

Conjunto básico de métricas.

Aspecto a medir	Nombre de Métrica
Tamaño	Cantidad de clases desarrolladas Cantidad de interfases desarrolladas
Reutilización - Agregación	Cantidad de colaboradores externos por clase (*)
Reutilización - Herencia	Cantidad de clases externas especializadas Cantidad de interfases externas extendidas Cantidad de clases que implementan Interfases externas
Herencia	Cantidad de jerarquías de clases desarrolladas. Cantidad de jerarquías de interfases desarrolladas. Cantidad de jerarquías extendidas de clases externas. (*)
Herencia-Especialización	Cantidad de niveles de especialización por jerarquía de clases. Cantidad de niveles de especialización por jerarquía de interfases. Cantidad de niveles agregados a jerarquías donde la raíz es externa. (*)
Herencia-Generalización	Cantidad de clases raíz no abstractas. Cantidad de clases raíz no abstractas que implementan Interfases
Herencia-Sobre-escritura	Porcentaje de métodos reemplazados en una Jerarquía. Porcentaje de Métodos Reemplazados en Jerarquías donde la raíz es externa. (*)
Polimorfismo	Cantidad de mensajes polimórficos enviados.
Granularidad de métodos.	Promedio de statements por método en una clase.
Responsabilidades públicas.	Cantidad de métodos de interfase por clase.
Colaboración - Agregación.	Cantidad de colaboradores internos por clase

Tamaño

[36] (*) No se describen las métricas Cantidad de colaboradores externos por clase, Cantidad de niveles agregados a jerarquías donde la raíz es externa, Porcentaje de Métodos Reemplazados en Jerarquías donde la raíz es externa, porque no han sido implementadas en la herramienta utilizada. Esta previsto implementarlas en la próxima versión.
La métrica Cantidad de jerarquías que usan herencia de subtipo, no se tiene en cuenta dadas las características del lenguaje. Todas las jerarquías definidas en java tienen un chequeo de tipo. El compilador de Java, al sobrescribir un método no permite cambiar el modificador de un método declarado públic, ni variar los parámetros de la firma. Si varío los parámetros de la firma no estoy sobrescribiendo (overrriding) sino que se define un nuevo método (overloading). En el caso del retorno no me permite variarlo sin variar los parámetros.

[37] Se encontró un artículo que plantea métricas basadas en Chidamber-Keremer en Java. Se lo descartó porque las conclusiones presentan las dificultades que este trabajo trata de solucionar. (J. Kaczmarek y M. Kucharski. Application of object-oriented metrics for java programs).

3.1 Cantidad de clases desarrolladas

Se cuentan las clases declaradas dentro del alcance de medición.

- ***Clases consideradas*: todas las clases independiente de los modificadores de acceso, las clases abstractas, clases raíz, subclases, clases inner[38].**
- ***Clases no consideradas*: las clases padres externas al alcance de medición, de las clases que están dentro del alcance del sistema.**

3.2 Cantidad de interfases desarrolladas

Se cuentan las interfases declaradas dentro del alcance de medición.

- ***Interfases consideradas*: todas las interfases definidas dentro del alcance.**
- ***Interfases no consideradas*: las interfases padres, externas al alcance de medición, de las interfases que están dentro del alcance del sistema.**

Reutilización – Herencia

3.3 Cantidad de clases externas especializadas

Se cuentan las clases externas al alcance de medición, que son extendidas.

- ***Clases a considerar*: clases externas que son extendidas, declaradas en las clases dentro del alcance de medición con la palabra reservada extends.**
- ***Clases a no considerar*: clases que pertenecen al alcance de medición y tienen subclases que pertenecen al alcance de medición. Clases que pertenecen al alcance de medición y son padres de clases externas.**

3.4 Cantidad de interfases externas extendidas

Se cuentan las interfases dentro del alcance de medición que extienden de interfases que no están dentro del alcance de medición.

- ***Interfases a considerar*: interfases externas que son extendidas, declaradas en las clases dentro del alcance de medición con la palabra reservada extends.**
- ***Interfases a no considerar*: interfases que pertenecen al alcance de medición y tienen interfases con hijas que pertenecen al alcance de medición. interfases que pertenecen al alcance de medición y son padres de interfases externas.**

3.5 Cantidad de clases que implementan interfases externas

Se cuentan las clases dentro del alcance de medición que implementan interfases que no pertenecen al alcance de medición.

- ***Clases a considerar*: las clases dentro del alcance, las clases inner; no se tiene en cuenta los modificadores de acceso[39].**
- ***Clases a no considerar*: clases fuera del alcance de medición.**

Herencia[40]

[38] Clases definidas dentro de otra clase.
[39] Public, Protected, Private.

3.6 Cantidad de jerarquías de clases desarrolladas

Se cuentan las clases raíz que tienen una jerarquía de por lo menos 1 nivel. Se usa la palabra reservada *extends para* identificar las jerarquías.

- ***Jerarquías a considerar*: aquellas donde la clase raíz pertenece al alcance de medición y tiene dentro del alcance de medición subclases que definen al menos un nivel.**
- ***Jerarquías a no considerar*: de interfases, las jerarquías que pueden existir con una clase inner.**

3.7 Cantidad de jerarquías de interfases desarrollada

Se cuentan las interfases raíz que tienen una jerarquía de por lo menos 1 nivel. La extensión se mide a partir de la palabra reservada extends.

- ***Jerarquías a considerar*: aquellas donde la interfase raíz pertenece al alcance de medición y tiene dentro del alcance de medición sub-interfases que definen al menos un nivel.**
- ***Jerarquías a no considerar*: de clases.**

Herencia-Especialización

3.8 Cantidad de niveles de especialización por jerarquía de clases

Se cuentan los niveles de las jerarquías diseñadas.

- ***Niveles a considerar*: se cuentan los niveles a partir de raíz.**
- ***Niveles a no considerar*: no se cuenta la extensión de Object.**

3.9 Cantidad de niveles de especialización por jerarquía de interfase

Se cuentan los niveles de las jerarquías de interfases, considerando que es igual a la cantidad de interfases, de la rama más profunda de la jerarquía, menos una interfase.

- ***Niveles a considerar*: se cuentan los niveles a partir de la raíz, considerando que la raíz de la jerarquía debe pertenecer al alcance de medición.**
- ***Niveles a no considerar*: ---**

Herencia-Generalización

3.10 Cantidad de clases raíz no abstractas

Se cuentan las clases raíz de las jerarquías diseñadas declaradas abstract.

---

[40] Cantidad de Jerarquías que Usan Herencia de Subtipo. Todas las jerarquías definidas en java usan herencia de subtipo, no tiene sentido implementar esta métrica en Java. El compilador de Java, al sobrescribir un método no permite cambiar el modificador de un método declarado públic, ni variar los parámetros de la firma. Si varío los parámetros de la firma no estoy sobrescribiendo (overrriding) sino que se define un nuevo método (overloading). En el caso del retorno no me permite variarlo sin variar los parámetros.

- ***Clases raíz a considerar*: todas las clases raíz declaradas como abstracta independiente del modificador de acceso que disponga.**
- ***Clases raíz a no considerar*: las clases raíz no abstractas.**

3.11 Cantidad de clases raíz no abstractas que implementan interfases

Esta métrica complementa la métrica de Cantidad de clases raíz no abstracta. Cuenta las clases raíz de las jerarquías diseñadas no declaradas abstrac, que implementan alguna interfase.

- ***Clases raíz a considerar*: Todas las clases no abstractas independiente del modificador de acceso de la clase, que implementan (implements) alguna interfase.**
- **Clases raíz a no considerar: las clases raíz que no implementan interfases, las clases que son raíces abstractas de una jerarquía.**

Herencia-Sobre-escritura

3.12 Porcentaje de métodos remplazados en una Jerarquía

Se cuentan los métodos remplazados[41], esto es métodos que tienen la misma firma que la clase padre y su comportamiento es remplazado y no extendido ni implementado. El promedio se calcula como el cociente de los métodos remplazados sobre los heredados[42].

- ***Métodos a considerar*: métodos sobrescritos que no usen la expresión super.nombreDelMetodoSobrescrito[43].**
- ***Métodos a no considerar*: métodos sobrescritos que usen la expresión super.nombreDelMetodoSobrescrito, métodos realizados[44]. Los métodos de las clases inner definidas dentro de una clase y que extienden algún comportamiento de otra clase. Los constructores de clase, los métodos native[45].**

Polimorfismo

3.13 Cantidad de mensajes polimórficos enviados[46]

Número de mensajes polimórficos enviados a tipos internos y externos al alcance de medición.

- ***Mensaje a considerar*: Un mensaje es contabilizado como mensaje polimórfico en el código fuente si se cumplen las siguientes condiciones:**
  a) enviado a una variable cuyo tipo lo define una clase
  - **La firma del mensaje debe estar definido en la clase raíz de la jerarquía de clases.**

[41] No se usa el término sobrescrito, porque hay diferentes formas de sobrescribir un método. En esta métrica la idea es determinar los métodos que no extienden el comportamiento del padre.
[42] Un método remplazado es un método heredado.
[43] Los métodos sobrescritos no figuran en el API como heredados, se los muestra en el "Method Summary".
[44] Método que implementa un método padre abstracto.
[45] La palabra **native** informa al compilador Java que la implementación del método esta realizada en otro lenguaje.
[46] No se ha implementado el cálculo de mensajes enviados a tipos externos. Esta previsto hacerlo en la nueva versión de la herramienta.

- **El mensaje debe ser aplicado sobre un objeto cuyo tipo pertenezca a la jerarquía de clases.**
- **La jerarquía de clases, deben tener dos o más subclases.**
- **El tipo correspondiente a la declaración del objeto (que recibe el mensaje) no puede pertenecer, a ninguno de los tipos que están en el último nivel, en la jerarquía de herencia.[47]**

b) enviado a una variable cuyo tipo lo define una interfase:

La firma del mensaje debe estar definida en la interfase

- ***Mensaje a no considerar:*** **si no cumple algunas de las características establecidas anteriormente.**

- ***Tipos a considerar*: clases e interfases que pertenecen al alcance de medición.**
- ***Tipos a no considerar*: clases que son hojas de la jerarquía de clases a la que pertenecen, clase padre abstracto que tiene una sola hija concreta, clases e interfases que no pertenecen al alcance de medición.**

Granularidad de métodos

3.14 Promedio de statements por método en una clase

Se calcula el promedio de los statements[48] por método de una clase.

- ***Métodos contados*: métodos de la clase, los métodos sobrescritos, métodos constructores. Se cuentan todos los métodos independientemente de los modificadores de acceso. Los que pertenecen a una clase inner definida en el alcance de un método. Los métodos native.**
- ***Métodos no contados*: los heredados.**

- ***Statements contados*: todos los statements de los métodos contados.**
- ***Statements no contados*: los statements definidos en bloques estáticos.**

Responsabilidades públicas

3.15 Cantidad de métodos de interfase por clase

Se cuentan los métodos declarados con la palabra reservada public de cada clase dentro del alcance de medición.

- ***Métodos a considerar*: propios y heredados, los constructores, los métodos abstractos[49].**
- ***Métodos a no considerar*: los métodos públicos de las clases inner –estos métodos se cuentan para su propia clase-, los métodos declarados con acceso default. Los métodos sobrescritos son contados como heredados.**

3.16 Cantidad de métodos por interfase

[47] Además, Likov y Gamma, "Design Patterns pag. 18" aconsejan que el tipo de una variable definido en el momento de la declaración, es conveniente que corresponda a un tipo abstracto y no a un tipo concreto, dado que asegura que se aplicará el polimorfismo en dicha variable en todos los casos.

[48] http://java.sun.com/docs/books/jls/second_edition/html/statements.doc.html#101241

[49] La versión actual de la herramienta usada no cuenta los métodos abstractos. Esta corrección será introducida en la versión posterior.

Cuenta los métodos de cada interfase definida dentro del alcance de medición.

- ***Métodos a considerar*: propios y heredados.**
- ***Métodos a no considerar*: ----**

Colaboración - Agregación

3.17 Cantidad de colaboradores por clase

Se cuentan los tipos de las variables de instancia y de clase.

- ***Variables a considerar*: las variables deben ser definidas como variables dentro del alcance de la clase o super clase que se evalúa, y ser invocadas o utilizadas en algún método de la clase que se evalúa.**
- ***Variables a no considerar*: las variables con tipos primitivos de datos y objetos comunes: "boolean", "byte", "short", "char", "int", "float", "long", "double", "Boolean", "Byte", "Short", "Carácter", "Interger", "String" , "Void", "Long", "Object", "StringBuffer". Variables declaradas dentro de un método. Las variables de clase invocadas en algún método de la clase solamente como argumento sin aplicarle ningún método.**

- **Tipos a considerar: clases e interfaces, tanto internas como externas al alcance de medición.**
- **Tipos a no considerar: Los tipos repetidos. Un tipo se contabiliza una sola vez.**

4 CASOS DE ESTUDIO

"Hay de hecho una literatura extensa sobre las métricas de software, incluyendo en la de desarrollo orientado a objetos, pero sorprendentemente pocas publicaciones son usadas en proyectos reales " Meyer

Se desarrolla un sistema para la medición automática de los productos[50]. En primera instancia, se hacen mediciones de las librerías de Java, y Javax[51]. Los resultados obtenidos sirven de marco de referencia para la definición de los intervalos de valores esperados, valores límites y valores probables.

Se plantean dos casos de estudios, sobre productos de una funcionalidad similar para facilitar el análisis comparativo.

Se seleccionaron tres productos del dominio de aplicaciones de interfase gráfica que cumplan con las funciones básicas de: dibujar, mover, cortar, pegar y soportar zoom. Los productos son:

- **Piccolo[52], y dos versiones anteriores del mismo producto Jazz_05 y Jazz_13.**
- **Jgraf[53]**
- **GEf[54]**

Para la medición de los productos se utilizan las herramientas descriptas en el Anexo correspondiente. Para facilitar el análisis de los valores se incorporan medidas estadísticas.

Se describe a continuación el análisis de las métricas de los dos casos de estudio. El primer caso de estudio compara tres versiones de un mismo producto. El segundo compara tres productos con funcionalidades similares.

Para analizar los valores obtenidos en las mediciones, para cada uno de los aspectos de la orientación a objetos considerados en el primer capítulo, se distinguen dos reflexiones:

- **la información que dan las métricas**
- **la información obtenida directamente del producto por la observación de las clases.**

---

[50] OOMTool, desarollado por Fernando Lafalle, alumno de la Facultad de Ingeniería de la Universidad Austral, en el contexto de un trabajo de grado. Actualmente se esta trabajando en las correcciones y en el desarrollo de nueva versión.

[51] Se midieron todos los paquetes de java, javax.sql, javax.swing, javax sin sql y swing, de Java 2 Platform, Standard Edition, version 1.4.1.

[52] Piccolo y Jazz han sido desarrollados por la Universidad de Maryland. El proyecto es liderado por Ben Bederson. Jesse Grosjean es el principal implementador de Piccolo, y actualmente está manteniendo Jazz & Piccolo. Aaron Clamage mantiene Piccolo.NET.

[53] Proyecto de SourgeForge.net

[54] Proyecto de Tigris.org, Open Source Software Engineering

En los anexos se describen cada uno de los productos.

4.1 Jazz05 – Jazz13 – Piccolo

Se muestran en forma comparativa los valores de las métricas obtenidas de las versiones anteriores a Piccolo: Jazz05, Jazz13 (Jazz05 es anterior a Jazz13) y los valores de las métricas correspondientes a Piccolo.

4.1.1 Tamaño

Cantidad de Clases desarrolladas	**62**	**129**	**30**
Cantidad de Interfases desarrolladas	**11**	**25**	**3**

En la tabla anterior, se analiza el tamaño. Se evidencia una evolución de versiones. Llama la atención el salto entre Jazz05 y Piccolo.

- ***Se espera un cambio significativo en el planteo de Piccolo.***
- **Se comprueba. El planteo básico de Piccolo se basa en lograr una versión fácil de usar. Se simplifica el diseño, logrando mantener las características del producto.**

4.1.2 Reutilización

Reutilización	**Jazz05**	**Jazz13**	**Piccolo**
Cantidad de Clases Externas especializadas	**14**[55]	**24**[56]	**6**[57]
Cantidad de Interfases Externas extendidas	**3**[58]	**7**[59]	**1**[60]
Cantidad de Clases que implementan Interfases Externas	**6**[61]	**72**[62]	**4**[63]
Porcentaje de Reutilización sobre desarrollado	**31,50**	**66,88**[64]	**33,33**

Se analizan las métricas de reutilización. Se destaca la métrica "Porcentaje de Reutilización sobre desarrollado"[65], porque da una idea de la reutilización a nivel de producto. La segunda versión duplica los porcentajes de reutilización.

- ***Considerando que la reutilización es algo bueno, llama la atención el valor bajo de la última versión***
- **La gran mayoría de las clases incorporadas en Jazz_13 (alrededor de 60), extienden de una clase de java o implementa una interfase. Esta proliferación de clases esta dada con la intención de dar al programador más cosas implementadas y favorecer una mayor flexibilidad.**

---

[55] Principalmente se extienden clases en la librería de util (java.awt.geom.Area, java.applet.Applet, javax.swing.Jframe, javax.swing.Jwindow, etc.), java.awt.AWTEvent
[56] Extiende de clases de java.
[57] Extiende de java, javax
[58] todas extienden de EventListener
[59] casi en su totalidad extienden de java.util.EventListener
[60] EventListener
[61] Printable, Cloneable, PathIterator, ImageObserver
[62] la mayoría implementa Serializable y Clonable (62)
[63] Serializable, Clonable, InputSource, Printeable.
[64] Este valor es más alto porque no se está contando cómo usa clase externas. La librería Component tiene la mayoría de las clases que tienen wrapper clases de java y javax. Conté 10 más. El porcentaje sería un 73%.
[65] Cociente de Reutilizadas sobre Desarrolladas.

Piccolo simplifica el diseño del producto, disminuye lo más posible la cantidad de clases, mantiene la reutilización inicial.
La funcionalidad que agrega en Jazz_13, fácilmente se puede desarrollar, porque las clases incorporadas en jazz_13 son extensiones de clases de java. Además, cómo se describe en el Anexo, es muy poco lo que no esta considerado en Piccolo, con respecto a las versiones anteriores.

Se detallan en la Tabla las librerías que importan los productos. En el caso de Píccolo, la herramienta no da información porque Piccolo no importa librerías. El detalle de las librerías ha sido incorporado posteriormente.

Jazz05	jazz13	Piccolo
java.io	java.io	no importa paquetes.
java.text	javax.swing.text	
java.awt.print	com.sun.image.codec.jpeg	java.io
java.awt.font	javax.swing.tree	java.text
java.awt.geom.	java.awt.print	
java.lang	java.awt.font	
java.util	java.awt.geom	java.awt.geom.
java.awt	javax.swing.plaf.basic	
javax.swing.event	java.lang	java.util
java.lang.reflect	java.applet	java.awt
java.net	java.util	Java.lang
java.awt.image	java.awt	Java.awt.print
java.awt.event	javax.swing.event	
javax.swing	javax.swing.border	
	java.lang.reflect	
	java.net	javax.swing
	java.beans	
	java.awt.image	
	java.awt.event	
	Javax.swing.plaf	
	Javax.swing	

- *La versión de Jazz13 hace un mayor uso de las librerías de swing.*
- **Se comprueba. Una de las modificaciones más significativas para el producto es la incorporación del Zcanvas, ZviewPort, ZscrollBox, ZcomboBox, que todas extienden de clases de la librería Swing.**

4.1.3 Herencia

Herencia	Jazz05	Jazz13	Piccolo
Cantidad de Jerarquías de Clases desarrolladas	4	6	3
Rango de niveles de Especialización por Jerarquía de Clases desarrolladas	[1 - 3]	[1 - 5]	[1 - 2]
Moda de niveles de Especialización por Jerarquía de Clases desarrolladas	1	1	-
Cantidad de Clases Raíz no Abstractas	3	3	3
Cantidad de Clases Raíz no Abstractas que implementan Interfases	3	2	2

- ***Porcentaje de cantidad de jerarquías y cantidad de clases: un 10 % es un valor que indica un diseño caracterizado por el uso de jerarquías.***

6,45	4,65	10

Se describen los porcentajes de las tres versiones.

- **Se cumple en Piccolo, son 3 jerarquías simples, que definen el corazón del Framework.**

edu.umd.cs.piccolo.event.PBasicInputEventHandler
edu.umd.cs.piccolo.activities.PActivity
edu.umd.cs.piccolo.PNode

En el caso de Jazz_05 tiene un porcentaje menor y jerarquías más complejas.

Los niveles sirven de alarmas.

- ***En el caso de Jazz13 llama la atención los 5 niveles, muchos niveles para 129 clases.***

Rango de niveles de Especialización por Jerarquía de Clases desarrolladas	[1 - 3]	[1 - 5]	[1 - 2]

Considerando que:

- **java.awt, 259 clases, 4 niveles**
- **javax.swing, 482 clases, 6 niveles**

- **Jazz_13 se torna una versión muy compleja, y la jerarquía que tenía 5 niveles pasa a tener 1 nivel en Piccolo**

- ***Se analizan las raíces de las jerarquías. Se mira el detalle de la métrica. En sí misma no da datos muy precisos. Se buscan aquellas jerarquías que no tienen la raíz abstracta y se analiza con detalle el planteo de las clases.***

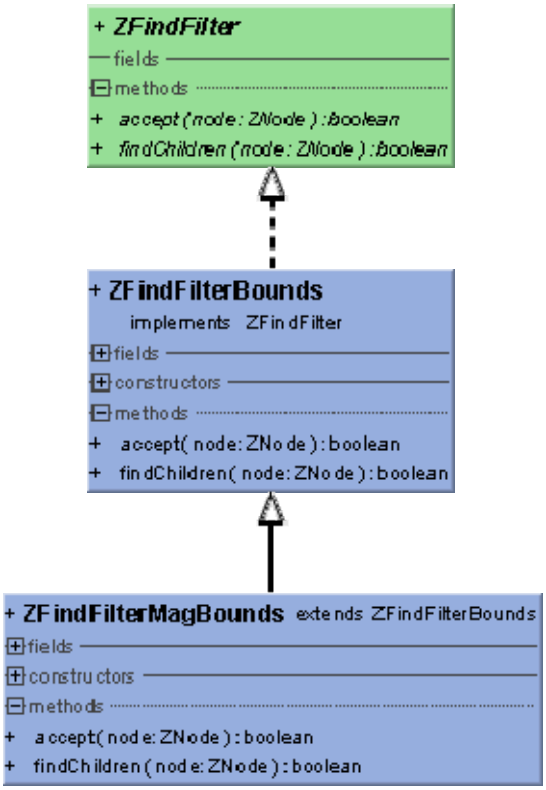
Cantidad de Clases Raíz no Abstractas	3	3	3
Cantidad de Clases Raíz no Abstractas que implementan Interfases	3	2	2

- **Conclusiones del análisis de las clases que son raíz de una jerarquía y no son abstractas:**
  - **En Jazz_05 y Jazz_13 es necesario reestructurar ZfindFilterBounds.**
  - **En Jazz_13 se encontró una clase que la documentación dice que es abstracta; se observa en el código que se comporta como abstracta (no es instanciada), pero no esta declarada como tal.**
  - **En general, en las diferentes versiones, casi no se usan clases abstractas, por lo tanto casi no hay métodos abstractos. Usa métodos vacíos lo que hace que se tenga que descubrir en la documentación la intención del diseñador.**

Se describe a continuación el resultado del análisis hecho sobre las jerarquías que tienen una clase raíz no abstracta.
a) Jazz_05. Las clases raíz que no son abstractas son:

edu.umd.cs.jazz.util.ZfindFilterBounds

En el caso de ZfindFilterBounds, posee el siguiente diagrama de clases:



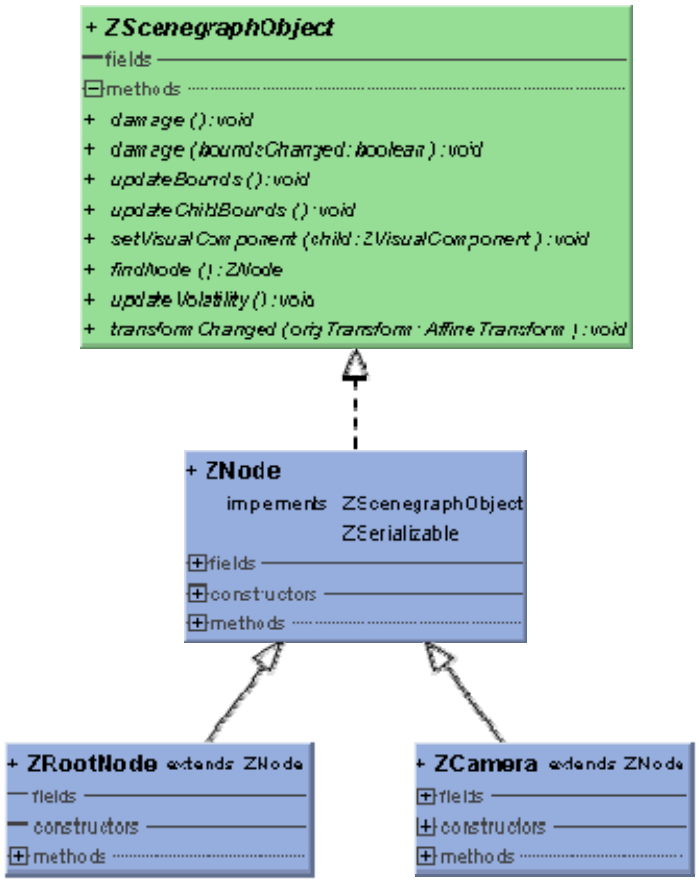
ZFindFilterBounds implementa una interfase llamada ZFindFilter y tiene una clase hija que se llama ZFindFilterMagBounds.
La interfase posee dos métodos:
- **public boolean accept(ZNode node);**
- **public boolean findChildren(ZNode node);**

La clase padre e hija implementan de distinta manera dichos métodos. No poseen, además, otras características visibles que muestren la diferencia entre las clases (Distinta funcionalidad, distintos colaboradores, etc).
Si bien, la estructura jerárquica parece mostrar un uso de polimorfismo, no se entiende el uso de una interfase. Visto de otra manera, la interfase permite que las

clases que la implementen hagan uso de sus métodos a su manera. Sería mejor entonces que las dos clases implementen dicha interfase o se haga uso de una clase abstracta padre.
Por otro lado los diseñadores pueden haber usado esta estructura para su reutilización. Aunque en este caso, tampoco se justificaría.

edu.umd.cs.jazz.scenegraph.Znode

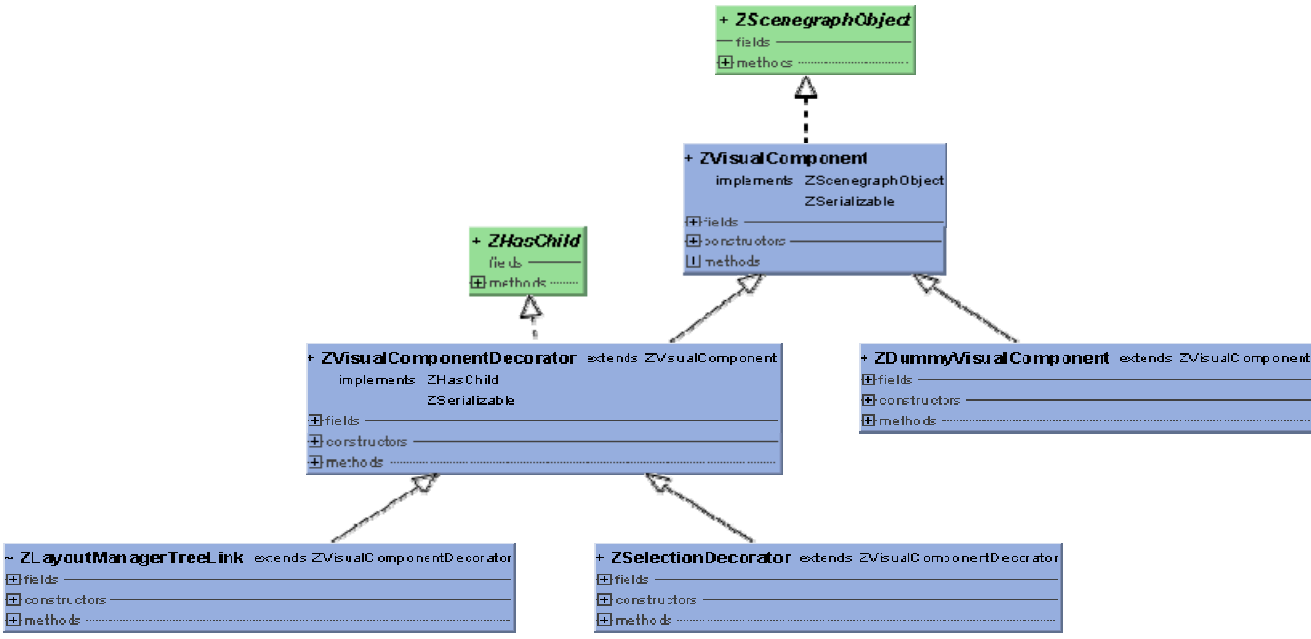
En el caso de ZNode posee el siguiente diagrama de clases:



ZNode implementa una interfase llamada ZscenegraphObject y posee dos clases hijas llamadas ZRaízNode y ZCamera. En este caso el uso de una clase padre no abstracta es razonable ya que hace uso de una interfase que se comporta como tal.

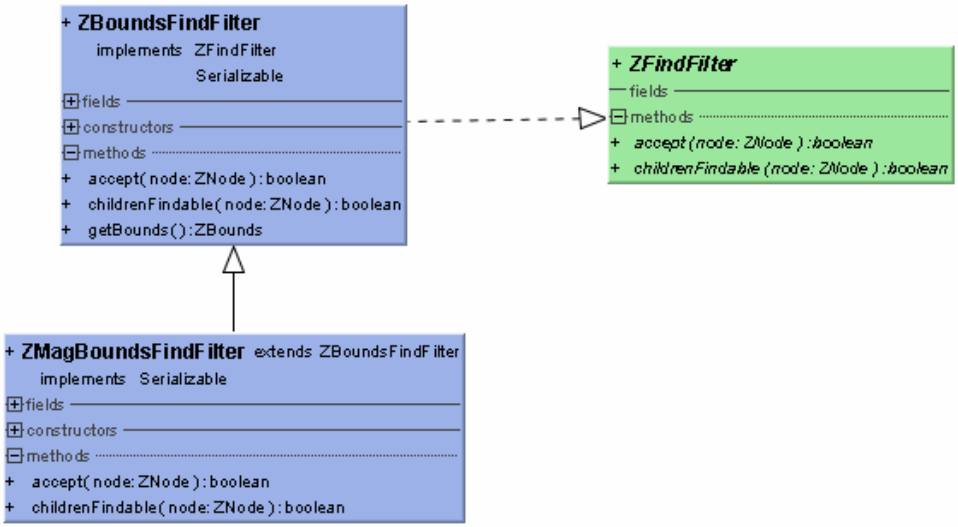
edu.umd.cs.jazz.scenegraph.ZvisualComponent

En el caso de ZvisualComponent posee el siguiente diagrama de clases:



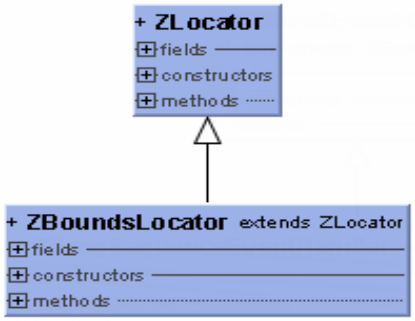
ZVisualComponent implementa una interfase llamada ZscenegraphObject y posee las clases hijas mostradas en la figura. Como en el caso de ZNode, el uso de una interfase reemplaza el uso de una clase abstracta padre.

b) Jazz_13. Clases no abstractas
edu.umd.cs.jazz.util.ZboundsFindFilter



ZBoundsFindFilter tiene una sola clase hija ZMagBoundsFindFilter. Los únicos dos métodos que posee ZMagBoundsFindFilter son métodos sobrescritos, que a su vez deben ser implementados en ZBoundsFindFilter por extender de la interfase ZfindFilter. El único método que hereda ZMagBoundsFindFilter es getBounds(). Si la idea era usar polimorfismo se podría haber hecho que ambas clases (ZboundsFindFilter y ZMagBoundsFindFilter) implementaran dicha interfase, teniendo cada una, por lo tanto, su método getBounds(). También se podría haber hecho una clase padre abstracta que implemente el getBounds() y de la que hereden las funcionalidades de ambas clases.
Sin embargo, creo que se ha mantenido así para que se entienda conceptualmente las funcionalidades de ambas clases, ya que como su nombre lo indica ZmagBoundsFindFilter parecería que fuera un subconjunto de ZboundsFindFilter.

edu.umd.cs.jazz.Zlocator
Zlocator se encarga de ubicar un punto dentro de un ZsceneGraphObject. Tiene una sola clase hija ZboundsLocator, que se encarga de buscar un punto dentro del bounds del ZsceneGraphObject. En este caso se sobrescriben sólo dos métodos de los cinco que posee el padre y agrega bastante funcionalidad.

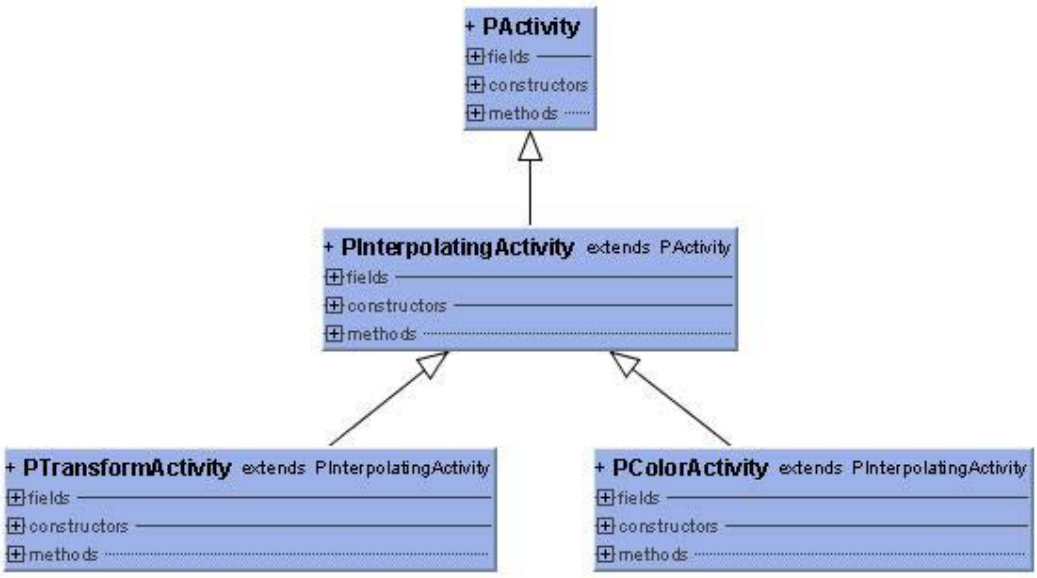


edu.umd.cs.jazz.event.ZFilteredEventHandler

ZFilteredEventHandler es raíz de una jerarquía más grande. El comentario de la clase explica que la misma es abstracta sin embargo no está declarada como tal. En cuanto a los usos de esta clase solo aparece cuando es extendida por las hijas, es decir, no es instanciada en ningún momento.

c- Piccolo

edu.umd.cs.piccolo.activities.Pactivity



Las Actividades se usan para lograr animaciones y otros comportamientos en la interfase.
PInterpolatingActivity permite interpolar dos estados (origen y destino) mientras dura la actividad
PColorActivity premite interpolar dos colores durante la animación.
PTransformActivity interpola dos transformaciones durante la actividad.
PActivity no es abstracta porque necesita ser instanciada. A su vez PInterpolatingActivity sobrescribe algunos métodos de la clase padre y se agregan nuevos para poder responder a las necesidades propias de su clase. Se considera que las clases presentan comportamientos distintos por lo cual cada una es una clase distinta. Además, se considera la intención del diseñador que puede haber creado esta jerarquía para luego agregar nuevas clases sin tener que modificar en forma excesiva el diseño.

- ***Las métricas relativas a Herencia están incompletas. Sólo consideran jerarquías donde la raíz es propia.***
- **No se diría que esta incompleta, en sí misma da información. Las jerarquías con raíz externa aportan también información.**

- ***La Moda de niveles de Especialización por Jerarquía de Clases desarrolladas no aporta información.***
- **Se comprueba.**

1- Rango de Porcentaje de Métodos Reemplazados en Jerarquías (PMRJ)

[6.06 - 33,33]	**[0,66 - [5 - 40] 66,66]**

Se toman valores límites como alarmas.

- **Se consideran alarmas valores mayores al 30%. Jazz_13 y Piccolo pueden tener métodos incorrectamente sobrescritos.**
- **Jazz_13, sobrescribe 2 métodos de un total de 5[66].**
  Piccolo [67], tiene una clase que sobrescribe 12 métodos de un total de 18 métodos heredados. Además se detecta que la jerarquía de PbasicInputEventHandler, tiene algunas características que llaman la atención y sugieren un re-planteo de esta. Una de las subclases de PbasicInputEventHandler es abstracta y no tiene métodos abstractos, tiene tres métodos vacíos.

Interfases.

Las interfaces permiten un uso alternativo del paradigma de herencia para la definición de tipos. Las métricas de interfases permiten hacerse una idea más cierta de la forma en que se están usando las interfases.

Cantidad de Interfases desarrolladas	**11**	**25**	**3**

Porcentaje de utilización de interfases propias (cantidad)con respecto a cantidad de clases.

17,74	**19,37**	**10**

Cantidad de jerarquías de Interfases desarrolladas	**0**	**2**	**0**	
Rango de niveles de Especialización por Jerarquías de Interfases desarrolladas	**0**	**1**	**0**	
Rango de Métodos de Interfase	**[0 - 8]**		**[0 - 10]**	**[1 - 5]**
Moda de Métodos de Interfase	**1**	**2**		**-**

- ***Se observa un mayor uso de interfases en Jazz_13, y un uso bajo en Piccolo.*** **Se simplifica el uso de las interfases en Piccolo, reduciéndola a:**

edu.umd.cs.piccolo.util.PNodeFilter
edu.umd.cs.piccolo.event.PInputEventListener
edu.umd.cs.piccolo.PComponent

- ***Las dos primeras versiones tienen interfases con 0 métodos.***
- **Corresponden a interfases usadas para definición de variables de uso común dentro de la aplicación. Se corrige en la versión de Piccolo.**

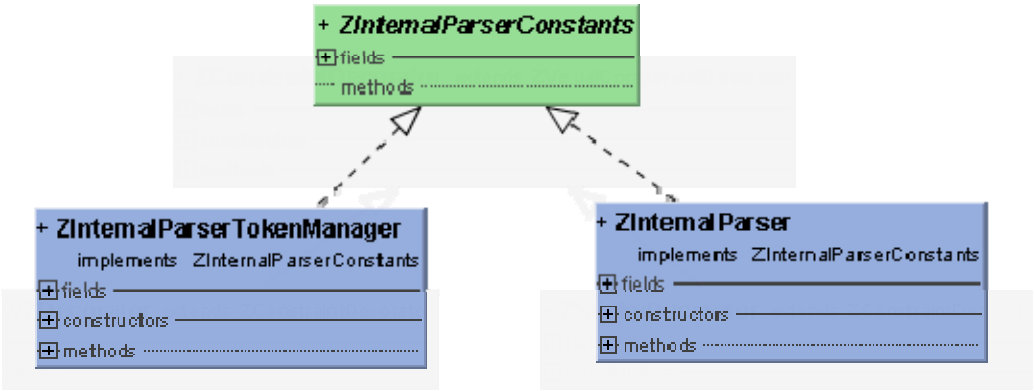
  Se detallan las clases correspondientes a interfaces con ningún método, diferenciando los productos.
  a) Jazz_05

  Existe una interfase sin operaciones

Edu.umd.cs.jazz.io.ZInternalParserConstants	**0**	**0**	**0**

[66] edu.umd.cs.jazz.ZBoundsLocator
[67] edu.umd.cs.piccolo.PInputManager

Existe una interfase sin operaciones ya que posee constantes solamente. Se ve que se quiso usar para que las clases que la implementen usen dichas constantes como propias. También, al no presentar una estructura de jerarquía apropiada, se realizó para no tener las mismas constantes en las dos clases que la implementan.



b) Jazz_13

Interfases sin métodos:

public interfase Zappearance
No es usada en ningún lado. Según el comentario de la interfase, se usa como ¨tag¨ para se extendida por todas las interfases que definen atributos visuales de un visual component.
Las interfases que extienden de ella son:
- ✓ **interfase ZpenPaint**
- ✓ **interfase ZfillPaint**
- ✓ **interfase ZpenColor**
- ✓ **interfase Zstroke**
- ✓ **interfase ZFillColor**

public interfase ZinternalParserConstants
Es implementada por dos clases. Contiene varias constantes que son usados por las clases que la implementan.
Las constantes no tienen los modificadores final y static.
Clases que la implementan:
- ✓ **class ZinternalParser**
- ✓ **class ZInternalParserTokenManager**

- ***La Moda de Métodos de Interfase aporta muy poca información.***
- **Se comprueba**

4.1.4 Polimorfismo

Polimorfismo	Jazz05	Jazz13	Piccolo
Cantidad de Mensajes polimórficos enviados	83[68]	234[69]	10[70]

- ***Jazz_05 y Jazz_13 son planteos flexibles. Piccolo es un producto poco flexible.***
- **Se comprueba. En Jazz_13, hay un aumento de complejidad con respecto a Jazz_05 y una mayor flexibilidad, al usar los beneficios del polimorfismo, tanto para tipos de interfase como para tipos de clases.**
  Piccolo tiene el valor más bajo de polimorfismo, de niveles en las jerarquías y muy pocas interfases.

4.1.5 Granularidad de métodos

Granularidad de métodos	Jazz05	Jazz13	Piccolo
Media de Statements por Método por Clase	5,90	4,95	3,59
Mediana de Statements por Método por Clase	4,07	3,66	3,03
Rango de Promedio de Statements por Método por Clase	[0 - 45,57]	[0 - 46,57]	[1,5 - 7,44]

- ***Se observa una disminución importante en la granularidad de los métodos, y un criterio de baja granularidad en la última versión.***
- **Se comprueba. En Piccolo, la clasePNode tiene un valor de 3,81. Siendo la clase principal de la aplicación y una clase con muchas responsabilidades, tiene una baja granularidad. La granularidad más alta la tiene AffineTransform, que es otra clase principal, porque permite variar de escala un elemento y trasladarlo.**

edu.umd.cs.piccolo.PNode	3.81

edu.umd.cs.piccolo.util.PAffineTransform	7.44

- ***Las dos primeras versiones tienen clases abstractas con todos los métodos abstractos.***

  Se muestran los valores para cada versión.

  Jazz_05

edu.umd.cs.jazz.event.ZNodeContainerAdapter	0.00
edu.umd.cs.jazz.event.ZnodeAdapter	0.00

  ZnodeContainerAdapter y ZnodeAdapter son clases que en la documentación dice que es abstracta, pero no es declarada como tal. Tiene 3 métodos vacíos[71].

  Jazz_13

[68] Los tipos de variable a los que envía mensajes polimórficos ZscenegraphObject (Interfase), Znode, ZVisualComponent
[69] Llama la atención el valor de la variable a los que envía mensajes polimórficos Zlist (interfase) que es 180.
[70] Los tipos de variable a los que envía mensajes polimórficos son PNode y Pcomponent (Interfase)
[71] Son clases que no están declaradas como abstractas, pero tienen métodos vacíos. Las que están resaltadas, son clases declaradas como abstractas pero no tienen métodos abstractos.

	0.00
edu.umd.cs.jazz.event.ZGroupAdapter	
edu.umd.cs.jazz.event.ZMouseAdapter	***0.00***
edu.umd.cs.jazz.event.ZNodeAdapter	**0.00**
edu.umd.cs.jazz.event.ZCameraAdapter	**0.00**
edu.umd.cs.jazz.event.ZTransformAdapter	**0.00**
edu.umd.cs.jazz.event.ZMouseMotionAdapter	***0.00***
edu.umd.cs.jazz.ZLeaf	**0.00**

- **Se encuentra una anomalía, con el uso de clases abstractas.**

- ***Sería suficiente contar con una de las dos medidas Media o mediana. Seleccionaría la mediana.***
- **Se comprueba. La mediana la uso para el cálculo de esfuerzo.**

4.1.6 Colaboración- Agregación

Colaboración - Agregación	**Jazz05**	**Jazz13**	**Piccolo**
Rango de Colaboradores Internos por Clase	**[0 - 10 [72]]**	**[0 - 14[73]]**	**[0 - 10 [74]]**
Mediana de Colaboradores Internos por Clase	**1**	**1**	**1,5**

- ***Miro valores extremos mayores a 10. Estimo que estos valores límites máximos apuntan a clases principales.***
- **Se comprueba. Facilita y agiliza la comprensión del paquete el hecho de comenzar a analizar el producto por estos valores.**

Se plantea si hay una versión que usa más intensivamente la colaboración. Se calcula el Porcentaje de la Cantidad de clases sin colaboradores internos / cantidad de clases.

- ***La segunda versión usa menos la colaboración.***

Porcentaje de clases que no tienen colaboradores.	**27,41**	**37,98**	**26,66**

- **Se comprueba lo observado.**

4.1.7 Responsabilidades públicas

Responsabilidades Públicas	**Jazz05**	**Jazz13**	**Piccolo**
Media de métodos de interfase por clase	**24,83**	**36,92**	**50,93**
Mediana de métodos de interfase por clase	**15**	**19**	**28**
Moda de métodos de interfase por clase	**4**	**4**	**29**

[72] edu.umd.cs.jazz.scenegraph.ZNode

[73]

edu.umd.cs.jazz.event.ZSwingEventHandler	14
edu.umd.cs.jazz.event.ZSelectionEventHandler	12
edu.umd.cs.jazz.ZCamera	11

[74]

edu.umd.cs.piccolo.PNode	10

Rango de métodos de interfase por clase	[1 - 105]	[1 - 167]	[4 - 168]

- ***Se observa un aumento de la media de las responsabilidades públicas en las diferentes versiones, de la mediana, de la moda y en el valor máximo del rango de métodos de Interfase por clase.***
- **Se comprueba.**

- **Jazz_05-Jazz13 al aumentar la media de responsabilidades públicas se espera que aumente el total de responsabilidades públicas del sistema.**

Total de Responsabilidades públicas	1540	4763	1528

- **Se comprueba. El aumento entre Jazz_05 y Jazz_13 se debe a que existe un aumento de prestaciones, además del tamaño.**

- **Jazz_13-Piccolo, disminuyendo significativamente el tamaño, al aumentar la media de responsabilidades públicas es posible que la funcionalidad soportada por el sistema, no sea proporcional al tamaño.**

- **Se comprueba. El aumento de responsabilidades públicas entre Jazz_13 y Picollo se debe a que se mantienen muchas de las prestaciones en Piccolo, disminuyendo significativamente el tamaño. Según se comprueba en la documentación y se observa en las clases, es posible hacer casi todo lo que se hace con Jazz_13, con Piccolo, a excepción de algunos aspectos que se detallan en el Anexo. Hay que tener en cuenta que "se puede hacer todo", no quiere decir que todo este implementado, como lo esta en Jazz_13.**
  Se observa que no hay un crecimiento desmedido en las responsabilidades de las clases, porque se mantiene el extremo máximo del rango de responsabilidades públicas por clase.

- ***La moda también es significativa. El valor más repetido es alto. Creo que el conjunto da información. No suprimiría ninguna.***
- **Se comprueba.**

Total de Responsabilidades públicas	1540	4763	1528

Esfuerzo (responsabilidades pub * mediana Statements)	6281,33	17464,33	4631,75

- ***Jazz_05 duplica el tamaño de Piccolo y tiene una cantidad de responsabilidades públicas similar a Piccollo.***
- **Observando el producto sólo se puede afirmar que Jazz_05 es más grande que Piccolo, y tienen la misma funcionalidad.**

- ***Jazz_13 cuatriplica el tamaño de Piccolo y triplica las responsabilidades públicas de Piccollo.***
- **Observando el producto sólo podría afirmar que Jazz_13 es más grande que Piccolo y que tiene una funcionalidad similar.**

- ***Jazz_05 duplica el tamaño de Piccolo y tiene un esfuerzo mayor que Piccollo.***

- **Observando el producto sólo podría afirmar que Jazz_05 es más grande que Piccolo.**

- ***Jazz_13 cuatriplica el tamaño de Piccolo y cutriplica el esfuerzo de Piccollo.***
- **Observando el producto sólo podría afirmar que Jazz_13 es más grande que Piccolo, y que tiene un esfuerzo mayor.**

4.1.8 Clases principales de la Aplicación

- ***Valores máximos en colaboración, granularidad de métodos y responsabilidades públicas apuntan a un grupo principal de clases.***

Jazz05

Muestro los valores máximos de:
a)granularidad de métodos

edu.umd.cs.jazz.util.ZLayout	**45,57**

b) Colaboradores y responsabilidades públicas (métodos propios, métodos heredados y total de métodos).

edu.umd.cs.jazz.scenegraph.ZNode	**10**		
edu.umd.cs.jazz.scenegraph.ZCamera	**28**	**77**	**105**

Jazz13

Muestro los valores máximos de: Colaboradores y responsabilidades públicas (métodos proios, métodos heredados y total de métodos.

edu.umd.cs.jazz.event.ZSwingEventHandler	**14**
edu.umd.cs.jazz.event.ZSelectionEventHandler	**12**
edu.umd.cs.jazz.ZCamera	**11**

15 clases con más de 100 responsabilidades públicas de la Jearaquía de Group

Piccollo

Muestro los valores máximos de: Colaboradores y responsabilidades públicas (métodos proios, métodos heredados y total de métodos.

edu.umd.cs.piccolo.PNode	**10**		
edu.umd.cs.piccolo.PNode	**134**	**0**	**134**

- **Se cumple en su mayoría. Descarto Zlayout, el resto apuntan a clases principales.**

Para las tres versiones, se detallan las clases que corresponden a los extremos máximos de responsabilidades públicas. En el caso de Piccolo se muestra a Pnode,

que es el padre de la clase que tiene el valor máximo. Se destaca que los valores máximos suelen apuntar directa o indirectamente a clases principales.

edu.umd.cs.jazz.scenegraph.ZCamera	**28**	**77**	**105**
edu.umd.cs.jazz.ZStickyGroup	**11**	**156**	**167**
edu.umd.cs.piccolo.PNode	**134**	**0**	**134**
Existe una clase que tiene 143, métodos. Aparentemente esta clase es padre de las clases con valores mayores a 100 (7 reportados).			

- ***Los valores extremos de los rangos suelen apuntar a clases principales del producto.***
- **Se comprueba.**

Nombre del producto:	**Jazz05**	**jazz13**	**Piccolo**
Cantidad de Clases desarrolladas	**62**	**129**	**30**
Cantidad de Interfases desarrolladas	**11**	**25**	**3**
Porcentaje Cantidad de Interfases desarrolladas / cantidad de clases	**17,74**	**19,37**	**10**
Reutilización			
Cantidad de Clases Externas especializadas	**14**	**24**	**6**
Cantidad de Interfases Externas extendidas	**3**	**7**	**1**
Cantidad de Clases que implementan Interfases Externas	**6**	**72**	**4**
Porcentaje de Reutilización sobre desarrollado	**31,50**	**66,88**	**33,33**
Herencia			
Cantidad de Jerarquías de Clases desarrolladas	**4**	**6**	**3**
Rango de niveles de Especialización por Jerarquía de Clases desarrolladas	**[1 - 3]**	**[1 - 5]**	**[1 - 2]**
Moda de niveles de Especialización por Jerarquía de Clases desarrolladas	**1**	**1**	**-**
Cantidad de Clases Raíz no Abstractas	**3**	**3**	**3**
Cantidad de Clases Raíz no Abstractas que implementan Interfases	**3**	**2**	**2**
Porcentaje de jerarquías sobre clases desarrolladas	**6,451**	**4,651**	**10**
Rango de Porcentaje de Métodos Reemplazados en Jer.	**[6.06 - 33,33]**	**[5 - 40]**	**[0,66 - 66,66]**
Cantidad de jerarquías de Interfases desarrolladas	**0**	**2**	**0**
Rango de niveles de Especialización por Jerarquías de Interfases desarrolladas	**0**	**1**	**0**
Rango de Métodos de Interfase	**[0 - 8]**	**[0 - 10]**	**[1 - 5]**
Moda de Métodos de Interfase	**1**	**2**	**-**
Polimorfismo			

Cantidad de Mensajes polimórficos enviados	**83**	**234**	**10**
Granularidad de métodos			
Media de Statements por Método por Clase	**5,90**	**4,95**	**3,59**
Mediana de Statements por Método por Clase	**4,07**	**3,66**	**3,03**
Rango de Promedio de Statements por Método por Clase	**[0 - 45,57]**	**[0 - 46,57]**	**[1,5 - 7,44]**
Colaboración - Agregación			
Rango de Colaboradores Internos por Clase	**[0 - 10]**	**[0 - 14]**	**[0 - 10]**
Mediana de Colaboradores Internos por Clase	**1**	**1**	**1,5**
Porcentaje de clases que no tienen colaboradores	**27,41**	**37,98**	**26,66**
Responsabilidades Públicas			
Media de Métodos de Interfase por Clase	**24,83**	**36,92**	**50,93**
Mediana de Métodos de Interfase por Clase	**15**	**19**	**28**
Moda de Métodos de Interfase por Clase	**4**	**4**	**29**
Rango de Métodos de Interfase por Clase	**[1 - 105]**	**[1 - 167]**	**[4 - 168]**

Total de Responsabilidades públicas	**1540**	**4763**	**1528**

Esfuerzo (responsabilidades pub * mediana Statements)	**6281,33**	**17464,33**	**4631,75**

4.1.9 Comentario de las métricas en su conjunto

- ***El Tamaño evidencia un cambio significativo entre las versiones. Dados los valores de las versiones, se debería encontrar un cambio significativo en el diseño entre Jazz13 y Piccolo. Entre Jazz05 y Jazz13 se espera una evolución del producto.***
- **Se comprueba.**[75]

- ***Las métricas nos resaltan los siguiente puntos de Piccollo:***
  No se apoya en interfases.
  Reutiliza menos.
  Su diseño se apoya en las jerarquías de clase.
  Es poco flexible.
  Tiene una menor granularidad de métodos y un aumento de responsabilidades públicas.
- **Se comprueba.**

- ***Las métricas nos resaltan los siguientes puntos de Jazz13 con respecto a Jazz05.***

  Aumento de Tamaño, es el doble.
  Triplica las responsabilidades públicas.
  Casi triplica el esfuerzo.
  Reutiliza más las librerías de javax.
  Aumenta los niveles de las jerarquías y la cantidad de jerarquías por lo tanto aumenta la reutilizabilidad del diseño.

[75] Cfr. Anexo de Piccolo

Desarrolla jerarquías de interfases.
Es más flexible.
Se mantiene la granularidad de métodos, a pesar del aumento de tamaño. Cosa positiva.
Disminuye el uso de colaboración pero hay un valor límite de cantidad de colaboradores más alto.
Aumenta la cantidad de responsabilidades públicas, y aumenta los valores/límites.
No se puede decir con exactitud en qué se apoya el diseño (jerarquías / interfases / colaboración). Sí se puede afirmar que la variación de versión se basa en la reutilización.

- **Se comprueban todos los puntos anteriores. Hay aspectos que para comprobarlos, se necesitaría un gran esfuerzo, por ejemplo la exactitud de los valores de las métricas[76].**

[76] En la comprobación de estos puntos no se han re-calculado los valores en su totalidad.

4.2 Piccolo-Jgraph-Gef

Se muestran en forma comparativa los valores de las métricas obtenidas de Piccolo, Jgraph y Gef.

4.2.1 Tamaño

Cantidad de Clases desarrolladas	**30**	**29**	**235**
Cantidad de Interfases desarrolladas	**3**	**13**	**22**
Porcentaje Cantidad de Interfases desarrolladas /cantidad de clases	**10**	**44,82**	**9,36**

- ***Tengo tres frameworks diferentes, dos de los cuales tienen un tamaño similar y un tercero que es casi ocho veces más grande.***
- **Salvo alguna excepción puntual los tres tienen una funcionalidad similar.**

Piccolo	JGraph	Gef
100% Java **Compatibilidad total con Swing** **Arrastra y libera** **Copia y pega** **Zoom**	**100% Java** **Compatibilidad total con Swing** **Arrastra y libera** **Copia y pega** **Zoom**	**100% Java** **Compatibilidad total con Swing** **Arrastra y libera** **Copia y pega** **Zoom**
	Ofrece XML	**Ofrece XML**
Usa el modelo de "scenegraph", que es común en los ambientes de 3D. Tiene una estructura jerárquica de objetos y cámaras	***" Un objeto de JGRAPH en realidad no contiene sus datos; esto simplemente proporciona una vista de los datos. Como cualquier componente de Swing no trivial, la componente gráfica toma datos del modelo de datos ".***	***"Model-View-Controller diseño basado en la librería de UI de Java hace a GEF capaz de actuar como un UI de estructuras de datos existentes, y también reduce el tiempo de estudio para desarrolladores familiares con Swing"***

La proliferación de clases de Gef se debe a que tiene jerarquías con varias subclases. Piccolo es un planteo simple y en Jgraf el planteo esta basado en interfases con implementaciones por default.
La cantidad de clases que existen en Gef dificulta la comprensión del paquete. Además, la distribución de clases en paquetes no tiene un ordenamiento adecuado. Facilita la comprensión del paquete la documentación.

El segundo framework, Jgraph, tiene un porcentaje alto de interfases, con respecto al número total de clases.

- ***Se realta el valor de 44,82% de Jgraf. Este porcentaje indica una característica del framework; se puede decir que el framework JGraph es "orientado a interfases".***
- **Sí, según se puede comprobar en el Anexo del producto la conceptualización de éste en primer lugar se hace en términos de interfases. La mayoría, a excepción de**

dos de ellas[77], tienen implementaciones; en muchas de estas implementaciones se dice explícitamente que son implementaciones por default.

Piccolo usa las interfases como una herramienta de diseño-programación típica generalizando comportamiento.
Gef tiene una intención de plantear el diseño a partir de interfases solamente en el paquete Graph. Esta intención no se continúa en el resto de los paquetes (cfr. Anexo)

4.2.2 Reutilización

Reutilización	Piccolo	JGraph	Gef
Cantidad de Clases Externas especializadas	6	12	40
Cantidad de Interfases Externas extendidas	1	3	14[78]
Cantidad de Clases que implementan Interfases Externas	4	16	83
Porcentaje de Reutilización sobre desarrollado	33,33	73,80	53,30

Piccolo tiene un porcentaje de reutilización bajo.

- ***Es una librería que no apoya sus características esenciales en la reutilización.***
- **Es así porque su objetivo es simplificar el diseño. Provee una colección de clases muy sencilla, fáciles de usar y comprender. Además caracteriza a este producto la facilidad con que se integra a aplicaciones en Java.**

Jgraph tiene el porcentaje de reutilización más alto.

- ***Se puede afirmar que el diseño esta basado en la reutilización.***
- **Sí. Describiría básicamente como un Framework para desarrolladores de Java. Usa todo lo que puede de estas librerías.[79] Caracterizaría el framework como un conjunto de clases que completa la librería de Java, manteniendo la complejidad de estas librerías. Especialmente reutiliza javax.swing.**

- ***Gef evidencia un porcentaje medio de reutilización.***
- **No hay duda que reutiliza, pero no llega a ser el rasgo que más lo caracteriza.**

Se detallan los paquetes que importan los productos.

[77] ValueChangeHandler y GraphSelectionModel
[78] Serializable y EventListener
[79] Lo único que llama la atención es que no usae la clase AffineTransform, para el traslado y escalamiento de las imágenes. Pero podría se perfectamente incorporada en alguna subclase.

Piccollo	JGraf	Gef
		** Detalle de Packages.
	Javax.swing.tree	javax.swing.tree
		org.apache.log4j.helpers
	java.awt.datatransfer	org.w3c.dom
java.io		java.io
java.awt.geom.		java.applet
java.text		javax.swing.text
java.util	Java.util	java.util.zip
java.awt		org.xml.sax
Java.lang	Java.lang	javax.swing.table
Java.awt.print		java.awt.print
		org.apache.log4j
		javax.swing.border
		java.lang.reflect
		java.awt.datatransfer
	java.awt.event	java.awt.event
		java.awt
		java.awt.image
javax.swing	Javax.swing	javax.swing
	javax.accessibility	Acme
	java.awt.dnd	java.lang
		Acme.JPM.Encoders
		java.util
		javax.swing.plaf.basic
		java.net
		java.beans
		javax.swing.event

4.2.3 Herencia

Herencia	Piccolo	JGraph	Gef	
Cantidad de Jerarquías de Clases desarrolladas	3	4	12	
Rango de niveles de Especialización por Jerarquía de Clases desarrolladas	[1 - 2]	1		[1 - 4]
Moda de niveles de Especialización por Jerarquía de Clases desarrolladas	-	1	1	
Cantidad de Clases Raíz no Abstractas	3	3	3	
Cantidad de Clases Raíz no Abstractas que implementan Interfases	2	3	2	
Porcentaje de jerarquías sobre clases desarrolladas	10	13,79	5,10	

- ***El diseño de Piccolo y Jgraf esta basado en jerarquías, dado que rondan los valores alrededor de un 10 %.***
- **Se cumple en el caso de Piccolo, no en el Jgraf. En Jgraf, las jerarquías a que apuntan las métricas, no pertenecen al grupo de clases principales, por eso no lo caracterizan. Además el rango de 1 nivel de las jerarquías de Jgraf, manifiestan que están poco desarrolladas.**

Gef tiene un valor bajo de jerarquías, con respecto a la cantidad de clases. Además tiene algunas jerarquías con numerosas subclases (como ser en el caso de Fig [80]).

- ***Los 4 niveles en Gef, no es un valor alto para la cantidad de clases***

java.awt, 259 clases, 4 niveles

- **Los 4 niveles de jerarquías, si bien están dentro de un intervalo esperado por el tamaño de la aplicación, manifiestan el grado de complejidad de una de estas jerarquías.**

- ***La Moda de niveles de Especialización por Jerarquía de Clases desarrolladas no aporta información.***
- **Se comprueba.**

- ***Se analizan las raíces de las jerarquías. Miro el detalle de la métrica. En sí misma no da datos muy precisos. Busco aquellas jerarquías que no tienen la raíz abstracta y las analizo con detalle.***
- **Piccolo, tiene una sola jerarquía propia donde la raíz no es abstracta ni implementa interfases, se la analiza y no se encuentran errores (cfr. Caso de estudio anterior).**
  Jgraf no tiene ninguna jerarquía propia donde la raíz no es abstracta ni implementa interfases. Igualmente Gef.

- ***Esta métrica esta incompleta. Sólo estoy mirando las jerarquías donde la raíz pertenece al alcance de medición. Es probable que este perdiendo alguna jerarquía importante al despreciar las jerarquías en donde la raíz esta fuera del alcance de medición.***
- **En Piccolo, la extensión de clases externas no generan jerarquía nuevas (la clase propia no tiene subclases).**
  En Jgraph existe 3 jerarquía, 1 de ellas es parte del grupo de clases principales:
  - **o *DefaultGraphCell extends DefaultMutableTreeNode implements* GraphCell, Cloneable**
  - **o GraphUI extends ComponentUI**
  - **o GraphViewLayerEdit extends AbstractUndoableEdit**

  En Gef, no son parte del grupo de clases principales.
  - **o TableMap extends AbstractTableModel**
  - **o PropSheet extends Jpanel**
  - **o ToolBar extends JtoolBar**
  - **o Cmd extends AbstractAction, son comandos, tiene alrededor de 50 sub-clases**

Rango de Porcentaje de Métodos Reemplazados en Jerarquías

Rango de Porcentaje de Métodos Reemplazados en Jerarquías (PMRJ)	**[0,66 - 66,66]**	**0**	**[0.63 - 100]**

- ***Gef presenta un valor muy alto de sobre-escritura. Es probable que exista un planteo incorrecto de una clase.***

[80] org.tigris.gef.presentation.Fig

- **Existe un error en el programa, el porcentaje esta mal calculado.**

Interfases

Estos valores completan la información de la cantidad de interfases con las que estoy trabajando. Sirve para hacerse una idea más cierta de la forma en que se estan usando las interfases.

Cantidad de Interfases desarrolladas	**3**	**13**	**22**

Porcentaje de utilización de interfases propias (cantidad)con respecto a cantidad de clases.

Porcentaje Cantidad de Interfases desarrolladas /cantidad de clases	**10**	**44,82**	**9,36**

	Piccoll o	JGraf	Gef
Cantidad de jerarquías de Interfases desarrolladas	**0**	**1**	**1**
Rango de niveles de Especialización por Jerarquías de Interfases desarrolladas	**0**	**1**	**1**
Rango de Métodos de Interfase	**[1 - 5]**	**[1 - 27 [81]]**	**[1 - 27[82]]**
Moda de Métodos de Interfase		**- 1**	**1**

Los valores de Jgraf comparativamente con Gef, son altos, dada la diferencia de tamaño.

- ***Estimo que refuerza la idea de que Jgraf esta orientado a interfases.***
- **Se comprueba. GraphModel pertenece al grupo de clases principales y es una interfase con muchos métodos.**

- ***La Moda de Métodos de Interfase aporta muy poca información.***
- **Se comprueba.**

4.2.4 Polimorfismo

Polimorfismo	**Piccolo**	**JGraph**	**Gef**
Cantidad de Mensajes polimórficos enviados	**10**	**44[83]**	**158[84]**

- ***El planteo de Piccolo es poco flexible, porque tiene un valor muy bajo de polimorfismo.***
- **Se comprueba en el producto porque la intención del producto es simplificar el diseño, una alta flexibilidad puede implicar una mayor complejidad del producto.**

[81] org.jgraph.graph.GraphModel, defines a suitable data model for a JGraph.
[82] org.tigris.gef.graph.MutableGraphModel, hija de GraphModel, permite modificaciones además de acceso como su padre (facede)
[83] No tiene mensajes polimórficos a clases, sólo a interfases.
[84] Mas de 100 de mensajes corresponde a Fig y FigNode

- ***El valor de mensajes polimórficos en Jgraf refuerza la idea de un Framework orientado a interfases.***
- **Se comprueba. Tiene mensajes polimórficos solo a tipos de interfases, no tiene mensajes enviados a tipos de clases.**

- ***El planteo de Gef no potencia las posibilidades de flexibilidad que tiene el producto.***
- **Comparando con Jazz_13, una aplicación con menor cantidad de clases, y menor cantidad de jerarquías, tiene un valor mayor de polimorfismo. Confirma el planteo que es difícil caracterizar este producto.**

Nombre del producto:	**jazz13**	**Gef**
Cantidad de Clases desarrolladas	**129**	**235**
Cantidad de Interfases desarrolladas	**25**	**22**
Herencia		
Cantidad de Jerarquías de Clases desarrolladas	**6**	**12**
Rango de niveles de Especialización por Jerarquía de Clases desarrolladas	**[1 - 5]**	**[1 - 4]**
Polimorfismo		
Cantidad de Mensajes polimórficos enviados	**234**	**158**

4.2.5 Granularidad de métodos

Granularidad de métodos	**Piccolo**	**JGraph**	**Gef**
Media de Statements por Método por Clase	**3,59**	**3,84**	**3,73**
Mediana de Statements por Método por Clase	**3,03**	**3,58**	**3,73**
Rango de Promedio de Statements por Método por Clase	**[1,5 - 7,44]**	**[0 - 6,88]**	**[0 - 41]**

- ***Jgraph y Gef tienen clases abstractas con todos los métodos abstractos.***
- **Se comprueba en el caso de Jgraph, al mismo tiempo llama la atención que no se defina una interfase, viendo que la tendencia de este framework es abstraer con interfases.**

org.jgraph.plaf.GraphUI	**0.00**

Se comprueba para Gef. Es una clase que tiene un método estático[85], sin nombre.

org.tigris.gef.util.logging.LogManager	**0.00**

- ***Se mantienen los valores medios, llama la atención el límite de 41 en Gef, al comparar las versiones. Por la mediana son valores puntuales altos.***
- **Se comprueba. Corresponde a una clase que dibuja un diamante al final de FigEdge.**

org.tigris.gef.presentation.ArrowHeadQualifier	**41.00**

[85] Para la invocación de un método estático no es necesario instanciar la clase, se invoca dirctamente la clase.

- ***Sería suficiente contar con una de las dos medidas Media o mediana. Seleccionaría la mediana.***
- **Se comprueba.**

4.2.6 Colaboración - Agregación

Colaboración - Agregación	Piccolo	JGraph	Gef
Rango de Colaboradores Internos por Clase	[0 - 10]	[0 - 11[86]]	[0 - 13[87]]
Mediana de Colaboradores Internos por Clase	1,5	2	1
Porcentaje de clases que no tienen colaboradores.	26,66	20,68	47,23

Se miran los valores extremos. Se busca la clase que tiene más colaboradores. Se analiza si hay una versión que usa más intensivamente la colaboración. Se calcula: Porcentaje de la Cantidad de clases *sin* colaboradores internos / cantidad de clases.

- ***Gef usa pocos colaboradores internos.***
- **Se comprueba. Los números más altos de colaboración no están dados en las clases principales, a excepción de Editor con 10 colaboradores.**

4.2.7 Responsabilidades públicas

Responsabilidades Públicas	Piccolo	JGraph	Gef
Media de Métodos de Interfase por Clase	50,93	24,32	33,98
Mediana de Métodos de Interfase por Clase	28	16,5	18
Moda de Métodos de Interfase por Clase	29	4	17
Rango de Métodos de Interfase por Clase	[4 - 168]	[2 - 125]	[0 - 226]

Totales de Responsabilidades públicas	***1528***	***681***	***7986***

- ***Si comparo Piccollo con Jgraph, la métrica dice que Piccolo tiene más responsabilidades públicas. Piccolo tiene más responsabilidades públicas (3 veces más) a igual tamaño. La moda y la media refuerzan la visión de que hay más responsabilidades públicas en Piccolo.***
- **Aspectos difíciles de percibir sin la métrica.**

- **Gef tiene dos clases sin responsabilidades públicas.**

Acme.IntHashtableEntry [88]	0	0	0
org.tigris.gef.util.logging.LogManager [89]	0	0	0

- **Se comprueba.**

---

[86] org.jgraph.plaf.basic.BasicGraphUI 11
Veo 23colaboradores. Es un Bugg.

[87] org.tigris.gef.properties.ui.TabPropFrame 13

[88] Es una clase de una librería librería externa

[89] Es una clase con un método sin nombre, caracterizado como static.

*Esfuerzo: Resp. Publicas * Mediana Statements*	*4631,75*	*2439,15*	*29842,42*

Piccolo al igual tamaño que JGraph, duplica-triplica las responsabilidades.

Gef es casi ocho veces más grande que Piccollo y quintuplica las responsabilidades.

- ***Se analiza el esfuerzo necesario para implementar una aplicación a partir de cada uno de los frameworks.***

- **Piccolo no tiene métodos abstractos[90]. 3 Interfases sin implemenación, con un total de 7 métodos.**

  Jgraf, 2 clases abstractas con un total de 13 métodos abstractos. Dos interfases sin clases que las implementan con un total de 26 métodos.

  Gef, hay que realizar un mayor esfuerzo para comprenderlo. Tiene 4 interfases sin implementar. Existe 1 clases abstracta sin subclases, con dos métodos abstractos.

4.2.8 Clases Principales de la Aplicación

- ***Analizo si colaboración, granularidad de métodos y responsabilidades públicas apuntan a un grupo clave de clases.***
- **Se cumple en Piccolo. Se muestran los colaboradores de Pnode y las responsabilidades públicas (métodos propios, métodos heredados, total de métodos) para Pnode y Pcamara.**

edu.umd.cs.piccolo.PNode	**10**		
edu.umd.cs.piccolo.PNode	**134**	**0**	**134**
edu.umd.cs.piccolo.PCamera	**34**	**134**	**168**

Se cumple para Jgraph, a excepción de GraphConstants. GraphConstants no es parte del grupo de clases principales pero nuclea constantes y métodos static que se usan con frecuencia en el producto. Muestro en la tabla colaboradores para BasicGraphUI, responsabilidades públicas (métodos propios, métodos heredados, total de métodos) para JGraph y GraphConstants, y métodos de interfaces para GraphModel.

org.jgraph.plaf.basic.BasicGraphUI	**11**		
org.jgraph.JGraph	**125**	**0**	**125**
org.jgraph.graph.GraphConstants	**87**	**0**	**87**
org.jgraph.graph.GraphModel	**27**		

En el caso Gef se cumple a excepción de TabPropFrame. La impresión que da este framework es que se han agregando desarrollos realizados por diversas personas, para necesidades específicas de cada una. Por eso las métricas dan cierta dispersión. Mostramos en las tablas siguientes granularidad de métodos para

[90] Cosa llamativa tiene una clase abstracta sin métodos abstractos. En el cometario da indicaciones de los métodos que se tienen que sobrescribir.

ArrowHeadQualifier, colaboradores para TabPropFrame y responsabilidades públicas métodos propios, métodos heredados, total de métodos) para las restantes.

org.tigris.gef.presentation.ArrowHeadQualifier	**41,00**

org.tigris.gef.properties.ui.TabPropFrame	**13**

org.tigris.gef.presentation.FigText	**67**	**159**	**226**
org.tigris.gef.demo.FigSampleNode	**1**	**221**	**222**
org.tigris.gef.presentation.FigNode	**30**	**191**	**221**

Estos valores apuntan a la clase Fig. Se describe a continuación un breve análisis de esta clase.

Como se observa en el modelo de diseño (cfr. Anexo), todas las figuras se basan en una clase raíz Fig. Esta clase tiene la particularidad que la mayoría de sus métodos están contenidos en otras clases de java. Los mismos están siendo redefinidos en vez de ser instanciadas las clases que los implementan (java.awt y swing). Hemos encontrado algunos ejemplos, que se repiten en la clase:

A- ***getX() y getY():*** **podrían haber sido invocados desde un colaborador del tipo java.awt.Pointer.**

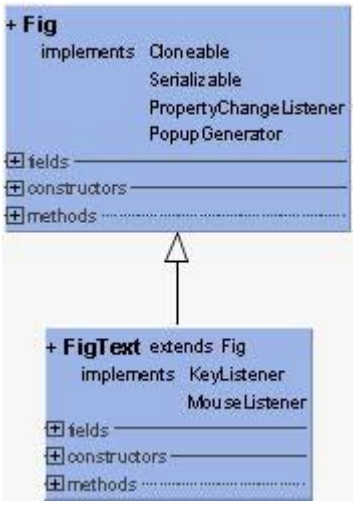
B- ***paint(), draw(), redraw(), drawDrashedLine():*** **son propios de la clase java.awt.Image, aunque alguno de ellos es abstracto.**

C- ***getBounds() y getBounds(Rectangle):*** **estos pertenecen propiamente a java.awt.Rectangle y javax.swing.JComponent respectivamente.**

No queda clara la intención del diseñador. Esto se podría solucionar afinando el diseño de la clase. Tampoco se comprende por qué no es declarada abstracta, ya que la misma no es instanciada en ningún momento, incluso en las aplicaciones. Se llega a concluir que este diseño es válido, pero a futuro puede ser complicado mantenerlo.

A partir de lo dicho sobre Fig, estudiamos los siguientes casos:

org.tigris.gef.presentation.FigText



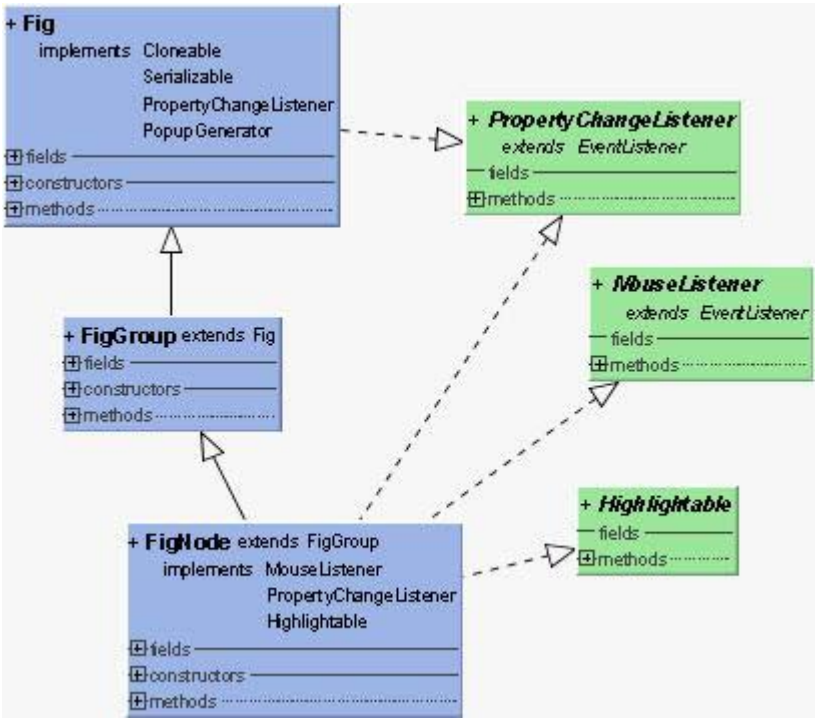
La clase FigText hereda de Fig, como se puede ver en el esquema superior, y con ello sus responsabilidades. Muchas de las mismas no son propias de FigText, como es el caso de getPopupAction(), quedando una subclase con métodos no adecuados a la intención de la clase. Ante este esquema, el diseñador se vio obligado a sobrescribir métodos en FigText y a agregar una cantidad significativa para generar la adaptación necesaria, llegando a un total de 67 métodos nuevos, siendo un llamado de atención sobre el diseño del producto.

La clase FigText, que comprende tantas responsabilidades, es muy difícil de reutilizar, pues aunque se requieran determinadas funciones de la misma, su costo es muy alto debido al resto de las responsabilidades obligadas a recibir.

Además, generar una optimización de la clase será mucho mas complejo por la alta responsabilidad asignada.

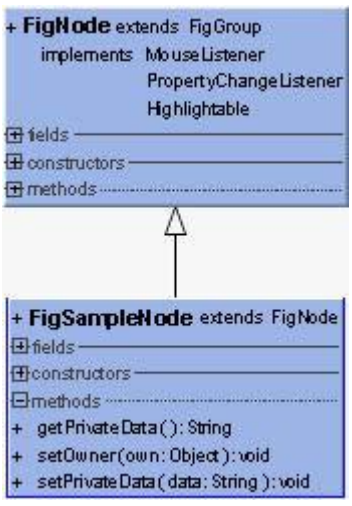
Una solución bastante simple hubiese sido heredar de las clases javax.swing.JTextPane, en vez de Fig, y hacer las adaptaciones necesarias.

org.tigris.gef.presentation.FigNode



La clase FigNode hereda de FigGroup y esta de Fig, clase principal de la aplicación, ambas con mucha responsabilidad, lo que obliga a FigNode a adaptarse sobre dichas responsabilidades sobrescribiendo métodos. Con este esquema el modelo presentado es aceptable, pues los métodos nuevos de la clase FigNode son tan solo 30 que, comparando con sus padres, se puede decir que esta dentro de los rangos aceptados por las métricas. Debe notarse que ante una crítica al diseño, esta se debe hacer sobre la jerarquía de padres de FigNode, y no sobre ella, pues son los padres quienes han asumido muchas responsabilidades, en especial Fig.

org.tigris.gef.demo.FigSampleNode



La clase FigSampleNode es una aplicación que hereda de FigNode, y lo que debemos notar es que se define tan solo un método nuevo y se sobrescriben dos. El problema radica en el diseño: mucha responsabilidad sobre los padres que obligan a los hijos a una alta adaptación y a aceptar responsabilidades innecesarias para los mismos. Si la estructura fuese mas clara no deberían sobrescribirse métodos en esta aplicación.

Se presenta un resumen comparativo de las métricas para los tres productos.

Nombre del producto:	**Piccolo**	**Jgraph**	**Gef**
Cantidad de clases desarrolladas	**30**	**29**	**235**
Cantidad de interfases desarrolladas	**3**	**13**	**22**
Porcentaje cantidad de interfases desarrolladas /cantidad de clases	**10**	**44,82**	**9,36**
			6,11
Reutilización			
Cantidad de clases externas especializadas	**6**	**12**	**40**
Cantidad de interfases externas extendidas	**1**	**3**	**14**
Cantidad de clases que implementan interfases externas	**4**	**16**	**83**
Porcentaje de reutilización sobre desarrollado	**33,33**	**73,80**	**53,30**
Herencia			
Cantidad de jerarquías de clases desarrolladas	**3**	**4**	**12**
Rango de niveles de especialización por Jerarquía de Clases desarrolladas	**[1 - 2]**	**1**	**[1 - 4]**
Moda de niveles de Especialización por Jerarquía de Clases desarrolladas	**-**	**1**	**1**
Cantidad de Clases Raíz no Abstractas	**3**	**3**	**3**
Cantidad de Clases Raíz no Abstractas que implementan Interfases	**2**	**3**	**2**
Porcentaje de jerarquías sobre clases desarrolladas	**10**	**13,79**	**5,10**
Rango de Porcentaje de Métodos Reemplazados en Jerarquías (PMRJ)	**[0,66 - 66,66]**	**0**	**[0.63 - 100]**
Cantidad de jerarquías de Interfases desarrolladas	**0**	**1**	**1**
Rango de niveles de Especialización por	**0**	**1**	**1**

Jerarquías de Interfases desarrolladas			
Rango de Métodos de Interfase	**[1 - 5]**	**[1 - 27]**	**[1 - 27]**
Moda de Métodos de Interfase	**-**	**1**	**1**
Porcentaje de Jerarquías de interfases sobre cantidad de interfases	**0**	**7,69**	**4,54**
Nombre del producto:	**Piccolo**	**Jgraph**	**Gef**
Polimorfismo			
Cantidad de Mensajes polimórficos enviados	**10**	**44**	**158**
Granularidad de métodos			
Media de Statements por Método por Clase	**3,59**	**3,84**	**3,73**
Mediana de Statements por Método por Clase	**3,03**	**3,58**	**3,73**
Rango de Promedio de Statements por Método por Clase	**[1,5 - 7,44]**	**[0 - 6,88]**	**[0 - 41]**
Colaboración - Agregación			
Rango de Colaboradores Internos por Clase	**[0 - 10]**	**[0 - 11]**	**[0 - 13]**
Mediana de Colaboradores Internos por Clase	**1,5**	**2**	**1**
Porcentaje de clases que no tienen colaboradores	**26,6666667**	**20,6896552**	**47,2340426**
Responsabilidades Públicas			
Media de Métodos de Interfase por Clase	**50,9333333**	**24,3214286**	**33,9829787**
Mediana de Métodos de Interfase por Clase	**28**	**16,5**	**18**
Moda de Métodos de Interfase por Clase	**29**	**4**	**17**
Rango de Métodos de Interfase por Clase	**[4 - 168]**	**[2 - 125]**	**[0 - 226]**

4.2.9 Comentario de las métricas en su conjunto.

- ***Piccollo. Se apoya en las jerarquías para plantear el diseño. Es normal el porcentaje de Interfases sobre clases (10%). Las interfases tiene una cantidad baja de responsabilidad. Tiene una buena granularidad, brinda muchos servicios. Reutiliza muy poco. Es poco flexible.***
- **Viendo las clases se ve un Framework sencillo, que no requiere un conocimiento de un experto en Java para su fácil uso. Tiene mucho implementado.**

- ***Jgraph: los valores de interfases son los más altos de los tres. También el porcentaje de jerarquías. Reutiliza, tiene el valor más alto de reutilización.***
- **En el diagrama de clases se observa un Framework orientado a interfases. Es una solución de diseño para hábiles programadores de Java. Tiene implementado lo**

esencial. No limita al programador, lo deja libre en el uso del lenguaje. Extiende el ambiente de Java.

- ***Gef: reutiliza medianamente. Pocas jerarquías con respecto al tamaño, puede haber mucha subclasificación, puede ser un Framework con mucha implementación de subclases, por eso da el valor de jerarquías bajo con respecto al total de clases. Bajo polimorfismo. Tiene 22 interfases y sólo 1 jerarquía de interfases. Baja colaboración. Los servicios prestados por las clases tiene una media baja. Es difícil caracterizar este diseño.***
- **Se comprueba. Es un framework y una librería completa, pero a la vez compleja. Se ha priorizado un contenido completo y extenso.**

- ***Se diría que Gef es una aplicación que ha crecido, pero sin un re-diseño adecuado. La impresión que da es que si Jazz13 hubiera seguido creciendo pero que no se le hubiera hecho un re-planteo de fondo, se obtendría un producto similar a Gef.***
- **Se comprueba[91]. Da la impresión de ser más una librería que un Framework[92].**
  Se muestra una tabla comparativa para observar la evolución supuesta de Jazz13.

Nombre del producto:	**Jazz05**	**jazz13**	**Gef**
Cantidad de Clases desarrolladas	**62**	**129**	**235**
Cantidad de Interfases desarrolladas	**11**	**25**	**22**
Porcentaje Cantidad de Interfases desarrolladas /cantidad de clases	**17,74**	**19,37**	**9,36**
Reutilización			
Cantidad de Clases Externas especializadas	**14**	**24**	**40**
Cantidad de Interfases Externas extendidas	**3**	**7**	**14**
Cantidad de Clases que implementan Interfases Externas	**6**	**72**	**83**
Porcentaje de Reutilización sobre desarrollado	**31,50**	**66,88**	**53,30**
Herencia			
Cantidad de Jerarquías de Clases desarrolladas	**4**	**6**	**12**
Rango de niveles de Especialización por	**[1 - 3]**	**[1 - 5]**	**[1 - 4]**

[91] Párrafos copiados de la documentación de Gef:

"Básicamente, quiero tener un editor gráfico bueno, estable y un framework de edición gráfico de estilo Mac. Lo necesito como un punto de partida para una herramienta que estoy construyendo para mi trabajo de investigación en el Doctorado, y pienso que otros pueden necesitar un framework similar...

Pienso que sería realmente interesante a organizar "un equipo de desarrollo de software virtual" en Internet de modo que: (1) no tenga que hacer toda la codificación yo mismo, (2) el framework es perfeccionado por el uso real, (3) logramos ganar experiencia en el desarrollo distribuido, (4) entro en contacto con desarrolladores profesionales para una futura colaboración, pruebas de usuario, y aún ofertas de trabajo. Como líder de equipo virtual yo me comprometo a integrar cambios, controlar la página web de proyecto, sugerir proyectos, y evitar la duplicación de esfuerzo... "

[92] Con esta expresión se quiere resaltar la idea de que lo que se epera de un framework es una arquitectura a extender, esto es a especializar, no se espera que brinde las clases ya espcializadas.

Jerarquía de Clases desarrolladas			
Moda de niveles de Especialización por Jerarquía de Clases desarrolladas	**1**	**1**	**1**
Cantidad de Clases Raíz no Abstractas	**3**	**3**	**3**
Cantidad de Clases Raíz no Abstractas que implementan Interfases	**3**	**2**	**2**
Porcentaje de jerarquías sobre clases desarrolladas	**6,45**	**4,65**	**5,10**
Rango de Porc. Mét. reem en Jerar. (PMRJ)	**[0,66 - 66,66]**	**0**	**[0.63 - 100]**
Cantidad de jerarquías de Interfases desarrolladas	**0**	**2**	**1**
Rango de niveles de Especialización por Jerarquías de Interfases desarrolladas	**0**	**1**	**1**
Rango de Métodos de Interfase	**[0 - 8]**	**[0 - 10]**	**[1 - 27]**
Moda de Métodos de Interfase	**1**	**2**	**1**
Polimorfismo			
Cantidad de Mensajes polimórficos enviados	**83**	**234**	**158**
Granularidad de métodos			
Media de Statements por Método por Clase	**5,90**	**4,95**	**3,73**
Mediana de Statements por Método por Clase	**4,07**	**3,66**	**3,73**
Rango de Promedio de Statements por Método por Clase	**[0 - 45,57]**	**[0 - 46,57]**	**[0 - 41]**
Colaboración - Agregación			
Rango de Colaboradores Internos por Clase	**[0 - 10]**	**[0 - 14]**	**[0 - 13]**
Mediana de Colaboradores Internos por Clase	**1**	**1**	**1**
Porcentaje de clases que no tienen colaboradores.	**27,41**	**37,98**	**47,23**
Responsabilidades Públicas			
Media de Métodos de Interfase por Clase	**24,83**	**36,92**	**33,98**
Mediana de Métodos de Interfase por Clase	**15**	**19**	**18**
Moda de Métodos de Interfase por Clase	**4**	**4**	**17**
Rango de Métodos de Interfase por Clase	**[1 - 105]**	**[1 - 167]**	**[0 - 226]**
Total de Responsabilidades públicas	**1540**	**4763**	**7986**
Esfuerzo (responsabilidades pub * mediana Statements)	**6281,33**	**17464,33**	**29842,42**

5 CONCLUSIONES

5. 1 Conjunto básico de métricas. Extensión de este conjunto básico.

Se ha planteado un grupo básico de métricas y una extensión de este grupo, los que permiten caracterizar un producto y señalar alarmas. Las alarmas facilitan el análisis de un producto y apuntan a valores fuera de un rango esperado. Los valores de las alarmas suelen pertenecer a las clases principales de la aplicación. El análisis de estos valores agiliza la comprensión y la observación del diseño de un producto de software.

Se trabaja sobre un conjunto básico de métricas. Algunas de ellas se definen como marco para la comprensión de otras.

Aspecto a medir	Nombre de Métrica
Tamaño	Cantidad de clases desarrolladas Cantidad de interfases desarrolladas
Reutilización -Agregación	Cantidad de colaboradores externos por clase
Reutilización - Herencia	Cantidad de clases externas especializadas Cantidad de Interfases externas extendidas Cantidad de clases que implementan Interfases externas
Herencia	Cantidad de jerarquías de clases desarrolladas. Cantidad de jerarquías de interfases desarrolladas. Cantidad de jerarquías extendidas de clases externas.
Herencia-Especialización	Cantidad de niveles de especialización por jerarquía de clases. Cantidad de niveles de especialización por jerarquía de interfases. Cantidad de niveles agregados a jerarquías donde la raíz es externa.
Herencia-Generalización	Cantidad de clases raíz no abstractas. Cantidad de clases raíz no abstractas que implementan interfases
Herencia-Sobre-escritura	Porcentaje de métodos reemplazados en una jerarquía. Porcentaje de métodos reemplazados en jerarquías donde la raíz es externa.
Polimorfismo	Cantidad de mensajes polimórficos enviados.
Granularidad de métodos.	Promedio de statements por método en una clase.
Responsabilidades públicas.	Cantidad de métodos de interfaz por clase.
Colaboración - Agregación.	Cantidad de colaboradores por clase

Sobre esta base se define:

- **un conjunto de mediciones estadísticas que facilitan la compresión de las series de datos obtenidas al aplicar las métricas (por ejemplo rango de variación de valores, mediana, media, moda, etc.)**
- **Porcentajes tomados con respecto al tamaño, total de responsabilidades públicas[93] y esfuerzo[94].**

[93] El total de responsabilidades públicas mide los servicios brindados por la clase.
[94] Medido como el producto entre responsabilidades públicas y mediana de statement.

Se sugiere la siguiente planilla de trabajo.

Nombre del producto:

Cantidad de clases desarrolladas
Cantidad de Interfases desarrolladas
Porcentaje de cantidad de Interfases desarrolladas sobre cantidad de clases

Reutilización
Cantidad de Clases externas especializadas
Cantidad de interfases externas extendidas
Cantidad de colaboradores externos
Cantidad de clases que implementan interfases externas
Porcentaje de reutilización sobre el total de clases desarrolladas

Herencia

Cantidad de jerarquías de clases desarrolladas
Rango de niveles de especialización por jerarquía de clases desarrolladas
Porcentaje de jerarquías sobre clases desarrolladas
Rango de porcentaje de métodos reemplazados en una jerarquía
Cantidad de clases raíz no abstractas
Cantidad de clases raíz no abstractas que implementan Interfases

Cantidad de jerarquías extendidas de clases externas
Rango de niveles de niveles agregados a jerarquías donde la raíz es externa
Rango de porcentaje de métodos reemplazados en jerarquías donde la raíz es externa

Cantidad de jerarquías de interfases desarrolladas
Rango de niveles de especialización por jerarquías de interfases desarrolladas
Rango de métodos de interfase
Porcentaje de jerarquías de interfases sobre cantidad de interfases

Polimorfismo
Cantidad de mensajes polimórficos enviados

Granularidad de métodos
Mediana de statements por método por clase
Rango de promedio de statements por método por clase

Colaboración – Agregación
Rango de colaboradores por clase
Mediana de colaboradores por clase

Porcentaje de clases que no tienen colaboradores. **Responsabilidades públicas** **Mediana de métodos de interfase por clase** **Rango de métodos de interfase por clase**
Total de responsabilidades públicas
Esfuerzo (responsabilidades públicas x mediana de statements)

5.2 Modo de uso de las métricas.

5.2.1 Consideraciones generales.

La valoración de los valores obtenidos se hace teniendo en cuenta un marco de referencia. El primer marco de referencia que se construye es el resultado de las mediciones de la librería de java y javax[95] cuyos valores se muestran en el anexo. Este marco, aunque es limitado, da una idea de las magnitudes esperadas.

La compresión de los valores crece con el mayor uso de las métricas. La realización de mediciones y su análisis, va armando un marco de referencia más amplio, necesario para la adecuada caracterización de los productos. Evidentemente, el tipo de producto -- o sea, "qué estoy midiendo"- sirve para comprender algunos valores reportados. La experiencia acumulada en las mediciones realizadas dan mayor nitidez y comprensión a los valores obtenidos.

Para hacer una comparación realista de productos es necesario comparar productos con funcionalidades similares. En general, cada dominio tiene valores límites y rangos de valores diferentes, pero la experiencia obtenida en las mediciones en otros dominios siempre aporta información.

La riqueza del planteo está en el conjunto básico planteado. Se debe tomar como un conjunto de medidas, donde los valores se comprenden en su conjunto. En forma individual, es muy pobre la información que brinda una métrica. No se puede afirmar que una métrica es más importante que otra, o que una suela dar más información que otra. De acuerdo con las características del producto que se esta analizando, sí se puede afirmar que para un producto en particular existe un subconjunto de métricas que caracterizan a un producto o que brindan más información sobre el producto. En general, todas las métricas han dado información para algún producto determinado, o han potenciado la apreciación de una característica puntualizada por otra métrica.

Dado que el planteo inicial busca encontrar métricas que midan las buenas prácticas de diseño, se agrupan las métricas por aspectos del diseño orientado a objetos. Estos aspectos son herencia, tamaño, reutilización, granularidad de métodos, responsabilidades públicas, colaboración, polimorfsmo.

[95] Se midieron todos los paquetes de java, javax.sql, javax-swing, javax sin sql y swing, org de Java 2 Platform, Standard Edition, version 1.4.1.

Se trata de caracterizar un producto, destacando cómo se han aplicado las buenas prácticas de diseño. Los valores se analizan mirando valores extremos, ya sea los extremos de los intervalos, o buscando valores que ya sean muy bajos o muy altos.

Las métricas dan un conocimiento rápido del producto, brindando valores con un grado de exactitud que sería imposible de determinar sin el uso del programa que mide en forma automática el producto, y un conjunto de alarmas que deben ser analizadas con mayor profundidad. Normalmente, el estudio de las alarmas lleva a una comprensión rápida del producto. Las alarmas suelen apuntar a las clases principales de la aplicación.

5.2.2 Experiencia en el uso de las métricas.

Se cuenta con un programa (cfr. Anexo) que emite una serie de valores por cada métrica. Dichos valores pueden ser: un valor único para cada clase[96] del producto a medir, o bien se obtiene un solo valor como resultado y un detalle esclarecedor de este valor[97].

Para facilitar el análisis del conjunto de datos obtenidos por el programa, se arma una planilla resumen y varias planillas de análisis de las series de valores más complejas. Se utiliza la planilla de cálculo Excel. Se importan los valores obtenidos como resultado de la medición automática a esta planilla. El libro se arma de la siguiente forma:

1. **En la primera hoja de cálculo se arma el resumen (cfr. Planilla de Trabajo del pto. 5.2.1).**
2. **Se copian en hojas de cálculo sucesivas las series de las métricas Colaboradores por clase, Métodos de interfase por clase, Métodos por interfase, Niveles de Jerarquías, Promedios de statement por clase. Con estos valores se calculan valores estadísticos y los gráficos de las series. Se usan las opciones:**
   - **Herramientas, Análisis de datos, Estadísticas Descriptivas, Resumen de Estadísticas,**
   - **Herramientas, Análisis de datos, Histogramas[98].**
3. **En la última hoja de cálculo se importan los valores obtenidos por el programa que mide en forma automática el código.**

Tamaño

El tamaño es el valor de referencia más importante. Muchos de los valores se pesan en relación a este dato. Por eso la cantidad de clases e interfases son los primeros valores que se individualizan.

Reutilización

Se consideran la reutilización de colaboradores externos y la reutilización lograda aplicando el paradigma de herencia. Se observan cantidades. El cálculo del Porcentaje de Reutilización sobre el total de clases desarrolladas, da un valor útil para sopesar la reutilización del paquete y compararlo con diferentes paquetes. Se considera que un valor menor al 30% es bajo y un valor mayor al 60% es alto.

Herencia

---

[96] Por ejemplo, en Cantidad de colaboradores internos por clase, para cada clase se calcula la cantidad de colaboradores
[97] Por ejemplo, en Cantidad de clases, se calcula el total de clases medidas y se detallan los nombres de todas las clases
[98] Está previsto incorporar en la nueva versión del producto la realización automática de esta planilla.

Se analizan las jerarquías propias, considerando la cantidad de jerarquías, los niveles de especialización por jerarquía, las clases raíz, y la sobre-escritura. Con respecto a la cantidad de jerarquías, se tiene como valor de referencia para el Porcentaje de jerarquías sobre clases desarrolladas el 10%. Un 10% en este item indica que el diseño del producto se apoya en el desarrollo de jerarquías propias.

Con respecto a los niveles de especialización, niveles mayores a tres suelen plantear desarrollos de cierta complejidad. Como valores extremos máximos se tienen[99]:

- **java.awt, con 259 clases, 4 niveles**
- **javax.swing, con 482 clases y 6 niveles.**

Se miran las clases raíz que no sean abstractas ni implementen interfases. Se analizan los extremos del Rango de Porcentaje de Métodos Reemplazados en Jerarquías. Se miran las clases que tengan un valor mayor a 30%.

Se analizan las jerarquías de interfases, niveles mayores a tres suelen plantear desarrollo de cierta complejidad. Como valores extremos máximos se tienen[100]:

- **java.awt, con 95 interfases, 4 niveles**
- **javax.swing, con 76 interfases y 2 niveles.**

Se consideran las jerarquías de clases donde la raíz es externa[101]. Suelen ser muy puntuales. Se centra el análisis en las jerarquías principales de la aplicación. Se analizan los extremos del Rango de Porcentaje de Métodos Reemplazados en Jerarquías donde la raíz es externa. Se miran las clases que tengan un valor mayor a 30%.

Polimorfismo

Se comparan los valores obtenidos teniendo en cuenta la cantidad de clases, jerarquías y de interfases. Se compara un producto con respecto a otro, teniendo en cuenta los items mencionados.

Clases Principales

En las métricas de Granularidad de métodos, Colaboradores por clase y Responsabilidades públicas, se miran valores extremos. Estos suelen apuntar a clases principales de la aplicación, hecho que agiliza el conocimiento del producto. Además refuerzan o completan las conclusiones que se van evidenciando con las métricas anteriores. Los valores máximos de estas métricas son:

- **Ganularidad: valores cercanos a 50**
- **Colaboradores: valores cercanos a 10**
- **Responsabilidades: difieren mucho con respecto al tipo de aplicación, por ejemplo:**
  java.awt, con 259 clases, [0 - 278]
  java.security, con 111 clases, [0 - 49]

Las mediciones estadísticas que resultan más útiles son los rangos y los porcentajes donde se usa como referencia valores de tamaño. En el caso de la media, mediana y

[99] Se toman estos valores de las librerías de java porque las mediciones se realizan con paquetes que desarrollan clases para la interfase gráfica.
[100] Se toman estos valores de las librerías de java porque las mediciones se realizan con paquetes que desarrollan clases para la interfase gráfica.
[101] Se tienen en cuenta aquellas clases externas que generan una jerarquía en la aplicación, esto es cuando dentro del alcance de medición existe más de un nivel.

moda, suelen ser válidos, porque refuerzan las conclusiones a las que se van arribando. En cuanto a los gráficos estadísticos, dan una visión rápida de la muestra.

Es posible plantear otras métricas basándonos en información estadística o mediciones básicas. Un ejemplo es el planteo de esfuerzo como el producto de la cantidad total de responsabilidades públicas por la mediana de statements.

La conclusión a la que se arriba suele basarse en los valores que llaman la atención por ser muy bajos, o bien muy altos.

5.3 Campo de aplicación de los resultados

A

- **los líderes de proyecto,**
- **las personas que deben realizar un juicio sobre la aplicación de las buenas prácticas de diseño orientado a objetos en productos no desarrollados por la persona que emite el juicio,**
- **las personas que deben seleccionar productos de terceros,**

les permite:

- **contar con una herramienta que facilita un conocimiento profundo y rápido del producto,**
- **cuantificar valores, lo que es muy difícil de hacer sin una herramienta automatizada,**
- **obtener una visión global del producto ,**
- **caracterizar el producto,**
- **determinar alarmas para detectar el no uso de las buenas prácticas**

A los constructores, les ayuda a:

- **cuantificar valores, lo que es muy difícil de hacer sin una herramienta automatizada,**
- **determinar alarmas para detectar el no uso de las buenas prácticas**

ANEXOS

Anexo I - Piccolo-Jazz13-Jazz05

Son tres versiones de un mismo producto. Piccolo es la última versión. Es un replanteo de Jazz. Jazz13 es una evolución natural de Jazz05.

Los tres productos proveen soporte para desarrolladores en 2D para uso del zoom en aplicaciones gráficas en Java. Usa el modelo de "scenegraph", que es común en los ambientes de 3D. Básicamente, esto significa que Piccolo tiene una estructura jerárquica de objetos y cámaras. Cada nodo en la jerarquía tiene un 2D AffineTransform que permite a cada objeto trasladarse arbitrariamente, variar la escala de la imagen o rotar. Además las cámaras tienen un AffineTRansform que especifica la vista dentro de la escena. Este enfoque simple es muy poderoso porque permite un punto de vista orientado a objetos desacoplado, para aplicaciones gráficas.

Cada versión define diferentes componentes visuales básicas, y las aplicaciones pueden extender estas componentes o definir las propias. Soportan objetos dinámicos que responden a diversos contextos, como ser la vista actual de la cámara. El modelo de la cámara soporta múltiples vistas, sobrevistas o vistas embebidas, donde la cámara mira a otra cámara dentro de la escena.

Piccolo[102]

Piccolo es un framework para construir aplicaciones que soportan el uso del zoom. Los elementos pueden ser agrandados o achicados, pueden reaccionar a las entradas del usuario y tener animación. Estas clases pueden definir una aplicación en forma completa o puede ser una parte de un sistema mayor.

La figura anterior muestra una secuencia de Frames de una interfase de zooming creada usando Piccolo. Este ejemplo muestra los aspectos fundamentales de una aplicación de Piccolo. En esta secuencia el usuario esta agrandando las palabras "Hello world!", un objeto Ptext.

Los objetos sobre el "canvas" son nodos (instancias de Pnode), en este caso un nodo Ptext. Los nodos son usados para representar las componentes de una interfase.

El framework tiene algunos nodos más requeridos, como ser (shapes, images, and text). El desarrollador tiene la oportunidad de definir nuevos nodos para sus propias interfases.

---

[102] http://www.cs.umd.edu/hcil/jazz/download/piccolo/PiccoloPatterns.html
http://www.cs.umd.edu/hcil/jazz/applications/
http://www.cs.umd.edu/hcil/jazz/download/piccolo/piccolodifference.shtml

En la figura que muestra el zooming, la interacción con el usuario ha sido creada con la ayuda de los event listener, instancias de PInputEventListener. Event listeners definen la forma en que la interfase reacciona a los eventos de entrada provocados por el usuario. En esta interacción de uso del zoom, el event listener reacciona a la entrada del mouse manipulando la transformación de la vista de la cámara, para crear el efecto de aumento o disminución de tamaño.

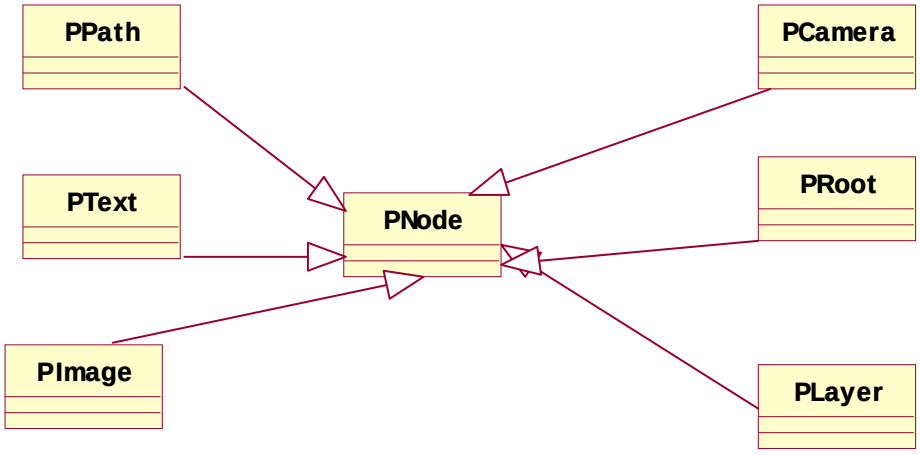
Piccolo tiene implementado eventListeners para aumento y disminución de tamaño para nodos camara, y nodos dragging en el canvas.

Todos los objetos que se muestran deben ser ubicados en un Pcanvas, de tal forma que pueden ser vistos y manipulados por el usuario. El canvas es una subclase de Jcomponent de tal forma que es fácil intergrarlas con aplicaciones de Java Swing. El canvas además mantiene una cámara y esta a su vez un nodo layer. El aumento o disminución de tamaño se logra realizar en Piccolo al trasladar y variar la escala de la transformación de la vista de la cámara en el tiempo. Nuevos nodos son agregados al nodo layer de cámara.

Piccolo trae un enfoque 2D diferente pero más tradicional. No hay separación entre los Snode y los ZvisulComponents, como se plantea en versiones anteriores. Por lo tanto, por ejemplo el rectángulo extiende directamente de Pnode, y puede ser ubicado directamente en el gráfico de la escena sin necesidad de ser envuelto ("wrapped") en un objeto tipo ZvisualLeaf. Piccolo ha construido muchos de los decoradores usados en Jazz_05 o los ZGroup de Jazz_13, en la clase Pnode. Por ejemplo, cualquier nodo tiene una AffineTransform, soporta hijos, y tiene las propiedades de transparencias.

Breve descripción de las clases principales.

Piccolo es un framework de manipulación directa de gráficos, que soporta la construcción de interfases de zoom. El diseño del Framework toma muchos elementos del diseño de las interfases de Jazz.[103]



[103] También toma elementos de "Morphic interfase frameworks".

Pnode
Es el concepto de diseño central en Piccolo. Cualquier objeto que quiere dibujarse en la pantalla debe heredar de esta clase. Además de dibujarse en la pantalla todos los nodos pueden tener otros nodos hijos. Estructuras visuales son construidas agrupando y sub-agrupando colecciones de nodos. Cada nodo tiene su propio AffineTransform, que es aplicado antes de que el nodo es dibujado en la pantalla. Esta transformación puede ser modificada a una determinada escala o puede ser trasladada. Esta transformación existe directamente encima del nodo, pero bajo los padres del nodo. Trasladar un nodo es trasladar el nodo y todos sus descendientes, pero no traslada los padres del nodo trasladado.

Pcamara
Son nodos que tiene una adicional viewTransform y una colección de capas más la colección de hijos que heredan del Pnode. Representa una vista sobre una lista de nodos Layer. La view transform es aplicada antes que se dibujen o se tomen las capas, pero no cuando se dibuja o se toma la cámara de los nodos hijos. Una cámara, que no es interna, puede también referenciar a un Pcanvas y enviar eventos de re-dibujo al canvas. El canvas podría preguntar a la cámara la posibilidad de dibujar las regiones dañadas de su superficie.

Player
Son nodos que pueden ser vistos por una o más cámaras. Mantienen una lista de cámaras que los están mirando, y notifican estas cámaras cuando son re-pintados.

Proot
Sirve como el nodo de tope más alto de la estructura de tiempo de ejecución de Piccolo. Todos los nodos son sus hijos o descendientes de sus hijos. El Pcanvas se comunica con el nodo raíz para manejar las novedades de la pantalla y disparar los eventos a sus hijos.

Pcanvas
Es una Jcomponent que es usada para ver un gráfico de escena en una aplicación de Swing. El Pcanvas ve el gráfico de la escena a través de la Pcamara. El envía los eventos de entrada de swing a la camara, y usa la cámara para dibujarse. Trasladando y escalando la view Transform de la cámara se logra el efecto de zoom.

Jazz13

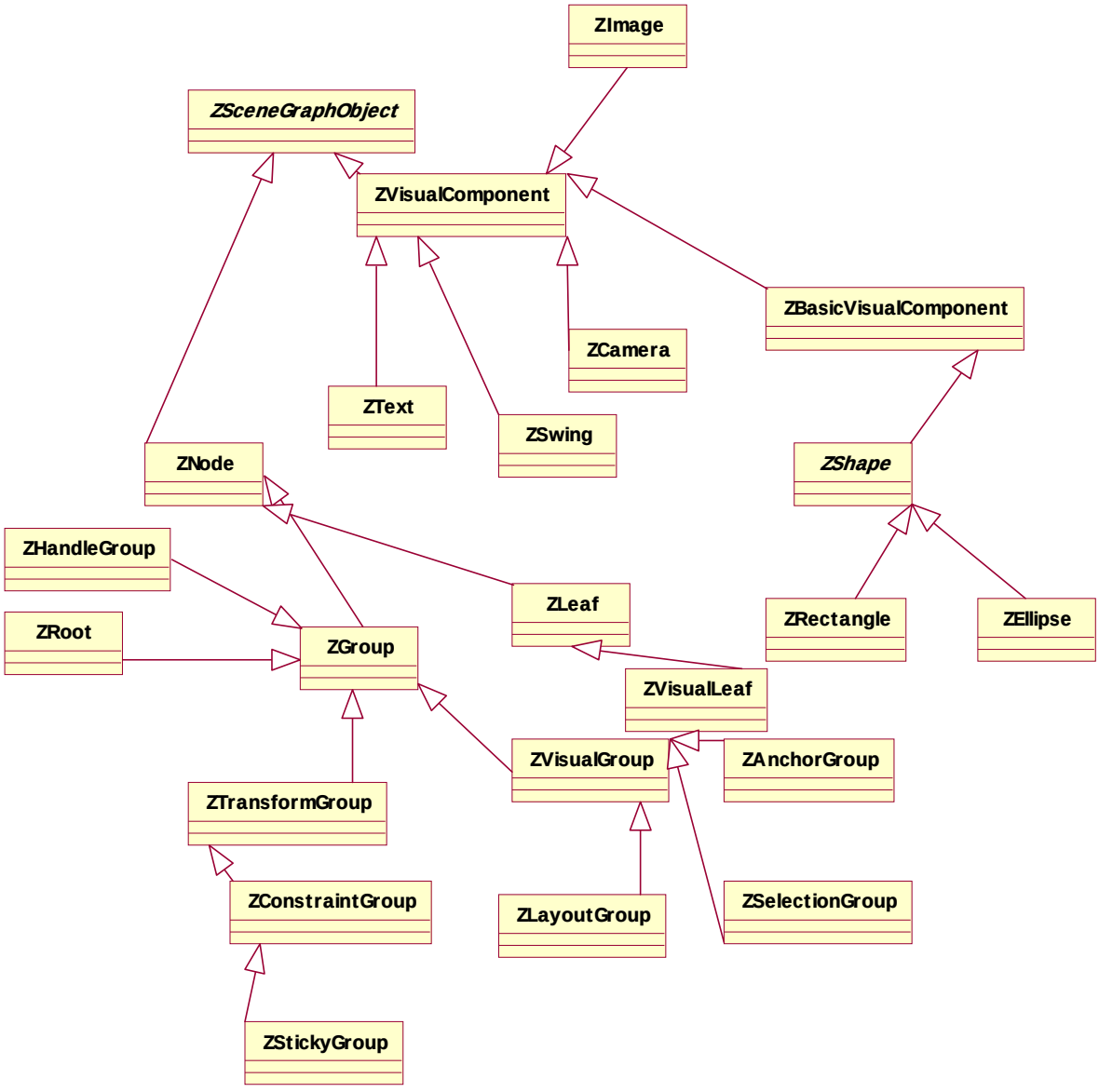
Los objetos que están en un canvas son componentes visuales, por ejemplo un Ztext. El framework tiene numerosas componentes visuales como ser, figuras (rectángulos, líneas, path, etc.), imágenes, textos o Swing, que es un wrapper usado para agregar componentes Swing a un ZCanvas. Además tiene padres, que es una ZlistNodes. Los ZGroups son Znodes con hijos.

Un efecto zoom, solicitado por el usuario, se realiza con la ayuda de un EventListener, implementado en ZoomEventhandler. Los EventListener definen la forma en que reacciona la interfase a la acción del usuario. En una interacción de zoom los ZoomEventhandler proveen los manejadores de eventos para el zoom básico de una cámara de Jazz, presionando el botón derecho.

Jazz_13 tiene implementados los EventListener para zooming, panning y selection en el canvas.

Todas las componentes visuales deben estar en un canvas, de tal forma que pueden ser vistas y el usuario puede interactuar. El canvas es una subclase de Jcomponent de tal forma que puede ser integrada fácilmente con Aplicaciones de Java Swing. El canvas tiene una cámara y una layer. ZlayerGroup, un único nodo que la cámara mira por encima, es considerado una capa porque otras aplicaciones pueden poner algún contenido debajo de este nodo que puede ser oculto o mostrado como una capa. Se logra el aumento o disminución de tamaño en Jazz_13, escalando o trasladando la transformación de la vista de la cámara en el tiempo.

Breve descripción de las clases principales.



SceneGraphObject, es una clase abstracta base de todos los objetos de Jazz. Provee los métodos básicos comunes a los nodos y a las componentes visuales.

Znode es la superclase común a todos los objetos en Jazz. Tiene una funcionalidad muy limitada, existe primariamente para soportar sub-clases.

ZGroup es un nodo con hijos. Las aplicaciones pueden usar Zgroup para agrupar hijos. Los grupos son típicamente usados cuando varios objetos deben ser tratados como una unidad semántica.

ZvisualComponent, es la clase base de los objetos que son ejecutados. Primariamente implementa tres métodos: paint(), pick(), and computeBounds(). Si es conveniente nuevas sub-clases van a sobrescribir pick.

Zcanvas es una simple componente de Swing, extiende a Jcomponent y sobrescribe los métodos apropiados de tal forma que cada vez que Java solicita que este widget sea re-dibujado, el requerimiento es enviado a Jazz para que sea ejecutado apropiadamente. También define un scenegraph simple que consiste en una raíz, una cámara y un nodo. Soporta la captura de la vista de la cámara sobre una Imagen.

ZCamera representa una vista sobre una lista de nodos. Una cámara puede mirar a cualquier número de ZlayerGroups, y puede especificar el lugar en el espacio que esta mirando, establecido por un arbitrario "affine transformation". Cuando un Zcanvas es creado, el crea automáticamente una cámara de "top-level", de igual tamaño que el canvas, adosada al canvas. Por lo tanto todo el escenegraph es ejecutado con el canvas. También es posible crear una cámara interna que actúa como un portal, o ventana interna; es un objeto dentro de la escena que mira sobre la escena.

Zroot
Extiende de Znode sobreescribiendo varios métodos para asegurarse que Zroot esta en la posición raíz del scenegraph. Todos los scenegraph comienzan con un Zroot que sirve como raíz del árbol del "scenegraph tree". Cada scenegraph tiene exactamente una raíz.

Jazz05

Jazz05 es un framework para construir clases que soportan zoom. Los elementos pueden ser agrandados o achicados, pueden reaccionar a las entradas del usuario, tiene animación. Estas clases pueden definir una aplicación en forma completa o puede ser una parte de un sistema mayor.

La clase principal es el Pnodo. El Pnode puede exhibir una apariencia visual si contiene un ZvisualComponent. El framework tiene algunas clases más requeridas como ser: ZRectagle, Ztext y Zimage, que son subclases de ZvisualComponent. Estas componentes visuales son decoradas con cadenas de propiedades como ser ZselecctionDecotaror o el ConstraintDecorator. El ZselecctionDecotaror muestra que ha sido seleccionado remarcando sus hijos con una línea de 1 pixel. El ConstraintDecorator aplica una simple restricción al hijo que modifica la transformación, controlando efectivamente el lugar que ocupar en el espacio.

La clase que esta interesada en procesar los Eventos de Znode puede implementar la interfase ZnodeListener o extender la clase ZnodeAdapter, implementando sus métodos porque están vacíos. Jazz-05 tiene, un ZoomEventhandlerRightButton que provee manejadores de eventos para zoom básico de una Camara de Jazz, presionando el botón derecho y un ZPanEventHandler para panning básico de la cámara, presionando el botón izquierdo, el ZselectionEventHandler para manejar eventos de la selección básica, y el ZlinkEventHandler para interactivamente crear hyperlinks.

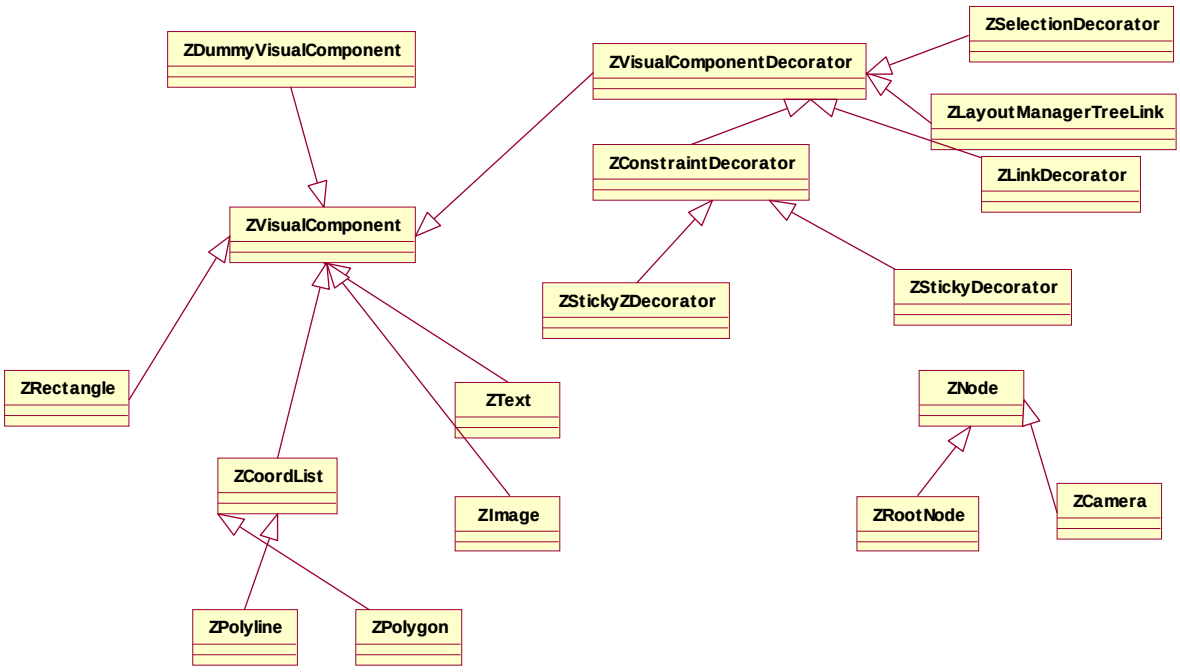
Todos los componentes visuales deben ser ubicados en un ZbasicFrame que es una subclase de Jframe, que es la base de aplicaciones simples y puede ser fácilmente integrado a aplicaciones de Java Swing.

Escalando y trasladando la viewTransform de la cámara se logra el aumento y disminución de tamaño.

Al nodo se le pueden agregar cámaras, hijos, se puede duplicar, macar alguna poción del nodo a ser re-pintada. Con la ayuda del LayoutManager muestras el árbol de hijos. Pinta la componente visual asociada y los hijos de este nodo.

El Znode tiene un objeto ZTransform tal que la transformación aplicada a este nodo es transmitida a todos los hijos.

Breve descripción de las clases principales.



Znode
Es la base de la estructura jerárquica. Puede tener cero o más nodos hijos. Puede exhibir una apariencia visual si contiene un ZVisualComponent en el slot visualComponent. Tiene todas las cámaras que explícitamente miran al scenegraph enraizado en este node. Otras cámaras pueden ver este nodo indirectamente, algún ancestro puede tener una camara en su vector.

ZvisualComponent
Permite la visualización de la componente. Tiene hijos que son las clases más típicamente usadas, como ser Zrectagule, Ztext, Zimage.

ZvisualComponent
Decora los ZvisualComponent.

Zcamera
Camaras son nodos que tienen una Afine Transform adicional y una colección de hijos que hereda del Pnode.
Representa una visión por encima del Nodo. Una cámara puede mirar hacia cualquier lado por encima de scenegraph como es especificado en una affine transformación. Un nodo puede tener cualquier número de cámaras sobre él. Cada cámara puede estar asociada con el nivel más alto de la ventana. Se le puede agregar o borrar una porción de la scenegraph que la cámara ve. Permite focalizar el objeto desde un punto 2D o desde un rectángulo.

ZRootNode
Extiende Znode para asegurarse que el nodo permanece en la posición de raíz, al buscarlos recursivamente.

ZbasicFrame
Define un simple Frame de Swing (top-level). Es la base para aplicaciones simples y además define handlers básicos para pan/zoom.

Diferencias entre Piccolo y Jazz[104]

Piccolo y Jazz soportan las mismas características básicas. Los dos tienen estructuras gráficas, y cámaras que pueden ver sobre la estructura transformando la vista. Por lo tanto se puede construir cualquier cosa que era posible construir en Jazz con Piccolo.

- **Facilidad de uso: El objetivo de Piccolo es crear ZUI[105] que tenga las mismas características que Jazz, pero que sea más fácil de usar desde el punto de vista de un programador. Casi todas las diferencias entre Jazz y Piccolo son cambios que tienden a hacer a Piccolo más fácil de usar.**
- **Complejidad: El diseño del framework de Jazz fue modelado después del "3D scene graph API". Contiene muchos diferentes tipos de objetos, cada uno con su específica funcionalidad, y estos objetos son diseñados para funcionar juntos en tiempo de ejecución y produciendo objetos más complejos. Da al framework más flexibilidad, pero requiere muchas clases y niveles de abstracción diferentes para soportarlo lo que lo hace más difícil de comprender. Piccolo tiene un enfoque diferente y más tradicional en 2D. No hay separación entre nodos y componentes visuales. Llevó mucho tiempo el desarrollo de Piccolo, dado que fue completamente re-escrito.**
- **Velocidad/memoria: Las diferencias de velocidad entre Piccolo y Jazz no son grandes, se tarda mucho tiempo para dibujar los gráficos de 2D en Java. Piccolo tiene mejoras en el tiempo requerido para la manipulación del gráfico de escena. Piccolo usa menos memoria que Jazz, dado que en vez de necesitar tres objetos para soportar la situación normal, usa un objeto.**
- **Características soportadas: soportan básicamente las mismas características. Todo lo que tiene Jazz puede ser construido con Piccolo. Se detallan algunas características que no se han implementado aún en Piccolo:**

    - **ZAnchorGroup – usado para especificar hiperenlaces**
    - **ZClipGroup – usado para especificar un "clip" grupo alrededor de nodos hijos**

[104] http://www.cs.umd.edu/hcil/piccolo/download/piccolo/piccolodifference.shtml
[105] Zooming User Interface

- **ZConstraintGroup – usado para restringir una transformación basada en algún cálculo.**
- **ZFadeGroup - todos los nodos de Piccolo soportan la transparencia, cosa que ZFADEGROUP fue usado para lograrla. Pero ZFADEGROUP también soportaba automáticamente el cambio transparente de objetos basada en la escala en la cual estaba siendo visto.**
- **Layout managers - Piccolo no tiene con ningún layout managers, pero todos los PNODES tiene un método layoutChildren () que fácilmente es sobrescrito y será llamado automáticamente cuando el tamaño de cualquiera de los hijos cambie.**
- **ZSelectionGroup - Usado por el sistema de administración de selección del Jazz.**
- **ZSpatialIndex - Usado para la recolección eficiente y la selección objetos que no serán pintados. No teniendo este efecto la velocidad varía muy poco, ya que la mayoría enorme de tiempo es gastada pintando objetos, no recolectándolos o seleccionándolos.**
- **ZSwing – Piccolo no soporta embeber objetos Swing en el Canvas.**

- **Características no soportadas: hay algunas características que no se van a ser nunca soportadas por Piccolo. Por ejemplo: Formato de archivo de Texto. La recomendación es que cuando se necesite un formato de archivo se deberá definir su propio formato.**

Anexo II - Jgraph

JGraph[106] es un componente gráfico con los fuentes disponibles, potente, ligero en cuanto a peso. Esta documentado.

Permite crear diagramas de flujo, UML o redes con muchos nodos. Provee una visión gráfica de las relaciones, lo que permite realizar un análisis con mayor precisión al poder visualizar el trabajo existente.



Características

- **100% Java**
- **Compatibilidad total con Swing**
- **Diseño claro y eficiente**
- **Ofrece XML**
- **Arrastra y libera**
- **Copia, pega**
- **Zoom**

Comentarios

"Esto es una biblioteca de dibujo de gráfico genérica para swing y es realmente fácil de usar. Es rápido. Es bien diseñado. Es flexible. Es justo lo que el doctor ordenó. Parece ser una perla entre proyectos de software libres. "
- Dave's Blog, http://diary.recoil.org/djs/

"Uso JGRAPH muy satisfactoriamente para construir una herramienta de visualización para un laboratorio de software en MIT. [...] Esto ocurre en el laboratorio de investigación de ingeniería de software (SERL) en el departamento aero/astro del MIT. JGraph es usado para visualizar gráficos y árboles que representan la especificación formal de un sistema de software (básicamente una descripción estado finito). [...] estamos impresionados por lo fácil que es usar este software... un buen trabajo. "
- Craig Lebowitz, Massachusetts Institute of Technology

Descripción

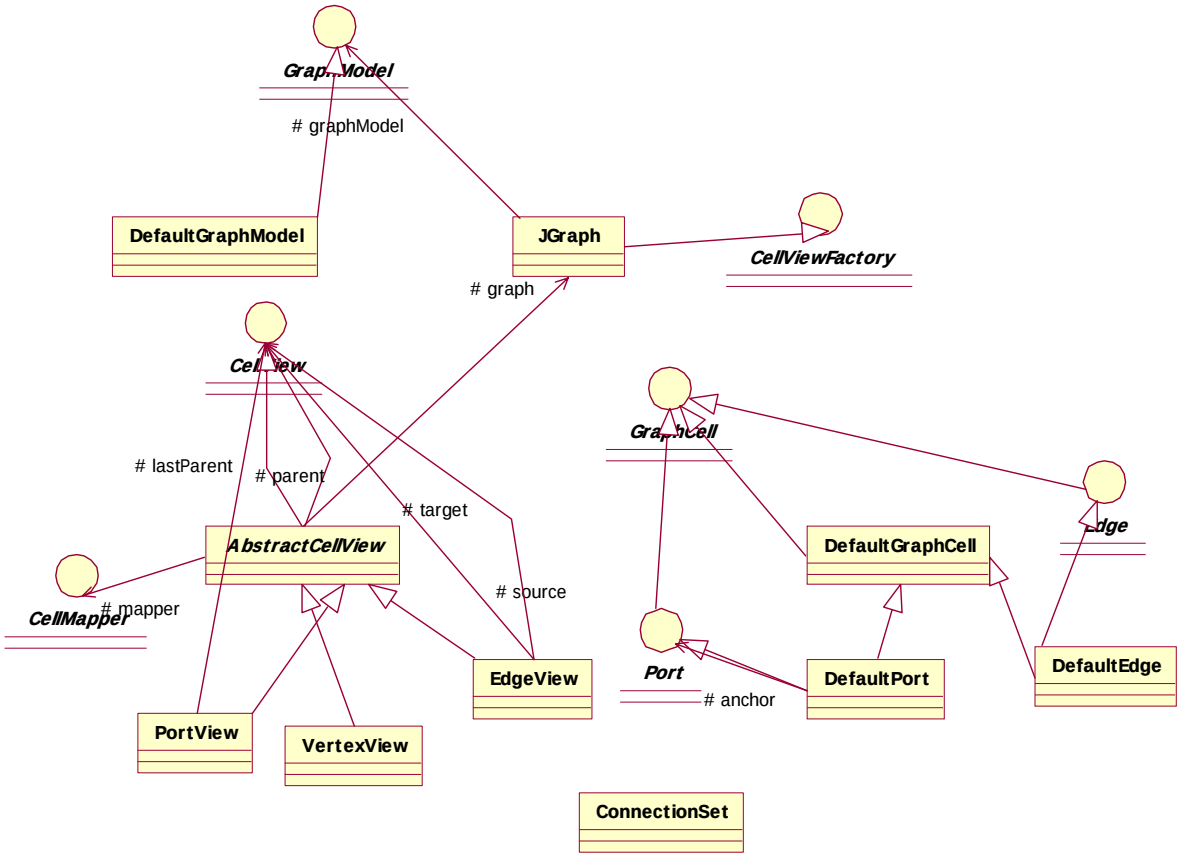
Los gráficos en Jgraph son usados como un paradigma para visualizar cualquier red de objetos relacionados, como por ejemplo: una carretera, una red de computadoras, una arquitectura de software o un esquema de una base de datos. Jgraph los dibuja y permite al usuario interactuar con ellos.

[106] http://www.jgraph.com/

Jgraph separa el modelo de la vista. El modelo es definido por la interfase GraphModel y contiene objetos que implementan la interfase GraphCell; la vista es representada por la clase GraphView class, y contiene objetos que implementan la interfase CellView. El mapeo entre celdas y vistas es definido por la interfase CellMapper.

Un modelo tiene cero o más vistas, y por cada celda en el modelo, existe exactamente un CellView en cada GraphView. El estado de estos objetos es representado por un map (key y value). Cada CellView une los atributos del correspondiente GraphCell con sus propios atributos.

En el gráfico se muestran las clases que pertenecen principales.



Jgraf

Es una clase de control que dibuja una red de objetos relacionados usando el conocido paradigma de grafo. No contiene los datos, simplemente provee una vista de los datos. Como cualquier componente de Swing no trivial, el grafo toma los datos del data model.
Jgraph muestra los datos dibujando elementos individuales. Cada elemento contiene exactamente un ítem de dato, el cual es llamado celda. Una cell pude ser tanto un vertex, un edge o a port. Los vertex tienen cero o más vecinos, y los edges tienen cero o un "source and target vertex ". Cada celda tiene cero o más hijos, y uno o más padres. Instancias de ports son siempre hijos de vértices.
Jgraf soporta edición de celdas, selección, redimensionamiento y traslado de arcos y vértices, estableciendo y removiendo conexiones. CellViewFactory, es la interfase que construye una vista para la celda especificada y la asocia con un objeto, usando un CellMapper. Setea el tamaño de la vista.

Modelo

GraphCell es una interfase que define los requerimientos para objetos GraphCells. DefaultGRaphCell es la clase que implementa por default a GraphCell. DefaultGraphCell Extiende javax.swing.tree.DefaultMutableTreeNode implementa GraphCell, Cloneable. DefaultGraphCell extiende a DefaultMutableTreeNode implementa GraphCell, Cloneable. Tiene dos subclases, DefaultEdge (tiene un source y un target) y DefaulPort (tiene edges) que son las implementaciones por default de un arco y un puerto.

Edge es una interfase que permite modificar y setear el source y el target.

Port es una interfase que permite agregar edges, remover edges, preguntar por los dejes, retorna un iterator. Los arcos no son conectados directamente al vértice source o target. Son conectados a los ports de un vértice. La relación entre un vértice y un port es de padre-hijo, comúnmente usada en nodos de un árbol.

ConnectionsSet, representa un conjunto de conexiones. Connections, Object that represents the connection between an edge and a port.

Vista

CellView, interfase que define las características de un objeto que representa la vista de un objeto del modelo. Permite realizar un refresh, update, identificar la celda correspondiente a la vista, etc. Cuando se crea una vista, JGraph asume que una celda es un vertex si no es una instancia de un Edge o Port, y llama el método createVertexView. Por lo tanto solamente se necesita sobrescribir este método para identificar n vertex oval y retornar la vista correspondiente.

AbstractCellView, es la implementación de CellView. Tiene una lista de hijos, una Jgraf, un CellMapper y un CellView padre. Permite ejecutar, escalar, trasladar, actualizar una CellView. Tiene los siguientes métodos abstractos: GetBounds(), getRender(), getHandler(). PotView, tiene un CellView "last parent".

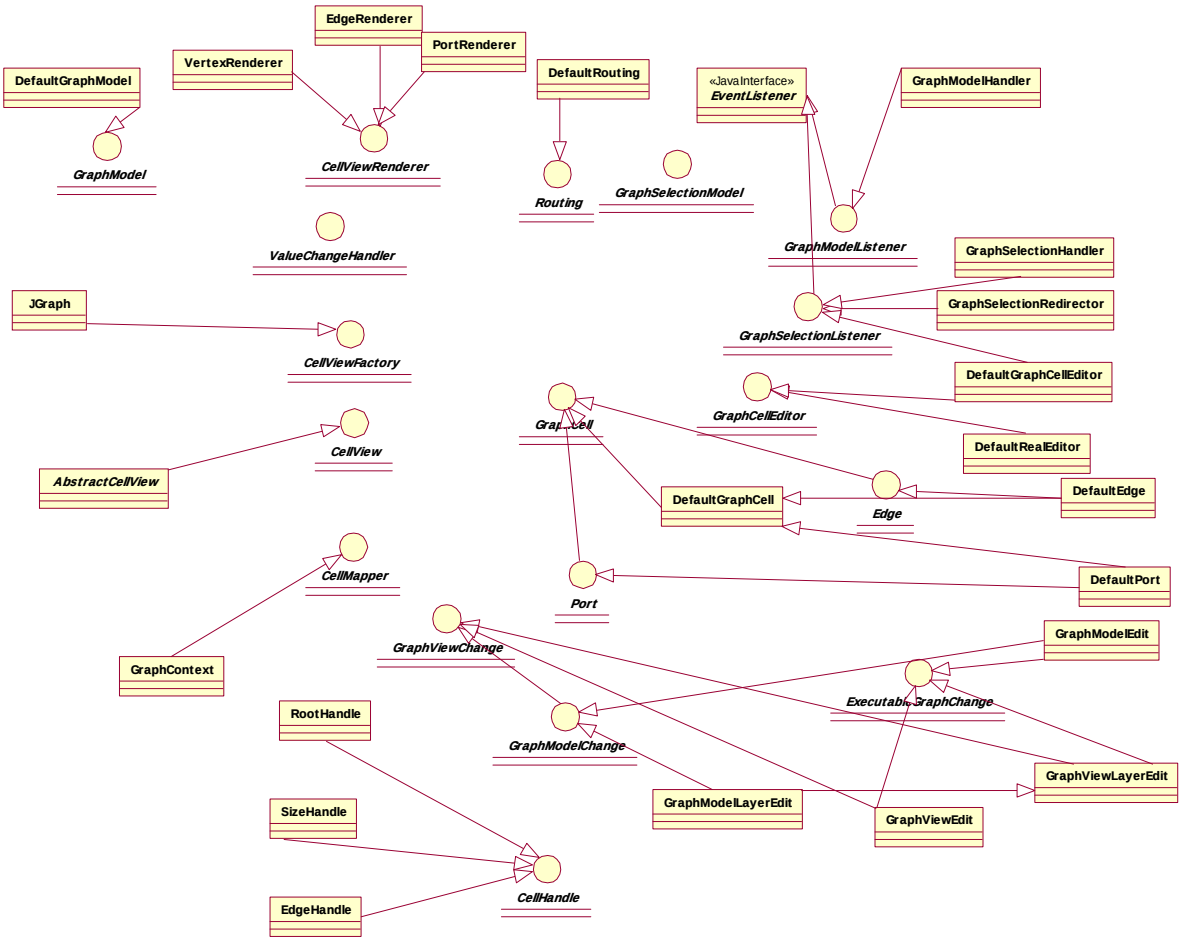
EdgeView, tiene un CellView source y target.

Paquetes

Event, contiene clases event e interfases de los listener que son usados para reaccionar a eventos disparados por JGraph.
Jgraph, contiene la clase Jgraph, que es el corazón de la aplicación.
Plaf, contiene la clase GraphUI la cual extiende de la clase ComponentUI de Swing.
Basic, contiene a BasicGraphUI, la cual es la implementación por default de GraphUI's.
Graph, el paquete jgraph.graph provee soporte para graph model, graph cells, graph cell editors, drivers, controladores, y ejecutadores.

Framework orientado a interfases

Se percibe en el diagrama que el planteo del producto parte de la conceptualización de una serie de interfases. Algunas de ellas son interfaces de clases principales, como ser GraphicCell, CellView, Edge, Port, CellMapper y CellHandle.


EdgeRenderer
PortRenderer
VertexRenderer
DefaultGraphModel
DefaultRouting
«JavaInterface»
EventListener
GraphModelHandler
GraphModel
CellViewRenderer
Routing
GraphSelectionModel
ValueChangeHandler
GraphModelListener
GraphSelectionHandler
JGraph
GraphSelectionRedirector
GraphSelectionListener
CellViewFactory
DefaultGraphCellEditor
GraphCellEditor
CellView
DefaultRealEditor
AbstractCellView
DefaultGraphCell
DefaultEdge
Edge
CellMapper
DefaultPort
Port
GraphViewChange
GraphModelEdit
GraphContext
ExecutableGraphChange
RootHandle
GraphModelChange
GraphViewLayerEdit
GraphModelLayerEdit
GraphViewEdit
SizeHandle
CellHandle
EdgeHandle

Anexo III - Gef - Java Graph Editing Framework.

El objetivo del proyecto de Gef es construir una librería de edición gráfica, que pueda ser usada para construir diversas aplicaciones.

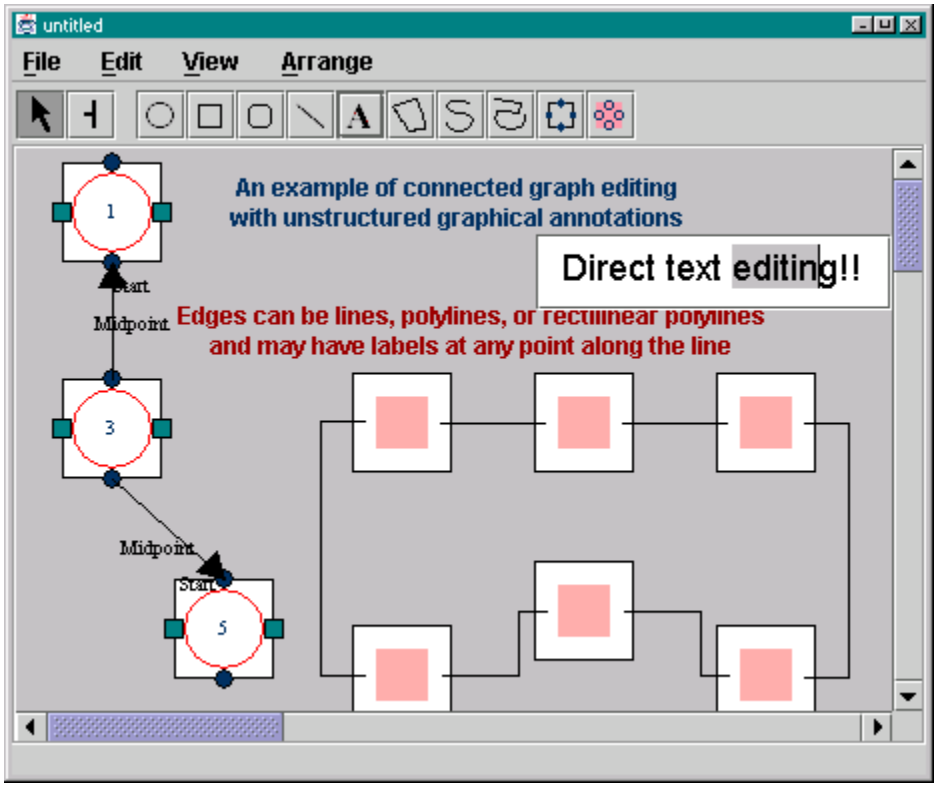
Características Generales

- **Un diseño base simple, pero que la evolución del producto da un resultado confuso. La documentación es clara, y centra la descripción en los elementos más importantes, lo que facilita la comprensión y uso del framework.**
- **Usa el modelo gráfico Node-Port-Edge que usan una importante mayoría de aplicaciones gráficas de grafos.**
- **El diseño Model-View-Controller basado en las librerías Swing Java UI hace que Gef este preparado para actuar como interfase de usuario para estructuras de datos existentes, minimizando el tiempo de aprendizaje de los desarrolladores que están familiarizados con Swing.**
- **Soporta interacción del usuario para movimiento, redimensionamiento o re-dibujo, etc. También soporta "broom alignment tool "[107] y botones de selección –acción.**
- **Soporta XML basado en el estándar PGML.**

[107] *Broom alignment tool:* Herramienta que sirve para reacomodar aquellos objetos que están en contacto entre sí. De esta manera, el usuario recibe una visión más clara del gráfico.

Figure 3. Using the broom to align and space objects.

Vista Previa



Descripción

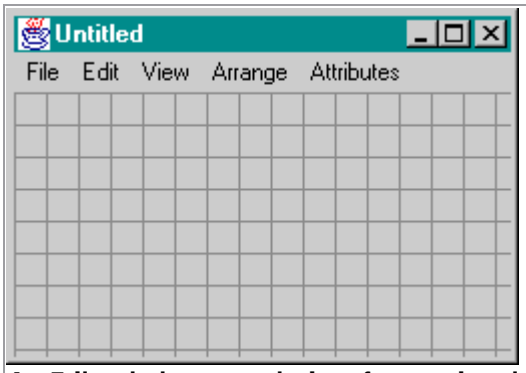
Hay dos niveles principales en Gef: el nivel de la red y el nivel del diagrama. El nivel de la red maneja nodos lógicos que pueden tener comportamiento y datos específicos de la aplicación. El nivel del diagrama presenta figuras de gráficos estructurados y desestructurados que visualmente representan la red y anotaciones varias.

"El requerimiento que nuevas características puedan ser agregadas sin modificar el código tuvo un gran impacto en el diseño. El diseño más obvio hubiera sido un editor que provee una gran cantidad de funcionalidad y puede ser extendido subclasificando. Se descartó esa posibilidad porque hace el código menos comprensible y porque la gente necesita tomar y seleccionar características, no agregar nuevas"[108]

Editor

Es la clase central del sistema. Hay una sola instancia de editor para cada diagrama mostrado en la pantalla. Actúa como un shell, no maneja los eventos de entrada, ni re-dibuja la pantalla, ni determina cuál es el ítem que el usuario a cliqueado, ni modifica la estructura de datos del diagrama. El Editor, pasa eventos y mensajes que soportan objetos a otras clases. Es el objeto raíz para las aplicaciones. Guarda la selección, ejecuta las acciones en un contexto seguro, provee menús, guarda el color de línea por default, color de rellenado, etc. Guarda la historia del undo (en un futuro).

[108] Jason Robbins



An Editor in its own window frame showing a LayerGrid and no DiagramElements. The menu is part of the editor. Editors can also exist in the web browser's window frame.

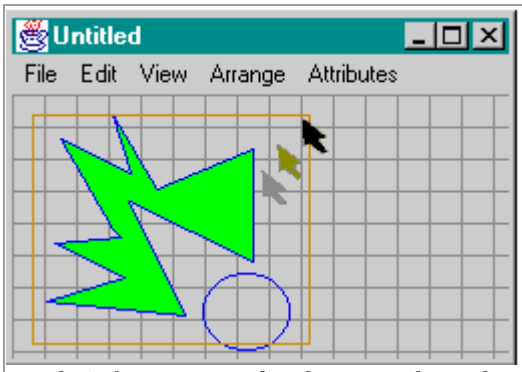
Cmd

Son clases que definen el método doit() que realiza una acción en el editor. Soporta undo. Clase abstracta que extiende de javax.swing.AbstractAction. Es una clase pensada para subclasificar los comandos del editor. Cuando se crea una instancia de Cmd se puede enviar el mensaje doIt() o undoIt(). Por ser una subclase de AbstractAction de la librería de Swing, los objetos Cmd pueden ser fácilmente agregados a los menús barra de herramientas.

Mode

Es una Interfase base para todos los modos de operación de Editor. Un modo es responsable de manejar todos los eventos que suceden en el Editor. Un modo define un contexto para interpretar los eventos. Los sistemas que usan Gef pueden definir sus propios modos subclasificando FigModifyingMode.

Interpretan la entrada del usuario, determina el próximo Mode, si es necesario. Dibuja gráficos temporales para indicar estado. ModeModify se ocupa de traslado y redimensionamiento. Subclases de ModeCreate crean nuevas figuras.



ModeSelect at work. The user has dragged the mouse, starting from blank space in the lower left to the upper right. The selection rectangle is drawn by ModeSelect.

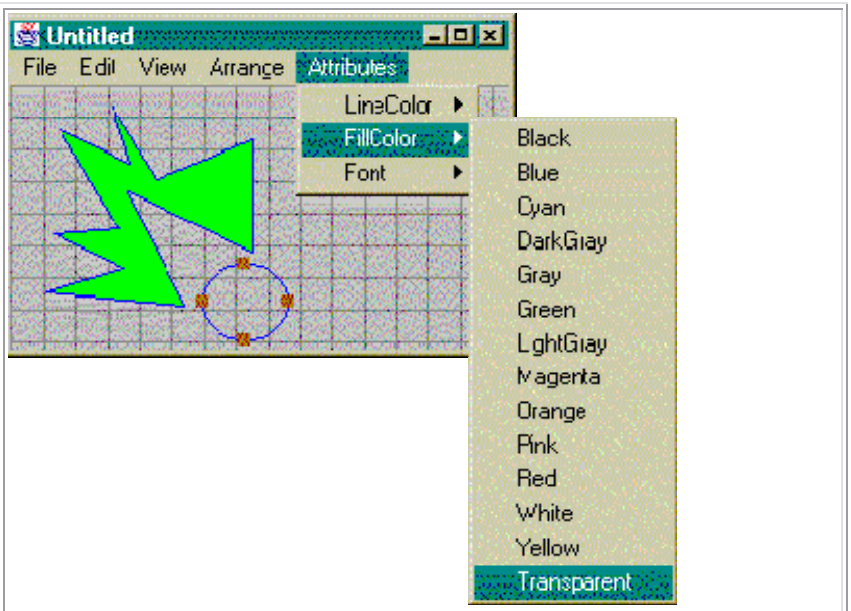
Guides

Son objetos que ayudan al usuario a hacer diagramas prolijos. Por ejemplo GuideGrid.

Fig

Son objetos dibujables que pueden ser mostrados y manipulados por un editor. Son elementos de dibujo básico como ser líneas, rectángulos, etc. FigGroups es la clase para los grupos de figuras.

Se sabe dibujar, notifica a los dependiente los cambios. Maneja algunos eventos como ser remover un vértice de un polígono. Especifica el tipo preferido de Selección. Guarda su propio color de línea, color de relleno, etc.

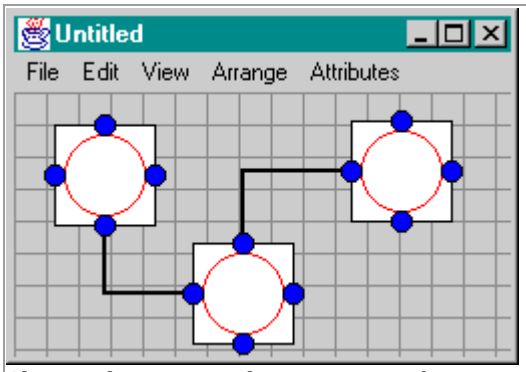


Here are a FigCircle and a FigPoly, both are subclassed from Fig.

NetPrimitive

Es la superclase abstracta de todos los objetos propios de un grafo. Provee implementación de cuatro subclases: NetNode, NetPort, NetEdge, y NetList.
DefaulGraphModel usa las subclases de NetPrimitive. Se puede además definir un GraphModel propio con las subclases de NetPrimitive también propias.

- **NetNode representa a un nodo gráfico, puede manejar eventos, chequea conecciones válidas**
- **NetPort es un puno de conexión sobre un nodo, chequea conecciones válidas**
- **NetEdge es un arco entre puertos**

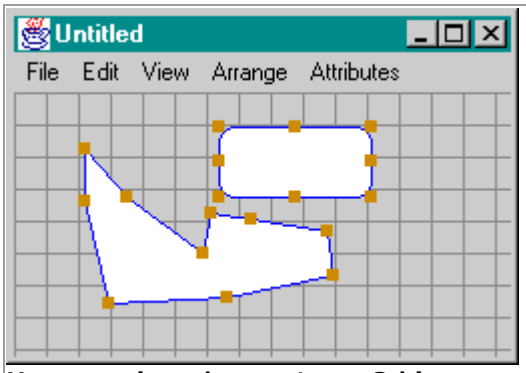


Shown here are three Perpectives and two ArcPerepectives. Not visible are the three NetNodes, twelve NetPorts, and two NetArc that represent the graph.

Layers

Sirven tanto para agrupar figuras en capas superpuestas transparentes y administrar el orden del re-dibujo, como para encontrar objetos por debajo de un punto dado por el mouse. Esto significa que sirven para mostrar y tomar listas.

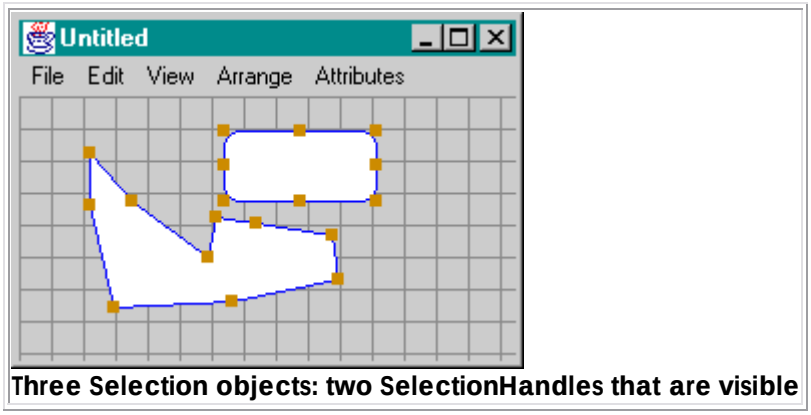
- **Dibuja objetos**
- **Notifica a los dependientes de las capas o a los Editores de los cambios.**
- **Pasa eventos a DiagramElements**
- **Mantiene el orden de los elementos del diagrama.**



Here are three layers: LayerGrid computes and draws the backgronud grid, LayerDiagram stores and draws some DiagramElements, LayerComposite contains these two Layers and maintains back-to-front ordering of sublayers.

Selecctions

Son objetos usados por el Editor cuando el usuario selecciona una figura. Son responsables de dibujar handles y manejar el arrastre sobre los handles. Pasa mensajes a la figura seleccionada.

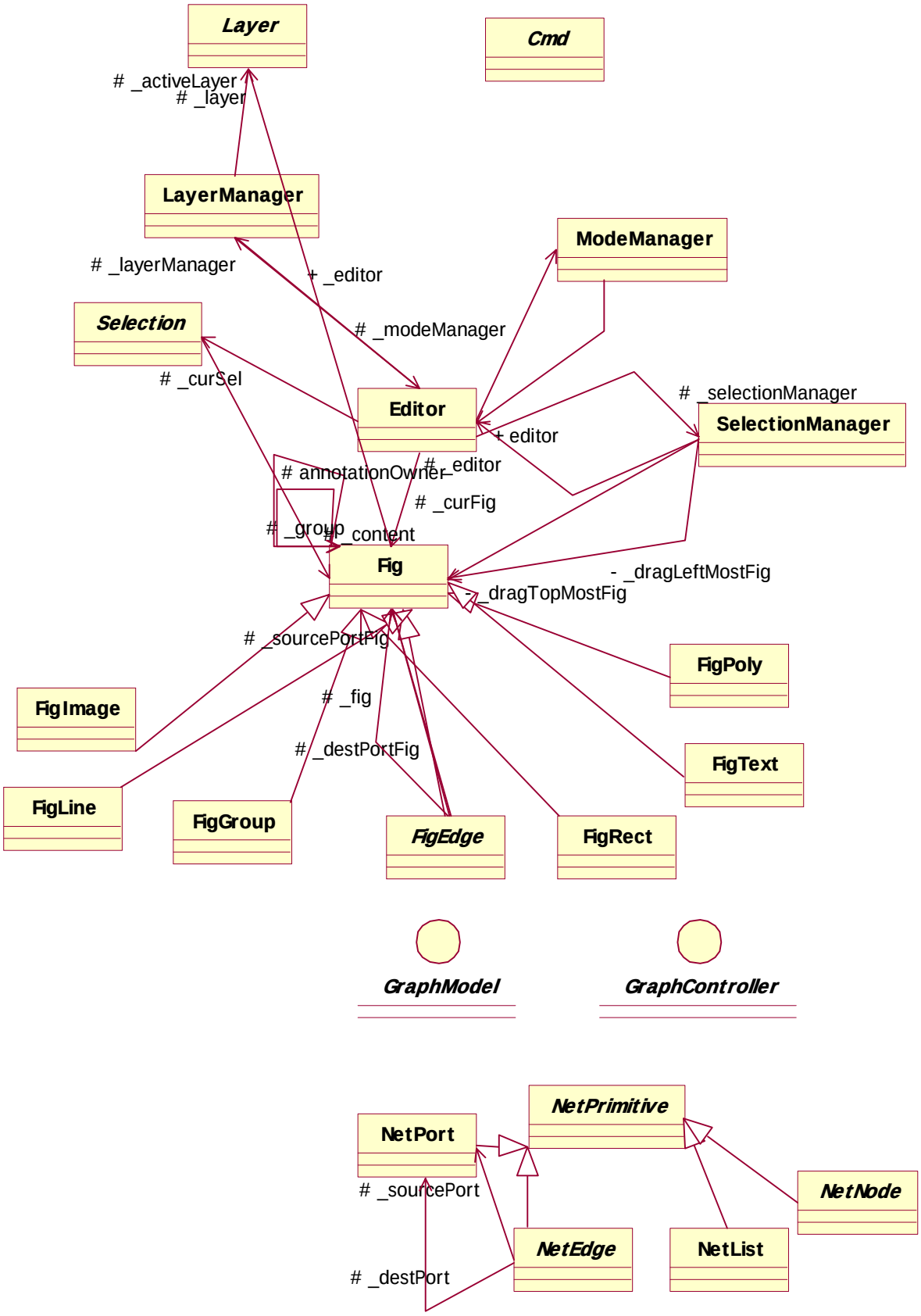


Three Selection objects: two SelectionHandles that are visible

GraphModel

Esta interfase provee un facade a una representación net-level. El programador define una clase modelo que da al widge el acceso a la aplicación de los datos. En esta forma hay una sola copia del dato, de tal forma que no hay problema con la sincronización de la presentación y dato. Esta interfase permite el uso de un objeto de la aplicación como nodo, port o edge, facilitando la visualización de una aplicación existente.

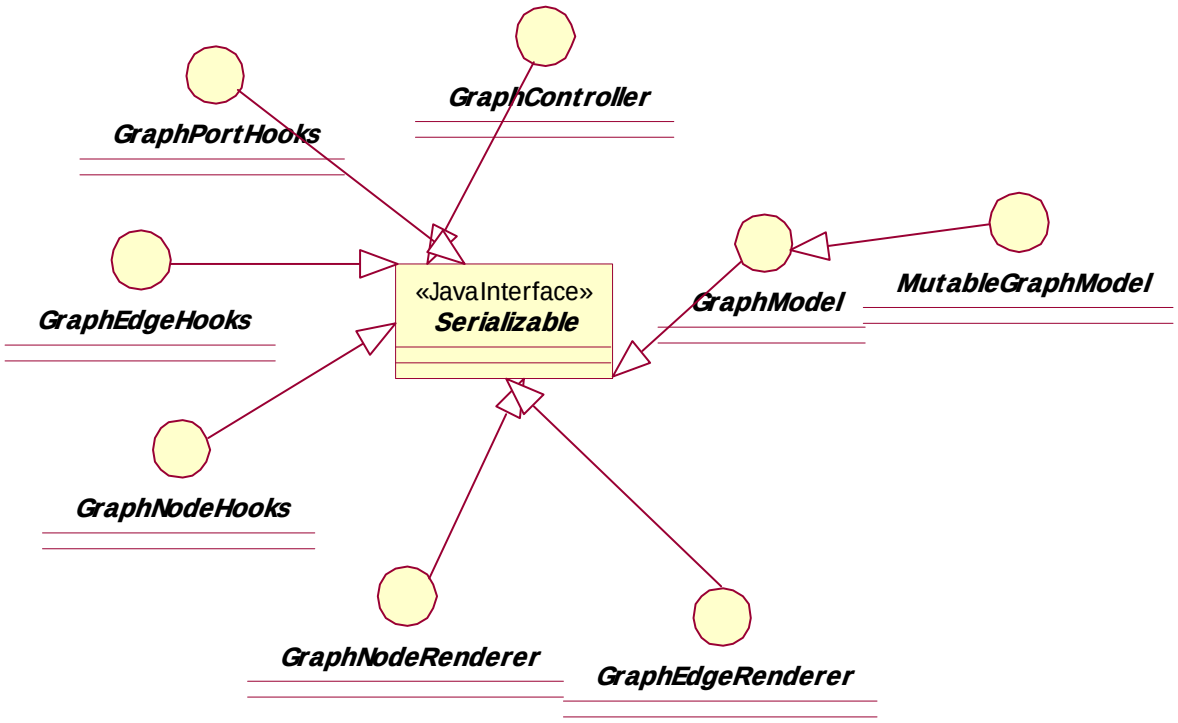
GraphController
Esta interfase es la base para cada clase que maneja el control del dato, objeto de la representación.


Layer
Cmd
_activeLayer
_layer
LayerManager
ModeManager
_layerManager
+ _editor
# _modeManager
Selection
_curSel
Editor
_selectionManager
SelectionManager
+ editor
# _editor
# annotationOwner
# _curFig
# _group
# _content
Fig
- _dragLeftMostFig
- _dragTopMostFig
_sourcePortFig
FigPoly
FigImage
_fig
_destPortFig
FigText
FigLine
FigGroup
FigEdge
FigRect
GraphModel
GraphController
NetPrimitive
NetPort
_sourcePort
NetNode
NetEdge
NetList
_destPort

Interfases.

GraphPortHooks, GraphEdgeHooks, GraphNodeHook proveen un conjunto de métodos que ports, edges o nodes podrían implementar en el GraphModel.

GraphEdgeRenderer, GraphNodeRender, una interfase para Fig Edge Factories. Similar al concepto a Swing class TreeCellRenderer.



Paquetes

La distribución y nombre de los paquetes dificulta la comprensión del producto. Por ejemplo en graph están las clases correspondientes al Model. Dentro de paquete graph existe un sub-paquete que se llama presentación y están las clases por default del Model.

Anexo IV - Descripción de las herramientas utilizadas

Se desarrolla un producto Object Oriented Metric Tool[109] (OOMTOOL) en Java[110], basado en el parser de Java Cross Referencig Tool Grammar. La herramienta permite:

- **Identificar un producto**
- **Caracterizar el producto**
- **Guarda Información sobre el producto.**
- **Identificar la ubicación física de las source class**
- **Seleccionar las métricas.**
- **Generar un archivo de texto donde están detallados los resultados de las métricas.**

El archivo de texto se importó a Excel, para facilitar el análisis de datos.

Para el análisis de los productos se usó la herramienta Rational XDE Developer for Java.

[109] Fue desarrollado por Fernando Lafalle, en el contexto de su Trabajo de Grado, de la carrera de Ingeniería en Informática de la Facultad de Ingeniería de la Universidad Austral. . Director del Trabajo: Gabriela Robiolo.
[110] JDK 1.4.0_0

Anexo V – Mediciones realizadas sobre Java – Javax

Anexo VI - Experiencias con alumnos

Se organizaron trabajos finales con tres grupos de alumnos de la materia Diseño de Sistemas[111] de la carrera de Ingeniería Informática de la Universidad Austral. Se plantearon trabajos de 20 hs. de duración. Se seleccionaron los productos, se midieron. A los alumnos se les entregaron las mediciones realizadas sobre productos. El objetivo a cumplir era constatar los valores de las métricas observando producto. A modo de guía se les armó un cuestionario, para localizarlos en las alarmas y valores más característicos de cada producto.

Durante el trabajo los alumnos intercambiaron opiniones con el docente, que aprovechaba para hacerles fijar prácticamente conceptos como, sobre-escritura, colaboración, patrones, etc. Los grupos de alumnos no habían anteriormente analizado el diseño (distribución de clases) de un producto. En general, habían trabajado con clases a nivel de código, pero no había obtenido a partir del código el modelo de clases[112], que les da un nivel de comprensión del producto más general. Fue para ellos un desafío poder sacar alguna conclusión sobre el planteo de diseño que tenía el producto y descubrir que eran capaces de abarcar un producto de más de cien clases, como es el caso de Gef y Jazz_05.

Se considera una experiencia altamente positiva para los alumnos. Se recogen algunos de los comentarios de los alumnos, con respecto a la experiencia realizada.

Piccolo

Jazz_05

"Jazz 05 presentó características normales dentro de los rangos admisibles de diseño, excepto en algunos casos, donde las métricas nos permitieron mostrar alarmas.
Algunas de ellas, como por ejemplo la cantidad de responsabilidades públicas por clase llegaban a superar las 100, o que la cantidad de métodos en interfaces fuera nula, mostraban advertencias de un posible mal uso del diseño.
A partir de estas alarmas, analizamos casos específicos que no mostraban coherencia, o en algunos casos mostraban redundancia. Por este lado nos dimos cuenta que las métricas fueron útiles, ya que permitieron una rápida focalización del problema.
La calidad del diseño en Jazz05, según las métricas, parece ser buena ya que no presenta problemas en ciertos puntos importantes como:

- **Reutilización**
- **Herencia**
- **Granularidad de Métodos**
- **Colaboraciones"[113]**

Jazz_13

"Lo que podemos deducir a través de las métricas es que este producto presenta un buen diseño si bien es muy complejo y por lo tanto difícil de entender.
La característica principal de Jazz 1.3 es su gran número de clases. A partir de las métricas se pudo centrar la atención en los puntos clave donde podría haber un error de diseño.

[111] Materia de tercer año de la carrera de Ingeniería Informática de la Universidad Austral
[112] Trabajaron con el IDEA y con una librería de clases que permite obtener el modelo de clases a partir del código.
[113] Luz Raca

Sin embargo las alarmas no fueron muchas. Existen clases con muchas responsabilidades y jerarquías con cinco niveles lo cual no es muy recomendable."[114]

Piccolo

"En base a la evolución del diseño y las pruebas realizadas considero que Piccolo es una librería que se puede adaptar fácilmente a cualquier proyecto desarrollando en Java con buenos resultados.
Creo que el manejo de la parte gráfica en general en Java es un tema crítico, y si bien no se ha podido hacer las suficientes pruebas para determinar la capacidad de Piccolo, la información que se provee en la página Web parece indicar que la performance lograda resulta bastante satisfactoria.

Las métricas permitieron:

- **Ver cómo fue evolucionando el diseño en las distintas versiones. El salto más visible es entre Jazz 1.3 y Piccolo. Esto se ve en la reducción de la cantidad de clases, jerarquías, interfaces, etc.**
- **Focalizar rápidamente posibles fallas en el diseño.**

En muchos casos la intención del diseñador no se ve reflejada en los resultados de las métricas.

La intención del diseñador intentamos comprenderla pero no contábamos con suficiente documentación / Tiempo de estudio del producto.

Los tres productos fueron analizados por un grupo de tres alumnos en 16 horas de trabajo".[115]

Jgraf

"Las métricas me ayudaron a individualizar las partes más importantes del producto. En poco tiempo me hice una idea del producto.
Además quiero destacar que lo que más me aportó fue la experiencia de enfrentarme con un producto desconocido, y poder comprenderlo en poco tiempo".[116]

Gef

"Se ha llegado a un punto en el cual la complejidad del producto requiere de un replanteo general del diseño, a fin de acercarse mas a la idea de un Framework. En una nueva versión del mismo debería ser simplificado el diseño, ya sea con más interfaces y clases abstractas, para que sea verdaderamente un *Framework*.
La calidad del producto es relativamente buena. Si bien cumple con todo lo que ofrece, su diseño no es tan simple de entender. Desde nuestro punto de vista, le falta maduración al producto.
Las métricas sin duda nos han ayudado a focalizar rápidamente posibles problemas de diseño, que de otra manera nos hubieran llevado más tiempo.

---

[114] Lucía Diez
[115] Luz Raca, Lucía Diez, Juan Chico.
[116] Pablo Castro

Si bien algunas métricas nos proporcionaron falsas alarmas, como por ejemplo en los porcentajes de sobre-escritura, generalmente nos dieron un buen panorama del diseño del producto.[117]

[117] Fernando Quintana, Mariano Carli, Ezequiel Rodriguez

BIBLIOGRAFÍA

Benlarbi, S and W, Melo , Polymorphism Measures for Early Risk Prediction, http://www.cistel.com/publications/Cistel-1999-02.pdf

Basili V.R., L.C. Briand, and W.L.Melo, A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering, 22(110): 751-761, 1996.

Briand, Lionel C., John W. Daly, and Jugen K. Wust, A Unified Framework for Coupling Measurement in Object-Oriented Systems, IEEE Transactions on software Engineering, vol. 25 No. 1, 1999.

Ceballos, Francisco Javier, Java 2, Alfaomega, México, ISBN 970-15-0604-9, 2000

Cooper, James, Design Patterns Java Companion, Addison Wesley, 1998

Gamma, Erich, Richard Helm, Ralph Jonson, John Vlissides, Design Patterns, Addoson-Wesley, ISBN O201633612, 1995

Harrison, R, S Counsell, R. Nithi, 'Experimental assessment of the effect of Inheritance on the Maintainability of Object-Oriented Systems', International Conference on Empirical AssesmentEASE'99 & Evaluation in Software Engineering, Keele University, Staffordshire, UK, 1999

Henderson-Sellers, Brian, Object-Oriented Metrics, Measures of Complexity, Prentica Hall, ISBN 0-13-239872-9, 1996.

Li 1998 a
Li, P, J, Kennedy and J, Owens . 'Assessing inheritance for the multiple descendant redefinition problem in OO systems', in *Proc. OOIS'97. 1997 International Conference on Object Oriented Information Systems*. Springer-Verlag London, London, UK, 1998.

Li 1998 b
Li, P, J, Kennedy and J, Owens . 'Mechanisms for interpretation of OO systems design metrics', in *Proc. Technology of Object-Oriented Languages. TOOLS 24*. IEEE Comput. Soc, Los Alamitos, CA, USA, 1998.

Liskov, Barbara, Data Abstraction and Hierarchy, Cambridge, http://citeseer.nj.nec.com/557944.html

Lorenz, Mark, Jeff Kidd, Object-Oriented software Metrics, Prentice-Hall, ISBN 0-13-179292-X, [1994]

Meyer, Bertrand, 1998
The role of object-orinted metrics. Una pequeña variación de este artículo fue publicado Computer (IEEE) como parte de Component and Object Technology en Noviembre 1998.

Jacobson, Ivar, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard, 1992
Object-Oriented Software Engineering, Addison-Wesley, ISBN 0-201-54435-0 (1997)

Java.Sun.com, Java 2 Platform, Standard Edition (J2SE)

Wirfs-Brock, Rebecca, Alan McKean, 2002
Object Design, Addison Wesley, ISBN 0-201-37943-0

Page-Jones, Meilir, Fundamentals of Object-Oriented Design in UML, Addison-Wesley, ISBN 020169946X, 2002

Unger, B. and L, Prechelt , The impact of inheritance depth on maintenance tasks - Detailed description and evaluation of two experimental replications. Techinical Report No. TR18/1998, Karlsruhe University, Germany1998.

GLOSARIO

Alarmas: las alarmas sirven para identificar valores que están fuera del rango considerado como probable u óptimo.

Arquitectura: describe la organización estática del software en subsistemas interconectados por medio de interfases y define a un nivel significativo cómo los nodos ejecutando estos subsistemas interactúan unos con otros.

Buenas prácticas de diseño orientado a objetos: conjunto de actividades de diseño orientado a objetos sobre las cuales existe un consenso sobre su forma de realización entre los autores más reconocidos.

Clases principales: conjunto de clases que hacen a la esencia del comportamiento del sistema.

Colaboración: una sociedad de clases, interfaces y otros elementos que trabajan conjuntamente para lograr un comportamiento cooperativo que es mayor a la suma de todos los elementos.

Agregación: una forma especial de asociación que especifica una relación de totalidad-parte entre el conjunto (el todo), y la componente (la parte).

Contratos: un acuerdo que perfila las condiciones de colaboración entre clases. Los contratos pueden ser escritos para definir las expectativas y obligaciones de tanto los clientes como los que proveen un servicio.

Diseño: La fase del desarrollo de software cuyo objetivo primario es decidir la forma en que el sistema será implementado. Durante el diseño de un sistema, se toman decisiones estratégicas y tácticas, para satisfacer los requerimientos funcionales y las exigencias de calidad.

Encapsulación: es el ocultamiento de la representación interna de un objeto. El objeto proporciona un interfaz que manipula los datos sin exponer su estructura subyacente.

Especialización: relación de especialización/ generalización en la cual objetos de un elemento especializado (el subtipo) son substituibles por objetos del elemento generalizado (supertipo).

Firma: (signature) el nombre y parámetros de una propiedad de comportamiento.

Framework: una micro arquitectura que proporciona una "template" (plantilla) extensible para usos dentro de un dominio específico.

"Template" (plantilla) que agrupa un conjunto de ítem que se pueden aplicar en situaciones similares. Puede ser extendido.

Generalización: relación de especialización/ generalización en la cual objetos de un elemento especializado (el subtipo) son substituibles por objetos del elemento generalizado (supertipo).

Granularidad de métodos: se buscan definir métodos que realicen una función bien determinada

Herencia: relación entre clases, en la que una clase comparte la estructura o comportamiento definido en otra.

Librería: conjunto de clases

Métrica: una medida cuantitativa del grado en que un sistema, componente o proceso posee un atributo dado /IEEE Standard Glossary of Software Engineering Terms IEEE93.

Niveles de herencia: de una jerarquía es igual a la cantidad de clases de la rama con mayor número de clases, menos una clase.

Objetos: instancias de clases.

Orientación a objetos: una forma de pensar y modelar el dominio de negocios/mundo real que localiza en encapsulación y modularización, cohesión semántica y clasificación, polimorfismo y herencia.

Polimorfismo: significa que el que envía un estímulo no necesita conocer la clase de la instancia receptora. La instancia receptora puede pertenecer a una clase arbitraria.

Reutilizar: volver a usar algo.

Responsabilidades: un contrato u obligación de un tipo o una clase.

Responsabilidades públicas: de una clase es la interfase de la clase. En Java los métodos públicos.

Sobre-escritura: (overriding) de un método existe cuando un método es heredado, la firma del método es igual a su definición original y la implementación es remplazada, extendida o provista.

Superclase: en una relación de generalización, la generalización de otra clase, la subclase.

Tipos: a descripción de un conjunto de las entidades que comparten características comunes, relaciones, atributos, y la semántica.