UNIVERSIDAD
REY JUAN CARLOS

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

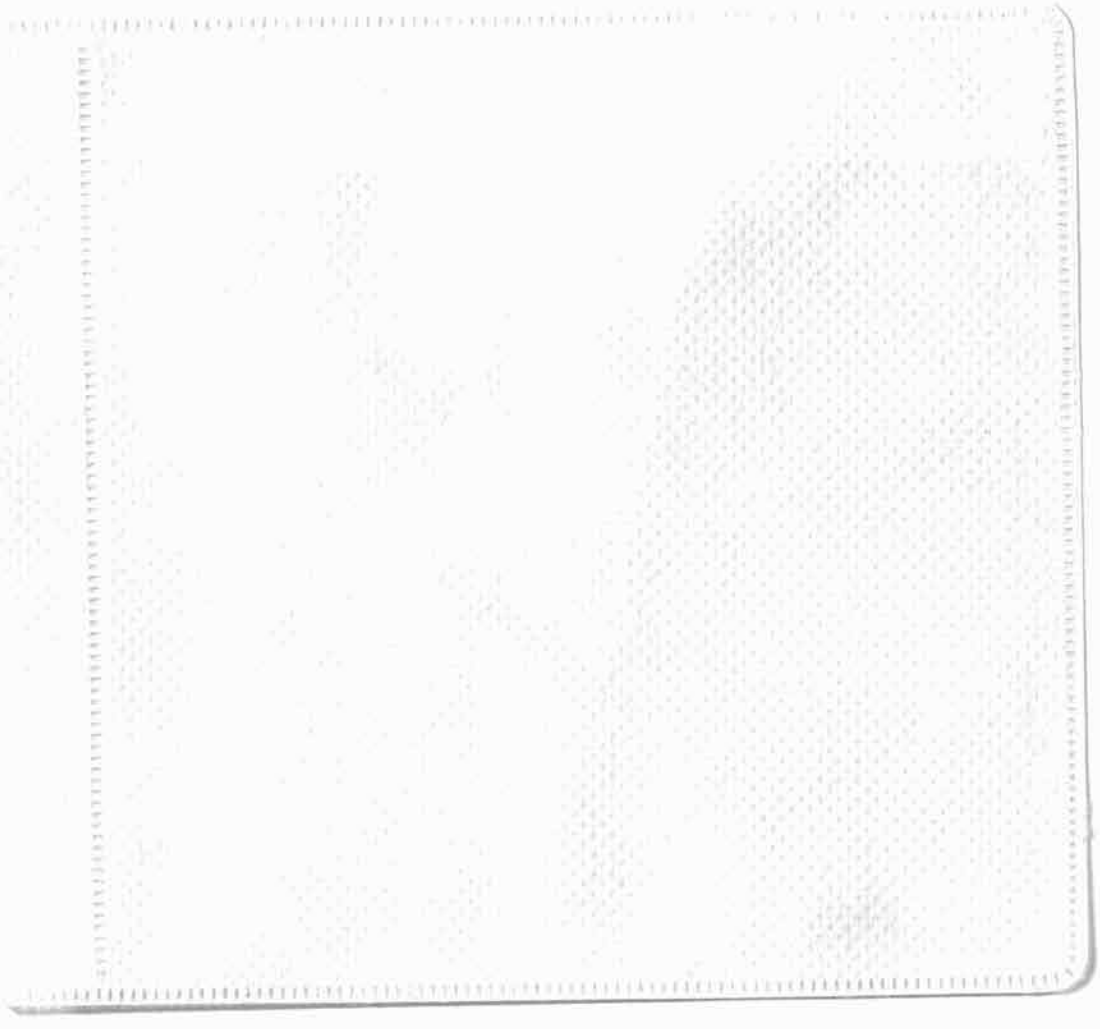
Curso Académico 2005/2006

Proyecto Fin de Carrera

DETECTOR DE BAD SMELLS

Autor : Daniel Cruz Martín

Tutor : Gregorio Robles Martínez



UNIVERSIDAD
REY JUAN CARLOS

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

Curso Académico 2005/2006

Proyecto Fin de Carrera

DETECTOR DE BAD SMELLS

Autor : Daniel Cruz Martín

Tutor : Gregorio Robles Martínez

–Well, there's egg and bacon; egg sausage and bacon; egg and spam; egg bacon and spam; egg bacon sausage and spam; spam bacon sausage and spam; spam egg spam spam bacon and spam; spam sausage spam spam bacon spam tomato and spam...

Índice general

Índice de figuras

Resumen

Un *bad smell* es un indicio de que existe código de poca calidad. La idea original es del libro de Martin Fowler (*Refactoring: Improving the Design of Existing Code*) donde se enumeran una serie de ellos. Este proyecto tiene como objetivo crear un software que detecte *bad smells* de manera automática a partir de código fuente escrito en el lenguaje de programación Python.

La aplicación que hemos desarrollado busca en el código fuente la presencia de siete *bad smells*: código duplicado, métodos largos, clases largas, métodos con gran número de parámetros de entrada, cláusulas *switch*, comentarios y variables de instancia temporales; ofreciendo como resultado de su análisis un informe estructurado y fácilmente comprensible (en HTML). Podemos ajustar el nivel de rigurosidad del análisis fijando los parámetros del programa (número máximo de líneas de un método, el máximo número de parámetros de un método, etc).

Además, para mostrar y poder discutir la validez de esta aproximación, se ha realizado un exhaustivo análisis a una gran cantidad de código fuente y con tres configuraciones distintas del programa. De esta manera, se han estudiado 655 paquetes de Debian Sarge que contienen código escrito en Python.

Capítulo 1

Introducción

En este capítulo presentamos los dos conceptos en los que se basa el proyecto: *bad smells* y refactorización. Ambos están estrechamente relacionados ya que es a partir del código mal escrito o de calidad cuestionable, del código con *mala pinta*, de donde surge la necesidad de modificarlo para mejorar su calidad mediante refactorización. En la última sección de este capítulo se mostrarán las principales características del lenguaje de programación en el que está escrita la herramienta que se ha realizado durante este proyecto.

La detección de *bad smells* en nuestro código plantea una serie de problemas durante el ciclo de vida del software que deben ser solucionados si se quiere producir código sencillo, estructurado y mantenible. La solución propuesta a estos olorosos problemas es la refactorización. ¿Qué significa refactorizar? La mejor definición la obtenemos del mismo Martin Fowler, que dijo: "Refactorizar es realizar modificaciones en el código con el objetivo de mejorar su estructura interna, sin alterar su comportamiento externo". De estas palabras sacamos ideas muy importantes que es necesario destacar. Refactorizar no es una técnica para encontrar y corregir errores en una aplicación, puesto que su objetivo no es alterar su comportamiento externo. No estamos depurando errores, de hecho, no modificar el comportamiento externo de la aplicación es uno de los requisitos de esta técnica. La esencia de la refactorización consiste en aplicar una serie de pequeños cambios en el código manteniendo su comportamiento. Cada uno de estos cambios debe ser tan pequeño que pueda ser completamente controlado sin miedo a equivocarse. Es el efecto acumulativo de todas estas modificaciones lo que hace de la refactorización una potente y beneficiosa técnica. Volveremos más adelante sobre el tema de la refactorización que acabamos de introducir. Antes es necesario profundizar un poco en la taxonomía de *malos*

olores que propuso Fowler para hacernos una idea del tipo de problema que estamos analizando.

1.1. Tipos de *bad smells*

¿Cuáles son estos *bad smells*?, ¿cuáles son los síntomas de que la evolución de nuestro proyecto se encamina hacia un caos?, ¿cuándo nuestro código empieza a oler? Martin Fowler propuso una lista con alguno de ellos como código duplicado, métodos largos, clases largas, cláusulas *switch*, comentarios, etc.

1.1.1. Código duplicado

El mismo código en más de un sitio es probablemente uno de los síntomas más claros de que nuestro código necesita una refactorización. La duplicación de código suele aparece cuando es necesario desarrollar una nueva funcionalidad que tiene relación con alguna otra que ya tenemos implementada. Es en ese momento cuando muchas veces solemos aplicar la solución más sencilla que es copiar el código similar para desarrollar la nueva funcionalidad. Replantearse el diseño para que podamos utilizar ese código supone tocar un código que estaba funcionando y rehacer el diseño y es posible que no tengamos la suficiente confianza para poder hacerlo. No obstante, copiar el código es una solución que es más costosa a largo plazo y que impedirá que nuestra aplicación crezca de una forma ordenada.

Tener código duplicado nos obligará, cuando realicemos un cambio, a realizarlo en muchas partes del código. Esto hará que cada cambio sea cada vez más costoso de implementar. Pero también tendremos otro problema si no memorizamos correctamente todos los sitios en los que se realizó esta copia ya que dejaremos partes de nuestro código en las que no se han implementado los cambios oportunos. Esto llevará a situaciones en las que será realmente complicado encontrar el error. Las posibles soluciones que nos aporta la refactorización a este problema son:

- Extracción de método: si dentro de una clase tenemos el mismo código diseminado por distintos sitios tendremos que crear un método que implemente la funcionalidad que estamos duplicando en esa clase.
- Extracción de clase: si tenemos el mismo código en clases distintas podríamos crear una

nueva clase en la que colocaríamos aquellos campos y métodos que serán heredados por las clases antiguas

- Substitución del algoritmo: si disponemos de dos métodos que hacen lo mismo pero con algoritmos distintos, elegiremos el algoritmo más claro y descartaremos el implementado de forma menos clara u óptima.

- Método de tipo Plantilla: existen métodos en subclases que realizan tares similares en el mismo orden, diferenciándose en las operaciones que realizan en cada tarea. En muchos de estos casos vemos código muy similar, pero no lo suficientemente parecido como para poder crear un método parametrizado que haga lo mismo. Una estrategia de refactorización que podemos emplear en este caso es extraer cada una de estas tareas en métodos que son llamados desde cada unos de los métodos originales de manera que el código de ambos métodos sea similar. Esto nos permitirá subir el método a la clase padre de manera que se convierte en un método cuyo código son invocaciones a métodos que serán resueltos en las clases hijas. Esto permite reutilizar código a un nivel más alto, permitiendo definir algoritmos o procesos genéricos cuyas implementaciones serán proporcionadas por las clases hijas. Podemos clarificar este concepto con el esquema 1.1, en la página 6.

1.1.2. Métodos Largos

Un método con un elevado número de líneas de código es igual a un método difícil de mantener que puede contener código duplicado. Hay que implementar pequeños métodos simples que funcionen bien antes que mastodontes con decenas de ramas *if* de las que quedarse colgado en busca de un error o que recorrer para introducir un cambio.

¿Cómo aplicamos la refactorización en este caso? Pues aplicando una práctica de programación (y de sentido común) básica: modularizando el código. Descomponiendo el método ciclópeo en métodos más pequeños que, debidamente combinados, ofrezcan la funcionalidad deseada. Esta estrategia de localizar aquellas partes del código que están relacionadas entre sí para extraerlas y así crear un nuevo método funciona la mayoría de las veces para adelgazar el tamaño de un método. Otras posibles refactorizaciones para atajar este *bad smell* ahondan en la estrategia presentada de extracción. Si tenemos un bloque *if-then-else* largo y complicado podemos crear métodos que evalúen las condiciones sin perder su semántica. Esta idea también

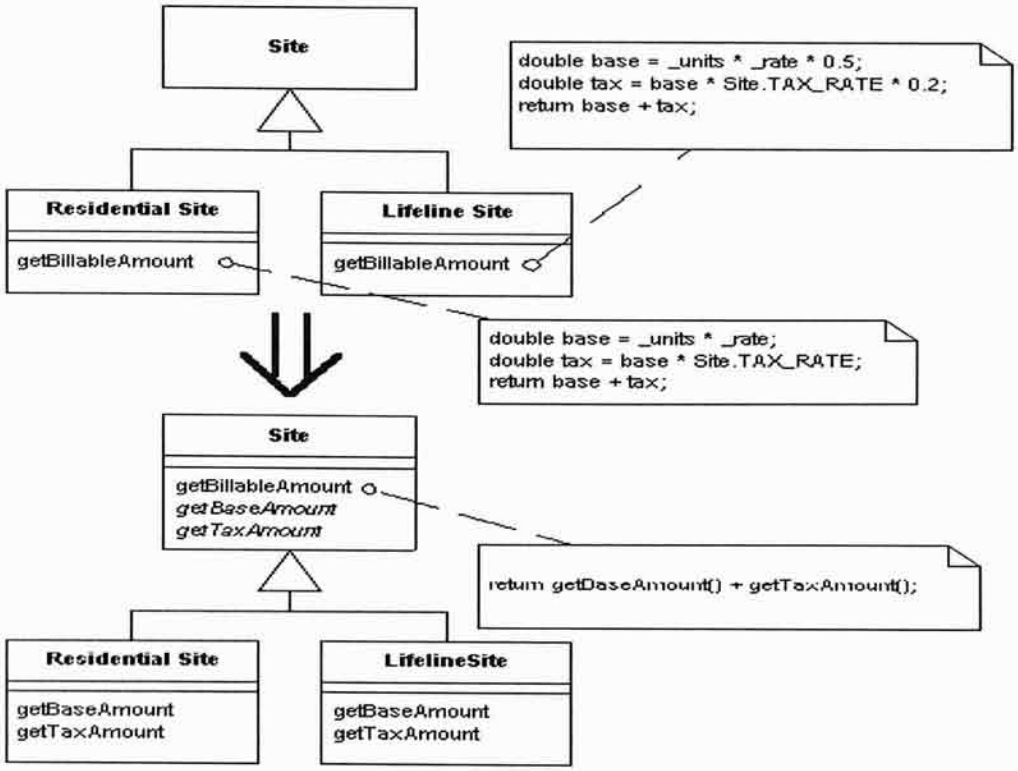


Figura 1.1: Método de tipo Plantilla

la podemos llevar a cabo con los bucles para encapsular de algún modo el proceso iterativo. Otra estrategia consiste en buscar las líneas comentadas. Un fragmento de código comentado debería ser refactorizado creando un método que haga lo que dice el comentario. Si llevamos esto último al extremo podemos afirmar que hasta una única línea debe ser refactorizada si necesita ser comentada.

1.1.3. Clases Largas

Una clase larga presumiblemente realiza muchas cosas, con lo cual contendrá muchas variables de instancia y, al ser muy extensa, es probable que contenga código duplicado y puede que sus métodos sean también muy grandes. Todas estas implicaciones empiezan a desprender ciertos aires mefíticos que nos avierten, quizá, de la necesidad de refactorizar. ¿Con qué ténicas contamos en este escenario?

- Extracción de clase: cuando tenemos una clase con multitud de métodos y datos podemos reestructurarla creando una nueva traspasándola aquellos campos y métodos que necesite

para realizar su función. Realmente lo que propone esta técnica es dividir una clase grande que hace todo en otras más pequeñas y manejables.

- Extracción de subclase: cuando una clase ofrece funcionalidades que sólo son instanciadas en determinadas ocasiones puede que sea necesario plantearse una jerarquía de clases en la que tengamos una subclase que implemente las funcionalidades que son usadas ocasionalmente en la clase padre.

1.1.4. Gran número de parámetros de entrada

Los métodos con muchos parámetros de entrada son difíciles de entender y además suelen cambiar cuando se necesita acceder a nuevos datos. No es necesario pasarle a un método absolutamente todo, sólo aquéllo que verdaderamente necesite para realizar su función o que pueda usar para obtener los datos que le hagan falta. Si tenemos un conjunto de datos que siempre van juntos podemos encapsularlos en un objeto que pasaremos como parámetro. Si invocamos un método de un objeto y el resultado de éste es pasado como parámetro a otro, nada nos impide quitar el parámetro de entrada y que sea el método receptor el que invoque dentro de su código al método que le proporciona el resultado que necesite. Ejemplo:

```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice (basePrice, discountLevel);

   es transformado en

int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice (basePrice);
```

Si obtenemos mediante el método get habitual varios datos de un objeto para pasárselos seguidamente a un método, puede que sea mejor pasarle al método el objeto entero y así reducir el número de parámetros de entrada.

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low,high);

   es transformado en

withinPlan=plan.withinRange(daysTempRange());
```

1.1.5. Cláusulas *switch*

Una de las características de la programación orientada a objetos es la poca cantidad de sentencias *case* en comparación con la programación imperativa. El principal problema que existe con este tipo de sentencias es que diseminan código duplicado ya que es muy común encontrarse el mismo bloque *case* esparcido en distintas partes de un programa. Cuando se necesita añadir una nueva condición a este tipo de sentencias hay que buscar y modificar todas estas sentencias, con el riesgo de que nos olvidemos de alguna. Por ello la noción de polimorfismo que la programación orientada a objetos nos proporciona es una forma elegante de tratar con este tipo de problema. Para intentar refactorizar este tipo de código podemos reemplazar las cláusulas condicionales que tenemos introduciendo polimorfismo. Es decir, nuestra situación inicial es que dependiendo del tipo de un objeto tenemos una serie de sentencias concionales que implementan acciones o comportamientos distintos. La forma de proceder para corregir esto es implementar una subclase con un método sobrecargado para cada una de las ramas concionales que tengamos y crear una clase padre abstracta. Es decir, tenemos un código como el siguiente:

```
double getSpeed() {
   switch (_type) {
     case EUROPEAN:
       return getBaseSpeed();
     case AFRICAN:
       return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
     case NORWEIGIAN_BLUE:
       return (_isNailed) ? 0 : getBaseSpeed(_voltage);
   }
   throw new RuntimeException ("Should be unreachable");
}
```

Podemos establecer la jerarquía de clases que vemos en la figura 1.2 de la página 9:

1.1.6. Comentarios

No se pretende eliminar todas las líneas comentadas que encontremos. Pero a veces los comentarios son usados como desodorante y es muy común encontrarse código prolijo en comentarios porque el código que hay más abajo es malo y poco claro. Si vemos que necesitamos comentar un fragmento de código deberíamos intentar refactorizarlo con alguna de las técnicas que hemos expuesto anteriormente e intentar conseguir que los comentarios sean superfluos. Tenemos que intentar que nuestro código hable por sí solo. Si tratamos de explicar lo que hace

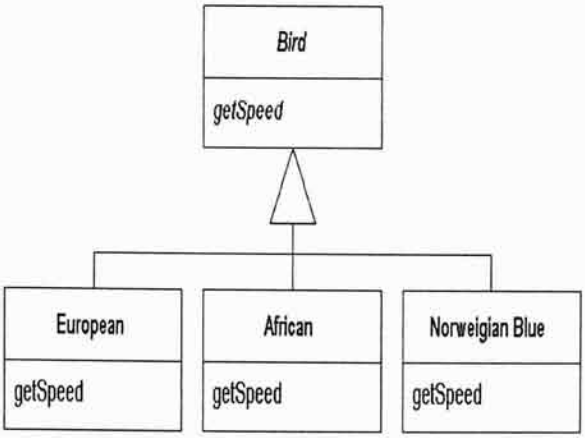


Figura 1.2: Reemplazar condicionales con polimorfismo

nuestro código estaremos duplicando información que en cualquier momento puede cambiar. Los comentarios son útiles para indicar por qué se realizó algo de cierta manera, ya que ese tipo de información suele ser útil para modificaciones futuras.

1.1.7. Variables de instancia temporales

Cuando en un objeto tenemos variables de instancia que son usadas sólo en determinadas circunstancias nos encontramos ante un código que es difícil de entender ya que lo normal es que un objeto necesite todas sus variables. Averiguar para qué sirve una variable de un objeto que a veces se usa y otras no puede ser frustrante si no se es el autor del código. Un ejemplo de esto se da cuando al implementar un algoritmo complicado se necesitan una serie de variables temporales y, para evitar pasar al método una lista de parámetros grande, se emplean variables de instancia que sólo son válidas durante la ejecución del mismo. Para evitar esto podemos crear una clase que implemente el algoritmo y tenga todas las variables de instancia que necesite. Si nos damos cuenta la situación que estamos describiendo encaja en el siguiente escenario: tenemos una clase que ofrece cada vez más funcionalidades y está creciendo de forma desordenada haciendo que su diseño original se esté desbordando.

1.1.8. Otros *bad smells*

Los anteriores son los *bad smells* que detecta el presente proyecto. La taxonomía de Fowler recoge hasta un total de veintidós, a continuación se presentan brevemente alguno de ellos:

- Cambio divergente: el software debe estar estructurado para que las modificaciones resulten sencillas. Si una clase tiene que cambiarse por diferentes motivos, puede que sea mejor dividirla en otras más pequeñas en las que se implementen los cambios necesarios.

- Cirugía en cascada: si un cambio en el software implica modificar el código de distintas clases podría ser mejor implementar los cambios en una nueva clase.

- Método envidioso: si un método está más interesado en los atributos (normalmente datos) de otra clase, puede que sea mejor que pertenezca a esa clase.

- Grupo de datos: se trata de encapsular en una clase aquellos datos que siempre aparecen juntos en distintos sitios. Un buen test: si borramos uno de los datos, ¿los que quedan tienen sentido?

- Obsesión por tipos de datos primitivos: a veces es mejor transformar un tipo de dato primitivo en una clase ligera para establecer de forma clara para qué sirve y las operaciones que soporta.

- Clase Perezosa: las clases que no realizan muchas cosas deben ser eliminadas.

- Cadena de mensajes:

  getA().getB().getC().getD().getE().doIt();

  El cliente está atado a una estructura de navegación.

- Hombre en medio: los objetos ocultan los detalles internos encapsulándolos y delegando en otras clases ciertas operaciones. Esto puede degenerar propiciando la existencia de clases sin mayor funcionalidad que la de ser meras intermediarias.

- Curiosidad malsana: este tipo de comportamiento se da cuando una clase accede a las partes privadas de otra.

- Clases alternativas con distintas interfaces: las clases que realizan funciones similares pero tienen nombres diferentes deberían ser modificadas para que compartan un protocolo común.

- Clases de sólo datos: cuando una clase no contiene más que campos de datos y métodos de acceso, sin mayor funcionalidad real. Si es una clase pública hay que hacerla privada. También hay que incorporar mayor funcionalidad a la clase.

- Herencia no deseada: si una subclase sólo necesita una parte del comportamiento de su padre es probable que la jerarquía de clases no sea la correcta.

- Generalización especulativa: uno de los errores más peligrosos que podemos cometer es tratar de generalizar en exceso. Normalmente este tipo de errores se comenten cuando se empieza a especular sobre las necesidades que tendrá nuestro proyecto en el futuro. Se trata de buscar soluciones muy sofisticadas que resuelven problemas diferentes a los que se tienen en el momento actual del desarrollo.

1.2. Refactorización

Hemos introducido el concepto de refactorización al principio de esta memoria y es necesario que lo ampliemos aunque antes recordaremos qué entendemos por refactorización:

- Refactorización es el proceso de modificar el código para mejorar su estructura interna sin alterar la funcionalidad que ofrece externamente.

- La refactorización no es la optimización de código en sentido de rendimiento. No se trata de solucionar "los bugs" (pero sí podría ayudarnos a encontrarlos o evitarlos).

- La refactorización busca producir código más fácil de entender, controlar, cambiar y mantener.

La refactorización es uno de los nuevos conceptos que se han introducido en la terminología del desarrollo de software apoyado por las metodologías ágiles. *Extreme Programming* (conocida en castellano como programación extrema), una de las metodologías ágiles más extendida, la incluye dentro del decálogo de prácticas que propone como fundamentales. ¿Por qué debemos refactorizar nuestro código? He aquí una serie de razones:

- Calidad: la más importante. Refactorizar es un continuo proceso de reflexión sobre nuestro código que permite que aprendamos de nuestros desarrollos en un entorno en el que

no hay mucho tiempo para mirar hacia atrás. Un código de calidad es un código sencillo y bien estructurado, que cualquiera pueda leer y entender sin necesidad de haber estado integrado en el equipo de desarrollo durante varios meses. Se acabaron los tiempos en que lo que imperaba eran esos programas escritos en una sola línea en la que se hacía de todo, tiempos en los que se valoraba la concisión aun a costa de la legibilidad.

- Eficiencia: mantener un buen diseño y un código estructurado es sin duda la forma más eficiente de desarrollar. El esfuerzo que invirtamos en evitar la duplicación de código y en simplificar el diseño se verá recompensado cuando tengamos que realizar modificaciones, tanto para corregir errores como para añadir nuevas funcionalidades.

- *Diseño Evolutivo* en lugar de *Gran Diseño Inicial*: en muchas ocasiones los requisitos al principio del proyecto no están suficientemente especificados y debemos abordar el diseño de una forma gradual. Cuando tenemos unos requisitos claros y no cambiantes un buen análisis de los mismos puede originar un diseño y una implementación brillantes, pero cuando los requisitos van cambiando según avanza el proyecto, y se añaden nuevas funcionalidades según se le van ocurriendo a los participantes o clientes, un diseño inicial no es más que lo que eran los requisitos iniciales, algo generalmente anticuado. Refactorizar nos permitirá ir evolucionando el diseño según incluyamos nuevas funcionalidades, lo que implica muchas veces cambios importantes en la arquitectura, añadir cosas y borrar otras.

- Evitar la reescritura de código: en la mayoría de los casos refactorizar es mejor que reescribir. No es fácil enfrentarse a un código que no conocemos y que no sigue los estándares que uno utiliza, pero eso no es una buena excusa para empezar de cero. Sobretodo en un entorno donde el ahorro de costes y la existencia de sistemas lo hacen imposible.

Refactorizar es por tanto un medio para mantener el diseño lo más sencillo posible y de calidad. ¿Qué se entiende por sencillo y de calidad? Kent Beck, uno de los creadores de la metodología ágil *Extreme Programming*, define una serie de características para lograr un código lo mas simple posible:

- El código funciona (el conjunto de pruebas de la funcionalidad de nuestro código pasan correctamente).

- No existe código duplicado.
- El código permite entender el diseño.
- Minimiza el número de clases y de métodos.

A pesar de todo ello, refactorizar parece ser muchas veces una técnica en contra del sentido común. ¿Por qué modificar un código que funciona? (si funciona no lo toques) ¿Por qué correr el riesgo de introducir nuevos errores?, ¿cómo podemos justificar el coste de modificar el código sin desarrollar ninguna nueva funcionalidad? No siempre es justificable modificar el código, no se reduce a una cuestión estética. Cada refactorización que realicemos debe estar justificada. Sólo debemos refactorizar cuando identifiquemos código mal estructurado o diseños que supongan un riesgo para la futura evolución de nuestro sistema. Si detectamos que nuestro diseño empieza a ser complicado y difícil de entender, y nos está llevando a un situación donde cada cambio empieza a ser muy costoso. Es en ese momento cuando debemos ser capaces de frenar la inercia de seguir desarrollando porque si no lo hacemos nuestro software se convertirá en algo inmantenible (es imposible o demasiado costoso realizar un cambio). Tenemos que estar atentos a estas situaciones donde lo más inteligente es parar, para reorganizar el código. Se trata de dar algunos pasos hacia atrás que nos permitirán tomar carrerilla para seguir avanzando. Los síntomas que nos avisan de que nuestro código tiene problemas ya los concemos, son los *bad smells*. Una vez identificado el *bad smell* deberemos aplicar una refactorización que permita corregir ese problema, como ya hemos visto anteriormente cuando explicábamos los diferentes tipos de *bad smells*.

Para comenzar a refactorizar es imprescindible que el proyecto tenga pruebas automáticas, tanto unitarias como funcionales, que nos permitan saber en cualquier momento al ejecutarlas, si el desarrollo sigue cumpliendo los requisitos que implementaba. Sin pruebas automáticas, refactorizar es una actividad que conlleva un alto riesgo. Sin pruebas automáticas nunca estaremos convencidos de no haber introducido nuevos errores en el código al término de una refactorización, y poco a poco dejaremos de hacerlo por miedo a estropear lo que ya funciona.

Conseguir un código sencillo es una tarea muy compleja. Como dice Carver Mead, una de las leyendas vivientes de Silicon Valley: "Es fácil tener una idea complicada. Es muy, muy complicado tener una idea simple".

1.3. Tecnologías utilizadas

Python es un lenguaje de *scripting* orientado a objetos. Proporciona la simplicidad y facilidad de uso de un lenguaje interpretado, así como las más avanzadas herramientas de programación propias de lenguajes destinados al desarrollo de sistemas (como C o C++). Destaca por su facilidad de aprendizaje y su portabilidad entre las distintas plataformas, tan sólo condicionada a la presencia de un intérprete disponible.

Nombremos algunas de las propiedades que caracterizan a Python como lenguaje de alto nivel:

- Tipado dinámico y fuertemente tipado. No requiere que nos tomemos la molestia de declarar variables ya que reconoce su tipo desde que se asigna un valor por primera vez. Eso sí, durante el tiempo de vida de una variable ésta sólo puede pertenecer a un tipo.

- Proporciona estructuras de datos flexibles y sencillas de usar como listas, diccionarios y strings, como parte intrínseca al lenguaje. Para procesar cada uno de estos tipos de objeto, incorpora un conjunto de operaciones que ahorrarán tiempo y esfuerzo al implementar tareas de uso habitual como ordenaciones, búsquedas, etc.

- Python posee una amplia colección de bibliotecas dedicadas a tareas específicas.

- Administra automáticamente la gestión de memoria. Se encarga de solicitar memoria en la creación de objetos y de liberarla cuando no van a volver a ser utilizados.

- Permite la construcción de sistemas de gran tamaño al incorporar herramientas como clases, módulos, y excepciones.

- A los programas escritos en Python pueden integrarse componentes escritos en otros lenguajes. Por ejemplo, mediante el API de Python/C, un código Python puede extender su funcionalidad incorporando componentes escritos en C o C++, lo cual convierte a Python en un lenguaje de prototipado rápido: las aplicaciones pueden ser implementadas en primera instancia con Python para aumentar su velocidad de desarrollo y posteriormente ciertas partes reescritas en C por motivos de eficiencia.

Con Python es perfectamente viable el desarrollo de proyectos software de gran entidad, ejemplo de ello son el servidor de aplicaciones *Zope* y el sistema de intercambio de ficheros

BitTorrent.

Otro aspecto que nos interesa particularmente es la incorporación, dentro de la completa biblioteca que ofrece, de módulos que manejan estándares comunes en Internet (HTML, FTP, XML, HTTP, ...), APIs para la comunicación con bases de datos (para gestores como PostgreSQL, Oracle o MySQL), también hay módulos incluidos que proporcionan E/S de ficheros, llamadas al sistema, sockets y hasta interfaces a GUI (interfaz gráfica con el usuario) como Tk, GTK, Qt entre otros.

El nombre del lenguaje proviene de la afición de su creador original, Guido van Rossum, por los geniales humoristas británicos *Monty Python*. Actualmente, Python se desarrolla como un proyecto de código abierto, administrado por la Python Software Foundation. Los usuarios de Python consideran a éste un lenguaje limpio y elegante para programar. La última versión estable del lenguaje es actualmente (Marzo de 2006) la 2.4.3.

Capítulo 2

Objetivos

Con el fin de elegir los proyectos teniendo en cuenta la calidad de su código hemos escrito esta aplicación, llamada *badsmcheck*, que nos permitirá hacernos una idea general sobre su calidad sin necesidad de tener que bucear por sus distintos módulos.

Es una herramienta dirigida, en principio, a *analistas y programadores* que pretende ser un primer filtro a la hora realizar el análisis de un proyecto. Estamos pensando en el siguiente ejemplo: tenemos un proyecto que queremos adaptar a nuestras necesidades, si después de pasarle nuestra herramienta al proyecto observamos muchas deficiencias del tipo que implican los *bad smells* puede que el código fuente con el que estamos tratando no sea el más adecuado. Otro posible escenario en la que esta herramienta puede resultar útil, es aplicarla durante el proceso de desarrollo de un proyecto para supervisar la calidad del código aplicando de forma continua la refactorización.

2.1. Funcionamiento

Esta aplicación se comporta tal y como nosotros haríamos manualmente una inspección de código en busca de *bad smells*: descargarnos los paquetes software, posiblemente descomprimir las fuentes, investigar los archivos con código fuente uno a uno para identificar los diferentes tipos de *bad smells*, apuntarlos y finalmente generar un informe con toda la información.

Sin embargo, gracias a la herramienta que se ha creado, una vez nos hayamos descargado el código del proyecto en el que estamos interesados, únicamente tenemos que llamar al programa pasándole como parámetro el software que nosotros queremos estudiar:

```
$ ./badsmcheck proyecto.tar.gz
```

A continuación, el programa descomprimirá el *tar.gz* y comenzará a buscar los siguientes *bad smells* en todos los ficheros del proyecto que estén escritos en Python:

- Clases largas
- Métodos largos
- Métodos con gran número de parámetros de entrada
- Comentarios
- Cláusulas *switch*
- Código duplicado
- Variables de instancia temporales

El programa discrimina la inclusión o no de los *bad smells* que vaya detectando en base a la configuración que tenga establecida. Esta configuración puede ser fijada al lanzar la aplicación.

Una vez analizado el fichero correspondiente, se evaluarán los resultados y se generará un informe. Este informe se detallará más adelante, pero podemos adelantar que consta de cuatro secciones, organizado de información más genérica a más específica.

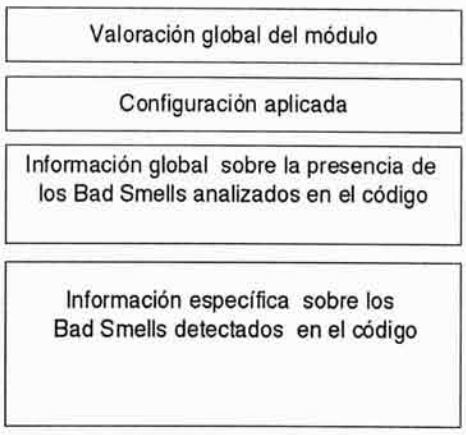


Figura 2.1: Estructura del informe.

Para permitir al usuario del programa interpretar los resultados de una forma rápida, el informe cuenta con un código de colores que muestra el estado en que se encuentra un *bad smell* determinado. Esto permite la comparación rápida de un mismo *bad smell* entre varios proyectos. Además del informe individual para cada uno de los ficheros se genera un documento índice que ofrece una valoración global del proyecto ofreciendo un resumen con la calificación que han obtenido cada uno de los módulos además de ofrecer un enlace al resumen de los ficheros analizados. En la página siguiente presentamos el diagrama de flujo de la aplicación.

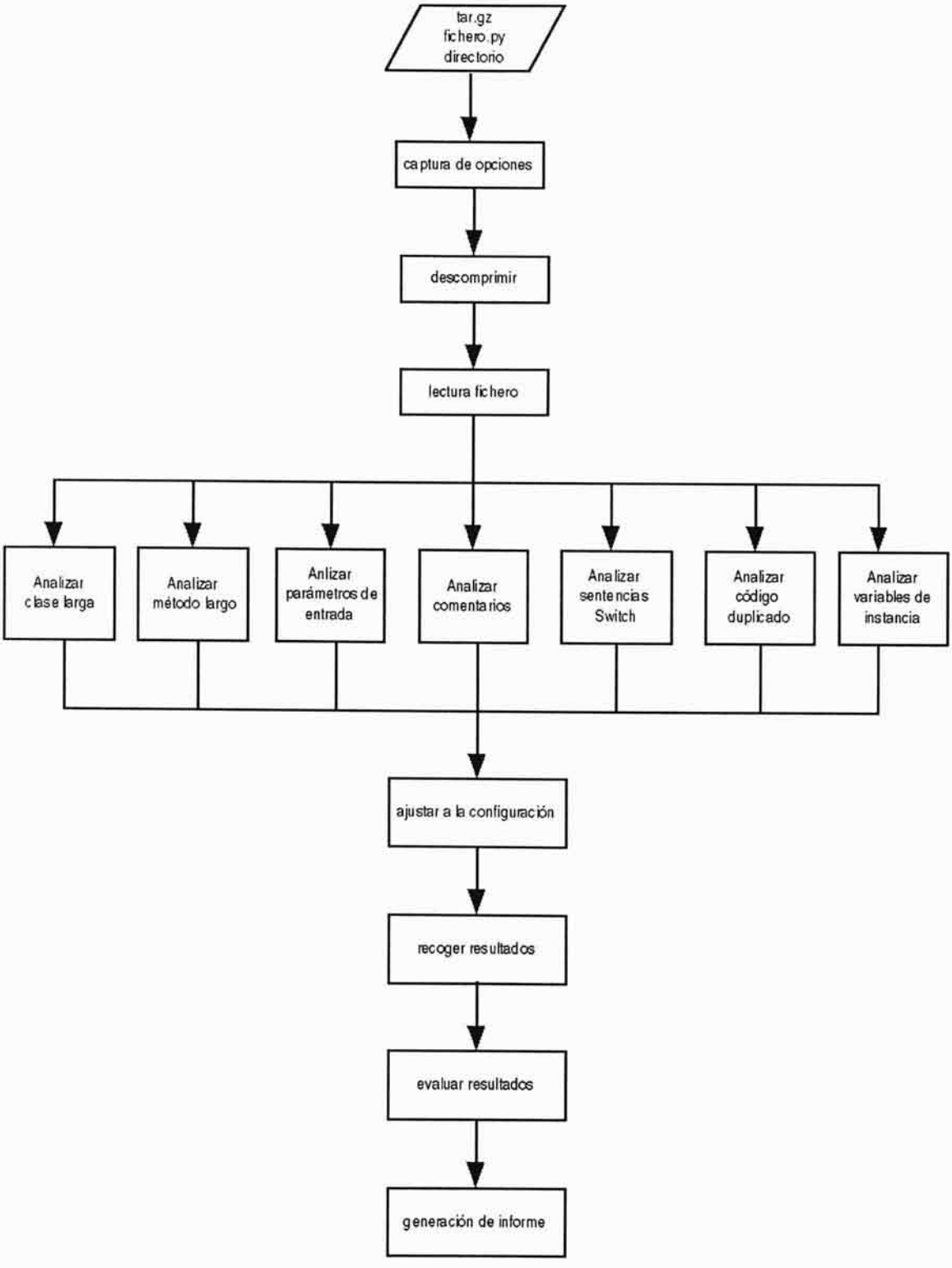


Figura 2.2: Diagrama de flujo de la aplicación.

Capítulo 3

Diseño e implementación

Para la realización del proyecto hemos ido marcando unas etapas intermedias antes de llegar al resultado final. Para ello hemos seguido un modelo de desarrollo en espiral y, en cada una de las iteraciones hemos ido añadiendo nuevas funcionalidades y características.

Se ha modularizado la aplicación para que el software sea más fácil de comprender y mantener. Aprovechando la orientación al objeto que incorpora Python, hemos estructurado el código creando clases que implementan las funcionalidades que el dominio del problema exigía. El diagrama de clases de la aplicación lo podemos observar en la página 29.

Las clases que se han desarrollado son:

- **Counter:** es la clase padre de la que heredan cada una de las clases en las que implementamos el análisis de los *bad smells* que detecta la aplicación. En sus variables de instancia están las listas portadoras que recogen los resultados de los análisis. No es una clase abstracta ya que contiene métodos comunes que son usados por todos sus hijos.

- **ClassCounter:** cuenta el tamaño de las clases y va incluyendo en la lista resultado el nombre, la posición en el código y la longitud de la clase si ésta sobrepasa los límites impuestos a la hora de configurar el programa.

- **MethodCounter:** esta clase cuenta el tamaño de los métodos y el número de parámetros de entrada de los mismos, incluyendo en la lista resultado el nombre, la posición en el código, la clase a la que pertenece (si procede) y la longitud del método si éste sobrepasa los límites impuestos a la hora de configurar el programa.

- **CommentsCounter:** esta clase controla en base a la configuración que se haya establecido el tamaño de los comentarios. Vierte a la lista resultado la posición en el código, la clase y el método (si procede) al que pertenece el comentario detectado.

- **SwitchCounter:** esta clase busca bloques *if-elif-else* en el código. Cuando encuentra uno que entra dentro de los parámetros establecidos, introduce en la lista resultado su posición, longitud y, si procede, la clase y el método al que pertenece.

- **DupCounter:** busca código duplicado e indica, si los bloques tienen la longitud que se haya establecido en la configuración, las líneas de código coincidentes.

- **TempFieldCounter:** ésta es la última clase dedicada a detectar *bad smells*. Explora los métodos constructores de clase buscando variables de instancia que estén declaradas y luego no se usan o bien aquéllas que en relación con el resto de las variables de instancia que componen la clase son usadas comparativamente menos. Es la clase más subjetiva de todas ya que los resultados que arroja deben ser interpretados con cuidado porque si bien podemos haber detectado un *bad smell* de bulto (en los análisis realizados se han llegado a detectar variables que nunca se usaban) también es posible que desde el punto de vista del diseño de la clase la menor instanciación de una variable sea algo completamente lógico y correcto. Es ésta una clase que empieza a introducir en el diseño cierto grado de incertidumbre que hace que estemos adoptando posturas interpretativas sobre el código. Discutiremos este hecho más adelante en mayor profundidad.

- **FileIterator:** verdadero motor de la aplicación. Esta clase itera sobre los ficheros Python leyendo cada línea y va instanciando los métodos polimórficos de las clases que hemos explicado antes para que realicen el análisis del *bad smell* que tengan asignado. Finalmente, usa métodos de la clase Resulter para ir recolectando los resultados. También se encarga de descomprimir, si es necesario, los ficheros *tar.gz*.

- **Resulter:** esta clase almacena las listas con los resultados que generan las clases descendientes de Counter y, una vez que se concluye el proceso de análisis, va construyendo los informes de salida llamando a los métodos que le facilita la clase HTMLwriter.

- **HTMLwriter:** esta clase realiza la escritura de cada una de las partes de los informes en HTML que ensamblará la clase Resulter.

- **Evaluator:** esta clase analiza el *bad smell* detectado y lo evalúa asignándole un nivel de gravedad. Esa calificación es pasada a la clase HTMLwriter para que pinte el código de color correspondiente.

- **Configurer:** esta clase es la que almacena la configuración del programa y es usada para determinar si se debe informar al usuario sobre los *bad smells* que se detecten.

A continuación, describiremos los pasos seguidos durante el desarrollo del programa y comentaremos algunos resultados obtenidos.

3.1. Primer paso: desarrollo de scripts independientes

Para empezar a aterrizar en el lenguaje Python, se desarrollaron pequeños scripts para tener código ejecutable que ofreciera cierta funcionalidad, sin entrar en detalles muy profundos de diseño. Una de las ventajas que brinda Python es que facilita un desarrollo rápido, por ello lo primero que se hizo fue implementar la gestión de archivos más básica (apertura, lectura y cierre). Una vez superada esta fase trivial se implementaron scripts independientes que aceptaban como entrada un fichero con extensión *.py* y controlaban los siguientes aspectos:

1. Contar el número de parámetros de entrada de los métodos que contenía el fichero
2. Contar el número de líneas de clases y métodos
3. Detectar líneas de código duplicado
4. Contar longitud de los comentarios

3.1.1. Lecciones aprendidas en el primer paso

En esta primera etapa se experimentó con el lenguaje, familiarizándonos con su sintaxis y adquiriendo las habilidades básicas para afrontar las siguientes etapas. Las pruebas realizadas con los scripts iniciales no se realizaron con código Python "real" como el que posteriormente sería objeto de un exhaustivo análisis.

3.2. Segundo paso: empieza el diseño de las clases

A partir de aquí se hizo evidente la necesidad de empezar a desarrollar una estructura de clases que nos facilitara el diseño. La primera clase que se implementó fue **FileIterator** que, básicamente, abre un archivo y lee línea tras línea su contenido. Fue casi inmediato pensar en pasar las líneas que leía esta clase a otras que modelasen cada uno de los *bad smells*. Por lo tanto los scripts individuales que se escribieron en el primer paso se transformaron en clases autosuficientes que procesaban las líneas que les pasaba el iterador de ficheros. La implementación de la clase iteradora condicionó el desarrollo posterior del proyecto ya que sólo había que seguir conectando módulos independientes a la salida de ésta para ir tratando distintos tipos de *bad smells*.

3.2.1. Problemas encontrados en esta fase

Llegados a este punto del análisis y con cuatro clases prácticamente implementadas había que decidir qué otros *bad smells* trataría nuestra aplicación. La lista de Martin Fowler es bastante amplia como hemos podido ver y tratar todos ellos desbordaría las dimensiones de este PFC. Podemos establecer una categorización de los *bad smells* clasificándolos en dos tipos: intraclases e interclases. Los intraclases se caracterizan por presentar defectos de diseño interno, los que trata la aplicación que hemos escrito son todos intraclases. Por el contrario, los interclases son los que contienen defectos en la forma en que las clases se comunican y se acoplan las unas con otras (método envidioso, hombre en medio, herencia no deseada).

3.3. Tercer paso: definiendo la herencia

En este paso se redefinió la estructura de clases y se creó la clase **Counter**. Esta clase es de la que descienden todas las clases que modelizan la detección de *bad smells* en el código. Se introdujo para evitar código duplicado ya que implementa métodos que son usados por todos sus hijos y además nos permitió usar el polimorfismo para implementar de forma elegante la funcionalidad real de las clases detectoras. Se diseñaron en esta etapa las últimas dos clases desciendentes de **Counter**: **SwitchCounter** y **TempFieldCounter**. Además se decidió que la clase **MethodCounter** controlara dos tipos de *bad smell*, la longitud de los métodos y el número

de parámetros de entrada porque así se agrupaba en la misma clase el análisis de dos *bad smells* estrechamente relacionados. Los resultados del programa en esta etapa se ofrecían por línea de comandos.

Por último, se creó la clase **Configurer** para controlar la configuración del programa.

3.4. Cuarto paso: presentación de resultados

En las etapas anteriores nos hemos dedicado a procesar ficheros para detectar una serie de *bad smells* e ir almacenando una serie de datos, como la longitud de las clases, el número de parámetros de entrada de los métodos, si existen líneas con código duplicado etc.

Todos estos datos son los que se escapan de los parámetros de configuración que hemos establecido. Es decir, si hemos establecido que consideramos que el número máximo de parámetros de entrada para los métodos son cinco sólo se habrán recogido los datos de aquellos métodos que superen ese número. Por lo tanto, los datos en "crudo" sobre los *bad smells* de las clases detectoras son almacenados en listas.

Hemos utilizado lógica difusa para determinar si el resultado de los análisis de cada uno de los *bad smells* es bueno, de mediana calidad o malo, representándolo en pantalla con tres colores (verde, amarillo o rojo).

Las clases que realizan estas funciones son las siguientes: **Resulter**, **HTMLwriter** y **Evaluator**

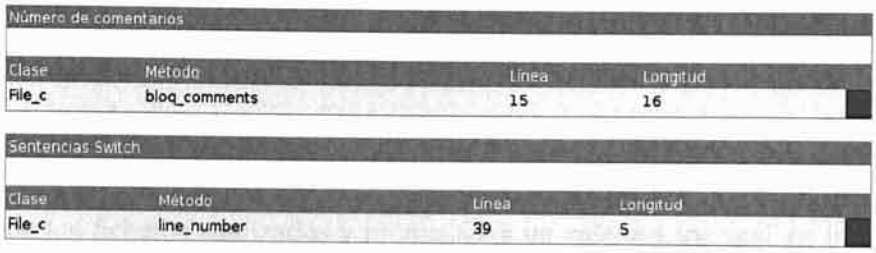
A partir del procesamiento de esas listas que se han ido almacenando generamos un informe en HTML como el que podemos ver en la figura 3.2 de la página 30.

En esta figura podemos ver un ejemplo de la salida del programa. Como se puede observar, se trata de una página HTML estática. Como comentamos anteriormente, la salida consiste en un informe con toda la información obtenida. En este informe no sólo se dan los datos numéricos que hemos extraído sino que, gracias a la lógica difusa hemos podido incluir un código de colores a través del cual va a ser mucho más fácil identificar la naturaleza de los *bad smells*. El código de colores simplifica la interpretación de los resultados. Podemos observar también la estructura seguida en la realización del informe. Como dijimos consta de cuatro partes bien diferenciadas en las que la información que se proporciona es cada vez más específica.

En este caso la aplicación ha detectado dos *bad smells*, el primero de ellos nos indica que

el tamaño de la clase que contiene el fichero de prueba sobrepasa ligeramente los límites que se habían establecido en la configuración. El otro *bad smell* que detecta la aplicación indica que el número de parámetros de entrada de uno de los métodos es excesivo.

A continuación presentamos la salida que ofrece el programa cuando detecta comentarios y bloques de sentencias condicionales. En ambos casos se ofrece la clase y el método donde se ha detectado el *bad smell*, la línea y la logitud del comentario o el número de ramas *elif*.



Número de comentarios

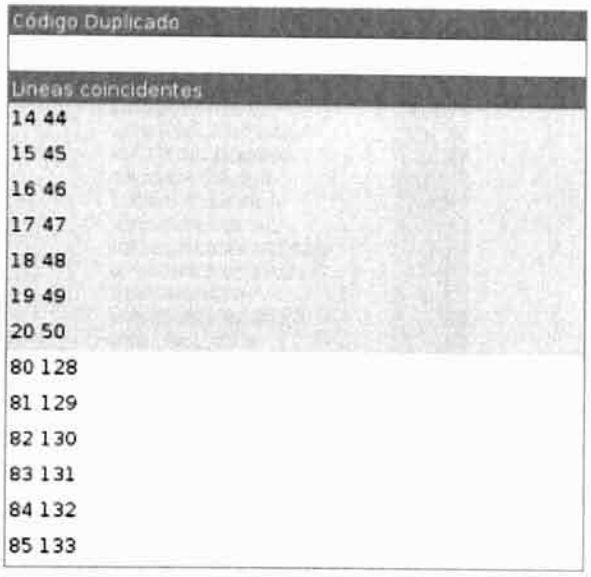
Clase	Método	Línea	Longitud
File_c	bloq_comments	15	16

Sentencias Switch

Clase	Método	Línea	Longitud
File_c	line_number	39	5

Figura 3.3: Ejemplo de salida de número de comentarios y sentencias switch.

La parte del informe que muestra el código duplicado se ofrece más abajo. En el ejemplo se advierte la existencia de dos bloques de código duplicado. Se ofrecen las líneas que coinciden para que se pueda inspeccionar directamente el código.



Código Duplicado

Líneas coincidentes
14 44
15 45
16 46
17 47
18 48
19 49
20 50
80 128
81 129
82 130
83 131
84 132
85 133

Figura 3.4: Ejemplo de salida de código duplicado.

Si alguna de las variables de instancia de las clases que tenga el fichero de entrada presenta

bad smells la salida la podemos observar en la figura 3.5 de la página 30.

Como podemos observar en la figura el análisis de las variables de instancia de esta clase ilustra a la perfección el tipo de situación que se comentó anteriormente sobre este *bad smell.* Nos encontramos con que existen variables de instancia que nunca usamos y otras que, en comparación con el resto, son usadas relativamente poco. Como el número de atributos de la clase que no son usados es alto puede que estemos tratando en este caso con otro *bad smell*: el de **Clase de sólo Datos**. Si además contiene muchos métodos de acceso/mutadores el diagnóstico es casi seguro.

Por último, cuando el paquete analizado se compone de varios módulos la salida del programa se estructura a partir de un índice que ofrece una valoración global del proyecto, presenta un resumen de los ficheros analizados y proporciona un enlace a los análisis individualizados de los ficheros.

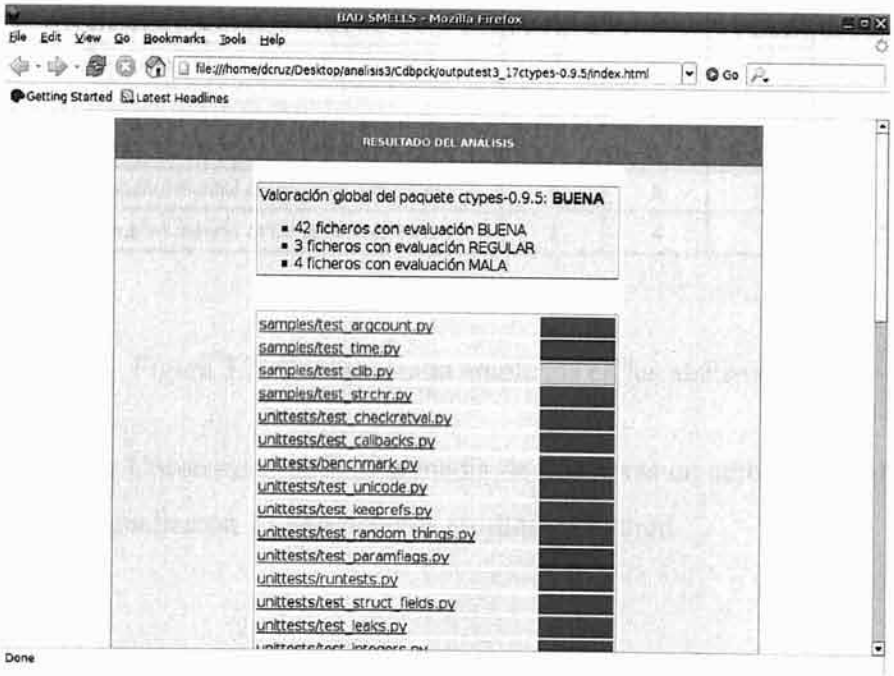


Figura 3.6: Ejemplo de salida del índice en HTML.

3.5. Quinto paso: estudio de paquetes de Debian Sarge

Una vez superadas las fases anteriores realizamos con nuestra aplicación un análisis exhaustivo a proyectos de software libre que tuviesen código escrito en Python. Se decidió analizar

paquetes pertenecientes a la distribución GNU/Linux *Debian Sarge* (ver la lista de paquetes completa en el apéndice C). Para poder realizar las pruebas mi tutor, Gregorio Robles, me proporcionó una cuenta en una máquina de la universidad con la suficiente potencia de cálculo para que se agilizase el análisis de todos los paquetes.

Para automatizar el análisis de los paquetes se escribió un *script* que iba copiando al directorio de trabajo los paquetes de Debian que estaban almacenados en la máquina y ejecutaba la aplicación *badsmcheck*. Los paquetes se fueron analizando por orden alfabético y una vez obtenidos los resultados se almacenaron en una computadora doméstica.

Se realizaron tres análisis con tres configuraciones distintas a 655 paquetes. La idea era realizar tres pasadas con diferente granularidad a cada paquete. La configuración usada en las pruebas se puede observar en la tabla siguiente:

Parámetro	análisis 1	análisis 2	análisis 3
Longitud máxima de las clases	18	25	50
Longitud máxima de los métodos	18	25	50
Numero máximo de parámetros de los métodos	2	3	5
Longitud máxima de los comentarios	6	8	18
Longitud máxima de las líneas duplicadas	6	8	18
Longitud máxima de las sentencias Switch	2	4	8

Figura 3.7: Configuración empleada en los análisis.

El servidor de la Universidad tardó una media de dos horas en completar cada uno de los análisis. En total se analizaron 17.846 ficheros escritos en Python.

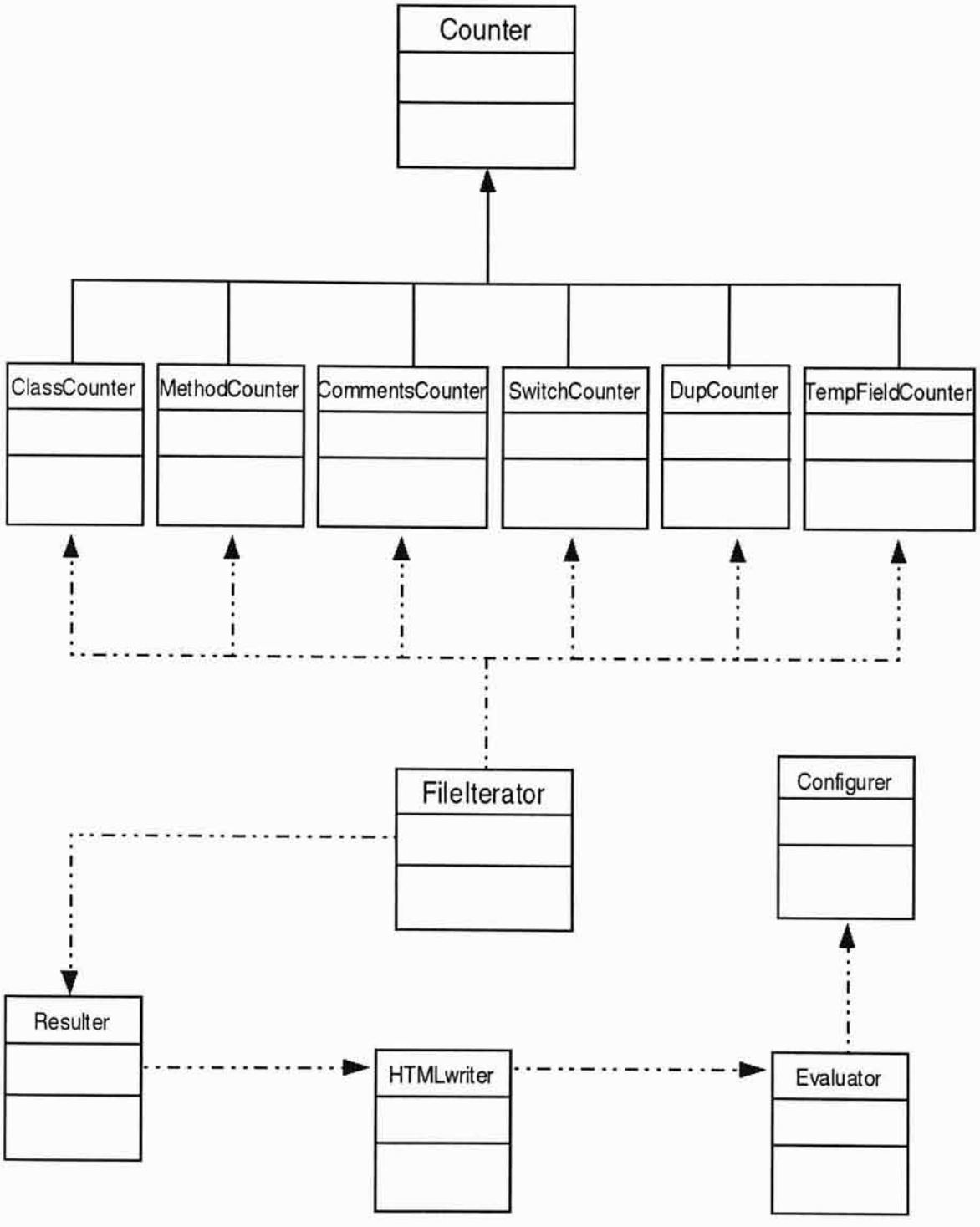


Figura 3.1: Diagrama de clases de la aplicación.

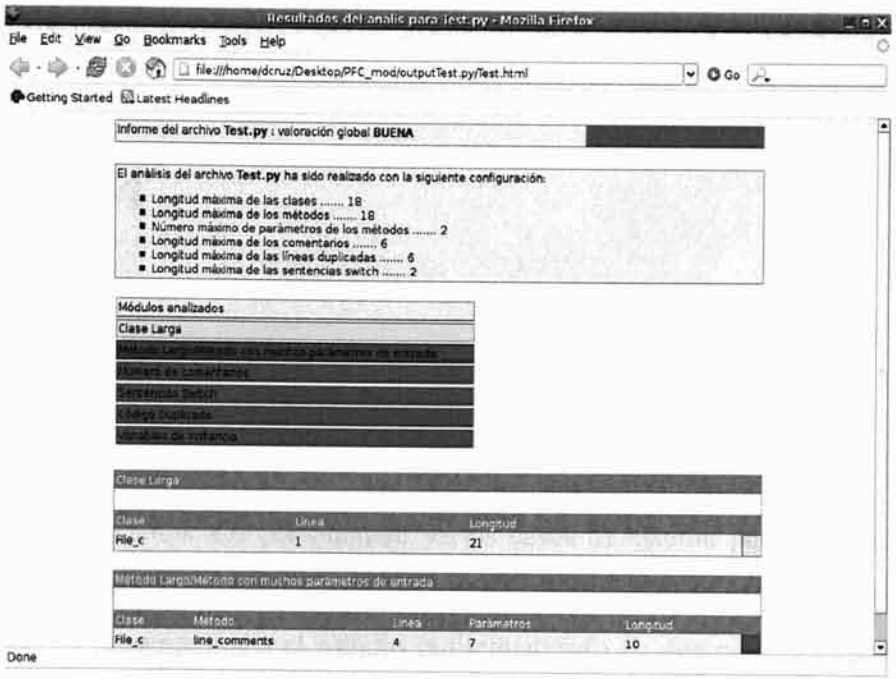


Figura 3.2: Ejemplo de salida del programa en HTML.

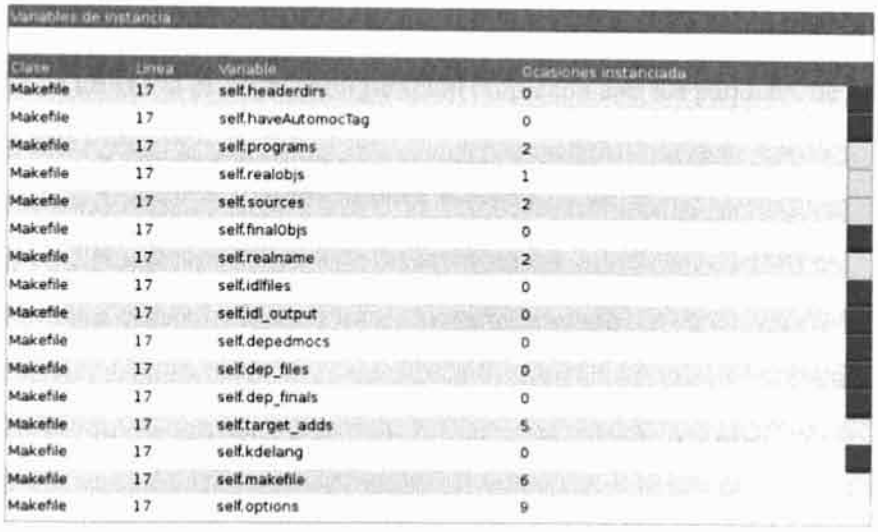


Figura 3.5: Ejemplo de variables de instancia cuestionables.

Capítulo 4

Casos de estudio

En esta sección vamos a ir presentando varios casos de estudio para mostrar qué ventajas pueden obtenerse si se emplea nuestra aplicación. Empezaremos con el propio código del proyecto, después presentaremos el estudio individualizado de dos paquetes (**python-ldap** y **pybliographer**) para finalizar con los resultados globales de todos los paquetes de *Debian* que han sido analizados

4.1. Primer caso de estudio: el PFC

Llega el momento de usar la aplicación con nuestro propio código para saber a qué *huele*. Para el análisis se utilizó la primera configuración empleada con los paquetes de *Debian Sarge*:

- Longitud máxima de las clases 18
- Longitud máxima de los métodos 18
- Número máximo de parámetros de los métodos 2
- Longitud máxima de los comentarios 6
- Longitud máxima de las líneas duplicadas 6
- Longitud máxima de las sentencias switch 2

Figura 4.1: Configuración usada en el análisis del PFC.

Los resultados generales que obtuvimos los podemos ver en la página siguiente.

¡Qué aparentemente perfecto es nuestro código! Si levantamos la tapa y pasamos a inspeccionar cada uno de los módulos que forman parte del PFC nos encontramos con que existen *bad smells* en el código (¡por supuesto!).

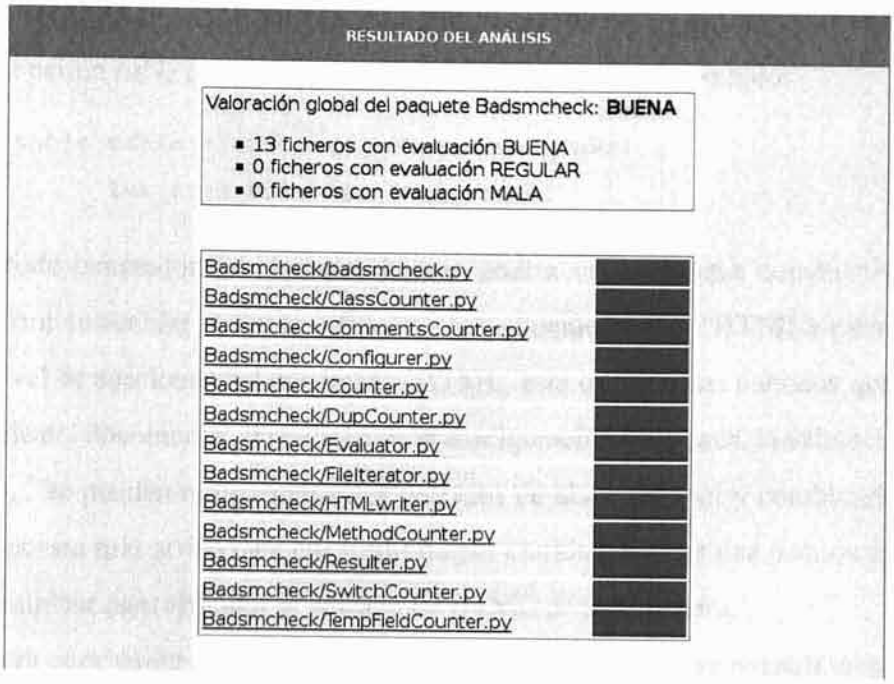


Figura 4.2: Valoración global de badsmcheck.

El más reiterativo de todos es el de Clase Larga, algo normal si consideramos que la longitud de las clases que entran dentro de la configuración es de 18 líneas o menos (*rice grain programming*). A nivel estadístico se presentan a contiuación la longitud de las clases que hacen funcionar a este PFC:

Clase Larga

Clase	Linea	Longitud
Counter	1	70
ClassCounter	5	62
CommentsCounter	4	138
MethodCounter	5	131
SwitchCounter	5	76
TempFieldCounter	4	118
FileIterator	12	100
Evaluator	9	106
HTMLwriter	11	311
Resulter	5	55
Configurer	4	48

Figura 4.3: Longitud de las clases de badsmcheck.

De todas las clases escritas **HTMLwriter** destaca por su longitud. Es la clase que escribe la salida en HTML y es probable que se pudiera compactar ya que sus métodos de escritura

son poco atómicos. Una posible mejora para estructurar su código sería establecer métodos que modelaran el patrón *table* que se repite en todos los informes, por ejemplo:

```
def write_table(table_tittle,num_columns,num_rows,
                lst_item_cols,lst_item_rows):
```

Este método generador debidamente iterado podría ser un bloque constructivo que sería reutilizado para ensamblar nuestros informes. Actualmente la clase **HTMLwriter** está modularizada a nivel de secciones del documento HTML, esto es, tenemos métodos que escriben la cabecera y pie del documento, el resumen de la configuración empleada, la valoración global de los módulos... Se pueden refinar más estos métodos de nivel superior y combinarlos con otros como el propuesto más arriba para conseguir mayor claridad. Esta es una posible estrategia que podríamos emplear para articular el volcado de HTML de otra manera.

La primera conclusión obtenida de esta pequeña disertación sobre nuestro propio código es la siguiente: a no ser que los *bad smells* sean de manual, la refactorización del código es más fácil cuando se conoce bien el código o se ha estudiado debidamente. Aún así, bien es cierto que la herramienta podría ayudar incluso con código desconocido, ya que resulta en un ahorro de tiempo considerable. Esto se debe a que aunque proporcione casos que haya que estudiar con mayor detenimiento, al menos nos ahorra la tarea de identificarlos.

En el caso del análisis de los paquetes *Debian* sólo podemos ofrecer de manera macroscópica un resumen de lo que hemos encontrado, ofreciendo estadísticamente algunos datos. Habría que ir paquete a paquete y estudiar el código para poder considerar si ha de ser modificado para mejorarlo. En cualquier caso, éste es un buen ejemplo de uso de la aplicación: se detecta un defecto y después de analizarlo se presenta una posible estrategia de refactorización.

Vamos a seguir examinando nuestro código. El siguiente *bad smell* donde nuestro proyecto presenta ciertas sombras es en el tamaño y número de parámetros de entrada de sus métodos. En la tabla de la página siguiente podemos ver aquellos métodos que se salen de la configuración.

Como podemos observar, algunos de los métodos no entran dentro de la configuración con la que se ejecutó la aplicación por unas pocas líneas o parámetros de entrada. Sin embargo sí podemos analizar algunos métodos, por ejemplo, el método **__labeling** de la clase **CommentsCounter**. Es un método que se emplea para "etiquetar" las listas que usamos para almacenar los comentarios que se hayan detectado. Almacenamos la clase y el método al que pertenece el comentario, su posición en el código y su longitud. ¿Qué le pasó a este método para que

Método Largo/Método con muchos parámetros de entrada

Clase	Método	Linea	Parámetros	Longitud
CommentsCounter	__prelabeling	15	3	18
CommentsCounter	__labeling	34	2	41
CommentsCounter	process	92	2	22
CommentsCounter	post_process	115	0	27
MethodCounter	__labeling	19	2	19
MethodCounter	__params_num	39	1	25
MethodCounter	__open_params_count	65	1	19
SwitchCounter	__labeling	10	2	21
SwitchCounter	__switch_up	32	1	22
TempFieldCounter	get_instanceVars	57	2	19
TempFieldCounter	post_process	99	0	23
FileIterator	__read_lines	63	0	36
Evaluator	get_grade	31	3	34
Evaluator	get_grade_tempfields	66	4	16
HTMLwriter	write_index	48	4	62
HTMLwriter	writer	192	4	44
HTMLwriter	write_doppels	237	3	45
HTMLwriter	write_tempfields	283	3	39
Resulter	__init__	7	3	5
Resulter	print_analysis	16	0	44
Configurer	__init__	6	0	25

Figura 4.4: Métodos de badsmcheck.

aumentara en longitud? Pues que en una primera implementación este método no contemplaba el hecho importante de que un comentario podía estar fuera de una clase y/o un método. Al arreglar este defecto hubo que introducir más líneas de código que lo hicieron alcanzar su dimensión y funcionalidad actual.

Otro método a destacar es **__read_lines** que pertenece a **FileIterator**. Es un método muy importante, ya que es el que instancia a todas las clases *contadoras* y las va alimentando con líneas de código para que el método polimórfico **process** entre en acción. Al finalizar el análisis de los ficheros pasa a la clase **Resulter** los resultados (listas de listas en realidad) obtenidos.

Pasamos a los métodos de **HTMLwriter**. Los métodos de esta clase que hemos destacado en la tabla de más arriba adolecen del problema que ya hemos discutido anteriormente. Se podrían introducir métodos de más bajo nivel que éstos para realizar la escritura a disco del fichero HTML y así poder extraer algo de su código consiguiendo aligerarlos y hacerlos más modulares y claros.

El caso de **write_index** es el más obvio de todos. Es el método que se encarga de escribir el fichero índice que ofrece el resultado de los análisis, la valoración global y permite navegar por los distintos ficheros que se han analizado. Su problema es que debería ser modulariza-

do introduciendo el código HTML que escribe dentro de funciones auxiliares que podrían ser reutilizadas por otros métodos de la clase. El siguiente método poco aseado es **writer**, que se encarga de escribir líneas como ésta:

Ya hemos comentado las posibles estrategias para refactorizarlo. Los últimos dos métodos de esta clase son **write_doppels** y **write_tempfields** que están especializados en escribir las líneas de código duplicadas y las que se refieren a las variables de instancia respectivamente. Esto se debe a que el formato de las listas resultado para el código duplicado y las variables de instancia temporales difiere del resto, que son tratadas por **writer**.

En la clase **Resulter** tenemos el método **print_analysis** que es el que se encarga de ensamblar los informes usando los resultados que le ha pasado **FileIterator** y los métodos de **HTMLwriter**.

Por último, tenemos a **__init__** que es el contructor de la clase **Configurer**, es el que se encarga de mantener la configuración de la aplicación en el fichero *settings.cfg*.

A continuación presentamos los resultados que hemos obtenido en los análisis de las sentencias *switch* en nuestro código:

Sentencias Switch

Clase	Método	Línea	Longitud
Evaluator	get_grade	31	3
HTMLwriter	write_objects_rslts	177	4

Figura 4.5: Sentencias switch en nuestro código

No se trata de prohibir el uso de bloques *if-elif*, pero en este caso podemos introducir mejoras en el código. Veamos una parte del código del método **get_grade**:

```
[...]
if isinstance(obj, ClassCounter):
  if obj.get_obj_length(lst) > (2 * self.conf.get_clm_limit()):
   rate = 0
   self.get_grade_OK_SO_BAD(0,'BAD')
  else:
   self.get_grade_OK_SO_BAD(0,'SO')
```

```
elif isinstance(obj, MethodCounter):
[...]
elif isinstance(obj, CommentsCounter):
[...]
```

Los bloques *elif* siguientes van evaluando si el objeto que se le pasa al método como parámetro es una instancia de alguna de las clases *contadoras* y, dependiendo de sus características, así es evaluado. No es un buen código. Como tenemos al propio objeto, mejor que establecer una serie de condicionales, tendría que ser el objeto el que se evaluara a sí mismo ya que puede usar los métodos públicos de la clase **Configurer** para poder comparar sus resultados con la configuración establecida.

En cuanto al método **write_objects_rslts** veamos su código:

```
def write_objects_rslts(self,lstobjs,fd):
  if isinstance(lstobjs, ClassCounter):
   self.writer(lstobjs,fd,4,0)
  elif isinstance(lstobjs, MethodCounter):
   self.writer(lstobjs,fd,6,1)
  elif isinstance(lstobjs, CommentsCounter):
   self.writer(lstobjs,fd,5,2)
  elif isinstance(lstobjs, SwitchCounter):
   self.writer(lstobjs,fd,5,3)
  elif isinstance(lstobjs, DupCounter):
   self.write_doppels(lstobjs,fd,4)
  else:
   self.write_tempfields(lstobjs,fd,5)
```

En fin, estamos en la misma situación de antes. Si tenemos el objeto, que sea él mismo quien nos proporcione sus datos mediante un método sobrecargado y así ocultamos los detalles de implementación. Porque los "números mágicos" que aparecen en la llamada al método **writer** son índices de la lista resultado que tiene cada objeto en su interior. Todo eso hay que encapsularlo apropiadamente. Posible solución:

```
def write_objects_rslts(self,lstobjs,fd):
```

```
for obj in lstobjs:
   self.writer(obj.get_printable_lst(),fd)
```

Mucho mejor. Introducimos un método en cada una de las clases *contadoras* que ofrece una lista con los datos que tienen que salir en los informes al método **writer** que es el que escribirá el código HTML adecuado. Así la clase **HTMLwriter** no conocerá ningún detalle de la implementación de los atributos de las clases descendientes de **Counter**. Con lo cual si fuera necesario realizar alguna modificación como, por ejemplo, añadir otra clase contadora no haría falta tocar el método **write_objects_rslts**. Simplemente tendría un elemento más en la lista de objetos que maneja y comprobará que es autocontenido y le proporciona sus resultados.

4.2. Segundo caso de estudio: python-ldap_2.0.4 y pybliographer_1.2.6.2

Para este caso de estudio escogimos dos paquetes de *Debian Sarge* y ejecutamos nuestra aplicación para observar los resultados que obteníamos, la configuración fue la misma que usamos para analizar el código de nuestro PFC. En la realización de este análisis procedimos de forma distinta a como se realizaron los análisis del primer y tercer caso de estudio al considerar el proyecto como un único elemento a estudiar.

4.2.1. python-ldap_2.0.4

El paquete **python-ldap_2.0.4** es un módulo que forma parte de las bibliotecas de soporte del protocolo LDAP de OpenLDAP. LDAP (Lightweight Directory Access Protocol) es un protocolo de red que permite el acceso a un servicio de directorio ordenado y distribuido para buscar diversa información en un entorno de red. LDAP puede considerarse una base de datos (aunque no es una base de datos relacional) sobre la que pueden realizarse consultas, realmente la información se almacena en directorios LDAP que están fuertemente optimizados para el rendimiento en lectura. El protocolo LDAP es utilizable por distintas plataformas y está basado en estándares, de ese modo las aplicaciones no necesitan preocuparse por el tipo de servidor en

que se hospeda el directorio.

OpenLDAP se trata de una implementación libre del protocolo que soporta múltiples esquemas por lo que puede utilizarse para conectarse a cualquier otro LDAP.

El paquete **python-ldap_2.0.4** contiene 34 ficheros con código escrito en Python. Los resultados que obtuvimos con la aplicación fueron:

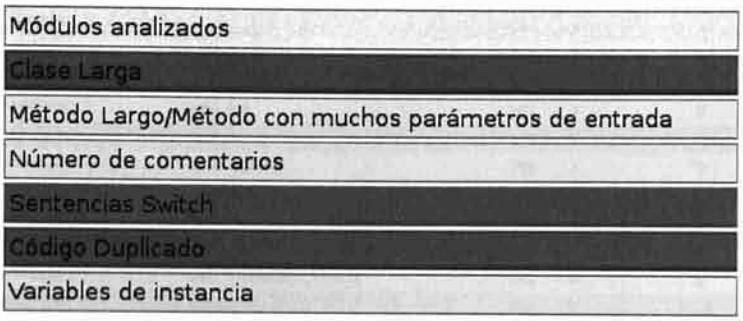


Figura 4.6: Resultados globales del paquete python-ldap_2.0.4

Como podemos observar el código del paquete **python-ldap_2.0.4** presenta dos defectos graves: el tamaño de sus clases y la presencia de código duplicado. Respecto al tamaño de sus clases hay que tener en cuenta que, al igual que en el primer caso de estudio, la configuración de la aplicación en este aspecto era muy restrictiva por lo que algunas de las clases que han "salido en la foto" presentan una longitud razonable y realmente no tenemos nada que objetar. Sin embargo nos hemos encontrado en el archivo **ldapobject.py** una clase, **SimpleLDAPObject**, que ocupa 557 líneas. Esta clase recubre a un objeto LDAP y ofrece ni más ni menos que 53 métodos. Estos métodos gestionan prácticamente todas las funcionalidades del protocolo LDAP: conectarse a un servidor LDAP, realizar búsquedas, añadir y borrar entradas en el directorio, etc.

Es esta una clase muy pesada que centraliza todas las operaciones LDAP y que está diseñada para convertirse en un clase delegada que podemos instanciar para atacar un servidor LDAP. Desde el punto de vista del diseño de la clase los desarrolladores del paquete han creado una clase que centraliza las funcionalidades del protocolo LDAP. Tener una clase tan grande dificulta su mantenimiento por ello se deberían extraer subclases que agruparan los métodos que tengan características similares. Es decir, una subclase que implemente los mecanismos de conexión a un servidor LDAP, otra para realizar consultas, otra para modificar la información almacenada

en el servidor. Podemos expresarlo de otra forma: a partir de la gran clase obtener otras más pequeñas que estén especializadas en un aspecto concreto del protocolo LDAP.

En la figura siguiente podemos ver algunos de los métodos de la clase **SimpleLDAPObject**

Método Largo/Método con muchos parámetros de entrada

Clase	Método	Linea	Parametros	Longitud
SimpleLDAPObject	__init__	58	4	12
SimpleLDAPObject	_ldap_call	77	3	28
SimpleLDAPObject	abandon_ext	122	3	11
SimpleLDAPObject	add_ext	137	4	9
SimpleLDAPObject	add_ext_s	147	4	3
SimpleLDAPObject	simple_bind	165	4	5
SimpleLDAPObject	simple_bind_s	171	4	6
SimpleLDAPObject	bind	178	3	6
SimpleLDAPObject	bind_s	185	3	6
SimpleLDAPObject	sasl_interactive_bind_s	192	4	5
SimpleLDAPObject	compare_ext	198	5	19
SimpleLDAPObject	compare_ext_s	218	5	9

Figura 4.7: Algunos métodos de la clase SimpleLDAPObject

Gracias a la configuración tan exhaustiva han salido prácticamente todos los métodos en el informe individualizado del fichero **ldapobject.py**. Los métodos de la figura que aparecen en rojo tienen ese color por su número de parámetros (recordemos la configuración, número máximo de parámetros de entrada fijado en dos).

Seguimos con la inspección de código, el fichero **async.py** contiene una clase llamada **AsyncSearchHandler** que sirve para procesar de forma asíncrona los resultados que nos devuelve un servidor LDAP. Esta clase ocupa 115 líneas, algo en principio no demasiado preocupante. Sin embargo observando su código nos encontramos con lo siguiente. Tiene declarados 5 métodos de los cuales:

- uno, **startSearch**, tiene 8 parámetros de entrada, 15 líneas de comentarios para explicar cada uno de los parámetros de entrada y una línea de código para inicializar una variable.
- otro método, **processResults**, que en principio no presenta ningún problema.
- tres métodos, **preProcessing**, **postProcessing** (invocados desde **processResults**) que únicamente están declarados. El método **_processSingleResult** tampoco está implementado.

```
def preProcessing(self):
```

```
        """
        Do anything you want after starting search but
        before receiving and processing results
        """
    def postProcessing(self):
        """
        Do anything you want after receiving and processing results
        """
    def _processSingleResult(self,resultType,resultItem):
        """
        Process single entry
        resultType
            result type
        resultItem
            Single item of a result list
        """
        pass
```

¿Qué está pasando? Si seguimos explorando el código nos encontramos más abajo la clase **FileWriter**, descendiente de **AsyncSearchHandler** que sólo implementa los dos métodos métodos **preProcessing** y **postProcessing**.

```
def preProcessing(self):
    """
    The headerStr is written to output after starting search but
    before receiving and processing results.
    """
    self._f.write(self.headerStr)

  def postProcessing(self):
    """
    The footerStr is written to output after receiving and
    processing results.
    """
    self._f.write(self.footerStr)
```

Muy escueto. *self._f* es un descriptor de archivo y *self.headerStr* y *self.footerStr* son dos strings. ¿Es necesaria la clase **FileWriter**? Estábamos rastreando un tipo de *bad smell* y nos

hemos tropezado con otro que nuestra aplicación no detecta *a priori*. ¿Es **FileWriter** una clase perezosa? Estimamos que sí, ya que si la implementación de los dos métodos que hemos expuesto fuese compleja podría justicarse su extracción de la clase **AsyncSearchHandler**, pero en este caso se podría colapsar la clase **FileWriter** dentro de la clase padre.

El método **_processSingleResult** que comentábamos antes está implementado en la clase **List** que es hija de **AsyncSearchHandler** y que es la que recoge los resultados de todas las búsquedas realizadas en el servidor LDAP.

En cuanto al código duplicado lo que la aplicación detectó fue la existencia de un método llamado def **__repr__(self):** en el fichero **ldapurl.py** que se encuentra duplicado en tres clases distintas: **LDAPUrlExtension**, **LDAPUrlExtensions** y **LDAPUrl**. El código del método en cuestión es:

```
def __repr__(self):
    return '<%s.%s instance at %s: %s>' % (
      self.__class__.__module__,
      self.__class__.__name__,
      hex(id(self)),
      self.__dict__
    )
```

Lo curioso es que este método privado no es usado en ningún momento en el código de las clases con lo cual puede ser eliminado.

4.2.2. pybliographer_1.2.6.2

El paquete **pybliographer_1.2.6.2** es una herramienta que nos permite manipular bases de datos bibliográficas, permitiéndonos visualizarlas, editarlas y realizar búsquedas. También es capaz de exportar las bases de datos bibliográficas a HTML. Pybliographer soporta diferentes sistemas de información bibliográfica como: BibTeX, Medline, Ovid y Refer.

Este es el aspecto que presenta un archivo de información bibliográfica (**.bib**) en BibTeX, en este ejemplo contiene una entrada que describe un manual de matemáticas:

```
@Book{abramowitz+stegun,
  author =        "Walter Larawitz and Irene A. Stegun",
  title =         "Handbook of Mathematical Functions with
                   Formulas, Graphs, and Mathematical Tables",
  publisher =     "Dover",
```

```
  year =          1964,
  address =       "New York",
  edition =       "ninth Dover printing, tenth GPO printing"
}
```

Una vez completado el análisis del paquete los resultados que obtuvimos a nivel de módulos fueron:

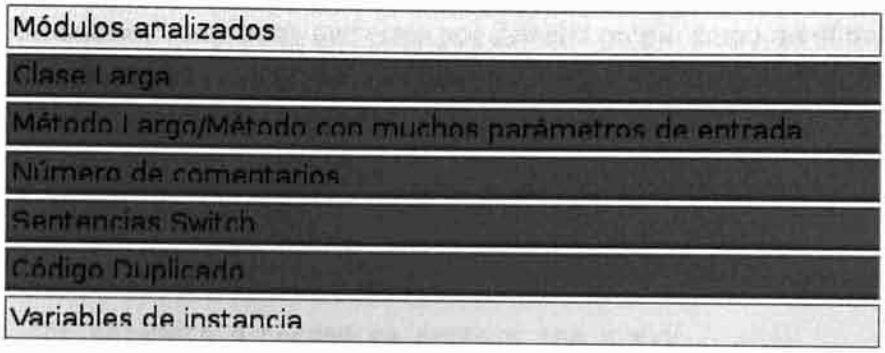


Figura 4.8: Resultados globales del paquete pybliographer_1.2.6.2

Son unos resultados a primera vista alarmantes, veamos qué ha provocado que la evaluación haya resultado tan negativa empezando por el tamaño de alguna de sus clases. En el fichero **Document.py** nos encontramos con la clase **Document** que ocupa 782 líneas (nada menos). Esta clase define el objeto *documento* de la GUI, sus 46 métodos implementan todas las operaciones que cuelgan del menú. El problema es que en esta clase no sólo se han implementado las operaciones que nos permiten manipular un documento tales como crear, salvar, abrir o cerrar sino que aparecen otras que no tienen la misma semántica. Por ejemlo: cortar y pegar, selecionar todo, omitir cambios, acerca de, eventos de teclado, refrescar resultados, búsquedas en BBDD bibliográficas, etc. Creemos que es mejor reorganizar en subclases aquellos métodos que ofrezcan funcionalidades relacionadas desde el punto de vista de la interfaz de usuario en vez de tenerlos todos en la misma clase.

En el fichero **Fields.py** nos encontramos con otra clase, **FieldsDialog**, que también forma parte de la GUI y que tiene una logitud importante: 422 líneas. Esta clase se encarga de ofrecer un cuadro de diálogo que nos permite configurar la estructura de los campos de las citas bibliográficas que creemos. El problema que observamos en esta clase es que se han mezclado dos elementos de diseño, ya que tenemos métodos que afectan al comportamiento externo

del cuadro de diálogo (*on_close, on_add, on_remove*) junto con los métodos que nos ofrecen la funcionalidad real que nos permite configurar la estructura de la información bibliográfica que estamos generando. ¿No sería mejor dejar la clase **FieldsDialog** con todos los eventos propios de una interfaz implementados en ella y crear una subclase que nos genere cada una de las pestañas que nos servirán para configurar la estructura de los campos?

En cuanto al tamaño de los métodos, si bien nos encontramos que muchos han sido incluidos en el informe debido a la configuración tan restrictiva que hemos empleado y no presentan aspectos "cuestionables", hay otros que están por derecho propio como **medline_query**. Este método se encuentra incluido en el fichero **Query.py**, ocupa 188 líneas y tiene 16 parámetros de entrada.

```
def medline_query (keyword,maxcount,displaystart,field,abstract,
epubahead,pubtype,language,subset,agerange,humananimal,gender,
entrezdate,pubdate,fromdate,todate):
# note all the parameters needed to perform the query
}
```

Aún hay más. En su código detectamos 126 sentencias *elif*. Todo son *elifs* en este método, como curiosidad su código ha sido incluido en el apéndice D. ¿Cómo refactorizamos este método? Borrándolo. No es un método comprensible. En todo el código fuente del paquete hay 144 sentencias *elif*, eso supone que **medline_query** protagoniza el 87,5 % del *bad smell* denominado "Cláusulas Switch" o "Sentencias Switch".

Por último, pasamos a analizar la incidencia del código duplicado en el paquete. Nos encontramos con que el método **author_desc** está duplicado en los ficheros **Generic.py** y **abbrv.py**, pero existen más coincidencias entre estos dos ficheros, ya que el método **string_key** está también presente en los mismos ficheros. La diferencia es que el que se encuentra en **abbrv.py** tiene un parámetro de entrada más y una línea de código diferente respecto al de **Generic.py**. Esto no acaba aquí, presentamos las sutiles diferencias del método **create_string_key** presente en **Generic.py** y **abbrv.py**:

En Generic.py:

```
def create_string_key (database, keys, fmt):
    table = {}
    for key in keys:
        s = string_key (database [key], fmt, table)
```

```
        table [s] = key

    skeys = table.keys ()
    skeys.sort ()

    return table, skeys
```

En abbrv.py:

```
def create_string_key (database, keys, fmt):
    table = {}
    oldnew = {} <----
    for key in keys:
        s = string_key (database [key], fmt, table, oldnew)
        table [s] = key
        oldnew [database [key].key.key] = s <----

    skeys = table.keys ()
    skeys.sort ()
    pybtextvar.oldnew = oldnew <----

    return table, skeys
}
```

Los siguientes métodos también están duplicados: **standard_date**, **last_first_full_authors**, **first_last_full_authors**, **full_authors**, **last_first_initials_authors**, **first_last_initials_authors** e **initials_authors**

Tenemos más código duplicado en este paquete: esta vez el fichero **Generic.py** coincide con el fichero **apa4e.py**. El código es idéntico salvo por 3 líneas.

¿Qué ocurre? Estos métodos no están contenidos dentro de ninguna clase, en principio pueden importarse y ser usados en otros módulos. Por lo tanto si lo que queremos es una biblioteca con métodos aislados entonces alguno de los tres ficheros sobra. Sin embargo, existen métodos que están en **Generic.py** y no están en **abbrv.py** y además algunos de los que están duplicados

presentan pequeñas diferencias, ¿es que uno de los ficheros es una versión obsoleta y ya no se usa?. Los métodos de estos tres ficheros sirven para controlar el estilo de los ficheros XML que gestiona la aplicación. Haciendo un búsqueda con el comando **find** en el directorio donde se descomprimió el paquete **pybliographer** seguimos encontrando sorpresas. Resulta que el módulo **Generic.py** no es usado por ningún otro, sin embargo descubrimos que **abbrv.py** y **apa4e.py** sí son usados. Armados con toda esta información podríamos borrar **Generic.py** ya que todo parece indicar que se trata de un módulo obsoleto.

Pero todavía queda por resolver el código duplicado de **abbrv.py** y **apa4e.py**. Deberíamos crear una clase que recogiera los métodos comunes de los dos ficheros y una subclase que tratara las excepciones.

En este caso la evaluación de *badsmcheck* del código del paquete **pybliographer** es acertada: huele mal.

4.3. Tercer caso de estudio: Debian Sarge

Para poder tabular los datos obtenidos en los tres análisis que realizamos en el servidor de la universidad se ha creado un *script* que *parsea* los informes HTML y extrae los resultados relevantes para nuestro estudio que se ha realizado con los paquetes de Debian Sarge listados en el apéndice C. La forma de trabajar del *script* es la siguiente: localiza la sección titulada **Módulos analizados** de los informes y va llevando un registro de la calificación obtenida en cada uno de los *bad smells*. Las tablas que presentamos a continuación tienen dos columnas: una roja y otra amarilla, lo que recoge la columna roja se refiere a los datos considerados por la aplicación como graves y la columna amarilla contiene los datos de carácter menos grave, tal y como se representa en los informes de los análisis.

El primer análisis era el más restrictivo de todos y el *bad smell* más destacado con diferencia (38,43 %) es el que se refiere al tamaño de las clases. El siguiente *bad smell*, con un 28,02 %, es el de número de comentarios en el código, tenemos que tener en cuenta que la aplicación no distigue los comentarios que documentan los módulos (proporcionando información sobre el autor, fecha de creación, etc.) de los meramente informativos, con lo cual los resultados obtenidos con esta configuración deben ser evaluados con precaución. Sin embargo, lo que nos orienta en este caso es el hecho de que si el comentario está localizado en las primeras líneas

Bad Smells detectados en el análisis 1		%		%
Clases Largas	8330	38,43	1704	11,23
Método Largo / Método con muchos parámetros de entrada	4510	20,80	6787	44,74
Número de Comentarios	6074	28,02	3919	25,83
Sentencias Switch	469	2,16	1425	9,39
Código Duplicado	1453	6,70	N/A	N/A
Variables de Instancia	842	3,89	1334	8.81

Figura 4.9: Resultados del primer análisis.

probablemente se trate de comentarios introducidos para documentar el código.

Los siguientes *bad smells* que más aparecen son los métodos largos y los que tienen muchos parámetros de entrada con un 20,80 %. Como en la aplicación se controlan de forma conjunta, en el informe que obtenemos como salida se sitúan en la misma línea sin diferenciar cuál de los dos (sino ambos) es el que ha motivado su inclusión en el mismo. Por ello hemos tenido que realizar otro *script* post-análisis para que nos ayudara a discernir con qué frecuencia se da cada uno de ellos. El *script* lee las líneas del informe donde se recoge la información que nos interesa y extrae el número de parámetros y longitud del método "cuestionable", comprueba cuál de los dos parámetros es el culpable de su presencia en el informe y, por último, toma nota de su calificación (mala o regular) para ir manteniendo un registro. Esta operación la repite con los informes de todos los paquetes que se han analizado.

Los resultados que arrojó este *script* fueron los siguientes:

Desglose del Bad Smell de Métodos del análisis 1	%	%
Método Largo	65,82	57,14
Método con muchos parámetros de entrada	34,18	42,86

Figura 4.10: Desglose del *bad smell* de métodos para el primer análisis.

Los resultados señalan que son los métodos largos los que suelen presentar con mayor frecuencia el defecto que motiva su inclusión en los informes que ha generado la aplicación. La primera tabla nos muestra que cuando la valoración de los *bad smells* es regular el tanto por ciento de métodos "cuestionables" sube a un 44,74 %.

En cuanto al código duplicado la aplicación no evalúa este *bad smell* en términos de malo o regular, para ella la presencia de código duplicado es malo y no admite gradación. La incidencia de este *bad smell* no es muy alta en el código (un 6,70 %), considerando lo restrictiva que era la configuración en este análisis.

La presencia de los últimos *bad smells* que hemos analizado, las sentencias *Switch* y las variables de instancia temporales, es muy baja en los paquetes analizados con un 2,16 % y un 3,89 % respectivamente.

Los resultados del segundo y tercer análisis son paralelos manteniéndose, en general, las mismas proporciones que se daban en el primer análisis, aunque los valores de cada uno de los *bad smells* disminuyen conforme la configuración empleada es más laxa. Lo que sucede con los resultados de éstos análisis es que los *bad smells* que aparecen en los informes obtienendo una mala calificación es porque presentan defectos importantes. Tenemos que destacar que las variables de instancia temporales no dependen de la configuración con la que hayamos ejecutado los análisis y por ello los valores recogidos en las tres tablas no varían. A continuación ofrecemos los resultados de los dos últimos análisis:

Bad Smells detectados en el análisis 2		%		%
Clases Largas	7198	43,65	2111	14,34
Método Largo / Método con muchos parámetros de entrada	3117	18,90	6191	42,05
Número de Comentarios	4541	27,54	4278	29,06
Sentencias Switch	113	0,68	808	5,49
Código Duplicado	680	4,12	N/A	N/A
Variables de Instancia	842	5,11	1334	9,06

Figura 4.11: Resultados del segundo análisis.

Desglose del Bad Smell de Métodos del análisis 2	%	%
Método Largo	78,10	64,57
Método con muchos parámetros de entrada	21,90	35,43

Figura 4.12: Desglose del *bad smell* de Métodos para el segundo análisis.

Observamos que al igual que en el primer análisis el *mal olor* más destacado en los análisis

Bad Smells detectados en el análisis 3		%		%
Clases Largas	4780	50,95	2439	23,05
Método Largo / Método con muchos parámetros de entrada	1667	17,76	3671	34,69
Número de Comentarios	1906	20,36	2789	26,36
Sentencias Switch	51	0,54	319	3,01
Código Duplicado	134	1,42	N/A	N/A
Variables de Instancia	842	8,97	1334	12,89

Figura 4.13: Resultados del tercer análisis.

Desglose del Bad Smell de Métodos del análisis 3	%	%
Método Largo	78,46	69,55
Método con muchos parámetros de entrada	21,54	30,45

Figura 4.14: Desglose del *bad smell* de Métodos para el tercer análisis.

2 y 3 es el que se refiere al tamaño de las clases, seguido del número de comentarios y del tamaño de los métodos (o el número de sus parámetros de entrada). Nos gustaría destacar que la incidencia de los *bad smells* más preocupantes como pueden ser sentencias *switch* y código duplicado es la más baja respecto a otros *malos olores*.

En resumen podemos concluir que los paquetes *Debian Sarge* presentan mayoritariamente como *bad smell* clases largas. Ya hemos visto que las clases que tratan de hacer muchas cosas contienen a menudo demasiadas variables de instancia y suelen contener código duplicado. Siempre que se pueda y esté justificado es recomendable intentar desacoplar las clases grandes en clases más pequeñas. Una de las ventajas de la programación orientada a objetos es que el diseño de los programas pueden organizarse de una forma arquitectónica que favorece el desarrollo y mantenimiento del código.

En cuanto al número de comentarios la incidencia de este *mal olor* en los paquetes analizados no es concluyente y puede ser obviada a falta de un análisis pormenorizado de todos los paquetes. Lo que hemos detectado es que la mayor parte de los comentarios corresponden a la documentación habitual y necesaria que identifican al autor o autores, versión, declaraciones sobre la naturaleza libre del código que estamos estudiando así como breves explicaciones sobre la funcionalidad de algún método. Aunque siempre podemos encontrar cosas como éstas:

```
# if date limits are provided, then the following will be added to keyword
```

```
# I will only allow this if the relative entrez date is not specified above,
#hence the elif command
# This is where I used the time function, gmtime()
# to get the current global mean time
```

Sobre el tamaño de los métodos y el número de parámetros de entrada su presencia en el código de los paquetes es destacable. Con la programación orientada a objetos podemos construir métodos que deleguen operaciones en otros métodos más pequeños y especializados. No hay que tener miedo a descomponer demasiado los métodos es más, una estrategia que podemos emplear al programar es que si necesitamos comentar un fragmento de código hay que extraerlo y crear un método. En cuanto al número de parámetros de entrada volvemos a recordar que no es necesario pasarle todos los parámetros a un método, es más fácil pasar un objeto que encapsule todos los datos que el método necesite para realizar su función. Aunque si no queremos establecer dependencias entre objetos tendremos que pasar los parámetros de la forma habitual, el problema es que si la lista de parámetros es muy grande o cambian a menudo puede que haya que replantearse la estrutura de clases.

La presencia de código duplicado es baja pero tenemos que recordar que los análisis realizados detectaban la presencia de códido duplicado únicamente dentro de los ficheros. Con lo cual es muy probable que exista código diseminado en distintos archivos de un paquete que se le haya escapado a la aplicación.

Sobre el último *bad smell* que detecta la aplicación, las variables de instancia temporales, conforme realizábamos los análisis nos dimos cuenta que de forma indirecta se podía detectar otro *bad smell*. Ya que si tenemos variables de instancia que son poco usadas dentro de la clase, algo que queda reflejado en los informes, y tenemos muchos métodos de acceso o mutadores entonces nos encontramos ante una *clase de sólo datos*.

Capítulo 5

Conclusiones

Los objetivos que nos planteamos al comienzo del proyecto fueron los de crear una aplicación que detectase *bad smells*. Una vez implementada la usaríamos para analizar proyectos de código abierto escritos en Python con el objetivo de estudiar qué tipos de *bad smells* presentaban.

La propuesta inicial del proyecto era abierta, teníamos una taxonomía de *bad smells* amplia y había que decidir cuáles de ellos trataríamos. La selección de los *bad smells* que finalmente se implementaron estuvo motivada por criterios cuantitativos. De ahí que las clases implementadas sean *contadoras* ya que consideramos que algunos de los *bad smells* propuestos por Fowler pueden ser programados pero introducen una cierta subjetivad. Estamos pensando, por ejemplo, en el *bad smell* presentado en esta memoria como *Hombre en Medio*. Puede darse el caso de que la introducción de una clase que actúe como intermediaria obedezca a un patrón de diseño lícito. Por ello creemos que la aproximación cuantitativa a este problema lo hace más útil ya que existe cierto consenso en que un método enorme es susceptible de ser modificado o que la existencia de código duplicado es algo a evitar. La nuestra es una experiencia piloto que intenta abordar un problema complejo del que existen pocas implementaciones equivalentes.

Estos objetivos que nos marcamos los hemos conseguido, ya que tanto la detección, cuantificación y presentación de resultados se han realizado con éxito. Todo ello a pesar de la magnitud de los datos de entrada y, aunque la sintaxis de Python no permita muchas frivolidades, tuvimos que lidiar con mucho código escrito de una manera "exótica".

5.1. Lecciones aprendidas

Durante todas las fases del desarrollo de la aplicación y gracias a tener que enfrentarme a este proyecto, he aprendido muchas cosas tanto técnicas como de investigación. Veamos alguna de ellas:

- La importancia que tienen las buenas prácticas de programación para producir software de calidad. La clasificación de Martin Fowler es una referencia imprescindible que hay que tener muy presente si se pretende escribir código orientado a objetos de calidad.

- Python como lenguaje de programación y la posibilidad que brinda para introducirse un poco más a fondo en la programación orientada a objetos. Sin duda, un lenguaje a tener en cuenta para desarrollar futuros proyectos. También querría destacar el uso de que he dado a Python como lenguaje de *scripting* en el ámbito laboral a partir de estar desarrollando este proyecto.

- Las labores de investigación que se han tenido que realizar para poder orientar el proyecto en la dirección que se ha considerado la correcta.

- Hay que destacar también la combinación de diferentes tecnologías como es la programación orientada a objetos y HTML, utilizada en los informes que ofrece la aplicación.

- El aprendizaje de LaTeX como lenguaje de procesamiento de textos y que es el que se ha utilizado para la realización de esta memoria y que, con mucha probabilidad, será utilizado en futuros documentos como tutoriales o memorias.

- Se ha visto a lo largo de la realización del proyecto que no todos los proyectos de software estudiados están escritos de la misma forma y que el hecho de que Python permita combinar programación orientada a objetos con programación estructurada ha dificultado la implementación de alguna de las clases.

- El uso de la distribución GNU/Linux Ubuntu como sistema operativo donde se ha desarrollado el proyecto. Habíamos manejado antes otras *distros* pero ésta nos ha conquistado 8-)

5.2. Limitaciones

Aunque este proyecto identifica con éxito *bad smells* y muestra informes detallados y resumidos de manera muy satisfactoria para el usuario, también tiene sus limitaciones. La más importante es que la evaluación es únicamente cuantitativa y no pretende poner "nota" a los programas que se analizan. Es el usuario el que debe realizar una inspección dirigida por los informes que genera la aplicación de aquellos aspectos del código que presenten *bad smells*. Por eso, aún cuando los informes se generen bastante rápido, no deja de ser una buena práctica inspeccionar manualmente aquellos aspectos que nos resulten más sorprendentes. En todo caso, si tenemos que elegir entre cien proyectos y lo hacemos por su ausencia de *bad smells*, nuestra herramienta permitiría discriminar un amplio conjunto de los mismos y centrarse en estudiar pormenorizadamente un reducido número (con todo lo que esto significa en cuanto a ahorro de tiempo).

Otra de las limitaciones del proyecto es que el análisis de clases es estático, no pudiendo realizar un análisis dinámico para saber cómo se comunican las clases unas con otras, algo sin duda interesante ya que nos ofrecería una visión de la arquitectura de los proyectos. Esta visión general sería el primer paso para poder implementar el análisis de los *bad smells* que calificamos anteriormente de interclases.

5.3. Expectativas de futuro

Este PFC se enmarca dentro de la labor investigadora que mi tutor, Gregorio Robles, lleva a cabo en el ámbito de la ingeniería del software libre. Consideramos que a partir de la experiencia adquirida con un número limitado de *bad smells* y con los proyectos en Python puede ser muy provechosa para futuras investigaciones cuyo objetivo sea identificar problemas en el código, y a ser posible hacerlo de manera cuantificada.

El proyecto debería evolucionar en dos direcciones una ya se ha apuntado anteriormente: la implementación de aquellos *bad smells* que analizan el comportamiento y comunicación de las clases, pasando de un análisis estático a uno dinámico, y añadir soporte a más lenguajes de programación como Java, C/C++ o PHP.

Resultaría también interesante exportar la información generada a otros formatos como PDF,

de este modo se podrían comunicar los resultados de la aplicación a un equipo de desarrolladores. Además se podrían integrar a la aplicación los scripts externos que hemos empleado para tabular los datos de todos los paquetes Debian para facilitar la interpretación de los resultados.

Apéndice A

Manual de usuario

El proyecto consiste en una aplicación de consola a la cual se le pasa como parámetro un archivo comprimido en *tar.gz* aunque se puede pasar también un *directorio* o un archivo con extensión *.py*.

```
$ ./badsmcheck proyecto.tar.gz
$ ./badsmcheck dir_proyecto
$ ./badsmcheck fichero.py
```

Si no se ha establecido ninguna configuración por defecto la aplicación nos requerirá los datos necesarios para su correcto funcionamiento. El directorio, archivos comprimidos y fichero de entrada tienen que estar en el mismo directorio que el programa.

A.1. El archivo de configuración

Como se ha comentado anteriormente, el archivo de configuración es fundamental para el correcto funcionamiento de la aplicación, contiene los parámetros que se necesitan para realizar la evaluación de los proyectos. Este archivo se puede modificar para ajustar la configuración como deseemos.

A.2. Los parámetros

Badsmcheck, además del archivo de configuración, introduce una serie de opciones que son específicas de esa ejecución. Estas opciones, que siempre han de ir antes que el proyecto que

queramos estudiar, vienen dadas como parámetros de entrada a la hora de llamar al programa y tienen que seguir esta sintaxis:

```
$ ./badsmcheck [-c] [-v] Input_Directory [Output_Directory]
```

Listado de parámetros y función:

- **-h**: esta opción muestra por pantalla un pequeño texto de ayuda que nos recordará la sintaxis que hay que seguir para arrancar el programa.
- **-c**: esta opción es necesaria y obligatoria para establecer la configuración del programa, si se detecta que no se ha introducido configuración alguna se solicitará al usuario.
- **-v**: al añadir -v observamos qué está haciendo nuestro programa en cada momento.
- **Output_Directory**: esta opción crea un directorio con los resultados de los análisis con el nombre que le haya asignado el usuario. Si no se le pasa este parámetro la aplicación escribirá los resultados en un directorio llamado por defecto *output*.

Apéndice B

Requisitos del sistema

El sistema tiene que cumplir un requisito básico para que *badsmcheck* funcione correctamente: debemos tener instalado el intérprete de Python en nuestro sistema, nada más. En particular, se ha probado con una versión 2.4 de Python, aunque suponemos que funcionará con todas aquéllas posteriores a la 2.2.

Apéndice C

Paquetes de Debian Sarge analizados

A continuación se ofrece un listado de los paquetes de Debian Sarge que han sido analizados. Estos paquetes cuentan con código fuente en Python:

a2ps_4.13b.orig.tar.gz
aap_1.072.orig.tar.gz
abiword_2.2.7.orig.tar.gz
abook_0.5.3.orig.tar.gz
adesklets_0.4.7.orig.tar.gz
aewan_0.9.6.orig.tar.gz
affix_2.1.1.orig.tar.gz
albatross_1.20.orig.tar.gz
allegro4.1_4.1.15.orig.tar.gz
alsa-driver_1.0.8.orig.tar.gz
amarok_1.2.3.orig.tar.gz
amaya_8.5.orig.tar.gz
amsn_0.94.orig.tar.gz
anjuta_1.2.2.orig.tar.gz
apache2_2.0.54.orig.tar.gz
apoo_1.1.orig.tar.gz
asciidoc_6.0.3.orig.tar.gz
atlas-cpp_0.4.94.orig.tar.gz
audacity_1.2.3.orig.tar.gz
axel_1.0b.orig.tar.gz
barcode_0.98.orig.tar.gz
bastille_2.1.1.orig.tar.gz

beautifulsoup_1.2+cvs20041017.orig.tar.gz
bg5ps_1.3.0.orig.tar.gz
bicyclerepair_0.9.orig.tar.gz
bind9_9.2.4.orig.tar.gz
bittornado_0.3.11.orig.tar.gz
bittorrent_3.4.2.orig.tar.gz
bld_0.3.2.orig.tar.gz
blender_2.36.orig.tar.gz
blogtk_1.0.orig.tar.gz
bluez-utils_2.15.orig.tar.gz
bnetd_0.4.25.orig.tar.gz
boa-constructor_0.3.0.orig.tar.gz
boost_1.32.0.orig.tar.gz
brltty_3.4.1.orig.tar.gz
bugzilla_2.16.7.orig.tar.gz
capisuite_0.4.5.orig.tar.gz
cbios_0.20.orig.tar.gz
c-cpp-reference_2.0.2.orig.tar.gz
celementtree_1.0.2.orig.tar.gz
cfv_1.18.orig.tar.gz
cheesetracker_0.9.9.orig.tar.gz
cheetah_0.9.16.orig.tar.gz
cherrypy_0.10.orig.tar.gz
clearsilver_0.9.13.orig.tar.gz
clientcookie_0.4.19.orig.tar.gz
cmake_2.0.5.orig.tar.gz
comedilib_0.7.22.orig.tar.gz
cplay_1.49.orig.tar.gz
crm114_20050415.orig.tar.gz
crossfire-maps_1.4.0.orig.tar.gz
cscvs_1.0pre25.patch.79.orig.tar.gz
cssed_0.3.0.orig.tar.gz
ctypes_0.9.5.orig.tar.gz
cuetools_1.3.orig.tar.gz
cupsys_1.1.23.orig.tar.gz
curator_2.1.orig.tar.gz
cvsbook_1.21.orig.tar.gz

cvs-syncmail_1.2+cvs.2004.05.02.orig.tar.gz
cvsutils_0.0.20020311.orig.tar.gz
cxref_1.6.orig.tar.gz
cyphesis-cpp_0.3.5.orig.tar.gz
dak_1.0.orig.tar.gz
dbus_0.23.4.orig.tar.gz
dcl_0.9.4.4.orig.tar.gz
decompyle2.2_2.2beta1.orig.tar.gz
decompyle_2.3.2.orig.tar.gz
devhelp_0.9.3.orig.tar.gz
devil_1.6.7.orig.tar.gz
dia_0.94.0.orig.tar.gz
diacanvas2_0.13.0.orig.tar.gz
dict-bouvier_6.revised.orig.tar.gz
dict-gazetteer2k_1.0.0.orig.tar.gz
dict-jargon_4.4.4.orig.tar.gz
dict-moby-thesaurus_1.0.orig.tar.gz
digitemp_3.3.2.orig.tar.gz
ding_1.3.orig.tar.gz
directoryassistant_1.4.orig.tar.gz
discover-data_2.2005.02.13.orig.tar.gz
distcc_2.18.3.orig.tar.gz
diveintopython_5.4.orig.tar.gz
djbdoc2man_1.1.orig.tar.gz
doclifter_2.1.orig.tar.gz
doc-linux-fr_2005.02.orig.tar.gz
dosage_1.5.2.orig.tar.gz
doxygen_1.4.2.orig.tar.gz
drqueue_0.60.0.orig.tar.gz
dstat_0.5.10.orig.tar.gz
duplicity_0.4.1.orig.tar.gz
ecasound2.2_2.4.1.orig.tar.gz
editobj_0.5.6.orig.tar.gz
egenix-mx-base_2.0.6.orig.tar.gz
egroupware_1.0.0.007-2.dfsg.orig.tar.gz
ekg_1.5+20050411.orig.tar.gz
elementtree_1.2.6.orig.tar.gz

enemies-of-carlotta_1.0.3.orig.tar.gz
epic4-script-hienoa_0.53.orig.tar.gz
epiphany-browser_1.4.8.orig.tar.gz
epiphany-extensions_1.4.5.orig.tar.gz
epydoc_2.1.orig.tar.gz
eric_3.6.2.orig.tar.gz
ethereal_0.10.10.orig.tar.gz
eyed3_0.6.4.orig.tar.gz
ezmlm-browse_0.10.orig.tar.gz
fet_3.12.30.orig.tar.gz
fetchmail_6.2.5.orig.tar.gz
file_4.12.orig.tar.gz
fixedpoint_0.1.2.orig.tar.gz
flawfinder_1.26.orig.tar.gz
flightgear_0.9.6.orig.tar.gz
fnorb_1.3.orig.tar.gz
fonttools_1.99+2.0b1+cvs20031014.orig.tar.gz
forg_0.5.1.orig.tar.gz
forgethtml_0.0.20031008.orig.tar.gz
forgetsql_0.5.1.orig.tar.gz
freeciv_2.0.1.orig.tar.gz
freedict_1.1.orig.tar.gz
freeradius_1.0.2.orig.tar.gz
freesci_0.3.4c.orig.tar.gz
freetype_2.1.7.orig.tar.gz
fsh_1.2.orig.tar.gz
fsp_2.81.b24.orig.tar.gz
gaby_2.0.2.orig.tar.gz
gadfly_1.0.0.orig.tar.gz
gaphor_0.5.1.orig.tar.gz
gazpacho_0.5.3.orig.tar.gz
gcompris_6.5.2.orig.tar.gz
gdal_1.2.6.orig.tar.gz
gdeskcal_0.57.1.orig.tar.gz
gdesklets_0.33.1.orig.tar.gz
geda-examples_20050313.orig.tar.gz
geda-utils_20050313.orig.tar.gz

genetic_0.1.1b.orig.tar.gz
geos_2.1.1.orig.tar.gz
getmail_3.2.5.orig.tar.gz
getmail4_4.3.5.orig.tar.gz
gforge_3.1.orig.tar.gz
ggi-doc_0.0.20040308.orig.tar.gz
ghc-cvs_20040725.orig.tar.gz
gif2png_2.4.7.orig.tar.gz
gjots2_2.1.1.orig.tar.gz
glabels_2.0.2.orig.tar.gz
gle_3.1.0.orig.tar.gz
glimmer_1.2.1.orig.tar.gz
gmailfs_0.4.orig.tar.gz
gmsh_1.60.1.orig.tar.gz
gnat-gps_2.1.0.orig.tar.gz
gnome-blog_0.8.orig.tar.gz
gnome-doc-utils_0.1.3.orig.tar.gz
gnome-mud_0.10.5.orig.tar.gz
gnue-common_0.5.14.orig.tar.gz
gnue-designer_0.5.6.orig.tar.gz
gnue-forms_0.5.11.orig.tar.gz
gnue-navigator_0.0.8.orig.tar.gz
gnue-reports_0.1.7.orig.tar.gz
gnugo_3.6.orig.tar.gz
gnumeric_1.4.3.orig.tar.gz
gnupginterface_0.3.2.orig.tar.gz
gnuradio-core_2.4.orig.tar.gz
gnushogi_1.3.orig.tar.gz
gnusim8085_1.2.89.orig.tar.gz
gpe-contacts_0.34.orig.tar.gz
gphpedit_0.9.50.orig.tar.gz
gpib_3.2.03.orig.tar.gz
gpsd_2.13.orig.tar.gz
gramps_1.0.11.orig.tar.gz
gr-audio-oss_0.5.orig.tar.gz
grmonitor_0.81.orig.tar.gz
gs-esp_7.07.1.orig.tar.gz

gs-gpl_8.01.orig.tar.gz
gst-python_0.8.1.orig.tar.gz
gtk+2.0_2.6.4.orig.tar.gz
guile-gnome-platform_2.7.99.orig.tar.gz
gvidm_0.8.orig.tar.gz
gvr_1.2.1.orig.tar.gz
gvr-lessons_0.2.orig.tar.gz
hal_0.4.7.orig.tar.gz
hamlib_1.2.4.orig.tar.gz
hashcash_1.17.orig.tar.gz
highlight_2.2.8.orig.tar.gz
hplip_0.9.2.orig.tar.gz
htmlgen_2.2.2.orig.tar.gz
hwdata_0.148.orig.tar.gz
hypermail_2.1.8.orig.tar.gz
iceme_1.0.0.orig.tar.gz
icepref_1.1.orig.tar.gz
iirish_2.0.orig.tar.gz
imcom_1.33.orig.tar.gz
imediff2_1.1.0+20041113.orig.tar.gz
imgseek_0.8.4.orig.tar.gz
imgsizer_2.7.orig.tar.gz
imhangul_0.9.11.orig.tar.gz
imlib_1.9.14.orig.tar.gz
inkscape_0.41.orig.tar.gz
inn2_2.4.2.orig.tar.gz
ipcheck_0.225.orig.tar.gz
ipython_0.6.13.orig.tar.gz
ircd-ircu_2.10.11.04.orig.tar.gz
iso-codes_0.44.orig.tar.gz
ispell_3.1.20.0.orig.tar.gz
ispell-lt_1.1.orig.tar.gz
jabber.py_0.5.0.orig.tar.gz
jack_3.1.1.orig.tar.gz
jaxml_3.01.orig.tar.gz
jftpgw_0.13.5.orig.tar.gzCláusulas Switch
jsch_0.1.19.orig.tar.gz

jumpnbump_1.50.orig.tar.gz
jython_2.1.0.orig.tar.gz
k3d_0.4.3.0.orig.tar.gz
kdeaddons_3.3.2.orig.tar.gz
kdebindings_3.3.2.orig.tar.gz
kdeedu_3.3.2.orig.tar.gz
kdesdk_3.3.2.orig.tar.gz
kdevelop3_3.2.0.orig.tar.gz
kdissert_0.3.8.orig.tar.gz
kernel-source-2.6.8_2.6.8.orig.tar.gz
kexi_0.1cvs20050408.orig.tar.gz
kfocus_1.0.2.orig.tar.gz
kid_0.6.3.orig.tar.gz
knapster2_0.5.orig.tar.gz
knoda_0.7.3.orig.tar.gz
knutclient_0.8.5.orig.tar.gz
kodos_2.4.5.orig.tar.gz
koffice_1.3.5.orig.tar.gz
kover_2.9.6.orig.tar.gz
kpsk_1.0.1.orig.tar.gz
krb5_1.3.6.orig.tar.gz
kudzu_1.1.67.orig.tar.gz
lcms_1.13.orig.tar.gz
lfm_0.91.2.orig.tar.gz
lg-issue107_1.orig.tar.gz
lg-issue108_1.orig.tar.gz
lg-issue110_1.orig.tar.gz
lg-issue56_2.orig.tar.gz
lg-issue63_2.orig.tar.gz
lg-issue66_3.orig.tar.gz
lg-issue67_4.orig.tar.gz
lg-issue74_2.orig.tar.gz
lg-issue77_2.orig.tar.gz
lg-issue79_2.orig.tar.gz
lg-issue82_2.orig.tar.gz
lg-issue83_2.orig.tar.gz
lg-issue85_1.orig.tar.gz

lg-issue92_1.orig.tar.gz
lilypond_2.2.6.orig.tar.gz
linkchecker_2.8.orig.tar.gz
liquidwar_5.6.2.orig.tar.gz
logilab-common_0.9.3.orig.tar.gz
lsbappchk_1.3.4.orig.tar.gz
luma_2.0.3.orig.tar.gz
lush_1.0+final.orig.tar.gz
lyskom-server_2.1.2.orig.tar.gz
lyx_1.3.4.orig.tar.gz
lzo_1.08.orig.tar.gz
m2crypto_0.13.orig.tar.gz
macchanger_1.5.0.orig.tar.gz
madman_0.93.0.orig.tar.gz
mailcrypt_3.5.8.orig.tar.gz
mailman_2.1.5.orig.tar.gz
makejail_0.0.5.orig.tar.gz
mapserver_4.4.1.orig.tar.gz
maxdb-7.5.00_7.5.00.24.orig.tar.gz
maxdb-buildtools_533920.orig.tar.gz
mayavi_1.4.orig.tar.gz
mbot_0.3.orig.tar.gz
mboxcheck-applet_0.3.orig.tar.gz
megahal_9.1.1.orig.tar.gz
meld_0.9.4.1+20050125.orig.tar.gz
memaid-pyqt_0.2.3.orig.tar.gz
mftrace_1.1.5.orig.tar.gz
mma_0.12.orig.tar.gz
moin_1.3.4.orig.tar.gz
moniwiki_1.0.9.orig.tar.gz
monotone_0.18.orig.tar.gz
moosic_1.5.1.orig.tar.gz
mozilla_1.7.8.orig.tar.gz
msc_1.1.1.orig.tar.gz
mtink_1.0.5.orig.tar.gz
mtx_1.2.16rel.orig.tar.gz
muddleftpd_1.3.13.1.orig.tar.gz

musiclibrarian_1.6.orig.tar.gz
mysql-admin_1.0.20.orig.tar.gz
mysql-query-browser_1.1.6.orig.tar.gz
nagios-plugins_1.4.orig.tar.gz
newt_0.51.6.orig.tar.gz
nget_0.27.1.orig.tar.gz
nicotine_1.0.8rc1.orig.tar.gz
ntlmaps_0.9.9.orig.tar.gz
ntop_3.0.orig.tar.gz
omnievents_2.6.1.orig.tar.gz
omniorb4_4.0.6.orig.tar.gz
ooo2dbk_1.3.13.orig.tar.gz
openal_0.2004090900.orig.tar.gz
openbox_3.2.orig.tar.gz
opencv_0.9.5.orig.tar.gz
openmsx_0.5.1.orig.tar.gz
openoffice.org_1.1.3.orig.tar.gz
openrpg_1.6.1.orig.tar.gz
optcomplete_1.2.orig.tar.gz
orpie_1.4.1.orig.tar.gz
pathological_1.1.3.orig.tar.gz
pexpect_0.999.orig.tar.gz
pfe_0.33.34.orig.tar.gz
phpgroupware_0.9.16.005.orig.tar.gz
phpreports_0.3.6.orig.tar.gz
pilot-link_0.11.8.orig.tar.gz
planner_0.13.orig.tar.gz
planner-el_3.27.orig.tar.gz
plplot_5.3.1.orig.tar.gz
pmock_0.3.orig.tar.gz
pnet_0.6.12.orig.tar.gz
pointless_0.5.orig.tar.gz
policycoreutils_1.22+0.orig.tar.gz
positron_1.1.orig.tar.gz
postgresql_7.4.7.orig.tar.gz
postnews_0.5.3.orig.tar.gz
potracegui_1.3.orig.tar.gz

pound_1.8.2.orig.tar.gz
prospect_0.9.8b.orig.tar.gz
psyco_1.4.orig.tar.gz
psycopg_1.1.18.orig.tar.gz
publib_0.38.orig.tar.gz
putty_0.58.orig.tar.gz
py2play_0.1.7.orig.tar.gz
pyao_0.82.orig.tar.gz
pybliographer_1.2.6.2.orig.tar.gz
pyblosxom_1.2.orig.tar.gz
pyca_20031118.orig.tar.gz
pycaml_0.81.orig.tar.gz
pychecker_0.8.14.orig.tar.gz
pyching_1.2.1.orig.tar.gz
pycurl_7.13.2.orig.tar.gz
pycxx_5.3.2.orig.tar.gz
pydb_1.01.orig.tar.gz
pydict_0.2.5.1.orig.tar.gz
pyflac_0.0.3.orig.tar.gz
pygame_1.6.orig.tar.gz
pygdchart2_0.beta1.orig.tar.gz
pygresql_3.6.1.orig.tar.gz
pygtkmvc_0.9.2.orig.tar.gz
pyid3lib_0.5.1.orig.tar.gz
py-libmpdclient_0.10.0.orig.tar.gz
pylint_0.6.4.orig.tar.gz
pymacs_0.22.orig.tar.gz
pymad_0.5.2.orig.tar.gz
pymodplug_1.1.orig.tar.gz
pymol_0.97.orig.tar.gz
pyogg_1.3.orig.tar.gz
pyopenal_0.1.4.orig.tar.gz
pyopengl_2.0.1.08.orig.tar.gz
pyopenssl_0.6.orig.tar.gz
pyorbit_2.0.1.orig.tar.gz
pyparallel_0.2.orig.tar.gz
pyparsing_1.2.2.orig.tar.gz

pypoker-eval_126.0.orig.tar.gz
pyrad_0.8.orig.tar.gz
pyrex_0.9.3.orig.tar.gz
pyrite-publisher_2.1.1.orig.tar.gz
pyro_3.4.orig.tar.gz
pyserial_2.1.orig.tar.gz
pyslide_0.4.orig.tar.gz
pysol_4.82.1.orig.tar.gz
pysol-sound-server_3.00.orig.tar.gz
pysvn_1.1.2.orig.tar.gz
pytables_0.9.1.orig.tar.gz
python2.3_2.3.5.orig.tar.gz
python-4suite_0.99cvs20050418.orig.tar.gz
python-adns_1.0.0.orig.tar.gz
python-apsw_3.1.3r1.orig.tar.gz
python-bibtex_1.2.1.orig.tar.gz
python-biggles_1.6.4.orig.tar.gz
python-biopython_1.30.orig.tar.gz
python-bsddb3_3.3.0.orig.tar.gz
python-bz2_1.1.orig.tar.gz
pythoncad_0.1.23.orig.tar.gz
python-cddb_1.4.orig.tar.gz
python-cjkcodecs_1.1.1.orig.tar.gz
python-crack_0.5.orig.tar.gz
python-crypto_2.0+dp1.orig.tar.gz
python-csv_1.0.orig.tar.gz
python-davlib_1.8.orig.tar.gz
python-dhm_0.5.orig.tar.gz
python-dns_2.3.0.orig.tar.gz
python-docutils_0.3.7.orig.tar.gz
python-email_2.5.5.orig.tar.gz
python-f2py_2.45.241+1926.orig.tar.gz
python-fuse_2.2.orig.tar.gz
python-gd_0.52debian.orig.tar.gz
python-gdchart_0.6.1.orig.tar.gz
python-gendoc_0.73.orig.tar.gz
python-geoip_1.2.0.orig.tar.gz

python-gnome_1.4.5.orig.tar.gz
python-gnome2_2.6.1.orig.tar.gz
python-gnuplot_1.7.orig.tar.gz
python-gtk2_2.6.1.orig.tar.gz
python-gtk2-tutorial_2.3.orig.tar.gz
python-gtkextra_0.22.orig.tar.gz
python-happydoc_2.1.orig.tar.gz
python-htmltmpl_1.22.orig.tar.gz
python-iconvcodec_1.1.2.orig.tar.gz
python-id3_1.2.orig.tar.gz
python-imaging_1.1.4.orig.tar.gz
python-irclib_0.4.4.orig.tar.gz
python-japanese-codecs_1.4.9.orig.tar.gz
python-kinterbasdb_3.1.orig.tar.gz
python-korean-codecs_2.0.5.orig.tar.gz
python-ldap_2.0.4.orig.tar.gz
python-libgmail_0.0.8+cvs20050208.orig.tar.gz
python-mysqldb_1.2.1c2.orig.tar.gz
python-numarray_1.1.1.orig.tar.gz
python-numeric_23.8.orig.tar.gz
python-omniorb2_2.6.orig.tar.gz
python-optik_1.4.1.orig.tar.gz
python-orbit_0.3.1.orig.tar.gz
python-osd_0.2.12.orig.tar.gz
python-oss_0.0.0.20010624.orig.tar.gz
python-pgsql_2.4.0.orig.tar.gz
python-pmw_1.2.orig.tar.gz
python-pqueue_0.2.orig.tar.gz
python-pylibacl_0.2.1.orig.tar.gz
python-pymetar_0.12.orig.tar.gz
python-pyxattr_0.2.orig.tar.gz
python-qt3_3.13.orig.tar.gz
python-reportlab_1.20debian.orig.tar.gz
python-rrd_0.2.1.orig.tar.gz
python-scientific_2.4.9.orig.tar.gz
python-scipy_0.3.2.orig.tar.gz
python-scipy-core_0.3.2.orig.tar.gz

python-setuptools_0.0.1.041214.orig.tar.gz
python-simpy_1.5.1.orig.tar.gz
python-slang_0.2.0.orig.tar.gz
python-soappy_0.11.3.orig.tar.gz
python-sqlite_1.0.1.orig.tar.gz
python-stats_0.6.orig.tar.gz
python-tclink_3.4.0.orig.tar.gz
python-tcpwrap_0.2.orig.tar.gz
python-textile_2.0.10.orig.tar.gz
python-tz_2005a.orig.tar.gz
python-uncertainities_0.001.orig.tar.gz
python-unit_1.4.1.orig.tar.gz
python-visual_3.1.1.orig.tar.gz
python-weblib_1.3.3.orig.tar.gz
python-xlib_0.12.orig.tar.gz
python-xml_0.8.4.orig.tar.gz
pytone_2.2.3.orig.tar.gz
pyvorbis_1.3.orig.tar.gz
pyvtk_0.4.66.orig.tar.gz
pyx_0.7.1.orig.tar.gz
pyxdg_0.8.orig.tar.gz
pyxine_0.1alpha2.orig.tar.gz
pyxmms_2.04.orig.tar.gz
pyxmms-remote_1.12.orig.tar.gz
pyzor_0.4.0+cvs20030201.orig.tar.gz
qm_2.2.orig.tar.gz
qscintilla_1.3.orig.tar.gz
qterm_0.4.0pre2.orig.tar.gz
qtorrent_0.9.6.1.orig.tar.gz
quantlib-python_0.3.9.orig.tar.gz
quiteinsane_0.10.orig.tar.gz
quiteinsanegimpplugin_0.3.orig.tar.gz
quixote_1.0.orig.tar.gz
quodlibet_0.10.1.orig.tar.gz
radiuscontext_1.88.orig.tar.gz
rapidsvn_0.7.0.orig.tar.gz
rawdog_2.4.orig.tar.gz

rdesktop_1.4.0.orig.tar.gz
rdiff-backup_0.13.4.orig.tar.gz
recode_3.6.orig.tar.gz
redland-bindings_1.0.0.2.orig.tar.gz
regina-normal_4.1.3.orig.tar.gz
reprepro_0.3.orig.tar.gz
revelation_0.3.4.orig.tar.gz
rgtk_0.7.0.orig.tar.gz
rhyme_0.9.orig.tar.gz
rosegarden4_1.0.orig.tar.gz
roundup_0.8.2.orig.tar.gz
rox_2.2.0.orig.tar.gz
rpy_0.4.1.orig.tar.gz
rss2email_2.54.orig.tar.gz
rsync_2.6.4.orig.tar.gz
rubber_0.99.8.orig.tar.gz
rubrica_1.0.12.orig.tar.gz
ruby1.8_1.8.2.orig.tar.gz
samba_3.0.14a.orig.tar.gz
sanduhr_1.0.orig.tar.gz
saods9_3.0.3.orig.tar.gz
scanerrlog_2.01.orig.tar.gz
scapy_0.9.17.orig.tar.gz
scgi_1.2.orig.tar.gz
scid_3.6.1.orig.tar.gz
scigraphica_0.8.0.orig.tar.gz
scite_1.63.orig.tar.gz
scons_0.96.1.orig.tar.gz
scribus_1.2.1.orig.tar.gz
sdcc_2.4.0.orig.tar.gz
sdl-stretch_0.2.3.orig.tar.gz
sgmltools-lite_3.0.3.0.cvs.20010909.orig.tar.gz
simpleparse_2.0.0.orig.tar.gz
simpletal_3.12.orig.tar.gz
simulavr_0.1.2.2.orig.tar.gz
sip4-qt3_4.1.1.orig.tar.gz
sitemap_2.3.orig.tar.gz

sketch_0.6.15.orig.tar.gz
sleuthkit_2.00.orig.tar.gz
slides_1.0.1.orig.tar.gz
slune_1.0.7.orig.tar.gz
smart_0.29.2.orig.tar.gz
smarteiffel_1.1.orig.tar.gz
snack_2.2.9.orig.tar.gz
snappea_3.0d3.orig.tar.gz
solarwolf_1.5.orig.tar.gz
solfege_2.0.4.orig.tar.gz
songwrite_0.12b.orig.tar.gz
sooperlooper_0.93.orig.tar.gz
source-highlight_1.11.orig.tar.gz
soya_0.9.2.orig.tar.gz
soya-doc_0.9.orig.tar.gz
spambayes_1.0.3.orig.tar.gz
spkproxy_1.4.7.orig.tar.gz
sqlobject_0.6.orig.tar.gz
sqlrelay_0.35.orig.tar.gz
squishdot_1.5.0.orig.tar.gz
stardict_2.4.3.orig.tar.gz
straw_0.25.1.orig.tar.gz
streamtuner_0.99.99.orig.tar.gz
subversion_1.1.4.orig.tar.gz
supertux_0.1.2.orig.tar.gz
supybot_0.80.1.orig.tar.gz
swig1.3_1.3.24.orig.tar.gz
sword_1.5.7.orig.tar.gz
syck_0.42.orig.tar.gz
sylpheed-claws_1.0.4.orig.tar.gz
synopsis_0.5.0.orig.tar.gz
tau_2.14.1.1.orig.tar.gz
tcllib_1.6.1.orig.tar.gz
teg_0.11.1.orig.tar.gz
tellico_0.13.3.orig.tar.gz
tessa_0.3.1.orig.tar.gz
tetex-base_2.0.2c.orig.tar.gz

tetex-src_2.0.2a.orig.tar.gz
thuban_1.0.0.orig.tar.gz
tinywm_1.3.orig.tar.gz
tix8.1_8.1.4.orig.tar.gz
tktable_2.9.orig.tar.gz
torch3_3.1.orig.tar.gz
torsmo_0.18.orig.tar.gz
trac_0.8.1.orig.tar.gz
tre_0.7.2.orig.tar.gz
ttcn3parser_20050130.orig.tar.gz
ttf-unfonts_1.0.1.orig.tar.gz
tuxpaint-config_0.0.5.orig.tar.gz
twisted_1.3.0.orig.tar.gz
txt2tags_2.0.orig.tar.gz
uligo_0.3.orig.tar.gz
ultrapossum_1.0rc5.orig.tar.gz
utf8script_1.0.orig.tar.gz
viewcvs_0.9.2+cvs.1.0.dev.2004.07.28.orig.tar.gz
viewmol_2.4.1.orig.tar.gz
vim-latexsuite_0.20041219.orig.tar.gz
vimoutliner_0.3.3.orig.tar.gz
vte_0.11.12.orig.tar.gz
vtk_4.4.2.orig.tar.gz
wavesurfer_1.8.1.orig.tar.gz
wily_0.13.41.orig.tar.gz
wmaker_0.91.0.orig.tar.gz
woody_0.1.6.orig.tar.gz
wv2_0.2.2.orig.tar.gz
xastir_1.4.1.orig.tar.gz
xbl_1.1.2.orig.tar.gz
xcircuit_3.1.19.orig.tar.gz
xemacs21-packages_2005.03.07.orig.tar.gz
xenophilia_0.8.orig.tar.gz
xfree86_4.3.0.dfsg.1.orig.tar.gz
xfs-xtt_1.4.1.xf430.orig.tar.gz
xmldiff_0.6.6.orig.tar.gz
xmms-coverviewer_0.11.orig.tar.gz

xprint_0.1.0.alpha1.orig.tar.gz
xracer_0.96.9.orig.tar.gz
xtalk_1.3.orig.tar.gz
xt-toolbus_0.25.orig.tar.gz
xxdiff_3.1.orig.tar.gz
yacas_1.0.57.orig.tar.gz
yapps2_2.1.1.orig.tar.gz
yodl_1.31.18.orig.tar.gz
zh-autoconvert_0.3.14.orig.tar.gz
zope2.6-verbosesecurity_0.5.orig.tar.gz
zope2.7-archetypes_1.3.1.orig.tar.gz
zope-backtalk_0.3.orig.tar.gz
zope-btreefolder2_1.0.1.orig.tar.gz
zope-callprofiler_1.4.orig.tar.gz
zope-cmf1.4_1.4.7.orig.tar.gz
zope-cmfactionicons_0.9.orig.tar.gz
zope-cmfformcontroller_1.0.3.orig.tar.gz
zope-cmfforum_1.0.orig.tar.gz
zope-cmfldap_2.0.orig.tar.gz
zope-cmfpgforum_1.0.0b.orig.tar.gz
zope-cmfphotoalbum_0.5.0.orig.tar.gz
zope-cmfplone_2.0.4.orig.tar.gz
zope-cmfquickinstallertool_1.5.0.orig.tar.gz
zope-cmfsin_0.6.1.orig.tar.gz
zope-cmfworkflow_0.4.2.orig.tar.gz
zope-cookiecrumbler_1.2.orig.tar.gz
zope-coreblog_1.0.orig.tar.gz
zope-docfindereverywhere_0.4.1.orig.tar.gz
zope-docfindertab_0.5.0.orig.tar.gz
zope-dtmlcalendar_1.0.15.orig.tar.gz
zopeedit_0.8.orig.tar.gz
zope-emarket_0.2.0a4.orig.tar.gz
zope-epoz_0.9.0.orig.tar.gz
zope-externaleditor_0.8.orig.tar.gz
zope-extfile_1.4.2.orig.tar.gz
zope-exuserfolder_0.50.0.orig.tar.gz
zope-filesystemsite_1.3.orig.tar.gz

zope-formulator_1.7.0.orig.tar.gz
zope-groupuserfolder_3.1.1.orig.tar.gz
zope-i18nfolder_2.02.orig.tar.gz
zope-i18nlayer_0.5.5.orig.tar.gz
zopeinterface_3.0.1.orig.tar.gz
zope-kinterbasdbda_1.0.orig.tar.gz
zope-kupu_1.1.1.orig.tar.gz
zope-ldap_1.1.0.orig.tar.gz
zope-ldapuserfolder_2.2.orig.tar.gz
zope-linguaplone_0.7.orig.tar.gz
zope-localizer_1.0.1.orig.tar.gz
zope-lockablefolder_0.1.0.orig.tar.gz
zope-loginmanager_0.8.8b1.orig.tar.gz
zope-mysqlda_2.0.8.orig.tar.gz
zope-parsedxml_1.3.1.orig.tar.gz
zope-photo_1.2.3.orig.tar.gz
zope-plonearticle_2.0.5.orig.tar.gz
zope-plonecollectorng_1.2.6.orig.tar.gz
zope-ploneerrorreporting_0.11.orig.tar.gz
zope-plonelanguagetool_0.5.orig.tar.gz
zope-plonetranslations_0.6.orig.tar.gz
zope-quotafolder_0.1.1.orig.tar.gz
zope-rdfgrabber_0.4.orig.tar.gz
zope-replacesupport_1.0.2.orig.tar.gz
zope-speedpack_0.3.orig.tar.gz
zope-testcase_0.9.6.orig.tar.gz
zope-textindexng2_2.0.8.orig.tar.gz
zope-tinytableplus_0.9.orig.tar.gz
zope-translationservice_0.4.orig.tar.gz
zope-ttwtype_0.9.1.orig.tar.gz
zope-verbosesecurity_0.6.orig.tar.gz
zope-xmlmethods_1.0.0.orig.tar.gz
zope-zaaplugins_2.21.orig.tar.gz
zope-zattachmentattribute_2.21.orig.tar.gz
zope-zms_2.1.2.7.orig.tar.gz
zope-znavigator_2.02.orig.tar.gz
zope-zpatterns_0.4.3p2.orig.tar.gz

zope-zshell_1.60.orig.tar.gz

zope-zwiki_0.37.0.orig.tar.gz

zsi_1.5.0.orig.tar.gz

zziplib_0.12.83.orig.tar.gz

Apéndice D

Código del método medline_query

```
def medline_query (keyword,maxcount,displaystart,field,abstract,epubahead,pubtype,
language,subset,agerange,humananimal,gender,entrezdate,pubdate,fromdate,todate):
  # note all the parameters needed to perform the query
  # Search with field limits
  if field == 'All Fields': field = 'ALL'
  elif field == 'Affiliation': field = 'AFFL'
  elif field == 'Author Name': field = 'AUTH'
  elif field == 'EC/RN Number': field = 'ECNO'
  elif field == 'Entrez Date': field = 'EDAT'
  elif field == 'Filter': field = 'FLTR'
  elif field == 'Issue': field = 'ISS'
  elif field == 'Journal Name': field = 'JOUR'
  elif field == 'Language': field = 'LANG'
  elif field == 'MeSH Date': field = 'MHDA'
  elif field == 'MeSH Major Topic': field = 'MAJR'
  elif field == 'MeSH Subheading': field = 'SUBH'
  elif field == 'MeSH Terms': field = 'MESH'
  elif field == 'Pagination': field = 'PAGE'
  elif field == 'Publication Date': field = 'PDAT'
  elif field == 'Publication Type': field = 'PTYP'
  elif field == 'Secondary Source ID': field = 'SI'
  elif field == 'Substance Name': field = 'SUBS'
  elif field == 'Text Word': field = 'WORD'
  elif field == 'Title': field = 'TITL'
  elif field == 'Title/Abstract': field = 'TIAB'
```

```
elif field == 'UID': field = 'UID'
elif field == 'Volume': field = 'VOL'

# Below is added to keyword if user wants items with abstracts only
if abstract: keyword = keyword + ' AND hasabstract'

# Below is added to keyword if user wants items that are listed on pubmed
# ahead of print
if epubahead: keyword = keyword + ' AND pubstatusaheadofprint'

# Below are publication type limits to add to keyword
if pubtype == 'Addresses': keyword = keyword + ' AND addresses[pt]'
elif pubtype == 'Bibliography': keyword = keyword + ' AND bibliography[pt]'
elif pubtype == 'Biography': keyword = keyword + ' AND biography[pt]'
elif pubtype == 'Classical Article': keyword = keyword +
' AND classical article[pt]'
elif pubtype == 'Clinical Conference': keyword = keyword +
' AND clinical conference[pt]'
elif pubtype == 'Clinical Trial': keyword = keyword + ' AND clinical trial[pt]'
elif pubtype == 'Clinical Trial, Phase I': keyword = keyword +
' AND clinical trial, phase I[pt]'
elif pubtype == 'Clinical Trial, Phase II': keyword = keyword +
' AND clinical trial, phase II[pt]'
elif pubtype == 'Clinical Trial, Phase III': keyword = keyword +
' AND clinical trial, phase III[pt]'
elif pubtype == 'Clinical Trial, Phase IV': keyword = keyword +
' AND clinical trial, phase IV[pt]'
elif pubtype == 'Comment': keyword = keyword + ' AND comment[pt]'
elif pubtype == 'Congresses': keyword = keyword + ' AND congresses[pt]'
elif pubtype == 'Consensus Development Conference': keyword = keyword +
' AND consensus development conference[pt]'
elif pubtype == 'Consensus Development Conference, NIH': keyword = keyword +
' AND consensus development conference, NIH[pt]'
elif pubtype == 'Controlled Clinical Trial': keyword = keyword +
' AND controlled clinical trial[pt]'
elif pubtype == 'Corrected and Republished Article': keyword = keyword +
' AND corrected and republished article[pt]'
```

```
elif pubtype == 'Dictionary': keyword = keyword + ' AND dictionary[pt]'
elif pubtype == 'Directory': keyword = keyword + ' AND directory[pt]'
elif pubtype == 'Duplicate Publication': keyword = keyword +
' AND duplicate publication[pt]'
elif pubtype == 'Editorial': keyword = keyword + ' AND editorial[pt]'
elif pubtype == 'Evaluation Studies': keyword = keyword +
' AND evaluation studies[pt]'
elif pubtype == 'Festschrift': keyword = keyword + ' AND festschrift[pt]'
elif pubtype == 'Government Publications': keyword = keyword +
' AND government publications[pt]'
elif pubtype == 'Guideline': keyword = keyword + ' AND guideline[pt]'
elif pubtype == 'Historical Article': keyword = keyword +
' AND historical article[pt]'
elif pubtype == 'Interview': keyword = keyword + ' AND interview[pt]'
elif pubtype == 'Journal Article': keyword = keyword + ' AND journal article[pt]'
elif pubtype == 'Lectures': keyword = keyword + ' AND lectures[pt]'
elif pubtype == 'Legal Cases': keyword = keyword + ' AND legal cases[pt]'
elif pubtype == 'Legislation': keyword = keyword + ' AND legislation[pt]'
elif pubtype == 'Letter': keyword = keyword + ' AND letter[pt]'
elif pubtype == 'Meta-Analysis': keyword = keyword + ' AND meta-analysis[pt]'
elif pubtype == 'Multicenter Study': keyword = keyword +
' AND multicenter study[pt]'
elif pubtype == 'News': keyword = keyword + ' AND news[pt]'
elif pubtype == 'Newspaper Article': keyword = keyword +
' AND newspaper article[pt]'
elif pubtype == 'Overall': keyword = keyword + ' AND overall[pt]'
elif pubtype == 'Periodical Index': keyword = keyword + ' AND periodical index[pt]'
elif pubtype == 'Practice Guideline': keyword = keyword +
' AND practice guideline[pt'
elif pubtype == 'Published Erratum': keyword = keyword +
' AND published erratum[pt]'
elif pubtype == 'Randomized Controlled Trial': keyword = keyword +
' AND randomized controlled trial[pt]'
elif pubtype == 'Retraction of Publication': keyword = keyword +
' AND retraction of publication[pt]'
elif pubtype == 'Retracted Publication': keyword = keyword +
' AND retracted publication[pt]'
```

```
elif pubtype == 'Review': keyword = keyword + ' AND review[pt]'
elif pubtype == 'Review, Academic': keyword = keyword + ' AND review, academic[pt]'
elif pubtype == 'Review Literature': keyword = keyword +
' AND review, literature[pt]'
elif pubtype == 'Review, Multicase': keyword = keyword + ' AND review, multicase[pt]'
elif pubtype == 'Review of Reported Cases': keyword = keyword +
' AND review of reported cases[pt]'
elif pubtype == 'Review, Tutorial': keyword = keyword + ' AND review, tutorial[pt]'
elif pubtype == 'Scientific Integrity Review': keyword = keyword +
' AND scientific integrity review[pt]'
elif pubtype == 'Technical Report': keyword = keyword + ' AND technical report[pt]'
elif pubtype == 'Twin Study': keyword = keyword + ' AND twin study[pt]'
elif pubtype == 'Validation Studies': keyword = keyword +
' AND validation studies[pt]'

# Below are language limits to add to keyword if chosen

if language == 'English': keyword = keyword + ' AND english[la]'
elif language == 'French': keyword = keyword + ' AND french[la]'
elif language == 'German': keyword = keyword + ' AND german[la]'
elif language == 'Italian': keyword = keyword + ' AND italian[la]'
elif language == 'Japanese': keyword = keyword + ' AND japanese[la]'
elif language == 'Russian': keyword = keyword + ' AND russian[la]'
elif language == 'Spanish': keyword = keyword + ' AND spanish[la]'

# Below are subset limits to add to keyword if chosen

if subset == 'AIDS': keyword = keyword + ' AND aids[sb]'
elif subset == 'AIDS/HIV journals': keyword = keyword + ' AND jsubsetx'
#X -  AIDS/HIV journals, non-Index Medicus
elif subset == 'Bioethics': keyword = keyword + ' AND bioethics[ab]'
elif subset == 'Bioethics journals': keyword = keyword + ' AND jsubsete'
#E -  bioethics journals, non-Index Medicus
elif subset == 'Biotechnology journals': keyword = keyword + ' AND jsubsetb'
#B -  biotechnology journals (assigned 1990 - 1998), non-Index Medicus
elif subset == 'Communication disorders journals': keyword = keyword +
' AND jusbsetc'
```

```
#C -  communication disorders journals (assigned 1977 - 1997), non-Index Medicus
elif subset == 'Complementary and Alternative Medicine': keyword = keyword +
' AND cam[sb]'
elif subset == 'Consumer health journals': keyword = keyword + ' AND jsubsetk'
#K -  consumer health journals, non-Index Medicus
elif subset == 'Core clinical journals': keyword = keyword + ' AND jsubsetaim'
#AIM - Abridged Index Medicus A list of core clinical journals created 20 years ago
elif subset == 'Dental journals': keyword = keyword + ' AND jsubsetd'
#D  -  dentistry journals
elif subset == 'Health administration journals': keyword = keyword + ' AND jsubseth'
#H -  health administration journals, non-Index Medicus
elif subset == 'Health tech assesment journals': keyword = keyword + ' AND jsubsett'
#T -  health technology assessment journals, non-Index Medicus
elif subset == 'History of Medicine': keyword = keyword + ' AND history[sb]'
elif subset == 'History of Medicine journals': keyword = keyword + ' AND jsubsetq'
#Q -  history of medicine journals, non-Index Medicus
elif subset == 'In process': keyword = keyword + ' AND in process[sb]'
elif subset == 'Index Medicus journals': keyword = keyword + ' AND jsubsetim'
#IM -  Index Medicus journals
elif subset == 'MEDLINE': keyword = keyword + ' AND medline[sb]'
elif subset == 'NASA journals': keyword = keyword + ' AND jsubsets'
#S-National Aeronautics and Space Administration (NASA) journals, non-Index Medicus
elif subset == 'Nursing journals': keyword = keyword + ' AND jsubsetn'
#N  -  nursing journals
elif subset == 'PubMed Central': keyword = keyword + ' AND medline pmc[sb]'
elif subset == 'Reproduction journals': keyword = keyword + ' AND jsubsetr'
#R -  reproduction journals (assigned 1972 - 1979), non-Index Medicus
elif subset == 'Space Life Sciences': keyword = keyword + ' AND space[sb]'
elif subset == 'Supplied by Publisher': keyword = keyword + ' AND publisher[sb]'
elif subset == 'Toxicology': keyword = keyword + ' AND tox[sb]'

# Age range will be added to keyword if desired
if agerange == 'All Infant: birth-23 month': keyword = keyword + ' AND infant[mh]'
elif agerange == 'All Child: 0-18 years': keyword = keyword + ' AND child[mh]'
elif agerange == 'All Adult: 19+ years': keyword = keyword + ' AND adult[mh]'
elif agerange == 'Newborn: birth-1 month': keyword = keyword +
' AND infant, newborn[mh]'
```

```
elif agerange == 'Infant: 1-23 months': keyword = keyword + ' AND infant[mh]'
elif agerange == 'Preschool Child: 2-5 years': keyword = keyword +
' AND child, preschool[mh]'
elif agerange == 'Child: 6-12 years': keyword = keyword + ' AND child[mh]'
elif agerange == 'Adolescent: 13-18 years': keyword = keyword +
' AND adolescence[mh]'
elif agerange == 'Adult: 19-44 years': keyword = keyword + ' AND adult[mh]'
elif agerange == 'Middle Aged: 45-64 years': keyword = keyword +
' AND middle age[mh]'
elif agerange == 'Aged: 65+ years': keyword = keyword + ' AND aged[mh]'
elif agerange == '80 and over: 80+ years': keyword = keyword +
' AND aged, 80 and over[mh]'

# Human or animal studies limit will be added to keyword if desired
if humananimal == 'Human': keyword = keyword + ' AND human[mh]'
elif humananimal == 'Animal': keyword = keyword + ' AND animal[mh]'

# Studies done on either females or males will be a limit of keyword
if gender == 'Female': keyword = keyword + ' AND female[mh]'
elif gender == 'Male': keyword = keyword + ' AND male[mh]'

# Past Entrez date range will be added to keyword; the number is the relative
#number of days prior to today
if entrezdate == '30 Days': entrezdate = '30'
elif entrezdate == '60 Days': entrezdate = '60'
elif entrezdate == '90 Days': entrezdate = '90'
elif entrezdate == '180 Days': entrezdate = '180'
elif entrezdate == '1 Year': entrezdate = '365'
elif entrezdate == '2 Years': entrezdate = '730'
elif entrezdate == '5 Years': entrezdate = '1825'
elif entrezdate == '10 Years': entrezdate = '3650'

# if date limits are provided, then the following will be added to keyword
# I will only allow this if the relative entrez date is not specified above,
#hence the elif command
# This is where I used the time function, gmtime() to get
# the current global mean time
```

```
elif fromdate != '':
  if todate == '':
     if pubdate == 'Publication Date': keyword = keyword + ' ' + fromdate + ':' +
     time.strftime('%Y/%m/%d', time.gmtime()) + '[dp]'
     elif pubdate == 'Entrez Date': keyword = keyword + ' ' + fromdate +':' +
     time.strftime('%Y/%m/%d',time.gmtime()) + '[edat]'
  else:
     if pubdate == 'Publication Date': keyword = keyword + ' ' + fromdate + ':' +
     todate + '[dp]'
     elif pubdate == 'Entrez Date': keyword = keyword + ' ' + fromdate + ':' +
     todate + '[edat]'

# Below is the actual call to the URL (PubMed's cgi): first to gain the pubmed UIDs
# and then to get the entries that is passed to pyblio to open
uids = query_info (keyword, field, maxcount, displaystart, entrezdate)
# get the pubmed UIDs and dump into uids variable

uids = string.replace (str(uids),'[','') # get rid of open bracket in string
uids = string.replace (str(uids),']','') # get rid of close bracket in the string
uids = string.replace (str(uids),' ','') # get rid of all the spaces in the string

if uids.strip () == '': return None

params = urllib.urlencode ({
    'db'     : 'pubmed',
    'report' : 'medline',
    'mode'   : 'text'
    })

url = "%s?%s&id=%s" % (fetch_url, params, str(uids))

file, data = urllib.urlretrieve (url)

return file
```