Refactoreo para Comprensión y Mantenimiento de Programas

Gustavo Villavicencio

FMA, UCSE

San Salvador de Jujuy, Agosto 2011

Table of contents

La siguiente clasificación es provista por Chikofsky y Cross [Chikofsky and II, 1990]:

Reingeniería de software

Consiste en el análisis y modificación de un sistema para transformalo en una forma distinta.

Ingeniería hacia adelante

Ingeniería de software convencional.

Ingeniería reversa o comprensión de programas

Analizar el código fuente para:

- Identificar los componentes del sistema y sus interrelaciones.
- Crear representaciones del sistema de un nivel superior al código.

Nota: Durante el proceso de comprensión de programa el sistema **NO** se altera.

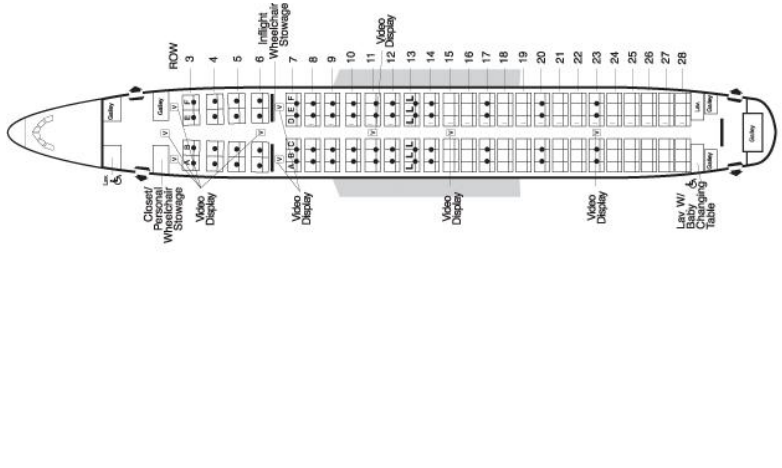
Terminología propia del entendimiento de programas

- Modelo mental
- Modelo cognitivo
- Soporte cognitivo
- Planes de programación
- Chunked code
- Referencia o señal
- Reglas de programación

Más detalles en [Storey, 2006].

La ingeniería aeronáutica:

- Como en mantenimiento de software, también usa **modelos mentales**.



- **Distinto al mantenimiento de software (recordar 1), hay un desensamblaje efectivo del artefacto**.

Necesidad del desensamblaje

- El artefacto es complejo.
- El artefacto es monolítico.

Ventajas del desensamblaje de un artefacto:

- Completa el entendimiento obtenido a partir del análisis de los modelos mentales.
- Corrige malos entendidos surgidos como consecuencia de modelos desactualizados.
- Facilita las tareas de mantenimiento.
- Permite detectar fallas futuras.

Comprensión de Programas en el Mantenimiento

Observaciones:

- La comprensión del artefacto de software se realiza mediante modelos mentales (descomposición virtual del artefacto).
- Al no haber un desensamblaje efectivo del artefacto el mantenimiento se realiza sobre el sistema complejo y monolítico.
- El mantenimiento sucesivo realizado en tales condiciones potencia la complejidad y la rigidez del sistema.

Cómo conseguir un desensablaje efectivo de un artefacto de software?

Refactoreo

Técnica destinada a cambiar la estructura del programa sin alterar la semántica [Fowler, 2010].

Motivación:

- Mejorar el entendimiento del sistema para favorecer su mantenimiento.
- Alentar el reuso.
- Mejorar eficiencia.

Más detalles en [Mens and Tourwé, 2004].

En el contexto funcional se propuso [Villavicencio, 2007, Villavicencio, 2011a]

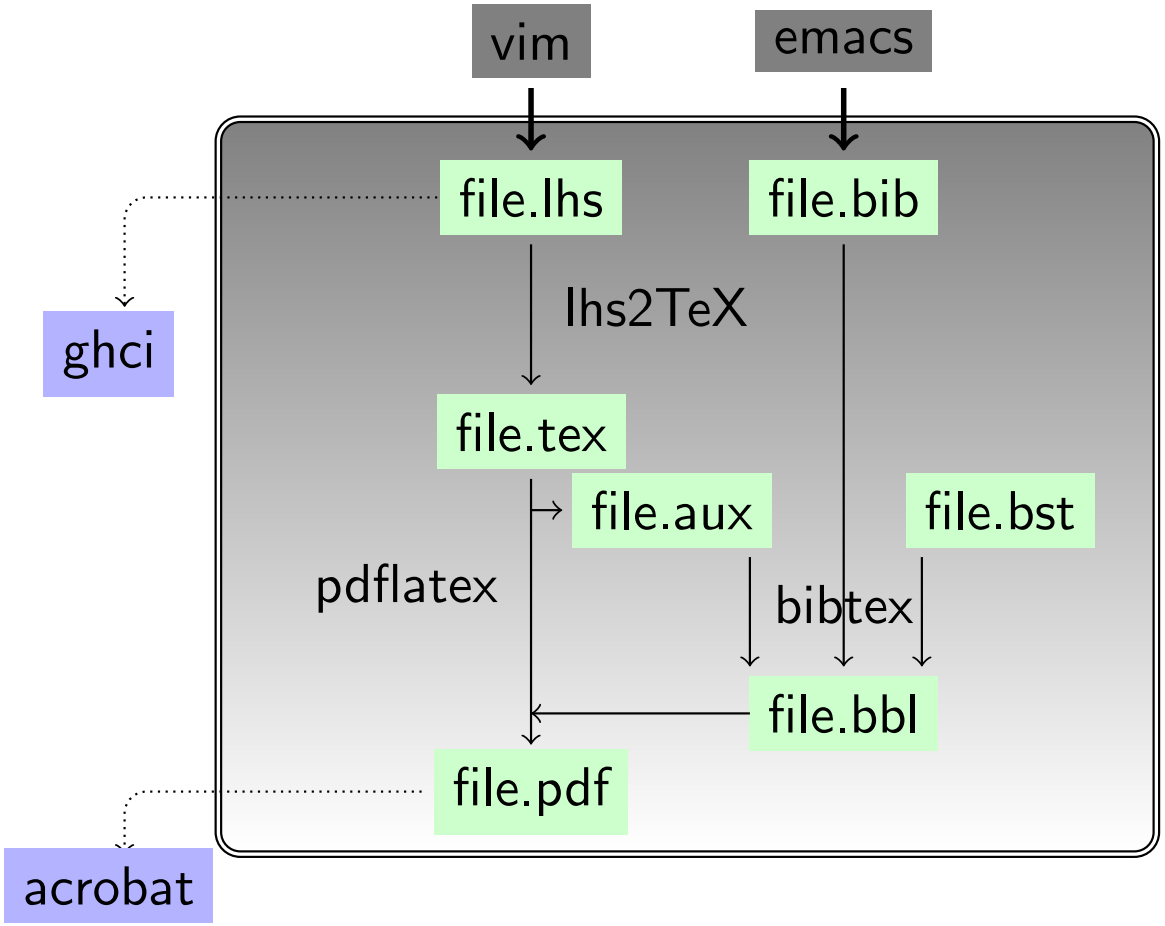
Refactoreo + Cálculo = Fórmula del programa

Proceso

Sobre el código fuente (en Haskell) se aplican técnicas de refactoreo hasta que se alcanza una representación que se pueda expresar en un diagrama conmutativo. Luego, por cálculo, se demuestra que la transformación es correcta.

$$
\begin{aligned}
&\mathit{cuentaCLiP}\ l = \mathit{loops}\ \ l\ \ (0, 0, 0, \mathit{False}) \\
&\mathit{loops}\ [\,]\ \ \ \ (c, w, l, b) \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ = (c, w, l, b) \\
&\mathit{loops}\ (h : t)\ (c, w, l, b) \mid \mathit{blanks}\ h \ \ = \mathit{loops}\ t\ (c + 1, w, l, \mathit{False}) \\
&\mathit{loops}\ (h : t)\ (c, w, l, b) \mid h \equiv \text{'\textbackslash n'} = \mathit{loops}\ t\ (c + 1, w, l + 1, \mathit{False}) \\
&\mathit{loops}\ (h : t)\ (c, w, l, \mathit{True}) \ \ \ \ \ \ \ \ \ \ \ = \mathit{loops}\ t\ (c + 1, w, l, \mathit{True}) \\
&\mathit{loops}\ (h : t)\ (c, w, l, \mathit{False}) \ \ \ \ \ \ \ \ \ = \mathit{loops}\ t\ (c + 1, w + 1, l, \mathit{True}) \\
&\mathit{blanks}\ c = (c \equiv \text{' '}) \vee (c \equiv \text{'\textbackslash t'})
\end{aligned}
$$

Breve Intermedio sobre Ingeniería de Documentos



Ejemplo (Continuación I)

Mediante la aplicación del refactoreo de división (splitting refactoring) el programa *cuentaCLiP* se descompone en los siguientes fragmentos (slices):

```
cuentaC l = loopc l 0
loopc []     c = c
loopc (h : t) c = loopc t (c + 1)
```

```
cuentaPal l = fst (loopPal l (0, False))
loopPal []     (w, b)              = (w, b)
loopPal (h : t) (w, b) | blanksw h = loopPal t (w, False)
loopPal (h : t) (w, True)          = loopPal t (w, True)
loopPal (h : t) (w, False)         = loopPal t (w + 1, True)
blanksw c = (c == '\n') || (c == ' ') || (c == '\t')
```

Ejemplo (Continuación II)

```
cuentaLi l = loopl l 0
loopl []     l           = l
loopl (h : t) l | h ≡ '\n' = loopl t (l + 1)
loopl (_ : t) l           = loopl t l
```

Este primer desensamblaje de *cuentaCLiP* conduce a reformular el mismo como sigue:

```
cuentaCLiPR1 l = (cuentaC l, cuentaPal l, cuentaLi l)
```

Observar:

- Hasta aquí la primera versión refactoreada de *cuentaCLiP* obtenida mediate la aplicación de splitting refactoring.
- La versión generada del programa es más **ineficiente** que la original.

Ejemplo (Continuación III)

Se aplica ahora, el refactoreo de eliminar los parámetros de acumulación.

- Se elimina el parámetro de acumulación del slice sobre *c*.

$$
\begin{aligned}
&\mathit{loopcr1}\ [\,] &&= 0\\
&\mathit{loopcr1}\ (h : t) &&= 1 + \mathit{loopcr1}\ t
\end{aligned}
$$

- Se elimina el parámetro de acumulación del slice sobre *l*.

$$
\begin{aligned}
&\mathit{looplr1}\ [\,] &&= 0\\
&\mathit{looplr1}\ (h : t) \mid h \equiv \texttt{'\textbackslash n'} &&= 1 + \mathit{looplr1}\ t\\
&\mathit{looplr1}\ (\_ : t) &&= \mathit{looplr1}\ t
\end{aligned}
$$

Ejemplo (Continuación VI)

- Se elimina el parámetro de acumulación del slice sobre w.
  - Pero antes se aplica folding refactoring!

```
cuentaPal2 l = fst (loopPalr1 l (0, False))

loopPalr1 []     (cw, bool) = (cw, bool)
loopPalr1 (h:t) (cw, bool) = loopPalr1 t (aux (cw, bool) h)

aux (c, b)     h | blanksw h = (c, False)
aux (c, True)  h             = (c, True)
aux (c, False) h             = (c + 1, True)
```

Ahora si se remueve el parámetro de acumulación

```
cuentaPal3 l = (fst ∘ loopPalr2) l

loopPalr2 []     = (0, False)
loopPalr2 (h:t) = aux (loopPalr2 t) h
```

Ejemplo (Continuación VII)

Pero la función *aux* puede aún ser refactoreada

$$
\begin{aligned}
&cuentaPal4\ l = (fst \circ loopPalr3)\ l \\
&loopPalr3\ [\,] \quad\quad = (0, False) \\
&loopPalr3\ (h : t) = (auxCuenta\ (loopPalr3\ t)\ h, auxBool\ (loopPalr3\ t)\ h) \\
&auxCuenta\ (c, b)\ h \mid (blanksw\ h \vee b \equiv True) = c \\
&auxCuenta\ (c, b)\ h \quad\quad\quad\quad\quad\quad\quad\quad = c + 1 \\
&auxBool\ (c, b)\ h = \neg\ (blanksw\ h)
\end{aligned}
$$

En consecuencia, la nueva versión refactoreada de *cuentaCLiP* es como sigue

$$
cuentaCLiPR2\ l = (loopcr1\ l, cuentaPal4\ l, looplr1\ l)
$$

Ejemplo (Final)

Observar:

1. En la última versión obtenida del programa se aplicaron los refactoreos de **remoción del parámetro de acumulación** y **folding**.
2. La versión obtenida es aún más **ineficiente** que la versión anterior también obtenida por refactoreo, y consecuentemente, también más **ineficiente** que la versión original.
3. **La versión original puede ser restituida aplicando la secuencia inversa de refactoreo, e invirtiendo cada uno de los refactoreos aplicados.**

La observación realizada en 2, es consistente con las afirmaciones realizadas en [Hu et al., 2006, Tullsen, 2002]. Esto es:

Eficiencia y entendimiento son propiedades opuestas de un programa.

Por lo tanto:

- No adherimos al entendimiento preventivo basado en refactoreo [Mens and Tourwé, 2004].
- El entendimiento preventivo no desensambla el sistema.
- El mantenimiento se realiza sobre el sistema complejo y monolítico potenciando aún más estas características.

El ítem 3 puede expresarse más formalmente como sigue:

- Los refactoreos destinados a desensamblar el programa se denominan reverse refactorings: $\overrightarrow{r_i}$ con $1 \leqslant i \leqslant n$, siendo n la cantidad de refactoreos en el catálogo.
- Los refactoreos destinados a restaurar el sistema se denominan forward refactorings: $\overleftarrow{r_i}$.
- Secuencia de refactoreo son los refactoreos aplicados sobre un sistema dado: $\overrightarrow{r_1}, \ldots, \overrightarrow{r_k}$, con $k \leqslant n$.
- Si se aplica la secuencia $\overrightarrow{r_1}, \ldots, \overrightarrow{r_k}$ para desensamblar el sistema, entonces la secuencia $\overleftarrow{r_k}, \ldots, \overleftarrow{r_1}$ es la que lo restaura.
- Por lo tanto, cada refactoreo de un tipo tiene su contraparte inverso de otro tipo: $\overleftarrow{r_i} = \overrightarrow{r_i}^{-1}$ y $\overrightarrow{r_i} = \overleftarrow{r_i}^{-1}$.

Más detalles en [Villavicencio, 2011b].

Un Nuevo escenario

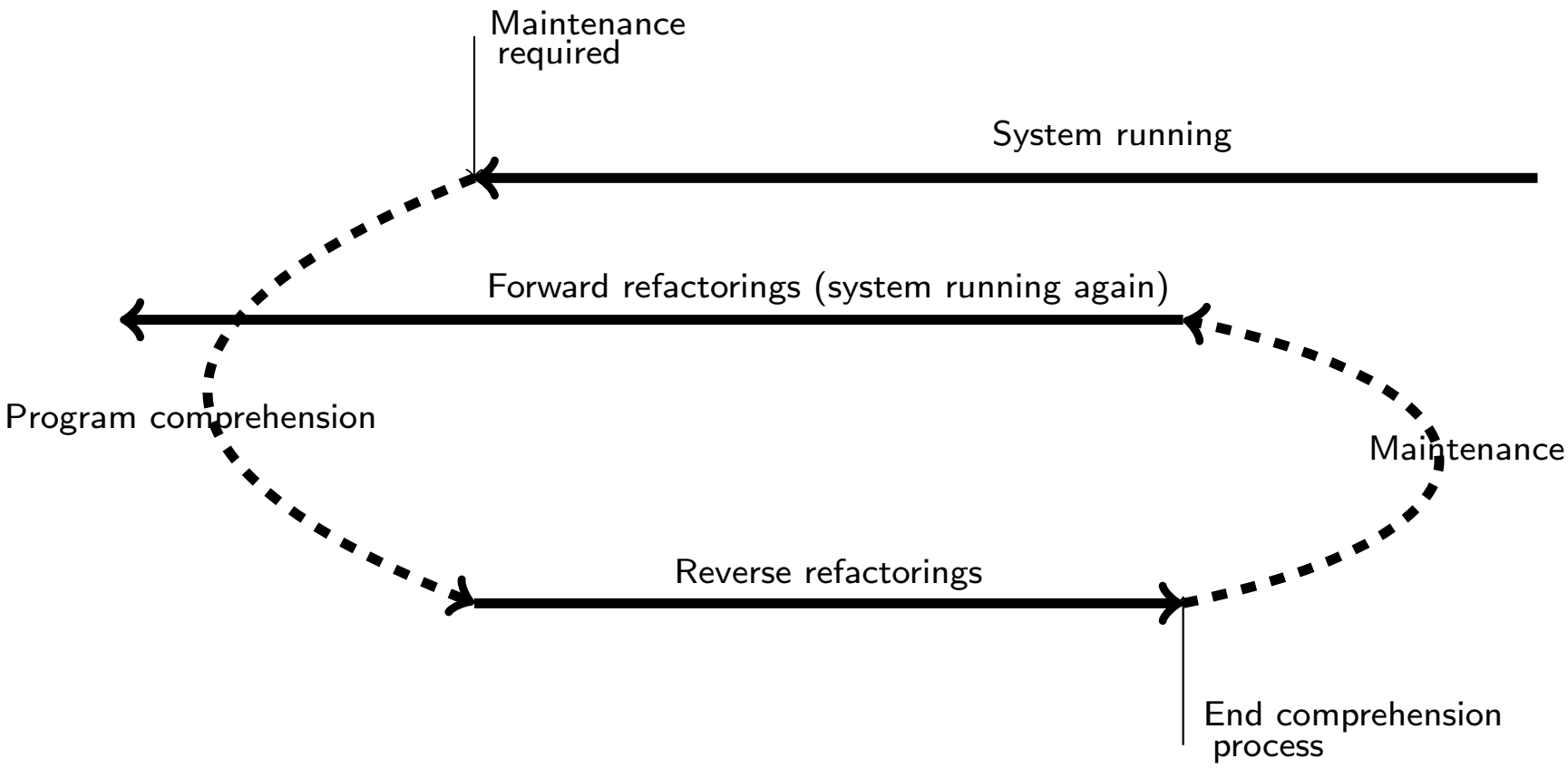


Figure: Maintenance scenario based on reverse and forward refactorings

Problemas emergentes

Del contexto anterior, surgen inmediatamente los siguientes interrogantes:

1. Dada una especificación de mantenimiento, cuál de las versiones obtenidas mediante reverse refactorings es la más apropiada para realizar el mantenimiento?
2. Luego de realizar mantenimiento sobre una versión desensamblada, podemos siempre restaurar el sistema mediante forward refactorings?
3. Cuál sería el mejor modo de cotejar las especificaciones de mantenimiento con las versiones del programa obtenidas por refactoreo?
4. Mejora contínua de las técnicas de refactoreo en ambas direcciones.
5. Los refactoreos forward y reverse, pueden modelarse como **transformaciones bidireccionales**?.

Posibles soluciones

Existen algunos indicios de solución a los problemas planteados. Específicamente:

- El interrogante planteado en el ítem 2 puede ser abordado mediante patrones de mantenimiento [Hammouda and Harsu, 2004].
- El aspecto referido en el item 4 puede ser abordado mediante el trabajo desarrollado en [Prete et al., 2010].
- Más investigación debemos realizar sobre los ítems restantes.

Desde otra perspectiva: Transformaciones bidireccionales

Alternativamente el mantenimiento soportado por refactoreo puede plantearse como un problema a resolver mediante transformaciones bidireccionales.

Transformación emparejada de software

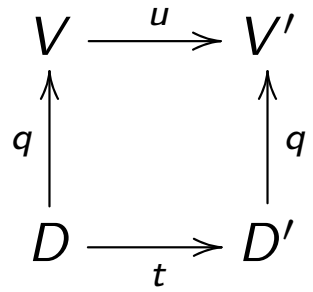
Múltiples artefactos de software se modifican para mantener la consistencia entre ellos.

Transformaciones bidireccionales

Caso particular de transformación emparejada destinadas a definir transformaciones coherentes de reconciliación entre dos artefactos de software para restablecer en cualquier momento la consistencia entre ellos.

Actualización de vistas

En base de datos relacionales se plantea el siguiente problema: La modificación de una vista genera una modificación en la base de datos.



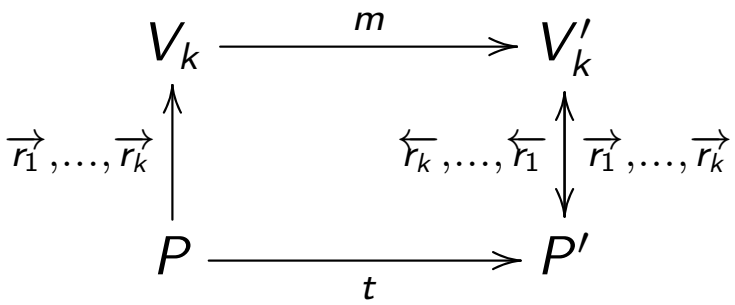
(1)

- q: Consulta a la base que genera la vista
- u: Actualización sobre la vista.

Problema

Cuál es la transformación t que refleja u?

Particularización al mantenimiento basado en refactoreo



(2)

- P : Programa original
- m : Mantenimiento realizado sobre la versión más comprensible del programa

Problema

Efectuado el mantenimiento m sobre la versión comprensible V_k obtenida por refactoreo, cuál es la transformación t a realizar sobre el programa original (monolítico e incomprensible) para obtener el programa modificado P'?

Fin. Gracias!


Chikofsky, E. and II, J. H. C. (1990).
Reverse engineering and design recovery: a taxonomy.
IEEE Software, 7(1):13–17.


Fowler, M. (2010).
Refactoring home page.
http://www.refactoring.com.


Hammouda, I. and Harsu, M. (2004).
Documenting maintenance tasks using maintenance patterns.
In *Proceedings of the Eighth European Conference on Software Maintenance and Reengineering*, Tampere, Finland. IEEE.


Hu, Z., Yokoyama, T., and Takeichi, M. (2006).
Optimizations and transformations in calculation form.
In Lämmel, R., Saraiva, J., and Visser, J., editors, *Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, volume 4143 of *LNCS*, Braga, Portugal. Springer-Verlag.


Mens, T. and Tourwé, T. (2004).
A survey on software maintenance.
IEEE Transactions on Software Engineering, 30(2):126–139.


Prete, K., Rachatasumrit, N., Sudan, N., and Kim, M. (2010).
Template-based reconstruction of complex refactorings.
In *26th IEEE International Conference on Software Maintenance*, Timisoara, Romania. IEEE.


Storey, M.-A. (2006).
Theories, tools and research methods in program comprehension: past, present and future.
Software Quality Control, 14(3):187–208.


Tullsen, M. A. (2002).
PATH, a Program Transformation System for Haskell.
PhD thesis, Yale University.


Villavicencio, G. (2007).
Refactoring for comprehension.
In *Draft Proceeding of the 8th. Trends in Functional Programming*, New York, USA. Seaton Hall University.


Villavicencio, G. (2011a).
A bottom-up approach to understand functional programs.
In *Fourth International C* Conference on Computer Science & Software Engineering (C3S2E 2011)*, Montreal, Quebec, Canada. ACM.


Villavicencio, G. (2011b).
Software maintenance supported by refactoring.
In *The 2011 International Conference on Software Engineering Research and Practice*, Las Vegas, Nevada, USA.
To appear.