41 Jornadas Argentinas de Informática

Concurso de Trabajos Estudiantiles 2012

Categoría:

Ingeniería del software, Bases de datos, Data warehouses y Minería de datos.

Título:

Desarrollo Distribuido de un Juego On-line utilizando Diseño por Contratos.

Autor:

Riberi, Franco Gaspar.

Docente:

Aguirre, Nazareno Matías.

Cátedra:

Ingeniería de Software Distribuida y Tercerizada (DOSE).

Institución:

Universidad Nacional de Río Cuarto.

Desarrollo Distribuido de un Juego On-line utilizando Diseño por Contratos

21 de mayo de 2012

Resumen

En este breve artículo, presentamos *Guess who?*, un juego on-line desarrollado de manera distribuida utilizando Diseño por Contractos. Este juego fue desarrollado de manera distribuida como trabajo de cátedra, en el marco de la asignatura Ingeniería de Software Distribuida y Tercerizada, la cual incluye el desarrollado de un proyecto, involucrando diferentes universidades del mundo, y coordinado por ETH Zurich.

Este proyecto, como todas las actividades de programación de la materia, fue desarrollado utilizando una metodología de desarrollo Orientada a Objetos denominada Diseño por Contractos. Además de los desafíos técnicos que este proyecto generó, por el uso de diferentes tecnologías de apoyo al desarrollo de software, el mismo generó desafíos comunicacionales y de interacción, propios del desarrollo en conjunto con personas de otros países, distribuidas geográficamente, que hablan otro idioma, etcétera.

Palabras clave: Desarrollo distribuido, Diseño por Contratos, Eiffel, Programación Orientada a Objetos (POO).

1. Introducción

Actualmente, el desarrollo de proyectos de software ha cruzado los límites de las fronteras en busca de nuevos desarrolladores, pasando de desarrollos locales a proyectos geográficamente distribuidos. Esta distribución geográfica plantea nuevos retos en cuanto a comunicaciones, especificaciones de requisitos de software (SRS), especificaciones de interfaz (API), gestión de proyectos, entre otros. Todos estas tareas deben realizarse ahora de manera distribuida, involucrando actores de diferentes culturas situados en regiones con diferentes cursos horarios, etcétera.

Luego, el desarrollo distribuido de software plantea problemas considerables. Si el desarrollo de un proyecto exitoso cuando todo sus integrantes se encuentran en el mismo sitio es una tarea difícil, imaginemos ahora dividiendo el equipo en diferentes continentes, zonas horarias, idiomas y culturas. Estos obstáculos pueden llevar a importantes retrasos o incluso a completos fracasos.

Con el objetivo de entrenar a los alumnos en el tipo de problemáticas mencionadas, ETH Zurich organiza desde hace ya varios años la asignatura **D**istributed and **O**ursourced **S**oftware **E**ngineering) (DOSE) [1]. La misma involucra un taller de desarrollo de software de forma distribuida, en colaboración con otras universidades situadas alrededor del mundo, coordinado por ETH Zurich.

Este proyecto no sólo brinda la oportunidad de enfrentarse a los retos del desarrollo de software distribuido, sino que también ayuda a comprender los típicos problemas de la ingeniería en software.

Desde 2010, la Universidad Nacional de Río Cuarto participa de DOSE. La asignatura que localmente incluye un curso previo de Diseño por Contratos, se denomina Ingeniería de Software Distribuida y Tercerizada. Este trabajo se realizó en el marco de la edición 2011 de DOSE, de la cual participaron universidades de Vietnam[1], Suiza[2], Italia[3], Rusia[4], Dinamarca[5], Hungría[6], España[7], Ucrania[8] y Argentina[9].

Describiremos las características del proyecto, los desafíos enfrentados y las lecciones aprendidas.

2. Preliminares

2.1. Desarrollo de Software Distribuido

El desarrollo de software de manera distribuida genera desafíos diferentes a los proyectos. Al involucrar diferentes equipos de personas que trabajan en un mismo proyecto situados en diferentes puntos geográficos, los miembros del proyecto no se pueden ver uno al otro cara a cara, pero todos trabajan en colaboración para obtener el mismo resultado final del proyecto. A menudo esto se hace a través de correos electrónicos, internet y otras formas de rápida comunicación a larga distancia.

Este tipo de proyectos enfrenta diferentes problemas tales como la comunicación (distancia, herramientas y métodos), la coordinación de actividades, deadlines, etcétera, diferencias culturales y/o de idioma, diferencias horarias, aislamiento, cómo asegurar la compresión entre personas si no están cara a cara.

El desarrollo distribuido presenta una serie de ventajas que lo hace sumamente aplicado en la práctica hoy en día, como un mayor acceso al talento (se puede trabajar con los mejores), mejor productividad, se organiza y aprovecha mejor el tiempo, comunicación síncrona y asíncrona, la estructura es mínima, los costos son bajos, se reduce el desplazamiento, etcétera.

2.1.1. Distribución del trabajo

En un desarrollo distribuido es necesario la división del trabajo entre los diferentes equipos. Para ello, existen esencialmente dos formas ortogonales:

- *Basado en procesos:* corresponde a la división tradicional del proceso de software (elicitación de requisitos, análisis, diseño, implementación, ejecución y prueba) y cualquiera de sus derivados.

[1]Hanoi University of Science and Technology, http://en.hustech.edu.vn.
[2]ETH Zurich, http://www.ethz.ch y University of Zurich, http://www.uzh.ch.
[3]Politecnico di Milano, http://www.polimi.it.
[4]ITMO, http://en.ifmo.ru.
[5]IT University of Copenhagen, http://www.itu.dk.
[6]University of Debrecen, http://www.lib.unideb.hu.
[7]Universidad Politécnica de Madrid, http://www.upm.es.
[8]Odessa Polytechnic National University, www.opu.ua.
[9]Universidad Nacional de Río Cuarto, www.unrc.edu.ar.

- *Basado en Clusters:* utilizando el concepto de "cluster" desde el desarrollo Orientado a Objetos [2]. En este enfoque, el sistema se descompone en un número, generalmente pequeño, de grupos formados por equipos. Cada equipo es responsable de un subsistema particular.

2.2. Diseño por Contratos

El Diseño por Contratos (DbC del inglés **D**esign **b**y **C**ontract) [2] [3], también conocido como Programming by Contract (Programación por Contratos), o Contract Programming (Programación Contractual), o Contract-first Development (Desarrollo con contrato primero) es una metodología para el diseño e implementación de aplicaciones y componentes popularizada inicialmente por el lenguaje de programación Eiffel. Básicamente consiste en tomar los elementos de diseño como participantes de una relación similar al contrato de negocios. Así, se pueden diseñar los componentes asumiendo que se cumplirán ciertas condiciones de entrada (*pre-condiciones*), mientras que se deberán garantizar ciertas condiciones de salida (*post-condiciones*), así como el invariante de clase[10]. Las post-condiciones e invariantes de clase se utilizan para ayudar a garantizar la corrección del programa sin sacrificar la eficiencia.

La idea central de DbC es una metáfora sobre cómo interactúan los elementos de un sistema de software para colaborar entre si, basándose en obligaciones y beneficios mutuos. La metáfora proviene del mundo de los negocios, en donde un "cliente" y un "proveedor" firman un "contrato" que define por ejemplo:

"El proveedor debe brindar cierto producto (obligación) y tiene derecho a que el cliente le pague una cuota (beneficio). El cliente paga una cuota (obligación) y tiene derecho a obtener el producto (beneficio)".

Ambas partes deben satisfacer ciertas obligaciones, como leyes y regulaciones, que se aplican a todos los contratos. De manera similar, si una rutina de una clase en programación orientada a objetos brinda cierta funcionalidad, podría:

- *Imponer ciertas obligaciones que se garanticen por cualquier módulo cliente que la invoque*: **la pre-condición de la rutina**. Corresponde a una obligación del cliente, y un beneficio para el proveedor (la rutina en si misma), ya que la libera de tener que gestionar casos por fuera de la precondición.

- *Garantizar cierto comportamiento a la salida*: **la post-condición de la rutina**. Corresponde a una obligación del proveedor, y obviamente un beneficio para el cliente.

- *Mantener cierta propiedad, asumida al momento de la entrada y garantizada a la salida*: **los invariantes de la clase**.

En conclusión, el contrato es la formalización entre obligaciones y beneficios de las rutinas invocadas e invocados. Se podría resumir al Diseño por Contrato en las siguientes tres preguntas que el diseñador del componentes debe preguntarse: *¿Qué espera?*, *¿Qué garantiza?* y *¿Qué mantiene?*

[10]Propiedades que se mantienen invariantes a pesar del procesamiento realizado por el componente.

Muchos lenguajes brindan facilidades para hacer verificaciones como estas. Sin embargo, DbC considera que los contratos son cruciales para crear software correcto, y que deben ser parte del proceso de diseño del software. De hecho, DbC fomenta escribir primero las aserciones.

Cuando se utilizan contratos, el código del programa por si mismo nunca debe intentar verificar las condiciones del contrato: la idea es que el código debe provocar una falla, siendo la verificación del contrato la red de contención. Este mecanismo de provocar fallos de DbC simplifica la depuración, ya que el comportamiento requerido para cada rutina está claramente especificado. Nunca se deben violar las condiciones del contrato en la ejecución del programa; por lo tanto se lo puede dejar activado para depuración y testing, o quitado completamente del código cuando éste está listo para ser enviado por motivos de rendimiento.

El Diseño por Contratos también facilita la reutilización de código, ya que el contrato para cada pieza de código documenta completamente la misma.

2.3. Lenguaje de Programación Eiffel

Eiffel [4][9][10] es un método de construcción de software y un lenguaje aplicable al análisis, diseño, implementación y mantenimiento de sistemas informáticos. Fue desarrollado originalmente por *Eiffel Software*, una empresa fundada por Bertrand Meyer[11]. El diseño de Eiffel se basa en la la programación orientada a objetos. El mismo tiene como objetivo permitir a los programadores la creación de módulos de software confiables y reutilizables. Eiffel soporta herencia múltiple, genericidad, polimorfismo, encapsulamiento, conversiones con seguridad de tipos, entre otros conceptos de programación. La contribución más importante de Eiffel a la ingeniería de software es el Diseño por Contratos. Este concepto es fundamental para Eiffel. Las declaraciones de pre-condición y post-condición se definen mediante las cláusulas require y ensure, respectivamente. Por ejemplo:

```
hanoi(n:INTEGER; desde:STRING; hasta:STRING; aux:STRING):
        LIST[STRING]
    require
        n>=0 and desde /= Void and hacia /= Void and aux /= Void
    do
        ...
    ensure
        Result.count = 2^n - 1
end
```

El compilador de Eiffel está diseñado para incluir la función y los contratos de clase en los distintos niveles. Eiffel Studio, por ejemplo, ejecuta todas las funciones y los contratos de clase durante la ejecución en el "modo de banco de trabajo". Cuando un archivo ejecutable se crea, el compilador se instruye por medio del archivo de configuración del proyecto para incluir o excluir a cualquier conjunto de contratos.

[11] Investigador, escritor y consultor en el campo de los lenguajes, nacido en Francia en 1950.

2.4. Eiffel Studio

Eiffel Studio es el IDE de desarrollo utilizado para Eiffel. Es multiplataforma y está disponible bajo un modelo de licencia dual, es decir, los usuarios pueden elegir cualquiera de las licencias comerciales o de código abierto.

3. Proyecto

3.1. Ediciones Previas

Este tipo proyecto se ha realizado previamente en distintas ediciones de la materia DOSE [5][12]. Cada edición cuenta con una evaluación [1] y una colección de datos recolectados durante las diferentes etapas por las cuales se atravesaron. De este modo, cada nueva edición intenta reparar las falencias detectadas mediante la experiencia, para lograr mejores desempeños en el desarrollo distribuido.

3.2. Descripción

El proyecto consistió en el desarrollo distribuido de una plataforma de juegos on-line basada en POO a través del Diseño por Contratos. Participaron un total de 10 universidades, nacionalizadas en los países ya mencionados. Dicha plataforma consiste en una interfaz general (Figura 1), desde la cual los jugadores pueden seleccionar un juego. Cuenta con 13 juegos, 12 de los cuales fueron desarrollados por diferentes grupos, y el restante fue provisto por la cátedra como ejemplo.

Para alojar el proyecto se utilizó el marco Origo [11]. La creación de un proyecto sobre Origo ofreció foros de discusión (usuarios, desarrolladores, dueños de proyectos), páginas wiki, apoyo a la gestión de configuración (Subversión), entre otros mecanismos.

El éxito del proyecto no sólo dependió del éxito de un equipo (local), sino que también dependió de los equipos socios distribuidos en diferentes países.

Figura 1: GUI principal del proyecto.

3.3. Planificación

El proyecto se distribuyo usando el enfoque basado en *clusters* o *subsistemas*. Cada grupo estuvo conformado por 2 o 3 equipos situados en diferentes países. A su vez, cada equipo estuvo conformado por 3 estudiantes como máximo. Las divisiones de subcomponetes dentro de cada grupo fueron:

- Lógica.
- Interfaz gráfica de usuario (GUI) y comunicación de red (NET).
- Inteligencia artificial (AI).

Este enfoque tiene una ventaja pedagógica importante dado que obliga a cada equipo a interactuar con el resto del grupo, particularmente para centrarse en la definición de las interfaces de programa (API).

3.4. Implementación

3.4.1. Fase 0: Configuración

Es esta etapa inicial, cada participante alumno completó información personal en la plataforma Origo, tal como nombre, ID de skipe, entre otros. Además cada equipo, eligió preferentemente un componente a desarrollar (GUI+NET, Lógica o AI) según el cual se formaron posteriormente los grupos.

3.4.2. Fase 1: Documento de alcance y Primera Comunicación

Según las preferencias de cada equipo, el grupo 1 quedó conformado por:

- *Argentina*: GUI y NET.
- *Italia*: Lógica.
- *Vietnam*: AI.

Una vez conformado el grupo, se realizó la primera comunicación. En la misma cada integrante de equipo se presentó frente al resto del grupo y en conjunto se escogió un juego a implementar el cual fue presentado ante ETH Zurich. El juego elegido fue el Guess Who? con algunas extensiones (e.g., soporte de multi-idioma y sala de chat).

Guess Who? [13] también conocido como *Adivina quién* o *¿Quién es Quién?* es un juego de adivinar para dos jugadores. Básicamente, cada jugador dispone de un tablero idéntico que contiene 24 dibujos de personajes identificados por su nombre. El juego empieza cuando cada jugador selecciona una carta al azar que contiene uno de los personajes disponibles. El objetivo del juego es determinar qué carta seleccionó el oponente a través de preguntas cuya respuesta es por si o por no, para eliminar candidatos. El juego finaliza cuando un jugador adivina la carta de su oponente.

Luego de la aceptación del mencionado juego, se continuó con el desarrollo del documento de alcance. En el mismo se definieron las reglas del juego, esto involucra, acciones válidas, fin de juego, entre otras. Se definió además el alcance del juego, es decir, se marcaron los límites del proyecto, describiendo que será implementado y que no. Se describió una primer aproximación de la arquitectura a utilizar la cual se compone de un sólo servidor central y una serie de clientes que se conectarán al mismo como se observa en la Figura 2. Se definieron los roles de cada integrante y las herramientas a utilizar para la comunicación.

Básicamente, se proporcionan dos modos de juego, online y offline, es decir, el jugador puede jugar Humano vs Humano (HvH), o Humano-Computadora (HvAI), respectivamente. Cuando un cliente quiere iniciar un juego HvH, se conecta al servidor, el mismo busca un oponente (si existe), y actúa como nodo de comunicación. No hay ninguna funcionalidad de autenticación, todos los jugadores son anónimos.

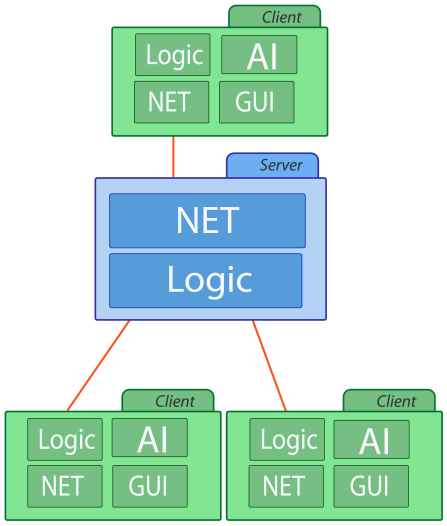

Figura 2: Diseño Arquitectónico

3.4.3. Fase 2: Documento de requisitos

En esta fase, cada equipo definió su documento de especificación de requisitos respetando las recomendaciones de la "Guía para la especificación de requerimientos de la IEEE" [14]. Este documento sirvió para identificar los requisitos específicos de cada componente, además de utilizarse para la verificación y validación de los mismo.

Algunos de los requisitos resultantes de esta etapa para el componente GUI+NET fueron: nuevo juego, interfaz gráfica por cada jugador, selección de personaje, construcción de una serie de preguntas, seleccionar un idioma, establecer comunicación, conexión con el componente lógico, sala de chat, entre otras. La funcionalidad de sala de Chat es una charla básica entre dos jugadores. Cada jugador es identificado con un nombre de usuario. Con el fin de generar las preguntas la interfaz gráfica de usuario muestra una serie de opciones posibles. Por ejemplo, el usuario marca la opción "tiene", a continuación, marca la opción "gafas", que forman la pregunta: "¿tiene gafas?".

El componente GUI también es responsable de la captura de los eventos desencadenados por el usuario y la invocación de features correspondientes, en especial los proporcionados por otros componentes (principalmente por la LÓGICA).

3.4.4. Fase 3: Especificación de interfaces usando contratos

Cada equipo definió su API haciendo uso de los contratos. Se definieron las interfaces de clases, definiendo para cada método su perfil, pre-condición y post-condición, estableciendo de esta manera un compromiso por cada componente a implementar. En esta etapa quedaron registrados los métodos ofrecido y requeridos por cada componente. Por ejemplo, el componente GUI+NET especificó el feature *onUpdate* el cual toma como parámetros una lista de cambios en el juego y actualiza la interfaz con ellos. El mismo deberá ser invocado desde el componente LÓGICA.

```
onUpdate (list's_change: LINKED_LIST [TUPLE[INTEGER,ANY]]) : Void
    require: list's_change \= Void and
             list's_change respeta un cierto formato.
    do
        ....
    ensure: la interfaz será actualizada.
```

3.4.5. Fase 4: Implementación en Eiffel

Esta etapa correspondió a la implementación propiamente dicha del juego. Fue necesario realizar constantes commits para mantener el proyecto actualizado para todo el grupo. En esta etapa se realizó la unión de los componentes implementados por cada equipo.

En la Figura 3 se pueden observar diferentes GUIs del *Guess Who?*.

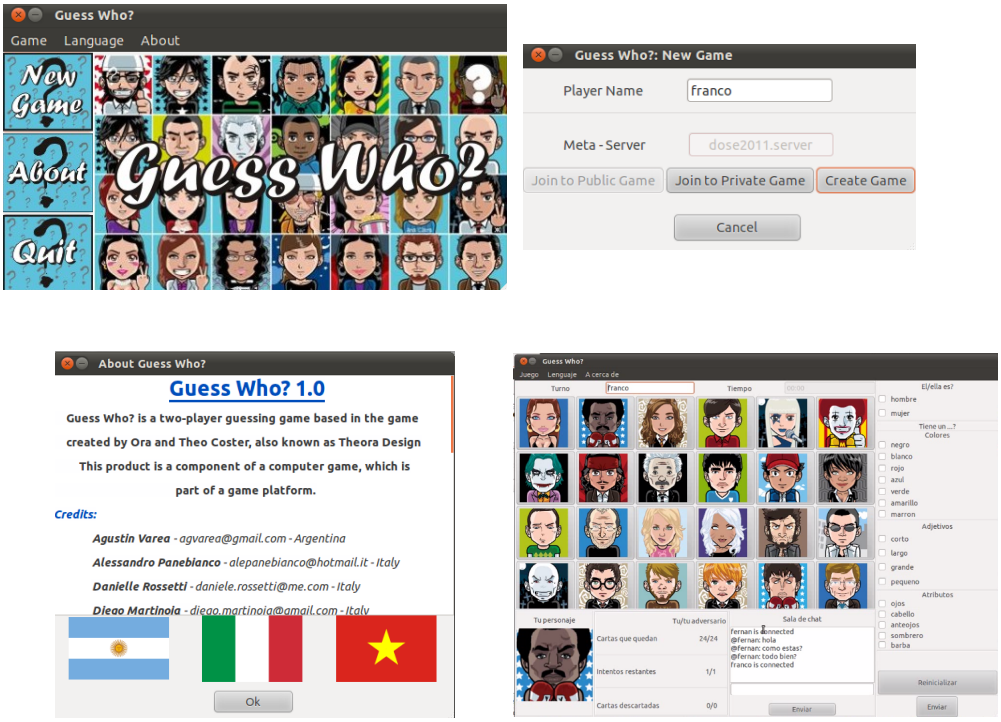

Figura 3: GUIs del juego *Guess Who?*.

3.4.6. Fase 5: Prueba

Esta etapa no sólo involucró el testing de cada componente por separado, sino que también su integración. Asimismo, fue necesario realizar casos de prueba desde el inicio del desarrollo, para poder testear cada componente independientemente de la implementación del resto.

3.4.7. Presentación y demo del proyecto ante ETH.

Por último, se realizó una presentación y una demo exhibiendo las funcionalidades del juego implementado.

Es importante destacar que cada fase contó con su respectivo deadline, además de una encuesta a cerca del esfuerzo involucrado, como así también tiempos, dificultades, entre otros atributos.

4. Conclusión

Los proyectos de desarrollo de software distribuido son más comunes día a día, por lo que esta experiencia fue totalmente productiva, no sólo en lo que respecta a lo profesional sino también en lo humano.

Desde el punto de vista profesional, este proyecto resaltó las condiciones que se dan al encarar el desarrollo de un proyecto de software de manera distribuida. Se comprendieron de forma clara los desafíos del desarrollo real del software en la industria. Se aplicaron diferentes conceptos y técnicas de la ingeniería de software. Se observaron las ventajas del uso de aserciones y especificaciones en el desarrollo de software. Se observó el grado de importancia en la definición de interfaces.

Desde el punto de vista humano, fue totalmente enriquecedor, dado que involucró desembolverse en un sistema distribuido, enfrentándose a los problemas propios de este desarrollo, tales como los horarios, balance entre vida personal y profesional, el idioma (no nativo), entre otros ya mencionados. Además, este proyecto fomentó el intercambio de culturas y conocimientos, el trabajo responsable en grupo involucrando estudiantes de prestigiosas universidades internacionales, validando así el aprendizaje local.

Referencias

[1] *"Teaching Software Engineering using Globally Distributed Projects: The DOSE Course"*. Nordio M., Di Nitto E., Ghezzi C., Tamburrelli G., Meyer B., Tschannen J., Kulkarni V. and Aguirre N., ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA. Copyright 2011 ACM 978-1-4503-0590-7/11/05.

[2] *"Object Oriented Software Construction", 2nd. Edition.* B. Meyer, Prentice-Hall, 2000.

[3] *"The Role of Contracts in Distributed Development"*. M. Nordio, R. Mitin, B. Meyer, C. Ghezzi, E. D. Nitto, and G. Tamburelli, In SEAFOOD, 2009.

[4] *"Eiffel: Analysis, Design and Programming Language"*. Bertrand Meyer, ISO/ECMA Eiffel standard, June 2006. Disponible en: http://www.ecma-international.org/publications/standards/Ecma-367.htm.

[5] *"The Allure and Risks of a Deployable Software Engineering Project: Experiences with Both Local and Distributed Development"*. Meyer B. and Piccioni M..

[6] *"Advanced Hands-on Training for Distributed and Outsourced Software Engineering"*. Nordio M., Mitin R. and Meyer B., Conference'04, Month 1–2, 2004, City, State, Country. Copyright 2004 ACM 1-58113-000-0/00/0004.

[7] *"How do Distribution and Time Zones affect Software Development? A Case Study on Communication"*. Nordio M., Estler C., Meyer B., Tschannen J., Ghezzi C. and Di Nitto E..

[8] *"Tutorial Origo"* Disponible en: http://www.origo.ethz.ch/wiki/origo_tutorial.

[9] *"Documentación Eiffel"* Disponible en: http://docs.eiffel.com/.

[10] *"An Eiffel Tutorial"* B. Meyer, First published July 2001. Corresponds to release 5.0 of the ISE Eiffel environment.

[11] *"Origo DOSE 2011"* ETH Zurich. http://dose2011.origo.ethz.ch.

[12] *"Dose 2010 course material and videos"* ETH Zurich. http://se.inf.ethz.ch/teaching/2010-H/dose-0273/index.html#slides.

[13] *"Guess Who?"* http://en.wikipedia.org/wiki/Guess_Who.

[14] *"IEEE Recommended Practice for Software Requirements Specifications"* Copyright © 1998 by the Institute of Electrical and Electronics Engineers, Inc. All rights reserved. Published 1998. Printed in the United States of America. ISBN 0-7381-0332-2.