Tema 5. Generación y Depuración de Aplicaciones

Contenidos

5.1 Concepto de plataforma. Software independiente de plataforma

5.2 Plataformas para el desarrollo de aplicaciones

5.3 Técnicas de depuración de programas

Objetivos

- Conocer conceptos y propiedades de calidad (portabilidad, productividad, etc.) a incorporar en el proceso de desarrollo de software.
- Conocer los fundamentos de los procesos de generación y depuración de aplicaciones.
- Conocer técnicas a aplicar en las diferentes fases de generación y desarrollo de programas.
- Comprender la importancia de plataformas y entornos de desarrollo y depuración de aplicaciones.

Bibliografía básica

- **[Agans06]** David J. Agans: "***Debugging: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems***", Amacom, 2006
- **[Bottcher18]** Evan Bottcher. "***What I Talk About When I Talk About Platforms".*** Disponible en: https://martinfowler.com/articles/talk-about-platforms.html (Último acceso: Abril 2020)
- **[Dooley16]** John F. Dooley. "***Software Development, Design and Coding: With Patterns, Debugging, Unit Testing, and Refactoring***", Apress, 2017
- **[Stallman18]** Richard Stallman, Roland Pesch, Stan Shebs. "***Debugging with GDB: The GNU Source-Level Debugger***", 12th Media Services, 2018
- **[Zeller09]** A. Zeller: "***Why Programs Fail***" *(2nd Edition),* Morgan Kaufmann Publishers (Elsevier), 2009

Concepto de Plataforma

Plataforma: Combinación de hardware y/o software utilizada para ejecutar aplicaciones software.

La versión más simple de plataforma puede ser una arquitectura de computadora o sistema operativo. Otros ejemplos son los navegadores web y sus plugin asociados, las máquinas virtuales (ejemplo, JVM), las plataformas de computación en la nube (Amazon Web Services, Google Cloud, Microsoft Azure, IBM Bluemix, etc.).

Ejemplos de plataformas:

Sistema Operativo	Arquitectura Hardware
Microsoft Windows	x86
Linux/Unix	X86, RISC, SPARC
Mac OS X	X86, PowerPC
Android	Dispositivos móviles basados en arquitecturas ARM, MIPS y x86
Java	Múltiples SOs para los que existen implementaciones de *Java Virtual Machine* , y por tanto múltiples arquitecturas

Clasificación del software

En cuanto a plataformas, el software se puede clasificar como:

1. **Dependiente de la plataforma particular** para la cual se desarrolla y ejecuta (bien sea una plataforma hardware, sistema operativo o máquina virtual)

2. **Multiplataforma,** cuando el software se ha desarrollado e interopera en varias plataformas.

Una **aplicación multiplataforma** se puede ejecutar en:

1. tantas plataformas como existan (caso ideal de **software independiente de la plataforma**), o

2. tan sólo en dos plataformas diferentes, por ejemplo, una aplicación multiplataforma se podría ejecutar en *Microsoft Windows* y *Linux* en arquitecturas x86.

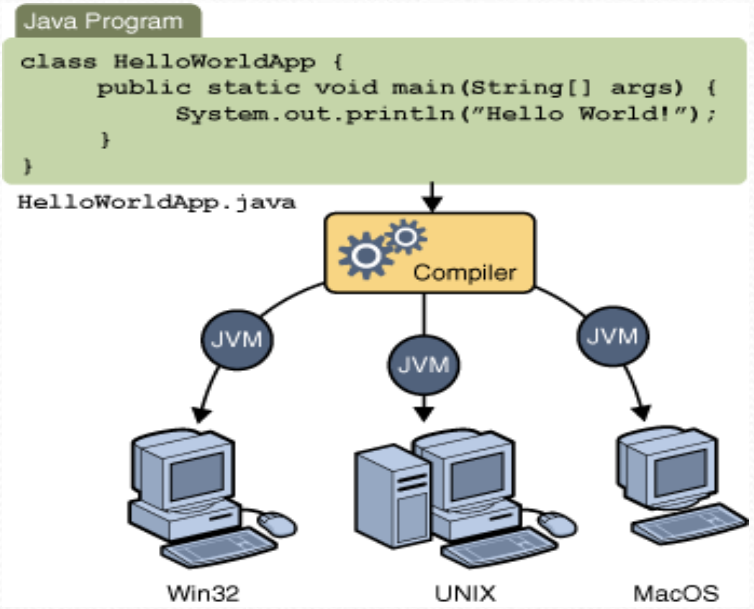
Software multiplaforma

El software multiplataforma puede dividirse en **dos tipos**:

1. Aplicaciones que requieren su **creación o compilación para cada plataforma** específica donde se ejecutarán.

2. **Aplicaciones que directamente se pueden ejecutar** en más de una plataforma sin preparación especial (p. ej., plataforma Java):

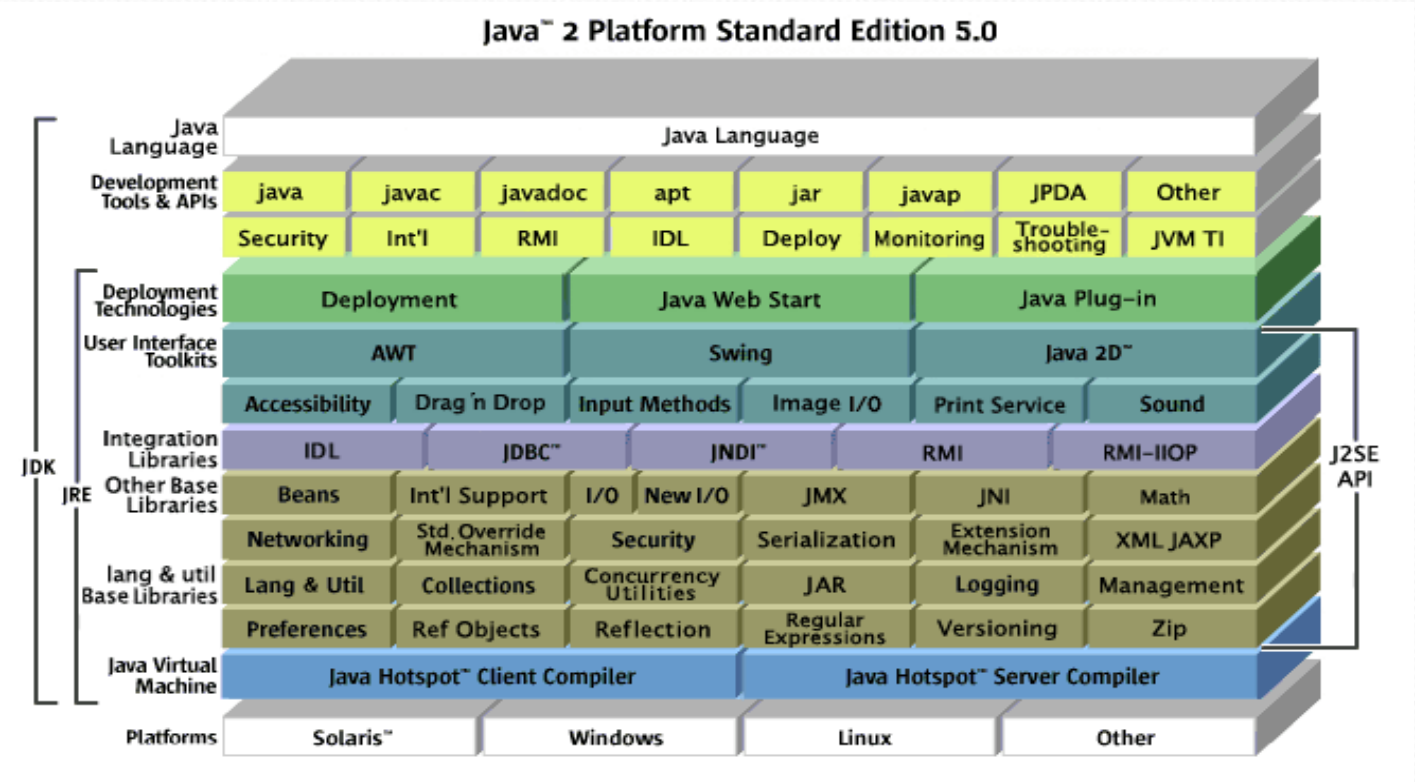
   Aplicaciones escritas en un **lenguaje interpretado** (o precompilado en un código intermedio) portable: el intérprete y los paquetes para su ejecución son estándares para varias plataformas.



Fuente figura:
http://parasitovirtual.wordpress.com/2010/09/23/introduccion-a-java/

Plataforma Java (1/3)

Ejemplo de plataforma a **mayor nivel de abstracción** (incluye lenguaje de programación) que el proporcionado por un sistema operativo.



Fuente figura: http://luizgustavoss.wordpress.com/2009/02/06/visao-geral-da-tecnologia-java/

Plataforma Java (2/3)

- **Requiere máquina virtual** JVM (*Java Virtual Machine*): el mismo código se puede ejecutar en todos los sistemas que implementen JVM.
- Los ejecutables Java **no se ejecutan nativamente** sobre el sistema operativo, es decir, ni Windows, Linux, Mac OS X, etc, ejecutan programas Java directamente. Sin embargo, Java Native Interface (JNI) permite el acceso a funciones específicas del sistema operativo.
- Para **aplicaciones Java móviles**: *Windows* y *Mac OS* utilizan *plugins* en los navegadores para su ejecución; y *Android* soporta Java directamente.

Fuente figura:
http://profejavaoramas.blogspot.com.es/2010/04/maquina-virtual-de-java-jvm.html

Plataforma Java (3/3)

- JVM ejecuta programas Java compilados a **lenguaje intermedio** (*bytecodes*) independiente de hardware y sistema operativo donde se ejecuta; los programas Java son **multiplataforma**, pero no la JVM (hay una para cada sistema operativo).

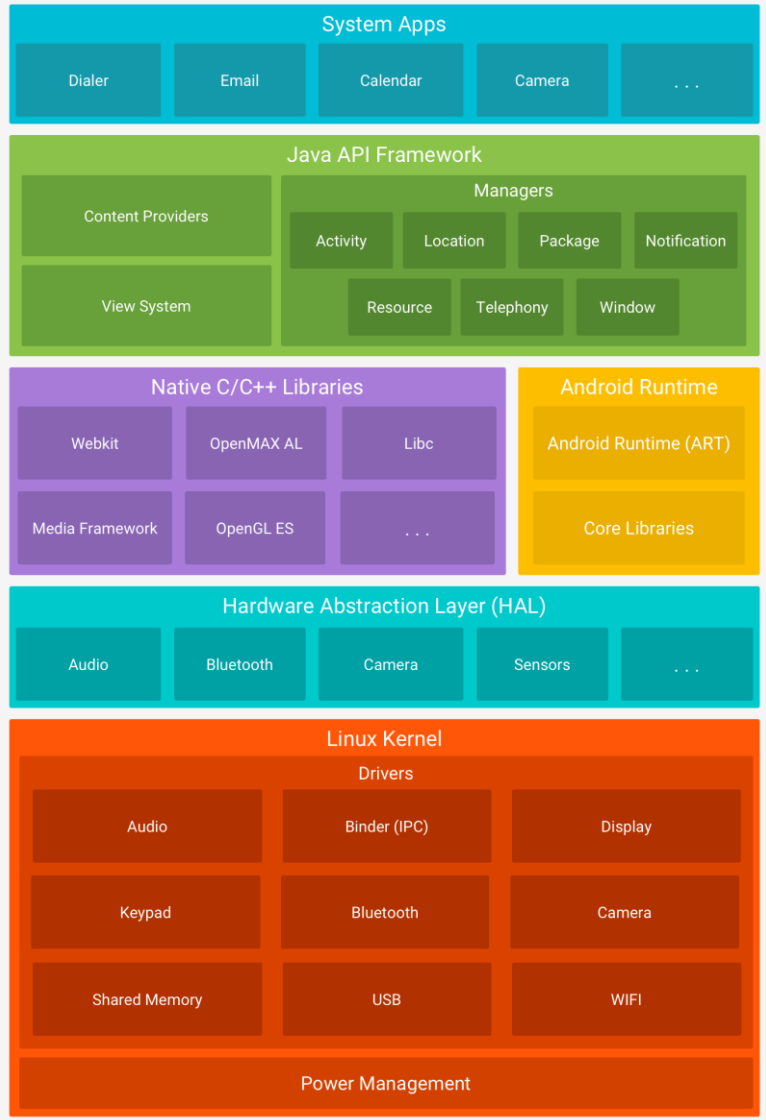
- Hay un compilador JIT (*Just In Time*) dentro de la JVM (desde la version 1.2) que traduce Java ***bytecodes* en instrucciones nativas** del procesador en tiempo de ejecución, las cuales se almacenan para su posterior reutilización.

- El uso del compilador JIT permite que, después de un breve retardo en la carga y prácticamente su total compilación, las aplicaciones Java se ejecuten **tan rápidamente como programas nativos**.

Fuente figura: http://profejavaoramas.blogspot.com.es/2010/04/maquina-virtual-de-java-jvm.html

Plataforma Android (1/3)

- La arquitectura de Android está formada por distintas capas, todas ellas basadas en software libre. Cada una de las capas utiliza elementos de la capa inferior para realizar sus funciones (pila de software):
  - **Kernel de Linux**. Es la base de la plataforma Android: generación de subprocesos, administración de memoria de bajo nivel, funciones de seguridad claves, etc.
  - **Capa de abstracción de hardware (HAL).** Módulos de biblioteca que implementan interfaces para un tipo específico de componente de hardware (ej. módulo de la cámara o de Bluetooth).

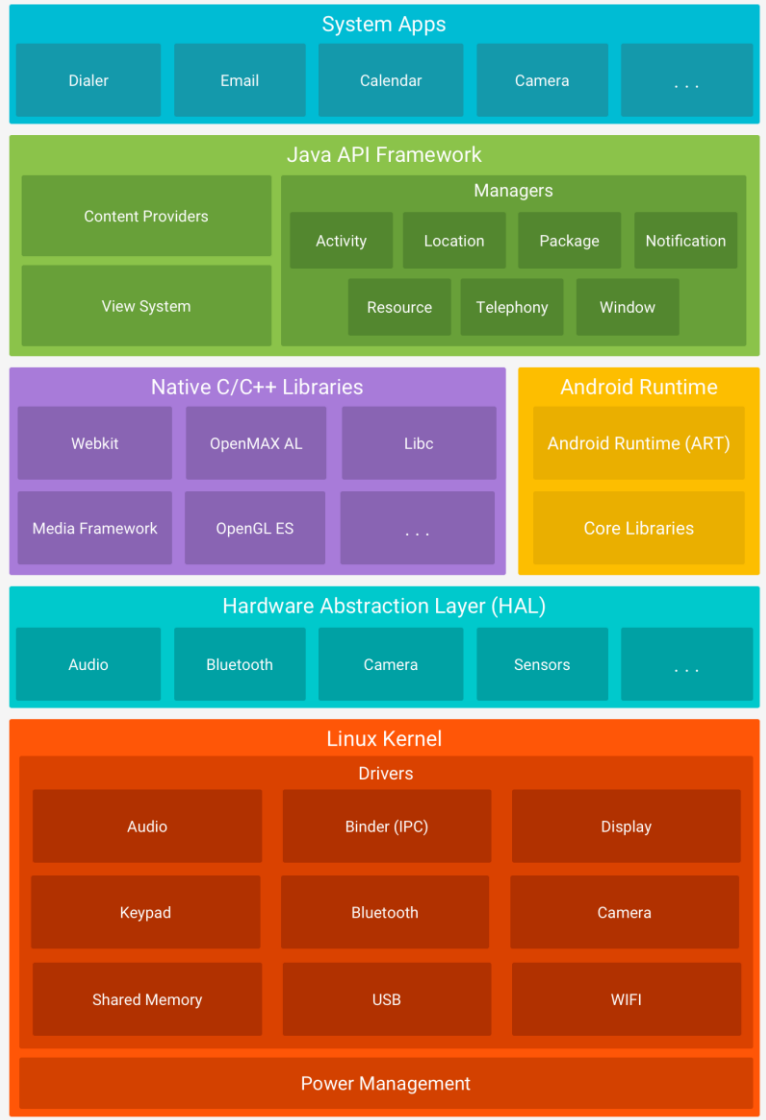


Fuente figura: https://developer.android.com/guide/platform

Plataforma Android (2/3)

- **Tiempo de ejecución de Android (ART).** El Ejecuta varias máquinas virtuales en dispositivos de memoria baja utilizando archivos DEX (formato de código de bytes diseñado especialmente para Android y optimizado para ocupar un espacio de memoria mínimo).

- **Bibliotecas C/C++ nativas.** Muchos componentes y servicios centrales del sistema Android, como el ART y la HAL, se basan en código nativo que requiere bibliotecas nativas escritas en C y C++.

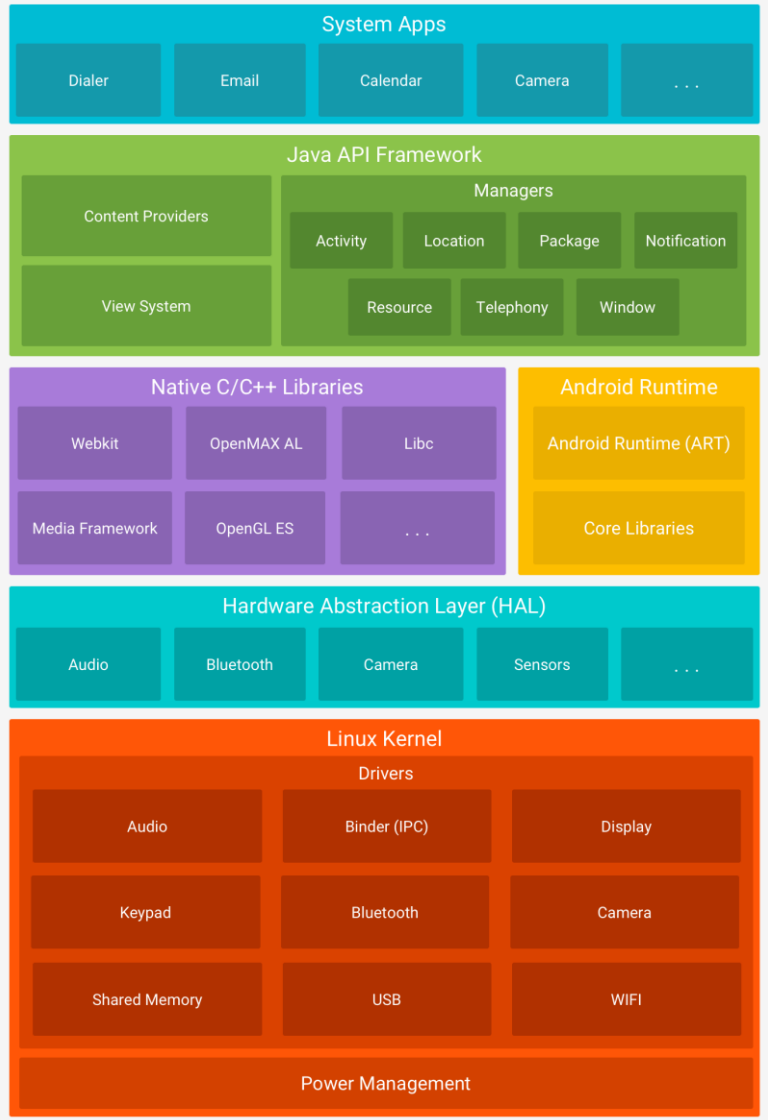


Fuente figura: https://developer.android.com/guide/platform

Plataforma Android (3/3)

- **Marco de trabajo de la API de Java.** El conjunto de funciones del SO Android está disponible mediante API escritas en lenguaje Java, necesarias para crear apps de Android simplificando la reutilización de componentes del sistema y servicios centrales y modulares.

- **Apps del sistema.** Se incluye un conjunto de apps centrales para correo electrónico, mensajería SMS, calendarios, navegación en Internet y contactos, entre otros elementos. Las apps del sistema funcionan como apps para los usuarios y brindan capacidades claves a las cuales los desarrolladores pueden acceder desde sus propias apps.



Fuente figura: https://developer.android.com/guide/platform

Herramientas básicas para el desarrollo software en Linux

Fases	Herramientas
Generador de código fuente	Editores de texto (**`pico`**, **`emacs`**, **`xemacs`**, ...)
Sangrado código fuente	**`indent`** sangra un programa en C sintácticamente correcto
Compilación código fuente	**`gcc`** y **`g++`** de GNU pre-procesa, compila, optimiza, y enlaza para generar archivos ejecutables
Gestión de software basado en módulos	**`make`** actualiza archivos en base a relaciones de dependencia previamente almacenadas
Gestión de bibliotecas	• **`ar`** permite crear y manipular archivadores en base a conjunto de archivos • **`ranlib`** genera y añade una tabla de índice de contenidos a archivadores acelerando la fase de enlazado • **`nm`** visualiza información de archivos objeto que ayuda a depurar bibliotecas
Control de versiones	CVS (Sistema Concurrente de Versiones) es una interfaz a RCS (Sistemas de Control de Revisiones), permite gestión de versiones en múltiples directorios y con múltiples desarrolladores

Integrated Development Environment (IDE)

- Entorno integrado de desarrollo (IDE): aplicación que proporciona un **conjunto de herramientas relacionadas** para el desarrollo del software:
  - creación y edición de código fuente;
  - generadores (compiladores, interpretes, enlazadores, gestores de bibliotecas, etc) de código objeto;
  - despliegue y depuración de programas.

- La integración de herramientas **contrasta** con el desarrollo utilizando las herramientas aisladas que incluye Linux (**`gcc`**, **`make`**, …)

- **IDEs ejemplo**: *Microsoft Visual Studio, Eclipse, SharpDevelop*, etc.

- IDEs actuales para el desarrollo de software **orientado a objetos (**ej: *Netbeans***)** incluyen otras herrramientas adicionales: navegadores para diagramas de jerarquía de clases, inspectores de objetos, etc.

Objetivos de un IDE

- **Máximizar la productividad** de los programadores con herramientas provistas de interfaces de usuario similares: todo el desarrollo se lleva a cabo bajo una misma aplicación.
- **Reducir tareas de configuración** de múltiples herramientas proporcionando un conjunto de facilidades de **forma cohesiva**.
- **Aprender rapidamente** a utilizar un IDE que integra manualmente todas las herramientas.
- **Acelerar el aprendizaje de lenguajes de programación**, por ejemplo, el código se puede analizar mientras se edita para conocer de forma inmediata errores léxicos, sintácticos, etc.

Tipos de IDE

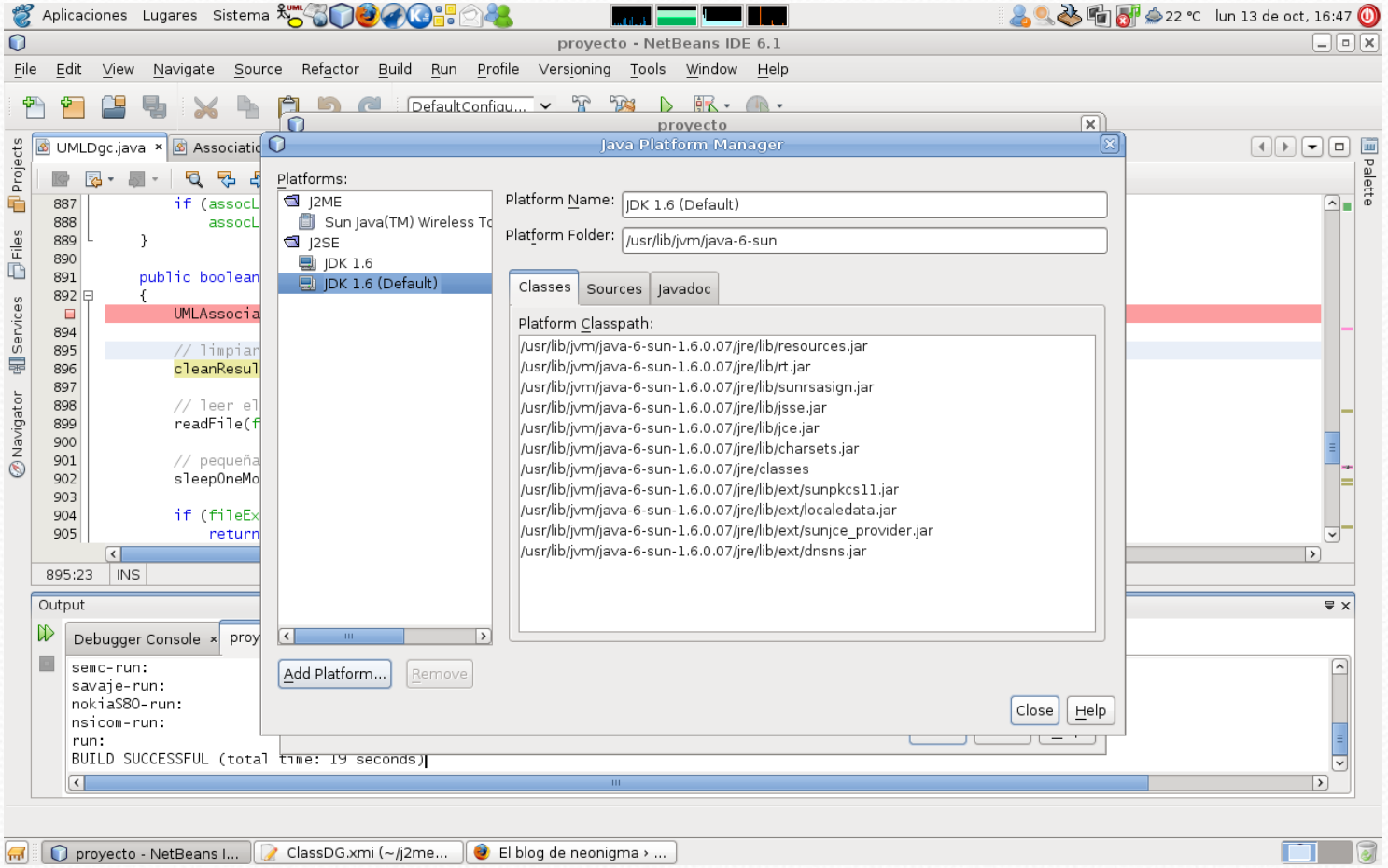
1. Dedicados a un **sólo lenguaje de programación** y con un conjunto de caracteríticas propias del paradigma de programación al cual pertenece (p. ej. herramienta para manejo de jerarquía de clases en orientación a objetos).

2. IDEs que soportan **múltiples lenguajes de programación**:

   a) Alternativamente mediante *plugins* (es posible instalar varios lenguajes al mismo tiempo): Los IDEs *Eclipse* y *Netbeans* soportan entre otros C/C++, Ada, Perl, Python, Ruby, y PHP; o

   a) Al mismo tiempo para un conjunto de lenguajes/plataformas relacionados: *Microsoft Visual Studio* y *Xcode* (OS X/iOS y lenguajes C/C++, Objective-C, Java, AppleScript, Python, …)

IDE ejemplo: *NetBeans* para *Java*



Fuente figura: http://www.nacho-alvarez.es/blog/2008/10/13/problema-con-sun-wireless-toolkit-bajo-netbeans-ventana-en-blanco/

IDEs y Programación Visual

- La **programación visual** hace uso de IDEs que permiten a los diseñadores/programadores crear nuevas aplicaciones **combinando bloques/nodos de código** mediante diagramas de estructura y de flujos, normalmente basados en el lenguaje UML (*Unified Modeling Language,* https://www.uml.org/)

- **Ejemplos**:

  1. *Lego Mindstorms System*: utilizando la potencia de navegadores como Mozilla.
  2. *KTechlab* :IDE abierto y simulador para desarrollar software para microcontroladores.
  3. *LabVIEW* y *EICASLAB*: especializados en programación distribuida.

Framework de desarrollo software

- **Framework de desarrollo software**: abstracción que proporciona software con funcionalidad genérica que puede cambiarse selectivamente mediante código de usuario para la creación de aplicaciones.

- Incluye herramientas similares a las encontradas en IDEs (compiladores, bibliotecas, etc.), así como una API (*Application Programming Interface*).

- Un framework tiene **caracteristicas clave para la reutilización software** que lo distinguen de otras alternativas tales como bibliotecas o bloques/nodos de código en IDEs.

Objetivos de un framework

- **Facilitar y reducir el tiempo el desarrollo** evitando dedicar tiempo a detalles de bajo nivel.
  - Por ejemplo, un desarrollador debe escribir la funcionalidad para la gestión de un sistema bancario en Web en lugar de los mecanismos para manejar peticiones y gestión de estado tales como crear hebras para atender una nueva petición cuando el resto de hebras ya sirven otras peticiones previas.

- Debido a la complejidad de las APIs, un framework conlleva un **tiempo extra de aprendizaje** inicialmente.

- Generalmente los frameworks se centran en **dominios específicos**: compiladores para diferentes lenguajes y plataformas, sistemas de soporte a la decisión, middleware, computación de altas prestaciones.

Características de un framework

- **Inversion de control:** a diferencia de las bibliotecas o aplicaciones normales, el código del framework invoca al código proporcionado por el usuario del framework.
- El framework tiene un **comportamiento útil por defecto**.
- **Extensibilidad**: un framework puede ser ampliado sobreescribiendo de forma selectiva o especializa su código con la funcionalidad específica proporcionada por el usuario.
- El código del framework no puede ser modificado, excepto por extensibilidad (como se ha comentado en el punto anterior).

Arquitectura de un framework

- Un framework se define mediante **componentes básicos**, y las relaciones entre ellos, que no pueden ser modificados (*frozen spots*).
- Existen partes predeterminadas donde el programador añade su código con la **funcionalidad específca** deseada (*hot spots*).
- **Ejemplo** de arquitectura de un framework orientado a objetos:
  - El framework consta de clases abstractas y concretas, y su instanción consiste en componer y especializar dichas clases.
  - Las clases definidas por el usuario reciben mensajes de las clases del frameword (inversión de control); los desarrolladores manejan ésto implementando los métodos abstractos de la superclase.

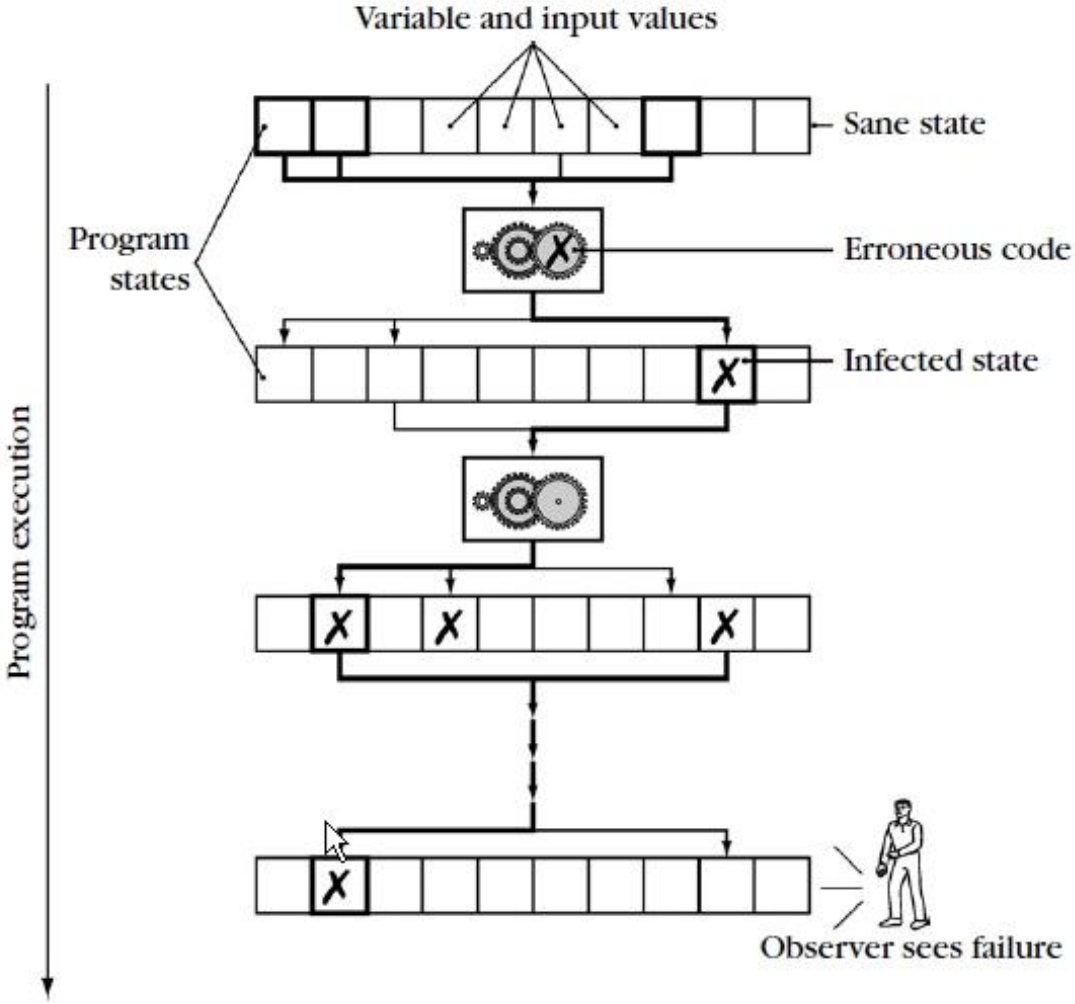
Definición y objetivos

- **Depuración**: es una actividad compleja consistente en encontrar y solucionar errores cometidos en el diseño y codificación de programas.
- **Objetivos** de las técnicas aplicadas a la depuración:
  - **Incrementar la productividad** encontrando y solucionando errores más rapida y efectivamente.
  - **Mejorar la calidad** de los programas eliminando defectos.
  - **Prevenir errores** como mejor solución.
  - **Mejorar lenguajes de programación y herramientas** identificando errores estáticamente y detectando el incumplimiento de invariantes dinámicamente, e.g., definición de sistemas de tipos en Java y C#.

Fases para la generación de fallos

- **Creación de un defecto**: Pieza de código creada por el programador que puede causar infección como consecuencia de un error, cambios en los requisitos originales, o interacciones impredecibles entre componentes (p. ej. programas distribuidos).
- **El defecto produce infección**. Los estados a alcanzar en la ejecución del programa difieren de los previstos por el programador.
- **Propagación de la infección**. Como un programa accede a su estado actual para su ejecución, una infección se puede propagar a otros estados.
- **La infección causa el fallo**: Error observable externamente en el comportamiento de un programa.

Ejecución de un programa como secuencia de estados (**Fuente fig.: [Zeller09]**)



Propiedades en la generación de fallos

- **Un estado de programa** viene definido por los valores de las variables y la posición de ejecución (contador de programa).
- Cada estado determina los siguientes estados hasta alcanzar el estado final.
- Las pruebas muestran la presencia de defectos pero nunca la ausencia:
  - Hay defectos que no provocan infecciones, y hay infecciones que no provocan fallos.
  - La ausencia de fallos no implica ausencia de defectos.
- **La cadena de infección** viene definida por la relación causa-efecto desde el defecto a fallo; la **depuración** consiste en identificar dicha cadena de infección para eliminar el defecto.

Fases de depuración

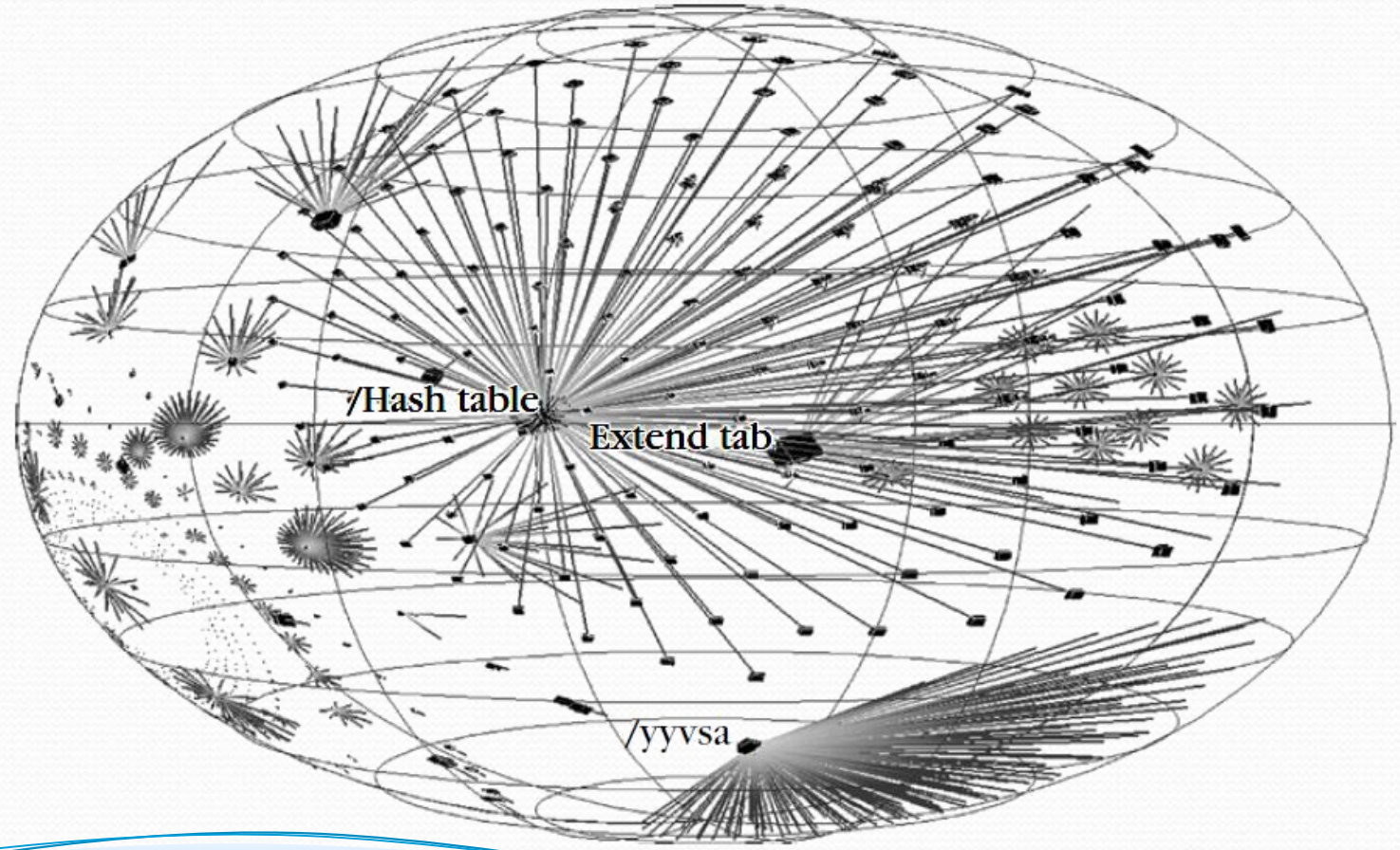
1. **Registrar el problema.**
2. **Reproducir el fallo (the failure)**: más complicado para programas no-deterministas y de larga ejecución.
3. **Automatizar y simplificar el caso de prueba.**
4. **Encontrar posibles origenes de la infección.**
5. **Centrar la búsqueda en los orígenes más probables.**
6. **Aislar la cadena de infección.**
7. **Corregir el defecto:** sencillo si está claro el defecto que produce el fallo.

Características del proceso de depuración

- Las fases 4-6 están directamente relacionadas con **comprender cómo se produce el fallo**.
- La depuración es un **problema de búsqueda** en dos dimensiones:
  1. **Espacio**: parte del estado (conjunto de variables) que está infectado.
  2. **Tiempo**: cuando tiene lugar la infección (estado).
- La búsqueda es de envergadura incluso para los programas sencillos:
  1. Los estados pueden comprender muchas variables.
  2. La ejecución puede constar de muchos estados.

Ejemplo de estado de ejecución:

- compilador GNU.
- 44.000 variables (vértices).
- aprox. 42.000 referencias entre variables (arcos) **(Fuente: [Zeller09])**



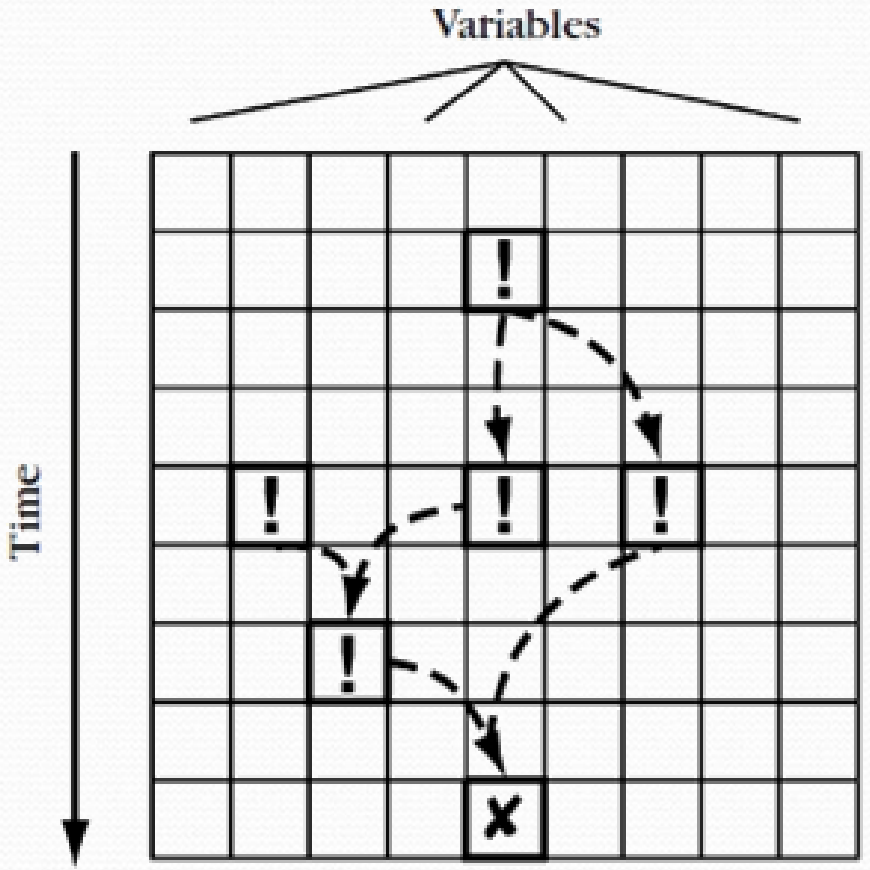
Búsqueda del defecto (Fuente: [Zeller09])

- Se aplican **dos principios básicos** para la búsqueda:

  1. separar estados sanos de infectados, y

  2. separar variables relevantes e irrelevantes.

- Un fallo puede ocurrir debido a ciertos valores de variables en estados anteriores (!), lo cual determina **dependencias que ayudan a localizar el defecto**.



Programación estructurada y depuración

La construcción de programas estructurados ayuda de forma importante a la depuración gracias a las siguientes características:

1. Contiene un conjunto de **funciones bien definidas**.
2. Utiliza **construcciones iterativas** (como *while* y *for*) en vez de *goto*.
3. Utiliza variables que tienen un propósito y a las cuales se les ha dado un nombre significativo.
4. Utiliza **tipos de datos estructurados** para representar datos complejos.
5. Utiliza **ADTs** (*Abstract Data Type*) o directamente el **paradigma de programación orientado a objetos**.

Depurador

Definición: Herramienta software que ayuda a **ejecución controlada de un programa** para ayudar a encontrar el defectos que producen un fallos.

Características:

1. Interactivo. Aunque se puede usar en modo *batch*, su rol principal es interactuar con el programador.
2. Proporciona un **conjunto de instrucciones** que indican las acciones de depuración a realizar.
3. Permite detectar **errores en tres categorías**:
   a. **Sintácticos**: detectados durante la fase de compilación o la de enlazado.
   b. **Ejecución**: p. ej., acceder fuera del espacio de direcciones asignadas (*segmentation fault*) o realizar una operación de división por cero, ambas produciendo la terminación del programa.
   c. **Lógicos**: p. ej., bucles que iteran más o menos veces de las previstas.

Tipos de errores comunes

Los tipos de errores más comunes que se producen cuando se programa son:

1. **Escribir código de una forma desestructurada**. Cuando un programa se estructura adecuadamente, se divide el problema en sub-problemas sobre los cuales se puede trabajar independientemente (razonar sobre su solución y probarla) y después componer las distintas partes. Esto facilita mucho la depuración posterior.

2. **Programar sin pensar**: Antes de comenzar a codificar hay que diseñar la solución al problema planteado, pensando las distintas alternativas de solución posibles y entendiendo el algoritmo que se ha de seguir.