

Tema 5. Generación y Depuración de Aplicaciones

Contenidos

- 5.1 Concepto de plataforma. Software independiente de plataforma.**
- 5.2 *Frameworks* 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, entornos, y *frameworks* de desarrollo y depuración de aplicaciones.

Bibliografía básica

- <http://en.wikipedia.org/wiki/Cross-platform>, consultado Diciembre 2012
- http://en.wikipedia.org/wiki/Software_framework, consultado Diciembre 2012
- <http://en.wikipedia.org/wiki/Debugging>, consultado Diciembre 2012
- [Zeller09] A. Zeller: "*Why Programs Fail*" (2nd Edition), Morgan Kaufmann Publishers (Elsevier), 2009

24-sep-2014

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. Ejemplos de plataforma:

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 <i>Java Virtual Machine</i> , y por tanto múltiples arquitecturas

Clasificación del software

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

1. **Dependiente de una plataforma particular** para la cual se desarrolla y ejecuta, bien sea esta hardware, sistema operativo o máquina virtual.
2. **Multiplataforma** cuando el software es implementado e interopera en varias plataformas.

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

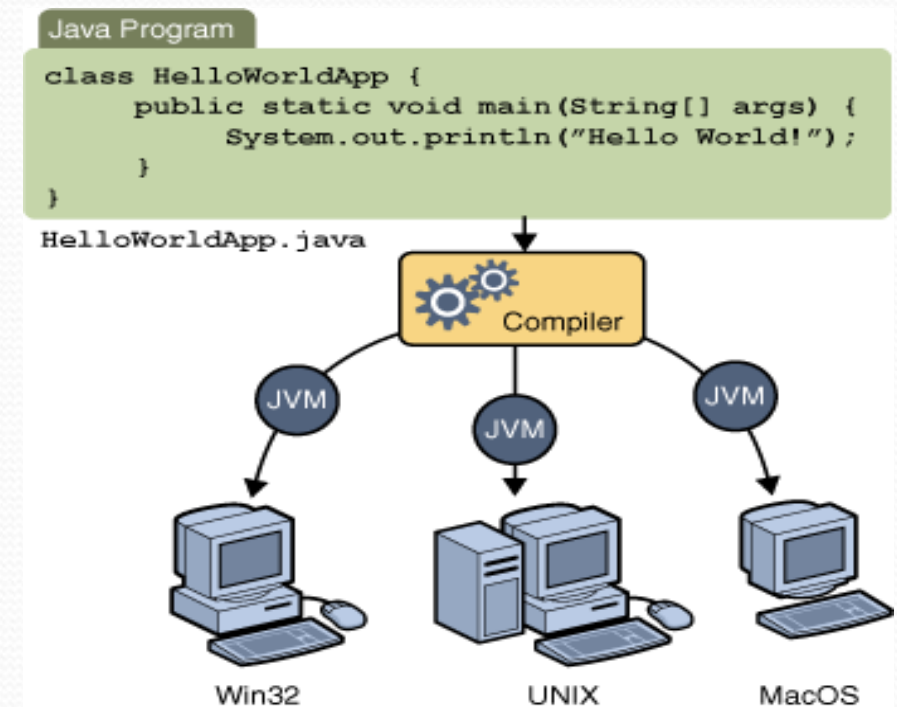
1. tantas plataformas como existan (caso ideal de **software independiente de la plataforma**), o
2. tan sólo en dos plataformas diferentes, por ejemplo, una aplicación multiplataforma se podría ejecutar en *Microsoft Windows* y *Linux* en arquitecturas x86.

Software multiplataforma

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

1. Aplicaciones que requieren su **creación o compilación para cada plataforma** específica donde se ejecutará.
2. Aquellas otras **aplicaciones que directamente se pueden ejecutar** en más de una plataforma sin preparación especial (e.g. plataforma Java):

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

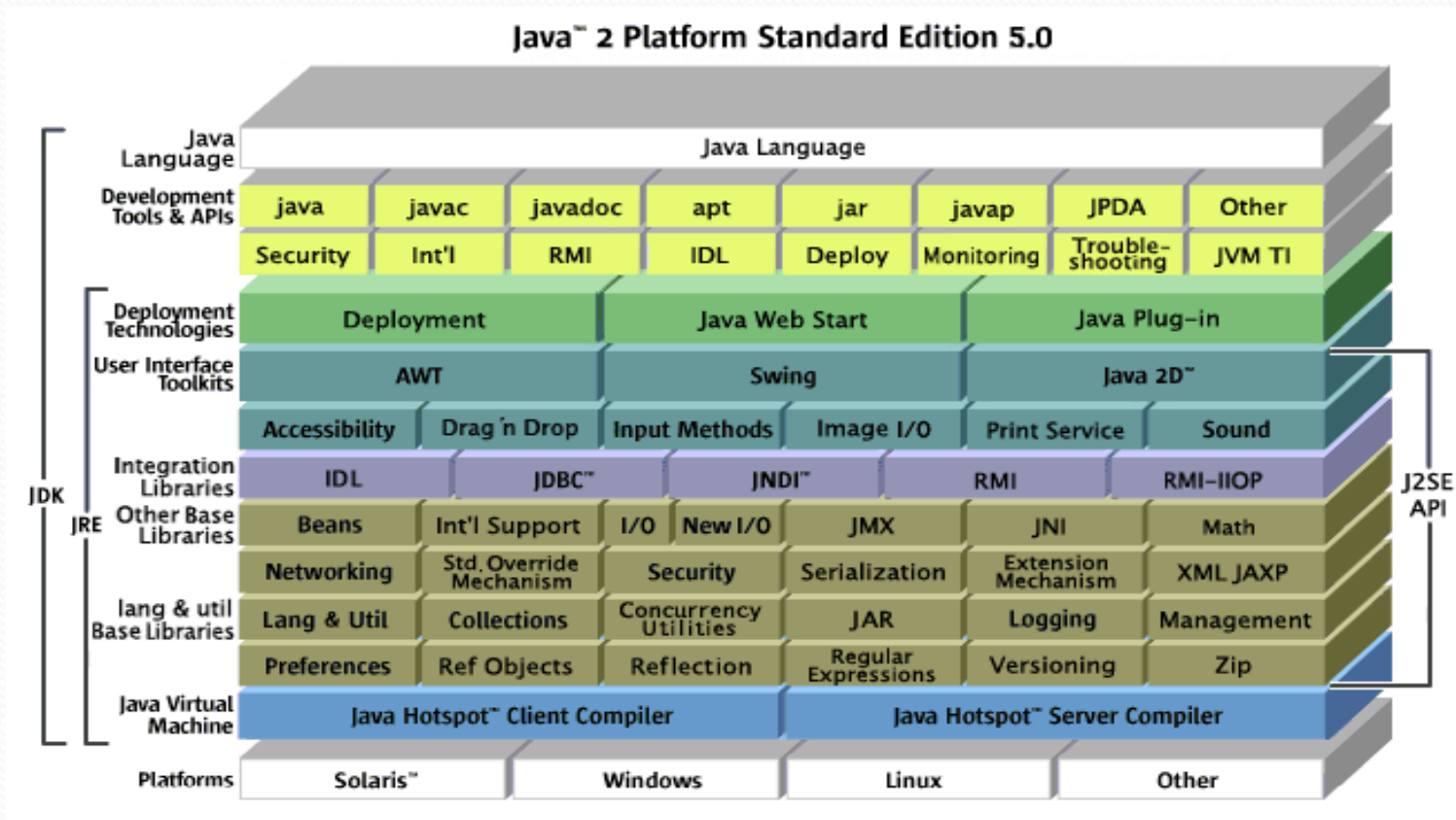


Fuente figura:

<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*), lo cual posibilita que el mismo código se pueda ejecutar en todos los sistemas que implementen JVM.
- Los ejecutables Java **no se ejecutan nativamente** sobre el sistema operativo, i.e., 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 JVM (version 1.2 en adelante) que traduce Java **bytecodes en instrucciones nativas** del procesador en tiempo de ejecución, las cuales son almacenadas 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>

Herramientas básicas para el desarrollo software en Linux

Fases	Herramientas
Generador de código fuente	Editores de texto (<code>pico</code> , <code>emacs</code> , <code>xemacs</code> , ...).
Sangrado código fuente	<code>indent</code> sangra un programa en C sintácticamente correcto
Compilación código fuente	<code>gcc</code> y <code>g++</code> de GNU pre-procesa, compila, optimiza, y enlaza para generar archivos ejecutables.
Gestión de software basado en módulos	<code>make</code> actualiza archivos en base a relaciones de dependencia previamente almacenadas
Gestión de bibliotecas	<ul style="list-style-type: none">• <code>ar</code> permite crear y manipular archivadores en base a conjunto de archivos.• <code>ranlib</code> genera y añade una tabla de índice de contenidos a archivadores acelerando la fase de enlazado.• <code>nm</code> 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, y 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 herramientas adicionales: navegadores para diagramas de jerarquía de clases, inspectores de objetos, etc.

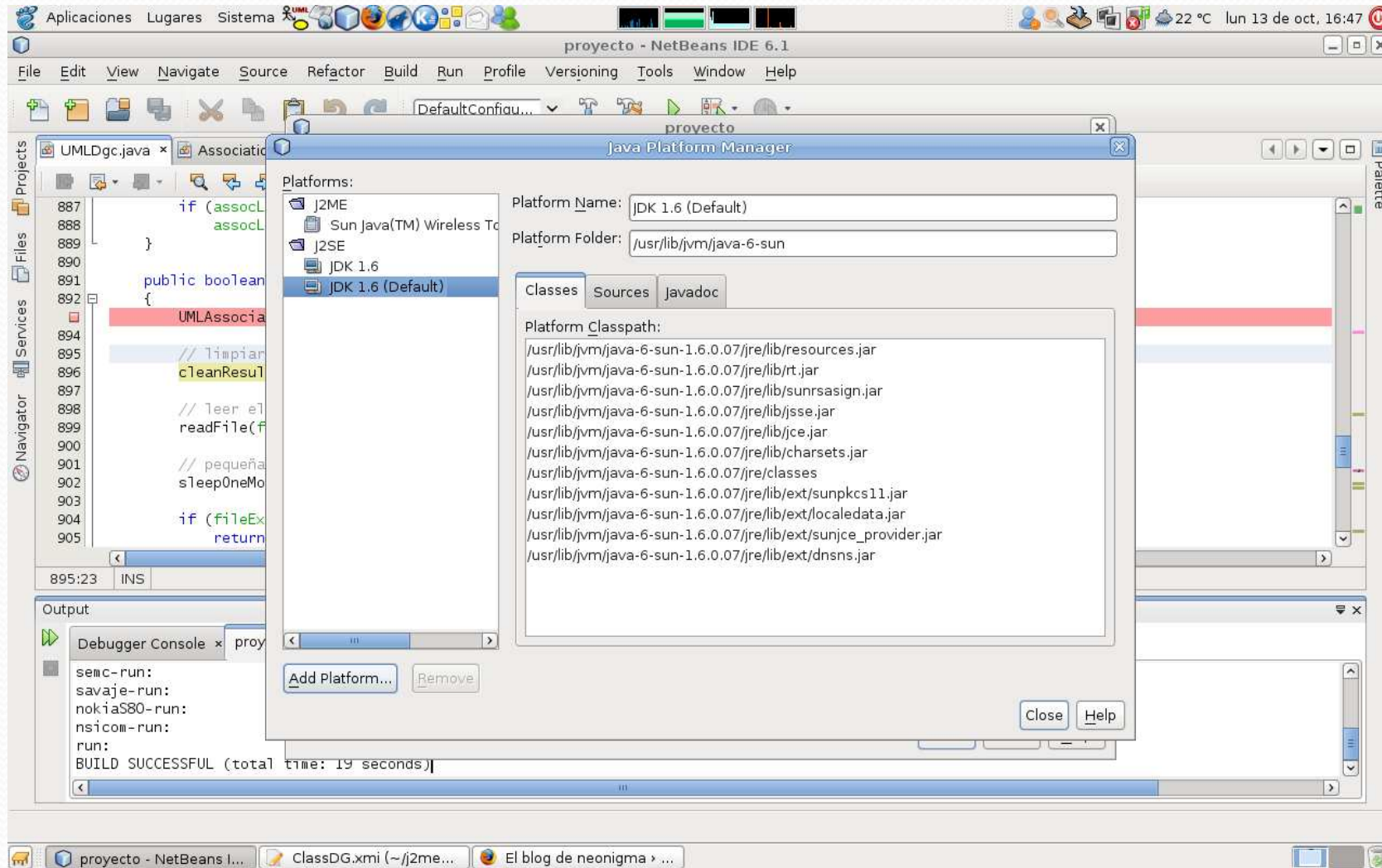
Objetivos de un IDE

- **Máximiza 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 rápidamente** a utilizar un IDE que integra manualmente todas las herramientas.
- **Acelerar el aprendizaje de lenguajes de programación**, e.g., el código se puede analizar mientras se edita proporcionando información inmediata acerca de errores léxicos, sintácticos, etc.

Tipos de IDE

1. Dedicados a un **sólo lenguaje de programación** y con un conjunto de características propias del paradigma de programación al cual pertenece (e.g. 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*^o



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 UML (*Unified Modeling Language*).
- **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, ...), así como una API (*Application Programming Interface*).
- Un framework tiene **características 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 sobrescribiendo 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 específica** deseada (*hot spots*).
- **Ejemplo** de arquitectura de un framework orientado a objetos:
 - El framework consta de clases abstractas y concretas, y su instancia consiste en componer y especializar dichas clases.
 - Las clases definidas por el usuario reciben mensajes de las clases del framework (inversión de control); los desarrolladores manejan esto implementando los métodos abstractos de la superclase.

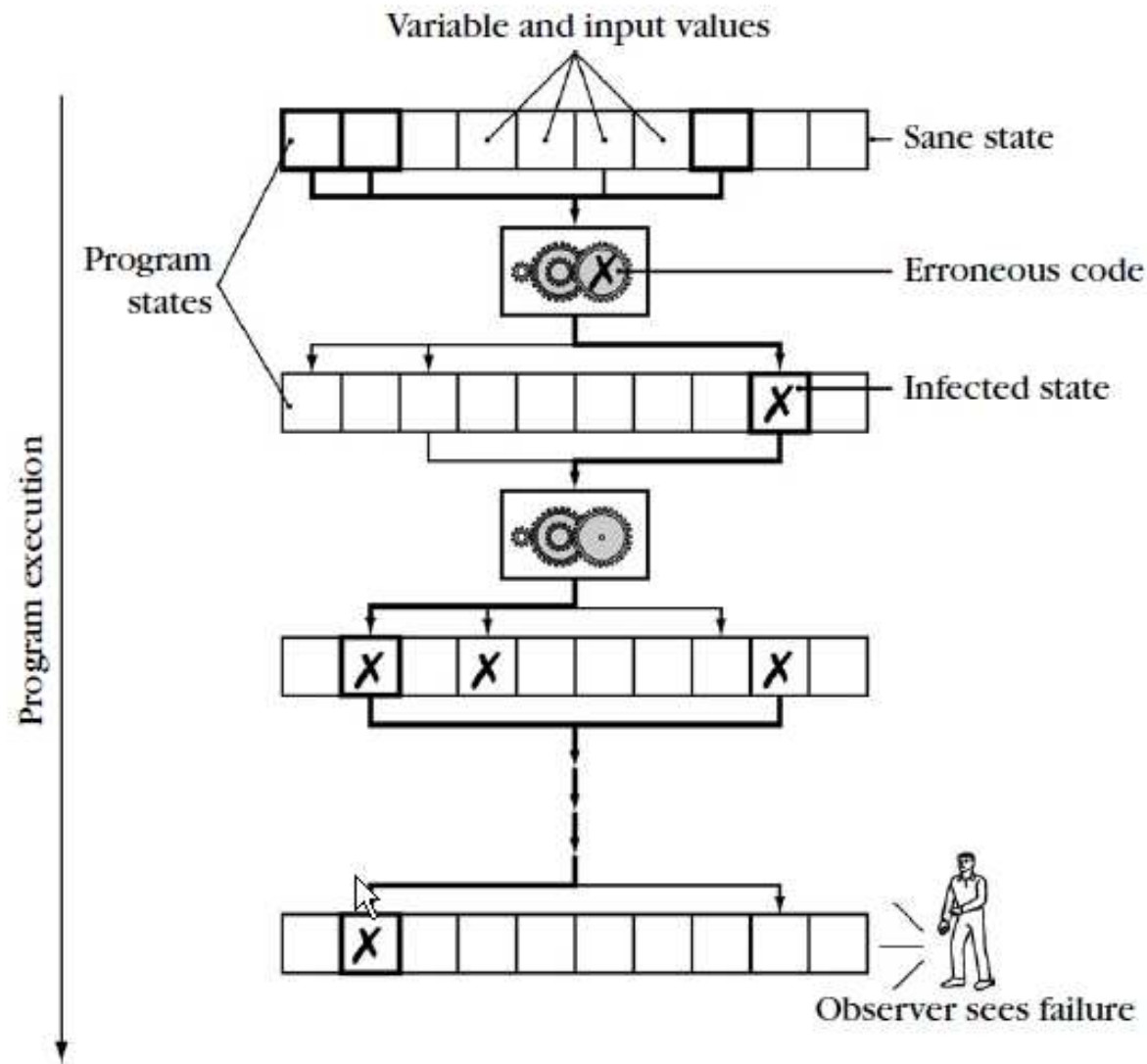
Definición y objetivos

- **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 rápida 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 (e.g. 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

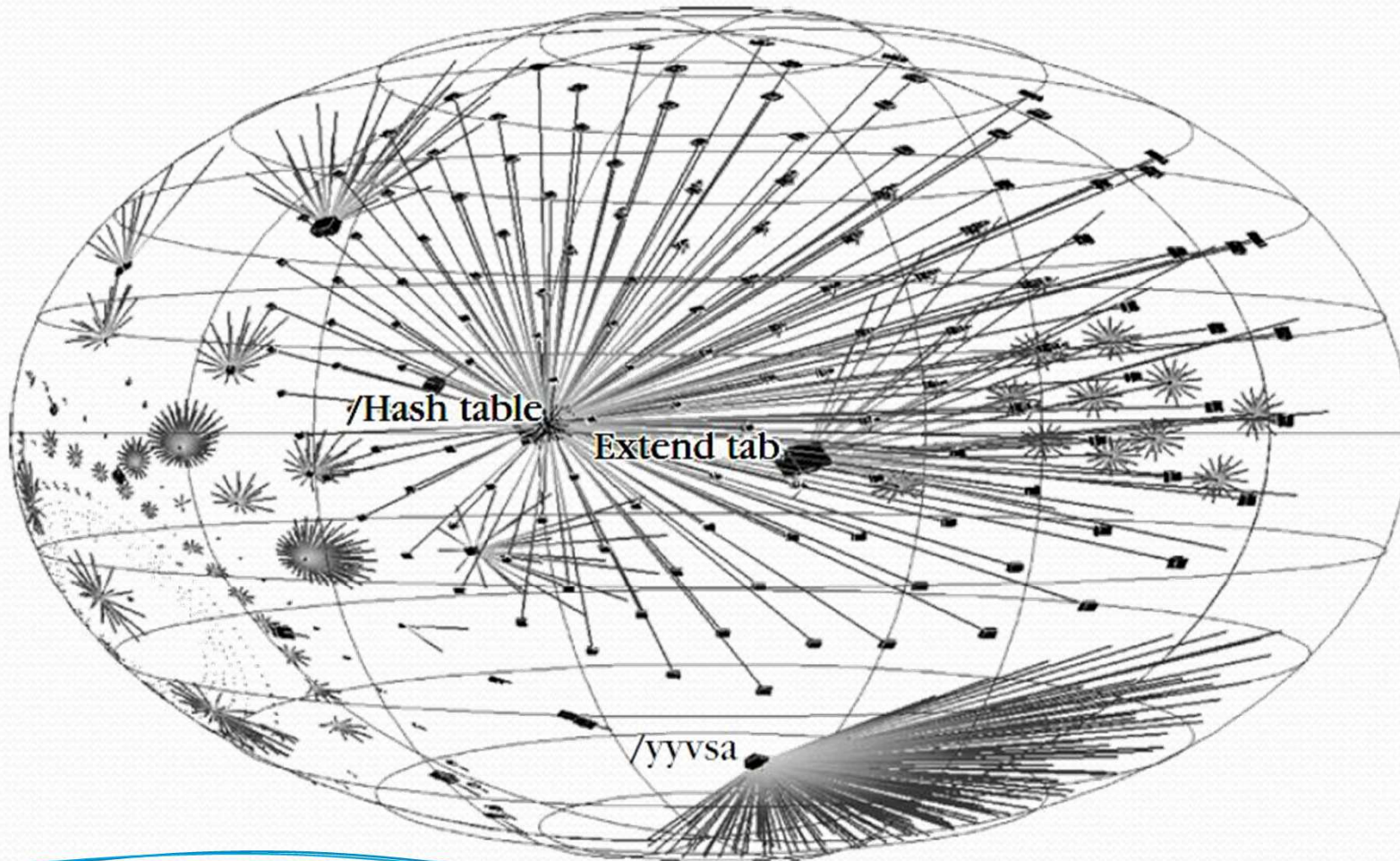
1. **Registrar el problema.**
2. **Reproducir el fallo (the failure):** más complicado para programas no-deterministas y de larga ejecución.
3. **Automatizar y simplificar el caso de prueba.**
4. **Encontrar posibles orígenes de la infección.**
5. **Centrar la búsqueda en los orígenes más probables.**
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.

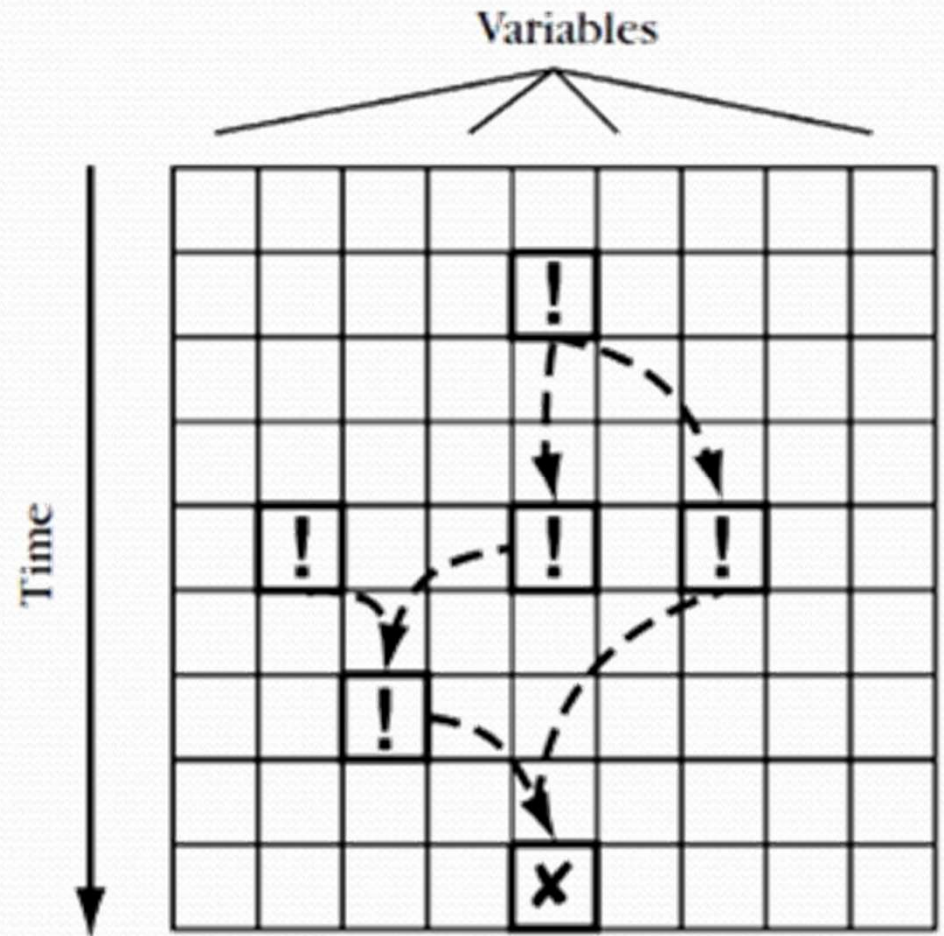
Ejemplo de estado de ejecución:

- 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 la **ejecución controlada de un programa** para ayudar a encontrar defectos que producen fallos.

Características:

1. Interactivo. Aunque se puede usar en modo *bach*, 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:** e.g., 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:** e.g., 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 subproblemas 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.