Tema 5. Generación y depuración de aplicaciones

1. Plataforma

1.1 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 SO. Ejemplos:

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

Notas:

Un programa normal va a depender de una plataforma (hardware (máquina) y software (SO)). Aunque algunos programas no necesitan SO (tienen un chip que hace la función de SO).

Normalmente, las plataformas nos ofrecen una máquina virtual.

1.2 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, SO o máquina virtual.
2. **Multiplataforma** cuando el software es implementado e interopera en varias plataformas.

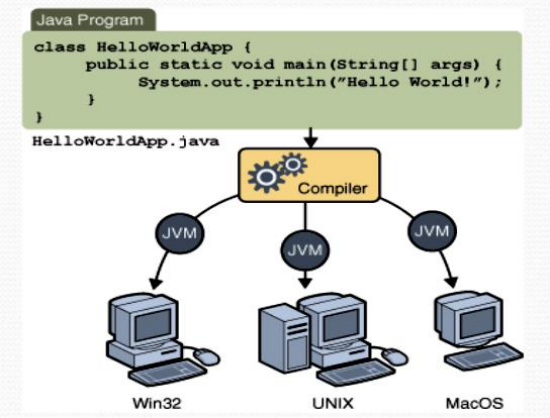
Una **aplicación multiplataforma** se puede ejecutar en:

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

1.3 Software multiplataforma

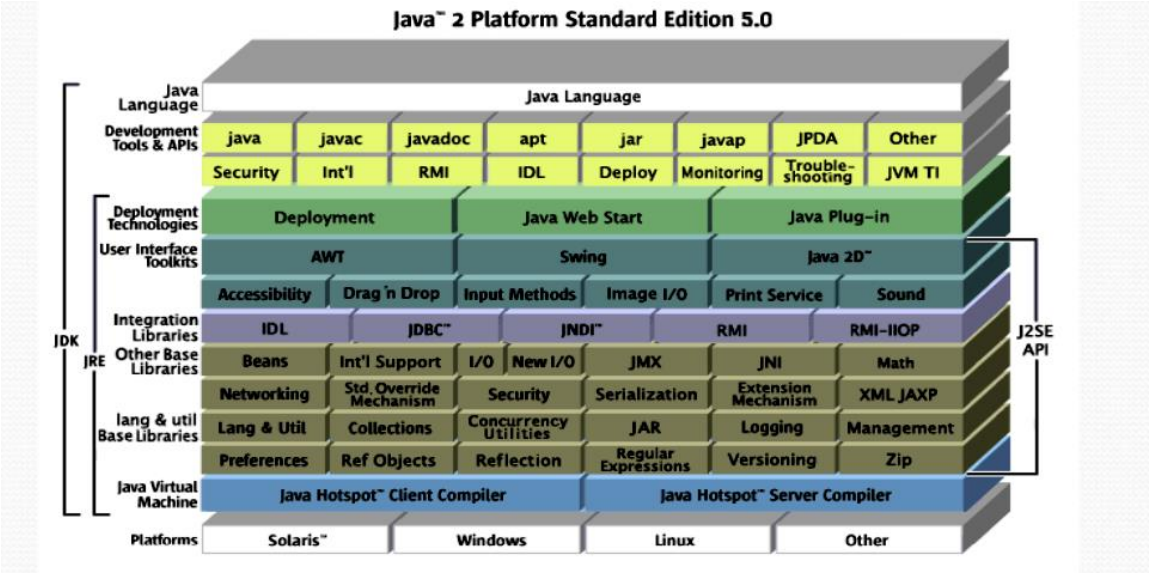
El software multiplataforma puede dividirse en **dos tipos**:

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.



1.4 Plataforma Java

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



- **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.
- 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.**

2. Framework de desarrollo de aplicaciones

2.1 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.

2.2 Integrated Development Enviroment (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, intérpretes, 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...*
- 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.

2.3 Objetivos de un IDE

- **Maximizar 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.

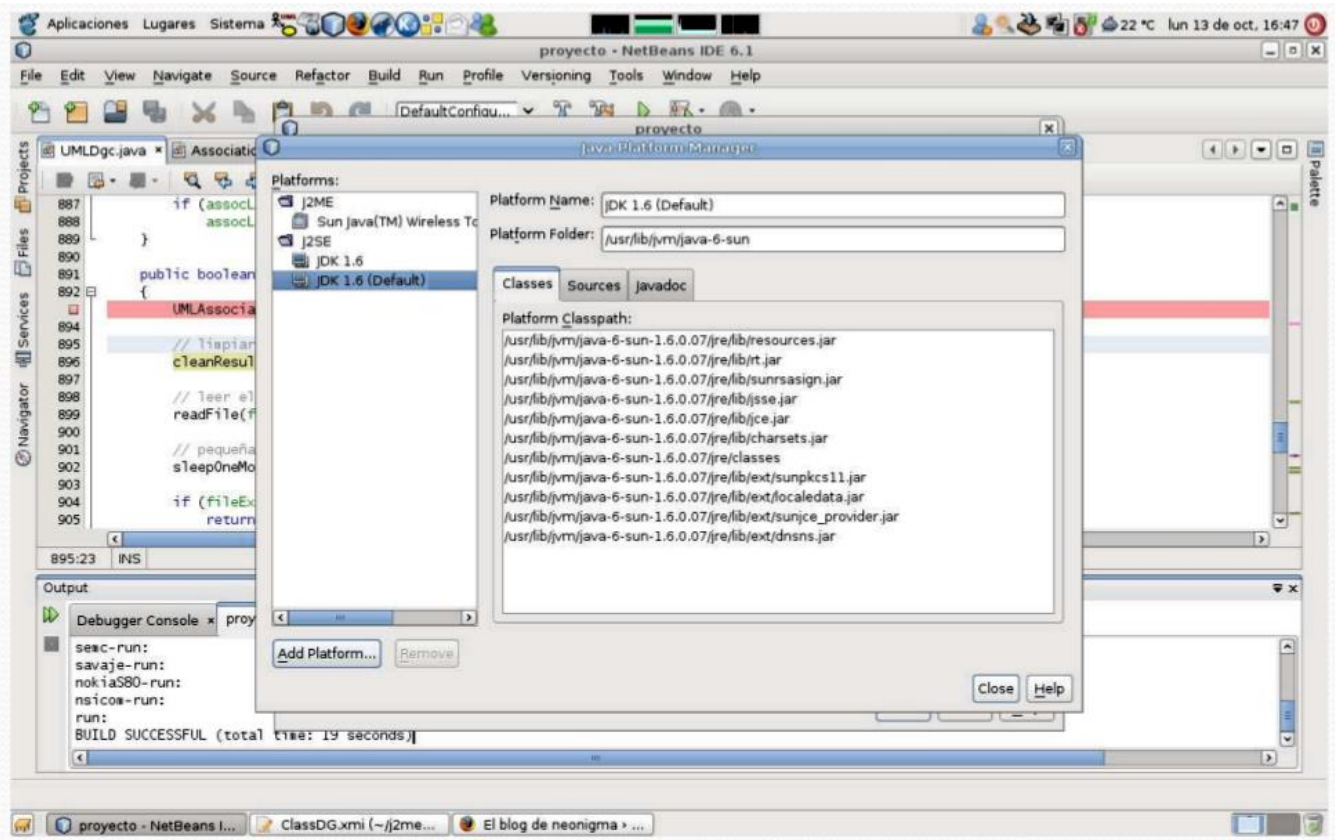
2.4 Tipos de IDE

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

   b) 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...)

2.5 IDE ejemplo: NetBeans para Java



2.6 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.

2.7 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.

2.8 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.

2.9 Características de un Framework

- **Inversión 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).

2.10 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 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.

3. Técnicas de depuración de programas

Apuntes de clase:

1. Hago el programa.
2. Quiero ver si cumple la especificación o diseño (lo que tiene que hacer), puedo hacerlo de dos formas:

2.1 Verificación formal de programas:

- Tengo una serie de sentencias escritas en un lenguaje y una especificación (es un conjunto de requisitos).
- Mi programa es un conjunto de órdenes.
- En la especificación de la semántica del lenguaje de programación se hace una **especificación axiomática.**
- ***Axiomática***: la especificación de la semántica del lenguaje de programación es un sistema deductivo (conjunto de axiomas y conjunto de diferencias).
- Hay dos **tipos de sentencias**:
  - Semánticamente simples: leer una variable y escribir una variable. Se especifican mediante axiomas.
  - Semánticamente compuestas: se especifican mediante reglas de diferencias.
- A partir de la especificación semántica se construye un **algoritmo** (Lógica de Hoare). Se construye esa lógica (que es el lenguaje de la lógica), meto el sistema deductivo, admite el lenguaje de programación y el de especificación como lenguajes correctos.
- Una vez construida esa lógica, mi programa se convierte en una **teoría** (conjunto de fórmulas bien formadas).
- Por lo que la verificación formal consiste, con este **sistema deductivo**, en probar que cada uno de los requisitos de mi especificación se cumplen. Por lo que hay que probar que los requisitos se cumplen en esta teoría, con el sistema deductivo.
- Esta comprobación lo hace un **humano**.
- Se hace en programas muy delicados para asegurarme de que están bien.
- Lo que estoy verificando es que la literatura del programa satisface las especificaciones.
- Pero no verifico que si cuando se ejecute se va a cumplir.
- Por lo que hacemos otra cosa: probar la ejecución de programas.

2.2 Forma general:

- Se diseña un conjunto de datos para ejecutar el programa de los cuales **sabemos los resultados**.
- Se diseña para que haga todos los **posibles recorridos** que tiene que hacer el programa.
- Por lo que así no tengo ningún motivo para pensar que no es correcto, pero no puedo pensar que es correcto totalmente.
- Suponemos que el programa **no funciona**: hay que corregir el programa. En programas complejos es más habitual que el fallo se detecte donde no se ha producido el fallo. Por lo que hay que ir haciendo trazas en la ejecución, para ver dónde falla.
- Cuando uno corrige el fallo, hay que tener cuidado con la **propagación de las correcciones**, porque yo he podido corregir algo, pero estoy desajustando otras cosas.

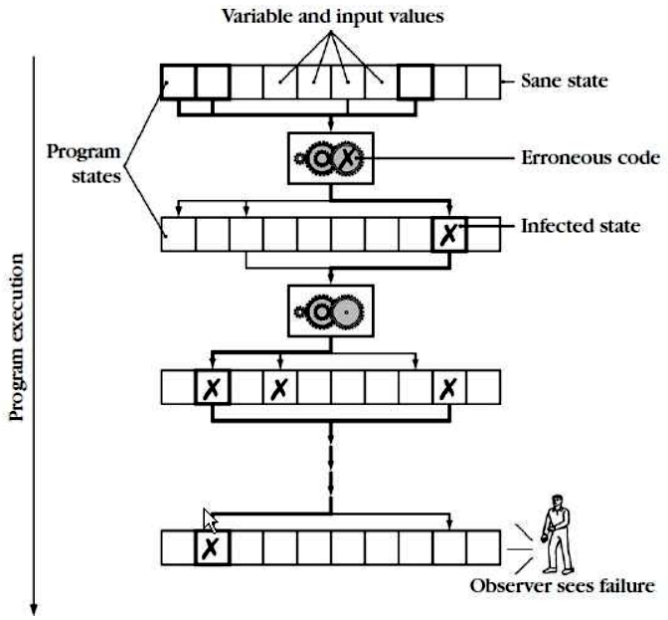
3.1 Definición y Objetivos

- **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#.

3.2 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.

3.3 Ejecución de un programa como secuencia de estados



3.4 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.

3.5 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.

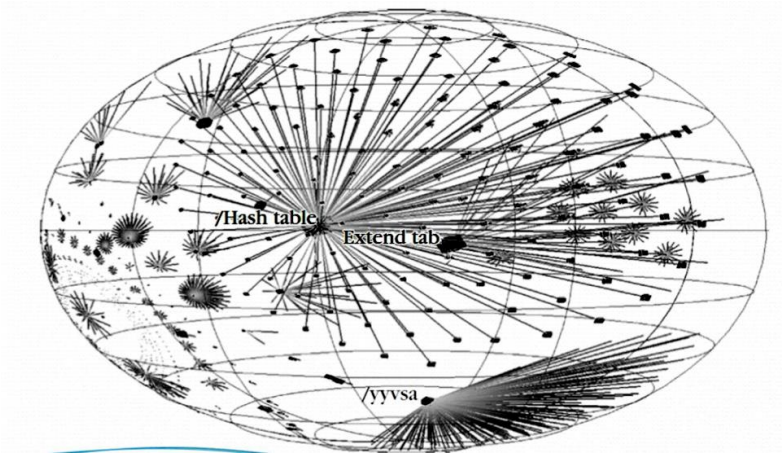
3.6 Características del proceso de depuración

- 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.

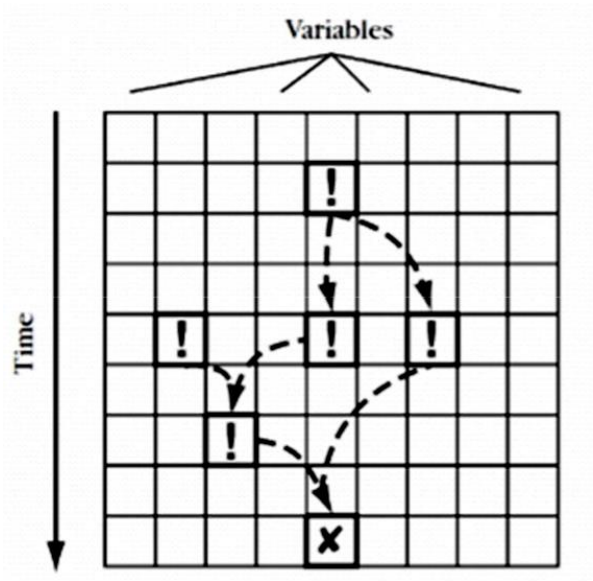
3.7 Ejemplo de estado de ejecución:

- Compilador GNU.
- 44.000 variables (vértices).
- Aproximadamente 42.000 referencias entre variables (arcos).



3.8 Búsqueda del defecto

- 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.**



3.9 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.**

3.10 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:**

   3.1 Sintácticos: detectados durante la fase de compilación o la de enlazado.

   3.2 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.

   3.2 Lógicos: e.g., bucles que iteran más o menos veces de las previstas.

3.11 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.