Análisis y Diseño de Sistemas 2

Administración de la configuración

Material de apoyo del curso Análisis y Diseño de Sistemas 2 de la USAC

Ing. Ricardo Morales

Primer semestre 2016

Análisis y Diseño de Sistemas 2

Administración de la configuración

Tabla de contenido

ADMINISTRACIÓN DE LA CONFIGURACIÓN

Objetivos

- Describir el concepto de administración de la configuración
- Conocer y entender los elementos que integran un sistema de control de versiones
- Identificar el valor de las prácticas recomendadas para implementar la administración de la configuración

Administración de la configuración

La administración de la configuración se refiere al proceso por el cual todos los artefactos relevantes a un proyecto y las relaciones entre ellos, son almacenadas, recuperadas e identificadas de manera única, y además son modificadas en el tiempo

Si se tiene una buena estrategia de administración de la configuración, se debería poder responder "si" a:

- ¿Puedo reproducir exactamente cualquiera de mis ambientes, incluyendo versiones del sistema operativo, niveles de parches, configuración de red, stack de software, aplicaciones y su configuración?
- ¿Puedo hacer fácilmente un cambio incremental a cualquiera de estos elementos y desplegarlo a cualquiera de los ambientes?
- ¿Puedo ver fácilmente que cambios han ocurrido a un ambiente y rastrearlo para ver exactamente que fue el cambio, quien lo hizo y cuando?
- ¿Puedo satisfacer todas las regulaciones de cumplimiento a las que estoy sujeto?
- ¿Es fácil para cada miembro del equipo obtener la información que necesita y hacer los cambios que necesita hacer?

Sistema de control de versiones

Un sistema de control de versiones, conocido también como control de fuentes, administración de código fuente o control de revisiones, es un mecanismo para mantener múltiples versiones de archivos, de manera que cuando se modifique un archivo aún se pueda accesar las revisiones previas.

El control de versiones permite:

- Retener y proveer acceso a cada versión de cada archivo que ha sido almacenado en el
- Colaborar entre equipos distribuidos en espacio y tiempo

Elementos del control de versiones

Repositorio

Un sistema de control de versiones es un sistema centralizado para compartir información

Su componente central es el repositorio, que es un almacenamiento centralizado o distribuido de datos. El repositorio almacena información en forma de un árbol de filesystem, una jerarquía de directorios y archivos.

Cualquier número de clientes se puede conectar al repositorio y crear o escribir archivos en el.

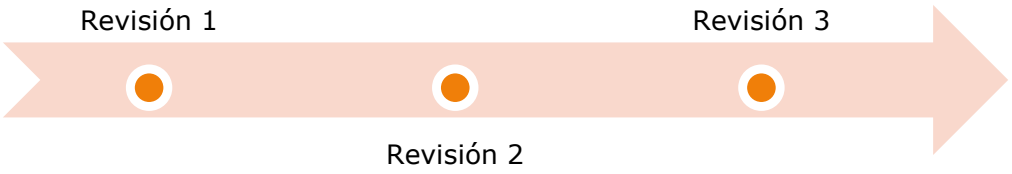
Área de trabajo

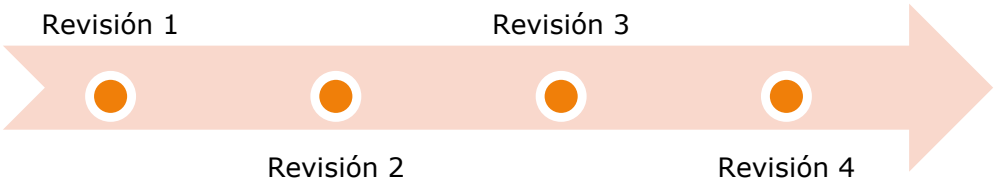
Es un lugar privado donde el integrante del equipo tiene los artefactos (archivos) que usa para llevar a cabo sus tareas.

Es una copia local (en el equipo del integrante del equipo) del repositorio o una parte de el, que esta sincronizada con el repositorio.

Revisión

Cada vez que se cambia un archivo u otro artefacto en el sistema de control de versiones, se crea una *revisión*.





Logs

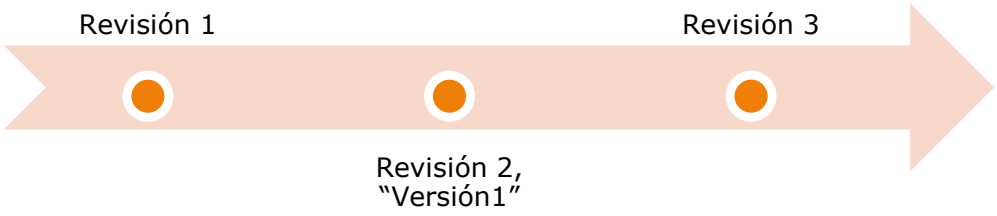
Cualquier sistema de control de versiones mantiene logs de los cambios realizados a los archivos en el repositorio.

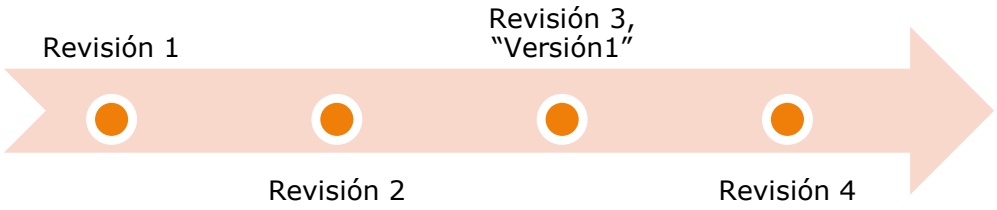
Al menos se tendrá el nombre del archivo, fecha de modificación, usuario de modificación y razón del cambio

La razón del cambio es importante, para conocer porque ha evolucionado el archivo de determinada manera.

Etiqueta (label)

En cualquier punto del tiempo, un snapshot del codeline (foto de una parte del repositorio en un momento del tiempo) contendrá varias revisiones de cada componente en el codeline.





El ejemplo muestra que en algún punto del tiempo se tiene la "Versión1", integrada por la revisión 2 del archivo 1 y la versión 3 del archivo 2. La siguiente ocasión que ocurre un cambio en el codeline, se tendrá la revisión 4 de archivo1 y la revisión 5 de archivo2

Cualquier snapshot ("foto") del codeline, que contienen una colección de revisiones de cada componente dicho codeline, es una *configuración* del codeline.

A cualquier configuración que se le da un nombre o número diferente es una versión del codeline, por ejemplo "versión1"

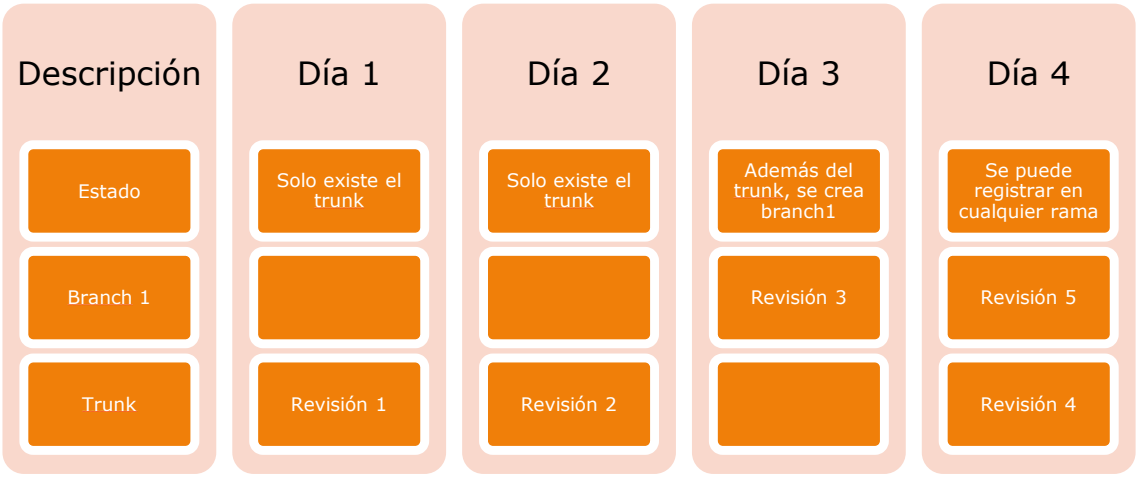
Si se elige identificar o marcar una versión como especial, se define una *etiqueta* (label). Se puede etiquetar un conjunto de revisiones que van en un release o que representan algo significativo para el equipo de trabajo

Rama (branch)

Conforme evoluciona un codeline, se puede descubrir que algún trabajo es derivativo de la intención del codeline. En ese caso, se podría hacer una rama (branch) tomando la revisión del archivo para que puede evolucionar de forma independiente.

Una *rama* de un archivo es una revisión del archivo que usa la versión del trunk (o rama principal) como punto de inicio y evoluciona de forma independiente.

Un ejemplo de uso de una rama es iniciar el trabajo de un nuevo release de un producto. Algunos cambios que se hacen en la rama también podrían ser necesarios en el trunk, así que se hace un *merge* para integrarlos de la rama al trunk.



Codeline

Un *codeline* es una progresión del conjunto de archivos fuente y otros artefactos que son parte de un componente de software, conforme cambia en el tiempo. Por ejemplo se puede tener el codeline versión 1 del producto procesador de palabras.

Un codeline contiene todas las versiones de todos los artefactos a través de un camino evolutivo.

En el caso mas simple, se puede tener un solo codeline que incluye todo el código del producto.

Los componentes de un codeline evolucionan a su propio ritmo y tienen revisiones que podemos identificar. Se puede identificar una versión del codeline a través de una etiqueta.

Cada codeline puede tener un propósito y el worskpace se puede poblar con snapshots identificables de varios codelines

Cada codeline tiene una *política*. Estas políticas definen:

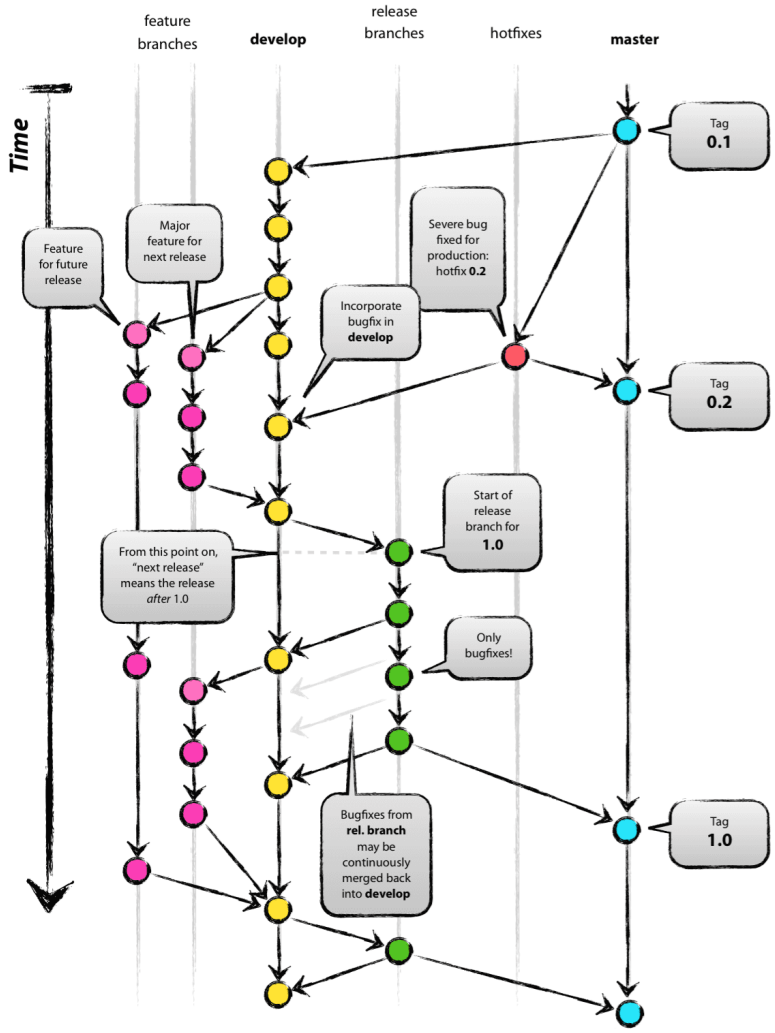
- El propósito del codeline y
- Reglas para cuando y como se pueden hacer cambios

Por ejemplo, desarrollo de características (features)

- Solo los equipos A y B pueden hacer cambios
- Se aplica un build diario centralizado, donde se corren pruebas
- Los archivos solo pueden ser agregados cuando la compilación y generación del build sean exitosos con la última versión "calificada"
- El codeline existe hasta la fase de pruebas alpha
- Se pueden agregar componentes externos luego de acordarlo con el administrador del proyecto

Otro ejemplo lo constituyen los flujos de trabajo de Git, una explicación detallada de los mismos se encuentra acá https://www.atlassian.com/pt/git/workflows#!workflow-feature-branch.

Un ejemplo adicional, se muestra en la siguiente gráfica.



Identificación de versiones

Un nuevo snapshot del codeline puede implicar conceptualmente una nueva versión a identificar.

Algunas versiones son significativas, incluyendo puntos en los cuales existe un release del producto o un build validado.

Estas versiones pueden identificarse a través de etiquetas. Existen diferentes métodos para esta identificación:

- Numérico:
  - Cada release tiene un identificador numérico único, que se expresa usualmente a través de 3 números separados por puntos
  - Una estructura usual es: mayor.menor[.revision[.build]]
- Fechas
  - Se identifica la versión asociada a una fecha, por ejemplo Producto24022012

- Año, se identifica la versión a un año, por ejemplo Office 2007

Breve historia de sistemas de control de versiones

El abuelo de todos los sistemas de control de versiones fue SCSS, escrito en 1972

CVS (Concurrent Version System), es una evolución de SCSS, escrito en 1985. Los archivos son la unidad de versionamiento

SVN, mejoró a CVS. Las revisiones son la unidad de versionamiento

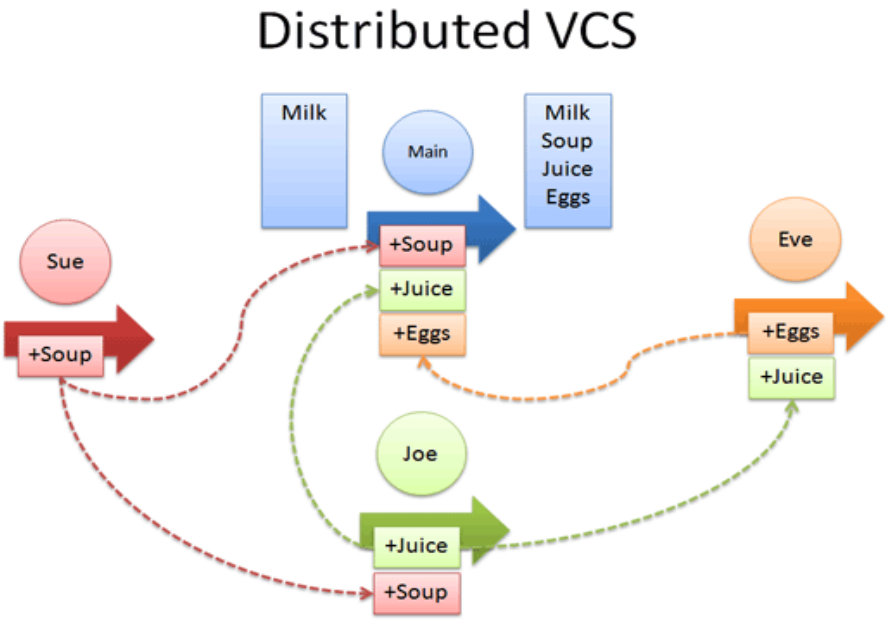
Sistemas comerciales

- Perforce
- AccuRev
- BitKeeper
- Clear Case

Sistema distribuido de control de versiones

El principio de diseño fundamental detrás de un DVCS es que cada usuario mantiene un repositorio en su computadora

No hay necesidad de un repositorio central "privilegiado", aunque algunos equipos designan uno por convención (para hacer integración continua)



Operaciones que se pueden realizar en un sistema de control de versiones

Para una descripción detallada de las diferentes operaciones que se pueden realizar en un sistema de control de versiones, ver:

- A visual guide to version control, http://betterexplained.com/articles/a-visual-guide-to-version-control/
- Pro git, http://git-scm.com/book/en/v2

- Intro to Distributed Version Control (Illustrated), http://betterexplained.com/articles/intro-to-distributed-version-control-illustrated/

Prácticas recomendadas

Mantener absolutamente todo en control de versiones

Cada artefacto relacionado a la creación del software debe estar bajo control de versiones

- Código fuente
- Pruebas
- Scripts de base de datos
- Scripts de build
- Scripts deployment
- Documentación
- Librerías
- Archivos de configuración del software
- Herramientas para desarrollar
- Configuración de ambientes (SO, configuración firewall,etc.)

La buena administración de la configuración es lo que permite implementar el resto de prácticas

No se recomienda tener en control de versiones la salida ejecutable de la compilación de la aplicación, esto porque:

- Son grandes y proliferan rápidamente
- Si se tienen un sistema automatizado de build, se debería poder recrear fácilmente a partir de correr un script de build

Hacer check in regularmente en el sistema de control de versiones

Cuando se incorporan cambios al control de versiones, se convierten en públicos, disponibles para todo el equipo

Esto implica que los cambios deben estar listos para lo que implica el check in

Una solución puede ser el crear branches y luego hacer merge hacia el trunk. Hay algunos problemas con este enfoque

- No se acopla bien a la integración continua, ya que el branch difiere la integración de nueva funcionalidad y los problemas se encuentran hasta hacer el merge
- Si varios desarrollares crean branches, el problema se incrementa, haciendo complejos los merges
- Los conflictos de merge no se solucionan automáticamente
- Es difícil hacer refactor al código base ya que los branches pueden modificar varios archivos, lo que complica el merge

Un mejor enfoque es desarrollar nuevas funcionalidades de forma incremental y hacerles commit en el trunk, de forma regular y frecuente

Para asegurar que no haya problemas con la aplicación en el commit, 2 prácticas son utiles:

- Correr las pruebas de commit antes de hacer el check in
- Introducir cambios de forma incremental

Usar mensajes significativos de commit

La razón mas importante para escribir un mensaje descriptivo de commit es que cuando haya un problema con el build, se pueda saber quien lo provocó y porque falló

Se puede usar un estilo con varios párrafos, donde el primer párrafo resume el cambio y el resto agrega mas detalle

Administración de dependencias, configuración de software y ambientes

Administrar dependencias – librerías

Las librerías externas vienen en forma binaria, a menos que se use un lenguaje interpretado, en cuyo caso se instalan globalmente

Hay algún debate acerca de tener un control de versiones de librerías

Maven es una alternativa para manejar librerías, permite crear un repositorio local o conectarse a un repositorio central de internet. Permite asociar las versiones de librerías que una versión de una aplicación utiliza

Administrar dependencias – componentes

Es una buena práctica dividir las aplicaciones en componentes.

Haciendo esto se limita el alcance de los cambios a la aplicación y se disminuyen los errores de regresión. También se promueve la reutilización y permite un desarrollo mas eficiente

Generalmente se inicia con un build monolítico que crea los binarios para la instalación de la aplicación

Si el sistema crece o hay componentes de los que dependen varias aplicaciones, conviene separar en componentes

Esta separación en componentes, es mejor manejarla a través de dependencias de binarios en vez de dependencias en código fuente

Estrategias para administrar el cambio

- Esconder nueva funcionalidad hasta que esté finalizada

  Se puede usar un branch y hacer merge hasta que se finalice un cambio, pero puede llevar a problemas en el release.

Una solución, para tener liberaciones rápidas, es publicar la nueva funcionalidad, pero no hacerla accesible a los usuarios.

Otra opción es publicar la funcionalidad y hacerla accesible a través de seteos de configuración.

- Hacer todos los cambios de forma incremental

  En la opción de usar un branch, la integración al final puede ser complicada

  Al hacer cambios pequeños, de forma incremental, se van resolviendo problemas para mantener la aplicación funcionando, previniendo problemas al final.

  Para esto, el análisis es importante, en el caso de cambios grandes que deban dividirse en cambios mas pequeños. En ocasiones, esto no será posible.

- Usar un branch por abstracción

  En vez de hacer un branch, se agrega una capa de abstracción sobre la pieza que cambiará.

  Se crea una nueva implementación en paralelo a la implementación existente y cuando está completo, se remueve la implementación actual y opcionalmente la capa de abstracción

  Las partes mas difíciles de manejar con esta estrategia son:

    El punto de entrada para el código base en cuestión

    Administrar cambios que se necesiten a la funcionalidad bajo desarrollo

- Usar componentes para desacoplar partes de la aplicación que cambian a diferentes ritmos

Administrar configuración del software

La información de configuración puede usarse para cambiar el comportamiento del software en tiempo de build, deploy y corrida

Por lo anterior, la configuración debe tratarse como el código: ser sujeta a una administración y pruebas

La flexibilidad a través de la configuración tiene un costo y podría hacer mas difícil un proyecto, por lo tanto debe existir un balance para ello

Los puntos en los que se puede inyectar información de configuración son:

- Los scripts de build pueden obtener la configuración e incorporarla en los binarios en *tiempo de build*
- El empaquetamiento del software puede inyectar configuración en *tiempo de empaquetamiento*, como cuando se crean ears
- Los scripts de deploy o instaladores pueden obtener la información necesaria o preguntarle al usuario por ella y pasarla a la aplicación en *tiempo de deployment*, como parte del proceso de instalación

- La aplicación puede obtener la configuración en *tiempo de inicio* o *tiempo de corrida*

Generalmente se considera una mala práctica inyectar configuración en tiempo de build o tiempo de empaquetamiento, debido a que se debe poder desplegar los mismos binarios a cada ambiente

El corolario de lo anterior es que cualquier cosa que cambie entre deployments necesita ser capturada como configuración y no agregada cuando la aplicación se compila o empaqueta

Administrar configuración de la aplicación

Hay 3 preguntas a considerar cuando se administra la configuración de la aplicación

- ¿Cómo se representa la información de configuración?
- ¿Cómo la accesan los scripts de deployment?
- ¿Cómo varía entre ambientes, aplicaciones y versiones de aplicaciones?

Para representar la configuración: Archivos de propiedades de windows, de java, xml

Para almacenarla, puede ser en: Una base de datos, un sistema de control de versiones o un directorio o registro

En cuanto al acceso configuración a la configuración, la forma mas eficiente de administrar la configuración es tener un servicio central a través del cual cada aplicación obtiene la configuración que necesita

La forma mas fácil para que una aplicación obtenga su configuración es a través del filesystem, es independiente de la plataforma y soportado en cada lenguaje, aunque es necesario mantener la sincronización

Otra alternativa es obtener la configuración de un repositorio centralizado como un RDBMS, LDAP o un web service.

Para modelar la configuración, cada setting de configuración puede ser modelado como una tupla, que depende de:

- La aplicación
- La versión de la aplicación
- El ambiente en el que corre

Algunos casos de uso a considerar al modelar son:

- Agregar un nuevo ambiente
- Crear una nueva versión de la aplicación
- Promover una nueva versión de la aplicación de un ambiente a otro
- Relocalizar un servidor de base de datos
- Administrar ambientes usando virtualización

Administrando ambientes

El principio a tener en mente cuando se administran los ambientes en los que corre la aplicación es que la configuración de ese ambiente es tan importante como la configuración de la aplicación

El problema de manejo de ambientes de forma manual se caracteriza por:

- La colección de información de configuración es grande
- Un pequeño cambio puede hacer fallar una aplicación o degradar severamente su rendimiento
- Una vez que hay una falla, encontrar el problema y arreglarlo toma tiempo indeterminado y requiere personal experto
- Es extremadamente difícil reproducir de manera precisa ambientes configurados manualmente, para propósitos de pruebas
- Es difícil mantener tales ambientes sin la configuración y el comportamiento de nodos puede variar

La clave para administrar ambientes es hacer su creación completamente automatizada, esto es esencial porque:

- Remueve el problema de tener piezas aleatorias de infraestructura cuya configuración es solo entendida por alguien que dejó la organización y no puede ser localizado

- Permite reparar un ambiente puede tomar horas, siempre es mejor poder reconstruir un ambiente en un tiempo predecible

- Es esencial poder crear copias del ambiente de producción para propósitos de pruebas.

La clase de información de configuración de ambientes que debería preocuparnos es:

- Los varios sistemas operativos en los ambientes, incluyendo sus versiones, niveles de parches y settings de configuración.
- Los paquetes adicionales de software que necesitan ser instalados en cada ambiente para soportar las aplicaciones, incluyendo versión y configuración
- La topología de red requerida para que funcione la aplicación.
- Cualquier servicio externo del cual dependa la aplicación, incluyendo versión y configuración.
- Cualquier dato u otro estado que esta presente en el (por ejemplo base de datos).

Finalmente, algunas herramientas para administrar ambientes son Puppet, CfEngine y la virtualización .