Test in a Clone of the Production Environment

If we test in a different environment, every difference results in a risk that what happens under test won't happen in production.

- Iron Age vs Cloud Age
- Virtualization, DevOps
- Infrastructure As Code

Stiamo pensando a server… ma nel caso di clients arbitrari?

UNIVERSITÀ DEGLI STUDI DI MILANO

Make it Easy for Anyone to Get the Latest Executable (2006)

Sorpassato da versioning e build automation?

Era utile per dare accesso a non sviluppatori:

- feedback rapido da utenti (XP)
- per demo da parte di personale di marketing
  - non per forza solo l’ultima versione
- non dice di metterlo nel repository del codice
  - release registry
  - package repository

Everyone can see what's happening

Continuous Integration is all about communication.

- stato della build della mainline (building, passed, coverage,…)
  - una volta si usavano dispositivi fisici (*lava lamp*,…)
  - ora più comuni dashboard virtuali
    - badges
    - integrazione con streumenti di comunicazione (email, sliack,…)
- storia e stato del progetto

UNIVERSITÀ DEGLI STUDI DI MILANO

Automate Deployment

To do Continuous Integration we need multiple environments[...]. Since we are moving executables between these environments multiple times a day, we'll want to do this automatically.

- abbiamo già parlato di *staged build*, *deployment pipeline*
- canary releases
- Blue-Green Deployment e Automatic fallback
- Continuous Integration, Delivery, Deployment

Continuous *Integration*	**Continuous *Delivery***
Integrazione frequente	Rilascio frequente
Mainline stabile	Ambiente di produzione stabile
Automazione build/test	Automazione deployment
Developer focus	Business focus

UNIVERSITÀ DEGLI STUDI DI MILANO

Stili di Integrazione

Riassumendo Fowler descrive tre stili principali di gestione dell'integrazione:

- **Pre-Release Integration**: L'integrazione avviene come una fase distinta del progetto, di solito alla fine dello sviluppo di unità separate. È tipica del modello a cascata e porta a integrazioni lunghe e rischiose.
- **Feature Branches**: Le funzionalità sono sviluppate in rami separati e integrate nel mainline una volta completate. Anche se più frequente della Pre-Release Integration, non garantisce un'integrazione giornaliera con il lavoro degli altri.
- **Continuous Integration**: Ogni sviluppatore integra il proprio lavoro nel mainline almeno quotidianamente. Questo approccio porta a integrazioni piccole e frequenti, riducendo i rischi e facilitando la collaborazione.

UNIVERSITÀ DEGLI STUDI
DI MILANO

8. Intro Software configuration management

Software Configuration Management

- Il Configuration Management nasce nell'industria aerospaziale negli anni '50. Alla fine degli anni '70 inizia a essere applicato nella produzione del software.

> Pratiche che hanno l'obiettivo di rendere sistematico il processo di sviluppo, tenendo traccia dei cambiamenti in modo che il prodotto sia in ogni istante in uno stato (configurazione) ben definito.

- Gli "oggetti" di cui si controlla l'evoluzione sono detti *configuration item* o (in ambito sw) *artifact*.

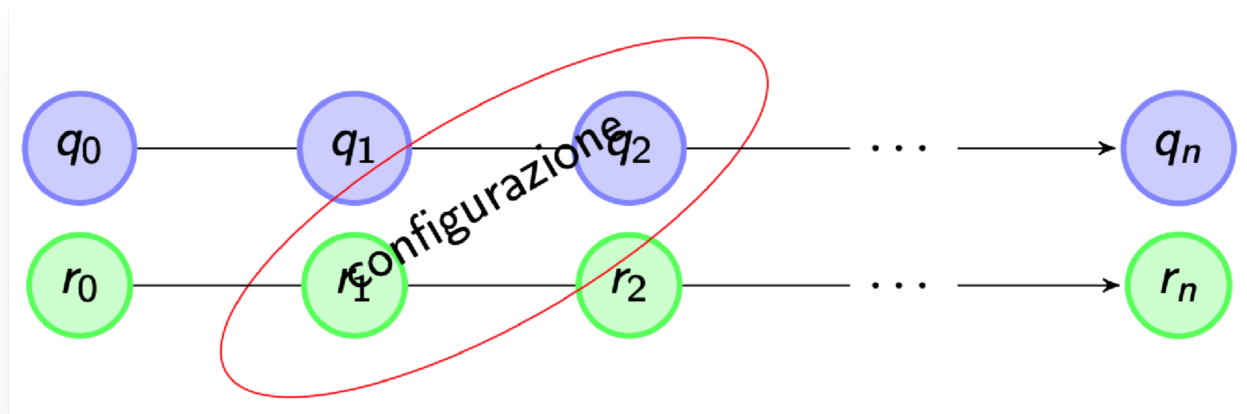
UNIVERSITÀ DEGLI STUDI DI MILANO

Tre scenari di esempio

- gestisce un posto dove mettere i lavori mentre sono in evoluzione, permettendo di richiamare velocemente una qualunque versione memorizzata degli stessi
- Permette la condivisione di tali lavori con altri gestendo accessi contemporanei ed aiutando a gestire i conflitti
- permette tracciabilità

Di cosa si occupano

- Gli *artifacts* classicamente sono file
- l'SCM permette di tracciare/controllare le revisioni degli *artifact* e le versioni delle risultanti *configurazioni*
- a volte fornisce supporto per la generazione del prodotto a partire da una ben determinata configurazione



SCM

- Gli SCM sono per lo più indipendenti da linguaggi di programmazione e applicazioni (una notevole eccezione è Monticello di `Smalltalk`): lavorano genericamente su file, preferibilmente fatti di righe di testo
  - anni ’80: strumenti locali (SCCS, rcs, …)
  - anni ’90: strumenti client-server centralizzati (CVS, subversion, …)
  - anni 2000: strumenti distribuiti *peer-to-peer* (git, mercurial, bazaar, …)

Local only	**Free/open-source**	RCS (1982) · SCCS (1973)
	Proprietary	The Librarian (1969) · Panvalet (1970s) · PVCS (1985) · *QVCS* (1991)
Client–server	**Free/open-source**	CVS (1986, 1990 in C) · CVSNT (1998) · QVCS Enterprise (1998) · Subversion (2000)
	Proprietary	AccuRev SCM (2002) · Azure DevOps (Server (via TFVC) (2005) · Services (via TFVC) (2014)) · ClearCase (1992) · *CMVC* (1994) · Dimensions CM (1980s) · *DSEE* (1984) · Integrity (2001) · Perforce Helix (1995) · SCLM (1980s?) · Software Change Manager (1970s) · StarTeam (1995) · Surround SCM (2002) · Synergy (1990) · Team Concert (2008) · Vault (2003) · *Visual SourceSafe* (1994)
Distributed	**Free/open-source**	*BitKeeper* (2000) · Breezy (2017) · *Code Co-op* (1997) · Darcs (2002) · *DCVS* (2002) · Fossil (2007) · Git (2005) · *GNU arch* (2001) · GNU Bazaar (2005) · Mercurial (2005) · Monotone (2003)
	Proprietary	Azure DevOps (Server (via Git) (2013) · Services (via Git) (2014)) · *TeamWare* (1992) · Plastic SCM (2006)

UNIVERSITÀ DEGLI STUDI DI MILANO

Cosa viene tracciato?

- Qualunque sistema si usi, occorre prendere due decisioni importanti, che influenzano la **replicabilità** della produzione
  1. Si traccia l'evoluzione anche di componenti fuori dal nostro controllo (librerie, compilatori, ecc.) ?
  2. Si archiviano i file che costituiscono il prodotto?

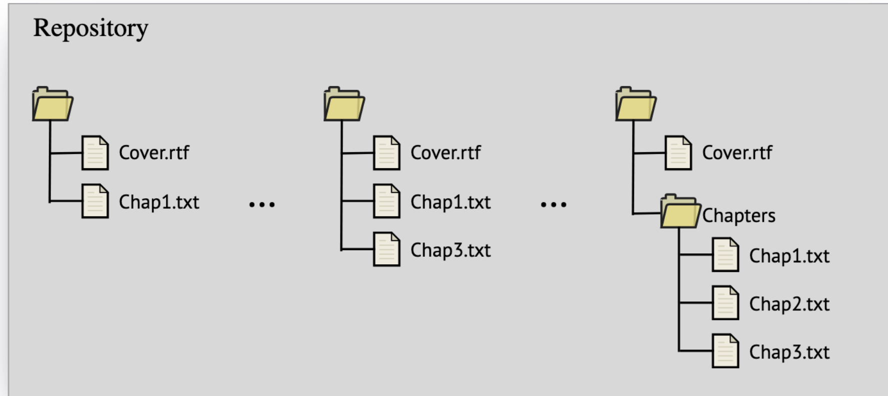
In entrambi i casi la risposta più comune è **NO**, perché molto *costoso* e *poco pratico* ma in questo caso la perfetta replicabilità viene perduta

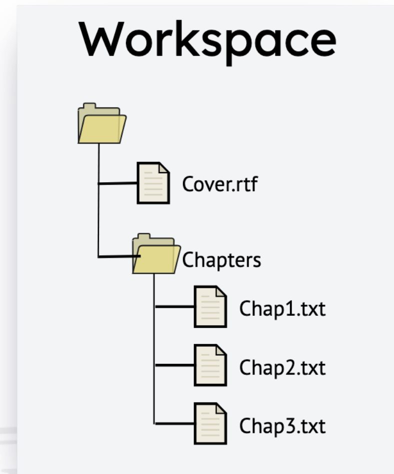
Il meccanismo base

- Il meccanismo di base per controllare l'evoluzione delle revisioni è che ogni cambiamento è regolato da:
  - **check-out** dichiara la volontà di lavorare partendo da una particolare revisione di un *artifact* (o una configurazione di diversi artifacts)
  - **check-in** (o **commit**) dichiara la volontà di registrarne una nuova (spesso chiamata change-set)
- Queste operazioni vengono attivate rispetto a un *repository*
  - Cioè producono spostamenti tra il *repository* (che contiene tutte le configurazioni) e il *workspace* (l'ambiente su cui si lavora nel filesystem)

UNIVERSITÀ DEGLI STUDI DI MILANO

Repository





- Il repository mantiene informazioni comprese date, etichette (tag), versioni, diramazioni (branches), etc
- Il repository può salvare anche solo le differenze tra una versione e l'altra
- Può essere centralizzato o distribuito

UNIVERSITÀ DEGLI STUDI DI MILANO

La regolazione del lavoro concorrente

- Quando il *repository* è condiviso da un gruppo di lavoro, nasce il problema di gestirne l'accesso concorrente:
  - modello "pessimistico" (RCS): il sistema gestisce l'accesso agli *artifact* in mutua esclusione attivando un lock al check-out
  - modello "ottimistico" (CVS): il sistema si *disinteressa* del problema e fornisce supporto per le attività di merge di *change-set* paralleli potenzialmente conflittuali

Il modello pessimistico è, nello sviluppo *software*, tanto irrealistico e ideale quanto il processo a "cascata".

Il modello ottimistico può essere parzialmente regolato tramite i rami paralleli di sviluppo (branch)

UNIVERSITÀ DEGLI STUDI DI MILANO

Il merge

Il merge è un'operazione essenziale e delicata. Possono esistere strategie diverse per gestirlo.

- Tenendoci sul generale possiamo identificare almeno queste macro situazioni (ragionando su *artifact* e *hunk*):
  - lavoro parallelo su *artifact* diversi
  - lavoro parallelo sullo stesso *artifact* ma *hunk* differenti
  - lavoro parallelo sullo stesso *artifact* e stesso *hunk*

Cosa è un *hunk*?

- un gruppo di righe consecutive che il mio tool (tramite qualche euristica) decide di considerare in maniera atomica.

UNIVERSITÀ DEGLI STUDI DI MILANO

La terminologia per i *merge*

La terminologia più usata fa riferimento alla coppia di programmi POSIX:

`diff` calcola la differenza fra due revisioni (R_0, R_1), calcolata per righe, cercando di minimizzare il numero di inserimenti e cancellazioni (ricerca della sottosequenza più lunga)

L'intersezione è: `a b c d f g j z`

Le differenze sono: `e h i q k r x y`
`+ − + − + + + +`

Il diff è diviso in **hunk** ("fette"): `e, h/i, q/krxy`

`patch` è il programma che permette di applicare il diff non solo a R_0 per ottenere R_1, ma anche a un qualunque R'_0 "vicino" a R_0 (che diventerà un R'_1), applicando alcune euristiche per ogni *hunk*.

R_0

```
a
b
c
d
f
g
h
j
q
z
```

R_1

```
a
b
c
d
e
f
g
i
j
k
r
x
y
z
```

```
4a5
> e

7c8
< h
---
> i

9c10,13
< q
---
> k
> r
> x
> y
```