

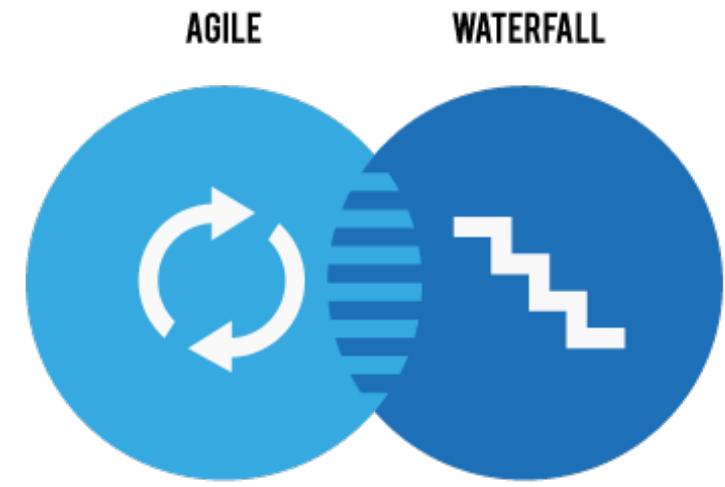
# Modelli di processo per lo sviluppo del software: i modelli agili



*Corso di Ingegneria del Software  
CdL Informatica Università di Bologna*

# Obiettivi di questa lezione

- I modelli di processo **agili**
- Il manifesto agile
- Extreme Programming (XP)



# Sviluppare software è un'attività sociale

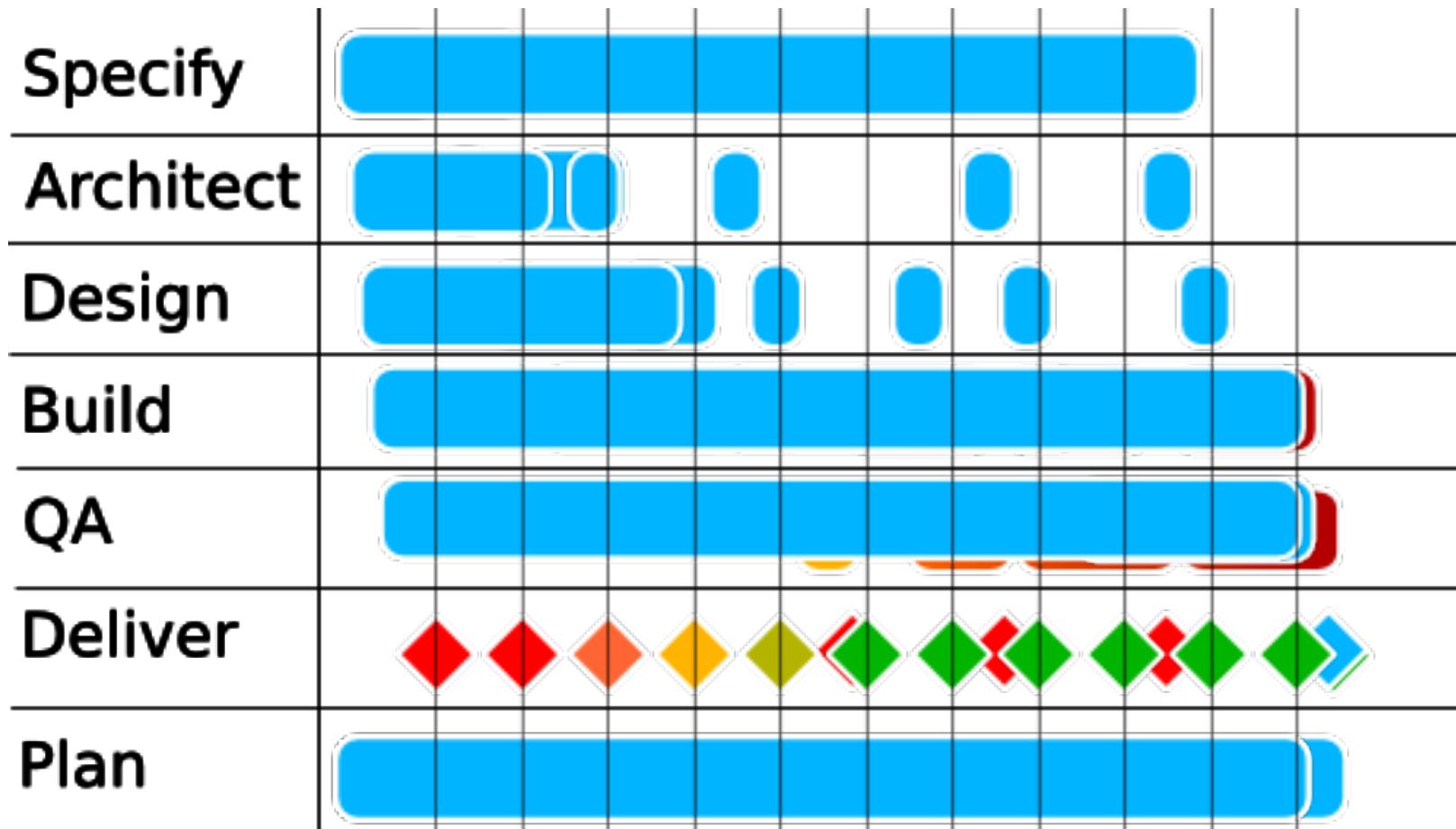
- Quasi tutti i problemi difficili nello sviluppo software riguardano le persone e non le tecnologie.
- Scrivere software non è difficile.
- Tuttavia, scrivere il software *\*giusto\** è molto difficile, perché occorre capire i bisogni degli utenti, negoziare i tempi di consegna, distribuire i compiti, fare compromessi
- Le persone capaci di fare bene il software sono rare: quando hanno successo di solito dimostrano sia abilità tecnica che buone capacità di relazione interpersonale

# La pianificazione spesso fallisce

- Alcuni problemi sono complessi, la soluzione richiede il contributo di molte persone
- La soluzione all'inizio non è chiara
- I requisiti della soluzione durante lo sviluppo cambieranno, perché gli utenti non sanno bene cosa vogliono
- Il prodotto può essere consegnato in versioni incrementali successive, ma non sappiamo quanti incrementi ci vorranno
- È richiesta una collaborazione stretta e un feedback rapido e continuo dagli utenti finali

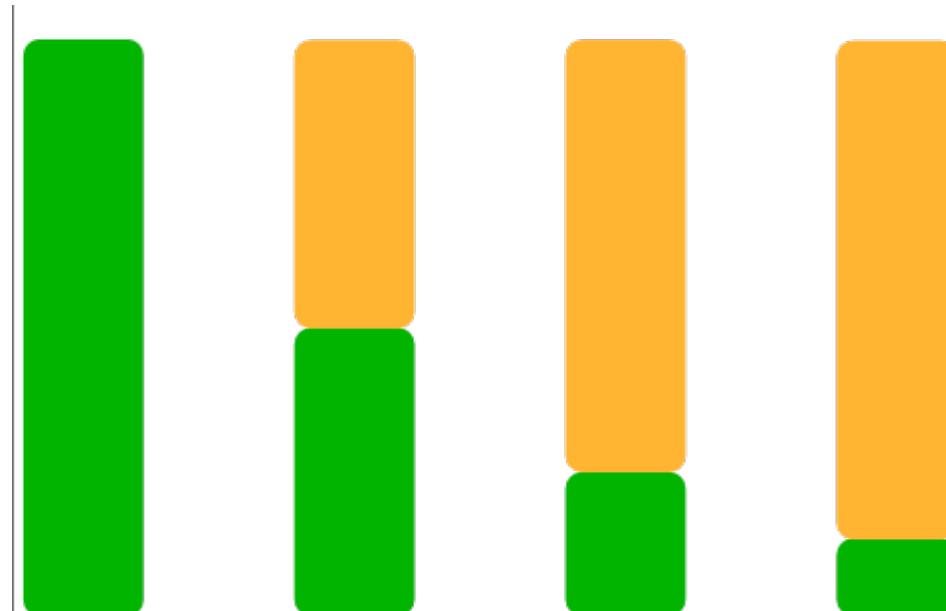
# L'interesse dell'utente:

in quale riga lo vedete?



# Volatilità dei requisiti

Ogni sei mesi – o anche meno – circa metà dei requisiti di un prodotto software perdono di interesse



Modelli di processo 2

[www.infoq.com/articles/the-curse-of-the-change-control-mechanism](http://www.infoq.com/articles/the-curse-of-the-change-control-mechanism)

# ....quindi...

- L'utente deve far parte del gioco fin dall'inizio
- Occorre evitare gli sprechi di tempo
- occorre adattarsi al cambiamento delle richieste dell'utente

# Migliorare i processi iterativi

- I processi iterativi devono produrre valore per gli stakeholder in modo incrementale e senza sprechi
- La filosofia **agile** ha lo scopo di evitare gli sprechi e le perdite di tempo – le attività inutili

# Etica del Movimento Agile

Stiamo scoprendo modi migliori di costruire il software facendolo e aiutando altri a farlo. Attribuiamo valore a:

Individui e interazioni più che a processi e strumenti  
Software che funziona più che a documentazione completa  
Collaborazione col cliente più che a negoziazione contrattuale  
Reagire al cambiamento più che a seguire un piano

I valori a destra sono importanti, ma noi preferiamo quelli a sinistra

[www.agilemanifesto.org](http://www.agilemanifesto.org)

Modelli di processo 2

# I principi agili

1. La nostra priorità è soddisfare il cliente mediante consegne anticipate e continue di software di valore
2. Le persone dell'azienda e gli sviluppatori devono quotidianamente lavorare assieme durante tutto il progetto
3. Le modifiche ai requisiti sono benvenute, anche nelle ultime fasi dello sviluppo
4. Consegnare di frequente software funzionante
5. Il software funzionante è la prima misura di progresso
6. I progetti si costruiscono attorno a individui motivati. Dategli l'ambiente ed il supporto di cui hanno bisogno, e confidate che facciano il loro lavoro
7. Le migliori architetture, requisiti, e design emergono da team auto-organizzanti
8. Il metodo più efficace ed efficiente di trasmettere informazioni verso e all'interno di uno sviluppo è mediante conversazioni faccia a faccia
9. I processi agili promuovono lo sviluppo sostenibile
10. L'attenzione costante all'eccellenza tecnica e al buon design esaltano l'agilità
11. La semplicità è essenziale
12. I team di sviluppo valutano la propria efficacia ad intervalli regolari e modificano di conseguenza il proprio comportamento

# I principi agili

1. La nostra priorità è **soddisfare il cliente** mediante consegne anticipate e continue di software di valore
2. Le persone dell'azienda e gli sviluppatori devono **quotidianamente lavorare assieme** durante tutto il progetto
3. Le **modifiche ai requisiti sono benvenute**, anche nelle ultime fasi dello sviluppo
4. Consegnare di frequente software **funzionante**
5. Il software funzionante è la **prima misura di progresso**
6. I progetti si costruiscono attorno a **individui motivati**. Dategli l'ambiente ed il supporto di cui hanno bisogno, e confidate che facciano il loro lavoro
7. Le migliori architetture, requisiti, e design emergono da **team auto-organizzanti**
8. Il metodo più efficace ed efficiente di trasmettere informazioni verso e all'interno di uno sviluppo è mediante **conversazioni faccia a faccia**
9. I processi agili promuovono lo **sviluppo sostenibile**
10. L'attenzione costante all'**eccellenza tecnica** e al buon design esaltano l'agilità
11. La **semplicità** è essenziale
12. I team di sviluppo **valutano la propria efficacia** ad intervalli regolari e modificano di conseguenza il proprio comportamento

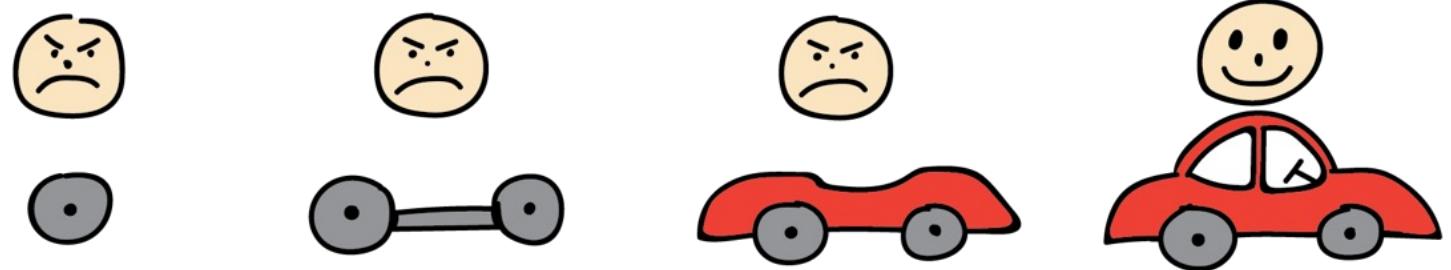
# Metodi agili

I metodi agili sono una famiglia di metodi di sviluppo che hanno in comune:

- **Rilasci frequenti** del prodotto sviluppato in modo incrementale
- **Collaborazione continua** del team di sviluppo col cliente
- **Documentazione** di sviluppo **ridotta**
- **Valutazione** sistematica e continua di **valori e rischi** dei **cambiamenti**

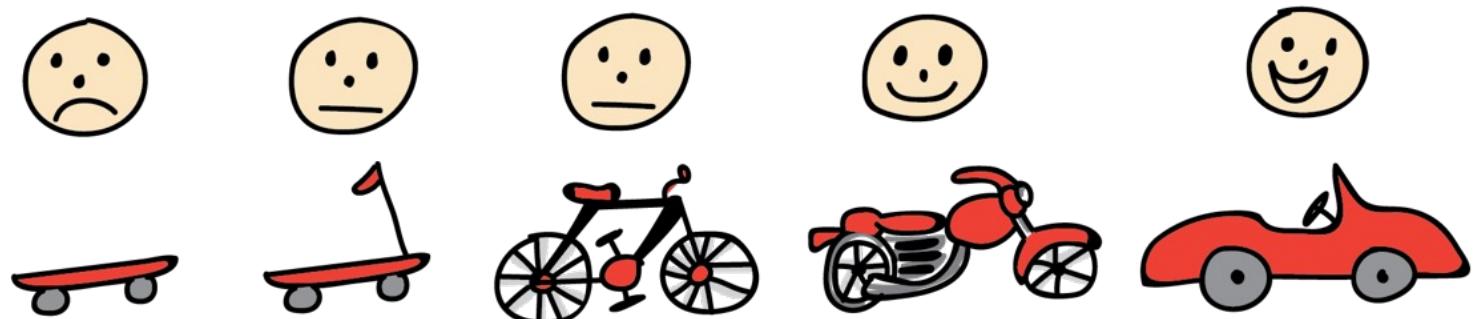
# Minimal Viable Product (prodotto minimo funzionante)

Not Like  
This!



---

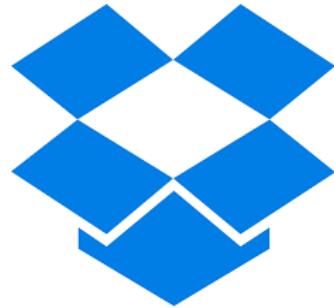
Like This!



# MVP: inizi di dropbox

<https://www.youtube.com/watch?v=xy9nSnalvPc>

[https://www.youtube.com/watch?v=qxFLfY7\\_Gqw](https://www.youtube.com/watch?v=qxFLfY7_Gqw)



# Metodi agili... ne esistono tanti!

- Extreme Programming (XP)
  - Scrum
  - Feature-Driven Development (FDD)
  - Adaptive Software Process
  - Crystal Light Methodologies
  - Dynamic Systems Development Method (DSDM)
  - Lean Development
- 
- DevOps
  - SAFe
  - Less

# eXtreme Programming (XP)

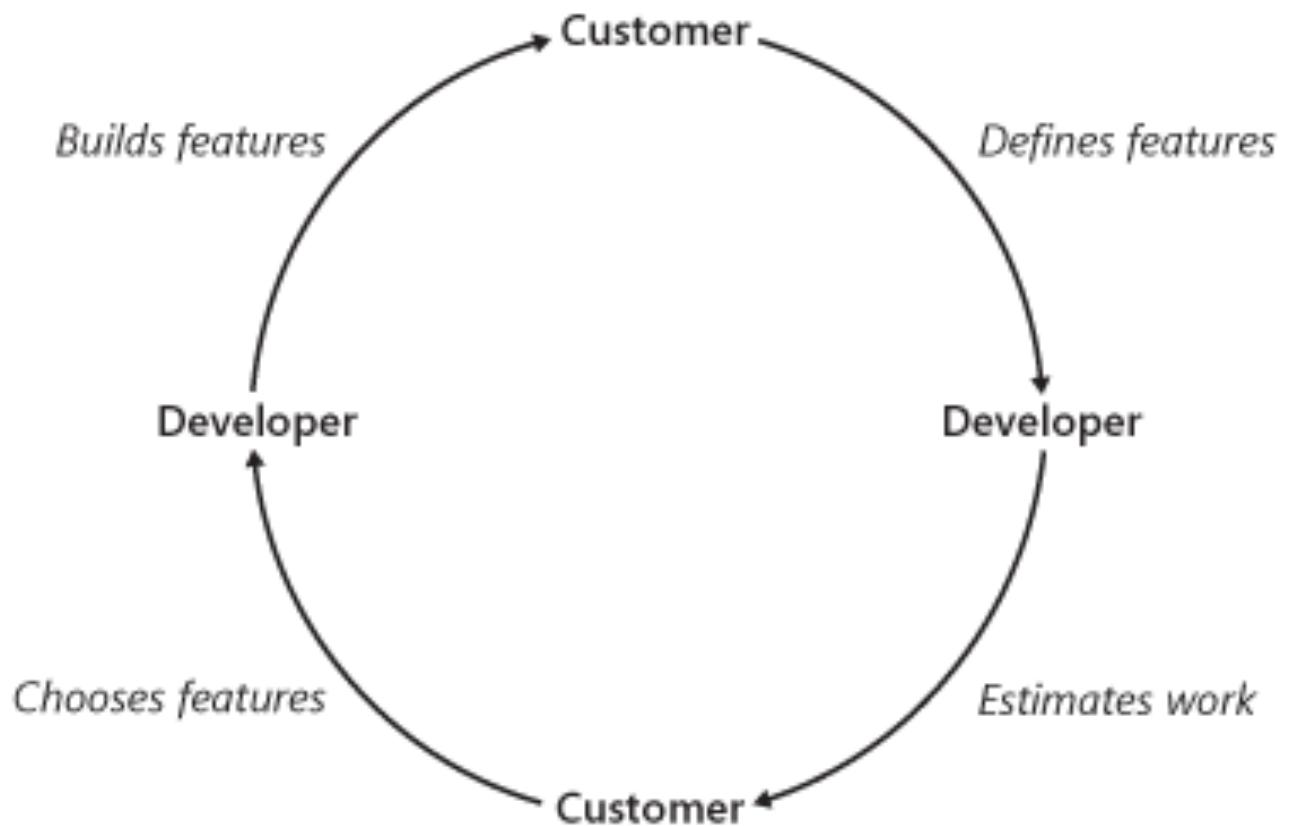
“Extreme Programming è una disciplina di sviluppo software basata sui valori di

- *semplicità,*
- *comunicazione,*
- *feedback, e*
- *coraggio”.*



Kent Beck

# La base della collaborazione agile in XP



# Il team XP

- Il team di sviluppo, di solito costituito da meno di dieci persone, è riunito nello stesso locale
- E' sempre presente un rappresentante del cliente, capace di rispondere a domande del team riguardo ai requisiti
- Il team deve usare comportamenti di sviluppo **semplici**, allo stesso tempo **capaci di informare tutti** sullo stato del progetto ma anche di **adattare** i comportamenti alla situazione specifica

# Le pratiche di Extreme Programming

- I requisiti sono “user stories”
- Il “planning game”
- Piccoli rilasci
- Cliente On-site
- “Prima i test, poi il codice”
- La metafora di riferimento
- Integrazione continua
- Proprietà collettiva del codice
- Settimana di 40 ore di lavoro
- Uso sistematico di standard di codifica
- Programmazione di coppia
- Refactoring
- Progettazione semplice

# I requisiti sono “user stories”

## User stories:

- Brevi frasi, usate al posto di documenti dettagliati di specifica dei requisiti
- Scritte assieme ai clienti: cosa si aspettano dal sistema
- una storia è descritta da una o due frasi in testo naturale, con la terminologia del cliente
- utili al team per stimare i tempi/costi del rilascio della nuova versione (incremento)

# Esempi di user stories

- Uno studente può acquistare online un abbonamento di parcheggio
- Gli abbonamenti di parcheggio si possono pagare con carta di credito
- Gli abbonamenti di parcheggio si possono pagare con Paypal
- Un professore può inserire i voti online
- Uno studente può visualizzare l'orario delle lezioni
- Uno studente può ottenere una registrazione della lezione
- Uno studente può iscriversi ad un corso se soddisfa i prerequisiti
- La registrazione di un corso si può visualizzare mediante un browser

# Esempio di scheda per user story

173. Students can purchase parking passes.

Priority: 8  
Estimate: 4

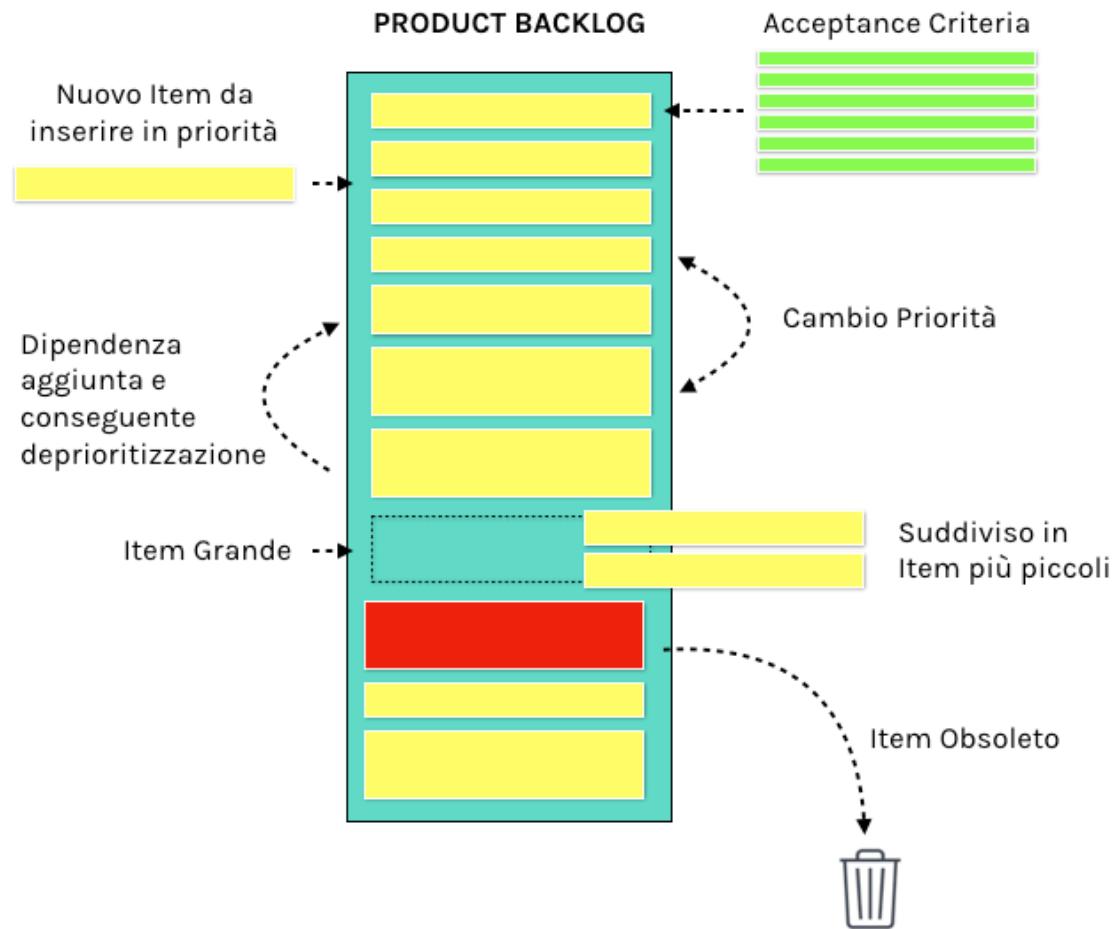
Story points

# Formato preferito per le US

Come studente  
voglio poter comprare on  
line un abbonamento  
mensile al parcheggio  
per non tirare fuori soldi  
tutti i giorni

→ Tipo di utente  
→ Obiettivo  
→ Motivazione

# Backlog di prodotto: insieme di US



<https://www.productheroes.it/product-backlog-agile-scrum/>

# Storie e iterazioni



# Il “Planning game”

- Le “**User Stories**” permettono di fare una pianificazione a breve termine
  - Una storia è una breve descrizione di qualcosa che vuole il cliente
  - La descrizione può essere arricchita da altre storie durante lo sviluppo
  - Le **priorità** tra le storie sono definite dal **cliente**
  - Le **risorse** necessarie (**story points**) e i rischi sono valutati dagli **sviluppatori**
- “The **Planning Game**”
  - Le storie di più alto rischio e priorità sono affrontate per prime, in incrementi “*time boxed*”
  - Il Planning Game si rigioca per ciascuna iterazione

# La presenza del cliente

- Il cliente (un suo rappresentante) è sempre disponibile per chiarificare le storie e per prendere rapidamente decisioni critiche
- Gli sviluppatori non devono fare ipotesi: risponde direttamente il cliente
- Gli sviluppatori non devono attendere le decisioni del cliente
- La comunicazione “faccia a faccia” minimizza la possibilità di ambiguità ed equivoci

# Il metodo MOSCOW per mettere in priorità le US

MOSCOW : acronimo per Must/Should/Could/Won't  
(DEVE / DOVREBBE / POTREBBE / NO)

- Must: funzioni che DEBBONO esserci nel prodotto
- Should: funzioni che DOVREBBERO esserci
- Could: funzioni che POTREBBERO esserci
- Wont: funzioni che NON INSERIREMO nella versione attuale

# Esempio

- L'utente potrà loggarsi con un username e passwd
- L'utente potrà registrare un nuovo account
- L'utente potrà vedere tutti i documenti Word inclusi nel file system
- L'utente potrà cancellare tutti i documenti Word
- L'utente potrà accedere ogni documento entro il file system
- L'utente potrà loggarsi con doppia autenticazione

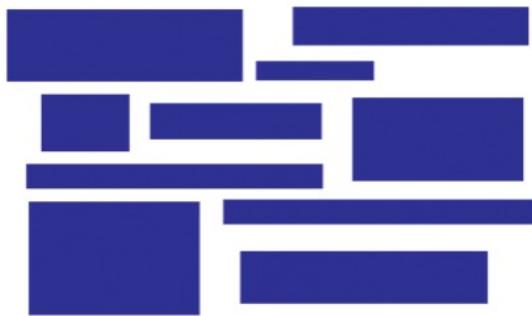
# Esempio

- L'utente DEVE loggarsi con con username e passwd
- L'utente DOVREBBE registrare un nuovo account
- L'utente DOVREBBE vedere tutti i documenti Word inclusi nel file system
- L'utente POTREBBE cancellare tutti i documenti Word
- L'utente DEVE accedere ogni documento entro il file system
- L'utente NON POTRÀ loggarsi con doppia autenticazione

In scope  
for this timeframe

(Project / Increment / Timebox)

Must Have



Typically  
no more  
than  
60% effort

Should Have



Typically  
around  
20% effort

Out of scope  
for this timeframe

Won't Have this time



Could Have



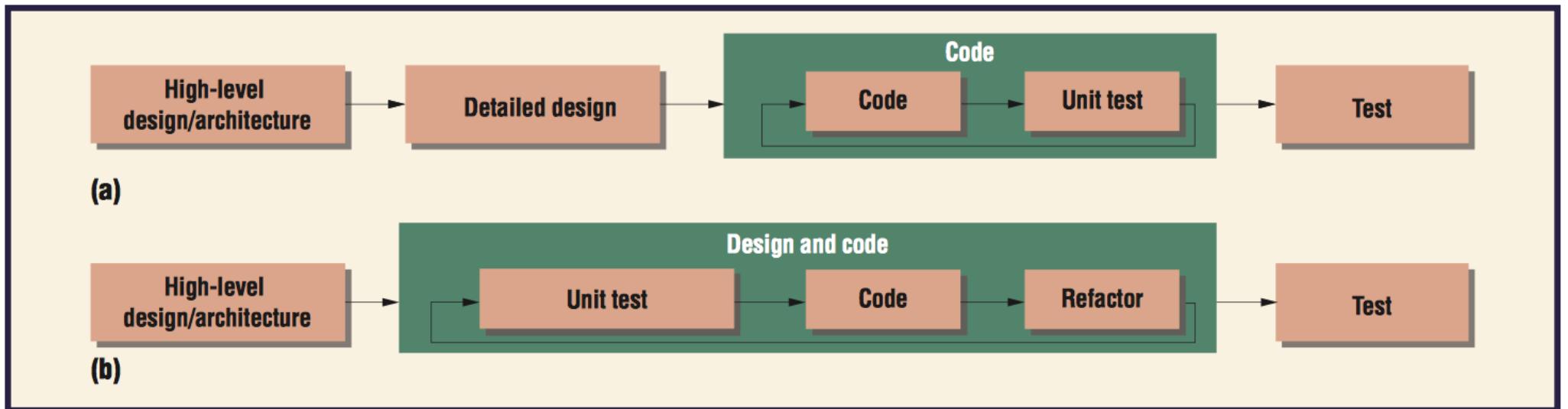
# Sviluppo test-driven

## Test-Driven Development (TDD)

- Scegliere una user story e definire i test prima del codice
- TDD è una tecnica di programmazione
- Automatizzare i test (es. usare xUnit)
- Deve girare tutto prima di proseguire con altro codice
- *Unit test vs acceptance test*



# Test classico vs test driven



a) Testing classico b) test-driven design come tecnica di programmazione

<https://semaphoreci.com/blog/test-driven-development>

# Test di accettazione

(customer tests)

## Test di accettazione

- Guidati dalle user stories
- Scritti col cliente
- Funzionano come “contratto”
- Misura del progresso



# Esempio

*Support technician sees customer's  
history on screen at the start of a call*

Esempio di user story su scheda

- Simulate a call with Fred's account number and verify that Fred's info can be read from the screen
- Verify that the system displays a valid error message for a non-existing account number
- Omit the account number in the incoming call completely and verify that the system displays the text "no account number provided" on the screen

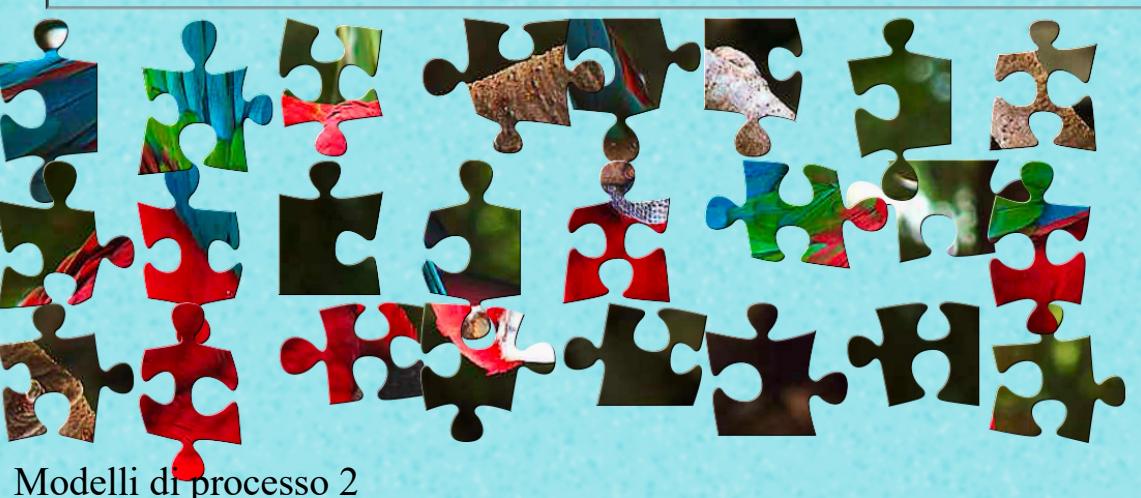
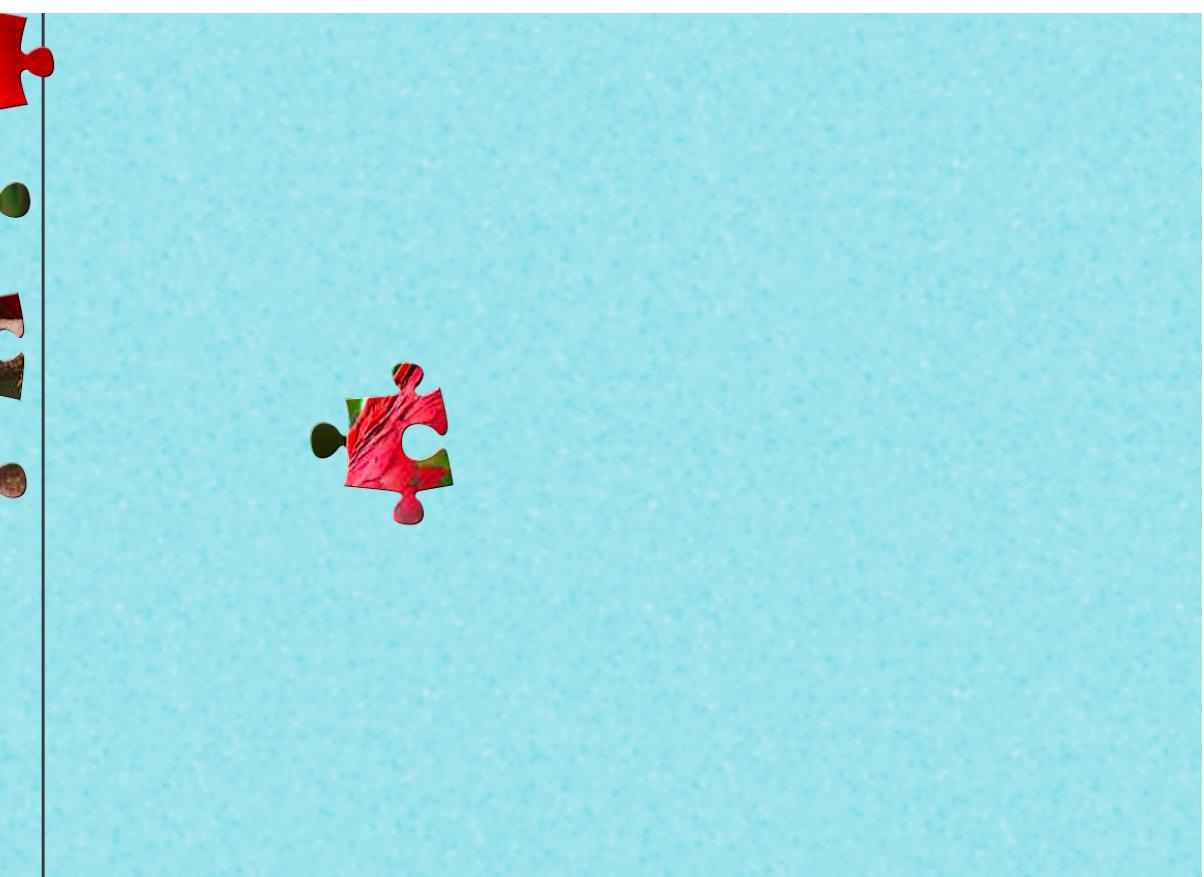
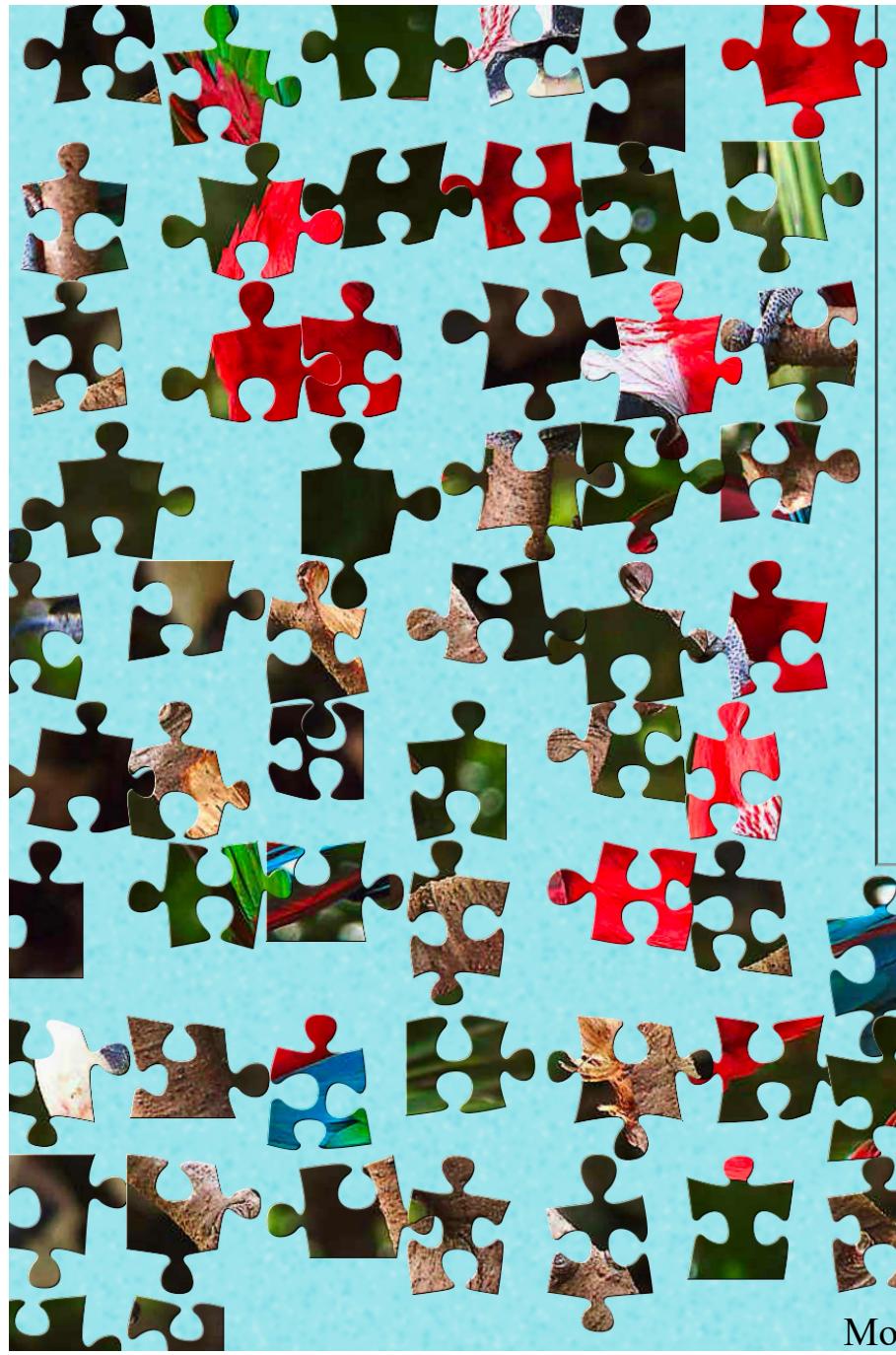
Esempio di test scritto sul retro della user story

# Piccoli rilasci

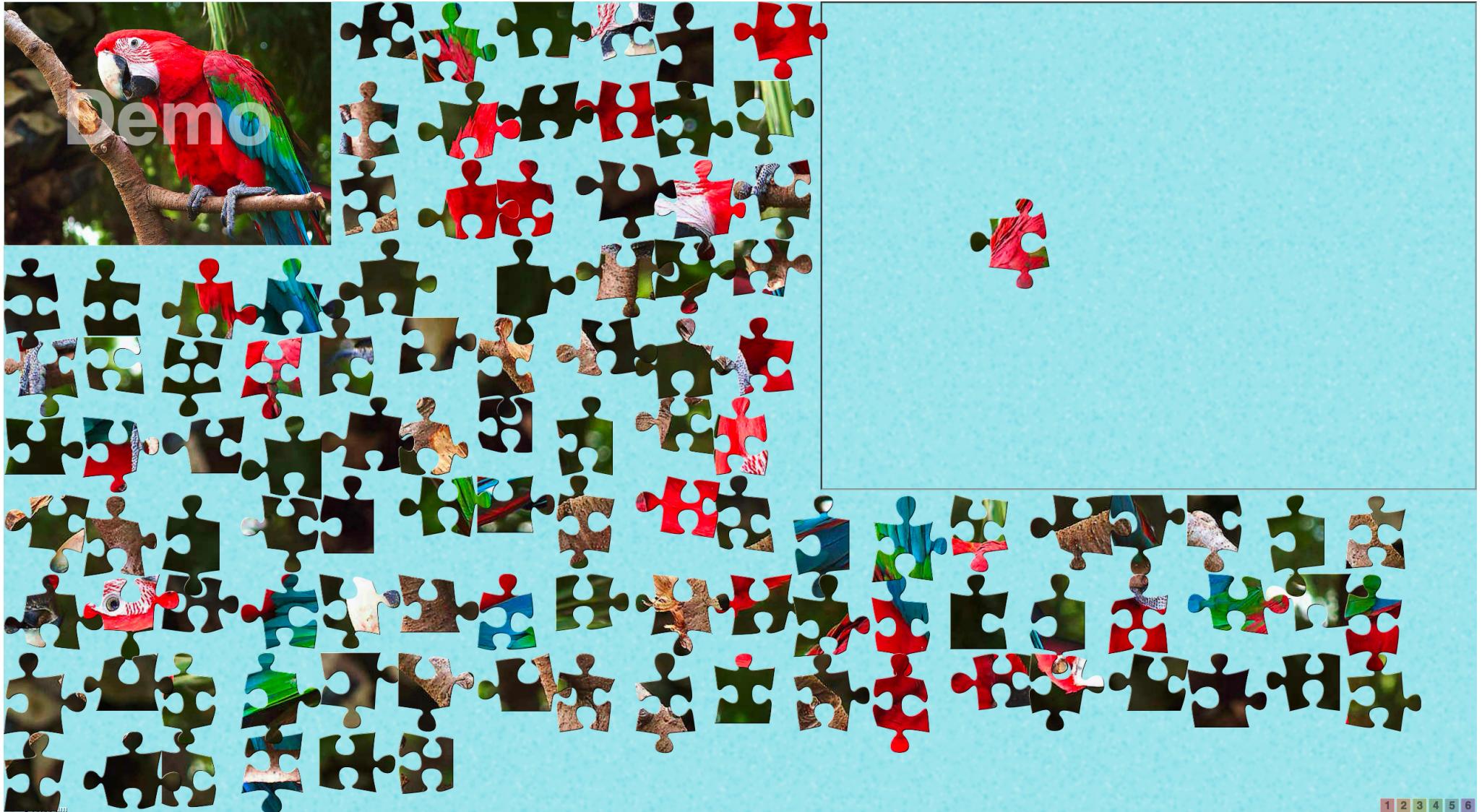
- Timeboxed (ovvero di durata “breve” prefissata)
- Minimali, ma comunque utili (**microincrementi**)
  - Mai cose come ‘implementare il database’
- Ottener feedback dal cliente presto e spesso
- Eseguire il “planning game” dopo ciascuna iterazione
  - Si voleva qualcosa di diverso?
  - Le priorità sono cambiate?

# La metafora

- I progettisti XP sviluppano una visione comune di come funzionerà il programma, detta “**metafora del sistema**”
- Esempio: *“questo programma funziona come uno sciame d'api, che cerca il polline a lo porta nell'alveare”* (sistema di information retrieval basato su agenti)
- Non sempre la metafora è poetica. In ogni caso il team deve usare un glossario comune di nomi di entità rilevanti per il progetto



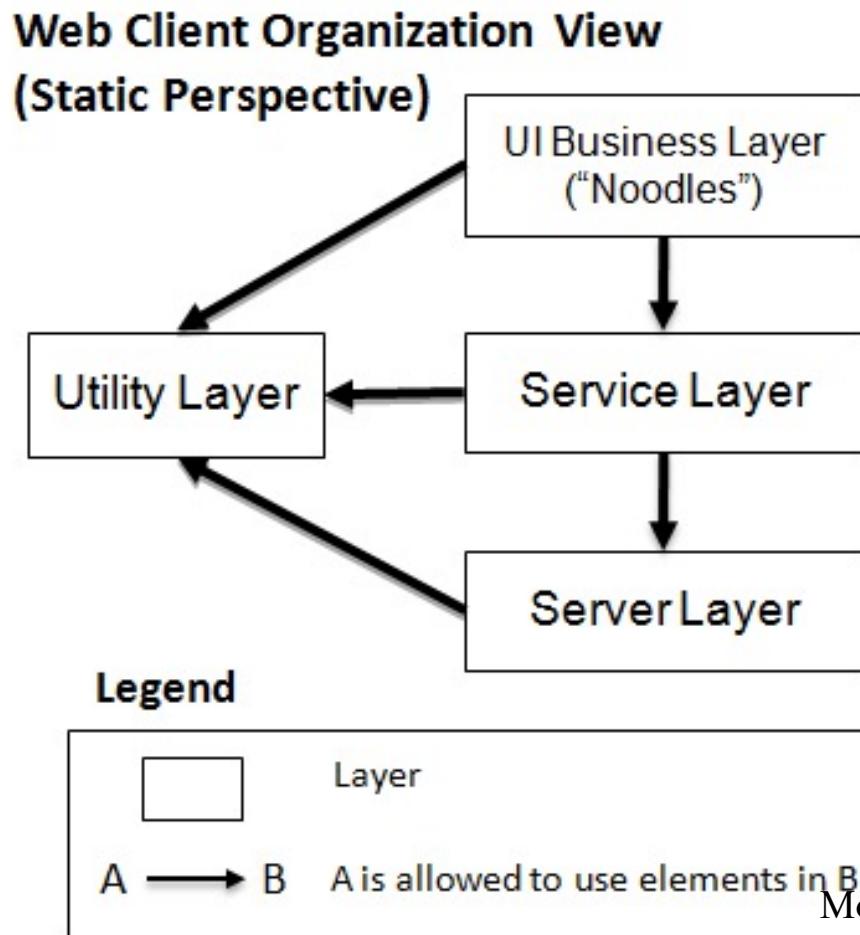
Modelli di processo 2



Modelli di processo 2

# Esempio di metafora

Fonte: neverletdown.net/topics/architecture/



The “Bento Box”

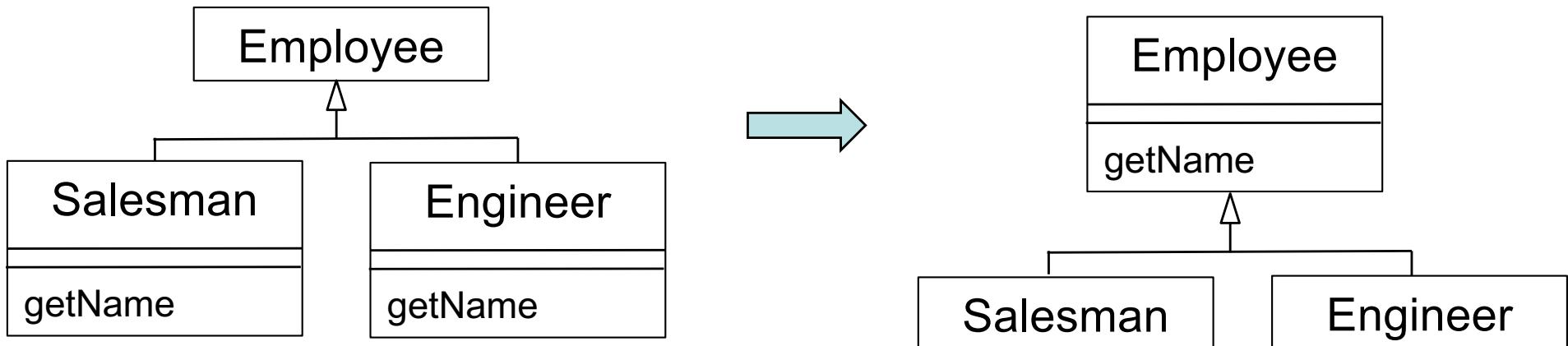
# Progettare in modo semplice

- No Big Design Up Front (BDUF)
- “Fare la cosa più semplice che possa funzionare”
  - Includere la documentazione
- “You Aren’t Gonna Need It” (YAGNI)
- **Opzione: usare schede CRC**  
(vedere le lezioni sui requisiti e sul design per un’ introduzione alle schede CRC)

# Rifattorizzare (refactoring)

- **Refactoring**: migliorare il codice esistente senza cambiarne la funzionalità
- Avere il coraggio di buttare via codice
  - Semplificare il codice
  - Rimuovere il codice ridondante
  - Cercare le opportunità di astrazione
- Il refactoring usa il testing per assicurare che con le modifiche non si introducano errori

# Refactoring: esempio



Le sottoclassi hanno ciascuna metodi con risultati identici

Spostando il metodo comune nella superclasse, si elimina la ridondanza.  
Eliminare le ridondanze nel codice è importante

# Pair programming

La programmazione di coppia  
(pair programming) è tipica di  
eXtreme Programming (XP)



Con la programmazione di coppia:

- Due progettisti lavorano allo stesso compito su un solo computer
- Uno dei due, **il driver**, controlla tastiera e mouse e scrive il codice
- L'altro, **il navigatore**, osserva il codice cercando difetti e partecipa al brainstorming su richiesta
- I ruoli di driver e navigatore vengono scambiati tra i due progettisti periodicamente (es. a metà giornata)
- Le coppie cambiano ogni giorno: lo scopo è che tutti prendano confidenza col codice (vedi: proprietà collettiva del codice)

# Pair programming migliora la qualità

	Progetto1: solisti	Progetto2: coppie
Dimensione (KLOC)	20	520
Team	4	12
Sforzo (mesi persona)	4	72
Produttività (KLOC/mp)	5	7.2
Produttività (KLOC/mpair)	n.d.	14.4
Difetti test unità	107 (5.34 difetti/KLOC)	183 (0.4 difetti/KLOC)
Difetti test integrazione	46 (2.3 difetti/KLOC)	82 (0.2 difetti/KLOC)

Fonte: Williams, *Pair Programming Illuminated*, AW, 2002

# Integrazione continua

- La coppia scrive i test ed il codice di un task (= parte di una storia utente)
- La coppia esegue tutto il test di unità
- La coppia esegue l'integrazione della nuova unità con le altre
- La coppia esegue tutti i test di regressione
- La coppia passa al task successivo a mente sgombra (capita una o due volte al giorno)
- L'obiettivo è prevenire l' "*Integration Hell*"

# Proprietà collettiva del codice

- Il codice appartiene al progetto, non ad un individuo
- Durante lo sviluppo tutti possono osservare e **modificare** qualsiasi classe
- Uno degli effetti del *collective ownership* è che il codice troppo complesso non sopravvive a lungo
- Affinché tale strategia funzioni occorre l’“integrazione continua”

# Passo sostenibile

- Kent Beck dice: “ . . . freschi e vogliosi ogni mattina, stanchi e soddisfatti ogni sera”
- Lavorare fino a tarda notte danneggia le prestazioni: uno sviluppatore stanco fa più errori, e alla lunga il gioco non vale la candela
- Se il progetto danneggia la vita privata dei partecipanti, alla lunga lo scotto lo paga il progetto stesso

# Convenzioni di codifica

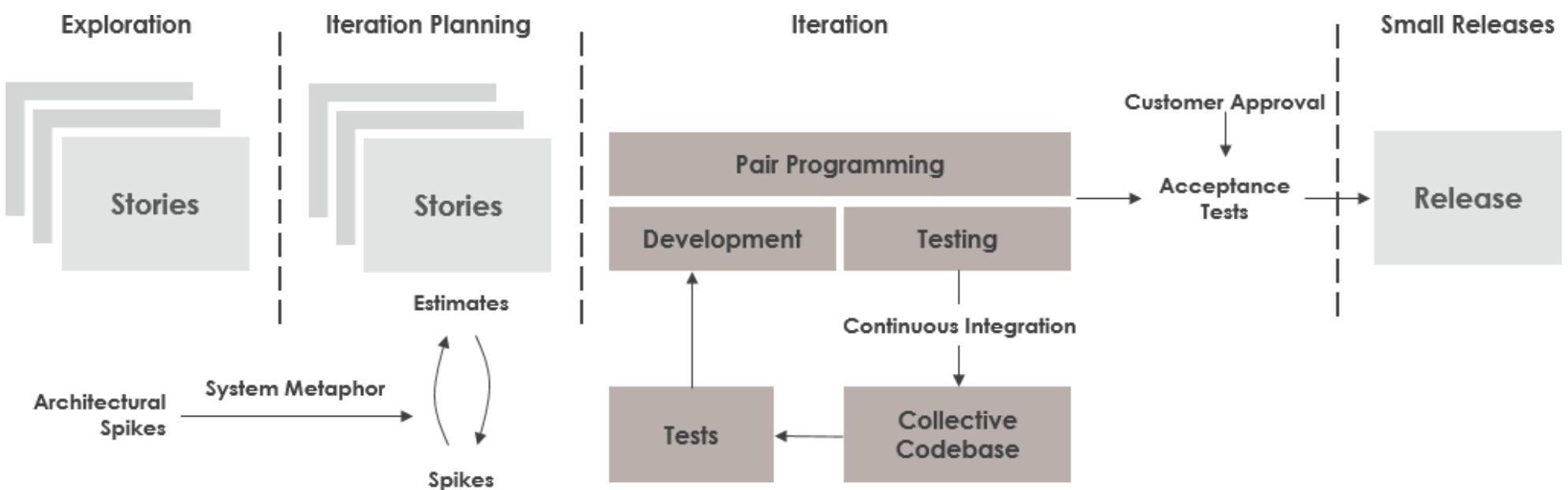
- Usare convenzioni di codifica
  - A causa del Pair Programming, delle rifattorizzazioni e della proprietà collettiva del codice, occorre poter addentrarsi velocemente nel codice altrui
- Commentare il codice
  - Priorità al codice che “svela” il suo scopo
    - Se il codice richiede un commento, riscrivilo
    - Se non puoi spiegare il codice con un commento, riscrivilo

<https://peps.python.org/pep-0008/>

<http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>

[https://en.wikibooks.org/wiki/Computer\\_Programming/Coding\\_Style](https://en.wikibooks.org/wiki/Computer_Programming/Coding_Style)

# XP: il processo



# Il 13º Principio: La riunione in piedi

- Ogni giorno inizia con una riunione di 15 minuti
  - Tutti in piedi (così la riunione dura meno) in cerchio
  - Ciascuno a turno dice:
    - Cosa ha fatto il giorno prima
    - Cosa pensa di fare oggi
    - Quali ostacoli sta incontrando
  - Può essere il momento in cui si formano le coppie

# Conclusioni

I metodi agili, e XP in particolare, prediligono:

- Piccoli team
- Il cliente è nel team - il Product Owner
- Accettano i cambiamenti dei requisiti
- Iterazioni brevi e consegne frequenti
- Praticano l'integrazione continua

# Autotest

- Cos'è un MVP (minimum viable product)?
- Quali sono i valori agili?
- Cos'è una user story?
- A cosa serve il metodo MOSCOW?
- Cos'è la proprietà collettiva del codice?
- Cos'è il refactoring?

# Letture raccomandate

- Cohn e Ford, Introducing an agile process to an organization, *IEEE Computer*, 2003
- Janzen & Saiedian, Does Test-Driven Development Really Improve Software Design Quality?, *IEEE Software*, March/April 2008

# Riferimenti

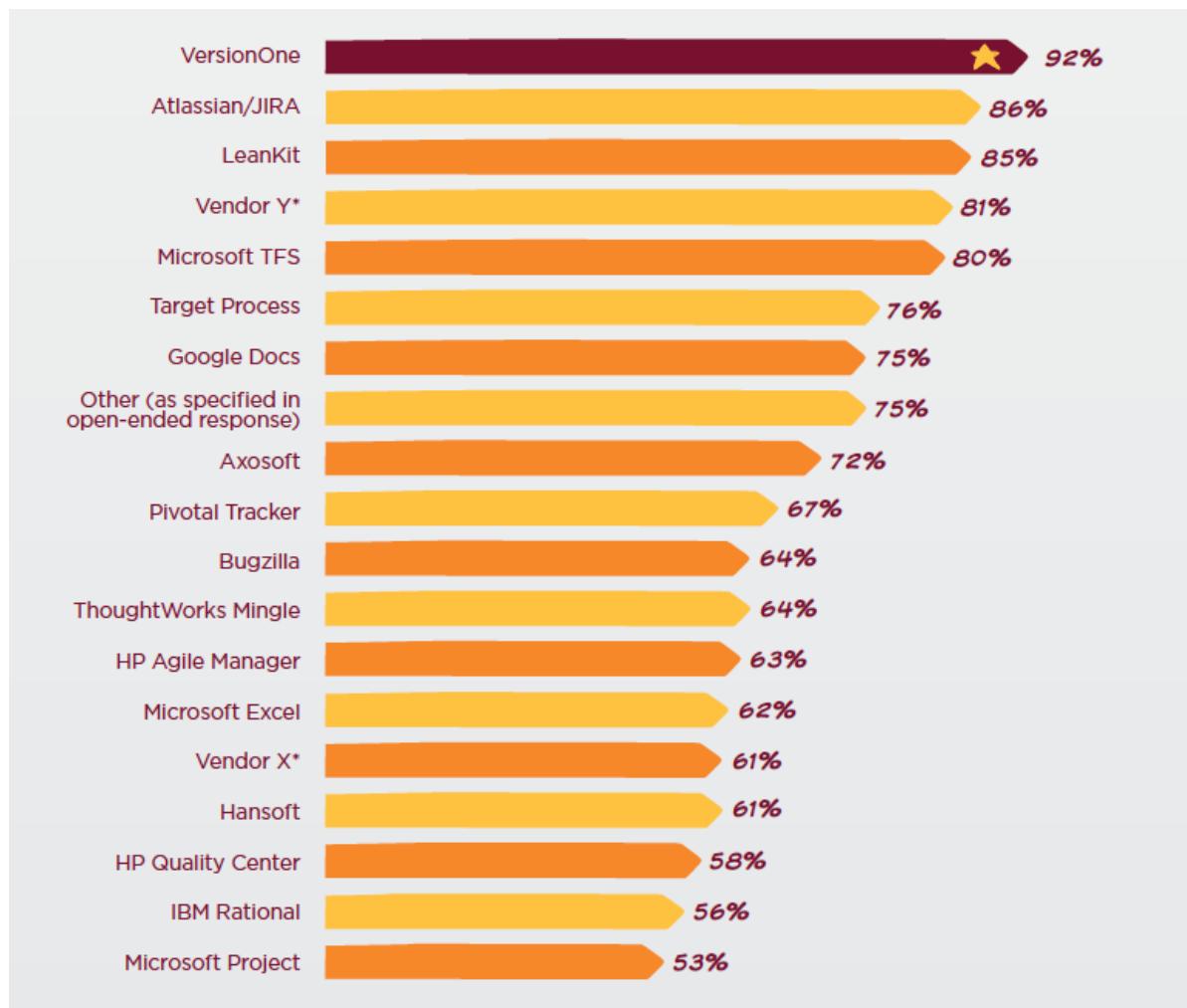
- Beck e altri, Manifesto for Agile development, 2001 [agilemanifesto.org](http://agilemanifesto.org)
- Beck & Andres, *Extreme Programming Explained: Embrace Change*, AW 2004
- Wilson, *Building software together*, <https://buildtogether.tech>

# Siti e blog

- Extreme Programming [www.extremeprogramming.org](http://www.extremeprogramming.org)
- Uncle Bob's blog [blog.cleancoder.com](http://blog.cleancoder.com)
- <https://refactoring.com>
- [xpl23.com/articles/](http://xpl23.com/articles/)

# Strumenti

- Trello/Taiga
- Slack/Mattermost
- Ant, XDoclet, JUnit, Cactus, Maven
- Git/GitLab
- Atlassian Jira
- agiletrack.net
- [www.planningpoker.com](http://www.planningpoker.com)
- [www.collab.net](http://www.collab.net)



Modelli di processo 2

# Domande?

