

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

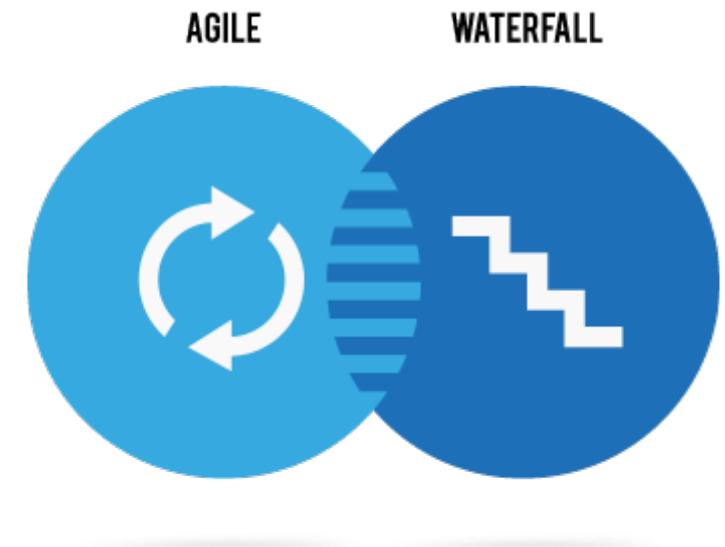
SS



*Prof. Paolo Ciancarini  
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)



Non è chiaro che cosa si vuole realizzare: richieste poco chiare o ambigue  
col tempo

## Certi sviluppi non si pianificano

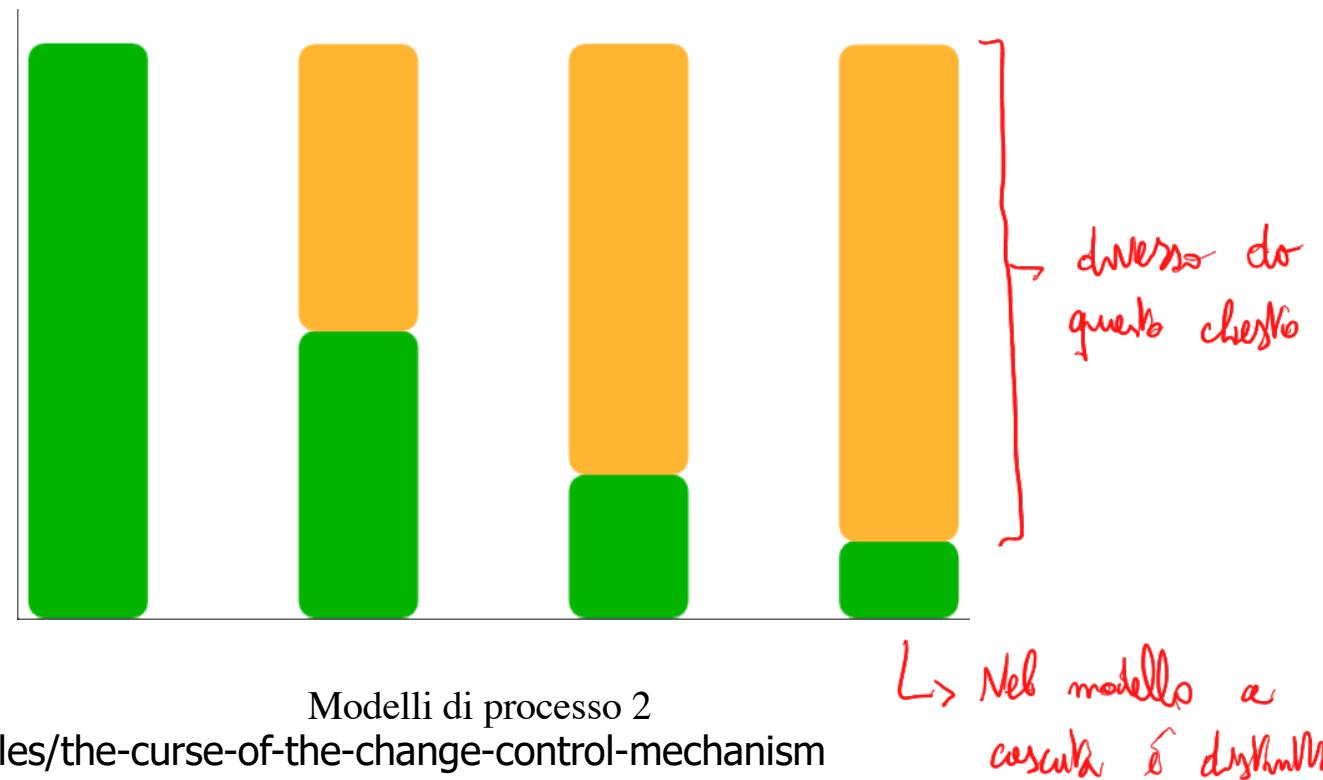
- Alcuni problemi sono complessi, richiedono il contributo di molte persone
- La soluzione all'inizio non è chiara
- I requisiti del prodotto-soluzione durante lo sviluppo probabilmente cambieranno
  - Normalissimo
  - incluso un pezzo
  - mano mano
  - ma dai feedback
- Il lavoro può essere articolato in incrementi
- È richiesta una collaborazione stretta e un feedback rapido e continuo dagli utenti finali

# Sviluppare software è un'attività sociale

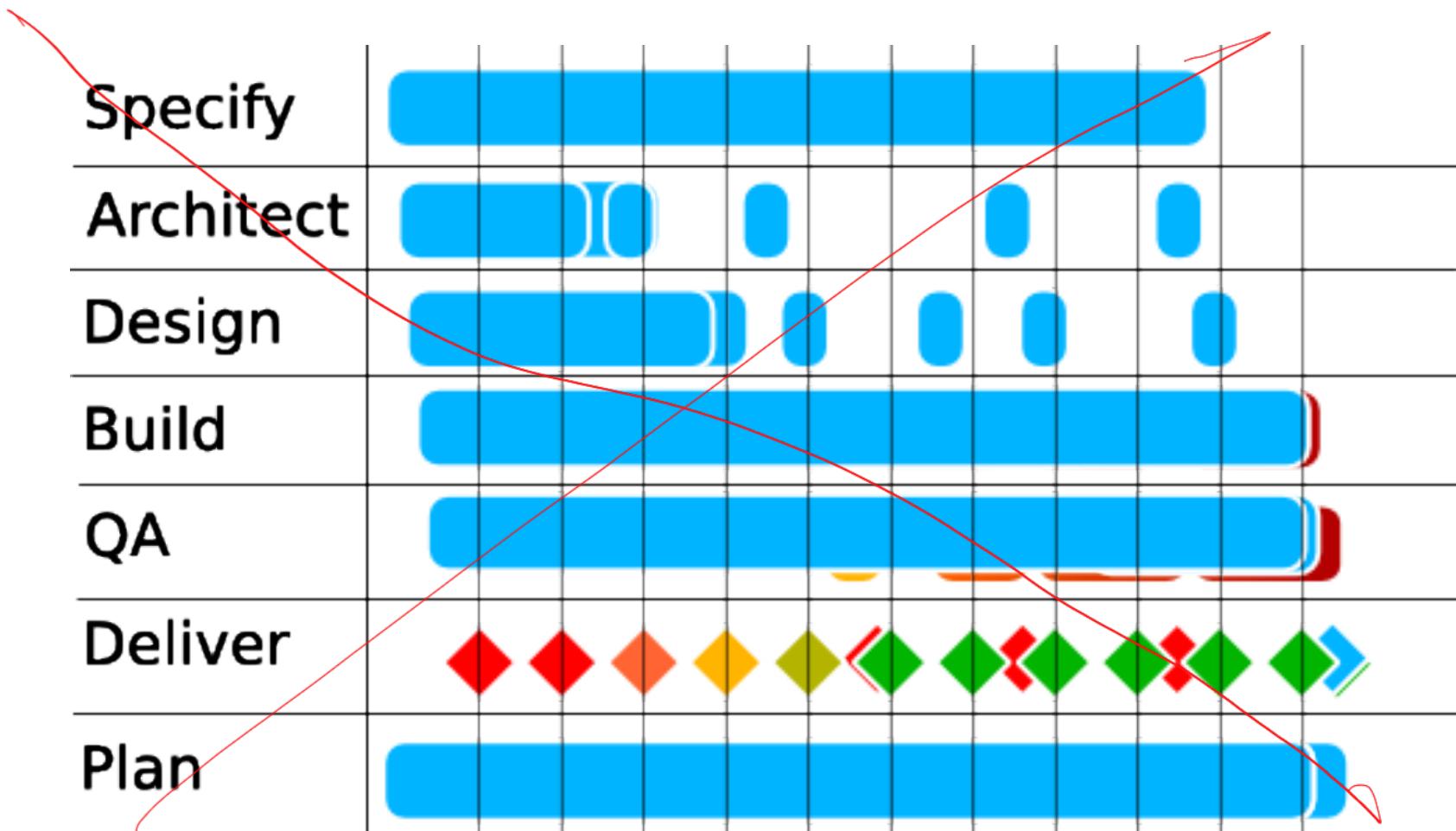
- Quasi tutti i problemi difficili nello sviluppo software riguardano le persone e non i computer.
- Scrivere software non è difficile. Scrivere il software giusto è molto difficile, perché occorre capire i bisogni delle persone, negoziare i tempi di consegna, distribuire i compiti, fare compromessi
- Le persone che lo fanno bene sono rare: quando hanno successo non è per l'abilità tecnica di programmazione che dimostrano, ma per le loro abilità relazionali interpersonali

# Volatilità dei requisiti

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



# L'interesse dell'utente



Modelli di processo 2

# Migliorare i processi iterativi

Modelli iterativi: prototipi vengono built-in quando ho requests chiavi

- 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 inutili di tempo

↳ Ciascuno dei releases è sempre utilizzabile

Non è quella finale, ma funziona. Con prototipi implementiamo alcune funzioni, se va bene lo build e ricomincio

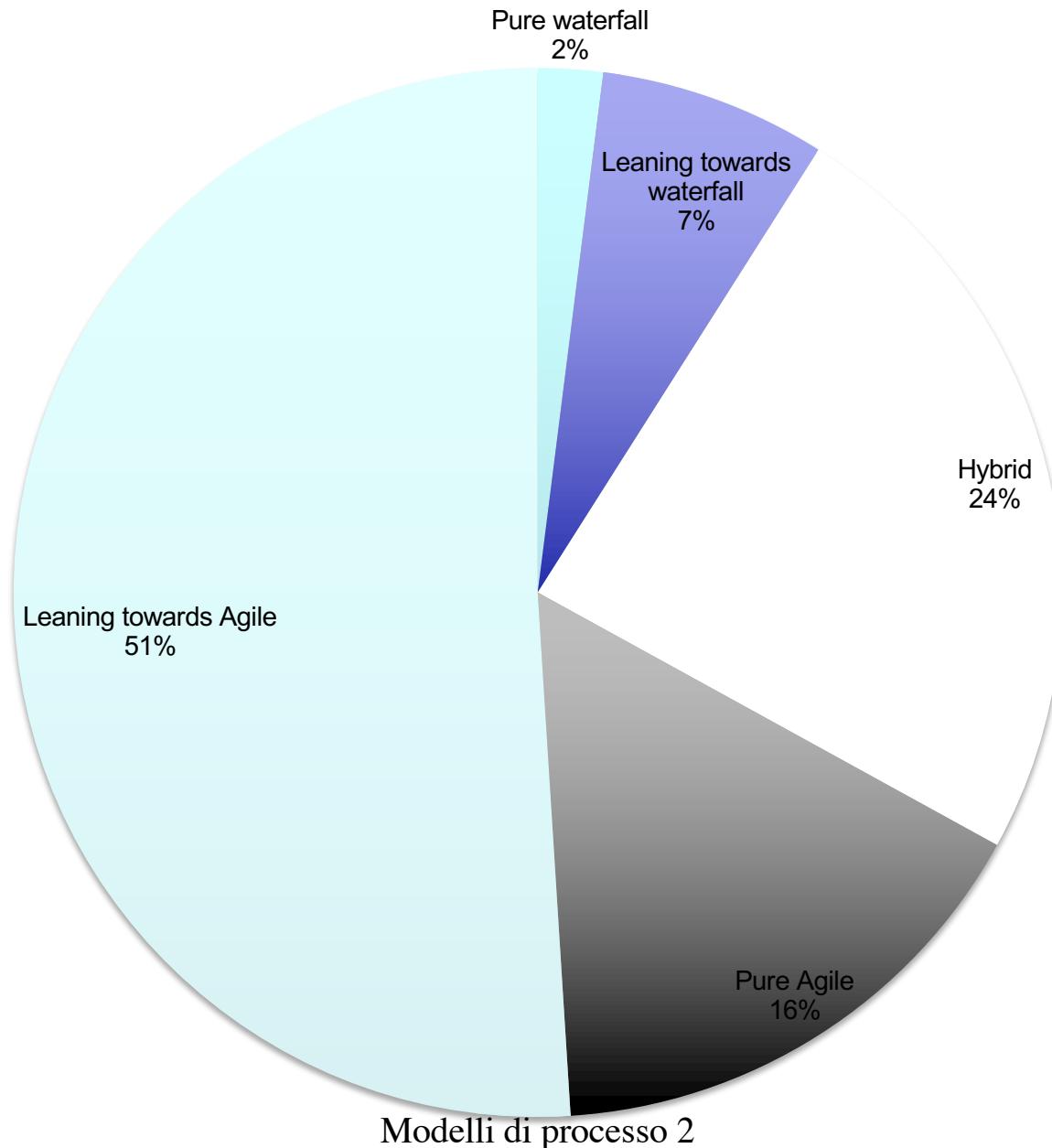
- [www.youtube.com/watch?v=Tj-lavaMkxU](https://www.youtube.com/watch?v=Tj-lavaMkxU)

# Metodi agili

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

## **Primary development method**

(source: 2015 HP survey of 601 sw professionals)



# Etica del Movimento Agile

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

→ Team, componenti e comunicazione

Modelli lineari

Individui e interazioni più che a processi e strumenti

Software che funziona<sup>4</sup> più che a documentazione completa documents

Collaborazione col cliente più che a negoziazione contrattuale

Reagire al cambiamento più che a seguire un piano

→ Ne ho chiesti 100 e 100 le ne faccio

\* Preferisco software da utilizzare che documentaz. complessissima

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

→ Più importanza a implementaz. che design

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

Modelli di processo 2

→ Perché il modello combina tanto da release a release

# 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** *più semplice è meglio è*
12. I team di sviluppo valutano la propria efficacia ad intervalli regolari e modificano di conseguenza il proprio comportamento
13. **Release, Sviluppo, architettura ecc: non imponere la forma al Team**

# 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:

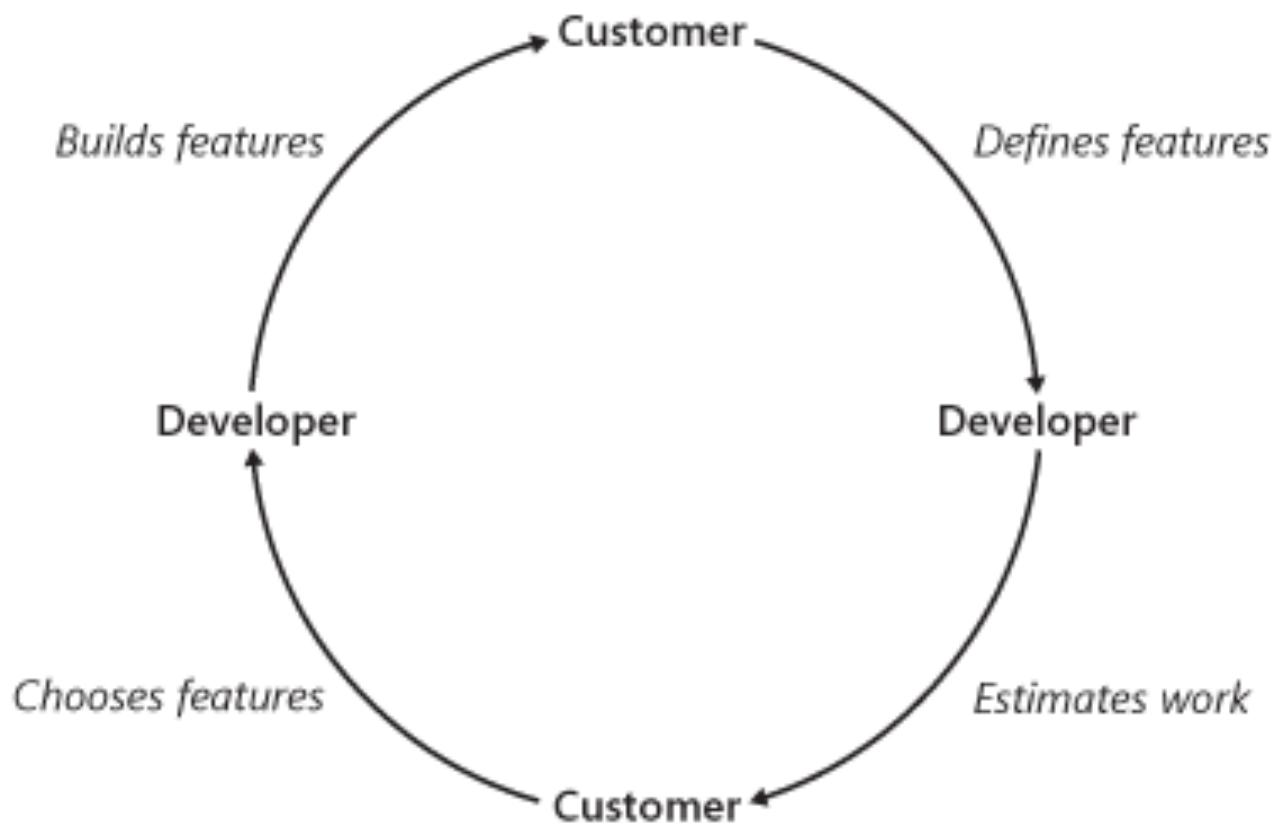
- **Rilasci frequenti** del prodotto sviluppato
- **Collaborazione continua** del team di progetto col **cliente**
- **Documentazione** di sviluppo **ridotta**
- **Valutazione** sistematica e continua di **valori e rischi** dei **cambiamenti**

# Lavorare per progetti vs lavorare per prodotti

	Progetto	Prodotto
Scope	Funzioni decise all'inizio	Lista di funzioni "correnti" in priorità
Agenda dei rilasci	Durata fissa	Rilasci multipli
Budget	Allocato all'inizio	Ciclico



# La base della collaborazione agile in XP



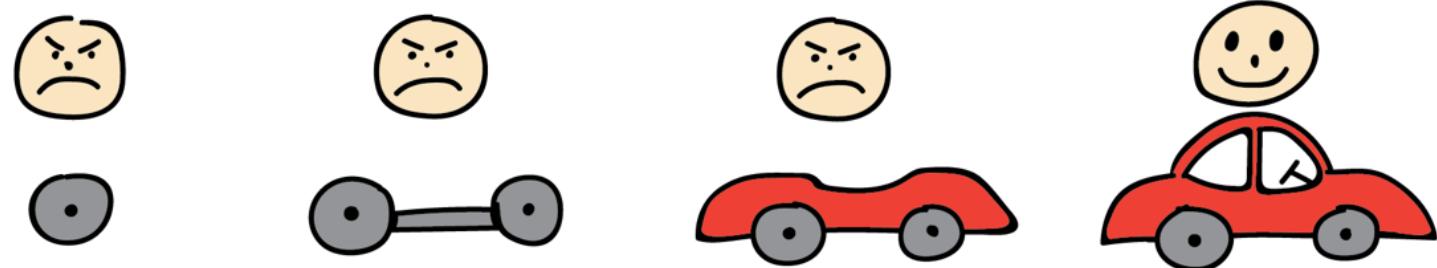
# Minimal viable product MVP

→ Considerare i rischi di sviluppo, quali sono le funzioni che devono realizzate per ottenere il massimo ritorno possibile.

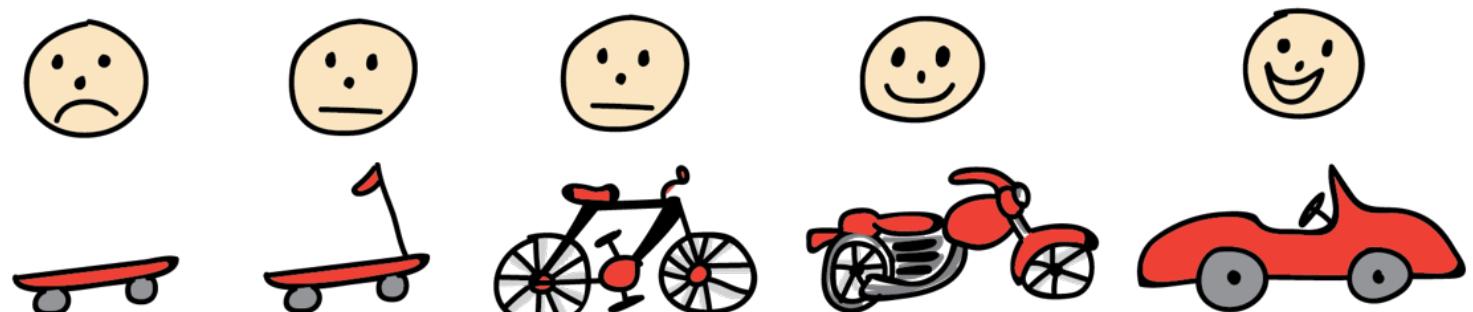
- Nello sviluppo di un nuovo prodotto, il prodotto minimo funzionante (**Minimum Viable Product "MVP"**) è il prodotto con il più alto ritorno sugli investimenti rispetto al rischio
- È una strategia mirata ad evitare di costruire prodotti che i clienti non vogliono, che cerca di massimizzare le informazioni apprese sul cliente per ogni euro speso. → es. carrello che non fa pagare
- Un MVP non è, quindi, un prodotto minimo, ma un processo iterativo di generazione di idee, prototipazione, presentazione, raccolta dati, analisi ed apprendimento.
- MVP è un fondamento del movimento “Lean”, alla base di Agile

# Minimal Viable Product (prodotto minimo funzionante)

Not Like  
This!



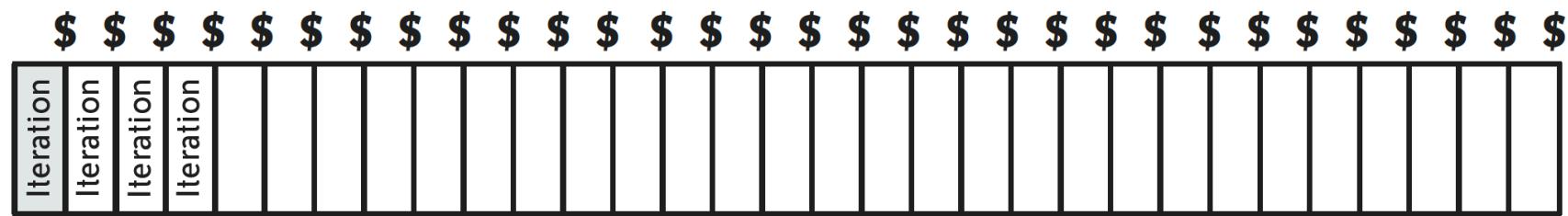
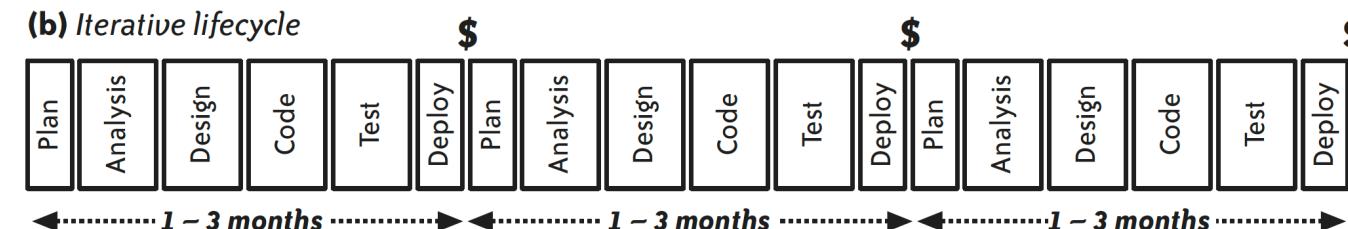
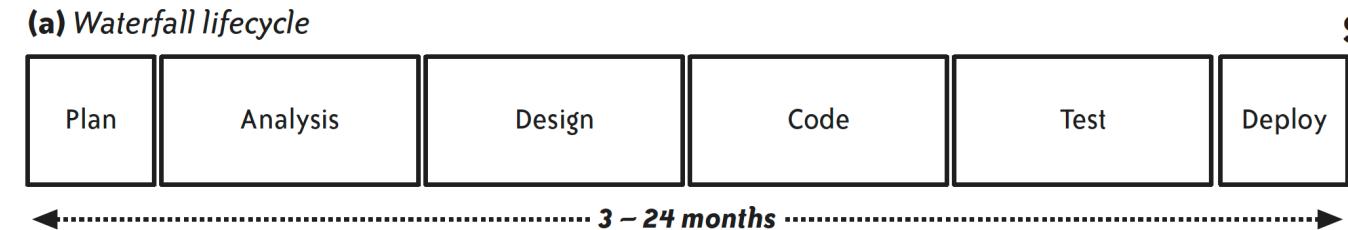
Like This!



↑  
Content  
der Anforderungen  
nur passendes Content

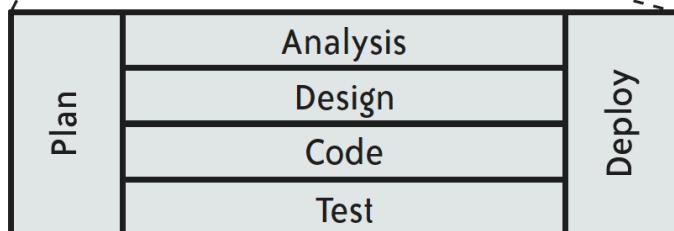
Modelli di processo 2

# Waterfall vs iterativo vs agile



Modello agile ha release continue (stretto)  $\$ =$  Potential release

differenze: release del software funzionante di continuo  
che puoi installare a usare

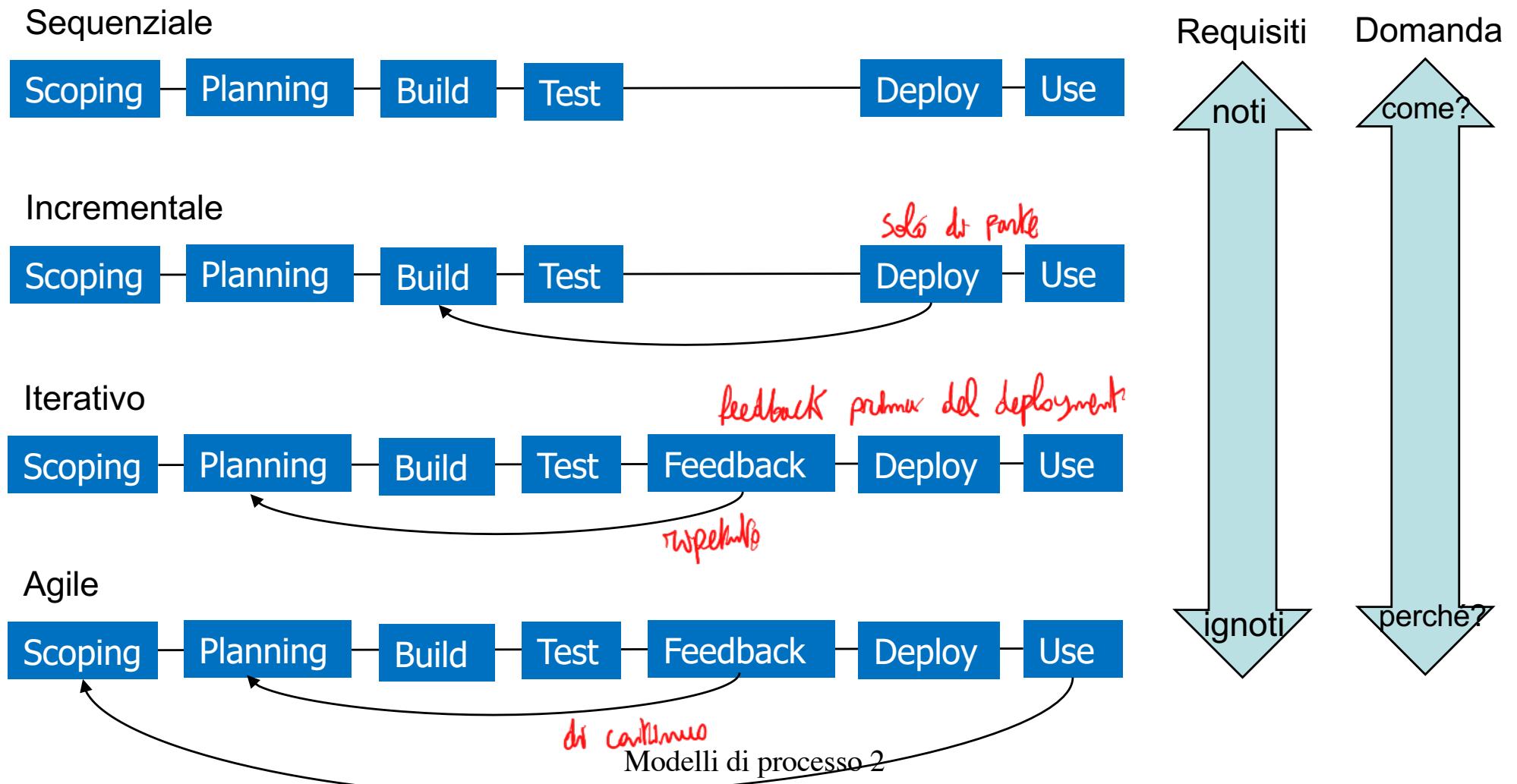


...1 week →

⇒ Documentar, a livello più elevato

• devo parlarti una breve cosa: sviluppo in quella settimana

# La diversità dei modelli di processo



# eXtreme Programming (XP)

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

- semplicità,
- comunicazione, *Tutti sanno di tutto*
- feedback, e *con lo user*
- coraggio”. *a seguire l'utente*



Kent Beck

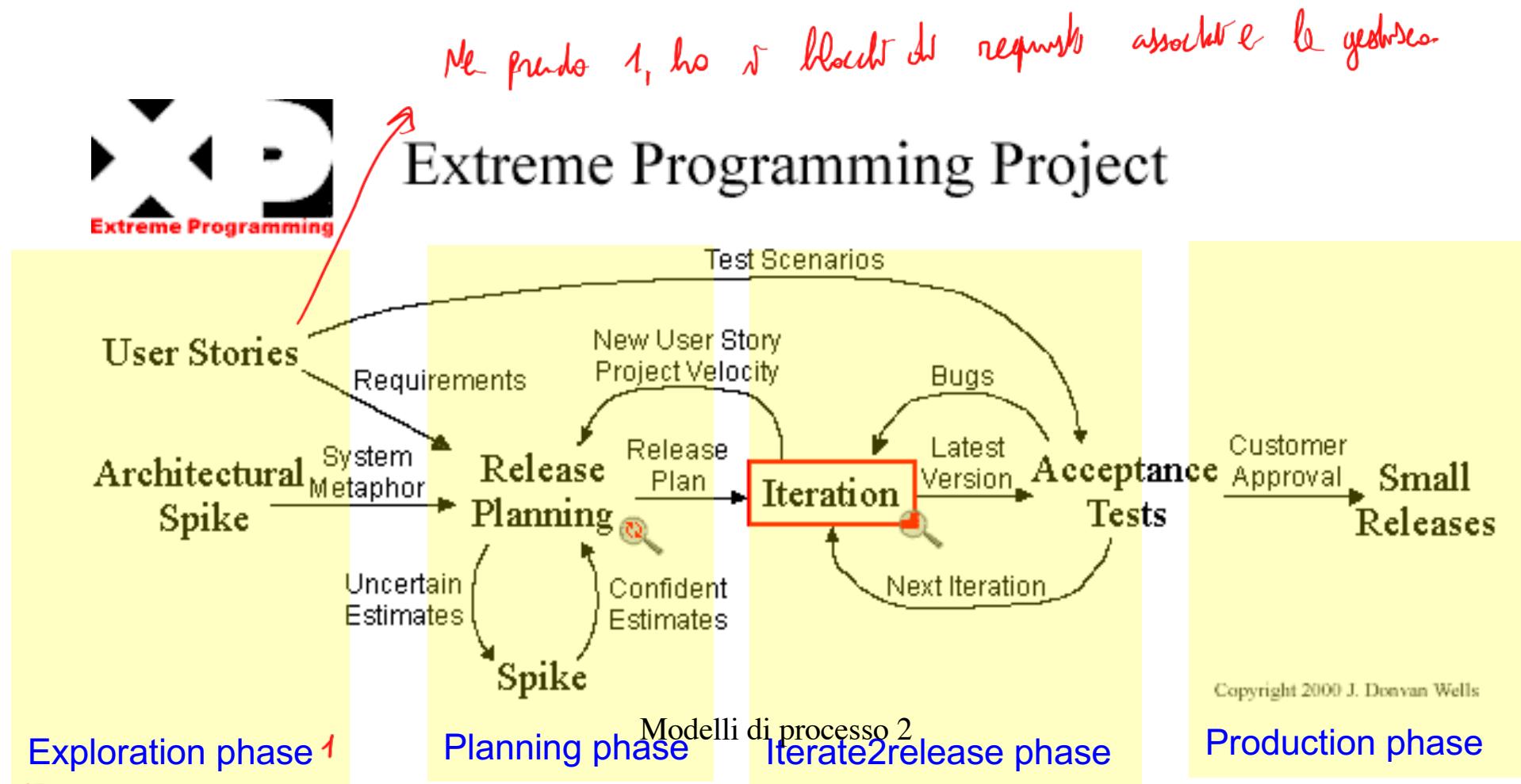
# Il team XP

→ possono avere anche più team

- Il team di sviluppo, di solito costituito da meno di dieci persone, è riunito nello stesso locale
  - comunicazione immediata
- E' sempre presente un rappresentante del cliente, capace di rispondere a domande del team riguardo ai requisiti
  - Obbligatorio: devono sapere se vado nella direz. questa
- 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

# eXtreme Programming (XP)

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



1) esplora requisiti da sviluppare: decido cosa andare a realizzare.

User Stories: "il cliente vuole poter compiere n. bigheggi"

Sparkle: elevata produttività: questo decido i requisiti deciso un'architettura generica del sistema

definiscono requisiti e scenario di test  $\Rightarrow$  Devo creare test che ottimizzino che sistema fa ciò che deve

2) Release planning: nella release immediata vedo cosa devo produrre, con delle stime incerte per capire cosa.

Sparkle: decido cose fare, ma poi si continua fino a quanto ho promesso cosa devo realizzare.

3) Iterazione: design, implementaz, test ecc.

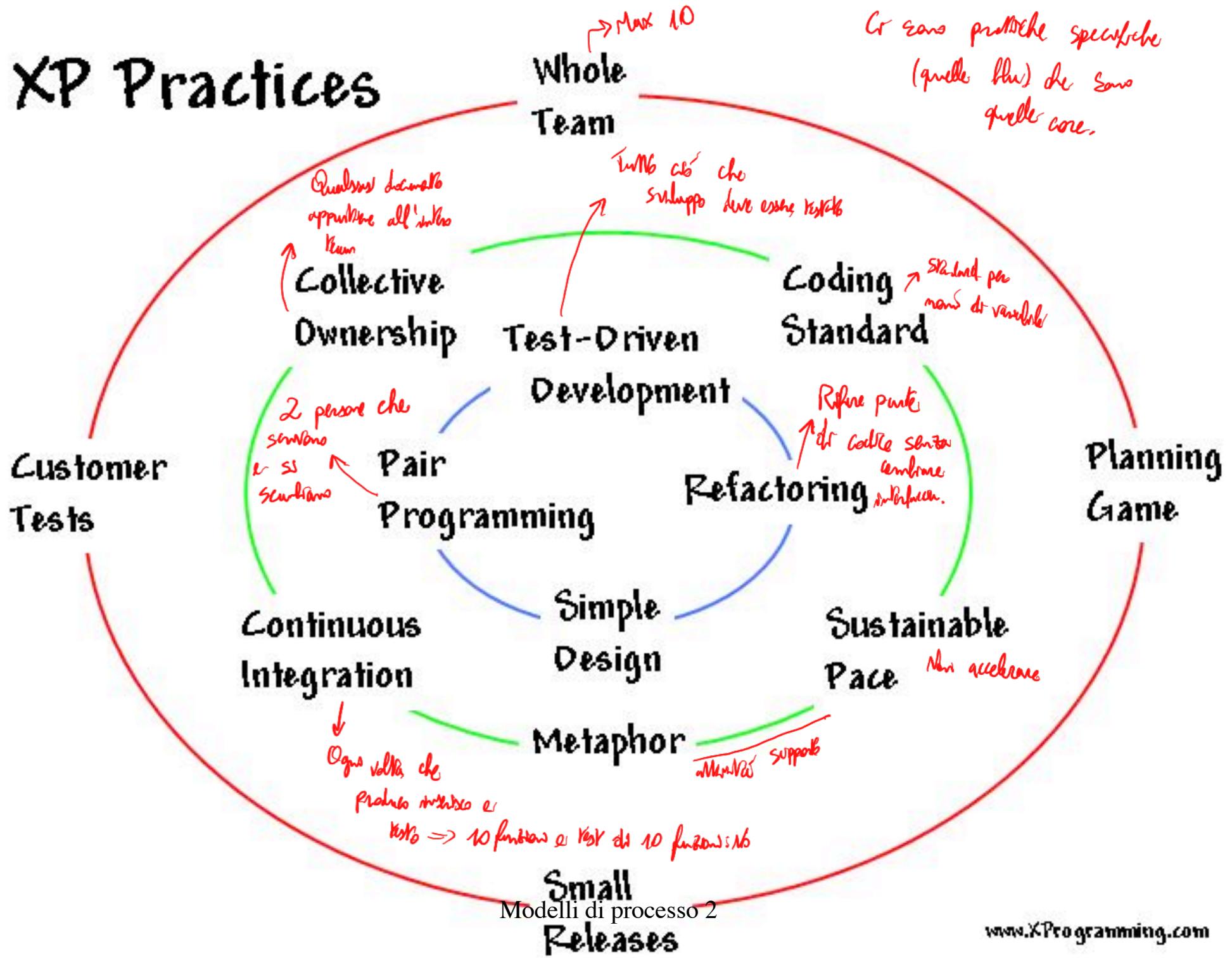
frecce multiuso: buy, aprire tempi struts e sviluppo cookie succedono

Torno al release planning: per sviluppare altro mi servono altre user stories, mi servono altri requisiti.

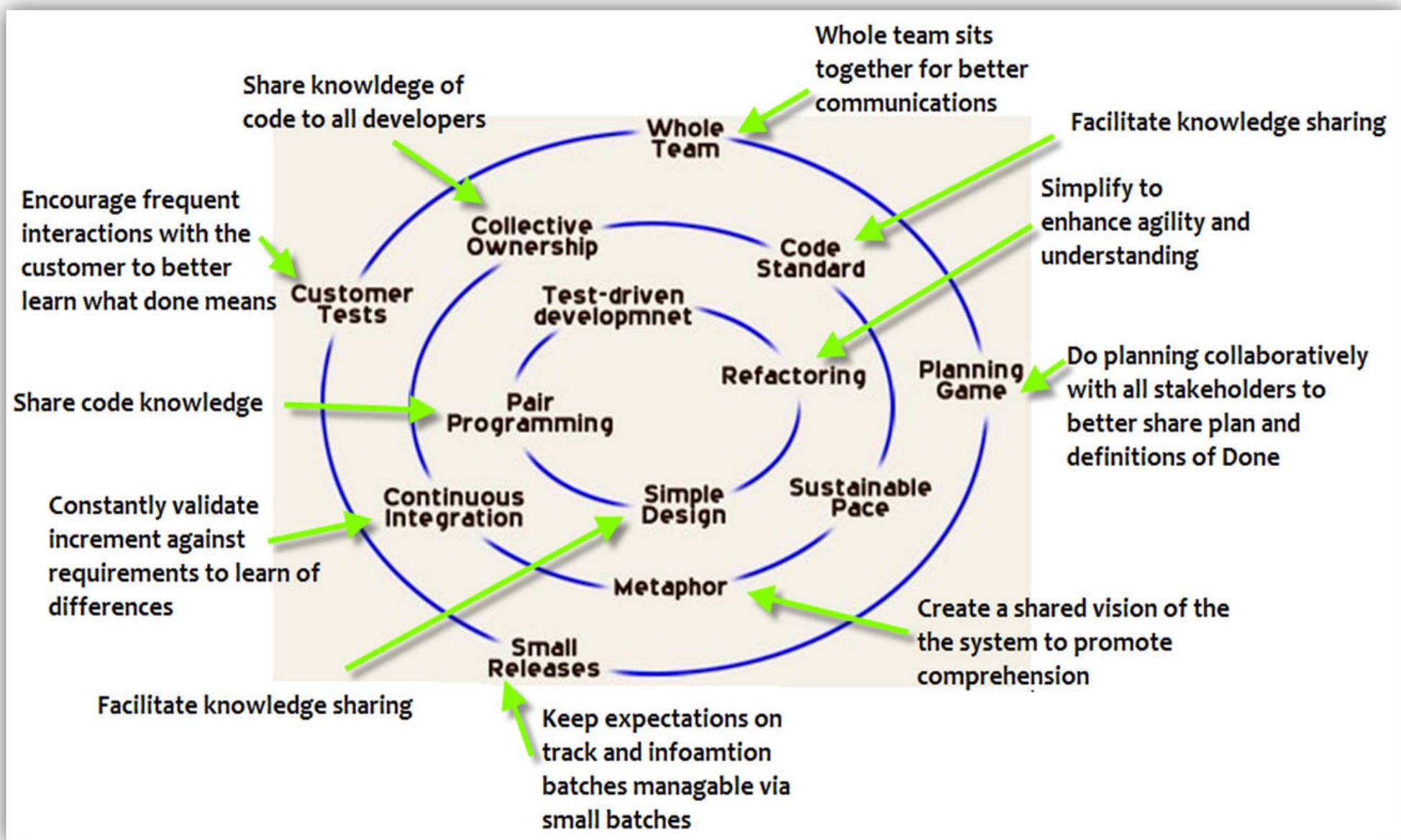
Velocity: impiego sui dati di velocità.

Acceptance test: approvaz. da chiave & risultato. Offro una baseline

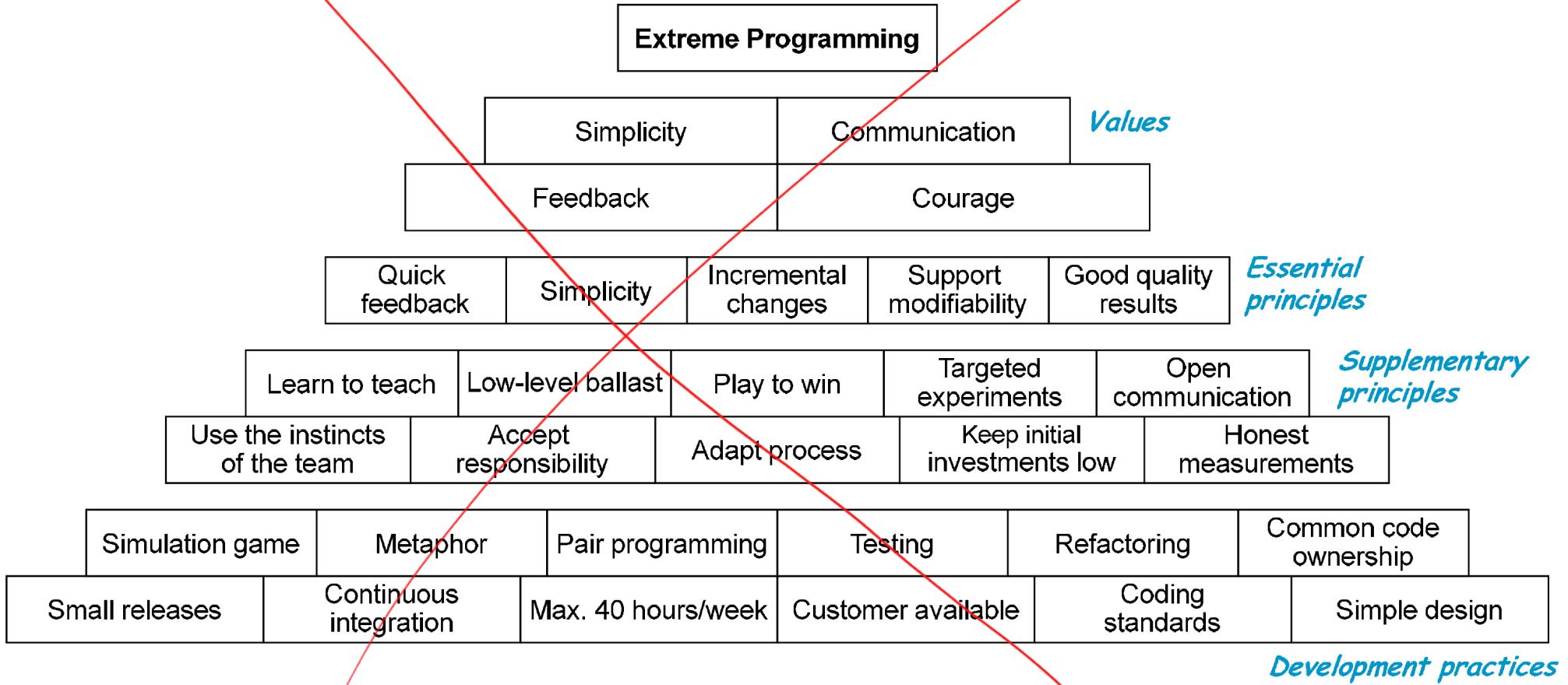
# XP Practices



# Knowledge Sharing in XP:



# Valori, principi, pratiche agili in XP



# 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:
  - usate al posto di documenti dettagliati di specifica dei requisiti
  - scritte dai clienti: cosa si aspettano dal sistema
  - una storia è descritta da una o due frasi in testo naturale, con la terminologia del cliente (*no techno-syntax*)
  - utili per stimare i tempi/costi di un rilascio (*release planning*)

*Descriz. di qualcosa che fa fare le storie al sistema*

# Esempi di user stories

*rappresenta intenz. tra utente e sistema*

- Students can purchase monthly parking passes online
- Parking passes can be paid via credit cards.
- Parking passes can be paid via PayPal
- Professors can input student marks
- Students can obtain their current seminar schedule
- Students can order official transcripts
- Students can only enroll in seminars for which they have prerequisites
- Transcripts will be available online via a standard browser

# Esempio di scheda per user story

ID

173.

Students can purchase parking passes.

→ questo valore ha user point Ha alta priorità, stima per i rischi bassa.

Priority: 8

Estimate: 4

Importanza

↳ Costo di sviluppo

Story points

Modelli di processo 2

# Il “Planning game”

- I **requisiti** vengono organizzati in “**User Stories**”
  - 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** (*costo* o *ore di lavoro*)
- “The **Planning Game**”
  - Le **storie** di più alto rischio e priorità sono affrontate per prime, in incrementi “*time boxed*”
- Il Planning Game si **rigioca** dopo ciascun **incremento**

# Storie e iterazioni



# 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
- Gli sviluppatori non devono attendere le decisioni del cliente
- La comunicazione “faccia a faccia” minimizza la possibilità di ambiguità ed equivoci

# 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)  
↳ importanti ma comunque a mano
  - Deve girare tutto prima di proseguire con altro codice
- 
- Unit test vs acceptance test

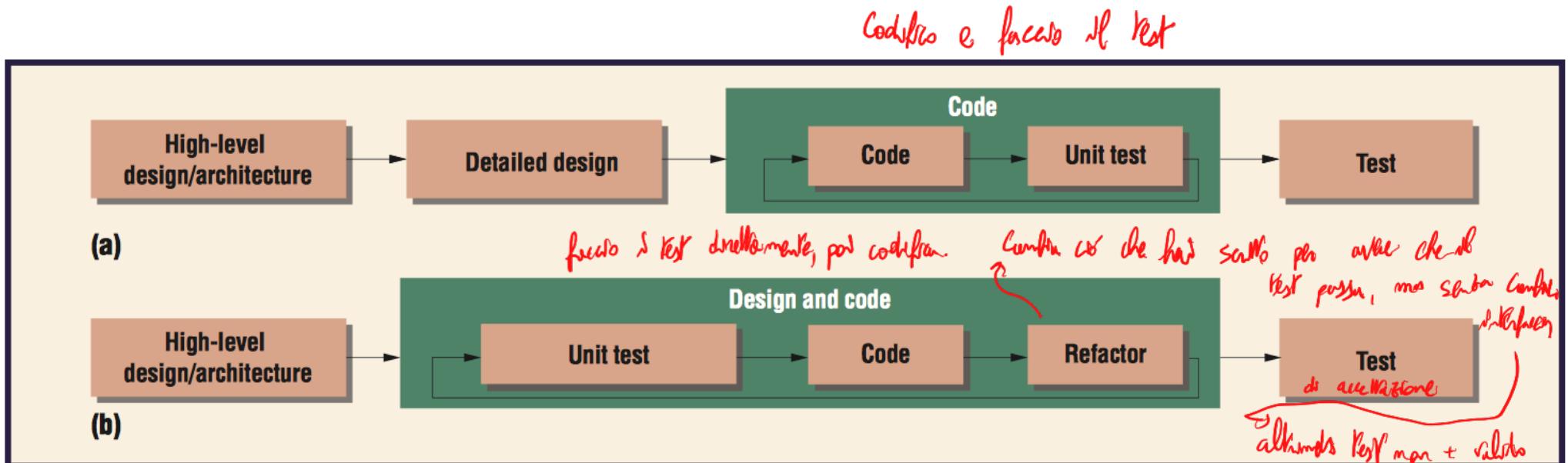
All'utente i test falliscono perché non c'è implementazione delle funzioni.

→ chiave per il Test

Prima ancora di mettere mano  
al codice, costruire una funzione  
di test



# Test classico vs test driven



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

# 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

Modelli di processo 2

Esempio di test scritto sul retro della user story

# Piccoli rilasci

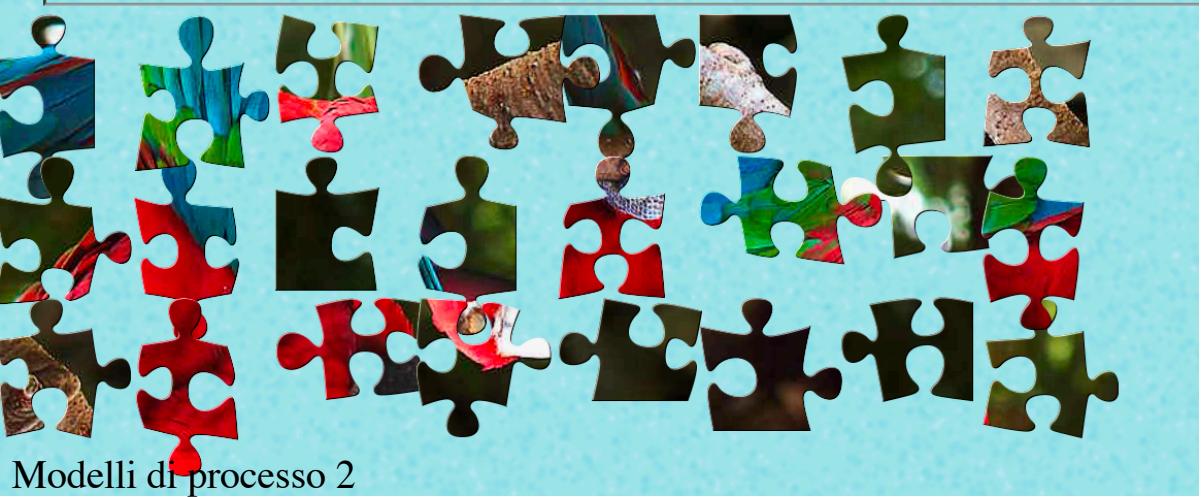
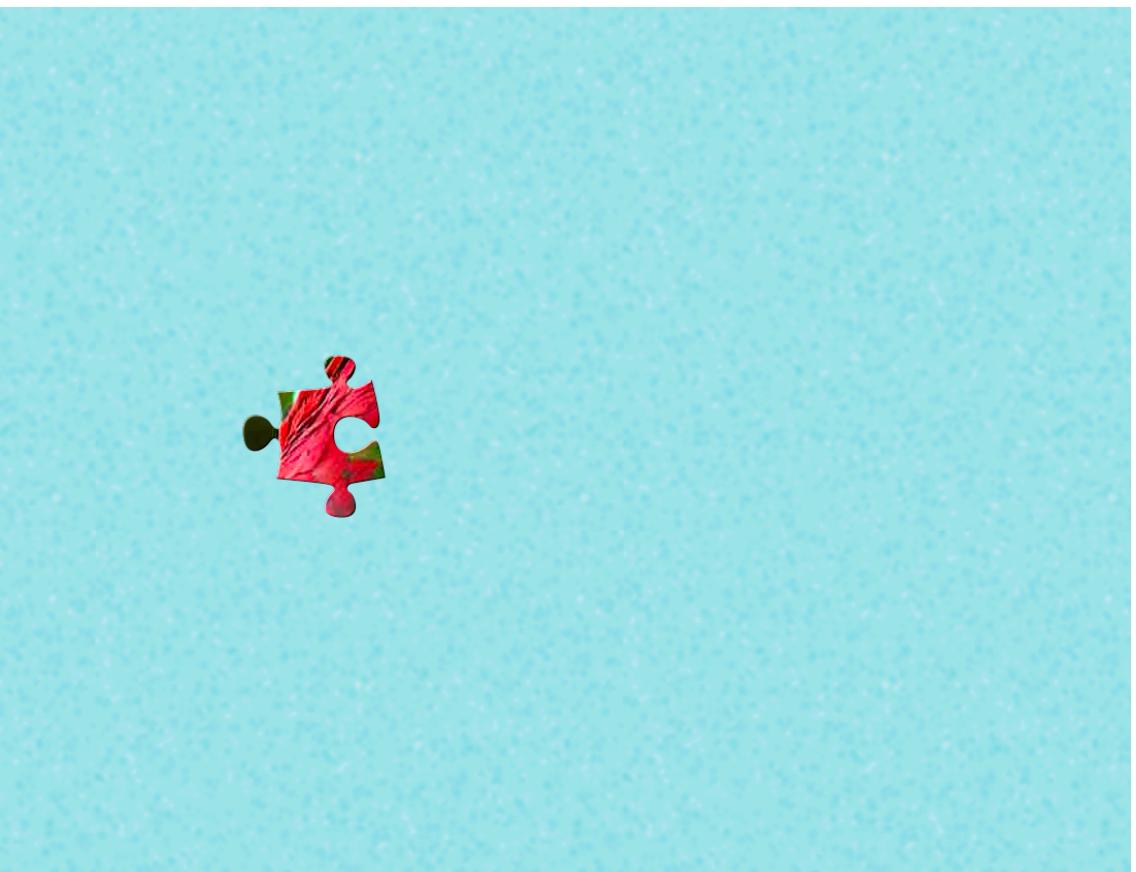
- Timeboxed (ovvero di durata “breve” prefissata)
- Minimali, ma comunque utili (**microincrementi**)
  - Mai cose come ‘implementare il database’
- Ottener feedback dal cliente presto e spesso
- Eseguire il “planning game” dopo ciascuna iterazione
  - Si voleva qualcosa di diverso?
  - Le priorità sono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

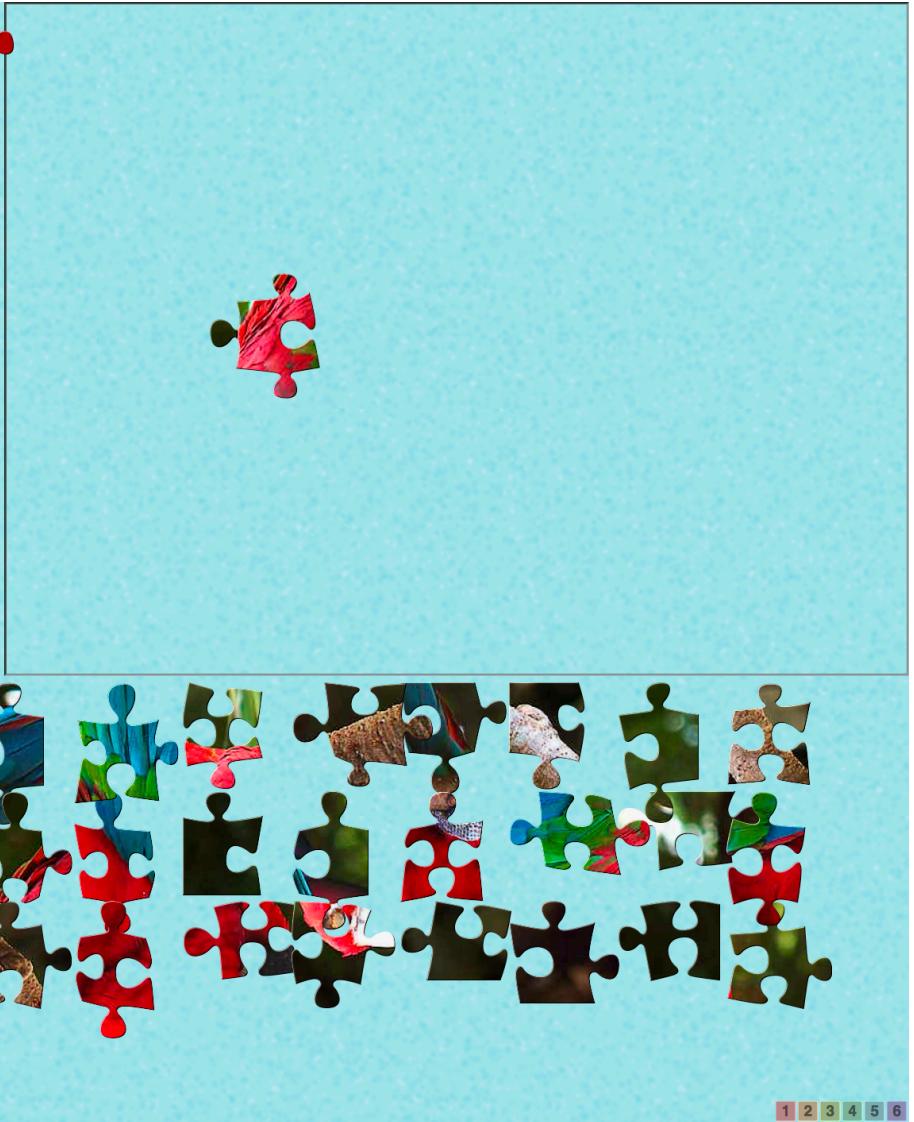
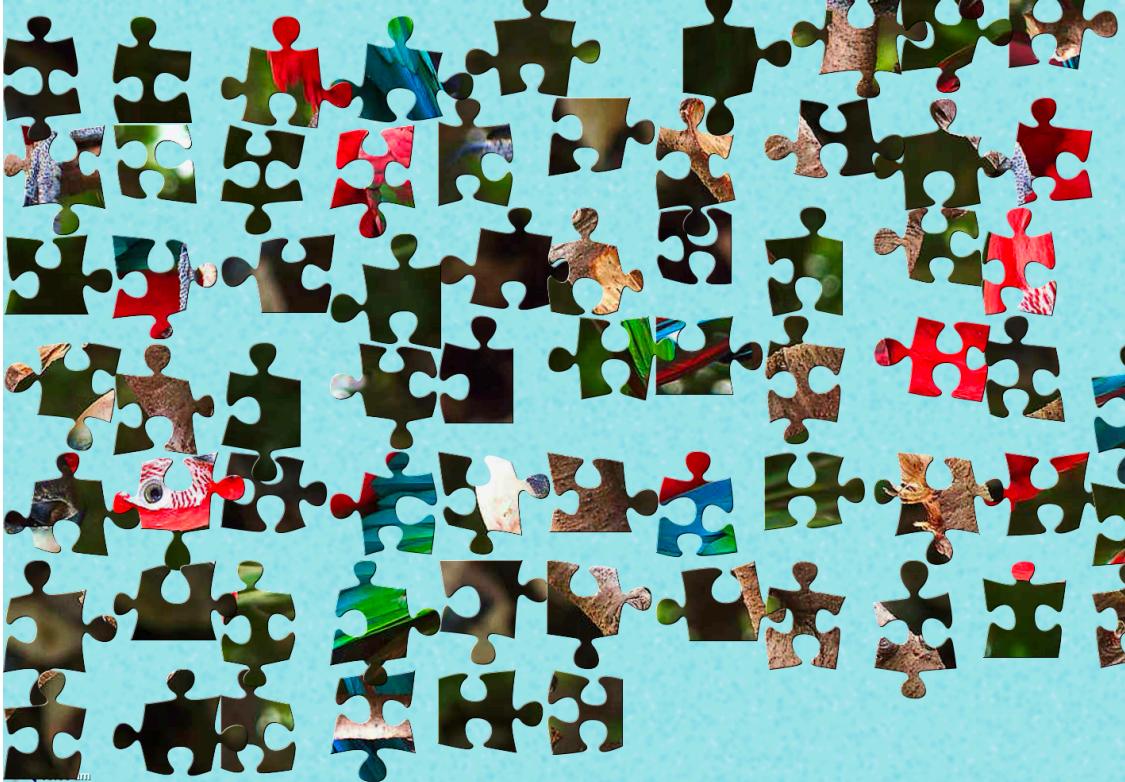
# La metafora

- 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



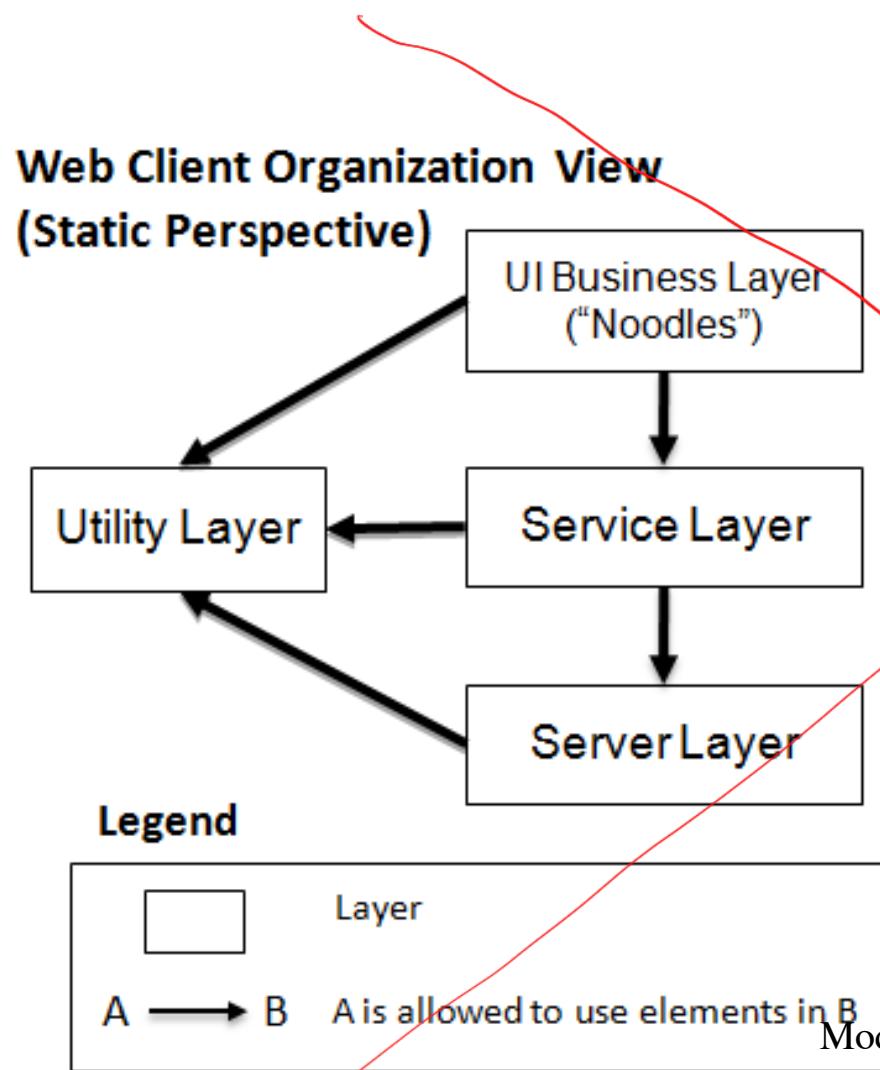
Modelli di processo 2





Modelli di processo 2

# Esempio di metafora



Fonte: neverletdown.net/topics/architecture/



The “Bento Box”

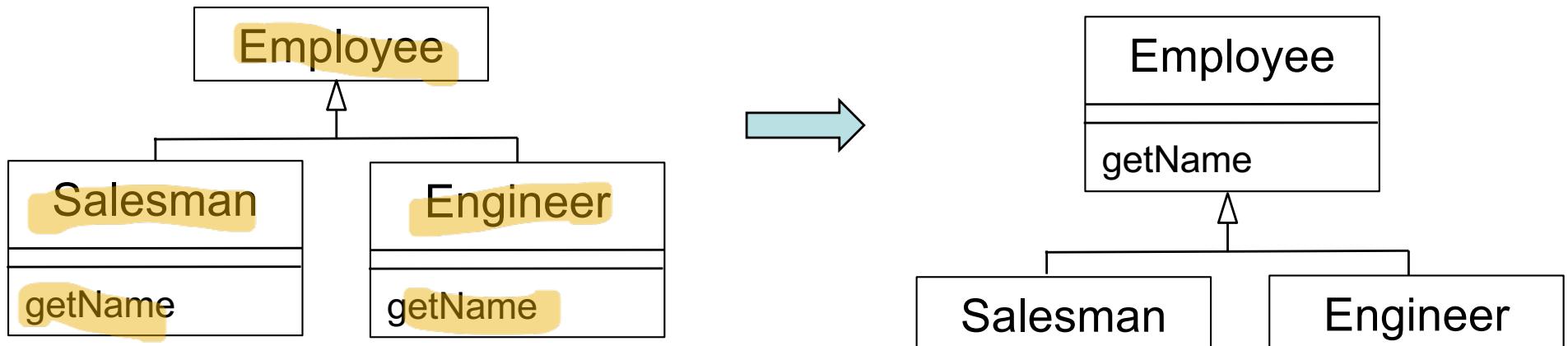
# Progettare in modo semplice

- **No Big Design Up Front** (BDUF) *Non fare subito un design bHOME dettagliato*
- “Fare la cosa più semplice che possa funzionare”
  - **Includere la documentazione**
- “**You Aren’t Gonna Need It**” (YAGNI) *Cose non richieste*
- **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	4
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)

*Problema: Se codice è molto semplice perdo tempo utile che gurbegno.*

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

*Midea: non 8 ore di lavoro sul progetto, ma 1 o 2 per fare crescita personale*

# Codifica

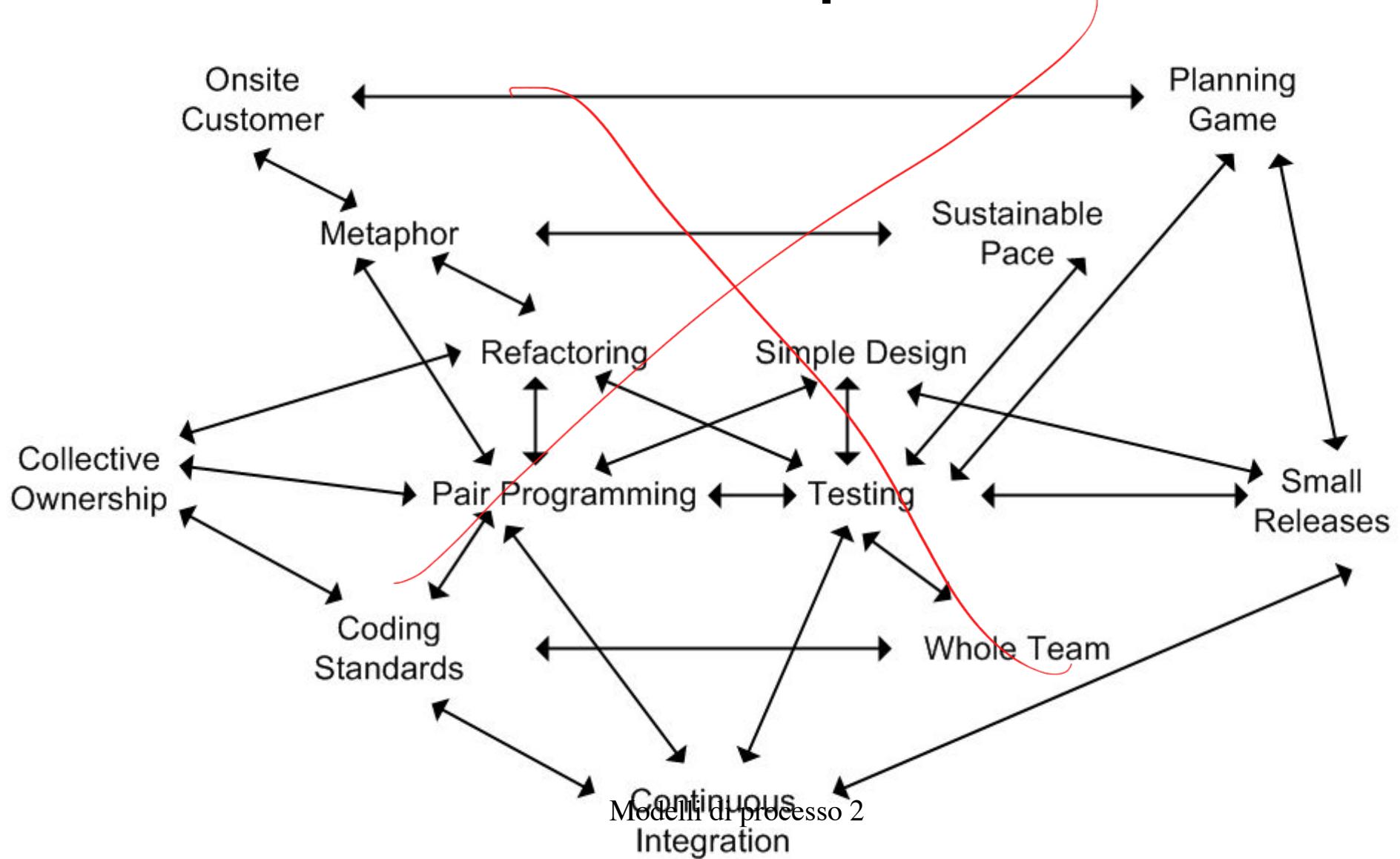
- Usare convenzioni di codifica
  - A causa del Pair Programming, delle rifattorizzazioni e della proprietà collettiva del codice, occorre poter addentrarsi velocemente nel codice altrui
- Commentare il codice
  - Priorità al codice che “svela” il suo scopo
    - Se il codice richiede un commento, riscrivilo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

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

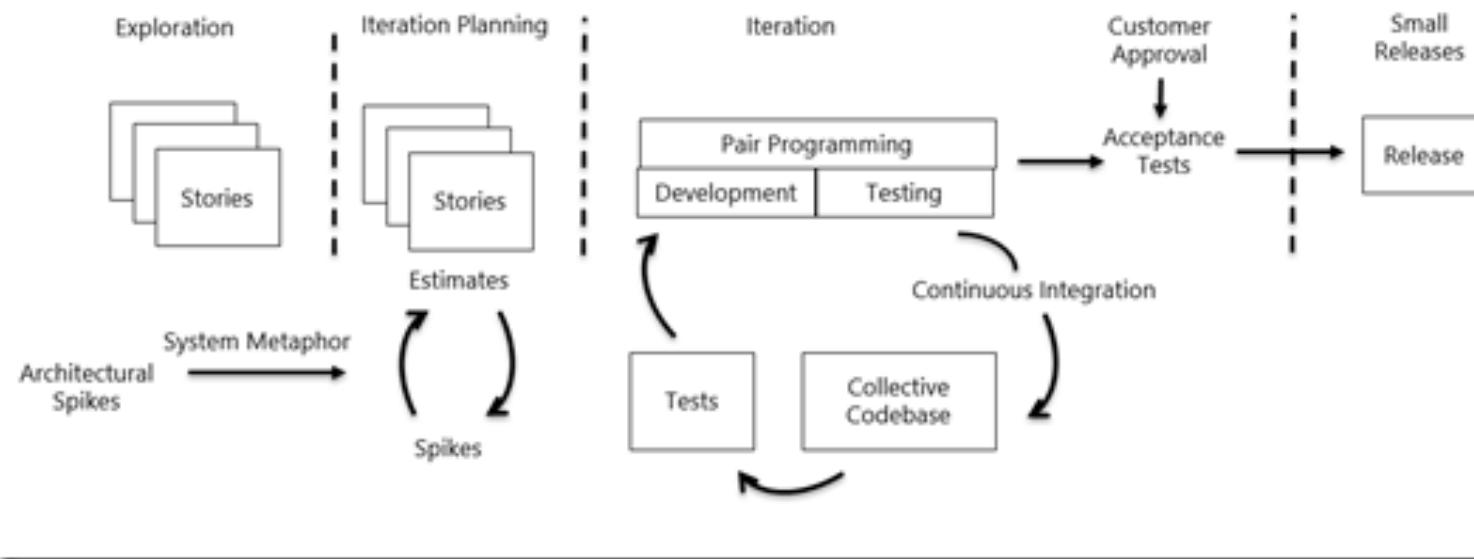
<http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsent7/html7vxconcodingstandardscodereviews.asp>

# Relazioni tra le pratiche XP



## Extreme Programming (XP) at a Glance



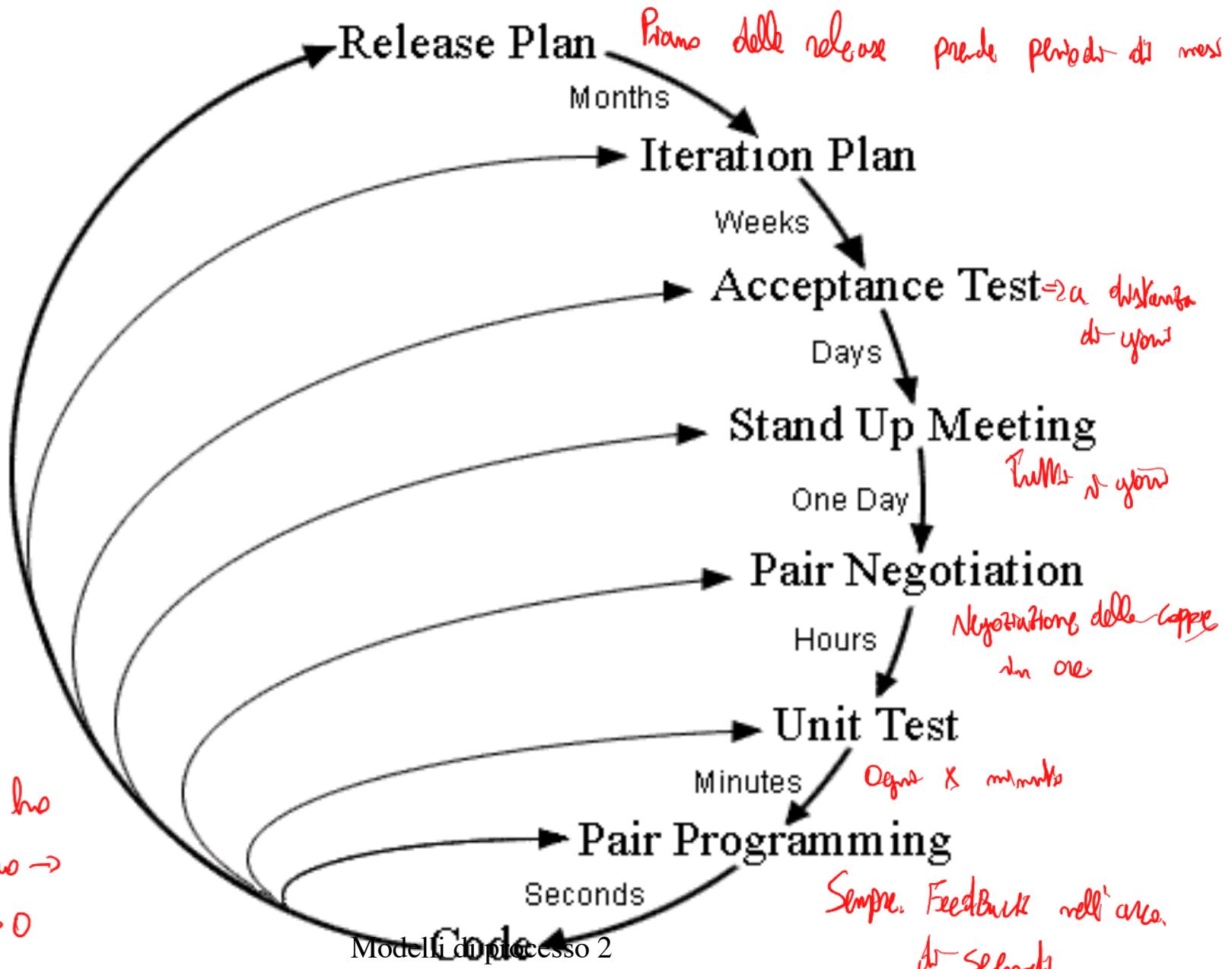
# Il 13º Principio: La riunione in piedi

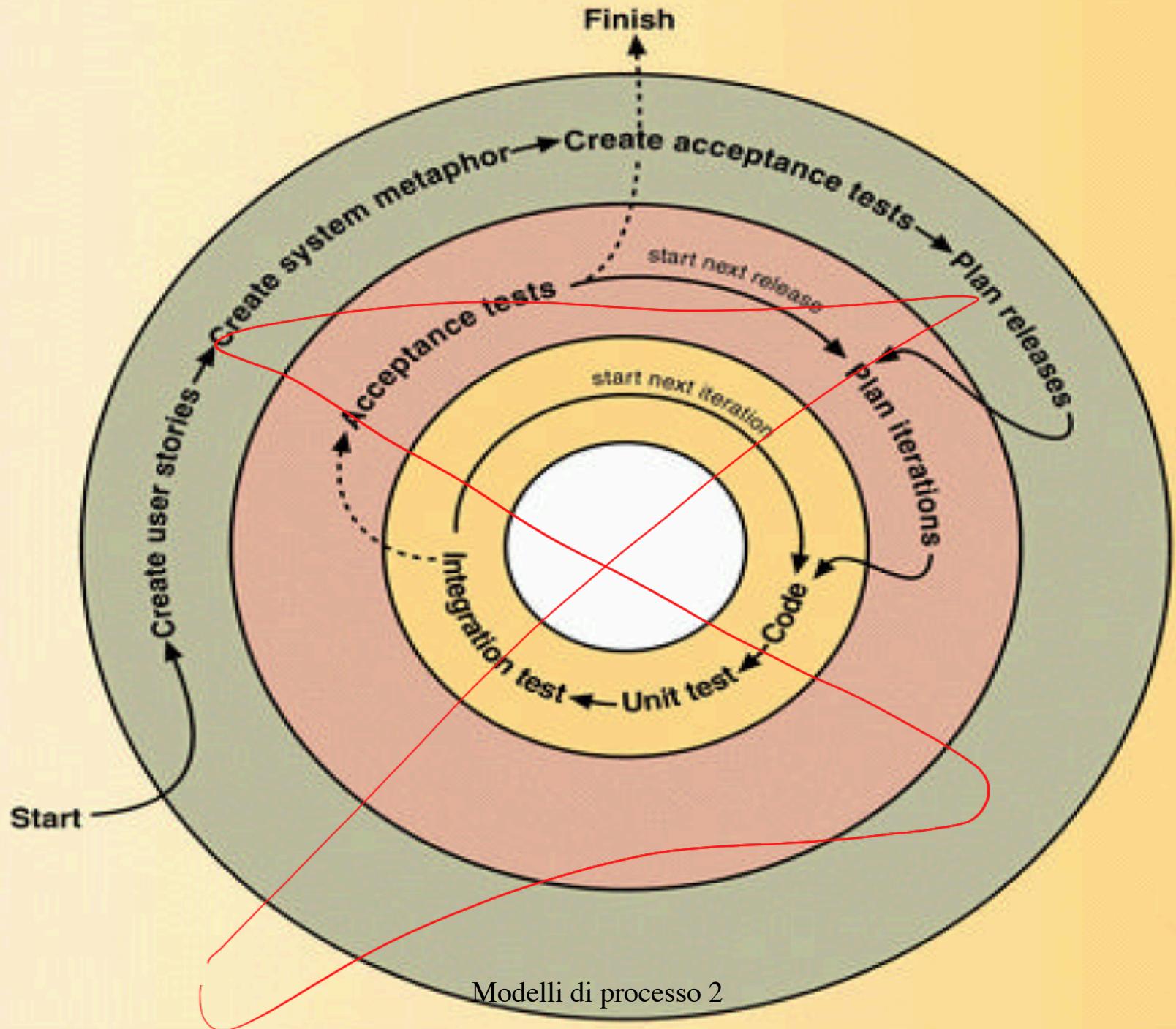
Modello agile sempre  
con questo rhincone

- 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

# Planning/Feedback Loops

Ciclo feedback





Modelli di processo 2

# Rischi del processo XP

- Il codice non viene testato completamente
- Il team produce pochi test utili per il cliente
- Il cliente non aiuta a testare il sistema
- I test “non funzionano” prima dell’integrazione
- I test sono troppo lenti
- Le storie sono troppo complicate *errore all'inizio per clienti*
- Il sistema è troppo difettoso *ma escono fiori di bug (punto 1)*
- QA vuole il documento “Specifiche dei requisiti”
- Il manager vuole la documentazione di sistema
- Il team è sovraccarico di compiti *↳ difficilmente si svilupperà*
- Il team deve affrontare scelte tecniche rischiose
- Alcuni membri (cowboy coders) ignorano il processo del team

→ e poi i test sono su valutare che sopravviva.  
è andata bene

Scegliere come testare le funzioni è  
complicatissimo.

es. alcuni test dipendono  
da altro. Es. funz. che visualizza  
i dati del database

documentazione

# Valutazioni dell'approccio XP

- Dall'industria abbiamo supporto anedottico "forte"
  - "Produciamo sw quasi senza difetti in metà tempo"
- Studi empirici
  - Le coppie producono codice di miglior qualità
    - 15% in più di casi di test superati (ovvero 15% meno errori)
  - Le coppie completano i loro compiti con minor sforzo individuale
    - Sforzo solo 15% in più (non 100%)
  - Molti programmatore all'inizio sono riluttanti a lavorare in coppia
    - Ma poi le coppie apprezzano di più il loro lavoro (92%)
    - Le coppie hanno più fiducia nei loro prodotti (96%)

→ "devo seguire questo schema, ma le varie componenti possono essere realizzate in agile."

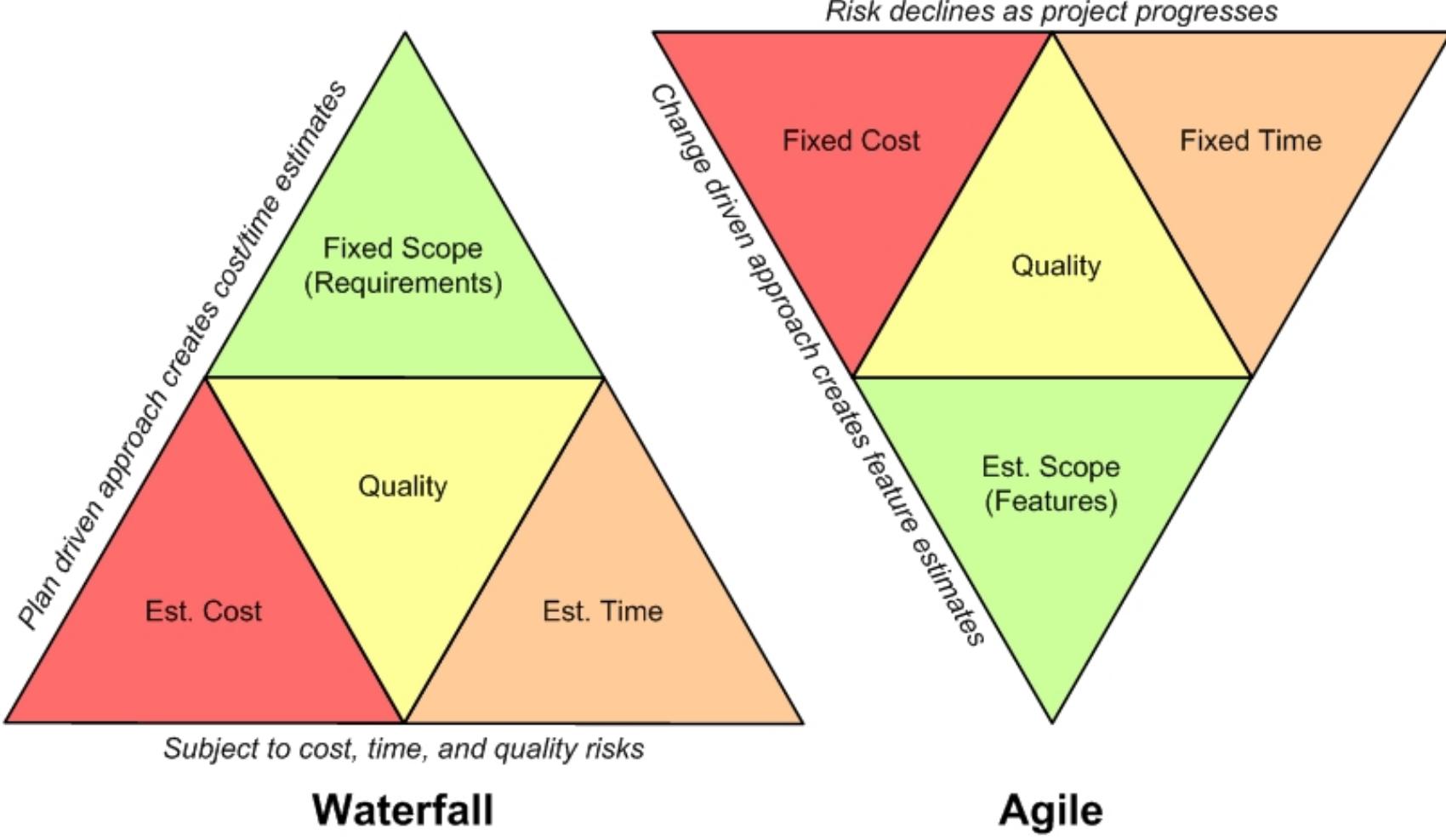
# Agile o pianificato?

lavorare

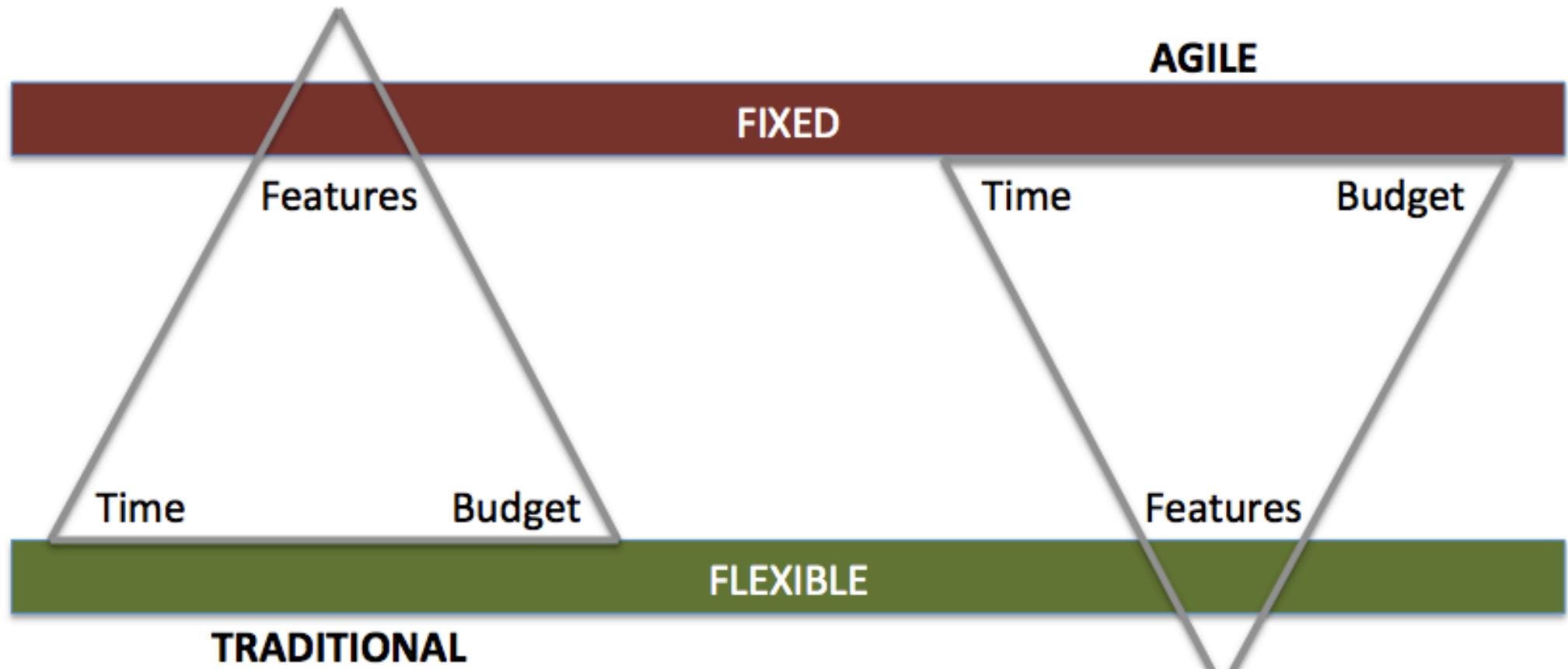
Software meno estremamente Complicato e stabile

Fattore	Pro-agile	Pro-pianificato
Dimensione	Adatto a team che lavorano su prodotti sw "piccoli". L'uso di conoscenza tacita limita la scalabilità	Adatto a grandi sistemi e team. Costoso da scalare verso i prodotti sw "piccoli" → qualche cosa possa avere un danno reale a chi lavora e chi fa
Criticità	Utile per applicazioni con requisiti instabili (es. Web)	Utile per gestire sistemi critici e con requisiti stabili → Se sistema è critico avere più uno stabile ma fa capire bene i responsi e tutto. Es. modello a V.
Dinamismo	Refactoring	Pianificazione dettagliata
Personale	Servono esperti dei metodi agili; con Scrum devono essere "certificati"	Servono esperti durante la pianificazione esperti dell'uno o dell'altro modello
Cultura	Piace a chi preferisce la libertà di fare	Piace a chi preferisce ruoli e procedure ben definiti

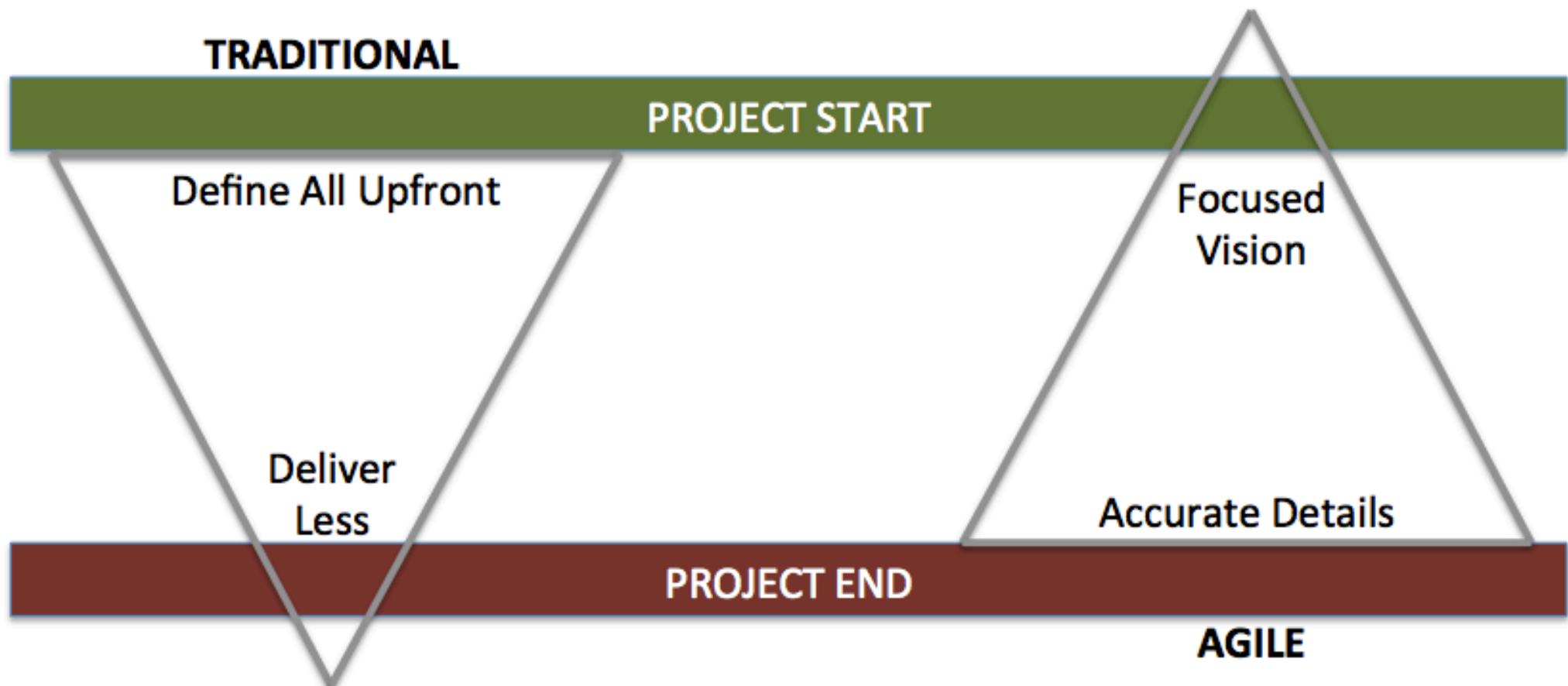
# Iron Triangle Paradigm Shift



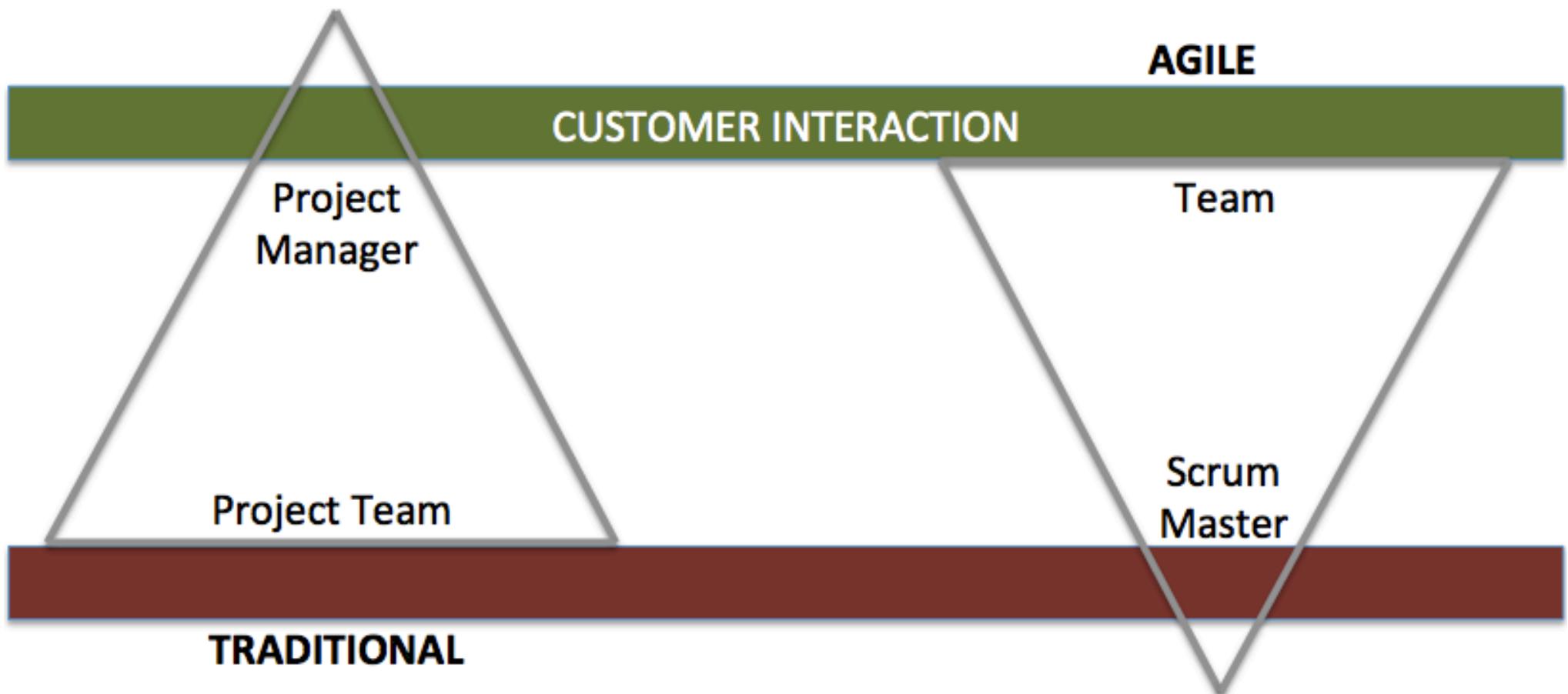
# Il triangolo del Project Management -1



## Il triangolo del Project Management -2



# Il triangolo del Project Management -3



# Conclusioni sui modelli agili

- Il team accetta i cambiamenti dei requisiti ovvero delle user stories
- Sviluppo guidato dai test
- Delivery rapide e frequenti
- Progetto semplice, refactoring sistematico e integrazione continua
- Coinvolgimento del cliente e responsabilizzazione dei membri del team
- Analisi retrospettiva

# Anti-metodologia?

Il movimento Agile non è contrario alle metodologie, anzi molti di noi vogliono ridare credibilità alla parola *metodologia*.

Vogliamo ripristinare un equilibrio.

Abbracciamo la **modellazione**, ma non per archiviare qualche diagramma in un polveroso repository aziendale

Abbracciamo la **documentazione**, ma non centinaia di pagine di tomi mai aggiornati e raramente usati.

**Pianifichiamo**, ma riconosciamo i limiti della pianificazione in un ambiente turbolento.

Jim Highsmith, *History: The Agile Manifesto*

# Oltre l'Agile

Come aspiranti artigiani del software stiamo spostando l'asticella dello sviluppo professionale del software praticandolo e aiutando altri ad apprendere l'arte. Noi apprezziamo:

- Non solo il software funzionante, ma anche **il software ben fatto**
- Non solo rispondere al cambiamento, ma anche **aggiungere valore costantemente**
- Non solo gli individui e le interazioni, ma anche **una comunità di professionisti**
- Non solo la collaborazione col cliente, ma anche **una partnership produttiva**

Cioè nel cercare gli elementi di sinistra abbiamo capito che quelli di destra sono indispensabili.

# Letture raccomandate

- Cohn e Ford, Introducing an agile process to an organization, *IEEE Computer*, 2003
- Bohem e Turner, Using Risk to Balance Agile and Plan-Driven Methods, *IEEE Computer*, 2003
- E. Raymond, *La cattedrale e il bazaar*, 1997  
[http://it.wikisource.org/wiki/La\\_cattedrale\\_e\\_il\\_bazaar](http://it.wikisource.org/wiki/La_cattedrale_e_il_bazaar)

# Riferimenti

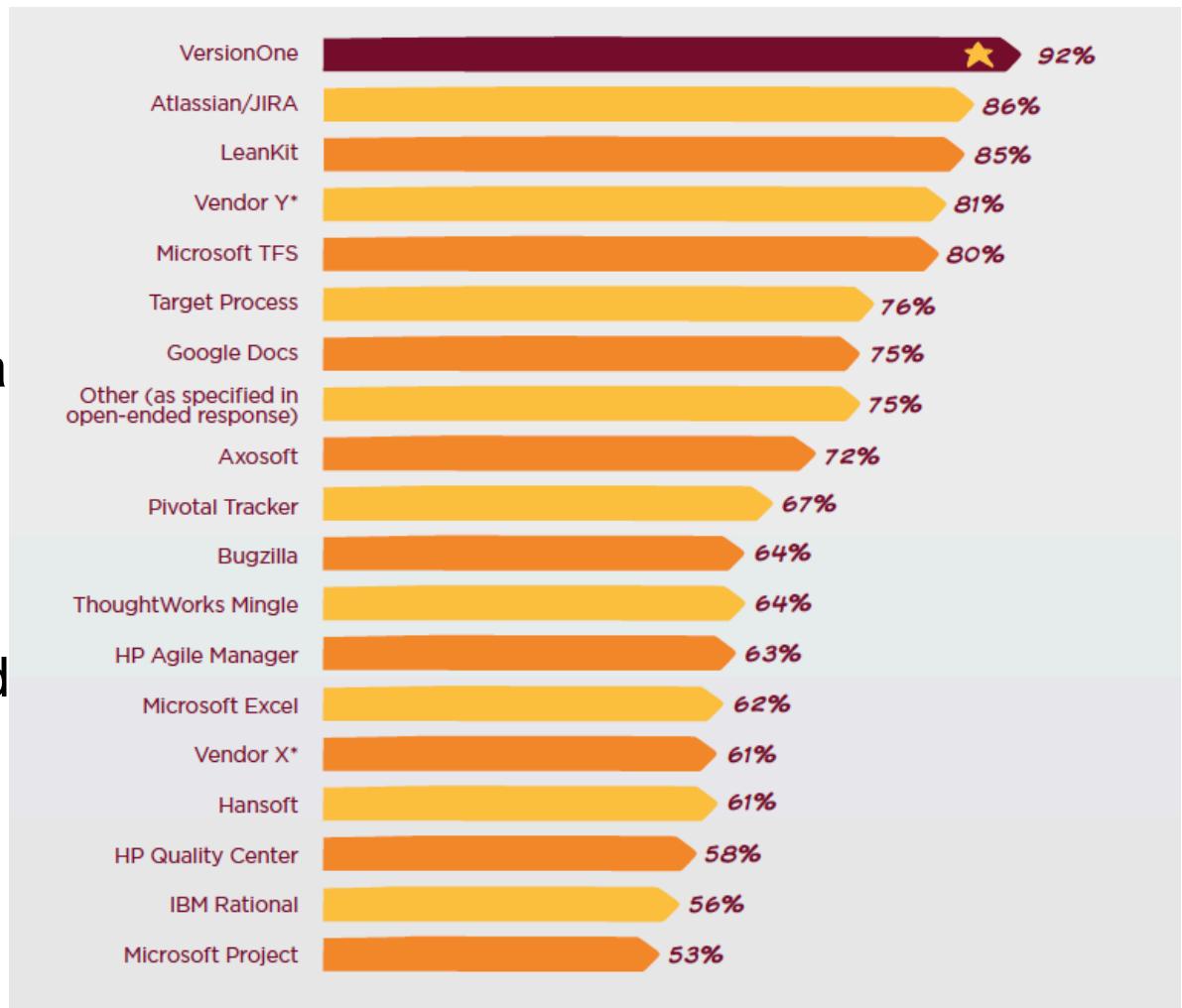
- Beck e altri, Manifesto for agile development, 2001 [agilemanifesto.org](http://agilemanifesto.org)
- Beck & Andres, Extreme Programming Explained: Embrace Change, AW 2004
- Rumpe, Agile modeling with UML, 2017

# Siti e blog

- Extreme Programming [www.extremeprogramming.org](http://www.extremeprogramming.org)
- Scrum [www.scrum.org](http://www.scrum.org)
- Uncle Bob's blog [blog.cleancoder.com](http://blog.cleancoder.com)

# Strumenti

- Trello
- Slack
- Ant, XDoclet, JUnit, Cactus, Maven
- [www.atlassian.com/software/jira](http://www.atlassian.com/software/jira)
- [agiletrack.net](http://agiletrack.net)
- [www.planningpoker.com](http://www.planningpoker.com)
- [www.versionone.com/product/agile-collaboration-tools/](http://www.versionone.com/product/agile-collaboration-tools/)



Modelli di processo 2

# Pubblicazioni di ricerca

- International Conference on Agile Software Development (XP)

# Domande?

