



UNIVERSITÀ DEGLI STUDI  
DI MILANO

# Ingegneria del software

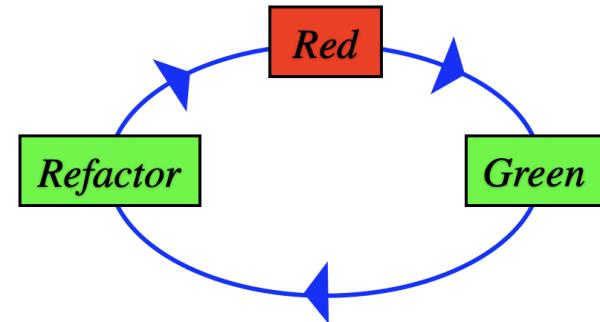
Lezione 3: eXtreme Programming

# eXtreme Programming

*Increment then simplify*



# Test Driven Development



- TDD è una tecnica di progettazione che guida verso il design più semplice
- TDD = test-first + baby steps
  - **Write a failing test**
  - **Make it pass**
  - **Refactor**

Repeat every 2-10 minutes

# eXtreme Programming: Le variabili in gioco

- portata
  - la quantità di funzionalità che si vogliono implementare
    - delicata perché mutevole
- tempo
  - il tempo che si può dedicare al progetto
- qualità
  - la qualità del progetto che si deve ottenere
- costo
  - le risorse finanziarie che si possono impegnare

# eXtreme Programming: i principi

- feedback rapido
- presumere la semplicità
- accettare il cambiamento
- modifica incrementale
- lavoro di qualità
- separazione degli interessi (*aspect* o *concerns*)
- astrazione e modularità
- anticipazione del cambiamento  
(*design for change*)
- generalità
- incrementalità
- rigore e formalità

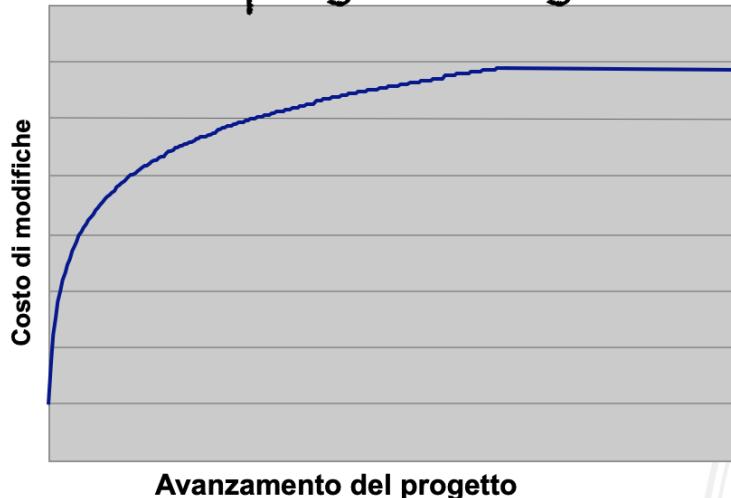
XP

CLASSICI (dal libro)

# Presumere la semplicità vs anticipare il cambiamento

- XP sembra dire di non pianificare per il futuro, per il riuso
- Boehm (1976) basandosi sullo studio di casi “reali” ipotizza una curva di tipo esponenziale per il costo di modifiche

Extreme programming sostiene



# Leprechauns ... cap.10

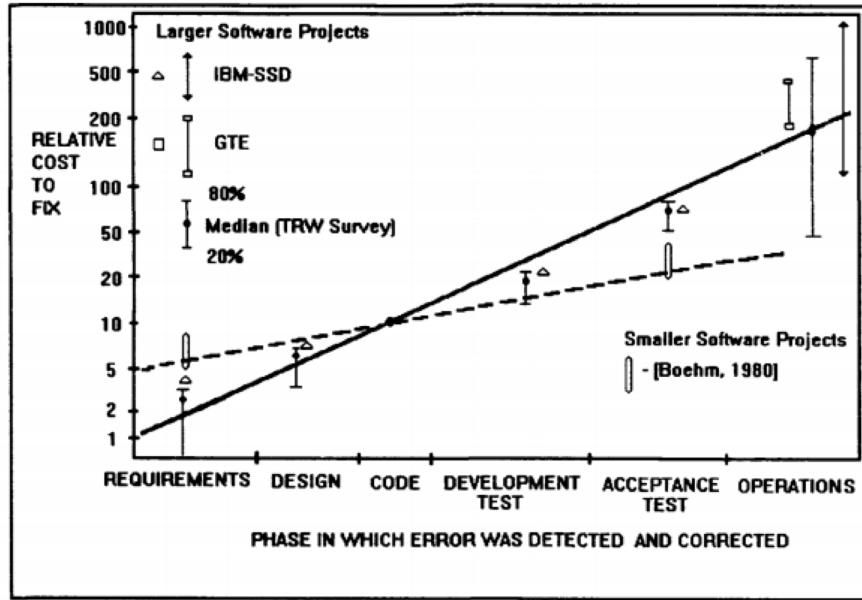
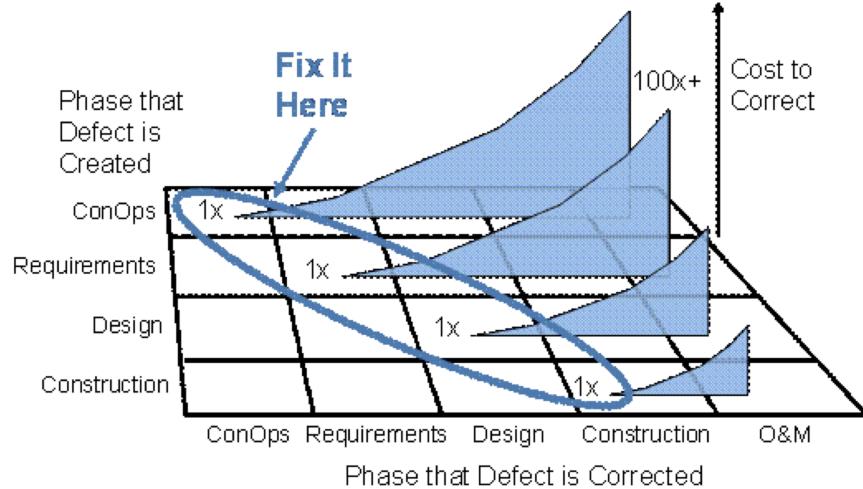


Figure 3. Increase in Cost-To-Fix or Change Software Throughout the Life Cycle (Boehm, 1981:40)



# Figure in gioco e responsabilità

## Manager e/o cliente

Ha responsabilità di decidere:

- la portata del progetto
- priorità tra funzionalità
- date dei rilasci

Ha diritto di:

- sapere che cosa può essere fatto, con quali tempi e a quali costi
- vedere progressi nel sistema, provati dal superamento di test da lui definiti
- cambiare idea, sostituire funzionalità e cambiare priorità

## Sviluppatore

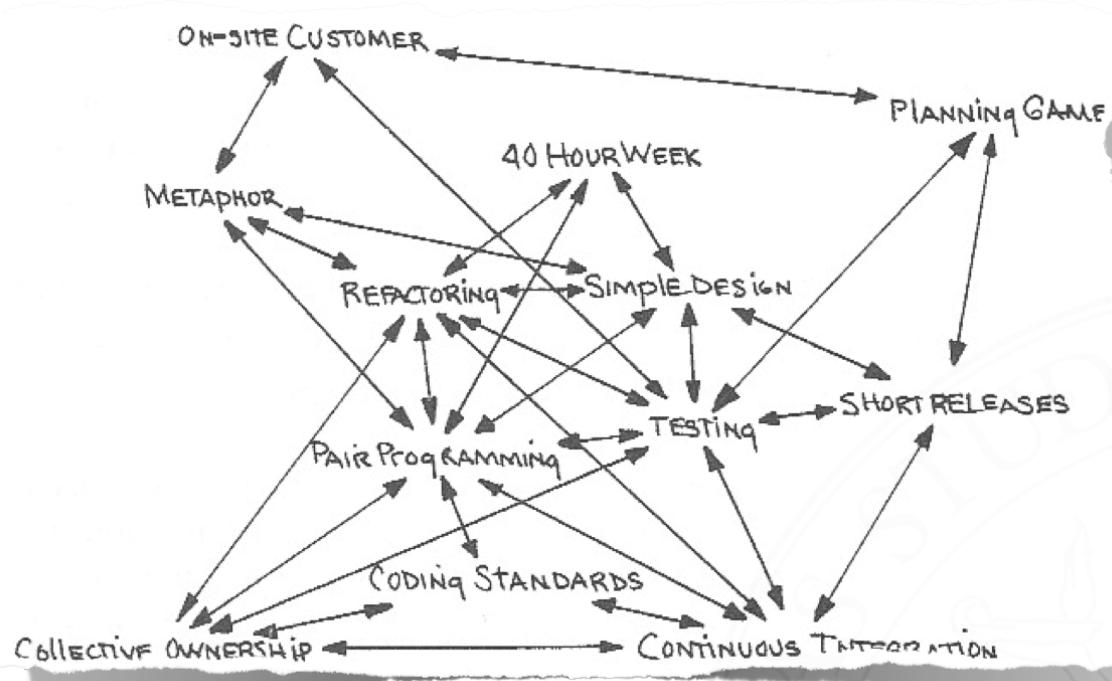
Ha responsabilità di decidere:

- stime dei tempi per le singole funzionalità
- conseguenze di scelte tecnologiche
- pianificazione dettagliata

Ha diritto di:

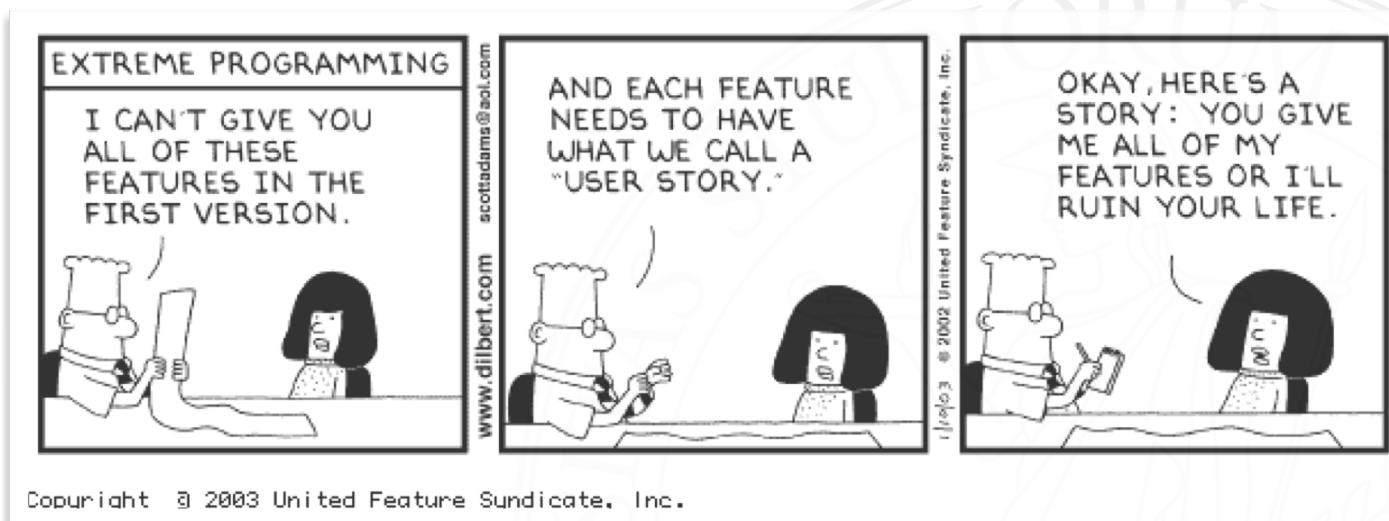
- sapere cosa è necessario attraverso dei requisiti chiari (storie di uso) con priorità
- cambiare stime tempi con esperienza
- identificare e indicare le funzionalità pericolose
- produrre software di qualità

# L'approccio



# 1. Planning game

- Basato su storie scritte dall'utente: una versione semplificata ed informale degli Use Cases di UML
- Vengono determinate le funzionalità del prossimo rilascio, combinando priorità commerciali e valutazioni tecniche.



# Planning game

- Il **cliente** prepara delle carte
  - breve frase di descrizione
  - caso di test che funge da test accettazione
  - il valore di business che ha per lui
- Gli **sviluppatori** stimano il tempo necessario
  - ognuno dice la propria stima
- Il **manager** sulla base di queste informazioni può decidere quali schede verranno implementate alla prossima iterazione

ID

frase che descrive uno scenario d'uso

TEST DI ACCETTAZIONE

Stima Tempo

VALORE

# Come si effettua una stima da parte del team

- Le stime sono molto differenti (ore vs giorni)
- Le stime sono quasi uniformi, ma molto alte
- Non c'è accordo nelle stime, ma simili?
  - si prende il tempo più basso? il più alto? la media?

## Problemi nelle stime condivise

- Perdita di tempo per troppa comunicazione
- Effetto àncora (Anchoring effect)

# Anchoring effect



# Stime agili: planning poker



- Vengono presentate brevemente le carte
- Il team può fare domande, richiedere chiarimenti e discutere per chiarire assunzioni e rischi
- Ogni componente sceglie una carta del poker rappresentante la propria stima
  - unità di misura?
- Le carte vengono girate contemporaneamente
- Chi ha espresso le stime più basse e più alte ha “un minuto” per spiegare le sue motivazioni e cercare di convincere gli altri
- Si ripete la votazione fino a quando si raggiunge unanimità

# Team Estimation Game

## PRIMA FASE: VALUTAZIONE COMPARATIVA

- I developer si mettono in fila
- Si fa una “pila” con le storie e si mette la prima carta al centro del tavolo
- Il primo developer della fila prende una carta dalla pila, la rilegge (ad alta voce) e la posiziona a sinistra (più semplice) a destra (più complicata) sotto (equivalente)
- Il prossimo developer può:
  - prendere una nuova carta dalla pila e posizionarla secondo le stesse regole (può anche distanziare due file e metterla in mezzo tra le due)
  - spostare una carta precedentemente posizionata (commentando la motivazione della sua azione)

<- Smaller

Larger ->

Da 1 punto,  
è molto più  
facile

# Team Estimation Game

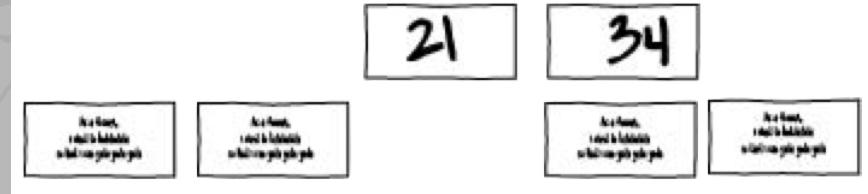
## SECONDA FASE: QUANTIFICARE DISTANZE

- ci si mette di nuovo in coda davanti al tavolo, il mazzo di carte questa volta è tipo quello del planning poker (ordinato e senza carte "strane"... quindi si è scelta una scala di valori)
- si posiziona la prima carta (di solito si parte dal valore 2 perché potrebbe esserci nella prossima iterazione qualcosa di ancora più semplice) sulla prima colonna
- il primo developer prende il valore successivo e lo posiziona sulla prima colonna che pensa abbia quel valore (rispetto a quello con valore 2) o tra due colonne
- Il prossimo developer può:
  - prendere una nuova carta valore dalla pila e posizionarla secondo le stesse regole (la prima colonna che gli sembra abbia bisogno di tale valore)
  - spostare una carta valore precedentemente posizionata (commentando la motivazione della sua azione)
  - se non ci sono carte nella pila e non vuole spostare esistenti, "passare"

# Team Estimation Game

## SECONDA FASE: QUANTIFICARE DISTANZE

- Si ripete il passo precedente fino a quando
  1. non ci sono più carte sulla pila E
  2. nessun developer vuole spostare una carta
- Le colonne senza una carta valore sopra, vengono assimilate alla colonna alla loro sinistra



# Stime agili: Team Estimation Game

## TERZA FASE: SCALA ASSOLUTA

- Si stima il tempo in ore/uomo di una delle carte più semplici (valore 2)...
- Si calcolano tutti gli altri come proporzioni

Necessario?

# Velocity

- capacità (osservata) di completare lavori da parte del team
- Sostituisce la necessità di rimappare unità ideali in tempi assoluti
  - dopo la prima iterazione, il team dirà che può sviluppare tanti "punti" quanti ne ha fatti alla iterazione precedente
- Permette di compensare tendenze sbagliate di stime all'interno del team
- NON deve essere usato come metro di valutazione tra team, o nel tempo
- NON si devono considerare storie non finite
- NON deve essere imposta

## 2. Brevi cicli di rilascio

- Per ridurre i rischi, la vita e lo sviluppo dell'applicazione sono scanditi dai rilasci di versioni del prodotto funzionanti
- Devono essere comunque rilasci significativi (responsabilità del management)

### 3. Uso di una metafora

- Serve come vista aggregante
  - *Sostituisce* in parte l'architettura del sistema
  - Fa capire gli elementi fondamentali e le loro relazioni
- Fornisce nuovi elementi di discussione
- Nuovo vocabolario con cui parlare con l'utente (quindi non tecnica informatica), ma anche ai nuovi venuti
  - Dovrebbe permettere nominazione classi e metodi omogenee

# 4. Semplicità di progetto

L'arte di massimizzare il lavoro non fatto (o da non fare)

- *One and once only*: cioè tutto quello che serve e senza duplicazioni
- KISS: keep it simple, stupid.
- Si contrappone al *Design for change* che viene visto come un appesantimento inutile
  - cosa succede se la *feature* per cui avevamo creato infrastruttura vengono tolte dal cliente?

# 5. Testing

- Abbiamo già accennato al TDD
- I clienti scrivono i test funzionali per aumentare la loro fiducia nel programma
- I programmatori scrivono i test di unità perché la fiducia nel codice diventi parte del programma stesso
  - Facilita refactoring... confidenza nelle modifiche che vengono apportate

# 6. Refactoring

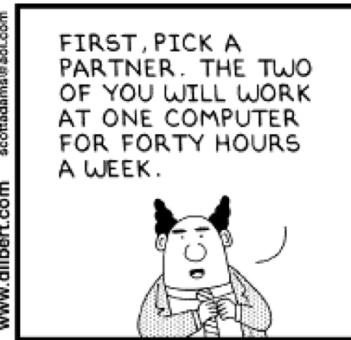
Modifiche al codice che non modificano le funzionalità

- Non bisogna avere paura di apportare modifiche che semplifichino il progetto
- O che rendano più semplice aggiungere una nuova funzionalità
- Deve essere fatto gradualmente ma con continuità e senza pietà
- Serve ad eliminare l'entropia che si creerebbe con le continue modifiche ed aggiunte

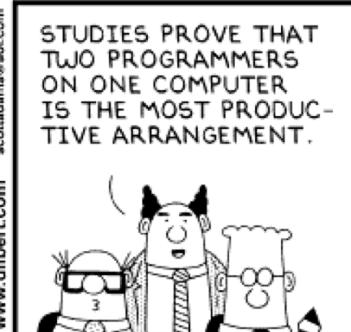
# 7. Programmazione a coppie

- Aiuta ad avere un controllo continuo del rispetto delle regole di XP
- Aiuta l'inserimento di nuovo personale e la sua formazione
- Aiuta a ottenere *proprietà collettiva* (conoscenza osmotica)
- Aiuta refactoring
- Ma dimezzare le persone che scrivono il codice dimezza la produttività del team?

# Pair programming secondo Dilbert



Copyright © 2003 United Feature Syndicate, Inc.



Copyright © 2003 United Feature Syndicate, Inc.