



# INGEGNERIA DEL SOFTWARE - 2025-26

# SVILUPPO AGILE DEL SOFTWARE

LEZIONE 6  
15/10/2025  
VINCENZO RICCIO

# RIFERIMENTI

- ▶ Sommerville - Paragrafi 3.1, 3.2

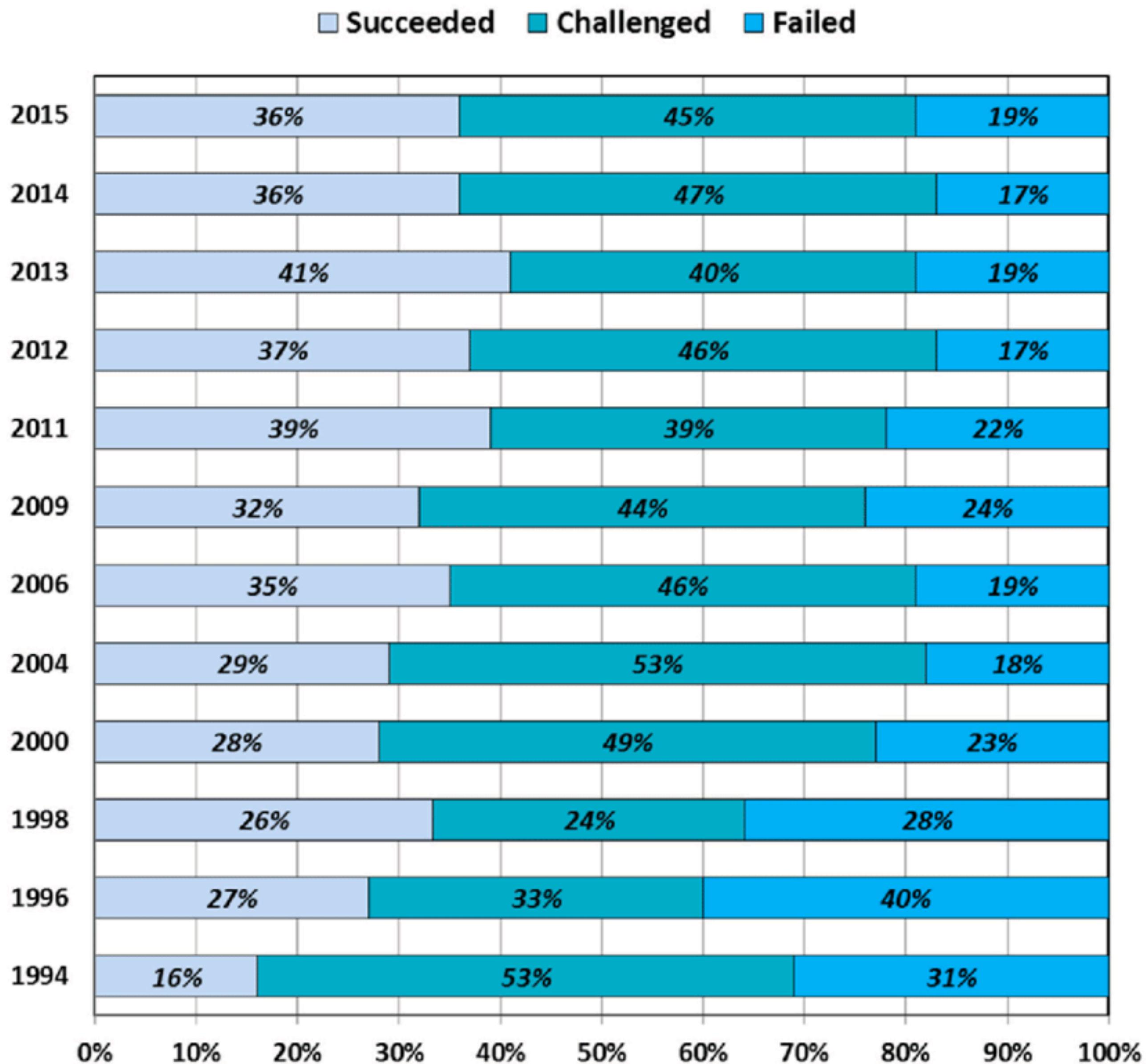
Vincenzo Riccio  
Ingegneria del Software 2025/2026  
Università degli Studi di Udine





# MOTIVAZIONI

# PREMESSA



- ▶ Secondo il CHAOS Report dello Standish Group fino al 2000:
  - Meno del 30% dei progetti software avevano successo
  - Progetti grandi fallivano più spesso
  - Solo metà delle feature richieste era effettivamente rilasciata

# PROBLEMA: INSTABILITÀ DEI REQUISITI SOFTWARE

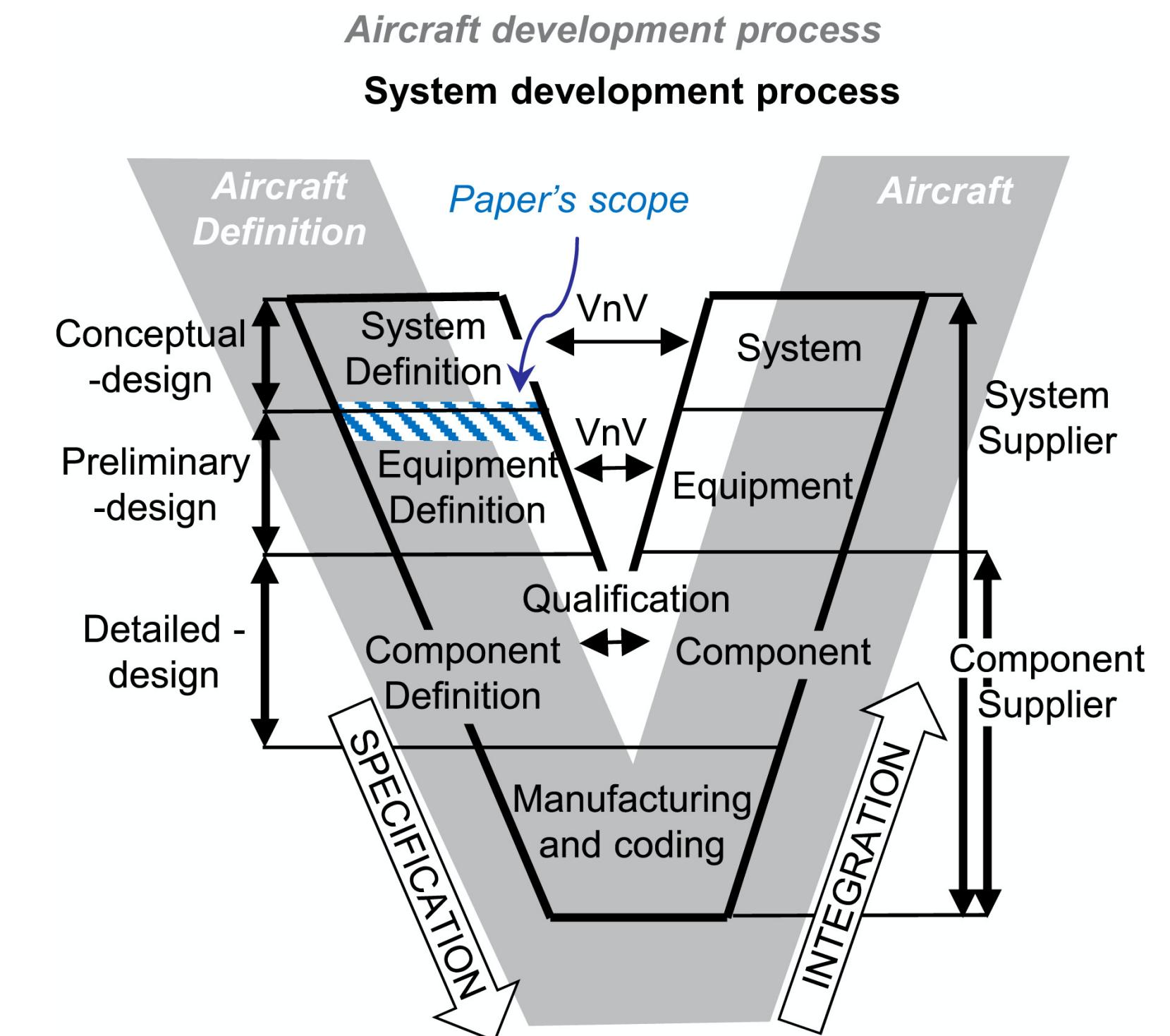


- ▶ Molti progetti hanno **requisiti non chiari, instabili e variabili**
- ▶ Le numerose prescrizioni da seguire, la quantità di documenti richiesta e una eccessiva rigidità rendono “pesante” il processo di sviluppo plan-driven, aumentando il rischio di fallimento o allungando i tempi di consegna
- ▶ In molti casi è richiesto un approccio flessibile (come il modello evolutivo) e “leggero”

# SUCCESSI DEI PROCESSI SOFTWARE PLAN-DRIVEN



- ▶ Gli approcci plan-driven allo sviluppo sono fondamentali per alcuni sistemi critici (ad es. sistemi di controllo per gli aerei moderni)
- ▶ I sistemi critici sono storicamente sviluppati:
  - Da team numerosi e geograficamente dislocati che devono essere attentamente coordinati
  - Attraverso processi che durano lunghi periodi
  - Per software che sarà usato (e quindi manutenuto) a lungo



# INSUCCESSI DEI PROCESSI SOFTWARE PLAN-DRIVEN

Vincenzo Riccio  
Ingegneria del Software 2025/2026  
Università degli Studi di Udine



- ▶ I modelli plan-driven si adattano poco al contesto dinamico e alla necessità di consegna rapida di molti progetti (specialmente il software aziendale)
- ▶ Aggiungono un eccessivo overhead (ad es. produzione di documentazione)
- ▶ Sono poco flessibili ai cambiamenti nei requisiti





# SVILUPPO RAPIDO DEL SOFTWARE

# SVILUPPO RAPIDO DI SOFTWARE UTILE

Vincenzo Riccio  
Ingegneria del Software 2025/2026  
Università degli Studi di Udine



- ▶ Nuove opportunità in un contesto globale in rapido cambiamento
  - Il software deve evolvere rapidamente per riflettere i cambiamenti nelle necessità dei committenti e utenti finali
- ▶ Presenza di prodotti concorrenti
  - Il software deve essere consegnato presto per essere competitivo sul mercato
- ▶ La **rapidità** dello sviluppo e consegna è spesso il requisito più critico per i sistemi software



# Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it.

Through this work we have come to value:

**Individuals and interactions** over processes and tools  
**Working software** over comprehensive documentation  
**Customer collaboration** over contract negotiation  
**Responding to change** over following a plan

That is, while there is value in the items on the right,  
we value the items on the left more.

Kent Beck  
Mike Beedle  
Arie van Bennekum  
Alistair Cockburn  
Ward Cunningham  
Martin Fowler

James Grenning  
Jim Highsmith  
Andrew Hunt  
Ron Jeffries  
Jon Kern  
Brian Marick

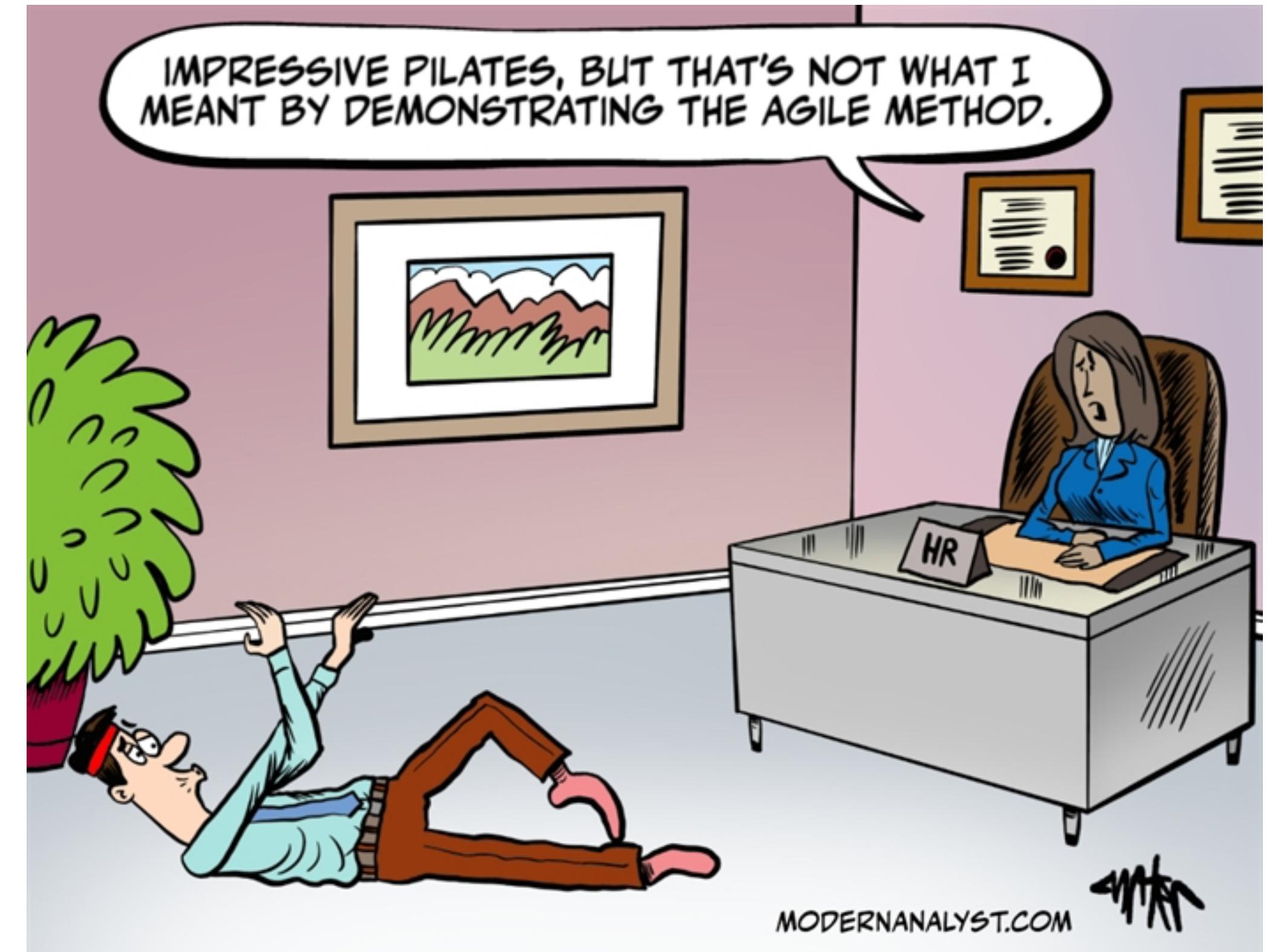
Robert C. Martin  
Steve Mellor  
Ken Schwaber  
Jeff Sutherland  
Dave Thomas



Alla fine degli anni 90 sono emersi i metodi agili per lo sviluppo software, mirati a ridurre radicalmente il tempo di consegna dei prodotti software



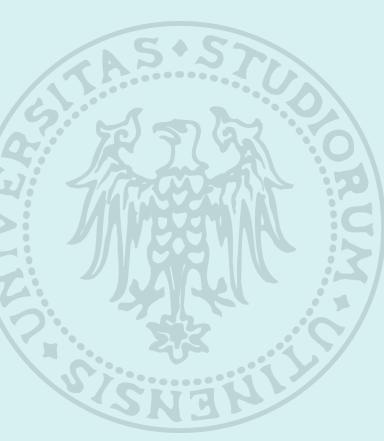
- ▶ Efficace risposta (rapida e flessibile) ai cambiamenti
- ▶ Efficace comunicazione fra tutti gli stakeholder
- ▶ Portare il cliente nel team di lavoro per avere feedback rapido
- ▶ L'agilità consente di avere una rapida, incrementale consegna del software





- ▶ **Documentazione minima**
  - Il focus è sul codice, invece che sulla progettazione
  - Non vi sono specifiche dettagliate
  - Overhead di documentazione limitati
- ▶ **Consegna rapida ed incrementale**
  - Il sistema è sviluppato in incrementi rilasciati frequentemente
  - Gli stakeholder sono coinvolti nella specifica e nella valutazione di ogni incremento, definendo insieme gli incrementi successivi
- ▶ **Strumenti di supporto**
  - Utilizzo di strumenti al supporto del processo di sviluppo, ad es. automatizzazione dei test

# PROCESSI PLAN-DRIVEN VS AGILI



- **Plan-driven:** Processi dove tutte le attività sono pianificate in anticipo e il loro avanzamento è misurato rispetto a quanto previsto dal piano. Le fasi del processo software sono distinte tra loro e gli output di ciascuna fase sono necessari per la fase successiva
- **Agili:** La pianificazione è incrementale e continua durante lo sviluppo del software. È quindi più facile modificare il processo per adeguarsi alle modifiche dei requisiti del cliente o del prodotto. Requisiti, progettazione e implementazione avvengono assieme

# PROCESSI PLAN-DRIVEN E AGILI



- ▶ Pratiche plan-driven e agili possono coesistere nello stesso processo:
  - Processi agili possono produrre documentazione di progettazione quando ritenuto necessario e possono includere attività pianificate. Lo scopo dei documenti di progetto è supportare la comunicazione e comprensione, anche sapendo che essi sono incompleti o imprecisi
  - Processi plan-driven possono essere incrementali
- ▶ Per grandi sistemi occorre trovare un compromesso tra processi pianificati e processi agili

# PROCESSI PLAN-DRIVEN E AGILI: DISCLAIMER

Vincenzo Riccio  
Ingegneria del Software 2025/2026  
Università degli Studi di Udine



- ▶ Abbiamo visto i modelli a sviluppo incrementale tra i processi plan-driven
- ▶ I processi agili pongono ancora maggiore enfasi sull'incrementalità e sulla rapidità degli incrementi
- ▶ Tuttavia i processi plan-driven sono solitamente organizzati in incrementi pianificati



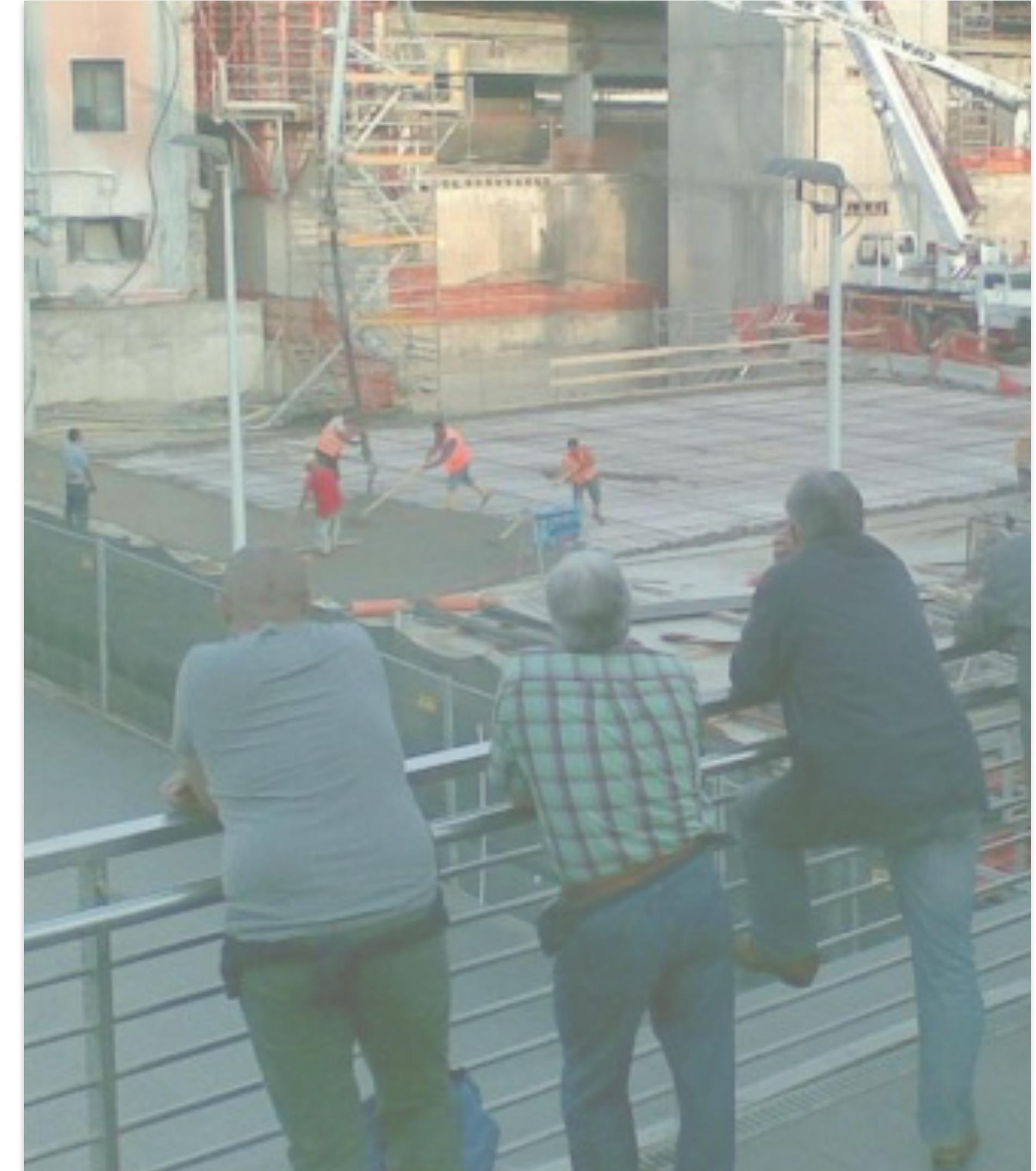


# PRINCIPI AGILI

# PRINCIPI AGILI



- ▶ **Coinvolgimento del cliente**
- ▶ I clienti sono coinvolti in tutto il processo di sviluppo
- ▶ I clienti intervengono a ciascuna iterazione del prodotto:
  - Valutano e validano l'iterazione
  - Forniscono nuovi requisiti del sistema o propongono modifiche alle iterazioni proposte
  - Assegnano una priorità a ciascun requisito richiesto





- ▶ **Accettare i cambiamenti**
  - Il prodotto non deve essere pianificato rigidamente
  - Bisogna prevedere che i requisiti del sistema potranno cambiare
  
- ▶ **Mantenere la semplicità**
  - Sia il prodotto che il processo di sviluppo devono essere più semplici possibili
  - Quando possibile bisogna lavorare attivamente per eliminare le complessità dal sistema





- ▶ **Sviluppo incrementale**
  - Il software è sviluppato e consegnato incrementalmente
  - Il cliente specifica i requisiti da includere in ciascun incremento
- ▶ **Persone, non processi**
  - Il processo di sviluppo non deve essere fortemente prescrittivo
  - I membri del team devono essere liberi di sviluppare il software secondo i loro metodi
  - Le capacità di ogni membro del team devono essere sfruttate al meglio, anziché ingabbiate in un processo rigido

# APPLICABILITÀ DEI METODI AGILI

Vincenzo Riccio  
Ingegneria del Software 2025/2026  
Università degli Studi di Udine



- ▶ Prodotti di **piccole o medie dimensioni**
- ▶ Prodotti **personalizzati** per cui c'è un chiaro impegno del cliente nell'essere coinvolto nel processo di sviluppo

- ▶ Prodotti in cui ci sono **pochi stakeholder** e non bisogna rispettare rigidi regolamenti
- ▶ Team **fisicamente vicini**, in modo che le comunicazioni siano informali e facilitate

# ESEMPIO DI TIPICO SVILUPPO AGILE



- Il prodotto evolve attraverso multiple brevi iterazioni
- Le iterazioni hanno durata costante (~ 2-3 settimane)
- Si rilascia un'applicazione funzionante al termine di ogni iterazione
- Con ogni nuova release si aggiungono quante più caratteristiche è possibile tra quelle con la più alta priorità richieste dal cliente
- Il cliente rivede ogni nuovo incremento e può ridefinire le priorità per la prossima iterazione
- Non si fallisce una data di consegna, piuttosto si rimanda lo sviluppo di qualche feature

# VANTAGGI DELLO SVILUPPO AGILE

Vincenzo Riccio  
Ingegneria del Software 2025/2026  
Università degli Studi di Udine



- ▶ Consegne predibili
- ▶ Rapida risposta ai cambiamenti dei bisogni utente
- ▶ Rischi attenuati grazie a cicli di consegna più brevi
- ▶ Alta produttività
- ▶ Clienti soddisfatti, successo dei progetti prodotto evolve attraverso multiple brevi iterazioni



# TECNICHE AGILI



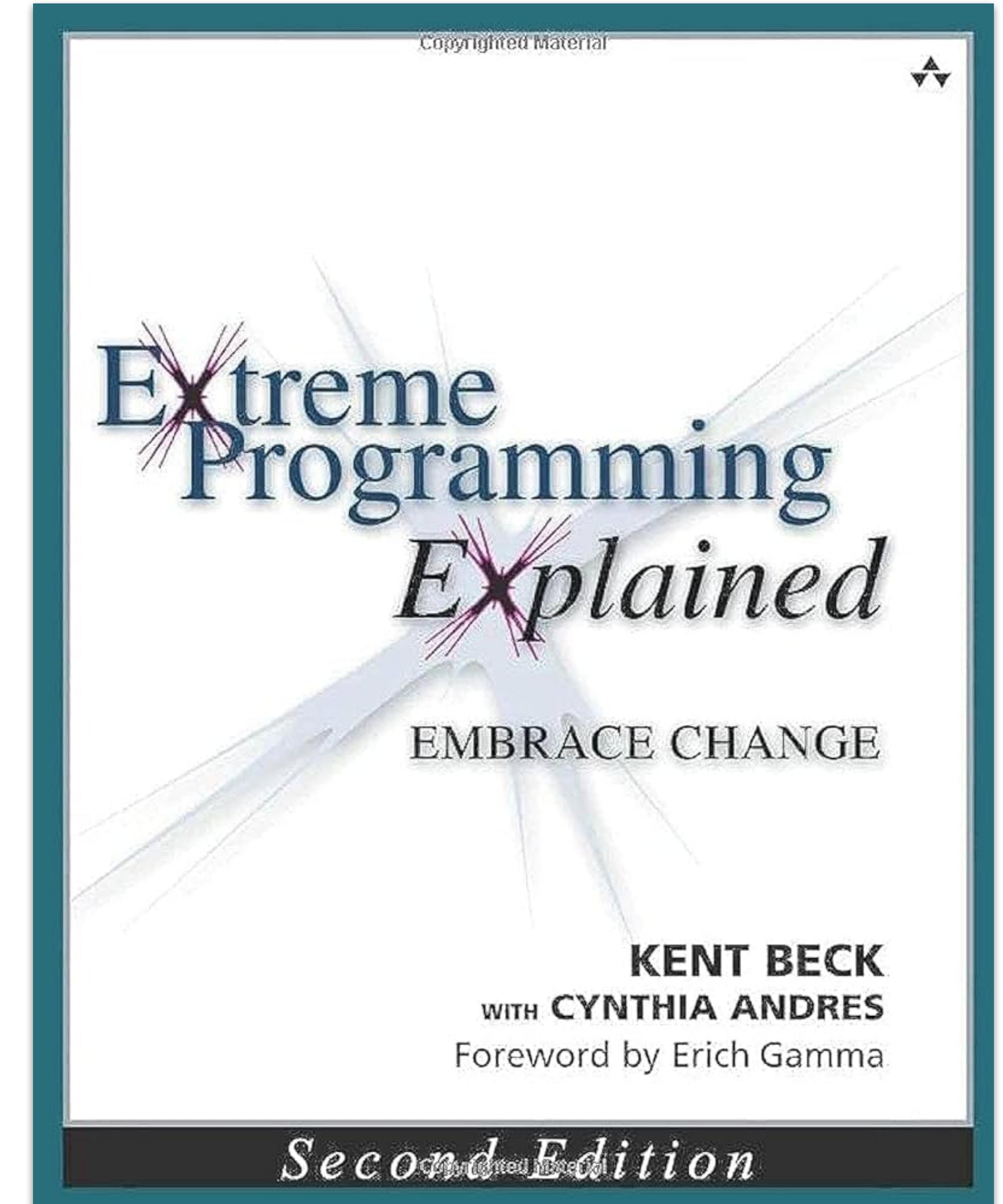
- ▶ Diverse proposte di metodi agili:
  - ▶ Extreme Programming (Kent Beck 1999)
  - ▶ SCRUM (Sutherland)
  - ▶ Feature Driven Development (De Luca & Coad)
  - ▶ Crystal (Cockburn )
  - ▶ DSDM (Dynamic System Development Method) Lean Software Development (Poppendieck)
- ▶ Ciascuno di questi metodi propone un diverso processo, ma tutti condividono gli stessi principi

# EXTREME PROGRAMMING

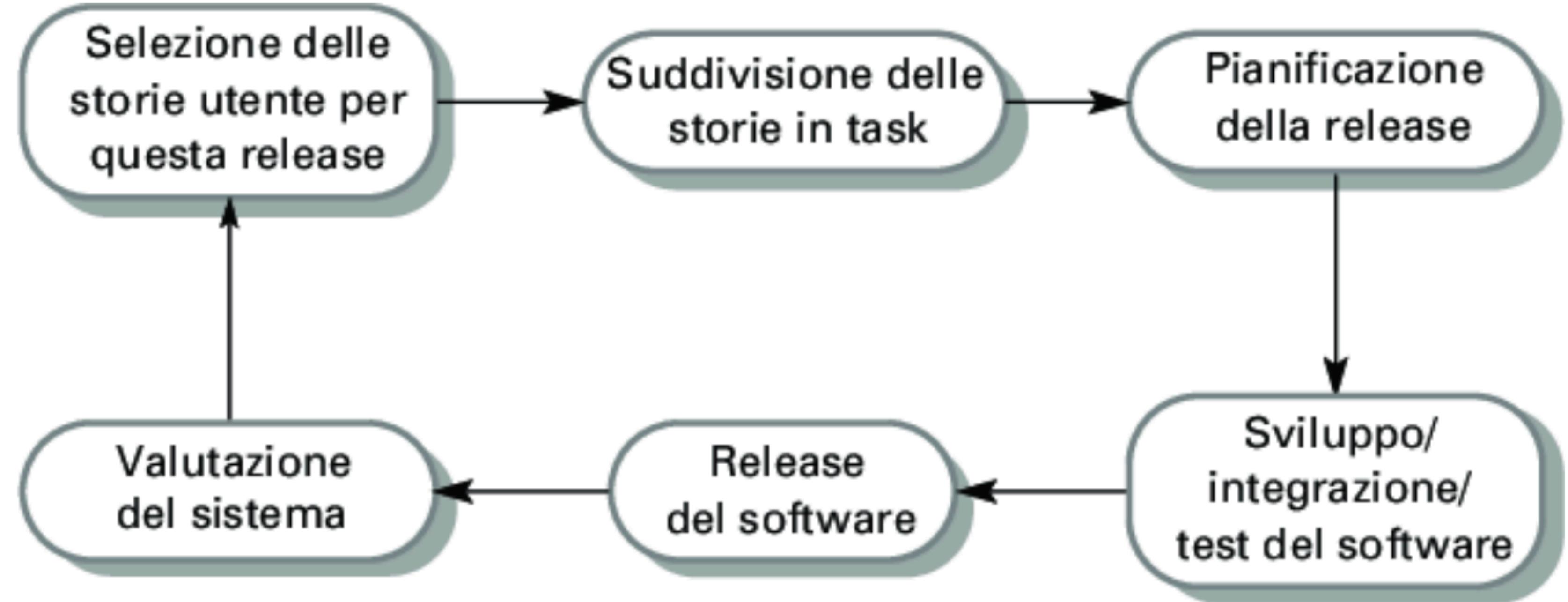
Vincenzo Riccio  
Ingegneria del Software 2025/2026  
Università degli Studi di Udine



- ▶ eXtreme Programming (XP) è il metodo agile più conosciuto
- ▶ Spinge le normali pratiche di sviluppo (ad es. sviluppo iterativo) a livelli “estremi”

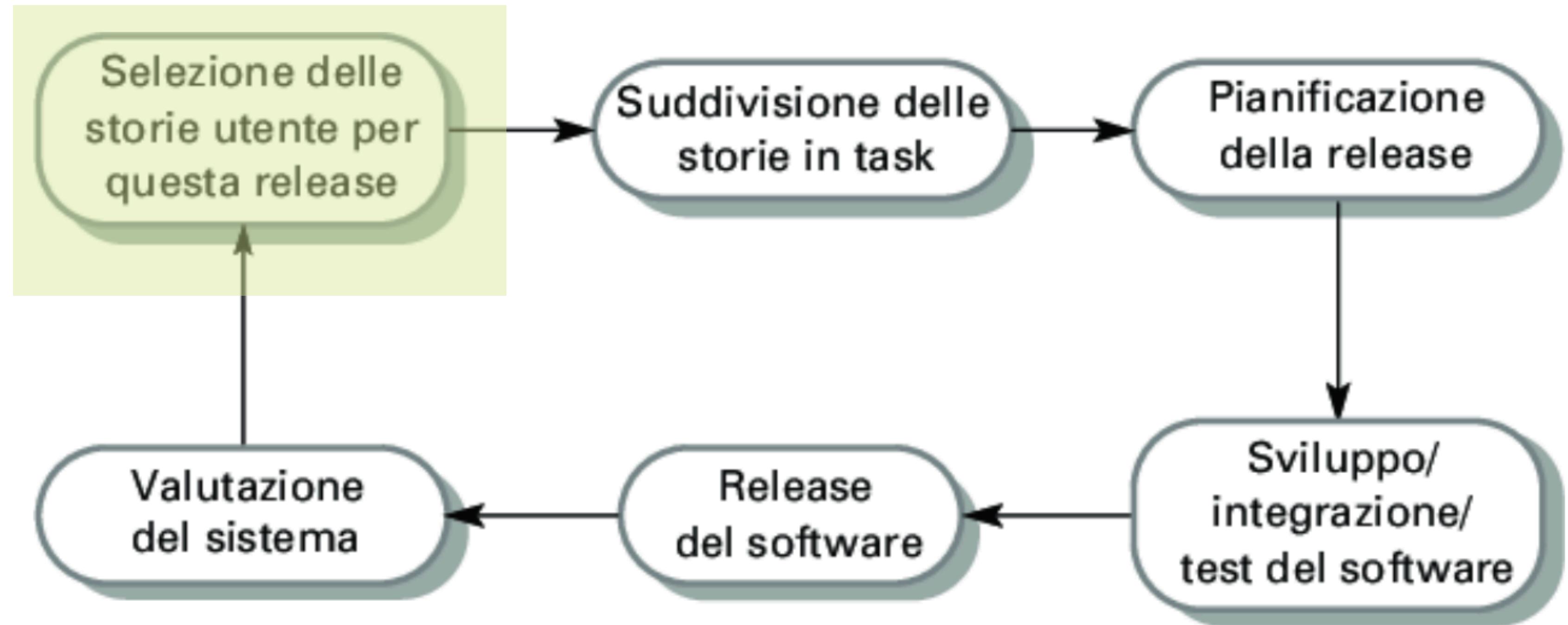


# EXTREME PROGRAMMING



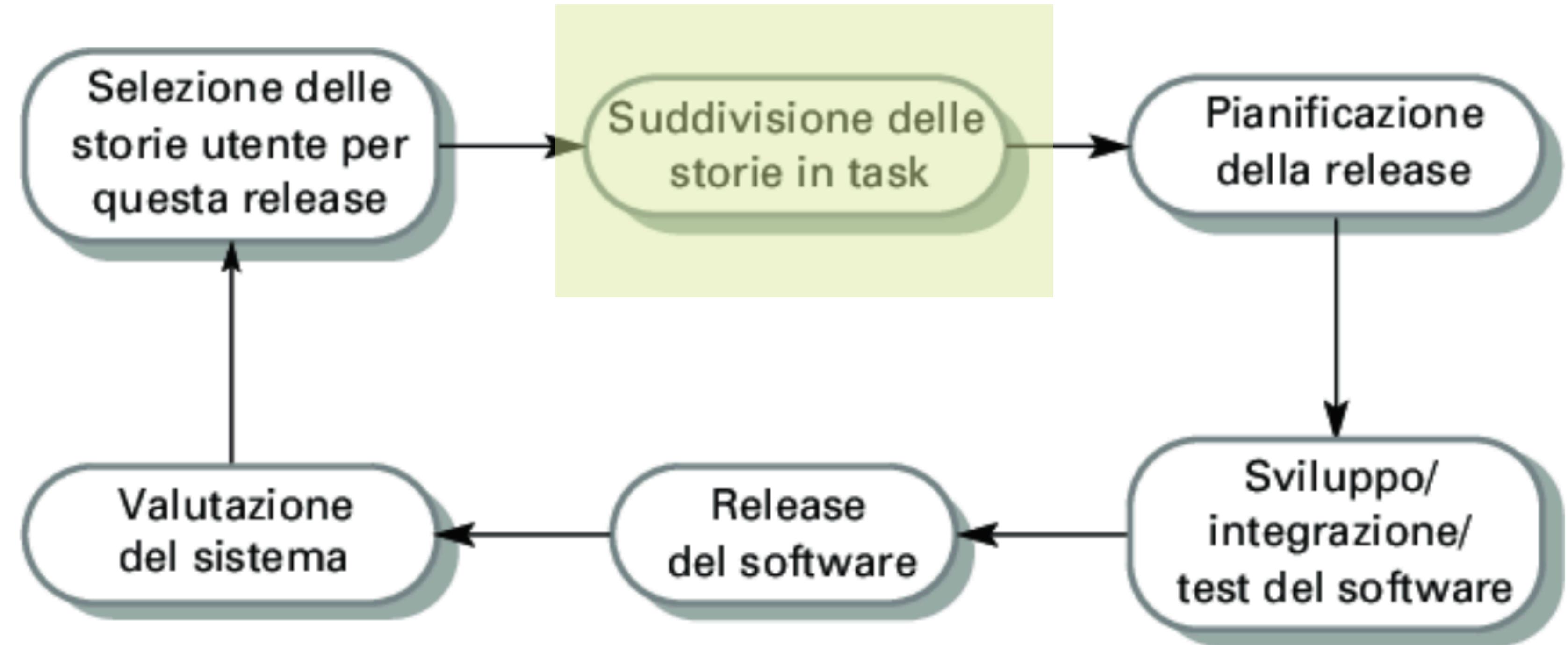
- Approccio iterativo estremo
- Piccoli e frequenti incrementi rilasciati al cliente (ad es. ~ ogni 2 settimane)

# EXTREME PROGRAMMING



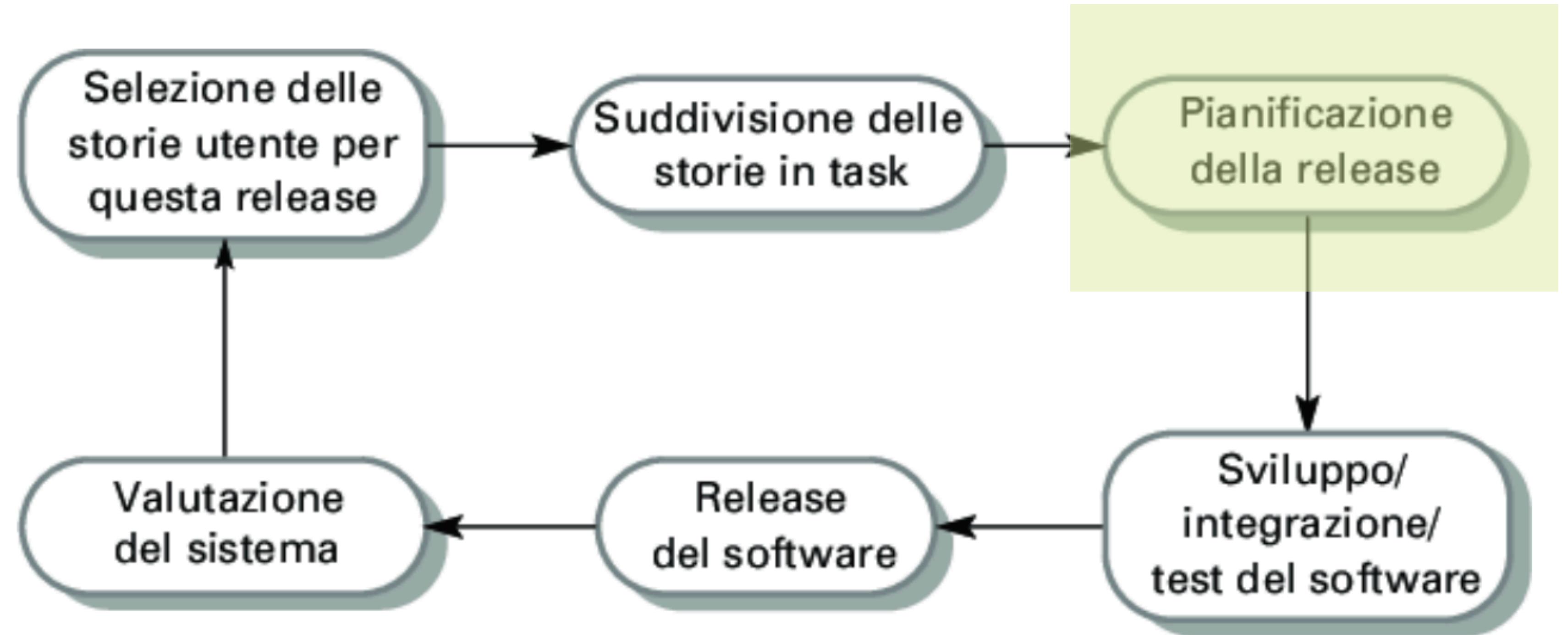
- I requisiti sono espressi come storie utente
- Le storie utente sono semplici scenari, che sono utilizzati come base per decidere quale funzionalità deve essere inclusa in un incremento del sistema

# EXTREME PROGRAMMING



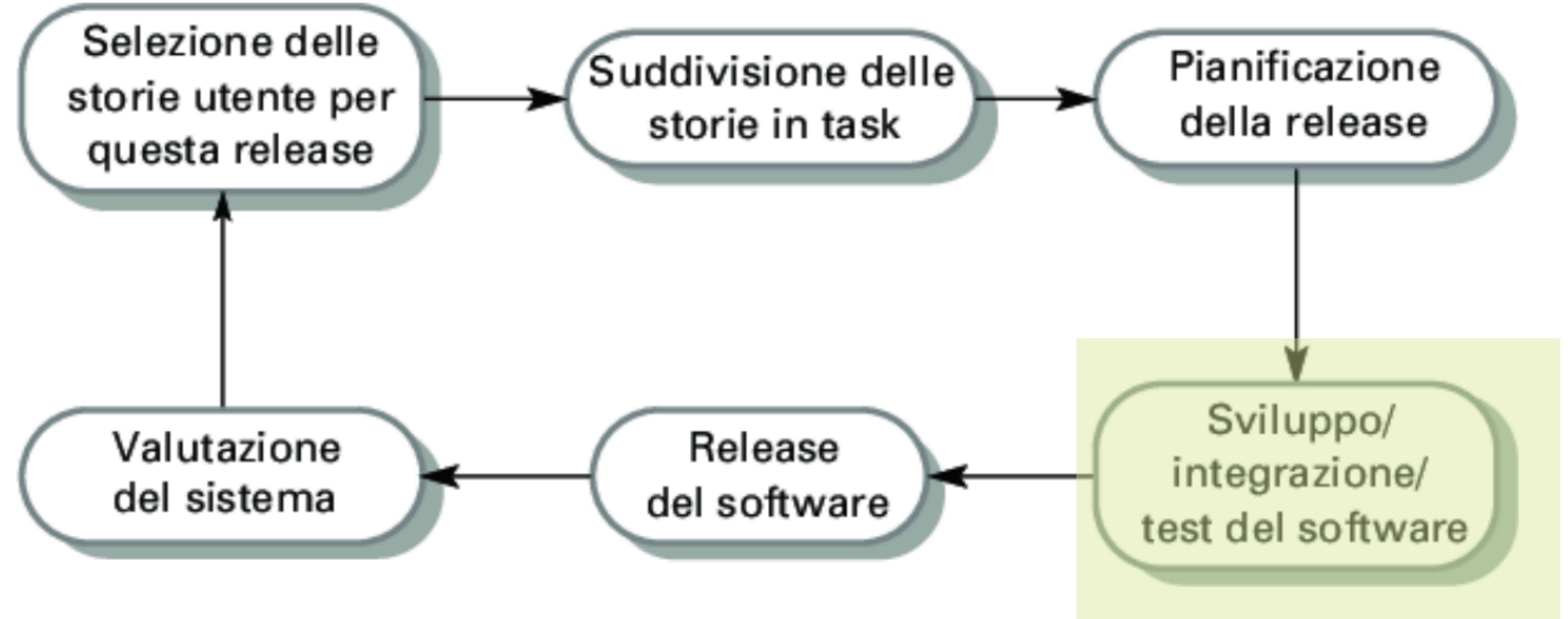
- ▶ Gli scenari sono divisi in task di sviluppo più semplici, che sono implementati direttamente (senza pesante progettazione e documentazione)
- ▶ I task costituiscono le unità principali dell'implementazione

# EXTREME PROGRAMMING



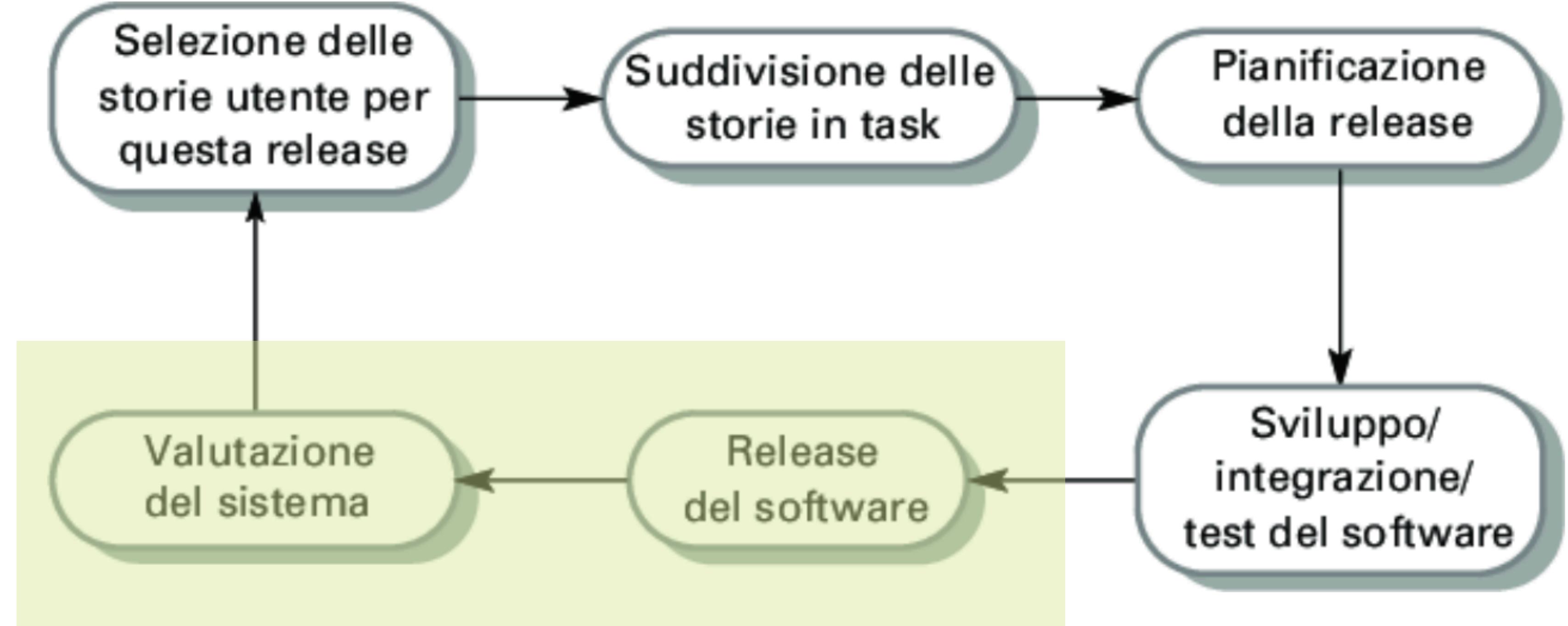
- Agilità non vuol dire assenza di pianificazione, ma pianificazione flessibile e non appesantita da documentazione eccessiva

# EXTREME PROGRAMMING



- I programmatori lavorano a coppie (pair programming)
- Sviluppano test per ogni task prima di scrivere il codice
- Tutti i test devono essere stati eseguiti con successo quando il nuovo codice viene integrato nel sistema

# EXTREME PROGRAMMING



- Il cliente deve essere coinvolto nello sviluppo
- Il cliente valida la release corrente, fornisce requisiti nuovi o modificati, partecipa alla selezione delle storie utente, definisce i test di accettabilità

# EXTREME PROGRAMMING E PRINCIPI AGILI



- **Coinvolgimento del cliente:** un rappresentante del cliente è on-site, parte del team di sviluppo
- **Accettare i cambiamenti:** appena un task è concluso, viene integrato nel sistema
- **Sviluppo incrementale:** piccoli e frequenti rilasci che aggiungono in modo incrementale nuove funzionalità
- **Mantenere la semplicità:** progetto più semplice possibile che soddisfi i requisiti correnti e costante attività di miglioramento del codice (*refactoring*)
- **Persone, non processi:** programmazione in coppia, proprietà collettiva del codice, lo sviluppo non richiede orari di lavoro eccessivamente lunghi e non sono accettabili troppi straordinari

# INFLUENZA DELL'EXTREME PROGRAMMING



- Nella pratica, XP così come proposto originariamente non è quasi mai adottato
- Non pratico perché richiede un cambio radicale nel modo di lavorare di un'organizzazione
- Tuttavia le pratiche chiave dell'XP hanno ispirato tutti gli approcci agili e sono spesso incorporate nei processi di sviluppo
  - Storie utente
  - Refactoring
  - Sviluppo preceduto dai test
  - Pair programming



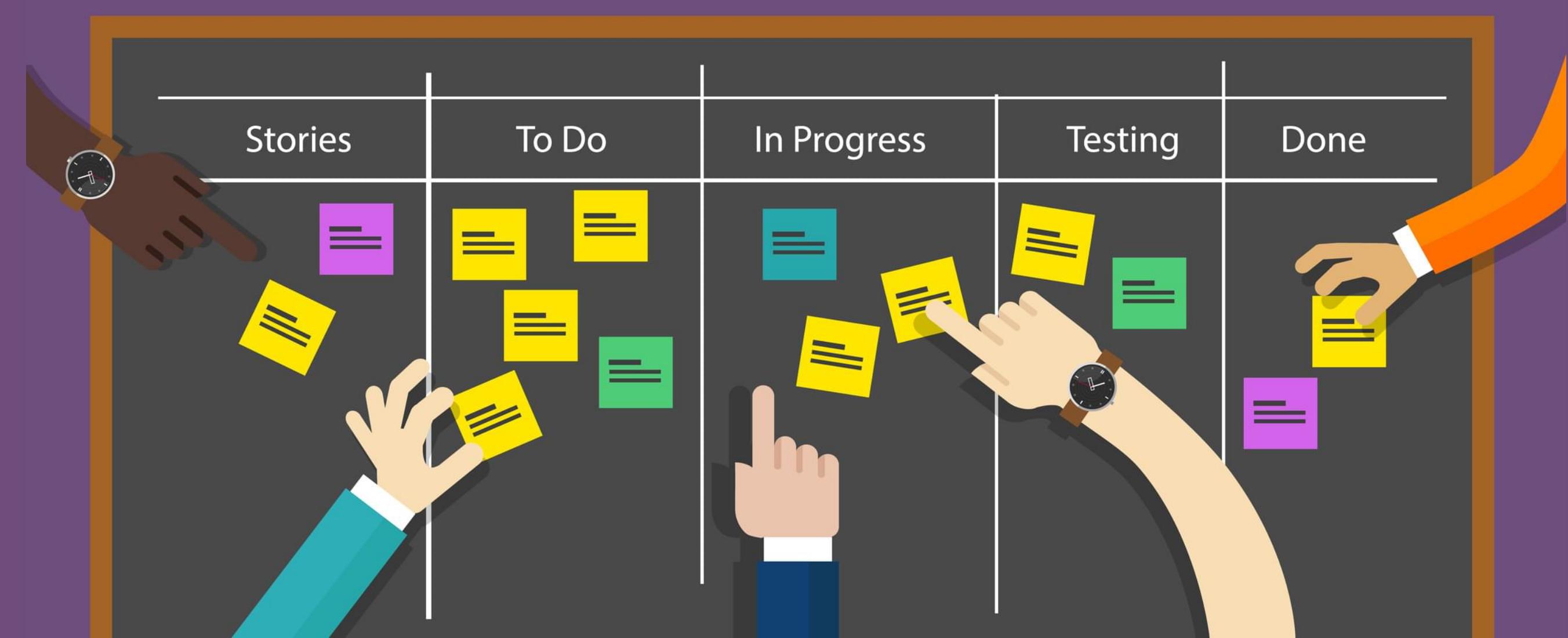
# TECNICHE AGILI

## STORIE UTENTE

# STORIE UTENTE



- ▶ Dette anche *scenari*, descrivono i requisiti degli utenti come scenari d'uso in cui l'utente potrebbe trovarsi
- ▶ Sono scritti da cliente e team su schede (**story cards**) che il team di sviluppo suddivide in **task** da implementare
- ▶ I task rilevati sono la base per definire la pianificazione delle iterazioni e le stime dei costi





- ▶ In XP il cliente (e/o l'utente finale) è parte integrante del team
- ▶ Il compito del cliente è prendere decisioni sui requisiti con il team
- ▶ Cliente e team definiscono una priorità e una stima dei costi per ciascuna storia, oltre ai criteri di accettazione
- ▶ In base a queste informazioni, il cliente e il team scelgono le storie che saranno incluse nella prossima versione del prodotto

<input type="radio"/> Story ID:	Story Title:
User Story:	
As a: <role> I want: <some goal>  So that: <some reason>	
Importance:  <input type="text"/>	
Estimate:  <input type="text"/>	
Acceptance Criteria	
And I know I am done when:	
Type:  <input type="checkbox"/> Search <input type="checkbox"/> Workflow <input type="checkbox"/> Manage Data <input type="checkbox"/> Payment <input type="checkbox"/> Report/ View	

# STORIE UTENTE - ESEMPIO DI STORY CARD



## Prescrizione dei farmaci

Kate è un dottore che desidera prescrivere un farmaco a un paziente che frequenta una clinica. La cartella clinica del paziente è già visualizzata sul suo computer, quindi fa clic sul campo del farmaco e può selezionare "farmaco corrente", "nuovo farmaco" o "formulario".

Se sceglie "farmaco corrente", il sistema le chiede di controllare la dose. Se Kate vuole cambiare la dose, digita la nuova dose e conferma la prescrizione.

Se sceglie "nuovo farmaco", il sistema suppone che Kate conosca quale farmaco prescrivere. Kate digita le prime lettere del nome del farmaco. Il sistema visualizza una lista di possibili farmaci che iniziano con quelle lettere. Kate seleziona il farmaco richiesto e il sistema risponde chiedendole di verificare se il farmaco selezionato è corretto. Lei digita la dose e conferma la prescrizione.

Se sceglie "formulario", il sistema visualizza una finestra di ricerca per il formulario approvato. Kate può cercare il farmaco richiesto e poi lo seleziona. Il sistema le chiede di verificare se il farmaco scelto è corretto. Lei digita la dose e conferma la prescrizione.

Il sistema controlla sempre che la dose sia entro i limiti consentiti; in caso contrario, chiede a Kate di cambiare la dose.

Dopo che Kate ha confermato la prescrizione, dovrà controllarla e poi fare clic su "OK" o su "Cambia". Se fa clic su "OK", la prescrizione viene memorizzata nel database di controllo. Se fa clic su "Cambia", il sistema riavvia il processo di prescrizione dei farmaci.

# STORIE UTENTE - ESEMPIO DI SUDDIVISIONE IN TASK



## Task 1: cambia la dose del farmaco prescritto

### Task 2: scelta dal formulario

### Task 3: controllo della dose

Il controllo della dose è una precauzione per verificare che il dottore non abbia prescritto una dose pericolosamente piccola o grande.

Utilizzando il codice del formulario come nome del farmaco, ricerca nel formulario le dosi minima e massima consigliate.

Confronta la dose prescritta con la minima e la massima. Se la dose è fuori dal range ammesso, visualizza un messaggio di errore per segnalare che la dose prescritta è troppo piccola o troppo grande. Se è all'interno del range, abilita il pulsante "Conferma".

# STORIE UTENTE - PRO



- Integrano la deduzione dei requisiti con lo sviluppo, invece di avere apposite attività di ingegneria dei requisiti, in modo da gestire i cambiamenti nei requisiti
- Più semplice relazionarsi con storie utente, anziché con un tradizionale documento di requisiti. Pertanto, coinvolgono maggiormente l'utente
- Sono ordinate in base a quelle che possono fornire un supporto utile all'azienda
- Se i requisiti cambiano, vengono aggiunte story cards e le storie non ancora realizzate possono essere modificate o scartate

# STORIE UTENTE - CONTRO



- Non è semplice stabilire se le storie utente coprono completamente i requisiti del sistema
- Clienti esperti potrebbero omettere scenari o task considerati ovvi ma che non lo sono per gli sviluppatori, fornendo quindi una descrizione incompleta del requisito



# TECNICHE AGILI

# REFACTORING



- ▶ Un principio basilare dell'ingegneria del software classica è di progettare pensando al cambiamento, riducendo così i costi di manutenzione futura
- ▶ In XP si rinuncia a gestire in anticipo i cambiamenti, perché si ritiene che le modifiche non si possano prevedere con certezza
- ▶ In alternativa, XP propone un continuo miglioramento del codice (**refactoring** [Fowler et al., 1999]) per rendere più semplice l'implementazione delle eventuali modifiche future
- ▶ Refactoring è un processo di miglioramento del codice che viene riorganizzato e riscritto per renderlo più efficiente e comprensibile, senza cambiarne le funzionalità

# REFACTORING



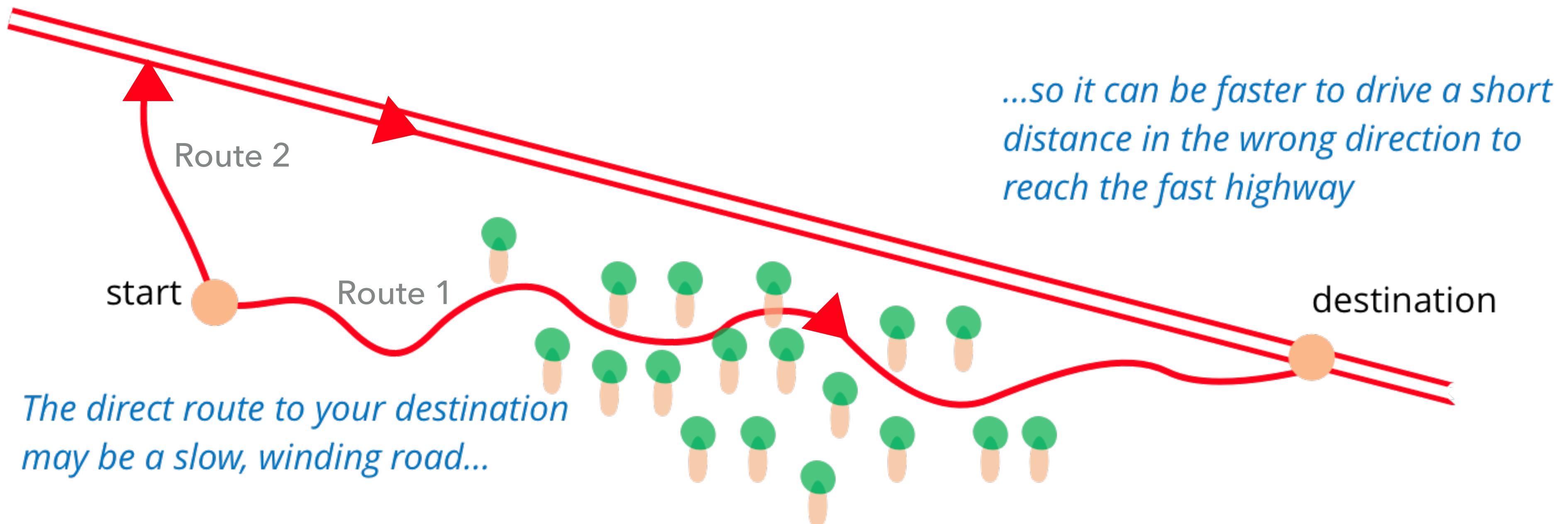
 Kent Beck 🌻  
@KentBeck

Follow ...

for each desired change, make the change easy (warning: this may be hard), then make the easy change

1:07 am · 26 Sep 2012

1,086 Reposts 205 Quotes 1,899 Likes 130 Bookmarks



# REFACTORING



- ▶ Il team di sviluppo cerca proattivamente aspetti del software da migliorare e implementa i miglioramenti immediatamente
- ▶ Il miglioramento può riguardare anche situazioni in cui non vi è una necessità immediata
- ▶ Si ritiene che un codice di più alta qualità, riduca la necessità di documentazione e faciliti le modifiche future

# I COMMENTI NON SERVONO A NULLA

0/6

codepizza

codepizza | Commenti non servono a nulla ✕

I commenti non rendono il tuo codice più leggibile, ma anzi lo degradano.

Se usi i commenti per documentare e spiegare scelte fatte nel codice stai sbagliando!

Arriverà il giorno in cui modificherai il codice senza aggiornare i commenti...

Inconsistenza che creerà degrado del codice e problemi nel team. 😤

(Ps: Se il codice è troppo complesso per leggerlo, forse sarebbe meglio rivederlo 😊)

Commenterai ancora il tuo codice? 🙅

#coding #programmer #informatica

3 w See Translation

Uno che parla così penserei che o ha esperienza 0 oppure che è alla prima settimana di corso su boolean

3 w 47 likes Reply See Translation

View replies (6)

Non avete mai ripreso codice vecchio di anni oppure lavorato in team? Direi che i commenti servono in quei casi.

3 w 57 likes Reply See Translation

View replies (9)

Liked by [REDACTED] and others

13 SEPTEMBER

- ▶ Ci sono anche posizioni estreme in cui si ritiene che un codice di più alta qualità elimini del tutto la necessità di documentazione, anche interna



# ESEMPI DI REFACTORING



- ▶ Rimozione di codice duplicato
- ▶ Rinominare classi, attributi e metodi per renderli più comprensibili
- ▶ Creazione di librerie che contengono codice utile a più classi o progetti

```
public class MyClass {  
  
    public int method1() {  
        int a = 0;  
        int b = 1;  
        a++;  
        b++;  
        return a + b;  
    }  
  
    public void method2() {  
        int a = 0;  
        int b = 1;  
        a++;  
        b++;  
        System.out.print(a + b);  
    }  
}
```

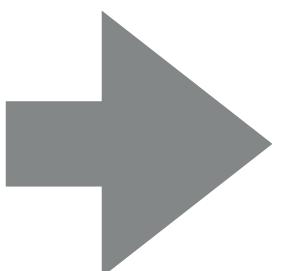
**Two clones**

# ESEMPI DI REFACTORING



- Esempio di metodi che condividono codice duplicato e possibile refactoring

```
def calcola_area_triangolo(base,  
altezza):  
    area = (base * altezza) / 2  
    return area  
  
def calcola_area_rettangolo(base,  
altezza):  
    area = base * altezza  
    return area
```



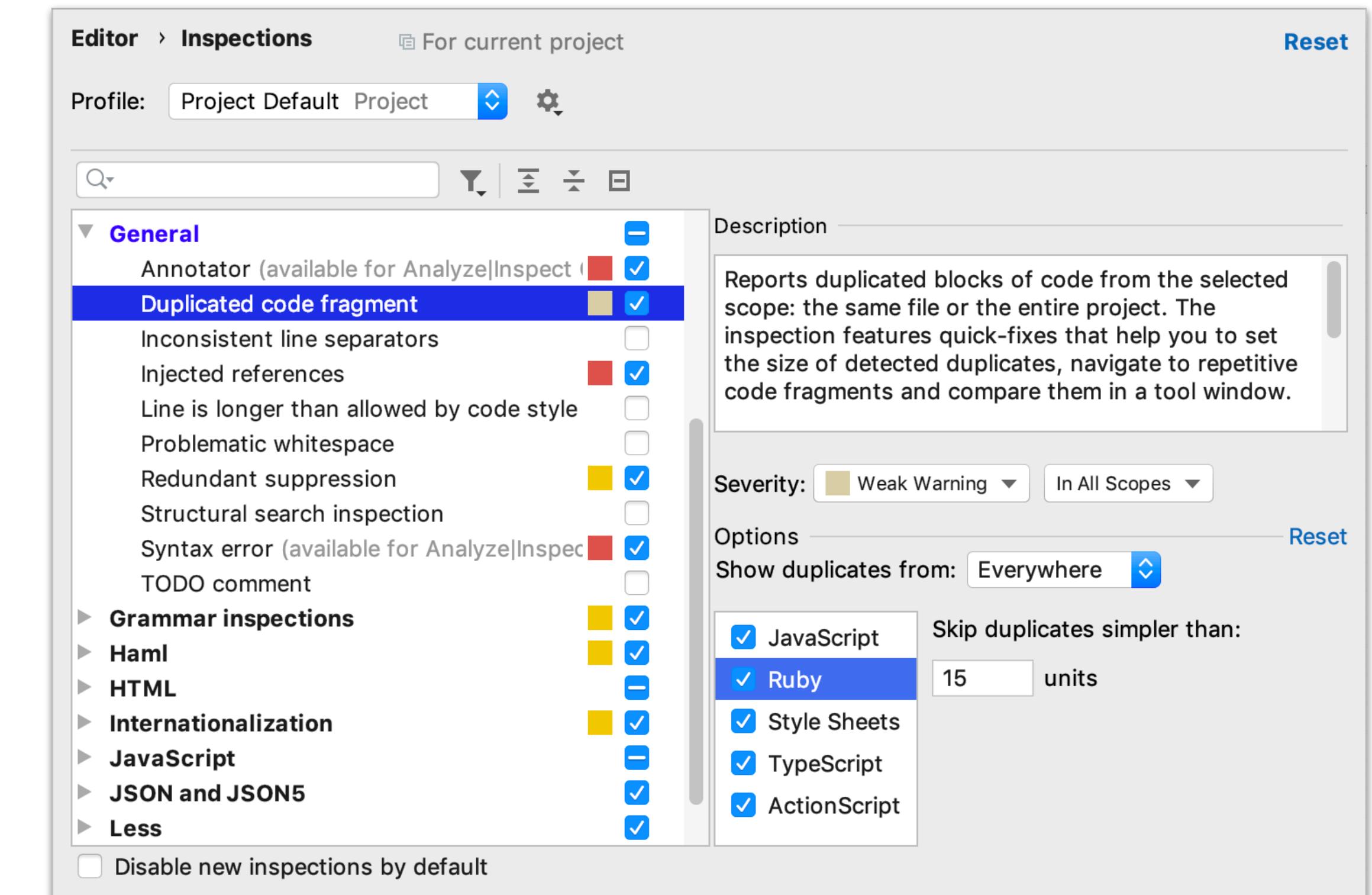
```
def calcola_area(base, altezza):  
    area = base * altezza  
    return area  
  
def calcola_area_triangolo(base,  
altezza):  
    area = calcola_area(base,  
altezza) / 2  
    return area  
  
def calcola_area_rettangolo(base,  
altezza):  
    area = calcola_area(base,  
altezza)  
    return area
```

# ESEMPI DI REFACTORING



- ▶ Gli ambienti di sviluppo integrato (IDE) come Jetbrains IntelliJ IDEA offrono funzionalità per aiutare lo sviluppatore nel refactoring

```
def calcola_area_triangolo(base,  
altezza):  
    area = (base * altezza) / 2  
    return area  
  
def calcola_area_rettangolo(base,  
altezza):  
    area = base * altezza  
    return area
```



# ESEMPI DI REFACTORING



- Mi raccomando di controllare sempre il risultato, soprattutto quando ci affidiamo a ChatGPT

Default (GPT-3.5)

V Ciao capo, ti fornisco del codice e tu mi suggerisci possibili azioni di refactoring

Ciao! Sarà un piacere aiutarti con il refactoring del codice. Per favore, incolla il codice di cui hai bisogno di assistenza, e sarò felice di darti suggerimenti su possibili miglioramenti. Assicurati di fornire dettagli sul linguaggio di programmazione in uso e una breve descrizione del problema che desideri risolvere o dell'area in cui desideri apportare miglioramenti.

def calcola\_area\_triangolo(base, altezza):  
 area = (base \* altezza) / 2  
 return area  
  
def calcola\_area\_rettangolo(base, altezza):  
 area = base \* altezza  
 return area

Default (GPT-3.5)

Il tuo codice sembra abbastanza semplice, ma potrebbe beneficiare di alcune piccole ottimizzazioni. Ecco alcune possibili azioni di refactoring:

1. **Unificazione dei calcoli di area:** Poiché sia il triangolo che il rettangolo coinvolgono calcoli di area, puoi creare una funzione generica per calcolare l'area di una forma geometrica con base e altezza come argomenti. In questo modo, eviti la duplicazione del codice.

python

```
def calcola_area(base, altezza):  
    area = base * altezza  
    return area
```

Ora puoi utilizzare questa funzione per calcolare l'area di qualsiasi forma geometrica con una base e un'altezza, come il triangolo e il rettangolo:

python

```
area_triangolo = calcola_area(base_triangolo, altezza_triangolo)  
area_rettangolo = calcola_area(base_rettangolo, altezza_rettangolo)
```

# REFACTORING - PRO



- ▶ Lo sviluppo incrementale tende a portare al deterioramento del codice. Il refactoring continuo mitiga tale deterioramento, migliorando la struttura e la leggibilità del codice
- ▶ Esistenza di tool per automatizzare alcune operazioni di refactoring

# REFACTORING - CONTRO



- ▶ Talvolta il refactoring a livello di codice non basta per supportare un cambiamento, ma è necessaria una modifica dell'intera architettura che è più costosa
- ▶ Bisogna trovare un compromesso tra tempo dedicato allo sviluppo di nuove funzionalità e refactoring



# TECNICHE AGILI

## SVILUPPO PRECEDUTO DAI TEST

# SVILUPPO PRECEDUTO DAI TEST



- ▶ In XP il testing è fondamentale: il software è testato dopo ciascun cambiamento
- ▶ Caratteristiche fondamentali:
  - **Sviluppo test-driven** - i casi di test da soddisfare sono scritti prima del codice e pertanto *guidano* lo sviluppo
  - **Automatizzazione dei test** - strumenti eseguono automaticamente tutti i test ogni volta che viene rilasciata una nuova versione
  - **Coinvolgimento del cliente** - il cliente è coinvolto nello sviluppo dei test di accettazione delle storie da essere implementate nell'iterazione successiva

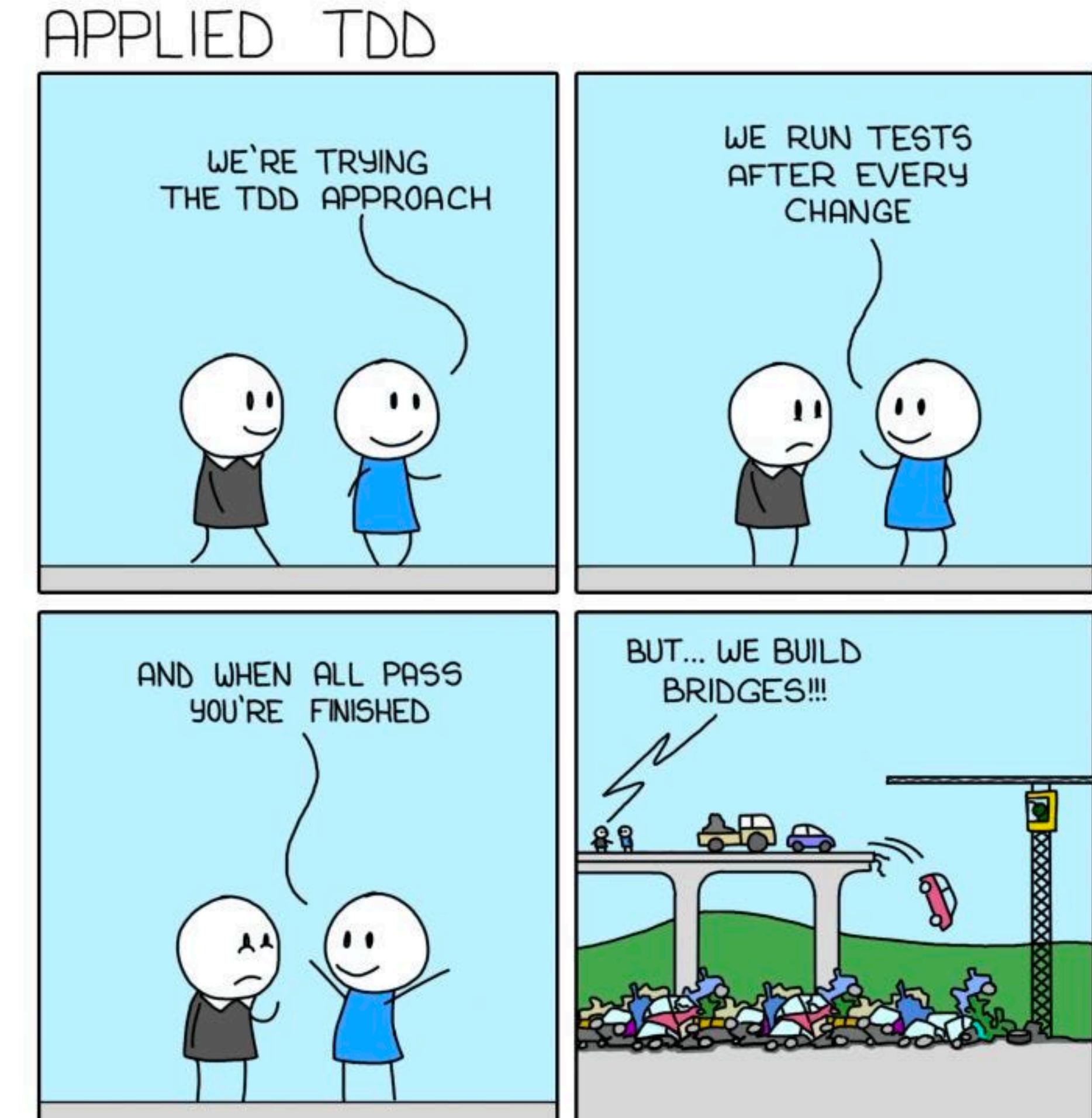
# SVILUPPO TEST-DRIVEN (TDD)



- ▶ Scrivere i test prima del codice (e non dopo) chiarisce i requisiti da implementare
- ▶ Questo aspetto è cruciale in assenza di specifiche accuratamente documentate che guidino i test di sistema (*vedere approcci plan-driven*)
- ▶ I test sono scritti come programmi, per essere eseguiti automaticamente. Ogni test simula l'invio degli input e controlla l'output
- ▶ Possibile eseguire i test mentre si programma in modo da scoprire subito eventuali problemi nel codice

# SVILUPPO TEST-DRIVEN

- ▶ Sia i test pre-esistenti che i nuovi test vengono eseguiti quando una nuova funzionalità viene aggiunta. Ciò permette di verificare che la nuova funzionalità non abbia introdotto errori
- ▶ Lo sviluppo non può procedere finché tutti i test non sono stati superati



# SVILUPPO TEST-DRIVEN



## Task 1: Visualizzazione Catalogo

## Task 2: Composizione Ordine

## Task 3: Verifica dati Pagamento

Il pagamento può essere fatto in due modi: attraverso carta credito, oppure specificando un numero di conto pre-registrato. Nel primo caso, il cliente inserisce un numero di 16 cifre ed una data di scadenza ed il sistema verifica la validità della carta, mentre nel secondo caso il cliente fornisce il numero del conto ed il sistema ne controlla la esistenza.

- ▶ Lo sviluppo test-driven parte dalle schede utente, suddivise in task



## Test 3: Test per il controllo validità della carta di credito

**Input:** una stringa di 16 caratteri e due interi (per mese e anno di scadenza)

### Test:

Controllare che tutti i caratteri della stringa siano interi

Controllare che il mese sia compreso fra 1 e 12 e l'anno sia maggiore o uguale di quello corrente

Controllare la validità della carta facendone richiesta al gestore della carta

**Output:** OK, oppure un messaggio di errore per carta non valida

- ▶ Ogni task genera uno o più test che controllano la correttezza del codice corrispondente al task



- I test sono scritti come programmi eseguibili, invece che essere forniti come un insieme di dati
- Il programma di test simula l'immissione di un determinato input e valuta se l'output ottenuto corrisponde all'output atteso dalle specifiche
- Framework come JUnit facilitano la scrittura ed esecuzione dei test
- I test sono eseguiti ogni volta che una nuova funzionalità è aggiunta al sistema
- L'automatizzazione rende più facile e rapida la fase di verifica e validazione

# SVILUPPO PRECEDUTO DAI TEST - PRO E CONTRO



- ▶ **Pro (fondamentale)**
  - La scrittura del test implica la definizione di un'interfaccia e di una specifica comportamentale della funzionalità da sviluppare, riducendo le possibili incomprensioni, ambiguità e omissioni
- ▶ **Contro**
  - Pratica onerosa per il cliente
  - Sforzo per tenere aggiornati i test interessati dalle modifiche del codice
  - I test potrebbero essere incompleti, ossia per varie ragioni non verificano tutti i possibili scenari che potrebbero verificarsi

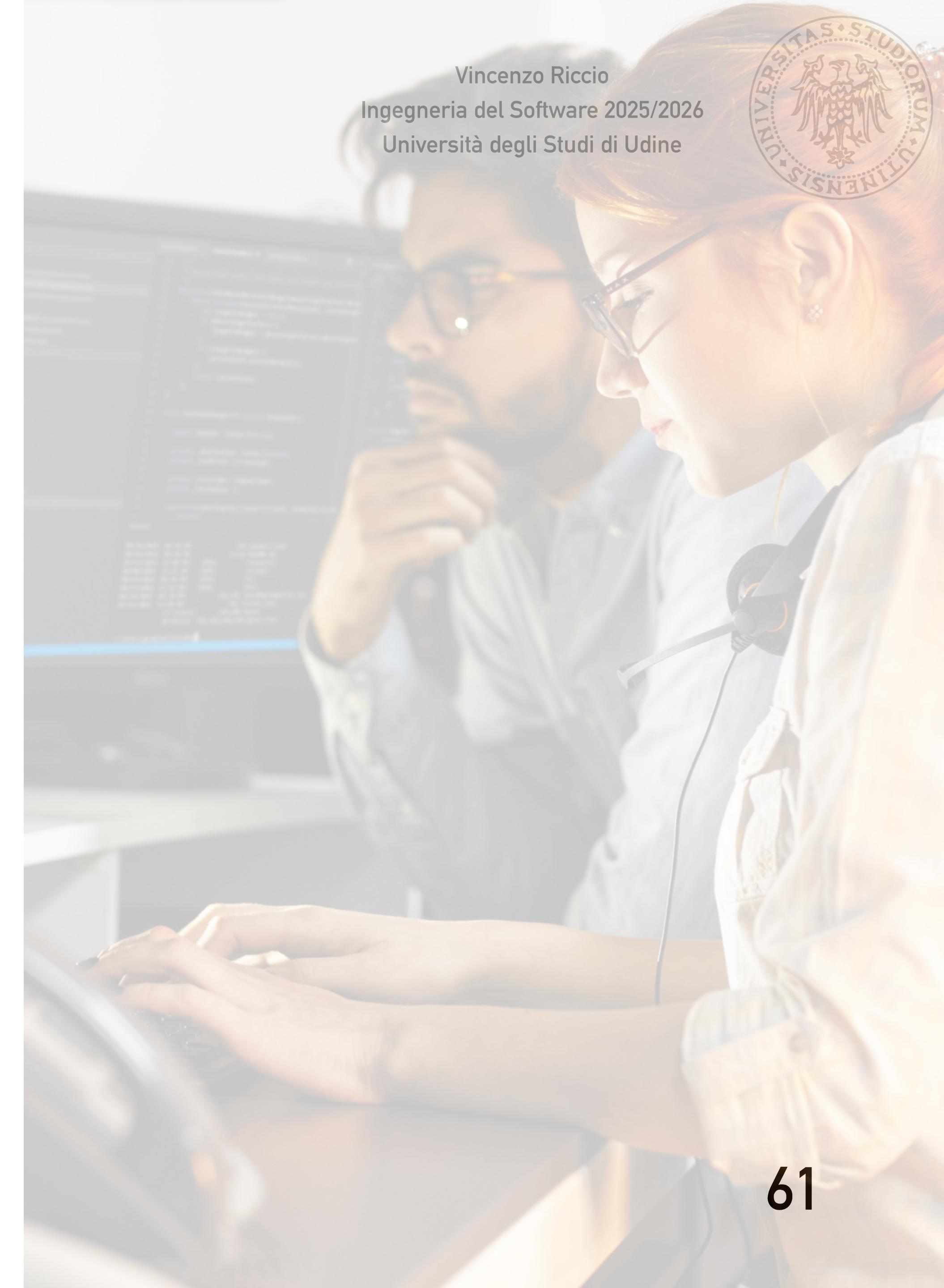


# TECNICHE AGILI

# PAIR PROGRAMMING

# PAIR PROGRAMMING

- ▶ I programmatori lavorano in coppie alla stessa postazione, e le coppie variano continuamente
- ▶ Il refactoring viene incoraggiato, perché è più facile che l'intero team ne benefici
- ▶ il software è di proprietà dell'intero team e i singoli non sono ritenuti responsabili dei problemi riscontrati nel codice (*egoless programming*). È il team che ha la responsabilità collettiva della risoluzione di questi problemi



# PAIR PROGRAMMING - DOMANDA



- ▶ Programmando a coppie si dimezza la produttività?



# PAIR PROGRAMMING - PRO E CONTRO



## ▶ **Pro**

- Aiuta a sviluppare il senso di proprietà del codice nel team e a diffondere la conoscenza nell'ambito del team
- Permette un processo di revisione informale: gli sviluppatori verificano reciprocamente il proprio lavoro e ogni linea di codice è controllata da più di una persona
- Riduce i rischi di fallimento dovuti a turn-over

## ▶ **Contro**

- Talvolta la programmazione in coppie può essere meno efficiente della programmazione individuale, soprattutto quando riguarda programmati esperti



# GESTIONE AGILE DELLA PROGETTAZIONE

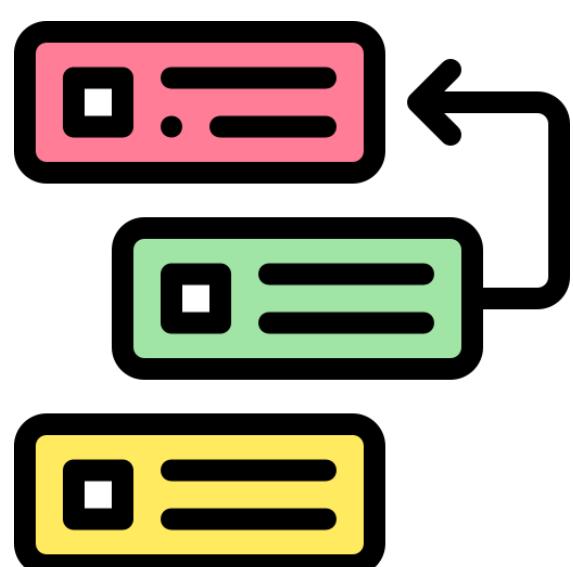
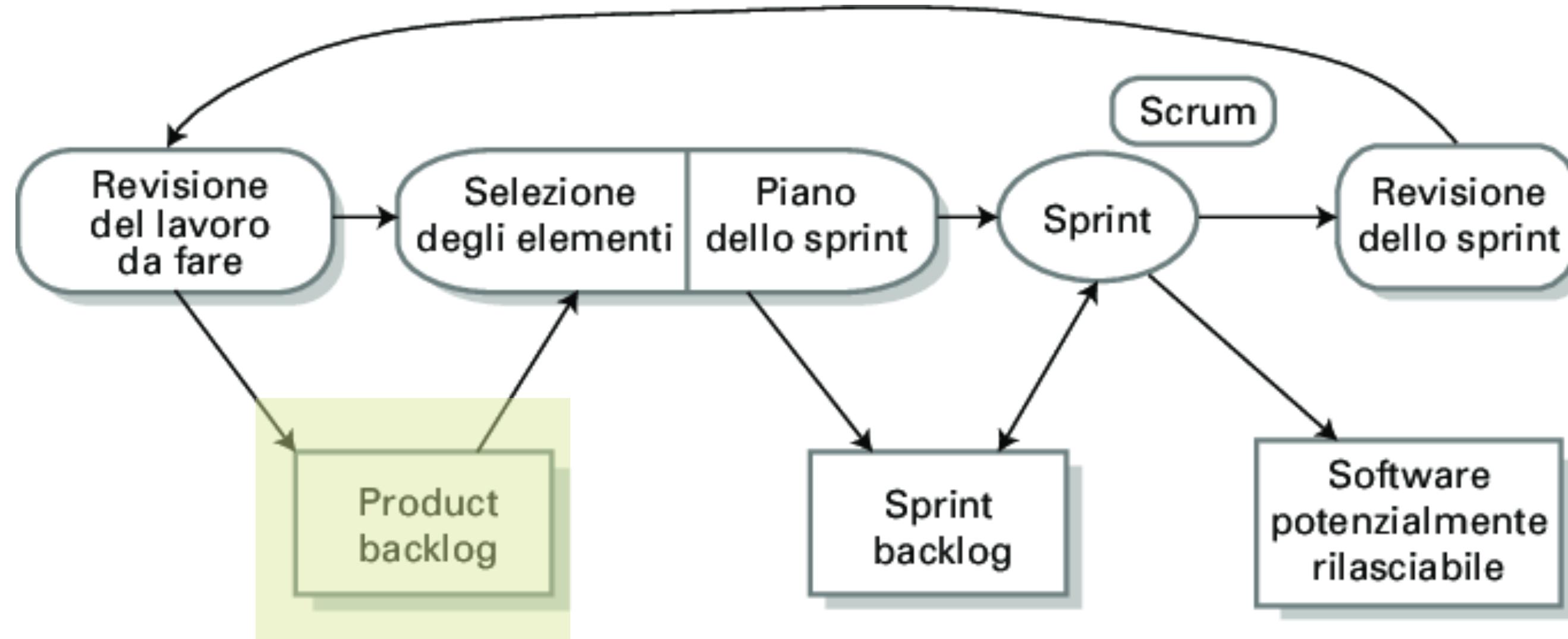
## SCRUM

# GESTIONE DELLA PROGETTAZIONE



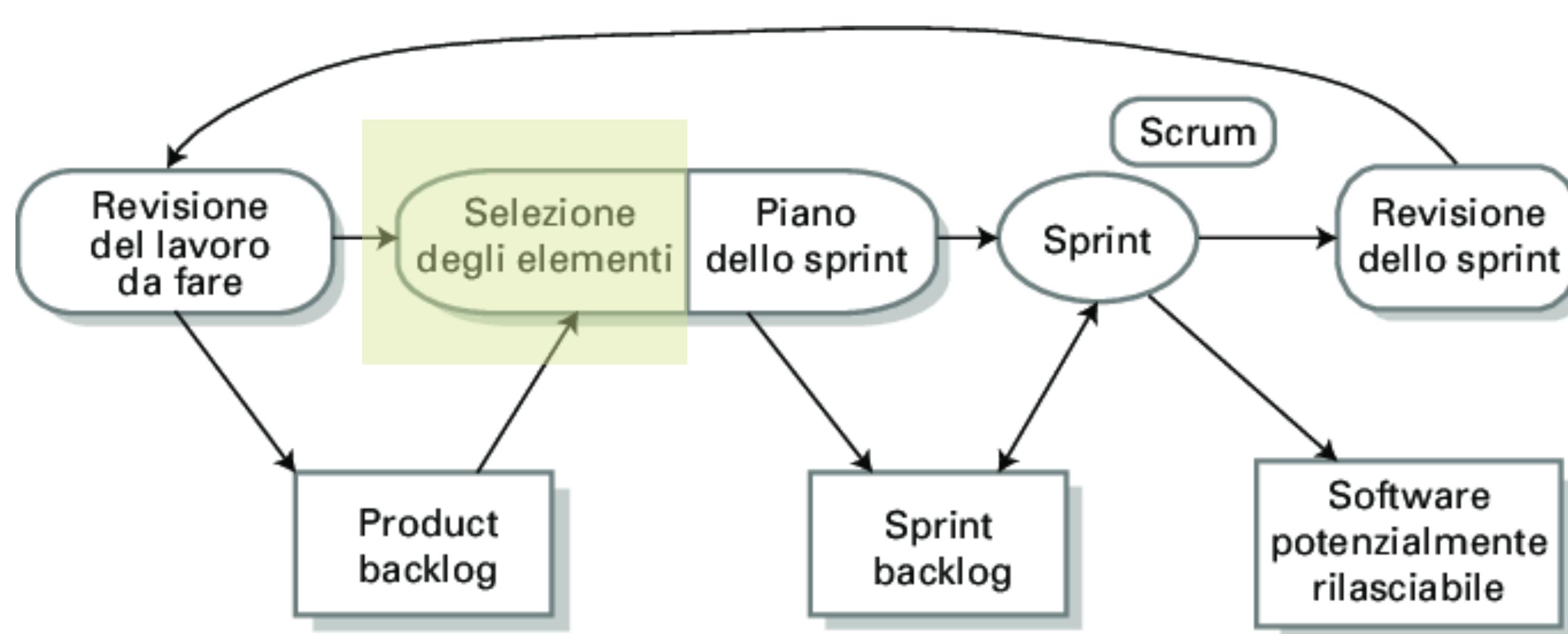
- ▶ La pianificazione informale proposta dai primi seguaci dei metodi agili si è scontrata con l'esigenza di visibilità del processo da parte dei manager della società di software
- ▶ I manager devono controllare il processo per sapere se raggiungerà i suoi obiettivi e sarà consegnato in tempo e nei limiti di budget previsti
- ▶ Il metodo agile Scrum (Schwaber e Beedle 2001; Rubin 2013) offre un framework per organizzare agilmente progetti e fornire, in una certa misura, una visibilità esterna su ciò che sta accadendo all'interno del team di sviluppo del prodotto software

# SCRUM



Il backlog non riguarda sempre e solo codice, ma anche architettura o documentazione

- ▶ Ogni sprint parte dal product backlog - la lista degli elementi, come le caratteristiche del prodotto, i requisiti e il miglioramento dell'ingegnerizzazione
- ▶ La versione iniziale del product backlog può essere derivata da un documento dei requisiti, da una lista delle storie utente

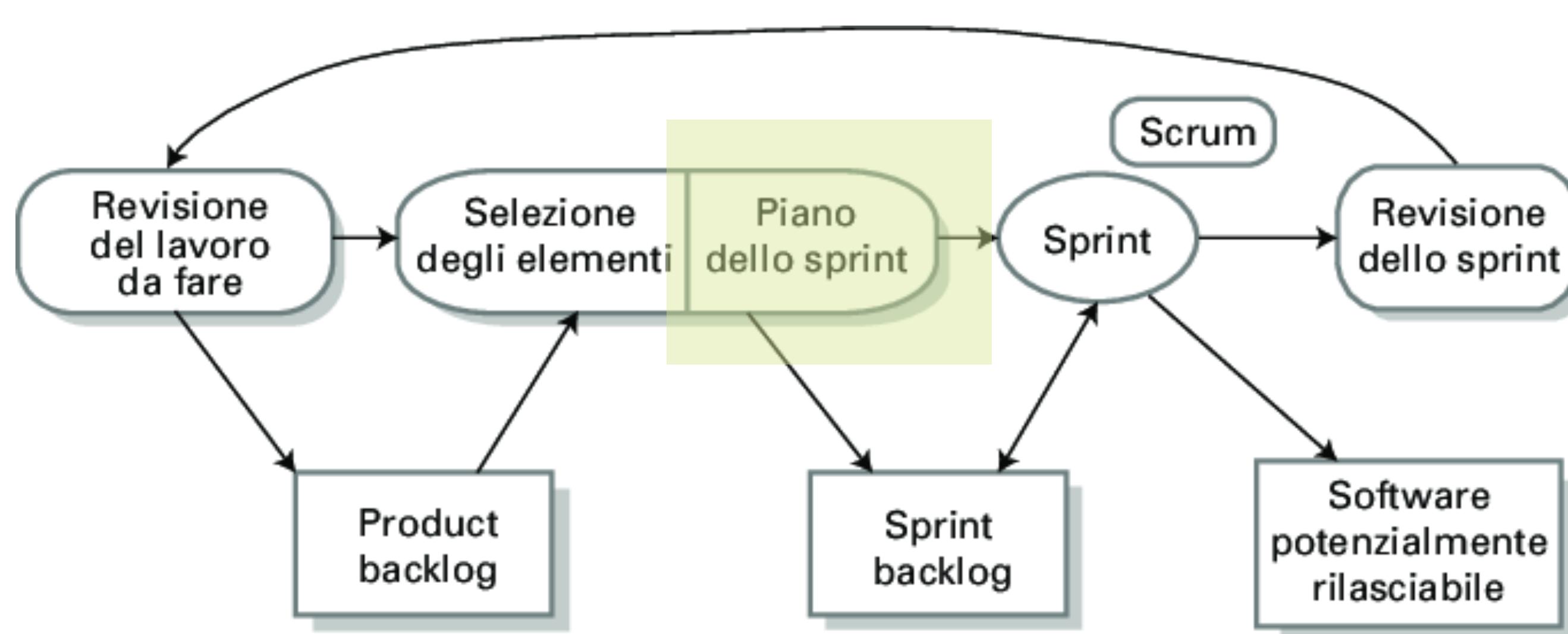


- ▶ All'inizio di ogni ciclo il product owner stabilisce le priorità del product backlog per definire quali sono gli elementi più importanti da sviluppare in quel ciclo

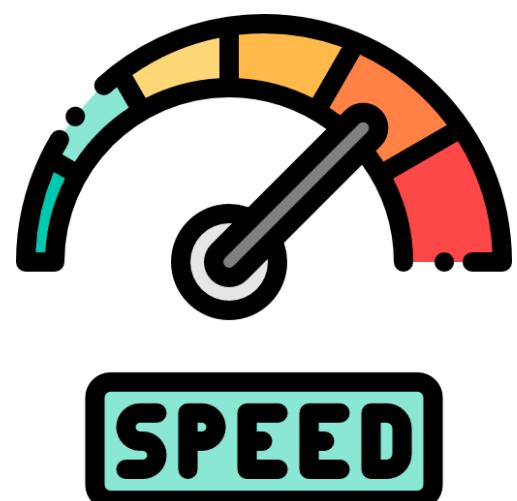


Il product owner deve identificare i requisiti del prodotto, stabilirne le priorità e rivedere continuamente il product backlog per garantire che il progetto continui a soddisfare i requisiti critici. Può essere un cliente, product manager o un altro rappresentante degli stakeholder.

# SCRUM

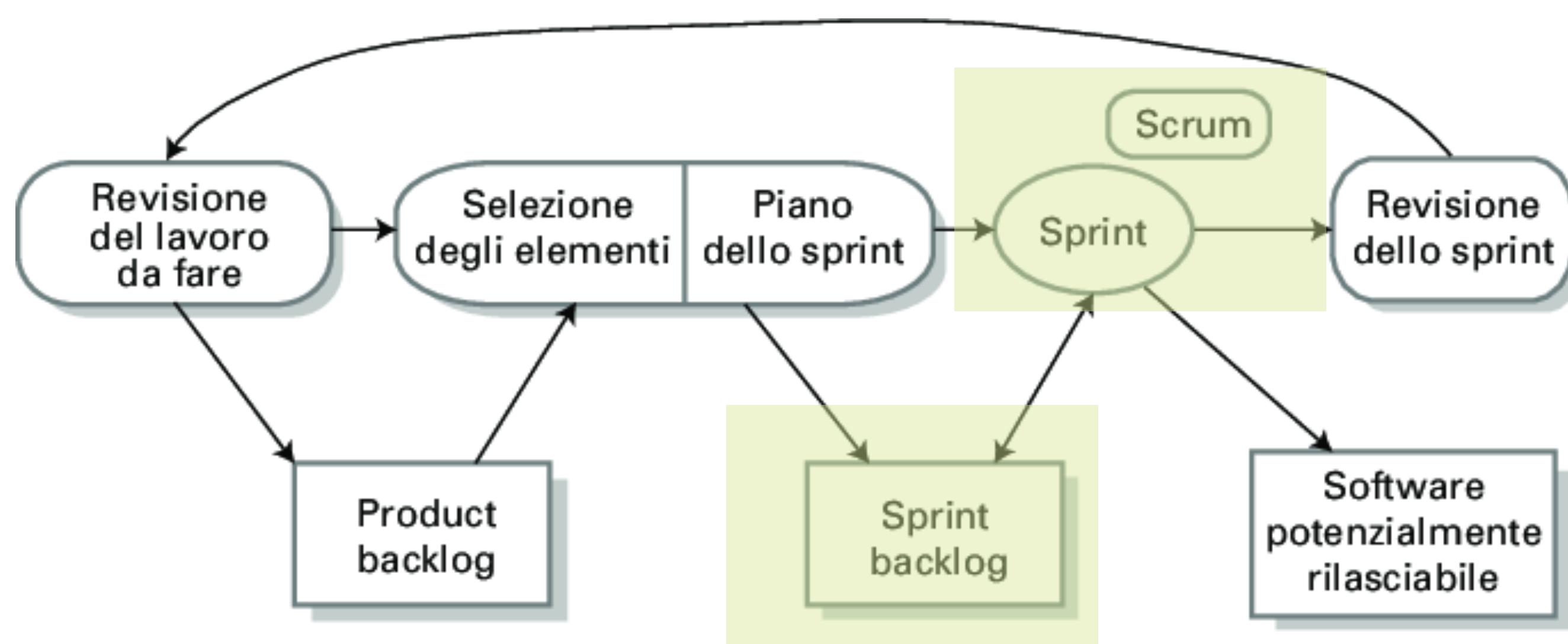


- ▶ Tutti i membri del team vengono coinvolti nella scelta degli elementi con priorità più alta che dovranno essere completati
- ▶ Valutano il tempo richiesto per completare questi elementi in base alla **velocità**



la velocità raggiunta nei precedenti sprint rappresenta una stima quanto lavoro del product backlog può essere svolto in un singolo sprint

# SCRUM

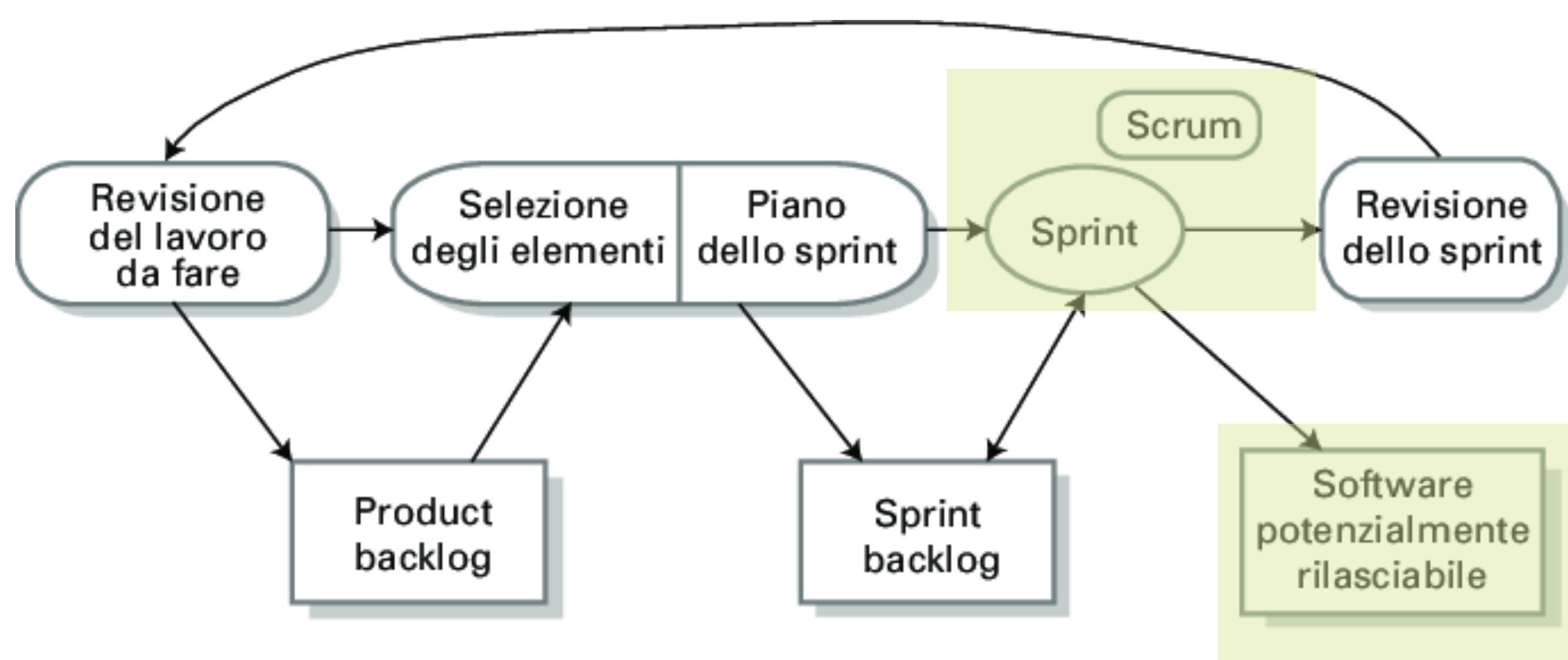


- ▶ La pianificazione dello sprint porta alla creazione di uno **sprint backlog** - il lavoro da svolgere in quello sprint
- ▶ Il team sceglie chi dovrà lavorare su determinati elementi, e avvia lo sprint
- ▶ Il team di sviluppo ha dimensioni contenute (circa 7 persone)



Lo sprint è un'iterazione dello sviluppo (~2-4 settimane)

# SCRUM

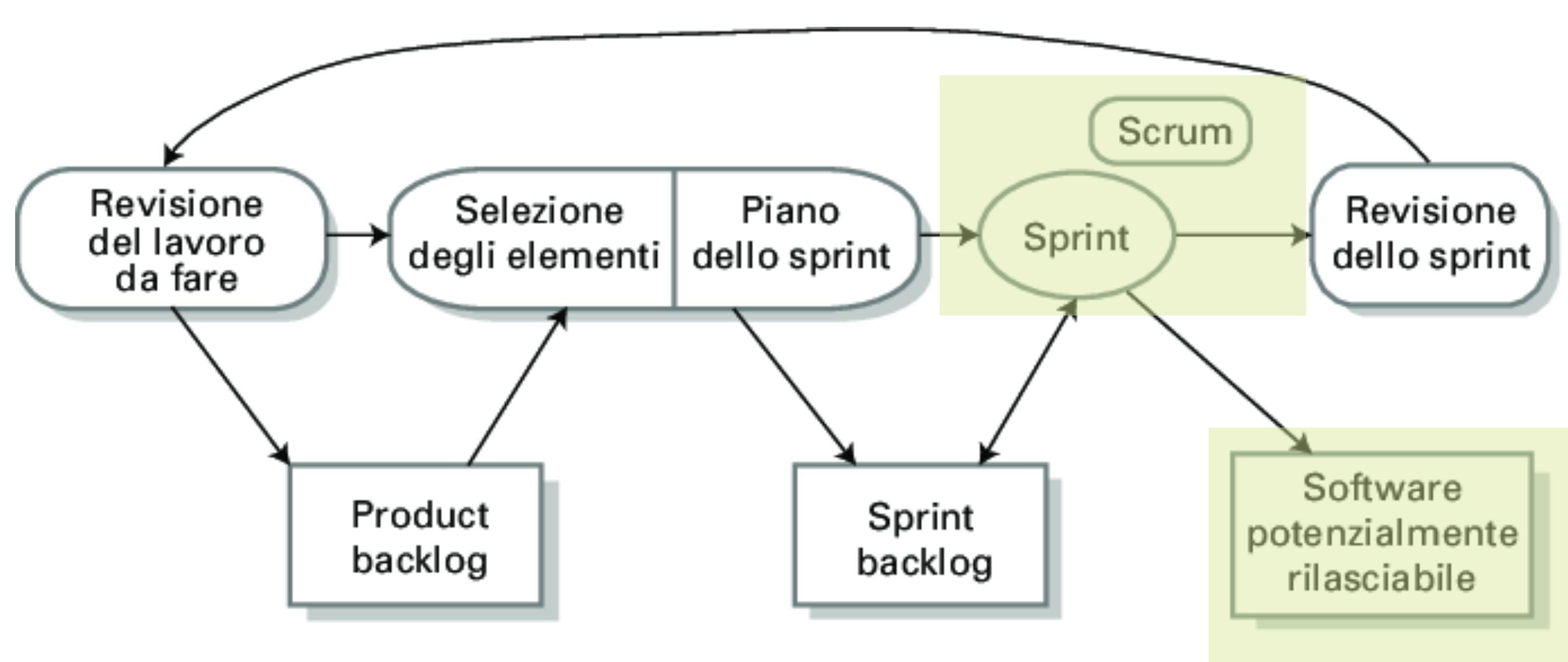


- ▶ Una riunione giornaliera del team di Scrum esamina l'avanzamento del lavoro e stabilisce le priorità del lavoro da svolgere in quel giorno.
- ▶ Lo scrum dovrebbe essere un breve incontro faccia a faccia di tutti i membri del team



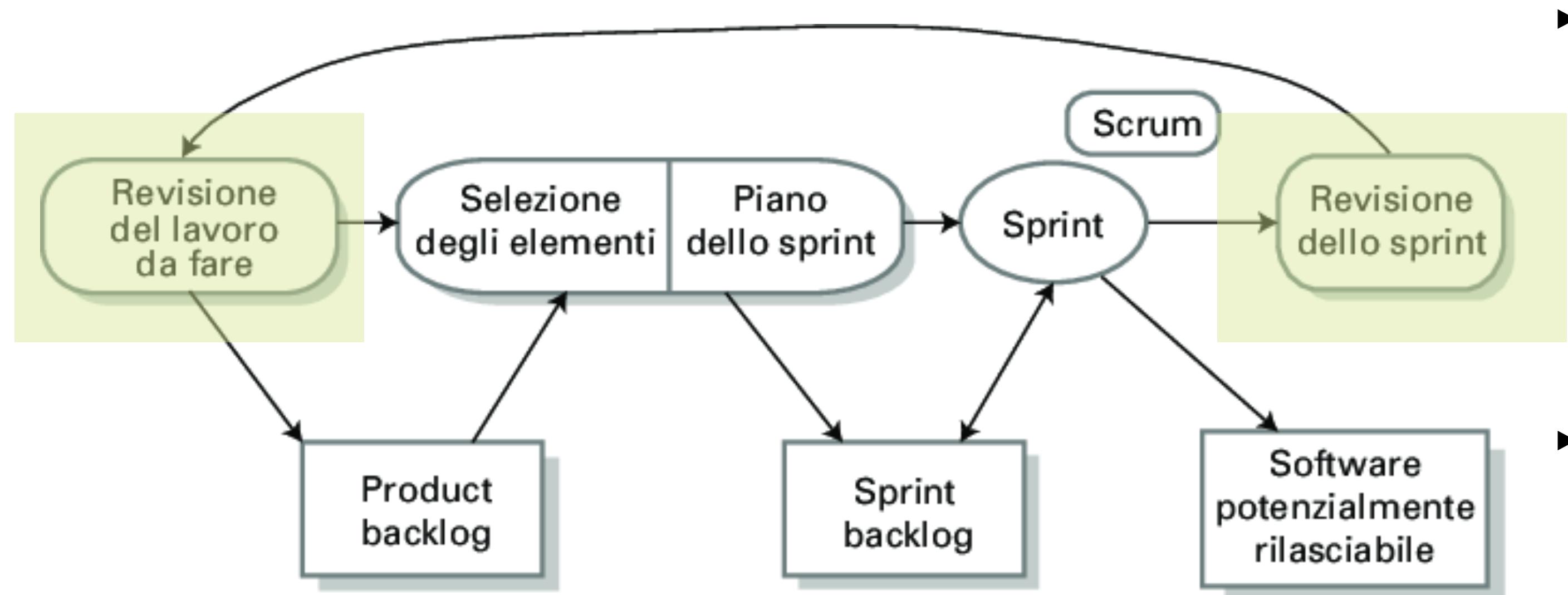
Lo ScrumMaster ha la responsabilità di garantire che il processo Scrum sia seguito e di guidare il team nell'uso efficiente di Scrum. Interfaccia con il resto della società ma non Project Manager classico

# SCRUM



- ▶ L'incremento del software che è consegnato da uno sprint è potenzialmente rilasciabile, ovvero deve trovarsi in uno stato finito e non occorre altro lavoro, come il testing, per incorporarlo nel prodotto finale
- ▶ Nella pratica, questo non è sempre realizzabile.

# SCRUM



- ▶ Alla fine di ogni sprint, si tiene una riunione di verifica, che coinvolge tutti i membri del team
- ▶ Questa riunione ha due obiettivi: (1) migliorare il processo; (2) input sul prodotto e sul suo stato per la revisione del product backlog