Sviluppo software in gruppi di lavoro complessi[1]

Mattia Monga

Dip. di Informatica
Università degli Studi di Milano, Italia
mattia.monga@unimi.it

Anno accademico 2018/19, I semestre

Lezione IV: Gruppi di lavoro agili (cont.)

Riassunto

- Sviluppare sw in gruppo è difficile (*The Tar Pit*): poiché le attività sono difficilmente *separabili* la complessità del coordinamento aumenta col quadrato del numero di componenti ("Legge di Brooks")
- La soluzione 'gerarchica': la squadra di lavoro gioca per il progettista (*cattedrale/sala operatoria*)
- L'esistenza di progetti *open source* con migliaia di partecipanti sembra mettere in dubbio la legge di Brooks
- L'organizzazione di questi gruppi è molto decentrata e le idee progettuali vengono da più parti (*bazaar*)
- *Legge di Linus*: le attività di verifica e convalida *sono* parallelizzabili
- In realtà però la *Tar Pit* e il *caos* sono sempre in agguato: più che dei *bazaar* disordinati, i progetti distribuiti tendono più ad assomigliare a dei *kibbutz*, con *valori tecnologici condivisi* e regole di partecipazione e sviluppo

Metodologie ‘agili’

Negli anni ’90 (dominati da “UML”) nascono molti approcci legati soprattutto dal fastidio per l’eccesiva enfasi data alla *documentazione del processo di sviluppo*.

- *eXtreme Programming* (XP)
- *Scrum*
- DSDM, Adaptive Software Development, Crystal, Feature-Driven Development, Pragmatic Programming, . . .

Meno rivoluzionari di quanto sembri a prima vista: i ricercatori dell’ingegneria del sw hanno sempre messo in evidenza la crucialità del processo di sviluppo (magari con l’esito di appesantirlo. . .) e studiato cicli di vita *iterativi*.

Il manifesto ‘agile’

Il termine agile è stato proposto da Martin Fowler. Nel 2001 viene pubblicato un *manifesto* dei ‘valori’ agili:
http://agilemanifesto.org/

- Gli individui e le interazioni più che i processi e gli strumenti
- Il *software* funzionante più che la documentazione esaustiva
- La collaborazione col cliente più che la negoziazione dei contratti
- Rispondere al cambiamento più che seguire un piano

I principi ‘agili’

1. Rilasciare *software* di valore, fin da subito e in maniera continua
3. Consegniamo frequentemente *software* funzionante
7. Il software funzionante è la principale misura di progresso

I principi ‘agili’

2. Cambiamenti nei requisiti, anche a stadi avanzati

4. Committenti e sviluppatori devono lavorare insieme quotidianamente

6. Conversazione faccia a faccia

I principi ‘agili’

5. Individui motivati e ben supportati

8. Sviluppo sostenibile: essere in grado di mantenere indefinitamente un ritmo costante
9. Eccellenza tecnica

11. Team che si auto-organizzano
12. A intervalli regolari il team riflette su come diventare più efficace

I principi ‘agili’

10. La semplicità — l’arte di massimizzare la quantità di lavoro non svolto — è essenziale

I principi ‘agili’

1. Rilasciare *software* di valore, fin da subito e in maniera continua
2. Cambiamenti nei requisiti, anche a stadi avanzati
3. Consegniamo frequentemente *software* funzionante
4. Committenti e sviluppatori devono lavorare insieme quotidianamente
5. Individui motivati e ben supportati
6. Conversazione faccia a faccia
7. Il software funzionante è la principale misura di progresso
8. Sviluppo sostenibile: essere in grado di mantenere indefinitamente un ritmo costante
9. Eccellenza tecnica
10. La semplicità — l’arte di massimizzare la quantità di lavoro non svolto — è essenziale
11. Team che si auto-organizzano
12. A intervalli regolari il team riflette su come diventare più efficace

Un manifesto *politico*

Il manifesto non è un metodo, né veramente un 'programma': parla ai cuori più che alle teste degli sviluppatori. È la dichiarazione di un disagio (non voglio essere pagato per scrivere documenti che nessuno legge) e la prefigurazione di un'utopia (il paradiso dei programmatori).
I metodi agili (principalmente XP e Scrum) fanno invece proposte concrete (non sempre facilmente identificabili come in linea col manifesto...)

I veri punti chiave

- *Team* piccoli e auto-organizzati, senza *manager* tradizionali, ma *facilitatori*
- Rifiuto di azioni e decisioni *big upfront*, sviluppo iterativo aperto alle variazioni in corso d'opera (rigorosamente regolate)
- Misura e controllo del processo di sviluppo, con pianificazioni con orizzonti temporali e funzionali ridotti
- Enfasi sul *testing*: non solo come *verifica & convalida*, ma come supporto alla progettazione, allo sviluppo e alla gestione delle variazioni

La parte più problematica è la *partecipazione della committenza*, che infatti è interpretati in maniera molto diversa dai vari approcci agili.

Semplicità e minimalismo

- *You aren't gonna need it* (YAGNI): non pensare né implementare una funzionalità finché non è davvero necessaria; realizzare la cosa più semplice che può funzionare.
- È in esplicito contrasto con il principio dell'ingegneria del sw classica "*Design for change*": se il cambiamento/adattabilità non è adeguatamente progettato costerà troppo.

User stories e *test*

"As a ... I want <business_functionality> so that <business_justification>"

```
assert_equal(fizzbuzz(1),1)
```

- Invece di *requisiti*, si usano storie d'uso, senza casi eccezionali, evidenza delle dipendenze...
- Invece di *specifiche*, si usano casi di test, con descrizioni estensive anziché intensive...

I *team Scrum*

http://www.scrumdesk.com/Download/Documents/AgileResources/ScrumGuidelines.pdf

- 7 ± 2 membri, *product owner*, *scrum master*
- Riunioni periodiche con scopi diversi, *daily stand-up*
- Il *product owner*: interfaccia col cliente/committente, fissa le priorità in base opportunità e rischi di *business*, gestisce il *backlog*
- Lo *scrum master*: cura il supporto al lavoro del gruppo, elimina gli impedimenti, fa rispettare le regole
- Gli altri: stimano la complessità del lavoro, identificano i rischi, dimostrano il progresso del prodotto

Le modalità di lavoro

- Il lavoro è frazionato in epopee, fatte di storie, rilasciate con *sprint* di 1-3 settimane
- *closed window rule*: durante uno *sprint* non si possono aggiungere funzionalità (se proprio è necessario, lo *sprint* ricomincia)
- Nelle riunioni di pianificazione i membri stimano la complessità con il *planning poker*, facilitato dallo *scrum master*, usando una 'storia di riferimento' come unità di misura: $\frac{1}{2}, 1, 2, 3, 5, 8, 13, 20, 40, 100$
- Nelle riunioni si identificano *pigs* (direttamente coinvolti) e *chicken* (solo interessati) che danno pareri solo se richiesto dai *pigs*

Le riunioni

Daily stand-up (15 min.) Cosa abbiamo fatto ieri, cosa facciamo oggi, ci sono impedimenti?

Planning (1-5 giorni) Pianificazione di uno *sprint*, definizione dello sprint backlog con la stima per ogni epopea/storia

Retrospettiva (30 min.) Alla fine di uno *sprint*, per migliorare

Review (1 ora) Alla fine di uno *sprint*, presentazione del lavoro agli *stakeholder*

Tecniche di lavoro

Ogni metodologia agile ne ha di specifiche, le più famose sono:

- *Pair programming*
- Codice condiviso (di proprietà collettiva)
- *Refactoring*
- *Test Driven Development* (TDD)
- *Velocity tracking*

Pair programming

Si programma a coppie, con una sola tastiera.

- Obbliga a rendere espliciti i ragionamenti
- Aiuta a mantenere il focus sull'obiettivo
- Diffonde la conoscenza totale della *codebase* (riducendo anche i rischi in caso di assenza di un collaboratore)

Questa (e TDD) è fra le tecniche maggiormente studiate sperimentalmente: nessuna evidenza che faccia differenza sulla qualità dei prodotti. La produttività, apparentemente dimezzata, rimane simile.

Codice condiviso

Tutto il *team* è responsabile di tutto il codice e può modificarlo a piacimento.

- Un'unica *codebase* (grazie agli scm come git, è facile usare le branch per evitare il "merge" troppo caotici)
- *Continuous integration* (possibile grazie a TDD)
- Il codice è una forma di comunicazione *broadcast*

La proprietà collettiva non è una buona ragione per rinunciare all'*information hiding*

Refactoring

refactoring

Martin Fowler, 2000: "*is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.*"

Sono piccole trasformazioni che non cambiano la semantica del codice, spesso attuabili automaticamente con un *editor* "consapevole" del linguaggio di programmazione.
Fowler mantiene un catalogo:
http://refactoring.com/catalog/

Esempi di *refactoring*

- Fattorizzazione di codice ripetuto in una funzione/metodo
- Campi attributo in metodi *getter*/*setter*
- Eliminazione di condizionali, sostituendoli con opportuni collegamenti dinamici (sottoclassi)
- Fattorizzazioni di comportamenti complessi in superclassi (eventualmente astratte)

Test-driven Development

Il *test* di unità viene scritto prima dell'unità stessa, servendo come "specifica" (ma senza la necessaria generalità!)

1. Aggiungi un *test*
2. Ripeti tutti i *test* assicurandoti che il nuovo *test* fallisca
3. Scrivi il codice dell'unità
4. Ripeti i *test* (questa volta dovrebbero passare)
5. *Refactoring* mantenendo il superamento dei *test*
6. Da capo

Ogni *bug* dovrebbe essere esaminato attentamente e diventare un nuovo caso di *test*

Supporto al *test*: *testing frameworks*

Librerie "xUnit" (JUnit, Kent Beck, 2002)

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {
  @Test
  public void evaluatesExpression() {
    Calculator calculator = new Calculator();
    int sum = calculator.evaluate("1+2+3");
    assertEquals(6, sum);
  }
}
```

Supporto al *test*: *mock objects*

Librerie che permettono di fare *behavior verification*, con oggetti "collaboratori".

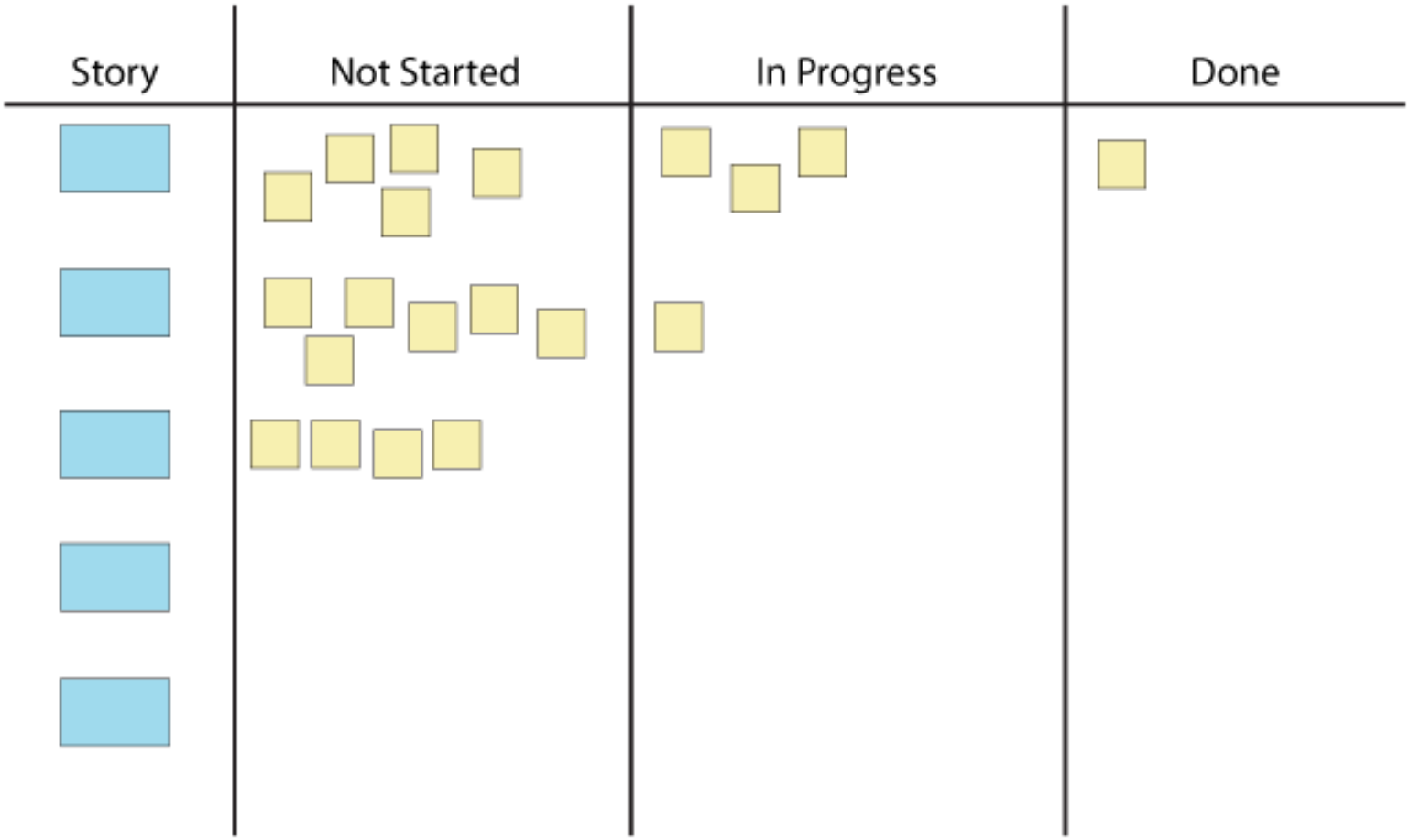
```
public class OrderInteractionTester extends MockObjectTestCase {
  public void testFillingRemovesInventoryIfInStock() {
    Order order = new Order("car", 50);
    Mock warehouseMock = new Mock(Warehouse.class);

    warehouseMock.expects(once()).method("hasInventory")
      .with(eq("car"),eq(50))
      .will(returnValue(true));
    warehouseMock.expects(once()).method("remove")
      .with(eq("car"), eq(50))
      .after("hasInventory");

    order.fill((Warehouse) warehouseMock.proxy());

    warehouseMock.verify();
    assertTrue(order.isFilled());
  }
```

La *task-board*

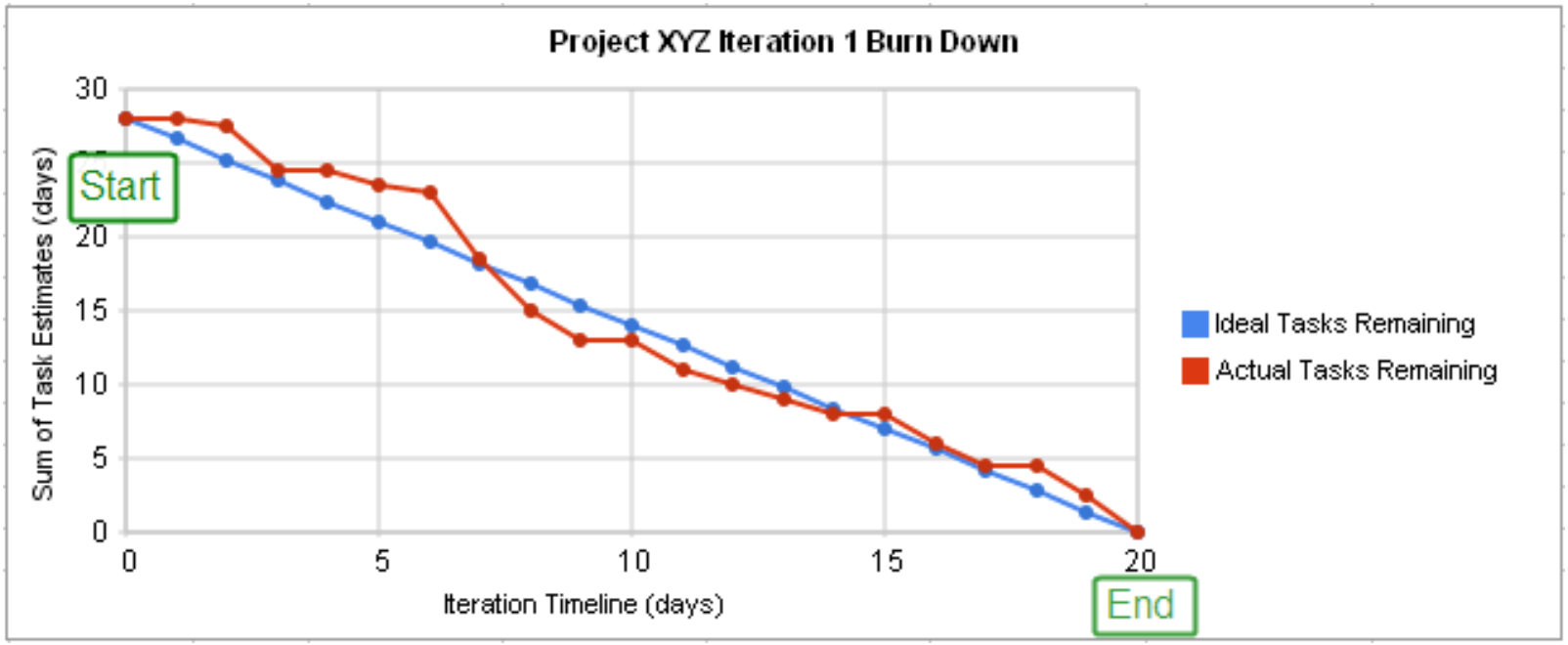


Velocity

Non è veramente una velocità, semmai uno "spazio percorso" in un tempo dato per fisso.

- In una iterazione (*sprint*) è la somma degli *item* in stato *Done*
- Se ne tiene traccia giornaliera con la *burn down chart*
- Inizialmente stimata riferendosi a $\frac{1}{3}$ del tempo a disposizione; con 6 programmatori e uno *sprint* di 2 settimane: $6 \times 5 \times 2 \cdot \frac{1}{3} = 20$

Burn down chart



(By I8abug - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=15511814)